Computer Graphics for Games 2024/2025

Explosive Chaos: Jinx's Playthings

A 3D scene inspired by *Jinx* from *Arcane*, showcasing her chaotic "toys" with Blinn-Phong lighting and a mix of realistic and stylized visuals, highlighting a cel-shaded *Hextech* crystal held by her iconic monkey.

Laura Romero Velasco Cunha

ist1112269

MSc in Computer Science and Engineering

laura.r.velasco.cunha@tecnico.ulisboa.pt

Abstract

This project presents a 3D scene using *OpenGL*, inspired by *Jinx* from *Arcane*, showcasing her explosive personality and "toys".

The *Blinn-Phong* lighting model was implemented to achieve realistic illumination, combining ambient, diffuse, and specular components, using a half-vector for realistic highlights and light attenuation to enhance distance perception.

To highlight key objects, as the *Hextec* crystal, a *Cel Shading* lighting model was used, assigning colors based on discrete intensity thresholds, creating a sketchy appearance, with a dark silhouette emphasizing the form, through make a second render of the mesh with expanded normals and *front-face culling*, and a rim light for edges' glow, mirroring the series' aesthetic.

Normal mapping enhances surface details, simulating depth and intricate textures like metal roughness and engraves using RGB normal maps that modify light interaction.

Procedural textures were generated for realistic wood, marble, and brick surfaces using *Perlin* noise, multi-octaves, and gradients. Wood textures employed smooth noise with linear interpolations, while marble and brick utilized rougher noise, accumulating values at different frequencies and amplitudes with cubic interpolations. All resulted in grayscale pixels subsequently mapped to colors.

The scene's complexity is managed by a *scene graph*, ensuring efficient handling of transformations, shaders, hierarchical relationships, among others.

1 Technical Challenges

In this section, the main issues and technical challenges tackled during the project development are described, along with what each challenge was aiming to achieve.

1.1 Generic scene graph

This challenge involves building a **scene graph** with scene nodes to manage **hierarchical** relationships between objects, transformations, shaders, textures, matrices, and more.

For this, I **adjusted the implementation** of the scene graph used in the **3D Tangram** project, which was inspired by the **graph provided in the theoretical classes** (figure 1), in order to match with the new requirements of this project.

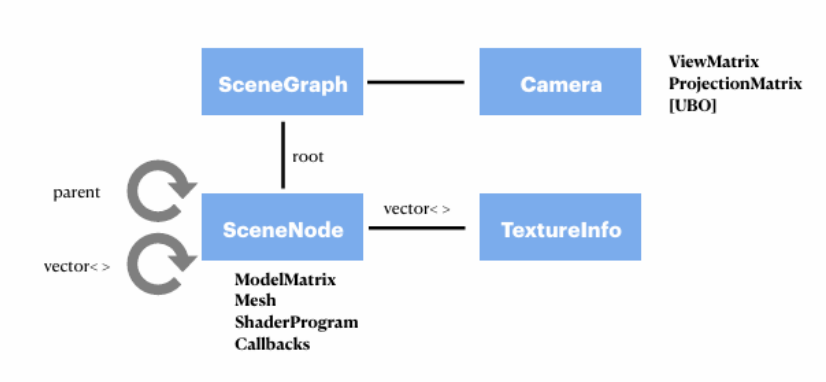


Figure 1: Scene Graph of theoretical classes

1.2 Illumination models

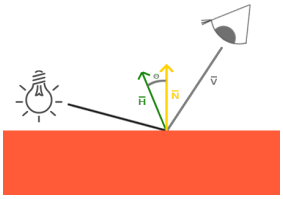
1.2.1 Non physically-based "photorealistic" lighting model: Blinn-Phong

This challenge focuses on implementing the ***Blinn-Phong* illumination model**, a non-physically based photorealistic lighting model.

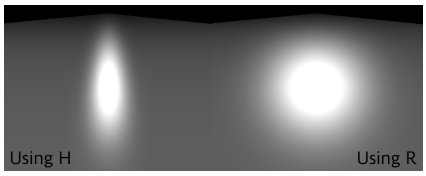
As a local illumination model, it only considers the interaction between light, the surface, and the camera position (view direction). The model uses **diffuse** (uniform light), **specular** (highlights), and **ambient** (indirect light) components, alongside **material** properties, to calculate final lighting, to achieve visually appealing highlights and specular reflections without the computational overhead of physically-based rendering.

The *Blinn-Phong* model also has the peculiarity that instead of only using the view and light directions, as *Phong*, it calculates an intermediate vector (**halfway vector**) for more efficient calculations in order to get a more realistic specular reflection (as shown in figure 2a, 2b), it also makes use of the object's normals and a shininess factor (figure 2c).

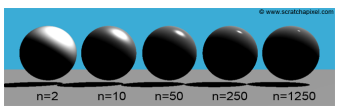
Knowing that, the objective is to create a **vertex** and **fragment shaders** that perform these calculations and integrate them into the scene, applying them to the majority of the objects.



(a) Blinn-Phong vectors [1]



(b) Phong vs Blinn-Phong specular reflection [2]



(c) Shininess factor [3]

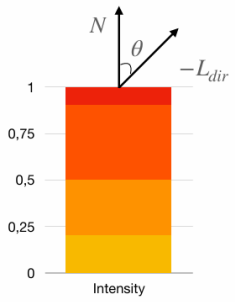
Figure 2: Blinn-Phong lightening model

1.2.2 Non-photorealistic lighting/shading model with silhouette: Cel Shading with Rim Light

This challenge addresses implementing a **cartoon-like aesthetic with flat color bands, defined silhouettes, and a glowing rim effect**, using a non-photorealistic lighting/shading model.

