Virtual Worlds User Guide

Introduction :

The Virtual Worlds application as it is implemented as of May 2022 is not in intended to be a fully functional production ready application at this time. It is, at this time, intended to be a functioning prototype like application to 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 center and that of every neighbour around it is not the same. However, a 3D space that does have a structure where the center 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 dodecahedral 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.

Application Overview

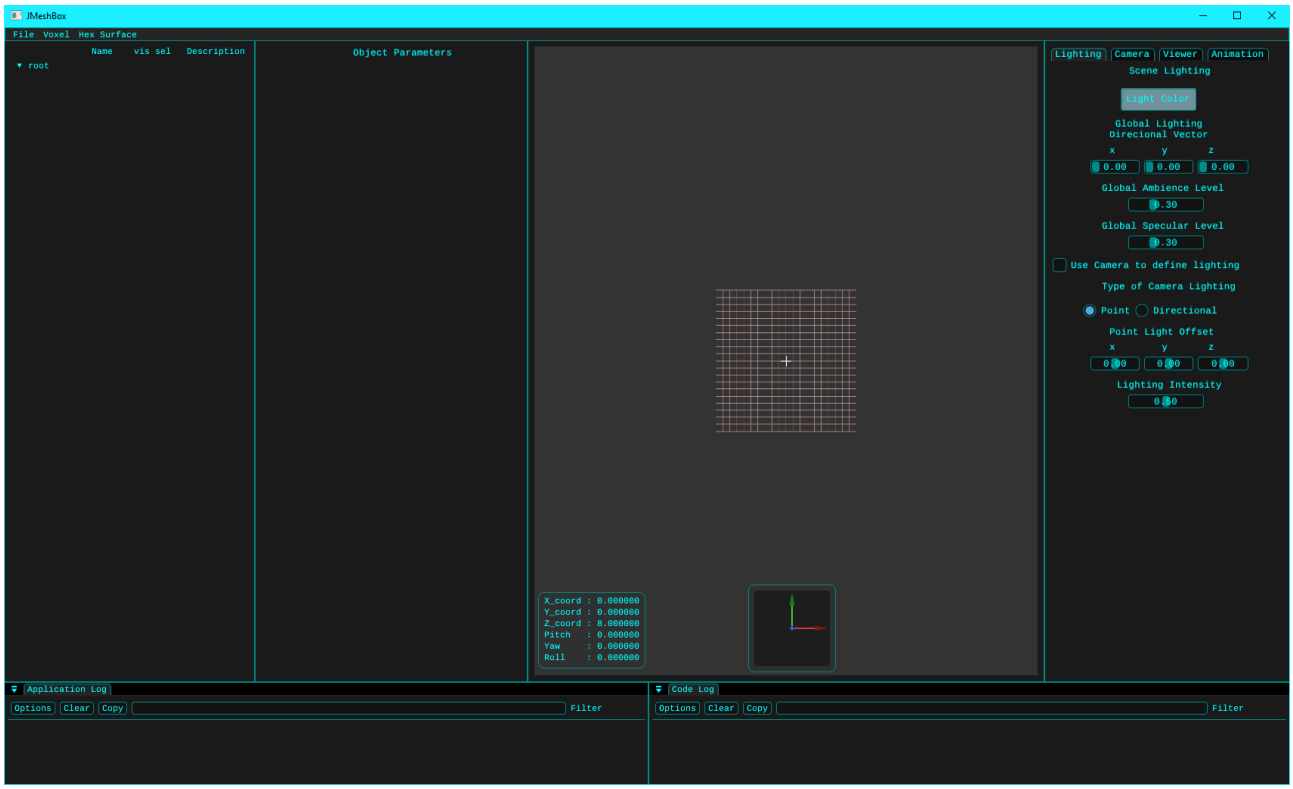


Fig 01

The Virtual Worlds application executable and all required files is located in the File Directory Build/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 as given in Fig 01 should appear. From left to right are four upper panels that are in order, the scene out liner panel, the parameters panel, the scene Viewer panel and scene properties panel. The two lower panels are the application logging and shader logging panels.

