DEVELOPERS' DOCUMENTATION FOR COMPUCELL3D

Version 3.4.1

Authors: Maciej Swat, Trevor Cickovski, James Glazier, Alex Dementosv, Benajmin Zaitlen

Last modified:09/25/09

This manual will guide you through the process of building new plugins and modules for CompuCell. Because of the modular design of the CompuCell3D you can get started quite quickly as in most cases you do not need to understand the entire code of the CompuCell3D. Of course, occasionally you might be forced to look up some parts of the code, but once you get some experience with plugin development understanding rest of CompuCell3D code will be easy.

Prerequisites:

We assume that you know C++ . To be more precise, you need to understand polymorphism, how virtual functions work and have basic knowledge of templates. We also assume that you are familiar with Cellular Potts Model, that is that you understand what CPM is all about and how it works.

We will try to demonstrate how to develop new plugins by using a concrete example. This is probably the best way to introduce you to CompuCell3D development. Before we go there it is probably a good idea to understand how CompuCell3D works.

CompuCell3D code basics

One of the most important classes in CompuCell3D , called Potts3D implements the Metropolis algorithm. The method that has the implementation of it has the following API:

```
virtual unsigned int metropolis(const unsigned int steps, const double temp);
```

If you look at the code inside this function you will discover that it performs many spin-flip (or pixel flips, if you prefer) attempts. Each flip consists of choosing randomly a point in the 3D lattice, lets call it *pt*. Then we look up a list of neighboring pixels of `pt` and randomly pick one of such pixels , let's call it `changePixel`. Now, what's going to happen next is that we will try to assign to `changePixel` a value (actually it is a pointer to a cell object) from `pt`. Before we allow to assign the value we need to check what energy change such a reassignment would cause. Looking in the code we find a call to a Potts3D's method called:

```
virtual double changeEnergy(Point3D pt, const CellG *newCell, const CellG
*oldCell)
```

In a nutshell, this method calls energy functions plugins and adds all the energy contributions from each of those plugins and returns as a result overall energy change due to the proposed spin-flip.

Actual call to this method looks as follows:

```
double change = changeEnergy(changePixel, cell, cellFieldG->get(changePixel));
```

This call tells you that the only piece of information that energy function plugins get from Potts are:

1) change point, i.e a location of the lattice at which spin reassignment will take place
2) pointer to *newCell* – this will be a pointer to a cell object that will be assigned to a *changePixel* if the spin flip gets accepted
3) pointer to *oldCell* – this is a pointer to a cell object that is currently (i.e. before spin flip) at the location of the *changePixel*

As you will see later, you can access more Potts3D information from the plugin, but this will be done in a slightly different way (also very straightforward)

Another very important type of plugins in CompuCell3D are field watchers. What they do is , every time a spin-flip takes place they update cell attributes such as volume, surface, list of neighbors, etc... depending what field watchers user has requested for his/her simulation.

The API for field watcher most important function (i.e. *field3DChange*) looks as follows:

```
virtual void field3DChange(const Point3D &pt, CellG *newCell, CellG *oldCell);
```

Again, as it was the case for changeEnergy function presented earlier we pass to field3Dchange same objects. It should be noted though that field3DChange is not called directly from Potts3D's metropolis method. It is called through WatchableField3D API

Another type of object that is called from Potts3D, although is not used that often is Stepper. It another type of plugin that is called every time spin-flip attempt. The difference is that it is called after all energy function and field3DChange functions have been called. We will explain why we need Steppers later when we will discuss implementation of the mitosis. For now, you should remember that there is a way to call a plugin *after* all registered plugins have been called, for a given spin flip attempt.

So far we have been dealing with plugins than are being called either every spin-flip attempt or every spin-flip. There is another category of modules of CompuCell3D which is being called every Monte Carlo Step or every number of Monte Carlo Step. Quick reminder, A Monte Carlo step consist of many spin-flip attempts (usually, but not always, equal to a number of lattice sites).

Steppable objects have three main methods that are being called by Compuell3D kernel:

```
1) void start()
2) void step(unsigned int _mcs)
3) void finish()
```

start() and *finish()* usually perform pre-/post-initialization (although there are other initialization methods that are also called, and we will discuss them later while workinig out a specific example).

step() is called every Monte Carlo step (or every user-predefined number of MCS's) and typically does various kind of “maintenance” or solves PDE's . Steppables are also used to initialize cell field.

Below you can find a pseudo-code for the CompuCell3D

```
Initialize Plugins
Initialize Steppables
Additional Initialization of Plugins
Additional Initialization of Steppables
Start Simulation
        for each Monte Carlo Step :
                for every spin flip:
                        pick random points
                        calculate energy functions and sum them (will call changeEnergy for every
                        registered plugin)
                        check if the spin flip gets accepted
                        do spin reassignment (at this step field3DChange functions will be called)
                        call steppers
                call steppables:
Finish Simulation
```

The above information should be sufficient to start coding new modules. In the following subsections we will work out examples of particular plugins. At the end of of this manual we will provide how to set up build system (makefiles etc... for CompuCell3D).

Note: *Plugins written for version 3.2.* of the CompuCell3D are NOT backward compatible with the version 3.3.0. The major change for the new version was done in parsing of the XML files. This significantly simplifies the code makes the plugin development much easier.*

Developing Plugins

Probably the best way to learn how to develop a plugin is to study an example. Here we will explain how to develop several plugins. The easiest plugin to learn and understand is a SimpleVolume plugin from DeveloperZone just to get feeling of what it is. Then we will consider the *VolumeTracker* plugin from the plugins directory that that tracks the cell volumes.

SimpleVolume Plugin

Let's open up the header file *SimpleVolumePlugin.h* from the directory DeveloperZone/SimpleVolume and consider the class SimpleVolumePlugin.

[SimpleVolumePlugin.h]

```
#ifndef SIMPLEVOLUMEPLUGIN_H
#define SIMPLEVOLUMEPLUGIN_H

#include <CompuCell3D/Plugin.h>
#include <CompuCell3D/Potts3D/Stepper.h>
#include <CompuCell3D/Potts3D/EnergyFunction.h>
#include <CompuCell3D/Potts3D/CellGChangeWatcher.h>
#include <CompuCell3D/Potts3D/Cell.h>
#include <CompuCell3D/dllDeclarationSpecifier.h>
#include <vector>
#include <string>
```

```
class CC3DXMLElement;

namespace CompuCell3D {
  class Potts3D;
  class CellG;

  class DECLSPECIFIER SimpleVolumePlugin : public Plugin , public EnergyFunction
  {
      Potts3D *potts;
      C3DXMLElement *xmlData;
      double targetVolume;
      double lambdaVolume;

  public:
      SimpleVolumePlugin():potts(0){};
      virtual ~SimpleVolumePlugin(){};

      //EnergyFunction interface
      virtual double changeEnergy(const Point3D &pt, const CellG *newCell,const
CellG *oldCell);

      // SimObject interface
      virtual void init(Simulator *simulator, CC3DXMLElement *_xmlData);
      virtual void update(CC3DXMLElement *_xmlData, bool _fullInitFlag=false);
      virtual std::string steerableName();
      virtual std::string toString();
  };
};
#endif
```

Note: *DECLSPECIFIER is a macro that is needed to compile dlls on Windows systems. On Linux/OSX this macro is automatically replaced with empty string.*

The purpose of the SimpleVolume plugin is to track cell volumes and add the volume energy. The class `SimpleVolumePlugin` inherits from two superclasses `Plugin` and `EnergyFunction`. Any plugin that you will develop should inherit from Plugin class. Additionally you can inherit also from other superclasses depending on the purpose of your class. For example, the class `NeighborTrackerPlugin` inherits also from `CellGChangeWatcher` as it tracks the cell neighbors and stores them in a list. In our case `SimpleVolumePlugin` inherits also from `EnergyFunction` because it calculates the change in the energy and adds and subtracts pixels from the cell depending on this value.

The most important task in our `SimpleVolumePlugin` development is to implement the `changeEnergy` function.

Let's look at the actual implementation of this function:

```
double SimpleVolumePlugin::changeEnergy(const Point3D &pt,
                                const CellG *newCell,
                                const CellG *oldCell)
{
```

```
      /// E = lambda * (volume - targetVolume) ^ 2
      double energy = 0.0;

      if (oldCell == newCell) return 0;

      //as in the original version
      if (newCell){
            energy += lambdaVolume *
                  (1 + 2 * (newCell->volume - targetVolume));
      }
      if (oldCell){
            energy += lambdaVolume *
                  (1 - 2 * (oldCell->volume - targetVolume));
      }

      return energy;
}
```

As it was discussed earlier the only piece of information that is passed to this function is the point of the lattice at which spin reassignment takes place and pointer to the current cell object (`oldCell`) at this point and pointer to the cell object (`newCell`) that will be assigned if the spin flip is accepted.

Notice that in the code we have 'if' statement that goes like this:

```
if (newCell)
```
or
```
if (oldCell)
```

Before accessing the object it is important that we make sure that a pointer is non-NULL. In CompuCell3D implementation there might be many lattice points which have null pointer. We call them 'Medium' . This is special way to mark non-cell objects. Always, check if you are not accessing null pointer when trying to access cell , as otherwise you will get segmentation fault error.

OK, back to the volume energy. The volume energy has the expression $E=\lambda(V-V_t)^2$ that depends on two parameters: λ and V_t stored in our `SimpleVolumePlugin` class in the attributes `lambda` and `targetVolume`. Of course you can come up with some other model and other expression for the energy function but then you will need to rewrite the `changeEnergy()` function that would redefine would suit the energy change to your model and may be introduce some other parameters. To pass the values for `lambda` and `targetVolume` you will need to specify them in the XML tags as shown in the snippet below.

```
<Plugin Name="SimpleVolume">
   <TargetVolume>25</TargetVolume>
   <LambdaVolume>2.0</LambdaVolume>
</Plugin>
```

For the `SimpleVolumePlugin` these parameters are the same for all cells. But you might want to make them different for different cell types. The example of how to do that is provided in the class `VolumePlugin` and the actual model that you can play with can be found in *dicty_try_40x40.xml*

under Demos directory.

