TDT4230: Graphics and Visualisation

Assignment 2

Bart van Blokland
Michael H. Gimle
Peder Bergebakken Sundt
Department of Computer and Information Science
Norwegian University of Science and Technology (NTNU)

February 3, 2022

- **Delivery deadline: March 6th, 2022 by 23:59.**
- **This assignment counts towards 5% of your final grade, unless the exam is made digital due to COVID, in which case it will be worth 10%.**
- All work must be completed individually.
- Deliver your solution on *Blackboard* before the deadline.
- Upload your report as a single PDF file.
- Upload your code as a single ZIP file solely containing the *src* and *shaders* directories found in the project.
- All tasks must be completed using C++.
- Use only functions present in OpenGL revision 4.0 Core or higher. If possible, version 4.3 or higher is recommended.
- The delivered code is taken into account with the evaluation. Ensure your code is documented and as readable as possible.
- Feel free to refactor the handout code if you prefer to have it structured in a different way.

Questions which should be answered in the report have been marked with a **[report]** tag.

Objective: Basics of texturing, and some techniques making use of them.

Preamble

Now that we have shed some light on the Phong lighting model in the previous assignment, hopefully the world feels ever so slightly brighter. However, the final result may still be a little... bland at this point. It needs something to spice things up a little. Such as some texture.

In this lab, we're going to decorate our scene with some fancy looking textures, each with their own effects. First, we'll render some text to the screen, followed by an implementation of normal mapping.

Task 0: Preparation [0 points]

If you did the last assignment, make sure you run the following command:

```
git pull
```

There shouldn't be any merge conflicts, but if there are, feel free to fix them any way you'd like, including simply making the changes included in the commit manually (there are not many).

If you didn't do the previous assignment, you must go back and do it at least up to and including task 2a. A lit scene is required to be able to complete this assignment.

Task 1: Fontastic decorations [2.5 points]

One common task you often eventually run into when working with OpenGL is the need to draw some text to the screen. Unfortunately, the OpenGL standard has absolutely nothing to help you here. Instead, there are three main ways of going about this.

First, you can chop each glyph in your font into a pile of triangles, and rasterise those. Unfortunately, this requires rendering a lot of triangles in order to closely approximate the curves present in many glyphs.

A second, but cheaper way is to render the string to a texture first, then render that texture instead. It is also possible to create a separate texture for each character, then render them one by one at runtime, although this may not always be efficient.

Finally, we can use the good old texture atlas. This is a single texture which contains all characters of your font, and means you don't need to switch between textures when rendering your string. Using this method is easiest when the font is monospaced; where each character is equally wide. The inexpensive rendering of these fonts is one of the main reasons they were commonly used in early games, where a blitter could easily copy and paste characters around without any complicated logic.

In this task, we'll use a texture atlas to render some text to the screen.

However, there are several things we need to set up in the process.

a) **[0.0 points]** First, make sure you download the zip file containing images from blackboard, and extract them in the designated res/textures directory in the project.

b) **[0.1 points]** The charmap.png texture contains all characters of the ASCII character set (although all the weird special characters are somewhat misrepresented).

   Read the contents of this image file into memory. I've included a handy image loading function in the `utilities/imageLoader.hpp` file.

c) **[0.1 points]** Create a function which takes in an image loaded into memory (such as the one from the previous task), and returns a texture ID. Since we need to load several other textures further down the line, it's handy having one of those around.

d) **[0.3 points]** The next step is to actually implement this function.

   As is customary with OpenGL functions, the first step involves creating a new empty texture. Like all OpenGL objects, we only get a reference to the texture, rather than some sort of data structure.

   The function to create a new texture is:

```
void glGenTextures(unsigned int count, unsigned int* textureID);
```

If you recall the `glGenBuffers` and `glGenVertexArrays` functions; this one is used in exactly the same way. If you only want to create a single texture, pass 1 into the `count` parameter, and a reference to the variable in which you'd like the texture reference to be stored using the & operator.

As with VAO's and VBO's, in order to make changes to a texture we need to bind it first. We can use the following function to accomplish this:

```
void glBindTexture(enum target, unsigned int textureID);
```

The target parameter effectively allows you to specify which type of texture you'd like to use your texture for. You'll rarely need to pass anything other than the target for "regular" 2D textures here: `GL_TEXTURE_2D`. As for the textureID parameter, pass in the texture ID you generated previously.

