

Virtual Worlds User Guide

Introduction :

The Virtual Worlds application as it is implemented as of March 2024 is not intended to be a fully functional production ready application at this time. It is, at this time, intended to be a functioning application to act as a base for further development and demonstrate the idea of using a different form of 3D voxel geometry and coordinate system that departs from the conventional cubic voxels that are used widely in many areas of computer computation and simulation.

In 3D space, for any one cubic voxel, the distance between its centre and that of every neighbour around it is not the same unless a neighbour is considered to be sharing a common face and not a common vertex point. However, a 3D space that does have a structure where the centre of any location in that space is of equal distance to all its neighbours is that of a hexagonal close packed spheres of equal radius, where the geometric shape of a Trapezo-rhombic dodecahedron can be arranged in such a manner that it forms a space filling geometry that can be utilised as a voxel.

It is of the view that a HCP sphere structure is superior than a cubic structure for use in procedural generation of geometric shapes, to simulate 3D environments and for cellular automata. Virtual Worlds had the initial ambition to be an application that generates procedural landscapes of a fractal nature utilising voxels and point clouds, hence the name. That ambition may not come to pass, but as a more practical purpose, and wanting to give and share something that could not be found elsewhere in my searches, is a demonstration of how a HCP Trapezo-rhombic dodecahedral voxel model can be implemented.

Section 01.0

Application Overview

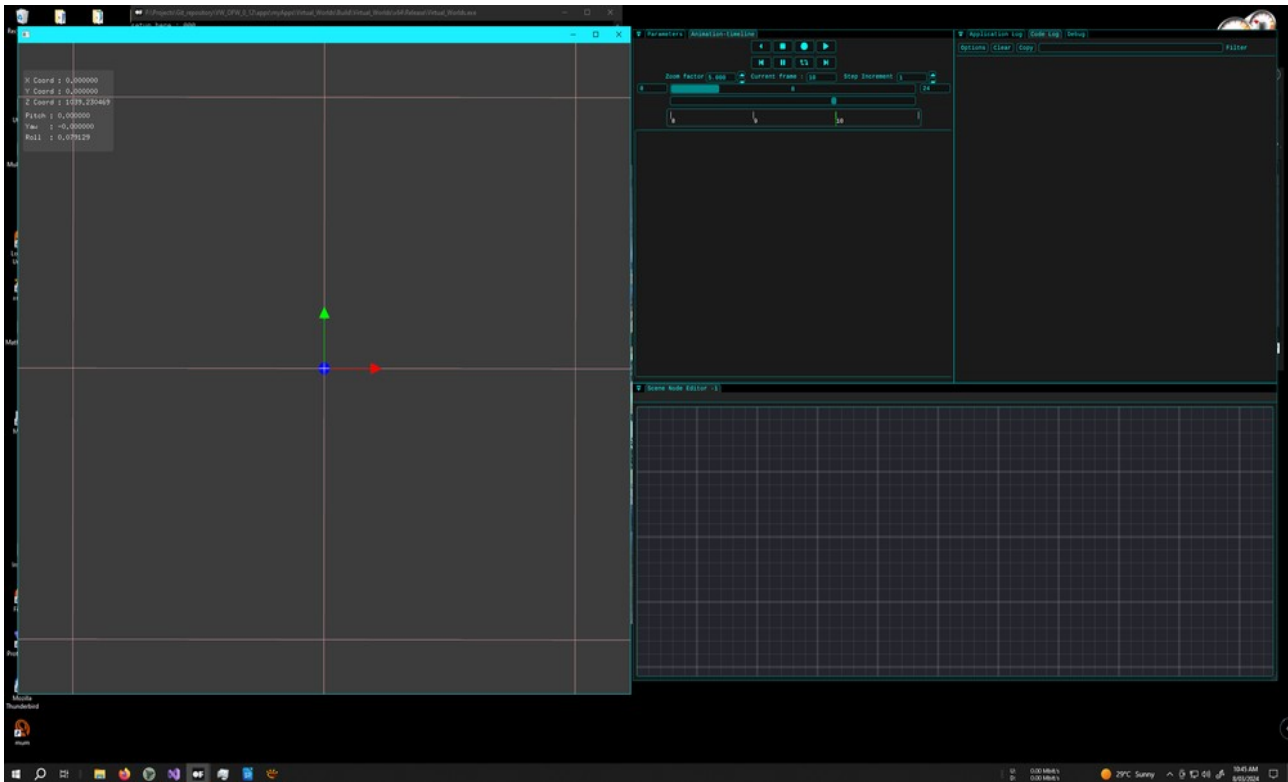


Fig 01.0.1

The Virtual Worlds application executable and all required files is located in the File Directory Build/Virtual_Worlds/x64/Release of the Virtual_Worlds project. All these files need to be copied to any other directory path if the user wishes to run it elsewhere.

By starting up the application, the application main user interface similar to the image given in **Fig 01.0.1** should appear.

On the left is the view port in which the 3D objects of the scene will be rendered.

On the right is the main application user interface. This user interface panels are dockable and may appear different in the arrangement of the panels according to user preference. The ImGui system that forms the basis of the GUI may interfere with the communication of widgets or user interaction between them. If any problems encountered, just redock the panels to overcome.

The user interface composes of the following panels

Node Editor panel : Is where all of the entity objects in the scene and functions or operations on them are created, deleted, imported, or saved, and are represented as a graphical node or rectangle box. This node editor is much similar many other node editors in its function to selects, display, link, and manage nodes.

Parameters Panel : Is where the parameter and variables data is displayed on the current selected scene entity object that is designated in the node editor panel, and settings for the view port display, camera and scene lights. The user, through the widgets displayed in this panel can make changes to the parameters and variables displayed in this panel , and/or perform tasks or actions that generate or modify graphical or other data to be displayed in the view port.

View port : Is where the view of the objects that are made visible in the scene are displayed, and where the user can interact with the viewer camera to move about the scene. A central cross hairs, camera information and navigational overlay are displayed with an initial x-y axis reference grid.

Timeline Panel : Is a control panel where the user can perform animations of the objects in the scene based upon the timeline tracks for specific scene and node elements that have been assigned to them.

Application Logger : Is where any messages that the application issues to inform the user of any application status are displayed.

Shader Logger : Is where any opengl shader compilation error messages that are generated while compiling shader code are displayed.

Section 01.1 The Node Editor

The node editor is generally the first point on startup of where the user will go to generate voxel point clouds and to perform tasks on any generated voxel cloud data. It is where all of the entity objects in the scene and functions or operations on them are created, deleted, imported, or saved, and where the management of scene objects is performed.

The Node editor is ImGui based in its operation and thus faces many of the limitations and restrictions that ImGui invokes.

To get started, clicking the right mouse button and releasing it will display a floating menu. Within this floating menu should be options to create a 3D data object, a group node and some other options.

Like most node editors, if nodes and their associated ports and links are displayed within the node editor, the user can left click on a node to move it, or a node port to create a link between nodes. As with most node editors, inputs are on the left side of the node, and outputs on the right.

By left clicking on a node to select it, the parameters panel will display a parameter widget with all the settings, data and functionality appropriate to the node selected. The user can then go to the parameter panel and perform any changes and/or tasks presented. Also after selecting a node, if the user right clicks on the node and releases it while it is selected, a floating menu specific to that node will appear for the user to interact with if that node has associated with it a menu selection from which to interact with.

If the user moves the mouse cursor over a link and left clicks it to select it, and then presses the right mouse button and releases it, a floating menu relevant to that link appears for the user to take action on.

If the user moves the mouse cursor over a node pin (port) and left clicks it select it, and then presses the right mouse button and releases it, a floating menu relevant to that pin appears for the user to take action on.

A special node called a group node can be created to open a child node editor of the current node editor. This group node editor is used to manage a scene of large nodes into a more manageable form. This group node node editor will behave just like the application root node editor in every way, except that it will have two special nodes that are used to import and export link data to and from this group node to its parent node editor. This is so a group node can act as if it is a single node.

If the user creates a group node, after selecting it, if the user makes a right double click, the node editor for that group node will be made visible for the user to interact with. Other wise a single right click will bring up a floating menu as with any node to perform actions on that group node, such as delete.

To deselect every thing, move the mouse cursor to a location within the node editor where no node editor entity lies beneath it and click the left mouse button.

All node editors have the capability to zoom in and out to a given limit and can be scrolled by clicking the scroll or mouse 3 button and moving the mouse.

Important :

Note : Due to an ImGui issue unable to be resolved as of the time of this writing, it seems that if one where to zoom in or out when there are one or more group node editors lying on top of each other, and the mouse position is common to one or more of these editors, all editors will zoom in and out at the same time. It seems this is an artefact of the way ImGui works and needs to be resolved.

Another faator is that when a node is selected and the user moves the mouse cursor into another node editor, any display in the parameter panel will disappear and not be displayed until the mouse cursor re-enters the selected nodes node editor. To edit parameter data for a selected node in a particular node editor, the user needs to move the mouse cursor directly from the node editor that the selected node is within to the parameter panel without entering any other node editor's exposed region.

02.0 Generating a HCP voxel object

HCP voxel objects are generated using a text file with an OpenGL compute shader program. So before doing anything else, the user needs to create or have an OpenGL compute shader program source code as a text file to use. Virtual Worlds already has some examples available for use in the examples directory of where the Virtual_Worlds executable is located. To find out how to create a Virtual Worlds compute shader code, see the documentation in section 4 on how to do this in the section writing a compute shader program.

To generate a HCP voxel object, move the mouse cursor to within the node editor described in section 01.1 and right mouse click over an empty region where one wishes to place a node to represent a HCP voxel object and release it.

A floating menu should appear with the option **Main Menu** displayed. Move the mouse cursor over this option to display a sub menu. One of these options should be **Create Node** Move the mouse cursor over this menu option and another sub menu should appear with a menu option **HCP Voxel** Move the mouse cursor over this menu option and yet another sub menu will appear with one option being **Create HCP Voxel node.**(Fig 02.0.1). Left click with the mouse on this option, and where the mouse cursor was placed in the node editor should be a HCP Voxel node similar to Fig 02.0.2

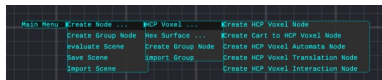


Fig 02.0.1



Fig 02.0.2

Move the mouse over this newly created node, select it by clicking the left mouse button while the mouse cursor lies over it. Depending on the configuration of the GUI one should see a panel on its own or grouped with others as a tab with the label **Parameters**. Display the parameters panel or any tab with the label **Parameters** to display it. Select the tab within the parameters panel that has the label **Object**. Within this object tab widget, more tab widgets should appear. Select the tab with the label **Generation**. A widget as in Fig 02.0.3 should be displayed. The parameters within this widget required to generate the HCP voxel data are from top to bottom

Expression button : to select the virtual worlds OpenGL compute shader program file that generates the HCP voxel data.

Generated voxel volume limits : Definition of the x-y-z volume limits of that the HCP voxel volume generation is to be confined to.

Compute invocation is a number that is relevant to the OpenGL compute shader process where the number indicates the number of threads running in parallel to be used in the calculation. Generally from documentation, the higher numbers are used if have a large number of calculations to be performed.

Execute button : Generate the HCP voxel data. Only press when everything is ready.

Min/Max surface values : These are the minimum and maximum permissible values that the voxels can possess to be designated as a valid voxel to be stored in computer memory and displayed on screen. At the time of this documentation, this should be left as the defaults as any other value may not work. Something for a future update.

Resolution step : Basically, this is the size of each HCP voxel.

Threshold : Generation of HCP voxels is of the form voxel value = $F(x,y,z)$ and the threshold is the minimum absolute value that this voxel value can have to be considered as a valid voxel to be stored in computer memory.

Display as points : Change the display of the generated data from points to voxels. This display is determined by the shader program and parameters that are defined in the the shader tab widget which is discussed below.

Voxel Scale : Widget to scale the HCP widget size when it is selected to not to display the data as points. Note : this is shader code dependent.

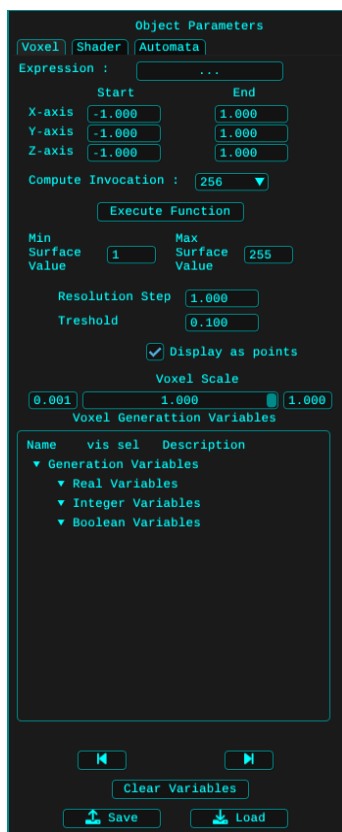


Fig 02.0.3

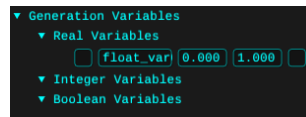


Fig 02.0.4

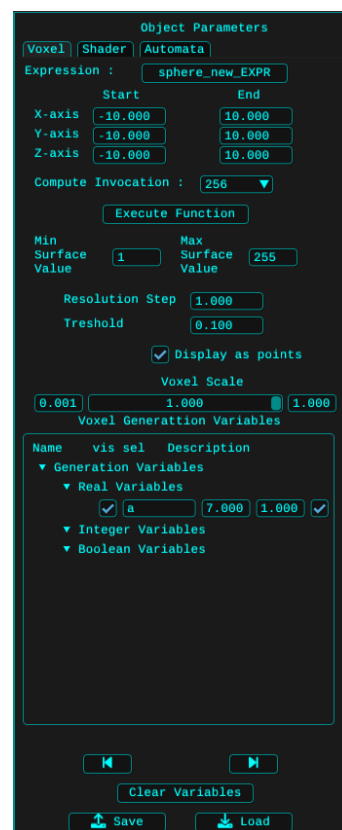


Fig 02.0.4

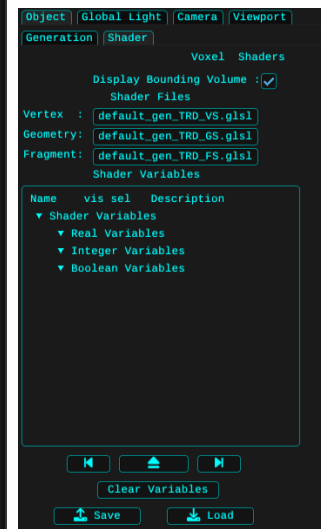


Fig 02.0.5

Voxel Generation Variables : A set of widgets that enable the user to manage and view the set of variables that are used to generate the volume of voxels within the volume as defined in the parameters above this widget. Any variables defined here must have an equivalent uniform shader variable that is defined in the compute shader program code that is used to generate the voxel volume or there will be a shader compilation error, and no voxel volume generated.

