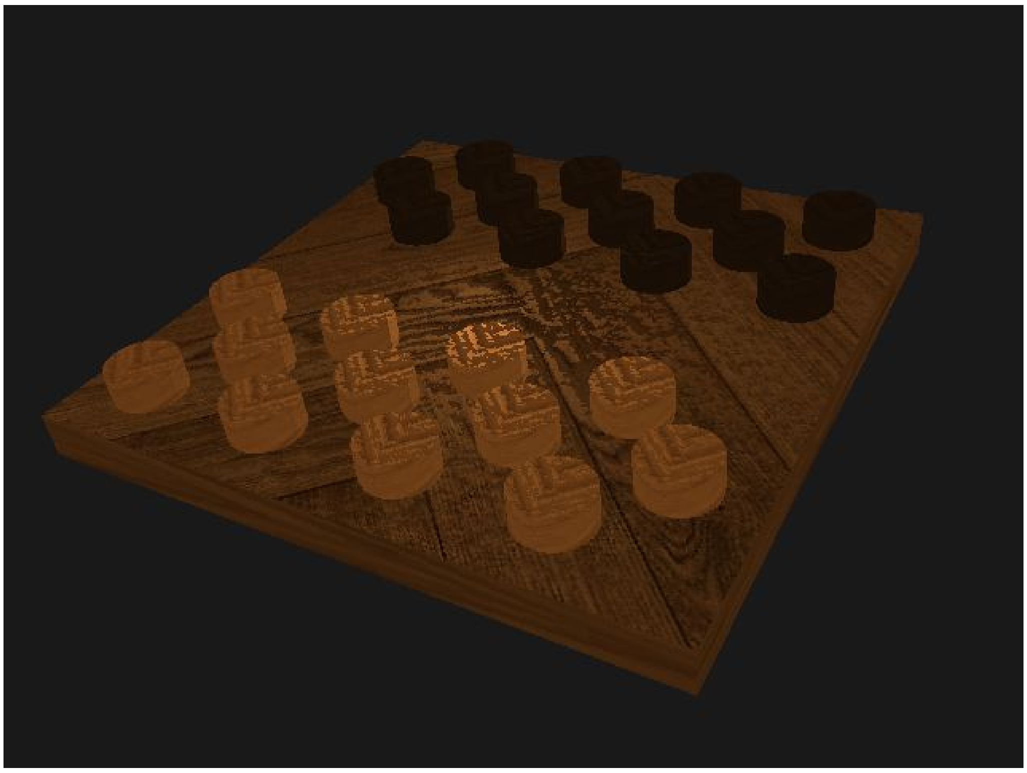
3D Animation and Visualisation 2020

Required:
Project Code + Report
(Online submission)

A game of Checkers

Our project concept was based on checkers, a board game where two players play against each other, each one with twelve pieces displayed on the board.



Marc Jelkic
84741
Leic-T
marc.jelkic@tecnico.ulisboa.pt

Guilherme Monteiro
90724
Meic-T
guilhermefam99@gmail.com

Nuno Laranjo
90760
Meic-T
nuno.miguel.laranjo@tecnico.ulisboa.pt

Pedro Leitão
90764
Meic-T
pedro.de.leitao@tecnico.ulisboa.pt

Abstract

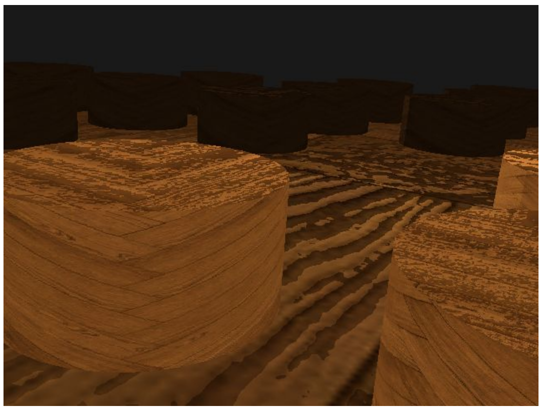
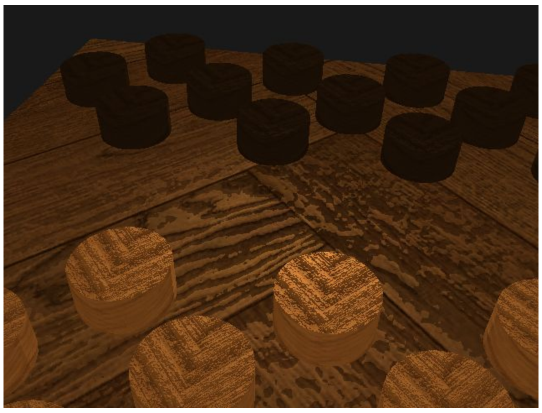
We chose to create a game of checkers due to its straight-forward yet challenging implementation of the various technical challenges.
First and foremost, we give an overall view of the project concept mentioning its description, all the different features we identified and their technical approach for their execution. This includes saving a snapshot of the scene using a third-party library, creating a realistic solid material for the game pieces and board, using a general scene graph handling hierarchical drawing for matrices, shaders and textures, rendering shader based special effects such as the bump map and finally generate a non-photorealistic lighting model which is represented by a smooth lighting on the center of the board. Some other technical challenges like mouse picking, shadows and bloom were not mentioned above because we did not manage to implement them. And finally, most technical challenges that we picked were fulfilled, yet due to poor time management and underestimating the unimplemented challenges we did not manage to complete the project on time.

1. **Concept**

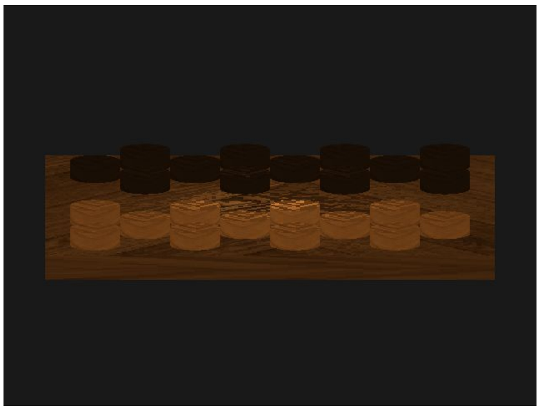
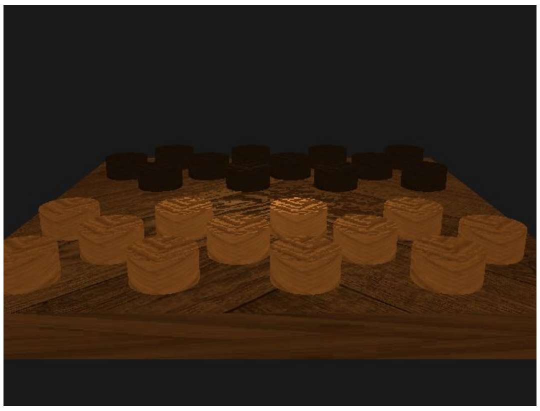
Our project concept is based on a match of checkers, which is a board game between two players, involving diagonal movement and mandatory captures. Each player controls twelve pieces, all equal between them, being usually white colored for one player and black colored for the other. In a checkers board, all the pieces are initially placed in the darker squares.

