Virtual Worlds Documentation

Voxel Data Storage

The voxel data is design to be stored as a data type that reflects the activity of a voxel. That is, to have a value of one to designate that the voxel at a particular location is active or on, or it is inactive and off. Such a data type that best reflects this is thus of the boolean kind, where a bit value within an integer type can store the bit settings of active or inactive. Integer numbers generally vary between one, two and four bytes in memory size, with each byte able to store 8 bits of information. To best conserve memory it is chosen that the best default integer size to use is the smallest being a one byte, eight bit integer.

A one byte integer thus is able to store eight voxel activity states. Each activity state can reflect the center of a voxel location by having a reference to the value of an origin, and a definition of what coordinate the byte and internal bit data refer to. Voxel data represents a data set of an evenly spaced tessellated geometric shape forming a grid or matrix. Thus the first bit of the first byte of an array of bytes can represent the origin point from which all other bits of every byte can have its coordinate calculated given the spacing between each tessellated voxel center.

Voxel grids or matrix structures represent 2D or 3D structures and thus the whole voxel data storage is an array of type integer. The array can by convenience be multidimensional, but it is more efficient to represent the voxel storage as a single one dimension array as by knowing the dimensions of the matrix, the spacing between each voxel center (size of voxel) and the origin that the first bit of the first byte represents, the coordinate of any bit within the array can be found, and the data written or retrieved.

The reason for choosing a one dimensional array to store the voxel matrix data is to enable the c++ vector data storage type to be used so as to have a flexible dynamic run time ability to have voxel arrays of any size available to be created and not have a large fixed array structure which has allocated memory that is not used.

Thus a class of data storage can be designed with these requirements as such in listing 1 Data types will be given later

listing 1 ::

```
class voxel_object_data_class{
public:
        voxel_size;
        matrix_origin;
        matrix_dimension;

        vector <8bit integer> voxel_matrix_data;

private:
}
```

This form of data storage over other documented uses of octrees and OpenVDB was simply because it was deemed unnecessary to do so, and that there was no advantage since the purpose of this app was not to require data to be directly linked to any voxel.

Voxels are generally given and most widely as a cubic shape as the cubic shape corresponds to the Cartesian coordinate system that much of computer graphics and mathematics uses. The cubic voxel shape suffers in that all the neighbours around any voxel do not have equidistant centers from each other.The neighbours not on the x, y or z axis have a distance of $\sqrt{2}$ voxel_size from the central voxel and there are up to 26 neighbours per voxel. E

However there is a second tessellated geometric shape that is better suited for 3D, which uses a spacing of hexagonal close packing of equal spheres. The geometrical tessellated shape of the **Trapezo-rhombic dodecahedron** can be used in the same manner as a 2D tessellated hexagon. As with the hexagon shape, the cartesian coordinates of the x centers of alternate hexagons in the y direction are offset from that of the hexagon on the origin x axis. Like wise it is the same in both x and y cordinates of alternative layers from the orgin x-y plane, and also a requirement to have the **Trapezo-rhombic dodecahedron** shape rotated 30

degrees to form the tessellation.

The trapezo-rhombic dodecahedron is the prefered method of tessellation and representation of voxel data as there are only 12 neighbours surrounding each voxel and each neighbour is equidistant from every other. The matrix to store the voxel is no different from that of the cubic cartesian in any way except the calculation of the center coordinate of each voxel to represent on the computer screen.

Thus there is a need to distinguish what type of voxel the voxel data represents which is done through a c++ class enumeration data type

```
enum class voxel_type_enum {cartesian_voxel, hcp_voxel};
```

which is added to the class in listing 1 as in listing 2:

listing 2 ::

```
class voxel_object_data_class{
public:
        voxel_size;
        matrix_origin;
        matrix_dimension;
        voxel_type_enum voxel_data_type;

        vector <8bit integer> voxel_matrix_data;

private:
}
```

