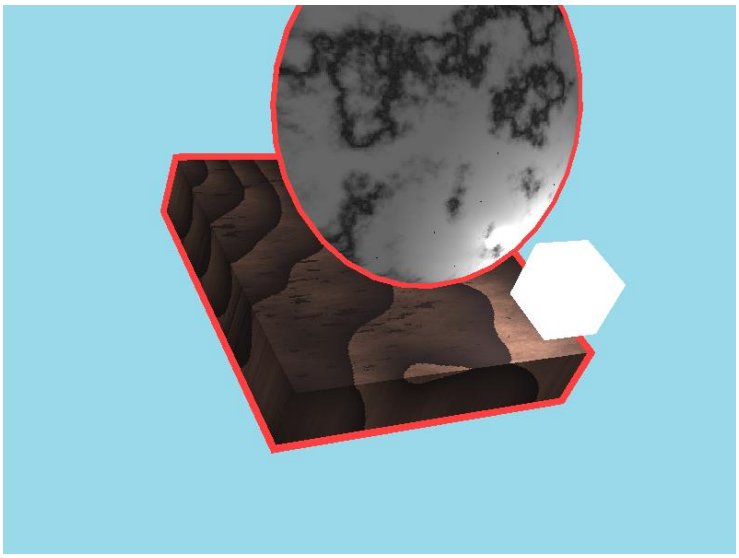
Computer Graphics for Games 2023

**Required:**
**Project Code + Report**
**(Online submission)**

# Texture Glow

An editor like application with saving,
realistic textures and lighting features in OpenGL



**<Tiago Alexandre Pereira Antunes>**
<ist199331>
<MEIC-A>
<tiago.a.p.antunes@tecnico.ulisboa.pt>

**Abstract**

An application in C++ and OpenGL. I implement a generic scene graph to ease the creation, rendering and manipulation of scenes. Scene node movement similar to already existing gizmo axis movement in other editors, by dragging the mouse and translating that movement into the objects orientation, while first requiring you to select the node you want to move first. I use the Stencil Buffer for the mouse picking and to render an outline around the selected object. Additionally I serialize the scene to a JSON file. We explore how Perlin Noise can be used to generate 3D textures such as wood and marble, and apply them to a model in the shaders. Finally, to make the scene have a realistic look, we implement the Blinn-Phong lighting model.

# 1. Technical Challenges

To illustrate the following challenges, the scene will be composed of a light, a flat cube with wooden texture, as a base, and a flat sphere with marble texture, as a floating top. The top is a child object of the top.

## 1.1. Generic Scene Graph

A generic scene graph should have generic nodes. All necessary data for drawing a node should be encapsulated in itself. This means that matrices, shaders, textures and callbacks should be handled automatically when needed. Much in the same way that other editors work, the scene graph should be a hierarchy, the position and rotation of a node are relative to its parent, so in the case of moving or rotating the parent node, the child node always keep its relative position to the parent.

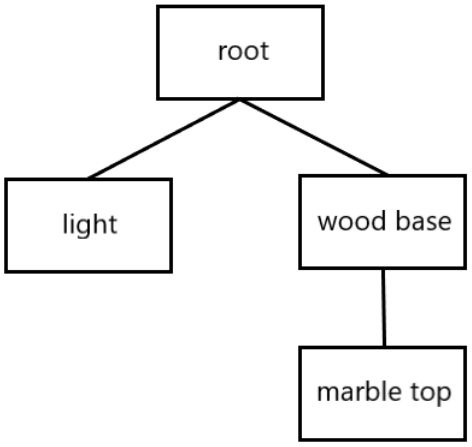



Figure 1: Pretended scene graph structure.

## 1.2. Scene Graph Serialization

An additional feature of the Scene Graph is the possibility to serialize to and deserialize from a file. This requires that we can translate all the node information into a file, of some format, and then reload the scene with the information from that file.

This should be possible to do, while the application is running, pressing a key will save in the scene and pressing another will load the scene seamlessly.

