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.

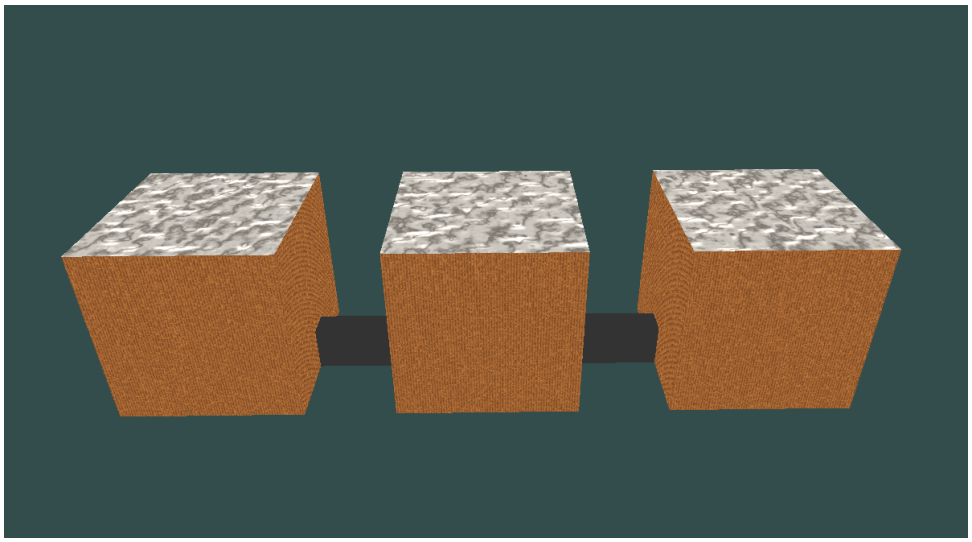


Figure 1: Structure and architecture of the 3D space as seen from the outside

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.

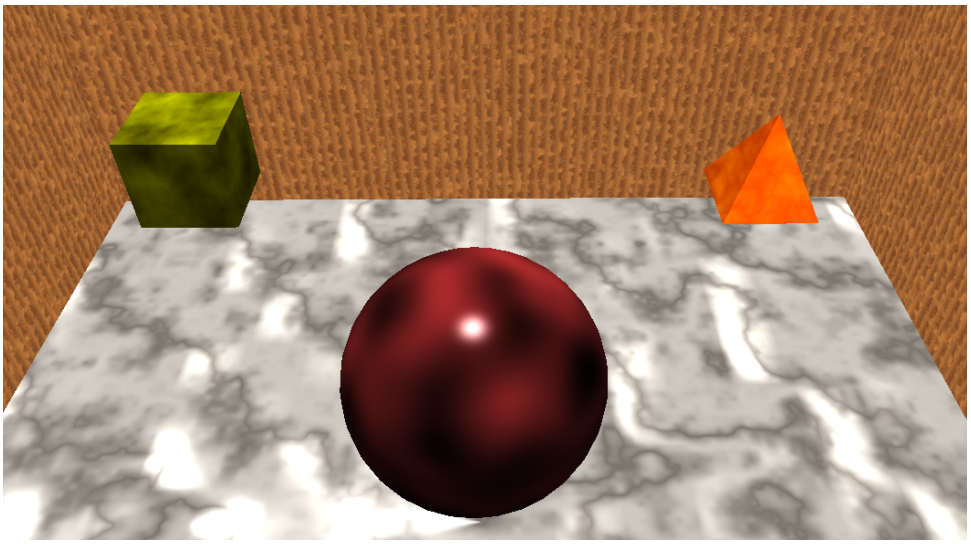


Figure 2: First room

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.

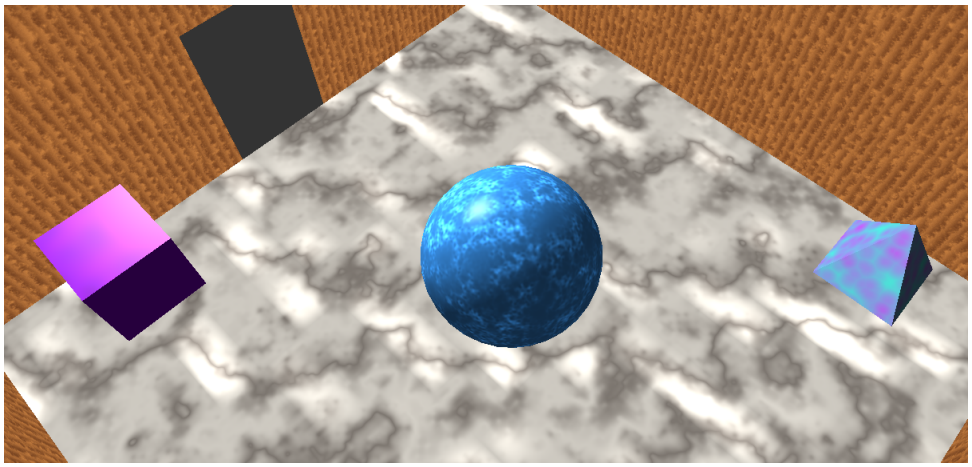


Figure 3: Second room

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.