We chose the checkers concept because we find it interesting to implement a famous board game in OpenGL, and because it was simpler than chess but also challenging.

The main features of our concept are to show a board of checkers all made of wood, where the detail of the wood texture can be seen both in all the pieces and in the board, just like the detail in the colors of the pieces with the white or black painting in the wood and also the light emission in both board and pieces, coming from a point in the scene. For that, we implemented a wood texture for the objects in our scene, the Bump Map and the Blinn-Phong lighting and shading model.

To see these details better, we can switch between Orthographic and Perspective projections with the P key, and we can also orientate the camera around with the mouse and move the camera with the W, A, S and D keys.

Our engine draws the scene through a scene graph.
It is also possible to capture the current state of the scene, by clicking in the V key.
We also tried to implement, although without success, the mouse picking, where we can recognize the color of a pixel in the scene with our mouse, and we can show the color in the console by left clicking it, but we couldn’t implement the recognition of a specific piece by its color and consequently move it.
We were also unable to implement the buffer based special effects, more precisely, reflections and shadows, and the scene post processing.

2. Technical Challenges

2.1. **Saving a snapshot** [Guilherme Monteiro]

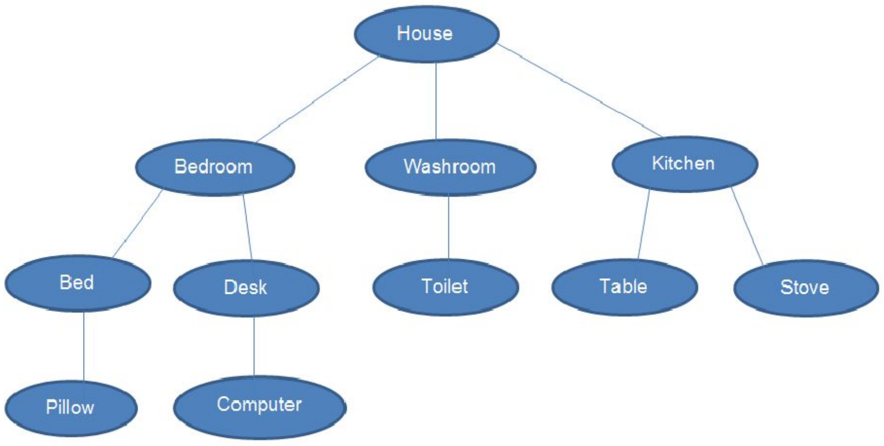
The first issue we encountered was how to save the data to a file, since OpenGL has no feature to do it. After searching we discovered a third-party library to do it using the glReadPixels() function. Another issue was deciding on what name the file should have. The best option was to use the timestamp, however files' names cannot include":" so our solution was to use ";" instead.

2.2. **Scene post processing (Bloom, Motion Blur)** [Guilherme Monteiro]

Due to the team's poor time management this challenge was not developed. We tried to reproduce the Bloom and the Motion Blur effect but the lack of time did not help. Our first try to replicate the Motion Blur was the simplest one, keeping a copy of the last framebuffer, then getting the object or the camera's moving vector and inverting it. After this we had to draw the scene and then draw the last framebuffer with some offset, we could repeat this as many times as we wanted to give the ideal effect. The Bloom effect required a lot of work to be done beforehand including lighting so it could be done as a post processing effect. The 3 shaders that were required did not make it in time.

2.3. **Generic scene graph handling hierarchical drawing** [Pedro Leitão]

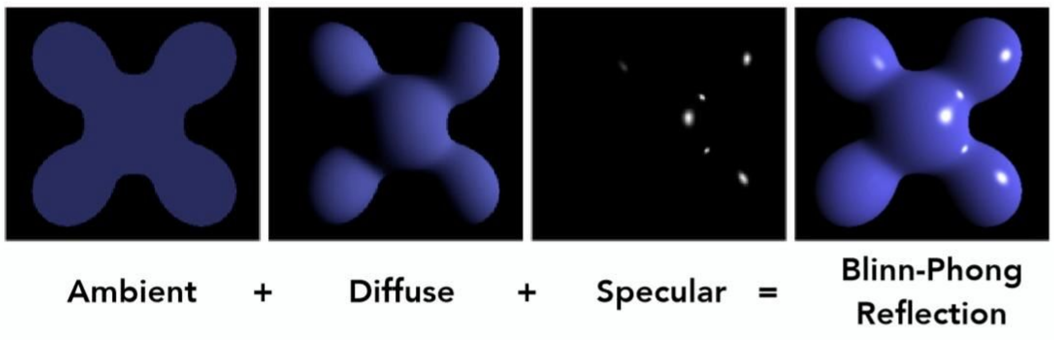
We took a while to figure out what the scene graph should do in concrete, which made us take a little longer to implement it.
When creating the nodes for each piece, when the scene was drawn we noticed it didn't draw any pieces, only the board. We later found out that the reason why that happened was because, when creating the nodes *"player1pieces"* and *"player2pieces"* (the node for all the player1 pieces and the one for all the player2 pieces, respectively) we added them as childs of the "pieces" node (the node for all the pieces) before adding all the nodes for every single piece as childs of the *"player1pieces"* and *"player2pieces"* nodes, and so those 2 nodes were assigned as childs of the "pieces" node as nodes without any childs. To fix that, we only added *"player1pieces"* and *"player2pieces"* as childs of the "pieces" node after adding all the nodes for every piece as their childs.
We also faced some runtime errors while drawing the scene and the objects in the scene were not drawn correctly. Those 2 issues were fixed by changing the *glDrawElements()* function to the *glDrawArrays()* one.



(An example of a scene graph)

2.4. **A non-physically based "photorealistic" lighting / shading model (Blinn-Phong)** [Pedro Leitão]

We faced some issues while generating the final components with all the colors, not representing a Blinn-Phong lighting and shading in our scene. That was caused because we were generating the lighting and shading components and consequently the colors in the CPU memory instead of the Fragment Shader, not showing a specific color for each pixel of the scene. To fix that, we made our light and shading calculations in the Fragment Shader.

