Cellulate

User Manual

Version 1.0

Written by: **Nick Pope**
James Dowling
Nick Williams
Tom Hirst
Robert Newman

25-04-2007

Cellulate

Table of Contents

1 Introduction

Congratulations on your download of Cellulate! Treated with care, respect and love, your copy of Cellulate will serve you well for many years.

Cellulate is a cellular automata modeller. It has the ability to run 1-, 2- or 3-dimensional cellular automata with any number of cell variables across numerous iterations powered by a user-defined script. It is possible to split the calculations over several machines by running a processing node on every machine you want to participate in the simulation.

The script determining the future state of cells can be easily edited using a subset of C++ (which is fully documented in sections 6 and 8). This script is then dynamically compiled on each of the processing nodes, increasing the speed of the simulation.

Cellulate was initially designed to simulate cell and bacterial growth but has grown into a fully-featured general purpose cellular automata modeller. It is capable of running any simulation for which a cellular automata is appropriate; the most common example of one, Conway's Game of Life, is provided in both 2D and 3D (see the `examples` directory).

Cellulate is designed on run on any platform that supports the wxWidgets windowing library. It has been successfully tested on OS X, Windows XP and Vista, and various distributions of Linux.

The latest version of Cellulate can always be found at http://cellulate.sf.net/.

2 System Requirements

2.1 Libraries

To successfully compile Cellulate, you will require the following libraries to be compiled and available, together with the development header files. If you are installing a binary, you don't need to worry about this.

2.1.1 wxWidgets

wxWidgets is a cross-platform windowing library. Cellulate requires wxWidgets version 2.6 or later (newer versions should also work, but 2.4 and below definitely don't work).

wxWidgets can be downloaded from http://wxwidgets.org/. If you're running Windows, you might have luck with wxPack, which can be downloaded from http://wxpack.sf.net/.

You will also need the Styled Text Control (stc) installed. This is distributed with wxWidgets, and can be built by using the build scripts found in *[Your Build Directory]*/contrib/src/stc in Linux/OS X, or *[Your Build Directory]*/contrib/build/stc on Windows.

The `wx-config` application called by the configure script should automatically reference all appropriate headers and library files for wxWidgets.

2.1.2 Boost

Boost is a set of portable libraries allowing features such as threading and regular expressions to be used reliably in cross-platform applications. Cellulate requires version 1.33 or later. It can be found at http://boost.org/.

Boost needs to be compiled and installed (or staged). Cellulate uses the boost_iostreams, boost_regex and boost_thread libraries. Note that the boost_iostreams library needs to be build **with** bzip2 support.

2.1.3 Berkeley DB (bdb)

Berkeley DB (bdb) is a simple yet very fast database library, used by Cellulate to store previous simulation runs. Cellulate requires version 4.2 or later. It can be downloaded from http://www.oracle.com/database/berkeley-db/index.html.

Cellulate requires the C++ library (db_cxx) to work properly.

2.2 Node Requirements

To work successfully, the node - the small application you need to run on each of the slave computers - needs to have access to a C++ compiler.

To maximise speed, the cell processing script you enter is compiled when you start the simulation, for which a C++ compiler on the node machine is required. You can use any compiler that is capable of generating libraries, but the use of programs other than g++, MinGW or the Visual Studio compiler will probably require the editing of the build configuration file, and aren't tested.

By default, on Linux or OS X, the GNU C++ Compiler (g++) is used. The g++ compiler is available with most Linux distributions, although you may need to install it through the package manager. On OS X, you will need to install the Developer Tools (available on the OS X installation CD).

On Windows, the use of MinGW is recommended. MinGW can be downloaded from http://www.mingw.org/ and the path will probably have to be edited in the build configuration file (see below).

2.3 Hardware Requirements

Running the simulation is expensive in terms of CPU, memory, and disk usage. Ideally, you'll want as much of each as possible. Larger simulations, or simulations with more variables, will require more resources and time to run properly.

At an absolute minimum, you will need around 96MB of memory free on each node, 128MB of memory free on the machine co-ordinating the simulation, and enough space on the co-ordinating machine to store the generated data. As a guide, a 100x100x100 simulation running for 1 iteration with a single boolean variable would require around 1MB of storage space.

It is strongly recommended that every machine has as much memory as possible. It is also worth bearing in mind the simulation speed will be limited by the slowest machine, since every machine will need to be at the same stage before one iteration of the simulation can be completed (although Cellulate will try and balance the workload when it can).

3 Getting Started

This section describes how to get Cellulate up and running. It assumes the libraries mentioned in the 'Requirements' section have been installed properly.

3.1 Installing from Binaries

In most cases, you can simply open the binary installer and follow the instructions. You do not need to install the libraries if you're installing this way. See the accompanying readme file for more detailed instructions.

3.2 Building from Source

If you want to compile Cellulate from source, this is the section for you. If you have opted for the pre-built binaries, skip this section.

3.2.1 Getting the Source

In most cases, you will want to download the most recent source tarball from the project site (http://cellulate.sf.net/). The current (and potentially broken) source can be checked out from the SourceForge subversion repository (SVN). The following command should check out the current version of the source to a 'cellulate' directory:

```
svn co https://cellulate.svn.sourceforge.net/svnroot/cellulate/trunk cellulate
```

3.2.2 Building the Source

Change to the newly-created trunk directory. We now need to run the included configure script to set Cellulate up for your machine. To do this, type in `./configure'. You may need to specify some of the following options:

- `--with-bdblib [PATH]`: This is the path to the Berekely DB library files.
- `--with-boostinc [PATH]`: The path to Boost's include files.
- `--with-boostlib [PATH]`: The path to Boost's library files.
- `--with-wx-config [PATH_TO_WX-CONFIG]`: Path to the wx-config application.
- `--prefix [PATH]`: The prefix to install all the files to. This is useful if you want to run Cellulate as a non-admin user; you can specify a path you can write to, and everything will be dumped there.

These should be the only options you need in the majority of cases. You can also use --help to get a listing of all the options you can specify.

When the configuration has finished, run `make`, followed by `make install` as root to put everything in the proper place.

4 Running Your First Simulation

This section talks you through running your first simulation. For this example, we will be implementing the Game of Life in 2D, which is arguably the most famous example of a cellular automata.

4.1 Starting Cellulate and Cellunet

The first thing we will need to do is start Cellulate. Cellulate is the program responsible for editing the simulation, managing the connected computers ('nodes'), and viewing the results. To start Cellulate under Windows or OS X, just double-click the application icon. Under Linux, you will need to run it from the command line; if it has been installed in a directory in your $PATH, you should just need to type in `cellulate`.

You should be presented with the window below.

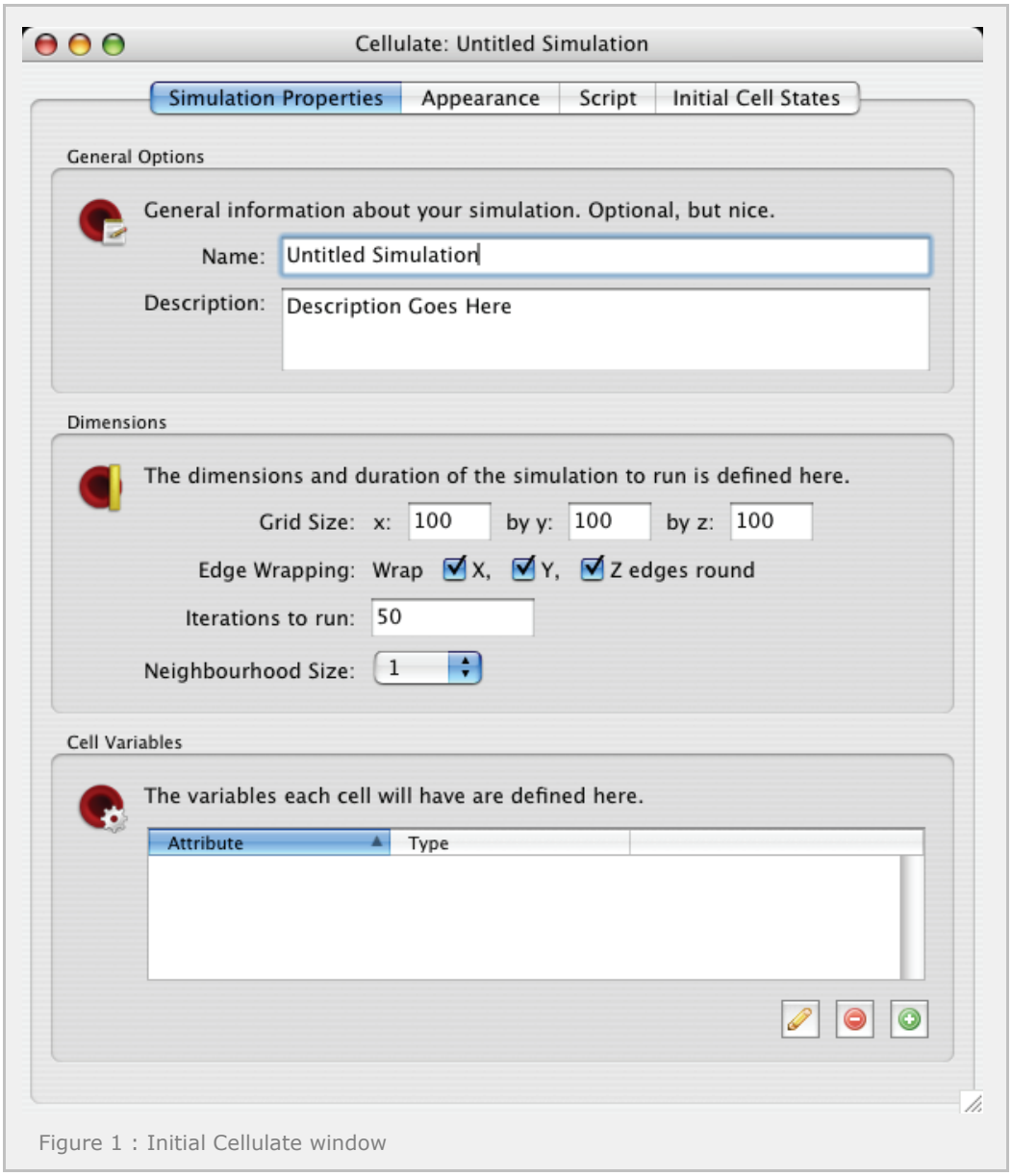

Figure 1 : Initial Cellulate window

At this point, it's probably a good idea to start at least one instance of Cellunet. Cellunet is the command-line application that does the actual processing of the cellular automata. It's designed to be run simultaneously on several machines, all of which connect to the original Cellulate application to report their results.

4.1.1 Finding Cellulate's IP Address

The first thing you will need to know is the unique network address ('IP Address') of the computer running Cellulate. The method of discovering it varies a bit depending on what OS you are using.

4.1.1.1 Windows

1. Go to the Start menu, Press Windows Key+R, type 'cmd' into the box that appears, and click 'OK'. You should be presented with a command console.
2. Type in `ipconfig /all`. You should get a list of properties; you are looking for the set of numbers under 'IP Address'.

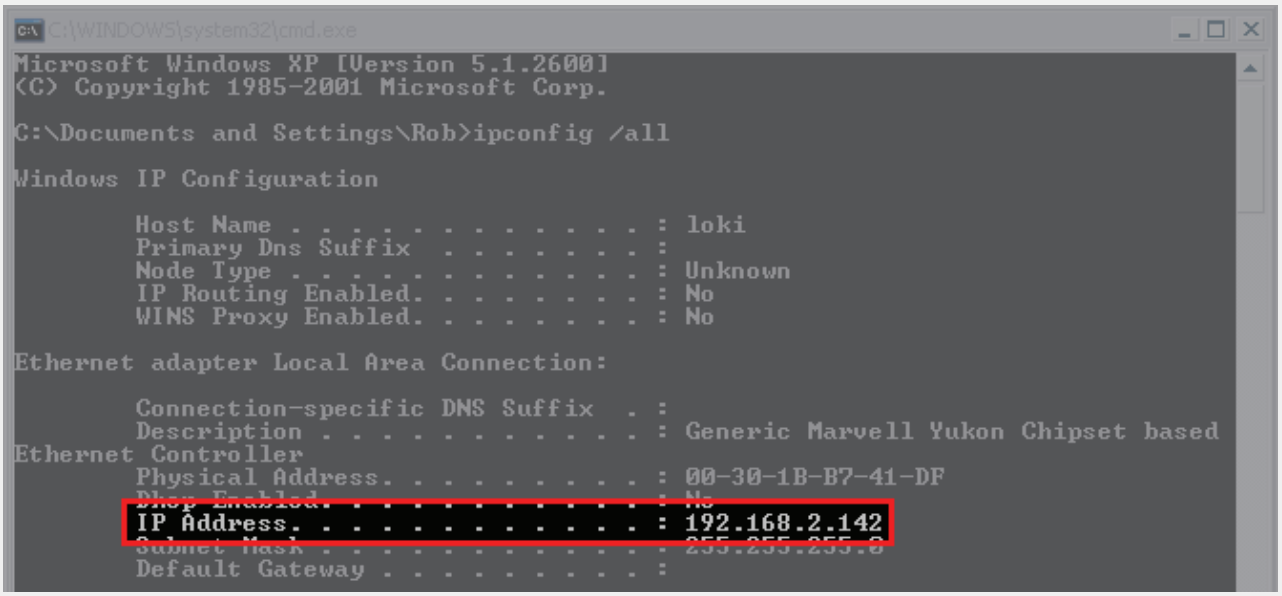

Figure 2 : Windows IP Address

4.1.1.2 Mac OS X

1. Go to the Apple menu, choose 'System Preferences...', click 'Network', and double-click on the connection you will be using to network the computers together.
2. Click the 'TCP/IP' tab. Your IP address should be shown on this page.