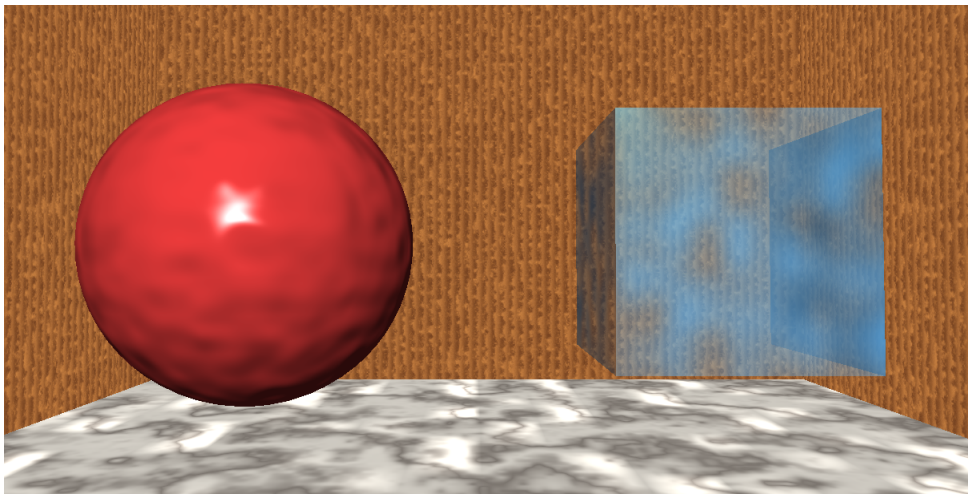


Figure 4: Third room

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.

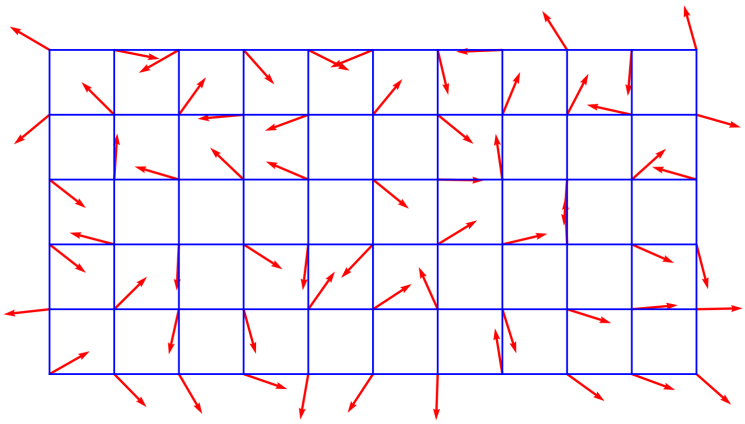
3 Algorithms and Techniques

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

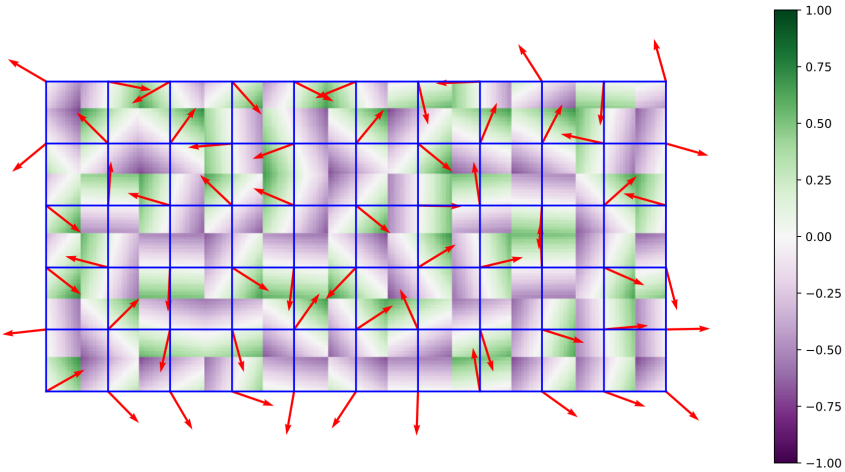
3.1 Noises

- **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:

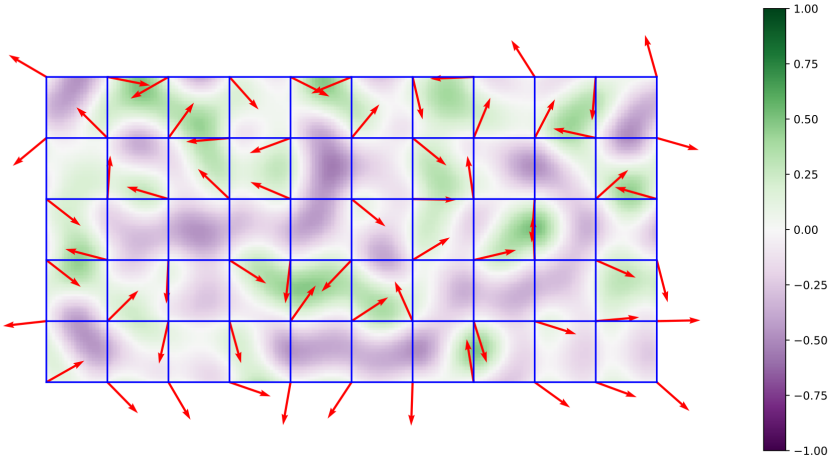
  1. **Grid definition**:
     Define an n-dimensional grid where each grid intersection has associated with it a fixed random n-dimensional unit-length gradient vector.



  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.

- **Perlin Noise on Normal Map**:
  This technique is often used to simulate surface details without the need for additional geometry, enhancing the realism of the rendered object. The implementation typically involves three steps:

  1. **Noise Generation**:
     Perlin noise is generated based on the fragment position. This noise is used to create small variations in the surface normals.
  2. **Normal Perturbation**:
     The generated noise is used to perturb the original normals. This involves adjusting the normal vectors based on the noise values, simulating a bumpy surface.
  3. **Lighting Calculations**:
     The perturbed normals are then used in the lighting calculations, affecting the diffuse and specular components of the lighting model.

- **Perlin Noise on Transparency**:
  This technique allows for the simulation of materials with varying opacity, such as frosted glass or organic surfaces. The implementation typically involves three steps:

  1. **Noise Generation**:
     Perlin noise is generated based on the fragment position. This noise value is used to determine the transparency level of each fragment.
  2. **Alpha Calculation**:
     The noise value is mapped to an alpha value, which controls the transparency of the fragment. This mapping is done using a linear interpolation between minimum and maximum alpha values.
  3. **Fragment Color Output**:
     The calculated alpha value is combined with the base color and lighting calculations to produce the final fragment color, including its transparency.

- **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.

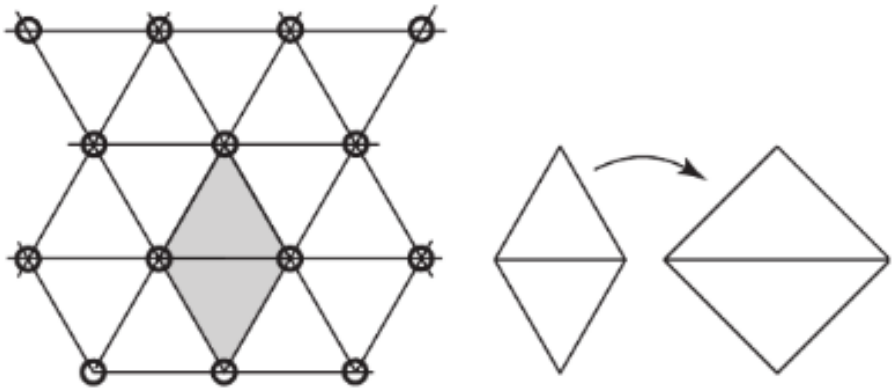


Figure 5: Skewed grid in 2D, made of triangles

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.

