Real Time Graphics Programming Project Report

Compagnoni Alessandro

UNIMI A.A. 2023/2024

UNIVERSITÀ
DEGLI STUDI
DI MILANO

Contents

1 Overview

This project implements a 3D environment using OpenGL, allowing users to navigate within 3 square rooms. Inside the rooms various 3D objects are placed, each rendered with a different material based on shaders that showcase various types of noise with different parameters, all freely adjustable by the user thanks to the dedicated UI.

2 Design Choices

2.1 Technologies

The following technologies were selected to ensure efficient rendering, cross-platform compatibility, and streamlined development for the project:

- **OpenGL Version 3.3+**:
  OpenGL's modern shader-based architecture was chosen to leverage direct control over vertex and fragment processing. By utilizing GLSL shaders, the pipeline enables complex procedural effects while maintaining high performance.
- **GLFW (Graphics Library Framework)**:
  GLFW provides robust window management and input handling, ensuring consistent behavior across operating systems, also its event-driven architecture simplifies interaction with user inputs.
- **GLM (OpenGL Mathematics)**:
  GLM mathematics library is optimized for graphics programming. It offers essential data types (e.g., vectors, matrices) and prebuilt functions for common transformations (e.g., translation, projection).
- **ImGui**:
  ImGui is a graphical user interface library for C++. It is simple, fast and portable and allows fast iteration and implementation of creation tools and visualization / debug tools.

2.2 Organization

The project is organized as follows:

- C++ file containing the code with the main function and all other functions called "Rooms.cpp", this file is located in the "src" directory.
- Directory called "shaders" containing all the fragment.glsl and vertex.glsl shader files for each of the materials used in the project.
- All the necessary dependencies and library located in the "include", "imgui" and "lib" directories.

2.3 Architecture

The scene is organized in 3 different square rooms, where each of the walls, floor and ceiling material's are based on a different shader: the walls' shader is made to make them look like wooden walls, while the ceiling and floor utilize a marble-like shader. The rooms are connected by 2 short corridors that utilize a dark-grey colored shader to appear neutral.

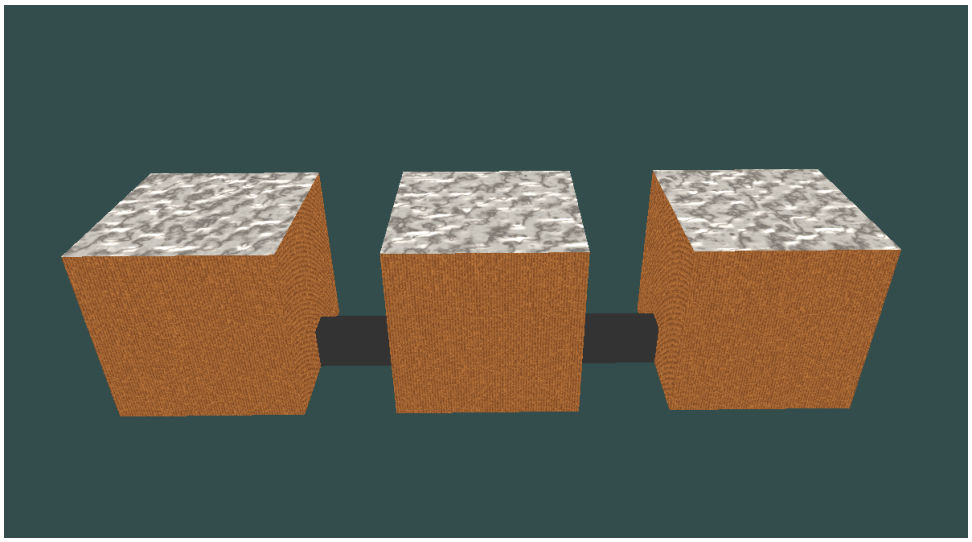


Figure 1: Structure and architecture of the 3D space as seen from the outside

Inside each of the first 2 rooms we can find 3 objects (a sphere, a pyramid and a cube), while in the last room we only find 2 objects of larger dimensions (a sphere and a cube).

In the first room we can find Perlin noise applied to all the objects: the sphere has a simple Perlin noise, the cube has Perlin noise with multiple octaves and the pyramid has Perlin Noise with turbulence.