Cel shading involves using banded diffuse light to apply material colors, where surfaces are colored with solid, flat colors without smooth transitions, by using specific threshold values, the object's normal and the negative of the light direction (Figure 3a). The **silhouette** emphasizes object contours in darker tones (figure 3b), and the **rim light** creates a glow along the object's edges, making it appear that is emitting light (figure 3c).

The objective is to implement **vertex and fragment shaders** for the monkey's crystal ball, aiming to produce a blue-toned sphere with a dark silhouette and a light blue rim.



(a) Cel shading vectors (from slides of theoretical classes)

(b) Cel shading with shillouette [4]

(c) Cel shading with rim light [5]

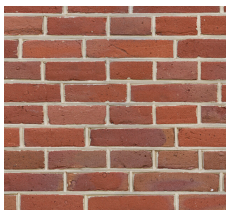
Figure 3: Examples of Cel shading

1.3 Realistic or stylized solid material using procedural noise: Wood, Bricks, Marble

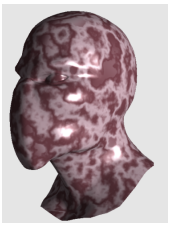
For this challenge, the objective is not to create a **noise function** from scratch but rather to utilize an existing one and **adjust its parameters** to generate various types of procedural, realistic, or stylized solid material textures. These include textures such as **wood** and **bricks**, inspired by the patterns shown in figures 4a and 4b, and **marble**, where the aim was to replicate patterns similar to those in figure 4c while using the color palette of the figure 4d.

It's also important to refer that the textures are **generated separately and then applied** to the objects in the scene.

(a) Wood pattern reference [6]



(b) Bricks pattern reference [7]



(c) Marble pattern reference (from slides of theoretical classes)

(d) Marble color reference [8]

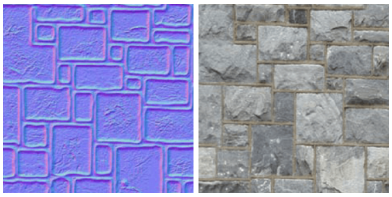
Figure 4: References of diverse types of materials

1.4 Shader-based special effects: Normal Mapping

This challenge aims to **simulate surface details** such as roughness in metals, engravings, and object shapes to make traces more realistic, without increasing the polygon count.

It works by using a special texture called a **normal map**, which stores information about the surface's normals, which are the perpendicular vectors to the surface at each point, and these are represented using **RGB** colors, where each color corresponds to a direction in 3D space (figure 5a).

The goal is to apply this normal map texture to the objects, adding it to its **shaders** in a way that alters how light interacts with the surface, creating the illusion of depth, bumps, and intricate details (figure 5b), by modifying the surface normals during lighting calculations, without adding more geometric detail to the model.



(a) Normal map and texture [9]

(b) Normal Mapping applied to an object [9]

Figure 5: Normal Mapping

2 Technical Solutions

In this section, the solution approach taken in the final scene is explained in detail, outlining how the previously mentioned technical challenges were addressed, detailing the *C++*, *OpenGL*, and *GLSL* implementations.

2.1 Generic scene graph

For the scene graph, the implementation is based on the following classes:

- ***Scene* class**:

  The *Scene* class (figure 6a) contains a *SceneNode*, which serves as the **root** node, the camera and its UBO, as well as update and draw functions, which rely on a *GraphicsAPI* that provides all the necessary ***OpenGL*** functions for rendering.

- ***SceneNode* class**:

  The *SceneNode* class contains an array of pointers to its *children*, a pointer to the *parent* node, and a pointer to a ***ShaderProgram*** that holds pointers to the fragment and vertex shaders (figure 6b).

  Each node provides access to these attributes, and if a **child** node does not have an associated ***ShaderProgram***, it will **iteratively traverse the hierarchy upwards until it finds a parent with it**, which will then be assigned to the child.

  The same logic applies to the transformation components within the ***Transform*** structure (scale, rotation, and position) and the matrix composed of these. The transformations **applied to parent nodes are successively added to those of the child nodes**, iterating through the hierarchy.

  For the **draw** and **update** functions, all the node's **children are iterated over**, and these functions are called on them.

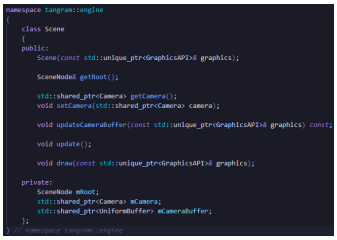
- **Concrete Node classes**:

  Depending on the type of lighting model and the characteristics of each object/model ([10], [11], [12], [13], [14]), specific nodes are required. These are implemented as **specialized classes that extend the *SceneNode*** class (figure 6c).

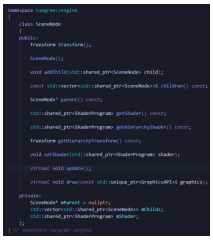
  On the **draw** function of the nodes, **uniforms** are set, and **textures** are bound according to the shaders associated with each node.

  It's also important to refer that the **directional light** passed to the nodes is a pointer to a struct that contains the direction, color, ambient, diffuse, specular components, and shininess.

