

Introduction to CompuCell3D



Maciej Swat
Biocomplexity Institute
Indiana University
Bloomington, IN 47405
USA



Developing Multi-Scale, Multicell Developmental and Biomedical
Simulations with CompuCell3D
Bloomington, 2011

IU Team: Susan Hester, **Julio Belmonte**, Abbas Shirinifard, Ryan Roper, Alin Comanescu, Benjamin Zaitlen, **Randy Heiland**, Dr. Dragos Amarie, Dr. Scott Gens, **Dr. James A. Glazier**, Dr. James Sluka, Dr. Sherry Clendenon, **Dr. Mitja Hmeljak**, Dr. Srividhya Jayaraman, Dr. Gilberto Thomas

Support: EPA, NIH/NIGMS, NAKFI, Indiana University

What will you learn during the workshop?

1. What is CompuCell3D?
2. Why use CompuCell3D?
3. Demo simulations
4. Glazier-Graner-Hogeweg (GGH) model – review
5. CompuCell3D architecture and terminology
6. XML 101. CC3DML-intro
7. Building Your First CompuCell3D simulation
8. Visualization – CompuCell Player
9. Python scripting in CompuCell3D
10. Building C++ CompuCell3D extension modules – for interested participants

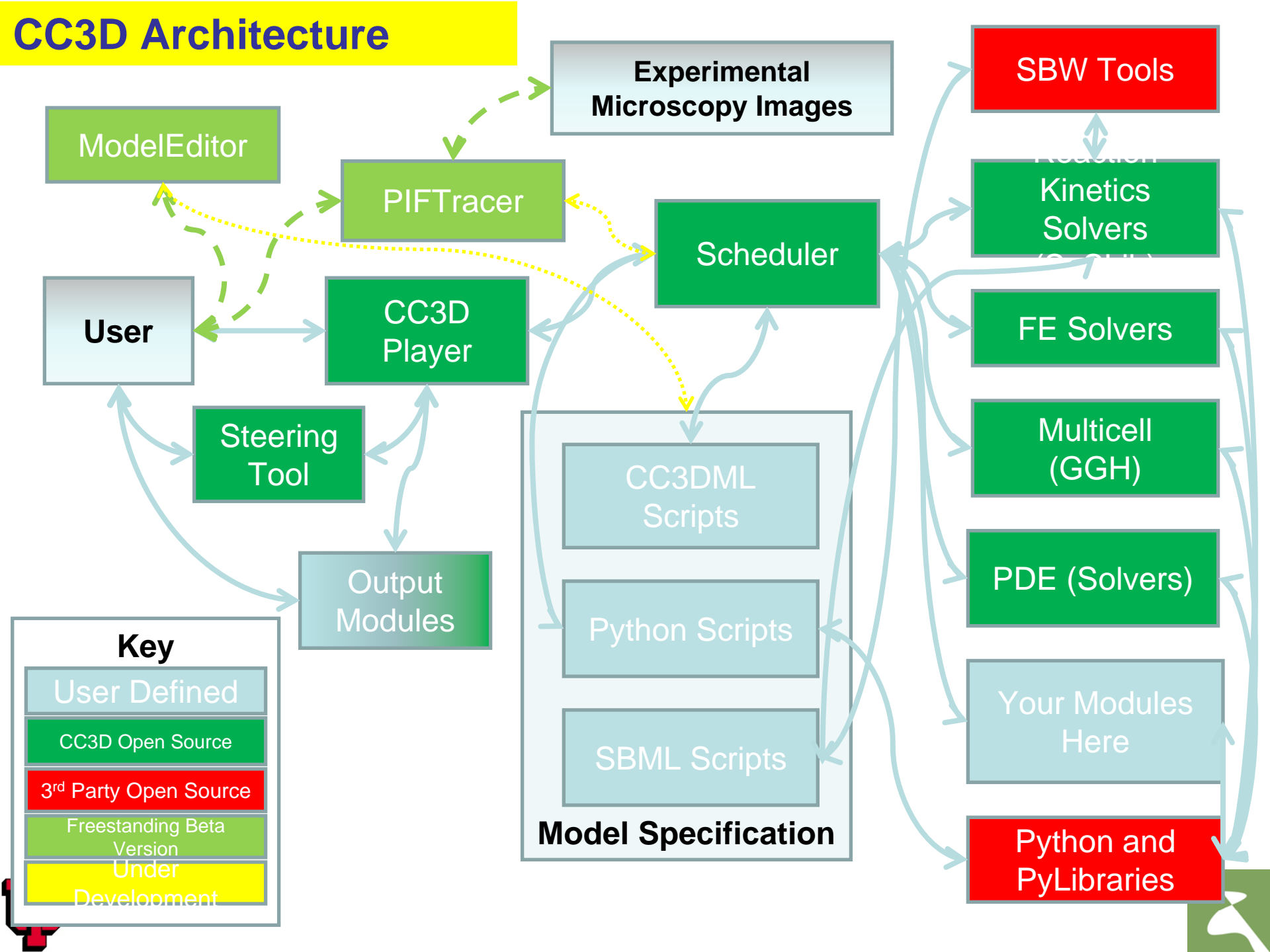


What Is CompuCell3D?

1. CompuCell3D is a modeling environment to build, test, run and visualize multiscale, multi-cell GGH-based simulations.
2. CompuCell3D has built-in standard scripting language (Python) that allows users to quite easily write extension modules that are essential for building sophisticated biological models.
3. CompuCell3D thus is NOT a hard-coded simulation of a specific biological system.
4. Running CompuCell3D simulations DOES NOT require recompilation.
5. CompuCell3D models described using CompuCell3D XML and Python script(s).
6. CompuCell3D platform is distributed with a GUI front end – CompuCell Player for 2- Dand 3-D visualization and simulation replay.
7. CompuCell3D provides a specialized model editor (Twedit) and initial condition generator (PifTracer).
8. CompuCell3D is a cross platform application that runs on Linux/Unix, Windows, Mac OSX.
9. CompuCell3D simulations can be easily shared and combined.



CC3D Architecture



Why Use CompuCell3D? What Are the Alternatives?

1. CompuCell3D allows users to set up and run their simulations in minutes to hours rather than weeks to months for custom code.
2. Most CompuCell3D simulations **DO NOT** need to be recompiled. To change parameters (in XML or Python scripts) or logic (in Python scripts) you just make the changes in the script or on the fly and run. Recompilation of hard-coded simulation is error prone and is accessible only to users with significant programming background.
3. CompuCell3D is actively developed , maintained and supported. The www.compuCell3d.org website provides manuals, tutorials and developer documentation. CompuCell3D has approx. 4 releases each year .
4. CompuCell3D has many users around the world, facilitating collaboration and module exchange, saving time when developing new models.
5. The Biocomplexity Institute organizes training workshops and mentorship programs. Those are great opportunities to learn biological modeling using CompuCell3D. For more info see www.compuCell3d.org

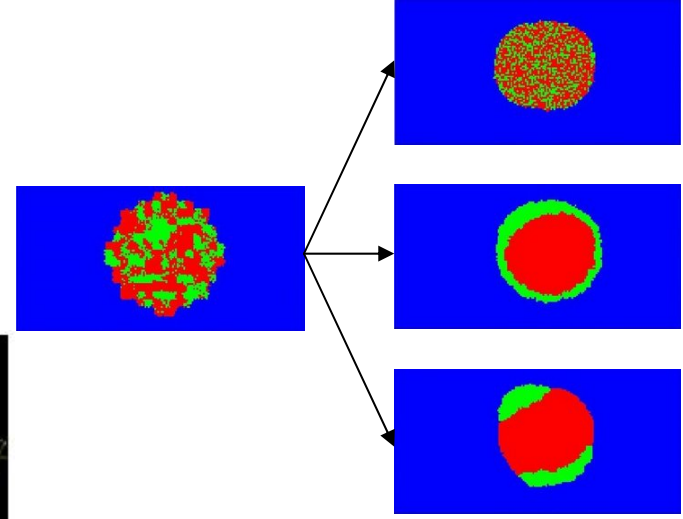
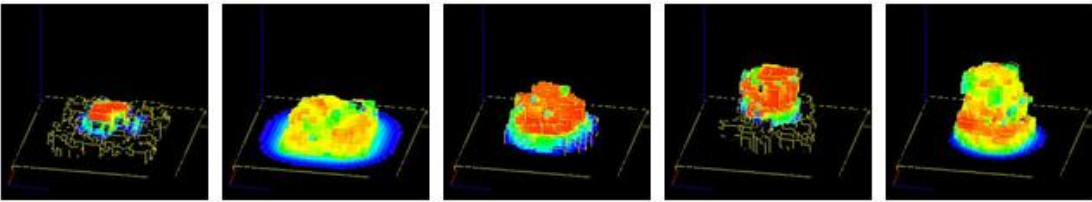
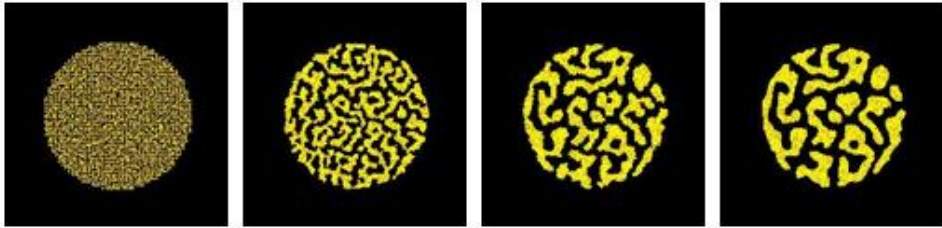
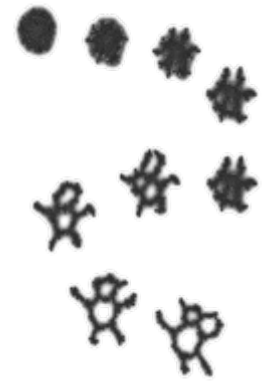
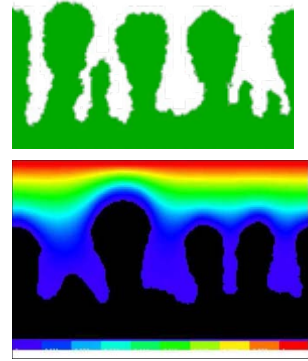
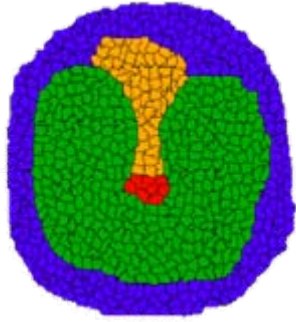


Why are model sharing and standards important?

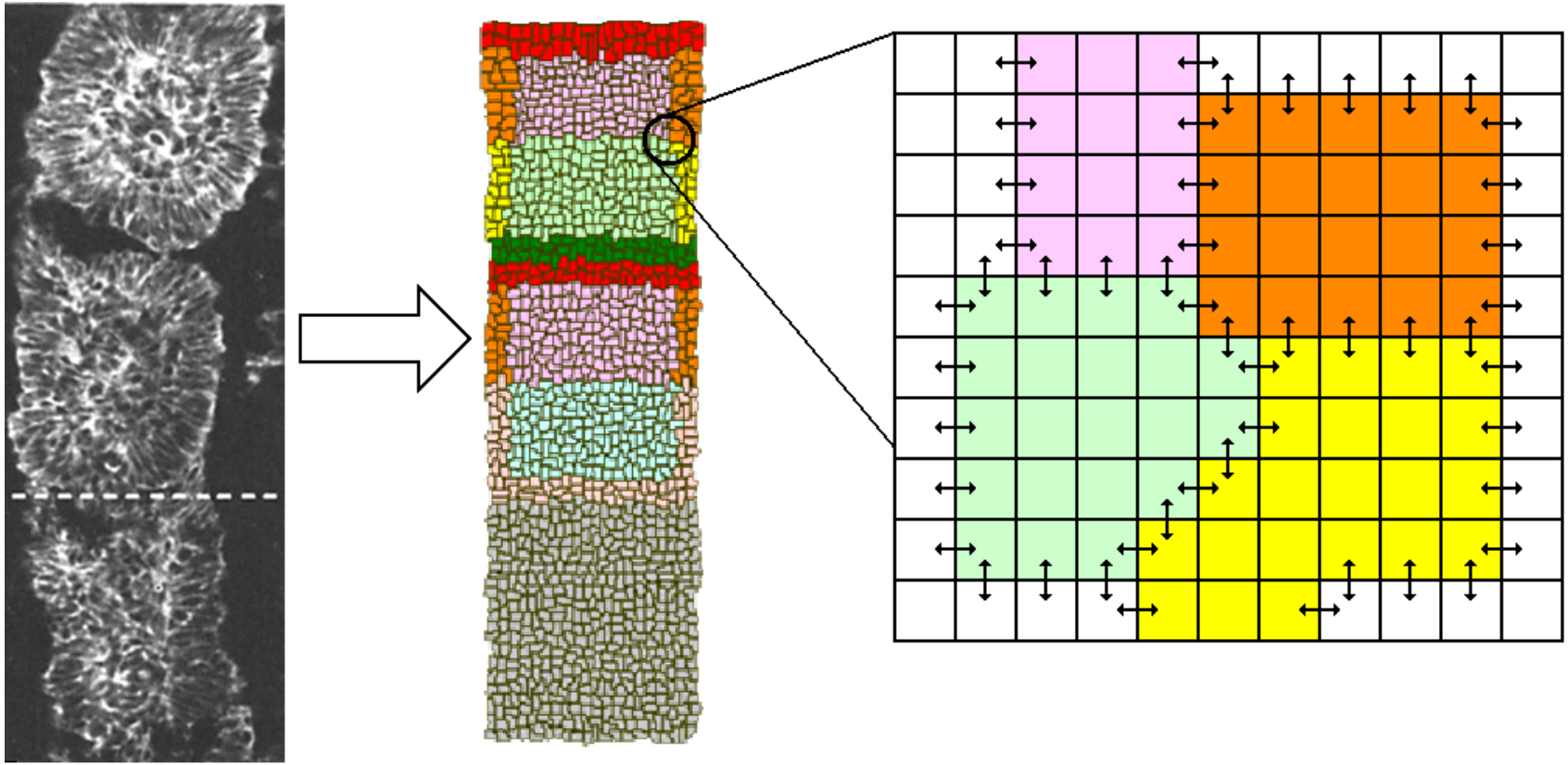
1. 99% of modeling done with custom written code is hard or impossible to reproduce or verify. In publications ,even ones including full code listings, authors often forget to describe details which are essential to reproducing their described work.
2. Using standard modeling tools improves the chances of your research being accepted and further refined by other scientists.
3. Standards allow people to spend more time working on new ideas and less struggling to reproduce old results .
4. Standards greatly improves research efficiency.
5. Bug tracking and detection are much more efficient with shared tools than with custom written ones. Bugs are also better documented for shared software.
6. Developing and sharing modules with other researchers is the best way to improve software modeling tools used by the research community.



Demo CompuCell3D Simulations



Review of the GGH Model



Key properties:

Cells live on a lattice.

Each cell occupies many lattice sites.

Each cell has a unique index.

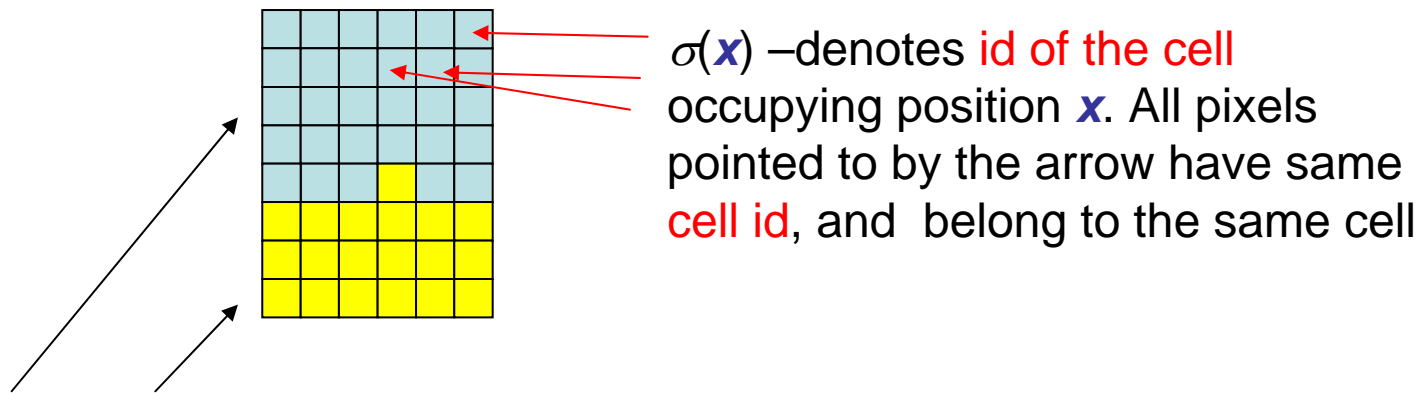
Each cell has a type—can have many cells of each type. *E.g.* a simple cell sorting simulation has many cells of type “**Condensing**” and many of type “**NonCondensing**”



The GGH Model Formalism Overview

- Configuration of Cells Evolves to Locally Minimize the Effective Energy, primarily by satisfying constraints) (Graner and Glazier, 1992)
- Key concept is differential adhesion between components: Contact energy depending on cell types (differentiated cells)

$$E = \sum_{\substack{\vec{x}, \vec{x}' \\ \text{neighbors}}} J(\tau(\sigma(\vec{x})), \tau(\sigma(\vec{x}')))(1 - \delta(\sigma(\vec{x}), \sigma(\vec{x}')) + \sum_{\sigma} \lambda_s(\sigma)(s(\sigma) - S_{\text{Target}}(\sigma))^2 + \sum_{\sigma} \lambda_v(\sigma)(v(\sigma) - V_{\text{Target}}(\sigma))^2 + E_{\text{chem}} + E_{\text{hapt}} + \dots$$



$\tau(\sigma(\mathbf{x}))$ denotes the **cell type** of cell with id $\sigma(\mathbf{x})$. In the picture above blue and yellow cells have **different cell types and different cell id**. Arrows mark different cell types



The GGH Model Formalism Overview—Dynamics

- To simulate the cytoskeleton-driven extension and retraction of cell membranes (including pseudopods, filopodia and lamellipodia). The GGH algorithm tries randomly to extend and retract cell boundaries one pixel at a time.
- At each attempt, it calculates the new configuration Effective Energy and accepts the new configuration according to the Metropolis algorithm: probability of configuration change

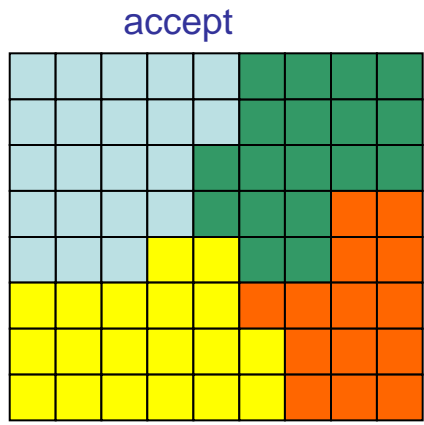
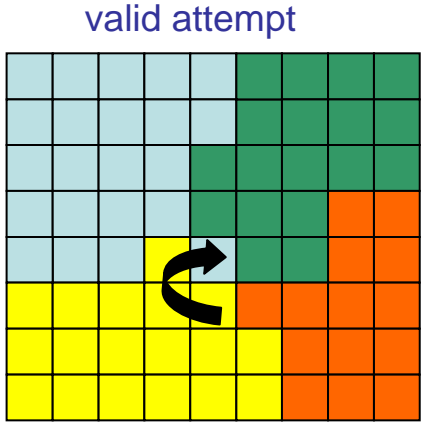
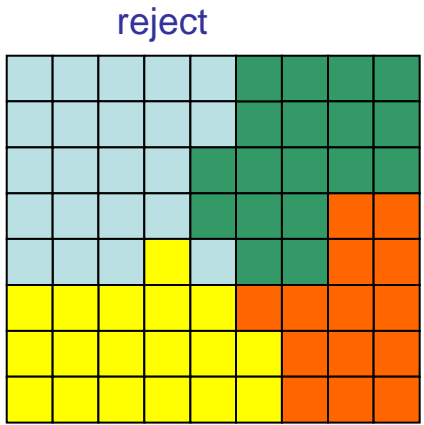
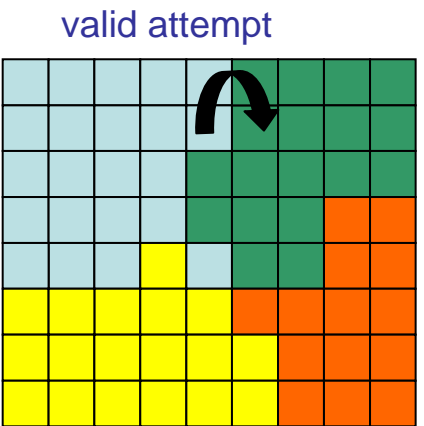
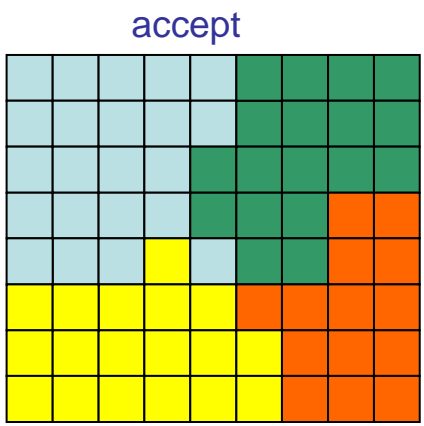
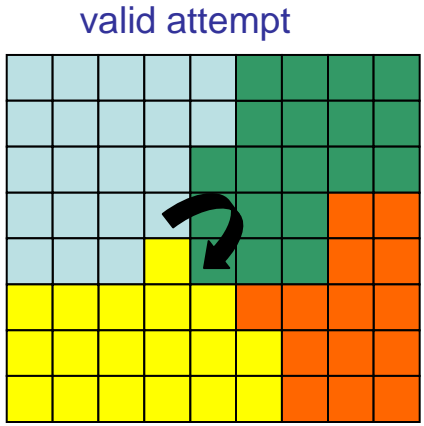
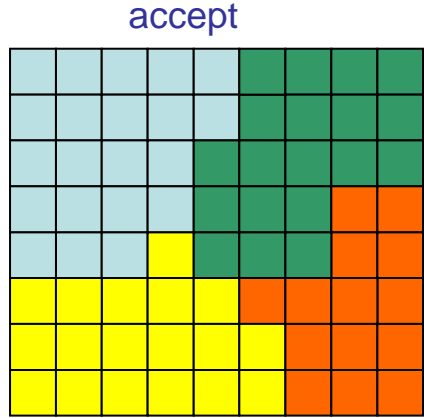
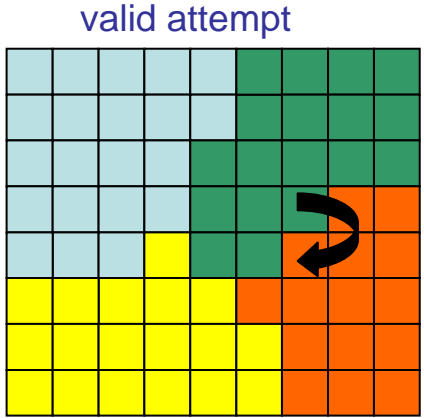
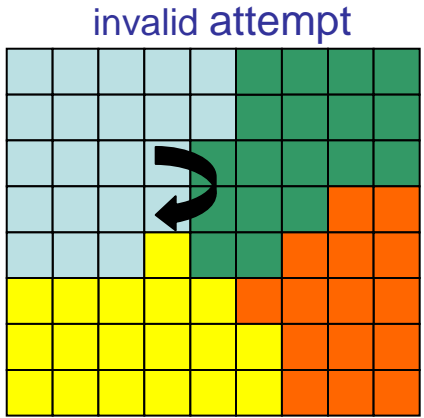
$$P(\Delta E) = 1, \Delta E \leq 0$$

$$P(\Delta E) = e^{-\Delta E/kT}, \Delta E > 0$$

- Result is movement with velocity proportional to the gradient of the Energy (or linear in the applied force).
- Configurations evolve to satisfy the constraints.
- When constraints conflict, evolve to balance errors.



More Detail on Pixel Copy Attempts



CompuCell3D Terminology and Relation to GGH

CompuCell has two basic time scales a **fast scale** and a **slow scale**:

Fast Scale:

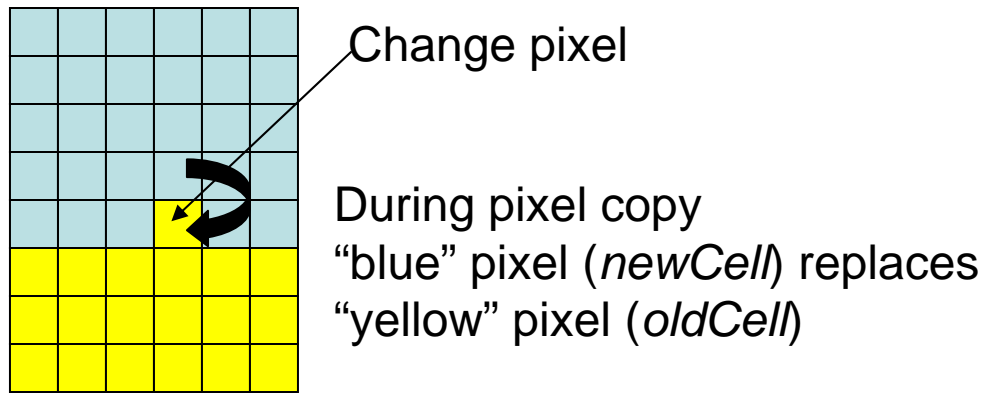
- A **Pixel-copy attempt** is an event where program randomly picks a lattice site and attempts to copy the pixel to a neighboring lattice site.
- CompuCell3D **Plugins** either calculate terms in the Effective Energy or implement actions in response to accepted pixel copies (**Lattice Monitors**). Most Plugins are coded in C++ for speed.

Slow Scale:

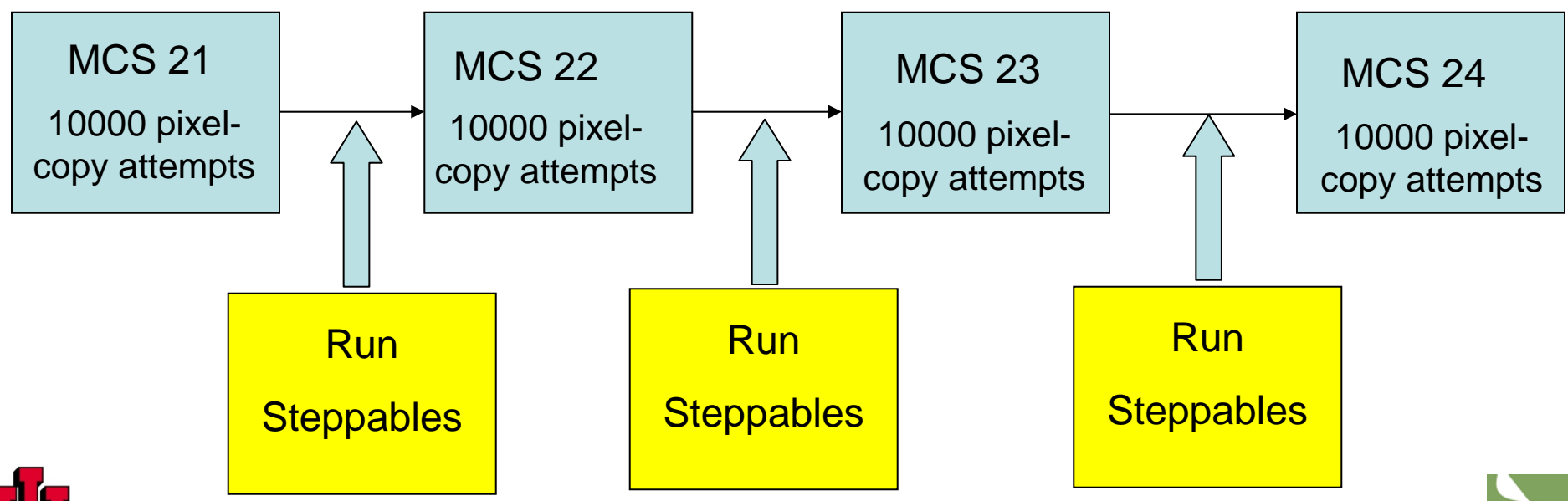
- A **Monte Carlo Step (MCS)** consists of a number of pixel-copy attempts equal to the number of lattice sites.
- CompuCell3D **Steppables** at the end of each MCS and at the beginning and end of simulations. Most customizations of CompuCell3D simulations use user-written Python Steppables



CompuCell3D Terminology – Visual Guide



100x100x1 square lattice = 10000 lattice sites (pixels)



Nearest neighbors in 2D and their Euclidian distances from the central pixel

| | | | | | |
|--|---|---|---|---|---|
| | | | | | |
| | 5 | 4 | 3 | 4 | 5 |
| | 4 | 2 | 1 | 2 | 4 |
| | 3 | 1 | ● | 1 | 3 |
| | 4 | 2 | 1 | 2 | 4 |
| | 5 | 4 | 3 | 4 | 5 |
| | | | | | |

| Nearest Neighbor Order | Number of nearest neighbors | Euclidian distance – square lattice |
|------------------------|-----------------------------|-------------------------------------|
| 1 | 4 | 1 |
| 2 | 4 | $\sqrt{2}$ |
| 3 | 4 | 2 |
| 4 | 8 | $\sqrt{5}$ |
| 5 | 4 | $\sqrt{8}$ |

Pixel copies could take place between any order neighbors.

