Trapezo-rhombic dodecahedral Geometrical Shaped Voxel

A Method to obtain an evenly distribution of Voxels in 3D space using the Hexagonal Close Packing of spheres

Abstract:

Introduction :

Voxels are commonly defined and represented as a coordinate in 3D space as a cube volume evenly spaced with a center placed according to a regular x,y,z grid. Neighbours of any such cube that share a common vertex coordinate have the distances from the centers of the cube not equal for all neighbouring cubic voxels. Cubes that share a face in the x,y,z, axis direction have a distance of one unit between centers, while those that share a point or line that is on a diagonal to any of these axis have a distance of $\sqrt{(2)}$ units. This causes inconsistencies and considerations of which neighbouring voxel is being used if the distance between neighbouring voxel centers is used for whatever purpose.

The desire is to create a voxel geometric system for 3D space that is evenly distributed from the center of any one voxel to any neighbour surrounding it similar to the hexagon geometry of a flat 2D plane. Such a system would allow better representations of 3D space to be used for the purposes of generating 3D volumes and surfaces more evenly, with the greater intention of being able to more accurately represent a geometry of evenly spaced coordinates to be used for simulation and cellular automata.

What is below is a solution and an application of how such an evenly geometrical distribution of voxels, and the shape of those voxels can be used to give a systematic graphical representation of a 3D volume in computer memory and store values to be used for 3D graphics, simulation and procedural cellular automata algorithms.

Related Work:

Compare your work with existing work or the same problem

Problem and data:

The first task is to find a geometrical distribution of points in 3D space that is analogous and similar to the hexagonal distribution geometry in 2D space. Such a distribution is the hexagonal close packing (HCP) of spheres where the distance from the center of any one sphere to any of its twelve neighbours are equal.

The second task is to form a data structure to store this HCP coordinate and voxel value data, and how to retrieve and write voxel values to this memory storage. A 3D HCP voxel matrix is not as simple as a cubic matrix stored as a 3D xyz matrix in that the indices of such a matrix directly corresponds to an xyz coordinate based upon a constant size of the cubic voxel. The coordinates of the HCP voxel center will have a different x,y coordinate spacing to its z coordinate, and an offset to each other that is different for each odd or even row or column depending on one of two different configurations of how the HCP geometry is defined.

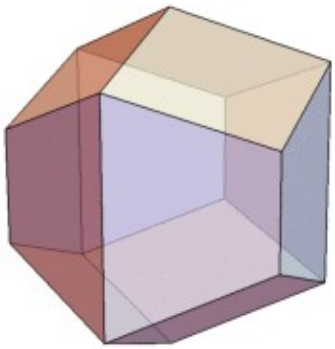
The third task is how to display the voxel value, volume or surface onto the computer screen.

Solution:

Hexagonal Close Packing Geometry to represent 3D Voxel Space

Section 1.0 : Defining the Trapezo-rhombic dodecahedral specification

Hexagonal close packing voxels is analogous to a 2D hexagonal grid where the distance from the center of any one hexagon to the center of each and every neighbouring hexagon grid cell is of equal distance. Hexagonal close packing geometry of voxels is in effect, a system of closely packed spheres of equal radius, where each voxel can be represented as a Trapezo-rhombic dodecahedral polygon shape,(fig 1)



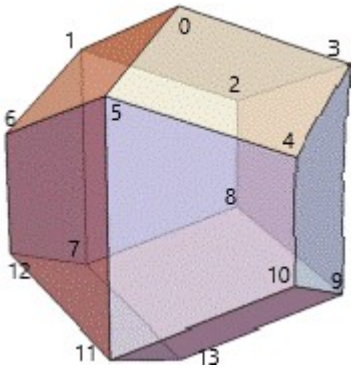


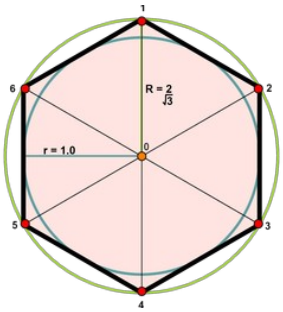
Fig 1 vertex indexes

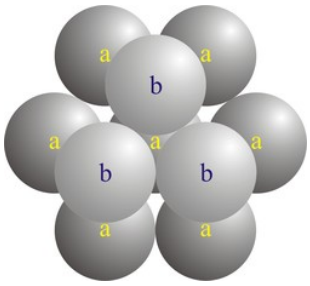
The Trapezo-rhombic dodecahedral polygon has twelve faces such that if spheres of equal size in a close packed hexagonal matrix were placed in the same space, the faces of each side would be perpendicular to the points at which each sphere would touch each other. The centre of each face is the location that a voxel sphere would touch its neighbour, and a line through it would connect the centres to each other. Thus the intersections of each of the planes as described above would form the boundaries of each of these polygonal surfaces.

There are six faces on the xy plane, that, if viewed from above or below, form a hexagonal grid pattern. To find the vertex coordinates of each of the surfaces about a central origin to construct the faces, a hexagonal pattern is used to find the x-y coordinates. There are two hexagonal configurations.

Configuration one : the direction to move in the x direction is in a straight line pattern, and on the y is staggered. Ie the hexagons are said to have a pointy top.
Configuration two : the direction to move in the y direction is in a straight line pattern, and on the x is staggered.Ie the hexagons are said to have a flat top.

Configuration one was selected as is is deemed easier to work with. For convenience, it was chosen that the radius of the sphere is of one unit and thus the x coordinate of each side in the x axis be one unit.(fig2). Thus the remaining x-y, vertex locations can easily be calculated with a central point (0,0,0). The z coordinate of each is mirrored about the xy plane





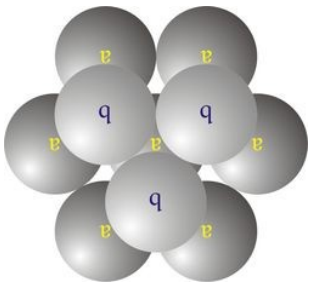


Fig 2 Fig 3a Fig 3b

The convention use in constructing the Trapezo-rhombic dodecahedral shape that is centered at the origin of 3D space, and looking down the z axis onto the x-y plane of Fig 1 is that vertex 0 is designated as the central positive z coordinate, and vertex 1-6 are the vertices of the positive z values beginning at the y=0 coordinate and rotating in a clockwise direction, (fig 2). Vetex 7-12 are the same as 1-6 but mirrored on the negative z plane. Vertex 13 is the same vertex as 0, and the mirror of it on the x-y plane.(fig 1)

For the x-y plane of packed spheres above or below this plane of closely packed spheres, three spheres can rest on this hexagon of six spheres below as an upright, or downward equilateral triangle. (Fig 3a and 3b). The upright arrangement (Fig 3a) would mean that the transition from the lower level to the upper level in a straight line on the y axis will occur in the positive y axis direction of this equilateral triangle configuration. To

travel in the negative y direction, a movement in either the positive or negative x direction. Conversely, the opposite would be the case for an inverted triangle configuration.(Fig 3b)

The inverted equilateral triangle arrangement (Fig 3b) was selected to specify the construction of the faces and conventions to use for the Trapezo-rhombic dodecahedral honeycomb structure that is used to construct the hexagonal close packing voxels structure. This specification requires the three rhombic shaped faces of the top and bottom to be as in fig 4a. The number of the face designates an ID to identify each face

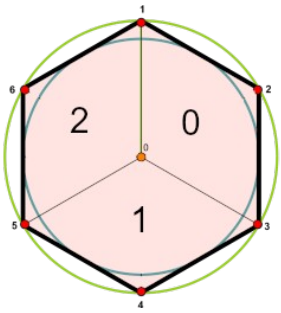


Fig 4a

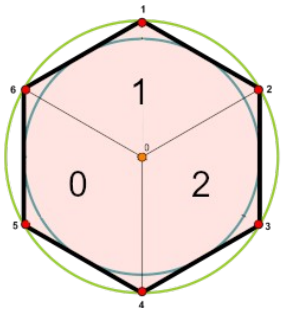


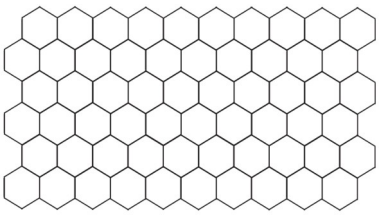
Fig4b

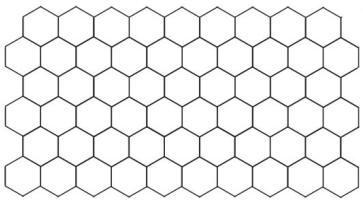
With this specification of the Trapezo-rhombic dodecahedral to be used, the z coordinates of each vertex can be found.

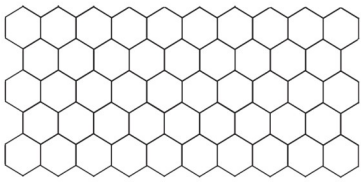
Section 1.1 : HCP Trapezo-rhombic dodecahedral voxel data storage.

Before considering how the HCP Trapezo-rhombic dodecahedral voxel matrix is to be stored in computer memory, the structure of that the HCP Trapezo-rhombic dodecahedral voxel matrix that will be represented in 3D space needs to be defined.

The common and well known conventional Cartesian 3D model of space as a cubic coordinate system which computers systems use for graphical display and memory storage is chosen as convenience and ease of use. Thus when defining the bounds of a 3D space, the HCP Trapezo-rhombic dodecahedral voxel matrix occupies a cubic 3D space defined by a x, y, z minimum and maximum boundaries.


10l,6j,nk even Layer
Fig 5a


10i,6j,nk even Layer
Fig 5b


10i,6j,nk odd Layer
Fig 5c

Because a HCP Trapezo-rhombic dodecahedral voxel is of a hexagonal nature, it will follow the same structure and behaviour as a 2D hexagonal grid. Consider a 2D hexagonal x-y plane defined as x-y dimension x_dim, y_dim (Fig 5a) where the first origin cell is defined as the bottom left hexagon of row 0, column 0. It can be seen than for equal number of hexagon cells in the x axis, there is an overshoot of one hex for each alternative odd row making the appearance of the grid to be less of a cube and more of a zig zag irregular shape. By taking away the last hexagon on each of the alternative odd rows (fig 5b), a more cubic form can be defined.

This has been selected to be the basis of the structure of the HCP Trapezo-rhombic dodecahedral voxel matrix in the z dimension as well where the first row of each alternative layer will have x_dim-1 voxels in the x direction and x_dim voxels for each alternative odd row. However each alternative z layer will have one less row similar to each alternative row in each layer having one less voxel for alternative rows.(Fig 5c)

There is one aspect of the HCP Trapezo-rhombic dodecahedral shape in the z direction that needs attention, and that is that the HCP Trapezo-rhombic dodecahedral shape of each alternative layer needs to be rotated by 60º to form a space filling lattice. The effect of this rotation is to change the orientation and order of the faces in the positive and negative z directions, as well as the vertices. Fig 4a displays the even z layer orientation and 4b the odd level orientation. The face ID numbers indicate the connecting faces of this orientation such that to move in a ± z direction, from one layer to the next one moves through the faces of the same face ID for each layer.