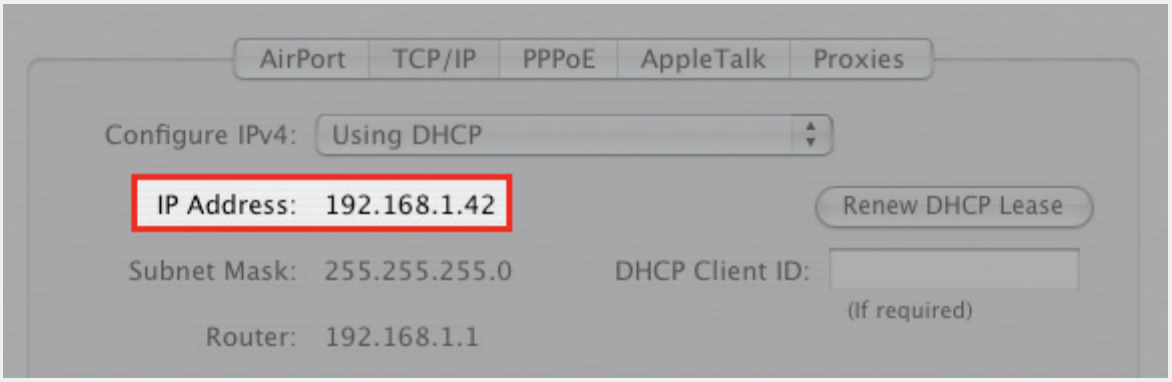

Figure 3 : OS X IP address

4.1.1.3 Linux

1. Start a console, and type in '/sbin/ifconfig'.
2. Under the heading for the interface you use to network the computers together, your IP Address should be shown after `inet`. You don't want the similar number shown after 'broadcast'; ignore that.

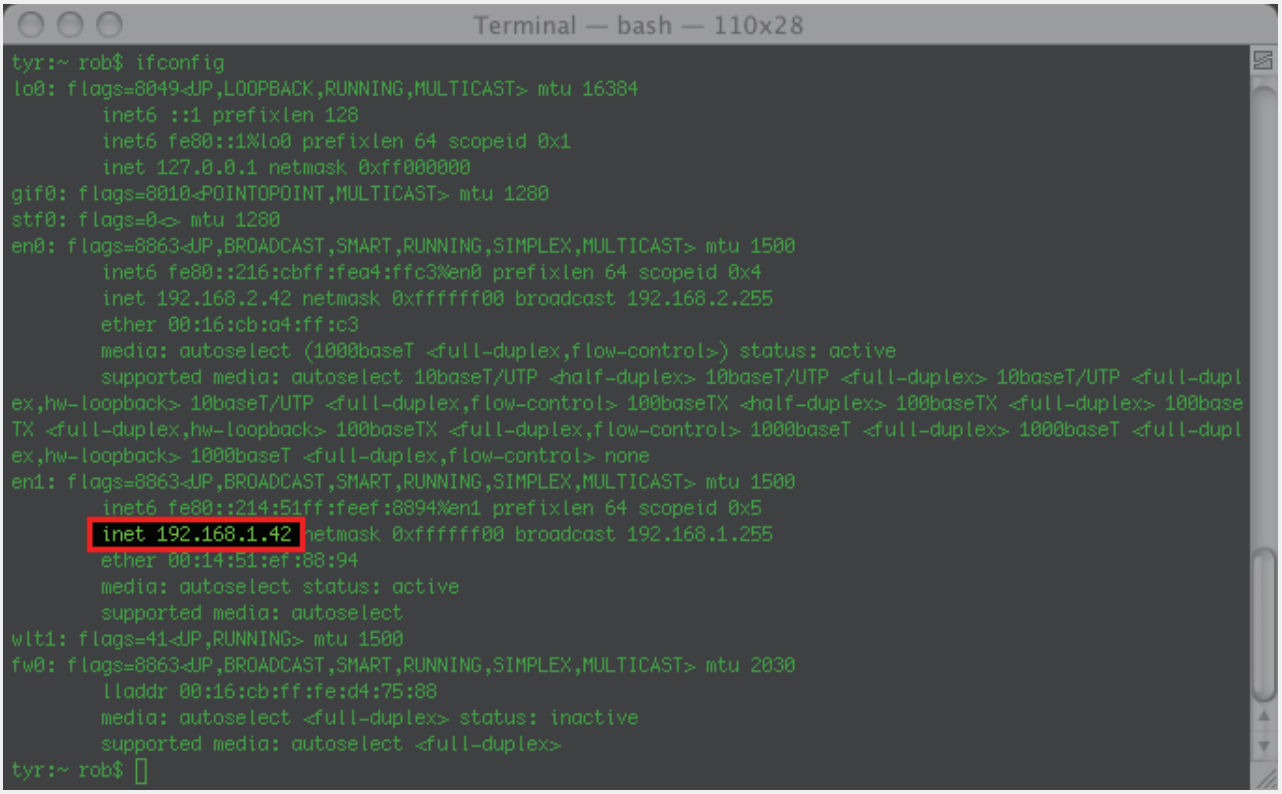

Figure 4 : Linux IP address

4.1.2 Starting Cellunet

Armed with your Cellulate IP address, start a new console on a machine you want to use for processing. This can be the same machine you're running Cellulate on, but for large simulations this is a really bad idea, and so is discouraged. Type in the path to the Cellunet command (or drag it's icon to the console window). Now, you will need to add some options to tell it which machine to connect to.

You can do this by typing ' -a [THE IP ADDRESS]' after the cellunet command you've typed in. The IP address is the IP Address you discovered above; the address of the computer running the Cellulate application.

Assuming everything goes okay, you should see something like the screenshot below. If you get errors about 'Could not connect to server', check the IP address you entered is correct and the Cellulate is still running on the first machine.

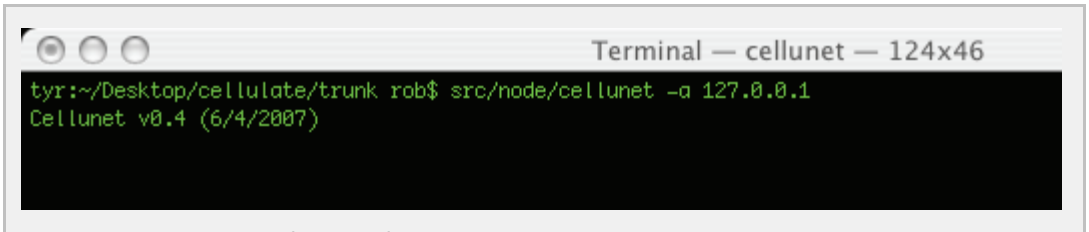

Figure 5 : Starting a celunet node

4.2 Setting Up Your Simulation

4.2.1 Entering a Name and Description

First things first: let's enter a name and description for our simulation. This is used to identify it and is optional, but it can help differentiate between simulations.

Make sure the 'Simulation Properties' tab is selected. For this example, we've entered the name and description shown below.

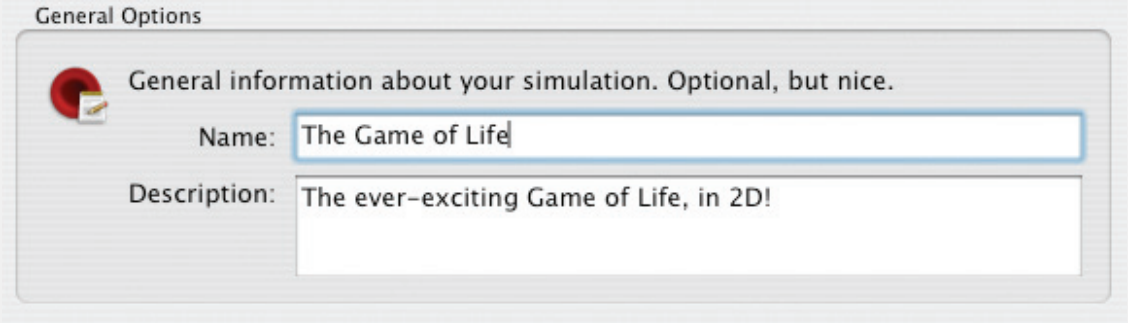

Figure 6 : The name and description of your simulation

4.2.2 Setting Sizes & Dimensions

Now, we need to define what dimensions the simulation should have. When doing this, it's important to bear in mind that bigger simulations will result in longer processing time, and consume more memory and disk space. It's also important to note that (right now) changing these values will cause the initial cell states to be reset; any initial states you have defined will be deleted.

As we are simulating 2D life, we will enter these values as shown below. Note that we have set one dimension to 1 as we want a 2D simulation. We have also turned off the Z wraparound. This governs whether cells at one edge will affect cells on the opposite edge. Leaving this turned on will cause any cell to be its own neighbour, and will cause unexpected results.

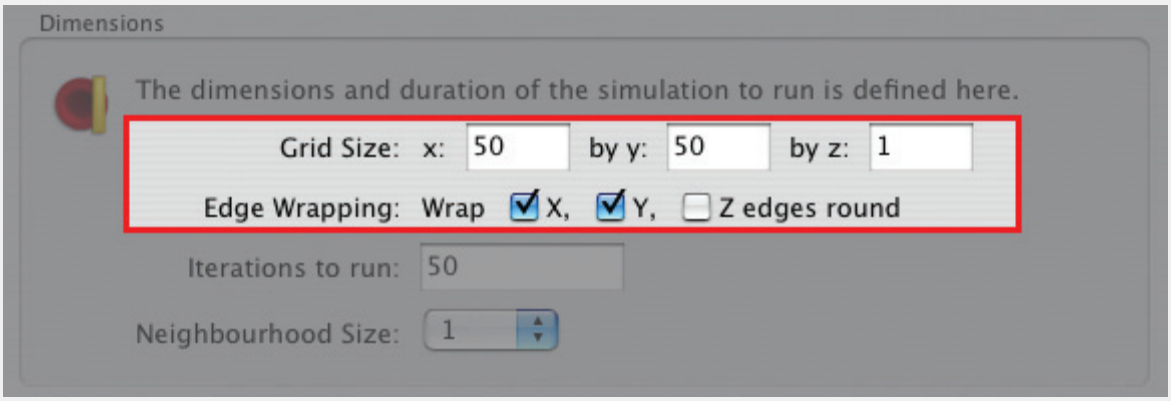

Figure 7 : Initial Rule Editor Window UI Design

We choose to let the simulation run for 50 iterations, and set the neighbourhood size to 1. The neighbourhood size describes the number of cells that any particular cell can be affected by (or affect). A neighbourhood size of 1 means that only the cells touching a given cell (including diagonals) can be affected by it - 26 cells in total. A neighbourhood size of two would increase this to be the 124 closest cells, and so on. Processing times and network traffic are affected by this.

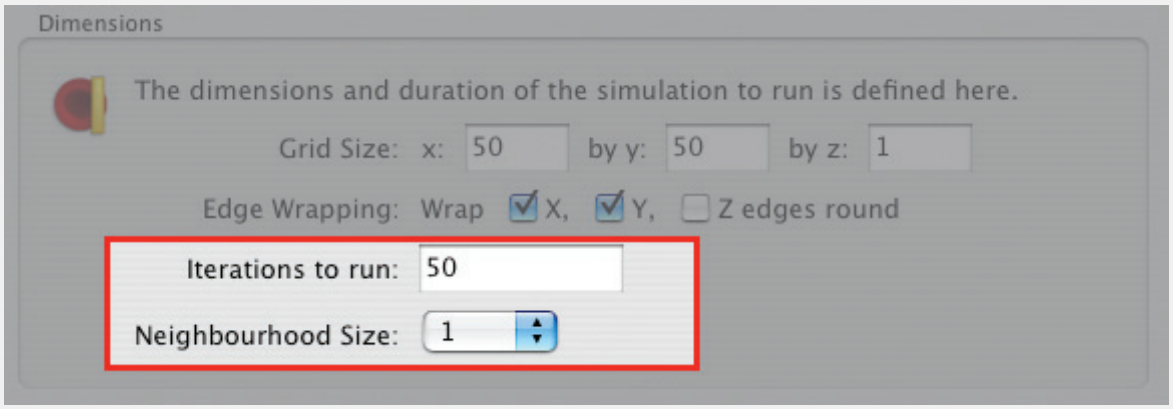

Figure 8 : Iterations and neighbourhood size

4.2.3 Creating Variables

The next thing to do is to define some cell variables. Variables can be used to store information about the current state of a cell; you may have a variable that represents how long a cell has been alive, or a variable that represents the amount of food a cell has. You can change a variable when the simulation runs using the script, and set an initial value for it before the simulation begins (more on that later).

As with the simulation size, adding, deleting or changing a variable will delete any initial states you have defined.

Initially, you need to tell Cellulate what variables you will be using, as well as what type they will be. A variable can be defined as a boolean, in which case the only values it can have are true or false, an integer of varying sizes, or a floating point decimal number.

For the game of life, we will only need one variable, which will represent whether the cell is alive or dead, which we will call 'cellalive'. To add it, first click on the plus icon below the list of variables:

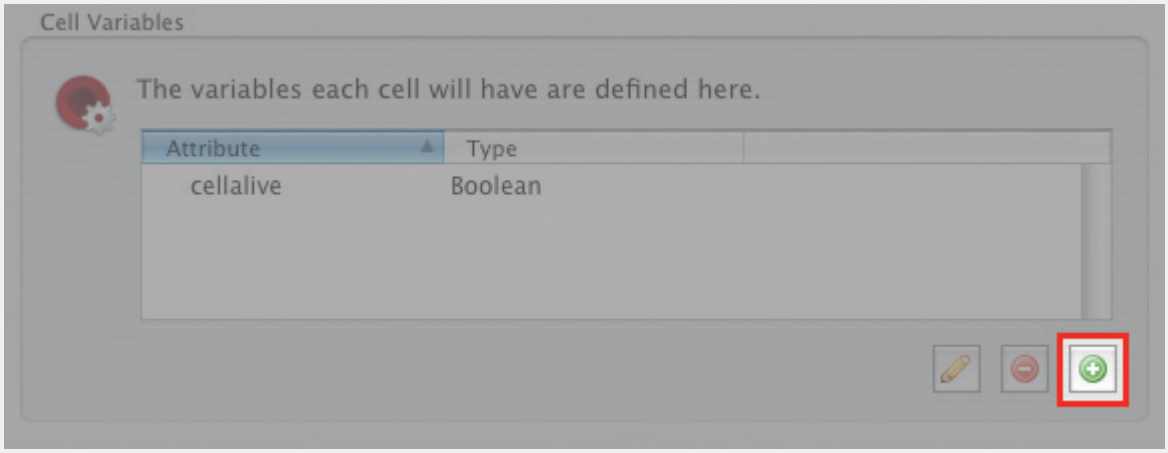

Figure 9 : The 'add variable' button

You should see a dialog like the one below. Enter the name as 'cellalive' and change the type to boolean since we will only need two states - 'true' to represent a living cell, or 'false' to represent a dead cell.