## 1.3.Mouse Picking and Node Movement

The node movement we want to implement is akin to what the gizmos allows in other editors. It should be possible to move any node in any axis we want and in the plane defined by two of them. It should also be possible to rotate them around their y and x axis. We should be able to click on a model on the screen and an outline should appear around the model of the nodes selected. Then we can drag the mouse to move the node in the direction we want and rotate it using the keys.

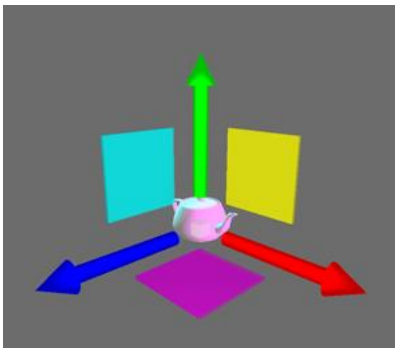


Figure 2: Planes and axis a node should move on.

## 1.4.Textures

I'll use the Perlin noise algorithm to produce a pseudo random noise, that we can then manipulate to create the textures that we want. Here we will create a texture similar to marble and another similar to wood. These textures should be created in 3D, so we'll also need to define how we map them to the model in the shaders, as if our model is a solid piece that was cut from a bigger piece.

Figure 3: Reference marble and wood textures.

## 1.5.Lighting Model

The last challenge of this project is to simulate realistic lighting on our scene. For this we'll implement the Blinn-Phong model, with different specular components for each texture used. Additionally, it should be possible to have a light that behaves like any other object, so the light position will have to be variable.

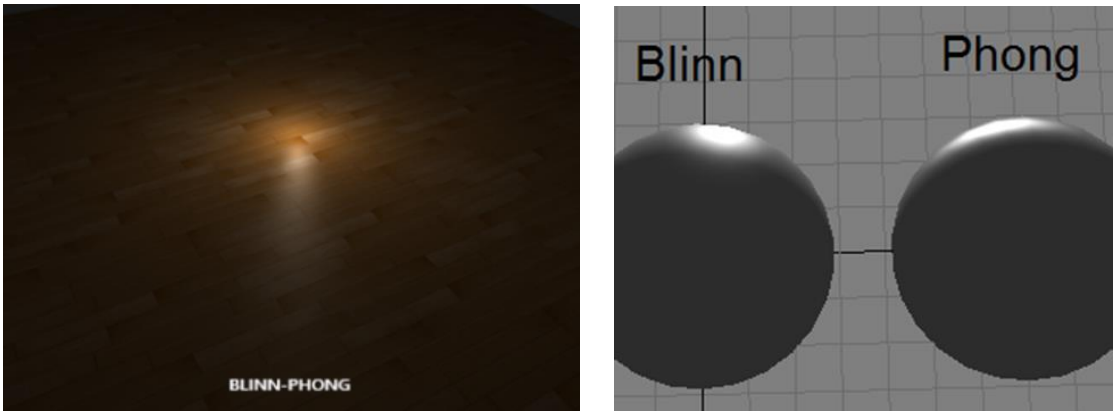



Figure 4: Blinn-Phong model reference images.

# 2. Technical Solutions

## 2.1. Generic SceneGraph

The scene graph contains a camera and a root node, which is the node that all other nodes will be children of. To draw the scene all we need to do is call, the SceneGraph class renderScene method from the application, all necessary operations are done internally. This method receives the elapsed time since the last frame, and will call the update on the camera, so it can process any rotation that might have happened on that frame. Then it calls update on its root node. Another important feature is the ability to search for a node, for this it calls the search function of the root node.

