Using the GLUT

This document provides a more detailed description of the minimum steps necessary to write build and execute an OpenGL 3D application that runs on the three desktop platforms Windows, OSX and GNU/Linux.

The GLUT is an old library but is still usable, comes as standard on a Mac OSX platform and is easily installed on the others. A newer, but completely compatible (no need to change any code - not even header files) called FreeGLUT can be used as a replacement. However FreeGLUT is not standard on OSX and there may be a bit of tweaking necessary to get it working.

As discussed in the book OpenGL has gone/is undergoing a radical revision with the older fixed functionality pipeline being replaced by the programmable shader pipeline, it is still possible to use the GLUT with this new scheme provided that none of the deprecated features are used in your part of the application.

To access the additional function in the API we recommend using the GLEW for Windows and Linux (on OSX many of the new functions are included in the normal OpenGL frameworks.)

Despite the radical changes in OpenGL it still offers a **compatibility mode** where the original fixed function (and first version of the shading language) approach may be used. In this document we will start by examining the use of the compatibility mode and then move on to discuss the changes that would be necessary to deploy the same application using the **core mode** of the latest versions.

> Basically in core mode you have to do all the geometry and shading calculations yourself in your own shader programs. Out goes the MODELVIEW and PROJECTION matrices, out goes all the transformation functions, `glRotate()` etc. Out go the lighting, materials and color functions. And out goes the single vertex specification, `glVertex(); glNormal();`.
>
> Vertices, (and vertex properties, normals, etc.) are transfered in blocks to the GPU, and drawn with functions like `glDrawArrays..();` `glDrawElements..();` Things like transform matrices, light positions, texture properties etc. are passed to the GPU processors using `uniform` qualified variables. And it is your responsibility to use these in the vertex, fragment, tessellation and geometry shader

processors to apply any transformations, lighting, shading and texturing.

Fortunately the approach to setting up a texture remains the same, of course, in the programmable pipeline textures must still be applied in the fragment shader from a *sampler*, however the use of samplers has not changed since the fist version of the shading language.

We start with our example in compatibility mode:

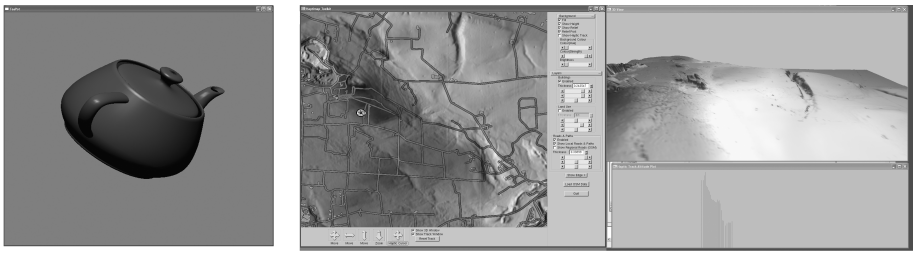
Compatibility mode

When it comes to desktop programs that use OpenGL there are two approaches: write native platform specific code, or use the GL Utility Toolkit (GLUT). To get an OpenGL program up and running very quickly on any desktop platform using the GLUT is the best option. The GLUT is readily available for all platforms pre-built DLLs are available for Windows and a pre-built package can be readily installed on all the GNU/LINUX flavors. On Macs running OSX it is installed with the developer tools package. From the programmer's perspective using the GLUT is essentially the same on all platforms. There are a couple of small differences which can be seen in the code examples accompanying this section. The GLUT offers facilitates to handle mouse events and keyboard interaction in a device independent way, and this make it a very powerful approach for developing an application when cross-platform deployment is important. It also makes the code you need to write very short and easy to follow. The GLUT library is very mature but still very usable, a slight variant called FreeGLUT is also popular because it offers the ability to exit the execution loop and return control control to the main program, something the GLUT itself cannot do.

We will now take a quick look at a very simple example that behaves in a similar way on all the desktop platforms, its appearance is illustrated . in figure **??**. In fact if you uses another little library, called the GLUI, that works on top of the GLUT it is possible to provide a versatile user interface, with text boxes, drop down menus etc. An example of this is also illustrated in figure 1 (the code for this example can be obtained as part of the HaptiMap toolkit which is referenced, as is the GLUI, via the book's on-line links.)