Note: *The* `VolumePlugin` *class that we used to give as an example in the Developers' Manual in previous versions of the CompuCell3D now became more complex because it now includes several plugins:* `Volume`, `VolumeFlex` *and* `VolumeLocalFlex`.

Let's learn how we got the formulas for the `changeEnergy()` that appear in the code. Recall that the volume energy function is given by expression:

$$E = \lambda (V - V_t)^2$$

Now, if a cell is going to gain extra pixel due to a spin flip (this will be new cell) that its change in volume energy is given by:

$$\Delta E_{new} = E_{after} - E_{before} = \lambda (V + 1 - V_t)^2 - \lambda (V - V_t)^2 = \lambda (1 + 2(V - V_t))$$

Similarly, if cell would lose one pixel (`oldCell`) due to a proposed spin flip than change of volume energy for this cell is given by:

$$\Delta E_{old} = E_{after} - E_{before} = \lambda (V - 1 - V_t)^2 - \lambda (V - V_t)^2 = \lambda (1 - 2(V - V_t))$$

Now, overall energy change due to a spin flip is a sum of ΔE_{old} and ΔE_{new} :

$$\Delta E = \Delta E_{old} + \Delta E_{new}$$

We hope you were able to match formulas in the code snippet with our derivation.

After we wrote the `changeEnergy()` function now we can move on and consider other member functions in the `SimpleVolumePlugin` class. Before we start our simulation we need to initialize it. The virtual member function `init(Simulator *simulator, CC3DXMLElement *_xmlData)` does just that. It passes the `Simulator` and `CC3DXMLElement` pointers to initialize the attributes `potts` and `xmlData`. Let's look at the implementation of this member function.

```
void SimpleVolumePlugin::init(Simulator *simulator, CC3DXMLElement *_xmlData)
{
      potts = simulator->getPotts();
      bool pluginAlreadyRegisteredFlag;
      Plugin *plugin =  Simulator::pluginManager.get( "VolumeTracker",
&pluginAlreadyRegisteredFlag);

      //this will load VolumeTracker plugin if it is not already loaded
      if(!pluginAlreadyRegisteredFlag)
            plugin->init(simulator);
      potts->registerEnergyFunctionWithName(this,toString());
      simulator->registerSteerableObject(this);
```

```
        xmlData=_xmlData;
        update(_xmlData);
}
```

First what this method does is that it assigns the *potts* attribute from *Simulator* object which passed as a parameter to the *init()* member function.

```
        potts = simulator->getPotts();
```

The *potts* attribute of the class *Potts3D* is one of main classes in the CompuCell3D. It implements the metropolis algorithm of spin flip, registers cell change watchers, sets the neighbor order and many other important operations.

We are sure you are puzzled why we have the next following lines:

```
bool pluginAlreadyRegisteredFlag;
Plugin *plugin =  Simulator::pluginManager.get( "VolumeTracker",
&pluginAlreadyRegisteredFlag);

//this will load VolumeTracker plugin if it is not already loaded
if(!pluginAlreadyRegisteredFlag)
        plugin->init(simulator);
```

As you may infer from the comment, this call loads another plugin that is needed by volume plugin. the *VolumeTracker*. *VolumeTracker* as its name suggests tracks changes and updates cell volume due to spin flips. Most important function there is *field3DChange()*. Next line registers *SimpleVolumePlugin* object so that it will be called every time a spin flip attempt has been made. To register we can also use the following statement:

```
potts->registerEnergyFunctionWithName(this, toString());
```

The method *toString()* returns the name of the plugin which in our example will be **"SimpleVolume".** Each plugin and steppable is register and searched by name in the *Simulator* object so make sure that you include the name of your plugin in the *toString()* method.

```
simulator->registerSteerableObject(this);
```

The next important step is to assign the pointer to the *xmlData* attribute. After that the *xmlData* will refer to the parameters specified in the XML file. This has to be included in each plugin and steppable if you want to get benefit from the XML file. After that you will need to call method *update(CC3DXMLElement *_xmlData, bool _fullInitFlag=false*) which just looks up the tags *<TargetVolume>* and *<LambdaVolume>* and copies the values of these tags to the attributes *lambda* and *targetVolume* (remember the parameters in the energy function?) as you can see in the implementation of the *update()* method:

```
void SimpleVolumePlugin::update(CC3DXMLElement *_xmlData, bool _fullInitFlag)
{
        //if there are no child elements for this plugin it means will use
```

```
changeEnergyByCellId
      if(_xmlData->findElement("TargetVolume"))
            targetVolume=_xmlData->getFirstElement("TargetVolume")->getDouble();
      if(_xmlData->findElement("LambdaVolume"))
            lambdaVolume=_xmlData->getFirstElement("LambdaVolume")->getDouble();
}
```

so that for the snippet of the XML file:

```
<Plugin Name="SimpleVolume">
   <TargetVolume>25</TargetVolume>
   <LambdaVolume>2.0</LambdaVolume>
</Plugin>
```

it will be equivalent to

```
targetVolume=25;
lambdaVolume=2.0;
```

The *init(Simulator *simulator, CC3DXMLElement *_xmlData)* method itself is called from the *Simulator::initializeCC3D()* (see the implementation in *Simulator.cpp*) method with other plugins and steppables right before the simulation starts.

So far we have implemented most important functionalities. Now we need to add one more thing. We write an implementation field that instantiates a template that will allow CompuCell3D to load *SimpleVolumePlugin* during run time. In general all plugins are loaded during run time. This way we do do end up with huge executables that contains an entire CompuCell3D functionality

The library that contains *SimpleVolumePlugin* implementations need to contain another a file where we instantiate *BasicPluginProxy* template. This file is called in our case *SimpleVolumePluginProxy.cpp*. Here is its content:

[SimpleVolumePluginProxy.cpp]

```
#include "SimpleVolumePlugin.h"
#include <CompuCell3D/Simulator.h>
#include <BasicUtils/BasicPluginProxy.h>
using namespace CompuCell3D;

BasicPluginProxy<Plugin, SimpleVolumePlugin>
volumeProxy("SimpleVolume", "Tracks cell volumes and adds volume energy
function.", &Simulator::pluginManager);

BasicPluginProxy<Plugin, SimpleVolumePlugin>
volumeEnergyProxy("SimpleVolumeEnergy", "Tracks cell volumes and adds volume
energy function.", &Simulator::pluginManager);
```

As you can see all it is quite mechanical to code this file. all you need to know (if you need to know more see the implementation of *BasicPluginProxy* and *BasicPluginManager.h*) is that in

order to enable loading SimpleVolumePlugin library you have to instantiate *BasicPluginProxy<Plugin, SimpleVolumePlugin>* template. In the constructor call :

```
volumeProxy("SimpleVolume", "Tracks cell volumes and adds volume energy
function.", &Simulator::pluginManager);
```

you provide Name of the plugin (this is how the plugin is referred to as in the XML file), in our case it is “SimpleVolume”, brief description of what plugin does, and pass address to an instance of *BasicPluginManager<Plugin>*.

At this point you may coalesce all this files into one shared library and you are done with the plugin development. It all may seem to be overwhelming at the first, but if you can pull out suitable examples from the existing code base and modify them, your task becomes much easier.

One nice thing about such code architecture is that you do need to worry about placing hooks to the plugin inside “main” CompuCell3D code. All you need to do is to provide library written according to the above “recipe” and you are done. In fact you do not need to even worry about remaining CompUCell3D code, because it remains untouched. This is pretty much all you need to know to write the plugins :).

Note: *As you might notice there is significant simplification in the new version 3.3.0 of the CompuCell3D such as there is no* `SimpleVolumeParseData` *class that stores the parsed data from nodes that are children to plugin node of the XML file. There is no* `SimpleVolumeEnergy` *class that calculates the energy change, since the* `SimpleVolumePlugin` *takes care of it. There no* `readXML()` *and* `writeXML()` *methods that parse and serialize data from and to XML file. Life is getting better! :)*

VolumeTracker plugin

As we mentioned earlier for a SimpleVolume plugin to run the simulation we must make sure that VolumeTracker plugin is loaded. Now we will present how to implement the VolumeTracker , that is the plugin that monitors volume changes and correspondingly updates cells' volume. As usual first we show the header file *VolumeTrackerPlugin.h:*

[VolumeTrackerPlugin.h]

```
 #ifndef VOLUMETRACKERPLUGIN_H
#define VOLUMETRACKERPLUGIN_H

#include <CompuCell3D/Plugin.h>
#include <CompuCell3D/Potts3D/Stepper.h>
#include <CompuCell3D/Potts3D/CellGChangeWatcher.h>
#include <CompuCell3D/Potts3D/Cell.h>
#include <CompuCell3D/dllDeclarationSpecifier.h>

class CC3DXMLElement;

namespace CompuCell3D {
  class Potts3D;
  class CellG;

class DECLSPECIFIER VolumeTrackerPlugin : public Plugin, public
CellGChangeWatcher, public Stepper
  {
      Potts3D *potts;
      CellG *deadCellG;

  public:
      VolumeTrackerPlugin();
      virtual ~VolumeTrackerPlugin();

      // SimObject interface
      virtual void init(Simulator *simulator, CC3DXMLElement *_xmlData);

      // CellChangeWatcher interface
      virtual void field3DChange(const Point3D &pt, CellG *newCell, CellG
*oldCell);

      // Stepper interface
      virtual void step();
      virtual std::string toString();
      virtual std::string steerableName();
  };
};
#endif
```

As you may see this class inherits *Plugin*, *CellGChangeWatcher* and *Stepper* and implements *init* from Plugin, *field3DChange* from *CellGChangeWatcher* and *step* from

Stepper. Those are the most important functions there. We need to write this plugin in the XML file if we want it to be loaded explicitly, however, we do not have to do it because, *SimpleVolume* plugin will make sure that *VolumeTracker* is loaded.
Let's explain how the *field3Dchange()* method is implemented. Here is the code:

```
void VolumeTrackerPlugin::field3DChange(const Point3D &pt,
                                CellG *newCell,
                                CellG *oldCell)
{
   if (newCell)
      newCell->volume++;

   if (oldCell)
      if((--oldCell->volume) == 0)
         deadCellG = oldCell;
}
```

It is really simple, as you can see. When a spin flip is made at location location *pt* a cell that is assigned to this location will have volume increased by one pixel (see *newCell->volume++*; statement, notice that ass we said it before we always make sure that we do not deal with medium – null pointer – before increasing volume of cell). Similarly, the cell that was at location 'pt' before spin flip will have its volume decreased by one (*--oldCell->volume*). Additionally when volume of the *oldCell* reaches zero we mark this cell for destruction. This marking is done by assigning *deadCellG* pointer to point to *oldCell* whose volume just reached zero. Actual destruction will take place in the *step* function (from *Stepper* interface). As we said it earlier, steppers are called once every objects which define *field3DChange* have been called.