In this application Qt is used to create the GUI and since it has a large number of predefined classes that can be ported to and from other C++ utilisations the Qt classes of QVector and Qvector3D were used. These could have versions to do the same functionality written or got from elsewhere, but it was not required at this time to do so. Implementing the Qt classes of QVector and Qvector3D within this class gives the voxel data structure as in listing 3:

listing 3 ::

```
typedef unsigned char  voxel_bit_type;
#define VOXEL_BIT_NUMBER 8;

class voxel_object_data_class{
public:
        float                   voxel_size;
        Qvoxel3D                matrix_origin;
        int_vector              matrix_dimension;
        voxel_type_enum         voxel_data_type;

        QVector <voxel_bit_type> voxel_matrix_data;

private:
}
```

Qt does not have a integer coordinate class defined so a makeshift vector 3 integer class defined as

```
struct QVector_int_struct_type {
        int x=0,y=0,z=0;
};
```

with

```
typedef QVector_int_struct_type int_vector;
```

is used.

The 8bit integer is defined as an unsigned char, but to add flexibility to at any stage use an integer of any

type the typedef declaration

```
typedef unsigned char  voxel_bit_type;
```

is used.

Also it is anticipated that the number of bits stored within the data type needs to also be defined, a

```
#define VOXEL_BIT_NUMBER 8;
```

is added.

Listing 3 gives the basis and as seen below almost everything that is needed to be defined for the voxel data storage class.

The C++ file that defines this class and all its components is voxel_data_storage.h

Data representation.

With this data class defined, what remains before any methods or functions are written is to define what coordinate the bytes and bit of each byte correspond to. It was chosen for a 2D grid that the bit of each byte would be the y axis coordinate as each x coordinate can be added more easily as a byte with an n bit integer representing the y coordinate. Also it was easier to envisage the data storage as boxes of bits where the bit in each byte represents a sequence of y coordinates than it was as a sequence of x coordinates.

Thus for a voxel 2D grid array of x-y dimension, n,m the number of bytes required to store the voxel data would be the n x m/VOXEL_BIT_NUMBER;

Adding a the z dimension L the number of bytes required to store the voxel data would be the n x m/VOXEL_BIT_NUMBER x L;

Given that each floating point 3D coordinate in computer memory takes up four bytes, the above data storage method can be a fraction for large numbers of voxel data stored as floating or 4 byte integer arrays.

A 64^3 grid would occupy ~ 32,768 bytes as opposed to 3,145,728. or about 1%

Allocating and initialising memory data.

Before any voxel matrix can be created, the voxel matrix dimensions must be given. For a 3D matrix the dimensions are given in the class of listing 3 as

```
matrix_dimension.x, matrix_dimension.y, matrix_dimension.z
```

In this voxel data storage class, to avoid confusion and having the incorrect physical memory assigned or accessed, the matrix_dimension variable must have the y component representing the actual physical computer memory represented. That is if a matrix grid is defined by the user or loaded into memory has ny coordinate components, then the physical memory has a physical y dimension of integer ny/VOXEL_BIT_NUMBER

This is often defined in the C++ code as corrected_ydim, corrected_iY or similar.

So the number of bytes that need to be allocated within the dynamic QVector array data container is

```
matrix_dimension.x = number x coordinate components
matrix_dimension.y = number y coordinate components/VOXEL_BIT_NUMBER;
matrix_dimension.z = number z coordinate components
```

and allocate

```
matrix_dimension.x*matrix_dimension.y*matrix_dimension.z   DS-01
```

bytes of memory.

The number of voxel coordinate components for each coordinate type is calculated by dividing the interval between coordinate axis limits and a defined voxel size. Ie

```
number x coordinate components = (x_max – x_min)/ voxel_size
```

number y coordinate components = (y_max – y_min)/ voxel_size
number z coordinate components = (z_max – z_min)/ voxel_size

Indexing of voxel coordinate .

To find the integer byte and bit location that represents a voxel at a certain coordinate, the next additional piece of data required is a coordinate origin reference that the first bit of the first byte represents. With this data and the voxel_size value and the dimensions of the voxel matrix data, the voxel data value at a specific location can be stored or retrieved.