For this to work our nodes are also generic, each node must contain an id, which is a number (root is always 0), a pointer to its parent (nullptr in the case of the root), as well as it's children nodes, if they exist. The node can have a mesh, shader program, callback and texture, storing the name and a pointer to the object for all. We only supply the name to the node, the object is obtained automatically from the manager at this point, to avoid always searching for it. The previous are all optional and are defined per node, if a node doesn't have one of these, then it will simply not do anything that requires it (e.g. a node without mesh or shader will not be drawn on screen). Additionally, I defined node Position, Rotation and Scale as separate properties of the node, these are relative to the parent, and are used to update the model matrix used for drawing. I did this so it would be easier to obtain the nodes transformations to then apply movement on it. This way to obtain the rotation of the node, I just need to return the rotation quaternion with no added work. The global position and rotation of the node are also available, and are very easy to obtain, the first can be obtained directly from the model matrix while the second is obtained by multiplying the nodes' rotation with its parent' rotation successively until the root.

The nodes' update method handles the update of the pending transformations for that frame, movement and rotation, these are stored in the node as additional variables and are what the user will generally add to, to get a smooth looking movement over time, since these are applied considering the elapsed time between frames. After these are handled it will draw the node, which will call the callback if it exists, bind the nodes'

shader, update it's model matrix with the transformation properties and draw the mesh. After that it will call update on it's children nodes so they can do the same.

The node's search method receives the id of the wanted node, if it's id is the wanted one it returns itself, otherwise it will call search on it's children until a match is found. If no match is found, nullptr is returned.

## 2.2.Serialization

For the serialization format I chose JSON, due to it's ease of use and availability of libraries. The library I used was Json for Modern C++ (check references [1]).

Before implementing serialization, I also created managers for mesh, shader program, callback, texture and sillouettes. I did this to ease how we save those values for the node. A manager contains a map, with a key which is a string, and a pointer to the object of type we are saving in that manager. It is implemented as a template class, so all managers are the same, just using a different type.

The serialization and deserialization of the scene can be done while the application is running seamlessly and are handled by their respective methods of the scene graph. Serialization starts by calling the serialization method of the root node, which returns a json object, provided by the library. We then add the json object returned by the camera serialization to it. Lastly, we write the json object to a file.
The deserialization works similarly, additionally deleting the existing camera and nodes. It reads from the file a json object, deserializes the camera, and then creates the root node and passes the json object to its deserialization method.

In the nodes' serialization we create a json object, to then return, as previously stated. Here we save the nodes' model matrix, position, rotation and scale using the glm to string methods and mesh name, shader name, callback name, texture name and sillouette name. We then create a new json object to store the children nodes' information and call serialize for each of them using their id as the key in the json object. Therefor the root, the base json object, will not have a serialized id, however, this is fine since we assume it to have id 0.

The deserialization method of the node receives the json object of the node we are deserializing. We retrieve all the information saved with custom functions for deserializing the mat4, quat and vec3 and assign the mesh, shader, callback, texture and sillouette by searching the saved name in their respective managers. Then as usual, we call the deserialization on each children node. For this we first create the node, with the key in the nodes' children json object and then call deserialize on it with the value assigned to that key.

## 2.3.Mouse Picking and Node Movement

For knowing what's in the position our mouse cursor is on when pressing the right mouse button, I use the stencil buffer. It allows us to give each pixel a number (8 bits), which will be the stencil value given to the fragment drawn on that location of the screen.

First, I enable stencil testing, setting the stencil operation to keep the value in the stencil buffer if the fragment fails the stencil test or the depth test, and to replace if they both succeed, this way we'll get the expected behaviour. Then we need to enable writing, by setting the mask to write on all 8 bits. The last step of the setup is to set the number we want to clean with, which happens at the beginning of each frame, in my case, I chose 255 (all bits to 1), the max number the stencil value can have, to minimize clashes with existing nodes.

I use a callback, which, before drawing the mesh, will set the stencil function to always pass the stencil test and the value will be the node id. With this, all fragments belonging to this object will have the same value as the node id.