Steppers are sort of “cleaners” that do “maintenance” after each successful spin flip.

Let's now look at the implementation of *step* function:

```
void VolumeTrackerPlugin::step()
{
      if (deadCellG) {
            potts->destroyCellG(deadCellG);
            deadCellG = 0;
      }
}
```

It is also very simple, namely, we check if *deadCellG* pointer has non zero value - if it does it means that due to last spin flip *oldCell* has reached zero volume and need to be destroyed *potts->destroyCellG(deadCellG)* . After cell is destroyed we need to do is to reset *deadCellG* to prevent destroying same cell twice (otherwise segmentation fault guaranteed).

We have all the functionality ready now we need to register *changeWatchers* and stepper objects with *potts* object. As you may remember *init* function does this trick:

```
void VolumeTrackerPlugin::init(Simulator *simulator,ParseData *_pd)
{
```

```
      potts = simulator->getPotts();
      potts->registerCellGChangeWatcher(this);
      potts->registerStepper(this);
}
```

and we are almost done with *VolumeTracker* plugin. The only thing that remains is to instantiate *BasicPluginProxy* template for *VolumeTracker*. Those is pretty mechanical (but important) task:

[VolumeTrackerPluginProxy.h]

```
#include "VolumeTrackerPlugin.h"
#include <CompuCell3D/Simulator.h>
#include <BasicUtils/BasicPluginProxy.h>

using namespace CompuCell3D;

BasicPluginProxy<Plugin, VolumeTrackerPlugin>
volumeTrackerProxy("VolumeTracker", "Tracks cell volumes",
&Simulator::pluginManager);
```

This should look familiar. now all you need to do is to pack all the files *(VolumeTrackerPlugin.cpp*, *VolumeTrackerPluginProxy.cpp*) into a shared library and you have another plugin ready.

At this point you should be able to explore CompuCell3D by yourself and try to understand how other plugins are built.

We are going to show you another useful “trick” which is very often needed in CompuCell3D plugin development, that is how to visit neighbors of a given pixel. Let's look at the implementation of the SurfaceTracker plugin, in particular at the *field3DChange* function:

[*SurfaceTrackerPlugin.cpp*]

```
void SurfaceTrackerPlugin::field3DChange(const Point3D &pt,
                                    CellG *newCell,
                                    CellG *oldCell)
{
      //this may happen if you are trying to assign same cell to one pixel twice
      if (newCell==oldCell)
            return;

      unsigned int token = 0;
      double distance;
      double oldDiff    = 0.;
      double newDiff= 0.;
      CellG *nCell= 0;
      Neighbor neighbor;
```

```
      for(unsigned int nIdx=0 ; nIdx <= maxNeighborIndex ; ++nIdx )
      {
            neighbor = boundaryStrategy-
>getNeighborDirect(const_cast<Point3D&>(pt),nIdx);
            if(!neighbor.distance)
                  continue;

            nCell = cellFieldG->get(neighbor.pt);
            if (newCell == nCell) newDiff-=lmf.surfaceMF;
            else newDiff+=lmf.surfaceMF;

            if (oldCell == nCell) oldDiff+=lmf.surfaceMF;
            else oldDiff-=lmf.surfaceMF;
      }

      if (newCell) newCell->surface += newDiff;
      if (oldCell) oldCell->surface += oldDiff;
}
```

Here you can see how we visit neighbors of the pixel *pt*. It is important to first initialize pointer to *boundaryStrategy* singleton class which manages neighbors this is done in the init function. Also note that the surface gets increased or decreased in the increments of *lmf.surfaceMF* they are multiplicative factors that are determined depending on the type of lattice. For a square lattice all multiplicative factors are 1 for other type lattices they are calculated by constraining the volume of the pixel to be 1. Multiplicative factors are obtained from *BoundaryStrategy* class usually in the *init* function of the plugin.

Developing Steppables

In older versions of CompuCell3D steppables (i.e. objects called every Monte Carlo Step) were merged in the main CompuCell3D library (libCompuCell3D on UNIX/OSX/Linux systems). Recently we have however adopted same methodology as for plugins i.e. steppables are loadable modules and they are developed using the same "philosophy" as plugins. This means that your steppable will need to instantiate properly *BasicPluginProxy* template, as we will show in the example.

To this end let's analyze one simple steppable to get an idea how to develop more complicated ones. In the CompuCell3D/src/steppables/FoamDataOutput subdirectory you can find source files for FoamData. traditionally let's analyze first a header file:

[FoamDataOutput.h]

```
#ifndef FOAMDATAOUTPUT_H
#define FOAMDATAOUTPUT_H

#include <CompuCell3D/Steppable.h>
#include <CompuCell3D/Field3D/Dim3D.h>
#include <CompuCell3D/plugins/NeighborTracker/NeighborTracker.h>
#include <CompuCell3D/dllDeclarationSpecifier.h>
#include <string>
```

```
template <typename Y> class BasicClassAccessor;

namespace CompuCell3D {
  class Potts3D;
  class CellInventory;

  class DECLSPECIFIER FoamDataOutput : public Steppable
  {
      Potts3D *potts;
      CellInventory * cellInventoryPtr;
      Dim3D dim;
      BasicClassAccessor<NeighborTracker> * neighborTrackerAccessorPtr;
      std::string fileName;
      bool surFlag;
      bool volFlag;
      bool numNeighborsFlag;
      bool cellIDFlag;

  public:
      FoamDataOutput();
      virtual ~FoamDataOutput(){};
      void setPotts(Potts3D *potts) {this->potts = potts;}

      // SimObject interface
      virtual void init(Simulator *_simulator, CC3DXMLElement *_xmlData=0);
      virtual void extraInit(Simulator *simulator);
      virtual std::string toString();
      // Begin Steppable interface
      virtual void start();
      virtual void step(const unsigned int currentStep);
      virtual void finish() {}
      // End Steppable interface
  };
};
#endif
```

As you can see *Steppable* API consists of the following functions:

```
   1) void start()
   2) void step(unsigned int _mcs)
   3) void finish()
```

In addition to that there are two more functions used to initialize a Steppable:

```
virtual void init(Simulator *_simulator, CC3DXMLElement *_xmlData=0);
virtual void extraInit(Simulator *simulator);
```

Now let's take a look at initialization of steppables. The *init* function fetches address to the potts3D object and from there we get an address to the cell inventory – a data structures that stores pointers to every cell sorted by cell id (this actually is not tha important at the moment. What matters that any look-up in cell inventory in O(log(N)) which might be important once you do simulation with large number of cells). It also should be said that most of the steppables iterate over set of cells and change their properties depending on requirements of the simulation.
Once *init* function has been called for all other steppables and plugins we can do extra round of

initialization. To enable extra initialization all you need to do is to provide implementation of `extraInit` virtual function in a seppable or plugin. This additional initialization is very often necessary as it is done when all the modules have been loaded into memory. At this point you can reference/access steppables and plugins from any module that you decided to load. Moreover you may even load certain modules from either `init` or `extraInit` functions *(in principle you can do it at any time but usually you do it in these two initialization functions, where it makes most sense). To get a reference to* `NeighborTracker` *plugin you use the* `Simulator::pluginManager.get("NeighborTracker")` *call. If the plugin has not been loaded* `Simulator::pluginManager.get` *will load it.*

```
void FoamDataOutput::init(Simulator *_simulator, CC3DXMLElement *_xmlData)
{
      potts = _simulator->getPotts();
      cellInventoryPtr = & potts->getCellInventory();
      CC3DXMLElement *outputXMLElement=_xmlData->getFirstElement("Output");

      ASSERT_OR_THROW("You need to provide Output element to FoamDataOutput
Steppable with at least
           file name", outputXMLElement);

      if(outputXMLElement)
      {
            if(outputXMLElement->findAttribute("FileName"))
               fileName=outputXMLElement->getAttribute("FileName");

            if(outputXMLElement->findAttribute("Volume"))
               volFlag=true;

            if(outputXMLElement->findAttribute("Surface"))
               surFlag=true;

            if(outputXMLElement->findAttribute("NumberOfNeighbors"))
               numNeighborsFlag=true;

            if(outputXMLElement->findAttribute("CellID"))
               numNeighborsFlag=cellIDFlag;
      }
}

void FoamDataOutput::extraInit(Simulator *simulator)
{
      if(numNeighborsFlag)
      {
            bool pluginAlreadyRegisteredFlag;
            NeighborTrackerPlugin * neighborTrackerPluginPtr =
(NeighborTrackerPlugin*)
             (Simulator::pluginManager.get("NeighborTracker",
&pluginAlreadyRegisteredFlag));
            if (!pluginAlreadyRegisteredFlag)
               neighborTrackerPluginPtr->init(simulator);
            ASSERT_OR_THROW("NeighborTracker plugin not initialized!",
neighborTrackerPluginPtr);
```

```
            neighborTrackerAccessorPtr = neighborTrackerPluginPtr->
getNeighborTrackerAccessorPtr();

            ASSERT_OR_THROW("neighborAccessorPtr  not initialized!",
neighborTrackerAccessorPtr);
      }
}
```

