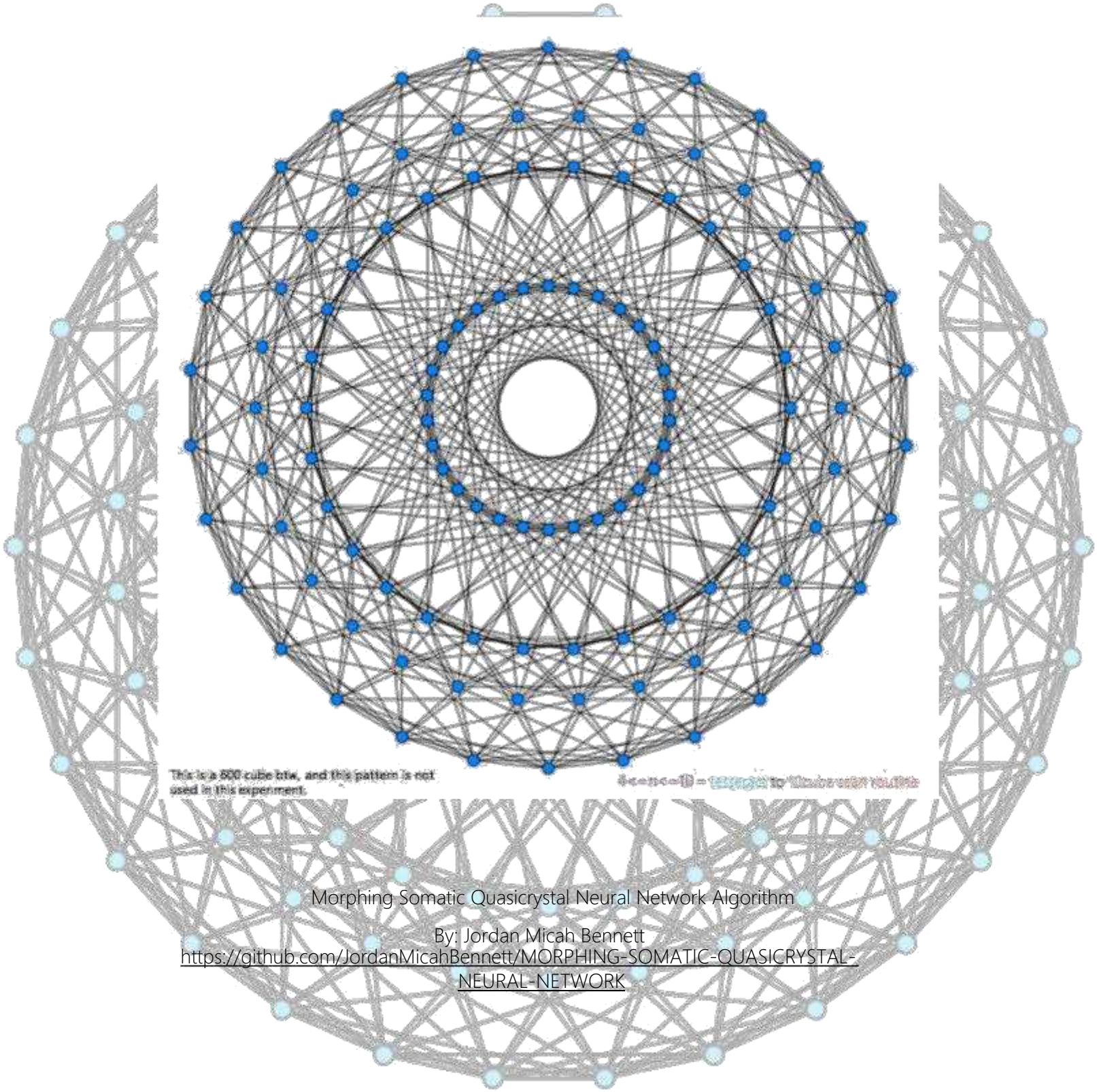


Morphing Somatic Quasicrystal Neural Network Algorithm

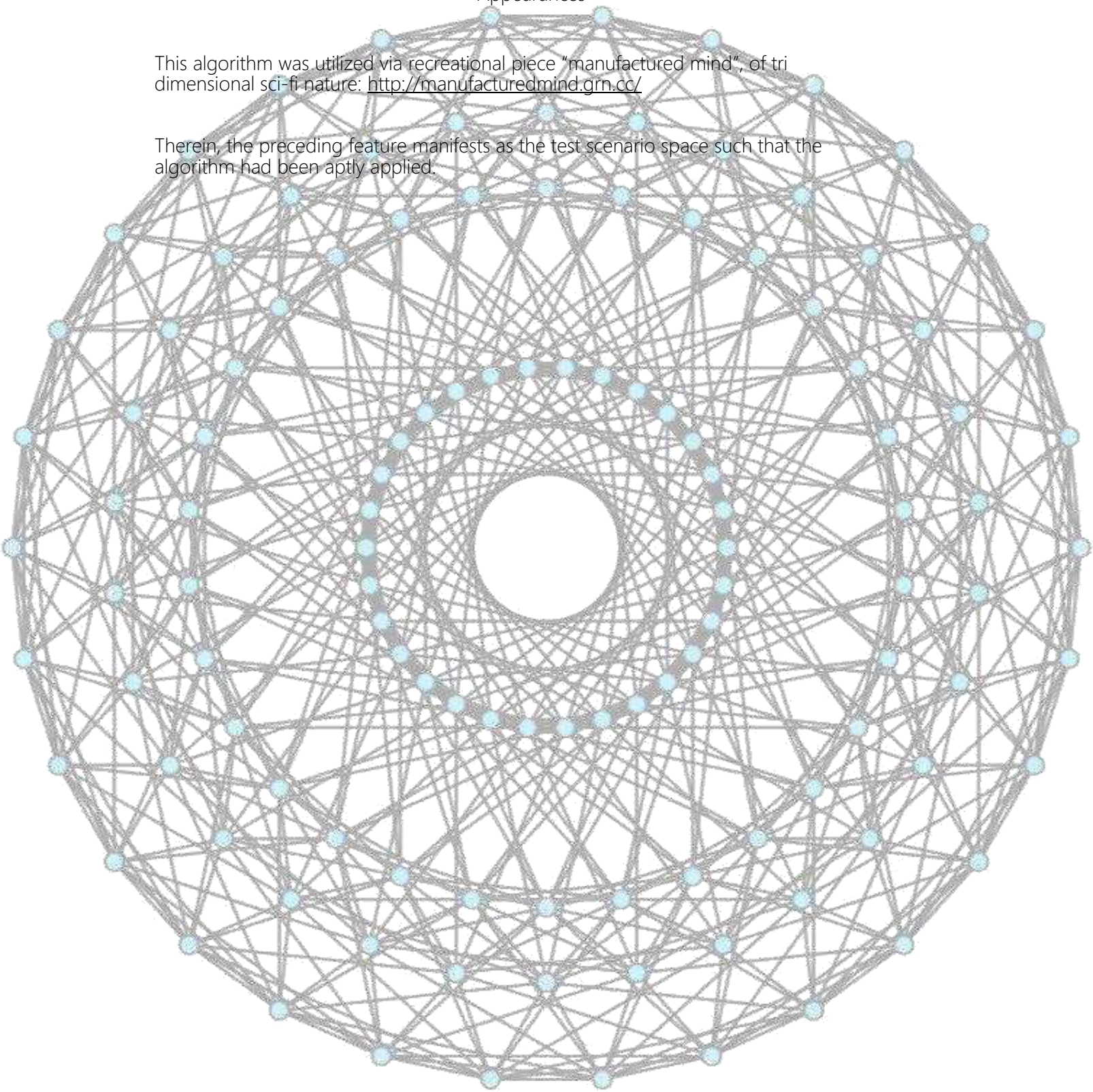
By: Jordan Micah Bennett

https://github.com/JordanMicahBennett/MORPHING-SOMATIC-QUASICRYSTAL-NEURAL-NETWORK

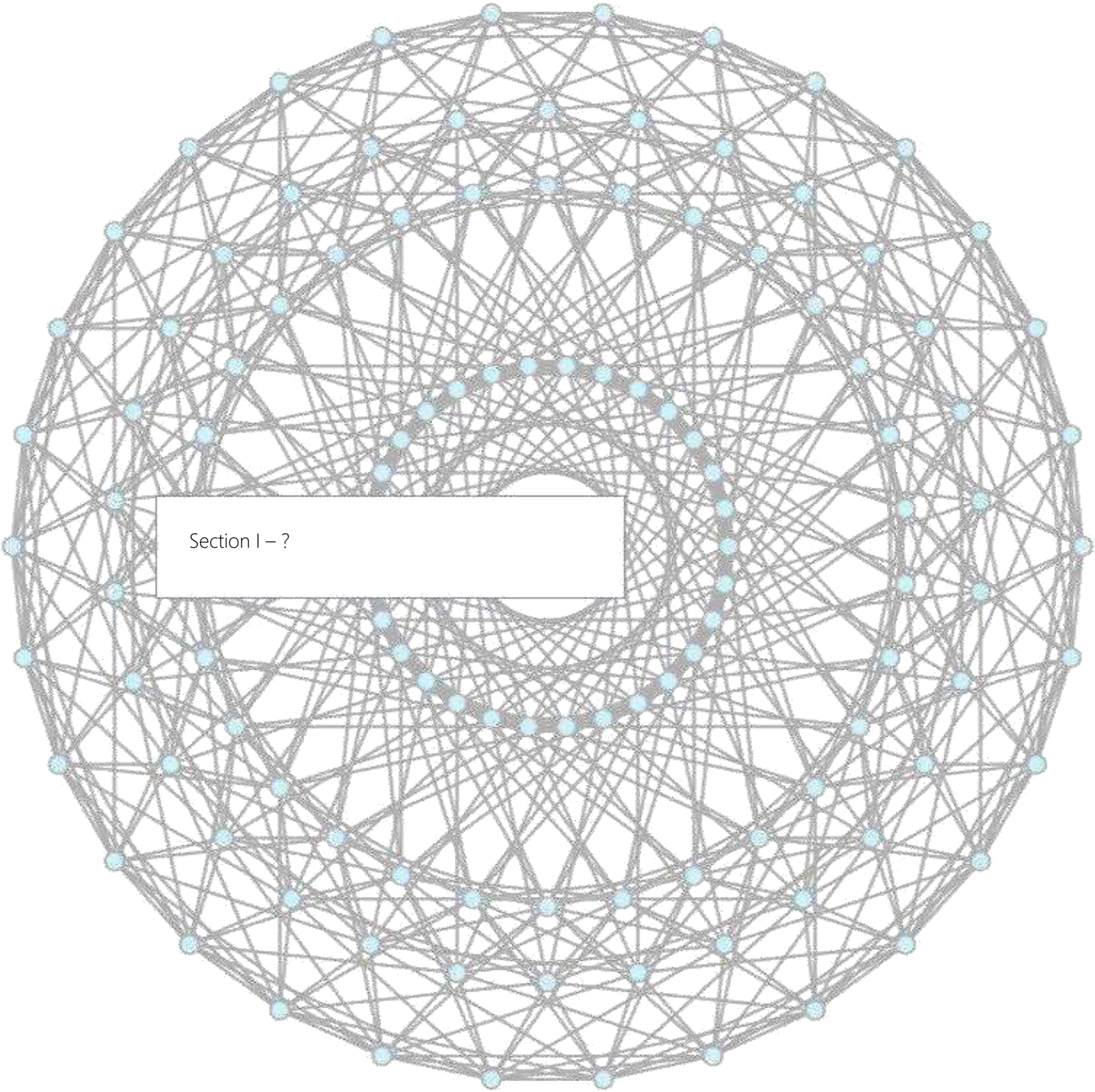
[0]Appearances

This algorithm was utilized via recreational piece "manufactured mind", of tri dimensional sci-fi nature: http://manufacturedmind.grn.cc/

Therein, the preceding feature manifests as the test scenario space such that the algorithm had been aptly applied.



Contents


Section I – ?

Introduction

The candidate AI is strictly centered upon search/rescue scan behaviour, where generally, actors (search agents) coordinate and span so as to recover/retrieve some item/entity. *See Reference 6.0*

Generally, activities encompassing search rescue/scan beaviourism include:

i. Locating missing hikers, persons

ii. Searching through debris after an earthquake

iii. Searching for lost aircraft or ships at sea and treasure hunting

As such, search process/problem encompasses:

i. Identifying the search area,

ii. Prompt application of systematic search pattern, with the highest probability of finding.

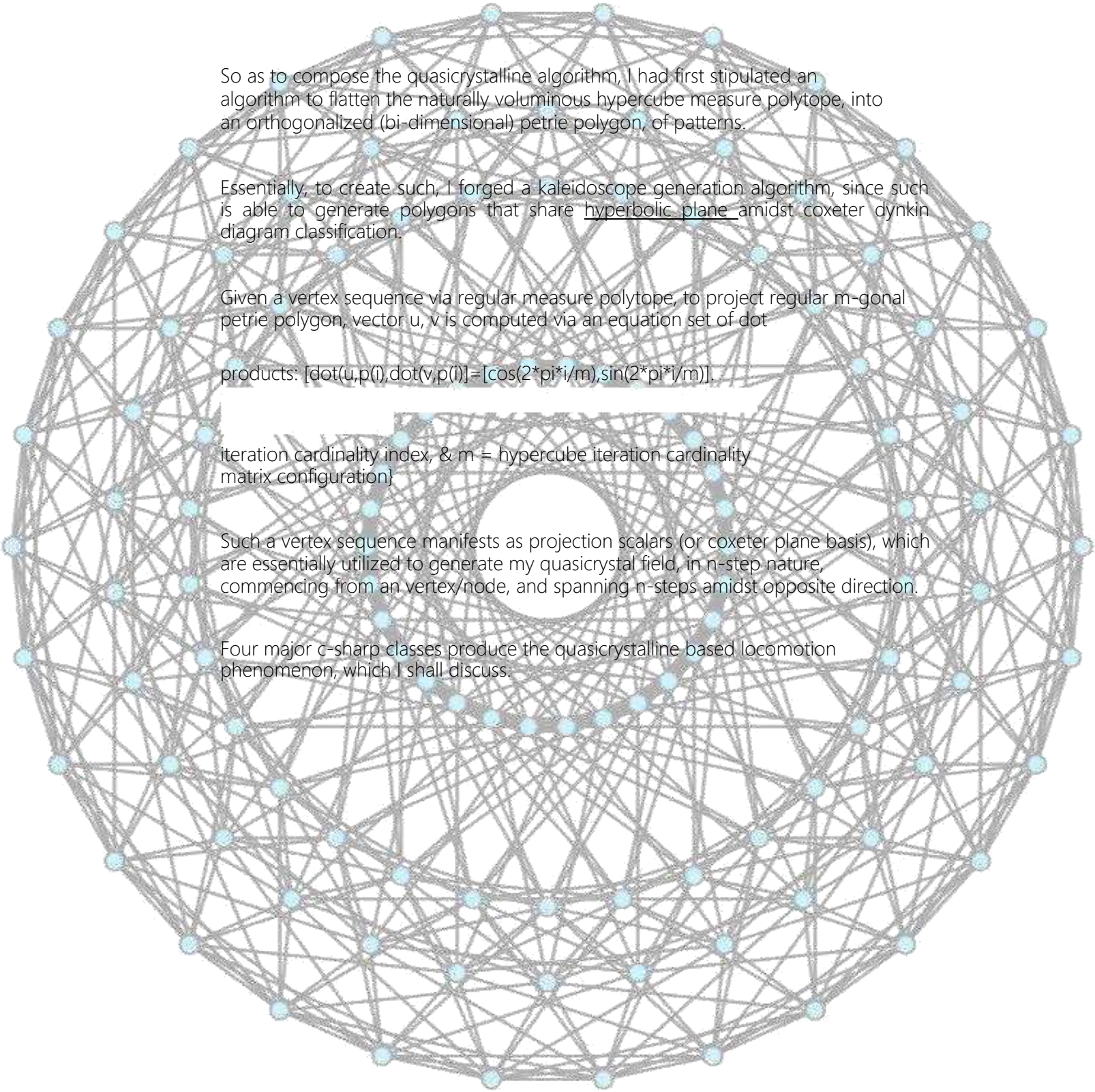
My AI locomotion algorithm exists as a measure to generate more efficient, more interesting and by extension easily implemented, dynamic AI locomotion/area scan mannerism. *See contents 5.1*

The problem

Though search/rescue algorithms exist, I detect means via which great enhancement is achievable. *See contents 5.1*

Though my algorithm provides a great segment of the complete solution I have intended to provide, I plan to implement two additional phases, to bolster such. (whereby search/scan efficacy amidst real agents could be improved by over 10%)*See contents 5.1*

Code Discussion



So as to compose the quasicrystalline algorithm, I had first stipulated an algorithm to flatten the naturally voluminous hypercube measure polytope, into an orthogonalized (bi-dimensional) petrie polygon, of patterns.

