COMP 2501 - Winter 2015 Tutorial #5
OpenGL: Displaying Text to Screen

Description:

The purpose of this tutorial is to show how to render text to a screen in OpenGL as well as how to use instancing to render multiple copies of an object.

Instructions:

Overview

Having text on the screen is useful not only for games but also for debugging purposes. Because OpenGL has no built-in way to simply print text, we will use a texture that contains all the characters and select from it the letters we want.

Another important fact when performing any rendering is reduce the number of times we send vertex array objects to the shader. When attempting to draw multiple characters of a string, we can bind the vao once and then use a loop to update the uniform variables and then draw. This way we only send the model information once but can draw reuse the model to draw as many characters as we want.

Task1

Look at textShader.vert and textShader.frag. These work similarly to the textureShader files, with an addition in textShader.frag that discards any pixels with an alpha value that is too low. This allows us to use images with a transparent background. There is also an addition in textShader.vert of a uniform variable `letter` that keeps track of which letter is being drawn, and the texture coordinates are adjusted to that character's position in the image.

Open Text.h. `Text` has been set up similarly to `Quad.h`, with an `initGeometry` function and a `render` function. Look at `initGeometry` - you'll notice that, unlike in `Renderable`, the vertices, indices, and texture coordinates are defined here. The vertices define a 1x1 square about the origin(0,0,0). The texture coordinates are defined in 1/128ths of the texture (because each character is one of 128 ASCII characters in the texture).

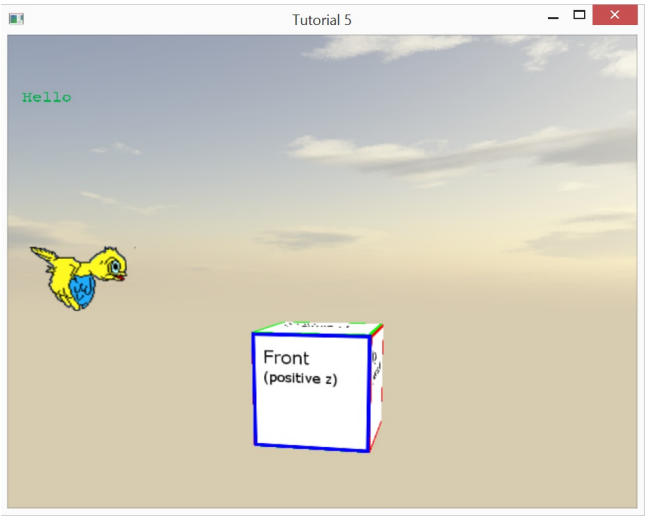
Next, go to the `render` function. You will need to create a scale matrix using `scaleMatrix(Vector3(letterWidth, letterHeight,1))` function. We won't be

using depth, so the z parameter can be 1. Create an MVP matrix defined by the product of the `parent` matrix, the translation matrix of the `position`, and the `scale`. Then calculate the character's position in the texture (`letterPosition`) using the `letter` and `IMG_RATIO` (the width of one character). Further down you can see how the `mvpMatrix`, `letterPosition`, and `color` are being sent to the uniform variables.

Go to Main.cpp `initObjects` function and take note of how where we are instantiating the `text` variable. The letter width is being set to 10 pixels and letter height to 15 pixels, the position and color of the text is being set in following calls.

Now go to the `renderWin1` function and find the `text->render()` call. One important aspect of rendering text is that we want the text on the screen no matter where we are looking. This can be done by building an orthographic projection matrix and ignoring the view matrix (or camera). Because we want to deal with a 2D surface to draw on we will use the `createOrthographicProjectionMatrix(win.width, win.height, -1, 1)`. This will create a view from (-width/2, -height/2) to (width/2,height/2) with a z range from -1 to 1. Now anytime we want to render to a heads up display(HUD) we will use this matrix.

Compile and run the code. You should see an 'H' place on the upper left of the screen (see image below). Now Alternate between modifying the `text` position using `text->setPosition()`, and calling `text->render()` to print out "Hello". Note that now you can move the camera and the text will stay in place.



Task2

Rendering one letter at a time is a bit wasteful and tedious to write. So let's modify `text.render` to print a string instead of a single character. First change the function parameter to take a `std::string message` instead of a `char`. Now create the following for loop at the end of the function:

```
for (int i = 0; i < message.length(); i++){
```

```
}
```

Now move the following lines inside the loop:

```
mvpMatrix = parent * translation * scale;
letterPosition = message[i] * IMG_RATIO //change letter to message[i]
glUniformMatrix4fv(mvpMatrixLoc,1,GL_TRUE, (GLfloat *)&mvpMatrix);
glUniform1f(letterLoc, letterPosition);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, (void*)0);
```

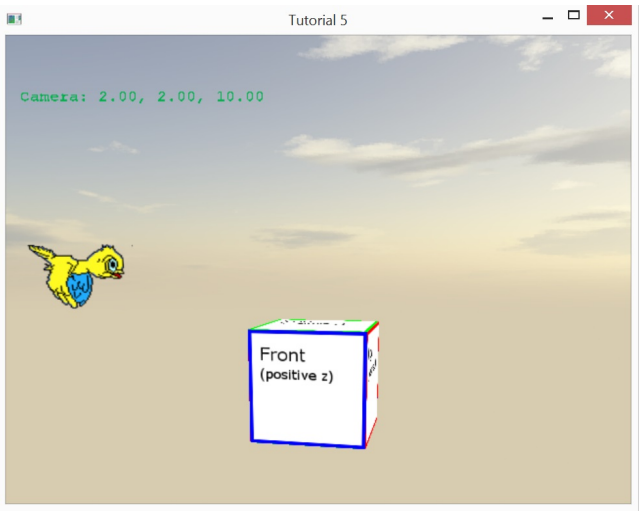
This loop will loop through the string and print out each character. The only problem is that all the character will print in the same position. Modify the translation matrix in the loop so that each character is positioned `letterWidth` to the right of the previous.

Go to main.cpp change what you wrote in Task 1 to use `text->render(orthographicMatrix,"Hello")`. Compile and run. You should have the same result as Task1.

Lets try printing out some debug information like for example the camera's position. Add the following code:

```
      char buffer[256];
      sprintf(buffer, "Camera: %.2f, %.2f, %.2f",
camera1->getPosition().x, camera1->getPosition().y,
camera1->getPosition().z);
      std::string message(buffer);
      text->render(orthographicMatrix, message);
```

Compile and run.



Bonus

Now that you can render a whole line of text, let's add the ability to render multiple lines of text. We will do with by adding line wrapping and checking for the newline character.

Use the `wrapLength` (-1 means no wrapping) property of `Text` to determine how many characters should appear before a wrap. Hints: the ASCII value of a newline('\n') is 10; keep a count of how many newlines there have been so you can determine the y position you are working at. Lastly, there is also a `lineSpacing` variable which should be used in combination with `letterHeight` to also help determine the change in y position. To test this try the following. Set the wrapLength to 40 and print the following message: "This sentence ends with a newline\nThis sentence should wrap at right here and then continue on this line"