Without the compressed y axis data using bit to represent a y location, the coordinate of the ith index_voxel_matrix_data element is given by first finding an index value for each of the x,y,z components as if the data was stored in a three dimensional array. Have these index values be iX, iY, and iZ.

```
=>          iX = (int) (coordinate.x()-matrix_origin.x())/voxel_size;
            iY = (int) (coordinate.y()-matrix_origin.y())/voxel_size;
            iZ = (int) (coordinate.z()-matrix_origin.z())/voxel_size;
```

and the matrix index can thus be calculated as

```
i =  iX*matrix_dimension.y*matrix_dimension.z + iY*matrix_dimension.z + iZ;
```
DS-02

But since the y matrix dimension is expressed as bits an not bytes, a correction needs to be made to the iY value to reflect this compression as

```
corrected_iY = iY / VOXEL_BIT_NUMBER;
```
DS-03

and **DS-02** becomes

```
i =  iX*matrix_dimension.y*matrix_dimension.z + corrected_iY*matrix_dimension.z + iZ;
```
DS-04

but this is not enough. The index of the voxel data matrix is found by DS-04, but not the bit location. This is simply done by taking the mod of the iY/ VOXEL_BIT_NUMBER value or

bit location = iY % VOXEL_BIT_NUMBER; **DS-05**

So two bits of information need to be retrieved to store/retrieve data from the voxel data matrix. The index number of the matrix integer value, and the bit location within the integer value.To store this information a data structure as follows is defined

```
struct voxel_matrix_index_struct_type {
       int             matrix_index;
       voxel_bit_type j_bit_index;
};
```

and the value of DS-04 is assigned to `matrix_index` and DS-05 is assigned to j_bit_index and thus the state of the voxel at location x,y,z can be assigned or retrieved by accessing the j_bit_index of the matrix_index of the voxel_matrix_data QVector array.

Class methods or functions

The specific requirements of functions or methods are to

1: allocate, clear, and delete memory allocations.
2:: get the voxel data index and bit location for a specific x,y,z location and perform voxel status assignments or retrieve voxel activity status

The principles of the voxel data storage class are given above to achieve these requirements. Specifics of the coding of performing these tasks are left to be described within the class itself in the file voxel_data_storage.h which is located in the Voxel_function_module of the Virtual_Worlds_Voxel project of the Virtual_worlds_suite project.

Voxel Cloud Object

The voxel cloud object is the main data storage construct of the virtual worlds voxel project.The design is based upon the principle of a enabling a hierarchical data structure of parent-children tree structure where the children are sub objects that make up and constitute a larger voxel object. This hierarchical structure allows and has as its core recursive routines to perform all the data management and storage retrieval of data to display as vertex data within an OpenGL, glsl shader graphics environment.

The voxel cloud object class is defined in the C++ file vw_voxel_cloud_object.h of the Virtual_Worlds_Voxel project

The recursion routines greatly simplifies and creates efficient code to perform data related tasks and manage the data. The voxel Cloud object stores as voxel data a voxel data storage object, and in addition when required, a QT QopenGLBuffer object class to store the vertex coordinate data of the center of each voxel to be displayed on the screen. In addition to the vertex data, provisions was made to also store color data, but for some unknown reason QT seems to have some issue with using and displaying more than one QopenGLBuffer object at a time. This is not a hinderence since it was decided and found that that the color data could be defined better within the GLSL shaders as they allow parallelisation of processing to occur.