Essentially, to create such, I forged a kaleidoscope generation algorithm, since such is able to generate polygons that share hyperbolic plane amidst coxeter dynkin diagram classification.

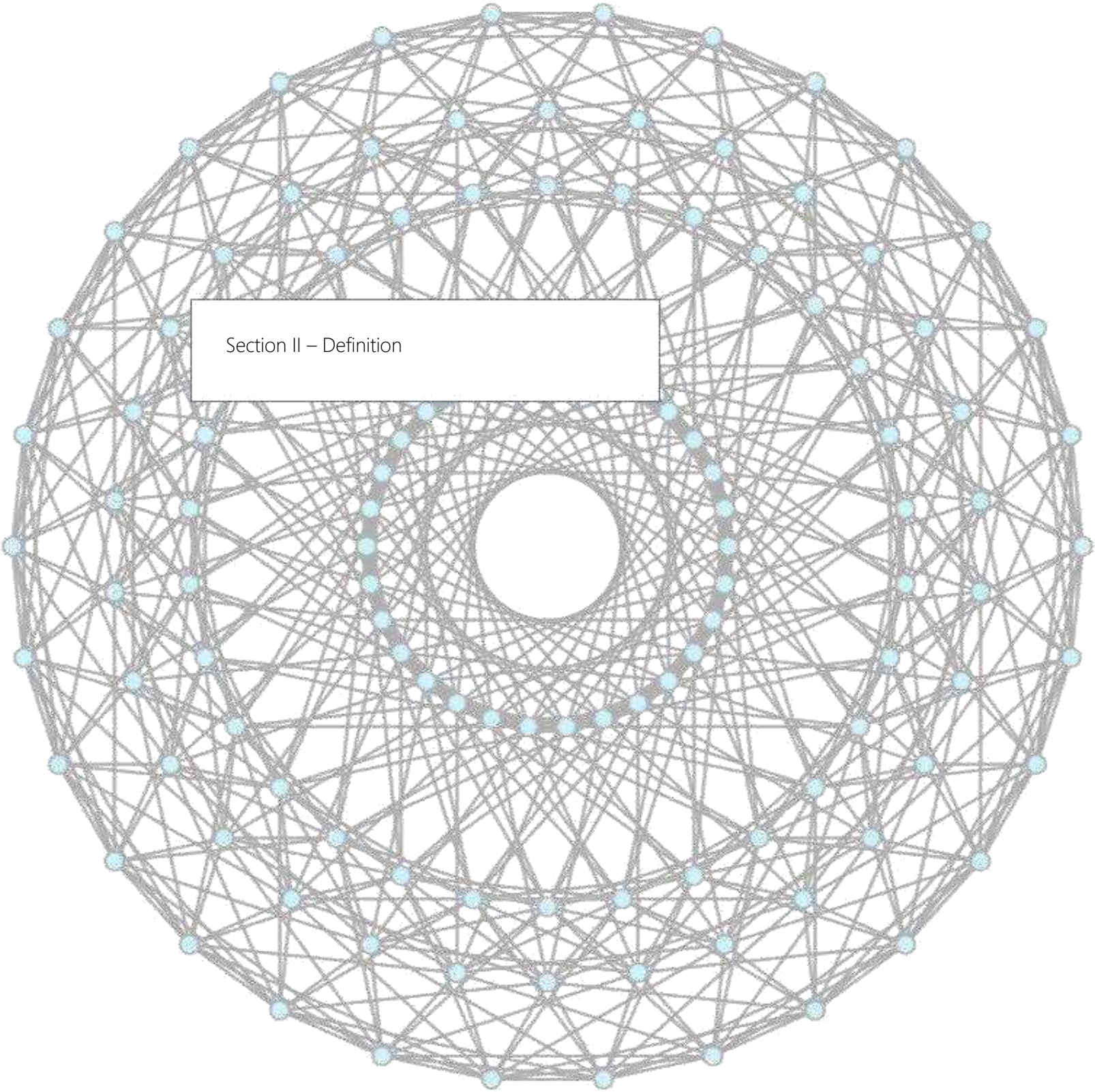
Given a vertex sequence via regular measure polytope, to project regular m-gonal petrie polygon, vector u, v is computed via an equation set of dot

products: [dot(u,p(i),dot(v,p(i)]=[cos(2*pi*i/m),sin(2*pi*i/m)].

iteration cardinality index, & m = hypercube iteration cardinality
matrix configuration}

Such a vertex sequence manifests as projection scalars (or coxeter plane basis), which are essentially utilized to generate my quasicrystal field, in n-step nature, commencing from an vertex/node, and spanning n-steps amidst opposite direction.

Four major c-sharp classes produce the quasicrystalline based locomotion phenomenon, which I shall discuss.



Section II – Definition

Definition

Whence computer science's memory management adroitly segments processes into workable chunks, in the synonymous like, my morphing quasicrystal somatic neural network algorithm splices the whole that is sample search space, therein originating morphing somatic nodes amidst saids central vectoring, thereby sequentially expanding amid the edge of such, in tandem with the equilibrium of space of pattern distribution emergent via.

Morphing quasicrystal somatic neural network algorithm is not quite the quotidian (quintessential) neural network. I approached the issue rather analogously; there exist no weights; each agent somatic-ally manifests as a unit node, wherein the quasicrystal field mechanism roles as the hidden layer that appropriately issues locomotion instructions. Verisimultiply (seemingly) amorphous, each agent possesses solely the ability to perceive target entities trivially, together with vectors that manifest as the attributes receiving instructions via the hidden field locomotion mechanism that induces adroit agent navigation potentiality.

Advantages:

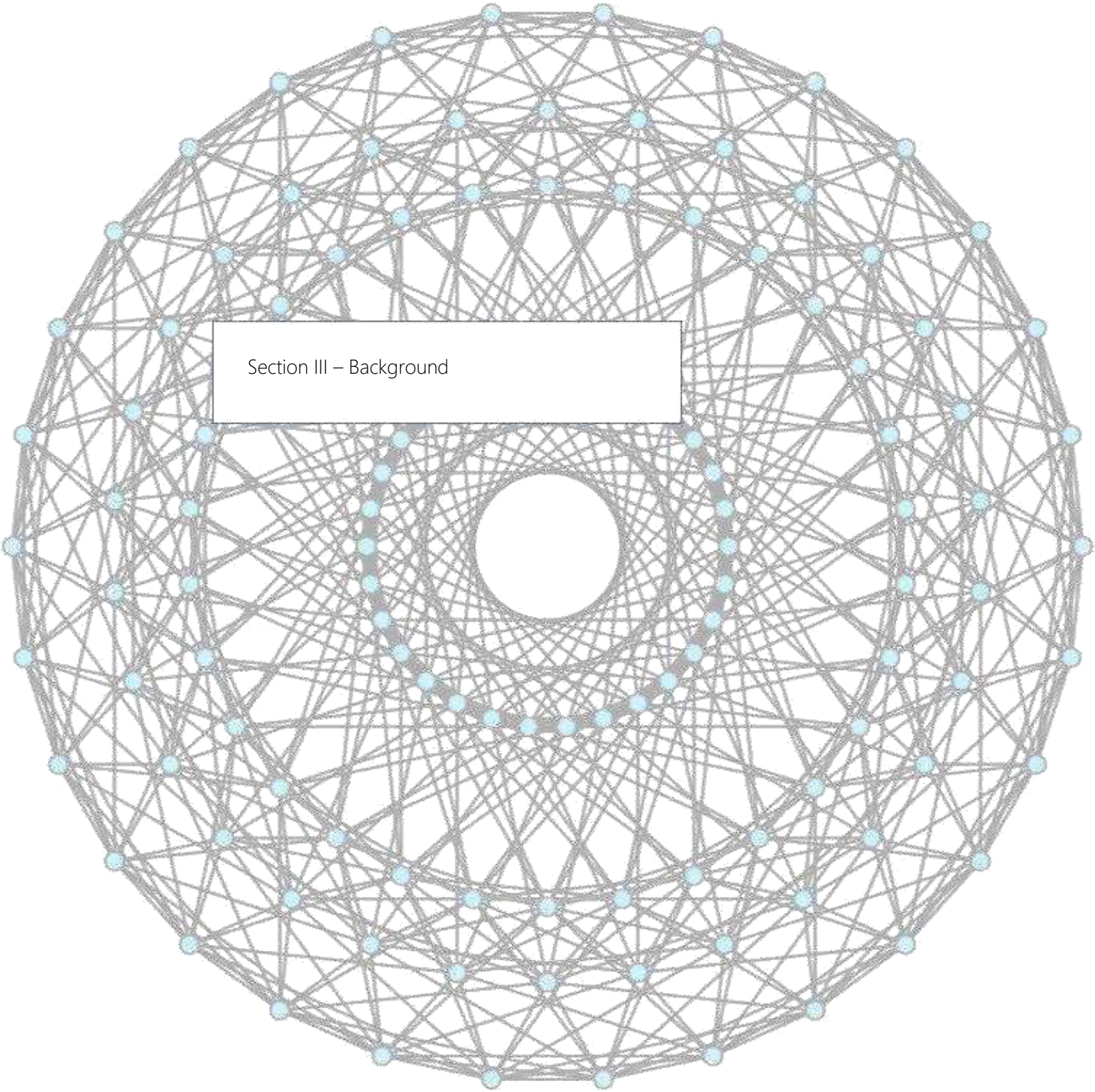
i. No training – agents traverse target space via quasicrystalline field pattern

ii. Enhanced efficiency? - potentially enhanced target space search efficacy, due to the adroit traversal behaviour emergent via field locomotion instruction sequence

iii. Context switching – target space variability is appropriately managed via variances of inbuilt quasicrystalline patterns

Disadvantages:

i. Un-optimized par reality - Not yet optimized for true search/rescue agents; the algorithm applies solely to virtual tri dimensional scenarios.. (as seen in www.manufacturedmind.grn.cc)



Section III – Background

Initially, amidst locomotion beaviourism, for my tri dimensional scan agents, I considered Craig Reynolds simulated flocking algorithm, which entails coordinated simulated animal locomotion routines.

However, upon further research, I discovered algorithms that more appropriately contacted search/scan beaviourism. *See Reference 6.1*

As such, as follows, are figures demonstrating initially considered search patterns:

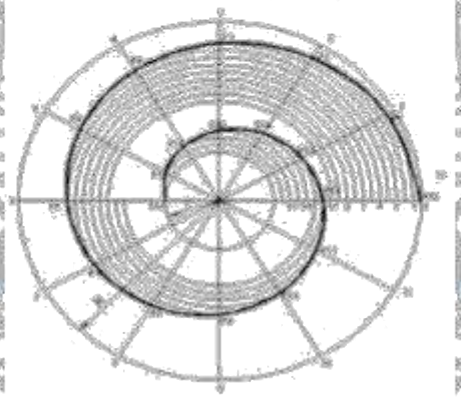


Figure 1.0. Archimedean Spiral Pattern

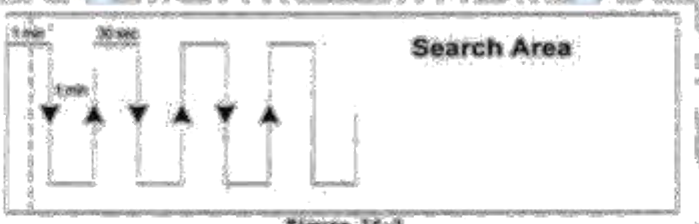


Figure 1.1. Creeping Line Pattern

Figure 1.2, Rectangular Spiral Pattern

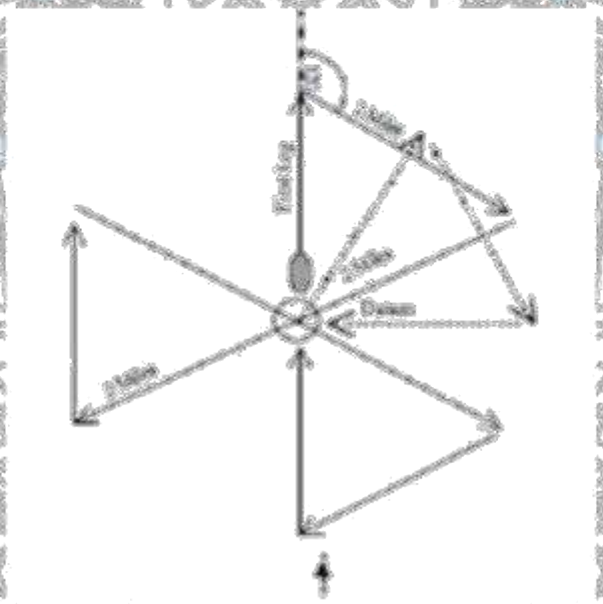


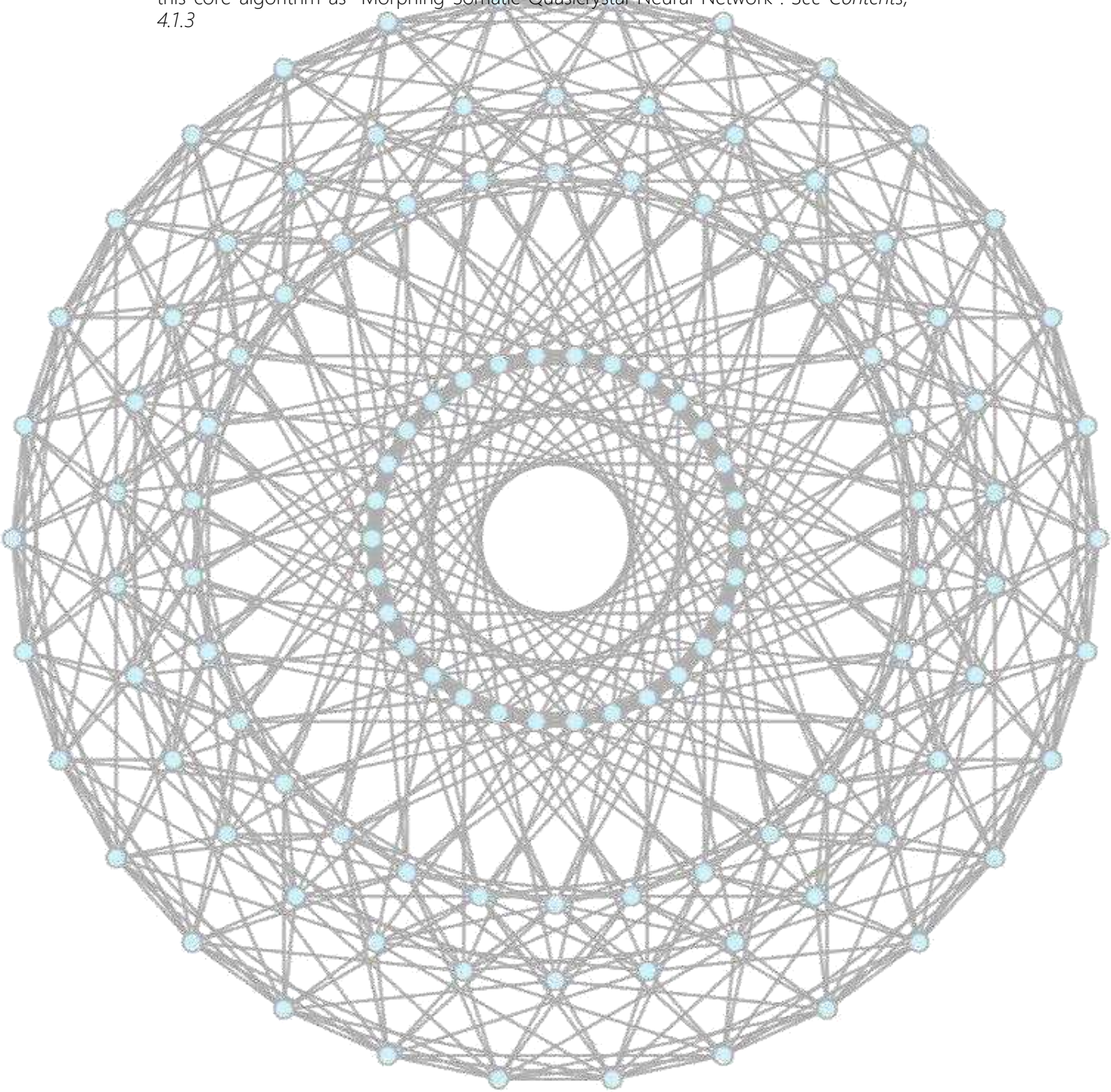
Figure 1.3, Sector Search Pattern

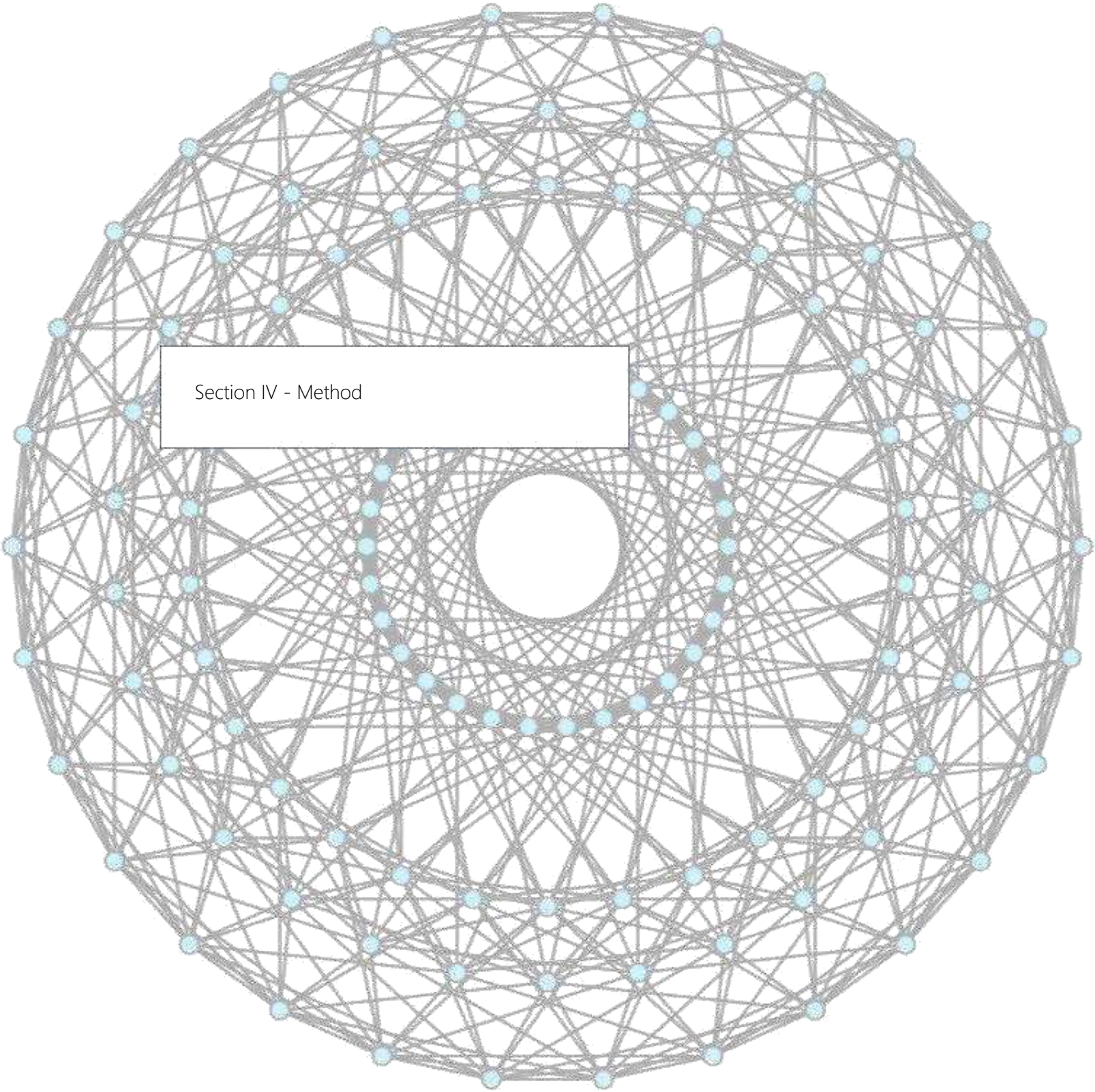
That, said, the aforesaid algorithms are all quite clever routines. However, none sufficiently appropriated. I desired a type of "morphing" algorithm that sub-trivially generated efficacious spanning/contracting mannerisms.

MSQNN essentially generalizes microcosmic behaviour ([specific-sample : electron diffraction fields encompass highly low-time-complex events]), vis a vis macrocosm in orthogonalized quasicrystalline variations (themselves, fundamentally, electron diffraction patterns), abound tri dimensional graph-search scenarios.

I had observed the quasicrystal structure, and noticed that its appearance synchronized neatly with my intentions/desires. I needed to determine whether such a pattern was creatable/feasible. I had prior trivial comprehensions beknownst vectors, and angles. However, my knowledge regarding quasicrystalline petrie polygon patterns were but nil. So as to combat certain ignorance, I utilized my Wikipedia user account, to converse with individuals *See Reference 6.2* who seemed to know about petrie polygons, such as octeracts or tesseracts. After composing, preparing and proposing intelligent questions and assumptions, the senior polygon users had relayed me with crucial guidelines, which in turn bolstered my confidence to generate the algorithm responsible for the core locomotion/scan/search/rescue beaviourism

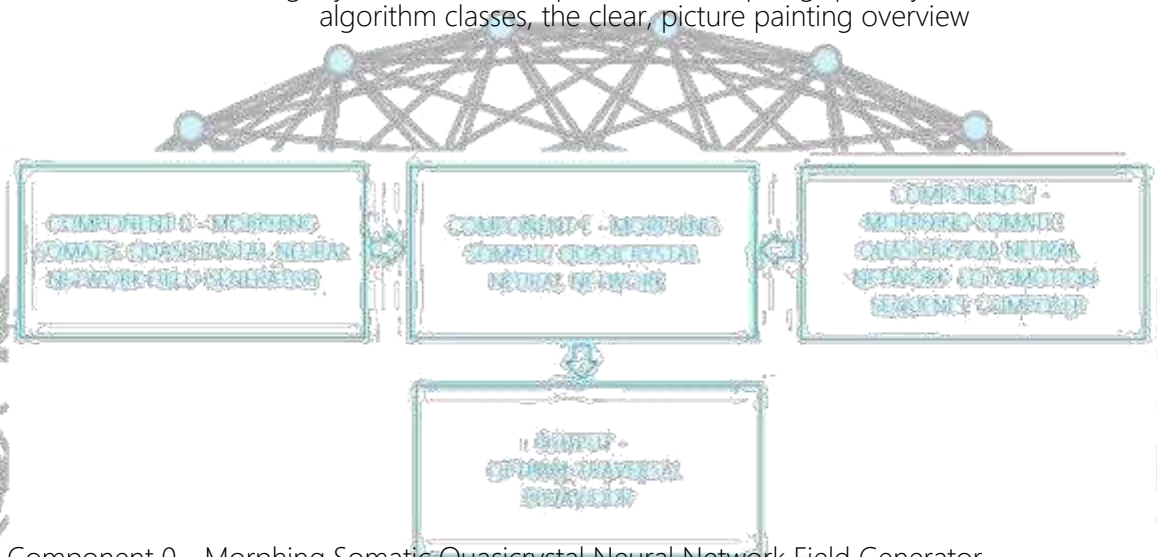
midst somatic neural node clusters insurgent in the tri dimensional test scenario. I label this core algorithm as "Morphing Somatic Quasicrystal Neural Network". *See Contents, 4.1.3*





Section IV - Method

4.1.1 Understanding my core method, sophisticated morphing quasicrystal algorithm classes, the clear, picture painting overview



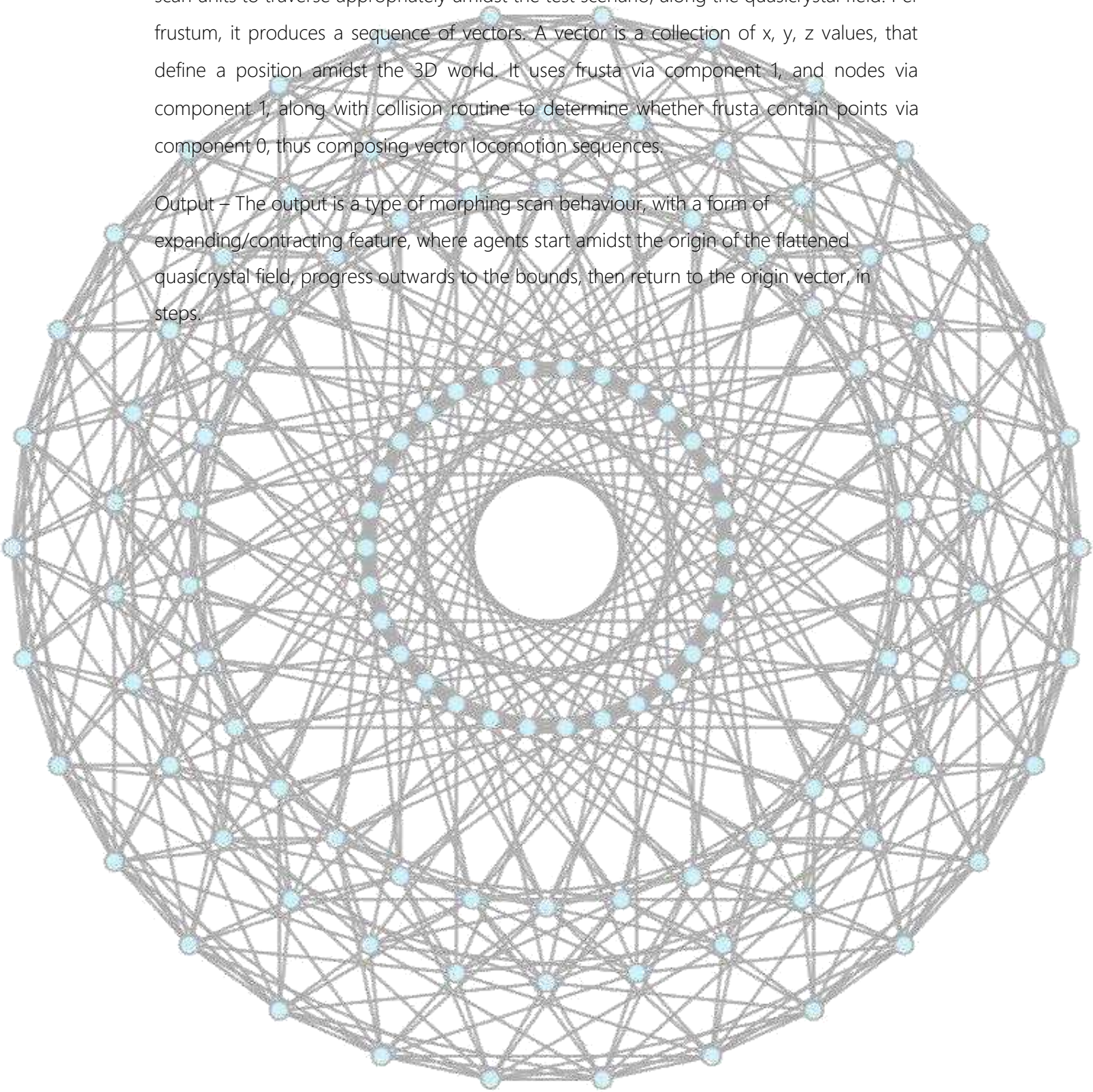
Component 0 - Morphing Somatic Quasicrystal Neural Network Field Generator - responsible for generating the node pattern, based on quasicrystal petrie polygon. A quasicrystal is an n-dimensional volume in nature, that may be flattened by projection/orthogonalization. My algorithm flattens 3 < n<= 10 dimensional quasicrystal, to produce AI node paths.

Component 1 – Morphing Somatic Quasicrystal Neural Network - using the node pattern generated by component 0 above, component 1 is responsible for generating agents, and agent frusta (frustum collection). A frustum is a box in games, normally used to check if things are within an object"s field of view. In the case of test scenario, a frustum occurs before a somatic scan unit, and has the ability to detect nodes generated via pattern above. This enables the agent to traverse the quasicrystal field optimally (from origin to edge)

Component 2 – Morphing Somatic Quasicrystal Neural Network Locomotion Sequence Composer - crucial class. This is the class used to automatically generate flattened quasicrystal flags (isolated paths) or rather locomotion sequences, which enable somatic

scan units to traverse appropriately amidst the test scenario, along the quasicrystal field. Per frustum, it produces a sequence of vectors. A vector is a collection of x, y, z values, that define a position amidst the 3D world. It uses frusta via component 1, and nodes via component 1, along with collision routine to determine whether frusta contain points via component 0, thus composing vector locomotion sequences.

Output – The output is a type of morphing scan behaviour, with a form of expanding/contracting feature, where agents start amidst the origin of the flattened quasicrystal field, progress outwards to the bounds, then return to the origin vector, in steps.