The GLUT is designed primarily to execute through a series of callback functions that are identified in an initialization phase. The essentials of all GLUT based program is presented in the four listings 0.1, 0.3, 0.4 and 0.4.

The program's entry point function shown in listing 0.1 performs three essential tasks. It defines the shape size and properties of the OpenGL window and displays it. Once the window has been displayed the OpenGL device and drawing contexts have been created by the GLUT and any calls to OpenGL library functions will apply to that window. The example uses its own function `Init()` to perform any *one-time* configuration. As the GLUT relies on callback functions to carry out rendering and user interaction, these are identified to the GLUT just before the main loop is started. The `glutMainLoop()` function never



(a) (b)

Figure 1: (a) A traditional teapot rendered using the GLUT library. (b)A multi window application rendered using the GLUT, and the GLUI which adds Dialog box type controls to any OpenGL application in a device independent way.

```
int main(int argc, char **argv){
 glutInit(&argc, argv);
 // create the window
 glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
 glutInitWindowSize(600, 600);
 glutCreateWindow("TeaPot");
 Init();       // Local functon
 // define callbacks
 glutReshapeFunc(Reshape);
 glutKeyboardFunc(Key);
 glutSpecialFunc(SpecialKey);
 glutDisplayFunc(Draw);
 // start
 glutMainLoop(); // Never returns
}
```

Listing 0.1: The main function setting up the GLUT.

returns, the program can be terminated by calling an exit function in response to a user action, but in a GLUT program any code placed after the main loop function call will not normally be executed.

In listing 0.1 no use is made of the value returned by `glutCreateWindow()`,. However it is possible to open several OpenGL windows at the same time using multiple called to `glutCreateWindow(.))` and in these cases the return value may be used to identify a specific window. It is easy to install separate callback handler functions for the different windows, this is done by calling the hander functions immediately after the window is created. A specific window can be refreshed by first activating it with a call to `glutSetWindow(...)`. Code fragments shown in listing 0.2 illustrates how to open two windows, set-up callbacks and post refresh commands to either one.

Once the viewing window has been created the further configuration of the OpenGL environment can be done in a completely device independent way. For example the code shown in listing 0.3 is typical of the actions done to set up the OpenGL pipeline. It also shows the minimal steps needed to configure a light

```
// create window 1 and set drawing callback
wID1 = glutCreateWindow("Window 1 Name ");
glutDisplayFunc(display1function);
..
// and the same for window 2
wID2 = glutCreateWindow("Window 2 Name ");
glutDisplayFunc(display2funciton);
..
glutSetWindow(wID1); // apply to window 1
glutPostRedisplay(); // refresh
..
glutSetWindow(wID2); // apply to window 2
glutPostRedisplay(); // refresh
```

Listing 0.2: Setting up multiple windows with the GLUT

```
void Init(void){
 // set background color
 glClearColor(0.5, 0.5, 0.5, 1.0);
 // set up a light
 float position[] = {0.0, 3.0, 3.0, 0.0};
 glLightfv(GL_LIGHT0, GL_POSITION, position);
 glEnable(GL_LIGHTING);
 glEnable(GL_LIGHT0);
 // define a material for the object
 float ambient[] = {0.1745, 0.01175, 0.01175};
 float diffuse[] = {0.61424, 0.04136, 0.04136};
 float specular[] = {0.727811, 0.626959, 0.626959};
 glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
 glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);
 glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
 glMaterialf(GL_FRONT, GL_SHININESS, 0.6*128.0);
 ... (Other configuration as required
 // build the object and store it
 // display list
MakeTeapot();

}
```

Listing 0.3: Initialising the drawing environment.

source. The last line in the code calls another example specific function that places the drawing commands in a *Display List* . OpenGL offers the helpful concept of the Display List as a method of recording "often used" drawing commands so that they do not have to be repeated every time the same sequence is needed. This is a very useful feature as it allows large numbers of drawing commands to be performed with a single function call. (The data in a display list is stored in the graphics memory and is therefore accessed rapidly.)