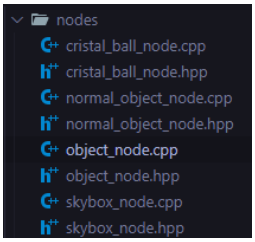
  - **CristalBallNode:** This node is used for the crystal ball object with **cel shading, silhouette, and rim light, without textures**. Its attributes include a directional light, the model, and the view position, as well as functions to set the rim, silhouette, and object color. As this node is designed for cel and silhouette shaders, the draw function sets the *front-face culling* for the silhouette (explained in detail in section 2.2.2).
  - **NormalObjectNode:** This node is for objects that use **Blinn-Phong shading with normal mapping and have applied textures**. Its attributes include the model, a directional light, textures, and the *Material* struct, which holds parameters for ambient, diffuse, specular, and emissive light, and shininess.
  - **ObjectNode:** This node is for objects that only use an **albedo texture with Blinn-Phong** shading. Its attributes include the directional light, model, and material with its components.
  - **SkyboxNode:** This node holds as attributes an array with the six images for the faces of the **skybox cube**, as well as the *cubemap* texture composed from them. It also includes the camera, an IBO (Index Buffer Object) for drawing the faces, and a VAO (Vertex Array Object) that includes the VBO (Vertex Buffer Object), layout, and IBO. On the **draw** function, it also sets *front-face culling*, depth, and binds the IBO and VAO (explained in detail in section 2.5).



(a) *Scene* class



(b) *SceneNode* class



(c) Types of nodes

Figure 6: Code of scene graph

2.2 Illumination models

2.2.1 Non physically-based "photorealistic" lighting model: Blinn-Phong

To implement the **Blinn-Phong illumination model**, I created **two variants**, resulting in four shaders: two pairs of vertex and fragment shaders. One pair handles Blinn-Phong with only albedo textures, while the other pair incorporates normal mapping as well as metallic and emission textures.

Regarding the Blinn-Phong implementation, both vertex shaders follow a **similar logic**. The key difference in the version with normal mapping lies in how the normal is obtained, as will be explained in section 2.4.

- **Vertex Shader:**

  The vertex shader requires **position**, **normal**, and **texture coordinates**, all sourced from the model's mesh through layouts/attributes, passed from a VBO via a VAO. It also uses the **camera's UBO**, which provides the view and projection matrices, as well as a **model matrix** passed as a uniform. Using these inputs, the following calculations are performed:

  1. **Position in Clip Space:** To compute the final vertex position in clip space, I calculated the multiplication of projection, view, model matrices, and the position vector from the attribute, and assigned to `gl_Position`.
  2. **Texture Coordinates:** The texture coordinates from the attribute are set into an output variable, enabling their use in the fragment shader.
  3. **Transforming Normals:** To ensure normals are transformed correctly without distortion, I calculated the model's transpose-inverse matrix, multiplied it by the normal attribute, and then set it into an output variable to be used in the fragment shader.
  4. **View Direction:** In world space, compute the vertex position by multiplying the model matrix uniform with the position attribute. Then, I obtained the camera's position in world space using the inverse of the view matrix. Finally, to calculate the view direction from the vertex to the camera position, I subtract the world position vector from the view inverse matrix.

- **Fragment Shader:**

  For the fragment shader, there are three input variables: **texture coordinates, normal, and view direction** from the vertex shader, an output variable `FragColor`, several **uniforms**, including the **texture**, **light components** (ambient, diffuse, specular, shininess), **light direction and color**, and a **material** struct containing the same properties as the light components, as already referred in section 2.1.

  For the shader with normal mapping, **additional uniforms** are included (detailed in section 2.4), as well as metallic and emission map textures (detailed in the topic "*Extras*" below). While the logic remains the same, these maps introduce additional visual effects.

  The implementation was based in the one present in the *"Lighting"* tutorial of *LearnOpenGL* [15], and also in the formula provided in the theoretical class, modified to include the half-vector (as explained in section 1.2.1):

$$I = M_e + M_a I_a K_a + \sum I_p F_{att} [M_d K_d (N.L) + M_s K_s (N.H)^n]$$

  So the following steps were performed:

  1. **Normalization:** Firstly, I normalized the normal vector (N), view direction (V), and the negative of light direction, because the light direction points from the fragment to the light source and its needed the opposite, (L).
  2. **Half-Vector Calculation:** To compute the half-vector, was used the normalized light and view directions: $H = normalize(L + V)$
  3. **Light and View Angles:** The calculation of the dot products for the light angle ($N.L$) and view angle ($N.H$), were performed and bounded to a maximum of 0 ensuring values are non-negative.
  4. **Attenuation Factor:** To ensure that two parallel surfaces with identical characteristics, but at significantly different distances from the light source, will appear differently, an attenuation factor was applied, in order to attenuate the light according to the distance, based on the formula of the theoretical classes:

$$F_{att} = \frac{1.0}{K_c + K_l d + K_q d^2}$$

     where K_c, K_l and K_q are, respectively, the constant linear and quadratic attenuation factors, and d is the distance from the light source to the object. Additionally, a minimum function is applied between 1 and the factor, as it represents the intensity of light that reaches the object after being attenuated by the distance, so the value 1 indicates the original light intensity, with no attenuation.
  5. **Ambient Light:** To calculate the ambient light, that simulates the indirect lighting, I multiplied the material's ambient color with the light color and the component of the ambient light ($M_a I_a K_a$).