We're now ready to configure our texture. Most notably, we can define its contents. This is where the image parameter of the function comes in handy.

The struct contains a width and height field, as well as a byte array of pixel values. The format of this pixel data is stored in the order (RGBARGBARGBA ..), with 8 bits per channel. This format should be identical to the one used by OpenGL.

Just like with vertex buffers, before a texture can be applied to a surface, it needs to be copied into the GPU's VRAM. You can use the following function to do so:

```
void glTexImage2D(
    enum target, int level, enum internalFormat,
    int width, int height, int border, enum inputFormat,
    enum type, void* data);
```

Lots of parameters here. We'll look at them one by one. The `target` one is (at least for our purposes) identical to the one passed into `glBindTexture`. Next, the `level` parameter can be used to specify different levels of detail of your texture. Since we're only specifying a single level here, just pass in the default, which is 0.

The one after specifies the number of components or channels you would like OpenGL to store in the texture's buffer in graphics memory. For storing 1, 2, 3 or 4 channels requires passing the parameters `GL_RED`, `GL_RG`, `GL_RGB`, and `GL_RGBA`, respectively.

The `width` and `height` parameters are the width and height of your image, in pixels. The `border` parameter exists for backward compatibility reasons, and must be set to 0.

Next, we need to specify how the buffer containing pixel data is formatted. In the final parameter of the `glTexImage2D()` function, we're basically handing over a pile of bytes, and as such we need to explain how to properly read its contents. This is done through the `inputFormat` and `type` parameters.

The `inputFormat` one basically specifies the number of channels internally. For keeping things understandable and consistent, pass in the same value as the `internalFormat` parameter here.

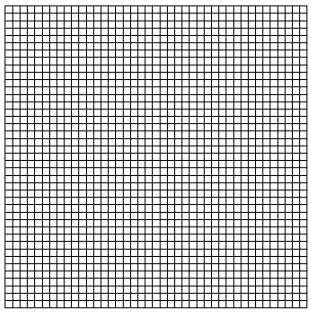
The `type` parameter specifies how many bits/bytes each channel requires, as well as whether you're passing in (un)signed integers of various widths or 16/32-bit floats. Regular PNG images by default include an alpha channel and use 8 bits to store the values of individual channels, which corresponds to the enum value `GL_UNSIGNED_BYTE`.

Finally, there remains a single parameter; the one where you give OpenGL a pointer to an array in memory where the contents of your texture are stored. If your data happens to be inside of a `std::vector`, you can get a pointer to its contents by writing `someVector.data()`.

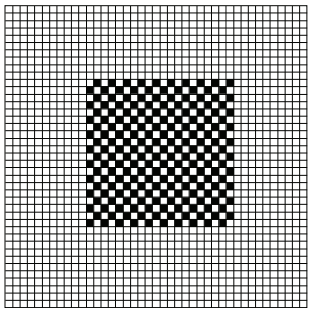
e) **[0.5 points]** We have now created a texture object, and copied our texture image into it. The next thing we need to do before returning the texture ID from the function is to configure how it's going to be sampled.

This may sound a little odd at first; we have hundreds if not thousands of pixels to choose from here. Why make such a big deal about this?

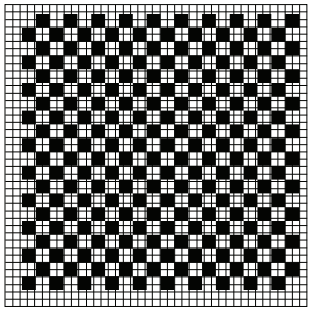
To answer this, let's say that the raster of pixels shown below are the pixels of your screen:



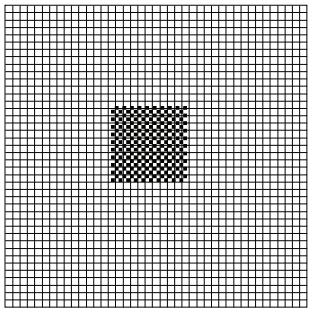
Now let's draw a simple checkerboard texture in there, exactly at the resolution of your screen:



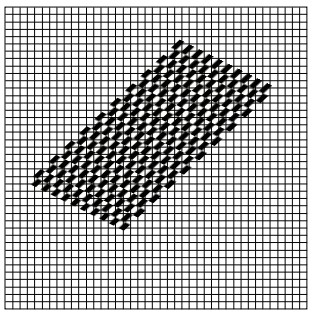
So far so good. Each pixel in the image maps to a pixel on screen, so we can copy the colour values directly. Unfortunately, this situation almost never happens. In reality, we also would like to be able to make our image a lot larger:



And in other cases, a lot smaller:



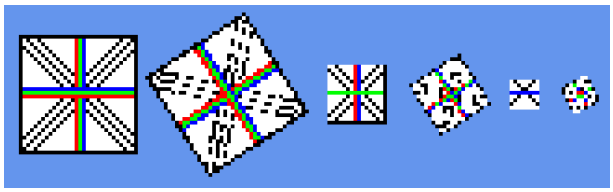
Even worse, as happens in 3D scenes, we'd like to render textured surfaces on which all kinds of transformations have been applied:



The question here is which colour each individual pixel should receive. In some cases there are multiple texture pixels (referred to as "texels") covering the same screen pixel, which means the answer isn't really clear-cut.

Fortunately, there are several ways of going about solving this problem, and OpenGL allows us to configure how it behaves.

The straightforward approach is to simply compute the pixel coordinates, and use the colour value of the pixel you landed on. This is known as *nearest neighbour* sampling. Unfortunately, this approach does not always produce good results. Here's an example of what happens when the image becomes very small [1]:



Notice how the image all the way to the right basically disappears? The problem here is something called "aliasing" (yes, the same kind anti-aliasing tries to reduce). It occurs whenever the sampling rate of a high-frequency signal is undersampled, and therefore not represented properly. In more practical terms, for the image on the left, there are a sufficient number of pixels being drawn in order to accurately display the entire image. However, the images further to the right steadily increase the number of pixels between each sample, as seen from the texture's perspective. This means that the "signal" of the texture has a greater frequency than the rate at which it is sampled.

This problem can be solved in one of two ways, both of which involve sampling more texels.

First, we can sample some neighbouring texels, and average the result. This is commonly done by sampling four pixels, and subsequently linearly interpolating their values to arrive at one colour. This helps a lot in many cases, but ends up with the same problems as nearest neighbour sampling when the texture is sufficiently far away or shrunk.

Alternatively, we can try to reduce the frequency of the texture when required. The idea is that if the difference between the sampling frequency of the screen and the texture is too high, we compute a different texture map that more accurately matches the sampling frequency. These alternate, smaller textures are referred to as "mipmaps".

They are generated by iteratively halving the resolution of the texture, and averaging the colour of each four pixels into one. Luckily, you don't need to implement this manually, because OpenGL can generate and manage these for you. However, you do need to explicitly generate them.

This is pretty easy, and you can use the following function to do so:

[1] Image credit: `https://blogs.msdn.microsoft.com/shawnhar/2011/04/29/texture-aliasing/`

```
glGenerateMipmap(enum target);
```

Make sure you call it in your texture generation function.

As with previous function calls, the `target` parameter is usually `GL_TEXTURE_2D`.

There's also a flipside of the aliasing problem; when the texture is too large compared to the distance between each sample, the texture may become less interesting to look at (depending on the texture of course). For these cases, we're interesting in finding good ways to "invent" additional pixel values based on the ones present in the texture. Amongst others, linear interpolation can be useful here too.

So now that we've seen some solutions to oversampling and undersampling, how do we instruct OpenGL which mechanism to use?

This is where the `glTexParameter()` function comes in:

```
void glTexParameteri(enum target, enum parameterName, int parameterValue);
```

This function allows a wide variety of sampling options to be set (such as the wrapping behaviour; what happens when you pass in texture coordinates that are larger than 1), but the sampling ones are the most relevant here. We'll talk about texture coordinates later. The `target` parameter is in our case pretty much always `GL_TEXTURE_2D`.

For the parameter name, there are two relevant ones. First, the `GL_TEXTURE_MIN_FILTER` is used for specifying how the texture should be filtered when it is rendered at a resolution where texels are smaller than pixels on screen. The *MIN* part stands for *Minification*. When using `GL_NEAREST` and `GL_LINEAR` as values for `parameterValue`, OpenGL will use nearest neighbour and bilinear interpolation, respectively.