Outliner panel : Is where all of the entity objects in the scene are created, deleted, imported, or saved, and have a tree node displayed for the user to select for actions to be taken on that scene entity object.

Parameters Panel : Is where the parameter and variables data is displayed on the current selected scene entity object that is designated in the outliner panel. The user through the widgets displayed in this panel can make changes to the parameters and variables displayed in this panel , and then generate data to be displayed in the viewer panel.

Viewer panel : 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 initilal x-y axis reference grid.

Properties Panel : Is a series of scene properties tab widgets where the user can change various render and camera parameters. An animation widget is present where the user can do basic animations of the objects in the scene in regards to the generation, opengl shaders and cellular automata parameters set for the entity objects in the scene.

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.

All these panels are dock-able, but it when undocking any of these panels, the ImGui system may interfere with the communication of widgets or user interaction between them. If any problems encountered, just redock to overcome.

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 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 the section writing a compute shader program.

To generate a HCP voxel object, the user must first create a HCP voxel object in the scene to be displayed. This is done in the outliner panel. Move the mouse into the area on the computer screen that defines the outliner panel into a empty space and click on the right mouse button. The following floating menu should appear as in Fig 02a. All entity objects must exist within an object group, so select the menu option Add object group. A default object group will now appear in the outliner panel. Select that group with the left mouse button or hovering over it to highlight it, and then press the right mouse button to bring up the floating menu. Move the mouse over and highlight the option Add Object, as in Fig 02b and select the sub menu option HCP Voxel. With the left mouse button. A new tree node item should appear as a child node of the group similar to Fig 02c

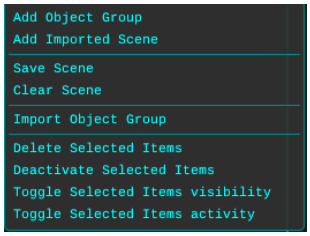


Fig 02a

Fig 02b

Fig 02c

Now move the mouse over this newly created tree node, highlight it, and select it using the left mouse button. In the parameters panel a series of widgets should appear as in fig 03a. The widgets displayed in the generation tab define the parameters required to generate the HCP voxel data. From top to bottom

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

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 relevent to the OpenGL compute shader process where the number indicates the number of threads running in parralel is used in the calculation. Generaly 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.

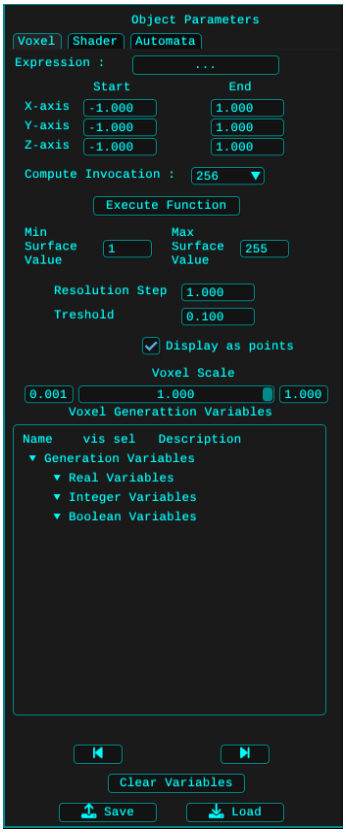
Min/Max surface values : These are the minimum and maximum permissable values that the voxels can posses to be designated as a valid voxel to be stored in computer memory and displayed on screen. At the time of this doumentation, this should be left as the defaults as any other value may not work Something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

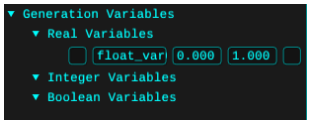
Resolution step : Basicly, this is the spacing between each HCP voxel.

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 that will be explained for the shader tab widget.

