NOITA "Sand System"

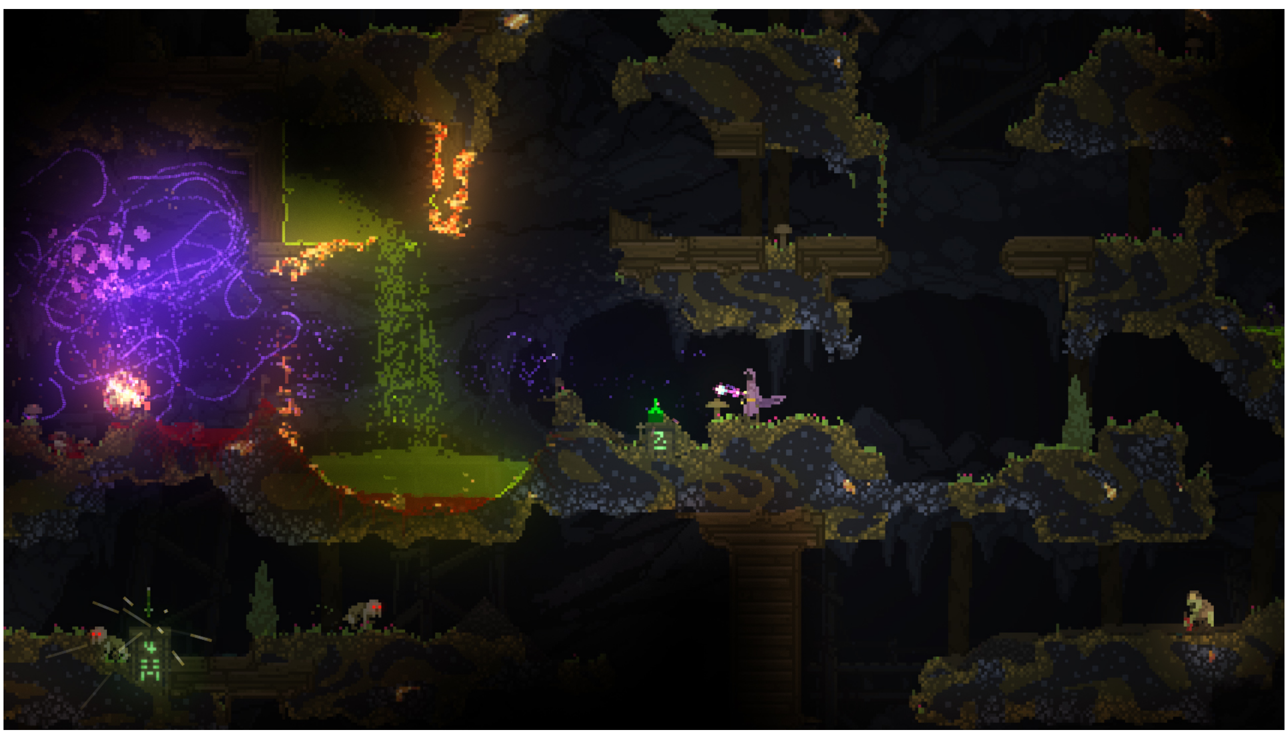
Clara RIPARD MINISINI
Lukas TAMAYO

April 2021

1 NOITA

"Dive into a world where every pixel simulates" is the concept promoted by the game "Noita" released in October 2020. But what 'simulates' means and how to implement it? This is what we wanted to dig while choosing this subject.

First, let's see what "Noita" is:

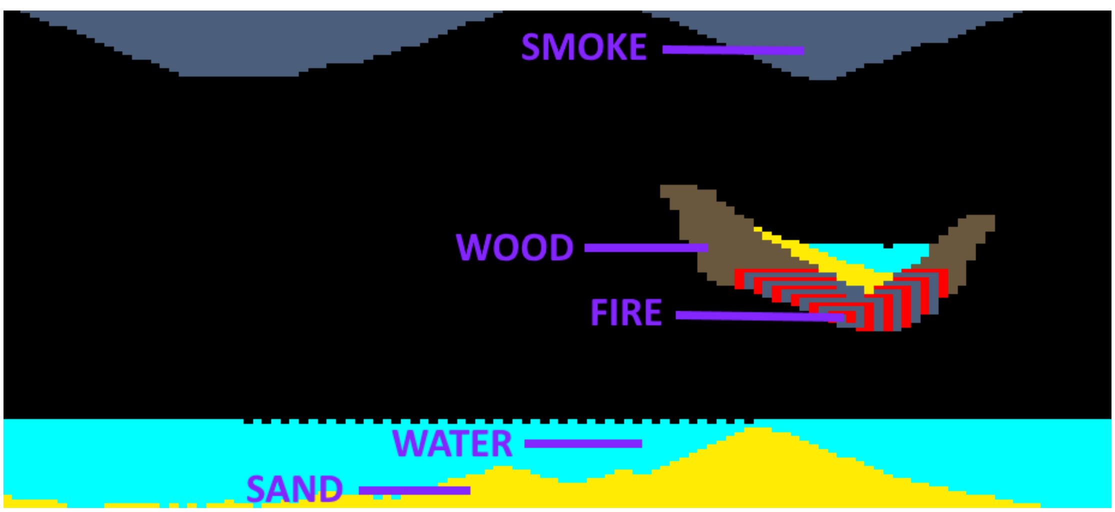


Noita screenshot

2 How?

If that seems quite incredible to figure out at first, but very brief researches will converge to one clear concept: Sand System. Every pixel is then considered as an element in the environment, constituted itself by multiple other pixels with the same behavior. The “Sand” referred here means that any of these pixels is up to react to any state change in the same way the others would. It can then burns, flows, corrodes, solidifies, etc. The idea here is to reproduce a credible – and plausible – behavior, and to apply this behavior to the neighbors, which can also apply this to other neighbors and so on.

For example, if the pixel as a state of ’wood’ and a mouse click set as ’fire’ applies, it may burn – but fire is known to expand providing a source of consumable energy is still reachable. Then the pixel will ’burn’, but if its neighbors are also made of ’wood’, then they will probably burn as well – and their neighbors as long as they are also flammable resources, and the original fire will expand until it is stopped – whereas there are no resources left, or it is countered by ’water’ for example.

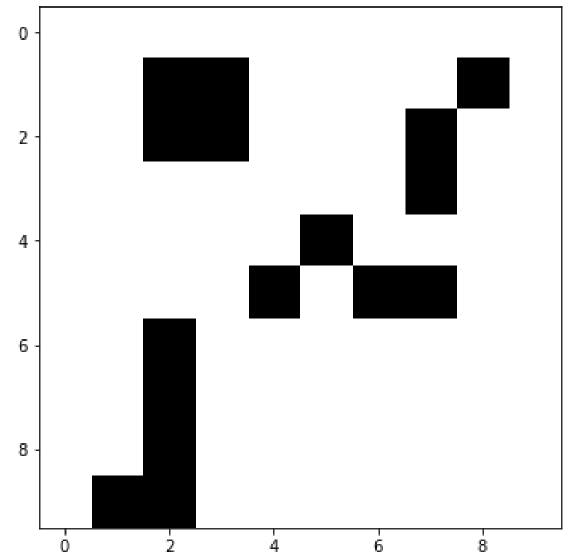


Sand, Water, Wood, Fire, Smoke

Once this system is clearly understood, there are two majors ideologies confronting each other to actually reproduce this feature: one implies physics, the other doesn’t. The GIF above has been coded without physics, simulating colors and effects through shader programming, while Noita has been.