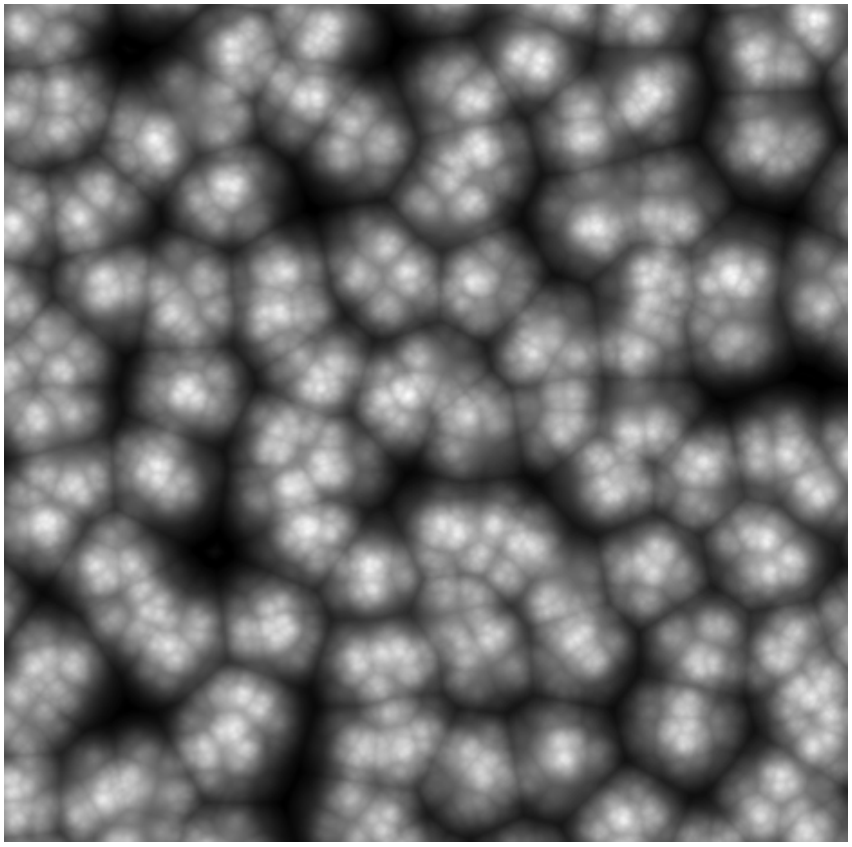


Figure 6: Example of multifractal noise

- **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.

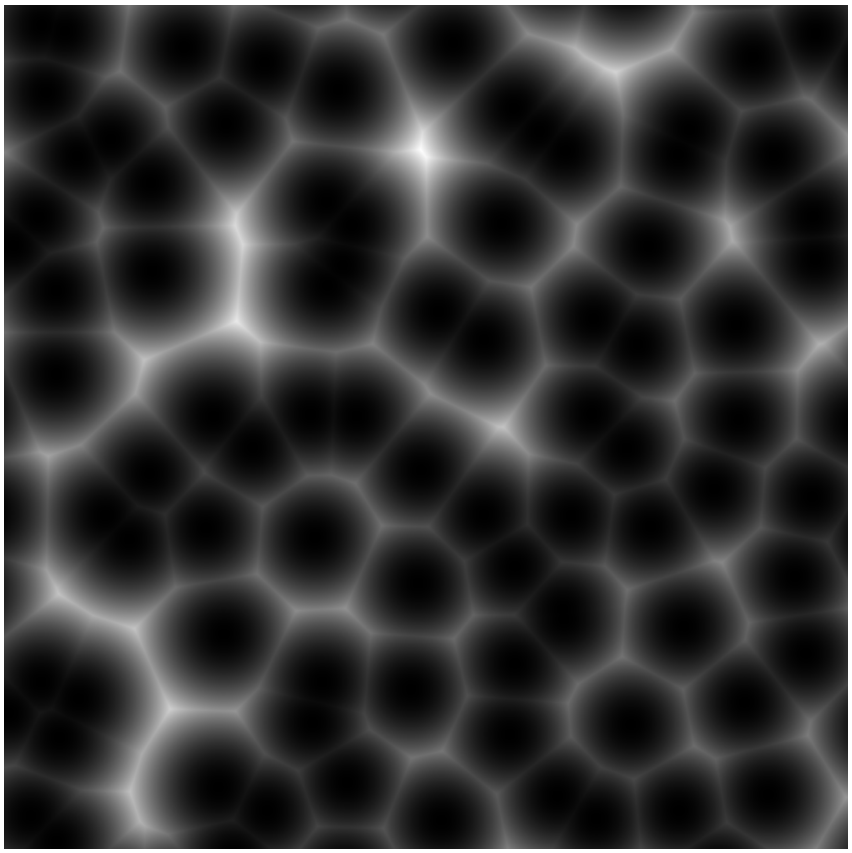


Figure 7: Example of cellular noise

4 Implementation Details

In this section are reported all the most relevant sections of the code, with particular focus on the noise generation and the shaders.

4.1 Noise Generation

1. **Perlin Noise and variations**:

