CppCraft Project Report

Niccolò Pozzetti

November 2021

1 Description

Procedural voxel terrain generation demo using Perlin noise algorithm.

This project is based on craft: a MineCraft C clone written by Michael Fogleman `https://www.michaelfogleman.com/projects/craft/`.

1.1 Blocks

The generated world is a sequence of atomic cubic structures called *Blocks*, where each Block has 2 attributes: *position* and *type*. The *type* determines the element represented by the block (for example *rock, glass, cloud, leaves, red flower, ...*) and some of its properties (for example opacity). The *position* is a *(x,y,z) integer-tuple* that represents the 3D absolute coordinate of a block.

1.1.1 Material Blocks

This is a subclass of Blocks which represents a material (rock, grass, sand, ...) which may be part of more complex structures such as:

Hill a sequence of *grass-material* adjacent blocks shaped like a concave parable where for each x,z coordinate there is a block for each y between *base-level-y* and *highest-block-y*.

Valley a sequence of *sand* horizontal-adjacent block at *base-level-y* only interrupted by hills

Tree sequence of *wood* vertical-adjacent blocks and *leaves* circular-adjacent blocks found on hills

1.1.2 Plant Blocks

the other subclass of Blocks that includes flowers and grass. Graphically these type of Blocks are like 2D images so they do not have an upper or lower face. They are always located above *Hills*.

1.1.3 Empty Blocks

These blocks represent the absence of a block in the specified position, so they are used as a trick to model the emptiness.

1.2 Chunks

Since generating and rendering infinite blocks would be impossible, they are grouped in a cuboid structure called *Chunk* where each chunk is tiled in *(p,q) integer-pair* horizontal coordinates.

When a Chunk is initialized with Perlin noise it takes all the Block coordinates delimited by its area and maps a Block type for each coordinate. When initialization ends, each one of the block that manages can be rendered in a row. If a Chunk is deleted and then recreated, it will have the same Block types since the generation algorithm produces the same results if position is the same (this type of random generation depends on coordinates).

1.3 Light

The light system is very simple: every block reflects diffuse light and light changes between faces because it is computed by the dot product between the ray end the face normal. An ambient light contribution is used to increase brightness of very dark areas and obtain the illusion of global illumination.

1.3.1 Shadows with ambient occlusion

Shadows are computed using *ambient occlusion*: a global illumination technique that computes the *darkness* of an area of the scene by considering the occluding volumes around it (volumes that should block light rays from the ambient).

Areas with the highest number of occluding volumes are concave areas, especially holes, but you can appreciate more the effect of AO with corners, since the the shadow along them increases the feeling of a *realistic* 3D volume instead of a *simulated* one.

Analytical definition

$$A(p) = \int_{\omega} V(p,\omega)(\omega \cdot n)\, d\omega \tag{1}$$

- n normal of a surface (a triangle).
- p point of the surface with normal n.
- ω front hemisphere (includes the normal n) centered in p.
- *V(p,ω)* occlusion value.
  - *0* if p is occluded from ω direction.

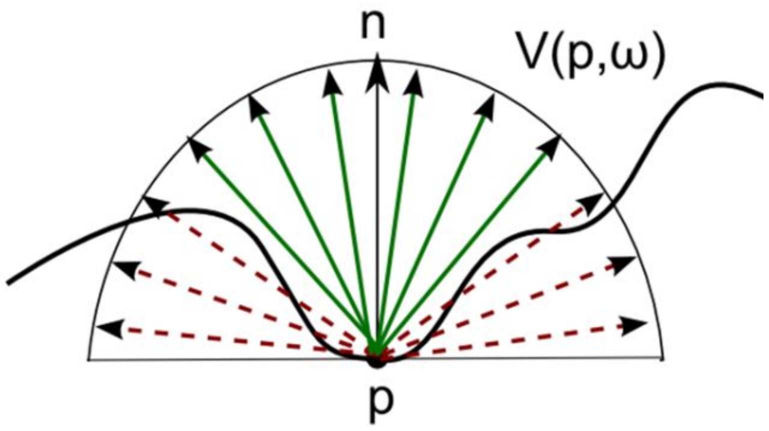


Figure 1: Visualization of *AO* analytical formula image source

   - *1* if p is not occluded from ω direction.