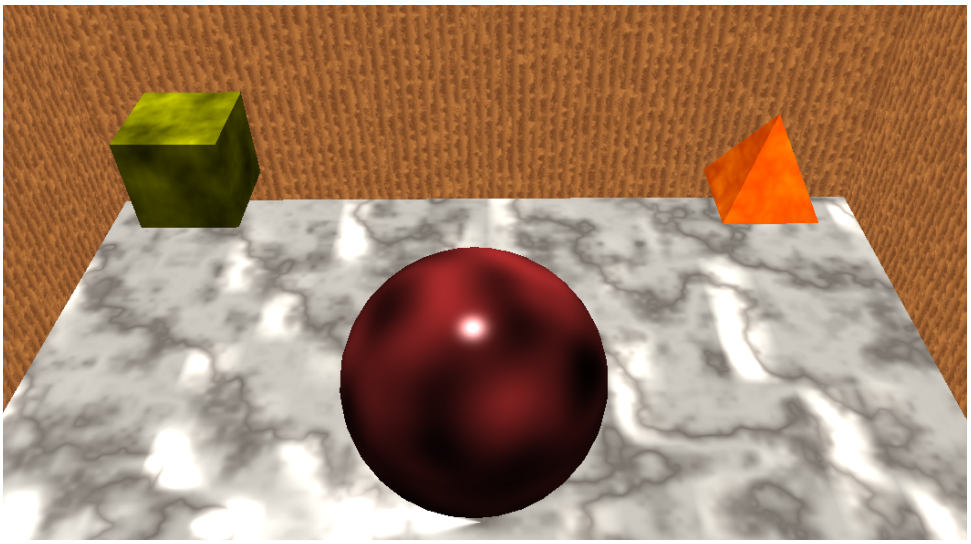


Figure 2: First room

In the second room we can find Simplex noise applied to the cube, Multifractal noise applied to the sphere and Cellular noise applied to the pyramid.

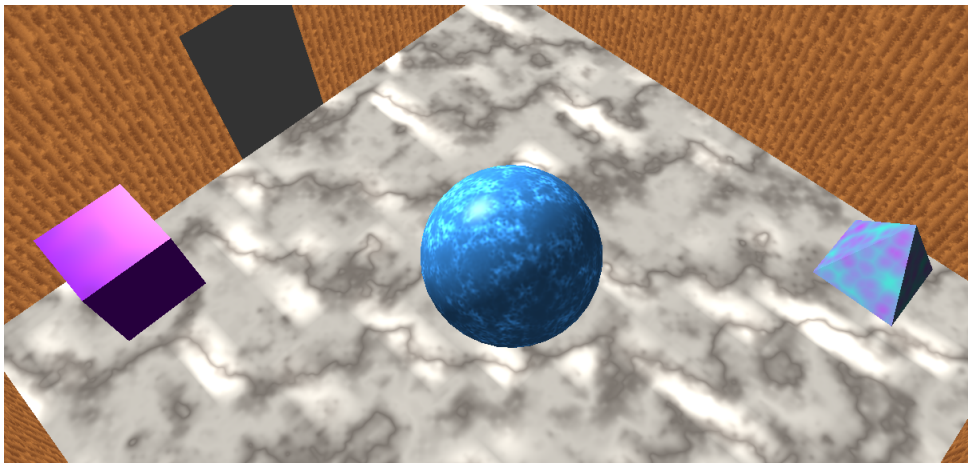


Figure 3: Second room

While in the first two rooms all types of noises where applied only to the diffusive color of the objects, in the third room we can find Perlin noise applied on the normal map in the case of the sphere, while for the cube the Perlin noise is applied to its transparency.