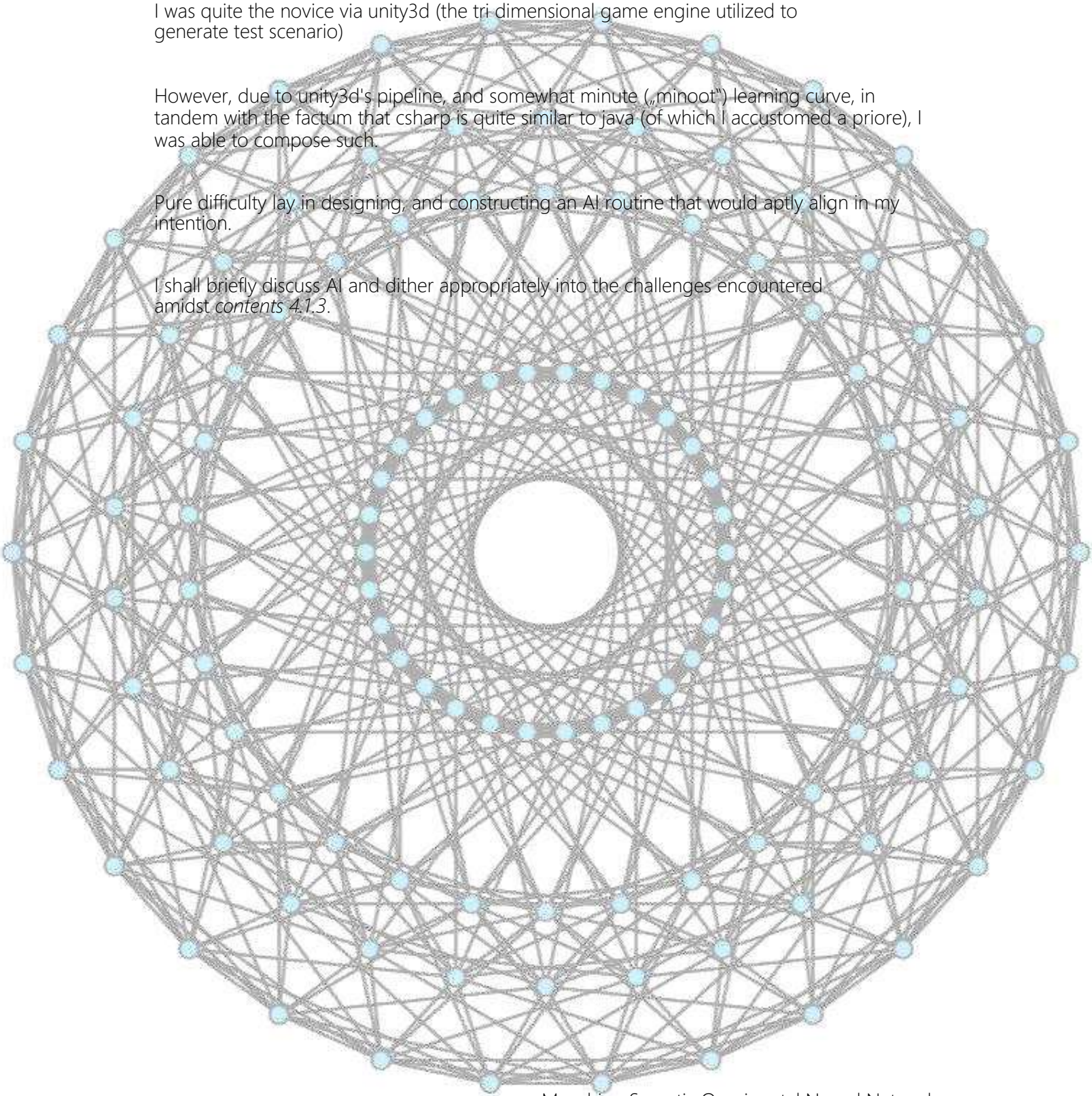
4.1.2 Difficulties

I was quite the novice via unity3d (the tri dimensional game engine utilized to generate test scenario)

However, due to unity3d's pipeline, and somewhat minute („minoot") learning curve, in tandem with the factum that csharp is quite similar to java (of which I accustomed a priore), I was able to compose such.

Pure difficulty lay in designing, and constructing an AI routine that would aptly align in my intention.

I shall briefly discuss AI and dither appropriately into the challenges encountered amidst *contents 4.1.3*.



4.1.3 Agent AI, Detailed

What is Morphing Somatic Quasicrystal Neural Network (M.S.Q N.N)?

Quintessentially, quasicrystals occur amidst nature as multi-dimensional, semiregular uniform n-polytope hypercubic voluminous structures.

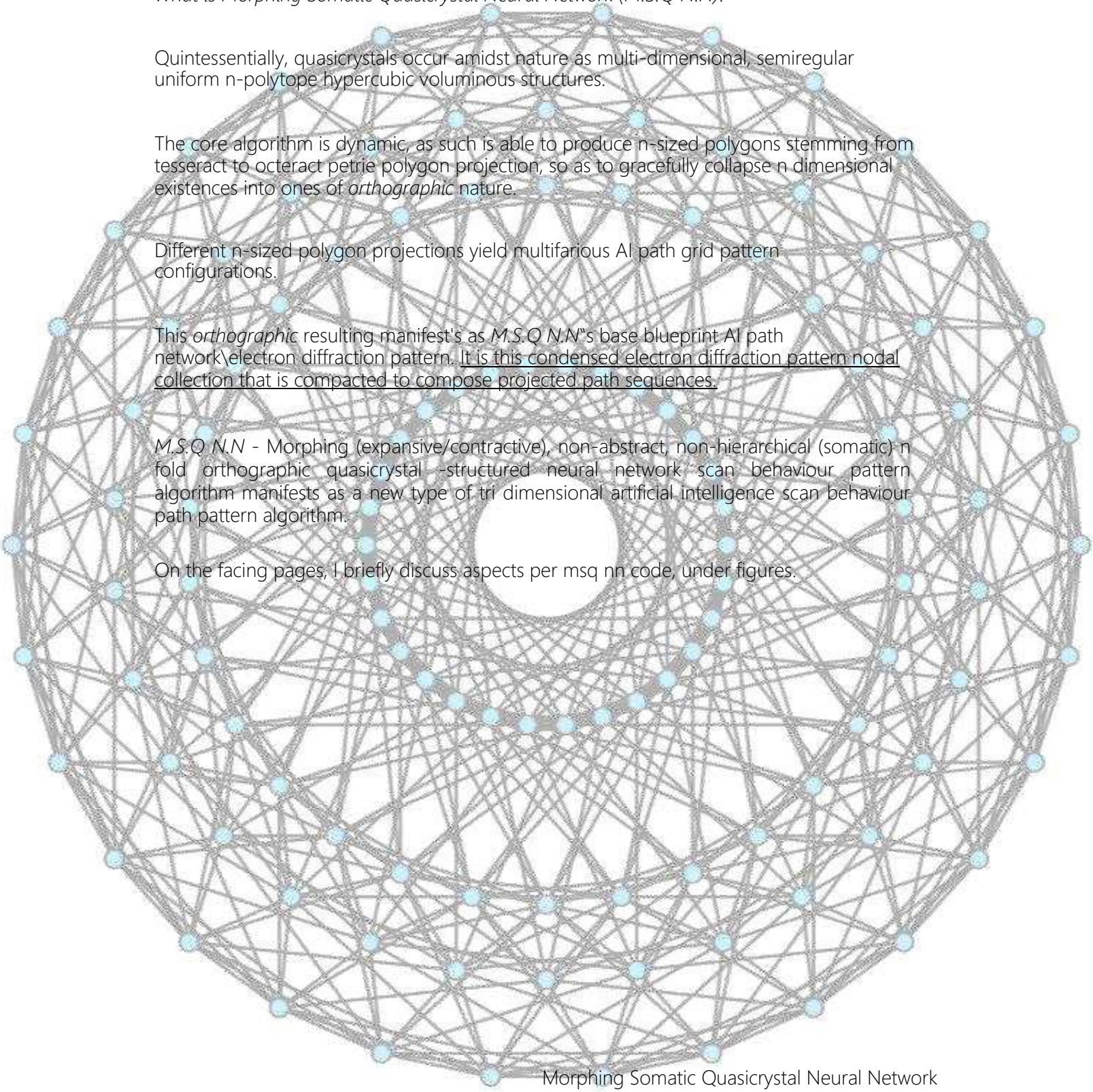
The core algorithm is dynamic, as such is able to produce n-sized polygons stemming from tesseract to octeract petrie polygon projection, so as to gracefully collapse n dimensional existences into ones of *orthographic* nature.

Different n-sized polygon projections yield multifarious AI path grid pattern configurations.

This *orthographic* resulting manifest's as *M.S.Q N.N*'s base blueprint AI path network\electron diffraction pattern. It is this condensed electron diffraction pattern nodal collection that is compacted to compose projected path sequences.

M.S.Q N.N - Morphing (expansive/contractive), non-abstract, non-hierarchical (somatic) n fold orthographic quasicrystal -structured neural network scan behaviour pattern algorithm manifests as a new type of tri dimensional artificial intelligence scan behaviour path pattern algorithm.

On the facing pages, I briefly discuss aspects per msq nn code, under figures.



4.1.3.1 M.S.Q. N.N Field Generator

I"ve written 64 c# classes, however I shall include the most crucial (a sequence of 3 small classes), dealing with my core method/mechanism, A.I:

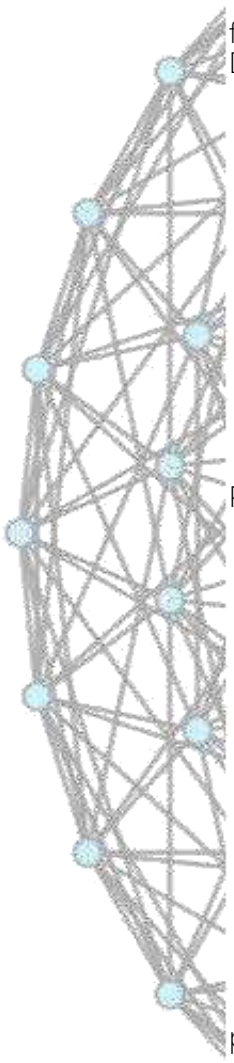
Morphing Somatic Quasicrystal Neural Network Diffraction Pattern Field Generator

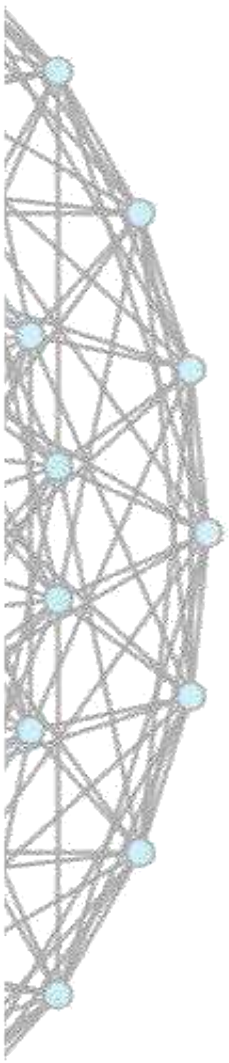
//Author >> Jordan Micah Bennett (manufactured mind (c) 2014) //Author Notes 1 >> This is based on a n dimensional petrie polygon / orthogonalized hypercube (range (tesseract TO dekeract))

//Author Notes 2 >> Credits --> Wikipedia users: TomRuen, Jgmoxness, Claudio Rochhini for quasicrystalline petrie polygon measure polytope information.

```
using UnityEngine;
using System;
using   System.Collections;
using   System.Collections.Generic;


public class MorphingSomaticQuasicrystalNeuralNetworkDiffractionPatternFieldGenerato r :
MonoBehaviour
{
      //establish unity3d game objects public
      List <GameObject> NODES; private
      LineRenderer edge; public GameObject
      FIELD;


      public float SPATIAL_DISPLACEMENT_VALUE; public
      int DIMENSION_CARDINALITY;
      public Vector3 FIELD_POSITION; private
      int VERTEX_CARDINALITY; public bool
      FIELD_TRANSPARENCY;
      public    float     FIELD_TRANSPARENCY_ALPHA;

      void   Awake  (   )
      {
            if ( SPATIAL_DISPLACEMENT_VALUE == null )
                  SPATIAL_DISPLACEMENT_VALUE = 30; //default to dense
grouping,    if    input    is    null


            if ( DIMENSION_CARDINALITY == null ) DIMENSION_CARDINALITY
                  = 4; //default to tesseract
computation,     if    input    is    null

            if   (   FIELD_POSITION     == null     )
                  FIELD_POSITION = new Vector3 ( 247.6164f, 8.124113f, - 772.6021f );
//default to dsr0 pos, if input is null

            generateField      (   );
            enableVisibility          (   FIELD_TRANSPARENCY_ALPHA  );
      }
```

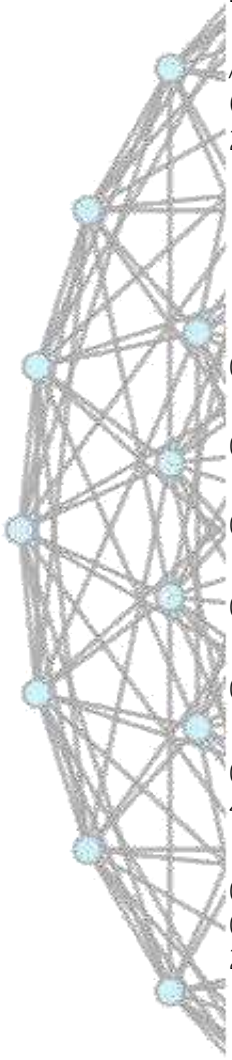
```
public void generateField ( )
{
```





```
VERTEX_CARDINALITY=     (1<<DIMENSION_CARDINALITY);//bitwise
leftshiftoperator unchanged


//computeedgecardinalityfromnchoose1*2^(n-1)                                    OR
n!/1!*(n-1)!*2^ (n-1)
    //formulae source>>     http://www.mathematische-
basteleien.de/hypercube.htm
        intNE=( int)(factorial(DIMENSION_CARDINALITY)/(
factorial ( 1 ) * factorial ( DIMENSION_CARDINALITY - 1 ) ) * Math.Pow ( 2,
DIMENSION_CARDINALITY - 1 ) );

    //establish electron diffusion pattern field FIELD = new
    GameObject ( );
    FIELD.name    = "msqnn_field";


    //establish node objects based on vertex cardinality NODES = new
    List<GameObject> ( );

    for   (   int    node  = 0;    node  < VERTEX_CARDINALITY;     node++ )
    {
        NODES.Add ( GameObject.CreatePrimitive (
PrimitiveType.Sphere ) );
        NODES [  node  ].name    = "msqnn_field_mesh_node";
    }


    //establish edge objects based on edge cardinality edge =
    gameObject.AddComponent<LineRenderer> ( );


    //establish       field
    for   ( int node = 0; node < VERTEX_CARDINALITY; node++ ) NODES [
        node ].transform.parent = FIELD.transform;
    edge.transform.parent          = FIELD.transform;


    //beginorthogonalized          quasicrystal        electron      diffraction
process
    double [ , ] v = new double [ VERTEX_CARDINALITY, DIMENSION_CARDINALITY ] ;
//Converted into c# capable array, for fixed buffer size, of undefined value
    int [ , ] e = new int [ NE, 2 ] ; //Converted into c# capable array, for fixed buffer
size, of undefined value

    string     [   ]   horizontal_projection_vector_scalars                =
    {

/*tesseract*/"0.270598050073,0.653281482438,0.653281482438,0.2705980500 73",

/*penteract*/"0.195439507585,0.511667273602,0.632455532034,0.5116672736
02,0.195439507585",
```

/*hexeract*/"0.149429245361,0.408248290464,0.557677535825,0.55767753582
5,0.408248290464,0.149429245361",



/*hepteract*/"0.118942442321,0.333269317529,0.481588117120,0.534522483825,0.481588117120,0.333269317529,0.118942442321",

/*octeract*/"0.097545161008,0.277785116510,0.415734806151,0.490392640202,0.490392640202,0.415734806151,0.277785116510,0.097545161008",

/*enneract*/"0.081858535979,0.235702260396,0.361116813613,0.442975349592,0.471404520791,0.442975349592,0.361116813613,0.235702260396,0.0818585 35979",

/*dekeract*/"0.069959619571,0.203030723711,0.316227766017,0.398470231296,0.441707654031,0.441707654031,0.398470231296,0.316227766017,0.203030723711,0.069959619571",
};

string [ ] vertical_projection_vector_scalars =
{

/*tesseract*/"-0.653281482438,-0.270598050073,0.270598050073,0.653281482438",

/*penteract*/"-0.601500955008,-0.371748034460,0.000000000000,0.371748034460,0.601500955008",

/*hexeract*/"-0.557677535825,-0.408248290464,-0.149429245361,0.149429245361,0.408248290464,0.557677535825",

/*hepteract*/"-0.521120889170,-0.417906505941,-0.231920613924,0.000000000000,0.231920613924,0.417906505941,0.521120889 170",

/*octeract*/"-0.490392640202,-0.415734806151,-0.277785116510,-0.097545161008,0.097545161008,0.277785116510,0.415734806151,0.490392640 202",

/*enneract*/"-0.464242826880,-0.408248290464,-0.303012985115,-0.161229841765,0.000000000000,0.161229841765,0.303012985115,0.408248290464,0.464242826880",

/*dekeract*/"-0.441707654031,-0.398470231296,-0.316227766017,-0.203030723711,-0.069959619571,0.069959619571,0.203030723711,0.316227766017,0.398470231296,0.441707654031",
};

//establish petrie polygon coxeter dynkin prebaked depth vector direction scalars
//we just distribute it z-wise appropriately per desired region.