- $\omega \cdot n$ dot product between direction and normal (as the angle between the normal increases, rays contribute is lower).
- *A(p)* value of *AO* at point p.
   - *0* means that p is completely occluded.
   - *1* means that p is completely exposed from ambient rays.

As you can see in equation 1 and figure 1 you have the highest possible value if there is no occluding volume near that and you have the lowest one if it is in a sort of underground hole with no light sources.

Face *Shadows* are computed by testing Blocks adjacency for each face vertex (ambient occlusion): light level is linearly decreased as the number of opaque Blocks increases.

Implementation The implementation adopted in this project uses scene geometry thanks to the Voxel paradigm (unlike *SSAO* that uses depth buffer of pixel neighbors).

1. Given the position of a Cube Block the method *getLightObstacles(const glm::ivec3 &blockPos)* from Chunk class computes a map of boolean values where there is an entry for each adjacent block (blocks connected to its faces and angles) which has a value of 1 if it is opaque or 0 if it is transparent or empty.
2. these values are passed to *BlockObject* 's constructor where for each face index we compute the AO contribution $\frac{1}{1+k}$ (k is the number of occluding

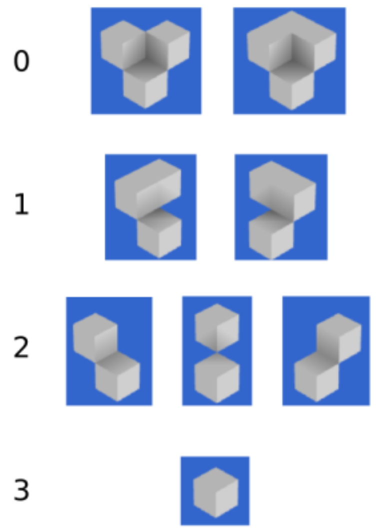


Figure 2: Different AO values depending on Blocks adjacent to the vertex `https://0fps.files.wordpress.com/2013/07/aovoxel2.png?w=293&h=410`

cubes) using the geometry information and we pass it to the *BlockShader* as a vertex VAO attribute.

3. vertex shader passes the AO contribution of each vertex to the fragment so it will be interpolated.
4. in the fragment shader the color of the fragment is multiplied with the AO contribution which is a float between 0 and 1.

In Figure 2 are taken into consideration the different light values of the upper Vertex of the lower Block (except in the last one where no Block occludes the upper Vertex, so light value is at maximum).

1.4 Camera

The 3D model is rendered using a virtual camera with perspective projection: this camera can move around the model using *WASD* and its front direction can be changed with mouse movement and Chunks are initialized and rendered when they are close enough (the *closeness* depends on a tunable integer). Chunks that are too far are deleted from *RAM* and *VRAM* and reloaded when camera returns near them (even the *proximity* factor can be tuned).

1.5 Fog

Color of distant Blocks is blended with sky color to obtain a foggy effect that gives the illusion that not rendered Chunks are simply hidden behind this virtual fog. This effect is heavier if *rendering distance* is set to low values, since Block 's true color vanishes more quickly.

1.6 Terrain Generation

1.6.1 Theory

Every Chunk generates a map of blocks using Perlin noise algorithm. Pseudo-random number generators output a discontinuous series of numbers given a seed, instead Perlin noise is a linear combination of shrinked continuous functions (). Each function member takes a *d-dimensional* coordinate as input and outputs a float scalar and it is defined in this way:

$$\begin{cases} noise(x) = 0 & \text{if } round(x) = x \\ noise'(x) = k(x) & \text{if } round(x) = x \end{cases} \tag{2}$$

Where x is a d-dimensional coordinate and *k(x)* is the gradient at coordinate x pseudo-random generated and gradient values between integers are interpolated.