Left and Right control buttons : These Buttons perform a decrement and increment of one or more of the variables defined in the voxel generations widget to be changed by a given step value, and then regenerate the voxel volume and display it on screen.

Clear variables button : Clears all the voxel generation variables

Save : Opens a dialog window to save the voxel parameters defined in this widget to a formatted text file of suffix .vgp (Voxel generation parameters)

Load : Opens a dialog window for the user to select a voxel generation parameters (.vgp) formatted text file to be imported into the application, and the values displayed in this widget ready to be used.

The details of creating a voxel generating compute shader program to be selected for use here is in the section 4 Create Voxel Compute Shader Program. For the present as this is an overview of generating a voxel volume and displaying it, it is only required for the user to select an existing example.

Step vg01 : Select the Expression Button at the top of this widget to bring up the file load dialogue.

Step vg02 : Navigate to the directory Examples/Generation/HCP_Voxels/ that the Virtual Worlds application resides in and select the file name sphere_new_EXPR.txt.

The button name should now be changed to sphere_new_EXPR. If not repeat steps vg01 and vg02 above.

Step vg03 : To define the volume that the HCP voxels will be generated over, change all of the start x,y,z, values to -10, and the end values to 10

Step vg04 : Go to the voxel generation variables section and with the mouse cursor, navigate to and hover over the Real Variables tree node widget to highlight it. Press the right mouse button to bring up the only available floating menu option “add variable” and select it to create a new real (ie floating) variable.

The variable widget that is displayed (Fig 02.0.4) is the same for all float and integer shader variables. From left to right.

Select and left mouse click the leftmost checkbox. This is to enable this variable to be used in the voxel generation compute shader program.

Select the text input widget and enter a to change the text value within this text box. This is a variable name that is defined and used in the compute shader program to generate a voxel volume cloud point data.

Select the floating number input widget and type in the value 7.0 .This is the initial or set value that this

variable is defined to be assigned to and used in the compute shader program to generate a voxel volume cloud point data.

Select the next floating point input widget and enter the value 1.0. This is the incremental step value to change the variable by as a step action if the increment or decrement button is selected.

Select and set the checkbox at the far right. This checkbox widget is to indicate to the application if the step variable is to be used or not when performing incremental or decremental step operations.

The voxel generation panel should now look like that of Fig 02.0.4.

The generation of a simple HCP voxel sphere can now be performed.

Step vg05 : Press the button labelled “Execute Function”

This will generate a HCP voxel point cloud of data. However there may not be any points displayed in the world scene view port. If not, perform the following.

Step vg06 : Select the tab with the label “Shader” to display the voxel object shader parameter widget. Fig 02.0.05

Step vg07 : Select the large middle button with the up arrow to update the and execute the shader programs associated with this voxel object.

Now in the viewer panel to the right of this voxel generation widget should be displayed some points that represent the centers of a the HCP Trapezo-rhombic dodecahedral shaped Voxels. The viewport camera may be too far from the origin and the user may need to zoom into the world scene by scrolling the middle mouse button to move close enough to see the generated points. These points are the centers of the equally spaced 3D hexagonal close packed spheres of radius. The points displayed should be of a volume in the shape of a low resolution sphere.

To see the HCP Trapezo-rhombic dodecahedral shaped Voxels, navigate the mouse to the checkbox labelled “Display as points” and uncheck it. A black or dark shape should now appear in the viewer panel.

This is a rather low resolution generation of a sphere and may look rather blocky or not spherical enough. To get a higher resolution and a more spherical shaped generated voxel volume, go to the resolution step number input widget and enter 0.5, and then press the execute function. A new generation of the voxel volume will be performed and replace the previous generation. What now appears in the viewer panel should look similar to Fig 02.0.7.

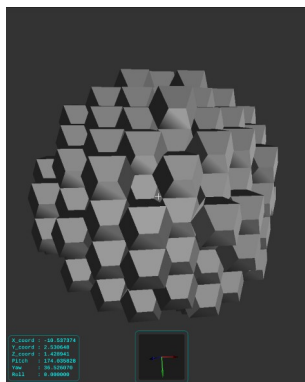


Fig 02.0.6

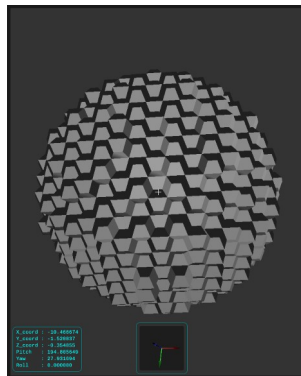


Fig 02.0.7

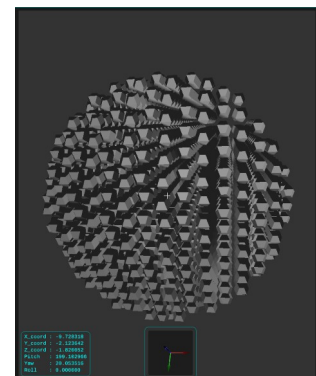


Fig 02.0.8

From here the user can then experiment with the generation variable that has been defined in step vg04. As defined, the variable a is actually the radius of this sphere. If one were to change the value of the variable a from 7.0 to some other value, and then press the Execute Function button, a new HCP voxel sphere would be created of that radius.

The other option to explore changing the variable is to change its value by stepping incrementally up or down by selecting the left and right control buttons. If the user were to press either of these, the value of the variable a would change by the step value defined as explained above. At this point, it is left to the user to explore performing this task to see how it works. Beware that if the leftmost checkbox is not in enable mode the generation will fail and a compute shader error message will be displayed in the code log panel.

The integer variable is the same in functionality as the real variable except that it is for integer generation compute shader variables. The boolean compute shader variable is for boolean variables and cannot have step functionality. Note : If any variable is added, deleted or any change is made to any of the variables activity, or name, then the Execution Function button will need to be pressed so as to recompile the compute shader program and generate the voxel volume data.

If the user would like to see the voxels resized to reveal the voxels on the interior of the sphere, by selecting and moving the voxel scale slider, the voxels will be resized. (Fig 02.0.8) This does not regenerate any voxels.

Without going into the detail of how to code a voxel generating compute shader program, this is basically all that is needed to be able to generate a HCP voxel volume of point data and view it on screen.

If the user so wishes, by pressing the save button, the user can select a directory and file name to save the defined parameters here for later use.

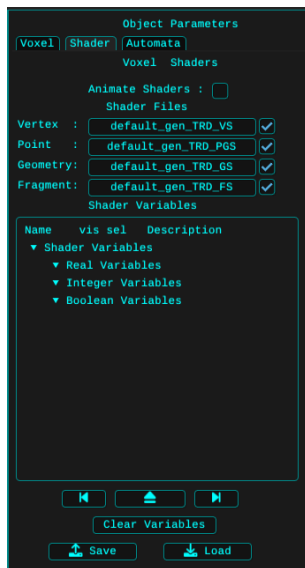


Fig 02.1.0

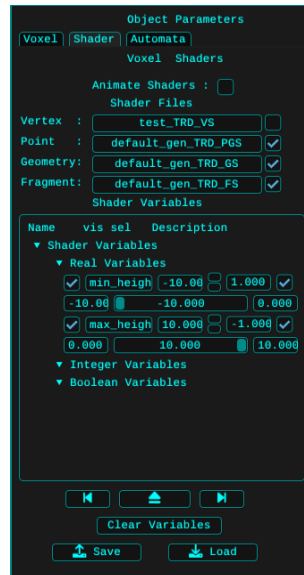


Fig 02.1.1

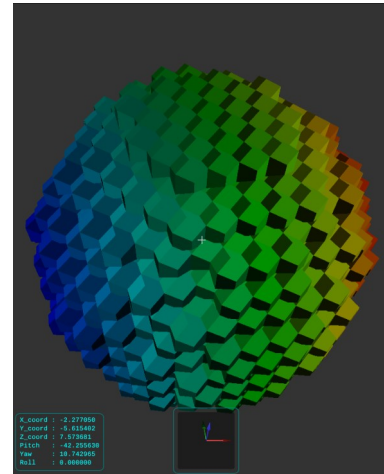


Fig 02.1.2

02.1 Visualising HCP voxel object

Any HCP Voxel generated object being displayed on screen utilises an OpenGL shader program. The points and HCP Trapezo-rhombic dodecahedron shaped Voxels are displayed on screen using a default shader program. Both of these can be substituted to obtain more advanced imagery by using a user defined shader program. This will be a brief description of how a given user defined program can be used without going into the details to much.

From the voxel parameters panel, select the Shader tab to bring up a widget as in Fig 02.1.0. This widget enables the user to select the shaders to use to display the HCP voxels onto the screen, and just like with the compute shader variables explained in the section 02.0 Generating a HCP voxel object, the user can change the values of a shader variable with one exception. The user can use a slider widget to change a variable value.

From top to bottom

Vertex, geometry and fragment shaders : A set of buttons that are used to select a user defined shader code file to generate a shader program. The vertex and fragment shaders are mandatory, and the geometry shader optional, but a geometry shader is used and needed to display hcp voxels as a Trapezo-rhombic dodecahedron.

Shader uniform Variables : This operates exactly as described for the compute shader variables described in section 2.0, with an additional slider widget to interactively change variable values for the real and integer variables.

Left and Right control buttons : These Buttons perform a decrement and increment of one or more of the variables defined in the shader widget to be changed by a given step value.

Generate Shader button : This is the large button between the increment and decrement buttons with an eject or up arrow decal. When any shader variable is added, deleted, renamed or activated etc, this button needs to be pressed to regenerate the shader program. If there is a compilation error, such as a variable name not found in the nominated shader programs above, the compilation error will be displayed in the code log panel.

Clear variables button : Clears all the shader variables

Save : Opens a dialog window to save the shader parameters defined in this widget to a formatted text file of suffix .twm.

Load : Opens a dialog window for the user to select a shader parameters (.twm) formatted text file to be imported into the application and the values displayed in this widget ready to be used.

To change to a less bland visualisation.

Step vv01 : Press the vertex shader labelled "Default_gen_TRD_VS" and navigate to the directory in which the application executable resides Examples/Shader/HCP_Voxel/ and select the vertex shader code file new_test_TRD_VS.glsl.

Step vv02 : Go to the Shader variables widget and as was performed with defining a voxel generation real variable, create two variables here. Define one with the name `min_height`, and the other with `max_height`. Check all the enable check boxes, and assign for the `min_height` variable , a value of -10.0, and a step value of 1.0. A variable slider will be visible below these widgets. The left float input widget is the minimum range value of this variable, and the right, its maximum. Enter -10.0 for the minimum, and 0.0 for the maximum. For the `max_height` variable, enter a value of 10.0, a step value of 1.0, and a slider min and max range value of 0.0 and 10.

This shader widget should appear as in Fig 02.1.1. The voxel display shader program can now be generated and the HCP voxel display updated to the functionality as defined by the shader code of the defined shader types.

Step vv03 : Press the generate shader button (middle button with up arrow). If the display disappears (it should not as this problem has been fixed) it is due to a programming bug having the voxel size set to zero. Just go back to the voxel generation tab widget and move the voxel scale slider to display the voxels. A view similar to Fig 02.1.2 should now appear in the viewer panel.

What the user can now do is move the variable sliders to change the min and max values that are displayed as a color range of the voxels. If the user now selects the increment and decrement buttons, the user will find that both variable values will change in the same step. If any of the step check boxes for enabling the step action is disabled, that variable will not change, while any that are active will.

If the user so wishes, by pressing the save button, the user can select a directory and file name to save the defined parameters here for later use.

This is a basic demonstration of the use of this shader widget. It is intended that the user can create new or use many a variety of shaders to display voxel volume data. The HCP Trapezo-rhombic dodecahedral shaped Voxels are actually drawn using the default geometry shader as only the point data that is observed by enabling the display as points checkbox in the voxel shader widget. It is envisaged that with more development, a whole series of different shader code will be created.

02.2 HCP voxel Cellular Automata

One purpose of creating the HCP spacial data structure and this application is to implement a usage of cellular automata for the purpose of simulation and modelling of real world phenomenon, or pure experimental and fractal exploration, with the ability to define any set of simple rules, an infinite realm of possible behaviours can be explored for computer graphics, art or science. The HCP Voxel 3D spacial data structure has the advantage over conventional cubic voxel modelling in that the centre of every neighbour of any voxel is of equal distance as the neighbours of a cubic voxel sharing one or more vertex points is not.

HCP voxel cellular automata is implemented as a node in the node editor, which needs to be linked to a HCP voxel generated scene object. To create a hcp cellular automata node, in the node editor where one wishes to place a hcp cellular automata node, right click with the right mouse button and release to bring up the floating menu, and and navigate through the menu to the menu option

Main Menu→Create Node ...→HCP Voxel ... → Create HCP Voxel Automata Node.

A HCP voxel automata node should be created. Make a node editor link from the hcp voxel node created above with the label Matrix output pin on the right of the node, to the hcp automata input matrix pin on the left of the cellular automata node.

Select the cellular automata node to bring up a widget as in Fig 02.2.1 in the parameters panel. This widget enables the user to define the HCP voxels cellular automata rules that determine the HCP voxel values from one iteration step to the next, and from these values and the shader program that is defined by the shader code defined in the shader widget tab display the HCP cellular automata results.

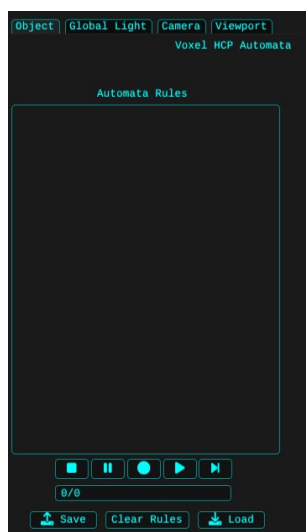


Fig 02.2.1



Fig 02.2.2



Fig 02.2.3



Fig 02.2.4

From top to bottom

Automata Rules : The section of this widget in which the user creates and manages the automata rules by which the application will use in an iterative step by step to determine what value to give to all of the individual HCP voxels within the volume defined in the voxel generation tab widget.

Automata control buttons : These are the automata control buttons to step the automata processing from one iteration to the next, play the automata rules, stop pause and record. As of this time, the record button is not implemented.

Progress bar : An indicator of the progress of the automata process

Clear variables button : Clears all the cellular automata rules variables.

Save : Opens a dialog window to save the cellular automata rules parameters defined in this widget to a formatted text file of suffix ABYTER.txt.

Load : Opens a dialog window for the user to select a cellular automata rules parameters (ABYTER.txt) formatted text file to be imported into the application and the values displayed in this widget ready to be used.

Given here is a simple example.

Step va01 : Move the mouse into the automata rules section and press the right mouse button to bring up a one option floating menu add rule. An automata rule should be added to this section of the widget and look like Fig 02.2.2.

