CMPUT 411/511—Computer Graphics

Assignment 4

Fall 2017
Department of Computing Science
University of Alberta

Due: 23:59:59 *Friday, December 8*
Worth: 20% of final grade for CMPUT 411
(15% of final grade for CMPUT 511)
Instructor: Dale Schuurmans, Ath409, daes@ualberta.ca

(Based on an assignment initially developed by Nathaniel Rossol.)

In this assignment, you will learn how to implement shaders in OpenGL (specifically, in GLSL). Shaders are basically mini rendering programs that run directly on the GPU and override OpenGL's default rendering pipeline. By doing so, one can achieve special material effects that are not possible with the default OpenGL pipeline. Shaders exploit the capabilities of GPUs and therefore can render complex effects very efficiently. By the end of this assignment, you will be able to implement a wide range of special material effects by using shaders.

Note A zip archive containing the directories and files you will need to complete this assignment are posted on the assignment web page in a4files.zip.

Note When finished, please submit a *single* .zip archive containing your updated version of the "student_code" directory (only) on **eclass**.

Question 1 – GLSL set up (2%)

First you will cover the steps needed to get a GLSL shader from a GLSL text file to creating a material effect rendering in real time. For reference, please consult the example `HelloSquare` in "a4files".

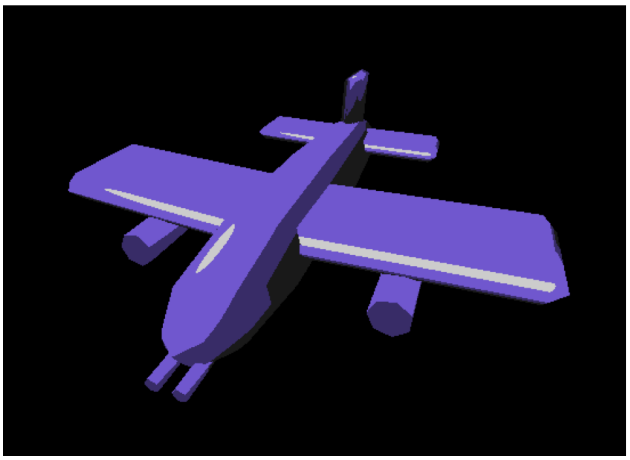
First, inspect "shader.h" in the "student_code" directory and look for the function "setShaders(char*, char*)". This is the function you will complete, using the following pseudo code as a guide.

```
Step 1: load and compile the vertex and fragment shaders

>> Create new handles for the vertex and fragment shaders
>> Read the shader programs into C strings
   (from "student_code/toon.vert" and "student_code/toon.frag" respectively)
>> Load the C strings to become the shader sources
>> Free the memory used by the C strings (since they have been copied to OpenGL)
>> Compile the vertex and fragment shaders
>> Verify that the compilations succeeded

Step 2: link and install the shading program

>> Create a new handle to the shader program
>> Attach the vertex and fragment shaders to this program
>> Link the shader program
>> Verify that the linking succeeded
>> Detach the vertex and fragment shaders (since they have already been linked)
>> Enable the shader program
```

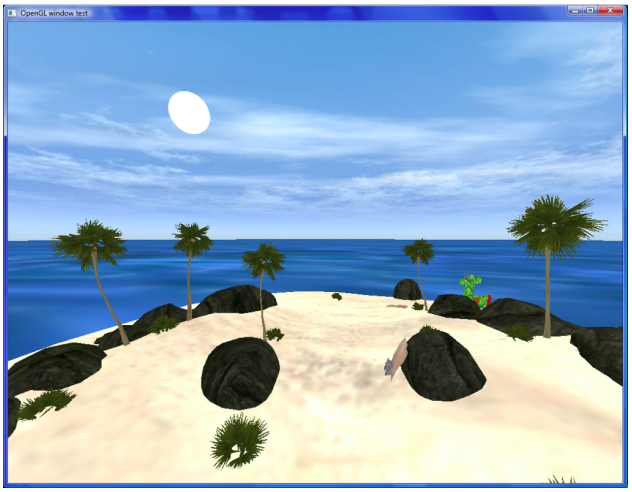


For your convenience use the function "readTxtFile(char *)" included in "shader.h", which takes in a filename and returns a C String (char*) that contains all of the text from the file. Use this function to read in the shader files. You can also use "glVerifyHandle(GLuint)" to verify each of the compilation and linking outcomes. Compile the overall program by running "`make Question1`".