Picking the object is done in the mouse button callback function. If the right mouse button was pressed, we call 'readPixels' with the mouse position (converted from glfw to OpenGL coordinates) specifying that we want the stencil value, which we put into a variable that keeps the id of the currently selected node. Here, I search the id on the scene graph to tell the node that it is selected, making it draw a sillouette around itself. A detail of this implementation is that because moving the parent node will move the child node, when telling the node that it is selected or deselected, we do this for all it's children as well. The sillouette is drawn at end of the draw of the node. The process is identical, differing in the callback and shader used and the scale applied to the mesh. In the callback we set the stencil function to pass if the stencil value is different from the id of the node, making it appear as an outline since the scale is bigger. We also disable writing to the stencil by setting the mask to 0 on all bits. After drawing the callback will set those values back to how they were.

Now let's see how we can move the objects. Our movement works by dragging the mouse, so we'll naturally implement it in the cursor callback. If the right mouse button is clicked, we'll see if any node is selected and we'll get a pointer to it. Then, to drag the object in their axis correctly we need to obtain their direction as we would see them if these vectors were drawn on screen, which we do by multiplying the unitary vector for each direction with nodes' global rotation. Each axis has a keyboard key assigned, and their movement is completely independent. The next steps are done for each axis without any knowledge of each other. If the axis key is pressed, we'll get the camera up and side vectors, and do a dot product between each of these and the direction of the axis we are processing. This will give what I call a multiplier for mouse movement on the x and y, that conditions how much the object will move depending on where you're seeing it from (i. e. if you're looking down on the z axis and trying to move in this axis the object will barely move, while if you're looking down the y axis and move on z it will move much more). The movement vector is then determined by multiplying the direction of the axis, this time without rotation (x is (1.0f, 0.0f, 0.0f)), with the mouse position difference in each axis multiplied by multiplier and summing both. The movement is rotated by the rotation of the node, so it is in the correct direction and is then added to the frame movement that gets accumulated during the frame.

Lastly, node rotation works by pressing keys that will rotate in each direction around an axis (only x and y axis). I implemented this on the key callback function by once again getting a pointer to the selected node. If a node is selected, we'll get the node x

and y axis by multiplying the axis with the node's rotation and then we determine the angle to rotate around each axis by reading the keys. We translate the rotation around each axis to a quaternion, and then obtain the combined rotation which we send into the frame rotation that gets accumulated during the frame.

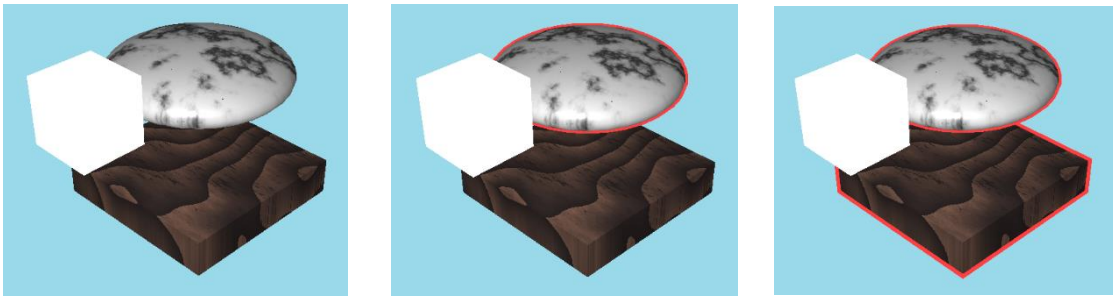


Figure 5: Default scene look. First, nothing selected, then top selected and finally base selected.