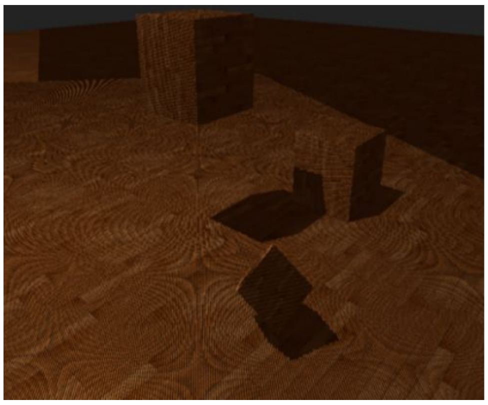


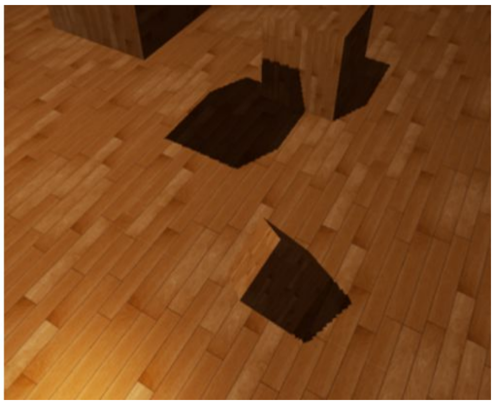
2.5. **Mouse Picking Objects** [Marc Jelkic]

This task was left unfinished due to its complexity and difficult adaptation on our base project. Shaders were a big part of the problem.
The idea of this challenge was to move each individual piece one by one by clicking with the left mouse button and dragging it to a new location on the board. We also decided to make it only move on two axis so it would always be parallel to the board.

2.6. **Buffer based shadows** [Marc Jelkic]

This task was not implemented due to poor time management and shader incompatibility. The photorealistic lighting created by Pedro Leitão was a starting point for this challenge. Creating shadows from it came to be harder than we initially planned. We aimed for a non-smooth shadow render first, followed by important improvements like removing the shadow acne.





2.7. **A realistic or stylised solid material for the objects of your scene (Wood)** [Nuno Laranjo]

The first difficulty we encountered when developing this challenge was figuring out what texture to use for it. Initially, we added the ability to load image (jpg) files and use them as textures applied to the fragment shader.
Although at some point we were prompted develop a noise function to generate the texture for the wood, it revealed itself to not look very realistic and it would also prove to be difficult to translate this noise function into one that would generate a normal map, to use alongside the main texture, which was absolutely necessary for the following challenge.
In the end, we kept the images as textures and used royalty-free images from texturise.club for both diffuse texture (texture for coloring) and normal map.
Regarding the transmission of information to the shaders, in more than one scenario, we found that we were sending information in the wrong order, which caused texture map coordinates, or other values (instead of the actual texture) to color the objects.
Another difficulty was the texture mapping of the objects, although models exported from Blender already contained the UV maps, it took a fair amount of time to import them and use them properly.

2.8. **Shader based special effects (Normal map)** [Nuno Laranjo]

This challenge was developed alongside the previous one. Many of the difficulties that had to be overcome for the diffuse texture to be applied, made it so they no longer had to be dealt with when applying normal maps.
Loading the normal map, downloaded from texturise.club, and sending the necessary information to the shaders was a breeze and very quickly we were able to display the normal map as diffuse texture for the object, for testing purposes. Applying the normal maps as normals for every individual fragment was also straight-forward.
The only main difficulty was making the normal maps work in conjunction with the lighting system created.

3. **Proposed Solutions**

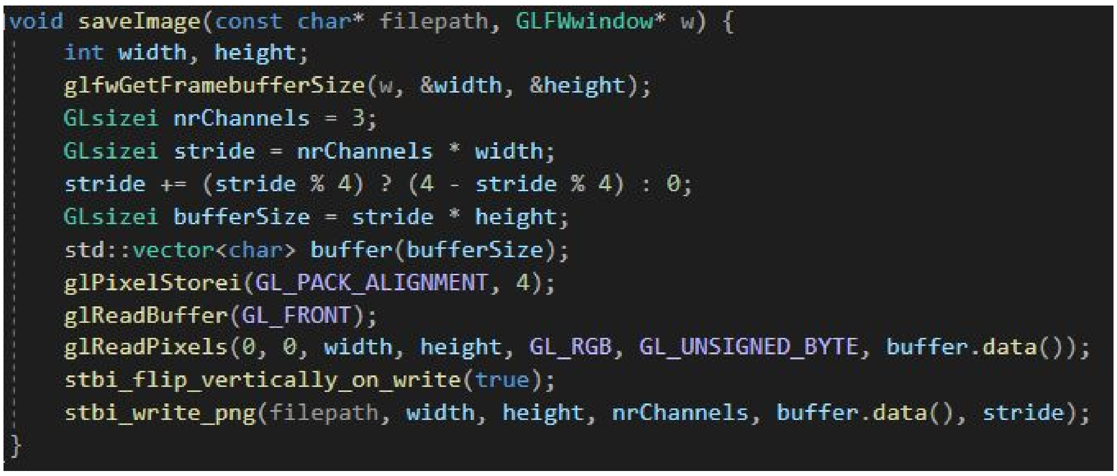
3.1. **Saving a snapshot** [Guilherme Monteiro]

3.1.1. Explored approaches

The first approach was to try and develop a function that could save data to a .png file. We decided that it would be easier and simpler to use a third-party library to do so. We started searching and we found several third-party libraries which would convert the data into a .bmp file. Since it was not ideal we then used the stb_image_write.h library. We also had to decide on how to get the window size. *glfwGetWindowSize()* gives us the logical size, however we need the size in pixels so the *glfwGetFramebufferSize()* was the solution for our problem.