Thus a HCP Trapezo-rhombic dodecahedral voxel matrix cube will have the appearance as in fig 6.

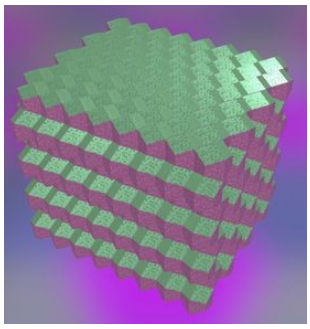


Fig 6

This is the basis to create a convention that is to be used to define the HCP Trapezo-rhombic dodecahedral voxel matrix. Thus having a computer storage as a conventional 3D vector array of some kind is not feasible, even if desirable.

Therefore the computer memory storage of the HCP Trapezo-rhombic dodecahedral voxel matrix is to be as a one dimensional array in preference to a three dimensional array to store the data elemental type that represents the state or other data that the voxel. This array needs to be dynamic in that the computer memory to store the voxel data can change depending upon the number of voxels defined. 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 by computation without storing them. 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 the C++ implementation using the GLM C++ library, the basic data storage of a HCP data model can have a source code of **Listing s1**

```
float                 voxel_size;
glm::vec3             matrix_origin;
glm::ivec3            matrix_dimension;

std::vector <voxel_data_type> voxel_matrix_data;
```

Listing s1

Section 1.2 Selection of voxel data storage data type and definition of data storage class

Voxel data is designed to be stored as a data type that reflects the activity of a voxel. That is, to have a value to designate that the voxel at a particular location is valid and can be used and displayed, or invalid and cannot be displayed. Such a data type that can be used to reflect this is an 8 bit byte integer which was chosen as a first option such that if a very large number of voxels are present, such in the millions, a one byte memory data type for the state of a voxel would use less memory.

The byte integer can be either signed or unsigned. An unsigned integer is chosen such that any zero value is designated as invalid, and a value from 1-255 defined as valid.

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 value of a byte of computer memory refers to. Voxel data represents a data set of an evenly spaced tessellated geometric shape forming a grid or matrix. Thus the first voxel value of an array of voxel values can represent the origin point from which all other voxels can have their coordinates calculated, given the spacing between each tessellated voxel center is constant.

Even though having a data type of an 8 bit integer would be ideal to store an 8 bit (ie 256) number of voxel states, and lower computer memory resources, a need to transfer this state data to an OpenGL shader program requires this data as a minimum be of type integer, as OpenGL in all its wisdom of design does not allow data types other than float, double or integer to be used to represent a number.

It was initially considered of having a four byte integer designated to carry four 8 bit integers representing four voxels to save computer memory could be implemented, but after testing, it was found to be complicated and also not practical. Also, OpenGL would need a single voxel value for each vertex point representing the center of a voxel sphere.

So in the end, it was decided to use a normal integer data type to represent a voxel state.

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 vector array as by knowing the dimensions of the matrix, the spacing between each voxel center (size of voxel) and the origin that the first value represents, the coordinate of any index within the vector 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 s1 :

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

        vector <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 defined 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 cubic 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.

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.

Section 1.3 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 1 as

matrix_dimension.x, matrix_dimension.y, matrix_dimension.z

or using the conventional voxel matrix coordinates

matrix_dimension.i, matrix_dimension.j, matrix_dimension.k

where i,j,k are used as substitutes for the C++ vector variables x,y,z.

The number of voxel coordinate components for each coordinate type i,j,k is not constant in respect to the voxel size as it is with the size of a cube in Cartesian coordinates. This is because as a sphere packed together into a 2D or 3D space equidistant according to their spherical shape does not translate into an equidistant displacement along the conventional Cartesian axis, but do so on the axis accordingly along where the spheres touch each other.

Thus using conventional Cartesian coordinates to define the dimension of a hexagonal close packed spherical voxel matrix requires a different displacement in each of the x,y,z axis directions. Thus a cube of Cartesian dimensions of equal size S in x,y,z directions would have a cubic voxel shape of equal dimensions D in the x,y,z axis. Ie dimension x = dimension y = dimension x = D. However a hexagonal close packed of **trapezo-rhombic dodecahedron** shape voxel would have different dimensions within the same Cartesian cube and having the same size S. ie dimension i ≠ dimension j ≠ dimension k.

However the dimensions of a voxel matrix grid of voxel size VS can be easily calculated. Consider that the voxel size is the radius R of the sphere that it represents in a hexagonal close packing of spheres of equal size. Then the centers of each voxel on the same row are separated in the x or i direction by 2R and

voxel dimension on the x or i axis = x size/(2*R) **E-1.1.1**

On the y or j axis, each sphere or voxel is on a row which are separated from the row above or below by a distance in Cartesian coordinates of $2*\sqrt{3}$ and

voxel dimension on the y or j axis = y size/($\sqrt{3}$*R) **E-1.1.2**

On the z or k axis, each sphere or voxel is on a layer of hexagonaly spaced spheres which are separated from the layer above or below by a distance in Cartesian coordinates of $2*\sqrt{6}/3$ and

voxel dimension on the z or k axis = z size/($2*\sqrt{6}/3$*R) **E-1.1.3**

Thus by use of **E-1.1.1,E-1.1.2,E-1.1.3** the dimensions of a voxel matrix grid can be found and a voxel matrix created to store voxel cell state data from a given rectanglular cubic shape given in Cartesian coordinates.

Section 2.0 Constructing and Accessing the HCP Trapezo-rhombic dodecahedral voxel matrix

The HCP Trapezo-rhombic dodecahedral voxel matrix is in essence, a hexagonal close packing lattice or matrix structure. The coordinates of the voxel (ie sphere) centres correspond to the coordinates of a HCP lattice which are known and can be used here. For the definition defined in sections 1.0 and 1.1, considering that **i** is the index of the voxel on the x axis, and **j** on the y axis and **k** on the z axis, for a sphere or voxel of size (close packing sphere radius) r the coordinate of the **i,j,k** voxel center is given by

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2\mathrm{i} + ((j + k) \bmod 2) \\ \sqrt{3}[j + \frac{1}{3}(k \bmod 2)] \\ 2\frac{\sqrt{6}}{3}k \end{bmatrix} r \qquad \text{- V0}$$

This gives a HCP matrix configuration for each alternative layer lying on top of each other as in fig 7. The black hexagon matrix depicts the even or **k** mod 2 = 0 layers, and the green hexagon matrix the odd, or **k** mod 2 = 1 layers of the voxel 3D cubic matrix. The green and black coordinates depict the plane **i,j** (ie x,y) index coordinates of the centres of the voxels (ie close packing sphere centers)of each of these layers. As can be seen, the odd layer can be considered as a displacement of all of the rows > 0 of the even layer -2/√3 units down in the Cartesian coordinate system..

For a given **i,j,k** coordinate in the voxel matrix, a Cartesian z,y,z, coordinate can be derived knowing the Cartesian coordinate value of the 0,0,0 voxel, and the size of the voxel, ie radius from relationship **V0**

However, since the voxel matrix is stored as a one dimensional vector array, the **i,j,k** index values specified have to have the vector array index value that corresponds to these given **i,j,k** index values calculated. To do this, the specification defined above in section 1.1 of the voxel matrix is crucial in finding this index value.

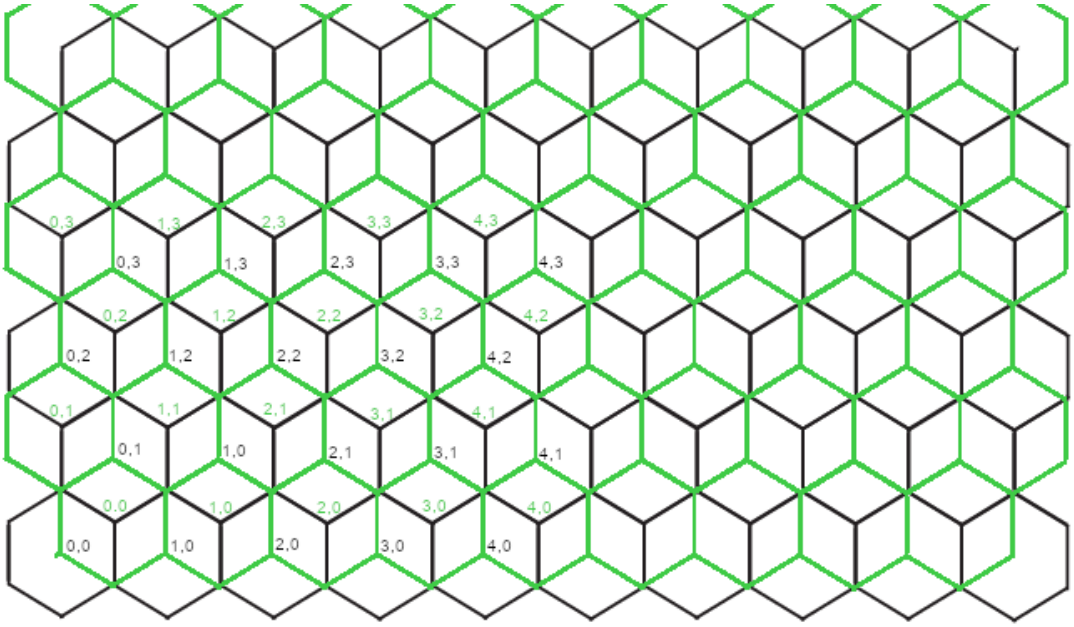


Fig 7

In finding the vector index, **vi** that corresponds to the HCP matrix coordinate index **i**,**j**,**k**, it is best to begin with finding the vector index **vi** of the even **k** = 0 level of a voxel matrix of x dimension x_dim and y dimension of y_dim. The specification definition has each even row (**j** mod 2 = 0) having x_dim voxels in that row, and each odd row (**j** mod 2 = 1) having x_dim-1 voxels in that row. This means that for any x,y coordinate of **i**,**j** in the **k**=0 level, the number of voxels up to the **j**-1 row needs to be calculated, and then the **i** coordinate on the row that the voxel is located on added to this total..

To do this, the number of even rows before the row that the voxel is located on is
(floor (**j** / 2) + **j** mod 2)x_dim
and the number of odd rows before the row the row the voxel is on is
(floor (**j** / 2))(x_dim-1).