If you implemented everything correctly, the program should run and produce an image of a rotating airplane with a toon shaded appearance. (If your implementation didn't work you will just see a grey scale airplane.) Note that you can use the same animation and camera controls as Assignment 3 to control the animation.

Question 2 – Color, texture and animation with GLSL

Consider the source code provided for Question 2. Run "`make Question2_3`". After compiling and running the program, you should be presented with an image from the following scene (assuming you have successfully completed Question 1).



To navigate in this scene, use the same camera controls as in Assignment 3.

This will be the starting scene. It might not seem so bad at first, but there are many issues if we look closer. First, the scene does not have any lighting; everything is lit to the same extent regardless of where the sun is (use the "LeftArrow" and "RightArrow" keys on your keyboard to move the sun across the sky). Also, some of the colours for things like the sun could use tweaking, and the sand texture on the island is very stretched out—maybe blending it with some grass would help too? All this and more will be accomplished through the use of shaders through the course of this assignment.

Note: Questions 2 and 3 involve modifying the code in "student_code" only! **Do not modify the C++ source code!** It should not be be submitted since it will not be marked.

(a) (1%) **Water - texture and animation**

First we consider working with the water material. Even though its texture file has no alpha channel, we can achieve translucency through use of a shader. Also, most water on the ocean is not perfectly still, so we will use a shader to animate the water as well.

This part is a tutorial, so simply follow the instructions to receive full marks.

First, look inside "a4files" for a directory named "student_code". All the shaders for this assignment are already placed there. Do not rename any of the shader files or they will stop working. Once you have completed the assignment, you'll be zipping-up and submitting the "student_code" directory. Inside the "student_code" directory look for the file called "oceanWater.vert", which contains the following vertex shader program:

```
uniform float currentTime;

void main() {
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Ignore the line "uniform float currentTime" for now (we will use it later). You can see that this shader program is extremely simple; only two lines of code. The first line says that the texture coordinates for this vertex should just be whatever was passed to the shader from the OpenGL application (i.e. this shader program does not modify the texture coordinates at all). The second line says that this vertex's position should be whatever position was passed from the OpenGL application multiplied by the current Modelview Matrix and Projection Matrix. You might be wondering how all these variables and arrays are being used without being defined. The answer is that any variable that begins with "gl_" represents a special variable you can obtain from the OpenGL application. For example, gl_Vertex gets the current vertex being rendered, and gl_ModelViewProjectionMatrix gets whatever the current Modelview and projection matrices are (multiplied together). gl_ MultiTexCoord0 gets the texture coordinates for the current vertex (i.e. the first set of coordinates just in case the vertex has multiple texture coordinates—something we will not be getting into in this assignment).

gl_Position is one of the "output" attributes that the vertex shader passes along to its corresponding fragment shader: the location we assign to gl_Position determines where the fragment will ultimately be rendered on the viewport. Likewise, the value assigned to gl_TexCoord[0] will be the location in the first layer of textures used by this vertex (i.e. you would use gl_TexCoord[1] to set the second layer of texture coordinates, but we will not get into that; for this assignment, all meshes only have one set of texture coordinates).

You might now be asking: "Why do we need a vertex shader program to transform the vertices when OpenGL already performs the same transformations anyway"? The reason you are re-implementing the vertex shader is that if you want to use any shaders for a scene object, you have to go all-in (for that object). You cannot just write a fragment shader for an object, you have to write both a vertex and fragment shader together. This means that even if we do not want to do anything fancy with the vertices, we still need to write a vertex shader like the one above to mimic what OpenGL would normally do by default otherwise all our vertices will remain untransformed (and be clustered around the world origin).

Next, open the fragment shader for the water named "oceanWater.frag":