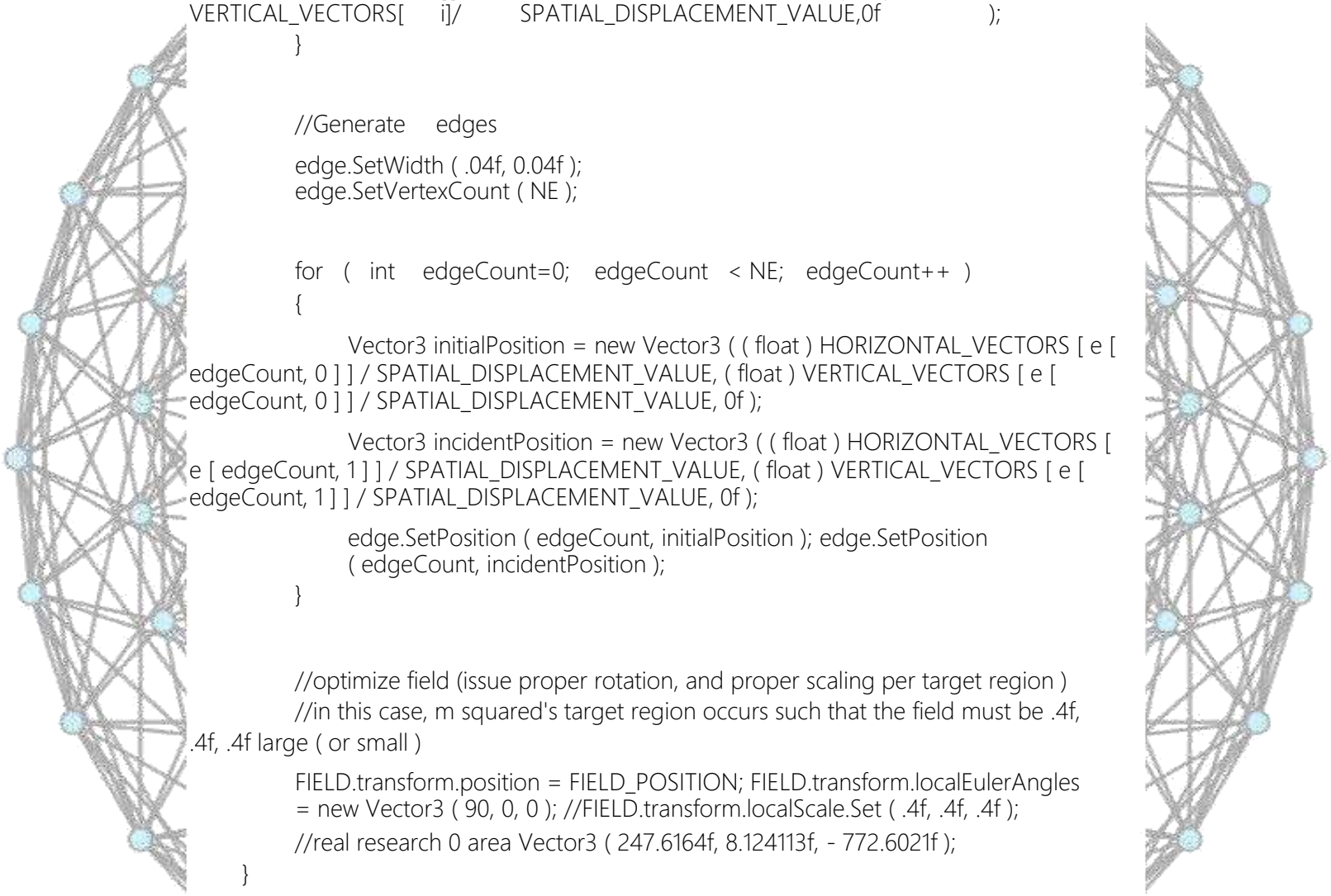
double [ ] HORIZONTAL_VECTORS = new double [VERTEX_CARDINALITY] ; //Converted into c# capable array

double [ ] VERTICAL_VECTORS = new double [VERTEX_CARDINALITY] ; //Converted into c# capable array

int i,j,k,l;

```
for ( i = 0; i < VERTEX_CARDINALITY; ++ i )
    for ( j = 0; j < DIMENSION_CARDINALITY; ++ j )
    {
        long rightShiftedIJindex = ( i >> j ) & 1; //newly introduced integer, for the facing/following right shift realignment measure.
        v [ i, j ] = rightShiftedIJindex == 1 ? -0.5 : 0.5; //Needed to realign, such that right shift operation worked with natural comparator, for easy translation into c#. ( i>>j ) &1 ? .5 :
.5; does not work in c#. C# sees ( i>>j ) &1 as an integer, rather than boolean, as seen in c++.
        //Converted v [ i ] [ j ] to v [ i,j ] and all other [ index ] [ index ] instances in the like
    }
l = 0;

for ( i = 0; i < VERTEX_CARDINALITY - 1; ++ i )
{
    for ( j = i + 1; j < VERTEX_CARDINALITY; ++ j )
    {
        double d = 0;
        for ( k = 0; k < DIMENSION_CARDINALITY; ++ k )
d += ( v [ i, k ] -v [ j, k ] ) * ( v [ i, k ] -v [ j, k ] ) ;
        d = Math.Sqrt ( d ) ;
        if ( d == 1 ) { e [ l, 0 ] =i; e [ l, 1 ] = j; ++l;}
    }
}


//Debug.Assert ( l==NE ) ; //Unity3d and C#'s Assert clash. However, this assert function is not really necessary. It is a mere precaution per orthogonal quasicrystal configuration change.

for(i= 0;i<VERTEX_CARDINALITY;++i)
{
    HORIZONTAL_VECTORS[i]=0;for(l =0;l< DIMENSION_CARDINALITY;++ l)HORIZONTAL_VECTORS[i]+=v[i,l]*( Double.Parse(horizontal_projection_vector_scalars[ DIMENSION_CARDINALITY-4 ].Split(',')[l]) );
    VERTICAL_VECTORS[i]=0;for(l= 0;l< DIMENSION_CARDINALITY;++ l)VERTICAL_VECTORS[i]+=v[i,l]*( Double.Parse(vertical_projection_vector_scalars [ DIMENSION_CARDINALITY-4 ].Split(',')[l]) );
}

doubleSX =800; doubleSY=800;
doubleB =64; doubleR =5;

doublesca =Math.Min( (SX-2*B)/2,(SY-2*B)/2);

for(i= 0;i<VERTEX_CARDINALITY;++i)
```

```
        {
            HORIZONTAL_VECTORS [ i ] = B+ ( HORIZONTAL_VECTORS [ i ] +1 ) *sca;
VERTICAL_VECTORS [ i ] = B+ ( VERTICAL_VECTORS [ i ] +1 ) *sca;
        }
```

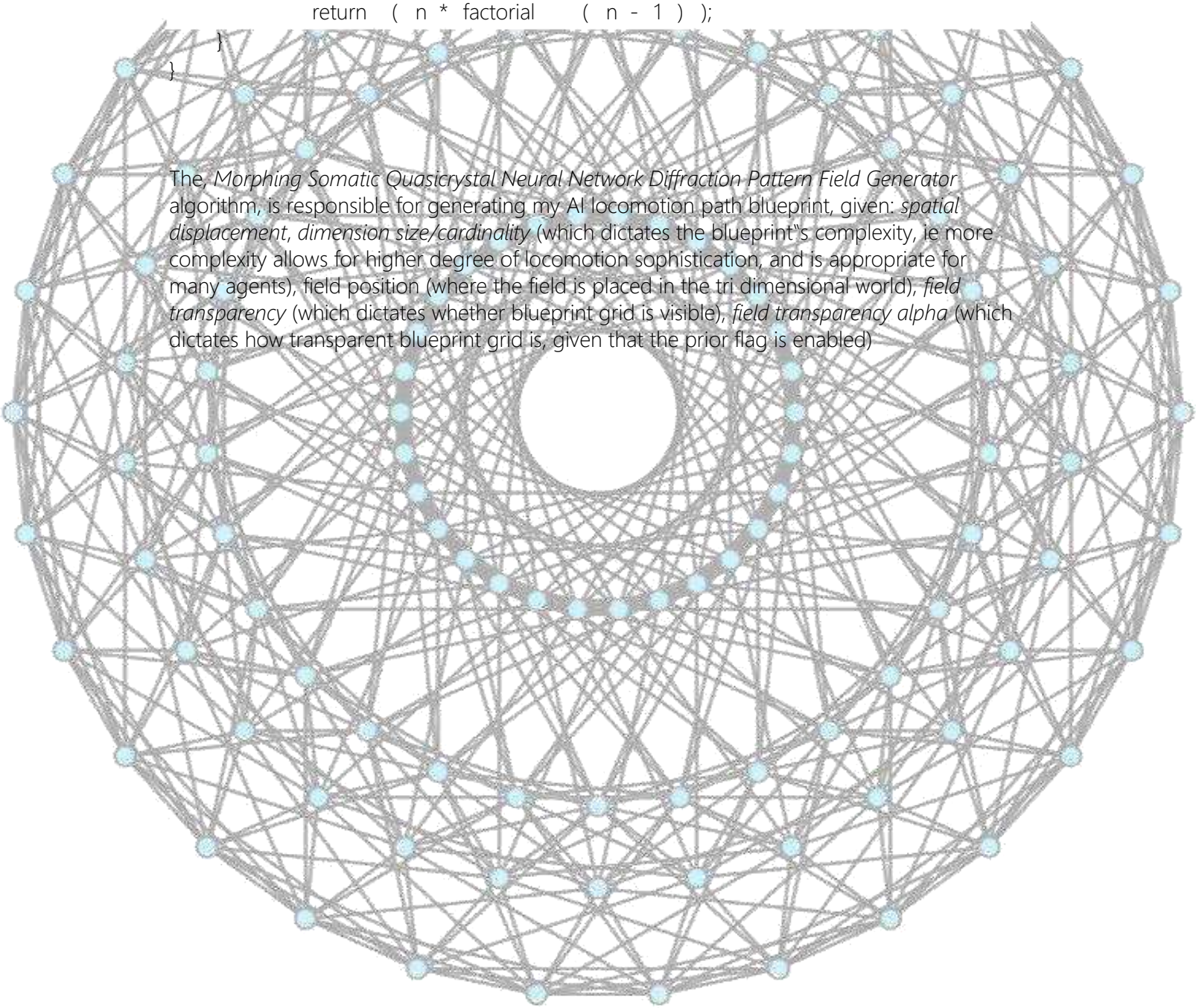


```
            //GENERATE  ORTHOGONALIZEDQUASICRYSTALFIELD
                        NODE
            //Generate    S
            for(i=0;i<              VERTEX_CARDINALITY;++i)
            {
                NODES[i].transform.position=newVector3                          ((float)
HORIZONTAL_VECTOR
S                       [i]     /SPATIAL_DISPLACEMENT_VALUE,(float)
VERTICAL_VECTORS[   i]/     SPATIAL_DISPLACEMENT_VALUE,0f           );
            }


            //Generate    edges
            edge.SetWidth ( .04f, 0.04f );
            edge.SetVertexCount ( NE );


            for  (  int   edgeCount=0;   edgeCount   < NE;   edgeCount++  )
            {
                Vector3 initialPosition = new Vector3 ( ( float ) HORIZONTAL_VECTORS [ e [
edgeCount, 0 ] ] / SPATIAL_DISPLACEMENT_VALUE, ( float ) VERTICAL_VECTORS [ e [
edgeCount, 0 ] ] / SPATIAL_DISPLACEMENT_VALUE, 0f );
                Vector3 incidentPosition = new Vector3 ( ( float ) HORIZONTAL_VECTORS [
e [ edgeCount, 1 ] ] / SPATIAL_DISPLACEMENT_VALUE, ( float ) VERTICAL_VECTORS [ e [
edgeCount, 1 ] ] / SPATIAL_DISPLACEMENT_VALUE, 0f );
                edge.SetPosition ( edgeCount, initialPosition ); edge.SetPosition
                ( edgeCount, incidentPosition );
            }


            //optimize field (issue proper rotation, and proper scaling per target region )
            //in this case, m squared's target region occurs such that the field must be .4f,
.4f, .4f large ( or small )
            FIELD.transform.position = FIELD_POSITION; FIELD.transform.localEulerAngles
            = new Vector3 ( 90, 0, 0 ); //FIELD.transform.localScale.Set ( .4f, .4f, .4f );
            //real research 0 area Vector3 ( 247.6164f, 8.124113f, - 772.6021f );
        }

        public   void   enableVisibility        (  float    alpha   )
        {
            if  (  FIELD_TRANSPARENCY  )
                for  (  int   node  = 0;   node  < VERTEX_CARDINALITY;    node++ )
                {
                    //of   course,    colours    range   from   0f   to   1f
                    NODES [ node ].renderer.material.shader = Shader.Find (
"Transparent/Diffuse" ); //first we need to set the shader to a diffusible gradient
```

```
                    NODES [ node ].renderer.material.SetColor ( "_Color", new Color ( 0f,
0f, 0f, alpha ) ); //next, set _Color ( as specified via unity3d documentation ) attribute to
input alpha
                }
```

```
}

long  factorial      (  int   n  )
{
      if ( n == 0 )
            return 1;
      else
            return   (  n  *  factorial      (  n  -  1  )  );
}
```

The, *Morphing Somatic Quasicrystal Neural Network Diffraction Pattern Field Generator* algorithm, is responsible for generating my AI locomotion path blueprint, given: *spatial displacement*, *dimension size/cardinality* (which dictates the blueprint‟s complexity, ie more complexity allows for higher degree of locomotion sophistication, and is appropriate for many agents), field position (where the field is placed in the tri dimensional world), *field transparency* (which dictates whether blueprint grid is visible), *field transparency alpha* (which dictates how transparent blueprint grid is, given that the prior flag is enabled)