Thus the total number of voxels of the coordinate **i**, **j** on an even **k** (z) layer is

vi = (floor (j / 2) + j mod 2)x_dim + (floor (j / 2))(x_dim-1) + i - **V1**

For an odd z layer, the number of voxels to the **i**,**j** coordinate is the inverse of V1 due to the inverse of the number of voxels per even and odd rows.

That is for an even **j** row on an odd **k** level there are (floor (**j** / 2))(x_dim-1) voxels on that row, and on an odd **j** row, there (floor (**j** / 2) + **j** mod 2)x_dim voxels on that row. Therefore the number of voxels of the coordinate **i**, **j** on an odd **k** (z) layer is

vi = (floor (j / 2) + j mod 2)(x_dim-1) + (floor (j / 2))x_dim + i - V2

And putting this all together a more general calculation of the number of voxels or index count on the **k** level of the voxel matrix for a voxel at coordinate (**I,j,k**) is

```
if (k mod 2 == 0)
        vi = (floor (j / 2) + j mod 2)x_dim + (floor (j / 2))(x_dim-1) + i
else                                                                    - V3
        vi = (floor (j / 2) + j mod 2)(x_dim-1) + (floor (j / 2))x_dim + i
```

In a similar fashion to the above, for any voxel coordinate (**i**,**j**,**k**) the number of voxels before the **k** layer that the voxel is located within needs to be calculated, and **V3** added to that total to give the correct vector array index **vi** within a one dimension storage vector. Ie

1) Count the number of even and odd layers before the **k** layer that the voxel is within
2) Calculate the total number of voxels within an odd and even layer
3) Multiply the number of even layers in 1) by the total number of voxels per even layer
4) Multiply the number of odd layers in 1) by the total number of voxels per odd layer
5) Add 3) and 4) and the result of V3

For an even z layer, the total number of voxels per layer can be found using **V1** and substituting 0 for **i** and y_dim for **j** giving

total voxels even k = (floor (y_dim/ 2) + y_dim mod 2)x_dim + (floor (y_dim / 2))(x_dim-1) - V4

For an odd z layer the number of rows as defined by the specification of the voxel matrix above has y_dim-1 rows and thus in **V2** above y_dim-1 is substituted for **j** and **i** =0 giving

total voxels odd k = (floor (y_dim-1/ 2) + y_dim-1 mod 2)(x_dim-1) + (floor (y_dim-1) / 2))x_dim - V5

Putting V4 and V5 together gives

```
if(k mod 2 ==0)
    total voxels even layer= (floor (y_dim/ 2) + y_dim mod 2)x_dim + (floor ( y_dim / 2))(x_dim-1)
else                                                                                              -V6
    total voxels odd layer= (floor (y_dim-1/ 2) + y_dim-1 mod 2)(x_dim-1) + (floor ( y_dim-1) / 2))x_dim
```

By using both **V3** and **V6**, for a given voxel matrix of coordinate (**i**,**j**,**k**), the index of the storage vector voxel data matrix can be calculated. The number of even k levels and number of odd k levels is as given for the number of even and odd rows for a given (**i**,**j**) coordinate on a even **k** level in **V1**. By substituting **k** for **j** and **V3** for **i** in **V1**, and **V6** for x_dim and x_dim-1 the index **vi** of the voxel data matrix vector array is

vi = (floor (k/ 2) + k mod 2)V6(k=0) + (floor (k / 2))V6(k=1) + V3(i,j,k) - V7

Listing 1 defines **V3**, **V6** and **V7** in c++ computer code that will give the index location within the one dimensional memory storage vector array that stores the voxel value. **V7** is the c++ function get_index_value, **V6** is get_z_layer_total and **V3** is get_z_layer_index_value in listing 1.

Listing 1

```
index_data_type get_z_layer_index_value(index_data_type iX, index_data_type iY, index_data_type iZ) {
    if(iZ % 2 == 0) // Even z level
       return (index_data_type(iY/2) + iY%2)*matrix_dimension.x + index_data_type(iY/2)*(matrix_dimension.x-1) + iX;
    else // Odd z level
       return (index_data_type(iY/2) + iY%2)*(matrix_dimension.x-1) + index_data_type(iY/2)*matrix_dimension.x + iX;
}

index_data_type get_z_layer_total(index_data_type iZ,index_data_type xdim, index_data_type ydim) {
    if(iZ % 2 == 0) // Even z level
        return (index_data_type(ydim/2) + ydim%2)*xdim + index_data_type(ydim/2)*(xdim-1);
     else // Odd z level
```

```
        return ((index_data_type(ydim-1)/2) + (ydim-1)%2)*(xdim-1) + index_data_type(ydim-1)/2*xdim;
}

index_data_type get_z_layer_total(index_data_type iZ) {
      return get_z_layer_total(iZ,matrix_dimension.x, matrix_dimension.y);
}

index_data_type get_index_value(index_data_type iX, index_data_type iY, index_data_type iZ) {
     return (index_data_type(iZ/2) + iZ%2) * get_z_layer_total(0) + index_data_type(iZ/2)* get_z_layer_total(1) +
     get_z_layer_index_value(iX,  iY, iZ);
}
```

Even though the voxel matrix data is a dynamic vector array, it is best to allocate memory and construct this vector array as a data block of memory in a single action, and not use a series of loops and appending a new voxel_data_type on each iteration. This saves a large amount of time and is greatly more efficient. To perform this action there needs to be a calculation of the total number of voxels that make up the matrix of dimension x_dim,y_dim,z_dim. This total is summing up all the voxels on the even and odd layers and is given in **V8**.

$$V_{total} = (\text{floor } (z\_dim/ 2) + z\_dim \bmod 2)V6(k=0) + (\text{floor } (z\_dim / 2))V6(k=1) \quad - \mathbf{V8}$$

The C++ code that is equivalent to V8 is as given in listing 2

Listing 2

```
index_data_type calculate_voxel_matrix_data_size(index_data_type xdim, index_data_type ydim, index_data_type zdim)
{
     return ( index_data_type(zdim/2)+ zdim % 2) * get_z_layer_total(0,xdim, ydim) +
               index_data_type(zdim/2) * get_z_layer_total(1,xdim, ydim);
}
```

Section 2.1 Iteration of the HCP Trapezo-rhombic dodecahedral voxel matrix

Knowing the dimensions of the voxel matrix, iteration through every voxel is not all too difficult if the iteration is performed in the same order as stored in the one dimensional voxel data matrix vector array. As described in section 2.0, Constructing and Accessing the HCP Trapezo-rhombic dodecahedral voxel matrix**,** the data is stored in a sequence of alternating X_dim voxels for each even row and X_dim-1 voxels for each odd rows in an even Z or **k** level which has Y_dim rows. For each odd **z** or **k** level there are Y_dim-1 rows with alternating X_dim-1 voxels on each even row, and X_dim voxels on each odd row.

Thus the iteration from vector index 0 to the end is to iterate in the order of z, y, x loops, taking into account if the z (layer), and y (row) value is even or odd. Listing 3 gives such an iteration series of loops.

Listing 3

```
for (k = 0; k < voxel_data.matrix_dimension.z && n < MAX_VOXEL_VERTICES; k++) { // Z axis level
      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
            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 row block

                  // ********* DO ITERATION STUFF HERE ********** //

              }
        }
}
```

Within the code of listing 3, a form the same algorithms **V3** and **V6** in section 2.0 HCP Constructing and Accessing the Trapezo-rhombic dodecahedral voxel matrix**,** are used to find the number of voxels in a row, and the number of rows to iterate through. dim_x is the number of voxels (or voxel cells) in the row being iterated and dim_y is the number or rows in the z layer being iterated.

Section 2.3 Find the voxel (i.jk) coordinate from a voxel data storage array index vi

There are times that for when processing the storage data as a vector a requirement of deriving the voxel coordinate from the voxel vector index is needed. That is, the inverse of **V7** in section 2.0.

To do this, the total number of voxels per even and odd layers are needed, and thus **V4** and **V5** are utilised to do this. By adding **V4** and **V5** together, a total number of voxels per even odd pair (**zt**) is given.

zt = V4 + **V5** **V2.3.1**

It can be deduced which layer pair (**lp**)**, vi** is in by taking the floor value of dividing **vi** by **zt** ie

lp = floor(**vi/zt**) **V2.3.2**

if **lp** is zero then **vi** is in either layer k=0, or k=1. if **lp** is one then **vi** is in either layer k=2, or k=3 and so on.

Now it is known which layer pair **vi** is in, next, is to find if **vi** is in the even or odd layer of this layer pair. To do this a similar method to those employed in section 2.0 can be applied.

1) Find the total number of voxels before the layer pair that **vi** lies within. ie Multiply **lp** given in **V2.3.2** by **zt** given in **V2.3.1**
2) Subtract the value of 1) from **vi** to give a relative index with an odd-even layer
3) Divide 2) by the total number of voxel cells that exist within an even layer (**V4**) and floor it
4) If 3) is greater than zero, it **vi** is in the odd layer **lp***2+1, otherwise it is in the even layer **lp***2

Putting this together in pseudo code

number voxels in previous layer pairs **nv** = **lp * zt**
relative voxel index pair index **rvi** = **vi – nv**

if(floor(**rvi/V4**) > 0)
voxel layer coord **k** = **lp***2+1
else **V2.3.3**
voxel layer coord **k** = **lp***2

Now that it is known which voxel matrix layer or **k** coordinate **vi** is present within, what remains is to find which row or **j** coordinate of that layer **vi** is within. To do this a relative index of the voxel index within its layer is calculated, and from this a determination of which row within this layer that **vi** is in can be determined.

The relative layer index of **vi** (**rvi)** is **vi** – vector index of voxel at voxel matrix coordinate (0,0,**k**) as given by **V7** in section 2.0 ie

rlvi = **vi** – **V7 for coordinate (0,0,k)** **V2.3.4**

A similar method to finding which layer **vi** is within is employed to find which row of the voxel matrix **vi** lies within using **V2.3.4**.

The total number of voxels for an odd-even row pair (**nvrp**) is
nvrp = 2*voxel matrix x dimension -1 **V2.3.5**
the row pair (**rp**) that **vi** lies within is
rp = floor(**rlvi/ nvrp)** **V2.3.6**

which then gives
number voxels in previous row pairs **nvprp** = **rp * nvrp**
relative voxel index pair index **rrvi** = **rlvi – nvprp**

To determine which row **rlvi** lies within depends upon whether the layer **k** that the voxel lies within is odd or even. If **k** is even then

if(floor(**rrvi/nvrp**) > matrix x dimension)
voxel row coord **j** = **lp***2+1
else **V2.3.7**
voxel row coord **j** = **lp***2

If **k** is odd then

if(floor(**rrvi/nvrp**) > matrix x dimension-1)
voxel row coord **j** = **lp***2+1
else **V2.3.8**
voxel row coord j = lp*2

To find the voxel **i** coordinate for an even layer, a count of all the voxels in the rows of the layer before the row **j** that the voxel lies within is performed, and then the relative voxel index is subtracted from this total to give the voxel **i** coordinte. This is dependent upon the layer **k** being odd or even.