Thus the voxel cloud object as a first basis a parent pointer to a parent Voxel Cloaud object, and an array of pointers to point to each child that the voxel cloud objet has created. The voxel cloud voxel data is a reference to the voxel storage data class defines in section Voxel data storage. In addition to the data, an id value and name to more easily identify and locate a certain voxel cloud object is included. This as a basis for the voxel cloud object class and is defined as

Listing VCO_1

```
Class vw_voxel_cloud_object_class{
public:
        vw_voxel_cloud_object_class* parent;

        int                                     id;
        QString                                 name;

        voxel_object_data_class                 voxel_object_data;

        QOpenGLBuffer                           ogl_vertex_buffer; // Must use
        QVector <vw_voxel_cloud_object_class*> children;

private:
}
```

Based upon the active voxels stored within this data structure,only those voxels that are active within the voxel_object_data structure have vertices assigned to the ogl_vertex_buffer so as use memory most efficiently as possible. This is also because Qt 5.x is 32 bit only and the QVector can thus has a maximum limit to hold of 2 gig of data which limits the number of 3D floating point numbers to about 390^3, or ~ 59 million points.

The voxel data stored within the voxel_object_data class object are not designed to do go any data transformations or manipulation such as scale, rotations or translations within the voxel cloud object. Any operations voxels under go are designed to be performed as part of functions that are external to the voxel cloud object class, but reference one or more voxel cloud objects to perform operations such as Boolean, addition subtraction that act on the voxel object data matrix arrays, and not on the vertices or coordinate data each voxel is located at. Such voxel operations produce a new voxel object data object or modify the states, or size of existing voxel object data matrices. In other words, the voxel object data is a fixed matrix in space defined at an origin of particular voxel size.

Creating the OGL vertex buffer data

The voxel data is visualised using OpengGL and GLSL shader programs. The method used to visualise the active voxels is to read the stored active voxel object data into a QT QOpenGLBuffer class object, and then use the QT opengl classes to copy this data into a GLSL vertex shader program.

To do this a QVertex array to store the vertex data needs to be created that has the vertices representing all the active voxels stored within it. Attempts to read one voxel vertex coordinate at a time into the QopenGLBuffer failed and was always empty no matter what efforts were made to try and get this to work. So a additional QVector data class object of a vertex data structure needs to be defined within the voxel cloud object class.
This was defined as

```
Qvector<vw_cloud_vertex_class *> vertices;
```

and ts added to listing VCO_1 to obtain

Listing VCO_2

```
Class vw_voxel_cloud_object_class{
public:
        vw_voxel_cloud_object_class* parent;

        int                                     id;
        QString                                 name;
        QVector <vw_cloud_vertex_class *>       vertices;

        voxel_object_data_class                 voxel_object_data;

        QOpenGLBuffer                           ogl_vertex_buffer; // Must use
        QVector <vw_voxel_cloud_object_class*> children;

private:
}
```

The vw_cloud_vertex_class is the vertex class used through out the virtual worlds projects and stores the vertex coordinate data and perform operations that can be performed on individual vertex data. It may therefore mean that due to the limitations of the QVector class, the number of vertices will not be of the magnitude to store all of the active voxel states if the voxel matrix is large.

Defining the voxel coordinate data values of the centers of each active voxel is not difficult to do and is performed within for loops iterating through the dimensions of the x, y and z dimensions and

1: Calculating the index and bit location that each iteration corresponds to.
2: Accessing the stored voxel object data and determining the status of the bit being accessed.
3: For each active voxel location, add a vertex coordinate to the vertices QVector array according which type of voxel (cubic or hcp) is stored

This is performed within the function define_vbo_vertices of the vw_voxel_cloud_class object, and the psedo code for this function being

```
voxel index = 0;

for(iX=0;iX<voxel_matrix x dimnension && number voxels < max permissible number voxels ;iX++){
   for(iY=0;iY<voxel_matrix y dimnension && number voxels < max permissible number voxels ;iY++){
      for(iZ=0;iZ<voxel_matrix x dimnension && number voxels < max permissible number voxels ;iZ++){

        for(bit location = 0; bit location < number bits for integer type; bit location ++){
                get voxel data value of the voxel matrix data array at location  voxel index
                if(bit value == active){
                    if(voxel type == cubic)
                        define vertex x,y,z value for cubic voxel center
                     else
                        define vertex x,y,z value for hcp voxel center
                }

                if(number voxels < max permissible number voxels){
                        add vertex data to the QVector vertices.
                }
         }
        voxel index ++;
        }
```

```
  }
}
```

The order of the vertices stored does not matter.