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





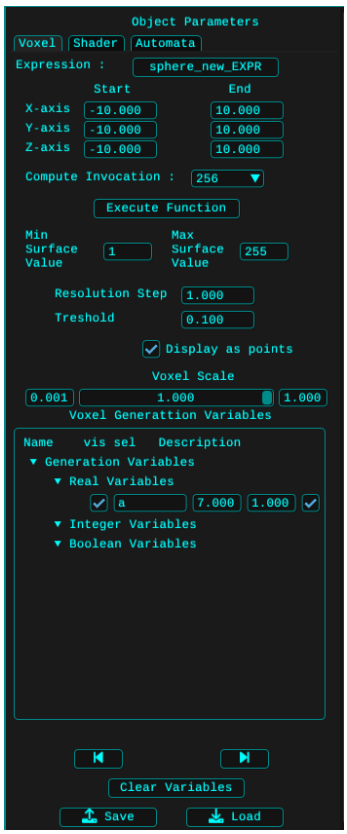


Fig 3a Fig 3b Fig 3c

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 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 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 dialog.

Step vg02 :Navigate to the Examples/Generation/HCP_Voxels/ directory 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 3b) is the same for all types of shader variables. From left to right.

Checkbox to indicate if the variable is defined and to be used in the shader program.

Select and left mouse click to enable this variable to be used in the voxel generation compute shader.

Name of the variable that is defined in the voxel generation compute shader code.

Select the text input widget and change the name to a.

The initial or set value that this variable is defined to be assigned to.

Select the floating number input widget and type in the value 7.0

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.

The default is 1.0 and this can be left as it is.

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 steps operations.

Select and set this widget to enable the step variable to be used.

The voxel generation panel should now look like that of Fig 3c.

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

Step vg05 : Press the button labelled “Execute Function”

This will generate a HCP voxel point cloud of data, and in the viewer panel to the right of this voxel generation widget will be displayed some points that represent the centers of a the HCP Trapezo-rhombic dodecahedral shaped Voxels. These points are the centers of the equally spaced 3D hexagonal close packed spheres of radius.

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. In the properties pane immediately to the right of the viewer panel, the scene lighting tab should be visible. If not select it to bring up the options for the lighting of the scene. A check box labelled “Use Camera to define lighting” should be visible. Select it to enable a light source to be defined from the camera location in the scene. If the camera is too close to or within the generated voxel shape, move the camera away from the object by selecting the number 2 or down arrow on the number keypad. (Note: Number lock must be disabled) Keep pressing this key until the scene in the viewer panel looks similar to Fig 4a.

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 4b.

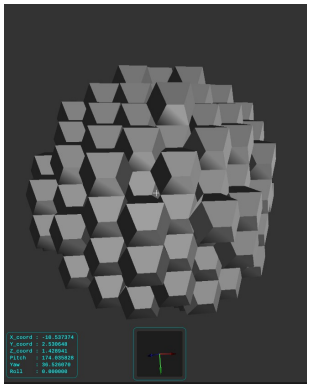


Fig 4a

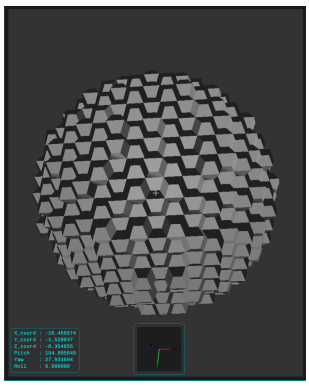


Fig 4b

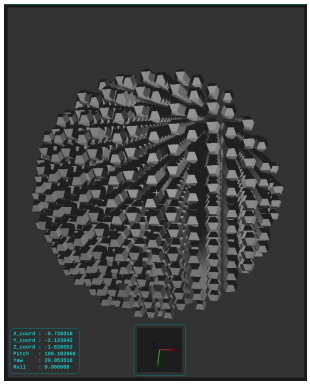


Fig 4c

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. I 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 bottom right code log.

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 4c) 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.