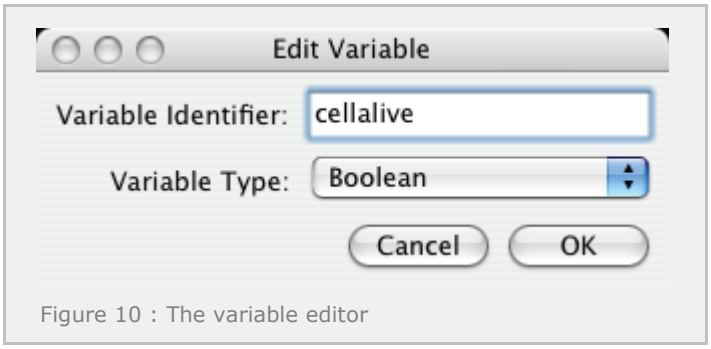

Figure 10 : The variable editor

The different sizes of integer, as well as the boolean type, are provided so as to minimise the amount of space the simulation will take up when finished. While you could, as an example, use a tiny integer instead of a boolean for representing 'cellalive' in the Game of Life, it could increase the amount of data generated by around 8 times!

4.2.4 Creating Cell Colouring

Now, we need to tell Cellulate how to colour the cells. We do this by defining specific colours for specific values of cell variables. Any cells not explicitly assigned a colour will not be shown. In this example, we will want living cells (cells whose 'cellalive' variable is exactly '1') to be shown in a nice masculine pink colour, and non-living cells not to be shown.

Click the appearance tab along the top of the window, and you should be taken to a screen with a list of colours. It should be empty right now. To add a colour, click the plus icon below it, and you will be presented with a window like the one shown below.

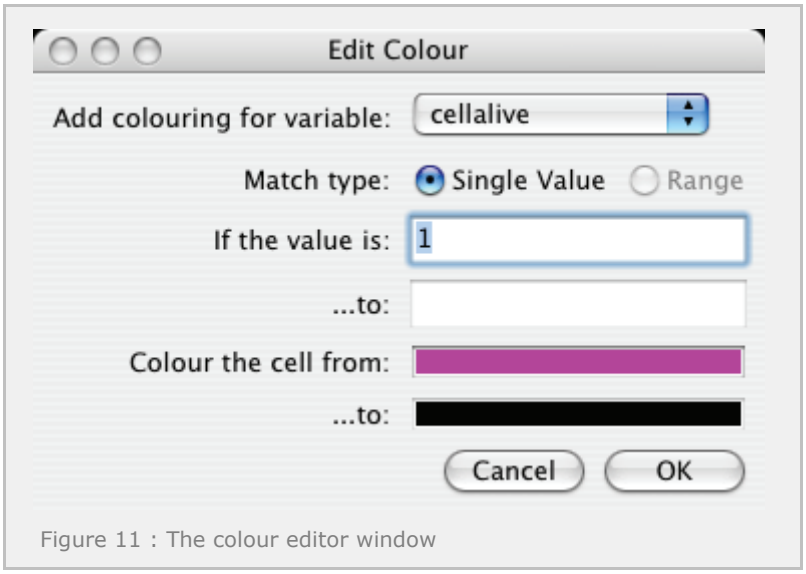

Figure 11 : The colour editor window

Firstly, ensure 'cellalive' is selected in the pop-up box at the top. This tells Cellulate which variable this colouring rule applies to. Next, we will need to choose whether we apply the colour to a single value of this variable, or a range of values. As this is a boolean and can only have two values, the range option is disabled, making the choice an easy one.

Next, we need to specify a value, which we will enter as '1' - this tells Cellulate to only apply this to cells that have the value '1', and only '1'. We don't need to enter an end value, as we are not entering a range.

Finally, we need to pick a colour. Click on the box next to the 'choose a colour...' label, and a colour picker should appear. Choose a colour you like the look of.

If you had wanted to use a range - which you must do for a floating-point variable - you would also need to specify an end value and an end colour in the same way. Cellulate would then gradient the cells according to where they lay in the range. Cells closer to the beginning value would have a colour similar to the starting colour you specify, cells closer to the end value would have a colour closer to the

end colour, and cells with values in the middle would be a blend of the two colours. If you want a range to be the same colour, you will need to make the start and end colours the same.

Now we are done with this dialog, you can press 'OK' to have Cellulate add it to the list box. A new row should appear, looking a little like:

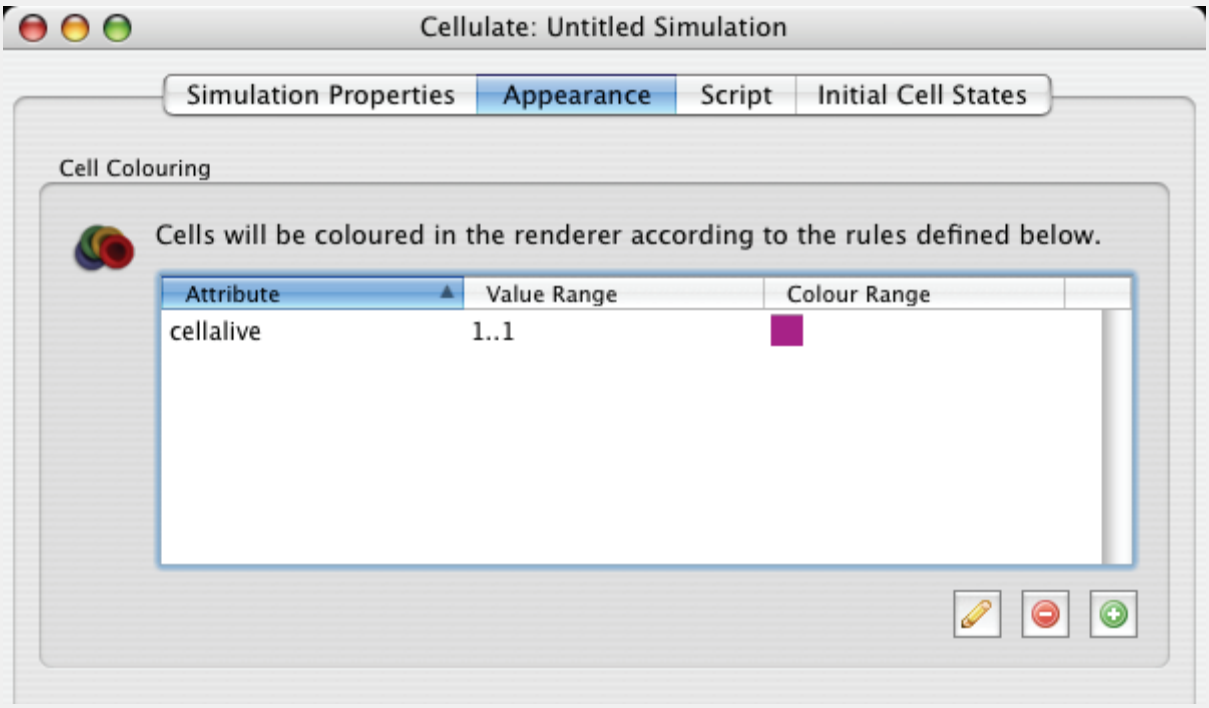

Figure 12 : Initial Rule Editor Window UI Design

It's worth remembering that if two or more colouring rules match one cell, the top matching colour will be the one that's used.

4.2.5 Defining the Cell Script

The cell script is one of the most important parts of your simulation. This script is used to update a cell's state, and is called once for every cell, for every iteration. It is written using a subset of C++, and is compiled on every node before the simulation runs.

To enter a script, click the 'Script' tab along the top of the main window. You should be presented with a screen similar to the following:

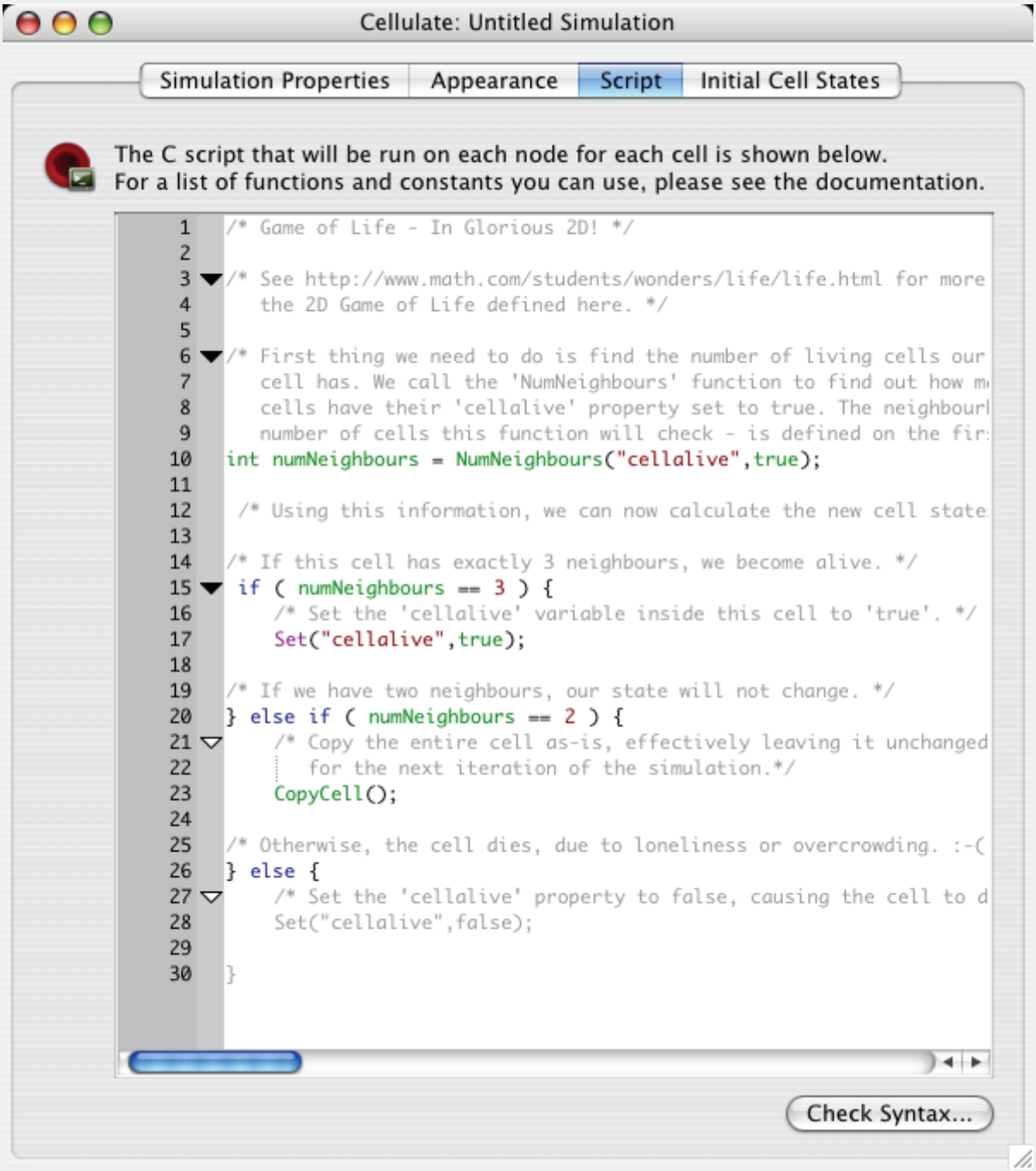

Figure 13 : Script editor window

The script you need to enter to simulate the 2D Game of Life is shown below.

```
/* Game of Life - In Glorious 2D! */

/* See http://www.math.com/students/wonders/life/life.html for more info on
   the 2D Game of Life defined here. */

/* First thing we need to do is find the number of living cells our current
   cell has. We call the 'NumNeighbours' function to find out how many neighbouring
   cells have their 'cellalive' property set to true. The neighbourhood size - the
   number of cells this function will check - is defined on the first setup page. */
int numNeighbours  NumNeighbours("cellalive",true);

 /* Using this information, we can now calculate the new cell states. */
```

```
/* If this cell has exactly 3 neighbours, we become alive. */
 if ( numNeighbours == 3 ) {
     /* Set the 'cellalive' variable inside this cell to 'true'. */
     Set("cellalive",true);

/* If we have two neighbours, our state will not change. */
} else if ( numNeighbours == 2 ) {
     /* Copy the entire cell as-is, effectively leaving it unchanged
        for the next iteration of the simulation.*/
     CopyCell();

/* Otherwise, the cell dies, due to loneliness or overcrowding. :-( */
} else {
     /* Set the 'cellalive' property to false, causing the cell to die. */
     Set("cellalive",false);

}
```

There are several things you should be aware of when writing scripts:

- A script can only check the values of cells inside its neighbourhood. Cells outside this neighbourhood will return an incorrect value.
- A change to the value of a variable inside a cell is only visible to surrounding cells when the current iteration is over.
- C++ variables defined in the script (e.g. numNeighbours above) are lost when it has finished executing after every cell. A cell cannot access a C++ variable defined while a previous cell was executing; variables should be thought of as going out of scope when the end of the script is reached.
- The use of the `new` or `malloc` keywords inside a script is not recommended. If you do use it, be very, very sure there are no leaks at all. Even the smallest leak will result in an obscene amount of memory being used.

When you are finished writing a script, you can press the 'Check Syntax...' button at the bottom of the window. If a node is connected, the program will ask it to test the script you've written. Any errors it encounters will be reported back to you.

For more information on writing scripts and the methods available to you, see the section on scripting later in this manual.

4.2.6 Defining Initial Cell States

Now we need to define what value each variable will be on the first iteration. This will form the basis of the states in the rest of the simulation; cell patterns and states in the next iterations will evolve from this.

In this example, we will create a glider. This is a group of cells that doesn't change in size, but moves slowly across the map by (appearing to) rearrange its cells. For more information on gliders and other interesting shapes you can make in the Game of Life, see http://www.math.com/students/wonders/life/life.html.

To create the glider, we will use the initial state editor; change to this using the 'Initial States' tab at the top of the main window. You should see a panel like the one below.