Now that the voxel data has been decompressed from bits to vertex data, it then further needs to be copied into Qt QOpenGLBuffer ogl_vertex_buffer object so as to able to be imported into an OpenGL GLSL shader program. Qt requires that there is an “OpenGL context” available and open to be able to do this, otherwise attempting to copy data into this buffer object will fail. Thus the copying of data into this buffer can only be performed within a Qt QOpenGLWidget class object and that the ogl_vertex_buffer needs to be bound before any function is called to copy data into it, and that it is released after the copying of data is complete.

The class function to perform this copying is the copy_vertices_to_OGL_vertex_buffer function. Once done, until there is a change in the voxel data and hence vertex data, there is no need to recreate the vertex data or copy it each time a new openGL graphics frame is to be displayed as all examples and code that has been looked at to display openGL do. This is the aim of why the ogl_vertex_buffer object exists within this class and not within the code to display openGL graphics.

This design has great difficulties as copying the vertex data into a QOpenGLBuffer failed in all but one method of code. Any QT QmessageBox that appears anywhere within the function to debug the code will crash the application. According to QT documentation,if all the vertex data was contained within a QVector<QVector3D> object, it should all be able to be copied into a QopenGLBuffer in one operation. But it was found that this could not be done. Instead it was found that the only method that worked was to copy one set of vertex data from the vertices object into a Qvector<QVector3D> and then use the Qt QOpenGLBuffer write function while keepin track of the offset to the correct buffer location within the QopenGLBuffer object.

The code to do this within the copy_vertices_to_OGL_vertex_buffer function is

```
QVector<QVector3D>   vbo_vertex;
offset = 0;
for (i = 0; i < number_vertices; i++) {
       vbo_vertex.append(vertices[i]->world_location);
       offset = i*number_vertex_data_elements * sizeof(GLfloat);

       ogl_vertex_buffer.write(offset,vbo_vertex.constData(),number_vertex_data_elements *
sizeof(GLfloat));

       vbo_vertex.clear();
}
```

Not efficient, but it works and time is saved by copying voxel data into doing this only. Preceding this there needs to be allocation of memory of the ogl_vertex_buffer object according to how many vertices are to be stored. This can all be seen in whole in the code of the function copy_vertices_to_OGL_vertex_buffer within the vw_voxel_cloud_object_class.

Calling of the ogl_vertex_buffer object in Qt

```
cloud->ogl_vertex_buffer.bind();
if(cloud->ogl_vertex_buffer.size() <=0) cloud->copy_vertices_to_OGL_vertex_buffer();
cloud->ogl_vertex_buffer.release();
```

Root Voxel Cloud Object

The root cloud object class is a very simple class that acts as the conduit for accessing all voxel object data to be displayed or manipulated by the application. In essence, it is a Qt QVector of pointers to the highest hierarchical voxel parent object data that has been created and which is to be displayed or manipulated and maintains the creation or deletion of. As inferred, there is only one Root cloud object for the whole application and all other classes that create or modify any voxel object data must have a reference to this class.

The root cloud object is defined as vw_root_voxel_cloud_object_class within the vw_voxel_cloud_object.h file of the Vitual_Worlds_Voxel_project.

Creating and Displaying a 3D Voxel space

Creating the 3D voxel space

The 3D voxel space that is stored in the computer memory is defined as a one dimensional dynamic vector array storing the voxel data.

The most appropriate data storage structure is a c++ vector array or its equivalent.

vector <voxel_data_type> voxel_matrix_data;

This vector array will have as its first or zeroth index entry the very first HCP Trapezo-rhombic dodecahedral voxel which is on the first zero row and zero column of the first x-y zero layer (or bottom) of the HCP Trapezo-rhombic dodecahedral voxel matrix cubic space that it is defined to exist within. From this point onwards, the word voxel will be a reference to the HCP Trapezo-rhombic dodecahedral voxel shape defined above.