When using mipmaps, the corresponding texture parameter enum values are `GL_NEAREST_MIPMAP_NEAREST`, `GL_NEAREST_MIPMAP_LINEAR`, `GL_LINEAR_MIPMAP_NEAREST`, and `GL_LINEAR_MIPMAP_LINEAR`. The part *before* each `MIPMAP` specifies whether to sample the nearest mipmap image only, or to sample from two and interpolate the results. The linear/nearest part *after* each `MIPMAP` specifies how to sample pixels on individual mipmap image, as discussed before.

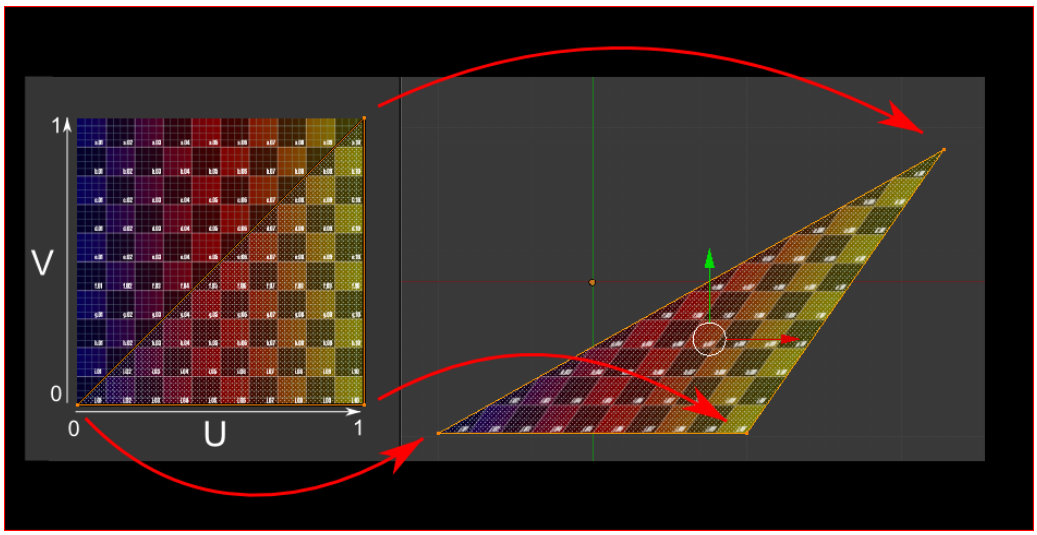
Similarly, for cases where the texels are larger than pixels on screen, you can specify the `GL_TEXTURE_MAG_FILTER` parameter, which determines the filtering behaviour in those situations. Here too you can use the `GL_NEAREST` and `GL_LINEAR` as parameter values.

After you've enabled mipmaps, and selected appropriate texture sampling parameters, we're ready to define *which part* of a texture should be mapped on to a given triangle.

f) **[0.3 points]** You see, the problem is that having one texture map per triangle is a little wasteful. As such, in a practical environment it's common to have one or a few texture maps per 3D model.

The mechanism employed by OpenGL for determining which triangle should use which part of the texture are texture coordinates. These are 2D points specified for each triangle in the object.

Texture coordinates need to be specified in a special coordinate system. In this systen, all UV coordinates are normalised. That is, rather than specifying the pixel (or "texel") coordinates directly, you specify a value between 0 and 1, such that $(texCoord.x * texImageWidth) = desiredPixelCoord.x$. Normalising coordinates, amongst others, allows you to use textures of different resolutions for the same model. A visual example is shown in the image below [2]:



Also, the x and y axis of the texture coordinate system are referred to as the u and v axis instead, respectively. The alternate common name for textures originate from the names of these axes; the UV map.

In order to render our string, we need to specify these texture coordinates.

If you look inside the utilities directory, you'll find a "glfont.cpp" file there. It contains most of the code needed to create the geometry buffers you'll need.

In order to make the correct letters appear in the right positions, we need to supply appropriate values for the texture coordinates. Unfortunately, while the vertex and index code is intact, the code responsible for setting up the appropriate texture coordinate values got damaged during shipping.

In case you weren't aware, there are a total of 128 symbols in ASCII.

Create a texture coordinate buffer in the `generateTextGeometryBuffer` function, and fill it with the texture coordinates needed to display text.