```
uniform sampler2D textureSample_0;
uniform float currentTime;

void main() {
    gl_FragColor = texture2D(textureSample_0,gl_TexCoord[0].st);
}
```

This shader is even simpler, only a single line of code! gl_FragColor is the single output of the fragment shader, and it is simply a vector with 4 elements (Red, Green, Blue, and Alpha Opacity). Here we use the special built-in function "texture2D()" to retrieve the appropriate texture pixel (aka. "texel") from the texture and then simply say that this pixel should be that colour. Again, we are just doing exactly what OpenGL would do if we were not using shaders (and had lighting disabled).

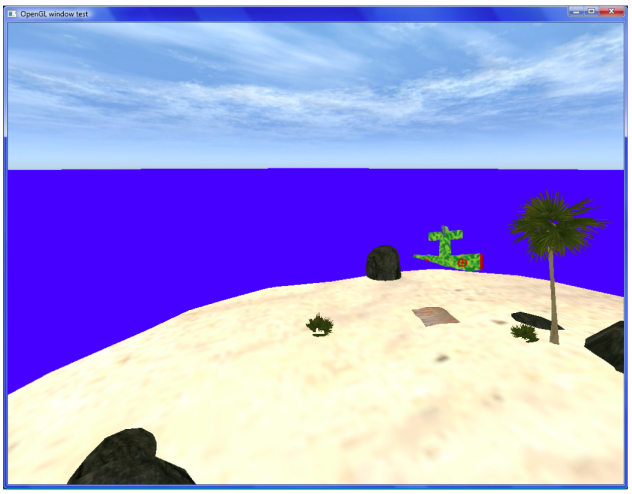
Where does gl_TexCoord[0] come from? Recall that we set this value in the corresponding vertex shader, and since the vertex shader always runs first, we can pass data from the vertex to the fragment shader, but obviously not the other way around.

Where does our textureSample_0 come from? As you can see, it is a uniform variable so that means that it is passed from our OpenGL application to the shader. Specifically, in this case, the program has already loaded "water.tga" from the "textures" directory and passed it to this shader. Now, we will try working with the shader a bit. Say that in the fragment shader you want the water to turn solid blue, then you can write:

```
uniform sampler2D textureSample_0;
uniform float currentTime;

void main() {
    gl_FragColor = vec4(0.0,0.0,1.0,1.0);
}
```

with the result:



If you do not obtain this result, check if any shader compilation error has occurred. Also, make sure you save your shader file before running your OpenGL application again. As expected, the shader program now ignores the texture and simply paints the water mesh completely blue regardless of any texture coordinates. Now put the code that draws the texture back in. You may have noticed that the water is fully opaque, even though in reality water is normally translucent. However, this particular water texture does not have an alpha-channel (you can open it up in Gimp to see for yourself) so it is rendered as fully opaque. In the shader, we can explicitly set the alpha value for each pixel as follows:

```
uniform sampler2D textureSample_0;
uniform float currentTime;

void main() {
    gl_FragColor = texture2D(textureSample_0,gl_TexCoord[0].st);
    gl_FragColor.a = 0.75;
}
```

The water should now look like this:

In the OpenGL application, alpha blending has been enabled for the water mesh, so we now have textured, translucent water. However, note that not all of the scene objects have been set up to use alpha-blending. For example, the trees and leaves use Alpha-test instead, so not all objects in the scene will support translucency like this—fair warning.

Now that our water is looking all nice and translucent, it is time to animate it! How might this be done? Recall that the texture coordinates control which part of a mesh receives which part of a corresponding texture. Thus, if we constantly change the

values of the texture coordinates, we can continually change which parts of a texture are being rendered on the mesh. The code below accomplishes this.

Note: This code is in the vertex shader "oceanWater.vert".