As a final part of the viewing process, the color framebuffer has to be mapped into the on-screen window. If the window size were always fixed, then the mapping could be defined during initialization, however when the output from the program is to appear in a window on a user's desktop it is likely that that user will change the size of the window or alter its aspect ratio, and therefore

```
static void Reshape(int w, int h){
 // fit viewoprt to window
 glViewport(0, 0, (GLint)w, (GLint)h);
 //configure an orthographic projection
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
 glOrtho(-6.0, 6.0, -6.0, 6.0, -1.0, 10.0);
 // configure an identity viewing trnansformation
 // look from (0,0,0) along the -Z axis
 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();
}
```

Listing 0.4: Resizing.

these changes must be accommodated by the mapping process. Responding to a change in aspect ratio of a window is especially important if the application wishes to maintain an undistorted appearance of the rendered output. In the GLUT any changes in window dimension will cause execution of the callback function specified to `glutReshapeFunc(Reshape);` The callback function in listing 0.4 is passed two arguments giving the new height and width of the output window, our example uses the resize callback to configure the transformation matrices and viewport so that a square region is mapped into the widow, no matter what its aspect ratio. Under these circumstances, the example could end up exhibiting distortion, result in distortion, for example circular objects may appear as ellipses. To maintain a one-to-one pixel size the *viewport*(window size) dimensions would have to be used to change the projection transformation.

```
static void Draw(void){
 // clear the background
 glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
 // save copy of MODELVIEW matrix.
 glPushMatrix();
 // move and rotate the object
 glTranslatef(0.0, 0.0, -5.0);
 glRotatef(rotY, 0.0,1.0,0.0);
 glRotatef(rotX, 1.0,0.0,0.0);
 glCallList(teaList); // draw the object
 // restore the previous marix.
 glPopMatrix();
 glutSwapBuffers();
}
```

Listing 0.5: Drawing.

Two of the possible callback functions are more or less essential. The scene most be drawn and the window resized. In our example this is done in functions `Draw(..)` and `Reshape(..)`, The GLUT is informed of these callback functions by passing their addresses as follows:

```
glutReshapeFunc(Reshape);
glutDisplayFunc(Draw);
```

In this short example the code to draw the teapot object has been omitted, it consists a small piece of code to render a polygonal mesh. Since this object itself does not change and all that the display function is doing is do is, change the orientation of the teapot so that we can view it from any angle, rendering can be considerably accelerated if the object is written into a display list during initialization, using the code:

```
glNewList(1, GL_COMPILE);  // The number "1" identifies the list
// Draw the object using any OpenGL functions
glEndList();  // finish building list
```

This display list is stored in graphics memory and then, when the object needs to be drawn, all that has to be done is, 1) generate the viewing transformation, and 2) tell the hardware to draw the contents of the display list. Display lists are very useful in optimizing the performance of an OpenGL program, they can contain texture mapping commands, but as yet, they are not implemented in the OpenGL for embedded systems.

To build this program for Windows it may be necessary to install the GLUT library. It will be necessary to add appropriate header files to the code:

```
#include <windows.h>
#include <stdlib.h>
#include <GL/glut.h>
```

and link with either a static library or stub library if the GLUT is being used as a DLL.

To build the example program for GNU/LINUX a makefile is required, listing 0.6 shows a very brief makefile that could be easily adapted for use with any project that requires to use the GLUT. The only line of any note is the one beginning `LIBS =` line which indicates that the GLUT and C run-time libraries are to be linked with the project. Note that for UNIX systems the library will have an actual filename of "libglut.a" with associated shared object libraries such as libglut.so (probably located in folder "/usr/lib". For OSX

```
TARGET1 = teapot
all: $(TARGET1)
CC = cc
CFLAGS = -Wall
LDFLAGS =
INCLUDES =
LIBS = -lglut -lrt
OBJS1 = teapot.o
%.o: %.c $(DEPS)
        $(CC) $(CFLAGS) $(INCLUDES) -c -o $@ $<
$(TARGET1): $(OBJS1)
        $(CC) $(LDFLAGS) -s -o $@ $^ $(LIBS)
```

Listing 0.6: LINUX makefile.