Now let's look how other core steppable functions are implemented in our *FoamDataOutput* example.

```
void FoamDataOutput::start() {}

void FoamDataOutput::step(const unsigned int currentStep)
{
      ostringstream str;
      str<<fileName<<"."<<currentStep;
      ofstream out(str.str().c_str());

      CellInventory::cellInventoryIterator cInvItr;
      CellG * cell;
      std::set<NeighborSurfaceData > * neighborData;

         for(cInvItr=cellInventoryPtr->cellInventoryBegin() ; cInvItr !
=cellInventoryPtr->cellInventoryEnd() ;++cInvItr )
         {
            cell=cellInventoryPtr->getCell(cInvItr);
            if(cellIDFlag)
               out<<cell->id<<"\t";

            if(volFlag)
               out<<cell->volume<<"\t";

            if(surFlag)
               out<<cell->surface<<"\t";

            if(numNeighborsFlag){
               neighborData = &(neighborTrackerAccessorPtr->get(cell-
>extraAttribPtr)->cellNeighbors);
               out<<neighborData->size()<<"\t";
            }
            out<<endl;
      }
}
```

This does not look too bad, especially given the fact that *start* function is empty and *finish* has its default implementation which is empty function as well. Well, this is not always the case, and especially in the start functions you perform all the initialization necessary before the actual simulation begins. For example you may want to assign target volume for each cell, in which case you would iterate over all cells and execute the for example following assignment:

```
cell->targetVolume=15; // I have assumed that cell is a pointer to cell object
```

Now, let's see how one iterates over cell inventory:

```
CellG * cell;
for(  cInvItr=cellInventoryPtr->cellInventoryBegin() ;
      cInvItr !=cellInventoryPtr->cellInventoryEnd() ;
      ++cInvItr )
{

      cell=cellInventoryPtr->getCell(cInvItr);
      //DO SOMETHING WITH THE CELL
}
```

Above code template is repeated in different variations quite frequently in the steppables.
You get a pointer to a cell object by passing the iterator to the `getCell` function.
In our case *`//DO SOMETHING WITH THE CELL`* is simply writing cell id, volume, surface and number of neighbors to the stream out (which in our case is associated with the file, or in other words we are writing to the file)
Notice how we get the number of neighbors of a cell:

```
neighborData = &(neighborTrackerAccessorPtr->get(cell-> extraAttribPtr)-
>cellNeighbors)
out<< neighborData->size()<<"\t";
```

Since cell neighbors are stored in *`cellNeighbors`* additional attribute (which is of a type *`std::set<NeighborSurfaceData>`*), all we need to do is to get a size of this set, which we do using the following statement:

```
neighborData->size()
```

However to access additional cell attribute, the one that is added during run time we do it using call of the type:

```
&(accessorPtr->get(cell->extraAttribPtr)->additionalAttribute)
```

this will return a reference to *`additionalAttribute`* of a *`cell`*. Compare it to actual example fro the code

One last thing, we need to provide a proxy for the steppable. As you remember in the case of plugins writing proxy implementation was very automatic task. The same is true in the case of steppables:

[FoamDataOutoutProxy.cpp]

```
#include "FoamDataOutput.h"
#include <CompuCell3D/Simulator.h>
#include <BasicUtils/BasicPluginProxy.h>

using namespace CompuCell3D;

BasicPluginProxy<Steppable, FoamDataOutput>
```

```
FoamDataOutputProxy("FoamDataOutput", "Outputs basic simulation data for foam
coarsening", &Simulator::steppableManager);
```

There is however one crucial difference, namely that you use *&Simulator::steppableManage*r in the constructor call instead if *&Simulator::pluginManager* as it was the case for plugins.

Our example is ready.

Above examples should give you at least an idea how to develop modules for CompuCell3D. We did not discuss all the code features but we will do our best to provide more complete documentation in the future.

Current Build System – CMake

CompuCell3D uses CMake as a build system. That means that every time you want to add new module you will need to write CMakeLists.txt for this new module and integrate it with existing source code base. Fortunately this task is very easy even for beginners. In the *DeveloperZone* directory there are examples of plugins and steppables that you may either use as a template for your own extension CompuCell3D modules. Let's see how one integrates extension module with existing source code. Assuming that you are in the *DeveloperZone* directory modify CMakeLists.txt file in this directory as follows

```
 SET_TARGET_PROPERTIES(${LIBRARY_NAME}Shared  PROPERTIES OUTPUT_NAME CC3D$
{LIBRARY_NAME}${LIBRARY_SUFFIX})
  INSTALL_TARGETS(/lib/CompuCell3DSteppables RUNTIME_DIRECTORY
/lib/CompuCell3DSteppables
  ${LIBRARY_NAME}Shared)

ENDMACRO(ADD_COMPUCELL3D_STEPPABLE)

add_subdirectory(SimpleVolume)
add_subdirectory(VolumeMean)
add_subdirectory(YourNewPlugin) # I assume this is a subdirectory of DeveloperZone
add_subdirectory(pyinterface)
```

The actual CMakeLists.txt configuration file for YourNewPlugin (in the YourNewPlugin directory) could look like this:

```
ADD_COMPUCELL3D_PLUGIN(YourNewPlugin YourNewEnergy.cpp YourNewPlugin.cpp
YourNewPluginProxy.cpp LINK_LIBRARIES ${CC3DLibraries})

ADD_COMPUCELL3D_PLUGIN_HEADERS( YourNewEnergy YourNewEnergy.h  YourNewPlugin.h)
```

Next, to make sure your new plugin is integrated with Python interface you need to go to pyinterface directory (inside DeveloperZone) and edit CmakeLists.txt file there:

```
SET(LIBS_AUX
   SimpleVolumeShared
```

```
   VolumeMeanShared
   YourNewPluginShared #adding your new plugin to linked modules
   PyPlugin
)
```

and also edit SWIG interface file CompuCellExtraModules.i:

```
%module CompuCellExtraModules
%include "typemaps.i"
%include <windows.i>

%{

#include "ParserStorage.h"

// ********* PUT YOUR PLUGIN PARSE DATA AND PLUGIN FILES HERE **********

#include <SimpleVolume/SimpleVolumePlugin.h>
#include <VolumeMean/VolumeMean.h>
#include <YourNewPlugin/YourNewPlugin.h>

// ********** END OF SECTION ****************************

#include "dllDeclarationSpecifier.h"

#define DECLSPECIFIER //have to include this to avoid problems with interpreting
by swig win32 specific c++ extensions...

#include <iostream>

using namespace std;
using namespace CompuCell3D;

%}

// C++ std::string handling
%include "std_string.i"

// C++ std::map handling
%include "std_map.i"

// C++ std::map handling
%include "std_set.i"

// C++ std::vector handling
%include "std_vector.i"

#define DECLSPECIFIER //have to include this to avoid problems with interpreting
by swig win32 specific c++ extensions...

%include "ParserStorage.h"

// ************ PUT YOUR PLUGIN PARSE DATA AND PLUGIN FILES HERE **********
// REMEMBER TO CHANGE #include to %include
```

```
%include <SimpleVolume/SimpleVolumePlugin.h>

// THIS IS VERY IMORTANT STETEMENT WITHOUT IT SWIG will produce incorrect wrapper
code which will compile but will not work
using namespace CompuCell3D;

%inline %{
   SimpleVolumePlugin * reinterpretSimpleVolumePlugin(Plugin * _plugin){
      return (SimpleVolumePlugin *)_plugin;
   }

%}

%include <VolumeMean/VolumeMean.h>

%inline %{
   VolumeMean * reinterpretVolumeMean(Steppable * _steppable){
      return (VolumeMean *)_steppable;
   }

%}

%include <YourNewPlugin/YourNewPlugin.h>

%inline %{
   YourNewPlugin * reinterpretYourNewPlugin (Plugin * _plugin){
      return (YourNewPlugin *)_plugin;
   }

%}

// ***************** END OF SECTION ***********
```

At this point you are ready to compile your plugins.

Developing CompuCell3D plugins under Windows using Cmake and MS Visual Studio 2005 (version 8)

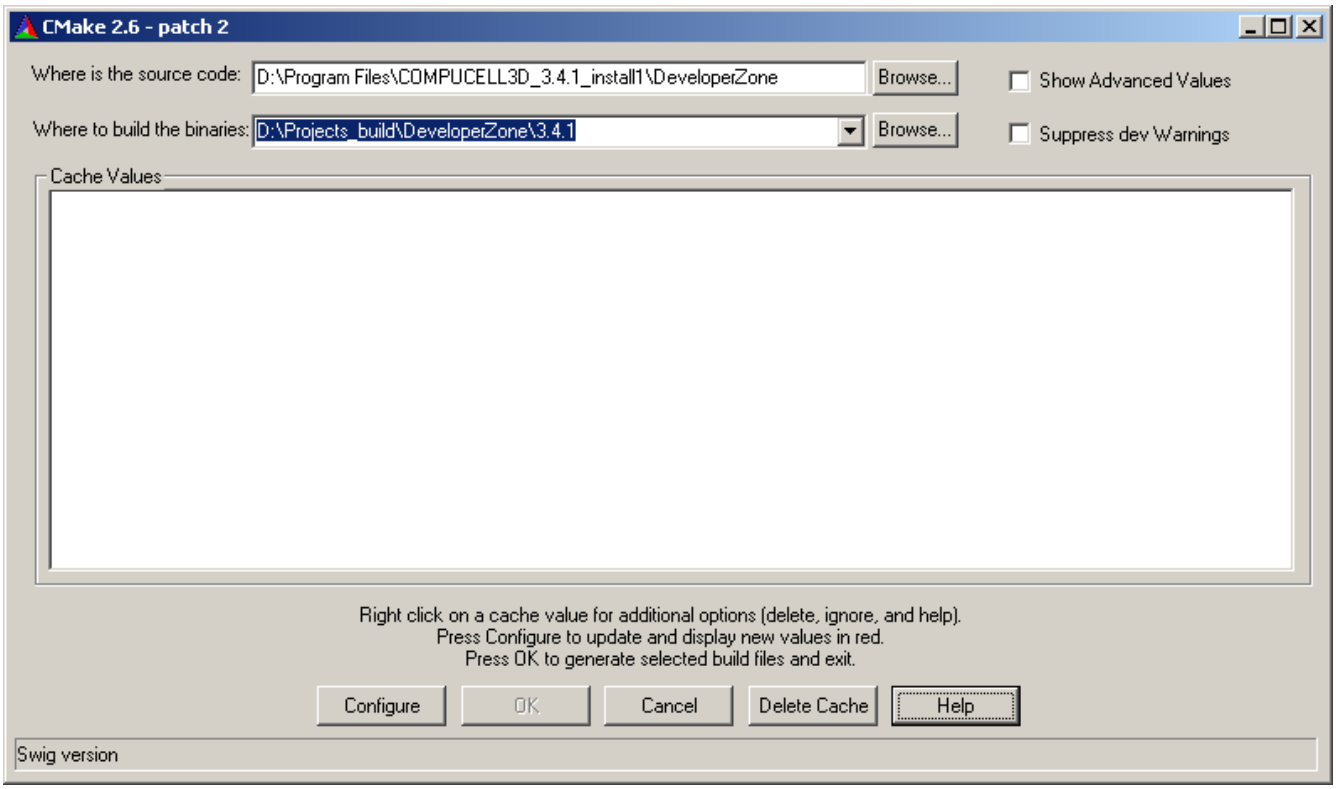
We assume that you have Visual Studio 2005, Cmake and SWIG installed on your machine. CMake is free (www.cmake.org) so is SWIG (www.swig.org). Visual Studio is not free and you might need to purchase appropriate license. Some universities have access to educational licenses for Visual Studio which are either free or have reduced price so you might want to check this before paying full price.

