TDT4230: Graphics and Visualisation

Assignment 1

February 17, 2020

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

- **Delivery deadline: February 23rd, 2020 by 23:59.**
- **This assignment counts towards 5% of your final grade.**
- 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.

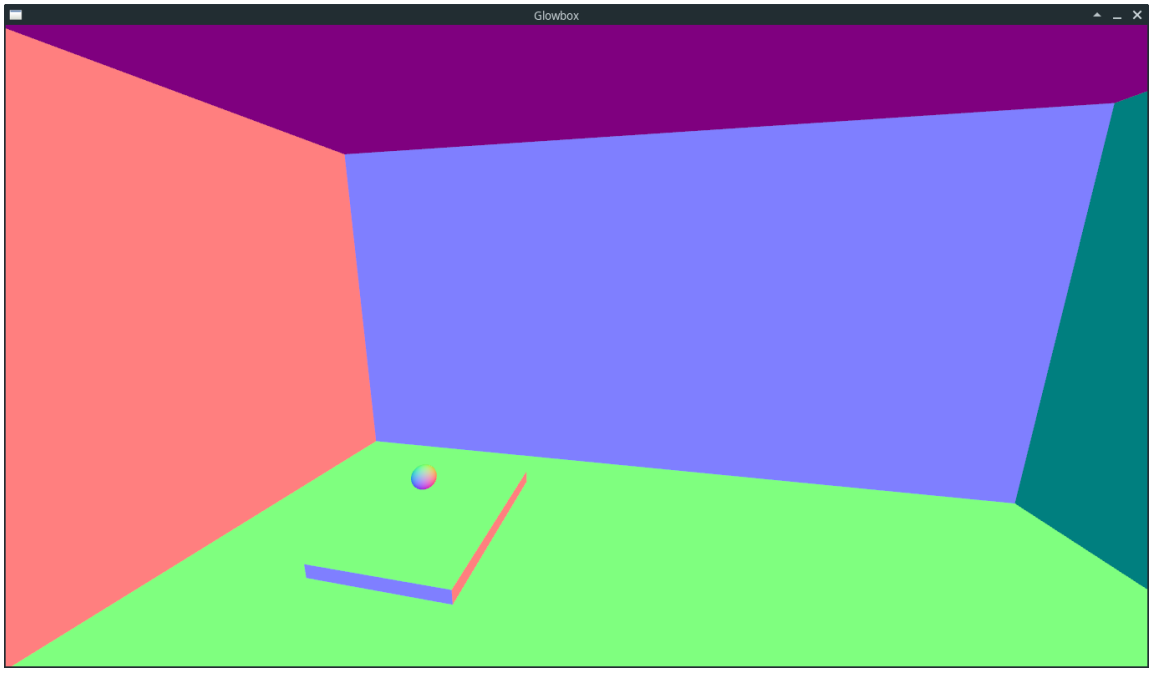
Questions which should be answered in the report have been marked with a [**report**] tag.

Objective: Understanding the basics of Phong lighting, the necessary transformations to implement it, and to experiment with shadows and coloured light.

Assignments Overview

We've created a small game for you. Nothing that's winning any prizes (apart from worst game of the year) any time soon, but it's definitely playable.

There's only a minor problem with it; it looks like a weirdly tame acid trip.



In the two assignments in this course, we're going to improve the looks of this thing in two different ways. First, we'll add some lighting by implementing the Phong lighting model. In the second assignment we'll spice things up even more with textures.

These assignments assume that you have done the OpenGL introductory assignments in TDT4195. If you have not done these, I recommend reading through the OpenGL Hitchhiker's Guide published on Blackboard, as well as looking into how Scene Graphs work.

Task 0: Preparation [0 points]

a) Ensure the following tools are installed:

- CMake
- Git
- A C/C++ compiler, such as GCC or MSVC++.

These have already been installed for you on the lab machines.

On Microsoft Windows, it may be easiest to install Microsoft Visual Studio Express, which can be downloaded for free and comes with the MSVC++ compiler, and it is the easiest to work with in combination with CMake. Alternatively, the full version of Microsoft Visual Studio can be downloaded for free through MSDNAA (Gurutjenesten has a page with a link).

b) Clone the Gloom repository using the command:

```
git clone --recursive https://github.com/bartvbl/TDT4230-Assignment-1.git
```

Note: the additional `--recursive` flag is required here for downloading the third party libraries the project depends upon.