one can either setup an Xcode project that includes the OpenGL and GLUT

frameworks, or as OSX is based on BSD UNIX, a simple makefile such as that in listing 0.7 ca be used. Note the use of the term *framework* to represent a library under OSX and the three frameworks that need to be linked to allow OpenGL to be used in the application. LINUX and OSX use slightly different

```
TARGET1 = teapot
all: $(TARGET1)
CC = gcc
CFLAGS =
LDFLAGS =
INCLUDES =
LIBS = -framework GLUT -framework OpenGL -framework Cocoa

OBJS1 = teapot.o
%.o: %.c $(DEPS)
        $(CC) $(CFLAGS) $(INCLUDES) -c -o $@ $<

$(TARGET1): $(OBJS1)
        $(CC) $(LDFLAGS)  -o $@ $^ $(LIBS)
```

Listing 0.7: OSX makefile.

paths for their library include files, `#include <GLUT/glut.h>` and `#include <GLglut.h>` respectively, so these different paths have to be accommodated in the code. However even with the small code differences and program building approach the template presented in this section can be built upon to deliver a cross platform application that could call on *all* the features and power of OpenGL to deliver stunning 3D graphics, with virtually no porting issues.

Core mode

The code for the teapot example considered in the previous section is not compatible with OpenGL's core mode because it uses the fixed function pipeline and makes use of deprecated functions and features such as display lists and the matrix stack.

As we explain in the book OpenGL has undergone a revolution and whilst most systems will support compatibility mode it is a good idea to investigate how to evolve our application so that it fully complies with the latest core features of the latest version.

There are many things that we will have to change:

- The Patch implementation of the teapot should be done using *tessellation shaders* but this feature is not always supported and so we will change the specification to a simple triangular mesh.) [1]
- The pipeline will need to be changed to the GPU programmable hardware with vertex and fragment shaders.

[1] A later example may show how to use the tessellation shader, but at this time we do not have suitable hardware available.

- The use of display lists will need to be changes, we will do this by using vertex buffers.
- The model, view and projection transformations will need to be done explicitly in our code.

One thing that we won't change is the use of the GLUT to set up the OpenGL drawing context and configure the drawing surface and viewport. It would be possible to change to the use of the FreeGLUT which is the modern replacement for the original GLUT but this is not necessary and we can modernize our example without having to change its main development library.

However we will need to gain access to the recent OpenGL functions and the best way to do this is to use the GLEW library. (Note that the GLEW library is NOT required for Mac OSX versions of our GLUT program because most of the recent OpenGL functions have prototypes already specified in the OpenGL framework.

To see how our application evolves the code is presented in four steps, but the intermediate steps are only given for the Windows platform, a Mac OSX version of the final stage is include too. The evolution is as follow:

1. `windows1:` This step switches the pipeline from fixed to programmable and includes a vertex and fragment shaders that are compatible with the GLES version 1.2,.apart from this change no other code changes are made to the main project file. However, additional files, common to all the projects, are added to the Visual studio project:
   (a) Code to read, compile and link the vertex and fragment shader programs.
   (b) Code to perform equivalent geometric transformations to those of the fixed function matrix stack and viewing and projection utility functions[2].
2. `windows2:` This stage of evolution replaces the use of all the fixed function (matrix stack) transformations and `GLU` functions with explicitly generated homogeneous matrices and passes them to the shaders through the use of`uniform` variables.
3. `window3:` To proceed further we have to replace the patch drawn version of the teapot object with a polygonated version. This requires a list polygons and a calculation of the vertex normal vectors. No other changes are made (from version 3) the teapot's vertices are still written into a display list which is then rendered.
4. `windows4:` This is the final version in which the teapot's vertices and copied vertex buffers and the vertices in the buffer are rendered with al call to `glDrawElements(..)` The shader codes are switched to use *core*

[2]These will be discussed again later

mode systax i.e and the GLSL shaders are written in GLSL version 3.3 and later compatible code.

We shall explore the code of the final version in a little more detail but first let us discuss the supplementary transformation generators from file "geom.c" (located in folder "common").

Transformations

In this example, functions provided in the file "geom.c" handle the generation and combination of the 4×4 matrices that previously would have been handled implicitly by the transformation functions, such as `glPushMatrix(..);` `glTranslate(..);` `glRotate(..);` and the view and projection functions `glFrustum(...)` `glOrtho(...)` `gluLookAt(..);` and `gluPerspective(..);`.

