BLG444E

Computer Graphics

2014 Spring

Report of Project 1

Date of Submission : 29.02.2014

040100118 Volkan İlbeyli
040100043 Mert Tepe

Instructor : Uluğ Bayazıt

Since this is the very basic hello world program of OpenGL, two of us did the entire project on our own and then in the end, combined our work into one project for the purpose of either of us having learnt the basics of OpenGL.

Task 2 : Keeping the aspect ratio of the square

Below are the list of changes made to the template code:

main.cpp

- `GLint h_uHeight, h_uWidth;` variables added to `struct SquareShaderState` as handles to window height and width, since they will be passed to the vertex shader as uniform variables, which will be used to keep the aspect ratio.
- `safe_glUniform1i(g_squareShaderState->h_uHeight, g_height);  // uniform height`
  `safe_glUniform1i(g_squareShaderState->h_uWidth, g_width);  // uniform width`
  function calls are added to `static void drawSquare()` function to bind the glsl variables (height and width of the window, in this case) to handles.
- `ss.h_uHeight = safe_glGetUniformLocation(h, "uHeight");`
  `ss.h_uWidth = safe_glGetUniformLocation(h, "uWidth");`
  function calls are added to `static void loadSquareShader(SquareShaderState& ss)` function to retrieve handles to uniform variables (which are defined in the shader program as the strings above).

asst1-sq-gl3.vshader

- `uniform int uWidth;`
  `uniform int uHeight;` uniform variable declarations added.

```
void main() {
    float ratio;
    float w = uWidth;
    float h = uHeight;
    float sign;

    if(uWidth > uHeight){
        ratio = (h/w)*abs(aPosition.x);
        if(aPosition.x == 0)
            gl_Position = vec4(0, aPosition.y, 0, 1);
        else{
            sign = aPosition.x / abs(aPosition.x);
            gl_Position = vec4(ratio * sign * uVertexScale, aPosition.y, 0, 1);
        }
    }
    else{
        ratio = (w/h)*abs(aPosition.y);
        if(aPosition.y == 0)
            gl_Position = vec4(aPosition.x * uVertexScale, 0, 0, 1);
        else{
            sign = aPosition.y / abs(aPosition.y);
            gl_Position = vec4(aPosition.x * uVertexScale, ratio * sign, 0, 1);
        }
    }

    //gl_Position = vec4(aPosition.x * uVertexScale, aPosition.y, 0, 1);

    vTexCoord = aTexCoord;
    vTemp = vec2(1, 1);
}
```

The vertex shader simply modifies the location of the vertex according the height / width ratio, thus keeping the aspect ratio constant.

Task 3 : A different, colored shape and keyboard interaction

We have, as suggested, written our own shaders, shader states, geometry for the second shape. Below are the changes:

main.cpp

- The position of the second shape is declared global, since we need to keep track of the vertex coordinates in order to redraw them when I,j,k,l keys are pressed

```
// global position
static GLfloat tpos[6] = {
    0, -.5,     //   x---x
    -.5, .25,    //   \ /
    .5, .25      //    x
};
```

- Shared pointers to geometry, shader states and the new texture are declared besides the addition of `colVbo` (the color vector) to the `struct GeometryPX` for giving each vertex a different color value.

```
static shared_ptr<SquareShaderState> g_squareShaderState;
static shared_ptr<TriangleShaderState> g_triangleShaderState;

// our global texture instance
static shared_ptr<GlTexture> g_tex0, g_tex1, g_tex2;

// our global geometries
struct GeometryPX {
  GlBufferObject posVbo, texVbo, colVbo;
};

static shared_ptr<GeometryPX> g_square;
static shared_ptr<GeometryPX> g_triangle;
```

- A new geometry for the triangle is defined. There is the color array of 9 points (3 vec3's) which is bound to array buffer after the position and texture. Texture positions are different than that of square's to fit the shield into the triangle that is drawn onto screen.

```
static void loadTriangleGeometry(const GeometryPX& g){

  GLfloat tex[6] = {
   .5, -.2,
   -.35, 1,
   1.35, 1
  };

  GLfloat col[9] = {
    1, 0, 0,
    0, 1, 0,
    0, 0, 1
  };

  glBindBuffer(GL_ARRAY_BUFFER, g.posVbo);
  glBufferData(
    GL_ARRAY_BUFFER,
    6*sizeof(GLfloat),
    tpos,
    GL_STATIC_DRAW);
  checkGlErrors();

  glBindBuffer(GL_ARRAY_BUFFER, g.texVbo);
  glBufferData(
    GL_ARRAY_BUFFER,
    6*sizeof(GLfloat),
    tex,
    GL_STATIC_DRAW);

  glBindBuffer(GL_ARRAY_BUFFER, g.colVbo);
  glBufferData(
      GL_ARRAY_BUFFER,
      9*sizeof(GLfloat),
      col,
      GL_STATIC_DRAW);

  checkGlErrors();
}
```

- Triangle shader state is declared with uniform variables of window height, width, and the texture handle and with attribute handles same as square shader state except for the color array handle.

```
struct TriangleShaderState {
    GlProgram program;

    // uniform variable handles
    GLint h_uHeight, h_uWidth;
    GLint h_uTex;

    // vertex attribute handles
    GLint h_aPosition;
    GLint h_aTexCoord;
    GLint h_color;

};
```