If you forgot to include the `--recursive` flag when cloning the repository, use the following command:

```
git submodule update --init
```

c) Compile the project.

Compiling should be as simple as running:

```
cmake ..
```

inside the build directory. You can afterwards open the generated solution on Windows, or run `make` on the generated makefile on UNIX systems.

On Linux, you may need to install some additional packages. There's a shell script in the "lib" directory.

If you, after compiling, get linker errors related to Glad, try running the glad script in the lib directory for your system, and re-run cmake. Please note you'll need python 3 installed for this.

d) Run the project. I recommend running the project with the –enable-music flag when you try it for the first time. Since the music may become a bit annoying afterwards, it's disabled by default.

Task 1: Let there be light [2.0 points]

To start things off, we'll implement the basic Phong lighting model, which is used nearly ubiquitously in realtime rendering. The computations themselves are not very hard to do, and you'll find plenty of sample shaders lying around on the web.

However, there are some gotchas and complications that you need to watch out for. In addition, I've noticed a large number of tutorials out on the web use older versions of OpenGL (most can be recognised by inputs using the "varying" keyword). Therefore, to ensure you get this thing right the first time around, I've structured this task as a kind of step by step tutorial.

The overall objective of the first task is to create a scene with multiple point lights, at least one of which is moving around.

I've written the code in such a way that most of the "infrastructure" should already be there. Feel free to modify it in any way you see fit, though.

a) **[0.2 points]** The first thing we need to do is create some additional scene nodes which represent our point lights.

   Representing lights as nodes makes it easy to determine where lights are located in the scene, and simplifies placing them relative to objects. For example, you'd like the light sources to move with the headlights of a car. Placing nodes inside the scene graph simplifies this, as you can utilise the matrix stack to compute the necessary relative transformations.

   The SceneNode struct defined in SceneNode.hpp has a field for node type, which allows you to determine whether a node is a point light at runtime. You'll probably want to track the ID of light sources somewhere, and changing SceneNode is a great place to add something to track that.

   Define at least three light sources, create scene nodes for them, and place them in the scene graph.

   At least one of the light sources should be moving. You can accomplish this either by animating it in the update function, or by attaching it to the ball or the pad, the latter of which is my personal recommendation, as it lets you move the light around with your mouse.

b) **[0.2 points]** Change the scene update function such that it computes the MVP and model matrices separately.

   This is needed because the perspective projection distorts angles. The Phong lighting model requires us to compute angles, so simply using the MVP is not an option. The produced image would otherwise not look accurate.

Pass any additional matrix/matrices you need into the vertex shader, creating separate uniform variables for it/them.

c) **[0.2 points]** Compute the positions of the lights, and create uniform variables for inserting the light positions into the shader.

You can compute the light positions in the node update by multiplying the origin of the coordinate space by the current transformation matrix. The origin of a 3d homogeneous coordinate space is $[0, 0, 0, 1]$.

Make sure these uniform variables are updated/set when performing the update pass of the scene.

d) **[0.2 points]** When we're transforming coordinates in the vertex shader, rotation transformations change the orientation of a surface. Logically, the normals would then also need to be recomputed to ensure they still represent a vector orthogonal to the surface.

There's only a minor problem: we can't use the regular model matrix for this. The problem is that this matrix may contain scaling and translation. When applied on a normal, this most likely means the normal will not be normalised again, and may even point in a completely different direction. So that's definitely a no-go.

We can, however, apply a small trick. We can compute the transpose of the inverse of the model matrix. Taking the top 3x3 part of this matrix yields the transformation without the unnecessary translations. Transforming the normal by this matrix and normalising it will give the correctly oriented normal.

Compute the normal matrix itself on the CPU side (in C++), then pass it into the vertex shader through a uniform.

e) **[0.2 points]** Now that all our shader inputs are ready to go, let's move over to the shaders themselves. We'll start by updating the vertex shader.

Fortunately, this one tends to be relatively straightforward, as most of the work of Phong is done in the fragment shader. We'll need to do two things in this shader.

First, transform the vertex and normal with the appropriate transformation matrices. Next, forward any (transformed) coordinates and shader inputs to the fragment shader.