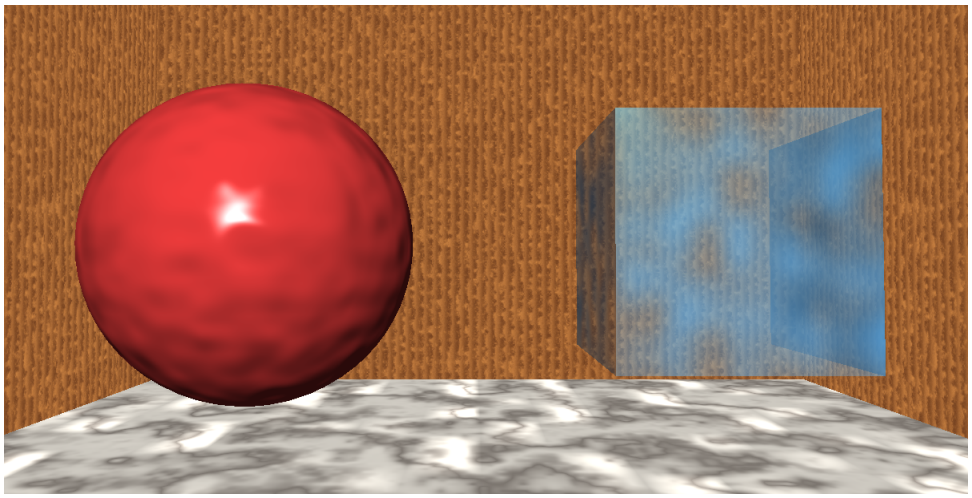


Figure 4: Third room

2.4 Controls

The user is able to move and look around in this space by using the WASD keys to move and the mouse to orient the camera.

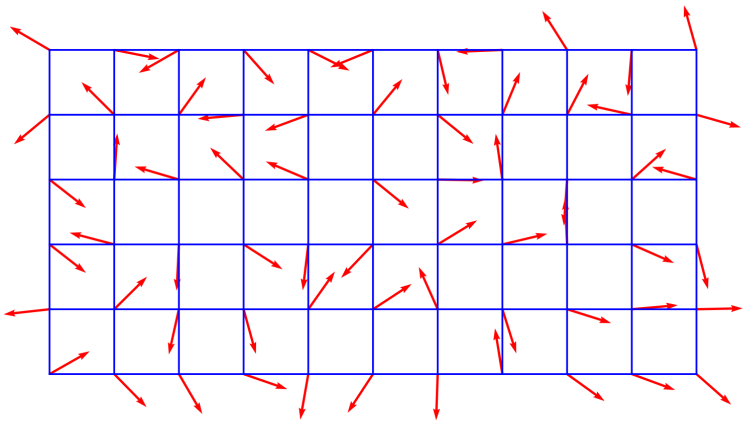
3 Algorithms and Techniques

This section will explain in detail all types of noise utilized and where the user can find them in the project.

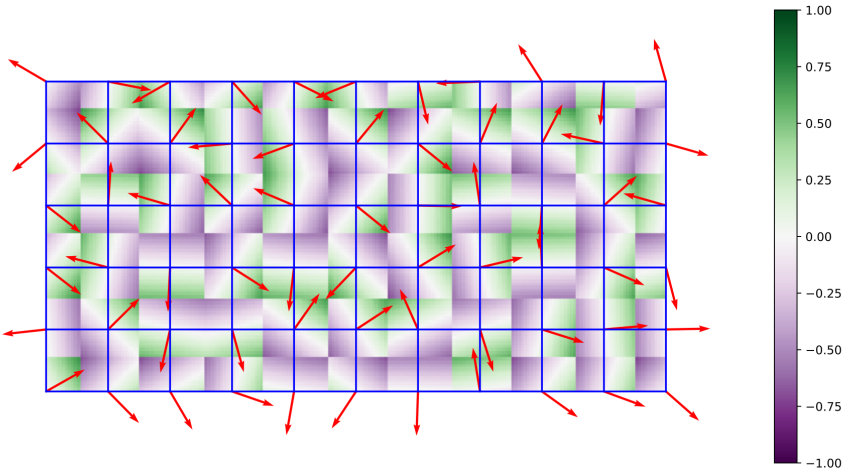
3.1 Noises

- **Perlin Noise**:
  Perlin noise is a gradient noise function that produces smooth, continuous random variations. It is widely used in computer graphics for procedural texture generation due to its natural appearance. The implementation typically involves three steps:

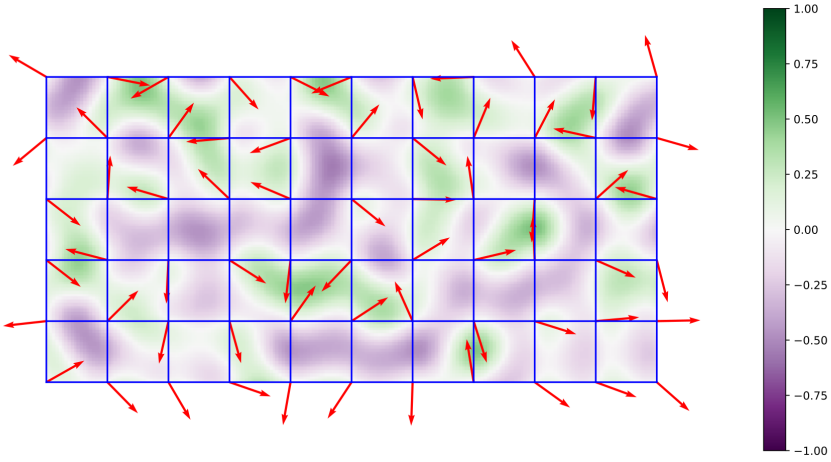
  1. **Grid definition**:
     Define an n-dimensional grid where each grid intersection has associated with it a fixed random n-dimensional unit-length gradient vector.



  2. **Dot Product**:
     For working out the value of any candidate point, first find the unique grid cell in which the point lies. Then, identify the 2n corners of that cell and their associated gradient vectors. Next, for each corner, calculate an offset vector. An offset vector is a displacement vector from that corner to the candidate point. For each corner, we take the dot product between its gradient vector and the offset vector to the candidate point. This dot product will be zero if the candidate point is exactly at the grid corner.



3. **Interpolation**:
   The final step is interpolation between the 2n dot products. Interpolation is performed using a function that has zero first derivative (and possibly also second derivative) at the 2n grid nodes. Therefore, at points close to the grid nodes, the output will approximate the dot product of the gradient vector of the node and the offset vector to the node. This means that the noise function will pass through 0 at every node, giving Perlin noise its characteristic look.



- **Perlin Noise (with Multiple Octaves)**:
  Perlin noise with multiple octaves is a technique that combines multiple Perlin noise functions to create a more complex and detailed noise pattern. It involves summing together multiple octaves of Perlin noise, each octave is a separate layer of noise with a different frequency and amplitude.

  Multiple octaves are combined by adding the noise values from each layer. Typically, the frequency is doubled, and the amplitude is halved for each successive octave. This creates a fractal-like pattern where larger features are overlaid with progressively finer details.

  Fractal Brownian Motion (fBm) is a common technique for combining octaves. It involves summing the noise values from each octave, weighted by their respective amplitudes. The result is a texture that has both large-scale and small-scale variations.

- **Perlin Noise (with Turbulence)**:
  Turbulence is a technique used in procedural texture generation to create chaotic, swirling patterns that resemble natural phenomena like smoke, fire, or flowing water. It is a variation of fractal Brownian motion (fBm) that uses the absolute value of noise to produce sharp, jagged features.

  Unlike standard fBm, which sums smooth noise values, turbulence takes the absolute value of each noise layer. This introduces sharp changes and more pronounced variations, which are characteristic of turbulent patterns.

  The final turbulence value is normalized by dividing by the maximum possible value, ensuring it remains within a usable range.

- **Simplex Noise**:
  Simplex noise is a type of gradient noise that improves upon classic Perlin noise by reducing directional artifacts and computational complexity. It is particularly useful for generating smooth, organic textures in graphics applications. The implementation typically involves five steps:

  1. **Coordinate Skewing**:
     Simplex noise uses a skewed grid to reduce the number of required gradient calculations.