- A shader load function, similar, if not copy of square shader loading function, is written.

```
static void loadTriangleShader(TriangleShaderState& ss){
  const GLuint h = ss.program; // short hand

  if (!g_Gl2Compatible) {
    readAndCompileShader(ss.program, "shaders/asst1-tr-gl3.vshader", "shaders/asst1-tr-gl3.fshader");
  }
  else {
    readAndCompileShader(ss.program, "shaders/asst1-tr-gl2.vshader", "shaders/asst1-tr-gl2.fshader");
  }

  // Retrieve handles to uniform variables
  ss.h_uTex = safe_glGetUniformLocation(h, "uTex");
  ss.h_uWidth = safe_glGetUniformLocation(h, "uWidth");
  ss.h_uHeight = safe_glGetUniformLocation(h, "uHeight");

  // Retrieve handles to vertex attributes
  ss.h_aPosition = safe_glGetAttribLocation(h, "aPosition");
  ss.h_aTexCoord = safe_glGetAttribLocation(h, "aTexCoord");
  ss.h_color = safe_glGetAttribLocation(h, "acolor");

  if (!g_Gl2Compatible)
    glBindFragDataLocation(h, 0, "fragColor");
  checkGlErrors();
}
```

- The draw function for triangle is defined (again similar to square drawing function, with addition of color array handle):

```
static void drawTriangle(){
    // activate the glsl program
  glUseProgram(g_triangleShaderState->program);

  // bind textures
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_2D, *g_tex2);

  // set glsl uniform variables
  safe_glUniform1i(g_triangleShaderState->h_uTex, 0); // 0 means GL_TEXTURE0
  safe_glUniform1i(g_triangleShaderState->h_uWidth, g_width);       // uniform width
  safe_glUniform1i(g_triangleShaderState->h_uHeight, g_height); // uniform height

  // bind vertex buffers
  glBindBuffer(GL_ARRAY_BUFFER, g_triangle->posVbo);
  safe_glVertexAttribPointer(g_triangleShaderState->h_aPosition,
                             2, GL_FLOAT, GL_FALSE, 0, 0);

  glBindBuffer(GL_ARRAY_BUFFER, g_triangle->texVbo);
  safe_glVertexAttribPointer(g_triangleShaderState->h_aTexCoord,
                             2, GL_FLOAT, GL_FALSE, 0, 0);

  glBindBuffer(GL_ARRAY_BUFFER, g_triangle->colVbo);
  safe_glVertexAttribPointer(g_triangleShaderState->h_color,
                             3, GL_FLOAT, GL_FALSE,0, 0);

  safe_glEnableVertexAttribArray(g_triangleShaderState->h_aPosition);
  safe_glEnableVertexAttribArray(g_triangleShaderState->h_aTexCoord);
  safe_glEnableVertexAttribArray(g_triangleShaderState->h_color);

  // draw using 6 vertices, forming two triangles
  glDrawArrays(GL_TRIANGLES, 0, 3);

  safe_glDisableVertexAttribArray(g_triangleShaderState->h_aPosition);
  safe_glDisableVertexAttribArray(g_triangleShaderState->h_aTexCoord);
  safe_glDisableVertexAttribArray(g_triangleShaderState->h_color);

  // check for errors
  checkGlErrors();
}
```

- Keyboard callback function modified to move the triangle when the i,j,k or l buttons are pushed down:

```
case 'i':  // up
    tpos[1]     += .1;
    tpos[3]     += .1;
    tpos[5]     += .1;
    loadTriangleGeometry(*g_triangle);
    break;
case 'j':  // left
    tpos[0]     -= .1;
    tpos[2]     -= .1;
    tpos[4]     -= .1;
    loadTriangleGeometry(*g_triangle);
    break;
case 'k':  // down
    tpos[1]     -= .1;
    tpos[3]     -= .1;
    tpos[5]     -= .1;
    loadTriangleGeometry(*g_triangle);
    break;
case 'l':  // right
    tpos[0]     += .1;
    tpos[2]     += .1;
    tpos[4]     += .1;
    loadTriangleGeometry(*g_triangle);
    break;
```

asst1-tr-gl3.vshader

- Pretty much the same shader with square vertex shader, except for the color array input for each vertex, which later sent to fragment shader as varying variables.

```
in vec3 acolor;
out vec3 vcolor;
void main(){
       .
       .
       .
       vcolor = acolor;
}
```

asst1-tr-gl3.fshader

```
#version 130

uniform sampler2D uTex;

in vec2 vTexCoord;
in vec3 vcolor;

out vec4 fragColor;

void main(void) {
  vec4 color = vec4(vcolor, 1);

  // The texture(...) calls always return a vec4. Data comes out of a texture in RGBA
format
  vec4 texColor = texture(uTex, vTexCoord);

  fragColor = texColor*color;
}
```

Fragment shader takes the uniform variable texture and applies it to the triangle. The color vector is passed to fragment shader as varying variable and then assigned to the vertex by the `fragColor` variable.