4th Oral Presentation

From

Team

Erwan Vasseure Aenora Tye Renaud Gaubert Romain Biessy

Contents

Chapter 1

Introduction

Markus “Notch” Persson spent two years on Minecraft full time, helped with five to ten people who were all experienced game developers.
As for us, students of Epita, we spent a mere nine month develeloping and, the result is an awesome game!

SkyLands is a Minecraft-like game in which we added some strategy. Thus you are able to create units and buildings in order to defeat your foes just like a strategy game, except that it’s in a Minecraft world!

Chapter 2

GUI

2.1 A windowed Game

Following the example given by Minecraft, we decided to give up on a full screen game and go for a windowed game.
This means that users can resize the window in which the game is to any size they want.

2.2 A GUI flawed

We recently discovered that Mogre, the 3d engine we were using was no longer maintained (and hasen't been for two years) and that it had tremendous flaws!
You already know that Mogre is a .NET wrapper for Ogre (a C++ 3d engine), which means that Mogre is just a "translator" however translating from C# to C++ isn't that easy.
Thus many functions which are in Ogre are not translated in Mogre and therefore, for some special cases, we need to access directly to the C++ functions or use pointers.

For the GUI it was worse, we could get Mogre to display images but never text, and we couldn't do anything about it because of a C# to C++ error...
We tried tinkering in the source of Mogre but we never could make our way out (they were using a weird procedure, wrapping from C++ to JAVA and the translating the wrapper in C#).
We tried using some of the new Ogre features which were added in the 1.8 version of Ogre (they are now writting the 2.0 version) however after to translating a few thousands lines of C++ to C#, without surprise we dsicovered that the latest version of Mogre (1.7.4) could not work with the new GUI features.

At that moment our spirit was really low, we didn't know what to do and couldn't really continue our Menus without doing horrible things.
However by sheer luck we discovered an awesome library, which allowed us to create Web UI which is just a fancy term for any interface built with HTML, JS, and CSS. Awesomium is a library which provides the special sauce that allowed us to integrate Web UI content in any .NET application!
And guess what! The C# version of Awesomium is just a wrapper of the C++ version!

2.3 An HTML5, CSS3 and JavaScript Menu

Finding a way to display our menus and what's more in HTML, CSS and JavaScript was a releif. However we had to completly destroy the old interface and thus rebuild our GIU from scratch. We first had to rebuild the previous GUI (we chose a different theme that time) and then the ingame HUD.