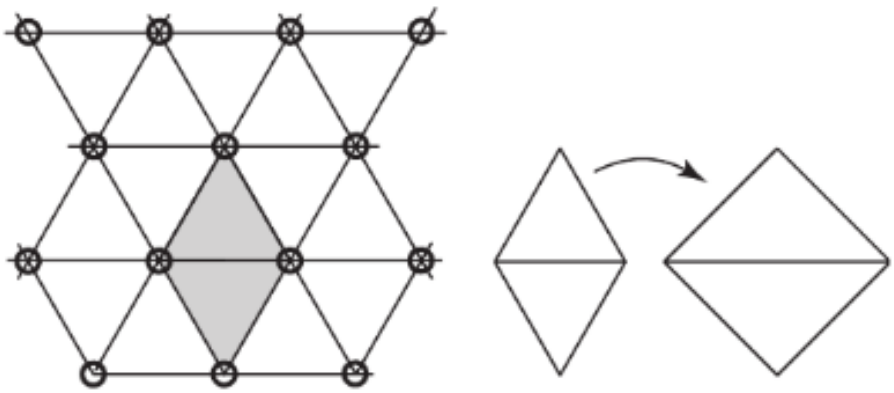


Figure 5: Skewed grid in 2D, made of triangles

  2. **Simplicial Subdivision**:
     The grid is divided into simplices (triangles in 2D, tetrahedra in 3D), which are the smallest possible convex shapes that can fill the space without gaps or overlaps.
  3. **Gradient Selection**:
     For each simplex, a random gradient vector is assigned to each of its vertices. These gradients are used to determine the direction of the noise variations.
  4. **Interpolation**:
     The noise values at the corners are interpolated using a smooth function to ensure continuous transitions across the grid.
  5. **Normalization**:
     The final noise value is normalized to ensure it remains within a usable range.

- **Multifractal Noise**:
  Multifractal noise is a complex noise function that combines multiple layers of noise with varying parameters to create highly detailed and intricate textures. It is particularly useful for simulating natural phenomena that exhibit self-similar patterns, such as landscapes, clouds, and other organic structures. Multifractal noise is characterized by its self-similar nature, where patterns repeat at different scales. This is achieved by layering multiple octaves of noise, each with a different frequency and amplitude.

  Unlike standard fBm, where amplitude and frequency are fixed for each octave, multifractal noise adjusts these parameters dynamically based on the current noise value. This creates a more complex and varied texture.

  The contribution of each octave is weighted by the current noise value, which influences the amplitude of subsequent octaves. This weighting creates a feedback loop that enhances the self-similar nature of the texture.

  Parameters: Aside for the number of octaves, there are 2 more parameters that influence the Multifractal Noise:

  - **Lacunarity**:
    Lacunarity controls the frequency scaling between octaves. Higher lacunarity results in more frequent changes in detail.
  - **Hurst Exponent (H)**:
    The Hurst Exponent determines the fractal dimension of the noise. Higher values of H result in smoother, more persistent patterns.