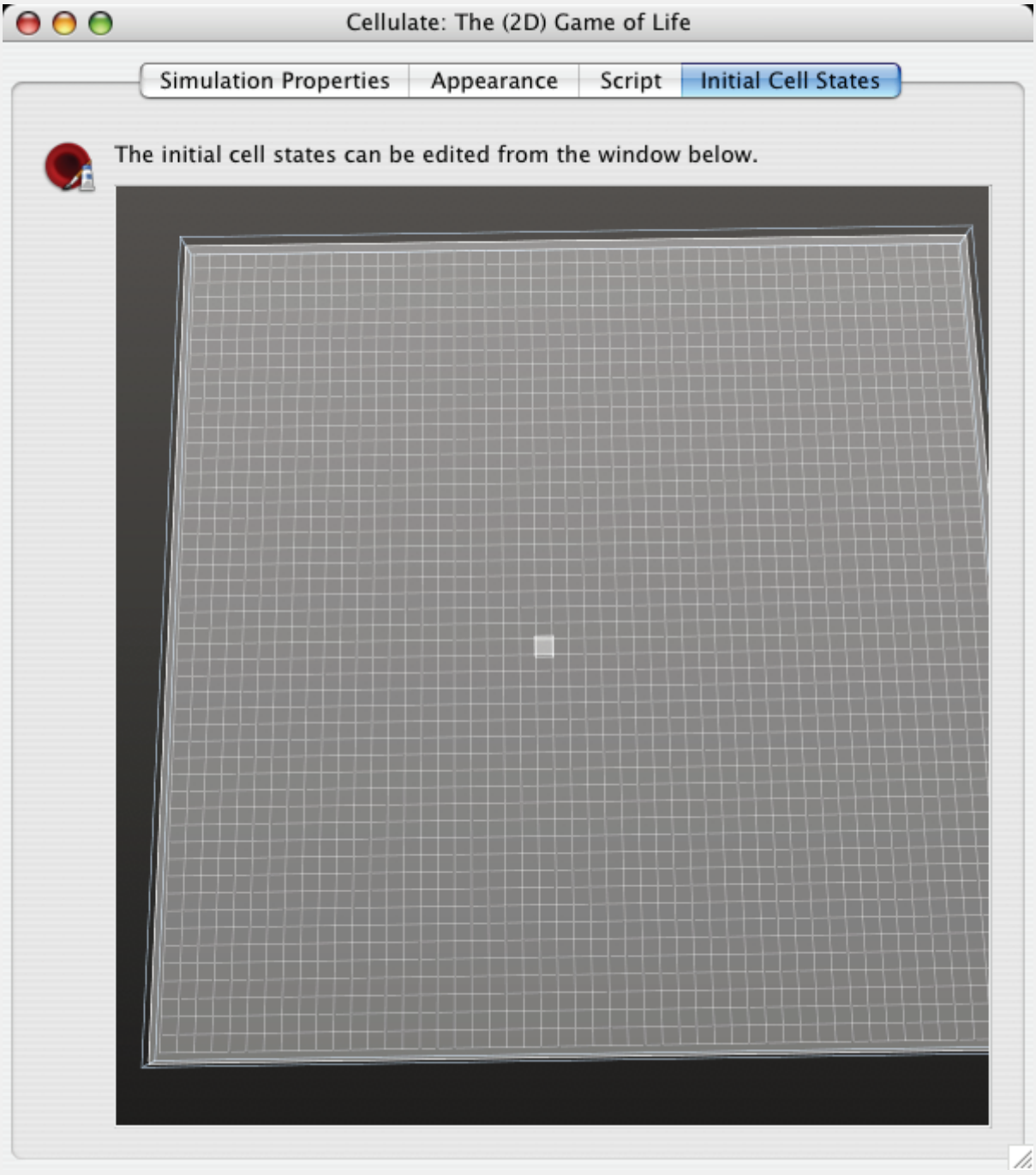

Figure 14 : Initial states editor pane

This shows your current simulation, and will probably be empty. Notice the select plane; this is the white grid overlaid on top of the simulation.

There are a few controls you will need to know at this point. Firstly, clicking and dragging with the left mouse button will rotate the model shown inside; you may need to do this so you can see the cell you want to edit. The scroll wheel (or the Page Up and Page Down keys) will zoom in and out. The right mouse button will allow you to move the simulation.

To select a cell, double-click somewhere on the simulation. If there's a coloured and visible cell there, it will be selected with a white box, and information about that cell will appear on the cell inspector. If there is not a visible cell, the corresponding empty cell intersecting the point you click and the select plane will be highlighted.

To create our glider, double-click on a empty cell on the grid. It should be highlighted in white, and the inspector window (shown below) should update to list the variables defined in that cell. We need to

change the 'cellalive' value to true; that is, mark the cell as being alive. To do this, click on the drop-down box next to 'cellalive' on the inspector, and choose 'true', changing the value of the variable.

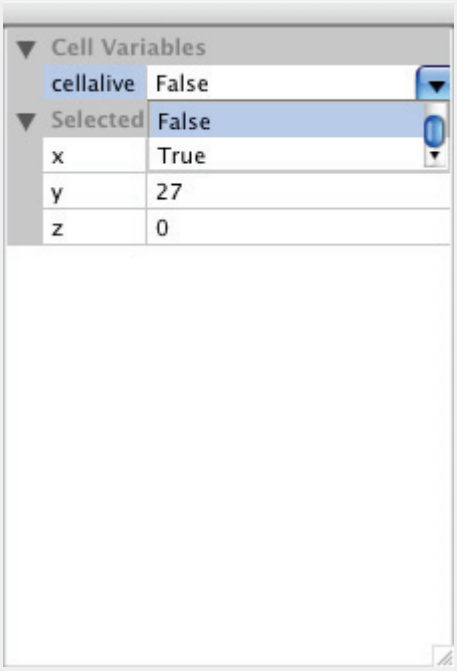

Figure 15 : Variable inspector

Keep doing this until you have a shape that looks like the one below. This is the simplest example of a glider in the game of life.

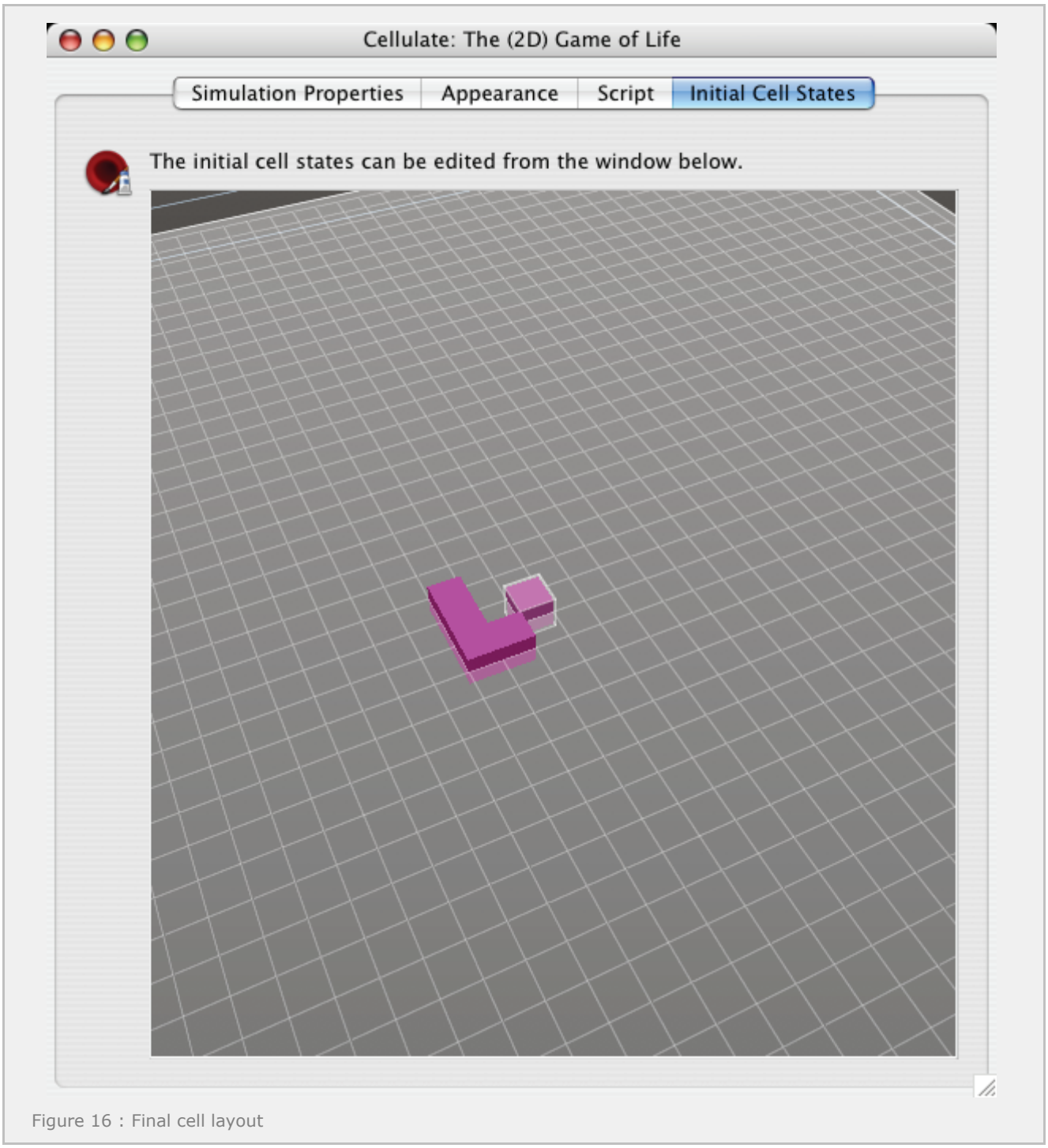

Figure 16 : Final cell layout

4.2.7 Running the Simulation

We are pretty much done setting up the simulation now. Save it (choose 'save' from the 'file' menu), and we are ready to run. Choose 'Run Simulation...' from the 'Simulation' menu. The program will check there is a node connected; if not, will prompt you to start one locally for testing purposes. You should get a window that looks like the one below.

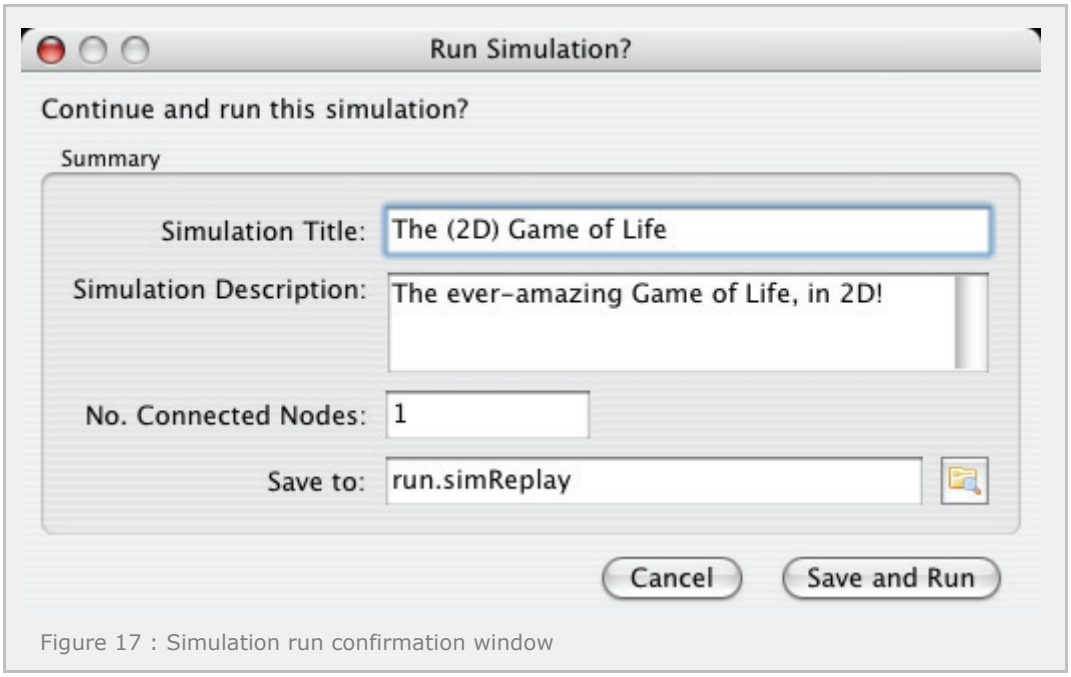

Figure 17 : Simulation run confirmation window

Check that everything's okay, especially the number of connected nodes, which should correspond to the number of machines you are running cellunet on (one in this example). Then, choose where to save the replay file to - you can do this with the folder button to the right of the 'Save to:' field. When you've done this, click 'Save and Run', and Cellulate will send the script out to all connected nodes and request they compile it. If something goes wrong, Cellulate will give you the errors the node reported back to it, otherwise you should see a window like the one below.

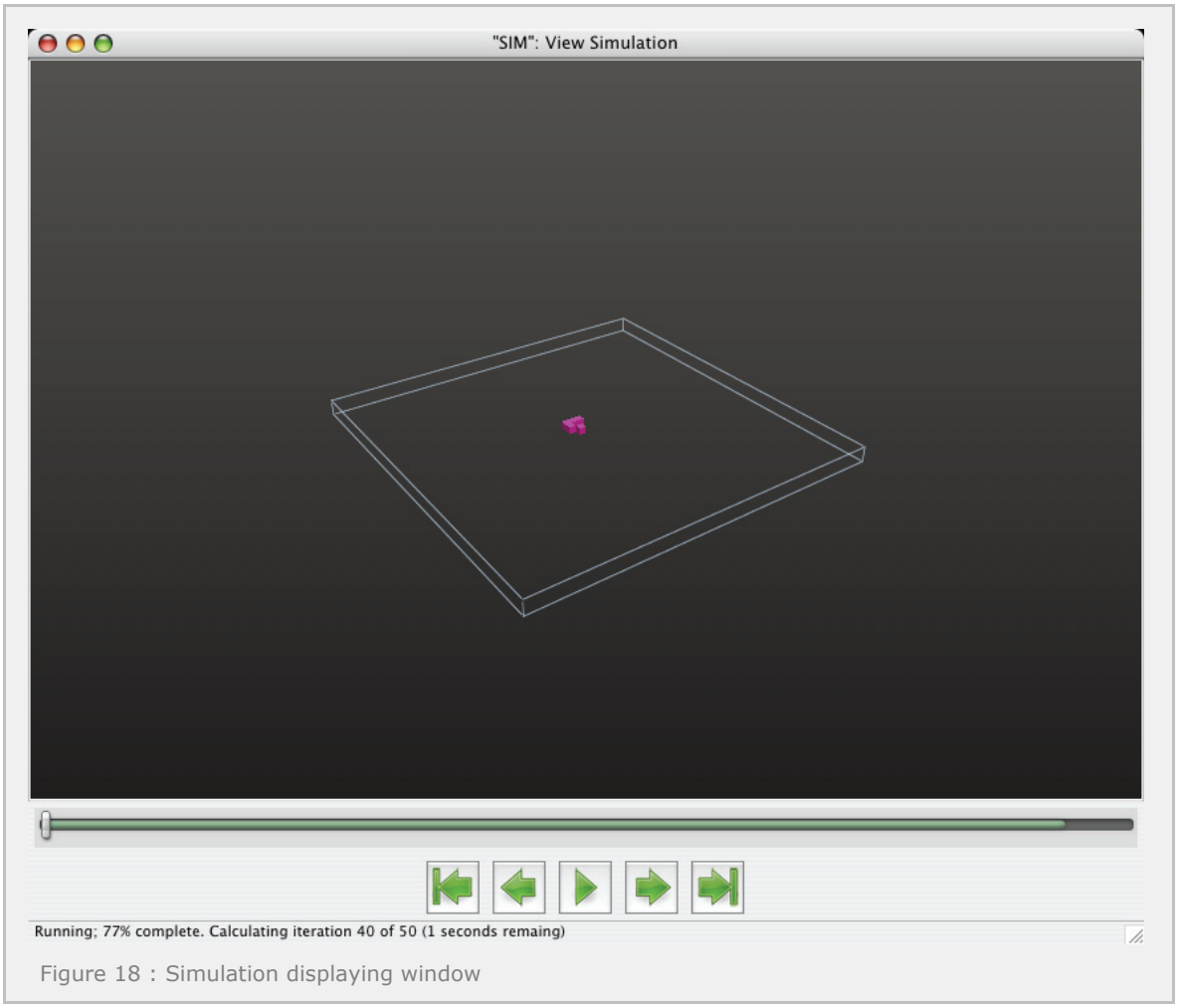

Figure 18 : Simulation displaying window

This shows the result of the simulation process. The cells shown are those defined in the first state; this is the initial state you defined in the editor window. To move to the next iteration of the simulation, click the right arrow underneath the renderer display; this should move it to the next iteration. You may notice the cells turn grey when you do this. This signifies the renderer is in the process of redrawing those cells, and they should change colour soon. You will hopefully see that the glider starts to 'move' diagonally across the grid.

