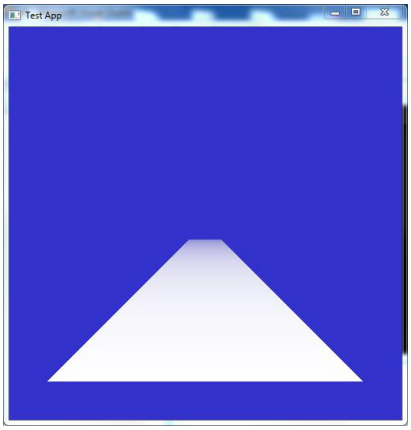
Lab 2 - Textures

Giving texture coordinates to each vertex



Set Lab2 as the startup project and run it. You will see a single colored quad drawn on screen. Now, take a look at the code and you will notice two large differences from the previous lab.

- **Assignment:** The quad is drawn as an indexed mesh. That means that instead of sending six vertices to the *glDrawArrays* function as in Lab1, a vertex buffer object containing the four vertices of the quad is created, and a second buffer object containing indices into this list is sent to the *glDrawElements* function. What is the point of this?

  _____

  _____

- **Assignment:** In the previous lab we drew the triangles essentially in 2D. Now, the four vertices are defined in three dimensional space and a projection matrix is sent along to the vertex shader. The perspective matrix is created with the following code:

  ```
  float4x4 projectionMatrix = perspectiveMatrix(fov, aspect_ratio, near, far);
  ```

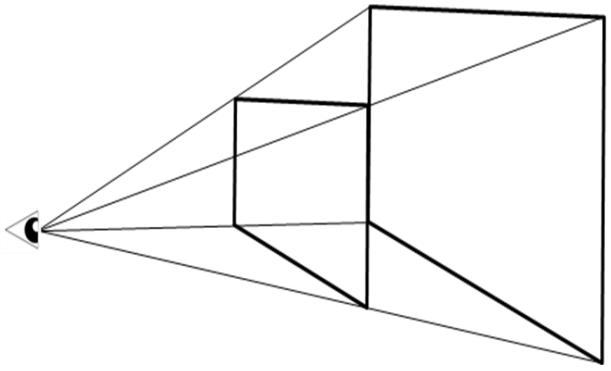
  What are each of these arguments for (draw in the picture if it will help you explain)?

  field of view: _____

  aspect ratio: _____

  near plane: _____

  far plane: _____



This quad may not look like much yet, but with a little imagination it could be the floor in a corridor of some old Wolfenstein game.Let's pretend it is. In this lab, we are going to make the floor look better by mapping a *texture map* to the floor

The first thing we need to do is to give each vertex a *texture coordinate* (often called *uv-coordinate*). This is done the same way as giving a vertex a position or a color. For starters, we will simply map the corners of the texture map to the corners of the quad. Just below where the vertex positions are defined, define the texture coordinates:

```
float texcoords[] = {
      0.0f, 0.0f,              // (u,v) for v0
      0.0f, 1.0f,              // (u,v) for v1
      1.0f, 1.0f,              // (u,v) for v2
      1.0f, 0.0f               // (u,v) for v3
};
```

Now we need to send these coordinates to the graphics board. Generate a buffer that will be used by OpenGL to store the texture coordinates (read more about buffer objects in the OpenGL 3.0 spec, §2.9):

```
glGenBuffers( 1, &texcoordBuffer );
glBindBuffer( GL_ARRAY_BUFFER, texcoordBuffer );
glBufferData( GL_ARRAY_BUFFER, sizeof(texcoords), texcoords,
     GL_STATIC_DRAW );
```

The variable `texcoordBuffer` (of type GLuint) is a number that identifies the buffer object. Don't forget to declare it together with the other buffer objects.
We will also need to access the texture coordinates in the vertex and fragment shaders. Locate the place where the indices of the vertex position and vertex color attributes are set, and add one for the texture coordinate attribute (read more about vertex attributes in the OpenGL 3.0 spec §2.20):

```
glBindAttribLocation(shaderProgram, 0, "position");
glBindAttribLocation(shaderProgram, 1, "color");
glBindAttribLocation(shaderProgram, 2, "texCoordIn");
```

We need to instruct OpenGL to take the values for `texCoordIn` from the `texcoords`-buffer that we created above. Do this by adding, near the end of `initGL()`:

```
glBindBuffer( GL_ARRAY_BUFFER, positionBuffer );
glVertexAttribPointer( 0, 3, GL_FLOAT, false, 0, 0 );
glBindBuffer( GL_ARRAY_BUFFER, colorBuffer );
glVertexAttribPointer( 1, 3, GL_FLOAT, false, 0, 0 );
glBindBuffer( GL_ARRAY_BUFFER, texcoordBuffer );
glVertexAttribPointer( 2, 2, GL_FLOAT, false, 0, 0 );
```