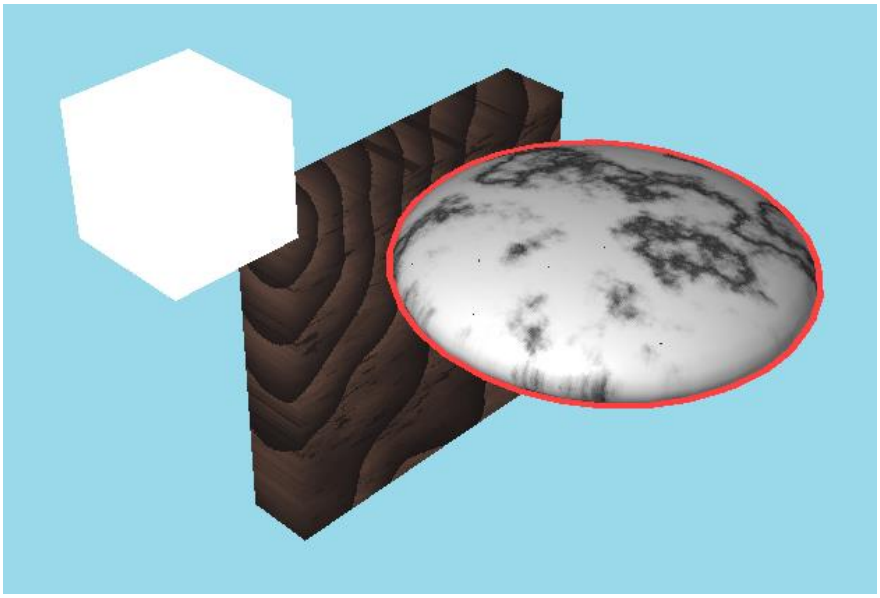


Figure 6: Scene configuration resulting from: select base -> rotate 90 degree (right arrow) -> rotate 90 degree (down arrow) -> select top -> rotate 90 degrees (top arrow) -> drag up on y axis.

## 2.4.Textures

For generating the base noise needed to create the textures from, I chose to implement the improved Perlin noise algorithm for 3D noise, based on some implementations I found online (check references [2]). This algorithm takes 3 numbers, which represent a position in space, like a 3D vector. The first step is to place this point on the grid, where points are defined for whole numbers and fractional numbers lie in between these. Here we need to determine the floor of each number and cap it to 255, to determine the starting coordinates of the unitary cube we are working on. After that, we calculate the value of the fade function for the fractional part of each number. This

will give us the interpolation value to later use to blend the values on each coordinate. To determine the gradient vector, we get the value from the permutation vector (from the original implementation) indexed by the whole x coordinate, then the one indexed by the previous added with the whole y coordinate and finally the one indexed with the previous added with the z whole coordinate, and then we use the 4 least significant bits of this hash value to determine our gradient vector from 12 possibilities, for each corner of the unitary cube. With this we now interpolate the dot product between the position inside the unitary cube and the gradient at each point of the cube, on each dimension, with the values obtained from the fade function. (Note: in the implementation, the gradient vector is implicit in the grad function, we never actually generate it, we just return the result of its dot product with the point inside the cube) In the end we get a number between -1 and 1.

To get a smooth and natural noise, we'll need one more step. I implemented a harmonic noise function that takes the number of octaves we want to sum the result of, a frequency value, a persistence value, and the x, y and z coordinates and returns the value of the smoothed noise on that point. Each octave is the result of calling the Perlin noise like follows:

*noise(frequency^octave * x, frequency^octave * y, z) / amplitude*
*amplitude = persistence^octave*

When summing each octave to the total value of the noise, it is divided by the sum of the amplitudes of all octaves, this will normalize the value but still give more significance to those that had higher amplitude. We get an final value between -1 and 1 which we then always change to a range between 0 and 1 before applying the operations to generate wanted textures.

To create each texture, these noise values are manipulated.
For wood, I generate 3 main layers. The first produces the look of wood, and it's obtained from multiplying the smoothed noise by 20 and then subtracting its floor, giving a value between 0 and 1 which will produce a degrade like effect between the hard lines of the wood. The second layer add some detail, it is again smoothed noise but this time I give the smooth noise function x and y scaled differently, which produces a stretched noise. The third is to add grains to the wood, it consists of producing another stretched noise texture and a very fine noise texture and mixing it with the previous, manipulating this so that everything is white except the darkest parts, which will have a look similar to wood grains. The final result is the multiplication of the 3 layers.