[2] Image credit: `http://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube/`

g) **[0.2 points]** We're getting close now. From the setup function in `gameLogic.cpp`, call the texture generation function with the loaded PNG image, and call the text generation function in glfont.h.

The function takes in three arguments;

- The text to be rendered.
- The ratio of an individual characters height over its width.
- The total width of the entire text-mesh.

The information you might be interested in knowing if you want the text to look "pixel perfect" is that the characters are 39 pixels tall and 29 pixels wide.

Finally, generate a VAO for the generated text string mesh.

h) **[0.3 points]** To simplify the rendering process, we'll create a special scene node type for geometry rendered in 2D (such as a user interface, or simply some text).

In `SceneGraph.hpp`, add an new type to the `NodeType` enum for 2D geometry, and create a field to hold the texture ID inside the SceneNode struct itself.

While you're here, also define an enum for normal mapped geometry, and a corresponding normal map texture ID in the struct. We'll use this in the other major task of this assignment.

Create a "2D geometry" type scene node, and add it to the root node of the scene graph.

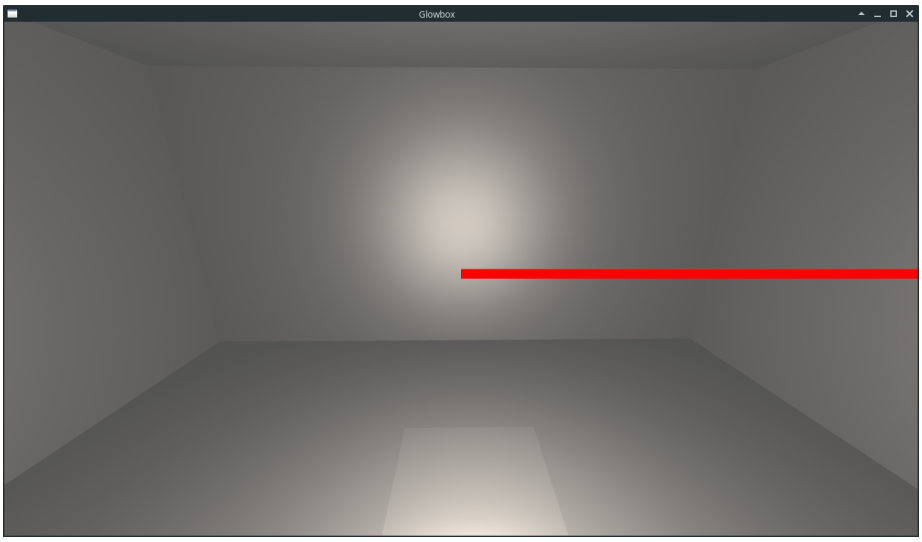
i) **[0.2 points]** Next, you can choose here between extending your existing shader and make it possible to enable and disable certain behaviour with uniform variables, or create a new shader pair. If you create a new shader pair, it's probably a good idea to draw all 2D SceneNodes after you're done drawing the 3D scene, so you won't have to re-set uniform values after every time you've drawn a 2D element.

We'll need it, because the Phong shader is not made to draw textured 2D unlit triangles.

Which route you take depends a bit on your personal preference, but **remember that uniform variables are associated with specific shaders**. If you switch between shaders, you need to make sure to update any necessary uniform variables it might need [3].

It is at this point you'll be able to see the geometry associated with the text. Try placing it at (0,0), and have the shader code responsible for drawing your text output the texture coordinates directly (or any other colour of your choosing, such as red). You should see something that looks similar to this;

[3] Fun fact: Uncharted 4 uses a single "monster shader" to render everything. They dynamically turn features on and off depending on what is being rendered. Engines such as unreal can handle the nitty gritty of such render state updates for you too.



j) **[0.4 points]** When drawing our 2D geometry, we don't want to be using a perspective projection matrix. There are a couple of different ways to approach drawing flat geometry on the screen, here's some of them;

- Don't use any projection matrix, making sure to place anything that should be visible within the 2x2x2 clipbox.
- Create an orthographic projection matrix with a width and a height of 1, and a corner at (0,0)
- Create an orthographic projection matrix with the same dimensions as the viewport, and a corner at (0,0).

I would personally recommend doing the last one, with the corner at the origin being the bottom left corner, because this allows you to move the text around the screen in a very precise (pixel perfect) way. You can find the dimensions of the viewport of the game in `window.hpp`.