Morphing Somatic Quasicrystal Neural Network

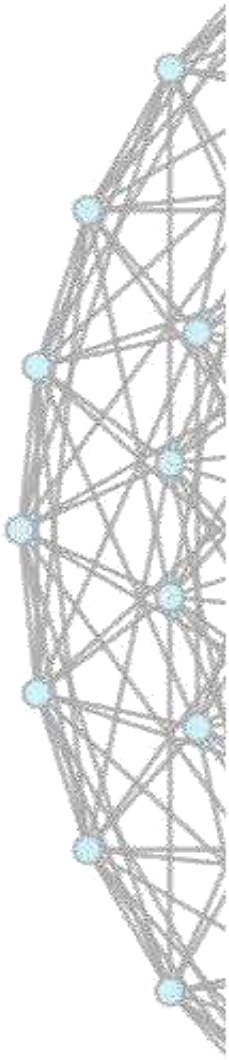
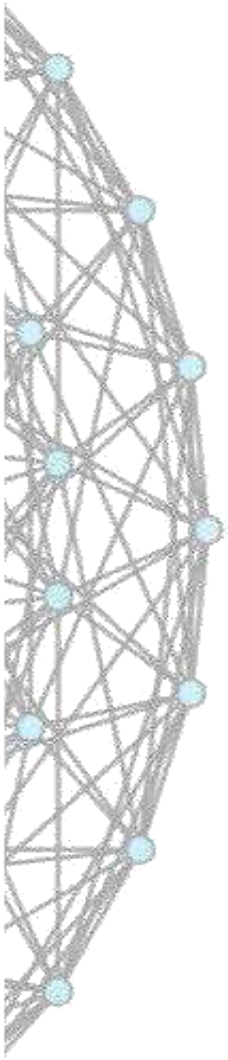
//Author >> Jordan Micah Bennett (manufactured mind (c) 2014)

```
using UnityEngine;
using System;
using System.Collections; using
System.Diagnostics;
using   System.Collections.Generic;

publicclass        MorphingSomaticQuasicrystalNeuralNetwork            :MonoBehaviour
{
      public    GameObject   AGENT;
      public    GameObject   AGENT_FRUSTUM;
      public List <GameObject> CLUSTER, FRUSTA; public
      GameObject CLUSTER_FIELD, FRUSTA_FIELD; public bool
      FRUSTA_VISIBILITY;
      public    float     FRUSTA_VISIBILITY_ALPHA;
      public List <Vector3> INITIAL_CLUSTER_VECTOR_GROUP,
INITIAL_FRUSTA_VECTOR_GROUP; //made public to be accessible amidst
'MorphingSomaticQuasicrystalNeuralNetworkGenerator'. No values need be passed to
these amidst run time.
      public    int    CLUSTER_CARDINALITY;

      //locomotion
      public List <Vector3> commencementVectors; public
      float commencementTime;

      public float TRAVERSAL_RATE; public
      IEnumerator waitEnumerator; public bool
      contractionQuery;
      public    int    AREA_TRAVERSAL_LIMIT_PADDING;

      void   Awake  (   )
      {
            CLUSTER = new List <GameObject> ( );
            CLUSTER_FIELD = new GameObject ( );

            CLUSTER_FIELD.name    = "msqnn_cluster";

            FRUSTA = newList       <GameObject>(      );
            FRUSTA_FIELD= newGameObject(               );

            FRUSTA_FIELD.name    = "msqnn_frusta";
```

```
INITIAL_CLUSTER_VECTOR_GROUP = new List <Vector3> ( );
INITIAL_FRUSTA_VECTOR_GROUP = new List <Vector3> ( );
```

```
            //locomotion
            commencementVectors = new List <Vector3> ( );
            commencementTime = Time.time;
            TRAVERSAL_RATE  = .009f;
            waitEnumerator = waitEnumeratorFunction ( 5 );
            contractionQuery = false;
            AREA_TRAVERSAL_LIMIT_PADDING = 38;
        }

        public void establishAgents ( Vector3 center, float clusterSpacing, float frustumSpacing
)
        {
            //we appropriately situate our agents ( antagonists ) amidst origin our
morphing quasicrystal based field.
            //our loops limit ends @ somewhere amidst the orgin of our field,
bounded in agent's cardinality
            INITIAL_CLUSTER_VECTOR_GROUP = new GameUtilities (
).getCircularVectorCollection ( center, clusterSpacing,
CLUSTER_CARDINALITY   );
            INITIAL_FRUSTA_VECTOR_GROUP = new GameUtilities
( ).getCircularVectorCollection ( center, frustumSpacing,
CLUSTER_CARDINALITY   );


            //establish variables needed to align agents facing outward float
            outwardClusterRotation = 0f,
outwardClusterRotationDisplacementFactor              = 15f;


            ///generate cluster based on INITIAL_CLUSTER_VECTOR_GROUP for ( int
            A = 0; A < CLUSTER_CARDINALITY; A ++ )
            {
                CLUSTER.Add ( ( GameObject ) Instantiate ( AGENT,
INITIAL_CLUSTER_VECTOR_GROUP [ A ], Quaternion.identity ) );
                CLUSTER [ A ].transform.parent = CLUSTER_FIELD.transform; CLUSTER [ A
                ].transform.localEulerAngles = new Vector3 (
0f,    outwardClusterRotation,          0f   );

                //'guessed' agent orientation outward computation
                outwardClusterRotation +=
outwardClusterRotationDisplacementFactor             *   CLUSTER_CARDINALITY;


                //locomotion
                commencementVectors.Add    (   CLUSTER  [   A  ].transform.position
);
            }
```

```
        //establish variables needed to align frustums facing outward float
        outwardFrustaRotation = 0f,
outwardFrustaRotationDisplacementFactor        = 15f;

        //generate FRUSTA based on INITIAL_FRUSTA_VECTOR_GROUP for (
        int A = 0; A < CLUSTER_CARDINALITY; A ++ )
        {
```

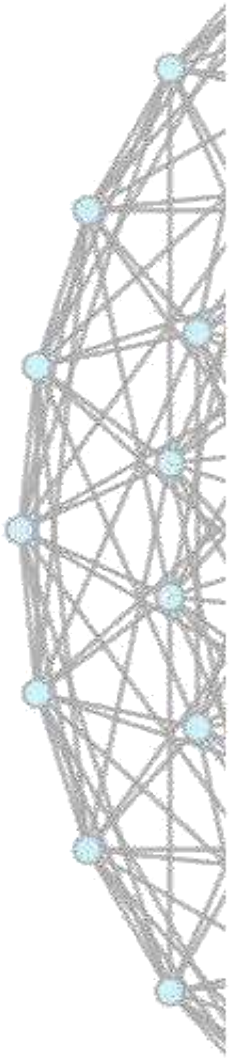
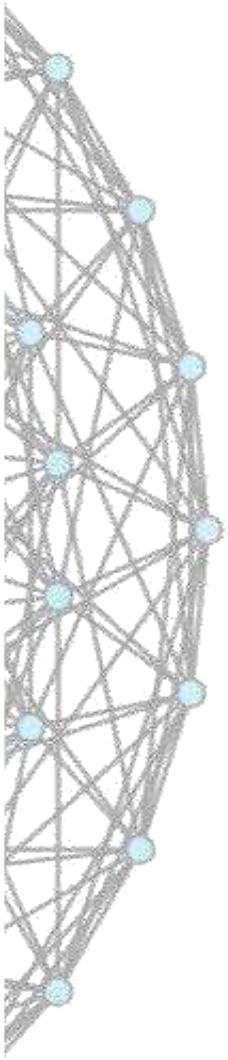
```
                FRUSTA.Add ( ( GameObject ) Instantiate ( AGENT_FRUSTUM,
INITIAL_FRUSTA_VECTOR_GROUP [ A ], Quaternion.identity ) );
                FRUSTA [   A  ].transform.parent          = FRUSTA_FIELD.transform;


                FRUSTA [ A ].transform.localEulerAngles = new Vector3 ( 0f,
outwardFrustaRotation, 0f );


                //fixed position adjustment. The system dyanmic albeit, however, this
fixes each frustum amidst our agent posisition wise a certain manner, such that it spans
                //outwards   upon  the  quaisicrystal        field.

                //need to extend to make this even more dyanamic, by accepting
'spans' and manipulating here appropriately, wrt to such span input.
                FRUSTA [   A  ].transform.Translate          (  new  Vector3   (  0f,   0f,
60f   )   );

                FRUSTA [ A ].transform.localScale = new Vector3 ( FRUSTA [ A
].transform.localScale.x, FRUSTA [ A ].transform.localScale.y, FRUSTA [ A
].transform.localScale.z * 15 );

                outwardFrustaRotation +=
outwardFrustaRotationDisplacementFactor * CLUSTER_CARDINALITY;


                //each frustum has it's own generated flag! //these are
                populated amidst
MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer //add
MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer, so as to
enable locomotion sequence construction.
                        //the added script utilizes collison bounds routines to detect rather
msq nn field nodes are within
                        //the bounds of any agent frustum. If so, the aforsaid script
automatically generates a List of <Vector3> elements, that
                        //consists of the accumilation of vectors found. FRUSTA [ A
                        ].AddComponent (
"MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer" ); //then
                        establish frustum object to offset collision
check, to fulfill somatic traversal node set construction FRUSTA [ A
                        ].GetComponent
<MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer> (
).establishFrustum ( FRUSTA [ A ] );
                        //then alter the discovered node set par sequence composer, such
that x and z components are maintained, but y components are toggled
                        //to default y set generated here {for valid y position = any cluster
unit's y configuration}
                        for ( int V = 0; V < FRUSTA [ A ].GetComponent
<MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer> (
).QUASICRYSTAL_POLYGON_FLAG.Count; V ++ )
                        {
                                Vector3   sequenceComposerVector     = FRUSTA   [   A
```

].GetComponent
<MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer> (
).QUASICRYSTAL_POLYGON_FLAG [V];

FRUSTA [A].GetComponent
<MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer> (

```
).QUASICRYSTAL_POLYGON_FLAG [ V ].Set ( sequenceComposerVector.x, CLUSTER
[ 0 ].transform.position.y, sequenceComposerVector.z );
                        }
                }


                //setup field positions
                CLUSTER_FIELD.transform.position = center;
                FRUSTA_FIELD.transform.position = center;



                // enableFrustumVisibilityControl
                enableVisibilityControl ( );
        }

        public  void  enableVisibilityControl           (  )
        {
                if  (  !FRUSTA_VISIBILITY       )
                        new GameUtilities       (  ).colourItems       (  FRUSTA,  0f,  0f,  0f,
FRUSTA_VISIBILITY_ALPHA ); else
                        new GameUtilities       (  ).colourItems       (  FRUSTA,  1f,  0f,  0f,  1f
);
        }

        public  IEnumerator  waitEnumeratorFunction       (  int  secondCardinality       )
        {
                yield  return  new  WaitForSeconds  (  secondCardinality       );
        }

        public  void  Update  (  )
        {
                print  (  ">>>"  + contractionQuery       );
                for  (  int  A  = 0;  A  < CLUSTER_CARDINALITY;  A  ++ )
                {
                        //expanding  behaviour
                        if ( CLUSTER [ A ].GetComponent <Animator> ( ).GetBool ( "Scanning" ) )
                        {
                                if  (  !contractionQuery       )
                                {

MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer
expaindingSequenceComposer = FRUSTA [ A ].GetComponent
<MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer> (
);

                                        if  (  expaindingSequenceComposer  !=  null  )
                                        {
```

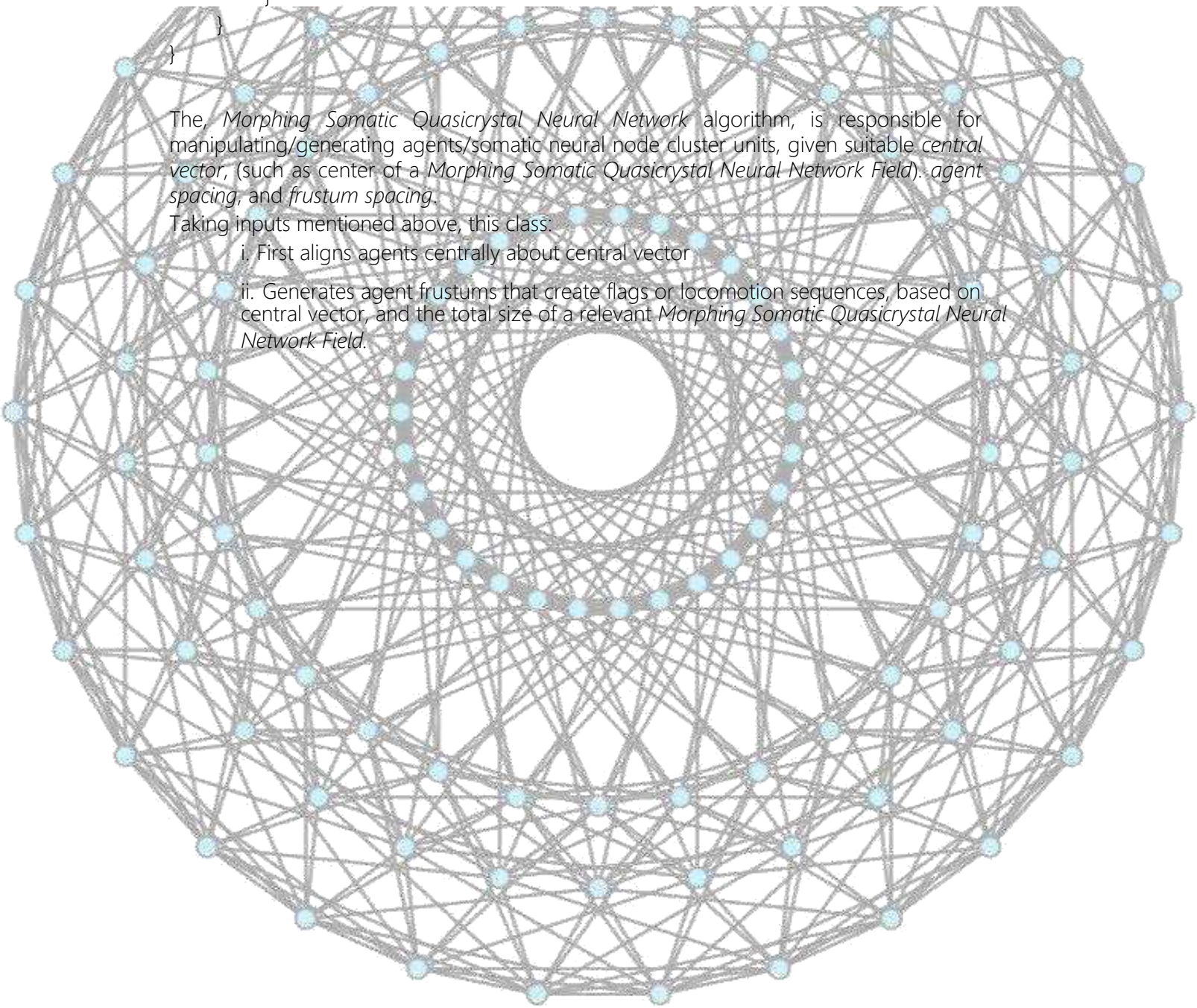
```
                                        List <Vector3> lerpVectors =
expaindingSequenceComposer.QUASICRYSTAL_POLYGON_FLAG;

                                        for ( int V = 0; V <
expaindingSequenceComposer.QUASICRYSTAL_POLYGON_FLAG.Count; V ++ )
                                        {
                                            if  (  waitEnumerator.MoveNext      (  )  )
```

```
                                                CLUSTER [ A ].transform.position =
Vector3.Lerp ( CLUSTER [ A ].transform.position, lerpVectors [ V ],
TRAVERSAL_RATE );
                                        }
                                }
                        }
                }

                //expanse/contraction          determinism
                if ( CLUSTER [ A ].GetComponent <Animator> ( ).GetBool ( "Scanning" ) )
                {

MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer
bluntSequenceComposer = FRUSTA [ A ].GetComponent
<MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer> (
);

                        print ( "contractionQuery >>> " + contractionQuery + " _CLUSTER
>>>" + CLUSTER [ A ].transform.position.z + " _QUASICRYSTAL_POLYGON_FLAG >>>" + (
bluntSequenceComposer.QUASICRYSTAL_POLYGON_FLAG [ 0 ].z -
AREA_TRAVERSAL_LIMIT_PADDING ) );
                        if ( CLUSTER [ A ].transform.position.z >= (
bluntSequenceComposer.QUASICRYSTAL_POLYGON_FLAG [ 0 ].z -
AREA_TRAVERSAL_LIMIT_PADDING ) )
                                contractionQuery = true;
                        else if ( CLUSTER [ A ].transform.position.z ==
bluntSequenceComposer.QUASICRYSTAL_POLYGON_FLAG [
bluntSequenceComposer.QUASICRYSTAL_POLYGON_FLAG.Count ].z )
                                contractionQuery      = false;
                }

                //contracting       behaviour
                if ( CLUSTER [ A ].GetComponent <Animator> ( ).GetBool ( "Scanning" ) )
                {
                        if   (   contractionQuery      )
                        {

MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer
contractingSequenceComposer = FRUSTA [ A ].GetComponent
<MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer> (
);

                                if   (   contractingSequenceComposer       !=   null    )
                                {
                                        List <Vector3> lerpVectors =
contractingSequenceComposer.QUASICRYSTAL_POLYGON_FLAG;
```

```
                                    for ( int V =
contractingSequenceComposer.QUASICRYSTAL_POLYGON_FLAG.Count; V > 0; V
--  )
                                    {
                                        if  (  waitEnumerator.MoveNext      (  )  )
```

```
                                                CLUSTER [ A ].transform.position =
Vector3.Lerp ( CLUSTER [ A ].transform.position, lerpVectors [ V ],
TRAVERSAL_RATE  );
                                    }
                                }
                            }
                        }
                    }
                }
            }
```



The, *Morphing Somatic Quasicrystal Neural Network* algorithm, is responsible for manipulating/generating agents/somatic neural node cluster units, given suitable *central vector*, (such as center of a *Morphing Somatic Quasicrystal Neural Network Field*). *agent spacing*, and *frustum spacing*.