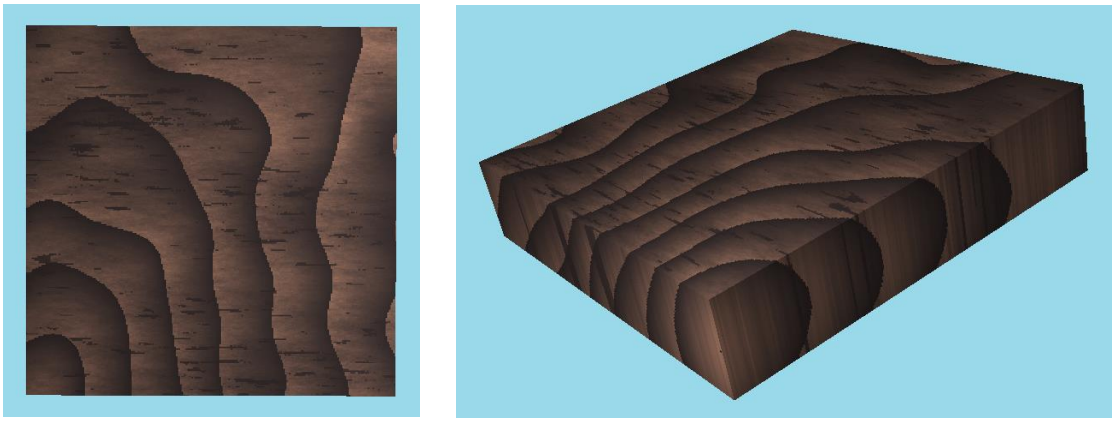


Figure 7: Wood texture composed by the 3 layers.

For marble, I start from a sin pattern as a base. The value we send to the sin function is the sum of the line frequency in the x and y axis, obtained from multiplying their value (the value passed to the noise function, between 0 and 1) with factors that define the wanted frequency. To these frequencies we then sum the value of the smoothed noise multiplied by a turbulence factor which will result in less straight lines the higher it is.

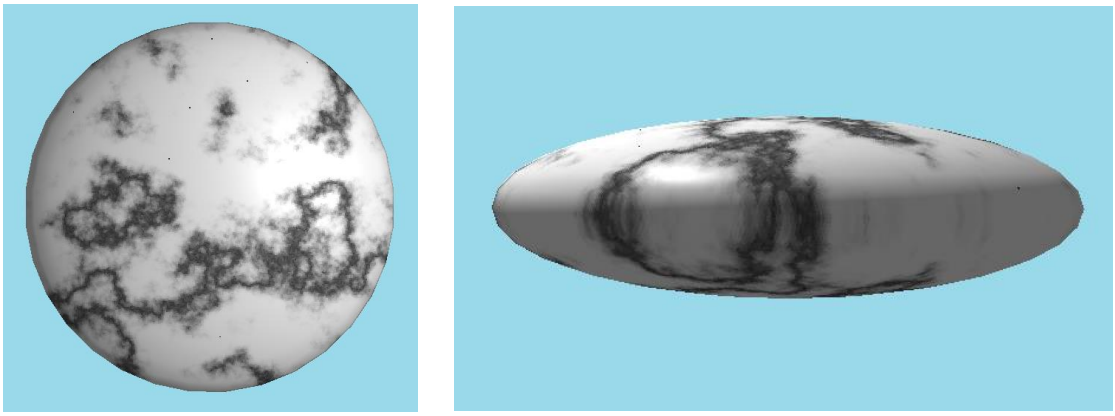


Figure 8: Marble texture with period of 0.5 on x, 1.0 on y and 5.0 of turbulence.

In the shaders, 3D texture is mapped to the object, for this I created a 3D vector that I pass to the fragment shader as input, which takes the vertex position and translates it to a vector where all coordinates are between 0 and 1 so we can get the texture. Since we created the texture fully on the CPU side, we simply retrieve the value of the texture at that point and use it to interpolate between the color of the lighter and darker spots of the texture.