   - **fragment_sphere1.glsl**:
     This shader is used to render the sphere in the first room, it uses simple Perlin noise to modify the diffusive color of the sphere.

```
vec3 mod289(vec3 x) { return x - floor(x * (1.0 / 289.0)) * 289.0; }
vec4 mod289(vec4 x) { return x - floor(x * (1.0 / 289.0)) * 289.0; }
vec4 permute(vec4 x) { return mod289(((x*34.0)+1.0)*x); }
vec4 taylorInvSqrt(vec4 r) { return 1.79284291400159 - 0.85373472095314 * r; }
vec3 fade(vec3 t) { return t*t*t*(t*(t*6.0-15.0)+10.0); }

float noise(vec3 P) {
    vec3 i0 = mod289(floor(P));
    vec3 i1 = mod289(i0 + vec3(1.0));
    vec3 f0 = fract(P);
    vec3 f1 = f0 - vec3(1.0);
    vec3 f = fade(f0);

    vec4 ix = vec4(i0.x, i1.x, i0.x, i1.x);
    vec4 iy = vec4(i0.yy, i1.yy);
    vec4 iz0 = i0.zzzz;
    vec4 iz1 = i1.zzzz;

    vec4 ixy = permute(permute(ix) + iy);
    vec4 ixy0 = permute(ixy + iz0);
    vec4 ixy1 = permute(ixy + iz1);

    vec4 gx0 = ixy0 * (1.0 / 7.0);
    vec4 gy0 = fract(floor(gx0) * (1.0 / 7.0)) - 0.5;
    gx0 = fract(gx0);
    vec4 gz0 = vec4(0.5) - abs(gx0) - abs(gy0);
    vec4 sz0 = step(gz0, vec4(0.0));
    gx0 -= sz0 * (step(0.0, gx0) - 0.5);
    gy0 -= sz0 * (step(0.0, gy0) - 0.5);

    vec4 gx1 = ixy1 * (1.0 / 7.0);
    vec4 gy1 = fract(floor(gx1) * (1.0 / 7.0)) - 0.5;
    gx1 = fract(gx1);
    vec4 gz1 = vec4(0.5) - abs(gx1) - abs(gy1);
    vec4 sz1 = step(gz1, vec4(0.0));
    gx1 -= sz1 * (step(0.0, gx1) - 0.5);
    gy1 -= sz1 * (step(0.0, gy1) - 0.5);
```

```
    vec3 g000 = vec3(gx0.x, gy0.x, gz0.x);
    vec3 g100 = vec3(gx0.y, gy0.y, gz0.y);
    vec3 g010 = vec3(gx0.z, gy0.z, gz0.z);
    vec3 g110 = vec3(gx0.w, gy0.w, gz0.w);
    vec3 g001 = vec3(gx1.x, gy1.x, gz1.x);
    vec3 g101 = vec3(gx1.y, gy1.y, gz1.y);
    vec3 g011 = vec3(gx1.z, gy1.z, gz1.z);
    vec3 g111 = vec3(gx1.w, gy1.w, gz1.w);

    vec4 norm0 = taylorInvSqrt(vec4(dot(g000, g000), dot(g010, g010), dot(g100, g100), dot(g110, g110)));
    g000 *= norm0.x;
    g010 *= norm0.y;
    g100 *= norm0.z;
    g110 *= norm0.w;
    vec4 norm1 = taylorInvSqrt(vec4(dot(g001, g001), dot(g011, g011), dot(g101, g101), dot(g111, g111)));
    g001 *= norm1.x;
    g011 *= norm1.y;
    g101 *= norm1.z;
    g111 *= norm1.w;

    float n000 = dot(g000, f0);
    float n100 = dot(g100, vec3(f1.x, f0.yz));
    float n010 = dot(g010, vec3(f0.x, f1.y, f0.z));
    float n110 = dot(g110, vec3(f1.xy, f0.z));
    float n001 = dot(g001, vec3(f0.xy, f1.z));
    float n101 = dot(g101, vec3(f1.x, f0.y, f1.z));
    float n011 = dot(g011, vec3(f0.x, f1.yz));
    float n111 = dot(g111, f1);

    vec3 fade_xyz = fade(f0);
    vec4 n_z = mix(vec4(n000, n100, n010, n110), vec4(n001, n101, n011, n111), fade_xyz.z);
    vec2 n_yz = mix(n_z.xy, n_z.zw, fade_xyz.y);
    float n_xyz = mix(n_yz.x, n_yz.y, fade_xyz.x);
    return 2.2 * n_xyz;
}
```

Figure 8: snippet of code used to implement Perlin noise

Explanation of the code:

- **mod289 and permute**:
  These functions are used to wrap indices and generate pseudo-random permutations, ensuring that the noise pattern repeats seamlessly across the grid.
- **taylorInvSqrt**:
  This function is used to normalize gradient vectors, ensuring they have unit length, which is crucial for consistent noise values.
- **fade**:
  This function is used to interpolate between the noise values at the corners of the grid.

The "noise" function proceeds to generate the Perlin noise following the procedure detailed in the section algorithms.

- **fragment_cube1.glsl**:
  This shader is used to render the cube in the first room, it uses Perlin noise with multiple octaves to modify the diffusive color of the cube.

```
float fbm(vec3 pos) {
    float amplitude = 0.5;
    float frequency = 1.0;
    float noiseSum = 0.0;
    float amplitudeSum = 0.0;

    // Add multiple octaves of noise
    for(int i = 0; i < 4; i++) {
        noiseSum += amplitude * noise(pos * frequency);
        amplitudeSum += amplitude;
        amplitude *= 0.5; // Halve the amplitude
        frequency *= 2.0; // Double the frequency
    }

    return noiseSum / amplitudeSum; // Normalize the result
}
```

Figure 9: snippet of code used to implement Perlin noise with multiple octaves

Explanation of the code:

- The for loop runs for a specified number of octaves, in this case 4.
- For each octave, the noise function is called with the position scaled by the current frequency.
- After each octave, the amplitude is halved, and the frequency is doubled.
- The final noise value is normalized by dividing by the sum of amplitudes to ensure it remains within a usable range.