In practice we use only the few first neighbors (1-4).

To specify a pixel-copy range of 2 in a simulation insert the CC3DML command :

`<NeighborOrder>2</NeighborOrder>`

In the `<Potts></Potts>` section of the simulation .

Contact energy calculations have their range specified separately

To specify an interaction range of 3 in a simulation insert the CC3DML command :

`<NeighborOrder>3</NeighborOrder>`

In the `<Plugin Name="Contact"> </Plugin>` section of the simulation .



Hexagonal Lattices

To reduce intrinsic lattice anisotropy of the square lattice, we can use a hexagonal lattice Instead.

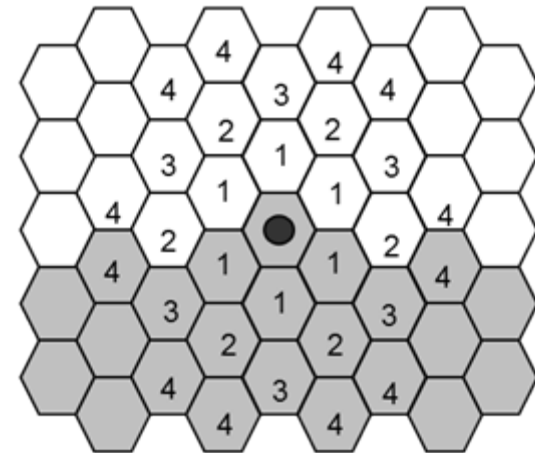
The area/volume of each pixel is fixed to 1, so the length scale changes when you move from square to hex lattices.

To specify a simulation on a hex lattice, insert the CC3DML command :

`<LatticeType>Hexagonal</LatticeType>`

In the `<Potts></Potts>` section of the simulation .

| | 2D Square Lattice | | 2D Hexagonal Lattice | |
|----------------|---------------------|--------------------|----------------------|----------------------|
| Neighbor Order | Number of Neighbors | Euclidian Distance | Number of Neighbors | Euclidian Distance |
| 1 | 4 | 1 | 6 | $\sqrt{2/\sqrt{3}}$ |
| 2 | 4 | $\sqrt{2}$ | 6 | $\sqrt{6/\sqrt{3}}$ |
| 3 | 4 | 2 | 6 | $\sqrt{8/\sqrt{3}}$ |
| 4 | 8 | $\sqrt{5}$ | 12 | $\sqrt{14/\sqrt{3}}$ |



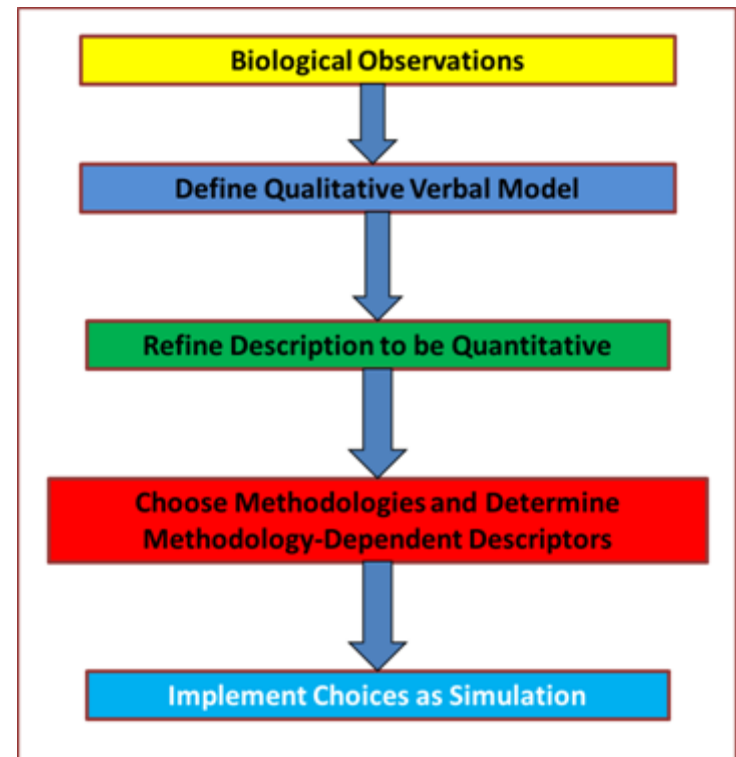
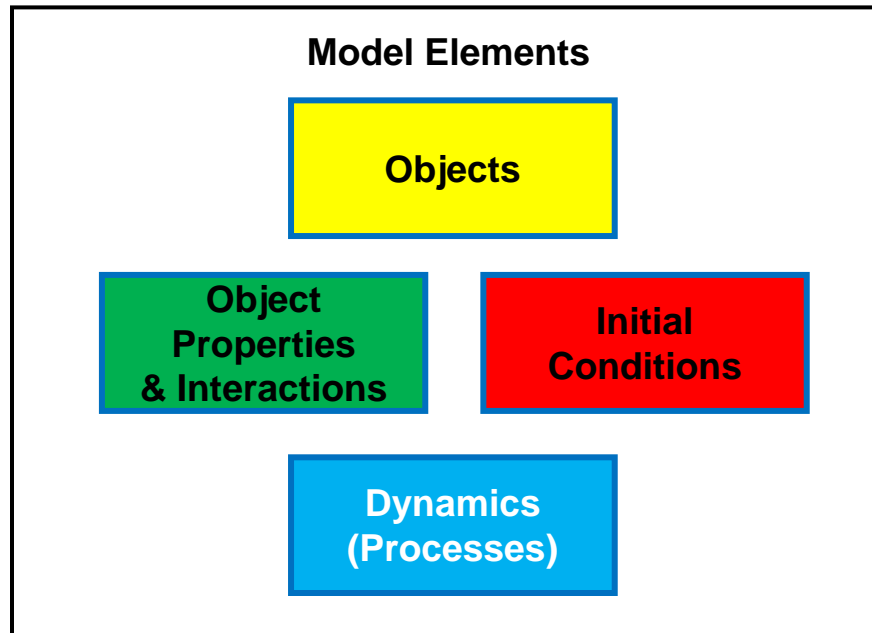
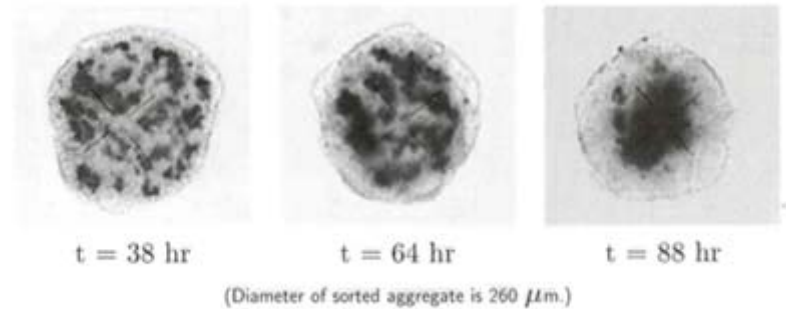
WARNING: a few functions may still not work properly for hex lattices.



Cell Sorting—The Simplest Model

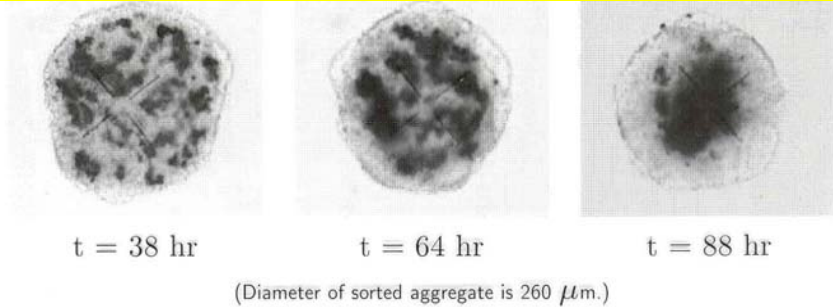
Simulate the evolution of a randomly mixed aggregate of two mesenchymal cell types due to Differential Adhesion and random cell motility.

Question—how does the outcome depend on the relative adhesion energies between the cell types and between the cells and medium?

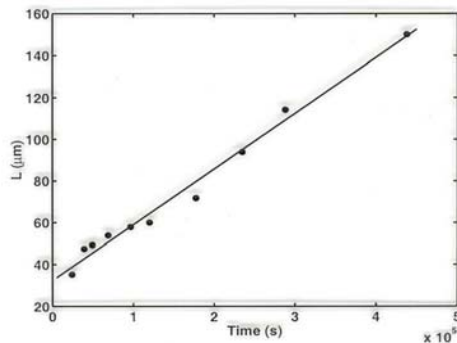


Cell Sorting—The Simplest Model

Biological Observations



For large volume fractions $\rightarrow L \sim (\sigma/\eta)t$



Slope = $2.7 \times 10^{-4} \mu\text{m s}^{-1}$

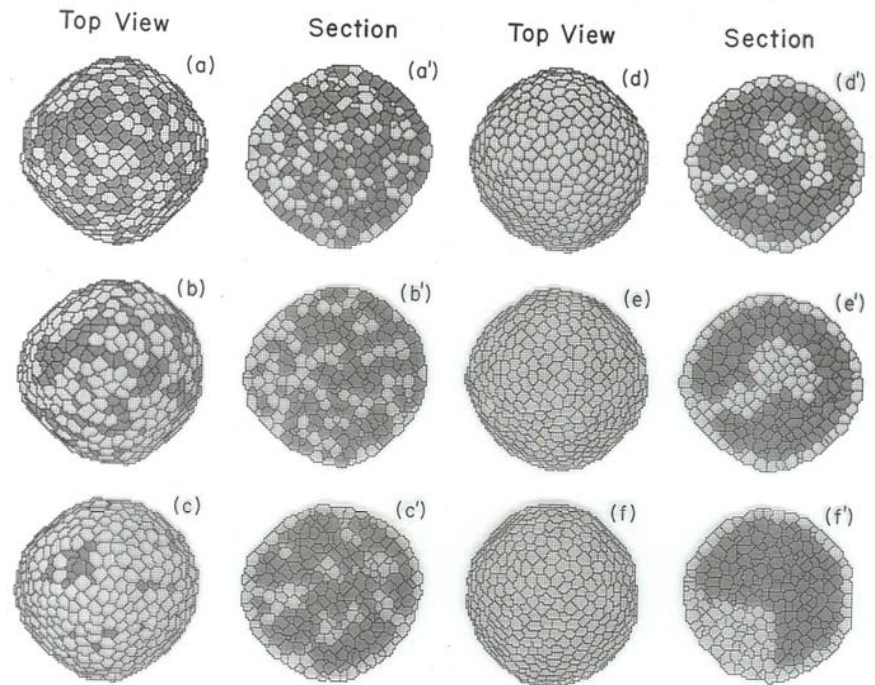
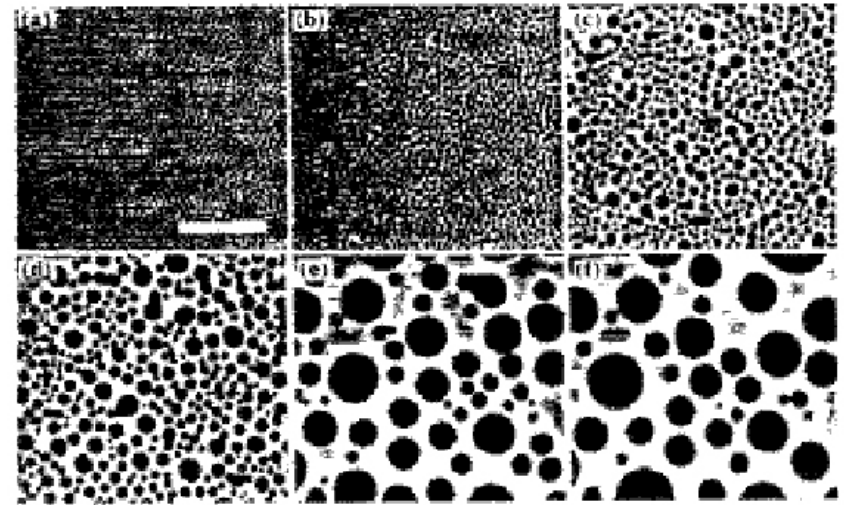


Figure 2.



Cell Sorting—The Simplest Model

Define Qualitative Verbal Model

- **Objects: Cells, Medium (Generalized Cell)**
- **Properties, Behaviors:**
 - Cells have Fixed Volumes and Fixed Membrane Areas
 - Medium has Unconstrained Volume and Surface Area
 - Cells are Adhesive
 - Cells have Intrinsic Random Motility
- **Interactions:**
 - Cells Adhere to each other and to Medium with an Energy/Area which Depends on Cell Type (simulating different types or densities of cadherins on each Cell Type)
- **Dynamics:**
 - Standard Potts Dynamics
- **Initial Conditions:**
 - Cells in a Blob Surrounded by Medium
 - In Blob, Cells Randomly Mixed



Cell Sorting—The Simplest Model

Refine Description to be Quantitative

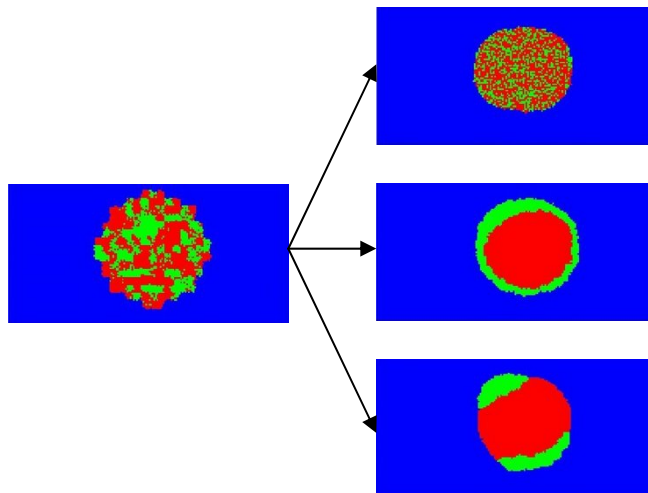
Three Cell Types: More Cohesive, Less Cohesive, Medium

$$H = \sum_{\substack{\vec{i}, \vec{i}' \\ \text{neighbors}}} J(\tau(\sigma(\vec{i})), \tau(\sigma(\vec{i}')) \{1 - \delta(\sigma(\vec{i}), \sigma(\vec{i}'))\} +$$

$$+ \sum_{\sigma} \lambda_{\text{volume}} (V(\sigma) - V_{\text{target}})^2$$

Random Blob Initial Conditions or
Adjacent Domains

Outcome Depends on J_s



| | |
|---|--|
| Initial configuration | |
| Cell Sorting $J_{\text{white,white}} = J_{\text{grey,grey}} < J_{\text{white,grey}}$ | |
| Cell Mixing $J_{\text{white,white}} = J_{\text{grey,grey}} > J_{\text{white,grey}}$ | |
| Engulfment $J_{\text{white,grey}} < J_{\text{white,medium}}$ $J_{\text{grey,medium}} < J_{\text{white,medium}}$ | |
| No cell cell adhesion $J_{\text{cell,cell}} > 2J_{\text{cell,medium}}$ | |



Describing CompuCell3D simulations

- Simulations are usually described using XML-based CC3DML and Python.
- For simple simulations CC3DML is sufficient. For more sophisticated ones you DO NEED Python.
- CompuCell3D distributions include many examples which you may use as a starting point for your simulations.
- Twedit++-CC3D allows users to autogenerate complex simulations within few seconds. We will use Twedit++-CC3D throughout the workshop.



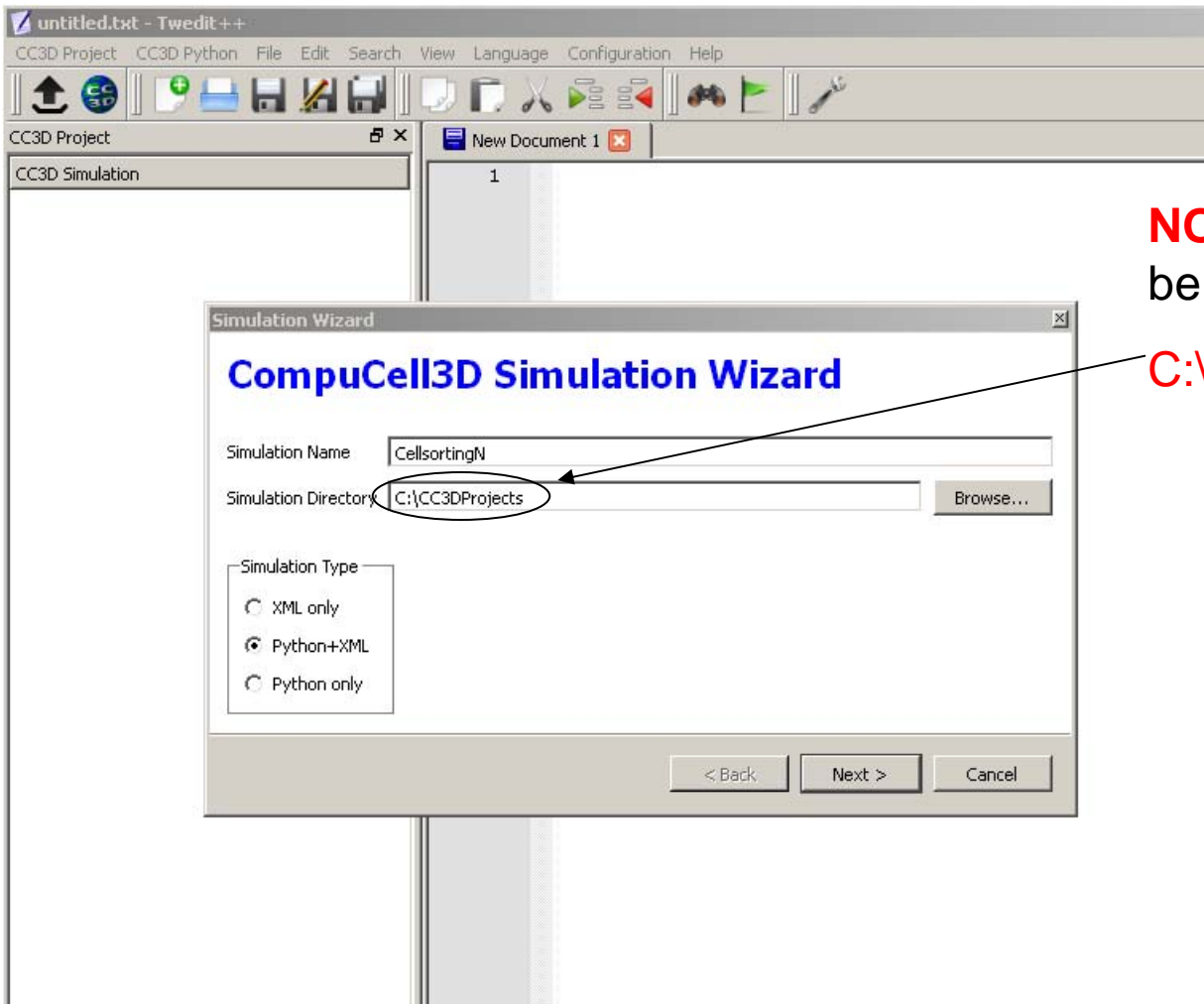
Using Twedit++-CC3D to Autogenerate Simulation Code Based on Top-Level Specifications

- **Specify basic simulation properties such as lattice dimension, cell membrane fluctuation amplitude , initial conditions etc...**
- **List all cell types**
- **List chemical fields (if any)**
- **Choose cellular behaviors and constraints**



Using Twedit++-CC3D part 1

- From CC3D Project menu select New Simulation Wizard...
- Type name of the simulation and choose languages which will describe – default choice is fine

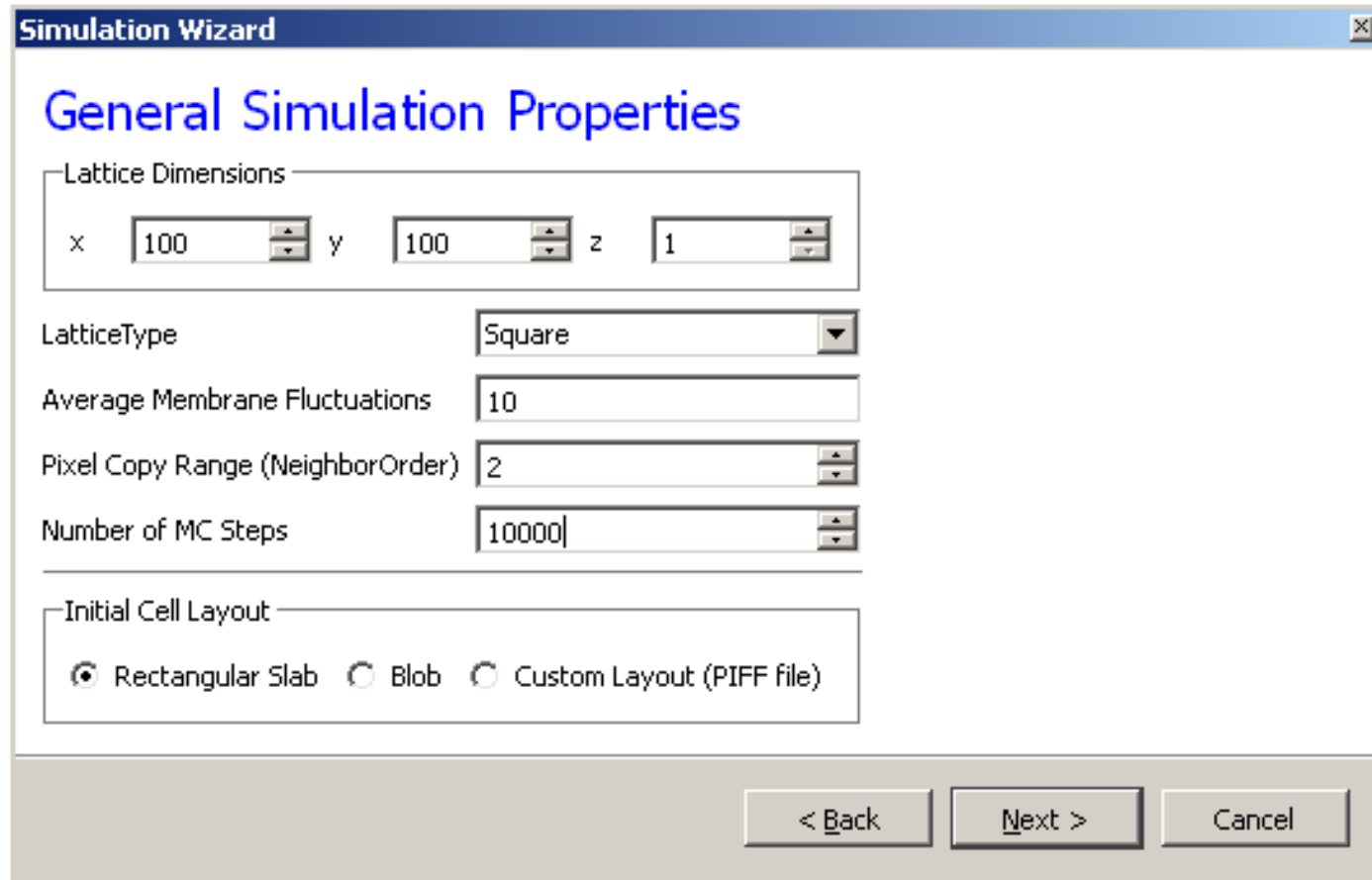


NOTICE: The simulation will be stored in

C:\CC3DProjects\Cellsorting

Using Twedit++-CC3D part 2

- Specify lattice dimensions, cell motility, number of MCS, initial conditions, lattice type, pixel copy distance



The image shows a screenshot of a software dialog box titled "Simulation Wizard". The main heading is "General Simulation Properties".

Lattice Dimensions

x: 100 y: 100 z: 1

LatticeType: Square

Average Membrane Fluctuations: 10

Pixel Copy Range (NeighborOrder): 2

Number of MC Steps: 10000

Initial Cell Layout

Rectangular Slab Blob Custom Layout (PIFF file)

Navigation buttons: < Back, Next >, Cancel



Using Twedit++-CC3D part 3

- List cell types

Simulation Wizard

Cell Type Specification

| | Cell Type | Freeze |
|---|-----------|--------------------------|
| 1 | Medium | <input type="checkbox"/> |
| 2 | Light | <input type="checkbox"/> |
| 3 | Dark | <input type="checkbox"/> |
| | | |

Clear Table

Cell Type Freeze

< Back Next > Cancel



Using Twedit++-CC3D part 4

- Choose cell behaviors and constraints. For cell sorting simulation we chose adhesive behaviors (implemented in the Contact module) and cell volume constraint (implemented in VolumeFlex module)

Simulation Wizard

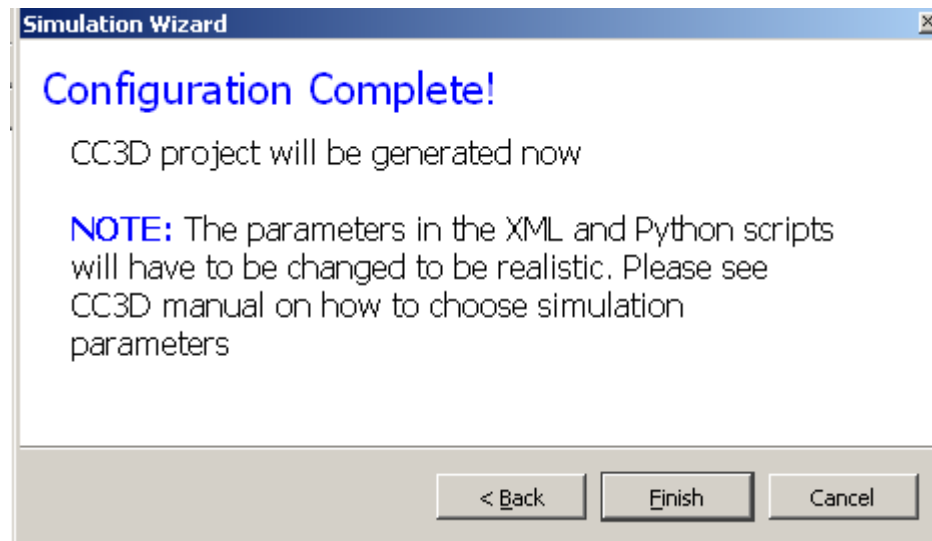
Cell Properties and Behaviors

| Cellular Behaviors | Constraints and Forces | Cellular Property Trackers |
|--|--|--|
| Adhesion <input checked="" type="checkbox"/> Contact <input type="checkbox"/> ContactInternal <input type="checkbox"/> AdhesionFlex <input type="checkbox"/> ContactLocalProduct <input type="checkbox"/> Compartments <input type="checkbox"/> FocalPointPlasticity <input type="checkbox"/> Elasticity <input checked="" type="checkbox"/> ContactMultiCad (deprecated) | Volume <input checked="" type="checkbox"/> volumeFlex <input type="checkbox"/> volumeLocalFlex Surface <input type="checkbox"/> SurfaceFlex <input type="checkbox"/> SurfaceLocalFlex Ext. Force <input type="checkbox"/> ExternalPotential <input type="checkbox"/> ExternalPotentialLocalFlex | <input checked="" type="checkbox"/> Center Of Mass <input type="checkbox"/> Cell Neighbors <input type="checkbox"/> Moment Of Inertia <input type="checkbox"/> Cell Pixel Tracker <input type="checkbox"/> Cell Boundary Pixel Tracker |
| Chemotaxis <input type="checkbox"/> Chemotaxis | Connectivity <input type="checkbox"/> Global (2D/3D) <input type="checkbox"/> Global (by cell id) <input type="checkbox"/> Fast (2D square lattice) | Aux. Modules <input type="checkbox"/> BoxWatcher <input type="checkbox"/> PIFDumper |
| Secretion <input type="checkbox"/> Secretion | Elongation <input type="checkbox"/> LenghtConstraint <input type="checkbox"/> LenghtConstraintLocalFlex (2D) | |
| Growth <input type="checkbox"/> Growth (Python) | | |
| Mitosis <input type="checkbox"/> Mitosis (Python) | | |
| Death <input type="checkbox"/> Death (Python) | | |



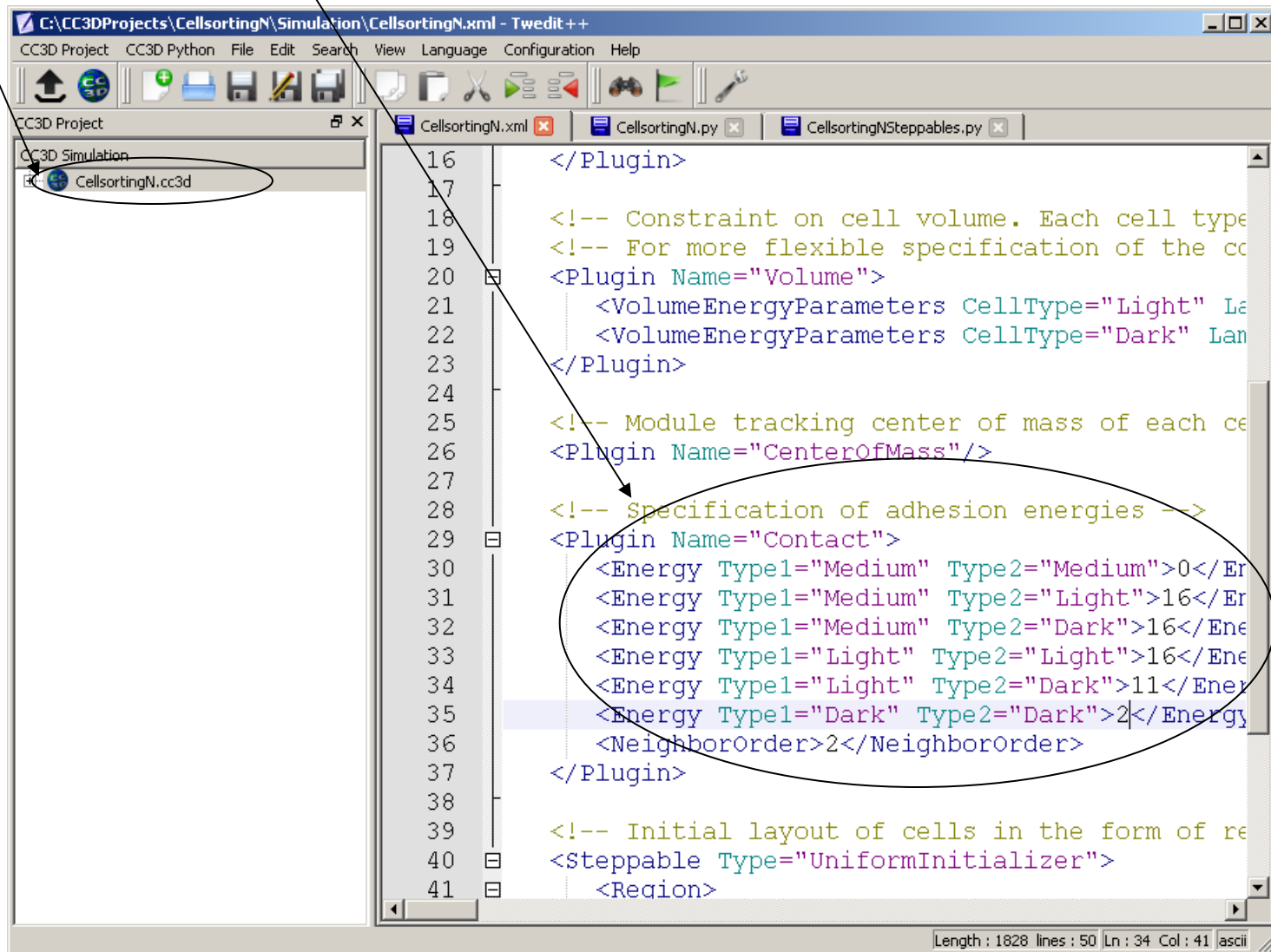
Using Twedit++-CC3D part 5

- Go to the last Wizard screen and click Finish. The simulation code will be generated. Now we have to manually edit parameters...