6. **Diffuse Light:** As the diffuse light scattered equally in all directions, I multiplied the light color with the attenuation factor, the material's diffuse color, the component of the diffuse light, and the light angle ($I_p F_{att} M_d K_d (N.L)$).
7. **Specular Light:** First, the specular reflection is calculated through the view angle raised to the shininess of the material, multiplied by the light intensity, and if the light angle is 0, the specular reflection is 0. Then, this component is multiplied by the component of the specular light, the attenuation factor, the light color, and material reflectance ($I_p F_{att} M_s K_s (N.H)^n$).
8. **Final Color:** The final color of the fragment is determined by combining the total light contributions ($ambient + diffuse + specular$), and then multiplied by the texture color sampled using texture coordinates. This includes the RGB lighting and the texture with an alpha value of 1.

- **Extras:**

  As an **extra option not included in the challenges** taken, I decided to use **metallic and emission map textures** as well. These are applied to the Blinn-Phong shader that uses the normal mapping technique, having two extra sample uniforms for them. The logic is almost the same as what was already explained, with differences only in:

  – The coordinates of the metallic texture are obtained using the `texture` function, getting a factor.
  – For the diffuse light, first was performed a mix, which is a linear interpolation, between the material's diffuse color and a `vec3(1.0)`, using the metallic factor obtained, and this result is the M_d.
  – For the specular light, a mix was performed between the material's specular component and the same component multiplied by the metallic factor. The metallic factor was again used as the mixing factor, resulting in the M_s.
  – The material also has an additional component, the emission intensity. If there is an emission map, the M_e will be calculated by using it and the material's emission intensity. Then, the value is added to the final fragment color component ($emissive + ambient + diffuse + specular$).

Figure 7: Scene objects illuminated by Blinn-Phong

2.2.2 Non-photorealistic lighting/shading model with silhouette: Cel Shading with Rim Light

To implement the **Cel shading with silhouette and rim light**, I based my implementation in some tutorials ([16], [17]), and I created four shaders: two pairs of vertex and fragment shaders, one for the cel and the other for the silhouette and rim light.

- **Cel shading:**

  – **Vertex Shader:**

  The vertex shader requires **position**, **normal**, attributes sourced from the model's mesh through a VAO, as well as the **camera's UBO** and model matrix uniform. Using these inputs, the following calculations were performed:

  1. **Transforming Normals to World Space:** To ensure normals are transformed correctly without distortion, I calculated the model's transpose-inverse matrix, multiplied it by the normal attribute, and then normalized it, and set it into an output variable.
  2. **World Space Position:** The vertex position is then transformed to world space by multiplying it with the model matrix and outputted in a variable.
  3. **Position in Clip Space:** To compute the final vertex position in clip space, I multiplicated the projection, view, and the position vector of the last step, and assigned them to `gl_Position`, determinating the vertex's position on the screen.

  – **Fragment Shader:**

  For the fragment shader, there are uniforms for **object and light color** as well as **light direction and view position**, with:

1. **Normalization:** Firstly, I normalized the normal attribute and the negative light vector, this last one to get the light direction toward the fragment.
2. **Diffuse Lighting Calculation:** Then, I calculated the dot product between both normalized vectors, representing how aligned the surface is with the light source. For the value was applied a max function between it and 0 for not being negative, which will imply no contribution from the back side.
3. **Cel Effect:** To get the characteristic cel shading effect, I made a block of *if-else* cases with defined thresholds for the intensity calculated in the previous step, turning the continuous value into a discrete one, categorizing the intensity into distinct bands of solid color, creating the "stepped" shading that defines cel shading, rather than smooth gradients.
4. **Base Color Calculation:** Finally, I multiplied the intensity value, now classified into one of the defined steps, with the object color and the light color. This resulting color is assigned to the `FragColor` variable, which is passed to the framebuffer for rendering the pixel.

- **Shilhoute and rim light:**

  Since the object with cel shading needs to have a silhouette, I added a **child node with the following shaders**.

  In this way, I'd draw **two meshes, the main one and the silhouette**. The second one is identical to the original but has its faces inflated, simulating the silhouette as it is slightly larger than the original. In the node's draw function (*CristalBallNode*, as referred in section 2.1), the *front-face culling* is set in order to draw the faces of the child mesh in reverse, once this node only has one child which is the silhouette, so this gives the appearance that is always behind the original mesh.

  – **Vertex Shader:**

  This vertex shader is quite similar to the one used for cel-shading, including the uniforms and attributes. The **normals** are also transformed in the same way to **world space**. Then, I performed a **silhouette position expansion**, defining a **value** for how much the vertex position should be expanded along the normal direction to create the silhouette effect. Each vertex is moved along the normal direction, visually enlarging the object to highlight the silhouette, achieved by multiplying the normal by the scale factor and adding it to the position. Finally, the **position in clip space** is calculated in exactly the same way.

  – **Fragment Shader:**

  For the fragment shader, there are several **uniforms** as the **object, light, rim light, and silhouette color**, as well as the **view position and light direction**. So, the following steps were performed:

  1. **Normalization:** I normalized the normal attribute, the negative light vector to get the light direction toward the fragment, and the subtraction of the view position with the `FragPos` attribute to get the view direction.
  2. **Rim Light Effect:** To achieve this effect, I first calculated the dot product of the normalized normal vector and the view direction to determine how much the surface is facing toward the camera. I then applied a max function to ensure that negative values are clamped to 0, meaning that only the parts of the surface facing the camera contribute to the rim lighting. Afterward, I made 1.0 minus this value to determine the intensity of the rim light, ensuring that the lighting increases at the edges (where the surface is less directly facing the camera). The intensity of the rim lighting is then raised to a certain power to enhance the effect near the edges.
  3. **Silhouette Color Calculation:** The final base color is calculated by combining the silhouette color with the rim lighting effect, giving the object both, the silhouette color and the rim light highlight at the edges. Finally, this value is passed to the `FragColor` variable.