For an even **k** layer

i = **rlvi** – (floor(**j**/2)+**j** mod 2)*matrix x dimension+(floor(**j**/2))*(matrix x dimension -1) **V2.3.9**

For an odd **k** layer

i = **rlvi** – (floor(j/2)+j mod 2)*(matrix x dimension -1)+(floor(j/2))*(matrix x dimension) **V2.3.10**

Thus for a given voxel vector storage array of a given index **l** a voxel matrix (I,j,k) coordinate can be found relatively easily.

Section 2.4 Find the voxel Cartesian coordinate (x,y,z) from voxel matrix coordinate (i.jk)

To calculate the center of a voxel Cartesian coordinate from its matrix coordinate, the size of a voxel cell, and the Cartesian coordinate of the voxel origin cell need to be defined and known. The centers of each matrix voxel cell is a center of a sphere in a hexagonal close packed lattice pattern, so the calculation of finding the Cartesian coordinate of a voxel cell of matrix coordinate can use the knowledge pertaining to hexagonal closed packed lattice of spheres of equal size.

To find the Cartesian coordinates for any voxel of voxel matrix coordinate I,j,k, V0 of section 2.0 is used.

Section 3.0 Finding the HCP Trapezo-rhombic dodecahedral vertex coordinates

To find the correct z coordinate of each vertex for the purpose of calculation or graphics, one needs to find first the z coordinate at which each neighbouring sphere touches the central origin sphere. On the xy plane this is is zero. To find the z coordinate for a Hexagonal Close Packing lattice of spheres of equal radius **r,** the formula

$$(x,y,z) = \left(2i + ((j+k) \bmod 2)\mathbf{r},\quad \sqrt{3}\ [j+\ \frac{1}{3}\ (k \bmod 2)],\quad \frac{2\sqrt{6}}{3}\ k\right)\mathbf{r} \quad (1)$$

is used

All the spheres touch the central origin sphere on the same z coordinate value, and thus to find the z coordinate of one touching sphere, all the others would have the same z coordinate value. The coordinate of the sphere that touches the origin sphere on face 0 as defined in fig 4a and fig S3.0.1 is

$$(x,y,z) = \left(1,\ \frac{\sqrt{3}}{3}\ ,\ \frac{2\sqrt{6}}{3}\ \right) = \left(1,\ \frac{1}{\sqrt{3}}\ ,\ \frac{2\sqrt{6}}{3}\ \right)$$

The coordinate at which the spheres touch the midpoint of the line (0,0,0) to (x,y,z) = (1, $\frac{1}{\sqrt{3}}$, $\frac{2\sqrt{6}}{3}$) is $(x_m,y_m,z_m) = (\ \frac{1}{2}\ ,\ \frac{1}{2\sqrt{3}}\ ,\ \frac{\sqrt{6}}{3}\ )$.

The vertices 1, 3, and 5 (Fig 1) are of the same z value as where the central sphere touches the spheres above it, which is $z_m = \frac{\sqrt{6}}{3}$. As 7, 9 and 11 (Fig 1) are the mirror of 1,3, 5, have a z coordinate $\frac{-\sqrt{6}}{3}$.

Similarly, vertices 2,4, and 6 all have the same z coordinate value and need to find only the z coordinate for one of them. This can be done by considering the point of contact of the two spheres touching each other on the rhombic plane of the 0 face (see fig 4a) is also the midpoint of the length of an equilateral triangle that makes up the rhombic plane of face 0. This midpoint has an equal distance to both the apex 0 vertex and the side 2 vertex. Finding the unit vector perpendicular to the point of contact between the two spheres and on the surface of the plane that makes up the rhombic shaped plane of face 0 will give a direction vector {l,m,n} that can be used to find the equation of a line in 3D space at point (x_0,y_0,z_0) by use of the relationship

$$\frac{x-x_0}{l}=\frac{y-y_0}{m}=\frac{z-z_0}{n} \qquad (2)$$

The unit directional vector can be found by use of the cross product of two vectors which will give the vector perpendicular to the plane that two vectors lie on. The desired resultant unit directional vector is on the plane of face 0 and the vector perpendicular to this face is the vector through the centres of the spheres touching at the point on this surface. Ie the vector from point ($\frac{1}{2}$, $\frac{1}{2\sqrt{3}}$, $\frac{\sqrt{6}}{3}$) to the origin (0,0,0)

A vector common to both the plane of face 0 and that perpendicular to it is from the point of contact to the vertex 3

Thus the cross product of the vector emanating from this point of contact on the surface of face 0 is the determinant of the matrix

$$\begin{vmatrix} i & j & k \\ \frac{1}{2} & \frac{1}{2\sqrt{3}} & \frac{\sqrt{(6)}}{3} \\ \frac{1}{2} & \frac{-\sqrt{(3)}}{2} & 0 \end{vmatrix} \qquad (3)$$

which gives the result

$$\frac{1}{\sqrt{2}}i+\frac{1}{\sqrt{6}}j-\frac{1}{\sqrt{3}}k \quad = \quad l\,i+m\,j-n\,k \qquad (4)$$

which is also a unit directional vector.

the equation (2) relationship can now be used to find the z value of vertex 0, 2,4, 6 by substituting the location of the vertex points x,y, and z , and the location of the point of contact for x_0,y_0,z_0 .

Thus have for the positive apex vertex 0

$$\frac{0-\frac{1}{2}}{\frac{1}{\sqrt{2}}}=\frac{0-\frac{1}{2\sqrt{3}}}{\frac{1}{\sqrt{6}}}=\frac{z-\frac{\sqrt{6}}{3}}{\frac{-1}{\sqrt{3}}} \quad \Rightarrow \quad \frac{-1}{\sqrt{2}}=\frac{-1}{\sqrt{2}}=\frac{z-\sqrt{6}}{\frac{-1}{\sqrt{3}}}$$

$$\Rightarrow z = \frac{\sqrt{6}}{3}+\frac{1}{\sqrt{6}} = \sqrt{\frac{3}{2}} \quad \text{for vertex 0}$$

for vertex 2,4, and 6 it is found that

$$z = \frac{\sqrt{6}}{3}-\frac{1}{\sqrt{6}} = \frac{1}{\sqrt{6}}$$

since the Trapezo-rhombic dodecahedral polygon is mirrored over the z = 0 plane, the z values are negative in the negative z space.

Therefore the complete HCP Trapezo-rhombic dodecahedral elemental polygon shape, and the coordinates that make it up to use in a HCP voxel matrix are defined as given in fig 8.

By using these constant values for a unit sized Voxel, the geometry of a Trapezo-rhombic dodecahedral polygon can be constructed at any scale or location for the purposes of visualisation on a computer screen.

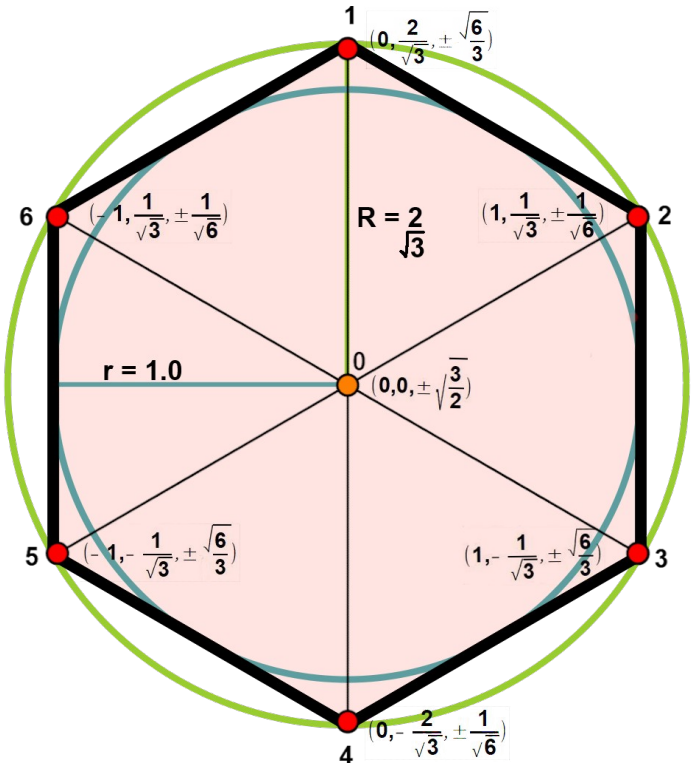


Fig 8.

Results:

Conclusion:

References:

Cellular Automata

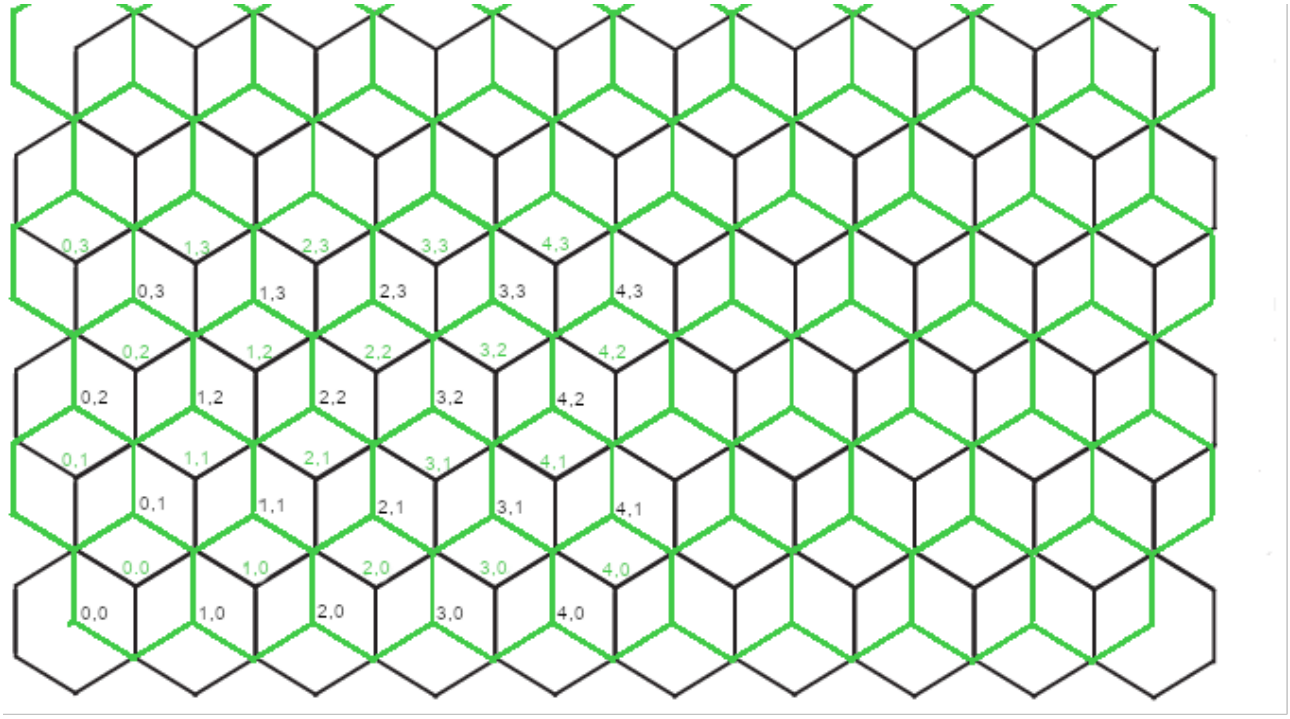


Fig CA1

Section 4.0 : Voxel cell neighbour specification

Fig CA1 is a representation of an even (black) and odd (green) k (z) layer or plane of the voxel data matrix and the associated i,j index coordinates as defined in the section **HCP Trapezo-rhombic dodecahedral voxel data storage**. The over lay of these odd and even hexagon grids gives the appearance of isometric cubes. The “faces” of these isometric cubes gives the direction or face that any one voxel has with its neighbour in the k layer above or below. Eg the even voxel located at i,j coodinate 1,0 (black color) has a rhombic shape directly below it with the upper lines in green and the lower lines in black that form the rhombic shape.