If this first voxel has a defined size, and coordinate location of its centre, then any voxel of any index within the vector array can have its central coordinate location, and hence vertex locations making up its shape defined without storing them by computation. Thus the voxel data type of the voxel vector array need only store data that relates to a value or structure to be used as necessary for display or simulation.

To facilitate conversion of the vector array index to a central coordinate or vise versa, the dimensions of the voxel matrix need to be known and stored. Thus the central elements of a c++ data class to store the voxel data has a beginning structure of elements as

```
float                              voxel_size;
vector3D  matrix_origin            = { 0.0,0.0,0.0 };
index_vector matrix_dimension      = {0,0,0};

vector <voxel_data_type>           voxel_matrix_data;
```

In creating a voxel data vector array of a 3D space, that 3D space has to have a x,y,z, bounds and a voxel size specified to calculate the i,j,k voxel memory index values and and allocate memory to store the voxel data.

A minimum and maximum x, y, z values are specified, as is the voxel size or spherical radius r to use and display on screen. From these inputs a the allocation of memory and assignments of data related to the voxel space is stored in memory.

The minimum x,y,z, value specifiies the coordinate of the centre of the voxel that is at the zero index that defines the origin coordinate of the entire voxel matrix and from which all the ceter coordinates of all voxels is calculated from.

The dimensions of the x,y,z axis is not as straight forward as it is for a regular cubic shape. The number of voxels in each of the x,y,z directions is dependent upon the Trapezo-rhombic dodecahedral shape and the distance from the center of one voxel to the other in the x,y,z directions These spacings are

$$x_{spacing} = 2\mathrm{r}$$
$$y_{spacing} = \frac{3}{\sqrt{3}} r \qquad \textbf{- VS 1}$$
$$z_{spacing} = 2\frac{\sqrt{6}}{3} r$$

Where r = voxel size.

The values of x,y and z spacing are in real numbers and not integers, and thus if the following relationships have a decimal component, reducing them to an integer value will give an incorrect value by one less than required.

$$X\_d = (x\ maximum - x\ minimum)/ X_{spacing}$$
$$Y\_d = (y\ maximum - y\ minimum)/ Y_{spacing} \quad \text{- VS 2}$$
$$Z\_d = (z\ maximum - z\ minimum)/ Z_{spacing}$$

To correctly calculate the dimensions of the voxel space if X_d, Y_d, or Z_d have a decimal component then add one to the reduced integer value given by **VS 2**

eg for X_dim

```
if( X_d – integer(X_d) > 0.0)
        X_dimensinon =    integer(X_d) +1
else                                                    - VS 3
        X_dimensinon =    integer(X_d)
```

With this information the data structure of the 3D space that is stored in the computer memory as described in section **HCP Trapezo-rhombic dodecahedral voxel data storage** is defined, and memory allocated to store the voxel data.

Creating the voxel space data values

The data that is stored in the the voxel matrix data vector array is in general, a value that determines the status of that voxel. The value stored in the very basic is to indicate if the voxel is active and to have that voxel displayed on a computer screen, or inactive not to be displayed.

To give a voxel a value at coordinate at index value i,j,k is simply to calculate the vector index value given by **V7** in the section **HCP Trapezo-rhombic dodecahedral voxel data storage** and assign a value to the data type that is defined for the voxel matrix.

The steps involved are

i: To create an empty 3D voxel space with each voxel data value set to a desired value (eg inactive)
Creating the 3D voxel space.

ii: Iterate throught the 3D voxel space and assign a value to each voxel data element.
Iteration of the Trapezo-rhombic dodecahedral voxel matrix

Virtual Worlds implements this by

1: Having the user specify the voxel space and voxel size to iterate over, and a text file containing the expression on which to specify what value to give to each voxel in the voxel space specified.

2: The inputs in 1: are stored and passed to a generation function where they are processed to create a voxel data matrix that is iterated through, and values assigned according to the defined expression.

With the voxel data matrix values defined and stored in the voxel data matrix, the voxels can then be displayed on the screen.