```
uniform float currentTime;

void main() {
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_TexCoord[0].s += currentTime*0.003;
    gl_TexCoord[0].t += currentTime*0.003;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Recall that in order to translate a point, we simply add another vector or scalar value to it. Here we are adding the scalar value "currentTime*0.003" to both the s and t components of the texture coordinate point. The result is that the texture coordinates will start translating continuously as currentTime is getting larger every frame (note that the value of gl_TexCoord[0] is reset at every frame however, same goes for any output variable: they only last for any particular vertex for a single frame).

Finally, the mysterious "currentTime" variable is used! Note that this variable is not some standard openGL variable (nor part of the GLSL language) because if it was, it would begin with "gl_". This variable is a "uniform" which means that it is a user defined variable passed to the shader from the OpenGL application. In this case, the OpenGL application for this assignment has been setup to pass a variable called "currentTime" to several shaders. As you might expect, this variable represents the current time—specifically the number of seconds that have passed since the application started. This variable is updated every frame and transferred to the fragment shader once every frame so that it can be used for animation effects on certain shaders.

Finally, our water shader is complete. If you followed along correctly, your water should now be gently animating. You can tweak the value of the constant "0.003" to make the water go faster or slower or even in different directions.

(b) (1%) **Radio buttons - color**

Now, it is time to finally to write your own fragment shader! Run the application and look at the emergency transponder radio in the middle of the island (remember, use the same camera controls as Assignment 3 to move and look around). You should see two perfectly white buttons near the bottom of the device. Your goal is to write a shader to make these buttons green instead. Write your solution in the file named "buttons.frag". Do not forget to save it before running the application again.

The result should look like the following.

You can make your buttons any shade of green you like, but make sure that they are fully opaque.

(c) (2%) **Sun - color and animation**

The next task will be to upgrade the material the sun mesh is using in the sky. The OpenGL application has been set up such that, as the sun moves across the sky, the colour of the sun's directional light gets redder and redder the closer the sun is to the horizon. Your goal for this part of the assignment is to write a shader (in "theSun.frag") that queries what the current *diffuse* colour is for `LIGHT0`, and then colours the sun to match it. As a result, your sun should get redder and redder the closer it is to the horizon (remember, you can use the "LeftArrow" and "RightArrow" keys to move the sun across the sky). The result should look like the following when the sun is low in the sky.

Note that the sun should get more and more white the closer it is to the top of the sky. The sun is using GL_LIGHT0 in the application so if you wanted to access this light's

properties, you would use the reserved shader variable gl_LightSource[0]. Specifically, if you wanted the light's diffuse colour (which is what we need for this part) you can get it as a vector (of size 4) via `gl_LightSource[0].diffuse`.

If you are curious about the other properties of the lights that can be queried by GLSL shaders, this site lists them all `http://www.lighthouse3d.com/tutorials/glsl-tutorial/lighting`. We will need some of these other properties for later questions.

Note: This question is doable in a single line of code.

(d) (2%) **Sky - color and animation**

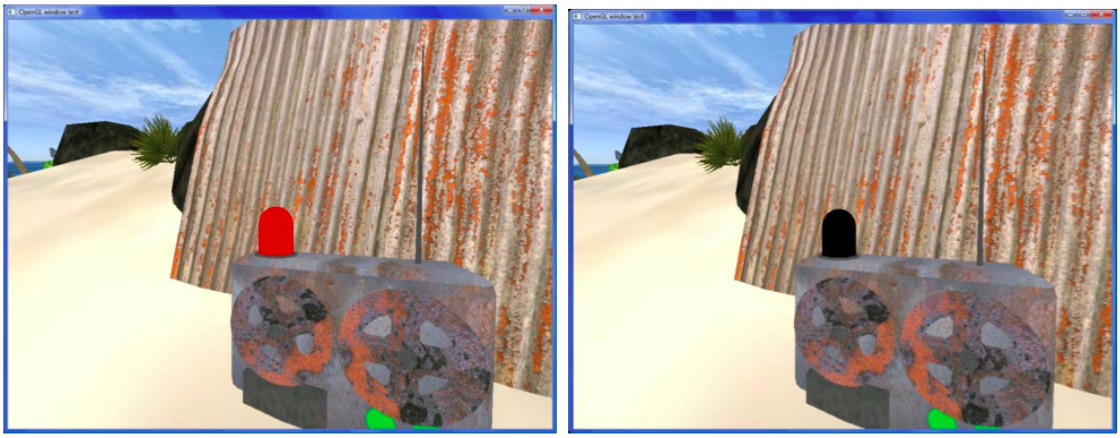
Okay, so the sun changes its colour now, but the scene still looks bizarre because the sky does not change colour with it, nor are any of the objects in the scene are affected by the supposedly changing sun. In this part, your goal is to have the sky change colour to match the sunlight colour as it changes across the sky. Write your shader code in "skyShader.frag". The result should look like the following image.