Using Twedit++-CC3D part 6

- Double click on project icon in the left panel to open simulation scripts. Go to Cellsorting.xml to fine tune cellular behaviors.

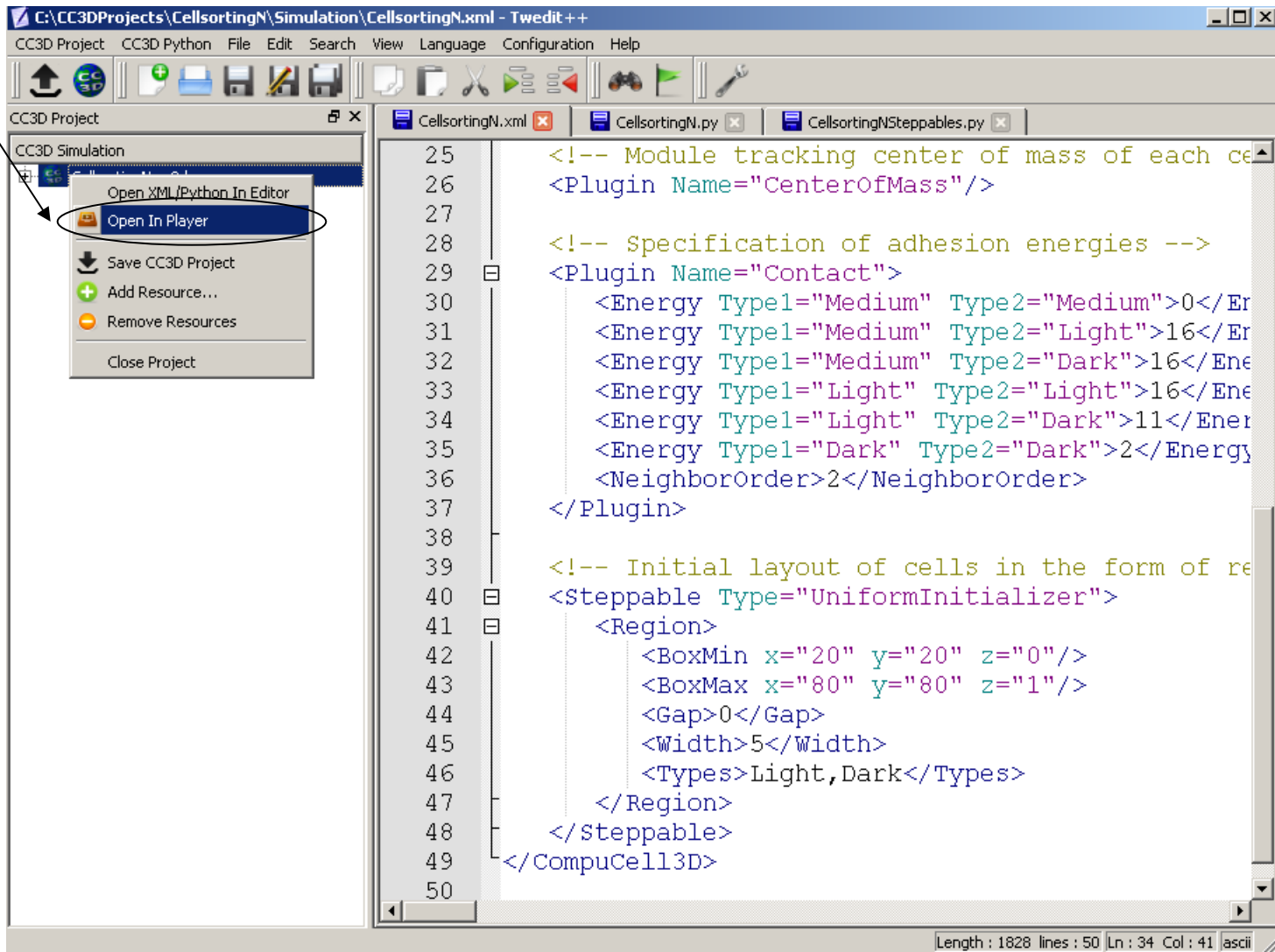


```
16 </Plugin>
17
18 <!-- Constraint on cell volume. Each cell type
19 <!-- For more flexible specification of the cc
20 <Plugin Name="Volume">
21     <VolumeEnergyParameters CellType="Light" La
22     <VolumeEnergyParameters CellType="Dark" Lan
23 </Plugin>
24
25 <!-- Module tracking center of mass of each ce
26 <Plugin Name="CenterOfMass"/>
27
28 <!-- Specification of adhesion energies -->
29 <Plugin Name="Contact">
30     <Energy Type1="Medium" Type2="Medium">0</Er
31     <Energy Type1="Medium" Type2="Light">16</Er
32     <Energy Type1="Medium" Type2="Dark">16</Ene
33     <Energy Type1="Light" Type2="Light">16</Ene
34     <Energy Type1="Light" Type2="Dark">11</Ene
35     <Energy Type1="Dark" Type2="Dark">2</Energy
36     <NeighborOrder>2</NeighborOrder>
37 </Plugin>
38
39 <!-- Initial layout of cells in the form of re
40 <Steppable Type="UniformInitializer">
41     <Region>
```



Using Twedit++-CC3D part 7

- To run generated simulation – right-click on project icon and choose “Open in Player”.



The screenshot shows the Twedit++ application window. The title bar reads "C:\CC3DProjects\CellsortingN\Simulation\CellsortingN.xml - Twedit++". The menu bar includes "CC3D Project", "CC3D Python", "File", "Edit", "Search", "View", "Language", "Configuration", and "Help". The toolbar contains various icons for file operations and simulation control. The left sidebar shows a tree view with "CC3D Project" and "CC3D Simulation". A context menu is open over the "CC3D Simulation" icon, with "Open In Player" highlighted. The main editor displays XML code for a simulation configuration, including plugins for "CenterOfMass", "Contact", and "UniformInitializer".

```
25 <!-- Module tracking center of mass of each cell -->
26 <Plugin Name="CenterOfMass"/>
27
28 <!-- Specification of adhesion energies -->
29 <Plugin Name="Contact">
30   <Energy Type1="Medium" Type2="Medium">0</Energy>
31   <Energy Type1="Medium" Type2="Light">16</Energy>
32   <Energy Type1="Medium" Type2="Dark">16</Energy>
33   <Energy Type1="Light" Type2="Light">16</Energy>
34   <Energy Type1="Light" Type2="Dark">11</Energy>
35   <Energy Type1="Dark" Type2="Dark">2</Energy>
36   <NeighborOrder>2</NeighborOrder>
37 </Plugin>
38
39 <!-- Initial layout of cells in the form of regions -->
40 <Steppable Type="UniformInitializer">
41   <Region>
42     <BoxMin x="20" y="20" z="0"/>
43     <BoxMax x="80" y="80" z="1"/>
44     <Gap>0</Gap>
45     <Width>5</Width>
46     <Types>Light,Dark</Types>
47   </Region>
48 </Steppable>
49 </CompuCell3D>
50
```

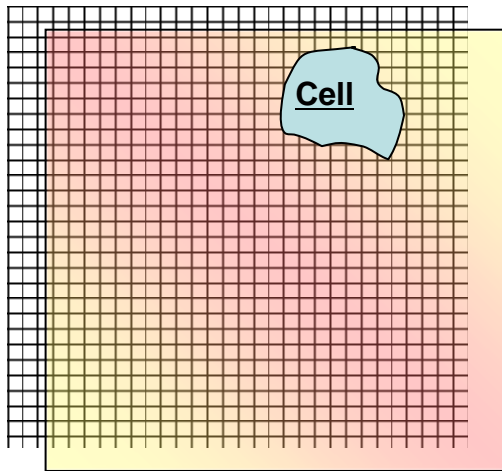
Length : 1828 lines : 50 Ln : 34 Col : 41 | ascii



Cell Sorting - walk through the code

We define Simulation using a script written in CompuCell3D Markup Language (CC3DML)

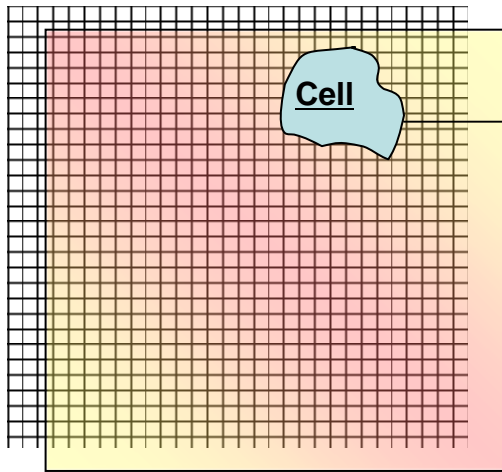
First: Define Lattice and Simulation Dynamics Parameters



```
< CompuCell3D>  
<Potts>  
  <Dimensions x="100" y="100" z="1"/>  
  <Steps>10000</Steps>  
  <Temperature>2</Temperature>  
</Potts>  
...  
</CompuCell3D>
```



Next, List all the Objects (here only cell types) in the simulation



```
<Plugin Name="CellType">  
  <CellType TypeName="Medium" TypeId="0"/>  
  <CellType TypeName="Light" TypeId="1"/>  
  <CellType TypeName="Dark" TypeId="2"/>  
</Plugin>
```

Note that Medium has TypeId =0. This is a REQUIREMENT in CompuCell3D.



TypeIds must be consecutive integers.



Cell Sorting—The Simplest Model

Implement Choices as Simulation

List object properties, behaviors and interactions

Volume

volume
volumeEnergy(cell)

```
<Plugin Name="Volume">  
<TargetVolume>25</TargetVolume>  
<LambdaVolume>1.0</LambdaVolume>  
</Plugin>
```



Cell

Contact

*contactEnergy(
cell1, cell2)*

```
<Plugin Name="Contact">  
  <Energy Type1="Medium" Type2="Medium">0  
  </Energy>  
  <Energy Type1="Light" Type2="Medium">16  
  </Energy>  
  <Energy Type1="Dark" Type2="Medium">16  
  </Energy>  
  <Energy Type1="Light" Type2="Light">16.0  
  </Energy>  
  <Energy Type1="Dark" Type2="Dark">2.0  
  </Energy>  
  <Energy Type1="Light" Type2="Dark">11.0  
  </Energy>  
</Plugin>
```



Mapping of CC3DML Syntax to Volume Constraint

$$E = \dots + \lambda^{volume} (v - V)^2 + \dots$$

```
<Plugin Name="Volume" >  
<TargetVolume>25</TargetVolume>  
<LambdaVolume>1.0</LambdaVolume>  
</Plugin>
```

Specifying Volume constraint for each cell type:

$$E = \dots + \lambda_{\tau}^{volume} (v_{\tau} - V_{\tau})^2 + \dots$$

```
<Plugin Name="Volume">  
  <VolumeEnergyParameters CellType="Light" LambdaVolume="2.0" TargetVolume="25"/>  
  <VolumeEnergyParameters CellType="Dark" LambdaVolume="2.0" TargetVolume="25"/>  
</Plugin>
```



Mapping of CC3DML Syntax to Surface Constraint

$$E = \dots + \lambda^{surface} (s - S)^2 + \dots$$

```
<Plugin Name="Surface" >  
<TargetSurface>25</TargetSurface>  
<LambdaSurface>1.0</LambdaSurface>  
</Plugin>
```

Specifying Surface constraint for each cell type:

$$E = \dots + \lambda_{\tau}^{surface} (s_{\tau} - S_{\tau})^2 + \dots$$

```
<Plugin Name="Surface">  
  <SurfaceEnergyParameters CellType="Light" LambdaSurface="2.0" TargetSurface="25"/>  
  <SurfaceEnergyParameters CellType="Dark" LambdaSurface="2.0" TargetSurface="25"/>  
</Plugin>
```



Mapping of CC3DML Syntax to Contact Energy Equation

$$E = \dots + \sum_{x,x'} J_{\tau(\sigma(x)),\tau(\sigma(x'))} (1 - \delta_{\sigma(x),\sigma(x')}) + \dots$$

```
<Plugin Name="Contact">
  <Energy Type1="Medium" Type2="Medium">0
</Energy>
  <Energy Type1="Light" Type2="Medium">16.0
</Energy>
  <Energy Type1="Dark" Type2="Medium">16.0
</Energy>
  <Energy Type1="Light" Type2="Light">16
</Energy>
  <Energy Type1="Dark" Type2="Dark">2.0
</Energy>
  <Energy Type1="Light" Type2="Dark">11.0
</Energy>
</Plugin>
```

You must specify a Contact Energy between each pair of cell types.

Contact Energies can be negative

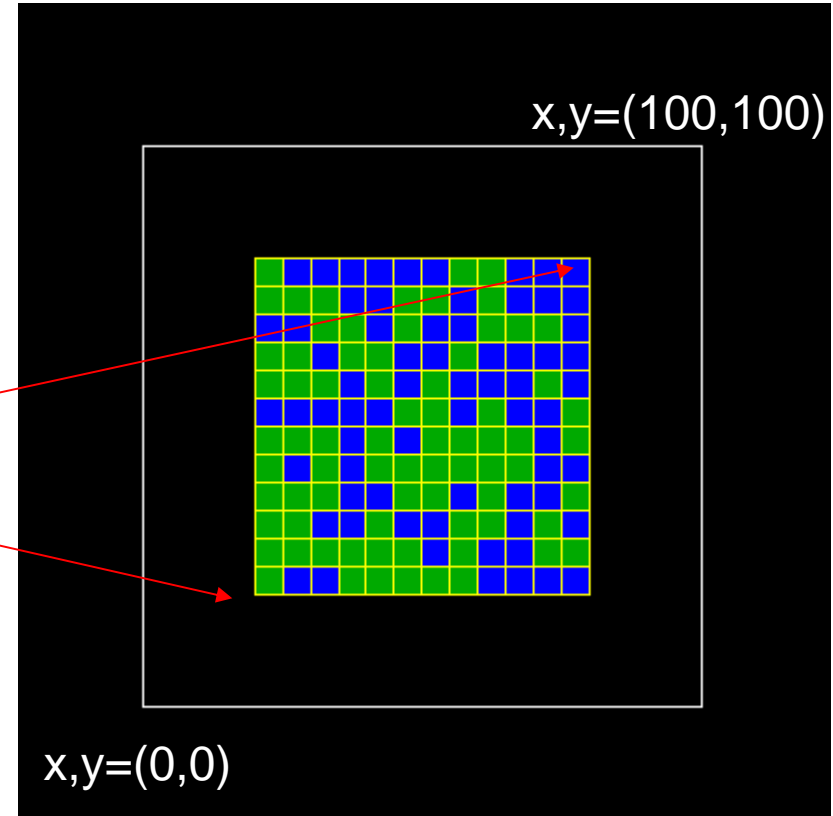
A smaller Contact Energy represents stronger adhesion



Define Initial Conditions (rectangular slab of cells)

Using built-in UniformInitializer Steppable:

```
<Steppable Type="UniformInitializer">  
  <Region>  
    <BoxMax x="80" y="80" z="1"/>  
    <BoxMin x="20" y="20" z="0"/>  
    <Gap>0</Gap>  
    <Width>5</Width>  
    <Types>Light,Dark</Types>  
  </Region>  
</Steppable>
```



CompuCell3D provides a number of ways to create initial cell configurations . UniformInitializer runs only once, at the beginning of a simulation and creates a rectangular slab of the specified cell types (Light and Dark).



Define Initial Conditions (rectangular slab of cells) details:

```
<Steppable Type="UniformInitializer">
```

```
<Region>
```

```
<BoxMax x="80" y="80" z="1"/>
```

```
<BoxMin x="20" y="20" z="0"/>
```

```
<Gap>0</Gap>
```

```
<Width>5</Width>
```

```
<Types>Light,Dark</Types>
```

```
</Region>
```

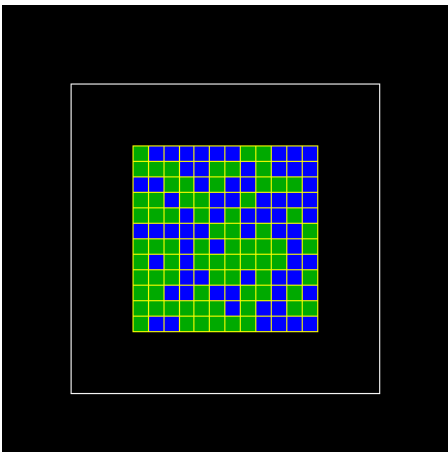
```
</Steppable>
```

Position of slab corners. Notice that in 2D max position of z coordinate has to be at 1 not zero. This is because 2D lattice is in fact 3D lattice with z dimension set to 1!

Separation between Adjacent Cells in Pixels (here 0)

Initial edge length of each square Cell

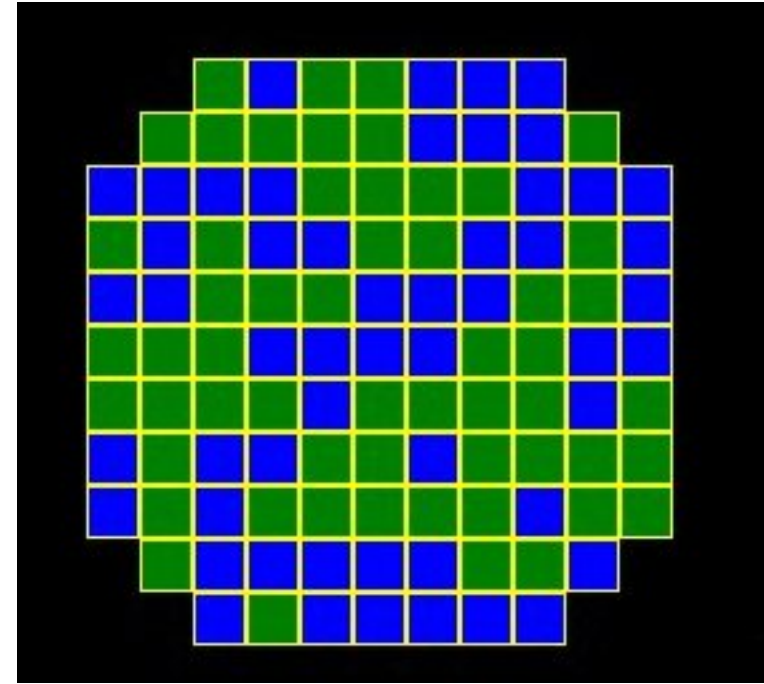
List of Cell Types to Include. If a Cell Type is repeated, the fraction of that Cell Type is proportional to the number of times it is listed



Define Initial Conditions (circular blob of cells)

Using built-in cell field BlobInitializer Steppable:

```
<Steppable Type="BlobInitializer">  
  <Region>  
    <Radius>30</Radius>  
    <Center x="40" y="40" z="0"/>  
    <Gap>0</Gap>  
    <Width>5</Width>  
    <Types>Dark,Light</Types>  
  </Region>  
</Steppable>
```

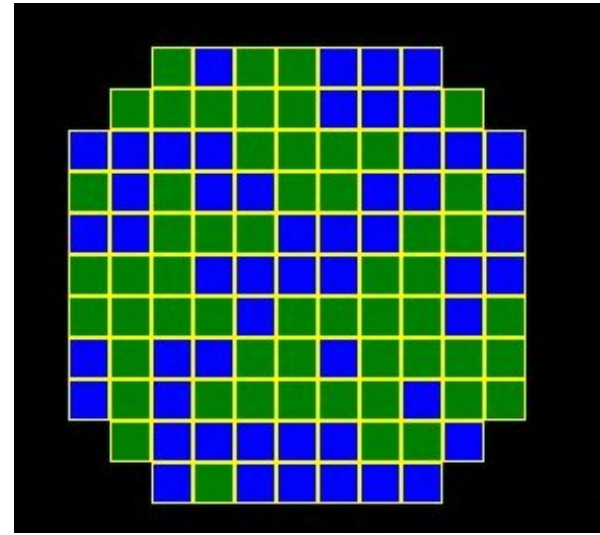


CompuCell3D provides a number of ways to create initial cell configurations . BlobInitializer runs only once, at the beginning of a simulation and creates a rough circle of the specified cell types.

NOTE: In the on-line code **Dark** cells are called **Condensing** and **Light** cells **NonCondensing**



Define Initial Conditions (circular blob of cells) details:



```
<Steppable Type="BlobInitializer">  
  <Region>  
    <Radius>30</Radius>  
    <Center x="40" y="40" z="0"/>  
    <Gap>0</Gap>  
    <Width>5</Width>  
    <Types>Dark, Light</Types>  
  </Region>  
</Steppable>
```

Radius of Disk of Cells

Position of Center of Disk of Cells

Separation between Adjacent Cells in Pixels (here 0)

Initial edge length of each square Cell

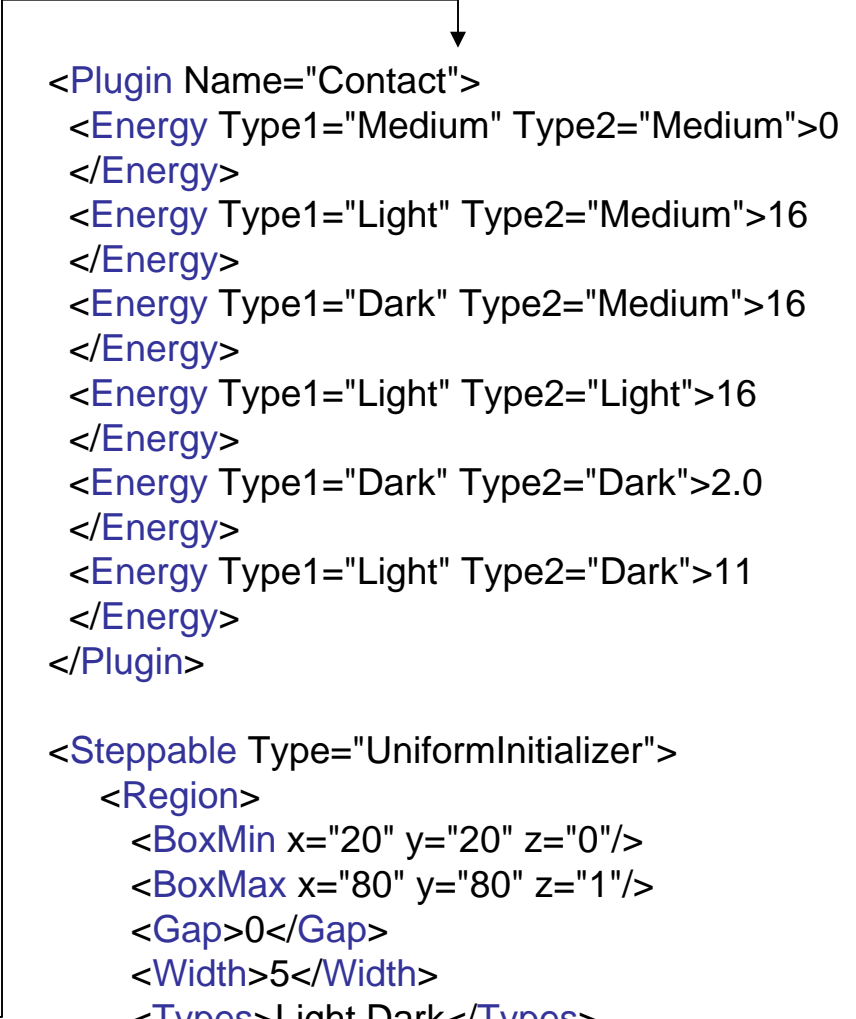
List of Cell Types to Include. If a Cell Type is repeated, the fraction of that Cell Type is proportional to the number of times it is listed

Putting It All Together - cellsort_2D.xml

```
<CompuCell3D>
  <Potts>
    <Dimensions x="100" y="100" z="1"/>
    <Steps>10</Steps>
    <Temperature>10</Temperature>
    <NeighborOrder>2</NeighborOrder>
  </Potts>

  <Plugin Name="CellType">
    <CellType TypeName="Medium" TypeId="0"/>
    <CellType TypeName="Light" TypeId="1"/>
    <CellType TypeName="Dark" TypeId="2"/>
  </Plugin>

  <!-- Replaced by-type constraint with global one -->
  <Plugin Name="Volume">
    <TargetVolume>25</TargetVolume>
    <LambdaVolume>1.0</LambdaVolume>
  </Plugin>
```



```
<Plugin Name="Contact">
  <Energy Type1="Medium" Type2="Medium">0
</Energy>
  <Energy Type1="Light" Type2="Medium">16
</Energy>
  <Energy Type1="Dark" Type2="Medium">16
</Energy>
  <Energy Type1="Light" Type2="Light">16
</Energy>
  <Energy Type1="Dark" Type2="Dark">2.0
</Energy>
  <Energy Type1="Light" Type2="Dark">11
</Energy>
</Plugin>

<Steppable Type="UniformInitializer">
  <Region>
    <BoxMin x="20" y="20" z="0"/>
    <BoxMax x="80" y="80" z="1"/>
    <Gap>0</Gap>
    <Width>5</Width>
    <Types>Light,Dark</Types>
  </Region>
</Steppable>
</CompuCell3D>
```

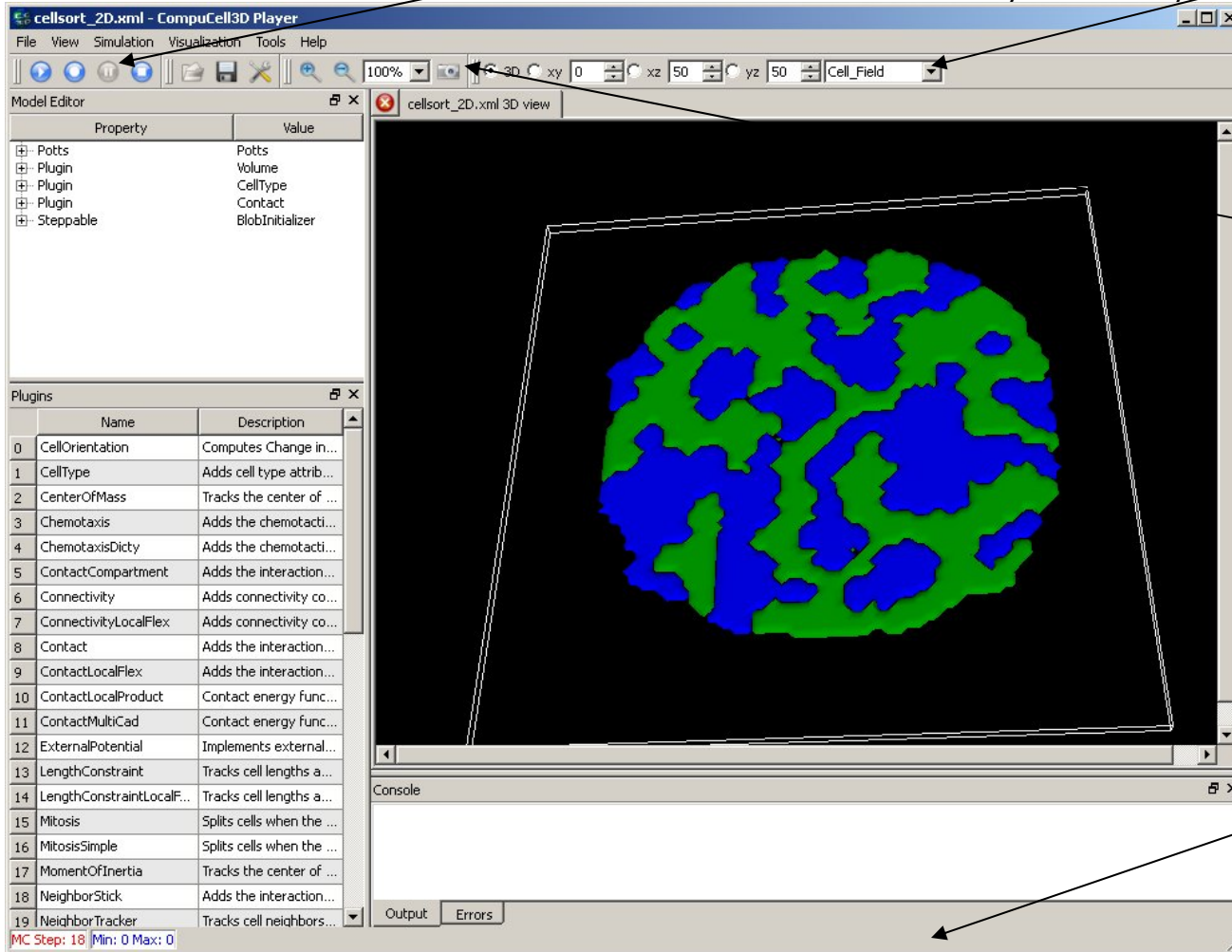


The same simulation in C/C++/Java/Fortran would take at least 1000 lines of code...