The amount of the simulation that's been calculated is represented by the green line in the slider above the buttons. The further towards the end it gets, the closer the calculation is to finishing. The handle on the slider represents your position within the simulation, and you can drag this around to change the frame you are looking at.

Manipulating the view shown by the renderer pane is similar to changing the view in the initial states window. As with the initial states window, there is a floating inspector that shows the state of the currently selected cell, which you can change by double-clicking on a cell in the window.

You can also display a clipping plane. This plane will hide all cells that are behind it, allowing you to select cells in the middle of a group. This isn't really that useful in this example simulation, as in a 2D simulation you can see all the cells. Using it is mentioned later in the manual.

For more information about the window, including the controls you can use to manipulate the renderer display, as well as the clipping plane, please see the chapter on this window later in the manual.

When you are done watching the simulation, you can close the window to return to the editor. The replay file you chose earlier will contain the data you generated, and can be watched again by selecting 'Replay Simulation...' from the file menu.

5 Using Cellulate

This section will give more detailed information on accomplishing a task using cellulate, as well as providing labelled screenshots to show you what the different buttons and features are.

5.1 Editing a Simulation

The simulation editing window is the window you will probably see most of. It allows you to edit a simulation, run a simulation you have created, or edit the application's preferences. See later sections for information on these tasks.

5.1.1 The 'Simulation Properties' Pane

This pane is used for editing general properties of a simulation. It should ideally be completed before moving on to other tabs; you must have defined at least one variable before you can switch to a different tab.

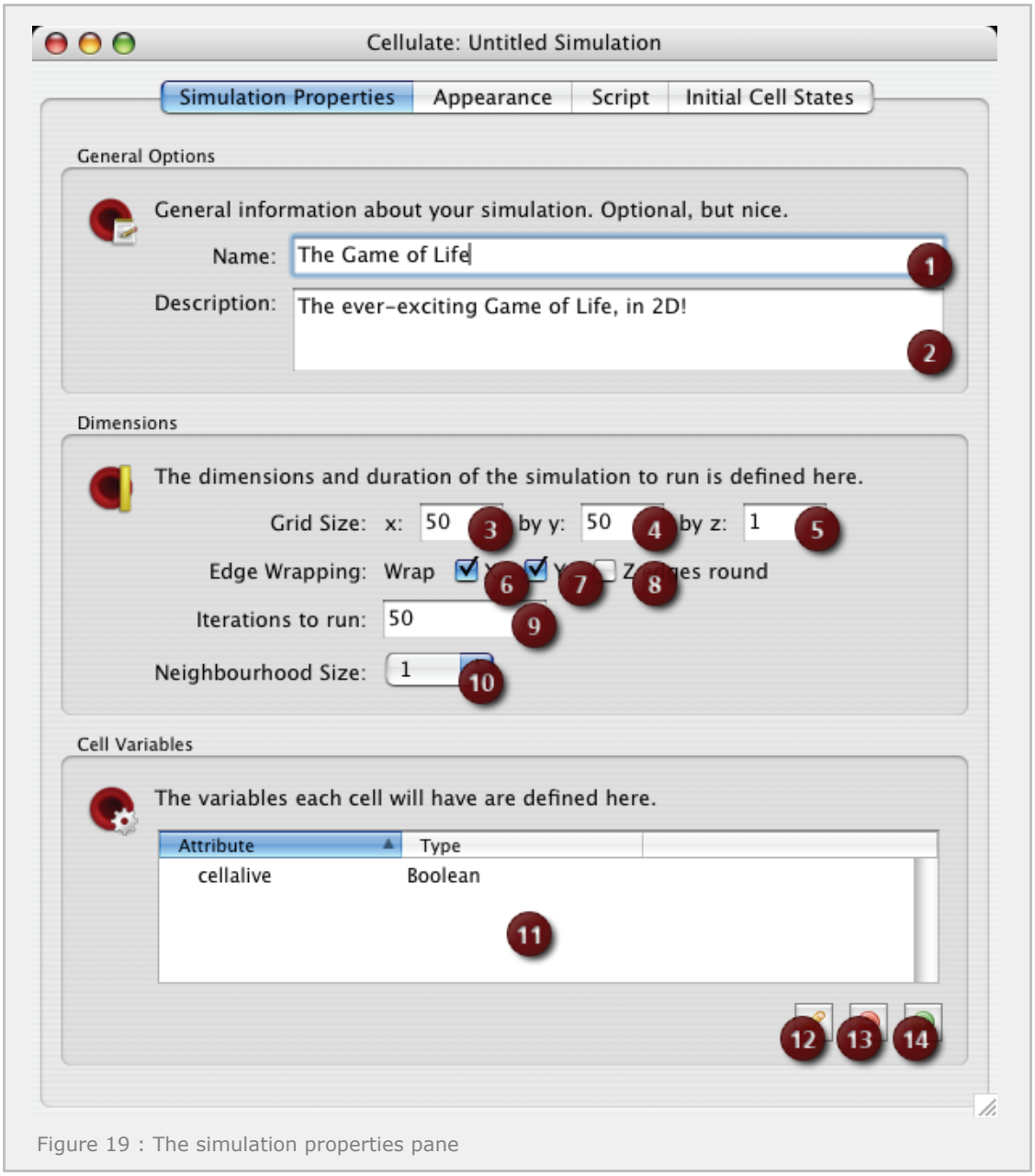

Figure 19 : The simulation properties pane

1. The name of the simulation. This is optional. This does not affect the simulation itself, but can be useful for labelling and describing the simulation.

2. The simulation description. As with the name, this is optional, but can help describe what a simulation is supposed to model.
3. The size of the simulation grid in the x-dimension. Changing any sizing attributes of the simulation will cause the initial states to be reset. Increasing the dimensions will increase the time the simulation takes to run, and the amount of space it takes to store the result.
4. The y-size of the grid.
5. The z-size of the grid. You may want to make this '1' if you want to run a 2D cellular automata.
6. Whether to wrap cells around the x dimension. If this is selected, cells on the edges of the x axis (which have their x coordinate as 0, for example) will have the corresponding cell on the other side of the grid as a neighbour. This will effectively mean the cells wrap around, and cells on one edge of the grid affect cells on the opposite edge.
7. Wrapping for cells in the y dimension.
8. Wrapping for cells in the z dimension. If you're running a 2D cellular automata, you will need to turn this off as otherwise a cell would have itself as a neighbour. This would cause interesting things to happen.
9. The number of iterations the simulation should show. The first iteration is the set of initial states you define. Iterations after that will be calculated by cellulate based on the specified rules. More iterations will take longer to run, and require more storage space.
10. The neighbourhood size. This is the number of cells each cell will have as a neighbour. With a neighbourhood size of 1, every cell a distance of one away (i.e. touching the cell) will be considered a neighbour - this is 26 cells. With a size of 2, cells a distance of 2 away will be neighbours - this is the closest 124 cells. Increasing this will increase the amount of network traffic and potentially the time to run the simulation. On the other hand, a cell can only refer to other cells inside its neighbourhood. If you try to refer to cells outside it, the results are undefined.
11. This is a list of variables a cell may have. Scripts may refer to these variables in the current cell, or any of its neighbours. They may also set the variable in the current cell only. Try and minimise the number of variables you use, and their size; using more variables than required can slow down the simulation, as well as requiring a lot more space to store.
12. This button allows you to modify the cell variable you have selected in the list above. See the section later for a description of the window that appears when you push this button.
13. This will remove the currently selected variable.
14. This will allow you to add a new variable.

5.1.2 The 'Appearance' Pane

This pane allows you to set the conditions for which a cell should have a certain colour. Cellulate will look through the list of colours you have defined in this pane. When it matches one, the cell will be coloured according to that rule. If no matching rule is found, the cell will be transparent (and not shown).

Cellulate allows you to apply a gradient to a range of values of a variable. If the variable is near to the start of the range, the cell will be given a colour near the start of the colour range you specify. If it is near the end, the colour will be closer to the end of the colour range.

If more than one colour matches, the first matching rule is the one that is applied.

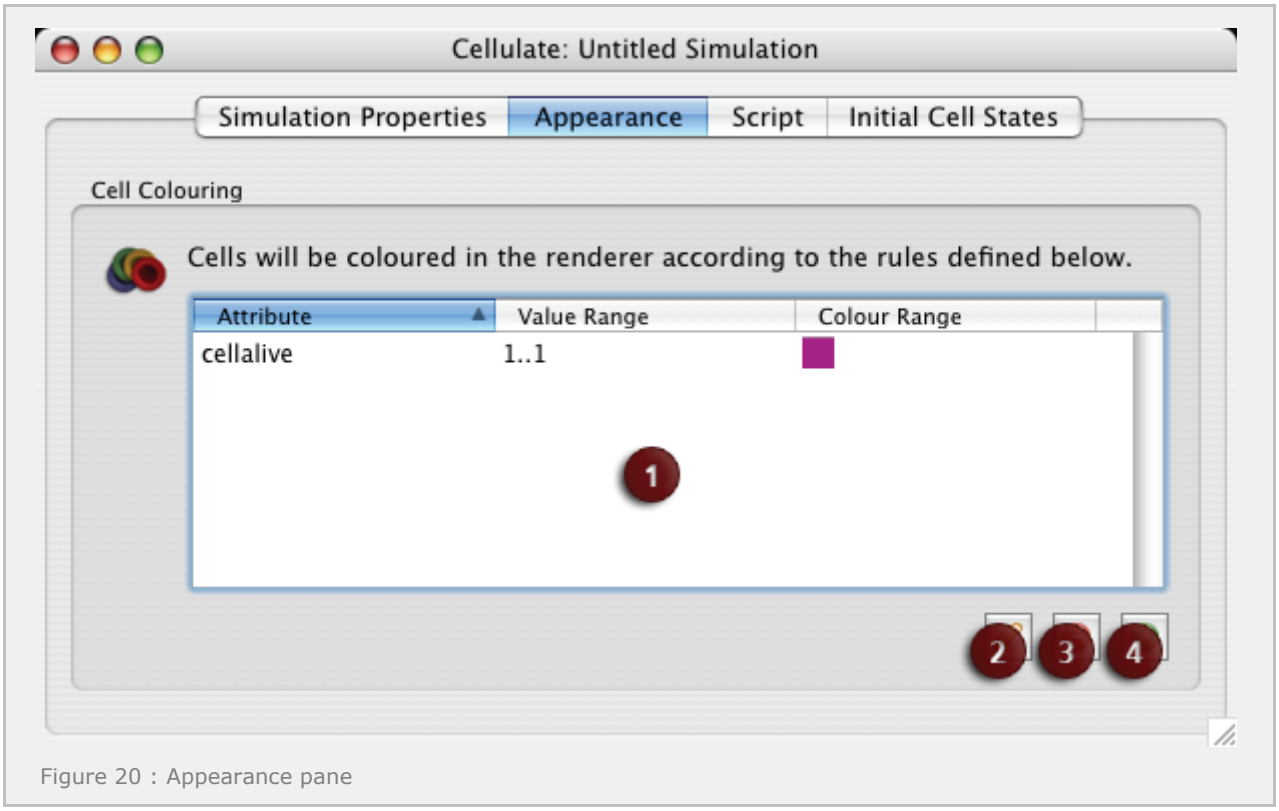

Figure 20 : Appearance pane

1. This shows the defined colouring rules. The variable the rule applies to, the range of values it applies to (if applicable; if it is just a single value, it will show it as '1..1') and the colour or gradient is shown in this box.
2. This allows you to modify the selected colouring rule.
3. This will remove the selected colouring rule.
4. This will show you a window for adding a new colouring rule to the list.

5.1.3 The 'Script' Pane

The script pane allows you to create a script for each cell in a subset of C++. The script you enter here will be sent to every node and compiled for maximum speed. For more information, please see the later chapter on scripting.