OK, let's get started. Make sure you have CompuCell3D version on your machine that contains a directory called DeveloperZone. If you download latest source version from our repository

`http://trac.compucell3d.net/svn/cc3d_svn/trunk/`

There you will find all the CMake files necessary to develop plugins There are also examples of plugins that you might study and reuse to build your own modules.

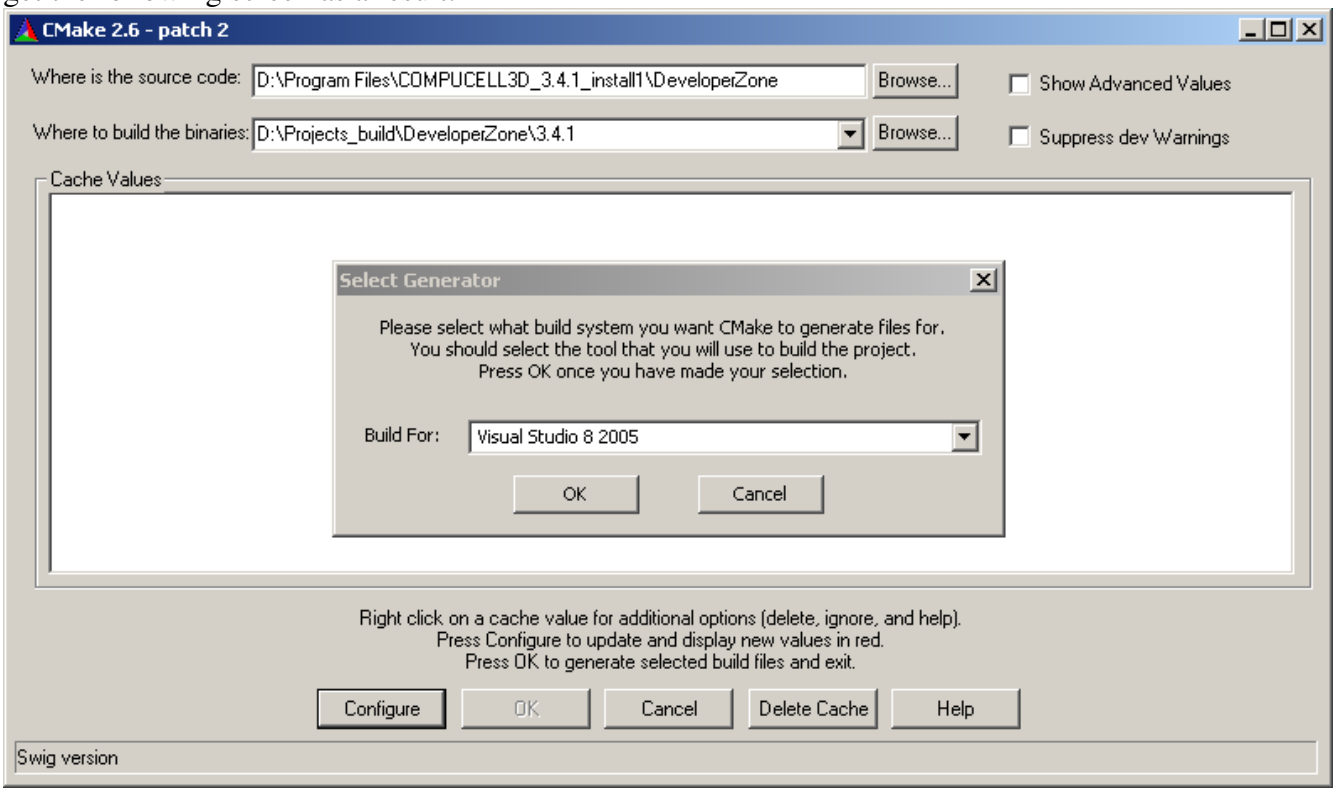
First thing you want to do is to open CMake on your machine and setup source and build directories:



notice that I put as a path to the source code for new plugins as `D:\Program Files\COMPUCELL3D_3.4.1_install1\DeveloperZone` and the directory where all object files and project files will be stored as

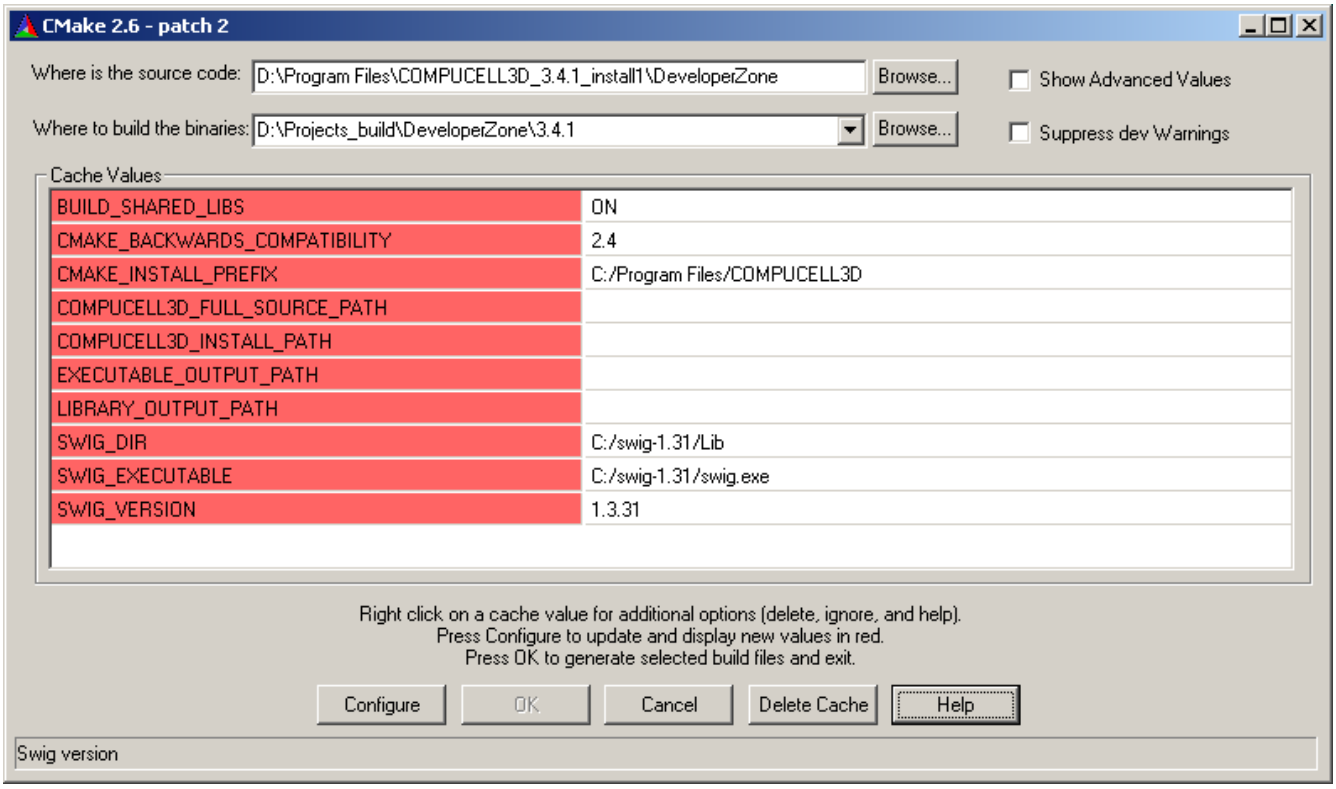
D:\Projects_build\DeveloperZone\3.4.1.
It is always a good to pick binary directory separately from source directory – as we did above.