Running Cell Sorting Simulation in CompuCell Player

Steering bar allows users to start or pause the simulation, zoom in , zoom out, to switch between **2D and 3D** visualization, change **view modes** (cell field, pressure field , chemical concentration field, velocity field etc..)



Player can output multiple views during single simulation run – **Add Screenshot** function

Information bar

Opening cell sorting simulation in CompuCell Player

Go to File->Open Simulation File and navigate to **C:/CC3DProjects/Cellsorting** directory. From this directory choose **Cellsorting.cc3d** project file.

The screenshot displays the CompuCell Player interface. The main window is titled 'cellsort_2D.xml - CompuCell3D Player'. An 'Open Simulation File' dialog box is open, showing the 'Look in:' path as 'cellsort_2D'. The file list includes:

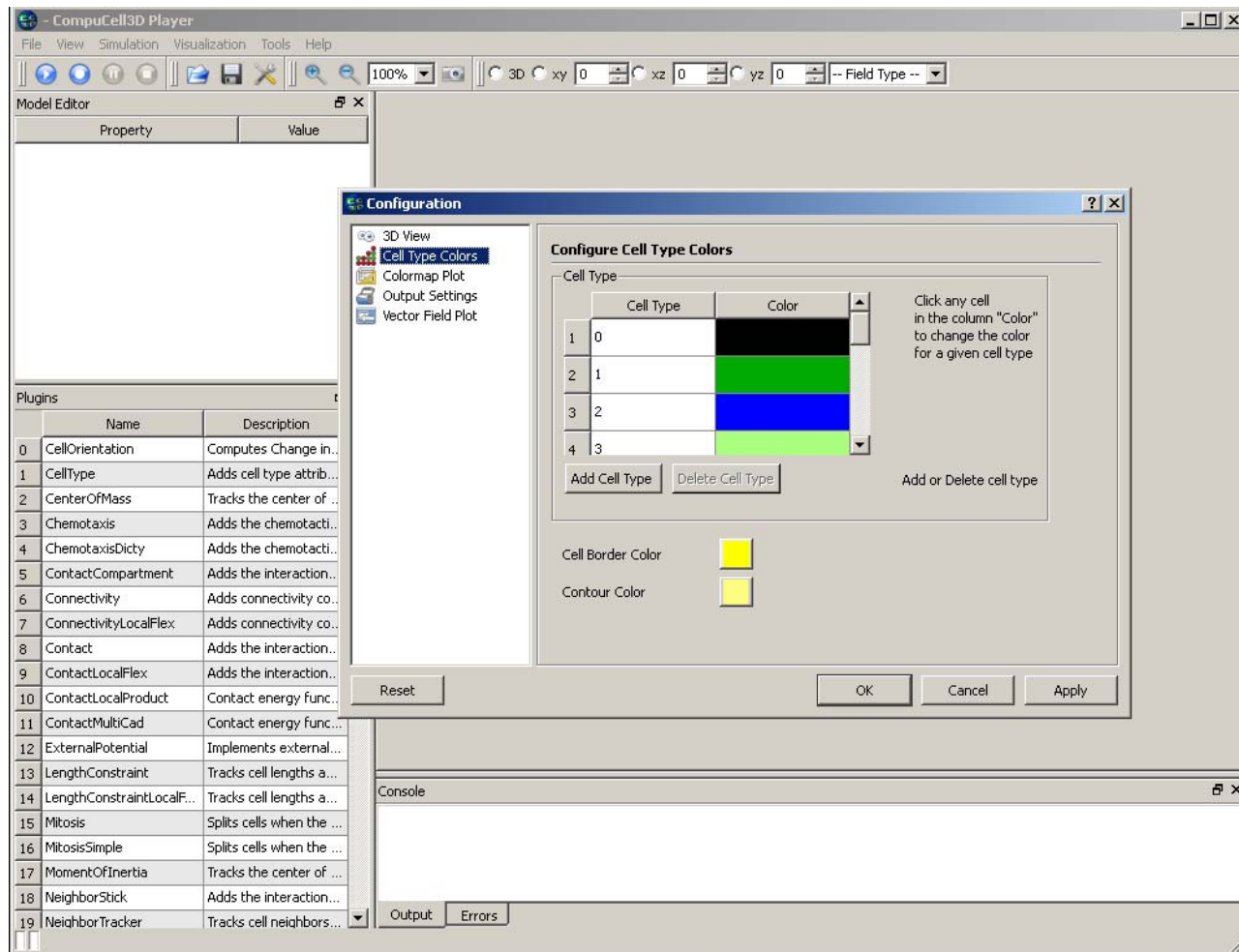
- cellsort_2D.xml
- cellsort_2D_boundary.xml
- cellsort_2D_PlayerSettings.xml
- cellsort_2D_variable_motility.xml
- cellsort_engulfment_2D.xml

The 'File name:' field contains 'cellsort_2D.xml' and the 'Files of type:' dropdown is set to 'XML files or Python scripts (*.xml *.py)'. The background interface shows a 'Plugins' list with 19 items, including 'CellOrientation', 'CellType', 'CenterOfMass', 'Chemotaxis', 'ChemotaxisDicty', 'ContactCompartment', 'Connectivity', 'ConnectivityLocalFlex', 'Contact', 'ContactLocalFlex', 'ContactLocalProduct', 'ContactMultiCad', 'ExternalPotential', 'LengthConstraint', 'LengthConstraintLocalF...', 'Mitosis', 'MitosisSimple', 'MomentOfInertia', 'NeighborStick', and 'NeighborTracker'. A 'Console' window is visible at the bottom.

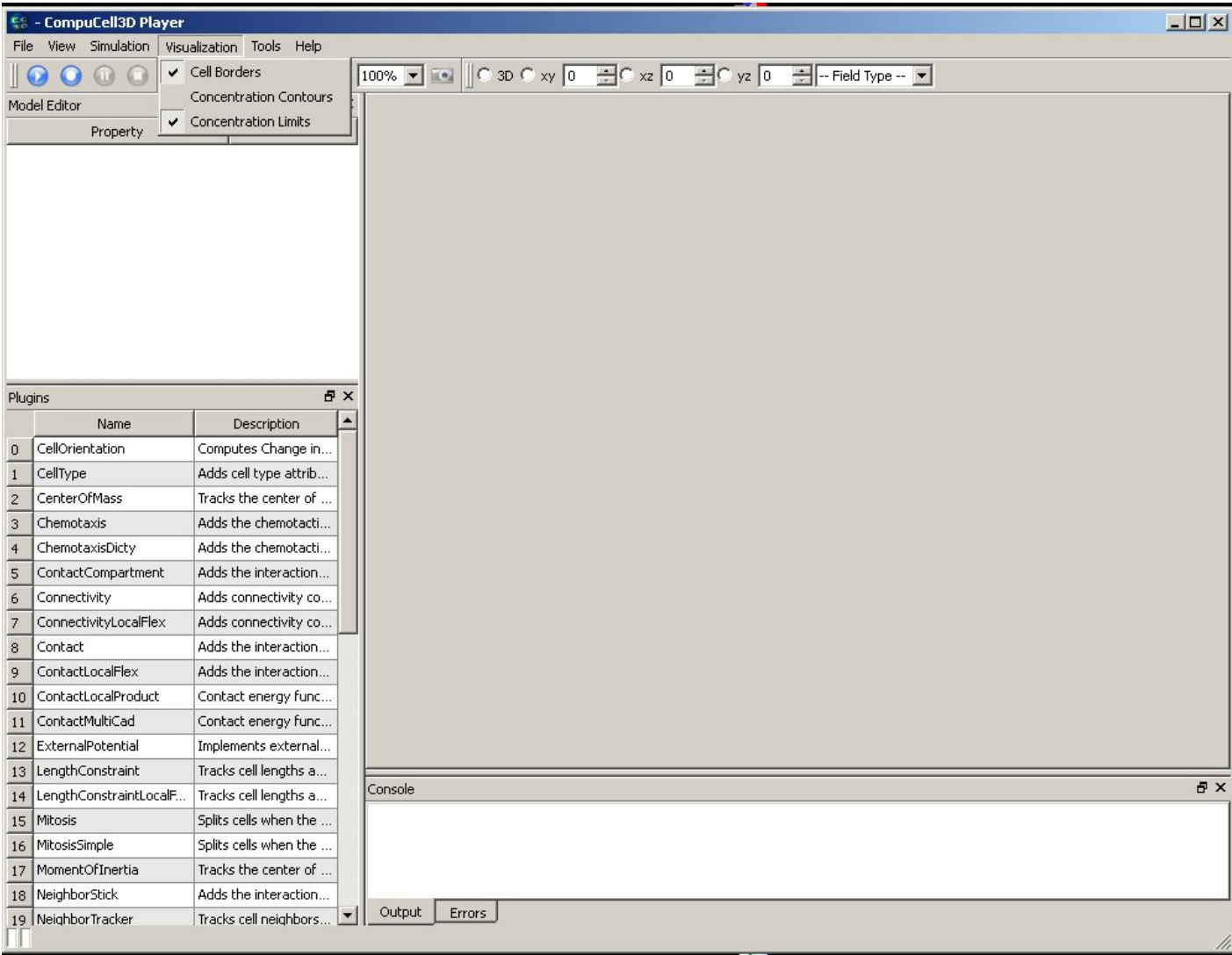


Configuring the Player

Most of Player's configuration options are accessible through **Tools->Configuration...** and **Visualization** menus.

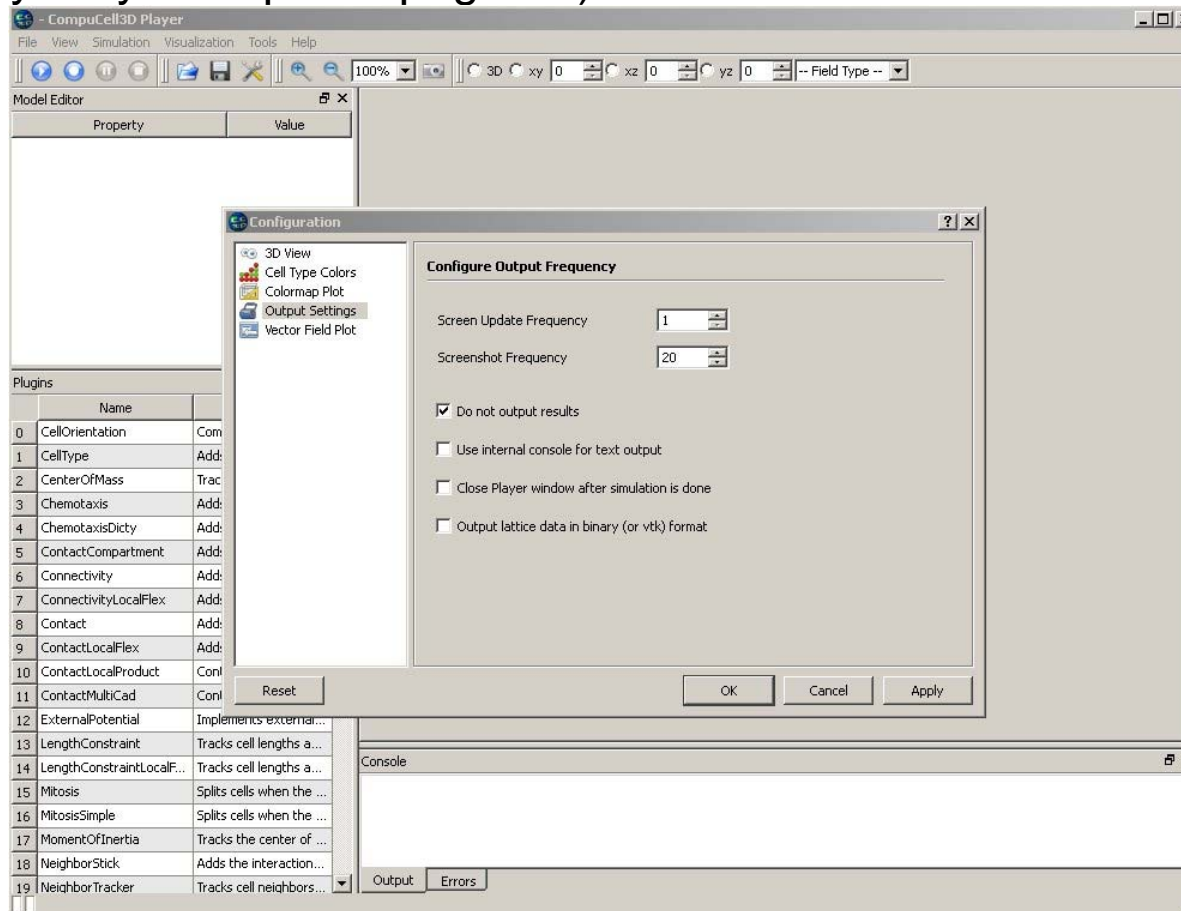


Visualization Menu allows you to choose whether in 2D cell borders should be displayed or not (in 3D borders are not drawn at all). You can also select to draw isocontour lines for the concentration plots and turn on and off displaying of the information about minimum and maximum concentration.



Screen update frequency is a parameter that defines how often (in units of MCS) Player screen should be updated. Note, if you choose to update screen too often (say every MCS) you will notice simulation speed degradation because it does take some time to draw on the screen. You may also choose not to output any files by checking “**Do not output results**” check-box. Additionally you have the option to output simulation data in the VTK format for later replay.

Screenshot frequency determines how often screenshots of the lattice views will be taken (currently Player outputs *.png files).



Screenshots

Screenshots are taken every “Screenshot Frequency” MCS

By default Player will store screenshots of the currently displayed lattice view.

In addition to this users can choose to store additional screenshots at the same time. Simply switch to different lattice view, click camera button. Those additional screenshots will be taken irrespectively of what Player currently displays.

Once you selected additional screenshots it is convenient to save screenshot description file (it is written automatically by the Player, user just provide file name). Next time you decide to run CompuCell3D you may just use command

```
compuCell3d.sh -s screenshotDescriptionFile_cellsort.txt -i cellsort_2D.xml
```

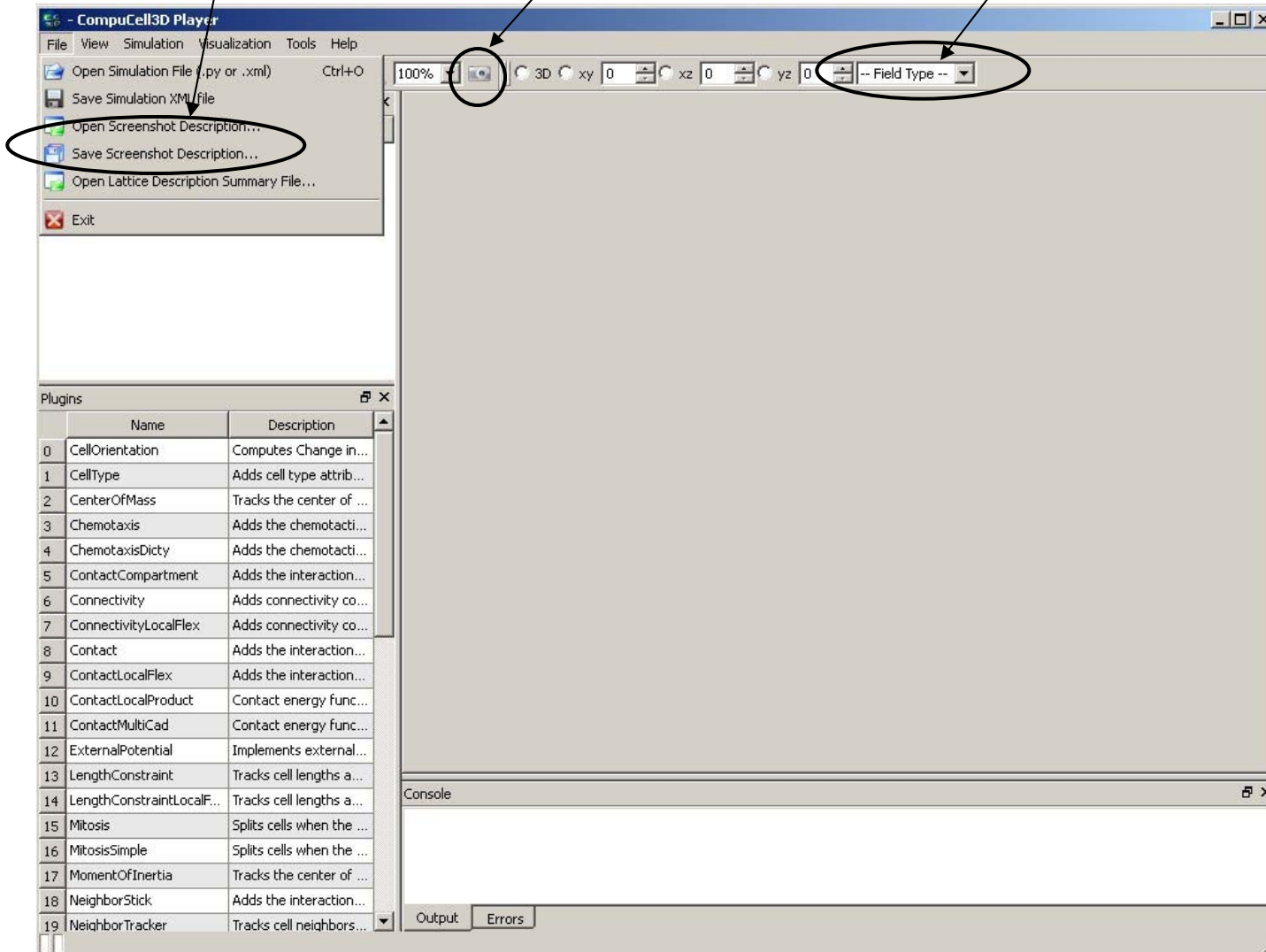
This will run simulation where stored screenshots will be taken



When you picked lattice views, you may save screenshot description file for later reuse

Click camera button on select lattice views

Notice, you may change plot types as well



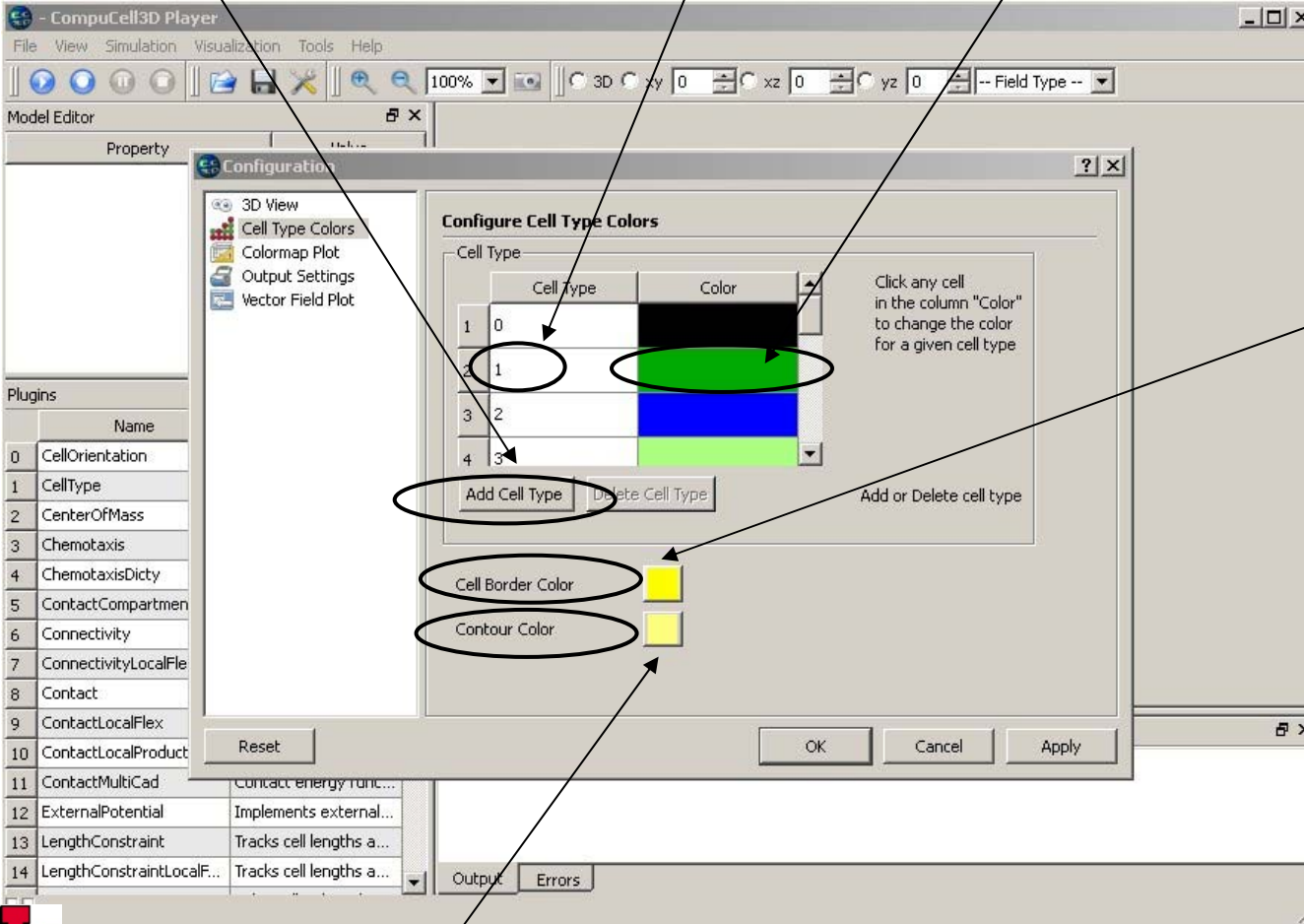
Configuring cell type colors

To enter new cell type click **"Add Cell Type"** button

Enter cell type number here

Click here to change color for cell type 1

Click here to change cell border color



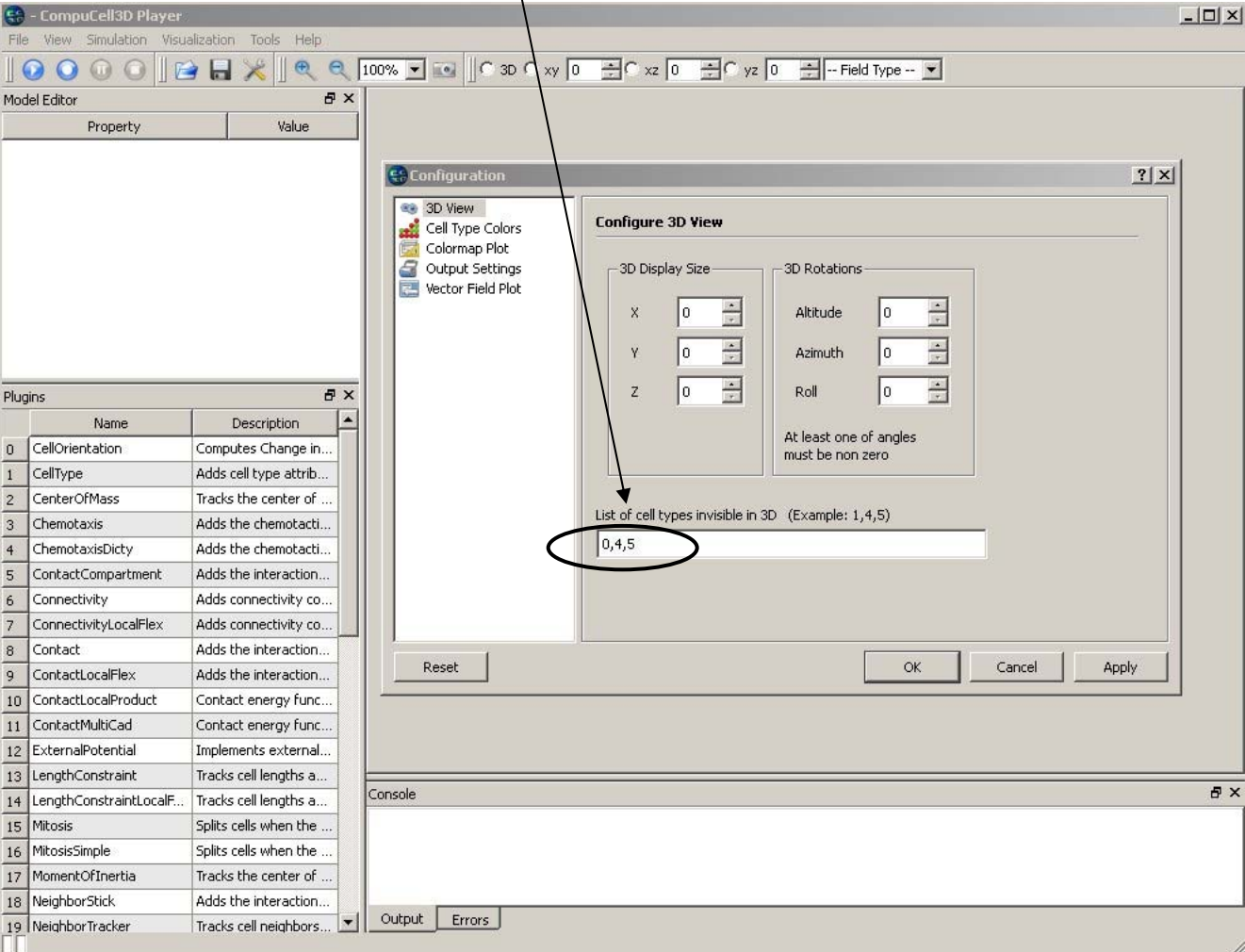
Click here to change isocontour color



Configuring cell types invisible in 3D visualizations

Sometimes when you open up the simulation and switch to 3D view you may find that your simulation looks like solid a parallelepiped. This might be due to a box made out of frozen cells that hides inside other cells. In this case you need to make the box invisible.

Type cell type number that you want to be invisible in 3D in this box. Notice, by default Player will not display Medium (type 0). Here we also make types 4 and 5 invisible



The screenshot shows the CompuCell3D Player interface. The Configuration dialog box is open, showing the 'Configure 3D View' section. The '3D Display Size' section has X, Y, and Z sliders all set to 0. The '3D Rotations' section has Altitude, Azimuth, and Roll sliders all set to 0. Below these sections is a text input field labeled 'List of cell types invisible in 3D (Example: 1,4,5)' containing the text '0,4,5'. A red circle highlights this text, and a red arrow points from the text above to it. The 'Plugins' table is visible in the background.