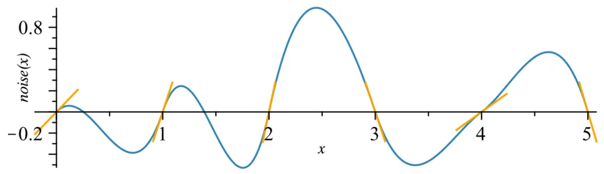


Figure 3: continuous noise function component for 1D input source

The final function is computed as:

$$Noise(x) = \sum_{n=0}^{N-1} a_n \cdot noise(f_n \cdot x) \tag{3}$$

Which has 3 parameters:

octaves (N) number of components

scale resolution of the octave (implementation dependent)

- this value is used to multiply the gradient and obtain consecutive values (dx if it is infinitesimal).
- higher values correspond to a coarser function and lower ones correspond to a finer one.

lacunarity (f_n) density of detail of ith-octave (frequency)

persistence (a_n) contribution to final result of the ith-octave (amplitude)]

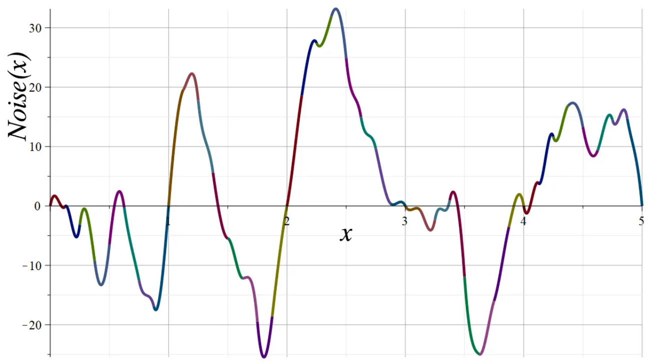


Figure 4: Final function for 1D coordinate source

1.6.2 Implementation

Terrain generation is implemented in this project with an algorithm similar to Perlin called Simplex noise: it has the same objective, but instead of gradients it uses *simplices* (n-dimensional triangles) for performance reasons.

1. Each Chunk is divided into a $ChunkSizexChunkSize$ horizontal grid.
2. each point of the grid identifies a 2D-integer-coordinate which is multiplied with a small float (the *scale* parameter) and passed to Simplex function.
3. the output is normalized between 0 and 1 so it is scaled with an integer constant to simplify computations.
4. if scaled output *Simplex(sx,sy)* is below a certain *treshold* it is considered part of a valley so at (x, *treshold*, z) coordinates a sand block is set else it is set a grass Block for each int coordinate in the range *treshold, Simplex(sx, sy)*.
5. using the same principle, a treshold is used to decide to put or not a tree or a plant onto a grass Block.
6. finally x,z couple is augmented with all y coordinates from the range (64,72) and these 3D coordinates are also fed to Simplex to decide to generate a cloud or not.

2 Development

While porting the original C project to C++ I run into problems mainly caused by misunderstanding of the original code and difficulties on handling large amounts of data. The original project can be divided into 5 parts:

1. rendering

2. terrain generation
3. terrain manipulation
4. collisions physics
5. saving and loading game

At first my goal was to implement the first four parts, but at the end I realized that the physical and interactive sides of the project were really difficult and deeply interconnected with the rendering and terrain generation side, so at the end I focused on the first 2.

Here I will describe the major problems I encountered during the development and corresponding solutions.

2.1 Linear Algebra Operations

Initially I started to implement my set of linear algebra tools but, but the project needed lots of them, so at the end I decided to use GLM since it gives lots of useful tools for computer graphics algebra.

2.2 Blocks rendering

Intuitively every Block should be considered as a separate renderable unit but this causes the following problems:

- Every face (even the hidden ones) will be computed and rendered
- For every Block there is an *OpenGL* buffer: this means that rendering a Chunks implies thousand of calls to *buffer allocation* and *rendering* (1 for each cube of the Chunk).

So I decided to integrate Blocks generation and rendering into Chunk functions:

1. every Chunk contains a map where the key is a position and the value is the type of Block stored in that position (no key for empty Blocks or Blocks outside Chunk)
2. this Block map is used to build geometry info about the Blocks:
   (a) the position and number of visible faces is computed for each Block
   (b) buffer is filled with visible faces ' vertices

At the end every Block is no longer a separate entity since it is a range of vertices into the buffer managed by the corresponding Chunk.