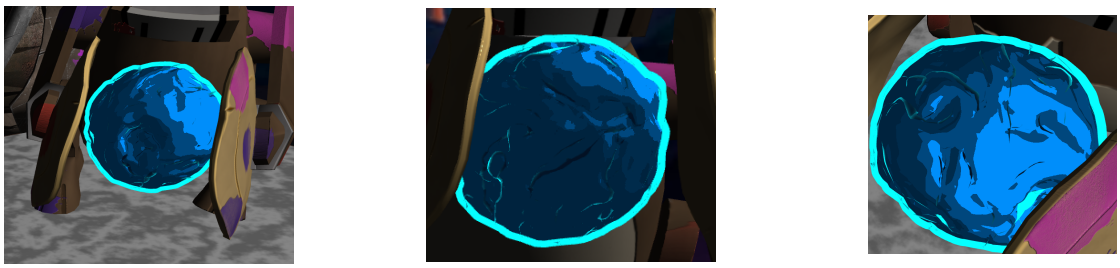


Figure 8: Scene object with Cel Shading, Silhouette, and Rim Light from different points of view

2.3 Realistic or stylized solid material using procedural noise: Wood, Bricks, Marble

To achieve realistic and stylized textures with procedural noise, I decided to use ***Perlin*** noise, as it is one of the most commonly used functions for this type of procedural textures.

2.3.1 Perlin Noise Function

To implement the noise function, I based it in the implementations from some papers ([18], [19]), videos ([20]), sites ([21], [22]) and repositories ([23], [24], [25], [26]), refining it with the help of *ChatGPT* [27]. This resulted in **two variants** of the *Perlin* function, one was used to generate the noise for the **marble** and **bricks**, and the other for generating **wood**, as the small differences in the function implementations make the noise more suitable for generating specific types of textures.

- **Auxiliary functions:**
  - ***randomGradient*:** This function generates a pseudo-random gradient vector based on two integer inputs, using a series of bit manipulations to produce a pseudo-random result, which is then used to create a vector representing the gradient, that is normalized, and then converted to a vector with sine and cosine components. This is useful as the **randomness in the directions of the gradients helps create natural patterns in the noise, avoiding obvious repetitions or regularities**.
  - ***dotGradient*:** The dot product function determines the noise value at that point, contributing to the **smoothness of the pattern** by calculating how the value of the noise changes depending on the gradient direction (from *randomGradient*) and the distance from the grid point.
  - ***cubicInterpolation*:** This function performs cubic interpolation between two values based on a weight, that represents how far between these two values the result should be. This interpolation produces **smoother transitions than linear interpolation**, making the resulting noise more fluid and less abrupt.
  - ***fade*:** This function applies a fade curve to a value, smoothing out the noise function. It uses a 5th-degree polynomial that allows the **interpolation** to ease in and out **smoothly**, reducing sharp transitions.
  - ***hash*:** This function generates a hash value from the input using bitwise operations and prime numbers, ensuring that the noise **pattern is repeatable for the same input coordinates without creating repetitive patterns**.
  - ***grad*:** The function computes the gradient vector based on the hash value and the input coordinates. It determines how much the noise value changes depending on the direction and distance from the grid point, contributing to the **diversity and randomness of the noise** pattern.

  It is important to note that the *randomGradient*, *hash* and *grad* functions were the parts I most relied on *ChatGPT* for guidance, as these were crucial in ensuring the natural random patterns.

- ***NoiseAttributes* struct:**

  To simplify the configuration of parameters such as **frequency, amplitude, octaves, and grid size**, I created a *NoiseAttribute* struct to encapsulate these values.
  - **Frequency:** Determines **how often** the noise **pattern repeats**. A higher frequency means more details in the noise, while a lower results in larger, smoother patterns.
  - **Amplitude:** Controls the **intensity or strength** of the noise values. A higher amplitude creates more variation in the noise, while a lower results in smoother transitions.
  - **Octaves:** Defines the **number of layers of noise** added together, with more octaves creating more complex and detailed noise, while fewer octaves create simpler, smoother patterns.
  - **Grid Size:** Specifies the **resolution** of the grid for generating noise, with smaller grid sizes leading to finer details, while larger grid sizes make the pattern coarser.

- ***MultiPerlin* Function:**

  This function generates a multi-octave *Perlin* noise, as it **accumulates noise values at different frequencies and amplitudes**, based on the parameters of the *NoiseAttributes*. This uses an internal ***Perlin*** noise function, that calculates the noise value at each point with the four neighboring grid points, applying **cubic interpolation and dot gradient**, and returning a smooth noise value.

  However, this kind of variant is often used for generating textures like **terrain or more intricate patterns**, as it incorporates **multiple octaves, where it adjusts both the frequency and amplitude for each one**, allowing more complex layered noise patterns.

- ***Perlin2D* Function:**

  This function is another variation of ***Perlin*** noise that uses a **similar multi-octave approach** but with **more detailed calculations**. It uses a 2D grid and performs a **linear interpolation** in both the x and y directions. The final noise value is a combination of multiple octaves with varying frequencies and amplitudes. The *fade* function is used to smooth the interpolation between the grid points, and the *grad* function is used to calculate the gradient values, allowing a more **fine-tuned control** over the resulting noise.

  This thinner noise is caused by computing the noise with octaves, that are applied sequentially using linear interpolation, which differs from the way noise layers are combined in *multiPerlin* function.