This indicates that the voxel in the even k layer at coordinate 1,0 has a neighbouring voxel in the odd k layer above or below it also at i,j coordinate 1,0. Similarly the odd level voxel at location 1,0 (green)has two other neighbours at i, j coordinates 2,1 and 1,1 on the even layers above and below it. The isometric rhombic shape with a coordinate at its center gives the face of the top and base of the HCP Trapezo-rhombic dodecahedral voxel shape corresponding to the even layer for black or odd layer for green colour the coordinate. It also indicates that a neighbour of that HCP Trapezo-rhombic dodecahedral voxel in the next layer above or below it is in the direction that the short diagonal that makes up the rhombic shape.

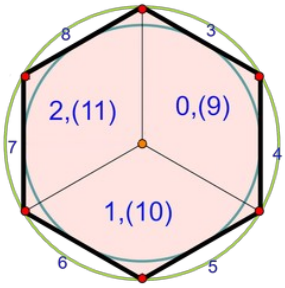


Fig CA 2a Even layer face id

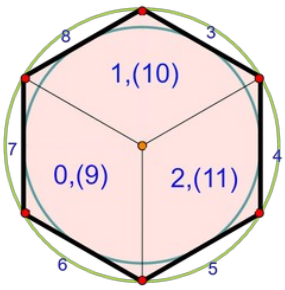


Fig CA 2b Odd layer face id

Fig CA 2a and 2b gives a specification of the faces and associated face for each even and odd k layer voxel. The numbers in brackets give the face id number of the faces at the base of the voxel. The face ids of the faces that have neighbours in the same k level do not change from the even to odd level. The face id also can have a corresponding neighbour id of the same id number assigned. Note that the faces for the even and odd layer voxels correspond to Fig CA1. Notice that the transition from an even level to an odd level or vise versa given in fig CA 2a and CA 2b occurs on a face of the upper surface of one voxel being shared with

the face of the lower surface, and that these shared surface id numbers have a difference of 9. ie face value 0 on an even or odd voxel shares a face of id value 9 with its neighbouring voxel on the next level above it etc. Similarly for voxels on the same level, there is a difference value of 3 between neighbouring face ids.

With this specification of the voxel faces, neighbours and the coordinates of alternating layers given in fig CA1, a table of neighbour coordinate relationships can be made, and from this table a calculation of the index in the voxel data storage matirix using relationship **V7** from the section **Constructing and Accessing the HCP Trapezo-rhombic dodecahedral voxel matrix**. Can be made.Table TCA1 gives such a relationship. It must be noted that from fig CA1 above, the coordinate of neighbouring voxels in the layer above and below also depends upon if the voxel is in a even or odd row

Even k voxel

Neighbour	Even Row Coordinate	Odd Row Coordinate
0	(i,j,k+1)	(i+1,j,k+1)
1	(i,j-1,k+1)	(i,j-1,k+1)
2	(i-1,j,k+1)	(i,j,k+1)
3	(i,j+1,k)	(i+1,j+1,k)
4	(i+1,j,k)	(i+1,j,k)
5	(i,j-1,k)	(i+1,j-1,k)
6	(i-1,j-1,k)	(i,j-1,k)
7	(i-1,j,k)	(i-1,j,k)
8	(i-1,j+1,k)	(i,j+1,k)
9	(i,j,k-1)	(i+1,j,k-1)
10	(i,j-1,k-1)	(i,j-1,k-1)
11	(i-1,j,k-1)	(i,j,k-1)

Odd k voxel

Neighbour	Even Row Coordinate	Odd Row Coordinate
0	(i,j,k+1)	(i-1,j,k+1)
1	(i,j+1,k+1)	(i,j+1,k+1)
2	(i+1,j,k+1)	(i,j,k+1)
3	(i,j+1,k)	(i+1,j+1,k)
4	(i+1,j,k)	(i+1,j,k)
5	(i,j-1,k)	(i+1,j-1,k)
6	(i-1,j-1,k)	(i,j-1,k)
7	(i-1,j,k)	(i-1,j,k)
8	(i-1,j+1,k)	(i,j+1,k)
9	(i,j,k-1)	(i-1,j,k-1)
10	(i,j+1,k-1)	(i,j+1,k-1)
11	(i+1,j,k-1)	(i,j,k-1)

TABLE TCA1

Section 4.1 : Cellular Automata algorithm

Table **TCA1** gives a basis from which to develop a cellular automata algorithm from which to write computer code to implement. Knowing all the neighbours of each voxel, and being able to retrieve the data from computer memory storage vector array, and by a system of rules based on these values, and that of the voxel itself, a new value for the voxel can be given and assigned to computer memory.

Cellular automata is the process of stepping from a present state of the overall voxel matrix to a new state through the process of applying these defined rules. Thus the cellular automata algorithm would be.

CA 1: Define an initial state of the voxel data matrix vector array.

CA 2: Define the cellular automata rules to apply to the neighbours of each voxel in CA 1 to produce a new voxel data matrix vector array.

CA 3: Create an empty voxel data matrix vector array that is at of the same dimensions as the voxel data storage matrix vector array in CA 1.

CA 4: Apply the rules defined in CA 2 to the voxel data storage matrix array defined in CA 1, and store the results in the voxel data storage matrix vector array of CA 2. When Complete delete CA 1 from Memory

CA 4a: Find the coordinate of each neighbour according to table CA1.
CA 4b: If the coordinate found in CA 4a is valid and falls within the dimension of the voxel array structure, retrieve the neighbour status value to be used by CA 2;
CA 4c: perform the rules defined in CA2 and assign the resultant status value to the voxel at its location in the empty voxel data matrix vector array created in CA 3:

CA 5: Display the results of CA 3.

CA 6: Repeat steps CA 3 to CA 5 as necessary.

Much of the details of this implementing this algorithm (CA 1, CA 3, CA 4a,CA 4b, CA 5) in virtual worlds is

described in the previous sections and left to examining the C++ code.

Section 4.2 : Cellular Automata rules

The neighbour rules that CA2 refers to are a set of 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 has a rule that is tested. If the rule is met then the rule as a whole can designated as a state of true. If any neighbour fails this in the rule being met, then the rule as a whole is classed as unmet or false. 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 according to the mathematical relationship as given in car01.

A (<,<=,ignore) VV (=, !=, < ,<=, ignore) B - car01

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 car02 form one particular neighbour

A < VV < B - car02

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 car03 form one particular neighbour

ignore VV <= B - car03

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 car04 form one particular neighbour

A < VV ignore - car04

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 car01, 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 car01 and the upper rule being the right side of car01.

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.

Section 5 : VOXEL COORDINATE SYSTEM

NOTE : The voxel index coordinate system used to identify the voxel coordinate as (column, row, level) (i,j.k) that is implemented by the application and documented through out this document and else where is not to be correlated with the coordinate system of this section. What is here is a proposal of a coordinate system that is to be at some time used and have a translation between the these two systems implemented in some way at a future date. If done so this section will be updated.

The voxel coordinate system is essentially similar to the hexagonal coordinate system used for a hexagonal map. Each flat even and odd plane in the Cartesian z direction is the same hexagonal arrangement of voxels, and thus the hexagonal coordinate system can be defined and used for each plane. For consistent coordinates between each plane, each plane has its origin placed at a common origin location, and for the purposes of this implementation that origin is as illustrated in **Fig CA1** , the bottom left corner of the voxel cube that makes up the bounds of the voxel volume.

The common hexagonal coordinate systems to use is the cubic and axial coordinate system as illustrated in **Fig VCS 01**.

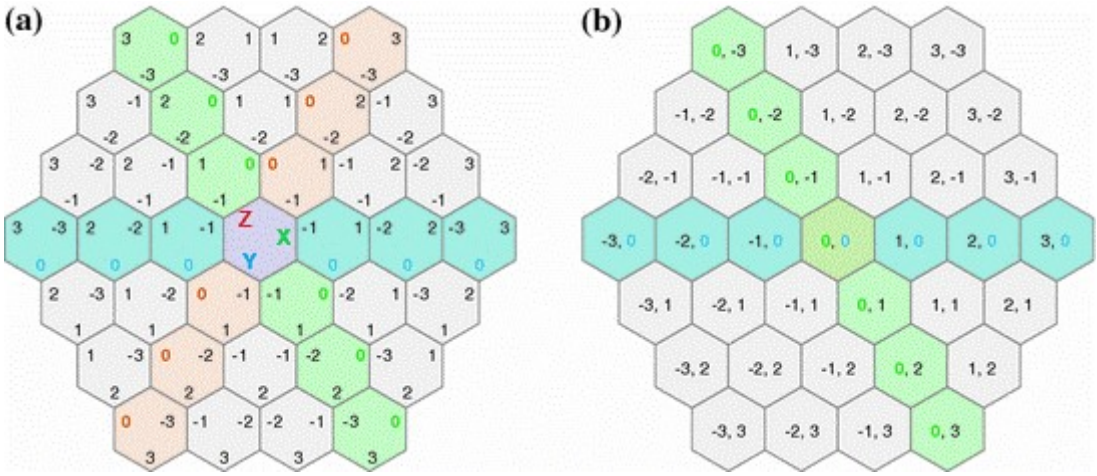


Fig VCS 01

Fig VCS 01a is the cubic coordinate system where three coordinate axis are used and **VCS 01b** is a axial coordinate system.

The cubic coordinate system is what will be used for the voxel coordinate system.