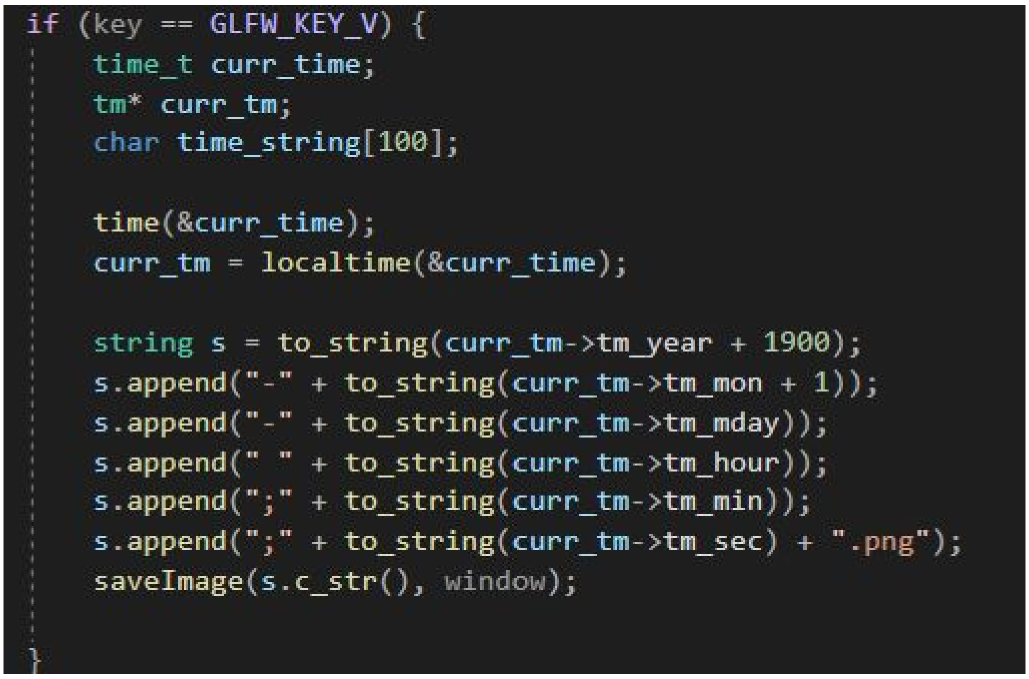
3.1.2. Final implementation

First we need the size of the GLFW window and we need to allocate a buffer. We used the *glfwGetFramebufferSize()* to get the window size. Also the number of bytes, of the formats of an image, used to allocate a single row of an image need to be a multiple of 4. OpenGL does the same thing to pack image data. So we made sure that *stride* was a multiple of 4. Then we read the image data using *glReadPixels()*, and we need to use *stbi_flip_vertically_on_write(true)* because without this the image would be flipped upside-down. This happens due to the fact that the y-axis of an image file goes downward while the y-axis of OpenGL goes upward.
Finally we use *stbi_write_png* to write the data to the .png file.
The file name is the current timestamp, we use the *time()* library to get the current time and to split the time into the different types of the timestamp. We choose the V key to save the snapshot.



```
void saveImage(const char* filepath, GLFWwindow* w) {
    int width, height;
    glfwGetFramebufferSize(w, &width, &height);
    GLsizei nrChannels = 3;
    GLsizei stride = nrChannels * width;
    stride += (stride % 4) ? (4 - stride % 4) : 0;
    GLsizei bufferSize = stride * height;
    std::vector<char> buffer(bufferSize);
    glPixelStorei(GL_PACK_ALIGNMENT, 4);
    glReadBuffer(GL_FRONT);
    glReadPixels(0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE, buffer.data());
    stbi_flip_vertically_on_write(true);
    stbi_write_png(filepath, width, height, nrChannels, buffer.data(), stride);
}
```

(the function to save the image data)



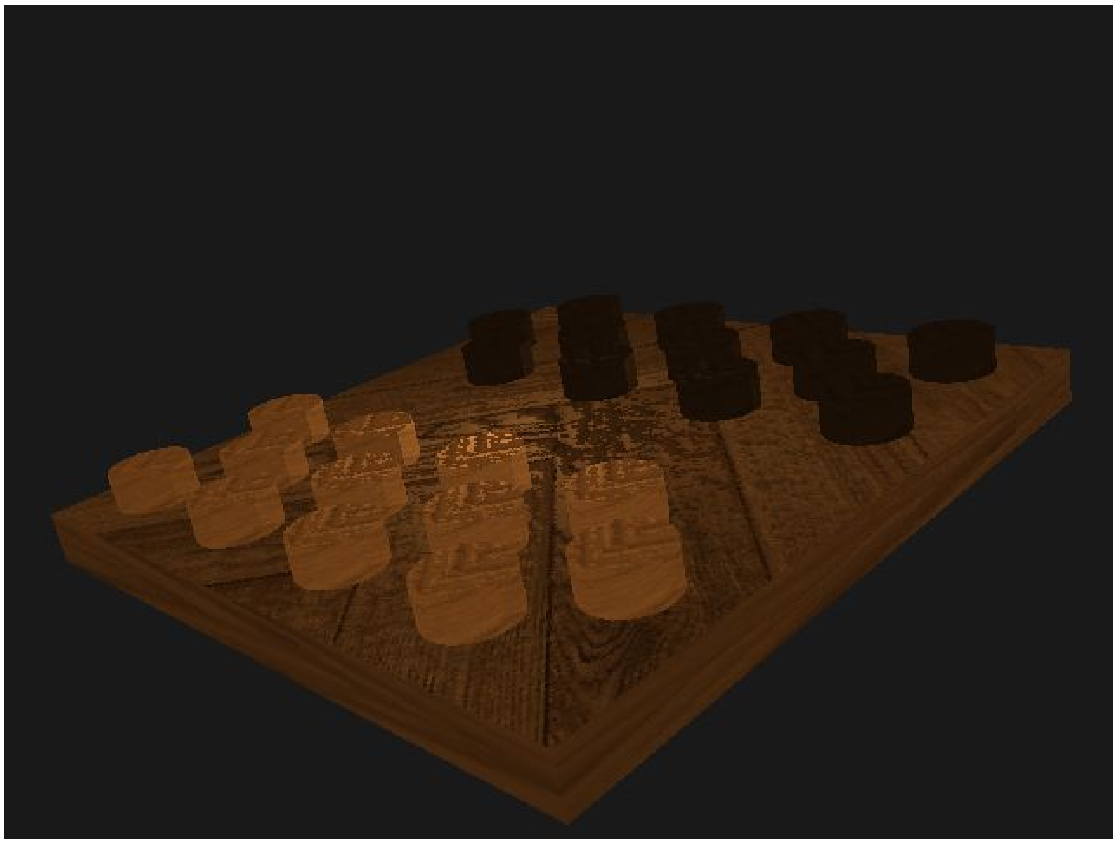
```
if (key == GLFW_KEY_V) {
    time_t curr_time;
    tm* curr_tm;
    char time_string[100];

    time(&curr_time);
    curr_tm = localtime(&curr_time);

    string s = to_string(curr_tm->tm_year + 1900);
    s.append("-" + to_string(curr_tm->tm_mon + 1));
    s.append("-" + to_string(curr_tm->tm_mday));
    s.append(" " + to_string(curr_tm->tm_hour));
    s.append(";" + to_string(curr_tm->tm_min));
    s.append(";" + to_string(curr_tm->tm_sec) + ".png");
    saveImage(s.c_str(), window);

}
```