Naturally, the sky should be almost completely normal when the sun is high in the sky if you implemented this correctly.

Note: This question is also doable in a single line of code.
(*Hint*: Multiplication is involved.)

(e) (1%) **Radio light - color and animation**

Next, take another look at that emergency transponder radio in the middle of the island. You will notice that it has a white bulb on top of it. We need to make this bulb red instead. However, we also want to show the machine is still active, so we will have the red bulb continuously blink on and off. Your goal is to write a fragment shader that makes the bulb red and continuously blink on and off. Write your shader code in "beaconLight.frag". Your result should look similar to the following result.



Obivously, this is another animated shader, so the "currentTime" uniform variable has been made available for your use again.

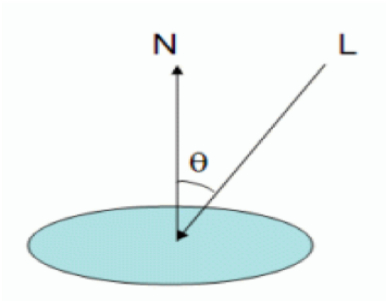
Note: The brightness of the light needs to oscillate over time. If you graph the brightness of the light over time it would look a lot like a *sine wave*. Therefore, recall that functions such as sin(), cos(), and dot() are all built-in functions of GLSL and work as you would expect.

(f) (2%) **Diffuse lighting**

Now it is time to address the lighting (or rather lack thereof) that is hampering our scene. In this part, your goal is to implement basic diffuse lighting for the objects in the scene. Most of the objects in the scene use the shader file pair "diffuseTexture.vert" and "diffuseTexture.frag". Your goal is to implement the standard OpenGL-style diffuse lighting and put this code in "diffuseTexture.vert" and "diffuseTexture.frag". When finished, many scene objects such as the rocks, airplane, trees, and plants will now be diffusely lit. The result should look like the following image.

Objects like the airplane and trees and rocks should all be affected (unfortunately, not the island or the metal sheets yet). Use the "LeftArrow" and "RightArrow" keys to move the sun across the sky and test the lighting from various angles.

Note that for this section, we are not yet using specular lighting, only *ambient and diffuse lighting*. As such, the intensity found will simply be a result of the dot product between the light source (which is directional in this case) and the normal of the vertex, as follows.



This tutorial explains the concept in more detail and provides some GLSL shader code that might be a good starting point `http://www.lighthouse3d.com/tutorials/glsl-tutorial/directional-lights-i`.

Do not forget to *add the effects of the ambient light* which has been set to vary with the sun angle too (note that *add* is meant literally—hint). None of the polygons facing away from the sun should be pitch-black because the scene has ambient lighting in it. Note that we are using coloured lighting here, so make sure you do not assume that the light colour is pure white.

Note: Your solution for this part is doable in about 4 or 5 lines of code in the vertex shader, and about 2 lines in the fragment shader.

Question 3 – Light and terrain mapping with GLSL

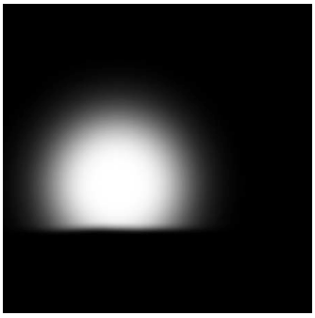
Now it is time to apply everything you have learned to implement a few more challenging (but interesting) shader problems. These are common shader tricks used in industry to achieve a wide range of effects and performance/visual optimizations. As with Question 2, we are still just using the shader files provided in the "student_code" directory. Do not modify any C++ code. Just fill in your shader code into the shader files as each question tells you to. At the end of the assignment you will finally zip up and submit "student_code.zip".