The next step is to start configuration of CMake build system by pressing configure button. You should get the following screen as a result:



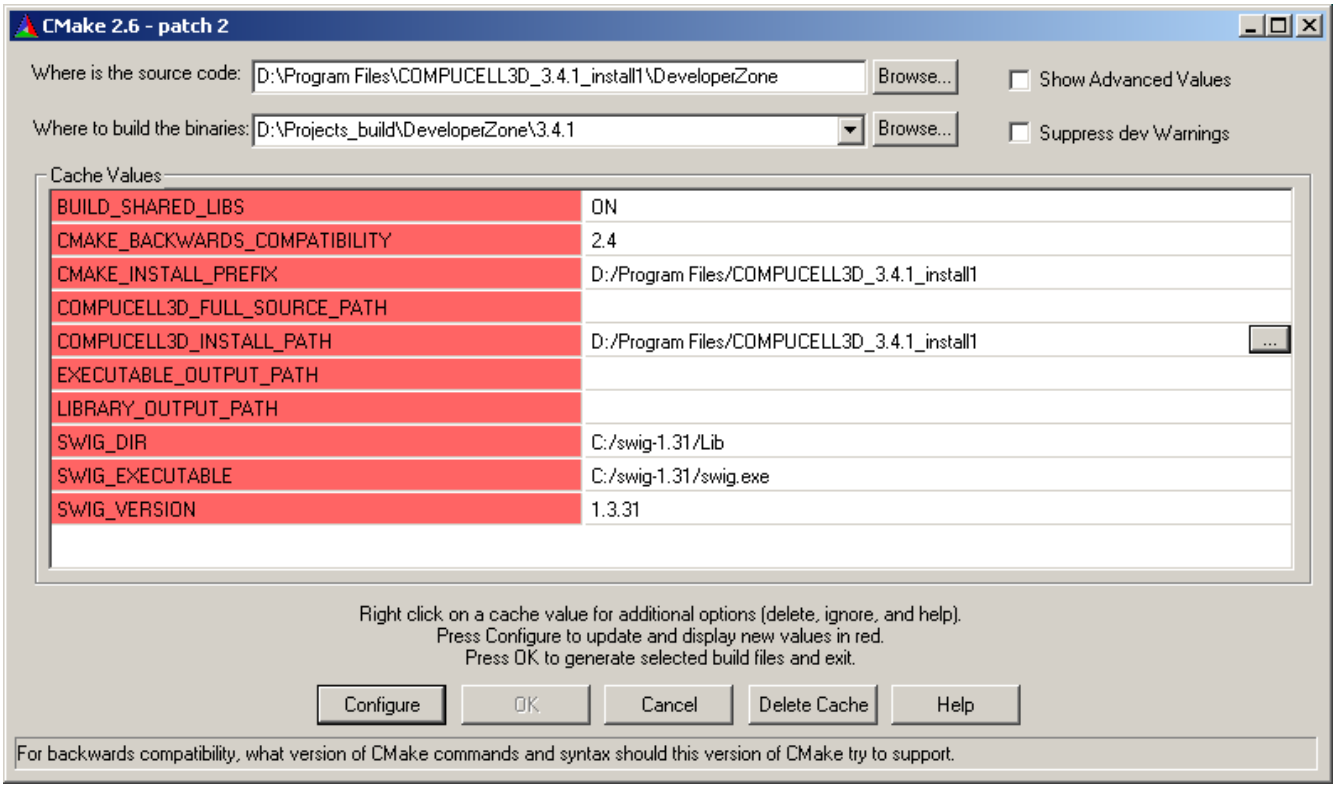
Here you need to choose Visual Studio 8 2008. There are other options to choose from but have tested CompuCell3D using Visual Studio 8 2005. If you are interested in compiling CompuCell3D using other compilers let us know and we can help you get started with this task

Anyway, after you picked Visual Studio 2005 you will get the following screen (followed by several messages or error messages – this is OK as some of them are just debug outputs. If SWIG is in your path, CMake will find it automatically, otherwise you migh have to point CMake to the location where SWIG is installed.

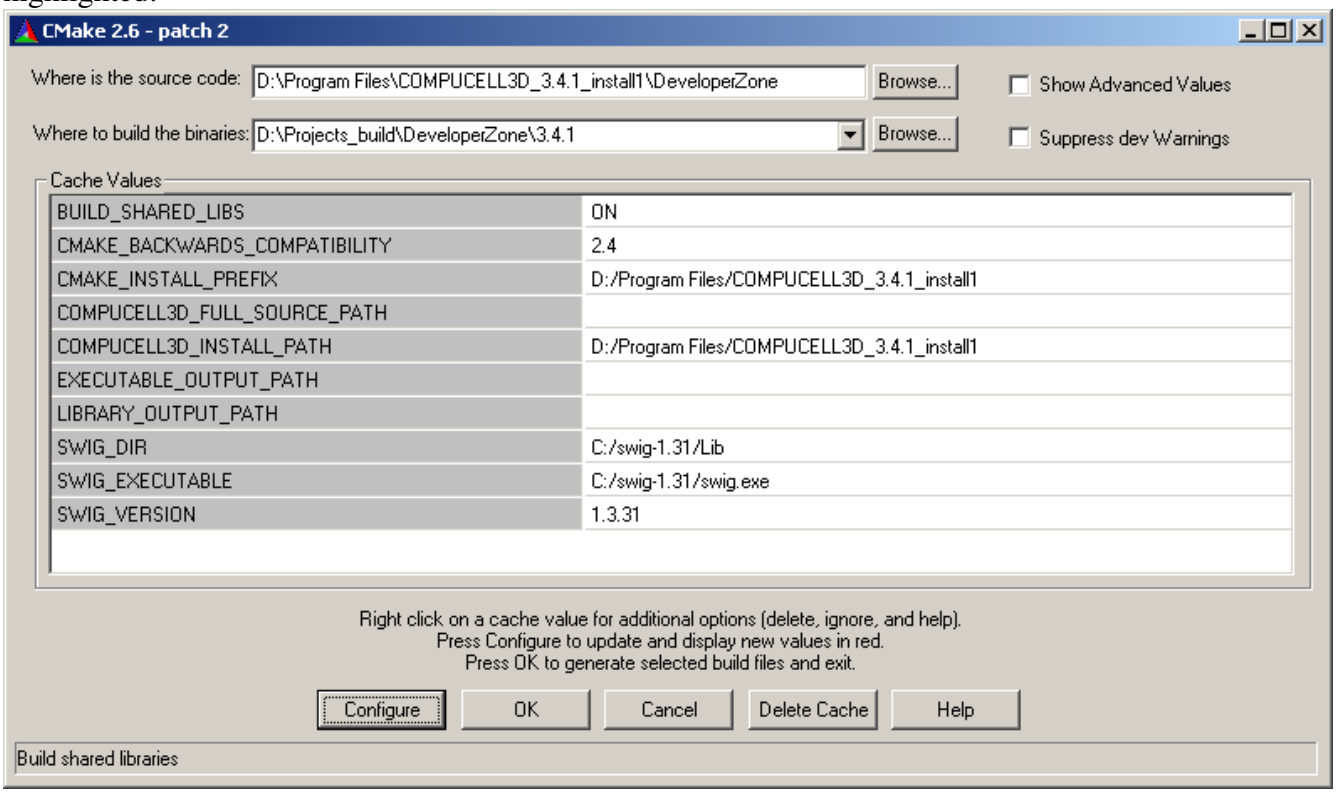


First thing you need to do is to point CMake to directory where SWIG is installed and also pick the directory where you have stored full CompuCell3D source code (this is the directory that contains *DeveloperZone* directory) and the directory where binary version of the CompuCell3D is installed *-COMPUCELL3D_INSTALL_PATH*. We also need to pick installation point for the new modules

- *CMAKE_INSTALL_PREFIX* and we will set it to the same value as *COMPUCELL3D_INSTALL_PATH* because we want new modules to be installed into same directory as current binary CompuCell3D installation.



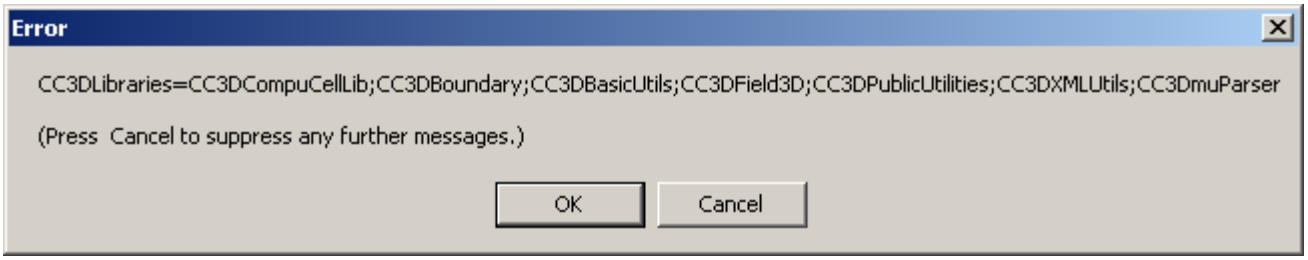
Next you need to click **Configure** few times until all red fields become grey and **OK** button is highlighted:



Notice – our CompuCell3D was installed in *D:/Program Files/COMPUCELL3D_3.4.1_install1* (*COMPUCELL3D_INSTALL_PATH*,