## 2.5.Blinn-Phong Lighting

To implement Blinn-Phong lighting the first thing to decide is in which space we want our calculations on. To simplify and not need to send the camera position to the shaders, everything is in view space.
On the vertex shader I obtain the position in view space, as well as the normal of the vertex by multiplying the mesh normal with the transpose of the inverse of the model view matrix. On the fragment shader, I perform all the lighting calculations, first I

calculate the light position in view space and determine the direction from the fragment position to the light, called light direction.

In the blinn-phong model the light affecting a point is composed by 4 components, emissive, ambient, diffuse and specular. To control the contribution of these to the final value I created a strength variable for each. The emissive component is simply a float, none of our materials are emissive so it is always set to zero. The ambient component is obtained by multiplying the light colour with the strength of the component. The diffuse lighting is obtained from the dot product between the fragment normal and the light direction, in case it is negative, it will simply be zero, because the face is not receiving light. This value is then multiplied with the component strength and the light colour. The specular component is obtained by first calculating a half vector, this vector is the sum of the light direction with the direction from the fragment to the camera, which can be obtained from the fragment position by negating it, since in view space the camera position is the origin of the space. Then we do the dot product between the normal and the half vector, in the same way as the diffuse lighting and then do the power of this value to a specular value, lower values produce a rubber look while higher values produce something closer to metal. For marble we use 64 and for wood 8. Then we multiply this value with the component strength and light colour.
To determine the attenuation, I used the formula 1 / (a + (b * distance) + c * distance * distance), where a = 1.0f, b = 0.09f, c = 0.032f, which produces a mostly linear drop-off until the squared distance component surpasses the linear one.
The final light value is obtained from summing the emissive component, the ambient component and the attenuation multiplied by the sum of the diffuse and specular components. This value is then multiplied by the texture colour to give the final colour of the fragment.

To create a light that we could move in the scene, I implemented a point light node as a subclass of the scene node. This node is only special in the sense that it needs to create a uniform buffer to put its position, and update it whenever it is changed so it can be accessed in the other shaders. Due to this the serialization also needs to store the nodes' uniform binding point for this node and a type field for all nodes, since not all nodes are equal now. The light colour is defined in each shader.

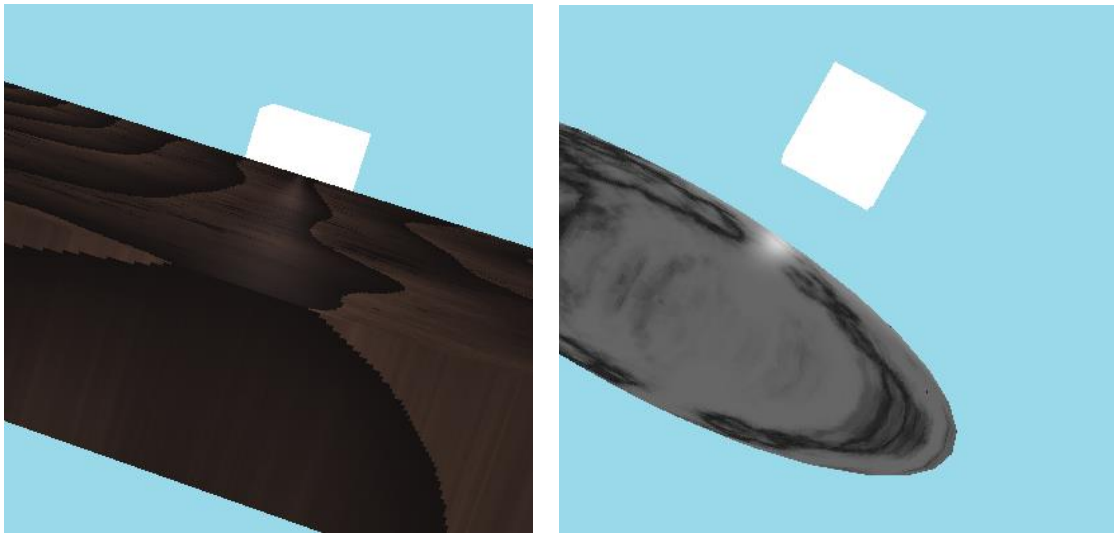


Figure 9: Specular effect on wood and marble. The results obtained are similar to reference images (see Figure 2).