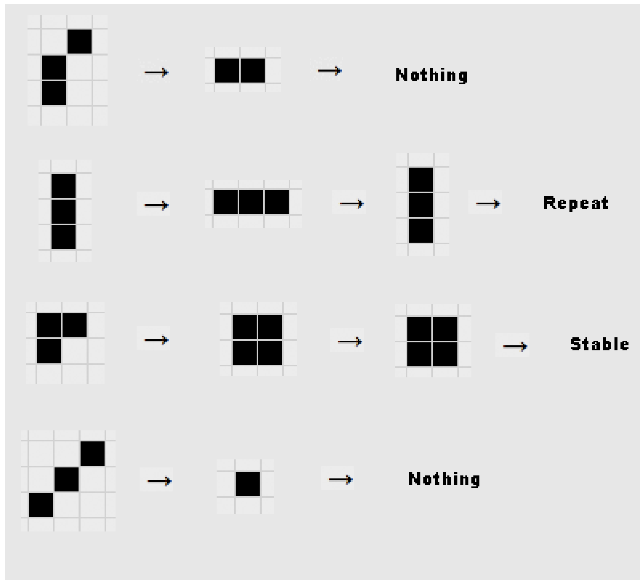
In any cases, some answers seem limpid – such as ’how to apply a state on a pixel?’ but some questionings remain – like ’how to duplicate this effect on neighbors?’ regardless the chosen method. The simplest answer would allow few more steps to understand how to do this: let’s talk about Conway’s 1970 “Game of Life”.

3 GAME OF LIFE



Game of Life example

We may not show something new, so here's a short explanation: cells are simulated autonomously on screen as 'alive' or 'dead'. If three neighbors are stated as 'alive', then it will either go to life or remain alive as well. On the contrary more or less than three neighbors will conduct to a 'dead' state.

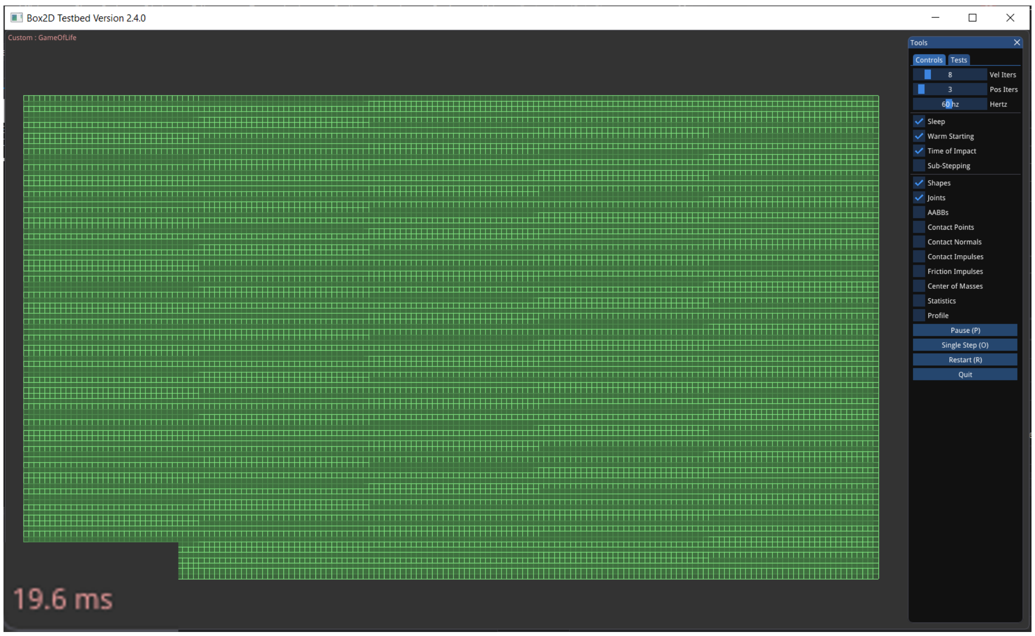


Game of Life concept

So at first we decided to focus on how we could recreate this feature without physics through coloring cells on screen, but after some first tests made on Unity that we will show later, we tried on the physics way. To experiment this method, we chose Box2D as support interface.

