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.

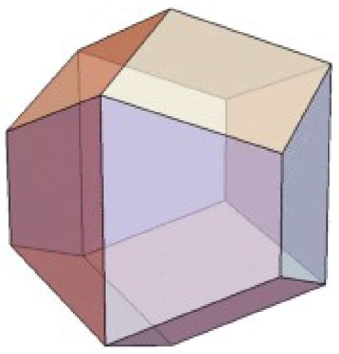
The third task is how to display the voxel value, volume or surface onto the computer screen.

Solution:

Hexagonal Close Packing Geometry to represent 3D Voxel Space

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, where each voxel can be represented as a Trapezo-rhombic dodecahedral polygon shape,(fig 1)



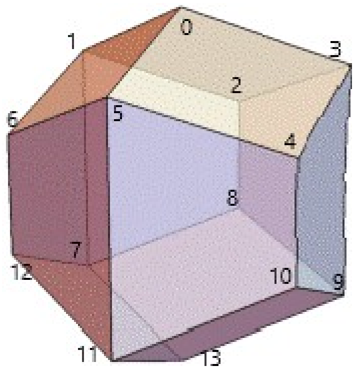


fig 1 vertex indexes

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 neighbouring sphere would touch itself, 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.
Configuration two : the direction to move in the y direction is in a straight line pattern, and on the x is staggered.

Configuration one is chosen. 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

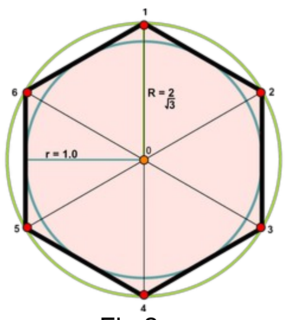


Fig 2

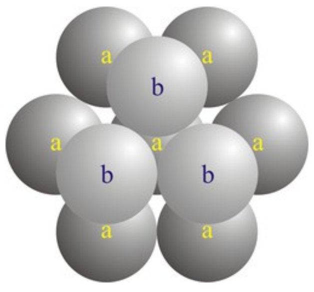


Fig 3a

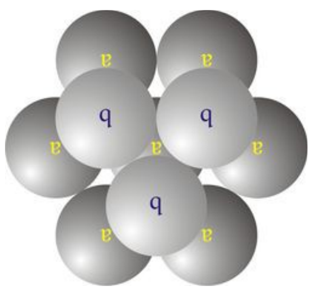


Fig 3b

The convention use in constructing the Trapezo-rhombic dodecahedral shape 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 a 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

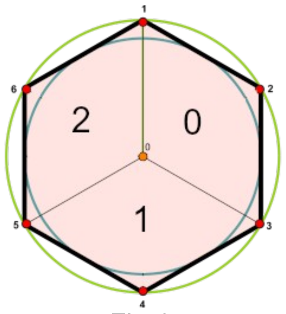


Fig 4a

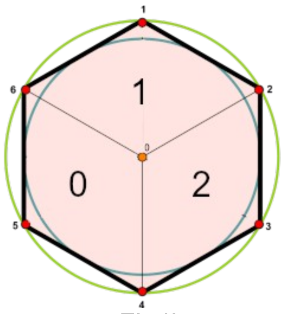


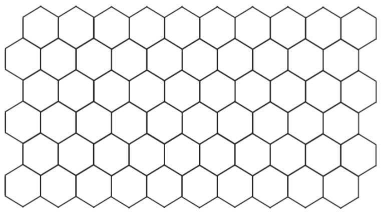
Fig4b

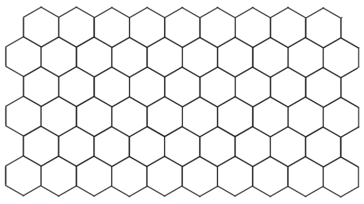
With this specification of the Trapezo-rhombic dodecahedral to be used, the z coordinates of each vertex can be found.

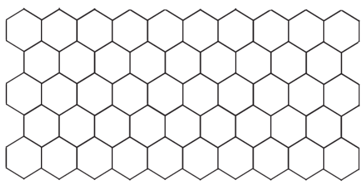
HCP Trapezo-rhombic dodecahedral voxel data storage.