Don't forget to normalise the normal *both* after transforming it with the normal matrix in the vertex shader *as well as* at the start of the fragment shader.

f) **[0.1 points]** Now it's time to move on to the meat of this task; the fragment shader.

As you might know, Phong computes light values based on four separate components; ambient, diffuse, specular, and emission.

Let's start with a constant colour for the ambient part. This component is independent of the light sources, and is added as a constant separately. You should hardcode it for now, unless you want to change it with a uniform.

g) **[0.3 points]** Next up are the diffuse and specular components, which need to be computed separately, and their colour is accumulated over multiple light sources.

We'll therefore need to loop over each active light source in the scene. You can either define this number with a uniform variable, or just hardcode your number of lights into the shader.

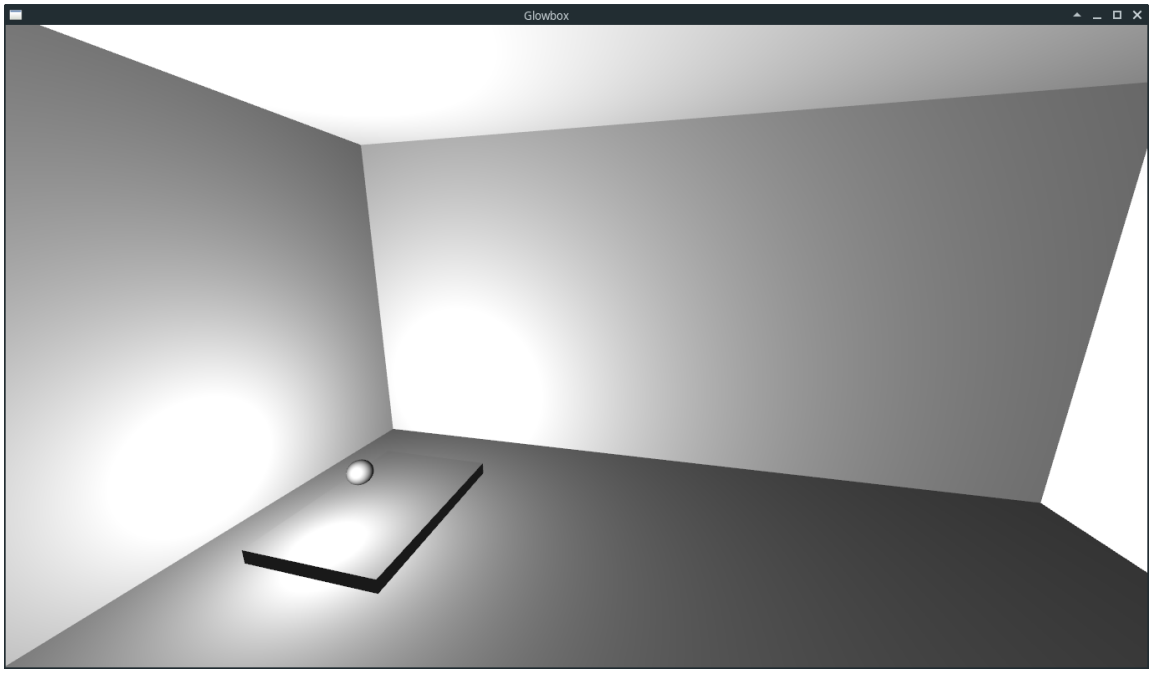
First up is the diffuse component. This involves calculating the light direction vector. This is a normalised vector that points in the direction of the light source from the (interpolated) vertex position. Make sure the light source and vertices are in the same coordinate space.

The diffuse intensity is simply the dot product (cosine) of the light direction vector and surface normal. To avoid affecting the other light components when the surface is facing away from the light source (which causes the cosine to be negative), it should be set to 0 when negative.

To obtain the diffuse colour, multiply the diffuse intensity with the surface's diffuse colour (which should be hardcoded for now, but will be a uniform later).

When the light is directly above a surface, this causes a lot of diffuse reflection, while surfaces at an angle relative to the light source reflect much less.

The scene below only has a single light source, but demonstrates what it should look like when only showing the diffuse and ambient components.



h) **[0.3 points]** We'll now compute the contributions of specular highlights for each of the light sources. As with the diffuse components, we calculate the specular colour for each light source separately, then simply add them together.