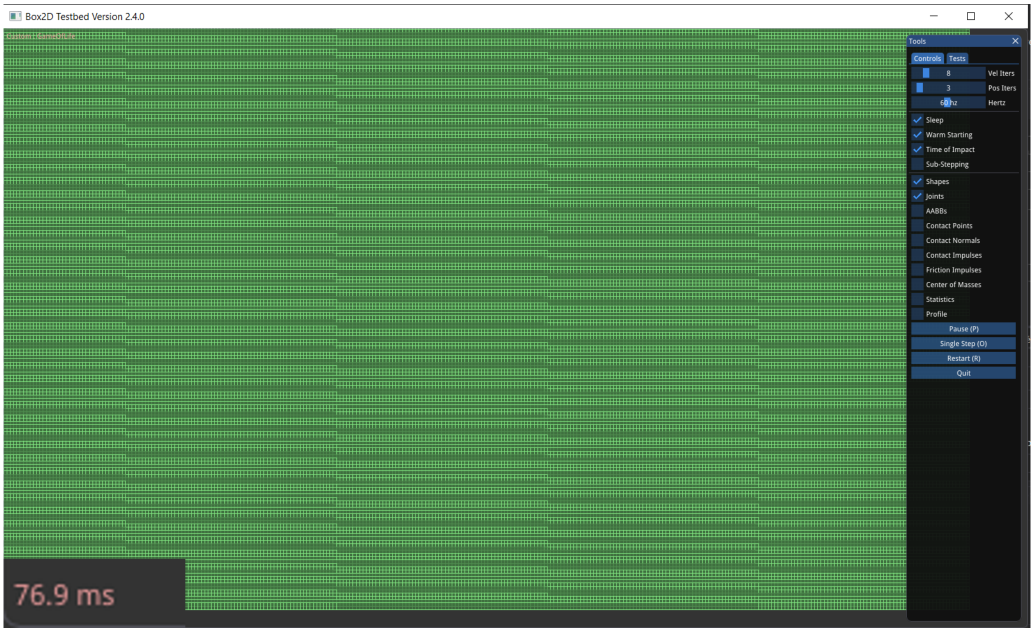
4 BOX2D

For a bit of clarity, Box2D is an engine that simulates plenty of 2D physics through multiple scenes (collision, gravity, impulsion, etc). In our case we wanted to have a full table of squares, so that we could apply the neighborhood principle. We just had to copy and adapt a scene that already exists – the squares pyramid, that we reorganized as a grid instead. And this single result was... Hardly relevant.



180x90 Box2D Grid performance

Printing a 160x90 grid assure a nice performance between 10 and 30 ms. But there are still very few cells if we want to simulate PIXELS! That is why we tried out to double this amount. And...



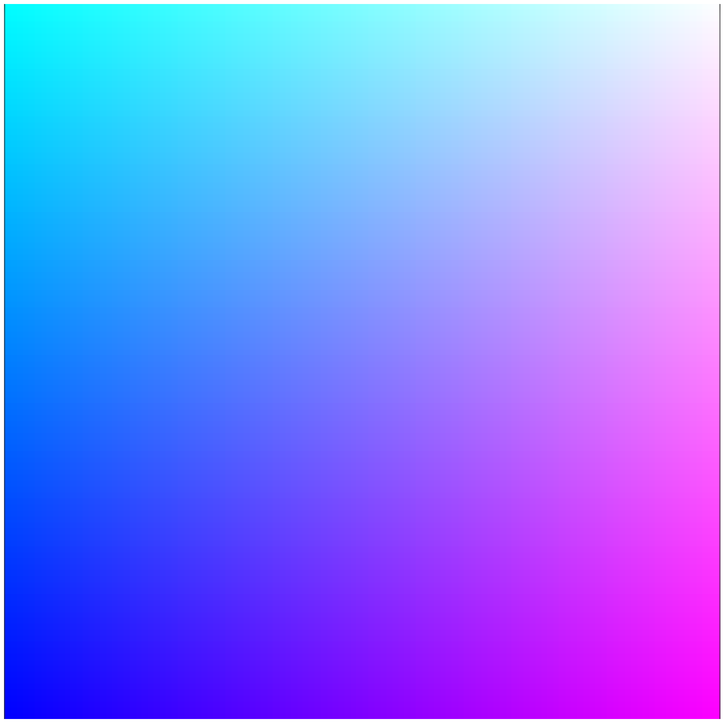
320x180 Box2D Grid performance

Printing a 320x180 cells fell down to a critical 60 to 100 ms. And we were still far, far from pixels. If only displaying squares with a deactivated physics caused such a lag, we admitted that this support was not adapted at all for our experience.