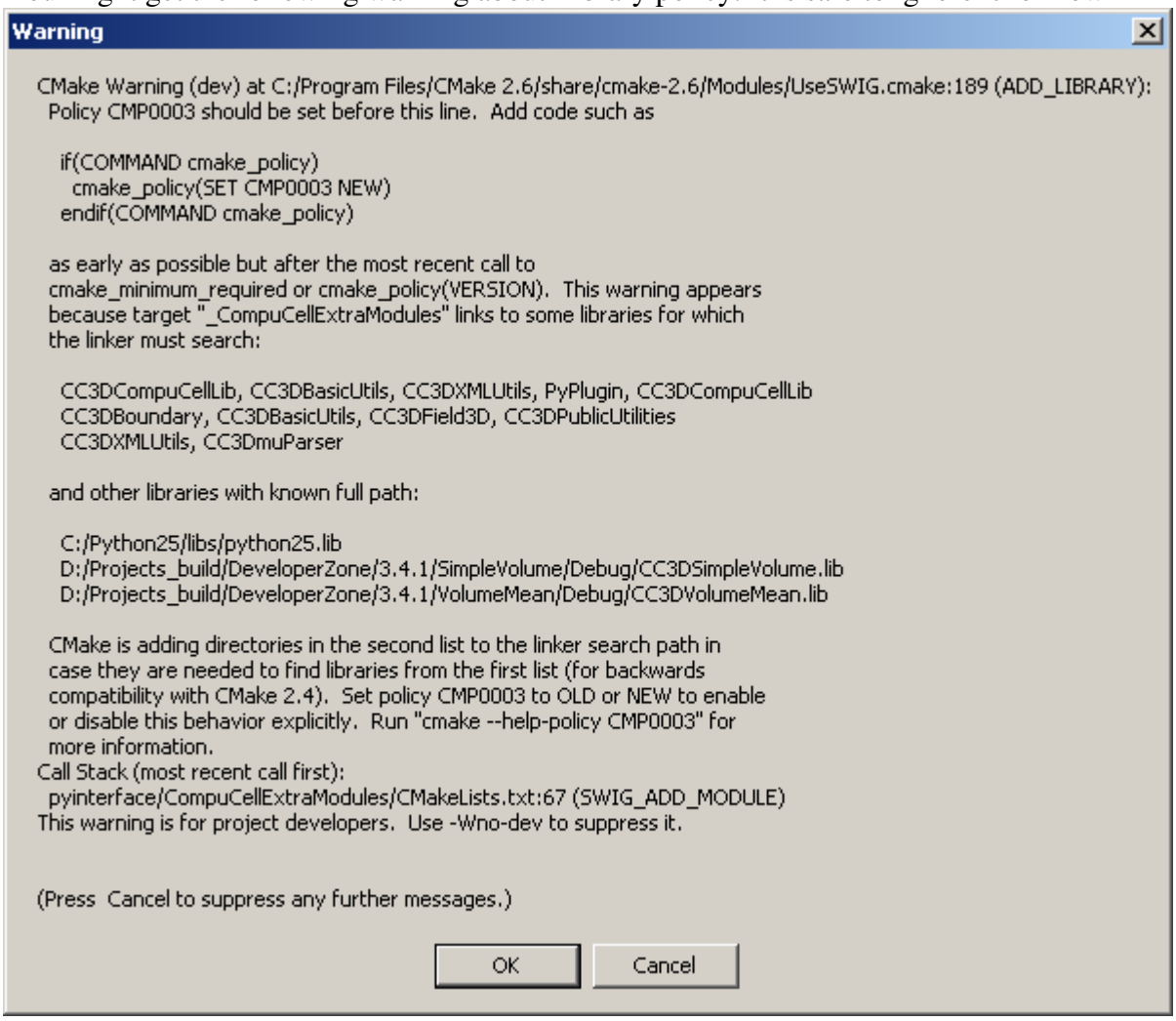
CMAKE_INSTALL_PREFIX), sand swig is located in *c:\swig-1.31* directory.
All other variables are set automatically by CMake .

The error messages that may pop up are OK because they are just message windows not actual errors:



The true error message would look similar but would begin with "CMake Error: ..."

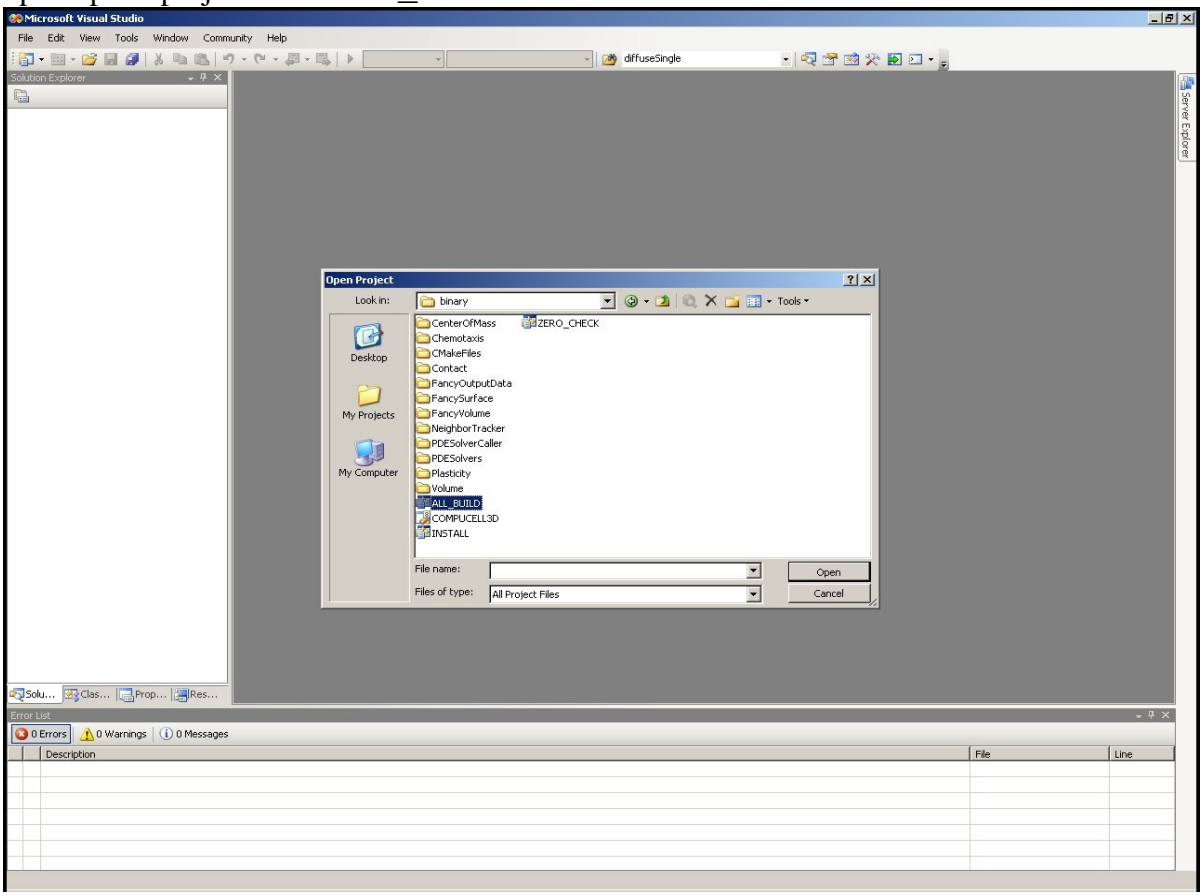
You might get the following warning about Library policy. It is safe to ignore it for now



Now, once you pressed OK button Visual Studio files were generated and you are ready to open them

up in Visual Studio.

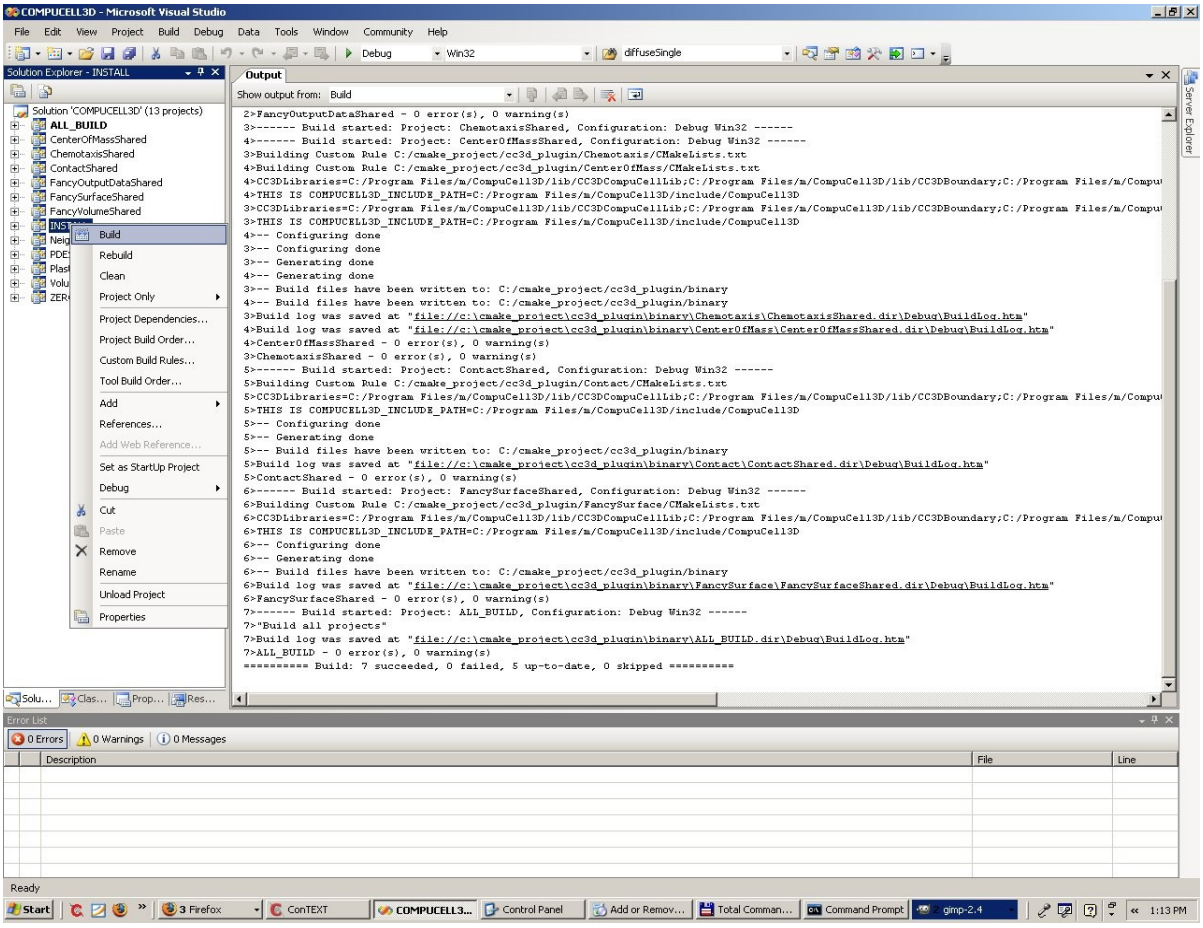
In my case we need to go to *D:\Projects_build\DeveloperZone\3.4.1* and from there open up the project called *All_BUILD*:

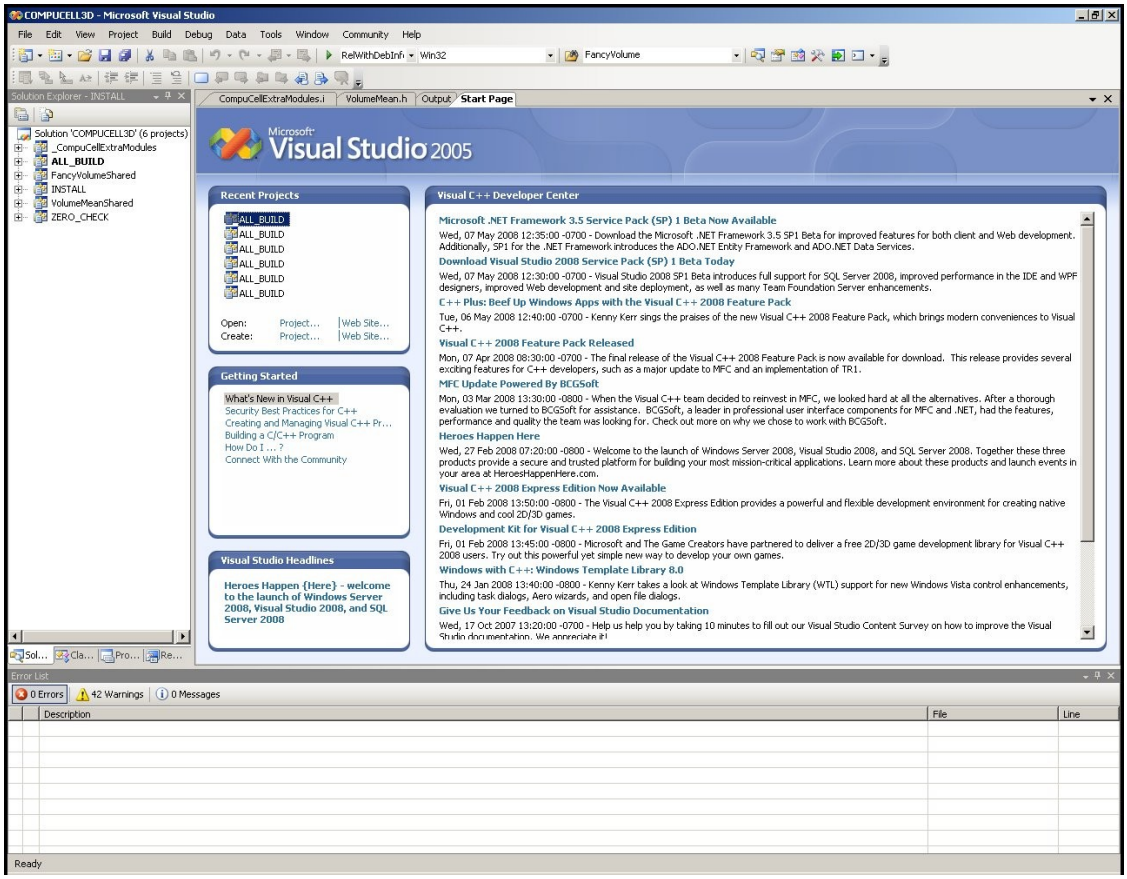


Before going any further go Build->Configuration Manager... and select Active solution configuration to be ReleaseWithDebugInfo or simply Release.

Next, to compile the project, right click *ALL_BUILD* and select build from pop up menu:

The compilation begins and once it finishes without errors all you need to do is to install newly created plugin into the place where other CompuCell3D plugins are installed. Right click on *INSTALL* project and select Build option:


COMPUCELL3D - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Window Community Help
Debug
Win32
diffuseSingle
Solution Explorer - INSTALL
Solution 'COMPUCELL3D' (13 projects)
ALL_BUILD
CenterOfMassShared
ChemotaxisShared
ContactShared
FancyOutputDataShared
FancySurfaceShared
FancyVolumeShared
Build
Rebuild
Clean
Project Only
Project Dependencies...
Project Build Order...
Custom Build Rules...
Tool Build Order...
Add
References...
Add Web Reference...
Set as StartUp Project
Debug
Cut
Paste
Remove
Rename
Unload Project
Properties
Output
Show output from: Build
2>FancyOutputDataShared - 0 error(s), 0 warning(s)
3>------ Build started: Project: ChemotaxisShared, Configuration: Debug Win32 ------
4>------ Build started: Project: CenterOfMassShared, Configuration: Debug Win32 ------
3>Building Custom Rule C:/cmake_project/cc3d_plugin/Chemotaxis/CMakeLists.txt
4>Building Custom Rule C:/cmake_project/cc3d_plugin/CenterOfMass/CMakeLists.txt
4>CC3DLibraries=C:/Program Files/m/CompuCell3D/lib/CC3DCompuCellLib;C:/Program Files/m/CompuCell3D/lib/CC3DBoundary;C:/Program Files/m/Compu
4>THIS IS COMPUCELL3D_INCLUDE_PATH=C:/Program Files/m/CompuCell3D/include/CompuCell3D
3>CC3DLibraries=C:/Program Files/m/CompuCell3D/lib/CC3DCompuCellLib;C:/Program Files/m/CompuCell3D/lib/CC3DBoundary;C:/Program Files/m/Compu
3>THIS IS COMPUCELL3D_INCLUDE_PATH=C:/Program Files/m/CompuCell3D/include/CompuCell3D
4>-- Configuring done
3>-- Configuring done
3>-- Generating done
4>-- Generating done
3>-- Build files have been written to: C:/cmake_project/cc3d_plugin/binary
4>-- Build files have been written to: C:/cmake_project/cc3d_plugin/binary
3>Build log was saved at "file://c:\cmake_project\cc3d_plugin\binary\Chemotaxis\ChemotaxisShared.dir\Debug\BuildLog.htm"
4>Build log was saved at "file://c:\cmake_project\cc3d_plugin\binary\CenterOfMass\CenterOfMassShared.dir\Debug\BuildLog.htm"
4>CenterOfMassShared - 0 error(s), 0 warning(s)
3>ChemotaxisShared - 0 error(s), 0 warning(s)
5>------ Build started: Project: ContactShared, Configuration: Debug Win32 ------
5>Building Custom Rule C:/cmake_project/cc3d_plugin/Contact/CMakeLists.txt
5>CC3DLibraries=C:/Program Files/m/CompuCell3D/lib/CC3DCompuCellLib;C:/Program Files/m/CompuCell3D/lib/CC3DBoundary;C:/Program Files/m/Compu
5>THIS IS COMPUCELL3D_INCLUDE_PATH=C:/Program Files/m/CompuCell3D/include/CompuCell3D
5>-- Configuring done
5>-- Generating done
5>-- Build files have been written to: C:/cmake_project/cc3d_plugin/binary
5>Build log was saved at "file://c:\cmake_project\cc3d_plugin\binary\Contact\ContactShared.dir\Debug\BuildLog.htm"
5>ContactShared - 0 error(s), 0 warning(s)
6>------ Build started: Project: FancySurfaceShared, Configuration: Debug Win32 ------
6>Building Custom Rule C:/cmake_project/cc3d_plugin/FancySurface/CMakeLists.txt
6>CC3DLibraries=C:/Program Files/m/CompuCell3D/lib/CC3DCompuCellLib;C:/Program Files/m/CompuCell3D/lib/CC3DBoundary;C:/Program Files/m/Compu
6>THIS IS COMPUCELL3D_INCLUDE_PATH=C:/Program Files/m/CompuCell3D/include/CompuCell3D
6>-- Configuring done
6>-- Generating done
6>-- Build files have been written to: C:/cmake_project/cc3d_plugin/binary
6>Build log was saved at "file://c:\cmake_project\cc3d_plugin\binary\FancySurface\FancySurfaceShared.dir\Debug\BuildLog.htm"
6>FancySurfaceShared - 0 error(s), 0 warning(s)
7>------ Build started: Project: ALL_BUILD, Configuration: Debug Win32 ------
7>"Build all projects"
7>Build log was saved at "file://c:\cmake_project\cc3d_plugin\binary\ALL_BUILD.dir\Debug\BuildLog.htm"
7>ALL_BUILD - 0 error(s), 0 warning(s)
========== Build: 7 succeeded, 0 failed, 5 up-to-date, 0 skipped ==========
Error List
0 Errors
0 Warnings
0 Messages
Description
File
Line
Ready
Start
Firefox
ConTEXT
COMPUCELL3...
Control Panel
Add or Remov...
Total Comman...
Command Prompt
gimp-2.4
1:13 PM



Now all the demo plugins were compiled and installed in the CompuCell3D installation directory on my windows machine.

To compile your own plugins you need to make directory that would contain *CMakeLists.tx*t file and source code for your plugin and then in add subdirectory to the *CMakeLists.txt* file in the main developer directory (*D:\Program Files\COMPUCELL3D_3.4.1_install1\DeveloperZon*e):

```
 SET_TARGET_PROPERTIES(${LIBRARY_NAME}Shared  PROPERTIES OUTPUT_NAME CC3D$
{LIBRARY_NAME}${LIBRARY_SUFFIX})
  INSTALL_TARGETS(/lib/CompuCell3DSteppables RUNTIME_DIRECTORY
/lib/CompuCell3DSteppables
  ${LIBRARY_NAME}Shared)

ENDMACRO(ADD_COMPUCELL3D_STEPPABLE)

add_subdirectory(SimpleVolume)
add_subdirectory(VolumeMean)
add_subdirectory(YourNewPlugin) # I assume this is a subdirectory of DeveloperZone

add_subdirectory(pyinterface)
```

Looking at the *CMakeLists.tx*t files for plugins one can see that configuring new plugin using CMake requires listing plugin's source files as parameters of *ADD_COMPUCELL3D_PLUGIN* and *ADD_COMPUCELL3D_PLUGIN_HEADERS* (or for steppables *ADD_COMPUCELL3D_STEPPABLE* and *ADD_COMPUCELL3D_STEPPABLE_HEADERS*) macros.

The actual *CMakeLists.txt* configuration file for *YourNewPlugin* could look like this:

```
ADD_COMPUCELL3D_PLUGIN(YourNewPlugin YourNewEnergy.cpp YourNewPlugin.cpp
YourNewPluginProxy.cpp LINK_LIBRARIES ${CC3DLibraries} )

ADD_COMPUCELL3D_PLUGIN_HEADERS( YourNewEnergy YourNewEnergy.h  YourNewPlugin.h)
```

The above documentation is for all operating system. The only platform dependent part presenteed here was that one that dealt with Visual Studio. Recent versions of CMake come with GUI that looks pretty much the same on all platforms therefore generating of makefiles, Kdevelop project , XCode project , or Visual Studio project exactly follows steps described above.