(naming the file with the timestamp)

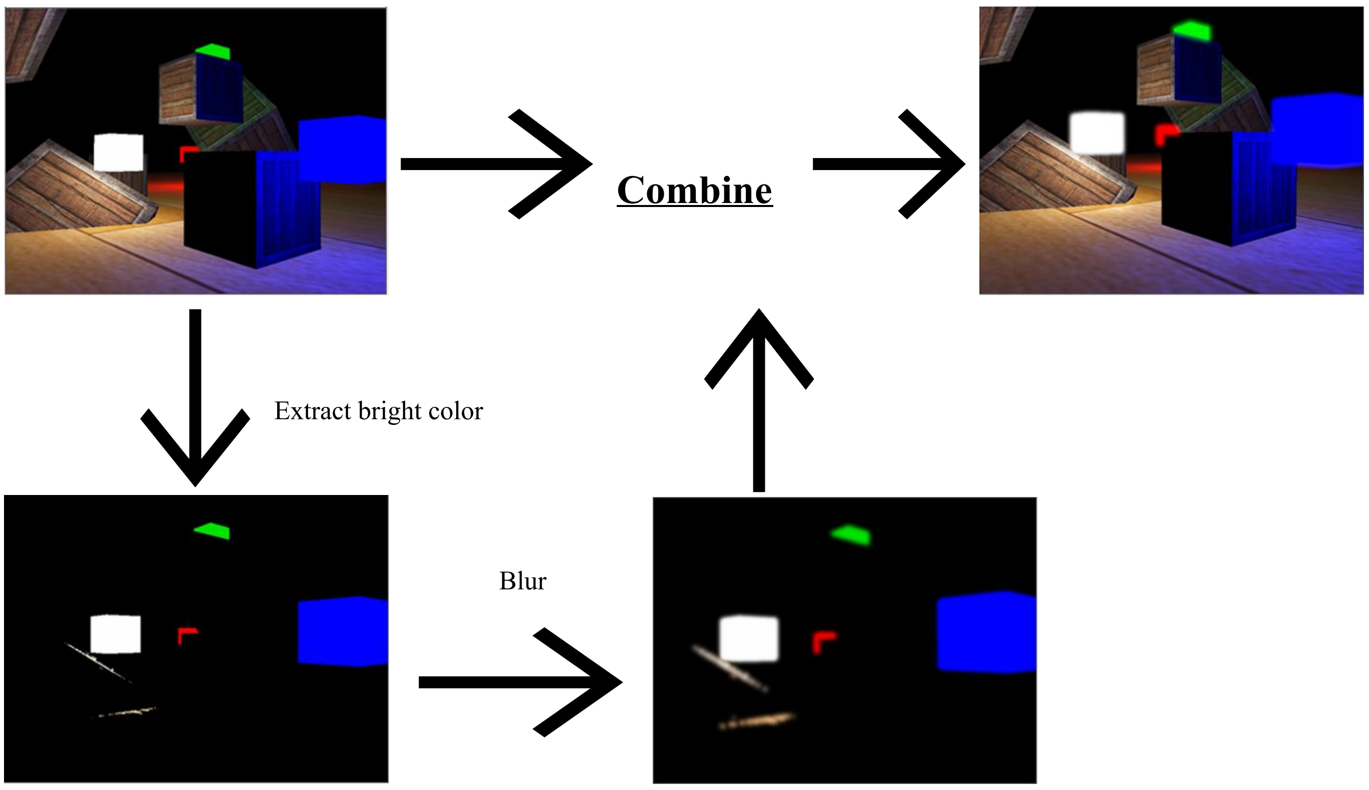


(a snapshot from our project using this functionality)

3.2. **Scene post processing (Bloom, Motion Blur)** [Guilherme Monteiro]

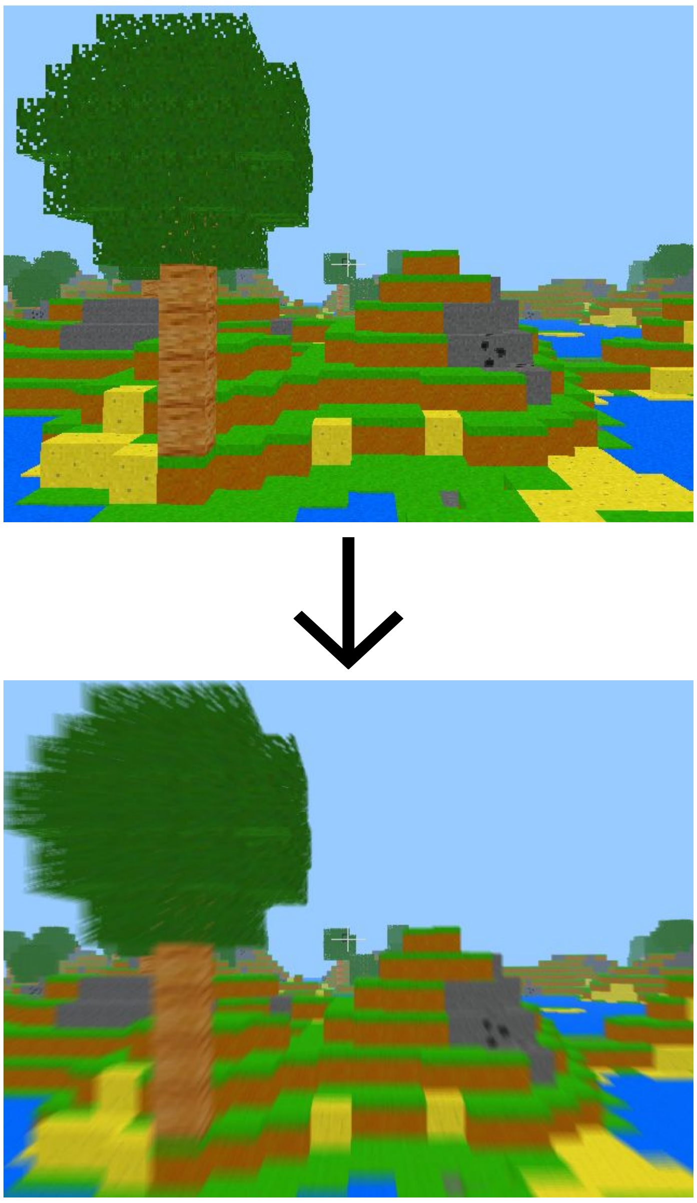
3.2.1. Explored approaches

For the Bloom effect we made a research and discovered that using 3 shaders was the optimal way of doing it.
One shader would extract all the fragments that exceeded a certain brightness, this would make the scene only show the bright colored regions. The second shader would blur this scene using the Gaussian Blur. Now the scene should be dark with only the bright colored regions but blurred. The last shader would combine the original scene and the blurred one. This way the bright regions should appear glowing.