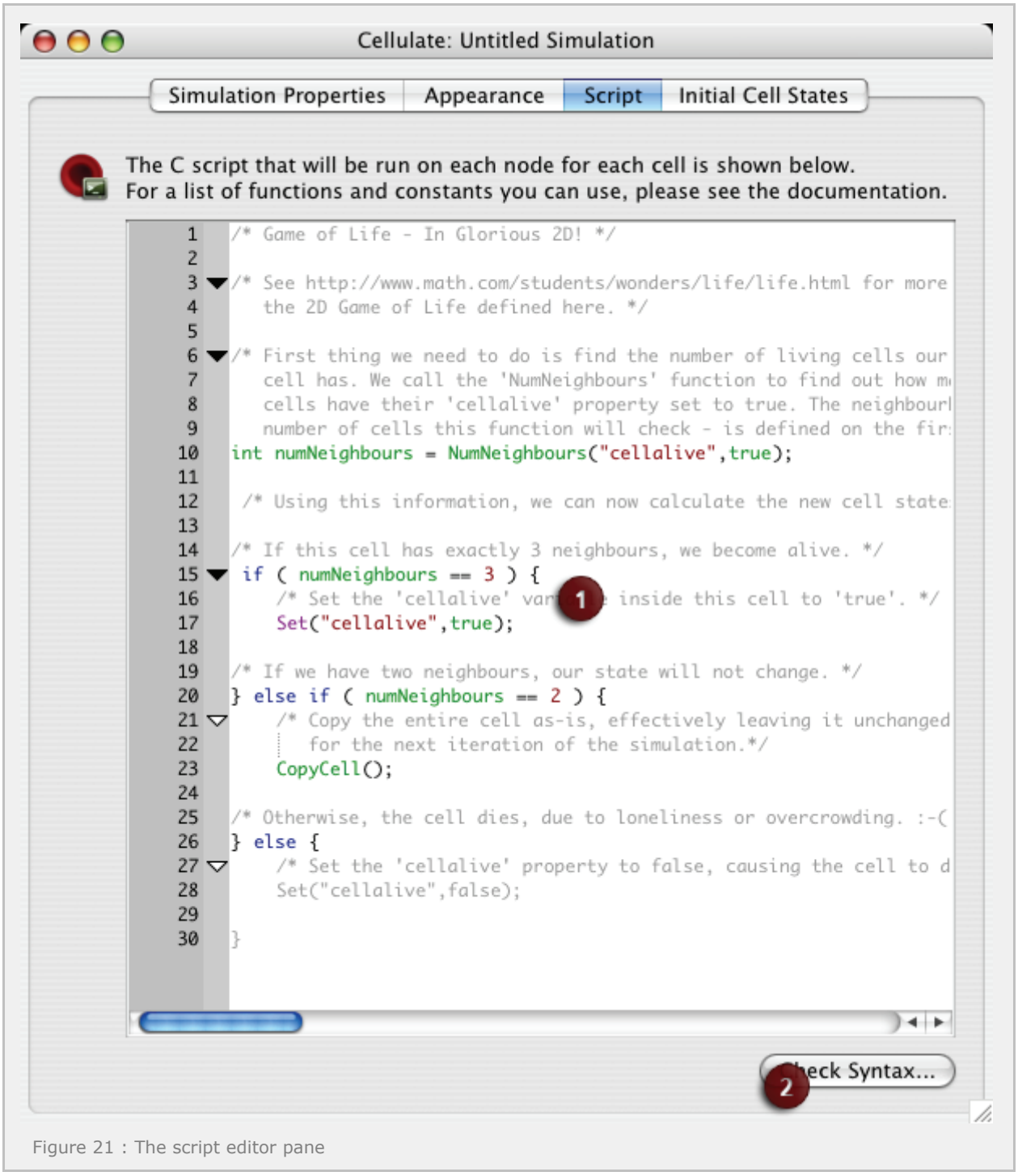

Figure 21 : The script editor pane

1. This is the C++ script that will be sent to the nodes for running.
2. This button sends the script to one node, to test the syntax and that it compiles properly. If compilation fails, the error message the node received from the compiler will be shown in a window, otherwise it will tell you that compilation succeeded.

5.1.4 The 'Initial Cell States' Pane

This pane allows you to define the initial states of cells in the simulation. Future iterations will be based on this initial 'seed' iteration. You can edit a cell by double-clicking it, and then changing the variables for that cell in the property list window to the side.

For a more comprehensive guide to manipulating the renderer, please see the appropriate section later in this manual.

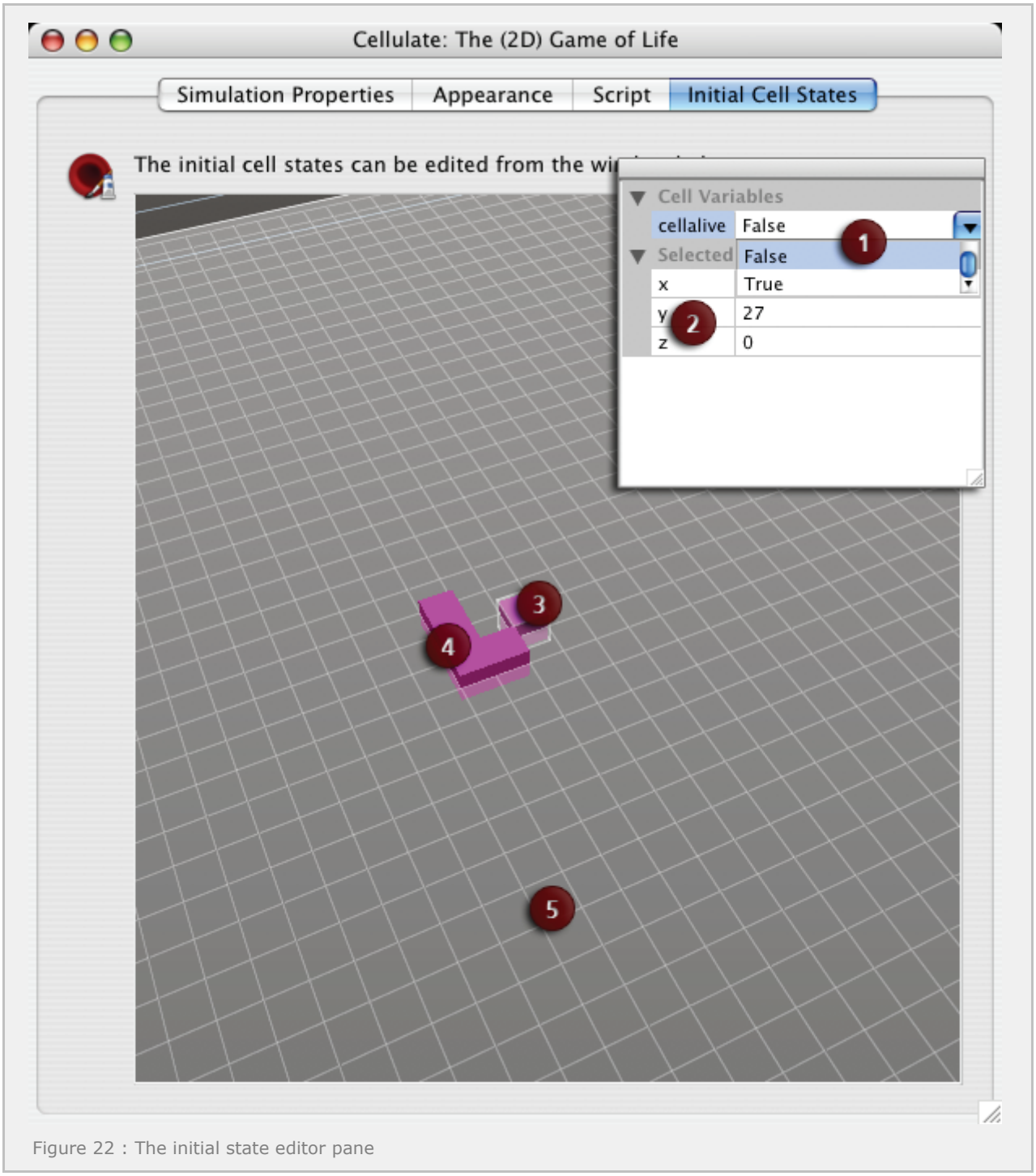

Figure 22 : The initial state editor pane

1. This section of the inspector shows the different cell variables the selected cell has. They can be edited by changing the values. The changes to colouring (if any apply) should be immediately visible in the states window.
2. This section of the inspector shows the x, y and z co-ordinates of the selected cell.
3. This is a selected cell (notice the white outline). Cells can be selected by double-clicking on them.

4. This is a normal cell. It is coloured pink because a colouring rule matched a variable in the cell.
5. This is the select plane. Normally, a cell with no interesting attributes set would not be visible and hence not selectable. When the select plane is shown, you can double-click on it, and the corresponding invisible cell will become selected. This allows you to edit variables in invisible cells. The select plane can be moved using the '+' and '-' keys. Double-right-clicking will change its orientation.

5.1.5 Editing a Variable

This window will appear when you choose to edit or create a cell variable. A variable can be one of the following types:

- A boolean: Either True (1) or False (0)
- A tiny integer: An integer number ranging from 0 to 15
- A small integer: An integer number ranging from -128 to 127
- A medium integer: An integer number ranging from -32768 to +32767
- A large integer: An integer number ranging from -2147483648 to 2147483647. This type should be avoided unless really necessary, since it requires a lot of storage space.
- A floating-point number. This uses the same amount of space as a large integer.

The large variety of types is provided to allow you to minimise the space a cell's variables take up.

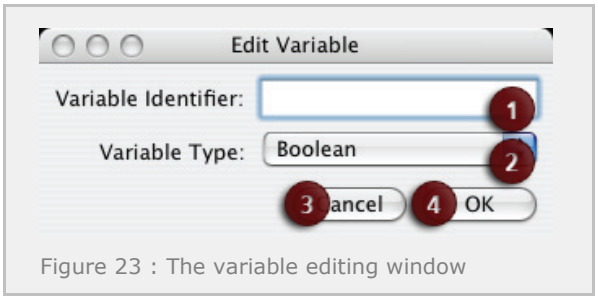

Figure 23 : The variable editing window

1. This is the name of the variable. The name is restricted to the characters a-z, A-Z, and 0-9. You will be warned if you cannot use the name you enter.
2. The type of the variable. Select the smallest size you can here to minimise the overall size of a cell and reduce the simulation's storage requirements.
3. Exit the window, losing any changes you made.
4. Close the window, saving changes.

5.1.6 Editing a Colouring Rule

The colouring window allows you to edit or create a rule to specify how cells should be coloured. Most variables can either be coloured if they match a single variable (in which case, you only need to choose one colour), or when they fall within a range (in which case you will need to choose an end colour as well). Rules for boolean variables cannot be defined as a range, and rules for floating-point variables cannot be a single value.

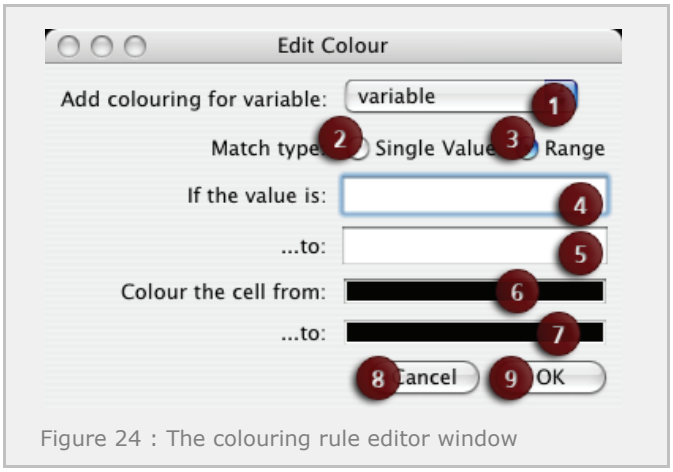

Figure 24 : The colouring rule editor window

1. Select the variable the rule should apply to.
2. If you want the rule to apply only for a single value, choose this option. This option is disabled when the variable is floating-point, since it would rarely, if ever, match one.
3. If you want a range of values for the rule, check this box. This is unavailable for booleans, since they can only take two values and you would never want to define a range for one.
4. This box shows the lower bound value to match. If you are not defining a range, this will also be the upper bound.
5. This box defines the upper bound of the rule. It is only applicable if you are defining a range.
6. This box specifies the lower bound colour.
7. This defines the upper bound colour. Again, it is only applicable if a range is being defined.
8. Exit the window, losing any changes you made.
9. Close the window, saving changes.

5.2 Starting a Simulation

When you have finished editing your simulation, you can start running it. To do this, choose 'Run Simulation...' from the Tools menu on the Editor window. If no node is connected, you will be presented with the option of starting one locally. This is not recommended for large simulations, but can be useful for quickly testing a small one. At least one node must be connected before a simulation can start.

A window will then appear allowing you to confirm your decision, and to choose where to save the replay file to.

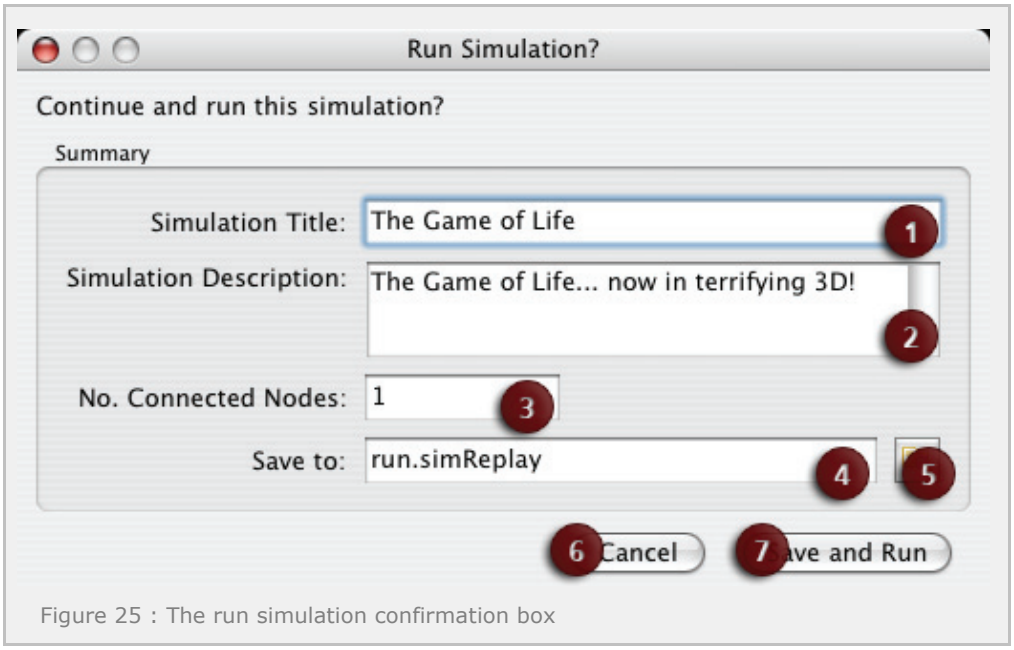

Figure 25 : The run simulation confirmation box

1. The name of the simulation you are about to run.
2. The description of the simulation.
3. The number of nodes you are about to run this simulation on. If this is wrong, press cancel, connect or disconnect nodes, and try again.
4. The path and filename you will be saving the replay file to. Change this using the button next to it.
5. Click this to change the replay file location. Type a name for the file in the window that appears, and choose a location to save it to.
6. Exit the window without running the simulation.
7. Save the simulation, and start running.