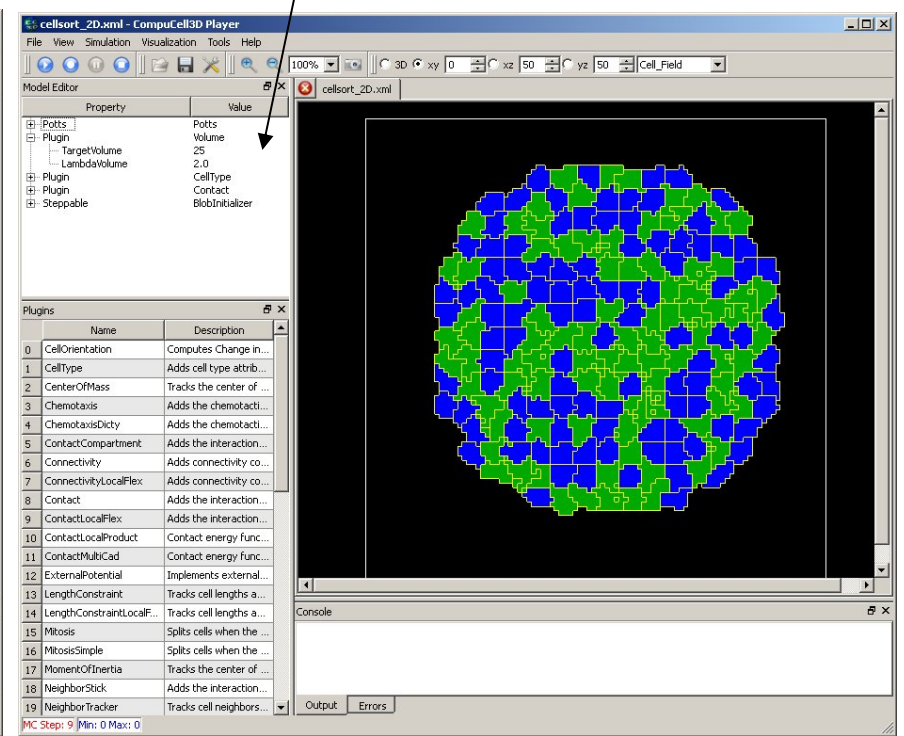
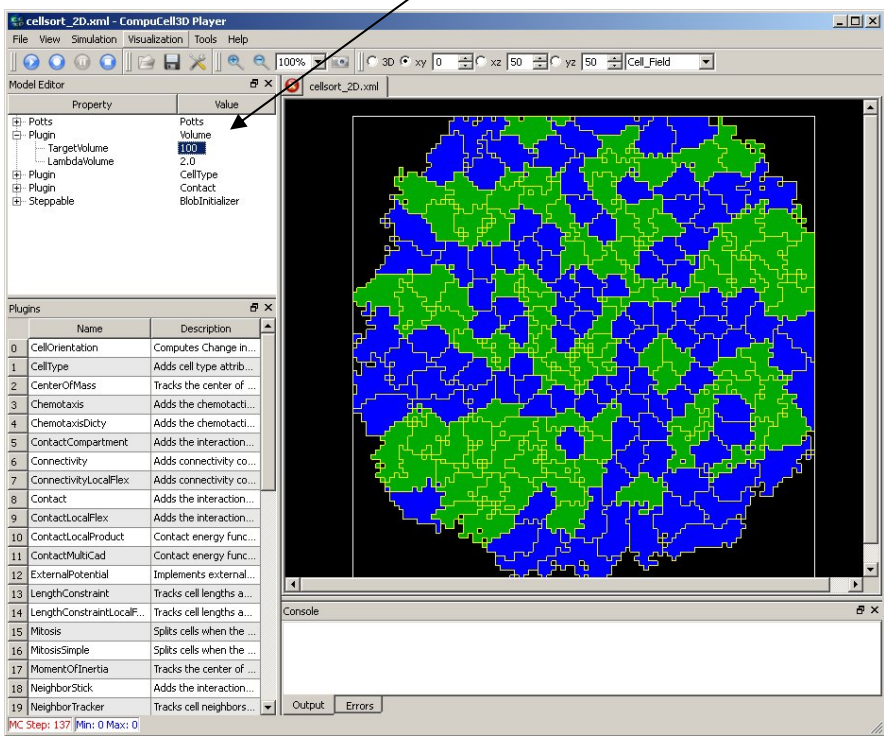
| Index | Name | Description |
|-------|---------------------------|---------------------------|
| 0 | CellOrientation | Computes Change in... |
| 1 | CellType | Adds cell type attrib... |
| 2 | CenterOfMass | Tracks the center of ... |
| 3 | Chemotaxis | Adds the chemotacti... |
| 4 | ChemotaxisDicty | Adds the chemotacti... |
| 5 | ContactCompartment | Adds the interaction... |
| 6 | Connectivity | Adds connectivity co... |
| 7 | ConnectivityLocalFlex | Adds connectivity co... |
| 8 | Contact | Adds the interaction... |
| 9 | ContactLocalFlex | Adds the interaction... |
| 10 | ContactLocalProduct | Contact energy func... |
| 11 | ContactMultiCad | Contact energy func... |
| 12 | ExternalPotential | Implements external... |
| 13 | LengthConstraint | Tracks cell lengths a... |
| 14 | LengthConstraintLocalF... | Tracks cell lengths a... |
| 15 | Mitosis | Splits cells when the ... |
| 16 | MitosisSimple | Splits cells when the ... |
| 17 | MomentOfInertia | Tracks the center of ... |
| 18 | NeighborStick | Adds the interaction... |
| 19 | NeighborTracker | Tracks cell neighbors... |



Steering the simulation

CompuCell3D Player will allow you to change most of the parameters of the XML file while the simulation is running.

Use steering panel to change simulation parameters. Make sure you pause simulation before doing this



Target volume = 100

Screenshot was taken before simulation had time to equilibrate

Target volume = 25



Exploring how different parameters affect cellular behaviors in cell sorting simulation

1. Vary cell membrane fluctuation amplitude (aka temperature)
2. Vary LambdaVolume, TargetVolume
3. Vary Contact Energy coefficients

Please refer to the Quick start guide to find set of exercises which will help you better understand the roles played by all parameters



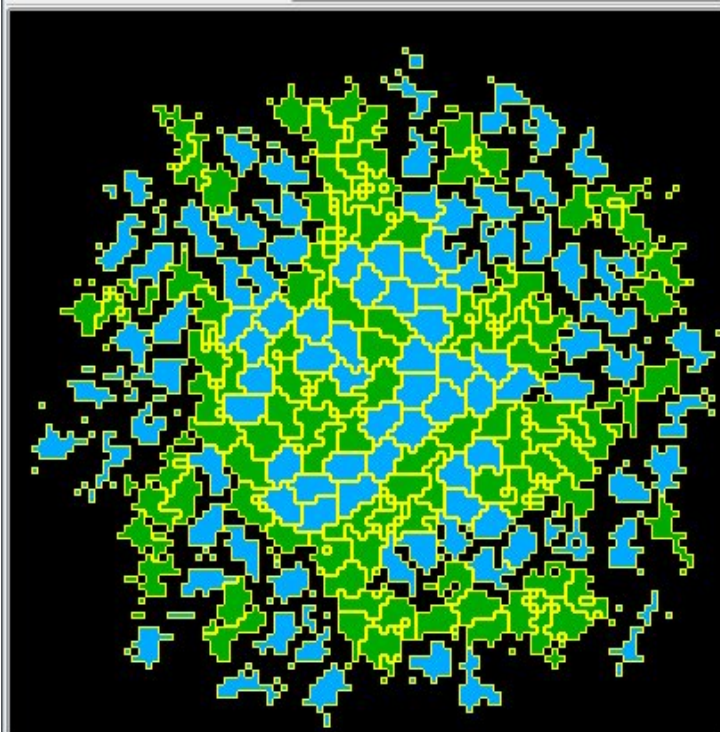
Practical way of guessing contact energy hierarchy

Basic facts:

- Cells that have high contact energies between themselves, when they come together they **increase** overall energy of the system. **Such cells tend to stay away from each other.**
- Cells that have low contact energies between themselves, when they come together they **decrease** overall energy of the system. **Such cells tend to cluster together.**
- Those two rules are helpful when determining contact energy hierarchy. Simply cells of one type like to be surrounded by those cells with which the contact energy is the lowest.
- And vice versa, if you want to make two cells not to touch each other, make sure that contact energy between them is high.



Examples of different contact energy hierarchies

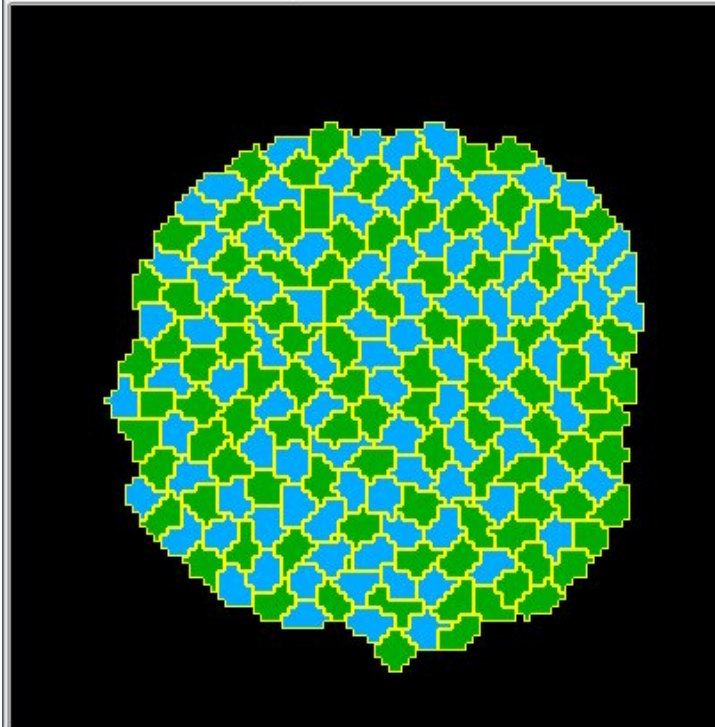


Cell sorting simulation where cells of both type like to be surrounded by medium. That is contact energy between **Condensing** and **Medium** as well as between **NonCondensing** and **Medium** is very low

$$J_{CM} = J_{NM} < J_{NN} < J_{CC} < J_{NC}$$



Examples of different contact energy hierarchies



Cell sorting simulation where cells of both type do not like to be surrounded by medium and cells of homotypic cells do not like each other

$$J_{NC} \ll J_{NN} = J_{CC} < J_{CM} = J_{NM}$$



CompuCell3D Subtleties

Now that we've seen and run a simulation, we can go back and review some general points:

1. Understanding XML
2. Running CC3D from command line (useful for running CC3D on clusters)
3. Replacing XML with corresponding Python syntax



Generic XML 101

CC3DML is an XML, which stands for eXtensible Markup Language. A standard way to exchange information between different applications.

XML Example:

```
<Sentence>
```

```
  <Text>It is too early to be in class</Text>
```

```
  <FontType>TimesNewRoman</FontType>
```

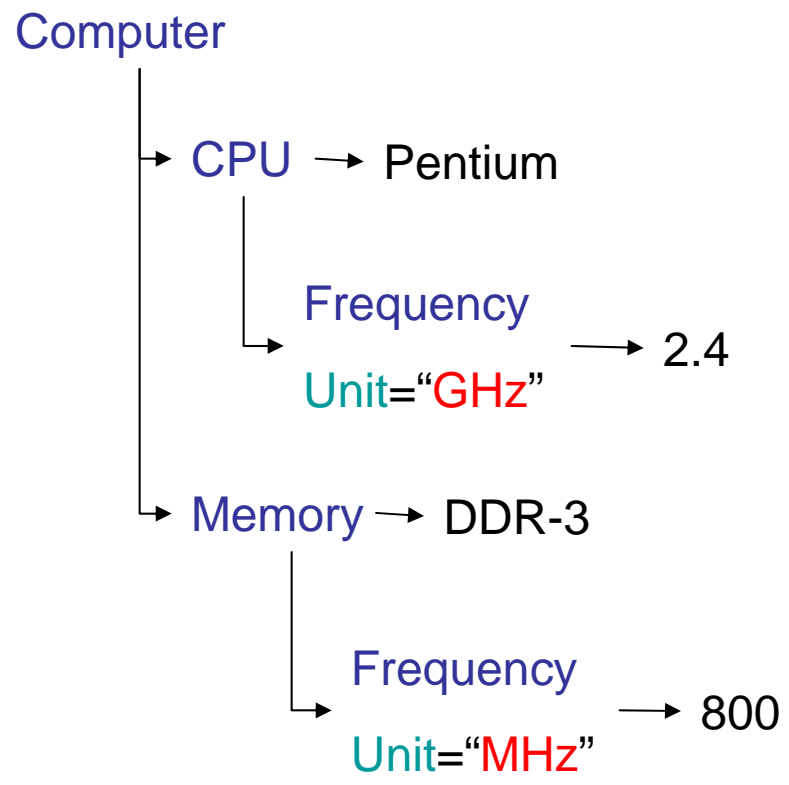
```
  <FontSize>12</FontSize>
```

```
  <DisplayHint Hint="AddFrameAround"/>
```

```
</Sentence>
```

XML is essentially a definition of hierarchical (tree-like) data structure

```
<Computer>  
  <CPU>Pentium  
    <Frequency Unit="GHz">2.4</Frequency>  
  </CPU>  
  <Memory>DDR-3  
    <Frequency Unit="MHz">800</Frequency>  
  </Memory>  
  ...  
</Computer>
```



Putting It All Together - Avoiding Common Errors in CC3DML code

1. The CC3DML **must** specify the simulation in the following order:

- Potts
- Plugins
- Steppables

If you mix, e.g. Plugins with Steppables you will get an error.

2. Remember to match every xml tag with a closing tag

```
<Plugin>
```

```
...
```

```
</Plugin>
```

3. Watch for typos – an error in the CC3DML syntax will generate an error pointing to the offending line

4. **Modify/reuse examples when possible, rather than starting from scratch – saves a lot of time**



Running a Simulation From the Command Line

You can start a simulation with or without CompuCell Player from the command line.

Open a console (terminal) and type:

./compuCell3d.command -i cellsort_2D.xml (on OSX)

./compuCell3d.sh -i cellsort_2D.xml (on Linux)

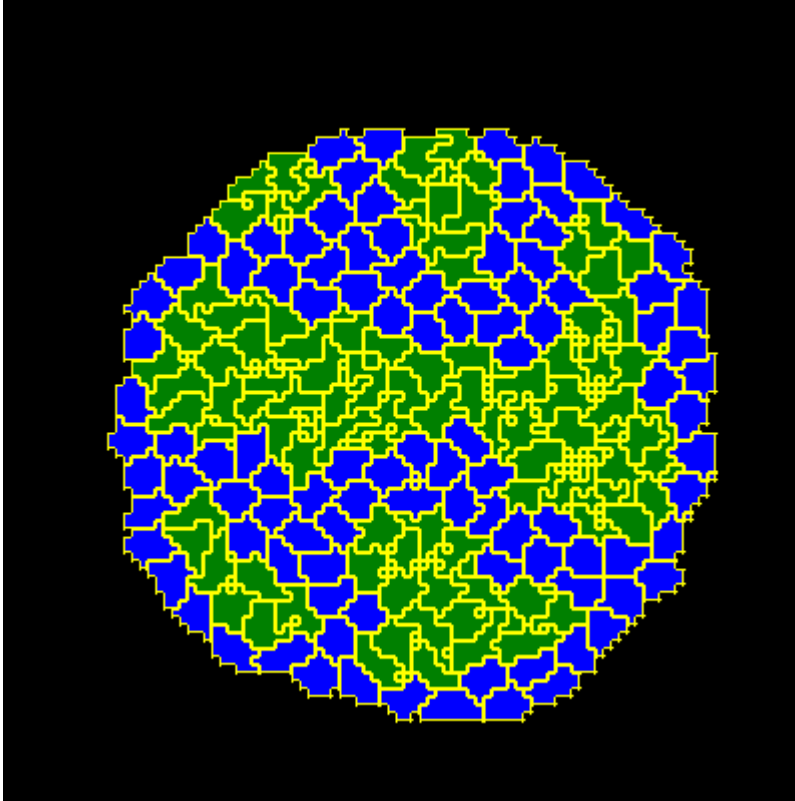
compuCell3d.bat -i cellsort_2D.xml (on Windows) – or simply double click the CC3D Desktop icon

Running CompuCell3D from the command line is required if you want to run in batch mode on a cluster. For more information about command line options see the “Running CompuCell3D” manual at www.compuCell3d.org.

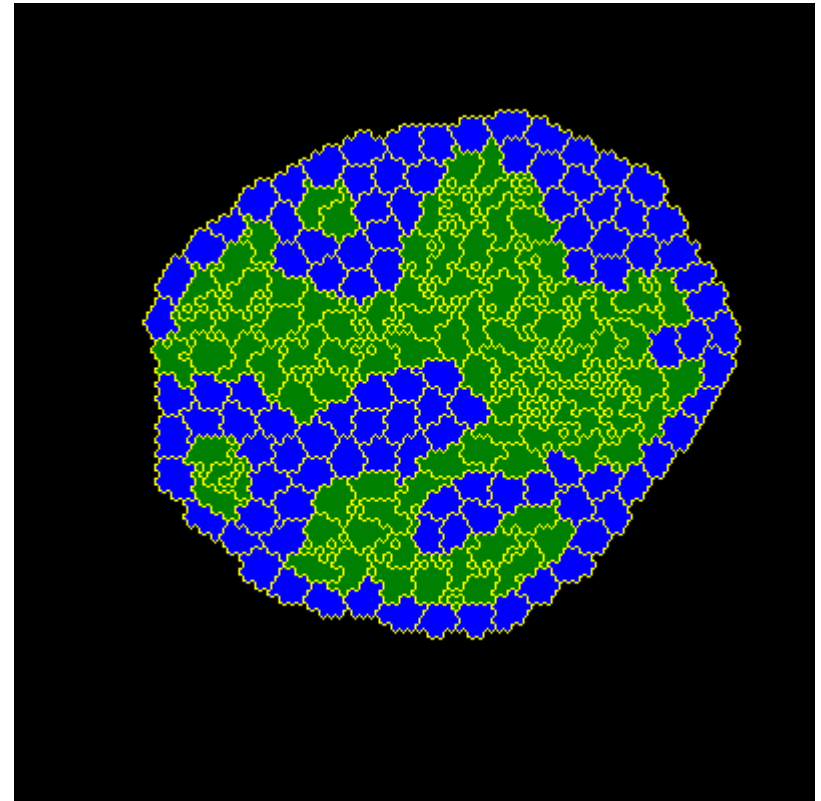


Cell-sorting simulation on square and hexagonal lattices

The simulation parameters were kept the same for the two runs



1000 MCS



1000 MCS

Replacing CC3DML with Python



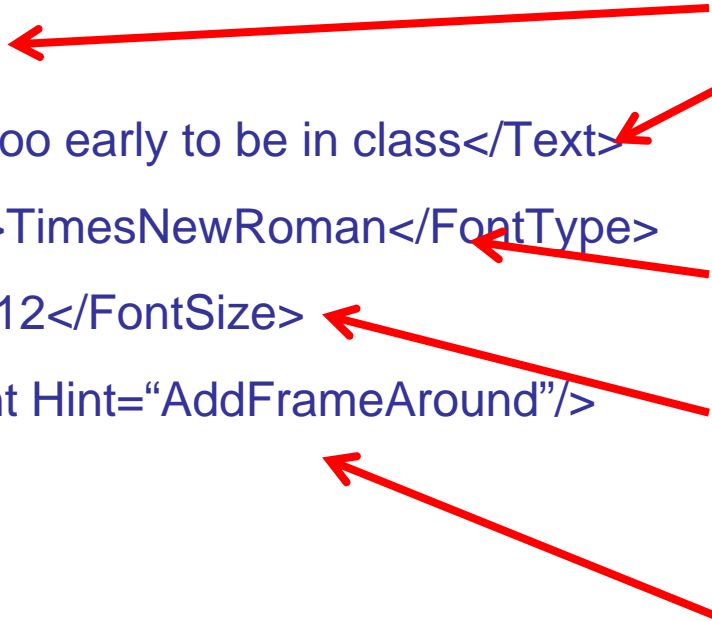
CC3D Supports Python Syntax Parallel to CC3DML Syntax

Generic XML Example:

```
<Sentence>  
  <Text>It is too early to be in class</Text>  
  <FontType>TimesNewRoman</FontType>  
  <FontSize>12</FontSize>  
  <DisplayHint Hint="AddFrameAround"/>  
</Sentence>
```

Parallel Python Example

```
def configureSimulation(sim):  
  Snt=ElementCC3D("Sentence")  
  Txt=Snt.ElementCC3D("Text",{  
    ,"It is too early")  
  Fnt=Snt.ElementCC3D("FontType",{  
    "TimesNewR")  
  fntSize=Snt.ElementCC3D("FontSize",  
    {},12)  
  Disp=Snt.ElementCC3D("DisplayHint",  
    {"Hint":"AddFrameAround"})
```



Choosing the Right Text Editor

Since developing CompuCell3D simulation requires typing some simple code it is important that you have the right tools to do that most effectively.

THE BEST EDITOR IS TWEDIT (supported by Consumer Research tests)

- On Windows systems we also recommend Notepad++ editor:

<http://notepad-plus.sourceforge.net/uk/site.htm>

- On Linux you have lots of choices: Kate (my favorite), gedit, mcedit etc.

- On OSX situation you may use Smultron

<http://sourceforge.net/projects/smultron/>

or TextWrangler

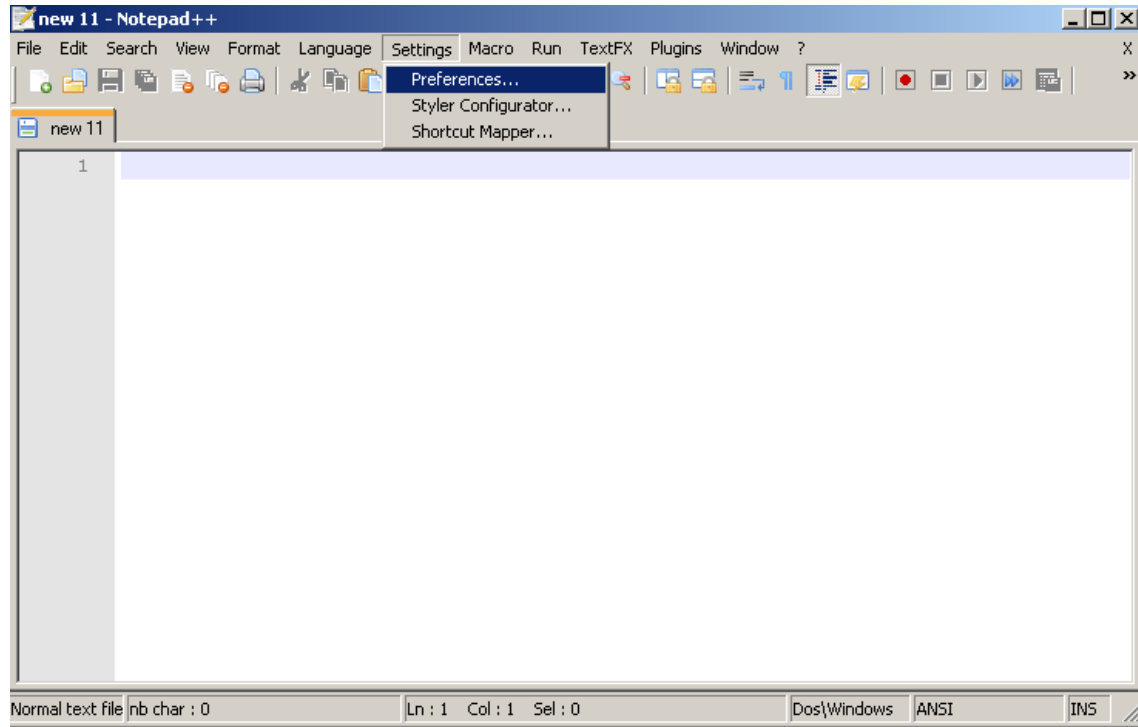
<http://www.barebones.com/products/textwrangler/>

And as usual, if nothing else works there is always vi, emacs and punch-cards



Configuring Notepad++ for use with Python

Go to Settings->Preferences...

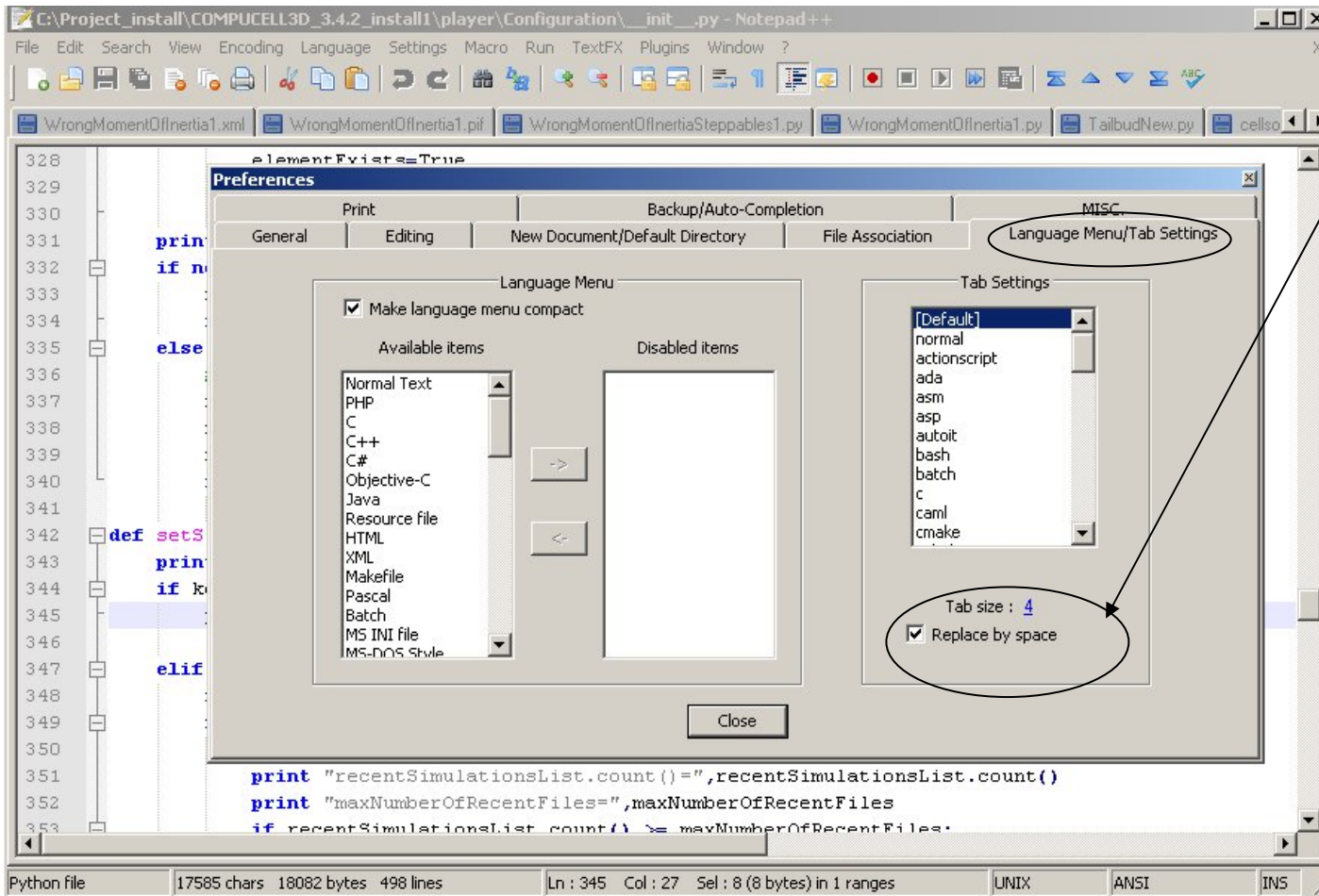


On the “Edit Components” tab change Tab Settings to :

Tab size: 4

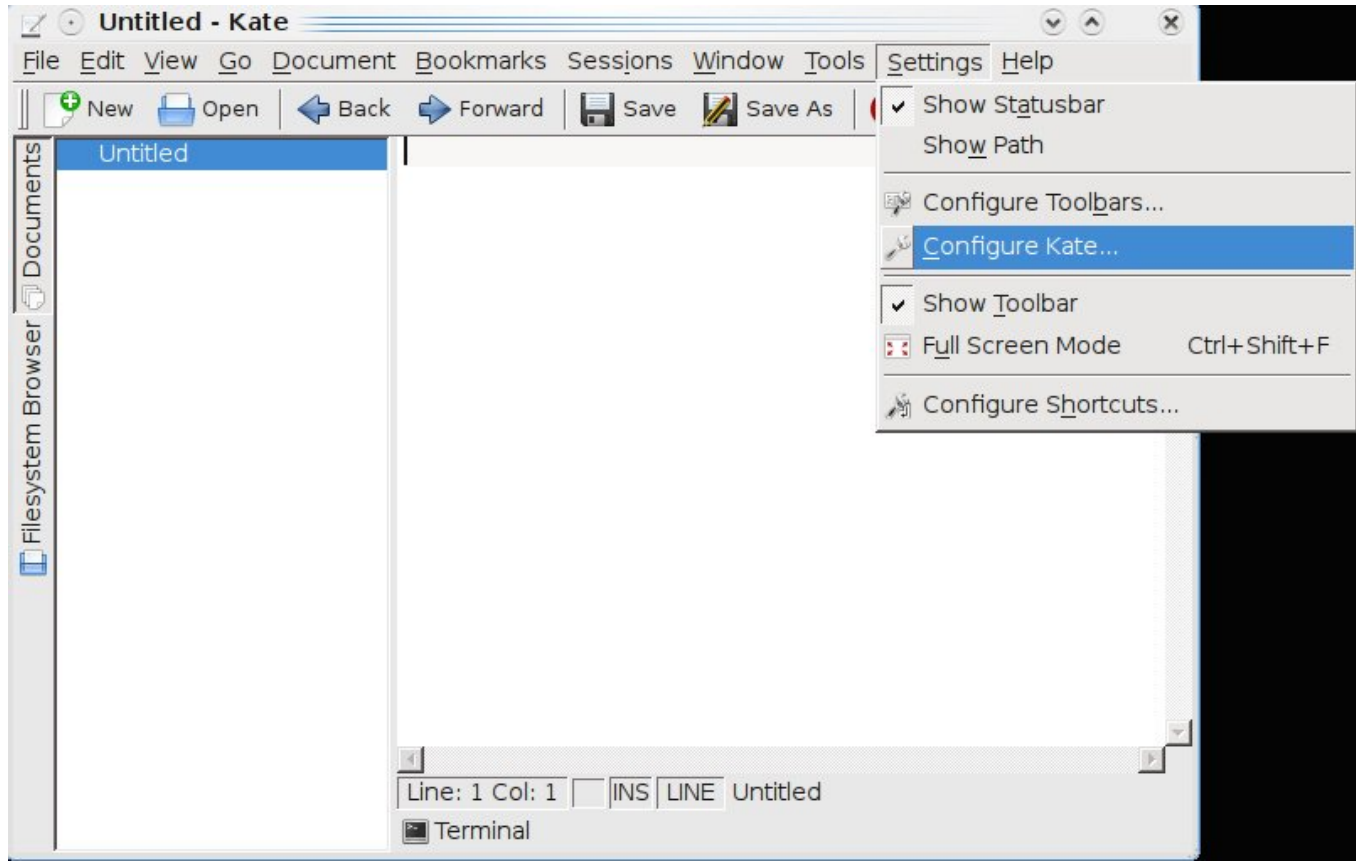
Replace by space: “checked”