2.3.2 Texture Generation

To generate the textures, I created three functions one for each texture type. All these share a common structure, they iterate across the pixels of the ***Image***, a custom struct I developed, that includes the **width, height, number of channels**, a pointer to the **pixel** data, and functions to load an image and save it as a **JPG**, both implemented with the help of the ***stb_img_write*** library [28].

Each texture function uses a ***Perlin*** noise function to generate a **grayscale pixel array** for the image, **applies an RGB** color mapping, and then **saves** the resulting image. These functions are called in the main and the generated images are saved into the folder `./release/source/assets/procedural` when the application runs.

- **Wood:**

  For the wood texture, I used the ***Perlin2D*** function, as it provides the fine details I needed. The parameters that best suited my requirements were **frequency = 2.8, amplitude = 0.5, octaves = 4, and gridSize = 1**, all passed through the *NoiseAttributes* structure. Each pixel's position was normalized using the dimensions of the image. These normalized coordinates were then scaled by **xScale = 8.5** and **yScale = 1.5** to adjust the noise's frequency. A **grain** effect was created using the noise function with these scaled coordinates, and to simulate **wood-like rings**, a **sine** wave pattern was combined with the grain and adjusted with **frequencyPattern = 17**, resulting in a repetitive, wavy effect that mimics the natural grain of wood. Finally, **color mapping** was applied to the pixels to generate the **wood** texture.

  The result was a **fine wood texture with subtle knots, in light brown tones with darker grains** as we can see in figure 9a.

- **Marble:**

  For the marble texture, I used the ***MultiPerlin*** function with the *NoiseAttributes* to generate a more granular and detailed noise, clamped to the range $[-1.0, 1.0]$ to ensure consistency and smooth transitions. A **cosine** function was applied to create a wavy pattern that simulates the veins found in marble, as described in the theoretical class' slides, mimicating the ones of the figure 4c.

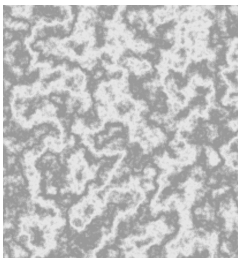
  I adjusted the **wave frequency** to **0.022** to control the density of the wave pattern. For the **cosine pattern**, combined with *Perlin* noise, a multiplier of **9.0** was used to introduce natural randomness to the veins. Finally, **color mapping** was applied to the pixels to produce a **marble**-like texture. So the result was a **slightly rough marble texture, with pronounced white and gray tones and noticeable patches**, as show in figure 9b.

- **Bricks:**

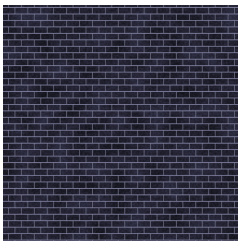
  To generate the brick textures, I defined parameters for the **brick width, height, mortar thickness, and mortar color**. I iterated through the pixels of the image, applying the ***MultiPerlin*** function with the specified *NoiseAttributes* for the bricks, as **frequency = 10.0f, amplitude = 0.8f, octaves = 5, and gridSize = 70**. The noise values were clamped to maintain consistency, and the background color was mapped accordingly.

  Afterward, I iterated again to draw both the vertical and horizontal mortar lines, also using the ***MultiPerlin*** function with the mortar attributes: **frequency = 20.0f, amplitude = 0.6f, octaves = 6, and gridSize = 80**, and applied the appropriate color mapping. Then the result was **rough-looking bluish-gray bricks with subtle tonal variations, complemented by a mortar with light gray tones and textured details**, as we can see in figure 9c.

(a) Procedural generated wood



(b) Procedural generated marble



(c) Procedural generated bricks

(d) Combination of all the procedural generated textures

Figure 9: Generated procedural textures with *Perlin* noise and its applications to the scene objects

2.4 Shader-based special effects: Normal Mapping

To implement **Normal Mapping**, which is a shader-based special effect, I need to create both a **fragment and a vertex shader**. These shaders were used in the *NormalObjectNode*, as discussed in the section 2.1. Since all objects using normal mapping are also rendered with the *Blinn-Phong* illumination model, these shaders include the *Blinn-Phong* implementation, as detailed in section 2.2.1.

Using this technique, which I based on the implementation presented in the *LearnOpenGL* tutorial [29], I was able to achieve **highly detailed objects** in the scene. All objects, except those with procedural textures or cel shading, are rendered with normal mapping.

- **Vertex Shader:**

  The vertex shader is similar to the one used in the *Blinn-Phong* shading model; however, it **differs in the way that it handles the normal vector**. In addition to the usual **attributes**, it includes the **tangent**, which is extracted from the model's *.obj* file and passed through a VAO to the shaders. The tangent is essential as we are working in tangent space, as it aids in texture mapping and computing the tangent-space normal vector. The **output** variables of the shader include the **texture coordinates, the view direction, and the TBN (Tangent, Bitangent, Normal) matrix**, which combines the tangent, the bitangent (calculated within the shader), and the normal matrix. The camera and model matrices are passed as **uniforms** to the shader.

  1. Firstly, I calculated the `gl_Position` and set the texture coordinates as an output variable in the same way as explained in the *Blinn-Phong* vertex shader.
  2. **Calculate TBN:** This matrix is used to **transform normals from tangent space to world space**. For the **tangent** vector T, I transform it from object space to world space by multiplying it by the model matrix, and then normalizing it. For the **normal** vector N, I transform it from object space to world space and normalize it as well, using the same model matrix multiplication. Finally, the **bitangent** vector B is calculated by taking the cross product of the normalized tangent and normal vectors, which results in a vector perpendicular to both, which is then transformed into a matrix. The subsequent steps are identical to step 4 in the *Blinn-Phong* shader.