When you choose to save and run, the script you entered will be sent to all connected nodes, which will then attempt to compile it. If any problems are encountered, you will be told about them. Otherwise, the simulation should begin and the simulation display window should appear.

5.3 Replaying an Existing simulation

To replay a simulation, you must have the corresponding simulation open in the editor window (Cellulate uses this to determine the variables and colours to display). You can then choose 'Replay Previous Simulation...' from the File menu on the editor window, and choose the .simReplay file in the file dialog that appears.

The simulation display window should appear, allowing you to move backwards and forwards through the simulation and view cell variable values. When you are done, you can close the window to return to the editor.

5.4 Changing Application Preferences

Various application settings can be changed by choosing 'Application Settings...' from the Tools menu (or 'Preferences...' from the Application menu under OS X). Most changes will take effect when the program is restarted.

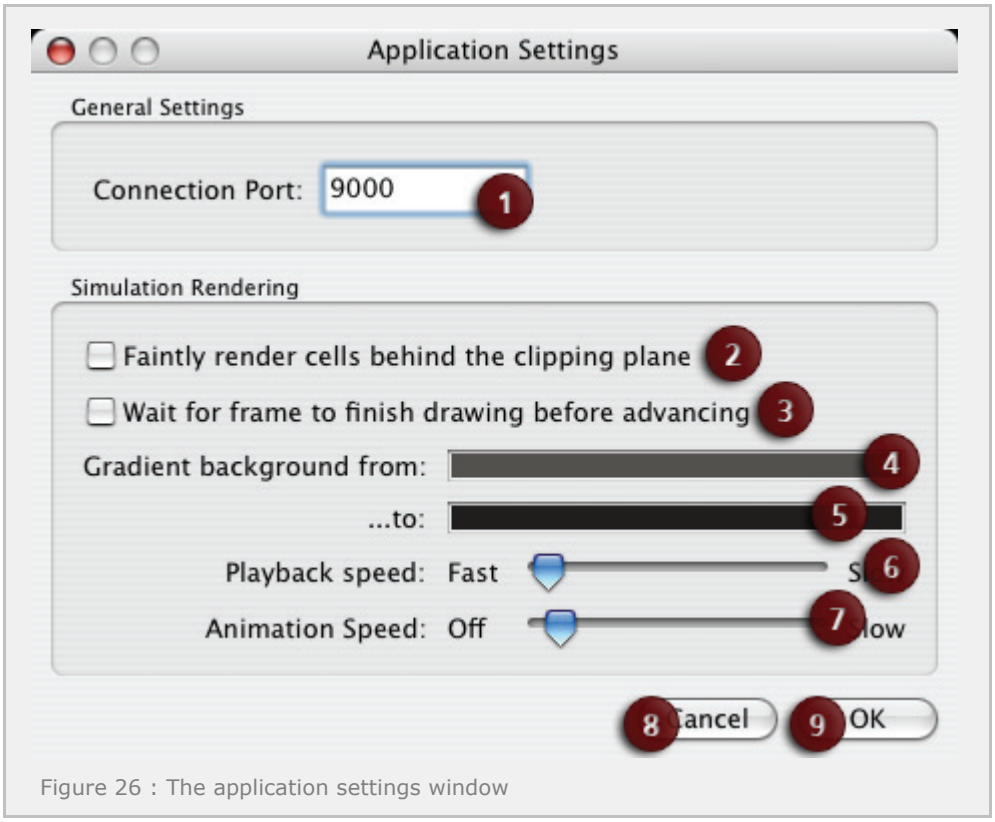

Figure 26 : The application settings window

1. This is the port that the Cellulate application listens for connections on. Changing this is useful if the port is already in use, or blocked by a firewall. This change will take effect when you restart the program. You will also need to pass the new port to Cellunet as a command-line parameter.
2. When using the clipping plane to view cells in the middle of a cluster, Cellulate will, by default, faintly render the clipped cells to give you an idea of the overall shape of the group. This option allows you to turn this off (i.e. hide the cells completely) which can be used to improve display performance.
3. If you are playing the simulation reasonably quickly, the frame may advance before the previous frame has finished rendering in its entirety. This option forces Cellulate to wait until the rendering completes before changing frame.
4. This is the top colour of the background gradient.
5. This is the bottom background gradient colour.
6. This controls playback speed: the delay before frames will advance when the play button is pressed on the display window.
7. This will affect transition animation speed, such as zooming in and out, and orbiting. Animations can be turned off completely by moving this slider all the way to the left.
8. Exit the window, losing any changes you made.
9. Close the window, saving changes.

5.5 Viewing a Simulation

When you are viewing a simulation that is in the process of being rendered, or replaying a simulation that has been recorded, you will be presented with a window like the one shown below.

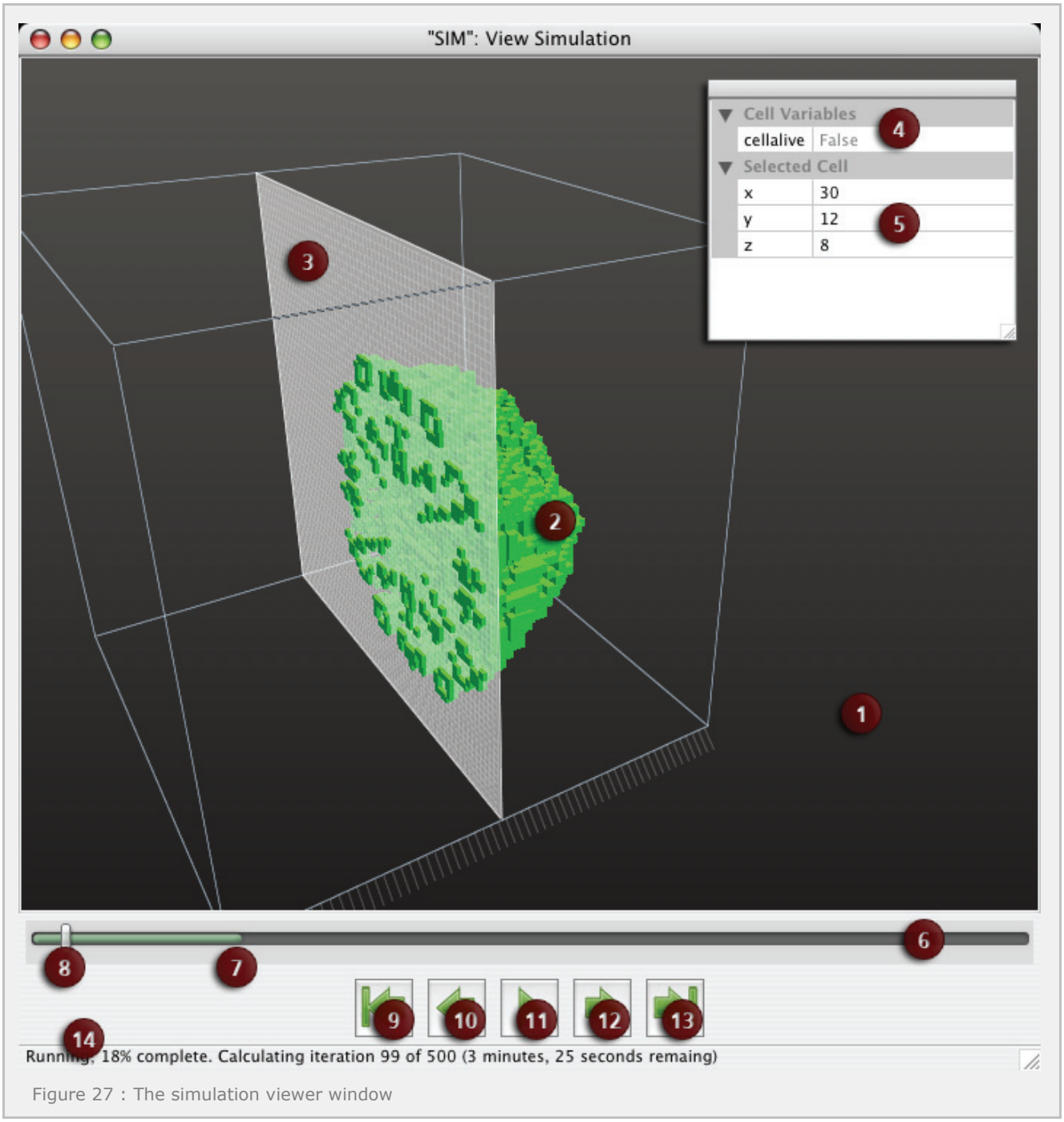

Figure 27 : The simulation viewer window

1. This is the renderer. It shows a 3D representation of the current state of the model. It can be rotated (using the left mouse button), panned (using the right mouse button) and zoomed in and out (using the scroll wheel). See the next section for more detailed controls.
2. This is the mass of cells. The screenshot in this example is from the 3D Game of Life, and the cells have been generated based on the rules and a small initial state. The values of a cell's variables can be viewed by double-clicking it; information about the selected cell is shown in the floating inspector window.
3. This is the clipping plane. It can be difficult, if not impossible, to view the middle of a group of cells if (as in the Game of Life), they form a structure like a ball or cube. The clipping plane allows you to hide all cells on one side of it, letting you view cells in the middle. It can be toggled on and off using the 'h' key, moved left and right using the '+' and '-' keys (or '[' and ']', or alt+scroll wheel),

the orientation can be changed by double-right-clicking, and the side that is hidden flipped using the 'j' key. These commands are summarised in the next section.
4. This shows variables of the currently selected cells, together with their values. They cannot be edited when viewing the simulation.
5. This shows the x, y and z co-ordinates of the currently selected cell.
6. This bar shows both how far through calculating the simulation Cellulate is, and the position of the frame you are viewing in relation to the entire simulation.
7. The green line inside the bar represents how far through calculating the simulation Cellulate is.
8. The handle shows you what frame you are looking at in relation to the entire simulation, and allows you to move around the simulation. Either drag it to where you want to look at, or click anywhere on the green bar to jump straight to that point.
9. This button moves back to the first iteration.
10. This button moves back one iteration.
11. This button toggles the simulation between playing and paused. You can adjust the delay before Cellulate advances to the next frame using the slider in the application settings window.
12. This button moves forward one iteration.
13. This button moves to the last available iteration (i.e. to the end of the green line).
14. This bar shows the current status of Cellulate. If Cellulate is calculating, it will give you a rough approximation of the amount of time left until the simulation has finished.

This window also lets you pause or stop the simulation calculation (if it is still happening). You can do this by choosing 'Pause Simulation...' or 'Stop Simulation' from the Simulation menu. When it is paused, it can be resumed in a similar way.

If the window is closed, Cellulate saves the data generated so far in the appropriate simReplay file. This can then be viewed at a later date.

5.6 Controlling the Renderer

The renderer is used in two main places: to define initial states, and to view the results of rendering. In both cases, it works in mostly the same way.

5.6.1 Controlling the Camera

Arguably the most important aspect of using the renderer is camera control. This allows you to rotate to see previously hidden cells, zoom in to see fine detail, and zoom out to get a broader view.

- To zoom in and out, use the mouse scroll wheel, or the 'Page Up'/'Page Down' Keys.
- To rotate the camera, click and drag with the left mouse button.
- To move the simulation to the left, right, up and down, click the right mouse button and drag. You can also use the 'a', 'd', 'r' and 'f' keys.
- To make the camera automatically orbit the simulation, press the 'o' key. To stop it, move the camera or press 'o' again.
- Holding down shift while zooming will allow you to zoom more quickly.

5.6.2 When Defining States

When you are editing the initial states, the renderer gains a few extra commands to make this task easier. In particular, it includes keys for manipulating the select plane, used to select invisible cells (cells with no colour rules defined that would otherwise be unselectable). When selecting cells by double-clicking, if there is no cell at the point you clicked, Cellulate will instead select the empty cell on the select plane at that point.

- To move the select plane, use the '+'/'-' or '['/']' keys, or press 'Alt' while using the scroll wheel.
- To change the orientation, double-right-click.

5.6.3 When Viewing Simulations

When viewing a simulation, the renderer includes an extra clipping plane, hidden by default. This plane can be used to hide all the cells behind or in front of it. This is useful for viewing a cross-section of a simulation; it lets you see cells in the inside of a group you would otherwise not be able to view.

- To toggle the clipping plane on and off, press the 'h' key.
- To move the clipping plane up and down, press the '+'/'-' or '['/']' keys, or press 'Alt' while using the scroll wheel.
- To change the orientation, double-right-click.

- To change the side of the clipping plane that is visible, press the 'j' key.

5.7 The Node (Cellunet)

The node is used by Cellulate to run the simulation. There needs to be at least one connected in order to run a simulation, but Cellulate will try and use as many as are available.

Once a node is connected to the Cellulate application, it will remain connected until you either terminate it (Ctrl-C, or through the node management window), or the Cellulate application exits. It will participate in whatever simulations are run during this time.

The node supports various configuration options that can be passed as command-line parameters. The most commonly-used are:

- `-a [IP address]`: Specifies the IP address to try to connect to. Defaults to 127.0.0.1 (the current machine).
- `-p [Port]`: The port number to try and connect on. This should match the one specified in the Cellulate application, and defaults to 9000.
- `-r`: Keep retrying the connection until it succeeds. Useful if the Cellulate application isn't running yet. This can be stopped using 'Ctrl-C'.
- `-y [0|1|2]`: The priority to run the node at. 0 is below average (only idle CPU cycles) and is default, 1 is normal priority, and 2 will run it at a higher priority. Note that this option may require administrative privileges.
- `-h`: Shows help in the form of a summary of all the command-line options, then exits.

5.7.1 Altering the Build Configuration

You may need to edit the command line the node uses to compile scripts. The configure script will guess the correct command line to use, together with appropriate paths, and most of the time it should do this correctly. If the guessed values are inappropriate for your system, you can edit it manually. The command line it uses to compile can be found in make.cfg, which is in either $PREFIX/share/cellulate/script, or the script directory in the same folder as the cellunet executable. Note that the location of the script files should not be changed unless you know what you are doing; the node expects to find the script and compiled library in the locations initially set.

5.8 Managing Connected Nodes

It is possible to view a list of nodes that are connected, as well as disconnect unwanted ones from the Cellulate application. To do this, choose 'Show Connected Nodes...' from the Tools menu of the editor.