Click on the number to change it



Configuring Kate for use with Python

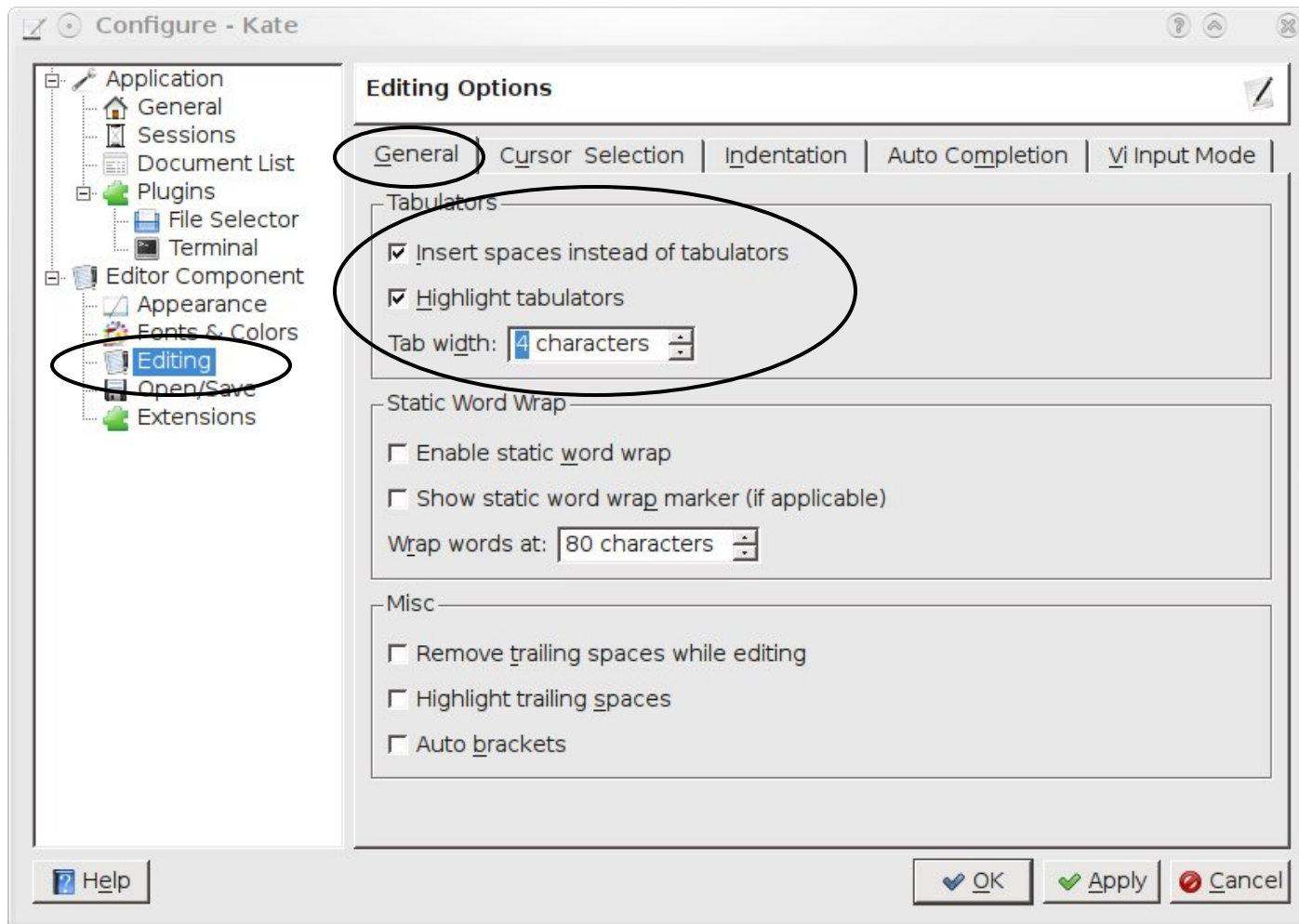
Go to Settings->Configure Kate ...



Click Editing and in the “General” Tab in “Tabulators” section set:

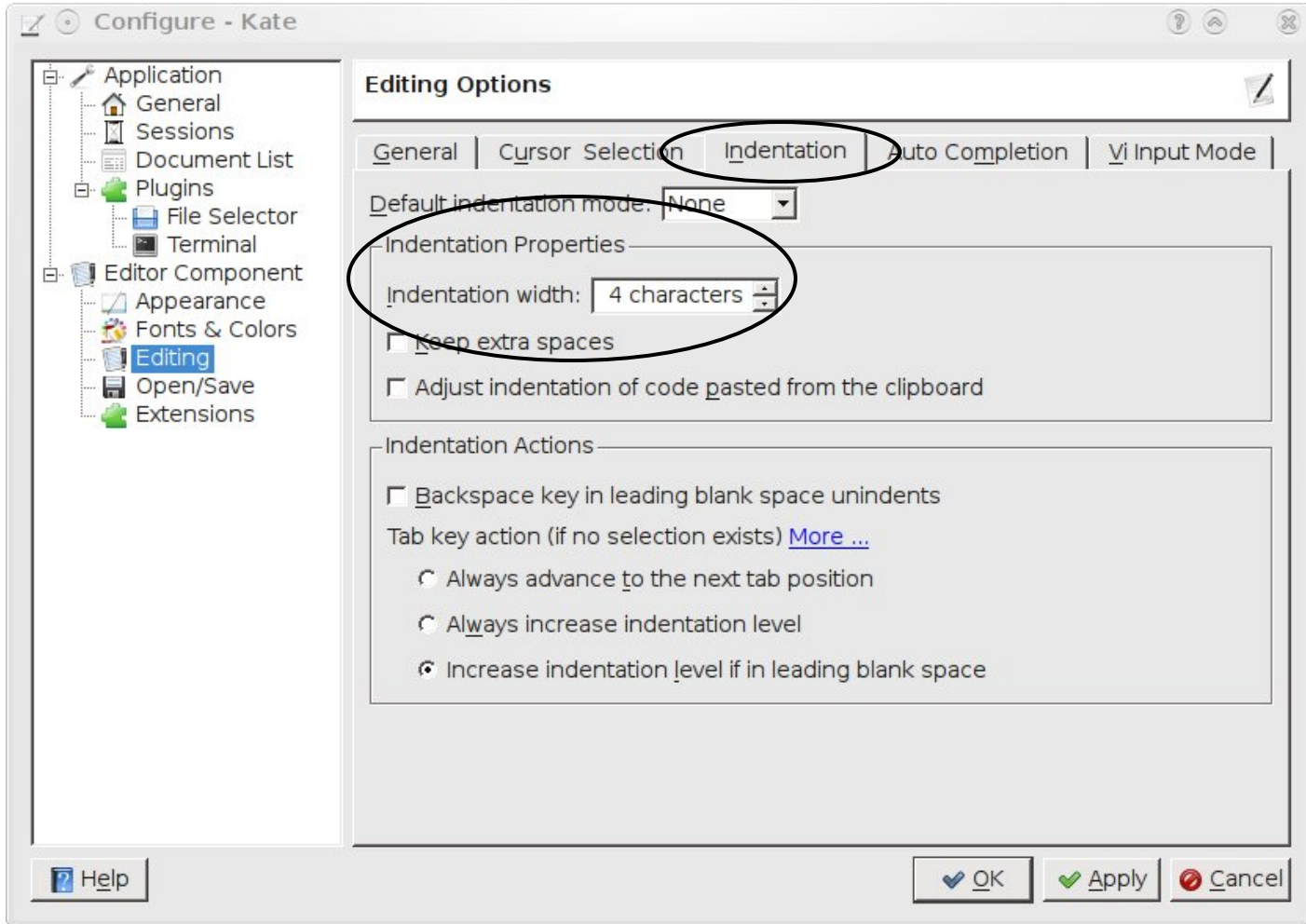
Insert spaces instead of tabulators: “checked”

Tab width: “4 characters”



On “Indentation” tab in “Indentation Properties” section set:

Indentation width: 4 characters



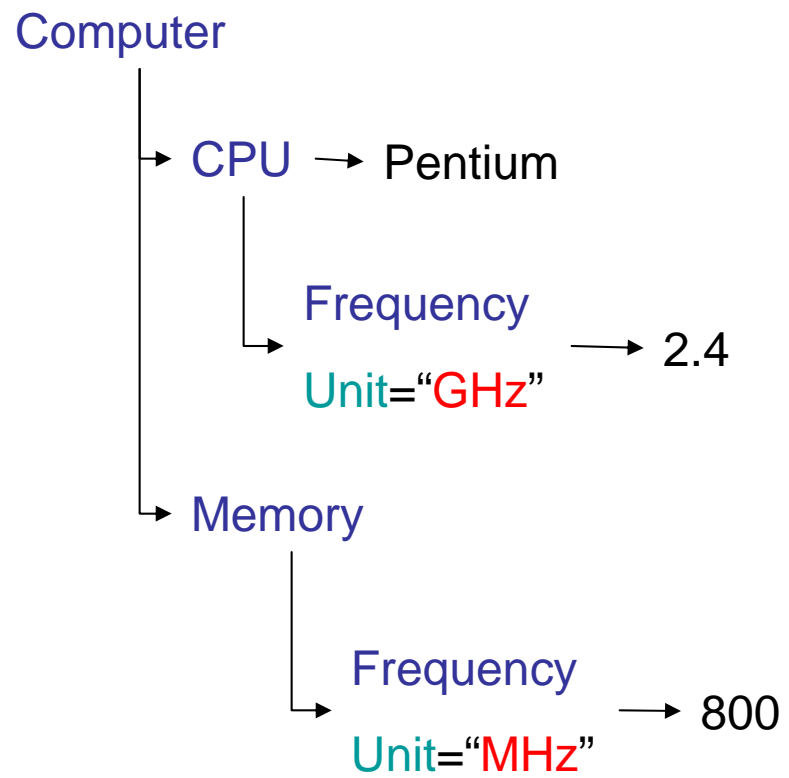
Using Python to describe entire simulations

- Starting with 3.2.0 versions you may get rid of XML file and use Python to describe entire simulation.
- The advantage of doing so is that you have one less file to worry about but also you may more easily manipulate simulation parameters. For example if you want contact energy between two cell types be twice as big as between two other cell types you could easily implement it in Python. Doing the same exercise with CC3DML is a bit harder (but not impossible).
- Python syntax used to describe simulation closely mimics CC3DML syntax. There are however certain differences and inconsistencies caused by the fact that we are using different languages to accomplish same task. Currently there is no documentation explaining in detail Python syntax that replaces CC3DML. It will be developed soon
- The most important reason for defining entire simulation in Python is the possibility of simulation steering i.e. the ability to dynamically change simulation parameters while simulation is running (available in 3.2.1)
- The way you replace XML in Python is purely mechanical and we will show it on a simple example



XML is essentially a definition of hierarchical (tree-like) data structure

```
<Computer>  
  <CPU>Pentium  
    <Frequency Unit="GHz">2.4</Frequency>  
  </CPU>  
  <Memory>DDR-3  
    <Frequency Unit="MHz">800</Frequency>  
  </Memory>  
  ...  
</Computer>
```



Building tree-like structure in a computer language (e.g. Python)

```
root=createElement(...parameters...)
```

```
child1=root.createElement(...parameters...)
```

```
child1_of_child1=child1.createElement(...parameters...)
```

```
child2=root.createElement(...parameters...)
```

```
child1_of_child2=child2.createElement(...parameters...)
```



Replacing XML with Python syntax:

```
import CompuCellSetup
from XMLUtils import ElementCC3D

cc3d=ElementCC3D("CompuCell3D")
potts=cc3d.ElementCC3D("Potts")
potts.ElementCC3D("Dimensions",{"x":100,"y":100,"z":1})
potts.ElementCC3D("Anneal",{ },10)
potts.ElementCC3D("Steps",{ },1000)
potts.ElementCC3D("Temperature",{ },10)
potts.ElementCC3D("NeighborOrder",{ },2)
```

```
<CompuCell3D>
  <Potts>
    <Dimensions x="100" y="100" z="1"/>
    <Anneal>10</Anneal>
    <Steps>10000</Steps>
    <Temperature>10</Temperature>
    <NeighborOrder>2</NeighborOrder>
  </Potts>
</CompuCell3D>
```

Notice , by using Python we have even saved few lines



Rules:

- To open XML document, create parent ElementCC3D:

```
cc3d=ElementCC3D("CompuCell3D")
```

- For nesting XML elements inside another XML element use the following:

```
potts=cc3d.ElementCC3D("Potts")
```

- If the element has attribute use Python dictionary syntax to list the attributes:

```
potts.ElementCC3D("Dimensions",{"x":100,"y":100,"z":1})
```

- If the XML element has value but no attributes use the following:

```
potts.ElementCC3D("NeighborOrder",{},2)
```

- If the XML element has both value and attributes combine two previous examples

```
potts.ElementCC3D("NeighborOrder",{"LatticeType":"Hexagonal"},2)*
```



Python-based simulation – template script

```
import sys
from os import environ
import string
sys.path.append(environ["PYTHON_MODULE_PATH"])


import CompuCellSetup

sim,simthread = CompuCellSetup.getCoreSimulationObjects()

configureSimulation(sim)

CompuCellSetup.initializeSimulationObjects(sim,simthread)

from PySteppables import SteppableRegistry
steppableRegistry=SteppableRegistry()
```

```
 CompuCellSetup.mainLoop(sim,simthread,steppableRegistry)
```



But you need to implement **configureSimulation** function:

Python

```
def configureSimulation(sim):  
    import CompuCellSetup  
    from XMLUtils import ElementCC3D  
    cc3d=ElementCC3D("CompuCell3D")  
    potts=cc3d.ElementCC3D("Potts")  
    potts.ElementCC3D("Dimensions",{"x":100,"y":100,"z":1})  
    potts.ElementCC3D("Steps", {},1000)  
    potts.ElementCC3D("Temperature", {},10)  
    potts.ElementCC3D("NeighborOrder", {},2)  
    cellType=cc3d.ElementCC3D("Plugin", {"Name":"CellType"})  
    cellType.ElementCC3D("CellType", {"TypeName":"Medium", "TypeId":"0"})  
    cellType.ElementCC3D("CellType", {"TypeName":"Condensing", "TypeId":"1"})  
    cellType.ElementCC3D("CellType", {"TypeName":"NonCondensing", "TypeId":"2"})  
    volume=cc3d.ElementCC3D("Plugin", {"Name":"Volume"})  
    volume.ElementCC3D("TargetVolume", {},25)  
    volume.ElementCC3D("LambdaVolume", {},2.0)
```



Continued...

```
contact=cc3d.ElementCC3D("Plugin",{"Name":"Contact"})
contact.ElementCC3D("Energy", {"Type1":"Medium", "Type2":"Medium"},0)
contact.ElementCC3D("Energy", {"Type1":"NonCondensing", "Type2":"NonCondensing"},16)
contact.ElementCC3D("Energy", {"Type1":"Condensing", "Type2":"Condensing"},2)
contact.ElementCC3D("Energy",{"Type1":"NonCondensing", "Type2":"Condensing"},11)
contact.ElementCC3D("Energy", {"Type1":"NonCondensing", "Type2":"Medium"},16)
contact.ElementCC3D("Energy", {"Type1":"Condensing", "Type2":"Medium"},16)
blobInitializer=cc3d.ElementCC3D("Steppable",{"Type":"BlobInitializer"})
blobInitializer.ElementCC3D("Gap",{},0) blobInitializer.ElementCC3D("Width",{},5)
blobInitializer.ElementCC3D("CellSortInit",{},"yes")
blobInitializer.ElementCC3D("Radius",{},40)
# next line is very important and very easy to forget about. It registers XML description and points
# CC3D to the right XML file (or XML tree data structure in this case)
```

CompuCellSetup.setSimulationXMLDescription(cc3d)

Full example:

Demos/PythonOnlySimulationsExamples/cellsort-2D-player-new-syntax.py



Example: Scaling contact energies – advantage of using Python to configure entire simulation

energyScale=10

```
def configureSimulation(sim):
```

```
    global energyScale
```

```
    .
```

```
    .
```

```
    contact=cc3d.ElementCC3D("Plugin",{"Name":"Contact"})
```

```
    contact.ElementCC3D("Energy", {"Type1":"Medium", "Type2":"Medium"},0)
```

```
    contact.ElementCC3D("Energy", {"Type1":"NonCondensing", "Type2":"NonCondensing"},1.6*energyscale)
```

```
    contact.ElementCC3D("Energy", {"Type1":"Condensing", "Type2":"Condensing"},0.2*energyscale)
```

```
    contact.ElementCC3D("Energy", {"Type1":"NonCondensing", "Type2":"Condensing"},1.1*energyscale)
```

```
    contact.ElementCC3D("Energy", {"Type1":"NonCondensing", "Type2":"Medium"},1.6*energyscale)
```

```
    contact.ElementCC3D("Energy", {"Type1":"Condensing", "Type2":"Medium"},1.6*energyscale)
```

It would be a bit awkward (but not impossible) to have same functionality in CC3DML...



Major Plugins and Steppables Available in CompuCell3D



Using different kind of lattices with CompuCell3D

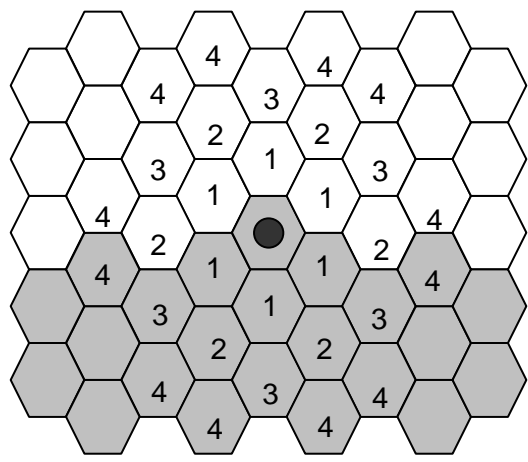
- Current version of CompuCell3D allows users to run simulations on square and hexagonal lattices.
- Other regular geometries (e.g. triangular) can be implemented fairly easily
- Some plugins work on square lattice only - e.g. local connectivity plugin
- Switching to hexagonal lattice requires only one line of code in the Potts section

`<LatticeType>Hexagonal</LatticeType>`

- Model parameters may need to be adjusted when going from one type lattice to another. This is clearly an inconvenience but we will try to provide a solution in the future
- Different lattices have varying degrees of lattice anisotropy. In many cases using lower anisotropy lattice is desired (e.g. foam coarsening simulation on hexagonal lattice). It is also important to check results of your simulation on different kind of lattices to make sure you don't have any lattice-specific effects.
- CompuCell3D makes such comparisons particularly easy

Nearest neighbors in 2D and their Euclidian distances from the central pixel

| | | | | | |
|--|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | 4 | 3 | 4 | |
| | 4 | 2 | 1 | 2 | 4 |
| | 3 | 1 | ● | 1 | 3 |
| | 4 | 2 | 1 | 2 | 4 |
| | | 4 | 3 | 4 | |
| | | | | | |



| | 2D Square Lattice | | 2D Hexagonal Lattice | |
|----------------|---------------------|--------------------|----------------------|----------------------|
| Neighbor Order | Number of Neighbors | Euclidian Distance | Number of Neighbors | Euclidian Distance |
| 1 | 4 | 1 | 6 | $\sqrt{2}/\sqrt{3}$ |
| 2 | 4 | $\sqrt{2}$ | 6 | $\sqrt{6}/\sqrt{3}$ |
| 3 | 4 | 2 | 6 | $\sqrt{8}/\sqrt{3}$ |
| 4 | 8 | $\sqrt{5}$ | 12 | $\sqrt{14}/\sqrt{3}$ |

SquareLattice:

Square in 2D

Cube in 3D

Hexagonal lattice:

Hexagon in 2D

Rhombic dodecahedron in 3D

Cell Attributes

CompuCell3D cells have a default set of attributes:

Volume, surface, center of mass position, cell id etc...

Additional attributes are added during runtime:

List of cells neighbors, polarization vector, Python dictionary or Python list etc...

To keep parameters up-to-date users need to declare appropriate plugins in the CC3DML configuration file.

For example, to make sure surface of cell is up-to-date users need to make sure that `SurfaceTracker` plugin is registered:

Include :

```
<Plugin Name="SurfaceTracker"/>
```


or use Surface plugin which will implicitly call SurfaceTracker

```
<Plugin Name="Surface">
```

```
  <LambdaSurface>0.0</LambdaSurface>
```

```
  <TargetSurface>25.0</TargetSurface>
```

```
</Plugin>
```

But here surface tracking costs you extra calculation of surface energy term:

$$E = \dots + \lambda (s - S_T)^2 + \dots$$

More Flexible Specification of Surface and Volume Constraints

```
<Plugin Name="VolumeFlex">  
  <VolumeEnergyParameters CellType="Amoeba" TargetVolume="150" LambdaVolume="10"/>  
  <VolumeEnergyParameters CellType="Bacteria" TargetVolume="10" LambdaVolume="50"/>  
</Plugin>
```

You may specify different volume and surface constraints for different cell types. This can be done entirely at the XML level.

$$E = \lambda^V_{\tau} (v_{\tau} - V_{\tau})^2$$

Type dependent quantities

$$E = \lambda^S_{\tau} (s_{\tau} - S_{\tau})^2$$


```
<Plugin Name="SurfaceFlex">  
  <SurfaceEnergyParameters CellType="Amoeba" TargetSurface="60" LambdaSurface="10"/>  
  <SurfaceEnergyParameters CellType="Bacteria" TargetSurface="12" LambdaSurface="20"/>  
</Plugin>
```

Even More Flexible Specification of Surface and Volume Constraints

<Plugin Name="VolumeLocalFlex"/>

$$E = \lambda^V_{\sigma} (v_{\sigma} - V_{\sigma})^2$$

<Plugin Name="SurfaceLocalFlex"/>

$$E = \lambda^S_{\sigma} (s_{\sigma} - S_{\sigma})^2$$


Notice that all the parameters are local to a cell. Each cell might have different target volume (target surface) and different λ volume (surface). You will need to use Python to initialize or manipulate those parameters while simulation is running. There is currently no way to do it from XML level. I am not sure it would be practical either.

Tracking Cell Neighbors

Sometimes in your simulation you need to have access to a current list of cell neighbors. CompuCell3D makes this task easy:

```
<Plugin Name="NeighborTracker"/>
```

Inserting this statement in the plugins section of the XML will ensure that at any given time the list of cell neighbors will be accessible to the user. You can access such a list either using C++ or Python. In addition to storing neighbor list, a common surface area of a cell with its neighbors is stored.

Tracking Center of Mass of Each Cell

Including

<Plugin Name="CenterOfMass"/>

statement in your XML code (remember to put it in the correct place) will enable cell centroid tracking:

$$x_{CM}^C = \sum_{i-pixel} x_i \quad y_{CM} = \sum_{i-pixel} y_i \quad z_{CM}^C = \sum_{i-pixel} z_i$$

To get a center of mass of cell you will need to divide centroids by the cell volume:

$$x_{CM} = \frac{x_{CM}^C}{V} \quad y_{CM} = \frac{y_{CM}^C}{V} \quad z_{CM} = \frac{z_{CM}^C}{V}$$

or use simpler syntax in Python

xCM=cell.xCOM

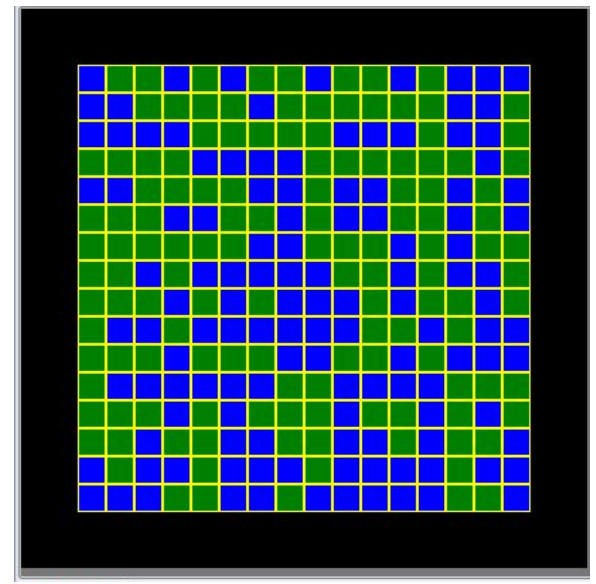
yCM=cell.yCOM

zCM=cell.zCOM

XML initializers - UniformInitializer

You may initialize simple geometries of cell clusters directly from XML

```
<Steppable Type="UniformInitializer">  
  <Region>  
    <BoxMin x="10" y="10" z="0"/>  
    <BoxMax x="90" y="90" z="1"/>  
  
    <Types>Condensing,NonCondensing</Types>  
  
    <Gap>0</Gap>  
    <Width>5</Width>  
  </Region>  
</Steppable>
```



Specify box size and position

Specify cell types – here the box will be filled with cells whose types are randomly chosen (either 1 or 2)

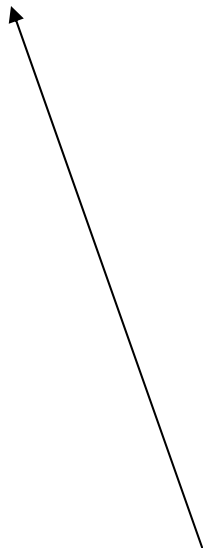
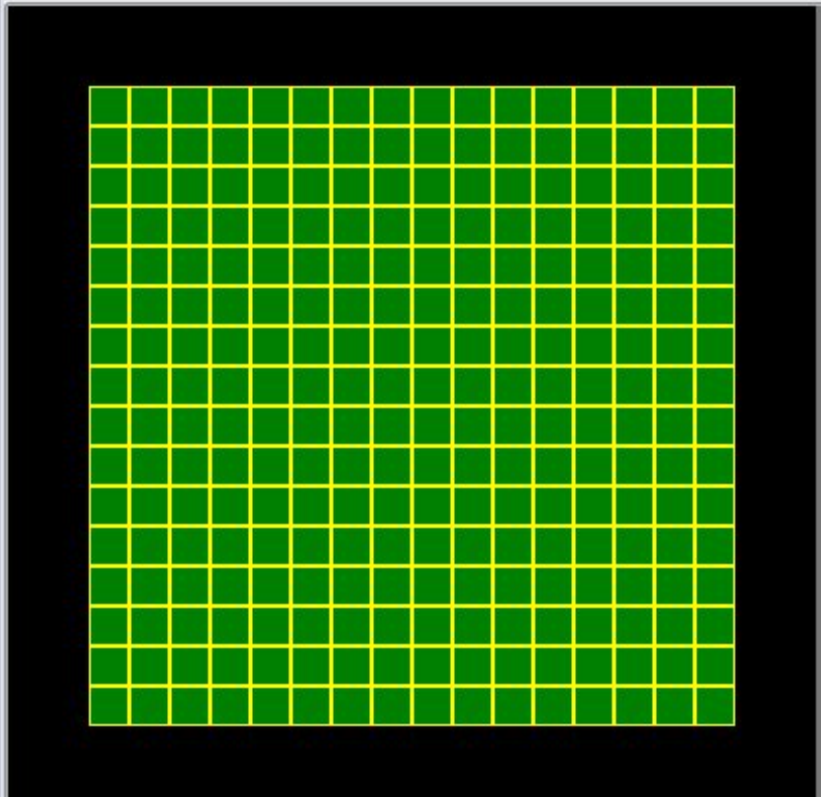
Choose cell size and space between cells



```
<Steppable Type="UniformInitializer">
  <Region>
    <BoxMin x="10" y="10" z="0"/>
    <BoxMax x="90" y="90" z="1"/>

    <Types>Condensing</Types>

    <Gap>0</Gap>
    <Width>5</Width>
  </Region>
</Steppable>
```