- **Fragment Shader:**

  In the fragment shader, the **TBN** matrix is added as an input variable, along with a `sampler2D` uniform for the normal map (explained in section 1.4).

  The logic remains the same, with the only change being that the **normal is retrieved from the normal map texture as an RGB vector**. Since normal maps store normals in the range $[0, 1]$, they are clamped to the range $[-1, 1]$, and then transformed from tangent space to world space by multiplying the **TBN** matrix (passed from the vertex shader) with the clamped normal. The subsequent logic follows the explanation in the *Blinn-Phong* in section 2.2.1, using this transformed normal.

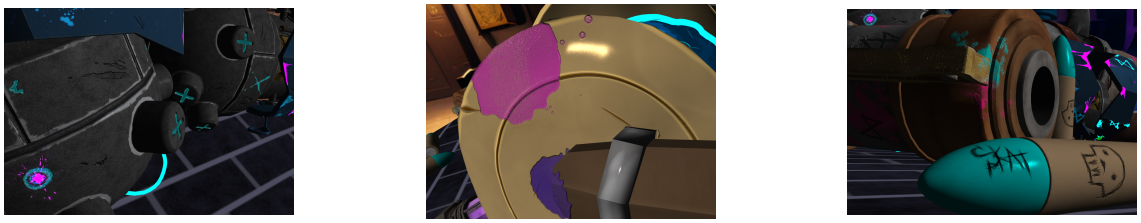


Figure 10: Normal Mapping technique applied to the scene objects

2.5 Extras

As an **additional challenge, not included in the graded requirements**, I implemented a **permanent rotation animation in the crystal ball** of the monkey, an also a **skybox** based on some tutorials ([30], [31]). Briefly, I created a pair of **shaders**:

- The **vertex** shader computes the position in clip space, assigns the original vertex coordinates to the `TexCoords` variable for later use in the fragment shader, and also calculates the final vertex position for rasterization.
- The **fragment** shader just samples the color from a *cubemap* texture using the texture coordinates, and sets this color as the output in `FragColor`.

On the **engine** side, I implemented the code to **load** a *cubemap* texture, and on the ***draw*** function of the *SkyBoxNode*, I set up *front-face culling* and adjusted the *Depth* tests accordingly.

Figure 11: Scene objects with the Skybox

3 Post-Mortem

In this section, I critically evaluate the development process, addressing not only technical challenges but also broader aspects such as project management and workflow strategies, with the aim of providing actionable insights for future endeavors.

3.1 What went well?

I think that overall, **everything went well**, and the decisions I made were the right ones.

Firstly, creating a task **schedule** and sticking to it was incredibly useful, as it allowed me to have enough time to address any unforeseen issues and still meet the deadlines, being a great workflow decision. Even though I didn't start implementing the project on the exact dates planned in the calendar because I was busy with other projects, I started with a slight delay but ultimately finished ahead of the deadlines I had set for myself.

Starting with the lighting, then the skybox, and only afterward focusing on mapping and procedural textures was also helpful. I **began with what I considered simpler tasks**, such as those involving shader code, and left the procedural textures, which I found more challenging, for the end. This allowed me to speed through the easier parts first.

Additionally, focusing **first on implementing the techniques and only afterward assembling** the scene with all the models, proved to be a good approach, as once the theoretical part was functioning, it was just a matter of making everything look polished.

Another key factor was that, in previous *Tangram* projects, my colleague and I had implemented a **highly abstracted engine**. This contributed to making the current code much more organized and readable, which became increasingly important as the number of classes grew. It significantly helped me create a mental map of what was or wasn't necessary to include.

3.2 What did not go so well?

Honestly, I don't think **anything went wrong**, there was just an **initial delay** in the schedule, which I managed to recover from quickly. I just think that I **slightly underestimated the complexity of procedurally generated textures**, although I knew from the beginning that this would be the most challenging part, adjusting the parameters to achieve the desired result turned out to be more difficult than expected.

Additionally, working with textures and objects presented a few challenges along the way. One issue arose when applying procedural textures to objects I had modeled myself, as they lacked texture mapping. This caused problems with the **texture coordinates not mapping correctly**, resulting in deformation of the texture. After some research, I discovered *Blender*'s UV editor, which allowed me to manipulate the texture coordinates and ensure proper mapping. Another related issue occurred with the wood texture, where its low pixel resolution caused it to appear **pixelated** on the object. To resolve this, I increased the pixel count when generating the image and adjusted the grid for the noise attributes, which fixed the issue and improved the texture quality.

The last small issue I encountered involved the implementation of the **silhouette** for cel-shading. Initially, I attempted to create both the cel-shading and the silhouette in the same shader, not realizing that for the implementation to work, two meshes needed to be rendered with *front-face culling*. After researching the topic further, I identified the mistake and corrected it by adding a child node to the mesh with the cel shaders, which now contain the silhouette shaders.

3.3 Lessons learned

If I could **go back**, despite everything going well, I would keep most things the same. However, I would spend more time **exploring *Blender***, as well as delving deeper into the **noise** functions and their **workings**, so this could have saved me time when addressing the texture-related issues I encountered.

So with this I think that the most valuable **lesson** I learned, and something I strongly recommend to everyone, is: take your time to learn and understand things thoroughly, even if you're afraid, or if you think that it won't succeed, once you truly understand something, everything becomes much easier.