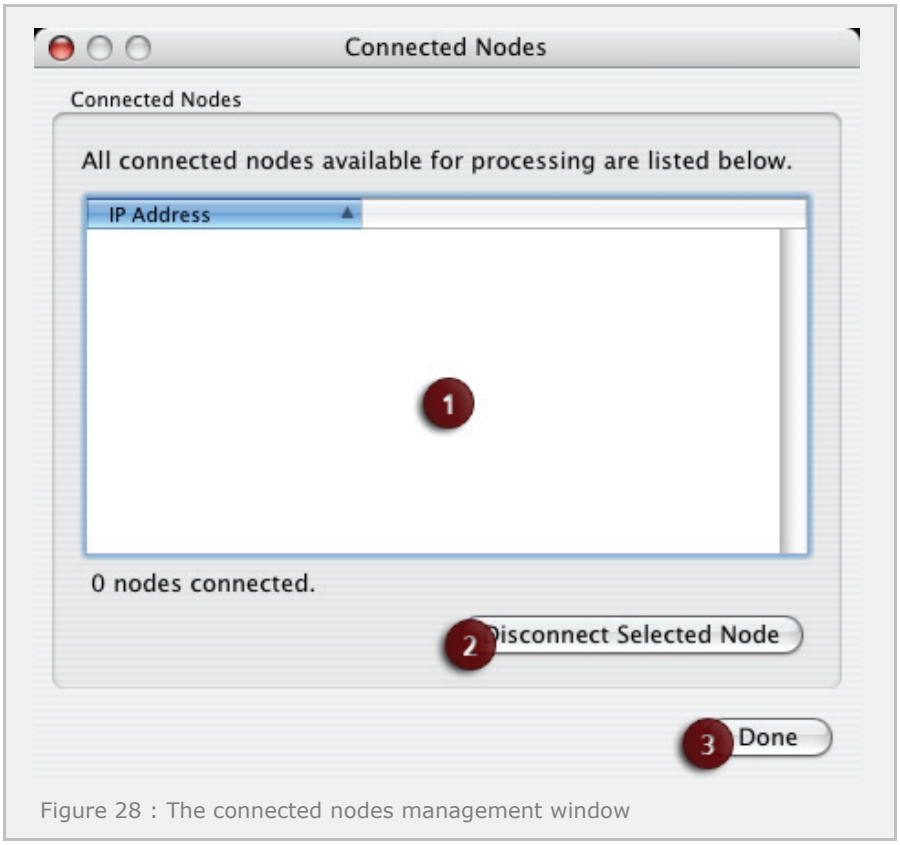

Figure 28 : The connected nodes management window

1. This shows a list of all connected nodes' IP addresses.
2. To disconnect a node, select it in the list above, and press this 'Disconnect' button. The node will be unceremoniously dumped from the exciting distributed network you've formed.
3. When you're done disconnecting things, you can close the window with this button.

6 Scripts

6.1 What is a Script?

The script is the method by which the user defines the rules of the simulation. It is written from the perspective of a single cell which is being updated. The aim is that by the end of the script you will have defined each different case for how you want a cell to be set.

It is written as a fragment of programming code. The script uses the syntax of the language C++ (since it is basically a subset of C++). It is therefore very similar to C and other high-level languages such as Java and Pascal, so users with any basic programming experience of such languages should find script-writing very straightforward.

6.2 Using Script Functions

A large amount of Cellular Automata (CA) can be created using these basic functions*:

- `Set()`: Sets the state of a variable in the current cell
- `Get()`: Gets the state of a variable in the current cell
- `GetNeighbour()`: Gets the state of a variable in a neighbouring cell

*Note: The parameters to these functions are omitted for clarity

For instance the well-known Conway's Game of Life CA can be written using just these three methods. For ease of use, a number of other methods have been provided to help the user write scripts faster and more efficiently.

A basic example of such a method is 'NumNeighbours()' which can check all neighbouring cells within the neighbourhood, see if they match some given value, and return the number of them that do. The need to do this is common in 'totalistic' cellular automata such as the Game of Life.

A complete list of available functions (for the current version) is given in the Functions Reference section at the end of the user manual.

6.3 Writing a Script - An Example

The following is an example rule script, using such methods to create the simulation rules. The example also shows the use of standard C++ syntax. (Note the 'if-else statements' and variable assignments, as in C++.)

```
/* Conway's Game of Life */

/* Constants for the life rules */
int r1 = 3;
int r2 = 4;
int r3 = 3;
int r4 = 2;

/* Count the number of neighbouring cells that are 'alive' */
int numNeighbours = NumNeighbours("cellalive",true);

/* Based on numNeighbours work out new cell state */

/* If not too many or too few, set this cell to alive */
if ( numNeighbours >= r1 && numNeighbours <= r2 ) {
        Set("cellalive",true);

/* If too many (overcrowding) or too few (loneliness) this cell dies */
} else if ( numNeighbours > r3 || numNeighbours < r4 ) {
        Set("cellalive",false);

/* Otherwise the state is maintained from the previous iteration */
```

```
} else {
	CopyCell();
}
```

As the comments say this script will run a particular case of Conway's Game of Life (GoL) in 3D. Now we will step through this script and see what is going on. If you wish to copy this example you will need to first set up a cell variable called "cellalive" of type boolean. Let us then begin:

```
/* Constants for the life rules */
int r1 = 3;
int r2 = 4;
int r3 = 3;
int r4 = 2;
```

Here, we have declared some integer variables and assigned values to them. These values will dictate how many 'live' neighbouring cells are required for the cell to be 'born', 'survive' or 'die'. This step is not essential, the values could be put into the if statements later, but this means we can easily come back to the simulation and change these values to see some different behaviour. (Note that variations of GoL are sometimes formally distinguished between by using these numbers - this is gol(3432)_3d).

```
/* Count the number of neighbouring cells that are 'alive' */
int numNeighbours = NumNeighbours("cellalive",true);
```

Now, we are using a script function that is provided for us called NumNeighbours(). This function checks every single neighbour (in 3D there are 26) and counts how many of these have the value we specify for the variable we specify. In this case (NumNeighbours("cellalive",true)) we are checking how many have their variable called "cellalive" set to true (i.e we want to know how many cells around us are 'alive'). We assign the result of the function call to a variable so we only need to call it once in the script.

```
/* Based on numNeighbours work out new cell state */

/* If not too many or too few, set this cell to alive */
if ( numNeighbours >= r1 && numNeighbours <= r2 ) {
	Set("cellalive",true);
```

Now we are going to work out how to set the current cell based on this value 'numNeighbours'. We are comparing it to the values of the variable r1 - r4 we assigned before.

Firstly, we want the cell to be set to 'alive', by setting our "cellalive" variable to true (Set("cellalive",true)) if the number of neighbours is between our values for r1 and r2 inclusive. We are saying a cell is 'born' if it has this number of neighbours.

```
/* If too many (overcrowding) or too few (loneliness) this cell dies */
} else if ( numNeighbours > r3 || numNeighbours < r4 ) {
	Set("cellalive",false);
```

Now, we want to set the cell to 'die', by setting our "cellalive" variable to false (Set("cellalive",false)) if the number of neighbours is greater than the value of r3 (in GoL we say the cell is 'overcrowded') or less than the value of r4 (the cell is 'lonely').

```
/* Otherwise the state is maintained from the previous iteration */
} else {
        CopyCell();
}
```

Finally, we want to set the cell to 'stay as it is' if none of these conditions are true, so we put this within the else statement and we use the method CopyCell(). CopyCell() basically means copy the entire cell's state exactly as it was in the previous iteration. If our cell has lots of variables, it would preserve all of these.

Now your script is ready so set some initial states, make sure you have set up the variable "cellalive" and some appearance properties and you are ready to run your simulation!

If you want to try out some of the other script functions, refer to the Function Reference section. Also looking at some of the other simulations in the examples directory will help you to see how they are used.

6.4 Advanced Use - Maths Functions

The user can access a number of standard C maths functions from the 'cmath' libary.

Warning: This is not particularly advisable for performance reasons since it can slow down simulations considerably. But users are free to do this if they wish. For further information please refer to a C Libary reference (http://www.cplusplus.com/reference/clibrary/cmath/). Available methods include:

- `pow()`
- `sqrt()`
- `rand()`
- `sin()`
- `rand()`

Note that a random number function 'Random' is provided (see function guide), but if users wish to use the cmath function rand(), they may. rand() is already seeded (outside of the main loop) sufficiently well that properly random results should occur with each simulation run. There is therefore no need to use the seeding function srand() as well.

6.5 Non-recommended Use

It is advisable not to use certain C++ methods and keywords such as:

- The `new` or `delete` keywords. (If you do, please be very careful!).
- Memory allocation methods such as `malloc()`.
- Use of sub-functions - this may work but is not recommended.
- The 'class' keyword to declare classes within scripts - This will not work.

6.6 Error Feedback

If your script contains an error when you attempt to run the simulation, the error will be reported in a dialog box. You can also check the script before attempting to run the simulation by pressing the 'Check Syntax' button.

6.7 FAQ - Common Problems

1. I do not update a cell and it disappears, why?

   If you do not specify what a cell should do, each of the cell's variables will return to their default values (0 for integers, false for boolean, 0.0 for floating-point). If you want a cell to 'stay the same' as in the previous iteration, use the function CopyCell() (see function reference section).

2. I do not understand the error feedback.

   Note that the feedback you are given is directly from the compiler, so error messages may be confusing if you are not used to C++ compiler output. The line number you are directed to for the first error is normally enough of an indicator to spot the error. If you still cannot locate the error, you can try Googling the output error (minus the line number etc) to find what it means.

3. I call Set() and then Get() which doesn't return the value I just Set!

   Get() returns the state of a cell variable from the previous iteration. Set() defines what state the cell variable 'will be set to' in the current iteration. Therefore you cannot Get a value you have Set previously in the script.

7 Troubleshooting

7.1 The node can't connect to the client

This is most likely caused by the port the node is trying to connect to being blocked.

- Firstly, check the Cellulate application is running. If it isn't, start it.
- Failing that, check the port isn't protected by a firewall.
- Check the port the application is listening on (as specified in the settings window) is the same as that the node is trying to connect to (9000 by default, but can be changed with the '-p' option on the command line, see Cellunet documentation for further information).
- Check that the node and Cellulate applications are the same version. Try re-compiling or re-installing if in doubt.

7.2 2D Cellular Automata don't work properly

- This is most likely because you forgot to turn off wrapping for the dimension you specified as being only one cell wide. If you haven't done this, every cell will have itself as a neighbour, and the simulation may produce unexpected results. You can toggle this in the 'Simulation Properties' tab of the editor window.

7.3 Cells are not coloured properly

The most likely reason for this is that the colouring rules are not being applied to the cell as you had intended.

- If a cell does not have any rules that apply to it, it will be invisible. Ensure there is a rule that matches the desired cell variable.
- If you specify a range, you will need to specify one colour that corresponds to the start of the range, and one that corresponds to the end of the range. Cellulate will calculate the colour for the cell depending on where it lies in that range; it is unlikely to be exactly either of the two colours you specified.
- If several rules apply to one cell, the first one that matches is the one that has an effect. Ensure there is only one colouring rule that matches the problematic cell.

7.4 I can't run my simulation

- Ensure at least one node is connected. Cellulate does not run the simulation itself; instead, it farms the processing out to every connected node. The application should offer to start one if you try running with no nodes available, but it is recommended you manuallly start a node to be sure.
- Ensure the script you entered is correct. You should get a dialog box with the error if the node was unable to compile your script due to a syntax error; check your code and correct any problems this window highlights. You may also want to refer to the scripting portion of the manual for more information.
- The node may not be able to find the C++ compiler. Check the make.cfg file and ensure the correct compiler and options are specified; see section 5.7.1 for more information.

7.5 Cells do not seem to have the right values

- A cell must have a value set each iteration; values do not 'carry over' from previous iterations. If you want to make a copy of the previous cell state without changing anything, you will need to use the CopyCell() method; see section x.x.x for more information.
- It is possible there is a logic error in your script; double-check it to be sure everything works as it should.

7.6 I get permission errors when trying to run a script

- This may be because you do not have permission to execute files on the partition Cellulate is compiling to (/tmp on Linux). You will need to enable execution for the script dynamic library that the node will try to create and run.

7.7 The simulation replay does not relate to the simulation

- This problem may be caused by loading the wrong .sim file. You need to open the original simulation file before starting the replay.
- Cellulate may have run out of disk space when saving the file. This will result in a corrupt replay file.

7.8 All the cells are gray

- The renderer may be updating the cells. They will be replaced by more up-to-date cells when the render gets around to it.
- If they are transparent, then the clipping plane is probably turned on. This is used to hide cells behind a movable plane, allowing you to see what's in a ball of cells. To turn it off, press 'h'.

7.9 Help! I'm lost and I don't know what I'm seeing!

- In Cellulate, press 'c'. This will reset the camera to the original position.

8 Function Reference

The following functions are available to the user:

8.1 Basic Functions

`Set(name, value)`
Set value of named cell variable to value specified

`value Get(name)`
Get value of named cell variable

`CopyCell()`
Copy cell state from previous iteration (cell 'stays the same' as previous iteration)

`SetToDefault()`
Set current cell's variables to default/empty values - optimises processing/rendering

8.2 Neighbours

Note that where required, 'relX, relY, relZ' are the relative coordinates used to specify a neighbouring cell (e.g. 1,0,0 is the neighbouring cell in the positive x-direction).

`value GetNeighbour(name, relX, relY, relZ)`
Get the value of a named variable of a specified neighbour.

`CopyNeighbour(relX, relY, relZ)`
Copy cell state of a neighbour to own state - i.e. the cell 'becomes its neighbour'

`value SumNeighbours(name)`
Sum all values of the the named variable state for every neighbour.

`int NumNeighbours(name, value)`
Number of neighbouring cells within the neighbourhood whose value of the named variable is EQUAL TO the value specified

`int NumNeighboursGT(name, value)`
Number of neighbouring cells whose value of the named variable is GREATER THAN the value specified

`int NumNeighboursLT(name, value)`
Number of neighbouring cells whose value of the named variable is LESS THAN the value specified

`int NumNeighboursGTE(name, value)`
Number of neighbouring cells whose value of the named variable is GREATER THAN OR EQUAL TO the value specified

`int NumNeighboursLTE(name, value)`
`NumNeighboursLTE` - Number of neighbouring cells whose value of the named variable is LESS THAN OR EQUAL TO the value specified

8.2.1 Other

`Increase(name, value)`
`Decrease(name, value)`
Assuming the named variable is of numerical type, increase or decrease by the value specified.

`int Random(highest,lowest)`
Returns a random number (integer) between highest and lowest (integer) values, inclusive.