Specular highlights occur when you see the reflection of a light in a polished or shiny surface. This requires us to compute two more vectors.

First, reflect the previously computed light direction vector in the normal. For simplicity, you can use GLSL's reflect() function here. Note however that this function requires a vector to point "into" the surface. Simply negating the light direction vector should do the trick.

Second, compute a direction vector that points from the (interpolated) surface location to the eye (camera). In order to do this, you have to pass the camera's position as a uniform to the fragment shader.

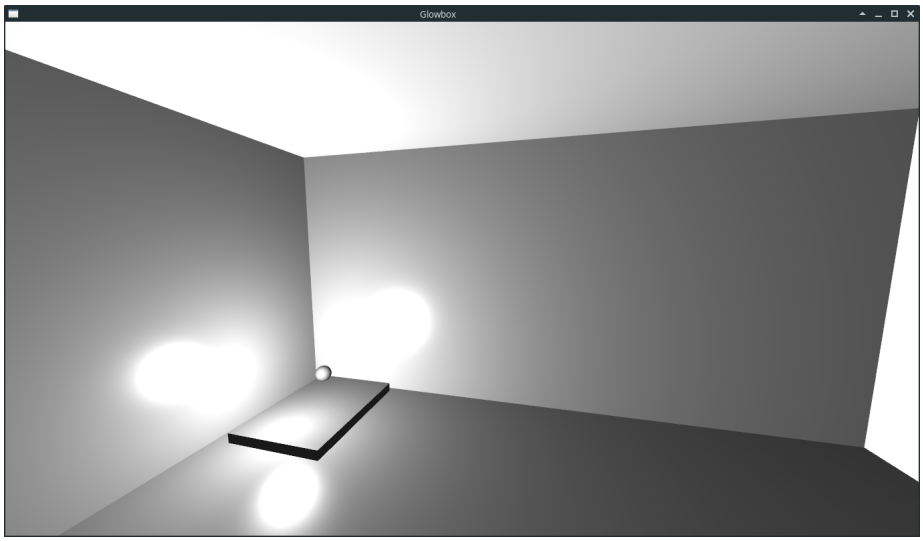
The specular intensity is determined by computing the dot product between the reflected surface-light vector and the surface-eye vector, by the power of a constant. This is called the "shininess factor". The shininess factor can be specified using a uniform variable, or a hardcoded value. Higher powers will cause a mode "focused" reflection, thereby representing a shinier surface.

If the specular factor is negative, you should set it to 0.

Multiply the specular intensity with a constant colour.

i) **[0.1 points]** Combine the accumulated specular and diffuse colours with the ambient and emission colours by adding them together. This results in the final colour of the pixel.

Once again, here's an image to give you an idea of what it should look like, using a shininess factor of 32. Feel free to use anything you like for the power though!



j) **[0.2 points]** **[report]** Put a screenshot of the resulting scene in your report.

Task 2: Sugar and Spice and everything Light [1.5 points]

a) **[0.5 points]** Imagine a street light in a dark street. The further away you walk from the light, the darker the street becomes. This effect is caused by that the photons from the street light are increasingly spread over a larger area.

We can model this "attenuation" in our fragment shader.

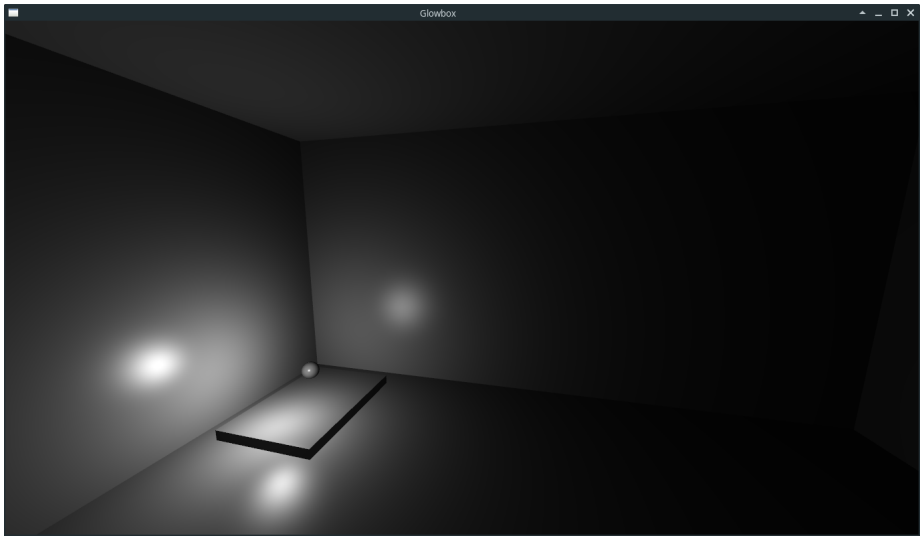
Since the light intensity decreases with distance, we'll need to compute the distance from our fragment to the light source. Next, the amount of light "left over" from the light source can be computed using the following equation:

$$L = \frac{1}{l_a + d \cdot l_b + d^2 \cdot l_c}$$

l_a, l_b, and l_c are constants that affect how quickly the light reduces in intensity. Play around with their values to see what works best for you. The effect of these factors are correlated with the scale with which the scene is drawn, which is a mostly arbitrary thing. In our case, you'll probably be most satisfied with the results of setting these values very low, on the scale of 10^{-3} to 10^{-2}.

Multiply the factor L by the diffuse and specular intensity of the corresponding light source, *before* adding it to the total light intensity.

Here's an example of what it could look like with a single light source.



b) **[0.5 points]** So, let's talk about banding.

You might have noticed it after completing the basic phong setup, but upon implementing attenuation it should become painfully obvious, due to the low-light areas. You should see some clearly delineated bands of intensities as you get farther away from the light source. These are the result of the simple fact that we don't have more than 256 different shades of gray on a computer monitor! Any gradient can suffer from the same problems, as it's the translation of a smooth mathematical concept into discrete pixel values which is the cause.

Luckily for us, there's a relatively simple solution to this issue; Dithering! [1]

There are *tons* of ways to do dithering, but the general idea is always the same; Make patterns that make the brain see colours that aren't really there.

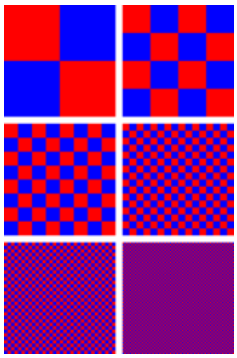


Image courtesy of Wikipedia.

[1] Dithering is actually a built-in part of OpenGL and is enabled by default, but its implementation is deferred to the implementing graphics driver, which means that it's not supported equally well on all platforms. The provided source code explicitly turns it off in order to bring attention to it.

Patterns for dithering usually have to be somewhat complex in order to prevent the brain from seeing them (we don't want to see the pattern in dithering, only the end result), but luckily there's an easy way to achieve a good-enough alternative; random noise.

There are tons of ways to generate noise out there, but our use case is very simple, so we're just going to go for a very simple noise function based on taking the fractional part of a really wild sine wave. This noise function has been included in the provided fragment shader.

We will then use *texture coordinates*, which we'll look at in more detail in the next assignment, to get a number that changes across the the surface of the object. We'll use this number as the seed for our random number generator, and use the random number generated to produce a random, very small change in intensity.

The effect of making this very subtle change in random places across the scene is that the very clear lines that are the result of banding will have their edges bleed into each other, and it ends up being almost impossible to see where the bands used to be.

At the very end of the fragment shader, when you set the output colour for the fragment, add the result of calling the `dither` function to the final fragment colour, using the provided texture coordinate as input.

The difference between dithered and non-dithered should be very clear when running the code. It might be hard to see on a picture, especially one that's displayed at a smaller resolution than what it was captured at, but it should be easy to see that it's working.

c) **[0.5 points]** **[report]** Our scene is starting to look pretty good, but there is something important missing; Shadows.

Making good looking shadows can be really complicated, so once again, we'll do what we've started getting pretty good at by now; cheating!

If we ignore the shadows that would be cast by the paddle, there's only a single object in the scene that can cast a shadow, and it's a sphere!