In the file "geom.c" , the functions `Orho(...);` `Frustum(...)` `Perspective(...);` and `LookAt(...);` take the same arguments as their original OpenGL deprecated equivalents and each returns the matrix `m[4][4]` containing the equivalent transformation.

Functions like `translate(..);` and `rotx(..);` return a matrix `m[4][4]` containing the equivalent matrix. Functions `m4by4(..);` and `c4to4(...);` may be used to combine and copy transformation matrices. In our program we must build the direct equivalent of the `gl_MODELVIEW` and `gl_PROJECTION` matrices and pass them to the vertex shader, where we have to apply them to the vertex positions.

> To match the OpenGL matrix element ordering it is **essential** that every matrix is transposed before it is passed to the shader programs. The reason for this is explained in the book.

Final Windows version

The version uses vertex data stored using an indexed scheme defined by lists:

```
const int num_model_vertices = 1152;
const int num_model_indicies = 1600;

GLfloat model_vertex_normals[1152*3];

GLfloat model_vertices [] = { ...}

uint model_indicies [] = { ...}
```

The transformations are defined by the variables:

```
GLfloat MM[4][4],   // model matrix (combinations of translations rotations etc.
        VM[4][4],   // view matrix
        MVM[4][4],  // combination of the model and view matrices
        PM[4][4],   // the projection matrix
        MVPM[4][4], // model view and projection matrices combined (the final matrix)
        NM[4][4];   // the matrix that transforms the model's surfac normal vectors
                    // (this is basically the same as the rotation part of the model
                    //view matrix  - it is only correct for NON scaled models)
```

We will be creating vertex attribute buffers to store the vertex coordinates, normal vectors and an index buffer to enable us to draw the vertices in an indexed order. These are identified by the global variables:

```
// These are the identifiers for the buffers used to
GLuint vertexbuffer;
GLuint normalbuffer;
GLuint indexbuffer;
```

The uniform variables and shader program must be identified in the C source and these are defined by the global variables:

```
GLuint programID;          // shader program
GLuint MVMid,MVPMid,NMid,  // trnasformation matrix uniform IDs
       NAid,NVid;          // vertex attribute buffer IDs (position * normal)
```

Since the program is based on the GLUT program introduced at the start of this document the set-up is very similar - we must compile and load the vertex and fragment shaders and get the locations of the shader's uniform variables and the vertex shader's input attribute buffers. This is all done in function `Init()` In this Windows example the `Init()` function also initializes the GLEW library and calls function `Teapot();` to assemble the vertex data buffers.

```
programID = LoadShaders( "simple.vert", "simple.frag" );
Ident(VM); // looking from (0,0,0) in direction (0,0,-1)
MVMid = glGetUniformLocation(programID, "MVM");
MVPMid = glGetUniformLocation(programID, "MVPM");
NMid = glGetUniformLocation(programID, "NM");
NAid=glGetAttribLocation(programID,"glNormal");
NVid=glGetAttribLocation(programID,"glVertex");
..
glUseProgram(programID);
..
Teapot();
```

To build the teapot we load the vertex position and surface normal data in to buffers, for example to copy the vertex position data the following code fragment is used:

```
glGenBuffers(1, &vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(model_vertices), model_vertices, GL_STATIC_DRAW);
glEnableVertexAttribArray(NVid);
glVertexAttribPointer(NVid,3,GL_FLOAT,GL_FALSE,0, (void*)0   );
// Note: (NVid) must match shader vertex attribute location
```

The index data is copied to the index buffer as follows:

```
glGenBuffers(1, &indexbuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexbuffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(model_indicies), model_indicies, GL_STATIC_DRAW);
```

Drawing the teapot can be done in a single call to `DrawElements(..)` as we have all the vertex data and index data already bound during the creation

of the teapot. If we had wanted to use several objects we could have creates several Vertex Array Objects and bound them one after another. we could even have used a large list of vertices and bound and drawn several different index buffers that address the same vertex list.