Taking inputs mentioned above, this class:

i. First aligns agents centrally about central vector

ii. Generates agent frustums that create flags or locomotion sequences, based on central vector, and the total size of a relevant *Morphing Somatic Quasicrystal Neural Network Field*.

4.1.3.3 M.S.Q. N.N Locomotion Sequence Gen

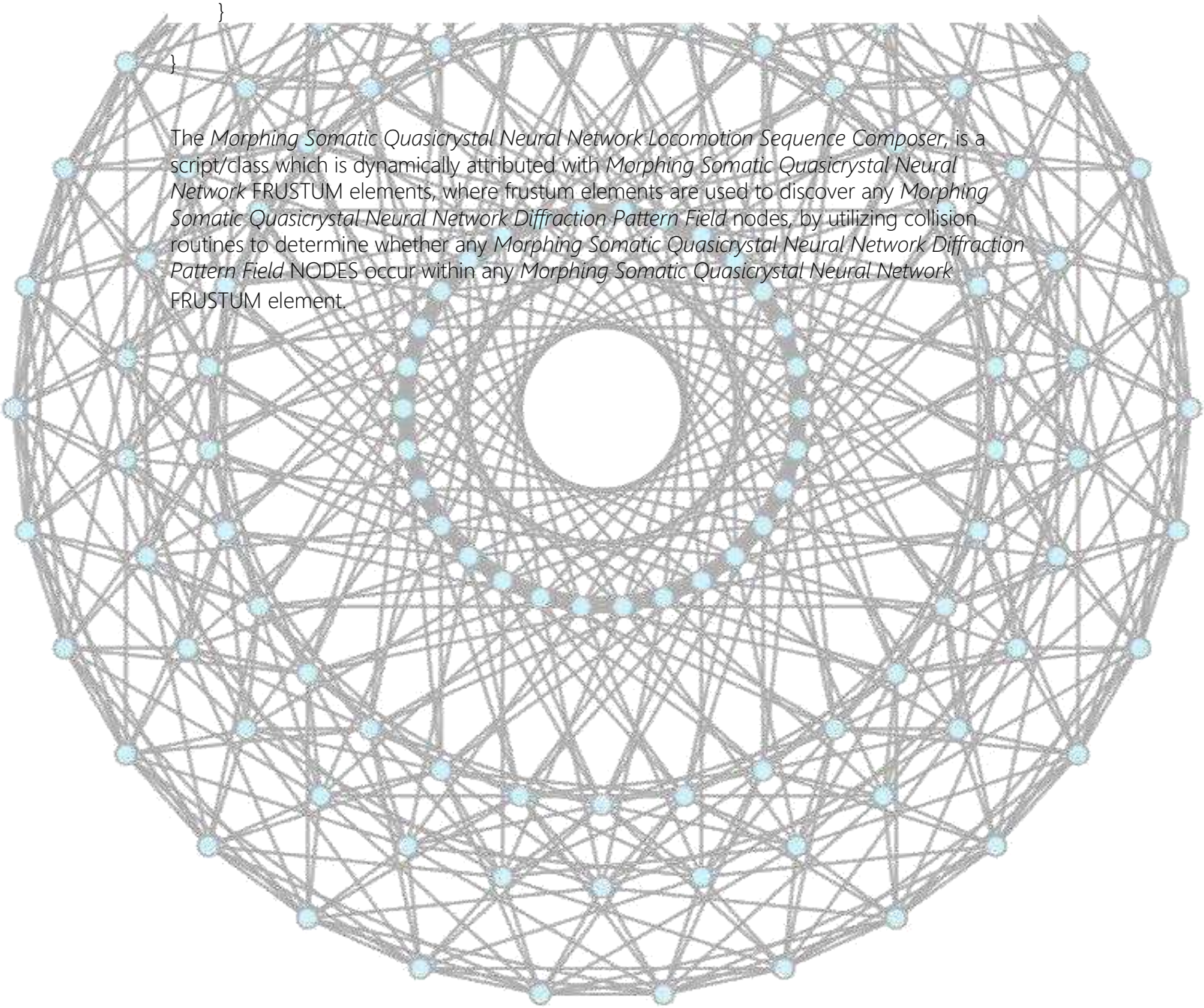
Morphing Somatic Quasicrystal Neural Network Locomotion Sequence Composer

//Author>> JordanMicahBennett (manufacturedmind(c)2014)

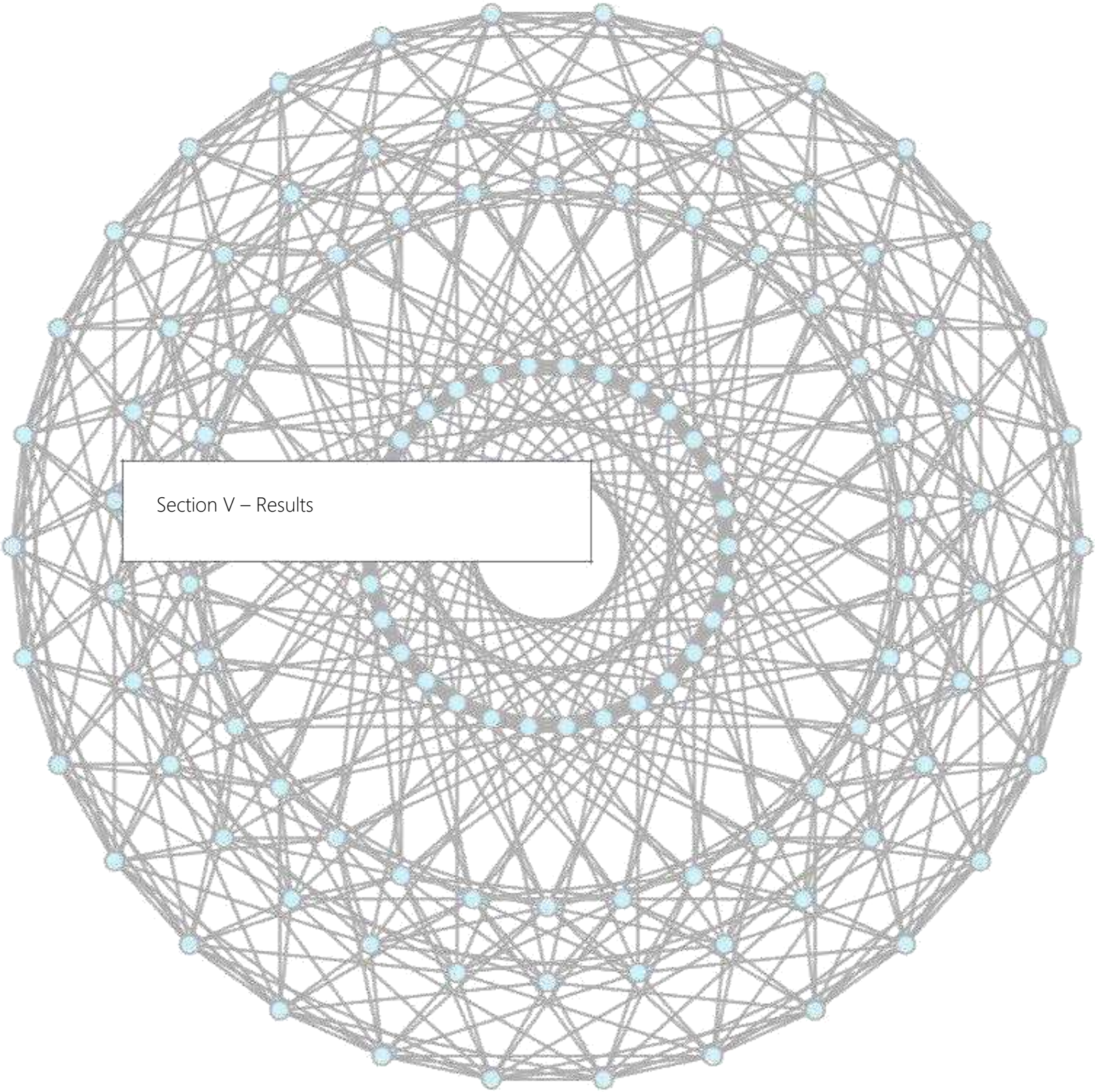
```
using   UnityEngine;
using   System.Collections;
using   System.Collections.Generic;


public class
MorphingSomaticQuasicrystalNueralNetworkLocomotionSequenceComposer :
MonoBehaviour
{
      private
MorphingSomaticQuasicrystalNeuralNetworkDiffractionPatternFieldGenerato
r fieldGenerator;
      private MorphingSomaticQuasicrystalNeuralNetwork nueralNetwork; public List
      <Vector3> QUASICRYSTAL_POLYGON_FLAG; //discovered nodes
{that all compose a single quasicrystal flag} public
      GameObject FRUSTUM;

      public   void   Awake  (   )
      {
            fieldGenerator = GameObject.FindGameObjectWithTag ( Tags.gameController
).GetComponent
<MorphingSomaticQuasicrystalNeuralNetworkDiffractionPatternFieldGenerat or> ( );
            nueralNetwork = GameObject.FindGameObjectWithTag (
Tags.gameController ).GetComponent
<MorphingSomaticQuasicrystalNeuralNetwork> ( );

            QUASICRYSTAL_POLYGON_FLAG = new List     <Vector3>   (   );
      }

      //simply checks if attached frustum contains any points within the quasicrystal field
      public   void   Update(      )
      {
            if ( FRUSTUM != null )
                  collisionCheck ( );
      }

      public   void   establishFrustum            (GameObjectvalue     )
      {
            this.FRUSTUM= value;
      }


      public   void   collisionCheck          ()
      {
            for   (int      N = 0;    N < fieldGenerator.NODES.Count;          N ++ )
```

```
        {
            if ( FRUSTUM.collider.bounds.Contains (
fieldGenerator.NODES [ N ].collider.transform.position ) )
            {
```

```
                        QUASICRYSTAL_POLYGON_FLAG.Add ( fieldGenerator.NODES [ N
].transform.position );

                        if ( fieldGenerator.FIELD_VISIBILITY ) new
                            GameUtilities ( ).colourItem (
fieldGenerator.NODES    [ N ],   0f,   0f,   1f,   4f  );
                    }
                }
            }
}
```



The *Morphing Somatic Quasicrystal Neural Network Locomotion Sequence Composer*, is a script/class which is dynamically attributed with *Morphing Somatic Quasicrystal Neural Network* FRUSTUM elements, where frustum elements are used to discover any *Morphing Somatic Quasicrystal Neural Network Diffraction Pattern Field* nodes, by utilizing collision routines to determine whether any *Morphing Somatic Quasicrystal Neural Network Diffraction Pattern Field* NODES occur within any *Morphing Somatic Quasicrystal Neural Network* FRUSTUM element.



Section V – Results

5.1 What does my quasicrystal based AI achieve

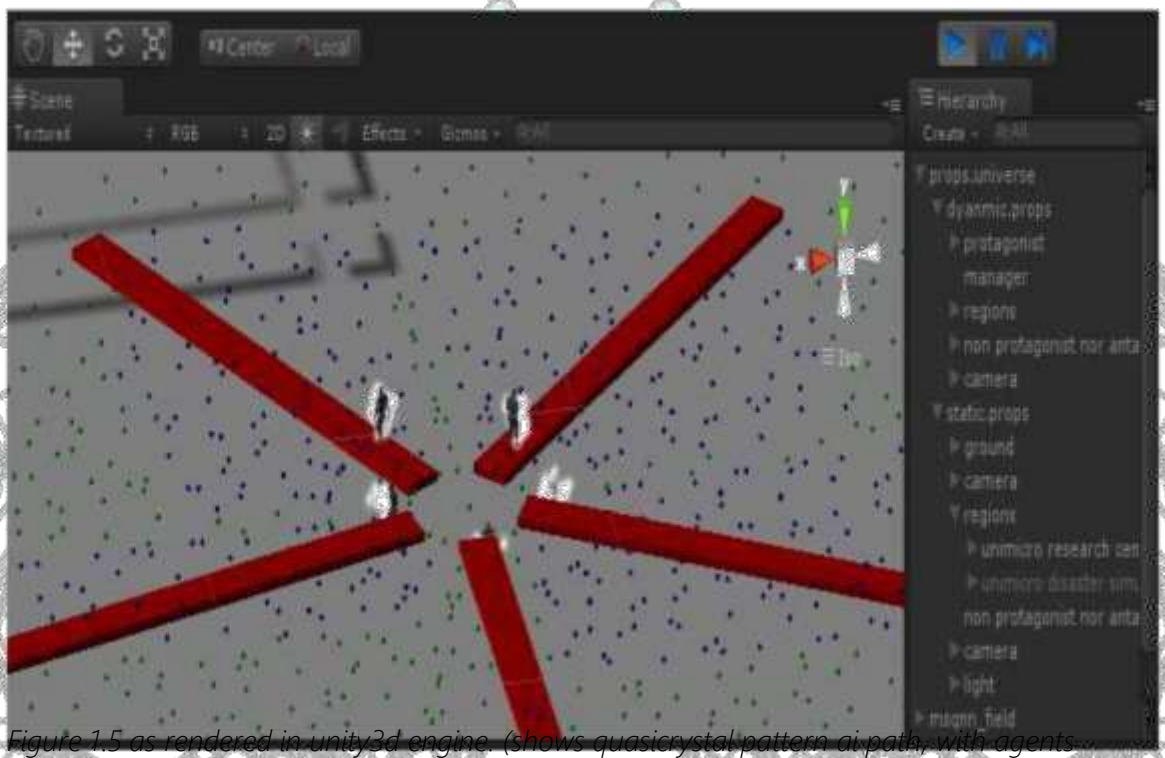


Figure 1.5 as rendered in unity3d engine. (shows quasicrystal pattern ai path, with agents about the origin of the flattened quasicrystal field, the red boxes, green and blue points are naturally invisible, but are toggle-able as a hack to display ai path in test scenario, and have visibility flags to display/hide in develop mode)

In figure 0.4 above, the red boxes represent frusta (frustum collection). A frustum is a volume that is normally used to detect objects. The green points represent nodes in a dekeract based quasicrystal pattern, generated via my dynamic morphing somatic quasicrystal neural network algorithm classes. In contents 3.1.1, I mentioned that a quasicrystal is an n-dimensional volume in nature, which may be reduced or flattened to a bi-dimensional structure or pattern. The blue points are subsets of the green points, or rather, the blue points are points that collide with the frusta or red boxes. These blue points are used to form the locomotion sequence per test scenario agent.