Displaying the voxels

The Voxel data stored in the computer memory has only the x,y,z, Cartesian coordinates of the centre of the very first or origin voxel defined and stored. However, due to the nature of openGL and the GLSL shaders which display 3D computer graphics, a central location must be defined for each voxel to be displayed on a screen within an openGL vertex buffer object file. Since it is only the voxels that meet a specified criteria that need to be displayed, only the central vertex locations of these voxels need to be copied into the openG: vertex buffer.

Thus an iteration through the voxel data matrix and storing the coordinates of those voxels that meet the criteria to be displayed on screen is the first step in this process. Such an iteration process is described in Listing 3 of section **Iteration of the Trapezo-rhombic dodecahedral voxel matrix**.

Equation **V0** of the section **Constructing and Accessing the Trapezo-rhombic dodecahedral voxel matrix** in which x,y,z coordinates are calculated from the index values i,j,k of the voxel data matrix is not totally correct as it is only suitable for non alternating starting locations of the first voxel on each row and each k (z) layer.

To take into account the alternating starting locations of the first voxel in each row the calculation of the x value has a voxel size value added to the total if the row is odd, and the k (z) layer if even. If the k layer is odd as well as the row, a value of 2 is subtracted from the total.

The y value has the value (√3 – 1)* voxel size added for the odd row for all k values.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2\mathrm{i} + ((j + k) \bmod 2) \\ \sqrt{3}[j + \frac{1}{3}(k \bmod 2)] \\ 2\frac{\sqrt{6}}{3}k \end{bmatrix} r \quad \text{Even row (j mod 2 = 0) all k}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 + 2\mathrm{i} + ((j + k) \bmod 2) r - 2(k \bmod 2) \\ \sqrt{3} + \sqrt{3}[(j-1) + \frac{1}{3}(k \bmod 2)] r \\ 2\frac{\sqrt{6}}{3} kr \end{bmatrix} \quad \text{odd row (j mod 2 = 1)}$$

Expressed in C++ code the voxel centre x,y,z coordinates are defined as in listing DL 1

listing DL 1

```
glm::vec3 get_voxel_cartesian_coordinate(index_vector voxel_coord, float voxel_size) {
        index_data_type i = voxel_coord.x, j = voxel_coord.y, k = voxel_coord.z;

        float sqrt3 = sqrt(3.0), third = 1.0 / 3.0, z_mult = 2.0 * sqrt(6.0) / 3.0, sqrt3_2 = sqrt(1.5);
        glm::vec3 voxel_cartesian_coordinate;

        if (j % 2 == 0) {
                voxel_cartesian_coordinate.x = ((float(i) * 2 + float(k % 2)) * voxel_size);
                voxel_cartesian_coordinate.y =((sqrt3 * float(j) + sqrt3 * third * float(k % 2)) * voxel_size);
        }
        else {
                voxel_cartesian_coordinate.x = ((-1.0 + float(i) * 2 + float(k % 2) + 2 * float((k + 1) % 2)) * voxel_size);
                voxel_cartesian_coordinate.y = ((sqrt3 + sqrt3 * (float(j) - 1) + sqrt3 * third * float(k % 2)) * voxel_size);
        }

    voxel_cartesian_coordinate.z = (z_mult * float(k) *voxel_size);

        return voxel_cartesian_coordinate;
}
```

Listing DL2 is the C++ function that is used to assign the Cartesian coordinates to a vector array to be then assigned to a buffer to be used in an OpenGL shader program to display the voxel data in graphical form