(a) (3%) **Light mapping**

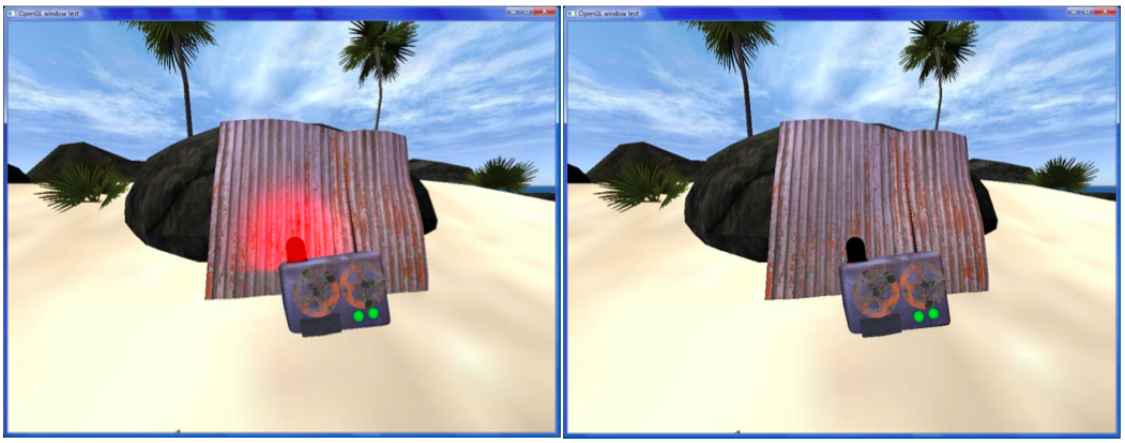
Lightmaps are simply textures that represent the lighting that is hitting a surface. For example, the image on the left below shows a scene in a game rendered using only its lightmap textures, and not its diffuse textures.

The image on the right shows the same scene with the diffuse texture combined with the lightmap. As a result one can achieve shadows and other effects without the need to perform complex lighting calculations for every pixel every frame, which makes for a huge amount of rendering optimization during runtime (at the cost of space).

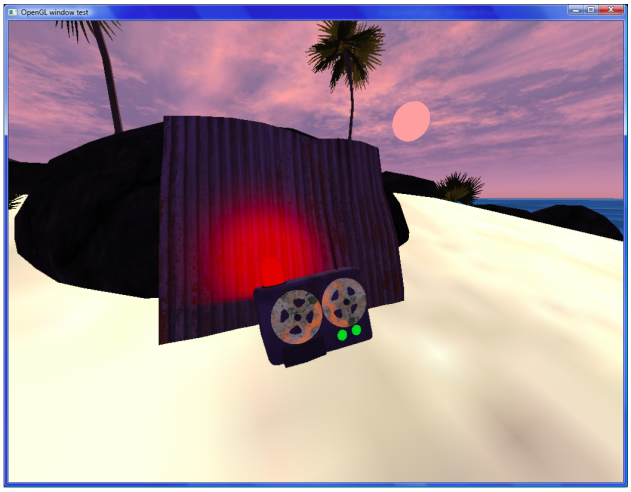
In our scene, since the sun moves across the sky, it is probably not a good source for a lightmap because lightmaps require non-moving lights. However, our emergency radio transponder beacon has a red light on it that does not move. The lightmap for this radio beacon light has already been provided to you, and looks like the following image.