With all of this in mind, the **advices** I would give to others following a similar path is:

- Attend classes because having someone with experience in the field explaining concepts makes it much easier to understand.
- Watch videos and do research, as tutorials and papers are great resources for learning how to implement things properly.
- Start with plenty of time to spare, as you never know what might go wrong. Break the work into small tasks and begin with simpler challenges so that you can feel accomplished when achieving small goals, and the end won't seem so far away.
- Focus on functionality first and aesthetics later, so even if you run out of time, you'll have the technical solution ready without wasting effort on superficial details.

4 Bibliography

[1] Joey de Vries. Learnopengl: Advanced lighting. https://learnopengl.com/Advanced-Lighting/Advanced-Lighting, 2016. Accessed: 2024-12-21.

[2] The instruction limit: Blinn-phong. https://theinstructionlimit.com/tag/blinn-phong. Accessed: 2024-12-20.

[3] Phong illumination model and brdf. https://www.scratchapixel.com/lessons/3d-basic-rendering/phong-shader-BRDF/phong-illumination-models-brdf.html. Accessed: 2024-12-20.

[4] Roblox Developer Forum. How to create a cel/toon shading effect on models using blender [outdated]. https://devforum.roblox.com/t/how-to-create-a-celtoon-shading-effect-on-models-using-blender-outdated/361005. Accessed: 2024-12-26.

[5] Hali Savakis. My take on shaders: Cel shading. https://halisavakis.com/my-take-on-shaders-cel-shading/. Accessed: 2024-12-26.

[6] Ryan King. Procedural material: Red brick wall. https://ryanking.artstation.com/projects/GaB8Q4?album_id=2507498. Accessed: 2024-12-26.

[7] Red brick wall wallpaper. https://www.photowall.com/pt/red-brick-wall-papel-de-parede. Accessed: 2024-12-26.

[8] Ryan King. Procedural marble material (blender tutorial). https://ryanking.artstation.com/projects/48eZmq. Accessed: 2024-12-26.

[9] Babylon.js documentation: More materials. https://doc.babylonjs.com/features/featuresDeepDive/materials/using/moreMaterials. Accessed: 2024-12-26.

[10] Sk4t. Jinx's bullet. https://skfb.ly/owsoF. Accessed: 2024-12-06.

[11] Prabhath077. The rocket launcher of jinx "fishbones". https://skfb.ly/ovHqx. Accessed: 2024-12-06.

[12] Zerostyle. Jinx's bomb. https://skfb.ly/oyJNF. Accessed: 2024-12-06.

[13] AK_Krause. Monkey keychain from valorant arcane bundle. https://skfb.ly/orAYZ. Accessed: 2024-12-06.

[14] Nim. Jinx's sheriff. https://skfb.ly/oyPKD, note = Accessed: 2024-12-06.

[15] Joey de Vries. Learnopengl - basic lighting. https://learnopengl.com/Lighting/Basic-Lighting, 2017. Accessed: 2024-12-20.

[16] GetIntoGameDev. Opengl shader programming 04: Cel shading. https://youtu.be/F199UtOLWlY?si=gaKAts3RdQOMRapE, 2022. Accessed: 2024-12-26.

[17] OGLDEV. Toon shading rim lighting // opengl tutorial 34. https://youtu.be/h15kTY3aWaY?si=ZmUkGxTN4ZA68IQi, 2022. Accessed: 2024-12-26.

[18] Nathan Thiesen. Perlin noise function. https://www.cs.montana.edu/courses/spring2005/525/students/Thiesen1.pdf, 2005. Accessed: 2024-12-27.

[19] Reynald Arnerin. A journey in a procedural volume: Optimization and filtering of perlin noise. https://inria.hal.science/inria-00598443/PDF/rapport_Reynald_Arnerin.pdf, 2011. Accessed: 2024-12-27.

[20] Zipped. C++: Perlin noise tutorial. https://youtu.be/kCIaHqb60Cw?si=ha4CM0PEiD_7kFFN, 2023. Accessed: 2024-12-28.

[21] Flafla2. Understanding perlin noise. https://adrianb.io/2014/08/09/perlinnoise.html, 2014. Accessed: 2024-12-27.

[22] Perlin noise. https://www.arendpeter.com/Perlin_Noise.html, 2016. Accessed: 2024-12-27.

[23] daniilsjb. Github: Perlin noise. https://github.com/daniilsjb/perlin-noise, 2024. Accessed: 2024-12-28.

[24] Reputeless. siv::perlinnoise. https://github.com/Reputeless/PerlinNoise, 2020. Accessed: 2024-12-28.

[25] Stefan Gustavson. Perlin noise implementation in c. https://github.com/stegu/perlin-noise/blob/master/src/noise1234.c, 2005. Accessed: 2024-12-28.

[26] Sean Barrett. stb_perlin.h. https://github.com/nothings/stb/blob/master/stb_perlin.h, 2017. Accessed: 2024-12-28.

[27] OpenAI. Chatgpt. https://openai.com/chatgpt, 2024.

[28] Sean Barrett. stb_img_write library. https://github.com/nothings/stb. Accessed: 2024-12-28.

[29] Joey de Vries. Learnopengl - normal mapping. https://learnopengl.com/Advanced-Lighting/Normal-Mapping, 2017. Accessed: 2024-12-26.

[30] Joey de Vries. Learn opengl - cubemaps. https://learnopengl.com/Advanced-OpenGL/Cubemaps, 2017. Accessed: 2024-12-20.

[31] Victor Gordan. Opengl tutorial 19 - cubemaps skyboxes. https://youtu.be/8sVvxeKI9Pk?si=1Az5YhhcBHAYV9dx, 2021. Accessed: 2024-12-20.