Next, we'll need to set the texture OpenGL will use when rendering the node's contents.

What we have to do in order to accomplish this is to map our shader to one of the 16 "texture mapping units". On the GPU, the process of texture mapping has primarily been implemented in hardware. Shaders can as such only use a specific number of textures at a time. This means a lot of the interpolation performed on pixel values can be done very quickly, but trades away some degree of flexibility by limiting the number of textures you can use at any given time.

The way OpenGL represents this is by assigning (or "bind") a specific texture to a specific texture unit. You can use the following function to accomplish this:

```
void glBindTextureUnit(unsigned int textureUnitIndex,
                       unsigned int textureID);
```

Here the `textureUnitIndex` is the index of the texture unit, which should be anywhere between 0 and 15 (because there are 16 texture units), and `textureID` is the reference to your texture which you previously generated with the `glGenTextures()` function.

Now that we have assigned our texture to a specific slot, we can use it in the fragment shader. Within a shader, OpenGL refers to a texture as a "sampler2D". It explicitly makes a distinction here between the texture itself, which contains the image, and the configuration of the procedure sampling values from it (more on this later).

We use a special layout qualifier to explain to OpenGL which texture unit we have bound our texture to. At the same time, we create a variable containing a reference to the sampler, which in turn samples values from our texture

```
layout(binding = 0) uniform sampler2D someSamplerVariable;
```

The `layout(binding = 0)` part specifies that the texture has been bound to texture unit 0. Set this value to the texture unit you bound your texture to if it was not unit 0.

To sample a colour value from the texture, we need two things; a sampler and a texture coordinate. The function which allows you to perform the sampling operation itself is:

```
vec4 texture(sampler2d sampler, vec2 textureCoordinate);
```

Note that the return value of the `texture()` function depends on the internal texture format specified in the `internalFormat` parameter of the `glTexImage2D()` function.

Also note that the `glBindTextureUnit()` function is only supported in OpenGL 4.5 and up. If your machine does not support this, you can use the `glActiveTexture()` function followed by a call to `glBindTexture()` instead.

Finally, render the triangles in the node's geometry buffer. They should now appear on screen.

k) **[0.1 points]** **[report]** Put a screenshot of the rendered text in your report.

If you'd like to give the drawn text some purpose, you can use it to instruct the player to click to start the game, or show a score tally.

Task 2: Colourful Questions [0.5 points]

a) **[0.2 points]** **[report]** As we've seen in the previous question, texture coordinates are used to denote which points on a texture should be mapped on to which vertices

of a triangle. Also, points which are sent out from the vertex shader to the fragment shader are interpolated, as a single triangle may contain a number of fragments.

One key detail is that this interpolation is not linear (unless explicitly disabled), as otherwise the rendered image would not look physically plausible.

What is the primary cause for interpolation to be non-linear, and why would a rendered image with linear interpolation of texture coordinates look incorrect?

b) **[0.2 points]** **[report]** The material provided for this assignment also came with a displacement map, but I chose not to include it in the provided textures.

   (a) Why would a displacement map be an inappropriate thing to use if we wanted more detailed features in the walls of the scene in this assignment?

   (b) In what way would we have to change the scene/meshes in order to reap a benefit from using a displacement map?

c) **[0.1 points]** **[report]** Consider a texture which is cut in half vertically, whose left side is completely red, and the right side green. The red side of the texture is now mapped on to two triangles forming a rectangle. Close to the camera, this red rectangle renders exactly as you'd expect it to. However, when moved far away instead, a downside of mipmaps becomes visible. What visual artefact would you expect to see, and why does that happen?

Task 3: Abnormal normals [2 points]

Textures are not only useful for specifying the colour of an object. You really should think of them as two-dimensional buffers of arbitrary data. For instance, you can create a texture where each pixel represents a particular rotation, and each row of pixels the rotation of a particular leaf on a tree. This allows you to store animation data that can be read out by the vertex shader.

Another common use case for textures is a technique called "normal mapping". It allows pretending like a scene has more triangles in it than it really has. In our scene in particular it can make a big difference, as the walls, floor, and ceiling all are made out of two triangles each.

In short, normal mapping functions by replacing the normal of a surface with the one from a texture, which allows for a much more fine-grained control over normals than the one normal per vertex (thus three per triangle).