The text input widget at the top left of the rule is the name of the rule being applied. Next to this is a button labelled "Neighbour Rules" to bring up a floating widget that defines the rule to be applied. The second line below text "rule" is the value that the voxel will be given if the defined rule is met. The start and stop entry values are the iteration step that the rule will apply over. The Activate check box indicates whether the rule will be applied.

Step va02 : Left click the mouse over the Neighbour Rules button. An Automata neighbour Rules popup widget should be displayed as in Fig 02.2.3.

The neighbour rules are the conditions that must be met for the cellular automata rule to be valid and the voxel set to the defined rule value. Each voxel has twelve neighbours, and each neighbour of the voxel is identified by the number on the left of this popup. For information of which neighbour these numbers refer to the section 4 Cellular Automata of the documentation file vw_HCP_Voxel_draft_01.pdf. A rule also can be defined for the self value of the voxel itself that is being evaluated.

An automata rule for each neighbour or self voxel is defined in this popup according to the mathematical relationship ar01 as given in section 4.2 of vw_HCP_Voxel_draft_01.pdf

$$A (<, <=, \text{ignore}) VV (=, !=, <, <=, \text{ignore}) B \quad - \text{ar01}$$

which states the rule is met for a neighbour if the voxel value (VV) of the neighbour voxel is one of any of the combinations of conditions that is within the brackets such that it falls within the range of value A, and value B. This may be explained better with some examples.

If have a rule ar02 form one particular neighbour

$$A < VV < B \quad - \text{ar02}$$

then this rule states that if the neighbours or self voxel value VV is between values A and B, then that rule condition is met for that voxel neighbour.

If have a rule ar03 form one particular neighbour

$$\text{ignore } VV \leq B \quad - \text{ar03}$$

then this rule states that if the neighbours or self voxel value VV is less than or equal to value B, then that rule condition is met for that voxel neighbour.

If have a rule ar04 form one particular neighbour

$$A < VV \text{ ignore} \quad - \text{ar04}$$

then this rule states that if the neighbours or self voxel value VV is greater than the value A, then that rule condition is met for that voxel neighbour.

So by use of the relationship defined in ar01, any automata rule pertaining to the integer value that the voxel neighbour has can be created with a very large set of possible rules. It is this method of defining the cellular automata rule for each voxel neighbour and itself that is represented in this popup widget, with the lower rule being the left side of ar01 and the upper rule being the right side of ar01.

Note : Virtual worlds currently has the voxel value VV in the one byte range of 1 to 255 that are valid, with the value 0 reserved as an invalid voxel value. So any value of zero for any voxel will not be displayed on screen

With modification, and for different types of data that the voxel value can have, other types of rules can be defined and implemented as desired, but for the purposes of this demonstration, the following simple set of rules will be applied.

Step va03 : Enter into the lower rule of neighbour 0 for the value 255. Leave the selection of relationships for the lower rule as the default negative sign. The negative sign means ignore the value. For the upper rule enter 1 as the value that the rule must use, and select the less than (<) rule relationship for the rule of voxel neighbour 0.

What this rule defined in step va03 means is that if the voxel value of the neighbour voxel 0 is less than 1, then the rule for that neighbour is met and the thus valid for this neighbour.

Step va04 : Repeat step va03 for voxel neighbours 1 and 2.

The cellular automata rule to be applied to all voxels should now appear as in Fig 02.2.4. Press the Accept button for this neighbours rule to be applied for this cellular automata rule, other wise it will be discarded upon exit.

Step va05 : Going back to the rule definition that was created in step va01 and displayed in Fig 02.2.2, enter in the stop widget a value of 5, and activate the rule by enabling the active checkbox Fig 02.2.5.

A cellular automata rule has now been defined to operate over the iteration step interval from 0 to 5. The start and stop iteration steps are so that multiple different rules such as defined above can be performed over different periods or iteration steps. This has not been thoroughly tested yet. For the purposes of this demonstration, the user can now perform the cellular automata rule by clicking on the next step button of the control buttons.(ie the right most button)

Do this a couple of times and see if there is a change in the appearance of the sphere. If not, move the mouse into the viewer panel and press the right mouse button to rotate the sphere until something similar to Fig 02.2.5 is seen.

What this rule does is to assign a value of zero to the voxels that have their neighbour 0,1 and 2 voxels of a value 0. The 0,1, and 2 neighbours are the topmost neighbours of any Trapezo-rhombic dodecahedral shaped voxel, so the hcp voxel sphere has all the voxels with no neighbours of a valid value (ie zero) being assigned as invalid (ie value zero) and thus not being displayed on screen.

Continue pressing the next iteration step control button until no more changes are made and the progress bar is complete. Fig 02.2.5. To restore the voxel volume to its original state, go back to the voxel generation tab widget and press the “execute function” to regenerate the data.

Now enter the Automata tab once again and reset the automata rules by pressing the stop button in the button controls (ie far left square shaped button) The progress bar should now be reinitialised. Press the play button (the second to last solid arrow button) and the cellular automata rule should now play over a period of five iterations.



Fig 02.2.5

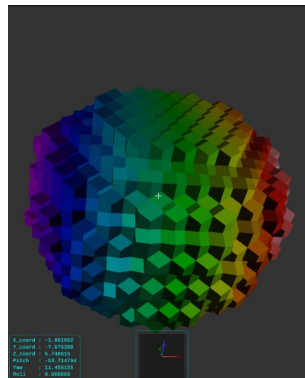


Fig 02.2.5

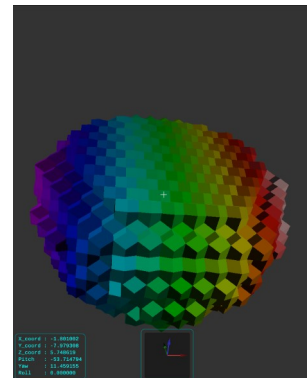


Fig02.2.5

If the user so wishes, by pressing the save button, the user can select a directory and file name to save the defined cellular automata rules here for later use.

02.3 Exporting a HCP voxel object Data

HCP voxel objects that are generated or modified by use of the cellular automata feature of this application can have the current state of the point cloud data of the HCP voxel that is displayed on screen exported as a .ply file to be imported into a third party 3D application such as Blender or Meshlab.

It is assumed that for this tutorial, the user has at least performed the steps and got the result as described in the section 2 **Generating a HCP voxel object**, and has on screen, something similar to Fig 02.0.7, or Fig 02.1.2.

Step ev01 : To export the generated HCP voxel point data as a point cloud, the user needs to make sure it is selected as the current voxel object. This is confirmed by having the node corresponding to the voxel object clicked and selected in the node editor panel of the scene objects such that the generation, shader or automata tab are displayed in the object parameters panel.

Step ev02 : With the voxel shader object selected in the node editor, the user can then right mouse click and release to display a floating menu for this node. Select the menu option

Node/Entity ...→ functions→ Voxel→ Export→As point Cloud

or one of the other export options.

Point cloud will export the entire generated voxel volume of valid voxel values as displayed to a .ply file

Point surface will export only the voxel point data of the generated voxel volume of valid voxel values that make up the surface of the volume to a .ply file

As face surface will export only the voxel data of the generated voxel volume of valid voxel values that make up the surface of the volume as triangles to a .ply file. This selection will recreate what the user sees on screen with only the external triangles that makes up the volume surface being exported. This exported face data is of individual triangles, of which the vertices of some will be the same. To obtain an emergence of all these into a single defined surface, an application like mesh mixer or blender will be needed to import and display this surface

Step ev03 : Select the menu option that one wants to export, and another menu sub menu selection will be visible that is the same for all types of export. For the purpose here, the selection of selected will suffice. A save file dialogue will appear for the user to select a file name location to save the exported data.

The exported data will have a file name with the format of

cp_object_name.ply for voxel volume centre point data as a point cloud

sp_object_name.ply for voxel surface centre point data as a point cloud.

sf_object_name.ply for voxel surface centre point data as a triangle vertex surface.

If the user were to use a 3D application like MeshLab, the exported data will appear as in Fig 02.3.1, 02.3.2 and 02.3.3.

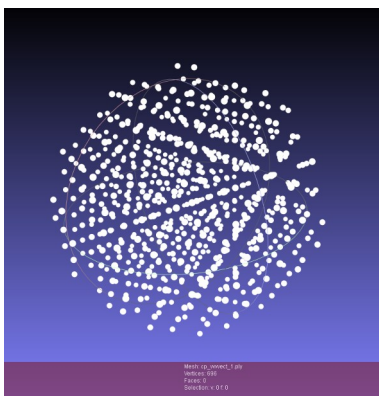


Fig 2.3.1

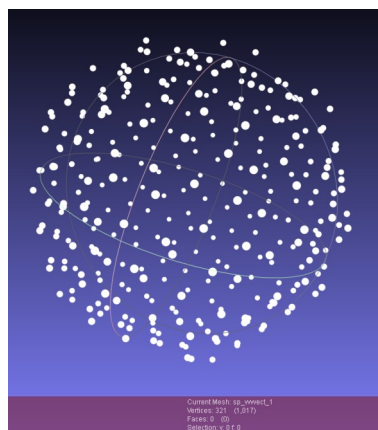


Fig 2.3.2

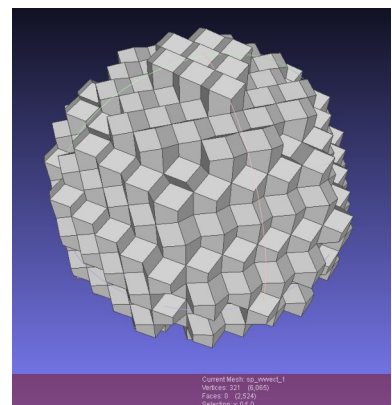


Fig 2.3.3

The HCP voxel volume point data of what is displayed on screen can be exported in any of these formats, and thus while performing the cellular automata steps, the cellular automata configuration of any step can be exported.

02.4 Create HCP Voxel Object From Cartesian Point Data

If one has a set of point data in Cartesian coordinates that is generated by a C++ function, or exists within a 3D formatted vertex file (eg a .ply formatted file), then this point data can be converted into a HCP voxel object to represent the point cloud data as voxels. This is achieved by a HCP voxel matrix of a given origin, size and range attributes that fit within the point data set volume in space that is to be imported. A test is performed to see if any point data exists within the volume of any of the HCP Trapezo-rhombic dodecahedron voxel cells. If a point does exist within a cell, the data attribute value for that cell within the computer data storage array is given a value to indicate one data point exists within that cell. If more than one point exists within a cell, that value is increased to represent the number of coordinate points that are found to exist within that voxel cell. Thus a resultant HCP voxel matrix is formed representing a kind of density of data points within each HCP voxel cell that can be displayed on screen.

An explanation how this is achieved is given in section 6 of the documentation file `vw_HCP_voxel_draft_01.pdf`

Presented here is an example of two methods to create a HCP voxel matrix of a cartesian point data set.

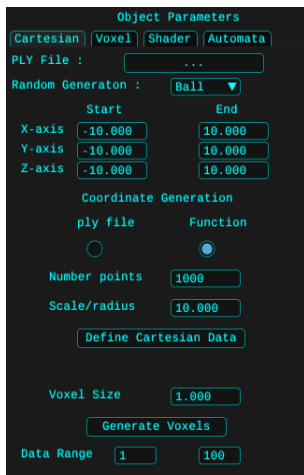


Fig 02.4.1

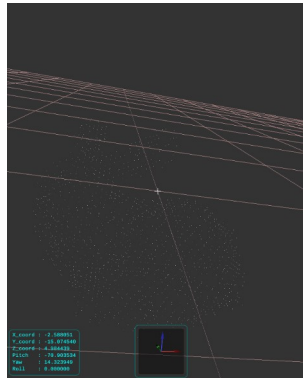


Fig 02.4.2

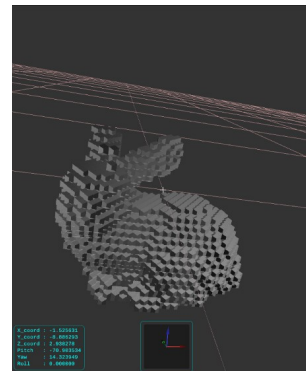


Fig 02.4.3

The following steps describe how to create a voxel matrix of HCP Trapezo-rhombic dodecahedron voxel cells from a .ply file.

Step vc01 : In a similar manner to creating a HCP voxel object, at a location in the node editor where one would like to place a cartesian hcp voxel node, with no node selected, click the right mouse button and release it to display a node editor's floating menu. Make the menu selection

Main Menu→ Create Node→ HCP Voxel→ Create Cart to HCP Voxel Node.

Select the created node and in the parameter panel a widget similar to that of Fig 02.4.1 will be displayed.

From top to bottom

PLY File : A button to bring up a file dialogue for the user to select a PLY file that contains vertex xyz point data to be used to create a HCP voxel matrix.

Random Generation : A button drop down list to for the user to select a method of generating a set of random xyz point data to be used to create a HCP voxel matrix. Available in this demonstration are Ball and Sphere. Ball generates a solid sphere of random points, and Sphere is a spherical shell of random points.

Definition of the x-y-z volume limits of that the HCP voxel volume generation is to be confined to. This is automatically updated to fit the Cartesian data that is generated or imported.

Coordinate Generation : Two radio buttons to select the type of point data that is to be used to create the voxel matrix. Ply is to use the nominated PLY file defined by the PLY button, or function to use the random data selected in the Random Generation drop down list.

Number Points : Is to specify the number of points that the random function is to generate if the function radio button is selected.

Scale/Radius : Is to specify a Scale factor of the imported xyz data points of the imported PLY file, if the PLY generation is selected, or the radius of the sphere of the random generated point data if the function generation is selected.

Define Cartesian Data : Is to import or generate the point data to be used to create the voxel matrix. This will also get the boundary of the volume that this data will occupy in 3D space and update the start

and end boundary limits. If the user wishes, these boundary limits can be changed.

Voxel Size : This is the size of the resultant voxel matrix cells. The size chosen will determine the level of detail of the point data to be viewed.

Generate Voxels : Pressing this button will create a matrix of HCP Trapezo-rhombic dodecahedron voxel cells. A value is given to each cell to indicate the number of points that exist within it.

Voxel Scale : Widget to scale the HCP voxel size when not displayed as points.

Data Range : Widget to select the range of number of points that each voxel cell must have for that voxel cell to be displayed on the screen.

Step vc02 : Select the .ply file selection button to open a file selection dialogue. Select the bunny .ply file that should be present in the Data directory that this application executable is located within. The button text should now have the text bunny present.

Step vc03 : If the ply radio button is not selected under Coordinate Generation, then select it to highlight and activate the selection that the unstructured Cartesian data in the ply file bunny.ply is to be displayed as a set of structured HCP Trapezo-rhombic dodecahedron voxel cells.

Step vc04 : The number of points has no effect in this case, so can be ignored, but for this bunny.ply data it is known that the x,y,z data values are in the range of ± 0.5 , which is rather small to represent as voxel cells, so need to be scaled by at least 10. If the Scale/Radius value is not a value of 10.0, then enter this value into the entry box.