To get the correct transformation the model, view, and perspective transformations must be built. We use our utility functions for this:

```
rotx(M1,rotX+270.0);  // 270 matches the Fixed Function rotation
roty(M2,rotY);
translate(M3,0.0,0.0,-5.0);
m4by4(M2,M1,MT);
m4by4(M3,MT,MM);   // buuld the model matrix
m4by4(VM,MM,MVM);  // combinen the model and view matrix
m4by4(PM,MVM,MVPM);  // model view and projection matriices combined
```

And pass then through to the vertex shader using:

```
transpose4(MVM); // MUST TRANSPOSE BEFORE PASSING TO OpenGL
glUniformMatrix4fv(MVMid, 1, GL_FALSE, &MVM[0][0]);
transpose4(MVPM); // MUST TRANSPOSE BEFORE PASSING TO OpenGL
glUniformMatrix4fv(MVPMid, 1, GL_FALSE, &MVPM[0][0]);
```

Then drawing the model is simple a matter of calling:

```
  glDrawElements(GL_TRIANGLES,num_model_indicies*3,GL_
                 UNSIGNED_INT,(void*)0);
 glutSwapBuffers();
```

The vertex shader picks up the vertex position and attributes from the buffer in which we placed them and the transformations from their respective uniform qualified global variables. In GLSL later than version 3.0 only one built in outpout variable is needed in vertex shader.

```
#version 330 core

// vertex attribute inputs
layout(location = 0) in vec3 glVertex;  // position
layout(location = 1) in vec3 glNormal;  // normal

uniform mat4 MVM;   // model view matrix
uniform mat4 MVPM;  // model view projection matrix
uniform mat4 NM;    // normal matrix (only 3x3 is used )

out vec3 normal;    // pass to fragment shader
out vec3 position;  // pass to fragment shader

void main(){
    position      = (MVM * vec4(glVertex,1)).xyz;
    normal        = (NM * vec4(glNormal,0.0)).xyz;
    // pass the built in variable
    gl_Position   = MVPM * vec4(glVertex,1);
}
```

The fragment shader collects its inputs from the vertex shaders and calculates ONE output, which we call `glFragColor;`

```
#version 330 core

layout(location = 0) out vec4 glFragColor;

in vec3 normal;
in vec3 position;

vec3 lightPos=vec3(0.0,0.5,0.0);
vec4 lightColor=vec4(1.0,1.0,1.0,1.0);
vec4 colour=vec4(0.8,0.2,0.2,1.0);

void main() {
 const float Specular=0.9;
 const float Diffuse=0.95;
 float diffuse=0.0,spec=0.0;

 vec3 ldir = normalize(vec3(lightPos-position));
 diffuse = max(dot(ldir,normal), 0.0);
 if(diffuse > 0.0){
   vec3  reflect = reflect(ldir, normalize(normal));
   spec = max(dot(reflect, normalize(position)), 0.0);
   spec = pow(spec, 32.0);
 }
 diffuse=clamp(diffuse,0.0,1.0);
 spec=clamp(spec,0.0,1.0);
 glFragColor = (colour * (1.0-Diffuse) + colour * Diffuse * diffuse)
              + lightColor* spec * Specular;
}
```

Note the Phong shading model and the short-cut assignment of "signs" compare with the fragment shader in project "Windows1"

```
// The reflection calculation should really be as below: (note the -ve signs)
//
// vec3 ldir = normalize(vec3(position-lightPos));  // incident direction
// diffuse = max(dot(-ldir,normal), 0.0);
// if(diffuse > 0.0){
//   vec3  reflect = reflect(ldir, normalize(normal));
//   spec = max(dot(reflect, normalize(-position)), 0.0); // from point to (0,0,0)
//   spec = pow(spec, 32.0);
// }
```

Building final version for Mac OSX

Currently we do not access to a Mac system that supports the GLSL shading language greater than version 1.2. This does not support the keywords `in` and `out` for passing variable between vertex and fragment shader. However the Mac version of GLUT and the OpenGL framework supports all the functions we have used, without the need for the use of the GLEW. We can therefore port the program to the OSX platform by simple demoting the GLSL shaders to version 1.2 and this is very easily done by changing the keywords `in` and `out` to `varying` and writing the fragment output color to the built in variable `gl_FragColor`

Compare these "Mac" shaders with the "Windows4" shaders to see how little needs to change.