2.3 Chunk loading and initialization

Initially Chunks were loaded sequentially with Block map initialization inside constructor, but the demo was unusable due to very long loading times since Block map generation with Perlin noise consists of lots of for loops. To avoid this problems I associated a thread with the function responsible of *Block map generation and Block buffer computation* so the the caller thread can continue execution and render visible Chunks, while rendering a *hole* if the Chunk in the corresponding position is not ready. I used thread library from *C++ standard library* for this operation.

2.4 Chunk removal

Chunks that are too distant are removed from memory to free up space and initially I used asynchronous threads to avoid the need of waiting Chunk initialization end (especially for the ones that were not ready when game window should close), but this caused errors since threads were still running at the and of main thread execution. At the end I chose joinable threads but to avoid stuttering or long waiting times I set the default value of *delete chunk distance* to an higher value than create distance.

2.5 Interaction components

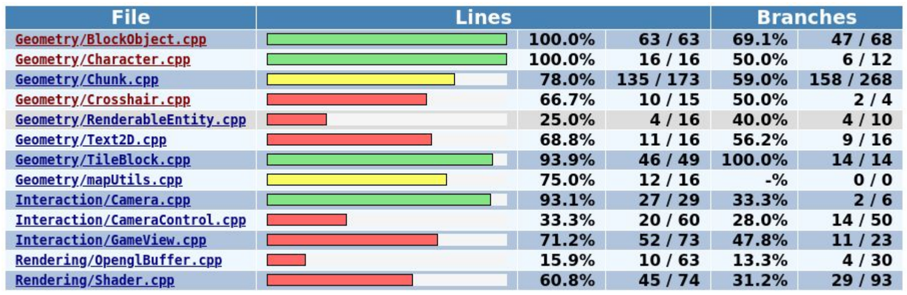
External input and window handling rely on a library called GLFW.

This library is written in C and and uses callback functions to handle input, so its function were very difficult to interface with my classes.

To solve this problem I decided to use these library only into a subset of my classes (*GameView, Scene and CameraControl*) and then in these classes I used the singleton pattern to always have 0 or 1 active instances of these classes. These classes are instantiated with a static method that returns a unique pointer, so the caller is the owner of the object and when this unique pointer is set to nullptr or the caller goes out of scope, the object destructor is called. This is especially useful with GameView class since this destructor terminates glfw. I decided to move ownership to callers because static objects get destroyed too late to free up dynamic memory data (this was reported on *stderr*), but callers are stack objects that get destroyed when they go out of scope.

2.6 Vertex attributes size

Since opengl buffers need info about the size of each attribute and these attributes change between vertex types (for example a 2D vertex has 2 floats for position and 3D vertex has 3 floats for position) I decided to use *SFINAE* to compute statically the size of these attributes, without writing different functions for different cases.



File	Lines			Branches	
Geometry/BlockObject.cpp	🟩	100.0%	63 / 63	69.1%	47 / 68
Geometry/Character.cpp	🟩	100.0%	16 / 16	50.0%	6 / 12
Geometry/Chunk.cpp	🟨	78.0%	135 / 173	59.0%	158 / 268
Geometry/Crosshair.cpp	🟥	66.7%	10 / 15	50.0%	2 / 4
Geometry/RenderableEntity.cpp	🟥	25.0%	4 / 16	40.0%	4 / 10
Geometry/Text2D.cpp	🟥	68.8%	11 / 16	56.2%	9 / 16
Geometry/TileBlock.cpp	🟩	93.9%	46 / 49	100.0%	14 / 14
Geometry/mapUtils.cpp	🟨	75.0%	12 / 16	-%	0 / 0
Interaction/Camera.cpp	🟩	93.1%	27 / 29	33.3%	2 / 6
Interaction/CameraControl.cpp	🟥	33.3%	20 / 60	28.0%	14 / 50
Interaction/GameView.cpp	🟥	71.2%	52 / 73	47.8%	11 / 23
Rendering/OpenglBuffer.cpp	🟥	15.9%	10 / 63	13.3%	4 / 30
Rendering/Shader.cpp	🟥	60.8%	45 / 74	31.2%	29 / 93

Figure 5: coverage result after running catch tests

3 Testing and coverage

3.1 Testing Tool

I decided to use *catch 2* to test code and I wrapped all test cases into a single executable called *catch_test* .