listing DL 2

```
bool define_vbo_vertices(int min_voxel_value, int  max_voxel_value) {
   index_data_type i, j, k;
   int n = 0, voxel_index = 0;
   float x, y, z;

   voxel_object_data_class voxel_data = voxel_object_data;
   voxel_data_type         volume_data;
   voxel_element_data_type volume_data_value;

   point_data_value_class  vertex;

   if (point_cloud.vertices.size() > 0) {
      point_cloud.vertices.clear();
      point_cloud.vertices.shrink_to_fit();
    }

    bool even_z_level;
    index_data_type  dim_x, dim_y;


   for (k = 0; k < voxel_data.matrix_dimension.z && n < MAX_VOXEL_VERTICES; k++) {
```

```
        if (k % 2 == 0)
          even_z_level = true;
        else
          even_z_level = false;

        dim_y = ((k + 1) % 2) * voxel_data.matrix_dimension.y + (k % 2) * (voxel_data.matrix_dimension.y - 1);

        for (j = 0; j < dim_y && n < MAX_VOXEL_VERTICES; j++) { // y axis integer block  of 32 integer bit nodes
           if (even_z_level) {
             dim_x = (j + 1) % 2 * voxel_data.matrix_dimension.x + (j % 2) * (voxel_data.matrix_dimension.x - 1);
         }
          else {
             dim_x = (j + 1) % 2 * (voxel_data.matrix_dimension.x - 1) + (j % 2) * voxel_data.matrix_dimension.x;
         }

          for (i = 0; i < dim_x && n < MAX_VOXEL_VERTICES; i++) {// x axis integer
             volume_data = voxel_data.voxel_matrix_data[voxel_index];
             volume_data_value = voxel_data.extract_voxel_data_element_value(data_storage_type_enum::value, voxel_index);

             if (volume_data_value > INVALID_VOXEL_VALUE && volume_data_value <= MAX_VOXEL_VALUE) {// HCP voxel
creation

                glm::vec3 voxel_cartesian_coord = voxel_object_data.get_voxel_cartesian_coordinate({ i,j,k });

                 x = voxel_object_data.matrix_origin.x + voxel_cartesian_coord.x;
                 y = voxel_object_data.matrix_origin.y + voxel_cartesian_coord.y;
                 z = voxel_object_data.matrix_origin.z + voxel_cartesian_coord.z;
                 voxel_element_data_type lower_range=voxel_element_data_type(min_voxel_value),
                                                 upper_range=voxel_element_data_type(max_voxel_value);

                if (lower_range == INVALID_VOXEL_VALUE) lower_range = INVALID_VOXEL_VALUE + 1;
                i f (n < MAX_VOXEL_VERTICES && volume_data_value >= lower_range && volume_data_value <= upper_range)
{
                   vertex.mPos   = { x,y,z };
                   vertex.mvalue = volume_data;

                   point_cloud.add_vertex(vertex);
                   n++;
                 }
              }
              voxel_index++;
        }
     }
}
```

Once iterating through the voxel data matrix vector array and storing all the locations of the centres of all the voxels to be displayed in a vector array, this data is then copied to an openGL vertex buffer array object. This vertex buffer array object is then passed to a GLSL vertex shader object in conjuntion with camera and other data such as lighting and color.

The GLSL vertex buffer does its thing with this data and passes it on to a GLSL geometry shader. The geometry shader is where the vertices, the faces, color and normals to create the display of the Trapezo-rhombic dodecahedral voxel matrix takes place. This utilises the parallel processing power of the GPU and saves much time and memory in the host.

For each voxel central location passed to the geometry shader, each vertex of the Trapezo-rhombic dodecahedral voxel is calculated by adding the appropriate x,y,z, values as given in fig 5 of the section Hexagonal Close Packing Voxels. Each vertex defined is given the same ID number To display the faces of each triangle on this shape, the vertices that define the triangle must follow a certain order. The normal vector for each triangle face also has to be correctly calculated for each face. This was actually done on a spreadsheet as the normals are a constant value for a given triangle surface, and the values entered in as a GLSL vec3 value.

The voxels for each odd k or z layer are rotated 60º and thus a calculation of which layer the voxel is in based upon its central coordinate, the origin coordinate and voxel size as the vertex buffer array index has no correlation to the voxel data matrix index.

And then a color needs top be calculated using a color model algorithm taking into account such things as shadow, specularity, diffuse etc. This color ultimately is passed to the fragment shader to display the resultant image.

All the above is not difficult, but is tedious, long winded to test and debug and requires a lot of near repetitive code to implement. It is best to view all the GLSL shader code for further information.