Your goal in this question is to use this lightmap to make the red light shine on the metal sheet behind it via use of the lightmap provided. Write your shader code in the file "lightmappedMetal.frag". Note that the metal should also be diffuse-lit as well so you can copy-paste your code from what you wrote for "diffuseTexture.vert" and "diffuseTexture.frag" and use that as a starting point. The lightmap should become red, and blink on and off so that it matches the blinking frequency of your beacon light. The result should look like this:



Note that the red light shine should be the same brightness regardless of the diffuse light. That is, the red shine comes directly from the red beacon so the amount of red light hitting the metal should be independent of the diffuse light around it. For example, when the sun sets behind the metal sheet, the red shine should still be quite bright (i.e. it is independent of the diffuse light, it is a completely separate light source).



Note that in "lightmappedMetal.frag" there are 3 user-defined uniform variables provided to you already by the OpenGL application that you will need to use.

```
uniform sampler2D textureSample_0;
uniform sampler2D textureSample_1;
uniform float currentTime;
```

The first one is the metal sheet texture, and the second texture sampler is the lightmap for this sheet. The third variable is naturally still the number of seconds elapsed since the application started.

Hint: It might help to think of the red light as an additional diffuse light that is already 100% fully aligned on the normal of the metal sheet for every pixel when computing the final combined lighting of the diffuse lighting from the sun and the red light (i.e. this means that you do not need to do a dot product, or anything fancy like that, plain-old addition is all you need).

Note: Your solution should probably be around 5 or 6 lines of code in the main body of both the vertex and fragment shaders if done properly.

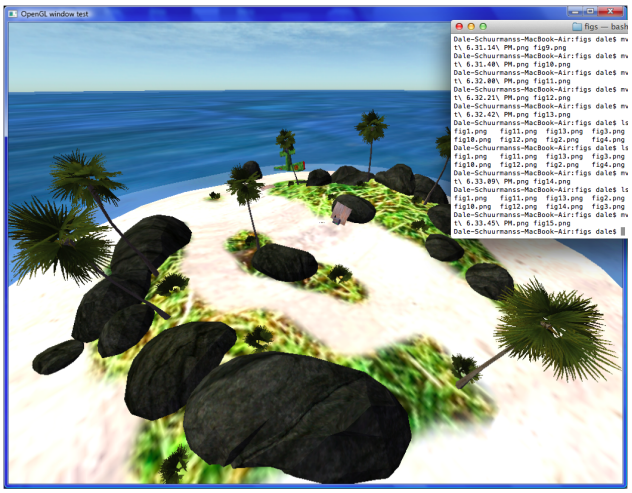
(b) (3%) **Terrain mapping**

This problem might be the most difficult, so you might want to save it for last. Try to build your solution iteratively, one step at a time.

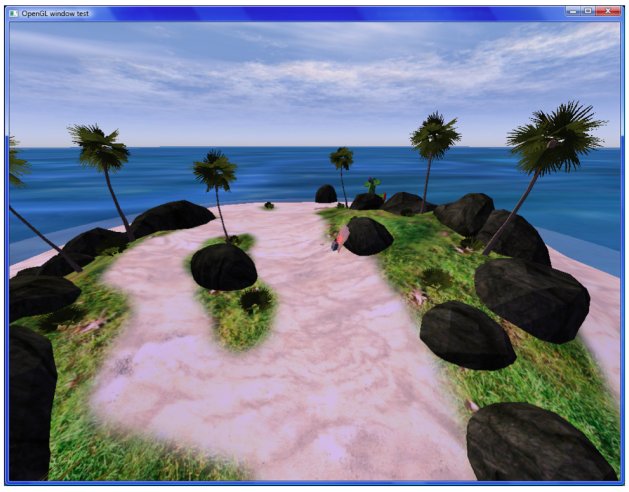
Many commercial games have large terrain sheets that form the base of their large outdoor environments. These terrain sheets are usually not textured with only a single texture, but rather several textures combined. In this question, you are going to attempt to apply multiple textures to the island that is in the middle of our scene, to give it some grass. However, we only want the grass on the top part of the island, not in or near the water (grass underwater would look weird). Fortunately, the grass texture we will be using has an alpha-channel that defines exactly where the grass should appear on the island terrain:

The *white* parts of the alpha mask (shown on the image on the right) should be mapped only with the *grass* texture, and the *black* parts define where the *sand* should

be. Naturally, gray areas (i.e. on the boundaries of the white and black areas) will have a little bit of both (but *only* the gray areas may share both textures). When you combine them together, the result should look like this:



While the grass is in the right place, as we can see it is horribly stretched out. This is because the *texture coordinates are too clumped-together* for this mesh. We need to spread them out (i.e. *scale them—hint*) to make them farther apart which will increase their tiling (for both the sand and the grass). The final result should be grass and sand that is no longer blurry and stretched, but nicely tiled across the island and still affected by diffuse lighting normally (see image below).