A "noise" function identical to the one seen before proceeds to generate the Perlin noise following the procedure detailed in the section algorithms.

- **fragment_pyramid1.glsl**:
  This shader is used to render the pyramid in the first room, it uses Perlin noise with turbulence to modify the diffusive color of the pyramid.

```
float turbulence(vec3 pos) {
    float sum = 0.0;
    float frequency = 1.0;
    float amplitude = 1.0;
    float maxValue = 0.0;

    for(int i = 0; i < 4; i++) {
        sum += abs(noise(pos * frequency)) * amplitude;
        maxValue += amplitude;
        amplitude *= 0.5; // Halve the amplitude
        frequency *= 2.0; // Double the frequency
    }

    return sum / maxValue; // Normalize the result
}
```

Figure 10: snippet of code used to implement Perlin noise with turbulence

Explanation of the code:

- The for loop runs for a specified number of octaves, in this case 4.
- For each octave, the noise function is called with the position scaled by the current frequency. The absolute value of the noise is taken to create sharp features.
- After each octave, the amplitude is halved, and the frequency is doubled.
- The final turbulence value is normalized by dividing by the sum of amplitudes, ensuring it remains within a consistent range.

A "noise" function identical to the one seen before proceeds to generate the Perlin noise following the procedure detailed in the section algorithms.

2. **Simplex Noise (fragment_cube2.glsl)**:
   This shader is used to render the cube in the second room, it uses Simplex noise to modify the diffusive color of the cube.

```
float snoise(vec3 v) {
    const vec2 C = vec2(1.0/6.0, 1.0/3.0);
    const vec4 D = vec4(0.0, 0.5, 1.0, 2.0);

    // First corner
    vec3 i  = floor(v + dot(v, C.yyy));
    vec3 x0 = v - i + dot(i, C.xxx);

    // Other corners
    vec3 g = step(x0.yzx, x0.xyz);
    vec3 l = 1.0 - g;
    vec3 i1 = min(g.xyz, l.zxy);
    vec3 i2 = max(g.xyz, l.zxy);

    vec3 x1 = x0 - i1 + C.xxx;
    vec3 x2 = x0 - i2 + C.yyy;
    vec3 x3 = x0 - D.yyy;

    // Permutations
    i = mod289(i);
    vec4 p = permute(permute(permute(
        i.z + vec4(0.0, i1.z, i2.z, 1.0))
        + i.y + vec4(0.0, i1.y, i2.y, 1.0))
        + i.x + vec4(0.0, i1.x, i2.x, 1.0));
```

```
    // Gradients: 7x7 points over a square, mapped onto an octahedron.
    float n_ = 0.142857142857;
    vec3 ns = n_ * D.wyz - D.xzx;

    vec4 j = p - 49.0 * floor(p * ns.z * ns.z);

    vec4 x_ = floor(j * ns.z);
    vec4 y_ = floor(j - 7.0 * x_);

    vec4 x = x_ *ns.x + ns.yyyy;
    vec4 y = y_ *ns.x + ns.yyyy;
    vec4 h = 1.0 - abs(x) - abs(y);

    vec4 b0 = vec4(x.xy, y.xy);
    vec4 b1 = vec4(x.zw, y.zw);

    vec4 s0 = floor(b0)*2.0 + 1.0;
    vec4 s1 = floor(b1)*2.0 + 1.0;
    vec4 sh = -step(h, vec4(0.0));

    vec4 a0 = b0.xzyw + s0.xzyw*sh.xxyy;
    vec4 a1 = b1.xzyw + s1.xzyw*sh.zzww;

    vec3 p0 = vec3(a0.xy, h.x);
    vec3 p1 = vec3(a0.zw, h.y);
    vec3 p2 = vec3(a1.xy, h.z);
    vec3 p3 = vec3(a1.zw, h.w);

    // Normalise gradients
    vec4 norm = taylorInvSqrt(vec4(dot(p0,p0), dot(p1,p1), dot(p2,p2), dot(p3,p3)));
    p0 *= norm.x;
    p1 *= norm.y;
    p2 *= norm.z;
    p3 *= norm.w;

    // Mix final noise value
    vec4 m = max(0.6 - vec4(dot(x0,x0), dot(x1,x1), dot(x2,x2), dot(x3,x3)), 0.0);
    m = m * m;
    return 42.0 * dot(m*m, vec4(dot(p0,x0), dot(p1,x1), dot(p2,x2), dot(p3,x3)));
}
```

Figure 11: snippet of code used to implement Simplex noise

Explanation of the code:

- The input vector v is skewed to determine the simplex cell it falls into, the corners of the simplex are then calculated.
- Gradients are calculated at each corner of the simplex, these gradients are used to compute the influence of each corner on the final noise value.
- The final noise value is normalized to ensure it fits within a usable range.
- The noise values at the corners are interpolated using a smooth function to ensure continuous transitions.