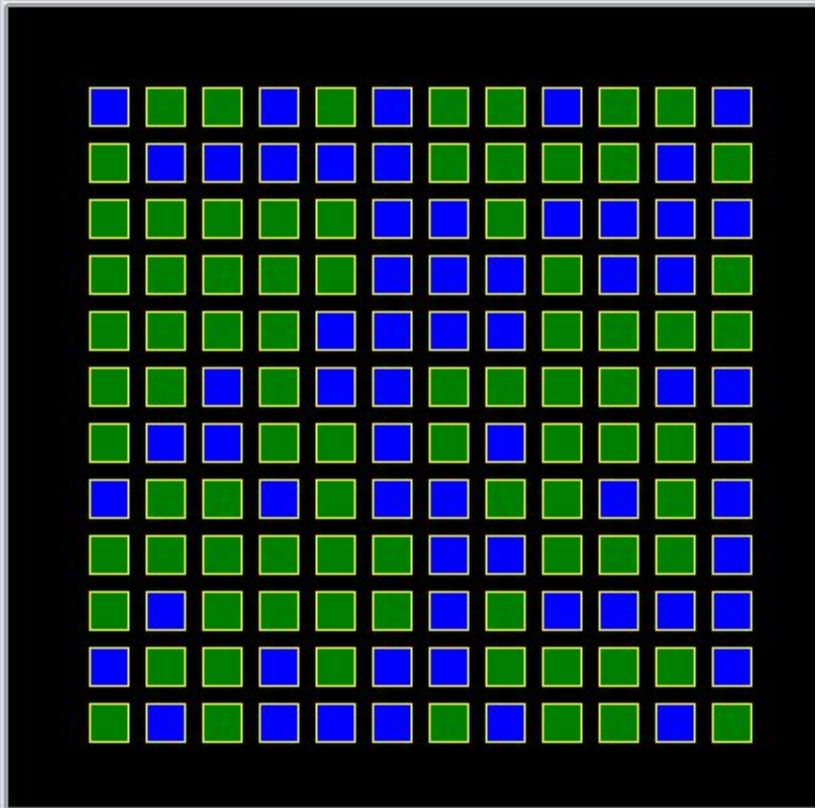
Notice, we have only specified one type (Condensing) thus all the cells are of the same type



```
<Steppable Type="UniformInitializer">
  <Region>
    <BoxMin x="10" y="10" z="0"/>
    <BoxMax x="90" y="90" z="1"/>

    <Types>Condensing,NonCondensing</Types>

    <Gap>2</Gap>
    <Width>5</Width>
  </Region>
</Steppable>
```



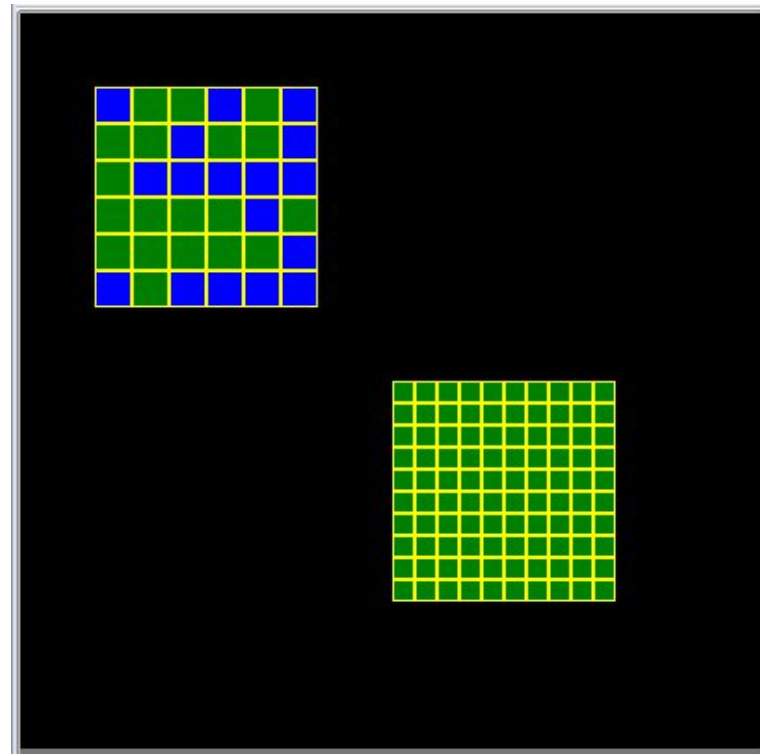
Introducing a gap between cells




```

<Steppable Type="UniformInitializer">
  <Region>
    <BoxMin x="10" y="10" z="0"/>
    <BoxMax x="40" y="40" z="1"/>
    <Gap>0</Gap>
    <Width>5</Width>
    <Types>Condensing,NonCondensing</Types>
  </Region>
  <Region>
    <BoxMin x="50" y="50" z="0"/>
    <BoxMax x="80" y="80" z="1"/>
    <Gap>0</Gap>
    <Width>3</Width>
    <Types>Condensing</Types>
  </Region>
</Steppable>

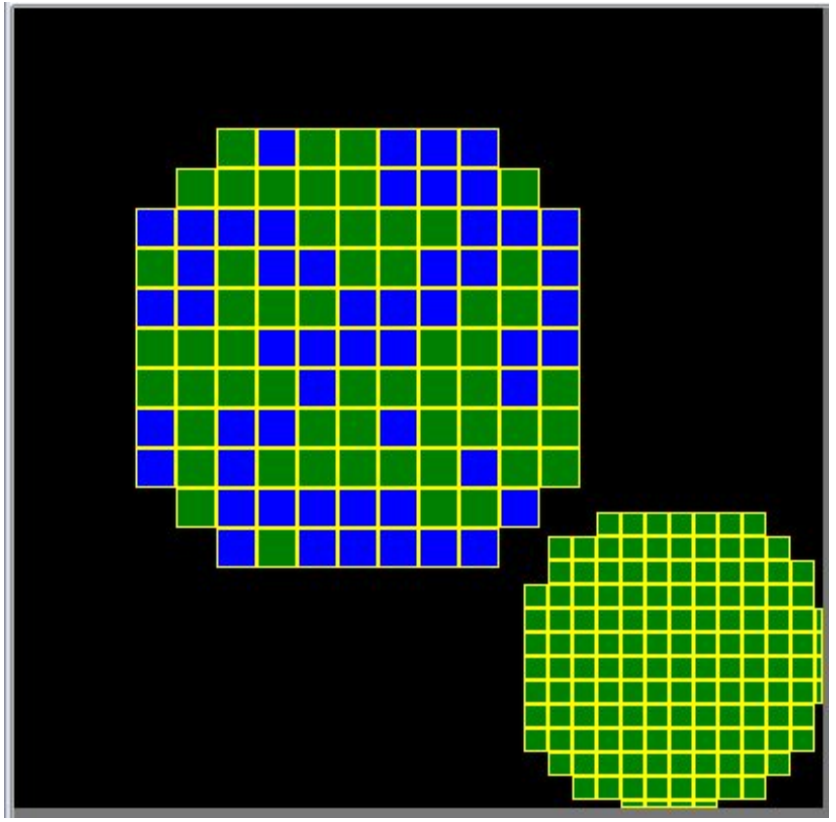
```



Notice, we have defined two regions with different cell sizes and different types



```
<Steppable Type="BlobInitializer">  
  <Region>  
    <Radius>30</Radius>  
    <Center x="40" y="40" z="0"/>  
    <Gap>0</Gap>  
    <Width>5</Width>  
    <Types>Condensing,NonCondensing</Types>  
  </Region>  
  
  <Region>  
    <Radius>20</Radius>  
    <Center x="80" y="80" z="0"/>  
    <Gap>0</Gap>  
    <Width>3</Width>  
    <Types>Condensing</Types>  
  </Region>  
</Steppable>
```



Defining two regions with different cell sizes and different types for BlobInitializer is very similar to the same task with UniformInitializer. There are some new XML tags which differ the two initializers.



Population control using initializers

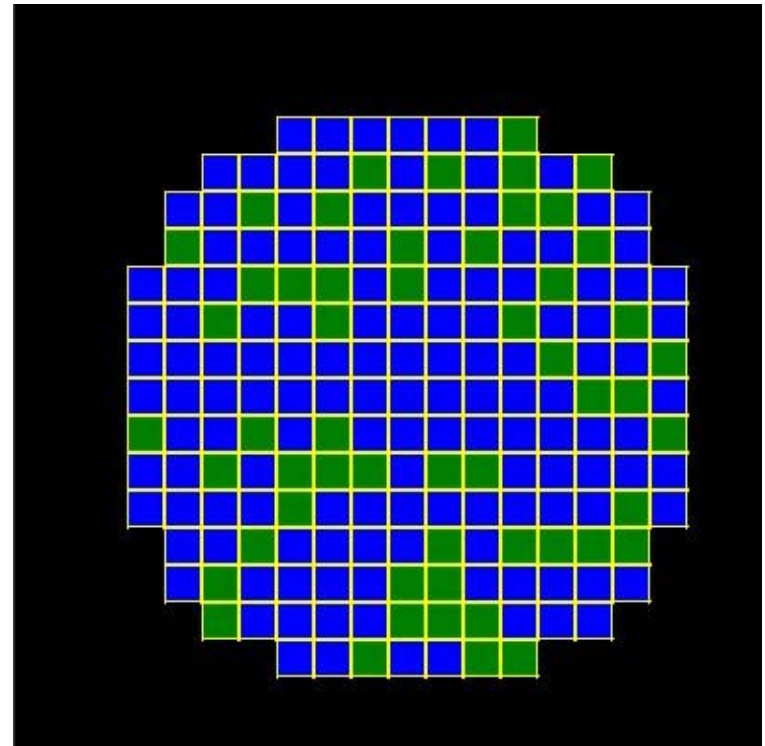
When using BlobInitializer of UniformInitializer you may list same type many times:

`<Types>Condensing,NonCondensing,NonCondensing,NonCondensing</Types>`

The number of cells of a given type will be proportional to the number of times a given type is listed inside the `<Types>` tag.

In the above example the 3/4 of cells will be NonCondensing and 1/4 will be Condensing

```
<Steppable Type="BlobInitializer">
  <Region>
    <Radius>40</Radius>
    <Center x="50" y="50" z="0"/>
    <Gap>0</Gap>
    <Width>5</Width>
    <Types>
      Condensing,
      NonCondensing,
      NonCondensing,
      NonCondensing
    </Types>
  </Region>
</Steppable>
```



Using PIFInitializer

Use PIFInitializer to create sophisticated initial conditions. PIF file allows you to **compose cells from single pixels or from larger rectangular blocks**

The syntax of the PIF file is given below:

```
Cell_id Cell_type x_low x_high y_low y_high z_low z_high
```

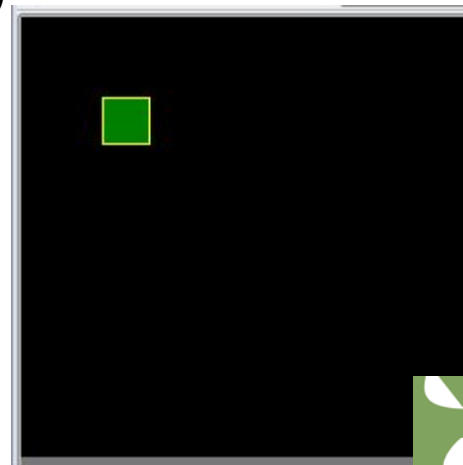
Example (file: amoebae_2D_workshop.pif):

```
0 amoeba 10 15 10 15 0 0
```

This will create rectangular cell with x-coordinates ranging from 10 to 15 (inclusive), y coordinates ranging from 10 to 15 (inclusive) and z coordinates ranging from 0 to 0 inclusive.

```
<Steppable Type="PIFInitializer">  
  <PIFName>amoebae_2D_workshop.pif</PIFName>  
</Steppable>
```

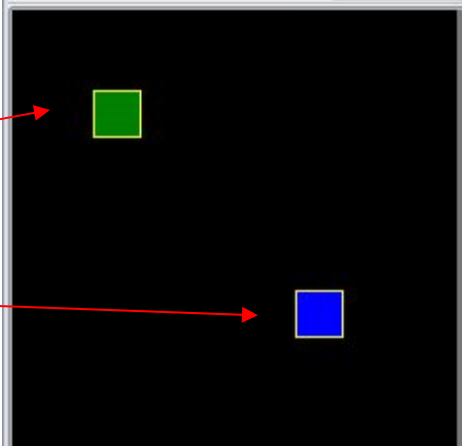
0,0



Let's add another cell:

Example (file: amoebae_2D_workshop.pif):

```
0 Amoeba 10 15 10 15 0 0
1 Bacteria 35 40 35 40 0 0
```

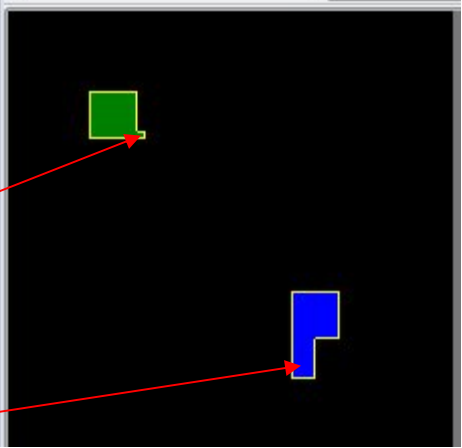


Notice that new cell has different cell_id (1) and different type (Bacterium)

Let's add pixels and blocks to the two cells from previous example:

Example (file: amoebae_2D_workshop.pif):

```
0 Amoeba 10 15 10 15 0 0
1 Bacteria 35 40 35 40 0 0
0 Amoeba 16 16 15 15 0 0
1 Bacteria 35 37 41 45 0 0
```



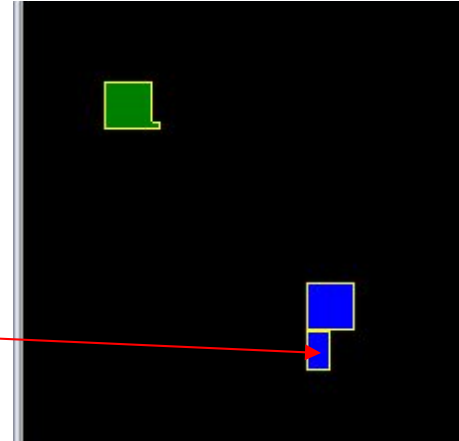
add pixels, start new pif line with existing cell_id (0 or 1 here) and specify pixels.



This is what happens when you do not reuse cell_id

Example (file: amoebae_2D_workshop.pif):

```
0 Amoeba 10 15 10 15 0 0
1 Bacteria 35 40 35 40 0 0
0 Amoeba 16 16 15 15 0 0
2 Bacteria 35 37 41 45 0 0
```



Introducing new cell_id (2) creates new cell.

PIF files allow users to specify arbitrarily complex cell shapes and cell arrangements. The drawback is, that typing PIF file is quite tedious task and , not recommended. Typically PIF files are created using scripts.

In the future release of CompuCell3D users will be able to draw on the screen cells or regions filled with cells using GUI tools. Such graphical initialization tools will greatly simplify the process of setting up new simulations. This project has high priority on our TO DO list.



PIFDumper - yet another way to create initial condition

PIFDumper is typically used to output cell lattice every predefined number of MCS. It is useful because, you may start with rectangular cells, “round them up” by running CompuCell3D , output cell lattice using PIF dumper and reload newly created PIF file using PIFInitializer.

```
<Steppable Type="PIFDumper" Frequency="100">  
  <PIFName>amoebae</PIFName>  
</Steppable>
```

Above syntax tells CompuCell3D to store cell lattice as a PIF file every 100 MCS.

The files will be named *amoebae.100.pif* , *amoebae.200.pif* etc...

To reload file , say *amoebae.100.pif* use already familiar syntax:

```
<Steppable Type="PIFInitializer">  
  <PIFName>amoebae.100.pif</PIFName>  
</Steppable>
```



See presentation by Mitja Hmeljak



Basic facts

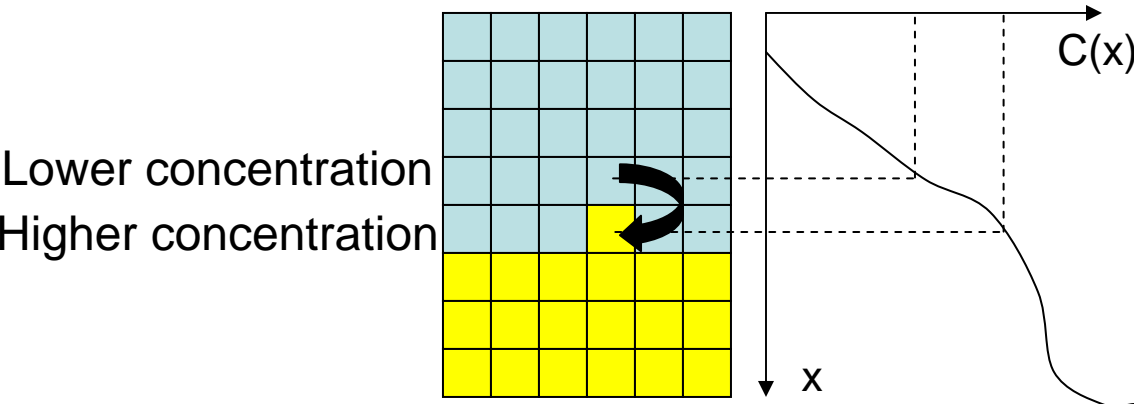
- Chemotaxis is defined as cell motion induced by a presence (gradient) of a chemical.
- In GGH formalism chemotaxis is implemented as a spin copy bias which depends on chemical gradient.
- Chemotaxis was first introduced to GGH formalism by Paulien Hogeweg from University of Utrecht, Netherlands
- In CompuCell3D Chemotaxis plugin provides wide range of options to support different modes of chemotaxis.
- Chemotaxis plugin requires the presence of at least one concentration field. The fields can be inserted into CompuCell3D simulation by means PDE solvers or can be created, initialized and managed explicitly from the Python level



Chemotaxis Term – Most Basic Form

$$\Delta E_{chem} = -\lambda(c(x_{destination}) - c(x_{source}))$$

If concentration at the spin-copy destination pixel ($c(x_{destination})$) is higher than concentration at the spin-copy source ($c(x_{source})$) AND λ is positive then ΔE is negative and such spin copy will be accepted. The cell chemotacts up the concentration gradient



Chemorepulsion can be obtained by making λ negative



Alternative Formulas For Chemotaxis Energy

$$\Delta E_{chem} = -\lambda \left(\frac{c(x_{destination})}{a + c(x_{destination})} - \frac{c(x_{source})}{a + c(x_{source})} \right)$$

$$\Delta E_{chem} = -\lambda \left(\frac{c(x_{destination})}{a \cdot c(x_{destination}) + 1} - \frac{c(x_{source})}{a \cdot c(x_{source}) + 1} \right)$$



$$\Delta E_{chem} = -\lambda(c(x_{destination}) - c(x_{source}))$$

```

<Plugin Name="Chemotaxis">
  <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF">
    <ChemotaxisByType Type="Amoeba" Lambda="300"/>
    <ChemotaxisByType Type="Bacteria" Lambda="200"/>
  </ChemicalField>
  <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF4">
    <ChemotaxisByType Type="Amoeba" Lambda="-300"/>
  </ChemicalField>
</Plugin>

```

Notice , that different cell types may have different chemotactic properties. For more than 1 chemical fields the change of chemotaxis energy expression is given below:

$$\Delta E_{chem} = \sum_{i-field} -\lambda_i(c_i(x_{destination}) - c_i(x_{source}))$$



$$\Delta E_{chem} = -\lambda(c(x_{destination}) - c(x_{source}))$$

```

<Plugin Name="Chemotaxis">
  <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF">
    <ChemotaxisByType Type="Amoeba" Lambda="300"/>
    <ChemotaxisByType Type="Bacteria" Lambda="200"/>
  </ChemicalField>
  <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF4">
    <ChemotaxisByType Type="Amoeba" Lambda="-300" SaturationCoef="2.0"/>
  </ChemicalField>
</Plugin>

```

$$\Delta E_{chem} = -\lambda \left(\frac{c(x_{destination})}{a + c(x_{destination})} - \frac{c(x_{source})}{a + c(x_{source})} \right)$$



$$\Delta E_{chem} = -\lambda(c(x_{destination}) - c(x_{source}))$$

```
<Plugin Name="Chemotaxis">
  <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF">
    <ChemotaxisByType Type="Amoeba" Lambda="300"/>
    <ChemotaxisByType Type="Bacteria" Lambda="200"/>
  </ChemicalField>
  <ChemicalField Source="FlexibleDiffusionSolverFE" Name="FGF4">
    <ChemotaxisByType Type="Amoeba" Lambda="-300" SaturationLinearCoef="2.0"/>
  </ChemicalField>
</Plugin>
```

$$\Delta E_{chem} = -\lambda \left(\frac{c(x_{destination})}{a \cdot c(x_{destination}) + 1} - \frac{c(x_{source})}{a \cdot c(x_{source}) + 1} \right)$$



- CompuCell3D has built-in diffusion, reaction diffusion and advection diffusion PDE solvers. Those are, probably most frequently used solver in GGH modeling.
- CompuCell3D uses explicit (**unstable but fast**) method to solve the PDE. Constantly changing boundary conditions practically rule out more robust, but slow implicit solvers.
- Because of instability users should make sure that their PDE parameters do not produce wrong results (which could manifest themselves as “rough” concentration profiles, “insane” concentration values, NaN’s - Not A Number etc...). Future release of CompuCell3D will provide tools to detect potential PDE instabilities.
- Additional solvers can be implemented directly in C++ or using BioLogo. BioLogo is especially attractive because it takes as an input human readable PDE description and generates fast C++ code.
- Typically a concentration from the PDE solver is read by other CompuCell3D modules to adjust cell properties. Currently the best way of dealing with this is through Python interface.



Flexible Diffusion Solver

```
<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.010</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
      <ConcentrationFileName>diffusion_2D.pulse.txt</ConcentrationFileName>
    </DiffusionData>
  </DiffusionField>
</Steppable>
```

Define diffusion field

Define diffusion parameters

Read-in initial condition

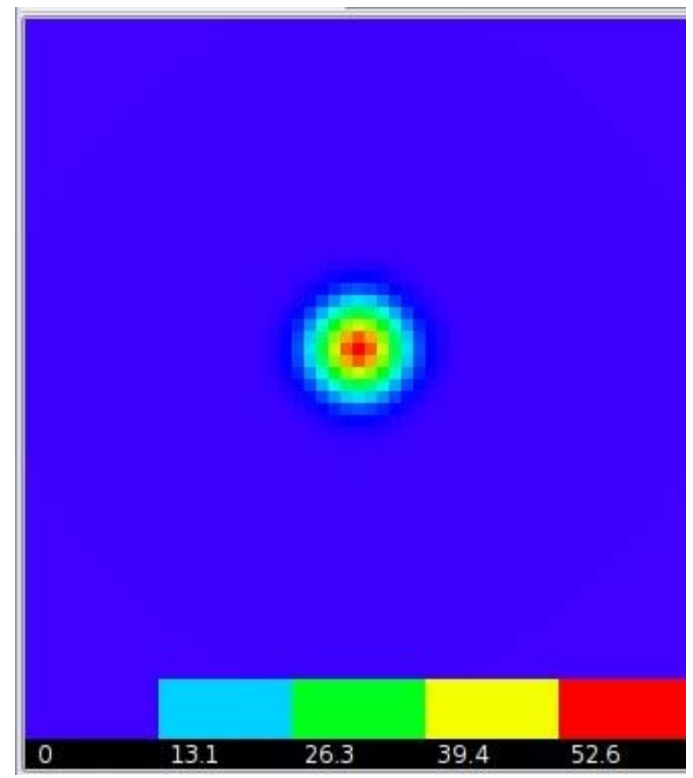
Initial Condition File Format:

x y z concentration

Example:

27 27 0 2000.0

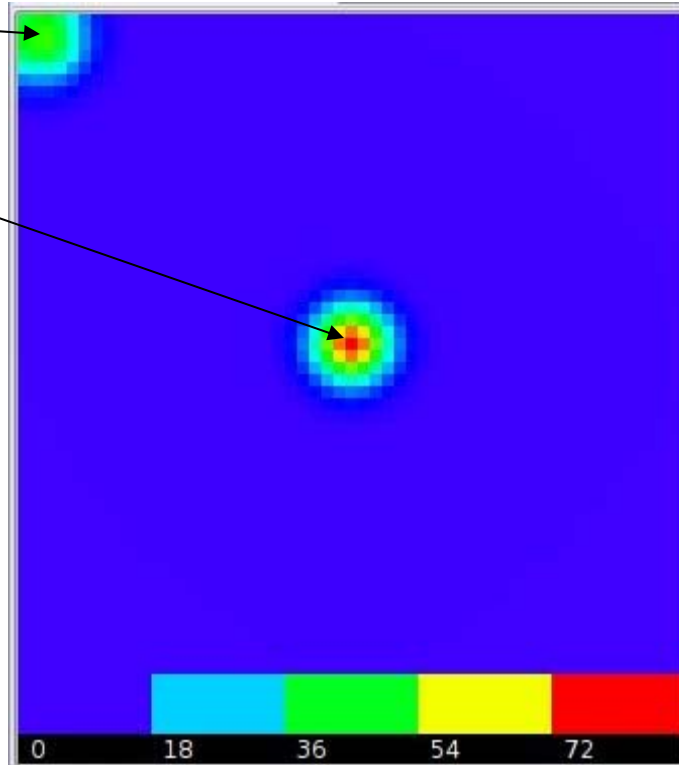
45 0 0.0 ...



Two-pulse initial condition

Initial condition (`diffusion_2D.pulse.txt`):

5 5 0 1000.0
27 27 0 2000.0



```
<Steppable Type="FlexibleDiffusionSolverFE">
```

```
<DiffusionField>
```

```
<DiffusionData>
```

```
<FieldName>FGF</FieldName>
```

```
<DiffusionConstant>0.010</DiffusionConstant>
```

```
<DecayConstant>0.000</DecayConstant>
```

```
<DoNotDiffuseTo>Medium</DoNotDiffuseTo>
```

```
<ConcentrationFileName>diffusion_2D.pulse.txt</ConcentrationFileName>
```

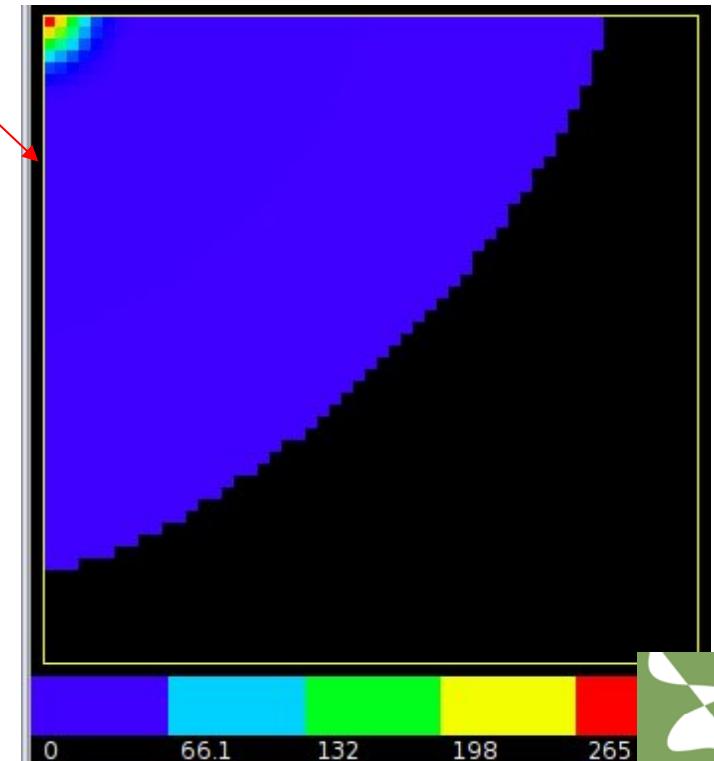
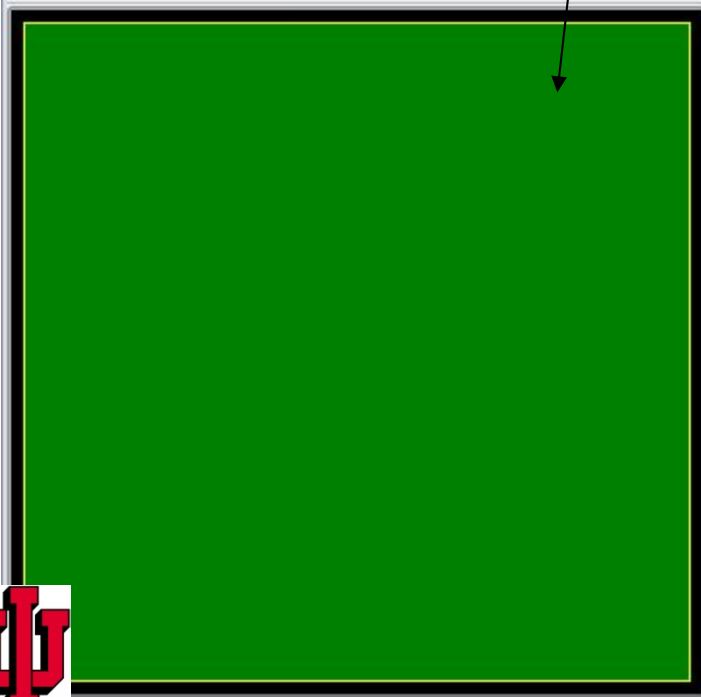
```
</DiffusionData>
```

```
</DiffusionField>
```

```
</Steppable>
```

You may specify diffusion regions

FGF will diffuse inside big cell and will not go to Medium



```
<Steppable Type="FlexibleDiffusionSolverFE">
```

```
<DiffusionField>
```

```
<DiffusionData>
```

```
<FieldName>FGF</FieldName>
```

```
<DiffusionConstant>0.010</DiffusionConstant>
```

```
<DecayConstant>0.000</DecayConstant>
```

```
<DoNotDiffuseTo>Wall</DoNotDiffuseTo>
```