The first parameter tells that attribute with index 2 will get its values from texcoordBuffer. The second and third parameter tells that each element consists of two floats. In addition, we need to enable the texcoord-array. Do this by adding:

```
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glEnableVertexAttribArray(2);
```

Loading the texture

To read a texture image from a file, we use **DevIL**, which is an open library for reading images of most formats (see http://openil.sourceforge.net/docs/index.php). We need to include ilutil.h and initialize DevIL. This latter is already done at the top of function initGL() by the two calls to:

```
ilInit();    // initiate devIL (developers Image Library)
ilutRenderer(ILUT_OPENGL); // initiate devIL
```

At the end of function initGL() is a good place to read the texture from file and hand it to OpenGL, so add this line:

```
texture = ilutGLLoadImage("floor.jpg");
```

Note that `texture` is a global variable, declare it at the top of main.cpp (it shall be an GLuint). The function `ilutGLLoadImage` returns an id, internally generated by calling `glGenTextures(1,&texture)`, (read all about textures in OpenGL 3.0 spec §3.9).

Drawing the floor with texture

In the `display()` function, we need to make sure that texture unit 0 is associated with our texture (which has its texture id stored in variable `texture`). Therefore, add this call, just before the call to glDrawElements():

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);
```

In the vertex shader, we need to declare a variable for the incoming texture coordinate, `texCoordIn`, and also a variable for the outgoing interpolated texture coordinate to the fragment shader. Add these lines to the top of the vertex shader in file *simple.vert*:

```
in    vec2  texCoordIn; // incoming texcoord from the texcoord array
out   vec2  texCoord;   // outgoing interpolated texcoord to fragshader
```

At the bottom of the `main()`-function in the vertex shader, add

```
texCoord = texCoordIn;
```

to send the texture coordinate to the fragment shader. For the fragment shader, the value in `texCoord` will be interpolated from the four values of `texCoord` that are output by the vertex shader for the four vertices of our quad (vertex, and other, shaders are described in the glsl 1.30 spec, and the usage is covered in OpenGL 3.0 spec §2.20).

In the fragment shader (file *simple.frag*) add a so called *sampler*. I.e., inform that the shader will use a texture that is connected to texture unit 0. In addition, add the variable for the incoming texture coordinate from the vertex shader. You do this by adding

```
uniform sampler2D colortexture;
in vec2 texCoord;
```

to the fragment shader. (Add to an empty line below the line containing `precision highp float` in file *simple.frag*). Also, replace

```
fragmentColor = vec4(outColor, 1.0);
```

in `main()` with

```
fragmentColor = texture2D(colortexture, texCoord.xy);
```

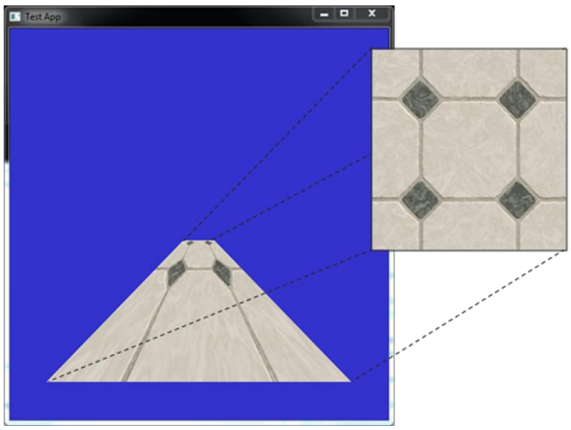
in order to fetch the color from the texture, instead.

We need to set the value of "colortexture" to zero, so that colortexture corresponds to texture unit 0 (a value of 1 will associate it with texture unit 1 and so on…). Do this, by adding the following two lines to the end of function initGL():