3. **Multifractal Noise (fragment_sphere2.glsl)**:
   This shader is used to render the sphere in the second room, it uses Multifractal noise to modify the diffusive color of the sphere.

```
float multifractal(vec3 pos, float H, float lacunarity, int octaves) {
    float value = 1.0;
    float frequency = 1.0;
    float amplitude = 0.5;
    float weight = 1.0;

    for(int i = 0; i < octaves; i++) {
        value *= (weight * snoise(pos * frequency) + 1.0);
        frequency *= lacunarity;
        weight = value;
        weight = clamp(weight, 0.0, 1.0);
    }

    return value;
}
```

Figure 12: snippet of code used to implement Multifractal noise

Explanation of the code:

- The function starts with an initial value of 1.0, and sets the initial frequency and amplitude.
- The loop iterates over the specified number of octaves.
- For each octave, the simplex noise function snoise is called with the position scaled by the current frequency. The result is adjusted by the current weight and added to the value.
- The frequency is multiplied by the lacunarity to increase the detail for the next octave. The weight is updated based on the current value and clamped to ensure it remains within a valid range.
- The final multifractal noise value is returned, representing the combined effect of all octaves.

A "snoise" function identical to the one seen before proceeds to generate the Simplex noise.

4. **Cellular Noise (fragment_pyramid2.glsl)**:
   This shader is used to render the pyramid in the second room, it uses Cellular noise to modify the diffusive color of the pyramid.

```
vec3 hash3(vec3 p) {
    p = vec3(dot(p, vec3(127.1, 311.7, 74.7)),
             dot(p, vec3(269.5, 183.3, 246.1)),
             dot(p, vec3(113.5, 271.9, 124.6)));
    return -1.0 + 2.0 * fract(sin(p) * 43758.5453123);
}

float cellular(vec3 p) {
    vec3 i_p = floor(p);
    vec3 f_p = fract(p);

    float min_dist = 1.0;

    // Search neighboring cells
    for(int k = -1; k <= 1; k++)
    for(int j = -1; j <= 1; j++)
    for(int i = -1; i <= 1; i++) {
        vec3 neighbor = vec3(float(i), float(j), float(k));
        vec3 point = hash3(i_p + neighbor);
        point = 0.5 + 0.5 * sin(point * 6.2831853); // Animate points
        vec3 diff = neighbor + point - f_p;
        float dist = length(diff);
        min_dist = min(min_dist, dist);
    }

    return min_dist;
}
```

```
float enhancedCellular(vec3 p, float scale, float intensity) {
    float n = cellular(p * scale);
    float n2 = cellular(p * scale * 2.0 + 5.0);

    // Mix different frequencies
    return mix(n, n2, intensity);
}
```

Figure 13: snippet of code used to implement Cellular noise

Explanation of the code:

- **hash3**:
  The hash3 function generates pseudo-random feature points within each grid cell. It uses a combination of dot products and trigonometric functions to ensure randomness.
- **cellular**:
  The input position p is split into its integer (i_p) and fractional (f_p) parts. The integer part identifies the grid cell, while the fractional part is used to calculate distances within the cell. The shader iterates over neighboring cells to find the nearest feature point: the distance to each feature point is calculated, and the minimum distance is stored. The feature points are animated by applying a sine function, which can create dynamic patterns over time.
- **enhancedCellular**:
  The function generates two layers of cellular noise at different scales, this layering adds complexity and detail to the texture. The two noise layers are mixed together using a specified intensity, this allows for blending between the different scales of noise, creating a more intricate pattern.

5. **Perlin Noise on Normal Map (fragment_sphere3.glsl)**:
   This shader is used to render the sphere in the third room, it uses Perlin noise to modify the normal map of the sphere.

```glsl
// Uniforms for noise and appearance
uniform float noiseScale;
uniform float normalStrength;
uniform vec3 baseColor;
uniform float glossiness;

void main() {
    // Generate noise-based normal perturbation
    vec3 noisePos = FragPos * noiseScale;
    float n = noise(noisePos);
    float nx = noise(noisePos + vec3(0.1, 0.0, 0.0));
    float ny = noise(noisePos + vec3(0.0, 0.1, 0.0));

    // Calculate normal perturbation
    vec3 perturbation = vec3(n - nx, n - ny, 0.0) * normalStrength;
    vec3 norm = normalize(Normal + perturbation);

    // Lighting calculations with perturbed normal
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);

    // Ambient
    float ambientStrength = 0.2;
    vec3 ambient = ambientStrength * baseColor;

    // Diffuse with noise-affected normal
    vec3 diffuse = diff * baseColor;

    // Specular with adjustable glossiness
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), glossiness);
    float specularStrength = 1.0;
    vec3 specular = specularStrength * spec * vec3(1.0);

    vec3 result = ambient + diffuse + specular;
    FragColor = vec4(result, 1.0);
}
```

Figure 14: snippet of code used to implement Perlin noise on normal map

Explanation of the code:

- **Noise Generation**:
  The position FragPos is scaled by noiseScale to control the frequency of the noise. The noise function is called to generate noise values at the current position and slightly offset positions (nx and ny).
- **Normal Perturbation**:
  The difference between the noise values (n - nx and n - ny) is used to create a perturbation vector. This vector is scaled by normalStrength to control the

intensity of the normal perturbation. The original normal Normal is adjusted by adding the perturbation vector, and the result is normalized to ensure it remains a unit vector.