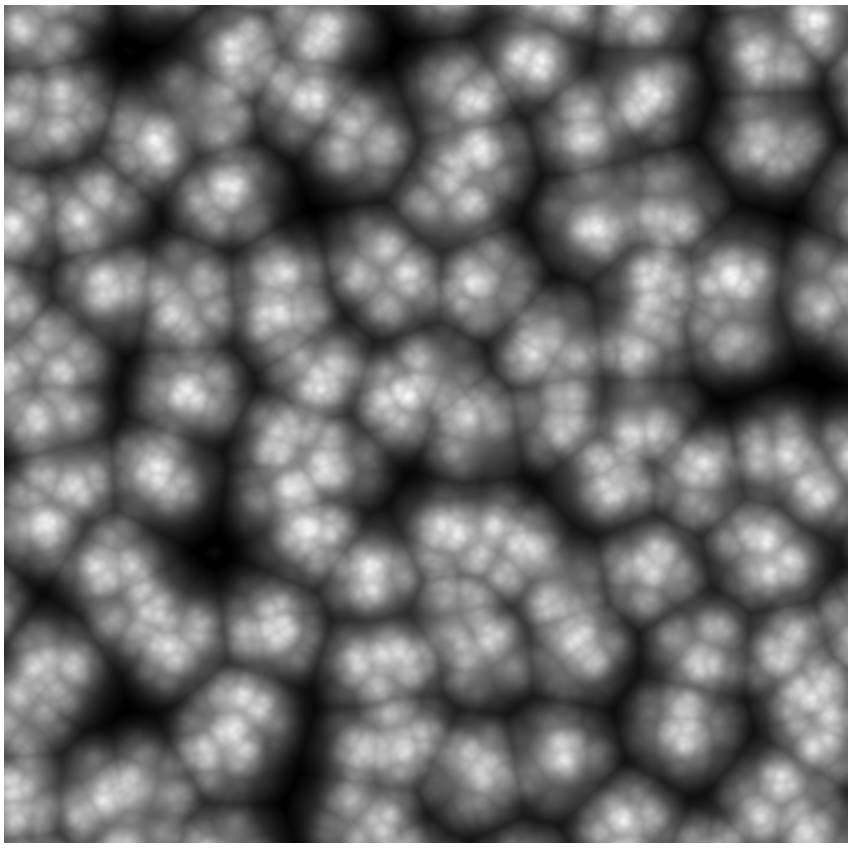


Figure 6: Example of multifractal noise

- **Cellular Noise**:
  Cellular noise, also known as Worley noise, is a type of procedural texture that generates patterns based on the distance to the nearest feature point in a grid. It is often used to create textures that resemble natural cellular structures, such as stone, skin, or organic patterns. The implementation typically involves four steps:

  1. **Grid Definition**:
     The grid is defined as a regular grid of points in the space.
  2. **Feature Point Generation**:
     A set of feature points are randomly distributed throughout the grid. These points act as the centers of the cellular structure.
  3. **Distance Calculation**:
     For each point in the grid, the distance to the nearest feature point is calculated. The minimum distance determines the noise value at that point.
  4. **Normalization**:
     The final noise value is normalized to ensure it remains within a usable range.

  The resulting noise pattern consists of cells, where each cell is centered around a feature point. The edges of the cells are defined by the equidistant lines between neighboring feature points.

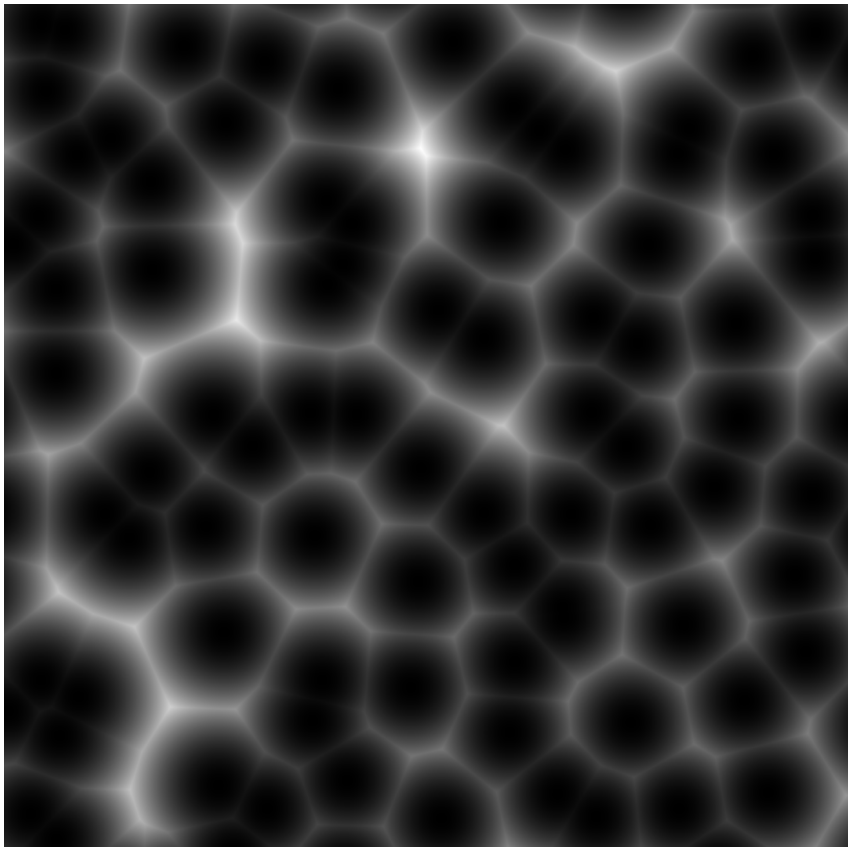


Figure 7: Example of cellular noise

4 Implementation Details