The choice of names for each axis is arbitrary, but to closely associate with the Cartesian coordinate system as close as possible, changes in the left and right voxels or hexagons are designated as the X-axis (indicated by right value within each hexagon) , up and down as the Y-axis, (indicated by bottom value within each hexagon) and the z axis the remaining diagonal axis (indicated by top left value within each hexagon)

These cubic and axial coordinate systems have documentation on their use and methods are commonly known and given in the appendices.

To adopt the cubic hexagonal coordinate system to the 3D voxel system as illustrated in **Fig CA1**, all that is needed is an additional coordinate to indicate which voxel hexagonal plane the voxel exists in. This additional coordinate or axis will be called the W-axis where +W is in the positive Cartesian Z direction, and -W in the negative Cartesian Z direction.

Using the cubic coordinate system, and being consistent with the specification of the faces for each voxel neighbour coordinate corresponding for each voxel face on any W plane is given as in **Table VCST 01**

Face/ Neighbour				
3	x+1	y	z-1	w
4	x+1	y-1	z	w
5	x	y-1	z+1	w
6	x-1	y	z+1	w
7	x-1	y+1	z	w
8	x	y+1	z-1	w

Table VCST 01

It does not matter whether the W layer is an even or an odd layer, the change in coordinate to any neighbour is the same.

To traverse to neighbours from an even to an odd layer, above or below a given voxel coordinate is by using **Fig CA1** as a guide given in **Table VCST02.**

Face/ Neighbour				
0	x	y	z	±w
1	x-1	y	z+1	±w
2	x-1	y+1	z	±w

Table VCST 02 even to odd layer

To traverse to neighbours from an odd to an even layer, above or below a given voxel coordinate is by using **Fig CA1** as a guide given in **Table VCST03.**

Face/ Neighbour				
0	x	y	z	±w
1	x+1	y	z-1	±w
2	x+1	y-1	z	±w

Table VCST 03 odd to even layer

As can be seen, traversing between each layer of the voxel grid and coordinate system obeys the same rule of an ordinary 2D hexagonal cubic coordinate system where the total of the change in coordinates between any two hexagonal cells is zero. Ie x+y+z = 0. It also does not matter if the row on each plane is even or odd.

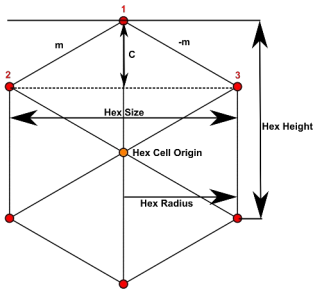
Section 6.0 VOXEL CELLS

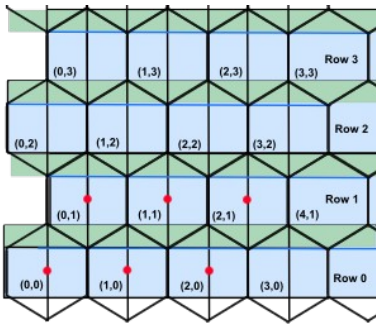
A voxel cell is the region, in the case of 2D voxel the area that the voxel is defined to occupy, or in the case of a 3D voxel, the volume that the voxel is defined to occupy. Thus in the case of using a Trapezo-rhombic dodecahedron (TRD) as to define a voxel geometry in 3D space, the volume that is occupied by a Trapezo-rhombic dodecahedron is a defined voxel cell.

The task given here is to find which TRD cell any given arbitrary 3D coordinate would be defined to be within so as to be able to perform whatever analysis, procedure, graphical display or other function for whatever purpose the user desires.

As a starting point, if one were to look at a matrix of TRD cells or structure from the z axis point of view onto the x-y plane as given in fig CA01, it is seen that each layer of the TRD matrix is in effect a 2D hexagonal grid structure. Thus to find which TRD cell any point P of a given coordinate that lies within it is to first consider that finding which hexagon on a x-y coordinate plane the x,y coordinate of P lies within.

To find which 2D hex cell a point of coordinate component x and y can be done by using the following method. For the hex configuration that is used, a definition of a hex cell attributes can be as is illustrated in fig VC01.





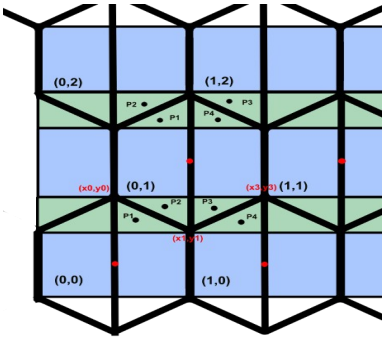


Fig VC01 **Fig VC02** **Fig VC03**

From the definitions defined in for hex height, size etc indicated in Fig VC01, it can be seen that a 2D grid of hexagons can be considered as a grid of rectangles as illustrated in fig VC02 with each odd row having its hex cell origin being offset from the even row by a value of one hex radius on the +x axis, and one hex height on the +y axis. However, there is an over lap of these rectangle indicated by the green color in fig VC02 where a point in this region needs to have a test to be performed on it to see if it is in the hex cell of the row that the rectangle is formed from, or in the neighbouring hex cell above. Such a situation is illustrated in Fig VC03.

Point P1 is within the hex cell of row 0, column 0, ie (0,0). Point P2 is within the hex cell of row 1, column 0, ie (1,0).Point P3 is within the hex cell of row 1, column 0, ie (1,0).Point P4 is within the hex cell of row 0, column 1, ie (0,1). From fig VC03 it can be seen that with reference to the vertex points 1,2 and 3 in fig VC01, point P1 is below the line connecting vertex points 1 and 3, of the hex cell of row 0, column 0, and point P2 is above this line and in the hex cell of row 1, column 0. Similarly, point P3 is above the line connecting vertex points 1 and 2 of hex cell of row 0, column 1 and lies within the hex cell of row 1, column 0, and point P4 is within the hex cell of row 0, column 1. Thus by finding the equation of these lines, and determining if the point P of x,y coordinate is above or below these lines, it can be determined which hex cell the point is in. With this information, an algorithm can be created and code written to find which hex cell a point with an x-y coordinate is in.

Step 1 : Find the relative x,y coordinate of the point P to the x,y coordinate of the hex grid origin cell.

The hex cell of (row, column) = (0,0) has a designated x,y coordinate which is defined as the origin x,y coordinate of the hex grid H. Thus the relative hex x,y coordinate of point P to hex grid origin hex H coordinate is

$$relative\_x = Px - Hx = grid\_x$$
$$relative\_y = Py - Hy = grid\_y$$

Thus grid_x and grid_y are coordinates relative ans local to the voxel matrix origin.

Step 2: Find the row and column that the point P is within.

From grid_y, the hex row that the point P is in can be deduced by taking the integer or floor value of the division of the relative y coordinate divided by the hex cell height. ie

row = floor value of (grid_y/hex height) - EVC-01

Because the column that the point P is in is dependent upon which row it is in as is illustrated in Fig VC02, it needs to be determined if P is in an odd or even row. An odd row is one where the mathematical relationship of row mod 2 = 1 is valid, and an even row if row mod 2 = 0.

ie

row_is_odd = row mod 2 = 1 EVC-02

From fig VC02, if the center of the even row hex cell (0,0) is the location of the hex grid origin in space, then the column that point P can have is a x coordinate being in the range of ± hex radius, to correspond to a column of value 0. Similarly, any even row hex cell with a center x coordinate Cx will have any point P being in that cell if it has an x coordinate of the range Cx ± hex radius. This is the same as with any odd hex cell.

Thus It can easily be seen that similar to the determination of the which row any point P is in, any point P that is in an odd row can have its hex column calculated by the same relationship as EVC-01. Ie

If row_is_odd then
column = floor value of (grid_x/hex size) - EVC-03a

To calculate which column a point P in an even row is in, EVC-03a can be modified to give the column for an even row if a value of the hex radius is added to the x coordinate value of P as it is offset by a value of hex radius to the left of the odd hex row. ie

If row is even then
column = floor value of ((grid_x + Hex radius)/hex size) - EVC-03b

Step 3 : Find if point P is within the hex cell of the calculated row, column given in step 2.

What step 1 and 2 have achieved is to determine the hex row and column that is given in the range from the lower blue line of fig VC03 to the upper green line for each row. If the coordinate P lies with the green zone, it needs to be determined which is the true row-column that P falls into.

To determine if the point P is indeed within the hex cell as defined in step 1 to 2, a final determination needs to be calculated to see if P is below the line or not as illustrated in fig VC03 as discussed above.

To do this, a relative coordinate from P to the hex cell origin (ie hex cell central coordinate) is calculated, and then this relative coordinate is used to determine if this relative coordinate is above or below the line of equation

relative_y = relative_x * m + C where relative_x < 0 EVC-04a
relative_y = -relative_x * m + C where relative_x ≥ 0 EVC-04b

To find the hex origin coordinate O = (Ox,Oy) where O is a local coordinate relative to the voxel matrix origin H defined in step 1;

For odd row
x_coordinate = column*hex_size + hex_radius = Ox EVC-05a
For even row
x_coordinate = column*hex_size = Ox EVC-05b

For both even and odd row

y_coordinate = row*hex_height = Oy EVC-05c

To find the x,y coordinate relative to the center of the hex cell that point P lies within, all that is needed is to subtract Ox from grid_x and Oy from grid_y Ie

relative hex cell x = grid_x – Ox = rel_x EVC-06a
relative hex cell y = grid_y – Oy = rel_y EVC-06b

From the relative value of x calculated in EVC-06a, it can be determined which of EVC-04a and EVC-04b should be used to determine which hex cell a point P exists within. But do do this, the slope m and the y axis intercept C need to be calculated.

In fig 8 of section 3,0 **Finding the HCP Trapezo-rhombic dodecahedral vertex coordinates**, it can be

seen that the slope m is 1/√3 regardless of the size of the hexagon. From this same figure, the value of C is the distance from the hex cell central origin coordinate (0,0) to vertex 1. This distance is given by the length of the side of the equilateral triangle from which the hexagon shape is constructed from. Ie

$$C = \frac{2}{\sqrt{3}} \text{ Hex Size} \qquad \text{EVC-07}$$

From fig VC03,

if P is in an odd row and rel_x ≥ 0 and y_rel ≥ EVC-04b then the column that point P is in is increased by one

and

if P is in an even row and rel_x < 0 and y_rel ≥ EVC-04a then the column that point P is in is decreased by one

and if P is in any row and y_rel ≥ EVC-04b or y_rel ≥ EVC-04a, then the row is increased by one

Otherwise no change is made to the row and column to the hex cell that the point P is determined to be in. Point P is in the blue zone as illustrated in fig VC03.