(Note: the sun is near the horizon to help show the sand details.)

You will need to write your shader code to do this in both "islandTerrain.vert" and "islandTerrain.frag". Note that in "islandTerrain.frag", "`uniform sampler2D textureSample_0`"

is the sandy texture for your island, and "`uniform sampler2D textureSample_1`" is your grassy texture, which has the alpha channel for blending the two textures together (the sand texture has no alpha channel at all).

Finally, note that when you scale the two texture coordinates, you need to make sure that you somehow use scaled coordinates for the RGB channels of the grass, texture, but somehow map the alpha channel (which should **not** have its tiling increased) using the original texture coordinates. Try to think carefully about how this might be done. (*Hint*: you can pass as many varying variables from the vertex shader to the fragment shader as you want). Do not forget that your island shader material needs to be affected by diffuse lighting just like the rest of your materials!

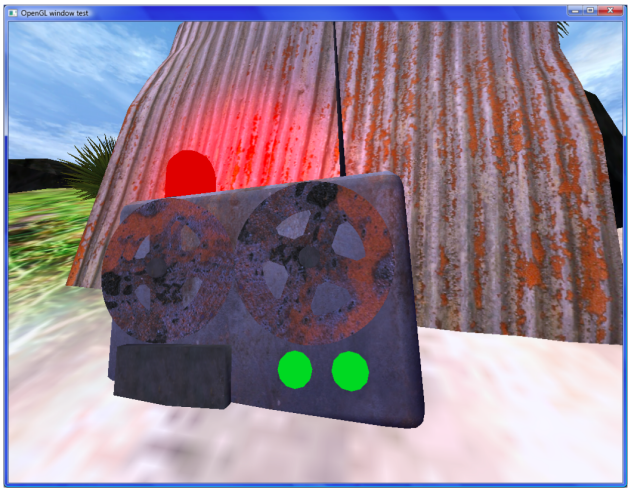
Note: You final solution should probably be about 6 to 7 lines of code in the main function of the vertex shader, and about 6 or 7 lines in the main function of the fragment shader as well.

(c) (3%) **Animated texture mapping**

A lot of graphics simulations make use of objects that spin (e.g. the propellers on an airplane, or the wheels on a vehicle). However, getting all the wheels on a vehicle or all the propellers on a plane to rotate all together using dynamic meshes can be tricky, because the vehicle itself could be moving, and a complicated rotation matrix stack may need to be built to make sure that the rotation is in line with the vehicle. Even if the wheels or propellers just spin on a non-moving vehicle, it would not be possible to light-map a moving mesh, or cache it in an OpenGL draw list to boost performance.

Therefore, a clever trick would be to *not* have the mesh itself move, but simply give it an animated material that rotates instead. Now its vertices could be cached in drawlists because they never move.

Remember how that emergency transponder beacon had two old rusty tape spindles on it? Now that the beacon has been activated, we should have the spindles slowly rotate to indicate that an emergency message is playing.



Note that, once again, the spindles should also be affected by diffuse lighting, use your code from the diffuse lighting question as a starting point.

The alpha channel for the metal texture (i.e. `uniform sampler2D textureSample_0`) that the spindles use looks like this:

The spindle meshes have been mapped around the origin so that the spindles appear round. In this case, you will want to transform all the channels of the texture by the same amount.

Hint: You will need to rotate the texture coordinate points around the texture's origin—do you remember what matrix to use to rotate points in a 2D space like this?

Write all your shader code for this part in "tapeSpindles.vert" and "tapeSpindles.frag".