Step vc05 : Press the "Define Cartesian Data" button to import the bunny.ply xyz point data set at a scale factor of 10.0. The values of the volume limits to create the voxel matrix should change and be in the range of ± 5 units.

Step vc06 : For a reasonable representation of the data, change the Voxel size to 0.2 units. Press the Generate voxels button.

In the viewer display, a default view of points representing the centers of the HCP voxel cells should appear in the shape of a rabbit in a sitting position. If necessary, rotate and/or move the camera to get a better view of this data.(Fig 02.4.2).

Step vc07 : To obtain an image of the HCP Trapezo-rhombic dodecahedron voxel cells, deselect the voxel "Display as points" checkbox. The viewer should now switch from a display of points to a display of voxel cells. If necessary, rotate and/or move the camera to get a better view of this data.(Fig 02.4.3).

To obtain an image of the HCP Trapezo-rhombic dodecahedron voxel cells such that the cells are colored and sized to give an indication of how many data points exist within each cell, select the shader tab (Fig 02.1.0) . Entering in the vertex shaders to use and defining shader variables is performed as in section 2.1

Visualising HCP voxel object. The steps to perform the task to define the shader code files and variables will not be described here. What the user can do is to import a parameter file with all these parameters defined.

Step vc08 : Select the "Load" button at the bottom right of the shader widget to bring up the file dialogue to select a shader parameter file to import. Navigate to the directory Examples/Shader/CtoV and select the file ctov_range.twm. This will load the shader parameters needed to display the HCP voxels in a different style.

Step vc09 : The directory path names to the vertex, geometry and fragment shader pathnames will probably be different to that on the user computer. One by one, select the button for each of these files to open a file dialogue and navigate to the directory

`<user directory this application exist within>\Examples\Shader\C_to_V\`

and select the default_CtoV_VS.glsl, default_CtoV_GS.glsl and default_CtoV_FS.glsl files that are listed. This will define the exact path name required to create the shader programs.

Step vc10 : Press the Shader update or regeneration button to generate the shader program. No error messages should occur and the voxels from the display may disappear. If an error message(s) are present, it is because an incorrect shader file has been selected or a variable name has been changed or is absent, or not defined to be used. Examine the error message and make the appropriate correction.

Step vc11 : If The shader compilation is successful and the display is blank, (it should not as this problem has been fixed) it is due to a programming bug having the voxel size set to zero that has not yet been fixed. Just go back to the voxel generation tab widget and move the voxel scale slider to display the voxels. A view similar to Fig 02.4.4 or Fig 02.4.5 should now appear in the viewer panel.

To flip between the views of Fig 02.4.4 or Fig 02.4.5, activate and deactivate the boolean variable scale_density that is defined in the shader tab widget. That is, the checkbox right of the boolean variable

scale_density.

The colors and size of the individual voxels give an indication to the number of coordinates that lie within each HCP voxel cell volume. By moving the slider widgets named min_density and max_density, the user can specify the range of data points that are to be displayed. Moving the min_density will mean that only voxels with data points above this value are considered, while moving the max_density slider means that only data points below this value are considered. Red colors are voxels with high number of data points in this specified range, while violet to purple, low number of points in the specified range that exist within the voxel cell.

The same procedure for displaying the data of a .ply file can be applied for a function as well. If the user now repeats the above steps vc02 to vc06, but select the radio button “Function” a sphere of random voxels similar to Fig 02.4.6 should be present.

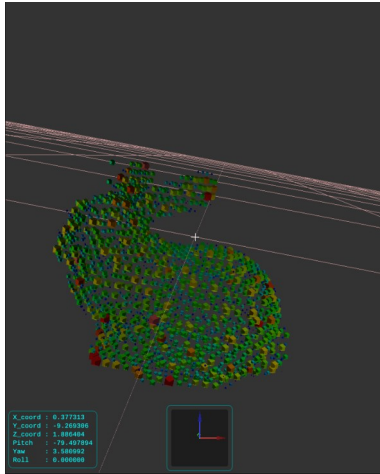


Fig 02.4.4

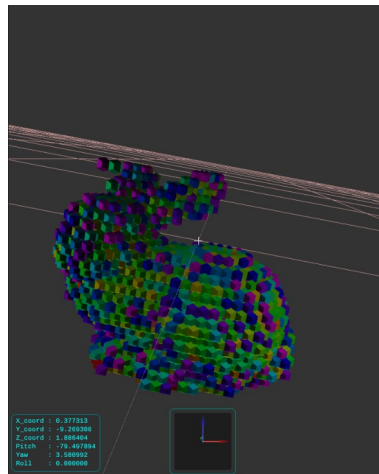


Fig 02.4.5

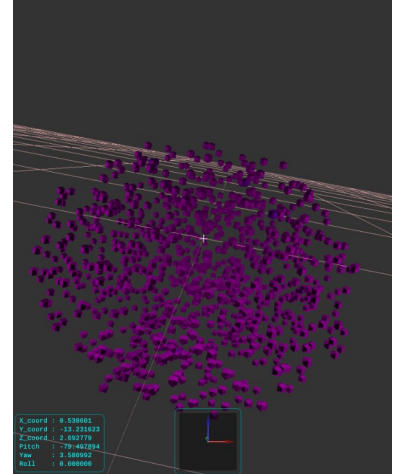


Fig 02.4.6

For experimentation, the voxel size and/or number of points for a function can be changed to get different results. Change in the voxel size is also a change in level of detail of the point data being displayed. If the voxel size is too small relative to the distribution or data, large blank spaces become apparent. If too large, only a few voxels, if not only one are present, and the shape of the data is lost.

This is the end of the instructive demonstration of the implementation of the concept of use of a hexagonal closed packed 3D spacial structure of voxels. It is hoped that this is enough to give an indication that there is a potential for wider developments that any reader may find worth investigating and inquiring about.

For further information on this topic and what coding has been written to implement this concept, please read the other documentation that is present with this document.

02.5 HCP Voxel Editor

Voxel editor applications exist and are widespread allowing users to create 3D volumes similar to building models using lego bricks. Such a rudimentary basic primitive voxel editor is present within the Virtual Worlds application and is utilised as a node in the node editor. It is intended to extend and expand this voxel editor into perhaps it's own module, or even application beyond what is presented here.

This first incarnation or version of a HCP Voxel editor is designed to act as a method of voxel generation, and can also import saved HCP voxel matrices to be edited and saved. It is aslo designed like any other generator of HCP voxel data, to have the voxel matrix available to be used elsewhere in the application, and in particular, for use in cellular automata.



Fig 02.5.1



Fig 02.5.2

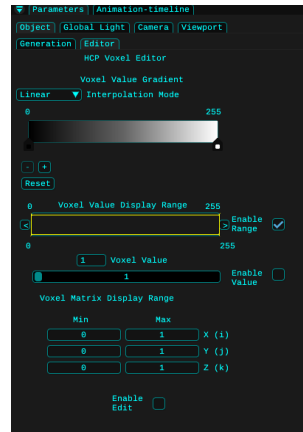


Fig 02.5.3

Section 02.5.1

To open the HCP voxel editor and start creating a HCP matrix

Step ve01 : In a similar manner to creating a HCP voxel object, at a location in the node editor where one would like to place a hcp voxel editor node, with no node selected, click the right mouse button and release it to display a node editor's floating menu. Make the menu selection

Main Menu→ Create Node→ HCP Voxel→ Create HCP Voxel Editor Node.

A node similar to that displayed in Fig 02.5.1 should be present.

Select the created node and in the parameter panel a widget similar to that of Fig 02.5.2 will be displayed.

The node HCP voxel editor has two tabs, a HCP voxel generation tab, and an editor tab. The generation tab specifies the parameters to generate a HCP voxel volume (Fig 02.5.2), and the widgets editor tab (Fig 02.5.3) gives control of the display and editing brushes to edit the HCP voxel data within the voxel volume.

Generation tab from top to bottom.

Display as points : Change the display of the generated data from points to voxels.

Voxel Scale : Widget to scale the HCP voxel geometry size when it is selected to not to display the data as points.

Voxel volume : Button to open a file dialog to select a voxel volume data set to import and edit. Any current selected files to import will have the name of that file displayed in this button

Generated voxel volume limits : Definition of the x-y-z volume limits of that the HCP voxel volume generation is to be confined to.

Limits of voxel volume : Button to read the voxel volume data file defined and the name displayed in the **Voxel volume** button text to find the limits of the volume that the data is defined to be within , and display the x-y-z limits in the **Generated voxel volume limits** widget.

Import voxel volume : Button to import the voxel volume data file defined and the name displayed in the **Voxel volume** button. The data imported will have the limits of the volume it is imported to be the same as defined by the voxel volume limits defined above.

Min Voxel value : The minimum voxel value that is to imported or created. Any voxel values less than this will be ignored.

Max Voxel value : The maximum voxel value that is to imported or created. Any voxel values greater than

this will be ignored.

Default Voxel value : The voxel value that is to be assigned to all voxels upon generation of a new voxel volume, or given to all voxels that are outside the minimum and maximum surface range.

Resolution step : The resolution or size of voxel grid cells that are to be generated. Note: the voxel size of any imported data is determined by the size of voxels defined by the imported data file.

Create New : Generate a new voxel matrix volume with the voxel value given by the default voxel value

Save Volume : Export and save the current voxel editor grid data into a file.

Save : Opens a dialog window to save the current voxel editor parameters defined in this widget to a formatted text file of suffix .vep.

Load : Opens a dialog window for the user to select a voxel editor parameters text file with the suffix .vep to be imported into the application and the values displayed in this widget ready to be used.

Step ve02 : Define the limits of the HCP voxel volume to edit, enter -10.0 for each of the start or minimum for the x,y,z axis, and 10.0 for the end or maximum for each of the x,y,z axis.

Step ve03 : In the default voxel value enter a value of 128. This is the value that every voxel will have upon generation.

Step ve04 : In the resolution step widget, enter a value of 1.0 if it is not already set to this value. This defines the resultant generated voxel volume matrix voxel size.

Step ve05 : Click on the Create New button to create a HCP voxel volume with the parameters entered in the previous steps.

In the View port display, a HCP voxel matrix of similar appearance to that of **Fig 02.5.4** should be present. If necessary, zoom in or out from the central origin of the scene using the middle mouse scroll wheel, and adjust the camera view via clicking on the left mouse button and move the mouse.

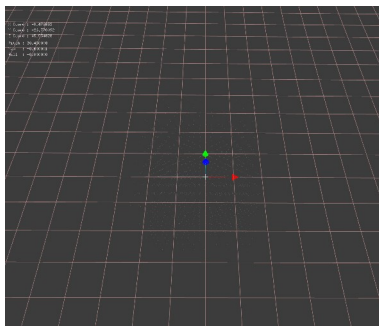


Fig 02.5.4

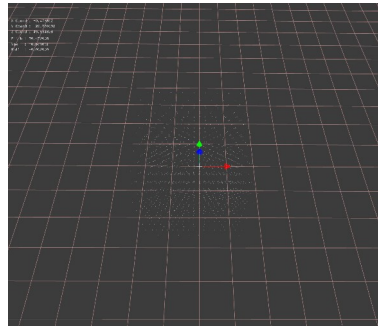


Fig 02.5.5

The points representing the HCP voxels will look to have a rather dark grey color. To change this

Step ve06 : Select the Editor tab to bring up the HCP voxel editor editor widget as displayed in Fig 02.5.3.

From top to bottom.

Voxel Value Gradient : This is a widget that defines a gradient of colors to be displayed in the voxel editor widget that reflects the values of each voxel being displayed. The number 0 to 255 at the top of the color bar gives the voxel value that the color in the color bar represents.

New colors can be added to the gradient by moving the mouse cursor into the color bar of the widget and left clicking at the position one wishes to add a new color, or by selecting the + button.

To delete a color, select the color slider widget and select the – button.

To edit a gradient color, select the color slider widget to highlight it, and then select the button with a color of that slider widget (after the + button) to bring up a color picker widget in which to change the gradient color at that color sliders location. The number entry after the color edit button is number that represents the percentage or location in relative terms to the value of 255. ie a multiplier of 255. eg .5 = $255 \times 0.5 = 128$.

The gradient colors can be displayed in a linear or step fashion. The selection of gradient type can be selected via the pull down menu at the top of this widget.

Voxel Value display range : This is a widget that defines the range of voxel values that are to be displayed.

There is a left or minimum range slider (<) to define the lower voxel value that is to be displayed, and a right or maximum slider (>) to define the upper voxel value to be displayed. By moving these sliders, the minimum and maximum voxel values to be displayed are changed and, those values are displayed below these sliders. The enable range check box next to this range slider widget enables or disables this widgets function. If not selected, all voxels of any value is displayed.

Voxel Value : This is a widget that will display all voxels that have the value specified in the editor if the

Enable Value check box is activated regardless of whether it falls in the Voxel value display range or not.

Voxel Matrix display range : This is a series of ImGui drag widgets that defines the limits of the voxel volume to be displayed in terms of the voxel matrix i,j,k coordinates. This allows the user to only have displayed a portion of the voxel matrix to be viewed and edited.

Enable Edit : Activating this check box widget opens the voxel editor voxel edit options

To get a better display of the created voxel volume

Step ve07 : Go to the Voxel value gradient widget and move the end color slider widget to the middle of the gradient, until a position is reached where the color of the points in the view port are a white color as in Fig 02.5.5.

Step ve08 : If the user moves the voxel value range sliders, either the min that goes above a value of 128, or the max that goes below 128, these points representing the voxels of value 128 will not be displayed. If these voxels are not displayed, then by entering 128 into the voxel value widget entry widget, or sliding the voxel value slider unit 128 is displayed, these voxels can be displayed by activating the enable value check box.

Step ve09 : If the user changes the values in the Voxel Matrix Display range widgets for the min and max x,y,z or i,j,k, axis, the display of the voxel matrix will change to display the range of the voxel matrix range in terms of the voxel matrix i,j,k, coordinates.

These widgets give the user the basic tools to analyse and view the voxel matrix that they are creating or editing.



Fig 02.5.6

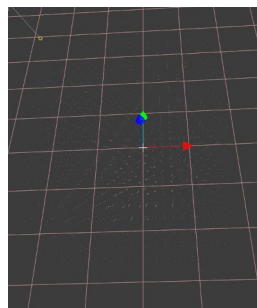


Fig 02.5.7

To start editing this voxel matrix activate the Enable Edit checkbox Widget to display the voxel editor widgets and activate the voxel editing mode in the view port of the voxel matrix created in steps **ve02** to **ve05**. Below the Enable Edit check box widget should appear a group of widgets similar to **Fig 02.5.6**. This is a very basic and rudimentary brush tool to edit voxels in the created and displayed voxel matrix.

Brush Value : Is the value that the brush will change the voxels value to that it overlays.

Brush Radius : Are the dimensions of the rectangular shape in world coordinates on a x-y-z axis that define the editing brush size.