The psuedo code of this algorithmic method for an even level HCP Trapezo-rhombic dodecahedral matrix is give below in Algorithm EVC-01

Algorithim EVC-01

```
hexagon_coordinates hexagon_cell_coord_from_cartesian(x,y) {
        hex_radius = hex_size / 2.0;
        hex_height = hex_radius * (sqrt(3.0f));

        grid_x = x – Hx;
        grid_y = y – Hy;

        row = floor(grid_y/hex height);

         row_is_odd = row mod 2 == 1;

        If (row_is_odd)
           column = floor(grid_x/hex size);
        else
           column = floor((grid_x + Hex radius)/hex size)

        hex_cell_center_y = row*hex_height;

        if(row_is_odd)
            hex_cell_center_x = column*hex_size+ hex_radius ;
        else
             hex_cell_center_x = column*hex_size;

         rel hex cell x = grid_x – hex_cell_center_x;
         rel hex cell y = grid_y – hex_cell_center_y;

        m = 1/sqrt(3.0);
        c   = hex_radius / (sqrt(3.0f));

        // Work out if the point is above either of the hexagon's top edges
        if (rel hex cell y >= (m * rel hex cell x + 2.0 * c) && rel hex cell x < 0){ // LEFT edge
            row++;
            if (!row_is_odd)
                column--;
        } else {
            if ( rel hex cell y >= (-m * rel hex cell x) + 2.0 * c && rel hex cell x >= 0) { // RIGHT edge
               row++;
               if (row_is_odd)
                   column++;
               }
        }

        return (column,row);
}
```

For an odd layer of the HCP Trapezo-rhombic dodecahedral matrix, the pseudo code code block would be similar to that of algorithm code bloc EVC-01, except that the hex grid would have its origin hex cell displaced from the even level origin hex by the hex radius on the x axis, and by hex_radius/√3 on the y axis as illustrated in fig CA1 in the section Cellular Automata. What is also seen is that the even rows for an odd level are equivalent to the odd rows on an even level, and the same for an odd row on an odd level being equivalent to an even row on an even level. Using the same procedure to form the pseudo code in algorithm CA01, the psuedo code algorithm for an odd level would be as in algorithm EVC-02

Algorithim EVC-02

```
hexagon_coordinates hexagon_cell_coord_from_cartesian(x,y) {
        hex_radius = hex_size / 2.0;
        hex_height = hex_radius * (sqrt(3));

        grid_x = x – (Hx + hex_radius);
        grid_y = y – (Hy + hex_radius/sqrt(3));

        row = floor(grid_y/hex height);

         row_is_odd = row mod 2 == 1;

        If (row_is_odd)
           column = floor((grid_x - Hex radius)/hex size)
        else
           column = floor(grid_x/hex size);

        hex_cell_center_y = row*hex_height;

        if(row_is_odd)
            hex_cell_center_x = column*hex_size;
        else
            hex_cell_center_x = column*hex_size+ hex_radius ;

         rel hex cell x = grid_x – hex_cell_center_x;
         rel hex cell y = grid_y – hex_cell_center_y;

        m = 1/sqrt(3.0);
        c   = hex_radius / (sqrt(3.0f));

        // Work out if the point is above either of the hexagon's top edges
        if (rel hex cell y >= (m * rel hex cell x + 2.0 * c) && rel hex cell x < 0){ // LEFT edge
            row++;
            if (row_is_odd)
                column--;
        } else {
            if ( rel hex cell y >= (-m * rel hex cell x) + 2.0 * c && rel hex cell x >= 0) { // RIGHT edge
                row++;
                if (!row_is_odd)
                    column++;
                }
        }

        return (column,row);
}
```

As can be seen in algorithim EVC-01 and algorithim EVC-02, there are subtle differences in the pseudo code where, because these are relevant to the even and odd layers of the HCP Trapezo-rhombic dodecahedral matrix, there is a kind of symmetrical swapping of some of the statements in the conditions of the code.

The next step is to find which HCP Trapezo-rhombic dodecahedral matrix level a 3D point P of coordinate (x,y,z) exists in so as to be able to determine which of the algorithms EV01 or EV02 to use to determine which voxel cell that point P exists within.

To do this, it can be considered that the levels of a HCP Trapezo-rhombic dodecahedral matrix is simlar to that of the rows of a hex grid. This would mean that there is a zone where it is indeterminate which level P is in due to the shape and orientation of the Trapezo-rhombic dodecahedron geometry that makes up a voxel cell as is illustrated by the green region for a hex grid in fig VC02 and fig VC03. But in this case instead of a line of constant slope that separates each row in this zone, there is a surface of a constant normal that separates each level.

Similarly to a 2D line, a point P can be determined which level it is in by use of finding the distance to the plane surface of the region of the voxel that P exists in. If the distance is negative, it exists within the voxel of the calculated level, and if the distance is positive, it exists within the level above.

To find which HCP Trapezo-rhombic dodecahedral matrix level a point P potentially exists within, the floor value of the z coordinate component of P divided by the voxel height will give this level value. Ie

$$\text{voxel level} = \text{floor}(\text{P z value}/\text{voxel height}) \qquad \text{EVC-08}$$

The distance D of P from a plane with a normal vector N is given by the mathematical relationship

$$D = \frac{\overline{PQ} \bullet N}{|N|} \qquad \text{EVC-09}$$

where $\overline{PQ}$ is the distance vector from point P to point Q that is anywhere on the plane surface, and N is the normal vector of the surface, ● is the dot product and |N| is the vector length of N.

Looking at how these plane surfaces as viewed from above the x-y plane in Fig CA 2a for an even layer, and Fig CA 2b for an odd layer of section 4.0, it can be seen that there are six surfaces that need to be taken into consideration, three for each layer. Also the surfaces are seen to be mirror images across the x axis, and the slope m of the lines defining each surface boundary is $\pm 1/\sqrt{3}$. By comparing the xy coordinates of P with the potential HCP Trapezo-rhombic dodecahedral matrix level that it exists in, a determination of which voxel surface region can be made and thus a determination can be made if it is within the nominated voxel or the one above it.

As with determining which hex cell row that P exists within as previously by use of a relative xy coordinate to the hex cell central origin, the same can be applied to determine which of these hex surface regions P is within.

So if P is defined to potentially be in an even level, then by using Fig CA 2b,

if x_relative is ≥ 0 and y_relative > -m * x_relative P is within region 0 of even level EVC-10a
if y_relative ≤ -m * x_relative and y_relative < m * x_relative P is within region 1 of even level EVC-10b
if x_relative is < 0 and y_relative ≥ m * x_relative P is within region 2 of even level EVC-10c

Normal vector of each of these regions is the vector from the voxel cell origin that P is nominated to be within, to the neighbouring voxel cell origin that the surface that defines the designated region Ie

$$N = \left(1, \frac{1}{\sqrt{3}}, 2\frac{\sqrt{6}}{3}\right) \quad \text{for region 0 of even level} \qquad \text{EVC-11a}$$

$$N = \left(0, -\frac{1}{\sqrt{3}}, 2\frac{\sqrt{6}}{3}\right) \quad \text{for region 1 of even level} \qquad \text{EVC-11b}$$

$$N = \left(-1, -\frac{1}{\sqrt{3}}, 2\frac{\sqrt{6}}{3}\right) \quad \text{for region 2 of even level} \qquad \text{EVC-11c}$$

For an odd level,

if x_relative is < 0 and y_relative ≤ -m * x_relative P is within region 0 of even level EVC-12a
if y_relative ≥ -m*x_relative and y_relative > m*x_relative P is within region 1 of even level EVC-12b
if y_relative is ≤ m*x_relative and x_relative > 0 P is within region 3 of even level EVC-12c

the normal vectors are

$$N = \left(-1, -\frac{1}{\sqrt{3}}, 2\frac{\sqrt{6}}{3}\right) \quad \text{for region 0 of odd level} \qquad \text{EVC-13a}$$

$$N = \left(0, \frac{1}{\sqrt{3}}, 2\frac{\sqrt{6}}{3}\right) \quad \text{for region 1 of odd level} \qquad \text{EVC-13b}$$

$$N = \left(1, \frac{1}{\sqrt{3}}, 2\frac{\sqrt{6}}{3}\right) \quad \text{for region 2 of odd level} \qquad \text{EVC-13c}$$

and thus in each case the normal vector length is always the same.

To obtain the distance vector $\overline{PQ}$ the same point Q can be used in all cases as the vertex point 0 of fig 8 in the section 3.0, Finding the HCP Trapezo-rhombic dodecahedral vertex coordinates shows this point is

shared by all plane surfaces. Thus Q = (0,0,√3/2).

With this information, an algorithm can be constructed to determine which level a point P exists within, and in turn, which voxel of coordinate column, row, and level P exists within.

The Voxel height is the distance between the vetex 0 and vertices 1,3 and 5 which is

voxel height = voxel radius*(sqrt(3/2) + sqrt(6)/3) EVC-14

A voxel matrix grid value for the z coordinate of P is calculated as are the grid_x and grid_y values for a Hex grid. Ie

grid_z = Pz – Hz EVC-15

and in the same manner

level = floor(grid_z/voxel height) EVC-16

and

level_is even = level mod 2 == 0 EVC-17

A psuedo code to find the 3D HCP Trapezo-rhombic dodecahedral voxel cell a point P exists within is defined in algorithm EVC-03.

Algorithim EVC-03

```
voxel_coordinate hexagon_cell_coord_from_cartesian(x,y,z) {
        voxel_radius = voxel_size/2.0;
        voxel_height = voxel_radius*(sqrt(3/2) + sqrt(6)/3);

        grid_z = z-voxel_origin.z;

        if(grid_z < -sqrt(6)/3)
            level = (int)((grid_z- voxel_height) / voxel_height);// take account of Pz below  qrt(6)/3
        else
            level = (int)(grid_z / voxel_height);

        level_is_even = level mod 2 ==0;

        if (level_is_even){
           grid_x = x-voxel_origin.x;
           grid_y = x-voxel_origin.y;
           return find_voxel_cell_coordinate in even level (grid_x,grid_y,grid_z,voxel_height,level);
        } else{
           grid_x = x-(voxel_origin.x+voxel_radius);
           grid_y = x-(voxel_origin.y+voxel_radius/aqrt(3));
           return find_voxel_cell_coordinate in odd level (grid_x,grid_y,grid_z,voxel_height,level);
        }
}
```

A final consideration in determining which voxel cell that a point P exists in is if it is deemed that the point P is above the surface of the Trapezo-rhombic dodecahedron voxel that it is defined to be above, then not only will the point P be in the level above the current nominated voxel level coordinate, but also have its row or column coordinate change as well as it is in the neighbouring voxel geometry volume. Table TCA1 of the section Cellular Automata indicates the changing of coordinates from any designated voxel to another. The top neighbours being indexed as the 0,1 and 2 neighbours corresponding to the same numbered surfaces.