5 UNREAL ENGINE 4

In the same time period we tried to make things work on Box2D, we were also trying to make the simulation work on Unreal Engine 4 because of its powerful use of C++ and being known in the industry as a very optimized game engine. Even though we didn't record a screenshot from our little experience, the performance was very similar to Box2D and it wasn't enough to create the sand simulation we wanted in the hope of making it work with gameplay. We came to the conclusion that we weren't doing it the right way.

6 UNITY 2020



First result on UNITY while dividing cells

As said before, Unity was actually our first choice but the opening test was also a fail. When we looked at MrFrenik's "Sand Simulation" (see references), we could see that the program was coded in C using shader rendering - something we were also able to do on Unity.

So the first step was to apply the "Game of Life" principle in a blank project so that we could manage the neighbor check behavior with both "ACTIVE" and "INACTIVE" states. Of course we wouldn't make the whole game for it would be a bit of digression, so once coloring cells from black to white - according to their 'living' state - was done, we were able to replace the "ACTIVE" state by actual elemental states.

The first cleared was of course "SAND", as it was the default behavior for reproducing some physics logic: the sand is presented as small grains that can fall and pile up when several occupy the same space.

Then we added some "WATER" with the same behavior, but lighter than sand - thus if two cells are on the same space, the heavier (sand) falls and water updates above.

To test it with static objects, we added the "WOOD" state that could not fall (unlike actual wood does, true) but able to contains some water or sand. If the wooden container was full, the grains fell down as well.

But "WOOD" would not have been a pertinent choice for us faced to "ROCK" for example if unable to react to other elements as sand and water did: this is why it became flammable with a "FIRE" state that burns wood, stops in water, with no impact on sand, and that creates "SMOKE" when in contact to wood. Its behavior repeats in space while there is some consumable on the way long.

"SMOKE" was the opposite of SAND, lifted up when fire was produced, and appears instantly when something is 'burnt' through "LAVA" or "FIRE".

Finally "LAVA" is "FIRE"'s liquid state, it can burn water if there is more, as well as reduces when totally flown. Only "SAND" does not react to a "LAVA" state.

```
public struct Cell
{
    private CellState currentState;
    public Color currentColor;
    public CellState nextState;

    6 references | Lukas TAMAYO, 20 hours ago | 1 author, 6 changes
    public CellState CurrentState
    {
        get { return currentState; }
        set
        {
            switch (value)
            {
                case CellState.INACTIVE:
                    currentColor = Color.black;
                    break;
                case CellState.ACTIVE:
                    currentColor = Color.white;
                    break;
                case CellState.SAND:
                    currentColor = Color.yellow;
                    break;
                case CellState.WATER:
                    currentColor = Color.cyan;
                    break;
                case CellState.LAVA:
                    currentColor = new Color(0.75f,0.288f,0.156f);
                    break;
                case CellState.WOOD:
                    currentColor = new Color(0.413f,0.346f,0.240f);
                    break;
                case CellState.FIRE:
                    currentColor = Color.red;
                    break;
                case CellState.SMOKE:
                    currentColor = Color.gray;
                    currentColor.a = 0.33f;
                    break;
                default:
                    break;
            }
            currentState = value;
        }
    }
}
```

Struct State

7 Conclusion

At first "NOITA" system seemed quite a puzzle for us ("how is it even possible and optimized at the same time") but when we dug out we came to the conclusion that this feature where every pixel 'simulates' on screen was actually pretty simple to reproduce - and there were several methods to do it! So as long as we know how to keep optimized the chosen support, we can program it in any of them. But when it comes to hundreds of thousands of pixels and dozens of different states interacting with each other on screen each frame, sometimes while some elements are animated, the choice of a custom engine adapted for such a 2D simulation seems pretty obvious. This provides to an oriented in pixel performance, particles stimulation and gameplay physics.

References

Conway's Game of Life Wikipedia page

"Recreating Noita's Sand Simulation in C and OpenGL — Game Engineering" by John Jackson on Youtube

"Fantastic Arcade 2018: Noita gameplay and technical presentation by Petri Purho" by Fantastic Arcade on Youtube