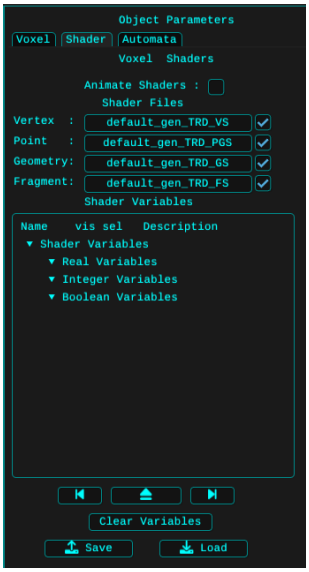


Fig 5a

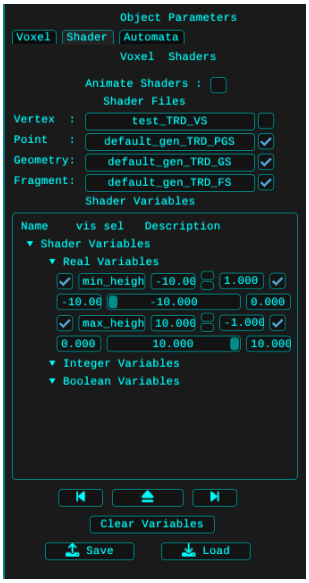


Fig 5b

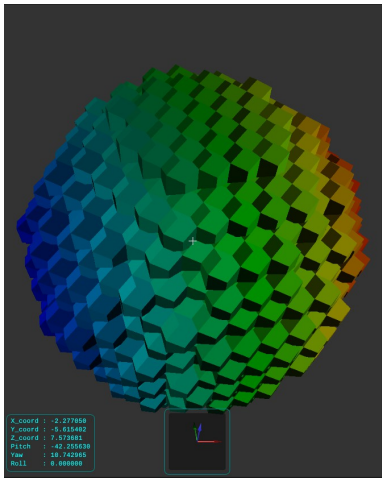


Fig 5c

Visualising HCP voxel object

Any HCP Voxel generated object being displayed on screen utilises an OpenGL shader program. The points and HCP Trapezo-rhombic dodecahedral shaped Voxels are dsiplayed on screen using a default shader program. Both of these can be substituded 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 5a.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 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

Animate Shaders : A checkbox to enable shaders for this voxel object to be used in the animation feature of this application. Ignore this since animations will not be discussed at all.

Vertex, point, geometry and fragment shaders : A set of buttons that are used just as the Expression button was in the Voxel generation tab widget. To select a user defined shader code file to generate a shader program. Next to each button is a check box. If the check box is enabled, the defined default shader code will be used. If unchecked, the selected user defined shader code that is displayed as text in the button is used. The default set up is to display the default shader code program code

Shader uniform Variables : This operates exactly as described for the compute shader variables 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 button between the increment and decrement buttons. 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, bottom right of the application.

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 button 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 test_TRD_VS.glsl. After doing this, uncheck the checkbox next to this button to select this glsl code file to be used in the shader program compilation.

Step vv02 : 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 as in fig 5b. 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). The display may go all blank 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 5c 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 fin 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.

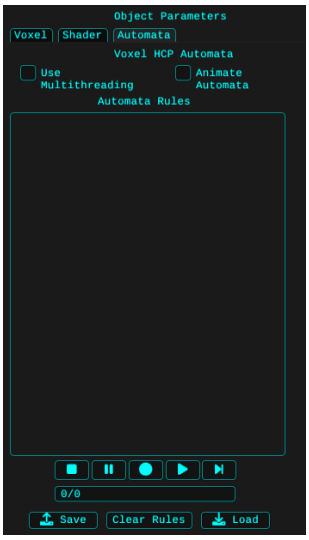
If the user so wishes, by pressing the save button, the user can select a directory and file name to save the defined parameters here for later use.

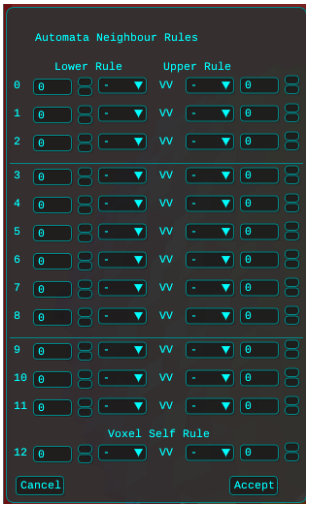
This is a basic demonstration of the use of this shader widget. It is intended that the user can create new or use many a variety of shaders to display voxel volume data. The HCP Trapezo-rhombic dodecahedral shaped Voxels are actually drawn using the default geometry shader as only the point data that 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.