So when forming an algorithm to determine the voxel coordinate of a point P in a voxel space, if it is found that P is within an undefined zone of a voxel level, and is above one of the surfaces of the nominated voxel of coordinate column, row, level (i,j,k), then P will be defined to be within the neighbour that the surface it is above and the voxel coordinate change as defined by Table TCA1 of section 4.0. Algorithm EVC-04 does this for an even level, and Algorithm EVC-05 for an odd nominated level that P exists within.

To envisage the regions of a Trapezo-rhombic dodecahedron voxel that fall within these ranges of uncertainty that need to be tested to find which true voxel cell a point P falls into, the image of a rapezo-rhombic dodecahedron voxel given in fig 1 of section 1 would have a plane going through vertices 6,2,8,12 for the xy plane, and 2,4,6 for the z axis. Fig VC04 illustrates such regions, where the green color represents the regions of uncertainty, and blue where a determination of which voxel cell a point P is located is definite. Note that fig VC04 has the lower region of the voxel surface that any testing is absent.

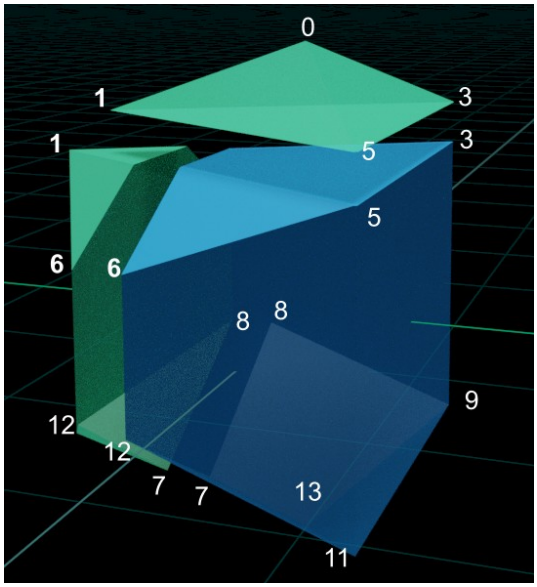


Fig VC04

The C++ implementation code that utilises these algorithms is exits within the voxel data storage class of the voxel module source code. Further documentation of this implementation and use of the functions are given in the vw_appl_imgui documentation.

Algorithim EVC-04

```
voxel_coordinate  find_voxel_cell_coordinate in even level (grid_x,grid_y,grid_z,voxel_height,level){
        hex size     = voxel_size;
        hex_radius = voxel_size / 2.0;
        hex_height = voxel_radius * (sqrt(3.0f));

        if (grid_y < -1.0 / sqrt(3.0))
            row = (int)((grid_y - hex_height) / hex_height);
        else
           row = (int)(grid_y / hex_height);

         row_is_odd = abs(row mod 2) == 1;

        If (row_is_odd)
           column = floor(grid_x/hex size);
        else
           column = floor((grid_x + Hex radius)/hex size)

        hex_cell_center_y = row*hex_height;

        if(row_is_odd)
            hex_cell_center_x = column*hex_size+ hex_radius ;
        else
             hex_cell_center_x = column*hex_size;

         rel hex cell x = grid_x – hex_cell_center_x;
         rel hex cell y = grid_y – hex_cell_center_y;
         rel hex cell z = grid_z - (level * voxel_height);

        m = 1/sqrt(3.0);
        c   = hex_radius / (sqrt(3.0f));

        // Work out if the point is above either of the hexagon's top edges
        if (rel hex cell y >= (m * rel hex cell x + 2.0 * c) && rel hex cell x < 0){ // LEFT edge
            row++;
            if (!row_is_odd){
                column--;
                row_is_odd = true; // P is in an odd row hex
                rel_x = (grid_x - (column * voxel_size)) - grid_radius; //calculate new relative x
            } else
                rel_x = rel_x + grid_radius;//calculate new relative x

            rel_y = grid_y - (row * grid_height);//calculate new relative y
        } else {
            if ( rel hex cell y >= (-m * rel hex cell x) + 2.0 * c && rel hex cell x >= 0) { // RIGHT edge
               row++;
               if (row_is_odd){
                   column++;
                   row_is_odd = false;// P is in an even row hex
                    rel_x = grid_x - (column * voxel_size);//calculate new relative x
                } else
                    rel_x = rel_x -  grid_radius//calculate new relative x

                rel_y = grid_y - (row * grid_height);//calculate new relative y
            }
        }

        // define voxel cell in regards to the z coordinate value and the voxel matrix attributes
        pq_vector     = { rel_x,rel_y,rel_z – sqrt(3/2) };
        m0 = -1 / sqrt(3);//slope of line in xy plane from point point 6 to point 3
        m1 =  1 / sqrt(3);//slope of line in xy plane from point point 5 to point 2

        if (rel_x >= 0 && rel_y > m0 * rel_x) {// region 0
          normal_vector = { 1.0f,1.0f / sqrt(3.0f),(2.0 * sqrt(6.0f)) / 3.0f };

          normal_vector_length = normal_vector.length();
          normal_dot_pq          = pq_vector dot normal_vector;
```

```
            distance_to_plane = normal_dot_pq / normal_vector_length;

            if (distance_to_plane < 0)
               voxel_coord.z = level;
            else {
               voxel_coord..x++;
               voxel_coord.z = level + 1;
            }

            return voxel_coord;
         }// end region 0

         if (rel_y <= m0*rel_x && rel_y < m1 * rel_x) {// region 1
            normal_vector = { 0.0f,-1.0f / sqrt(3.0f),(2.0 * sqrt(6.0f)) / 3.0f };

            normal_vector_length = normal_vector.length();
            normal_dot_pq        = pq_vector dot normal_vector;

            distance_to_plane = normal_dot_pq / normal_vector_length;

            if (distance_to_plane < 0)
               voxel_coord.z = level;
            else {
                  voxel_coord.y--;
                  voxel_coord.z = level + 1;

                  return voxel_coord;
         }// end region 1


         if (rel_x < 0 && rel_y >= m1 * rel_x) {// region 2
            normal_vector = {-1.0f,-1.0f / sqrt(3.0f),(2.0 * sqrt(6.0f)) / 3.0f };

            normal_vector_length = normal_vector.length();
            normal_dot_pq        = pq_vector dot normal_vector;

            distance_to_plane = normal_dot_pq / normal_vector_length;

            if (distance_to_plane < 0)
                voxel_coord.z = level;
                else {
                    voxel_coord.z = level + 1;
                    if (!row_is_odd) voxel_coord.x--;
                }
                    return voxel_coord;
             }
         }// end region 2
}//  end find_voxel_cell_coordinate in even level
```

Algorithim EVC-05

```
voxel_coordinate  find_voxel_cell_coordinate in odd level (grid_x,grid_y,grid_z,voxel_height,level){
        hex size     = voxel_size;
        hex_radius = voxel_size / 2.0;
        hex_height = voxel_radius * (sqrt(3.0f));

        if (grid_y < -1.0 / sqrt(3.0))
            row = (int)((grid_y - hex_height) / hex_height);
        else
           row = (int)(grid_y / hex_height);

         row_is_odd = abs(row mod 2) == 1;

        column = floor((grid_x + Hex radius)/hex size);

        hex_cell_center_y = row*hex_height;

        if(row_is_odd)
            hex_cell_center_x = column*hex_size - hex_radius ;
        else
             hex_cell_center_x = column*hex_size;

         rel hex cell x = grid_x – hex_cell_center_x;
         rel hex cell y = grid_y – hex_cell_center_y;
         rel hex cell z = grid_z - (level * voxel_height);

        m = 1/sqrt(3.0);
        c   = hex_radius / (sqrt(3.0f));

        // Work out if the point is above either of the hexagon's top edges
        if (rel hex cell y >= (m * rel hex cell x + 2.0 * c) && rel hex cell x < 0){ // LEFT edge
            row++;
            if (row_is_odd){
                column--;
                row_is_odd = false; // P is in an odd row hex
                rel_x = (grid_x - (column * voxel_size)) – grid_radius;//calculate new relative x
            } else
                rel_x = rel_x + grid_radius;//calculate new relative x

            rel_y = grid_y - (row * grid_height);//calculate new relative y
        } else {
            if ( rel hex cell y >= (-m * rel hex cell x) + 2.0 * c && rel hex cell x >= 0) { // RIGHT edge
               row++;
               if (!row_is_odd){
                   column++;
                   row_is_odd = true;// P is in an even row hex
                   rel_x = grid_x - (column * voxel_size);
                } else
                   rel_x = rel_x – grid_radius;//calculate new relative x

                rel_y = grid_y - (row * grid_height);//calculate new relative y
             }
        }

        // define voxel cell in regards to the z coordinate value and the voxel matrix attributes
        pq_vector     = { rel_x,rel_y,rel_z – sqrt(3/2) };
        m0 = -1 / sqrt(3);//slope of line in xy plane from point point 6 to point 3
        m1 =  1 / sqrt(3);//slope of line in xy plane from point point 5 to point 2

        if (rel_x <= 0 && rel_y < m0 * rel_x) {// region 0
           normal_vector = { -1.0f,-1.0f / sqrt(3.0f),(2.0 * sqrt(6.0f)) / 3.0f };

           normal_vector_length = normal_vector.length();
           normal_dot_pq           = pq_vector dot normal_vector;

           distance_to_plane = normal_dot_pq / normal_vector_length;

           if (distance_to_plane < 0)
               voxel_coord.z = level;
```

```
            else {
                if(!row_is_odd) voxel_coord.x--;
                voxel_coord.z = level + 1;
            }

            return voxel_coord;
        }// end region 0

        if (rel_y >= m0*rel_x && rel_y > m1 * rel_x) {// region 1
            normal_vector = { 0.0f,1.0f / sqrt(3.0f),(2.0 * sqrt(6.0f)) / 3.0f };

            normal_vector_length = normal_vector.length();
            normal_dot_pq        = pq_vector dot normal_vector;

            distance_to_plane = normal_dot_pq / normal_vector_length;

            if (distance_to_plane < 0)
                voxel_coord.z = level;
            else {
                    voxel_coord.y++;
                    voxel_coord.z = level + 1;

                    return voxel_coord;
        }// end region 1


        if (rel_x > 0 && rel_y <= m1 * rel_x) {// region 2
            normal_vector = {1.0f,1.0f / sqrt(3.0f),(2.0 * sqrt(6.0f)) / 3.0f };

            normal_vector_length = normal_vector.length();
            normal_dot_pq        = pq_vector dot normal_vector;

            distance_to_plane = normal_dot_pq / normal_vector_length;

            if (distance_to_plane < 0)
                voxel_coord.z = level;
                else {
                    voxel_coord.z = level + 1;
                    if (!row_is_odd) voxel_coord.x++;
                }
                    return voxel_coord;
            }
        }// end region 2
}//  end find_voxel_cell_coordinate in odd level
```