If the user now goes to the view port, it should be seen that there is a yellow circle surrounding one of the displayed vertex points and a line drawn with a number at its end. This number is the voxel matrix vertex index number. By moving the mouse about, the user will see that the circle and number will change. This circle around the vertex data point and its index can be considered as the selected central voxel that is to be edited and depending on the brush dimensions, change the voxels that fit within the volume dimensions of the editing brush. To demonstrate this.

Step ve10 : Go to the voxel value gradient and move the slider with the white color to the far right or at the value of 255. The displayed voxels should now be a mid dark grey color.

Step ve11 : Enter in the Brush Value widget a value of 250

Step ve12 : Enter or change in each of the brush radius value ImGui drag widget to a value of 5 by double clicking within each widget and entering 5.

Step ve13 : Go to View port and with moving the mouse, select a voxel to edit it and the surrounding voxels to this value of 250 and press the space bar. The selected voxel and its neighbours should now appear with a white color indicating they have been changed to a value of 250. **Fig 02.5.7**.

This is the basis of the HCP voxel editor operation. By changing the brush value and radius, more voxel editing can be done, and by changing the voxel value gradient, voxels of different values can be envisioned with colors according to this gradient. The newly created and edited Voxel matrix data can be exported for future use.

To save the HCP voxel matrix data for future use and editing.

Step ve14 : Select the Generation tab to go back and display the generation tab widget that was instructed to do in **step vw01**. The generation tab widget of **Fig 02.5.2** should be displayed.

Step ve15 : Select the Save Volume button to bring up a dialogue to enter a file name to save the HCP voxel volume data to. Navigate to a desired directory to save the file to and enter a name to save the

voxel volume data to. If an existing voxel volume data file exists and one wishes to overwrite it, select the file and follow the usual steps as with any file dialogue to save the data.

To import an existing HCP voxel matrix data file to edit its data.

Step ve15 : In the same voxel editor generation widget as specified in **step ve14**, Select the Voxel Volume button to bring up a dialogue to select a file name to import the HCP voxel volume data from. Navigate to a desired directory where the file to import exists, and select the file and follow the usual steps as with any file dialogue to select the file to import.

Step ve16 : To initialise the limits of the volume of data to import, press the “Limits of Voxel Volume” button widget to have the software read the selected file to extract the volume limits. If desired, the user can then modify these limits to contract or expand the voxel volume matrix for importing data into and for editing.

Step ve17 : Select the Import Voxel Volume button widget and the data that exists within the limits specified will be imported for display and editing as described in steps **ve06** to **ve13**.

To view the HCP voxel volume as Trapezo-rhombic dodecahedral geometry shapes instead of as a point cloud deselect the Display as points check box at the top of the generation tab widget.

Is recommended that all editing is performed with the voxel matrix displayed as points as it can be difficult to make selections of a specific hex grid cell when displaying the grid as Trapezo-rhombic dodecahedral geometry.

03.0 Generating a hexagonal Voxel object

Hexagonal voxel objects are generated using a text file with an OpenGL compute shader program. So before doing anything else, the user needs to create or have an OpenGL compute shader program source code text file to use. Virtual worlds already has some examples available for use in the examples directory of where the Virtual_Worlds executable is located. To find out how to create a Virtual Worlds compute shader code, see the documentation on how to do this in section 4 writing a compute shader program.

To generate a hexagonal voxel object, move the mouse cursor to within the node editor as described in section 03.1 and right mouse click over an empty region where one wishes to place a node to represent a hexagonal voxel and release it.

A floating menu should appear with the option Main Menu. Move the mouse cursor over this option to display a sub menu. One of these options should be **Create Node** Move the mouse cursor over this menu option and another sub menu should appear with a menu option **Hex Surface** Move the mouse cursor over this menu option and another sub menu will appear with one option being **Create Hex surface Node.** (Fig 03.0.1). Left click with the mouse on this option, and where the mouse cursor was placed in the node editor should be a Hex surface node similar to Fig 03.0.2

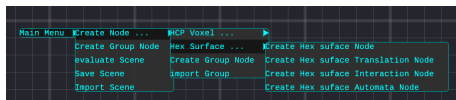


Fig 03.0.1



Fig 03.0.2

Select this newly created node so as to highlighted it with the left mouse button. Depending on the configuration of the GUI one should see a panel on its own or grouped with others as a tab with the label Parameters. Display the parameters panel or any tab with the label Parameters to display it. Select the tab within the parameters panel that has the label **Object**. Within this object tab widget, more tab widgets should appear. Select the tab with the label **Generation**. A widget as in Fig 03.0.3 should be displayed. The parameters in this widget are what are required to generate the hexagonal voxel data. From top to bottom.

Expression button : to select the virtual worlds OpenGL compute shader program file that generates the hexagonal surface voxel data.

Definition of the x-y area limits of that the hexagonal surface voxel area generation is to be confined to.

Compute invocation is a number that is relevant to the OpenGL compute shader process where the number indicates the number of threads running in parallel is used in the calculation. Generally from documentation, the higher numbers are used if have a large number of calculations to be performed.

Execute Function : Generate the hexagonal surface voxel data. Only press when everything is ready.

Min/Max surface values : These are the minimum and maximum permissible values that the hexagon voxels can posses to be designated as a valid voxel to be stored in computer memory and displayed on screen.

Resolution step : This is the size of the hexagonal surface voxel.

Threshold : Generation of hexagonal surface voxels is of the form voxel value = $F(x,y)$ and the threshold is the minimum absolute value that this voxel value can have to be considered as a valid voxel to be stored in computer memory.

Display as points : Change the display of the generated data from points to some other form. This display is determined by the shader program and parameters that are defined in the shader tab widget which is discussed below.

Voxel Scale : Widget to scale the size of the hexagonal voxel when it is selected to not display the data as points. This is shader program dependent and thus not be implemented or used.

Voxel Generation Variables : A set of widgets that enable the user to manage and view the set of variables that are used to generate the area of voxels within the area as defined in the parameters above this widget. Any variables defined here must have an equivalent uniform shader variable that is defined in the compute shader program code that is used to generate the hexagonal area or there will be a shader compilation error and no voxel area generated.

Left and Right control buttons : These Buttons perform a decrement and increment of one or more of the variables defined in the hexagonal generations widget to be changed by a given step value, and then regenerate the hexagonal area and display it on screen.

Clear variables button : Clears all the voxel generation variables

Save : Opens a dialog window to save the voxel parameters defined in this widget to a formatted text file of suffix .hgp (hexagonal generation parameters)

Load : Opens a dialogue window for the user to select a hexagonal generation parameters (.hgp) formatted text file to be imported into the application, and the values displayed in this widget ready to be used.

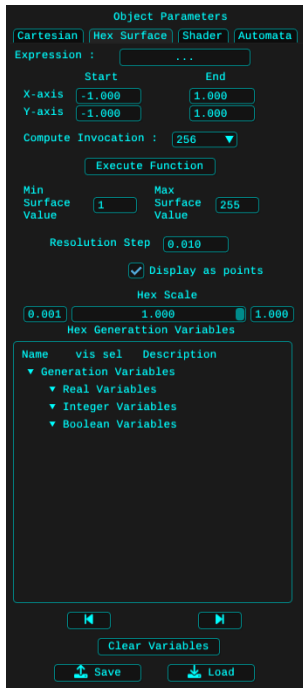


Fig 03.0.3

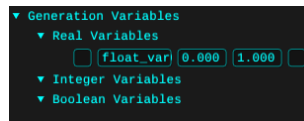


Fig 03.0.4

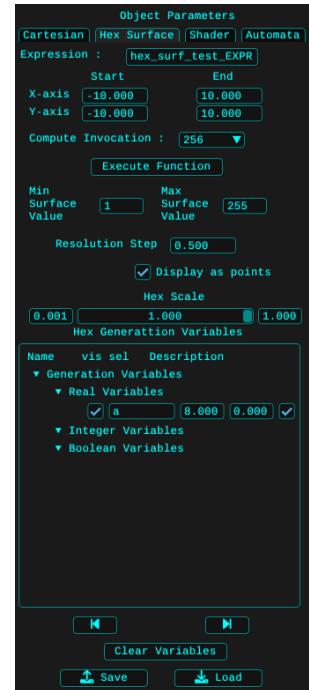


Fig 03.0.5

The details of creating a hexagonal surface generating compute shader program to be selected for use here is in the section 4, Create Voxel Compute Shader Program. Creating a hexagonal surface compute shader program is identical to that for a HCP voxel with the exception that the output value is the z value of a surface function of the form $z = f(x,y)$. For the present as this is an overview of generating a hexagonal surface area and displaying it, it is only required for the user to select an existing example.

Step hg01 : Select the Expression Button at the top of this widget to bring up the file load dialogue.

Step hg02 : Navigate to the directory Examples/Generation/Hex_surfaces/ that the Virtual Worlds application resides in, and select the file name hex_surf_test_EXPR.txt.

The button name should now be changed to hex_surf_test_EXPR. If not repeat steps hg01 and hg02 above.

Step hg03 : To define the area that the hexagonal surface voxels will be generated over, change all of the start x,y values to -10, and the end values to 10

Step hg04 : Go to the hex generation variables section and with the mouse cursor, navigate to and hover over the Real Variables tree node widget to highlight it. Press the right mouse button to bring up the only available floating menu option “add variable”, and select it to create a new real (ie floating) variable.

The variable widget that is displayed (Fig 03.0.4) is the same for float and integer shader variables. From left to right.

Select and activate the left checkbox by selecting it with the left mouse button. This is to enable this variable to be used in the voxel generation compute shader program.

Select the text input widget and change the text to a. This is the name of a variable that is defined in the voxel generation compute shader code to generate a hexagonal voxel point cloud of a 2D surface.

Select the floating number input widget and type in the value 8.0. This is the value that this variable is defined to be assigned to.

The next floating point input widget is the incremental step value to change the variable by as a step action if the increment or decrement button is selected. If it is not 1.0 set this value to 1.0.

The last checkbox widget is to indicate to the application if the step variable is to be used or not when performing incremental or decremental step operations. Select and set this checkbox widget to enable the step variable to be used.

The hexagonal generation panel should now look like that of Fig 03.0.5.

The generation of a simple hexagonal voxel surface can now be performed.

Step hg05 : Press the button labelled “Execute Function”

This will generate a hexagonal 2D surface voxel point cloud of data, and in the viewer panel this voxel generation 2D surface will be displayed and the points that represent the centers of a surface of hexagonal shaped Voxels. If there are too few points to see this surface adequately, increase the number of points by entering a value of 0.1 in the resolution step float input box and repeat **step hg05**. If the camera is too close

to or within the generated voxel shape, move the viewport camera by scrolling the middle mouse button or by pressing the number 2 or down arrow on the number keypad. (Note: Number lock must be disabled) The 2D surface should look similar to Fig 03.0.6.

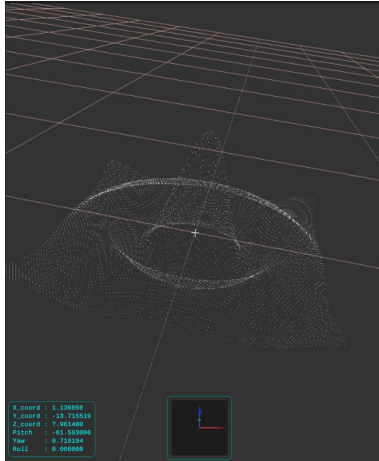


Fig 03.0.6

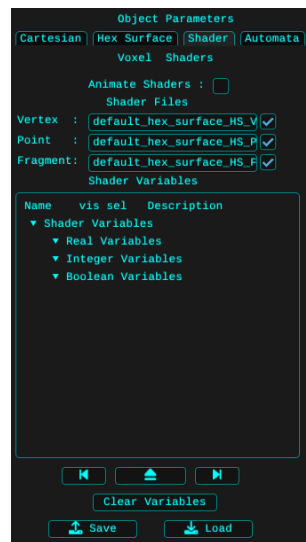


Fig 03.0.7

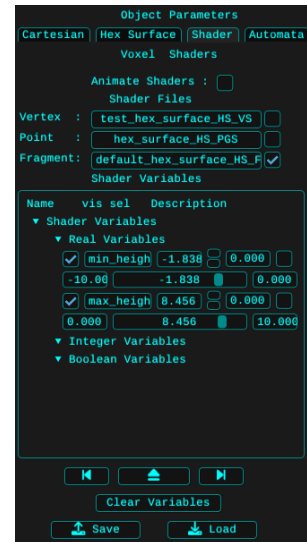


Fig 03.0.8

To see the hexagonal voxels surface as hexagon columns, an existing shader program exists that can be used to do this.

Step hg06 : Select the tab Shader in the parameters panel to bring up the shader widget as in **Fig 02.1.0** of the section **Visualising HCP voxel object**.

Step hg07 : Select the tab Shader in the parameters panel to bring up the shader widget as in **Fig 03.0.7**. Press the vertex shader button labelled “Default_hex_surface_VS” and navigate to the directory Examples/Shader/Hex_surface/ in which the application executable resides and select the file

hex_surface_HS_new_VS.glsl

Repeat this for the geometry shader by selecting the file hex_surface_HS_GS.glsl.

Step hg08: Go to the Shader variables widget and as done with defining a voxel generation real variable, create two variables here. Define one with the name min_height, and the other with max_height. Check all the enable check boxes, and assign for the min_height variable , a value of -10.0, and a step value of 1.0. A variable slider will be visible below these widgets. The left float input widget is the minimum range value of this variable, and the right, its maximum. Enter -10.0 for the minimum, and 0.0 for the maximum. For the max_height variable, enter a value of 10.0, a step value of 1.0, and a slider min and max range value of 0.0 and 10.

This shader widget should appear similar to that displayed in Fig 03.0.8. The voxel display shader program can now be generated and the hexagonal voxel display updated to the functionality as defined by the shader code of the defined shader types.

Step hg09 : Press the generate shader button (middle button with up arrow). The display may go all blank (it should not as this problem has been fixed) due to a programming bug having the voxel size set to zero that has not yet been fixed. Just go back to the voxel generation tab widget and move the voxel scale slider to display the voxels. A view similar to Fig 03.0.9 should now appear in the viewer panel. If the hexagon surface voxels are too small, change the resolution step value to 0.5 and repeat step hg05.

What the user can now do is move the variable sliders to change the min and max values that are displayed as a color range of the voxels. If the user now selects the increment and decrement buttons, the user will find that both variable values will change in the same step. If any of the step check boxes for enabling the step action is disabled, that variable will not change while any that are active will.

If the user so wishes, by pressing the save button, the user can select a directory and file name to save the defined parameters here for later use.

This is a basic demonstration of the use of this shader widget. It is intended that the user can create new or use many a variety of shaders to display voxel surface data. The hexagonal shaped Voxels are actually drawn using the defined geometry shader as only the point data that is observed by enabling the display as points checkbox in the voxel shader widget. It is envisaged that with more development, a whole series of different shader code will be created.