HCP voxel Cellular Automata

One purpose of creating the HCP spacial data structure and this application was 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 center 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.

From the voxel parameters panel, select the Automata tab to bring up a widget as in fig 6a.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.





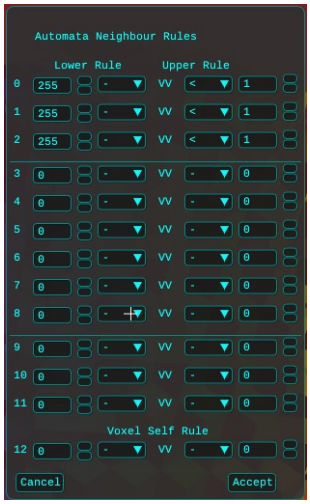


Fig 06a Fig 06b Fig 06c Fig 06d

From top to bottom

Use Multithreading : A checkbox to enable the processing of the cellular automata rules to be performed as a threaded process. Unfortunately, at this time, this is unable to be performed as the migration of code from Qt to ImGui and using Visual Studio 2022 has lead to linking errors that could not be reconciled.

Animate Shaders : A checkbox to enable the cellular automata rules for this voxel object to be used in the animation feature of this application. Ignore this since animations will not be discussed at all.

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 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 06b.

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 06c.

The neighbour rules are the conditions that must be met for the cellula 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 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 ar01.

A (<,<=,ignore) VV (=, !=, < ,<=, ignore) B - ar01

which states the 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 - 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

ignore VV <= B - 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 ignore - 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 06d. 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 06b, enter in the stop widget a value of 5, and activate the rule by enabling the active checkbox fig 07a.

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 panal and press the right mouse button to rotate the sphere until something similar to fig 07b 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 07c. 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 07a

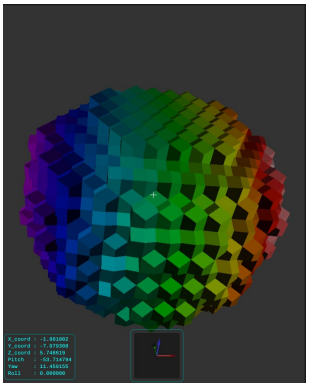


Fig 07b

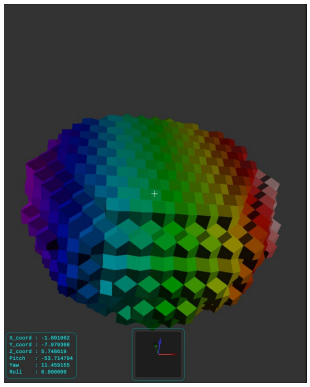


Fig07c

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.

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.

Writing the Code for the Compute and OpenGL Shaders

Creating the code for the Virtual Worlds compute and OpenGl shaders is identical, except that the compute shader code is for generating data, while the OpenGL shader code is to display graphics.

Virtual worlds 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, vertex, geometry or fragment shader 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.

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

vertex;

voxel_value;

value;

mvpMatrix;

camera_loc;

camera_front_vector;

camera_up_vector;

camera_right_vector;

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.

The way that the shader program is put together and compiled is that there are default shader code files for certain sections of the shader program, and these are combined with the custom code that has been created in the above format as a string of text to be passed to be compiled by the application.

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*a-(x*x+y*y+z*z)) > 0){
      value = uint(abs(a*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 eqvg-01, and if it greater than zero, assign the voxel at location x,y,z (which is a location of the center 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, of 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 data.

This but the simplest of examples of how a custom compute shader snippet code can be defined. All other custom vertex, geometry and fragment shader code is done in the same way.

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