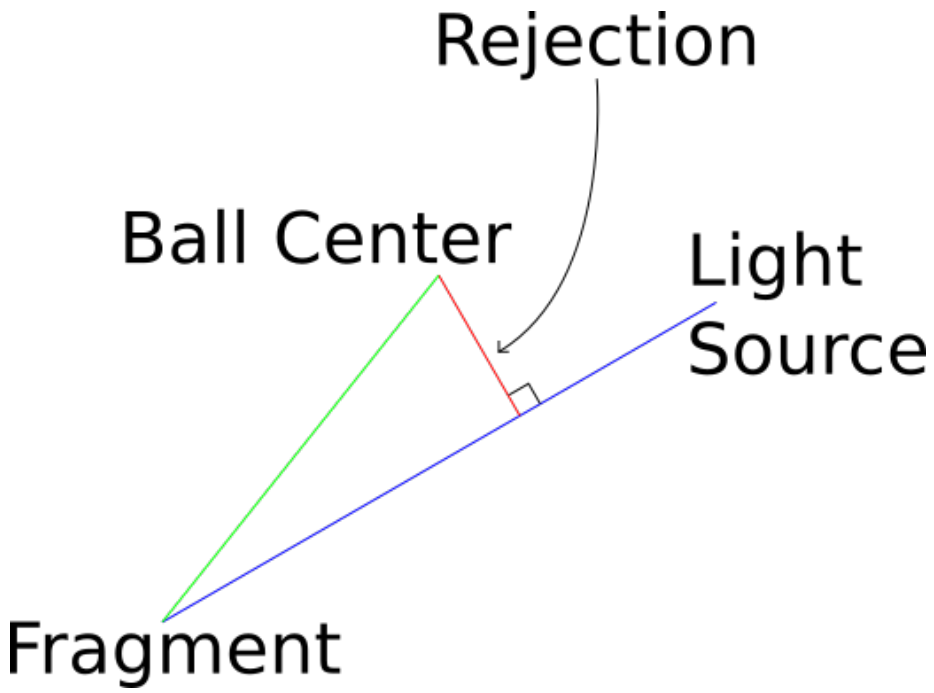
If we do some very simple "ray tracing" we can check, for each fragment, if any of the light sources are obscured by the sphere, and only add that light source's contribution to the final fragment colour if that's not the case.

The first thing we have to do is to make sure we can see the ball's position and radius from the shader. The position has to be sent in as a uniform, but the radius can be hardcoded.

Then, we compute two vectors; One from the fragment's position to the light source and one from the fragment to the center of the ball.

If we project the vector to the ball onto the vector to the light source, we get a vector pointing to the point along the ray to the light source which is closest to the the

center of the ball. The vector from this point and back to the center of the ball is called the "rejection" of the ball vector *on* the light-vector. If we compare the length of this vector to the known radius of the ball, we can see if the line passing from the fragment to the light source would have passed through the ball!



The formula used to compute the rejection is shown below, along with a glsl snippet, because I'm such a nice guy.

$$\boldsymbol{x} - \boldsymbol{y} * \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{\boldsymbol{y} \cdot \boldsymbol{y}}$$

```
vec3 reject(vec3 from, vec3 onto) {
    return from - onto*dot(from, onto)/dot(onto, onto);
}
```

If we only add the light contribution when the length of this vector is smaller than the radius will yield some simple shadows. There are however, some situations where this model fails, so we need some additional parts to make it right.

Specifically, there are two situations where the model misbehaves;

- We don't want the ball to cast a shadow if the light source is closer to the fragment than the ball.