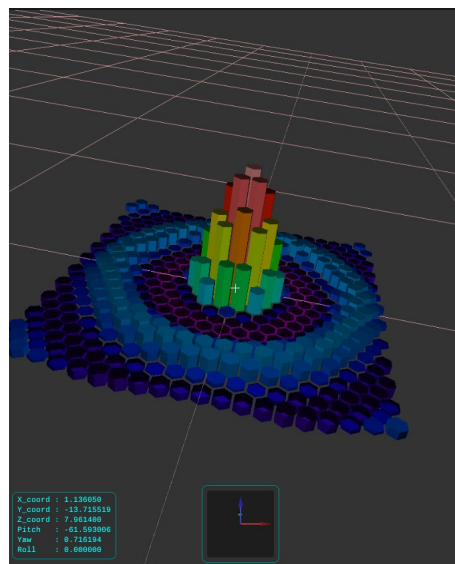


Fig 03.0.9

03.1 Hexagonal Cellular Automata

One purpose of creating the hexagonal spatial data structure and this application is to implement a usage of cellular automata for the purpose of simulation and modelling of real world phenomenon or pure experimental and fractal exploration. With the ability to define any set of simple rules, an infinite realm of possible behaviours can be explored for computer graphics, art or science. The hexagonal voxel 2D spatial data structure has the advantage over conventional cubic voxel modelling in that the centre of every neighbour of any voxel is of equal distance as the neighbours of a cubic voxel sharing one or more vertex points is not.

The hexagonal voxel cellular automata is implemented as a node in the node editor, which needs to be linked to a hexagonal voxel generated scene object. To create a hexagonal cellular automata node, in the node editor where one wishes to place a hexagonal cellular automata node, right click with the right mouse button and release to bring up the floating menu, and navigate through the menu to the menu option

Main Menu→Create Node ...→Hex Surface... → Create Hex Surface Automata Node.

A hexagonal voxel automata node should be created. Make a node editor link from the hexagonal voxel node created above grid output pin on the right of the node to the hexagonal automata input grid pin on the left of the cellular automata node.

Select the hexagonal cellular automata node to bring up a widget as in Fig 03.1.1 in the parameters panel. This widget enables the user to define the hexagonal voxels cellular automata rules that determine the hexagonal voxel values from one iteration step to the next, and from these values and the shader program that is defined by the shader code in the shader widget tab display the hexagonal cellular automata results.

From top to bottom

Automata Rules : The section of this widget in which the user creates and manages the automata rules by which the application will use in a iterative step by step to determine what value to give to all of the individual hex voxels within the area defined in the voxel generation tab widget.

Automata control buttons : These are the automata control buttons to step the automata processing from one iteration to the next, play the automata rules, stop pause and record. As of this time, the record button is not implemented.

Progress bar : An indicator of the progress of the automata process

Clear variables button : Clears all the cellular automata rules variables.

Save : Opens a dialog window to save the cellular automata rules parameters defined in this widget to a formatted text file of suffix AHSR.txt.

Load : Opens a dialogue window for the user to select a cellular automata rules parameters (AHSR.txt) formatted text file to be imported into the application and the values displayed in this widget ready to be used.

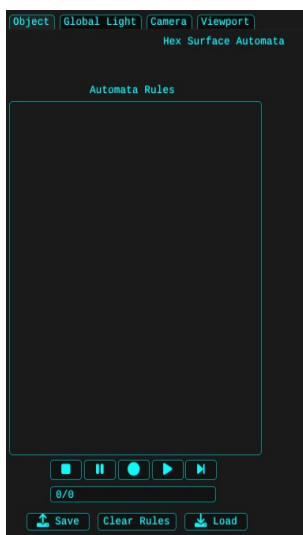


Fig 03.1.1



Fig 03.1.2



Fig 03.1.2

Given here is a simple example.

Step ha01 : Move the mouse into the automata rules section and press the right mouse button to bring up a one option floating menu add rule. An automata Rule should be added to this section of the widget and look like Fig 02.2.2.

The text input widget at the top left of the rule is the name of the rule being applied. Next to this is a button labelled "Neighbour Rules" to bring up a floating widget that defines the rule to be applied. The second line below text "rule" is the value that the voxel will be given if the defined rule is met. The start and stop entry values are the iteration step that the rule will apply over. The Activate check box indicates whether the rule will be applied.

Step ha02 : Left click the mouse over the Neighbour Rules button. An Automata neighbour Rules popup widget should be displayed as in Fig 03.1.2.

The neighbour rules are the conditions that must be met for the cellular automata rule to be valid and the voxel set to the defined rule value. Each hex voxel has six neighbours, and each neighbour of the voxel is identified by the number on the left of this popup. For information of which neighbour these numbers refer to section 4 Cellular Automata in the documentation file Virtual Worlds Voxel. A rule also can be defined for the self value of the voxel itself that is being evaluated.

An automata rule for each neighbour or self voxel is defined in this popup according to the mathematical relationship as given in har01.

$A (<, <=, \text{ignore}) VV (=, !=, <, <=, \text{ignore}) B$ - har01

which states the rule is met for a neighbour if the voxel value (VV) of the neighbour voxel is one of any of the combinations of conditions that is within the brackets such that it falls within the range of value A, and value B. This may be explained better with some examples.

If have a rule har02 from one particular neighbour

$A < VV < B$ - har02

then this rule states that if the neighbours or self voxel value VV is between values A and B, then that rule condition is met for that voxel neighbour.

If have a rule har03 from one particular neighbour

$\text{ignore } VV \leq B$ - har03

then this rule states that if the neighbours or self voxel value VV is less than or equal to value B, then that rule condition is met for that voxel neighbour.

If have a rule har04 from one particular neighbour

$A < VV \text{ ignore}$ - har04

then this rule states that if the neighbours or self voxel value VV is greater than the value A, then that rule condition is met for that voxel neighbour.

So by use of the relationship defined in har01, any automata rule pertaining to the integer value that the voxel neighbour has can be created with a very large set of possible rules. It is this method of defining the cellular automata rule for each voxel neighbour and itself that is represented in this popup widget, with the lower rule being the left side of har01 and the upper rule being the right side of har01.

Note : Virtual worlds currently has the voxel value VV in the range of 1 to 255 that are valid, with the value 0 reserved as an invalid voxel value. So any value of zero for any voxel will not be displayed on screen

With modification, and for different types of data that the voxel value can have, other types of rules can be defined and implemented as desired, but for the purposes of this demonstration, the following simple set of rules will be applied.

Step ha03 : Enter into the lower rule of neighbour 0 the value 0. Change the selection of relationships for the lower rule by selecting the rule drop down selection menu and select the < (less than) rule condition. The less than sign means that any value less than 0 will mean that the rule has not been met. For the upper rule enter 0 as the value that the rule must use, and select the negative sign (-) rule relationship for the upper rule of voxel neighbour 0. The negative sign means ignore the value.

What this rule defined in step ha03 means is that if the voxel value of the neighbour voxel 0 is greater than 0, then the rule for that neighbour is met and the thus valid for this neighbour.

Step ha04 : make sure all other rules have their rule conditions set to an ignore (ie minus) rule.

The cellular automata rule to be applied to all voxels should now appear as in Fig 03.1.2. Press the Accept button for this neighbours rule to be applied for this cellular automata rule, other wise it will be discarded upon exit.

Step ha05 : Going back to the rule definition that was created in **step ha01** and displayed in Fig 03.1.2, enter the value 6 in the rule input entry box, and enter in the stop widget a value of 5. Activate the rule by enabling the active checkbox Fig 03.1.3.

A cellular automata rule has now been defined to operate over the iteration step interval from 0 to 5. The start and stop iteration steps are so that multiple different rules such as defined above can be performed over different periods or iteration steps. This has not been thoroughly tested yet. For the purposes of this demonstration, the user can now perform the cellular automata rule by clicking on the next step button of the control buttons.(ie the right most button with an arrow like decal and a bar at its right)

Do this a couple of times and see if there is a change in the appearance of the hex surface. If not, move the mouse into the viewer panel and press the right mouse button to rotate the sphere until something similar to Fig 03.1.4 is seen.

What this rule does is to assign a value of 6 to the voxels that have their neighbour 0, having a value greater

than the value 0. Thus all the hex voxels in Fig 03.1.4 have a zero neighbour with a value greater than 0. Looking at Fig 03.0.9, Fig 03.1.4 looks to be correct.

Continue pressing the next iteration step control button a few times and at the end, one should end up with something that looks like Fig 03.1.5. To restore the hex surface to its original state, go back to the hex generation tab widget and press the “execute function” to regenerate the data.



Fig 03.1.3

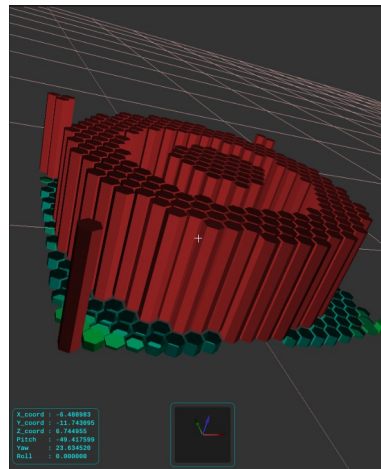


Fig 03.1.4

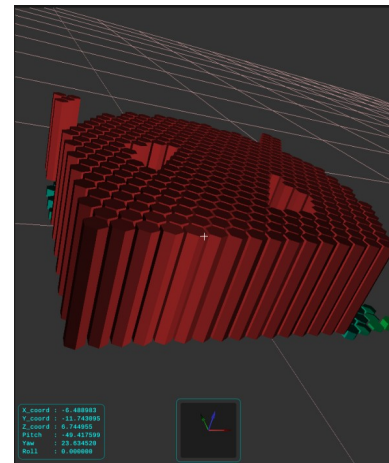


Fig 03.1.5

Now enter the Automata tab once again and reset the automata rules by pressing the stop button in the button controls (ie far left square shaped button) The progress bar should now be reinitialised. Press the play button (the second to last solid arrow button) and the cellular automata rule should now play over a period of five iterations.

If the user so wishes, by pressing the save button, the user can select a directory and file name to save the defined cellular automata rules here for later use.

03.2 Create Hexagonal Voxel Object From Cartesian Point Data

If one has a set of point data in Cartesian coordinates that is generated by a C++ function, or exists within a 3D formatted vertex file (eg a .ply formatted file), then this point data can be converted into a hexagonal voxel object to represent the point cloud data as a surface of voxels. This is achieved by a hexagonal voxel grid of a given origin, size and range attributes that fit within the point data set surface in space being imported such that, If a point does exist within a cell, the data attribute value for that cell within the computer data storage array is given a value to indicate one data point exists within that cell. If more than one point exists within a cell, that value is increased to represent the number of coordinate points that are found to exist within that voxel cell. Thus a resultant hex voxel matrix is formed representing a kind of density of data points within each HCP voxel cell that can be displayed on screen.

An alternative method of generating a hexagonal voxel object from Cartesian point data is that if the Cartesian data has a z value that corresponds to a surface value. Then the resultant voxel value for a voxel is an average of all of the z values that exist within that voxel area. Thus a resultant visualisation of that data can be representative of a surface and not the number of points that exist within a voxel ares or cell.

The following steps describe how to create a hexagonal grid of hexagonal voxel cells from a .ply file.

Step hc01 : Create a Hex surface voxel object in the node editor panel as described in section 3.0 **Generating a hexagonal Voxel object**, and select the voxel node object to display its generation paramaters in the panel as in Fig 03.2.1.

There is a radio button at the top of this widget that selects the method for generating 2D hexagonal voxel surface point cloud data. To generate a hexagonal 2D voxel surface from Cartesian coordinates stored in a .ply file, the radio button with the label Cartesian Coordinate needs to be selected. This will change the widgets that are displayed from those needed to generate a hexagonal voxel surface from a shader program, to those needed to inport a .ply file or from a random generation.

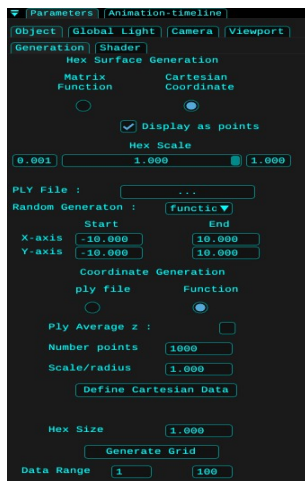


Fig 03.2.1

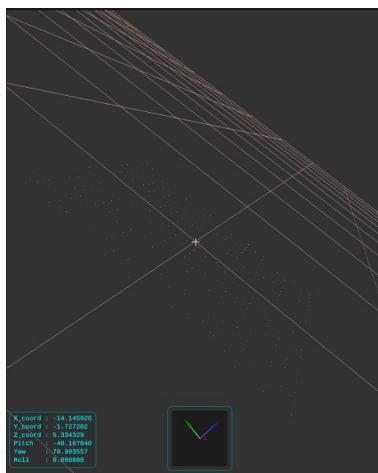


Fig 03.2.2

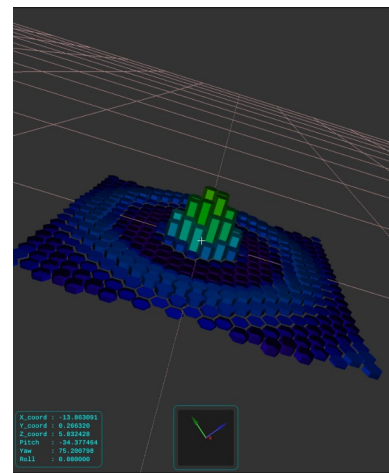


Fig 03.2.3

From top to bottom, the hex surface generation

PLY File : A button to bring up a file dialogue for the user to select a PLY file that contains vertex xyz point data to be used to create a hexagonal voxel grid.

Random Generation : A button drop down list to for the user to select a method of generating a set of random xyz point data to be used to create a hexagonal voxel grid. Available in this demonstration are function, square and circle. Square generates a zero plane square of random points, Circle a zero plane circle of random points, and function generates a surface of random points of the function $z = \sin(r)/r * \text{scale}$ where r is the radius from the origin point (0,0)..

Definition of the x-y area limits of that the hexagonal voxel grid generation is to be confined to. This is automatically updated to fit the Cartesian data that is imported.

Coordinate Generation : Two radio buttons to select the type of point data that is to be used to create the voxel grid. Ply is to use the nominated PLY file defined by the PLY button, or function to use the random data selected in the Random Generation drop down list.

Ply average z : Is to specify if the resultant value that is given to the hexagonal grid voxel cell when a .ply file is used to import Cartesian coordinate data is the number of points within a voxel cell, or is an average of the z values of the point data that are found to be within that voxel cells area domain.

Number Points : Is to specify the number of points that the random function is to generate if the function radio button is selected within the **Coordinate Generation** widget.

Scale/Radius : Is to specify a Scale factor of the imported xyz data points of the imported PLY file, if the PLY generation is selected, or the radius of the sphere of the random generated point data if the function generation is selected within the **Coordinate Generation** widget.