3.2 Tests structure

For every tested class there is a source file named with pattern *classNameTest.cpp*. 3 classes has not been tested:

Scene this is a wrapper used to simplify the main, so it has not been tested

Item this class has not been included in the final demo

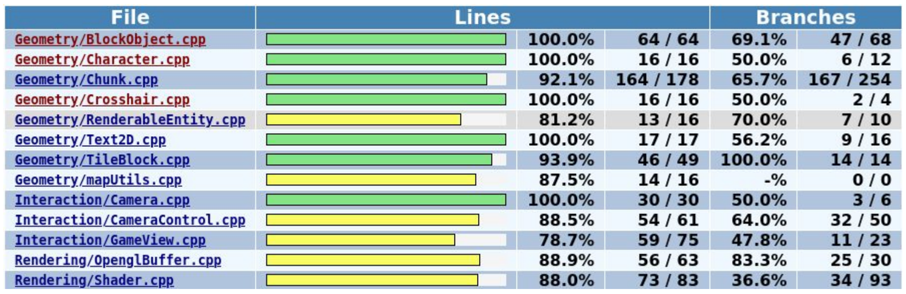
Sphere this class has not been included in the final demo

Since some classes contain code used by render the model and handle interaction, I did not test that code because his effects are only visible through user input and window output (rendering).

3.3 Coverage results

As you can see in figure 5, classes containing methods which use rendering (*glad* and *glfw*) functions or interaction (*glfw*) functions have a low coverage percentage, but if you run the demo (that uses the interaction and rendering functions) percentages drastically increases as seen in figure 6.

This is obvious, but since these functions are used for interaction between model and user, if they work as they should, the input-output should reflect the user' s expectations.



File	Lines			Branches	
Geometry/BlockObject.cpp	🟩	100.0%	64 / 64	69.1%	47 / 68
Geometry/Character.cpp	🟩	100.0%	16 / 16	50.0%	6 / 12
Geometry/Chunk.cpp	🟩	92.1%	164 / 178	65.7%	167 / 254
Geometry/Crosshair.cpp	🟩	100.0%	16 / 16	50.0%	2 / 4
Geometry/RenderableEntity.cpp	🟨	81.2%	13 / 16	70.0%	7 / 10
Geometry/Text2D.cpp	🟩	100.0%	17 / 17	56.2%	9 / 16
Geometry/TileBlock.cpp	🟩	93.9%	46 / 49	100.0%	14 / 14
Geometry/mapUtils.cpp	🟨	87.5%	14 / 16	-%	0 / 0
Interaction/Camera.cpp	🟩	100.0%	30 / 30	50.0%	3 / 6
Interaction/CameraControl.cpp	🟨	88.5%	54 / 61	64.0%	32 / 50
Interaction/GameView.cpp	🟨	78.7%	59 / 75	47.8%	11 / 23
Rendering/OpenglBuffer.cpp	🟨	88.9%	56 / 63	83.3%	25 / 30
Rendering/Shader.cpp	🟨	88.0%	73 / 83	36.6%	34 / 93

Figure 6: coverage result after running the demo *app*

4 Memory checks

4.1 Address and Undefined Behavior Sanitizer

I compiled and linked my project files with these flags and then I run both *demo executable* and the *tests* written with catch but no errors were reported.

4.2 Valgrind Memcheck

I run code with *Valgrind Memcheck tool* and I printed a log in doc folder. The log reported lots of leaks whose origin is the call to *glfwInit* (which is called by *GameView* class constructor): this call executes lot of system level code connected to *X11 libraries* used by GLFW to handle windows. *glfwTerminate* is always called so these leaks must not be caused by glfw of my code. I saw that part of the leaks was also caused by video drivers, so this may be the cause of them.

5 Static Analysis

I have checked my code with *clang-tidy* and it did not find any errors, but it reported some warnings regarding implicit conversions and use of constants inside *if commands.* I did not change these lines of code since implicit conversion are kept for readability and constants are part of the *tunable parameters* which are decided before code compilation.