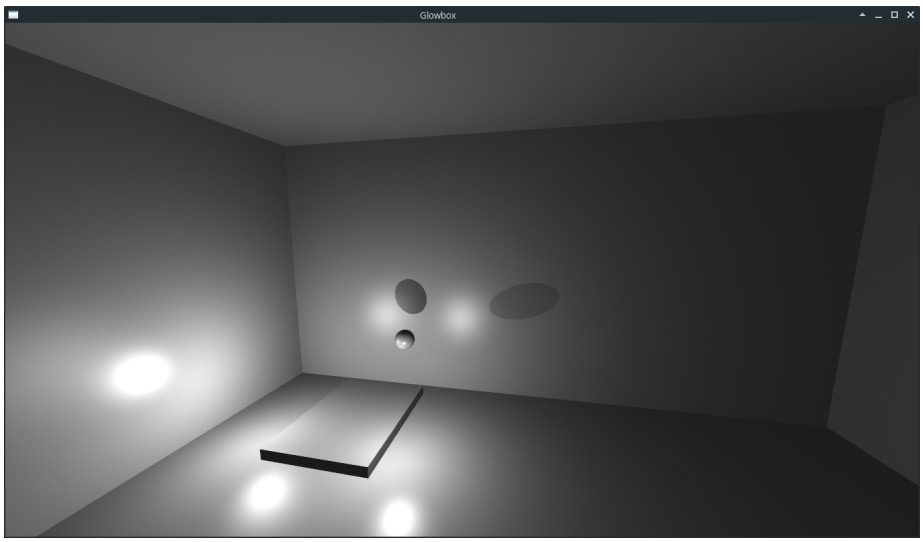
- The vectors from the fragment to the ball and the fragment to the light source shouldn't be pointing in opposite directions.

These issues can be easily solved by *not* casting a shadow if;

- The length of the vector from fragment to light is shorter than the vector from fragment to ball.
- The dot product of thoses two vectors is less than zero.

Implement shadows as described above (or in any other way, if you want a challenge!), and show a picture in your report.

Here's a picture to give you a general idea of what the end result should look like;



Task 3: Restoring colour vision [1.5 points]

Now that we have working lights that look somewhat nice, we're going to insert a splash of colour into the equation!

a) **[1.0 points]** The first step is going to be changing how we send parameters to the shader, in order to make it easier to send multiple similar values. Specifically, we're going to be using structs to send in a position and a colour for each light source.

First, define a struct representing a light source in the fragment shader. It should have at least two members; a vec3 for the position of the light source and a vec3 for its colour.

You can define a struct in glsl the same way as you would in c++;

```
struct MyStruct {
    float aFloatyNumber;
    vec2 aPositionIn2d;
};
```

You can then use the struct as the type of a uniform or a uniform array, like this;

```
uniform MyStruct anArrayOfStructs[5];
```

Note that we don't specify the `layout()` like we were doing with the earlier uniforms. There's a good reason for this! Each member of a uniform struct is going to take (at least) 1 uniform location, and we would really like to be able to access the exact position without having to do arithmetic to figure out where to store a value. If we, for example, want to add a new member to a struct, we would have to re-do all of our work to figure out the new positions of things.

When setting the values for a struct in a shader, you have to set each of the members individually. In order to accomplish this, you have to ask the shader for the locations of the various uniform struct members. This can be done using the function `glGetUniformLocation(GLint shaderProgramID, char* nameOfUniform);`. We have provided a convenience function which allows you to get a uniforms location in a shader using a c++ `String` instead.

It looks like this;

```
GLint location =
    shader->getUniformFromName("anArrayOfStructs[0].aFloatyNumber");
```

Once you have the location, you can set the uniform the normal way;

```
glUniform1f(location, 2.71f);
```

This would result in `anArrayOfStructs[0].aFloatyNumber` returning 2.71 in the shader.

Formatting strings nicely is a bit of a hassle in c++, so we've included the `fmt` library, see the end of the `initGame` function to see an example of its use.

In `updateNodeTransformations`, set the positions and colours of the light sources, using the light id associated with the `SceneNode` to set the appropriate shader struct's values.

b) **[0.5 points]** **[report]** Finally we get to the good bit! All of the ground work has been done, and making colours appear should be pretty trivial from here. Replace the hardcoded light colour used in the fragment shader with the colours you've sent in with the structs from the previous task. Choose a different colour for each one of

your light sources. I recommend sending in "pure" colours, red, green and blue, to properly observe the effects of colour blending.