Define Cartesian Data : Is to import or generate the point data to be used to create the voxel matrix. This will also get the boundary of the surface that this data will occupy in 2D space and update the start and end boundary limits. If the user wishes, these boundary limits can be changed.

Hex Size : This is the size of the resultant hexagonal voxel grid cells. The size chosen will determine the level of detail of the point data to be viewed.

Generate Voxels : Pressing this button will create a matrix of hexagonal voxel cells. A value is given to each cell to indicate the number of points that exist within it, or for the case of importing data from a .ply file and activating the Ply average z option. The average z value of all the points within a voxel cell.

Data Range : Widget to select the range of number of points that each voxel cell must have for that voxel cell to be displayed on the screen.

Step vc02 : Select the .ply file selection button to open a file selection dialogue. Navigate to the directory Data that this application executable is located within and select the file named

hex_surface.ply

The button text should now have the text hex_surface displayed.

Step hc03 : If the ply file radio button is not selected in the **Coordinate Generation** widget, then select it to highlight and activate the selection that the unstructured Cartesian data in the ply file hex_surface.ply is to be displayed as a set of structured hexagonal voxel cells. If the Ply Average z checkbox is not activated, then activate this option by selecting the checkbox so as to have it highlighted with a tick mark.

Step hc04 : The number of points has no effect in this case, so can be ignored. The hex_surface.ply data is known to have the x,y,z data values are in the range of ± 10.0 , which should be fine to leave the

Scale/Radius value as 1.0. If it is not 1.0 then change the Scale/Radius value to 1.0.

Step hc05 : Press the “Define Cartesian Data” button to import the hex_surface.ply xy point data set at a scale factor of 1.0. The values of the volume limits to create the voxel matrix should change and be in the range of ± 10 units.

Step hc06 : For a reasonable representation of the data, change the Hex size to 0.5 units. Press the Generate voxels button.

In the viewer display, a default view of white points representing the centers of the hexagonal grid voxel cells should appear in the shape of a flat grid. If necessary, rotate and/or move the camera to get a better view of this data. (Fig 03.2.2).

To obtain an image of the hexagonal voxel cells such that the cells are colored and sized to give an indication of how many data points exist within each cell, select the shader tab (Fig 03.0.7). Select the shader tab to use and define some shader variables as was performed as in the section 3.0 **Generating a Hexagonal voxel object**. The steps to perform the task to define the shader code files and variables will not be described here. What the user can do is to import a parameter file with all these parameters defined.

Step hc07 : Select the “Load” button at the bottom right of the shader widget to bring up the file dialogue to select a shader parameter file to import. Navigate to the directory Examples/Shader/Hex_Surface and select the file hex_surface_columns.twm. This will load the shader parameters needed to display the hexagonal voxels.

Step hc08 : The directory path names to the vertex, point, geometry and fragment shader pathnames will probably be different to that on the user computer. One by one, select the button for each of the vertex and point files to open a file dialogue and navigate to the directory

`<user directory this application exist within>\Example\Shader\Default\Hex_surface\`

and select the hex_surface_HS_new_VS.glsl, and hex_surface_HS_GS.glsl, files that are listed. For the fragment shader program file, select the fragment shader button and navigate to the directory

`<user directory this application exist within>\Resources/Shaders/Default/Hex_surface`

and select the file

`default_hex_surface_HS_FS.glsl`

Step hc09 : Press the Shader update or regeneration button to generate the shader program. No error messages should occur. If an error message(s) are present, it is because an incorrect shader file has been selected or a variable name has been changed or is absent, or not defined to be used. Examine the error message and make the appropriate correction.

Step hc10 : If the display is blank (it should not as this problem has been fixed), it is due to a programming bug having the voxel size set to zero that has not yet been fixed. Just go back to the voxel generation tab widget and move the voxel scale slider to display the voxels. A view similar to Fig 03.2.3 should now appear in the viewer panel.

The colors and height of the individual hexagonal voxel columns give an indication of the average z value of the points that exist within each hexagonal voxel cell. By moving the slider widgets named min_density and max_density, the user can specify the range of data points that are to be displayed. Moving the min_density will mean that only voxels with data points above this value are considered, while moving the max_density slider means that only data points below this value are considered. Red colors are voxels with high number of data points in this specified range, while violet to purple, low number of points in the specified range that exist within the voxel cell.

Step hc11 : Go back to the Generation widget, and deselect the Ply average z checkbox. Repeat steps hc05 and hc06 so as to generate for each hex voxel cell, the number of points that exist within that cell. The display should change and have a similar appearance to that of Fig 03.2.4.

The same procedure for displaying the data of a .ply file can be applied for a function as well. If the user now repeats the above steps hc02 to hc06, but select the radio button “Function”, and from the drop down list of the Random Generation option select square, what should be seen in the display is something similar to Fig 03.2.5. The column heights in this case represent the number of points that lie within each hexagonal voxel cell.

If user were to select the option Circle from the random generation drop down list, change the Scale/Radius value to 5.0 and perform step hc05, the start and end coordinate range should change to ± 5 units. Change the x and y start coordinates to -10.0, and the x and y end coordinates to +10.0. Performing step hc06 should result in something similar to Fig 03.2.6 being displayed. A circle of random points of radius 5 units.

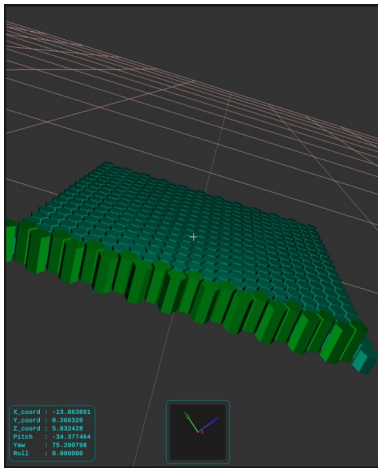


Fig 03.2.4

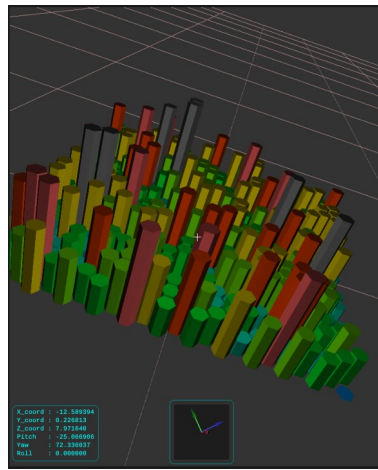


Fig 03.2.5

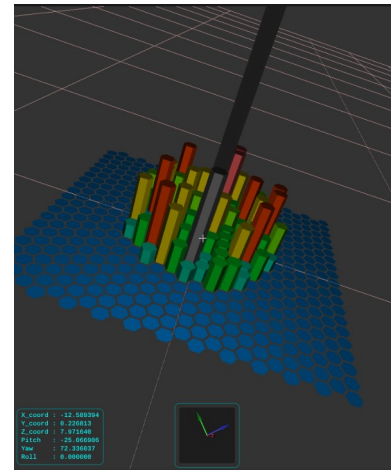


Fig 03.2.6

A provision to allow a function of z coordinates can be also created in code to allow a display similar to Fig 03.2.3 to be viewed rather than just the number of points that exist within each hexagonal voxel cell. Select function from the random generation drop down list, make sure the Ply Average z option is activated and that the scale/radius value is set to something like 5.0. Repeat steps hc05 and hc06 to generate the hexagonal voxel data, and a display similar to Fig 03.2.7 can be obtained after modifying the min_height and max_height of the shader variable sliders. Note that the heights of these columns are low because an average of all of the data points within each cell is calculated. The height of the columns can be changed by changing the scale/radius value and repeating steps hc05 and hc06.

Other possibilities can be applied in code to create other forms of data for computer graphics, simulation and automata. It is up to the user programmer to create such applications.

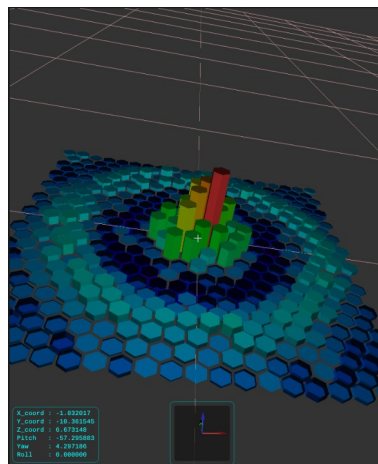


Fig 03.2.7

03.3 Hex Surface Editor

Hex editor applications exist and are widespread allowing users to create 2D hex maps or surfaces. Such a rudimentary basic primitive Hex editor is present within the Virtual Worlds application and is utilised as a node in the node editor. It is intended to extend and expand this Hex editor into perhaps it's own module, or even application beyond what is presented here.

This first incarnation or version of a Hex surface editor is designed to act as a method of hex generation, and can also import saved Hex grid to be edited and saved. It is also designed like any other generator of Hex surface data, to have the hex grid available to be used elsewhere in the application, and in particular, for use in cellular automata.



Fig 03.3.1

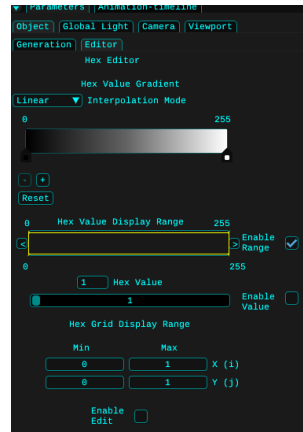


Fig 03.3.3

Section 03.3.1

To open the Hex editor and start creating a Hex grid.

Step ee01 : In a similar manner to creating a Hex surface object, at a location in the node editor where one would like to place a hex surface editor node, with no node selected, click the right mouse button and release it to display a node editor's floating menu. Make the menu selection

Main Menu→ Create Node→ Hex Surface→ Create Hex Surface Editor Node.

A node similar to that displayed in Fig 03.3.1 should be present.

Select the created node and in the parameter panel, a widget similar to that of Fig 03.3.2 will be displayed.

The node Hex surface editor has two tabs, a Hex surface generation tab, and an editor tab. The generation tab specifies the parameters to generate a Hex surface grid (Fig 03.3.2), and the widgets editor tab (Fig 03.3.3) gives control of the display and editing brushes to edit the Hex surface data within the hex grid.

Generation tab from top to bottom.

Display as points : Change the display of the generated data from points to voxels.

Hex Scale : Widget to scale the displayed Hex size when it is selected to not to display the data as points.

Height Scale : Widget to scale the displayed Hex value displayed as a z coordinate. A height scale of zero gives a z coordinate of zero and displays the points and hexagonal representation as a flat 2D surface. Any positive scale value will give a scaled value in the z coordinate axis that represents the value of the hexagonal voxel cell. No negative scales are permitted.

Hex Surface : Button to open a file dialog to select a voxel volume data set to import and edit. Any current selected files to import will have the name of that file displayed in this button

Generated Hex surface limits : Definition of the x-y grid area limits of that the Hex surface generation is to be confined to.

Limits of Hex grid : Button to read the hex surface data file defined and the name displayed in the **Hex Surface** button text to find the limits of the area that the data is defined to be within , and display the x-y limits in the **Generated Hex surface limits** widget.

Import Hex Grid : Button to import the hex surface data file defined, and the name displayed in the **Hex Surface** button. The data imported will have the limits of the volume it is imported to be the same as defined by the hex surface limits defined above.

Min Surface value : The minimum hex value that is to be imported or created. Any hex values less than this will be ignored.

Max Surface value : The maximum hex value that is to be imported or created. Any hex values greater than this will be ignored.

Default Surface value : The hex value that is to be assigned to all hexes upon generation of a new hex volume, or given to all hexes that are outside the minimum and maximum surface range.

Resolution step : The resolution or size of hex grid cells that are to be generated. Note: the hex size of any imported data is determined by the size of hexes defined by the imported data file.

Create New : Generate a new hex surface with the hex value given by the default hex value

Save Volume : Export and save the current hex editor grid data into a file.

Save : Opens a dialog window to save the current hex editor parameters defined in this widget to a formatted text file of suffix .hep.

Load : Opens a dialog window for the user to select a hex editor parameters text file with the suffix .hep to be imported into the application and the values displayed in this widget ready to be used.

Step ee02 : Define the limits of the Hex surface volume to edit, enter -10.0 for each of the start or minimum for the x,y axis, and 10.0 for the end or maximum for each of the x,y axis.

Step ee03 : In the default hex value enter a value of 128. This is the value that every hex will have upon generation.

Step ee04 : In the resolution step widget, enter a value of 0.5 if it is not already set to this value. This defines the resultant generated hex surface grid hex size.

Step ee05 : Click on the Create New button to create a Hex surface volume with the parameters entered in the previous steps.

In the View port display, a Hex surface matrix of similar appearance to that of **Fig 03.3.4** should be present. If necessary, zoom in or out from the central origin of the scene using the middle mouse scroll wheel, and adjust the camera view via clicking on the left mouse button and move the mouse.

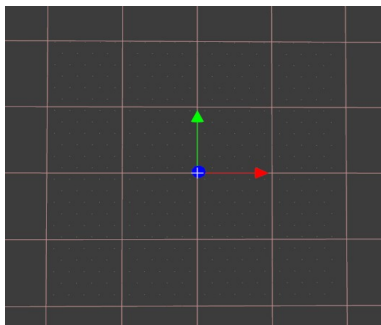


Fig 03.3.4

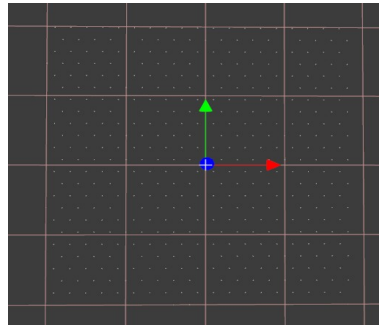


Fig 03.3.5

The points representing the Hex surfaces will look to have a rather dark grey color. To change this

Step ee06 : Select the Editor tab to bring up the Hex surface editor editor widget as displayed in Fig 03.3.3.

From top to bottom.

Hex Value Gradient : This is a widget that defines a gradient of colors to be displayed in the hex editor widget that reflects the values of each hex being displayed. The number 0 to 255 at the top of the color bar gives the hex value that the color in the color bar represents.

New colors can be added to the gradient by moving the mouse cursor into the color bar of the widget and left clicking at the position one wishes to add a new color, or by selecting the + button.

To delete a color, select the color slider widget and select the – button.

To edit a gradient color, select the color slider widget to highlight it, and then select the button with a color of that slider widget (after the + button) to bring up a color picker widget in which to change the gradient color at that color sliders location. The number entry after the color edit button is number that represents the percentage or location in relative terms to the value of 255. I.e a multiplier of 255. eg .5 = $255 \times 0.5 = 128$.

The gradient colors can be displayed in a linear or step fashion. The selection of gradient type can be selected via the pull down menu at the top of this widget.

hex Value display range : This is a widget that defines the range of hex values that are to be displayed.