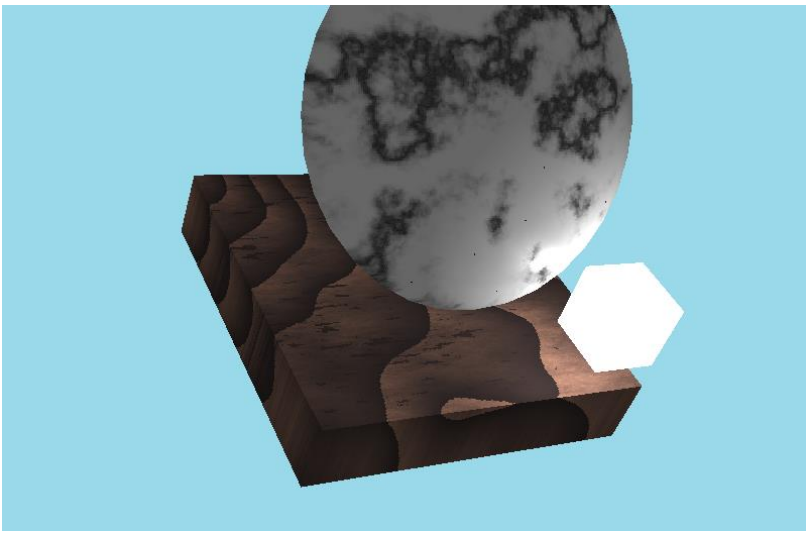


Figure 10: Overall scene lighting. Marble shows to be shinier than wood.

# 3. **Post-Mortem**

## 3.1. What went well?

The scene graph and node design is very modular, this meant that adding new nodes and features to nodes towards the end of development was very fast and easy to do.

The movement of the nodes also came out simpler than I expected, it offers a great freedom of movement, with few lines of code. Storing the rotation of the node especially as an individual component also came in handy for this.

The managers made adding and removing objects of the types there was a manager for very fast, since I made a function to add an element to each of these, all I needed to do was call it with a name and one or two values. This also made serialization of these very

simple. The creation of new managers for other types only requires 2 lines, which I also appreciated.

### 3.2. What did not go so well?

When trying to create 3D texture initially the size of the 3D array was too big and would crash the application. Then I tried to create a 2D texture with vectors for each level of the texture, however the resulting texture didn't look right. The problem was that by allocating 2D vector the memory positions weren't continuous like the OpenGL texture function was expecting so I created a 1D vector with enough space for the 2 dimensions and used it like a 2D vector.

Although the node search works well for normal nodes, when I tried to generalize the camera as just another node, I had to explicitly convert the type to access what I needed because it always returns a node of the base type. This seemed to give me some troubles in the end I kept the camera as a special component of the scene graph.

Deserialization for the light node and the camera is not fully handled by the class itself. We need special code in all nodes to deserialize a light node specifically, which would not be manageable if we had many node types, and for the camera this is done in the deserialize function of the scene graph which is not ideal.

Due to making the lighting model in view space, while I previously had the normal matrix calculated on the CPU side for efficiency reasons, I now have it on the shader, because I have no easy way of obtaining the currently active camera view matrix from any node.

### 3.3. Lessons learned

The main lesson is to make changes incrementally, making sure everything still works after each, especially when working with something that we can't easily test, shaders. I also learned some C++ things that would avoid many problems I had through the development.

## References

[1] Json for C++ | https://github.com/nlohmann/json

[2] Perlin noise reference implementations | https://github.com/daniilsjb/perlin-noise/blob/master/db_perlin.hpp | https://github.com/sol-prog/Perlin_Noise/blob/master/PerlinNoise.cpp