How pixel-perfect physics works?

I always wondered how pixel-perfect physics works in such games like Worms: Armageddon, Liero and similar games. It was quite a challenge to implement such physics in my game. I was aiming to quite large maps - 10000x10000 pixels in size, and it was rather difficult to find an optimal way to check collisions between physical bodies and set of 100 millions (!) of pixels.

The main difference between my game and others: the game has fluid physics and requires very fast collision detection&response algorithm to grind up thousands of physical particles and calculate collision detection between those particles and millions of pixels.

In The Dwarf Adventure terrain represented as a rectangular image where each pixel contains color (R, G, B, 3 bytes) and some additional info – set of flags packed in 16-bit integer (2 bytes), so each pixel takes 5 bytes of memory. The flags should be described more carefully. Each bit of a flag was used to give a pixel some property, for example:

1. Will pixel participate in collision or it will be ignored (so called “solid” pixels)
2. Can pixel be visible for ray-tracing operation or will be transparent for them (suitable for glass, for example) (so called “transparent” pixels)
3. Can pixel be destroyed
4. And so on

As you can see it is very compact approach of storing properties of a pixel, especially when you have 100kk pixels – in this case the whole terrain will take only 476 Mb of memory, which is very low amount for modern hardware (just look at the memory amount consumed by your browser when you reading this article).

Circle was chosen as a shape of physical body – it is the simpliest shape, which allows checking collisions very fast. Circle shape allows doing a fluid simulation, which is another very interesting part of the game. More complex shapes, like polygons require more complex collision detection algorithms (like SAT and so on).

My very first algorithm was pretty straight-forward: iterate over every pixel in the terrain and check intersection between every physical body to find a push vector which will allow me to push body outside of “solid” parts of a terrain. As was expected, straight-forward approach was very slow and the game was totally unplayable: I was able to play with decent FPS only on small maps (1500x700px) with very little amount of physical bodies (less than 20-30) on quite powerful processor – 8 cores with 3.8 GHz, of course now I’m using only single core, but be patient – I’ll talk about multithreaded simulation quite soon. The main disadvantage of this approach is very obvious – we doing a lot of unnecessary operations: checking collisions with pixels that are too far away from physical bodies. So it the time to do optimizations!

Firstly, we should uniformly divide our terrain into a set of square cells of same size – I choose 256x256 pixels cell. Each cell will contain its portion of pixels of the terrain. And now we can modify our algorithm: instead of checking every pixel of the terrain, let’s firstly check intersection of a physical body with a terrain cell and if they are not intersecting we can discard a huge amount of pixels! With this very simple optimization, I increased performance up to 50 times!

But we can go further – modern processors have at least 2 cores nowadays and more commonly 4 cores. We can use this to check collisions in parallel. To do so, we can uniformly split a set of physical bodies into more small sets and do collision checks in different threads – each thread will use its own set of bodies. Such easy approach can speed up physics up to N times, where N – cores count of your processor. But I should mention that in real world you can speed up physics in (0.85N) times.

The game also have more specific optimizations: we can go further and divide terrain cell into set of smaller cells and each sub-cell will have pixels that are stored linearly in memory – this sort of optimization aimed to make calculations more cache-friendly.

That’s all for now, I would like to know if you interested in such kind of articles with technical details. Thanks for reading!