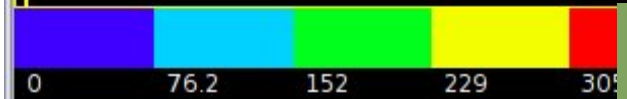
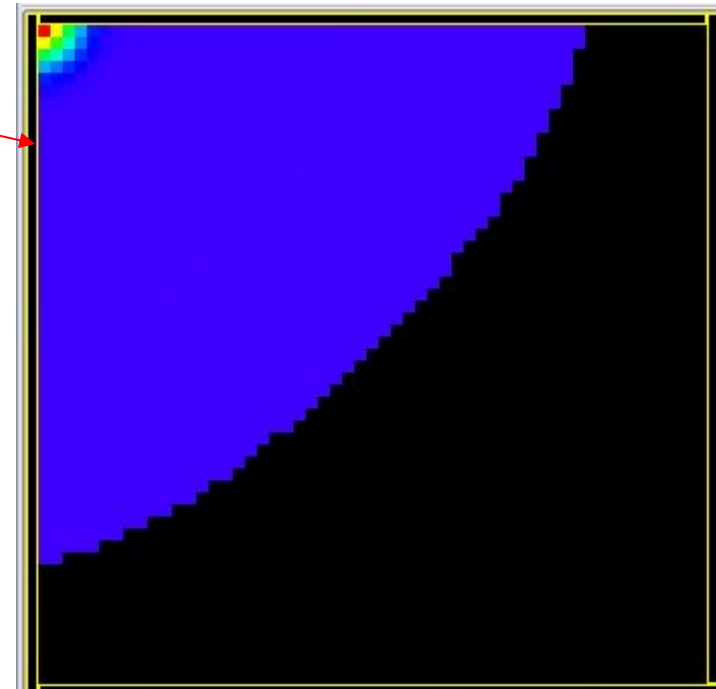
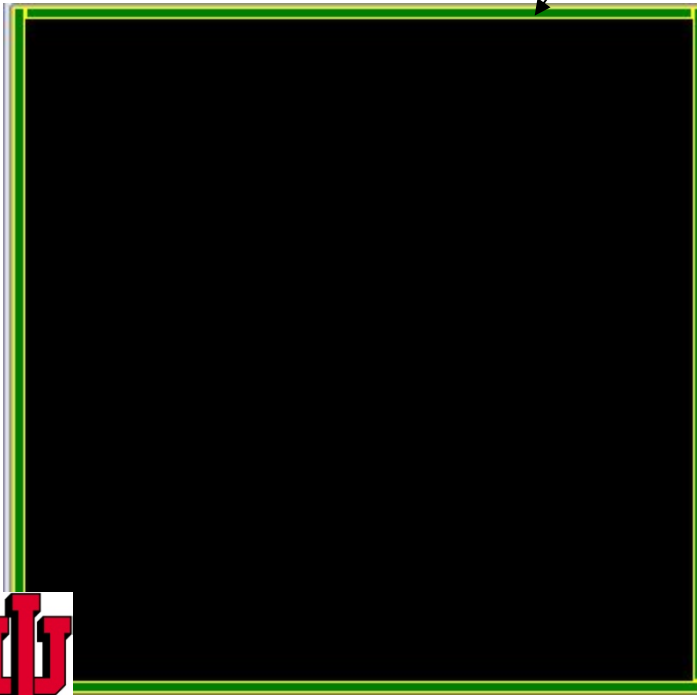
```
<ConcentrationFileName>diffusion_2D_wall.pulse.txt</ConcentrationFileName>
```

```
</DiffusionData>
```

```
</DiffusionField>
```

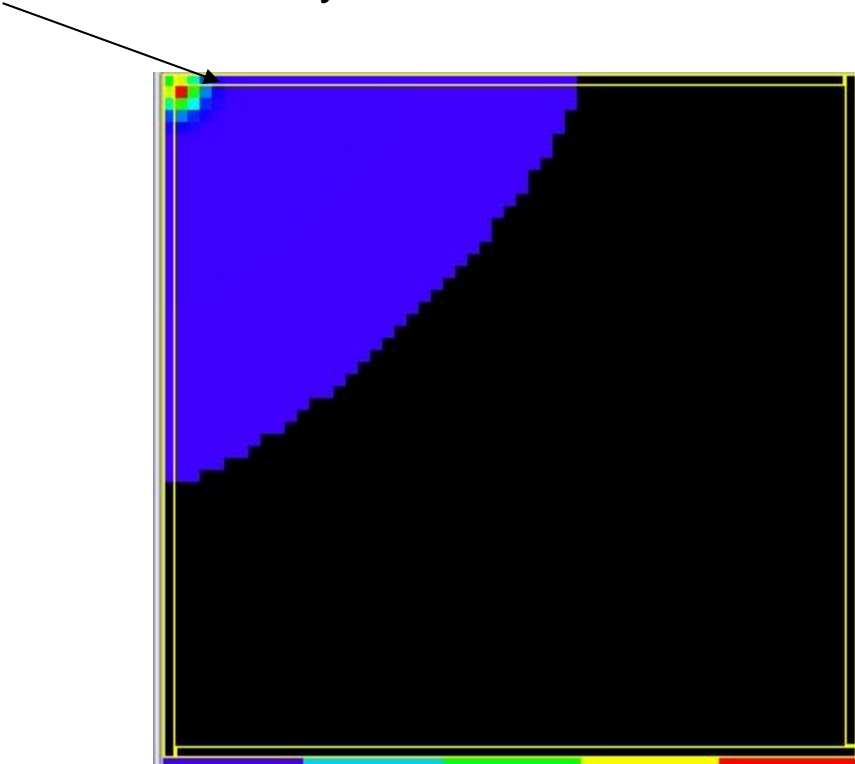
```
</Steppable>
```

FGF will not diffuse to the Wall



```
<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.010</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
      <!--DoNotDiffuseTo>Wall</DoNotDiffuseTo-->
      <ConcentrationFileName>diffusion_2D_wall.pulse.txt</ConcentrationFileName>
    </DiffusionData>
  </DiffusionField>
</Steppable>
```

Now FGF diffuses everywhere



By default PDE solver is called once per MCS. You may call it more often, say 3 times per MCS by including PDESolverCaller plugin:

```
<Plugin Name="PDESolverCaller">  
  <CallPDE PDESolverName="FlexibleDiffusionSolverFE" ExtraTimesPerMC="2"/>  
</Plugin>
```

Notice, that you may include multiple **CallPDE** tags to call different PDESolvers with different frequencies.

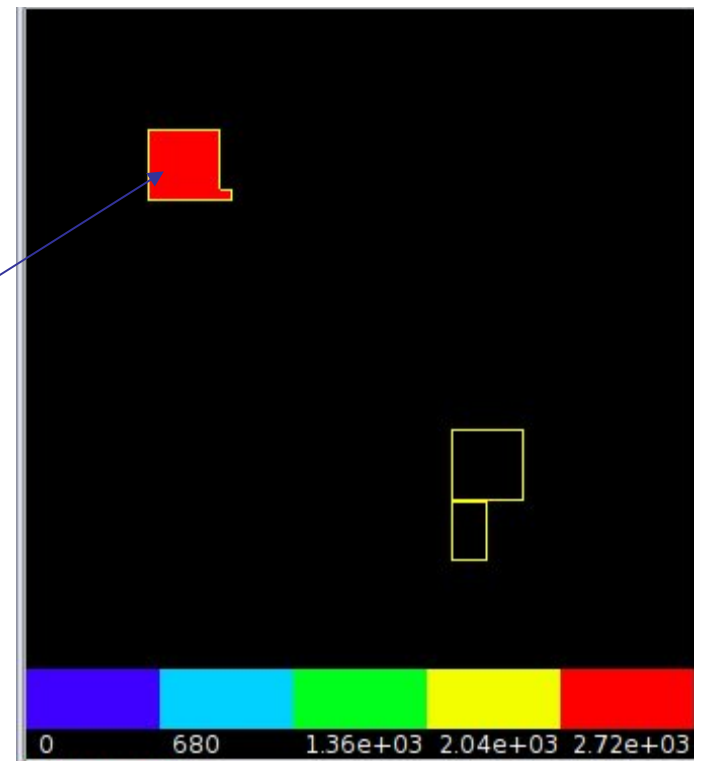
You typically use this plugin to avoid numerical instabilities when working with large diffusion constants



Secretion

CompuCell3D offers several modes for including secretion in your simulations. Let's look at concrete examples:

```
<Steppable Type="FlexibleDiffusionSolverFE">  
  <DiffusionField>  
    <DiffusionData>  
      <FieldName>FGF</FieldName>  
      <DiffusionConstant>0.000</DiffusionConstant>  
      <DecayConstant>0.000</DecayConstant>  
    </DiffusionData>  
    <SecretionData>  
      <Secretion Type="Amoeba">20</Secretion>  
    </SecretionData>  
  </DiffusionField>  
</Steppable>
```

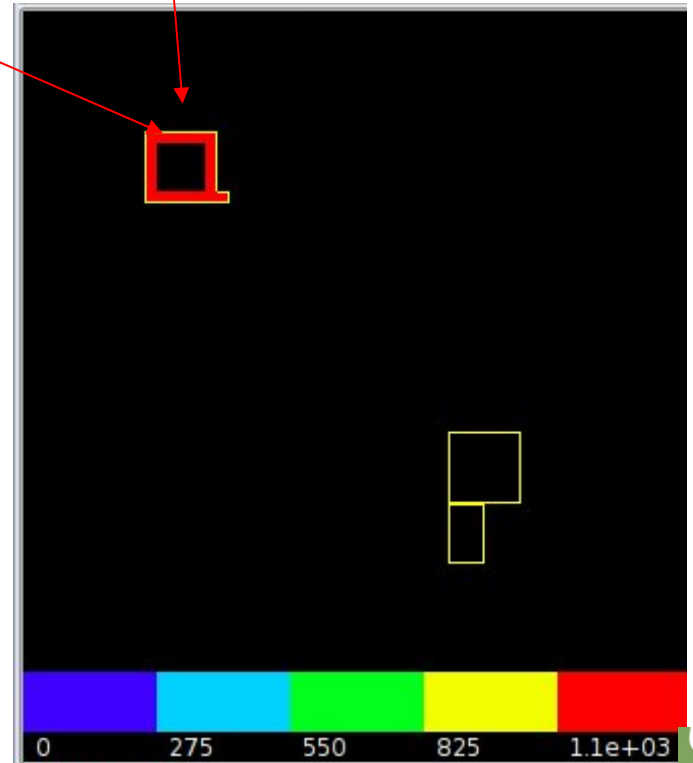


We turned diffusion off and have cells of type Amoeba secrete FGF. Secretion takes place at every pixel belonging to Amoeba cells. At each MCS we increase the value of the concentration at those pixels by 20 units.



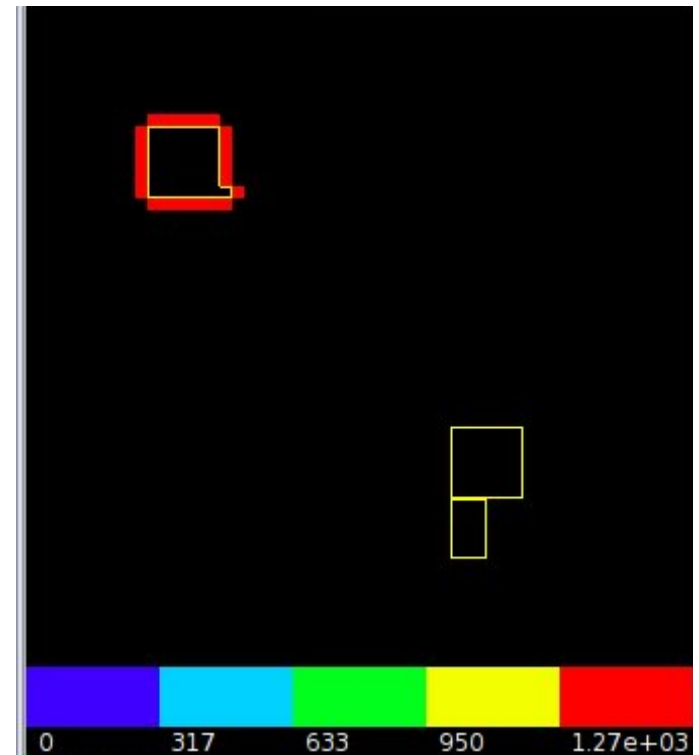
```
<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.000</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
    </DiffusionData>
    <SecretionData>
      <SecretionOnContact Type="Amoeba" SecreteOnContactWith="Medium">20.1</SecretionOnContact>
    </SecretionData>
  </DiffusionField>
</Steppable>
```

Secretion will take place in those pixels belonging to Amoeba cells that have contact with Medium



```
<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.000</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
    </DiffusionData>
    <SecretionData>
      <SecretionOnContact Type="Medium" SecreteOnContactWith="Amoeba">20.1</SecretionOnContact>
    </SecretionData>
  </DiffusionField>
</Steppable>
```

Secretion will take place in those pixels belonging to Medium cells that have contact with Amoeba

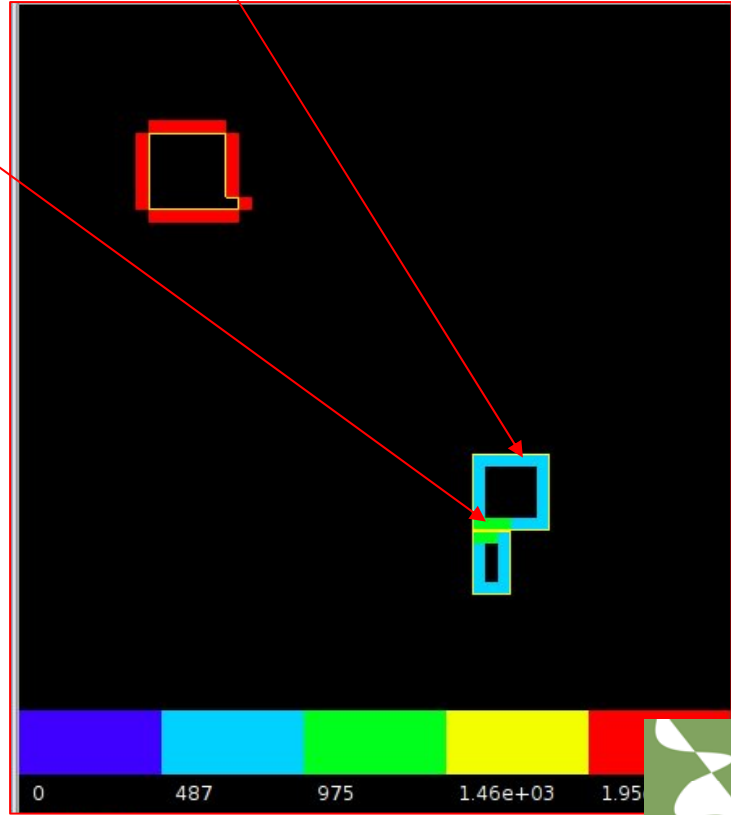



```

<Steppable Type="FlexibleDiffusionSolverFE">
  <DiffusionField>
    <DiffusionData>
      <FieldName>FGF</FieldName>
      <DiffusionConstant>0.000</DiffusionConstant>
      <DecayConstant>0.000</DecayConstant>
    </DiffusionData>
    <SecretionData>
      <SecretionOnContact Type="Medium" SecreteOnContactWith="Amoeba">20.1</SecretionOnContact>
      <SecretionOnContact Type="Bacteria" SecreteOnContactWith="Bacteria">10.1</SecretionOnContact>
      <SecretionOnContact Type="Bacteria" SecreteOnContactWith="Medium">5.1</SecretionOnContact>
    </SecretionData>
  </DiffusionField>
</Steppable>

```

1. Secretion will take place in those pixels belonging to Medium cells that have contact with Amoeba.
2. There will be secretion in pixels of Bacteria cells that have contact with medium.
3. Secretion will also take place in those pixels of bacteria cells that have contact with other bacteria cells



$$\frac{\partial c_1}{\partial t} = D_1 \nabla^2 c_1 + f_1(c_1, c_2, \dots, c_N)$$

$$\frac{\partial c_2}{\partial t} = D_2 \nabla^2 c_2 + f_2(c_1, c_2, \dots, c_N)$$

⋮

$$\frac{\partial c_N}{\partial t} = D_N \nabla^2 c_N + f_N(c_1, c_2, \dots, c_N)$$

Solving general set of above PDE's can be tricky because functions 'f' can have arbitrary form. There are two ways to deal with this problem:

1. For each set of PDE's write new PDE solver. This is not a bad idea if you can do it "on the fly". If you can write a code that automatically generates and compiles PDE solver you will see no performance degradation
2. Use fast math expression parser that will interpret mathematical expressions during run time

CompuCell3D 3.4.1 uses the second solution. The reason was that it was the simplest to implement and also one does not have to bother about compilers installed on users machines. However such PDE solver will not be as fast as the compiled one



Let's consider a simple example

$$\frac{\partial F}{\partial t} = 0.01\nabla^2 F + F - F^3/3 + 0.3 - H$$

$$\frac{\partial H}{\partial t} = 0.01\nabla^2 H + 0.08F - 0.064H + 0.056$$

```
<Steppable Type="ReactionDiffusionSolverFE">
```

```
<DiffusionField>
```

```
<DiffusionData>
```

```
<FieldName>F</FieldName>
```

```
<DiffusionConstant>0.01</DiffusionConstant>
```

```
<ConcentrationFileName>Demos/diffusion/FN.pulse.txt</ConcentrationFileName>
```

```
<AdditionalTerm>F-F*F*F/3+0.3-H</AdditionalTerm>
```

```
</DiffusionData>
```

```
</DiffusionField>
```

```
<DiffusionField>
```

```
<DiffusionData>
```

```
<FieldName>H</FieldName>
```

```
<DiffusionConstant>0.01</DiffusionConstant>
```

```
<AdditionalTerm>0.08*F-0.064*H+0.056</AdditionalTerm>
```

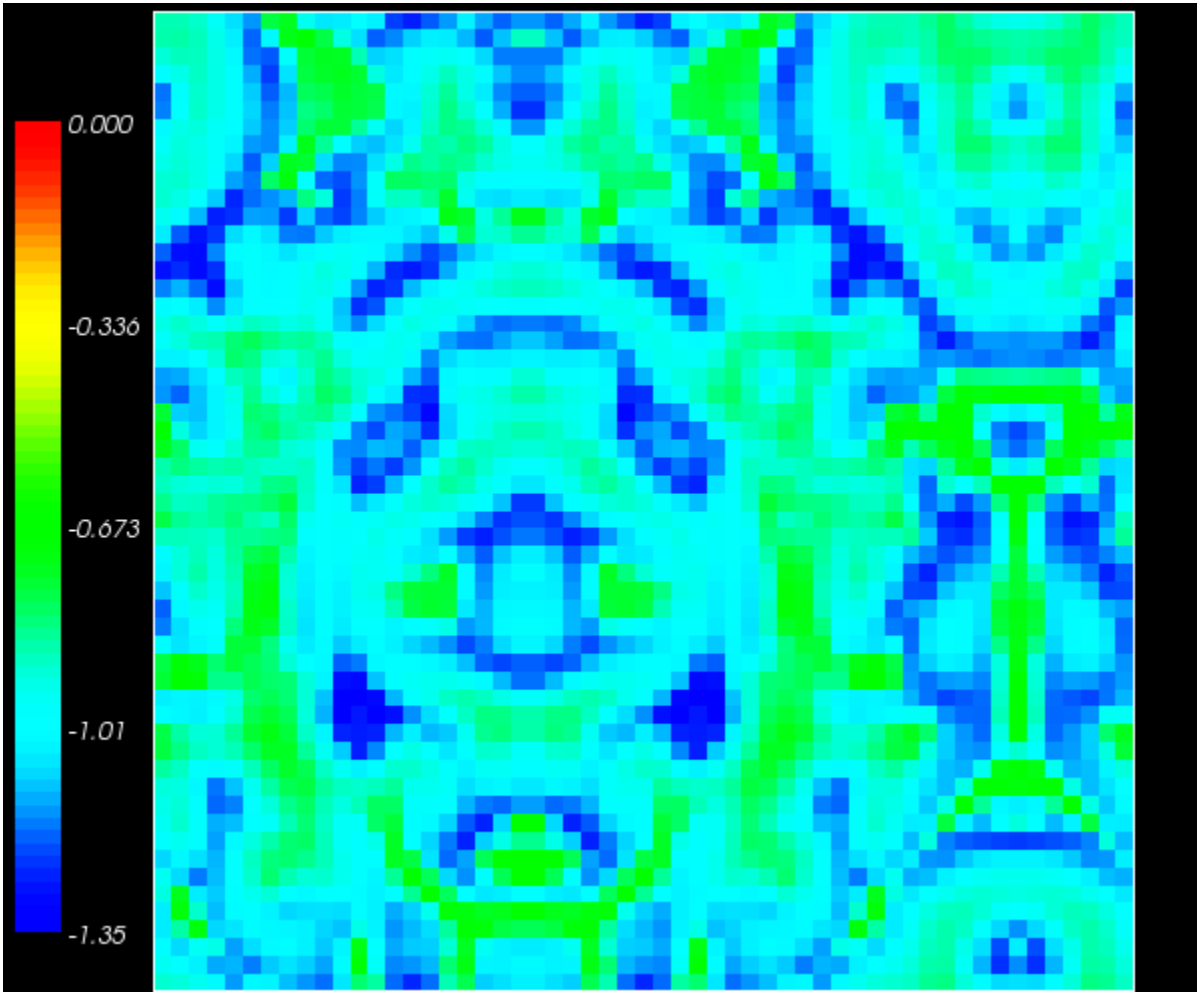
```
</DiffusionData>
```


```
</DiffusionField>
```

```
</Steppable>
```



Functions of F and H are coded using quite naturally looking syntax. The output of the above simulation with periodic boundary conditions may look like



 quite interesting that the slowdown due to interpreting user defined functions is small.



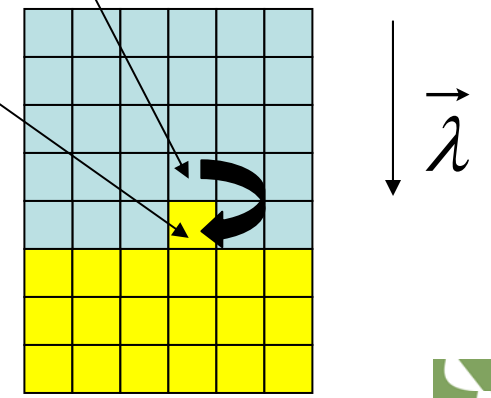
Imposing Directed Motion of Cells

One can impose artificial spin flip bias that would have an effect of moving cell in the direction OPPOSITE to Lambda vector specified below. The magnitude of the lambda vector determines the “amount” of spin copy bias.

```
<Plugin Name="ExternalPotential">  
  <Lambda x="-0.5" y="0.0" z="0.0"/>  
</Plugin>
```

$$\Delta E_{external_potential} = -\vec{\lambda} \cdot (\vec{x}_{destination} - \vec{x}_{source})$$

ΔE will be negative (favoring spin copy) →



Connectivity plugin ensures that **2D cells** are not fragmented and are simply connected. It decreases probability of certain spin flips which are can break connectedness of a cell. Users can specify energy penalty that will be incurred if the spin copy is to break connectedness of the cell.:

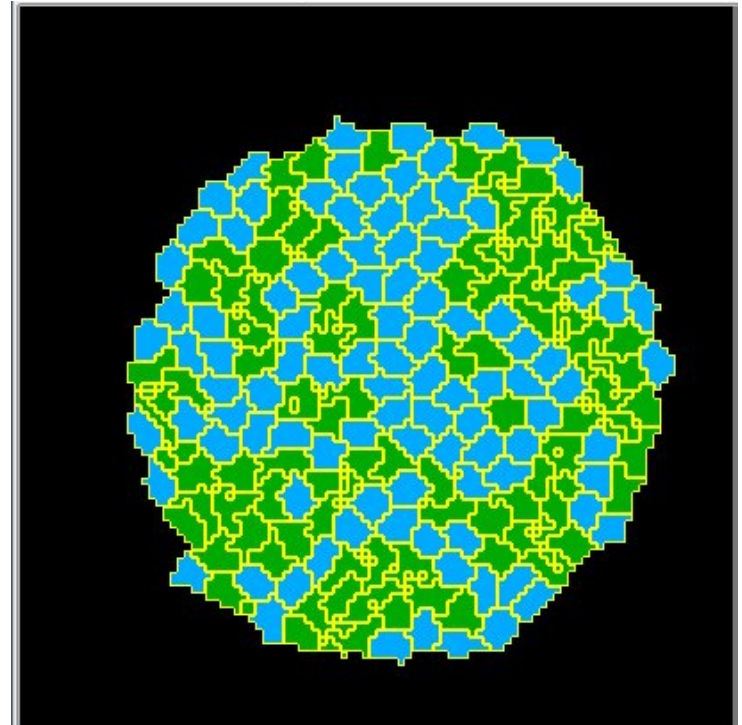
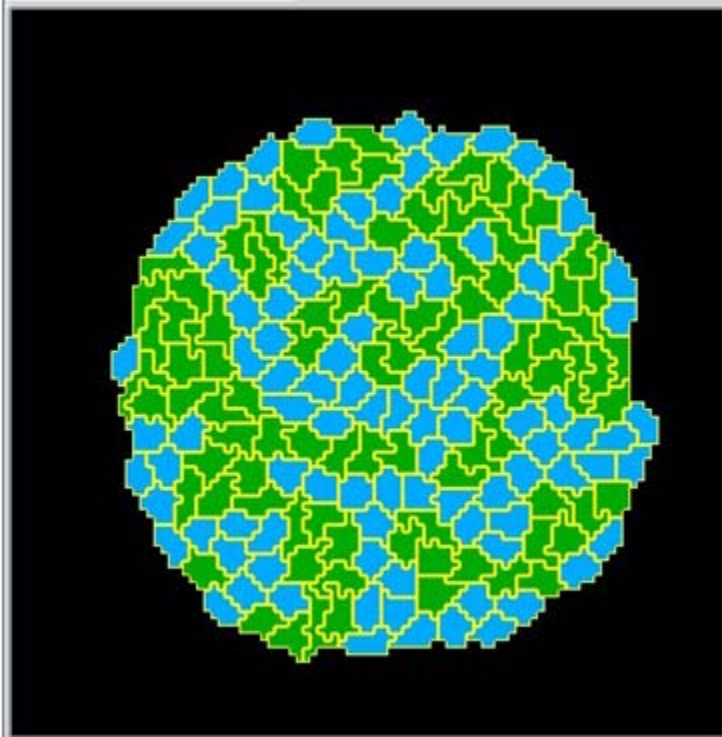
Syntax:

```
<Plugin Name="Connectivity">  
  <Penalty>100000</Penalty>  
</Plugin>
```

Note: this plugin will not work properly with hexagonal lattice



Cell sorting simulation with and without connectivity plugin



Length Constraint Plugin

Length constraint plugin is used to force cells to keep preferred length along cell's longest axis (we assume that cells have elliptical shape):

```
<Plugin Name="LengthConstraint">
```

```
  <LengthEnergyParameters TargetLength="15" LambdaLength="2.0"/>
```

```
</Plugin>
```

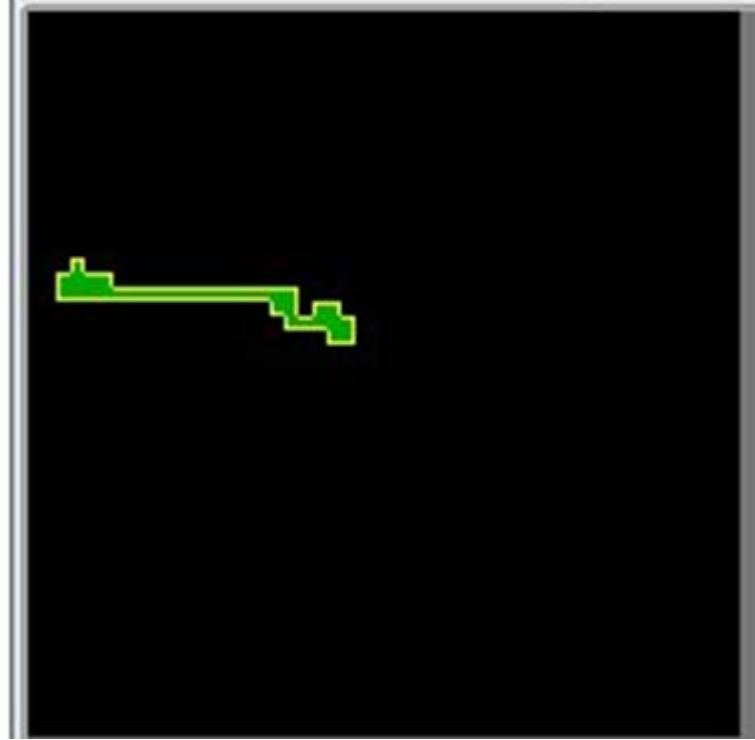
The LambdaLength and TargetLength play similar role to LambdaVolume and TargetVolume from Volume Plugin.

IMPORTANT: Length Constraint Plugin has to be used together with connectivity plugin or else cells might become fragmented. The applicability of the LengthConstraint and Connectivity Plugins is limited to 2D simulations.

For more information see

“Cell elongation is key to in silico replication of in vitro vasculogenesis and subsequent remodeling” by **Roeland M.H. Merks** *et al* Developmental Biology 289 (2006) 44– 54





Note: this plugin will not work properly with hexagonal lattice