(The Bloom Effect)

For the Motion Blur Effect there were some options of implementation available to choose. We tried to use *glAccum()* which is the accumulation buffer. However *glAccum()* is deprecated since OpenGL 3.0 and it was removed in OpenGL 3.1 so we move on to the next option.
One of the simplest methods was to keep a copy of the last framebuffer, then get the object or the camera‘s moving vector and invert it. After this we had to draw the scene and then draw the last framebuffer with some offset, we could repeat this as many times as we wanted to give the ideal effect.



(The Motion Blur Effect)

3.2.2. Final implementation

This challenge was not implemented.

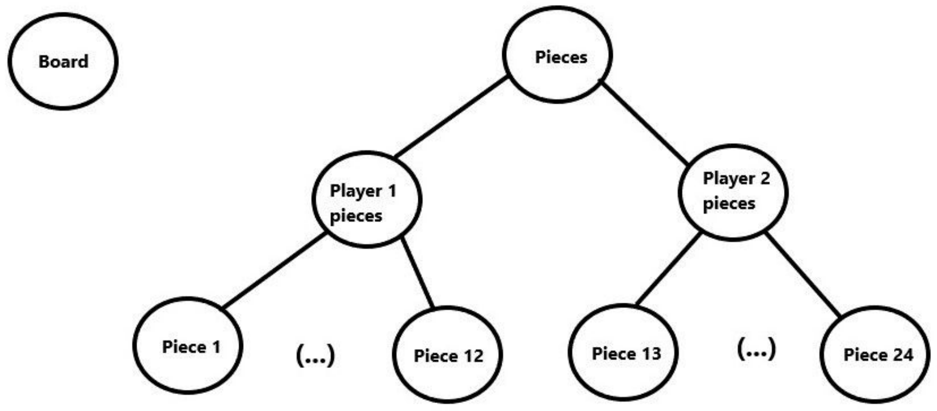
3.3. **Generic scene graph handling hierarchical drawing** [Pedro Leitão]

3.3.1. Explored approaches

Our first approach was to create every component for each node (model matrix, mesh and shader program) individually, not applying the transformations of parents to their children, causing us to not have a functional scene graph.
So, after that, we changed our scene graph to create the model matrix, mesh and shader program for each node based on the ones from his parent node.

3.3.2. Final implementation

We created 2 functions in the main c++ file, one to create the scene graph, createSceneGraph(), and the other to draw every node in the scene graph, drawScene().
We used for loops to implement both functions. Used two for loops in createSceneGraph(), one to create the nodes for every single piece for player1 and the other to do the same for the pieces for player2, were we actualizate the model matrix depending of the initial position in the board of that specific piece. We also actualizate the shader program for every new node.
For the drawScene(), we use a for loop to reach each node of the graph, with other for loops inside of it to reach every child of those nodes, and consequently draw them, by calling a function in the Renderer class, that receives the information of that specific node and calls the shader programs with that information, generating the final colors of the object and finally rendering the primitives from array data with the glDrawArraysFunction().
We created 2 different meshes, one for the board (a parallelepiped) and one for the pieces (a cylinder).



(Our scene graph)

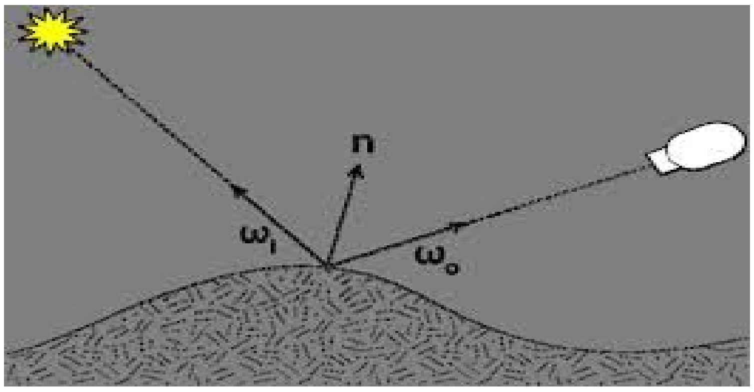
3.4. **A non-physically based "photorealistic" lighting / shading model (Blinn-Phong)** [Pedro Leitão]

3.4.1. Explored approaches

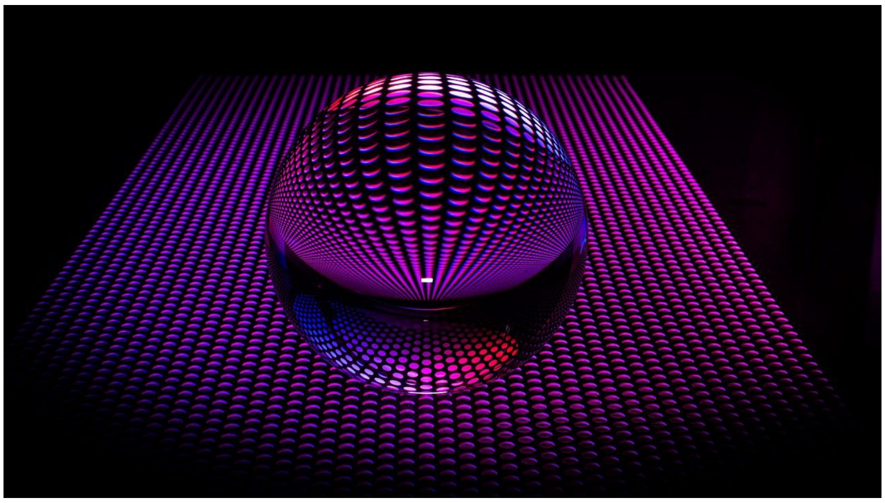
Our first approach was to calculate the Blinn-Phong components in the CPU (in the Shader class c++ file), instead of the GPU, which made us have a very ineffective lighting and shading calculation. So, we changed that to create two shader files, one for the vertex shader and the other for the fragment shader, using them to calculate the Blinn-Phong components, and consequently generating the color for each pixel, instead of for each vertex like we did on our first approach.