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.


10I,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 will also 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 60° 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.

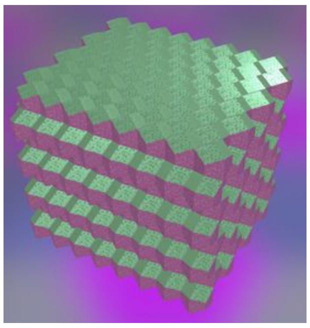


Fig 6

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 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;
```

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. Thus the coordinates of the voxel centres correspond to the coordinates of a HCP lattice which are known and can be used here. For the definition defined above 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 (radius) r the coordinate of the I,j,k voxel center is given by

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

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, or x,y index coordinates of the centres of the voxels 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..

So 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 of the voxel matrix is crucial in finding this index value.

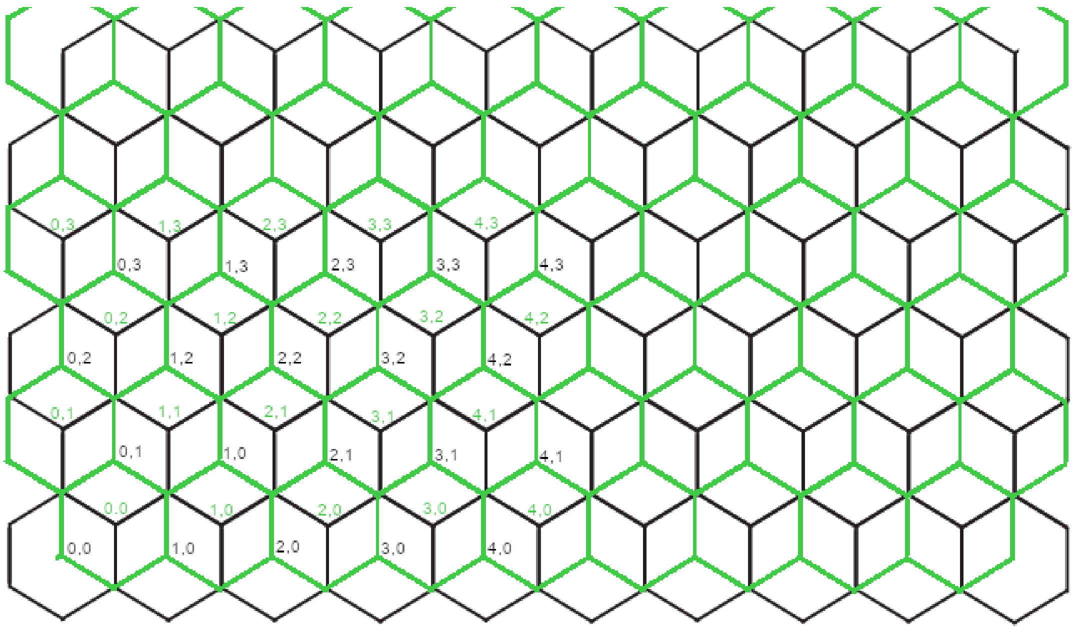


Fig 7

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 vi of the even k = 0 level of a voxel matrix of x dimension x_dim and y dimention 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 the the i coordinate added to this total..

To do this, the number of even rows is (integer (j / 2) + j mod 2)x_dim, and the number of odd rows is (integer (j / 2))(x_dim-1). Thus the number of voxels of the coordinate i, j on an even z layer is

vi = (integer (j / 2) + j mod 2)x_dim + (integer (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, and thus the number of voxels of the coordinate i, j on an odd z layer is

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

And putting this all together a more general calculation of the index count on the k level of the voxel matrix is

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

For any voxel coordinate (i,j,k) the number of voxels before the k layer needs to be calculated and **V3** added to that total to give the correct vector array index vi.

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 = (integer (y_dim/ 2) + y_dim mod 2)x_dim + (integer (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 = (integer (y_dim-1/ 2) + y_dim-1 mod 2)(x_dim-1) + (integer (y_dim-1) / 2))x_dim - **V5**

Putting V4 and V5 together gives

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

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

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);
}
```

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 the section **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 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 in the row being iterated and dim_y is the number or rows in the z layer being iterated.

