A few words about the Cellular Automaton

Robin Glattauer

September 1, 2011

Abstract

I documented my code with doxygen, but of course this a good way to document code and not to document an idea. So for getting the overall picture of it, I wrote this short introduction. It should serve as a quick guide, telling what a cellular automaton is, why I use it and how I designed my classes.

1 What is a Cellular Automaton

It is a set of discrete entities, which can have discrete states and change their states depending on their environment. Let's give an example for this, because the last sentence is horrid: the maybe best example for it are cells (tada). Which kinda is the basic idea behind it. Cells are discrete because there is always one cell or two or seven cells, but never 2.3 cells. And we assign them discrete states like: "living" or "dead" or others.

The idea now is, that with every timestep the situation evolves. So when a cell for example finds that there is space to expand, it might create new cells around itself. And when there's no space, it might just do nothing. And when it's surrounded by dead cells it might think "oh, what the heck" and go for suicide too.

So how the cells are evolving, what their states are and so on, depends completely on the implementation. A word said here: No, my Cellular Automaton is not capable of doing that. It is designed for pattern recognition in track search.

2 The Cellular Automaton for Pattern Recognition

So far it sounds like the Cellular Automaton would be an interesting tool to simulate cells or population growth, but that's not, what I'm using it for.

What is a pattern? As hard as it is to define "pattern" exactly, what we might say is, that a pattern follows some rules: like the pattern of tiles on a chess board will always follow the rule, that there is a black tile next to a white tile. And a cellular automaton has rules too: there are the rules how to change the state. So if one implements rules in a cellular automaton that resemble the rules of a pattern one wants to find and then lets cells following this pattern survive and others die, at the end we are left with living cells forming the pattern.

2.1 Track Search

What I want to do is Track Search in the forward direction. So essentially I have a lot of spacepoints stemming from a track and need to find out how they are connected. I have 7 layers of discs (all normal to the z axis) in the forward direction I can use. And there is a magnetic field in z direction, so all the charged particles, that cause the hits will run on helix tracks. As a helix is well defined with 3 points, I could gather all triplets of possible points and make helixes and then try to pick up other points from other layers. Usually I would do this using a Kalman Filter, but this means a lot of use of a maybe time consuming Kalman Filter at an early stage.

INSERT: picture of this

If a segment has another one connected to its inner point with the same state and the angle between them is not too big its state is raised by 1. This is repeated again and again. As you can see the innermost segments will always stay at state 0. the next generation of segments can reach state 1 (if they have another segment connected to it and the angle is right) and so on.

So at the end we will have segments with different states. When a segment in the highest generation has a neighbor which itself has one and so on, i.e. it is connected until the innermost segment, it could be a track and is therefore kept. This is kinda logical: a real track has to reach it's point of origin and needs to follow a certain path (which is not too curly, as we don't want those tracks).

Of course a real track may trick us:

- There can be layers left out, because of multiple reasons (hardware doesn't always get 100%, the geometry, etc.).
- There might be multiple scattering
-

And it is therefore important to take care of these problems and be careful.