- **Lighting Calculations**:
  The perturbed normal norm is used in the lighting calculations, this affects the diffuse and specular components, creating the appearance of a textured surface. The diffuse component is calculated using the dot product of the perturbed normal and the light direction. The specular component is calculated using the reflection of the light direction around the perturbed normal, with the glossiness controlling the sharpness of the specular highlight.

A "noise" function identical to the one seen before proceeds to generate the Perlin noise following the procedure detailed in the section algorithms.

6. **Perlin Noise on Transparency (fragment_cube3.glsl)**:
   This shader is used to render the cube in the third room, it uses Perlin noise to modify the transparency of the cube.

```
// Uniforms for noise and appearance
uniform float noiseScale;
uniform float noiseIntensity;
uniform vec3 baseColor;
uniform float minAlpha;
uniform float maxAlpha;

void main() {
    // Generate noise
    float noiseValue = noise(FragPos * noiseScale) * 0.5 + 0.5;

    // Calculate alpha based on noise
    float alpha = mix(minAlpha, maxAlpha, noiseValue);

    // Lighting calculations
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);

    // Ambient
    float ambientStrength = 0.3;
    vec3 ambient = ambientStrength * baseColor;

    // Diffuse
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * baseColor;

    // Specular
    float specularStrength = 0.5;
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    vec3 specular = specularStrength * spec * vec3(1.0);

    vec3 result = ambient + diffuse + specular;
    FragColor = vec4(result, alpha);
}
```

Figure 15: snippet of code used to implement Perlin noise on transparency

Explanation of the code:

- **Noise Generation**:
  The position FragPos is scaled by noiseScale to control the frequency of the noise. The noise function is called to generate a noise value, which is normalized to the range [0, 1] by multiplying by 0.5 and adding 0.5.
- **Alpha Calculation**:
  The mix function is used to interpolate between minAlpha and maxAlpha

based on the noise value. This determines the transparency level of the fragment, with higher noise values resulting in higher transparency (or vice versa, depending on the values of minAlpha and maxAlpha).

- **Lighting Calculations**:
  The base color and lighting calculations (ambient, diffuse, and specular) are performed as usual, using the normalized normal vector and light direction.
- **Fragment Color Output**:
  The final fragment color is constructed using the calculated lighting components and the interpolated alpha value. The FragColor output includes the RGB color and the alpha transparency.

A "noise" function identical to the one seen before proceeds to generate the Perlin noise following the procedure detailed in the section algorithms.

4.2 Main code

This section is a short summary that explains how the code in the main file Rooms.cpp is structured:

- **Global Variables**:
  Camera and movement variables are defined to control the user's view and navigation through the 3D space. Shader program IDs are stored for different objects (cubes, spheres, pyramids) in each room.
- **Noise Parameters**:
  Parameters for different noise types (Perlin, Simplex, Multifractal, Cellular) are defined for each room.
- **Shader Setup**:
  A function readShaderFile is provided to read shader source code from files, which is then used to compile and link shader programs.
- **Input Handling**:
  Functions like "mouse_callback" and "processInput" handle user input for camera movement and interaction, such as toggling mouse capture and moving the camera with keyboard keys.
- **Object Setup and Rendering**:
  Functions are defined to set up vertex data and buffers for different objects (cubes, spheres, pyramids), rooms and corridors. These functions also compile and link shaders for rendering. Rendering functions use the shader programs to draw objects with the specified noise parameters and transformations.
- **Cleanup**:
  Cleanup functions are provided to delete OpenGL resources (VAOs, VBOs, EBOs, shader programs) when they are no longer needed.
- **ImGui Rendering functions**:
  The "renderNoiseControls" function is used to render the ImGui window, which contains the noise parameters for each room.

- **Main Function**:
  The main function initializes the application, creates a window, and enters the main loop, which continues until the window is closed, it also handles the loop, updating the view and projection matrices, and rendering the GUI and the objects using the various functions described in details here in each room. In the end, it cleans up the resources calling the apposite functions and exits the program.

5 Conclusions

The project is a simple but effective way to show and understand how noise can be used to modify the appearance of objects in a 3D scene in a OpenGL context. It also showcases the various properties, advantages and disadvantages of the different noise types, and how they can be used to create different effects.
Compagnoni Alessandro, 2025