2.3.1 Main Menu

Chapter 3

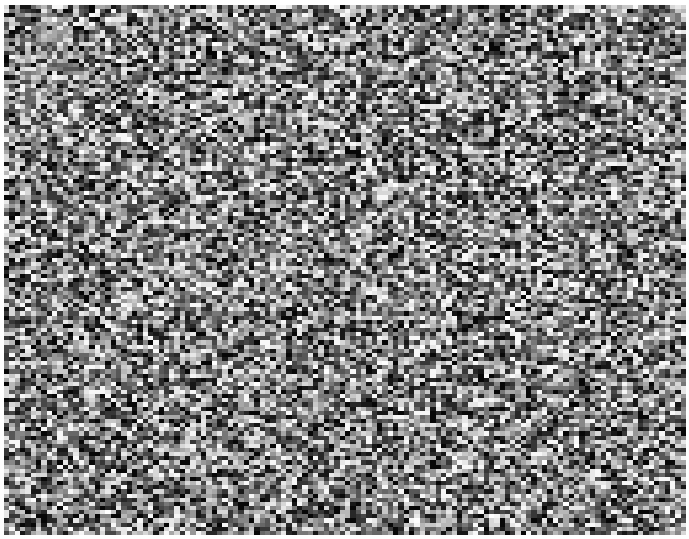
Random Terrain Generation

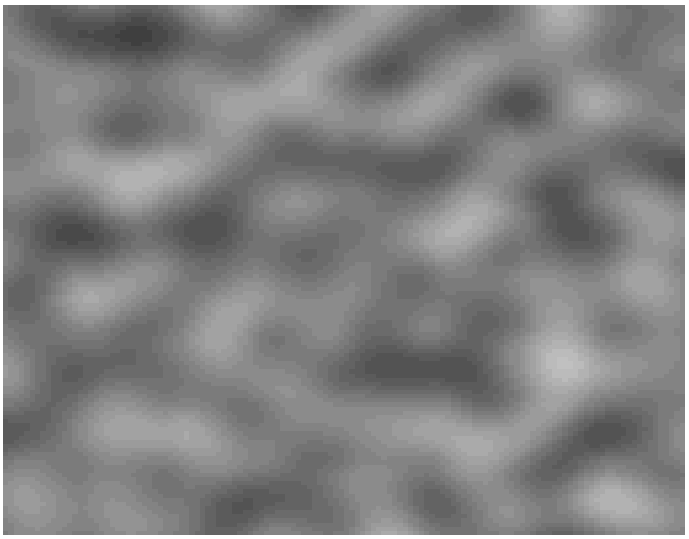
3.1 Perlin Noise

If you look at many things in nature, you will notice that they are fractal. They have various levels of detail. A common example is the outline of a mountain range. It contains large variations in height (the mountains), medium variations (hills), small variations (boulders), tiny variations (stones) . . . you could go on.
Look at almost anything: the distribution of patchy grass on a field, waves in the sea, the movements of an ant, the movement of branches of a tree, patterns in marble, winds. All these phenomena exhibit the same pattern of large and small variations. The Perlin Noise function recreates this by simply adding up noisy functions at a range of different scales.

3.1.1 Noise

Perlin noise generates coherent noise over a space. Coherent noise means that for any two points in the space, the value of the noise function changes smoothly as you move from one point to the other – that is, there are no discontinuities.





3.1.2 Generating Perlin Noise

The outline of our algorithm to create noise is very simple. Given an input point P, look at each of the surrounding grid points. In three dimensions there will be eight.
For each surrounding grid point Q, we choose a pseudo-random gradient vector G. It is very important that for any particular grid point you always choose the same gradient vector.

Compute the inner product G . (P-Q). This will give the value at P of the linear function with gradient G which is zero at the grid point Q.

When we have 8 of these values. We interpolate between them down to the point, using an S-shaped cross-fade curve (eg: 3t2-2t3) to weight the interpolant in each dimension (we could use a sin function but it would be too slow). This step requires computing 8 S curves, followed by 8-1 linear interpolations (the value has a dimension of degree 1).

We then create an array of the width, height and length of the terrain and fill it with the perlin noise value. We then iterate through the array's 3 dimensions and calculate a noise value for each cells of the array.
The noise value is based on experimentations and advice given by experienced people. At the begining, the value was computed using a simple linear function, however it is now computed using a polynomal function ad two noise arrays :

```
double noiseValue = (noise[xx, yy, zz]) * noise2[xx, 255 - yy, zz] + noise[
    xx, yy, zz] - System.Math.Abs(1 / smoothHeight * (yy - smoothHeight -
    minElevation + 20))
```

Where smoothHeight correspond to (maxElevation - minElevation) / 2. The noise value computed is a value between -1 and 1 and if it is lower than 0 we consider that it is air.

3.2 Different Islands

The smoothHeight value allows us to make different type of Islands, such as plains or even mountains. Because when smoothHeight's value is High, the values computed by noiseValue are higher and tend to be more scattered whereas low values tend to be more concentrated on one level.

3.3 Decorating the Islands

By default, the terrain is just a big cube of stone. However, when generating it, we add air blocks and, solid blocks under air blocks are usually grass (it can be set to another block).
Since we do not want to iterate through the terrain multiple times, this is done when generating the terrain thanks to little algorithm trick :

```
for (int xx = 0; xx < this.mIslandSize.x * Cst.CHUNK_SIDE; xx++) {
    for (int zz = 0; zz < this.mIslandSize.z * Cst.CHUNK_SIDE; zz
        ++) {
      for (int yy = 250; yy > 0; yy--) {
```

We create the terrain from the top, which allow us to check for each block if it's under an air block and if that's true, place a block of grass.

In case the block we just created is solid, we calculate it's distance from the surface and if we told the program beforeHand that there was a special block at the block's particular depth then the type of block will be the one we set, else it's stone.

Thus in most of the Islands, the first layer is grass and the next two are dirt.

3.4 Structures

3.5 Displaying

3.6 Save and Load

Chapter 4

Physics

4.1 Collisions

4.2 Fall

Chapter 5

Gameplay

5.1 Exploring the terrain

5.2 Adding Blocks

5.3 Removing Blocks

5.4 Inventory, Selector

5.5 Crafting

Chapter 6

Strategy

6.1 Build your own Base

6.2 Buy your army of Robots

6.3 Fight against foes

6.4 PathFinding

Chapter 7

Graphics

7.1 High / Low definition

7.2 Sinbad's skins

7.3 Particles

Chapter 8

Scenario Editor

8.1 Basic Idea

8.2 Structures

8.3 Scripting

Chapter 9

Sound Engine

Chapter 10

Website

Chapter 11

Conclusion