In this task, we'll implement normal mapping to add a bunch of detail to our walls.

If you're still using the coloured light sources from the previous assignment, now is probably

a good time to only be using a single, white light source. This is going to make it easier to see if anything has gone wrong.

a) **[0.2 points]**

Set the node type of the box scene node to the textures/normal mapped type you created earlier.

b) **[0.1 points]** In the texture handout, there's a diffuse texture and a normal map. Load both, and create an OpenGL texture object from them. Store these in the scene node fields you created before.

c) **[0.2 points]** Either create another shader pair for rendering normal mapped geometry (duplicate the phong shader from last time as a starting point), or add any uniform variable flags that allow you to enable and disable the normal mapping feature. The latter alternative is definitely easier for this task.

In the renderNode() function inside gamelogic.cpp, add another case for rendering the normal mapped node. Set any state your shader needs. If you're using a separate shader, make sure to bind the shader and set the computed light position for every shader program.

Also bind each texture to the appropriate texture unit, and then render the node.

d) **[0.2 points]** Ensure that normal mapped meshes are rendered with the scene node's diffuse texture.

e) **[0.3 points]** Modify the fragment shader such that the normal is read from the normal map texture, rather than the one from the shader input. Note, however, that the normals in the texture are stored in a format where each axis ranges between 0 and 1. By doubling their values and subtracting one, you can get them into the regular normal vector ranges of -1 and 1.

This also causes normal map textures to generally have a blue tint to them, as normals never really go below zero on the x-axis. The blue channel is therefore almost always greater than 0.5.

f) **[0.1 points]** **[report]** Try running the program now :)

The result should look super detailed.

Put a screenshot in your report showing what it looks like.

However...

g) **[0.4 points]** At this point, the back wall should look great, but everything else might seem a little off. Specifically, it's probably going to be looking quite a bit darker than you'd expect when the light gets near.

Notice how it seems like the light source is a spotlight?

This is caused by the fact that the normal map normals are specified in so-called "tangent space", and are not reoriented relative to the scene.

So what do we do to solve this? We need to compute the so-called *TBN matrix*. It's essentially the normal matrix we saw last time, but used to correct normal map normals instead.

The first step in this process is computing two additional values per vertex; the tangent and bitangent. The following tutorial gives a pretty good explanation on how to do that:

`http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/`

Extend the `generateBuffer` function in `glUtils.cpp` to compute the tangent and bitangent vectors, and then add them as VAO attributes. Alternatively, you can create a function which binds a VAO and "appends" the additional buffers to it. It is not necessary to do the *Indexing* part of the linked tutorial, as the model we're texturing doesn't use any shared vertices.

Mind your coordinate spaces! The tutorial linked above uses *Camera Space* as its chosen coordinate space within which to do all lighting computations. If you followed the previous assignment, you're likely to be using *Model Space* instead. Make sure you understand what kind of implications this has for the code you see on the tutorial! You can end up with a correct solution in three different ways; By doing everything in *Tangent-*, *Camera-*, or *Model Space*. This task is definitely not a freebie despite linking to a tutorial with source!

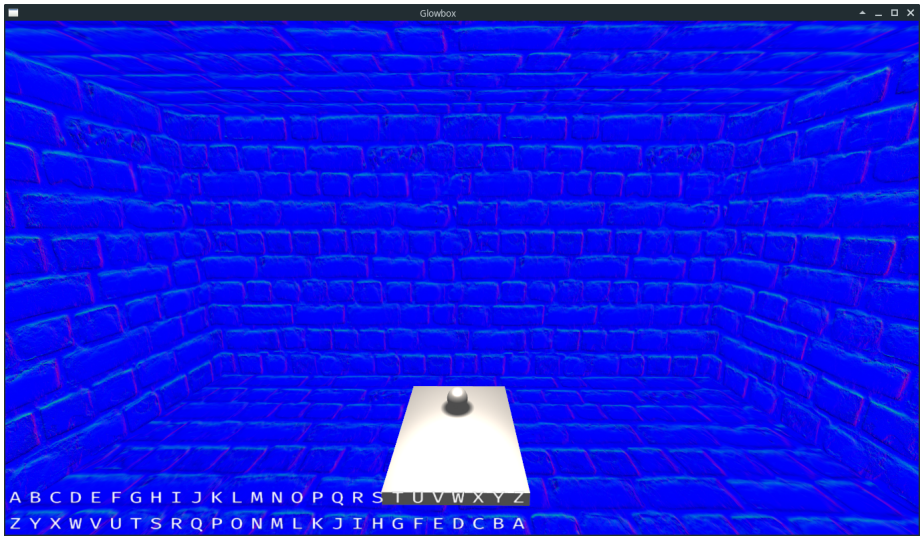
h) **[0.3 points]** In the vertex shader, combine the new tangent and bitangent shader inputs with the existing normal one to construct the TBN matrix, and pass it on to the fragment shader.