3.4.2. Final implementation

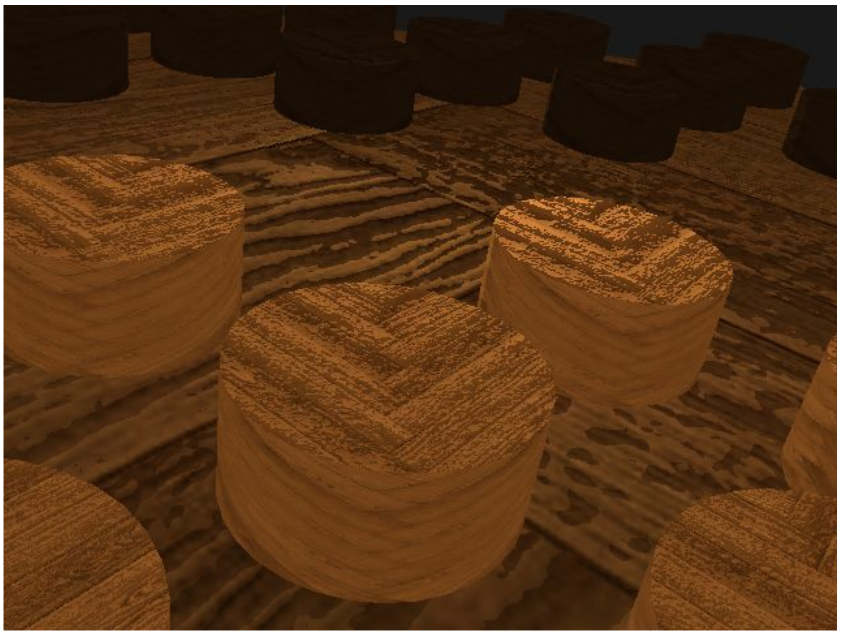
We created 2 files, the “base.vert” and the “base.frag”, which are the vertex and fragment shaders respectively, and to the vertex shader we sent all the information about each piece to be rendered, just like the mesh and the light and the material components and its painted color, and calculated on that shader the fragment position, based on the vertex positions putted into the scene.
Fragment position this that we transferred, together with the light and the material components and the painted color, to the fragment shader, were we calculated the colors of the Emissive, Ambient, Diffuse and Specular components and finally putting them together and combining them with the painted color of the object, generating the final color of the pixel to be drawn.



(Blinn-Phong Reflection Model)



(An example of the application of the Blinn-Phong model)

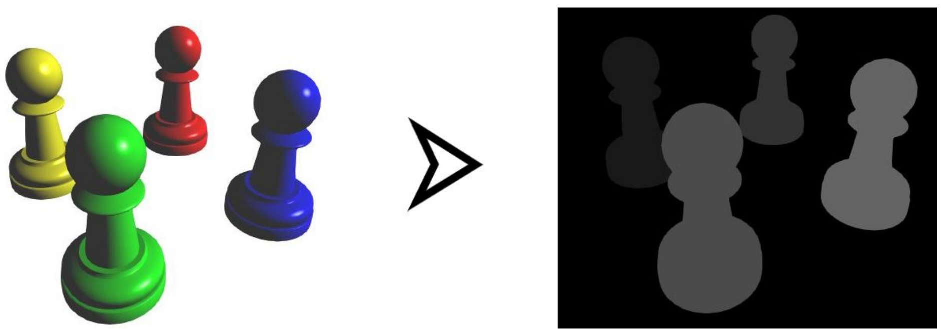


(The reaction of the lighting in the board and pieces of our scene)

3.5. **Mouse Picking Objects** [Marc Jelkic]

3.5.1. Explored approaches

After an initial research about the subject, we encountered a relevant geometric algorithm that might be a possible solution for our problem: **Ray Casting**. This idea seemed perfect at first glance but after diving into the details of its implementation and a short talk with our teacher, we concluded that this solution was too complex and demanding for what we had in mind. Our teacher mentioned **depth mapping** and **color primitives** as a way to simplify the task we had to achieve.
We started off by showing the mouse cursor on screen by changing *"GLFW_CURSOR_DISABLED"* to *"GLFW_CURSOR_NORMAL"*. Next we tried to read each and every pixel from the running application so that we can extract valuable information such as the colors (RGBA) of each piece and its depth by calling the "*glReadPixels()"* function **twice**. This part was successful so we moved on to the next part, which was identifying every game piece with a different primitive. This step was harder than anticipated due to not having actual objects to attribute color to yet.
As this part was on hold and the depth mapping was mentioned, we decided to shift our work to rendering shadows across the board. Since then, no more progress has been made. The next steps would have been to identify every game piece with a different primitive and move them accordingly.



(Primitives on objects)

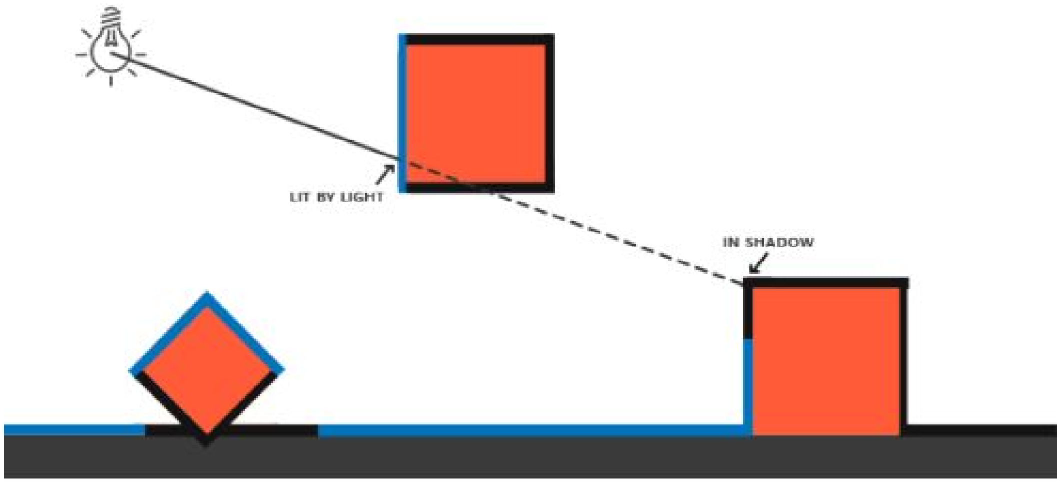
3.5.2. Final implementation

This challenge was not implemented.