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, the formula

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

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 is

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

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 are of the same z value as where the central sphere touches the spheres above it, which thus 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

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

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

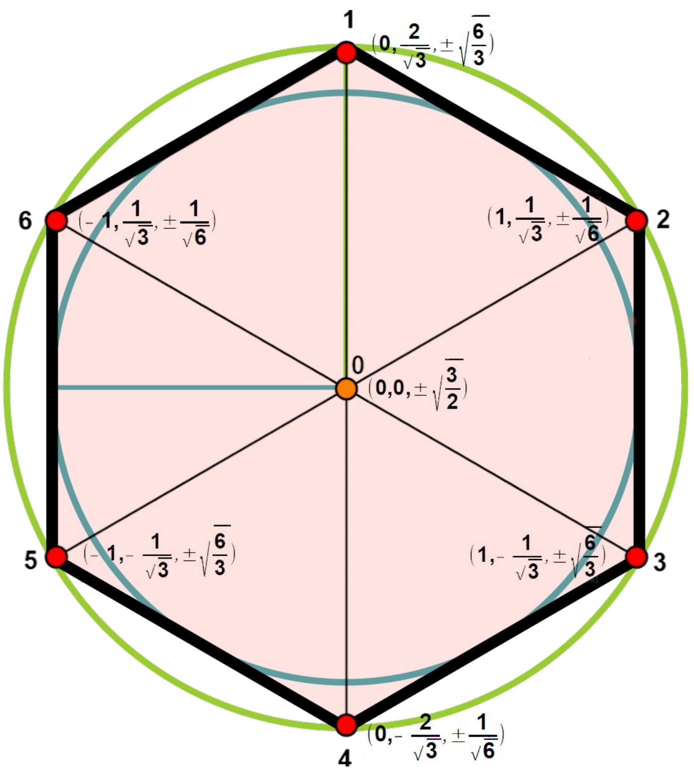
for vertex 2,4, and 6 it is found that

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

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.

Thus 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.



1
$(0,\frac{2}{\sqrt{3}},\pm\sqrt{\frac{6}{3}})$
2
$(1,\frac{1}{\sqrt{3}},\pm\frac{1}{\sqrt{6}})$
6
$(-1,\frac{1}{\sqrt{3}},\pm\frac{1}{\sqrt{6}})$
0
$(0,0,\pm\sqrt{\frac{3}{2}})$
5
$(-1,-\frac{1}{\sqrt{3}},\pm\sqrt{\frac{6}{3}})$
3
$(1,-\frac{1}{\sqrt{3}},\pm\sqrt{\frac{6}{3}})$
4
$(0,-\frac{2}{\sqrt{3}},\pm\frac{1}{\sqrt{6}})$

Fig 8.

Results:

Conclusion:

References:

Cellular Automata

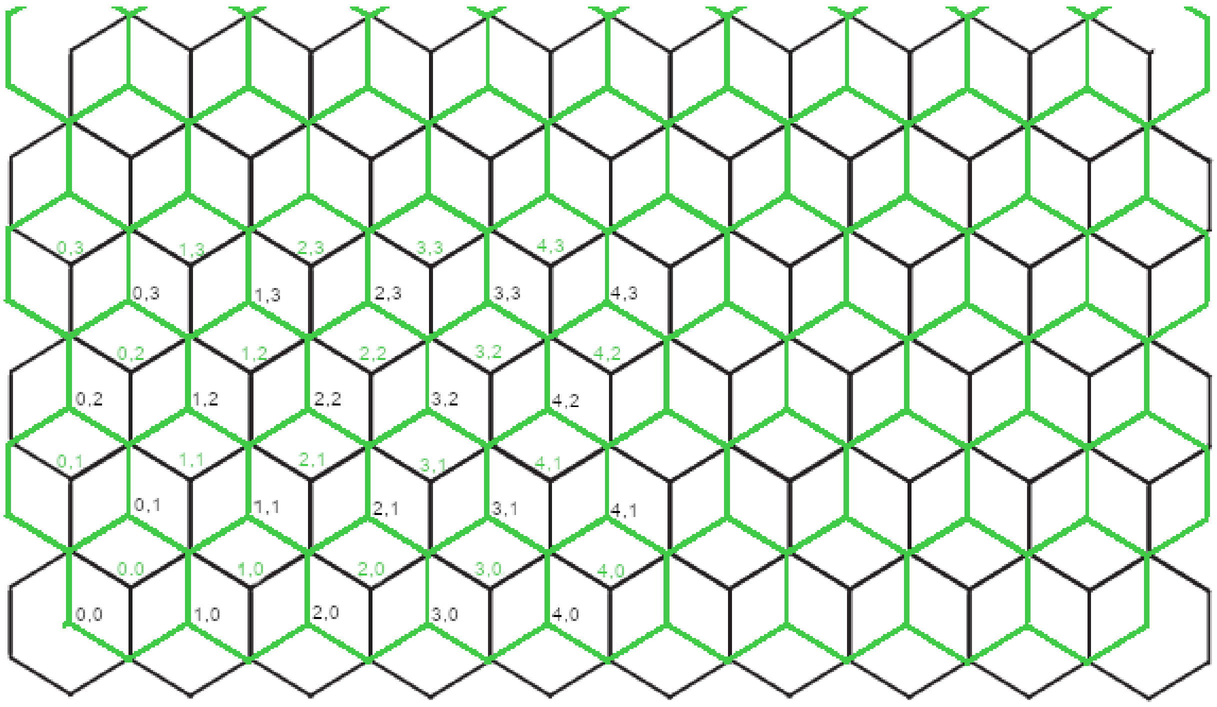


Fig CA1

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.

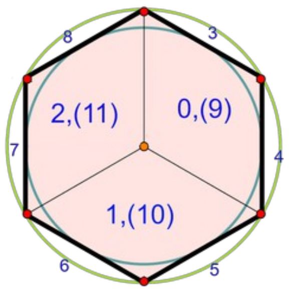


Fig CA 2a Even layer face id

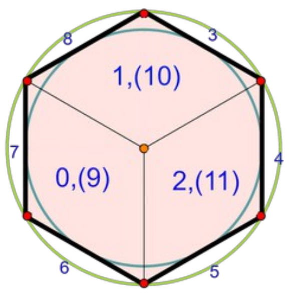


Fig CA 2b Odd layer face id

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

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.

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.

VOXEL COORDINATE SYSTEM

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**.

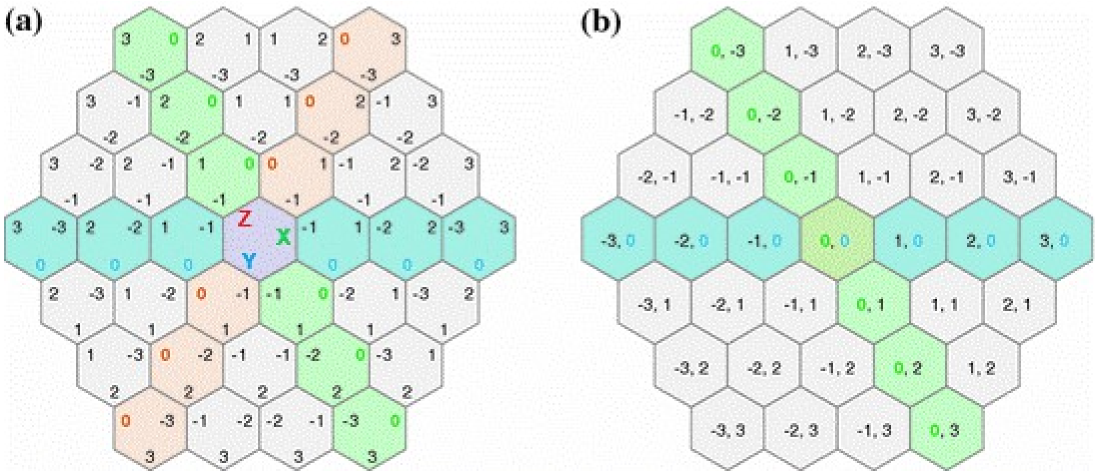


Fig VCS 01

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.