i) **[0.1 points]** In the fragment shader, transform the normal read out from the normal map texture with the TBN to obtain the corrected normal.

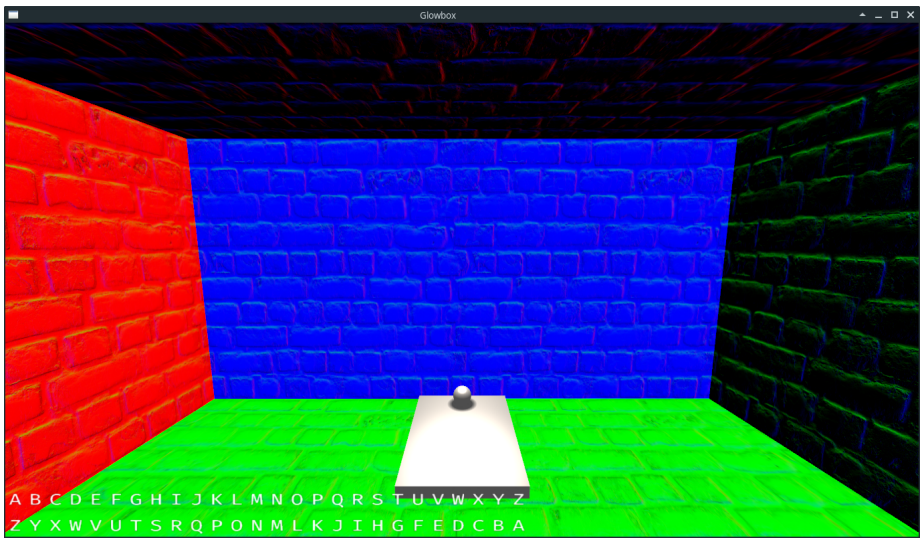
I *strongly* recommend making sure your TBN matrix is properly constructed by setting the output colour to a TBN-corrected sample from the normal matrix.

Here's what you should see before and after applying the TBN matrix, if you went for model space as your final space. Notice how the blue colour on the walls apart from the back wall now is predominantly on the sides of the bricks that face towards the camera.

```
texture(normalMap, uv).xyz * 2 - 1
```




```
TBN * (texture(normalMap, uv).xyz * 2 - 1)
```



j) **[0.1 points]** [**report**] And that's it! Your scene should now look like a realistic breakout clone!

Add a screenshot of your final scene to the report.

Task 4 (OPTIONAL): BONUS TASKS [at most 0.5 bonus points]

Rough around the edges. [0.3 points] [report]

The material texture you were given for this assignment also includes a roughness map. Roughness is the opposite of "shininess", and influences how focused a specular reflection should be in various parts of the scene.

In order to utilise this roughness map in order to bring a tiny bit of extra fidelity to our scene, we first have to make a change to how we compute the specular component of the lighting model. There are quite a few ways to do this, but for simplicity's sake, we're going to replace the "shininess factor" used in the last assignment with the following formula:

$$\frac{5}{\text{roughness}^2}$$

Now we have to get the values associated with the roughness map into the shader, the same way we sent in the normal and diffuse maps earlier.

Finally, we can sample from the roughness map and use this to compute the specular component of the intensity.

The result from doing this won't be nearly as flashy as what we get from adding the normal mapping, and it would look much better if we were using an even more complicated lighting model and environment mapping, but this should result in a minor aesthetic improvement to the scene.

Attach a picture of your roughness mapped scene for some cool, cool bonus points!

Impress me! [0.1-0.5 points]

If you have a cool idea or you've done something interesting, this is the bit that I put in so I can give you points if I'm impressed by what you did!