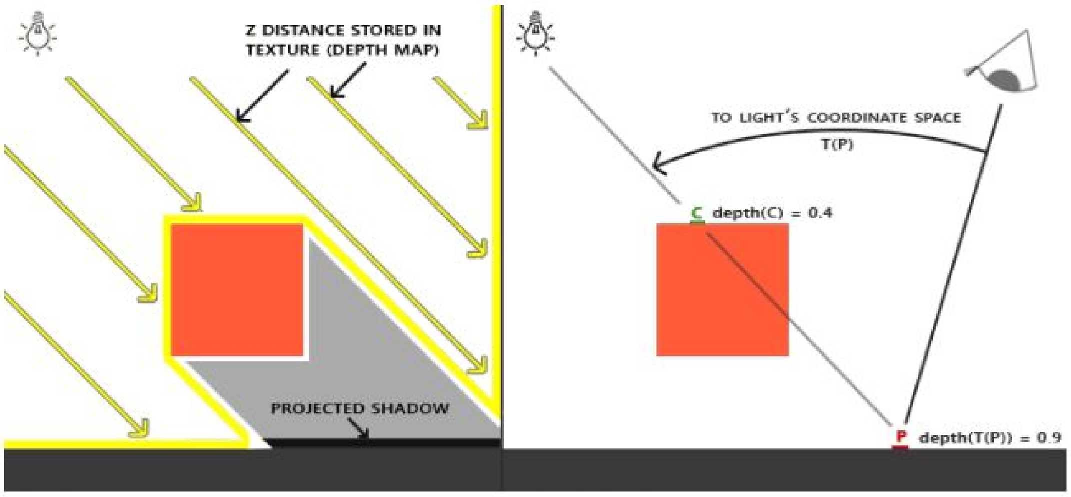
3.6. **Buffer based shadows** [Marc Jelkic]

3.6.1. Explored approaches

For this challenge we modelled our approach from the *"Learn OpenGL"* book about **Shadow Mapping.** Having some insight of what the **depth map** is about helped us understand what to do next. The idea is to first render the scene from the light's perspective and then we render the scene as normal and use the generated depth map to calculate whether fragments are shadowed or not.



(Shadowing from the light's perspective)


Z DISTANCE STORED IN TEXTURE (DEPTH MAP)
PROJECTED SHADOW
TO LIGHT'S COORDINATE SPACE
T(P)
C depth(C) = 0.4
P depth(T(P)) = 0.9

(Projecting shadows)

Following the book's walkthrough, we started working on the depth map by creating a **framebuffer** object to render the map, followed by making a 2D texture that we'll use as the framebuffer's depth buffer and finally we can attach the generated depth texture as the framebuffer's depth buffer.

Then, by changing **viewports**, binding framebuffers and rendering the scene twice, we should have been able to see some shadows. Except there's one small detail missing: **the lighting**.

We still need to configure the lighting, although this has already been done from the photorealistic lighting. By picking that spotlight we were close to finishing the shadow implementation, **yet two other problems emerged**.

The first one being the same problem the mouse picking task had: **shaders**. For the depth map to work we should have created another shader program with an empty fragment shader. This went against our base structure of the project which only included one and only shader.

The second problem was the **double rendering** made for the depth buffer mapping. For some reason, by trying to draw the scene twice it would throw a **write access violation** exception.

Unfortunately, both of these problems still persisted until the end.

3.6.2. Final implementation

This challenge was not implemented.

3.7. A realistic or stylised solid material for the objects of your scene - Wood [Nuno Laranjo]

3.7.1. Explored approaches

As previously mentioned, the first approach we took was to load image files and use them as diffuse textures for the objects. For this, we used a single-file public domain library called stb_image - it allowed us to load images into buffers and generate mipmap textures within OpenGL. We also had at one point loaded multiple textures for the two different tones of wood but we ended up replacing that with a single texture in the end. After the first delivery, we faced the criticism that the wood could be created through noise functions. We explored this approach but were only able to experiment with some very simple noise functions, however, soon we realized that it didn't look realistic, and would be very difficult to generate a normal map to use alongside this noise texture. Looking back, if we were able to generate some sort of perlin noise while also generating a coherent normal map for that texture, we would've had some very interesting results. In the end, we removed this entirely, and kept the image loading, which we thought looked far superior.

3.7.2. Final implementation

In the end, we kept the most straight-forward approach, we loaded the images through stb_image and generate mipmaps for them, which we use in the shaders. Texture mapping also was done through the values included in the .obj files which were exported from blender, imported into our program, and sent to the shader. The color differences were achieved by multiplying our only diffuse texture by a light-brown color as well as a dark-brown color within the fragment shader, giving us two different tones of wood with the use of a single texture.

3.8. Shader based special effects [Nuno Laranjo]

3.8.1. Explored approaches

Implementing the shader based special effect (normal maps) was relatively simple. We initially just sent the normal maps into the fragment shader and applied it as color for the fragments, for testing purposes. After we verified that was working as intended, we attempted to use arbitrary normals for the vertices, applied alongside the normal map, and, after that was concluded, all that was missing was using the vertex normals included in the .obj files. The normal maps work alongside the lighting system also implemented by the team, so it was a matter of experimenting with different values, light positions and intensities, before we got a result we were satisfied with.

3.8.2. Final implementation

The normal maps are also loaded with the stb_image library, they are then sent to the fragment shader where they are aligned with the surface normals, using the normals from the .obj files. The normal map is then applied in color calculation during the diffuse and specular lights calculation, which results in objects that appear to have more mesh complexity than they really do.

4. Post-Mortem (max 2 pages)

4.1. What went well?

We worked well as a team, we didn't just stick to the tasks assigned to us, but also helped each other in their respective tasks.
We learned a lot about OpenGL, computer graphics and C++ language while working on this project and also developed other skills like programming, teamwork, algebra and physics.

4.2. What did not go so well?

Although we had a weekly development plan, we did not stick to it. We did not know how to manage the time we had, in order to conciliate it with other courses evaluations.
We also took a lot of time to fully understand the real objective of most of the tasks and had some difficulty in their implementation, being not able to implement some tasks of the ones we had assigned.

4.3. Lessons learned

If we could go back to the start of this project, while taking all the knowledge we gathered while doing it, we would orientate our time better in order to fully understand how the different tasks of the project worked before actually starting trying to implement them without knowing what to do. We would also try to fully understand the theory before coding.

We learned that we need to start implementing our future solutions only when we fully understand the concept, and should not leave everything to the last days before the delivery because we can get an unexpected bug or external issue and not be able to correct that in time.

We would advise future students of this course to orientate well between the team and with the time.

References

https://www.lighthouse3d.com/tutorials/opengl-selection-tutorial/
https://github.com/nothings/stb/blob/master/stb_image_write.h
https://lencerf.github.io/post/2019-09-21-save-the-opengl-rendering-to-image-file/
https://learnopengl.com