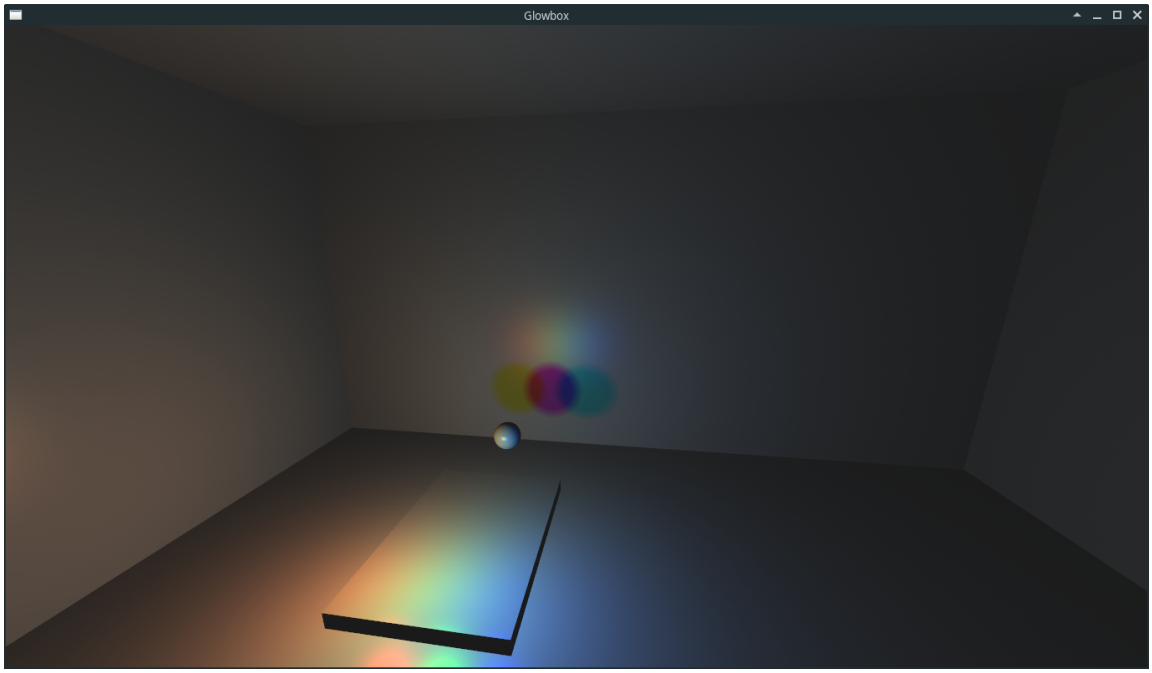
The colour of the light sources should affect both the diffuse and the specular component, while the colour of the surface itself (which is just white in this assignment) will not impact the colour of the specular reflection at all. This part in particular is something a lot of tutorials online will get wrong, but it's a necessity if one wants a lighting model which accurately reflects how things work in the real world. A blue object in a room only illuminated by a red lamp is in fact going to appear black, due to the fact that an object being "blue" means that it absorbs all parts of visible light *other than* blue.

A great way to see if you've done everything correctly is to put all of the lights on top of each other and make them red, green and blue, as this *should* result in a scene with a single white light source if you've done everything correctly!

The shadows cast by the coloured light sources should be exhibiting some interesting behaviour as well, which is consistent with how things look when using coloured bulbs in the real world.

Attach a picture of your scene with coloured shadows, including at least one example of shadows overlapping.

Here's an example of what it can look like when everything is put together, including soft shadows, which is the bonus task!



Task 4 (OPTIONAL): Getting softer with age [at most 0.5 bonus points]

This task is optional, but doing it can give you some extra points, although your assignment grade cannot be more than 5%.

In the real world, shadows rarely have a very hard edge. This is partly because real world light sources are never *actually* point lights, and partly because of other light scattering phenomena.

For the bonus task, we're once again going to do some cheating, and make our shadows soft, like in the picture above.

There are a few ways to do this, especially if you want your shadows to be more or less soft based on how far away the object casting the shadow is from the surface the shadow is being cast upon, but here's a suggestion for a pretty simple way to accomplish this effect;

When you have the ability to compare against a radius to see if a point is in shadow, like we do after the shadowing part of this assignment, you can simply do the math again with a slightly larger radius to determine if a point should be "softly" shaded. The part of the shadow which is definitely in the shadow area can stay dark, but as the distance gets into the "soft shadow radius", we can start adding more and more of the computed light contribution, until the distance is greater than the soft shadow radius, in which case there's no shadow applied at all.

Doing a simple linear interpolation between full shadow and no shadow over the "ring" marked out by the distances less than the soft shadow radius but greater than the hard shadow radius is enough to produce these much more pleasing shadows.

Making soft shadows essentially amounts to adding the following to your current model;