There is a left or minimum range slider (<) to define the lower hex value that is to be displayed, and a right or maximum slider (>) to define the upper hex value to be displayed. By moving these sliders, the minimum and maximum hex values to be displayed are changed and, those values are displayed below these sliders. The

enable range check box next to this range slider widget enables or disables this widgets function. If not selected, all hexes of any value is displayed.

hex Value : This is a widget that will display all hexes that have the value specified in the editor if the Enable Value check box is activated regardless of whether it falls in the hex value display range or not.

Hex Matrix display range : This is a series of ImGui drag widgets that defines the limits of the hex surface to be displayed in terms of the hex grid i,j coordinates. This allows the user to only have displayed a portion of the hex grid to be viewed and edited.

Enable Edit : Activating this check box widget opens the hex editor hex edit options

To get a better display of the created hex volume

Step ee07 : Go to the hex value gradient widget and move the end color slider widget to the middle of the gradient, until a position is reached where the color of the points in the view port are a white color as in Fig 03.3.5.

Step ee08 : If the user moves the hex value range sliders, either the min that goes above a value of 128, or the max that goes below 128, these points representing the hexes of value 128 will nor be displayed. If these hexes are not displayed, then by entering 128 into the hex value widget entry widget, or sliding the hex value slider unit 128 is displayed, these hexes can be displayed by activating the enable value check box.

Step ee09 : If the user changes the values in the hex grid display range widgets for the min and max x,y or i,j axis, the display of the hex grif will change to display the range of the hex grid range in terms of the hex grid i,j, coordinates.

These widgets give the user the basic tools to analyse and view the hex grid that they are creating or editing.

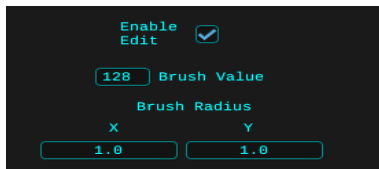


Fig 03.3.6

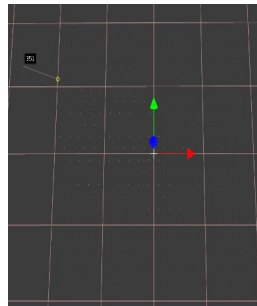


Fig 03.3.7

To start editing this hex matrix activate the Enable Edit checkbox Widget to display the hex editor widgets and activate the hex editing mode in the view port of the hex matrix created in steps **ve02** to **ve05**. Below the Enable Edit check box widget should apper a group of widgets similar to **Fig 02.5.6**. This is a very basic and rudimentary brush tool to edit hexes in the created and displayed hex matrix.

Brush Value : Is the value that the brush will change the hexes value to that it overlays.

Brush Radius : Are the dimensions of the rectangular shape in world coordinates on a x-y axis that define the editing brush size.

If the user now goes to the view port, it should be be seen that there is yellow circle surrounding one of the displayed vertex points and a line drawn with a number at its end. This number is the hex grid vertex index number. By moving the mouse about, the user will see that the circle and number will change. This circle around the vertex data point and its index can be considered as the selected central hex that is to be edited, and depending on the brush dimensions, change the hexes that fit within the grid dimensions of the editing brush. To demonstrate this.

Step ee10 : Go to the hex value gradient and move the slider with the white color to the far right or at the value of 255. The displayed hexes should now be a mid dark grey color.

Step ee11 : Enter in the Brush Value widget a value of 250

Step ee12 : Enter or change in each of the brush radius value ImGui drag widget to a value of 5 by double clicking within each widget and entering 5.

Step ee13 : Go to View port and with moving the mouse, select a hex to edit it and the surrounding hexes to this value of 250 and press the space bar. The selected hex and its neighbours should now appear with a white color indicating they have been changed to a value of 250. **Fig 03.3.7**.

This is the basis of the Hex surface editor operation. By changing the brush value and radius, more hex editing can be done, and by changing the hex value gradient, hexes of different values can be envisioned with colors according to this gradient. The newly created and edited hex grid data can be exported for future use.

To save the Hex surface grid data for future use and editing.

Step ee14 : Select the Generation tab to go back and display the generation tab widget that was instructed to do in **step ee01**. The generation tab widget of **Fig 03.3.2** should be displayed.

Step ee15 : Select the Save Volume button to bring up a dialog to enter a file name to save the Hex surface grid data to. Navigate to a desired directory to save the file to and enter a name to save the hex grid data to. If an existing hex grid data file exists and one wishes to overwrite it, select the file and follow the usual steps as with any file dialog to save the data.

To import an existing Hex surface matrix data file to edit its data.

Step ee15 : In the same hex editor generation widget as specified in **step ee14**, Select the hex Surface button to bring up a dialog to select a file name to import the Hex surface grid data from. Navigate to a desired directory where the file to import exists, and select the file and follow the usual steps as with any file dialog to select the file to import.

Step ee16 : To initialise the limits of the surface data to import, press the “Limits of hex Gid” button widget to have the software read the selected file to extract the volume limits. If desired, the user can then modify these limits to contract or expand the hex surface grid for importing data into and for editing.

Step ee17 : Select the Import hex Grid button widget, and the data that exists within the limits specified will be imported for display and editing as described in steps **ee06** to **ee13**.

To view the Hex surface volume as hex geometry shapes instead of as a point cloud deselect the Display as points check box at the top of the generation tab widget.

Is recommended that all editing is performed with the hex grid displayed as points as it can be difficult to make selections of a specific hex grid cell when displaying the grid as hexagon geometry.

Section 4

Writing the Code for the generation of voxel data using a Compute Shader program

Virtual worlds compute shader code uses a system that requires only certain customised snippets of code to create the data and display graphics without the need to reuse and copy blocks of the same required glsl code in every file. Without too much further detail, a simple format of this snippet code to be in a Virtual Worlds glsl compute,program is given in listing sc01

```
begin_function
<user defined functions and variables>
end_function

begin_expression
<user defined variables and main function>
end_expression
```

Listing sc01

Between the begin_function and end_function block identifiers the functions and variables to use in them are defined.

Between the begin_expression and end_expression block identifiers is the main function code and the variables to use in the main function. The main function block code "main() {" to begin the main function or "}" to end the main function must not be used as this is added by the Virtual Worlds application.

In using the application, the user may define some uniform variables such as "a" in the **Generating a HCP voxel object** section, or the uniform variables "min_height", "max_height" in the **Visualising HCP voxel object** section. These uniform variables are added to the final shader code text string that is compiled by the application, so there is not need to add any additional text like "uniform variable_name" to these section blocks. Doing so will create a compilation error that a variable has been redefined.

The code however must have any uniform variable defined in the application used in the custom shader snippet code to be of exactly the same name. A common compile error that can occur is to use names in the shader code that is different, or is not defined by the user in the application.

Another restriction in writing this custom shader code is that there are certain reserved words for variables that cannot be used or a redefinition of variable names compilation error is generated.

For all of the the shader code the user should not use the version number of the glsl code. eg #version 430 core

The compute shader custom user code has a few restrictions on the names of variables that the user cannot use. The following variable names are reserved and should not be redefined.

```
voxel_size
threshold
min_surface_value
max_surface_value
e_time
frame
origin
matrix_dimension_x
matrix_dimension_y
matrix_dimension_z
min_voxel_value
max_voxel_value
invalid_voxel_value
value
```

The voxel generation shader program will have other internal variable names that are reserved as well as function names depending on which type of data generation is being performed. If the user gets a

compilation error because of a redefinition, a renaming of that variable or function will be needed in the custom shader snippet code.

When a compilation error occurs, the code logger will display the entire shader code program with line numbers and the compile error message to be corrected. The error message should only concern the custom code that was created in a text file by the user. The user should correct the errors, and then attempt to submit that code to be used in the voxel generation or displaying graphics.

Below in listing sc02 is an example of the compute shader code used to generate the sphere of the section **Generating a HCP voxel object**.

```
begin_function

end_function

begin_expression
    if((a*(x*x+y*y+z*z)) > 0){
        value = uint(abs(a*(x*x+y*y+z*z)));
    } else
        value = invalid_voxel_value;
end_expression
```

Listing sc02

The generation of the voxel vertex data is of the form of using the mathematical relationship

$$\text{voxel value} = f(x,y,z) = \text{radius}^2 - (x^2+y^2+z^2) \quad \text{eqvg-01}$$

what the code in listing sc02 does is to calculate the mathematical relationship defined by eq vg-01, and if it greater than zero, assign the voxel at location x,y,z (which is a location of the centre of a HCP voxel) this value, otherwise assign an invalid value to the HCP voxel at that x,y,z location. Without going in to details, voxels with an invalid value are not displayed by the application.

As can be seen from this code, no user defined functions have been made, but there are variables a, x,y,z, value and invalid_voxel_value that are not defined. Even though there are no custom functions or variables defined, the function section with the block code identifiers begin_function and end_function must be present, or an error will occur in attempting to read the file that this custom program snippet exists in.

For the HCP compute voxel generation shader, x,y,z are reserved variable names defined by the core HCP compute shader code that the user cannot access. However the variable a is a uniform variable, and the same variable name defined in the section **Generating a HCP voxel object** that was given in the generation variables widget. This variable a is the radius that the user can change in the application to use to explore or get different results in generating voxel matrix data.

The variable invalid_voxel_value is a uniform variable that is defined by the application, and value is an internal shader variable of type unsigned integer that is used to assign a value from the compute shader program back to the application so as to be stored and used in displaying this generated data. This value must be used in every generation of voxel data.

This is the simplest of examples of how a custom compute shader snippet code can be defined.

Other examples of using this system can be found in the Examples directory in which the executable file of Virtual Worlds exists in.

Creating custom shader programs to display generated voxel data

The vertex, geometry and fragment shaders used to display the generated voxel data are all dynamically generated and managed by the Virtual Worlds application during its execution. Thus a certain knowledge and skill of the glsl programming language will be needed for one to create custom shaders to display voxel data outside the provided shaders.

Virtual Worlds is designed as closely as possible to compile vertex, geometry and fragment shader code written in a similar fashion to C. Include statements of separate external files can be defined so as to better organise and reuse code. With this a design concept, predefined glsl files need to be included with every vertex, geometry and fragment shader code file. A main function must be present as with all glsl to compile correctly and run.

For the creation of a custom shader program, default template shader programs for the vertex, geometry and fragment shaders exist within the directory Assets/Shaders/Default/, which can be copied and saved to a designated directory to be modified. There are include statements within these templates that are to load mandatory predefined text files of glsl code that are essential for the display of voxel point cloud data. A relative path name from the location of the source code should be reflected in this include statement. If a compilation error occurs that these predefined glsl files cannot be found because the path name is incorrect, then the user needs to modify the line(s) with the include statement to give the correct path name to these glsl files, as with any normal C/C++ code.

The code of a template for any shader program is of the structure given in listing CSP-01.

```
#version 450 core

// ----- Shader Reserved Uniforms -----

// #include "shader_basis_code/Universal_reserved_uniforms.glsl"

#include "shader_basis_code/vs_reserved_uniforms.glsl"

// Comment uncomment following as needed
// layout(location = 1) in vec4 color;
// layout(location = 2) in vec3 normal;
// ----- User Defined Uniforms -----

// ----- Application dynamically defined uniforms -----
// Do not delete next line with DDU as application defined uniforms are placed here
// Must exist in every glsl code unless user wishes to manually enter uniforms that
// the application generates.
// #DD#

// ----- Shader Reserved functions -----

// ----- User Defined Functions -----

void main(){
    vertex = vec3(vertex_data.x, vertex_data.y, vertex_data.z);

    gl_Position = vec4(vertex, 1.0f); // required as a min
    // gl_Position = mvpMatrix * vec4(vertex, 1.0f); // required as a min if not using geometry shader program
    value = int(vertex_data.w); // required as a min

    vs_out.varyingColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Listing CSP-01

Within this template are sections below the comments statements with User Defined where the user can add their own variables and/or functions. Other sections are given that that need to be present for the Virtual Worlds application to add code to before compilation begins.

The shader program will have variable names that are reserved as well as function names depending on which type of data generation is being performed. If the user gets a compilation error because of a redefinition, a renaming of that variable or function will be needed in the custom shader snippet code.

For the vertex, geometry, fragment shaders the reserved variable name that should not redefine are

```
vertex;
voxel_value;
value;
modelViewProjectionMatrix
mvpMatrix;
```

```

uCameraPos;// OpenFrameWorks
uCamera_forward;
uCamera_up;
uCamera_right;

light_color;
lighting_direction;
ambience
specular_strength
lighting_intensity
camera_lighting_type
use_camera_lighting
lighting_camera_offset
t_frame
uniform float voxSize;
voxel_matrix_dimension;
voxel_origin;
voxel_hcp_z_increment;
voxel_min_surface_display_value;
voxel_max_surface_display_value;
vEnabledFaces;
flat out int gEnabledFaces;
VertexData
vs_out;
fvaryingColor;
raw_color;

```

If any of these are used in a custom vertex, geometry or fragment shader, a compilation error will occur.

These glsl code text files are created and modified using an external text editor of the users choice as there is no internal text editor that the virtual worlds application uses. Once a coded text file is created, the user then selects this shader file for the appropriate shader type under the shader tab of the selected application object that this shader program is to be used on. (**Fig 02.1.0, Fig 02.1.1**) To compile the custom shader program, click the checkbox on the right to have the box deselected (ie no tick marked) to indicate that the use of a the designated custom shader program is to be used, and then click the update button. (middle button with up arrow above the "clear variables" button after the list of shader variables.

The user can create uniform variables that the user can use and interactively change in the shader program by creating a shader variable of a particular datatype and name in the shader variables section of the object shader tab. The user can use this variable in their code of the same name and datatype to interact with the display of graphical data.

If the compilation was successful, (even if the display is not due to logical or other errors) then a message will be given in the application log. If not, then a message will be displayed stating this, and a listing of the resultant shader code and error message given in the code log panel. When errors occur or modifications need to be made, the user can make these changes in the text editor and recompile the shader code. The application will automatically discard the old shader program and define a new one to be used.

With knowledge and skill, the user can create a set of shader programs to display the data in any way they wish, with interactivity of changing variables in the shader variable section of the object file definition.

Examples

A series of examples to give an illustration of some of the capabilities of the Virtual Worlds application is given in the directory Examples. These examples are defined formatted text files of different types that are to be loaded into the appropriate sections of the application.

This directory is divided into

Automata : Simple cellular Automata rules

Generation : Examples of surfaces and volumes that can be generated

Shader : location of the default shaders that the application requires to function and examples of some simple custom shader programs that the user can load and compile, and then copy and modify

Each of these directories has a subdirectory related to the object type that this application currently has as modules that can be generated and viewed.

Also within the Examples directory are formatted files that are fully configured to display a geometry that the user can load into the application and then generate the objects, shaders and run any defined cellular automata rules.

Use the included user guide text as a guide to make changes, experiment or export geometry.