```
// Bind the shaders
glUseProgram( shaderProgram );
// Get the location in the shader for uniform "colortexture"
int texLoc = glGetUniformLocation( shaderProgram, "colortexture" );
// Set colortexture to 0, to associate it with texture unit 0
glUniform1i( texLoc, 0 );
```

Fragment, and other, shaders are described in the glsl 1.30 spec, and the usage is covered in OpenGL 3.0 spec §3.12.

Now run the program again and enjoy the textured floor. Not exactly what we wanted is it? The floor is much longer than it is wide (20 units wide and 300 units long to be exact) but we map the square texture one-to-one on the quad, so the texture will be very stretched out. Instead, lets repeat the texture several times in the z-direction. Change the texture coordinates as follows:

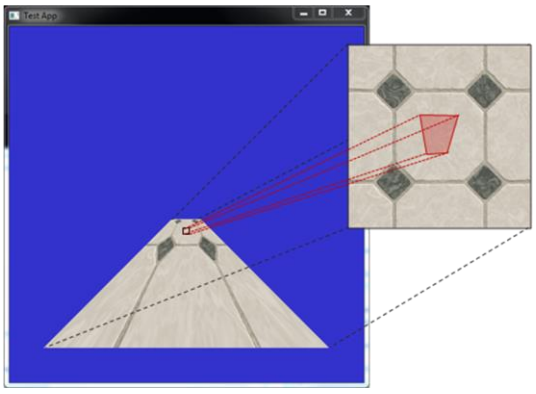


```
float texcoords[] = {
      0.0f, 0.0f,
      0.0f, 15.0f,
      1.0f, 15.0f,
      1.0f, 0.0f
};
```

Then we need to tell OpenGL what to do with texture coordinates outside the (0,1) range. Add the following lines right after loading the texture:

```
texture = ilutGLLoadImage("floor.jpg");
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);
//Indicates that the active texture should be repeated,
//instead of for instance clamped, for texture coordinates >1 or <0.
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

The call to `glActiveTexture(GL_TEXTURE0)` specifies that the following commands affect texture unit 0. The call to `glBindTexture(GL_TEXTURE_2D, texture)` associates texture unit 0 with the image with id= `texture`. The next two lines tell OpenGL that texture coordinates above 1.0 or below 0.0 shall simply wrap around.
Run the program again.



Texture filtering

To improve the image we are now going to look at texture filtering. A problem we have run into is that parts of the texture are being shrunk to a much smaller size than the original image. So, when fetching a color from the texture for a pixel, the texture coordinate could actually map to any texel in a large area on the texture, but it will simply pick the texel on the texture coordinate that the middle of the pixel maps to. A simple way to improve the result is to use mipmapping.
Right after the glTexParameter calls, add the lines:

```
glGenerateMipmap(GL_TEXTURE_2D);
// Sets the type of filtering to
// be used on magnifying and
// minifying the active texture. These are the nicest available options.
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR_MIPMAP_LINEAR);
```

The call to `glGenerateMipmap()` generates the mipmap levels on the bound textures. The following two lines tell OpenGL to use the mipmap when minimizing a texture and to do linear filtering when magnifying. Run the program again.

You will notice that the image is now a lot less noisy, but that its mostly a blur far away. Add this line as well:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, 16.0f);
```

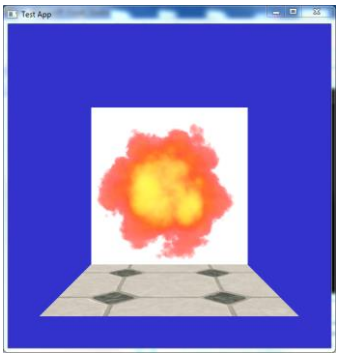
which enables an extension, not part of core OpenGL 3. Nevertheless, this extension is available on your machines (this and other extensions are listed in the OpenGL registry). It enables the nicest level of anisotropic filtering. (Note that `glTexParameter` sets state per texture object and does so for the currently bound texture – in our case `texture` which is bound by the call to `glBindTexture` above.) What happens with the final image and why?

Transparency

No fake wolfenstein scene is complete without an explosion. We will create one by drawing a new quad and texture it with a picture of an explosion. Create a new vertex array with data for positions and texture coordinates so it looks something like the image below.

Also load a new texture ("`explosion.png`") and use it when drawing the new quad.

When you are done, you should see something like the image to the right. This texture has an *alpha* channel as well as color channels. The alpha channel simply says how transparent each texel is. To get the transparent effect, we will have to enable *blending* (see OpenGL 3.0 spec §4.1). Add the following lines in function `display`() somewhere before the call to your second `glDrawElements` ();. The first line enables blending. The second specifies how the blending should be done. With these parameters the destination will receive an interpolated color value of *α*source color + (1- α)*destination color*, which faithfully corresponds to the behavior of transparency.



```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Note that to get correct transparent behavior, all overlapping transparent objects/polygons should be rendered in back-to-front order. It is common to first render all non-transparent objects in any order, and then sort all transparent objects/polygons and render them last. This transparency calculation does not require the presence of any alpha channel in the framebuffer.

Now run the program again and enjoy the exciting scene.

When done, show your result to one of the assistants. Have the finished program running and be prepared to answer some questions about what you have done.