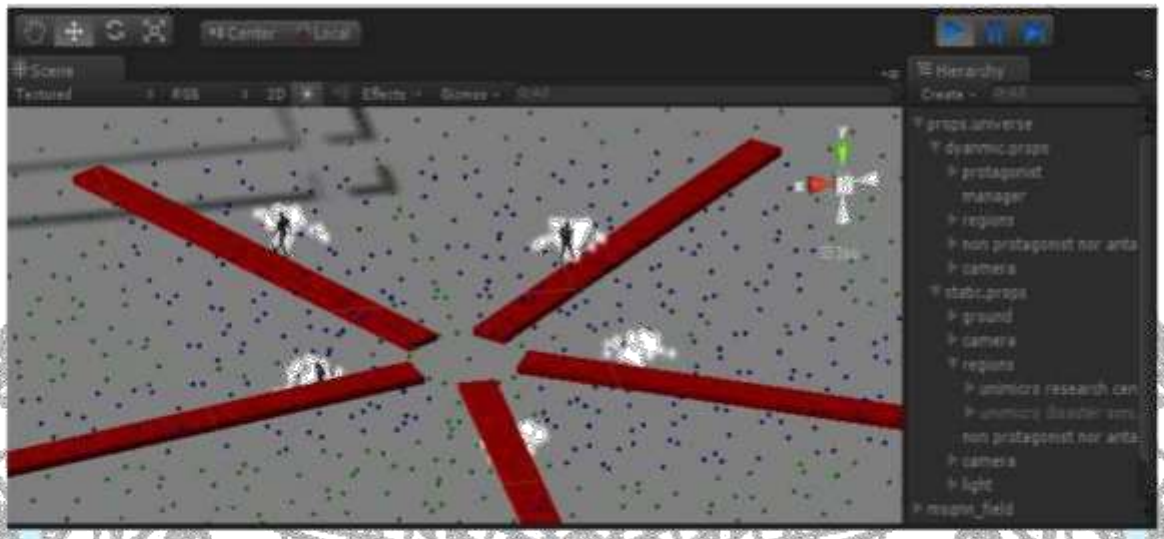
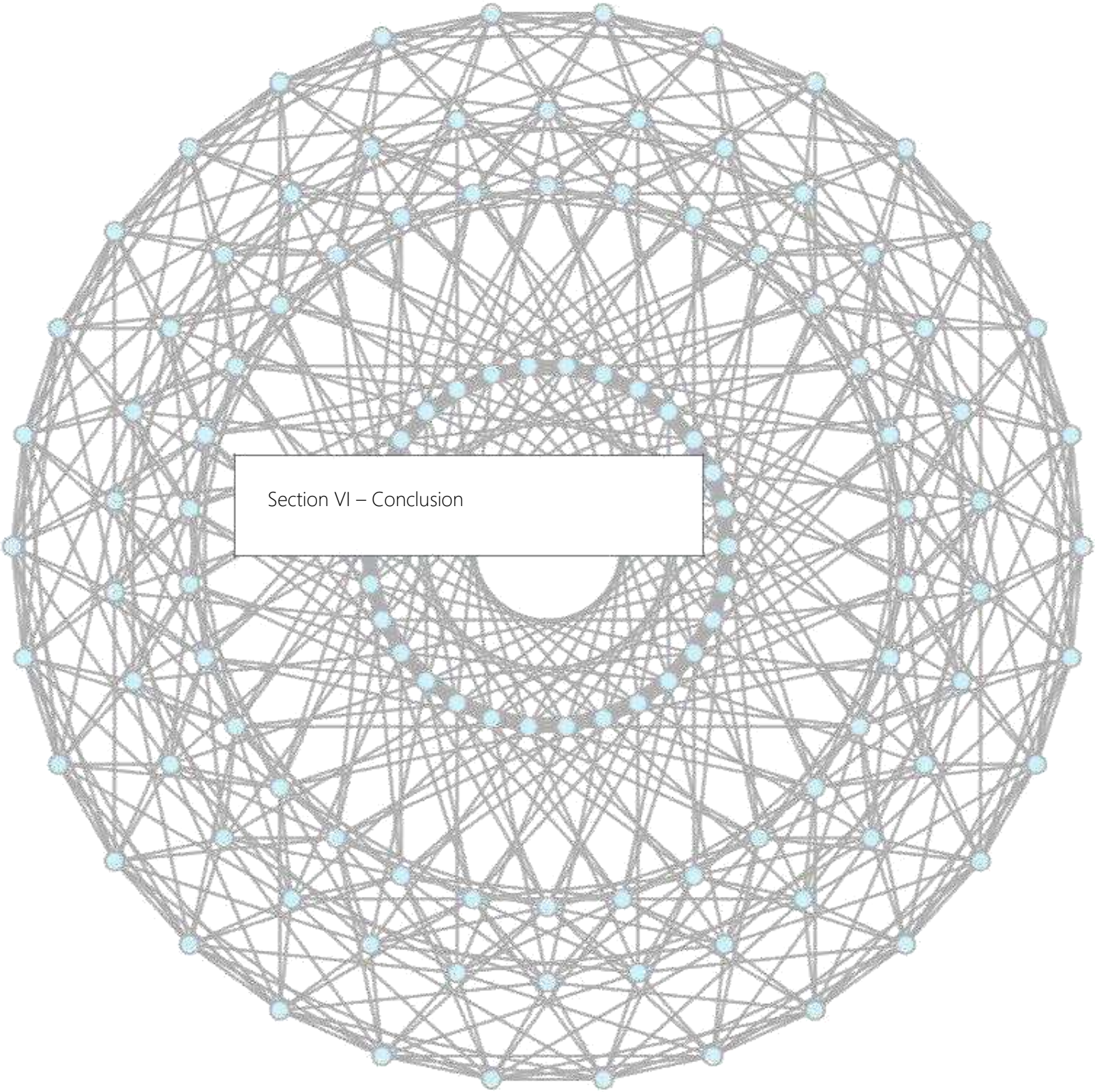


Figure 1.6 as rendered in unity3d engine. (shows quasicrystal pattern at path, with agents spread out away from the origin)

The output is a form of coordinated scan behaviour with a type of morphing or contracting or expanding feature. In other words, the test scenario agents progress outwards along the blue nodes chosen by my *Morphing Somatic Quasicrystal Neural Network Locomotion Sequence Composer* class. As discussed in contents 3.1.1, this class takes the *Morphing Somatic Quasicrystal Neural Network and Morphing Somatic Quasicrystal Neural Network Generator* as input, and uses frusta and nodes respectively from each, to choose points for agent locomotion. Points are selected and stored as a sequence of vectors, which are then later used for continuous locomotion. When the agents have reached the last vector in the sequence, (which represents the bounds of the field) they halt, then return amidst the field‟s origin. This is why I describe the behaviour as morphing, (expansive/contractive) since agents begin in the origin, progress to the bounds of the flattened quasicrystal field, then return amidst the origin, repeating the progress, and thus effectively scanning the test scenario environment world.



Section VI – Conclusion

6.1 Conclusion, Implementation success

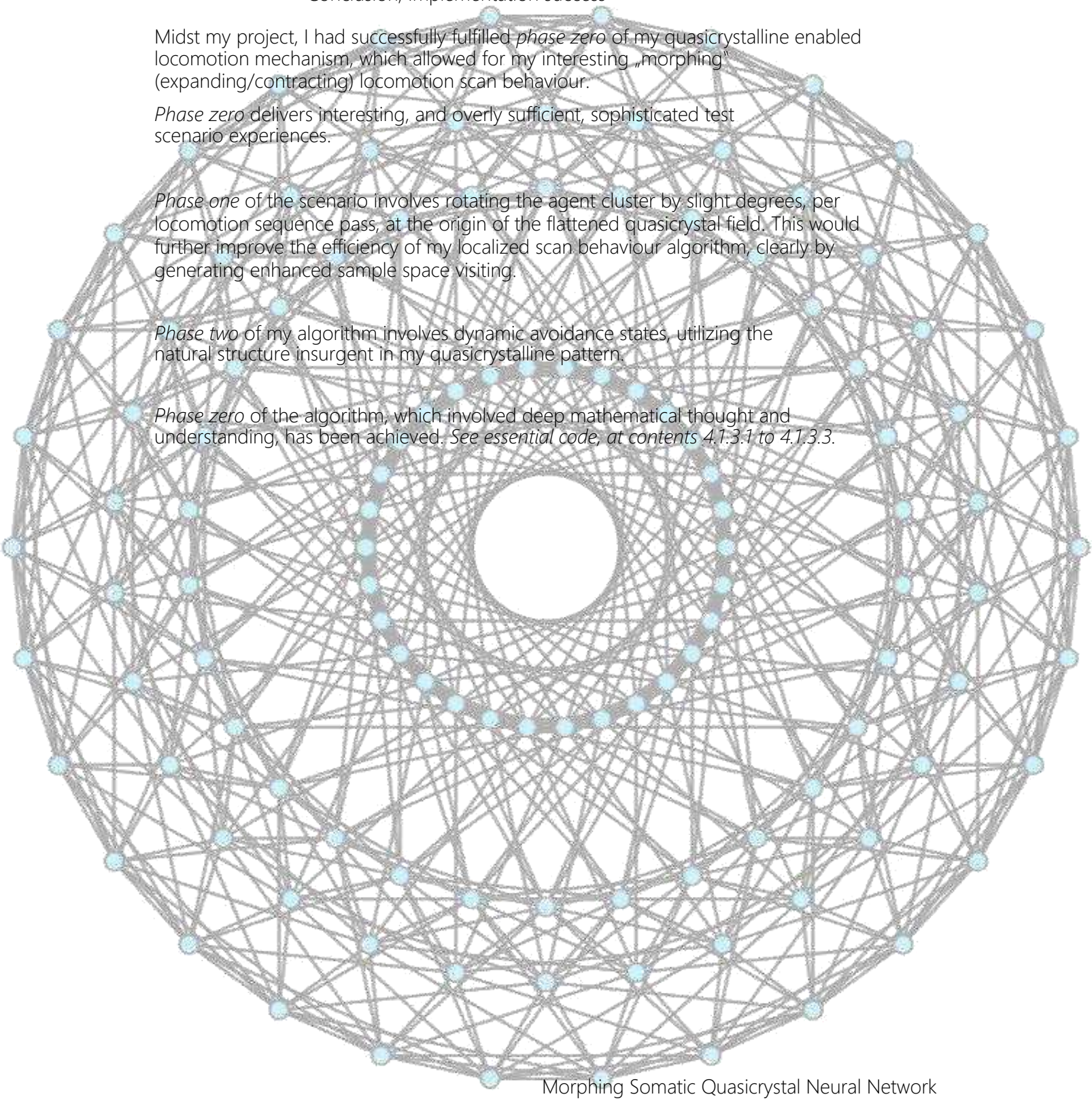
Midst my project, I had successfully fulfilled *phase zero* of my quasicrystalline enabled locomotion mechanism, which allowed for my interesting „morphing" (expanding/contracting) locomotion scan behaviour.

Phase zero delivers interesting, and overly sufficient, sophisticated test scenario experiences.

Phase one of the scenario involves rotating the agent cluster by slight degrees, per locomotion sequence pass, at the origin of the flattened quasicrystal field. This would further improve the efficiency of my localized scan behaviour algorithm, clearly by generating enhanced sample space visiting.

Phase two of my algorithm involves dynamic avoidance states, utilizing the natural structure insurgent in my quasicrystalline pattern.

Phase zero of the algorithm, which involved deep mathematical thought and understanding, has been achieved. *See essential code, at contents 4.1.3.1 to 4.1.3.3.*



[6.2] Where do my tools end, and my work (sizably difficult work) begin?

Unity3d is the best solution, when it comes to producing professional games/scenarios, at reasonable pricing. (I used free version)

As such, unity3d provides many facets or resources, including diverse foundries of libraries, and materials such as textures. Though I freshly wrote 64 classes, I were merely either accessing the primitives already premade in the unity3d engine, or characters which I modeled using make human. For example to make my character move, I would write something like:

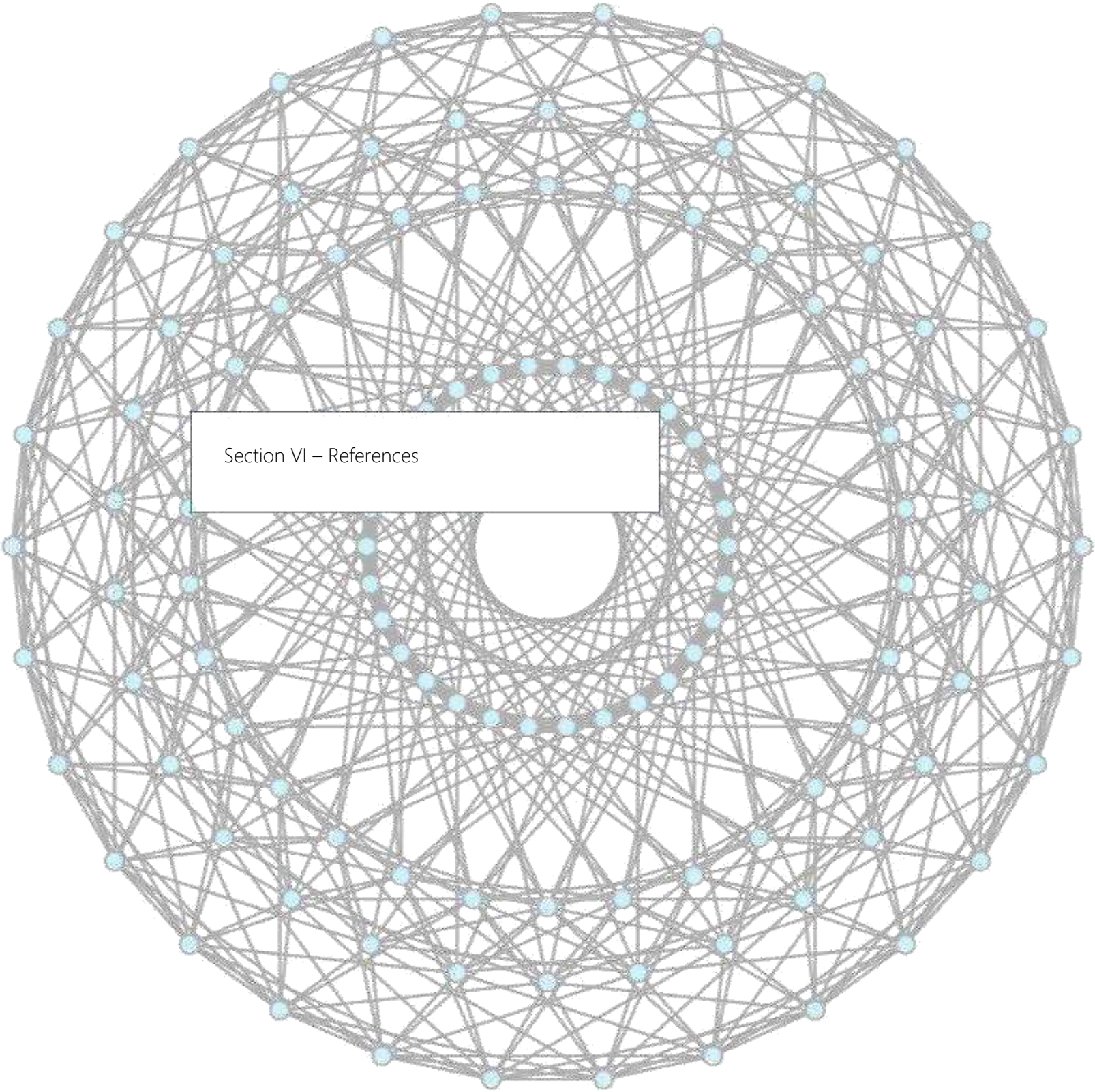
```
if ( Input.GetKeyDown ( "Up" ) )
    mainCharacter.transform.translate ( 10f, 0f, 0f );
```

Above, unity3d provides all the functions, all I needed to do was import my main character object, then refer to in by "mainCharacter", then use unity3d"s attribute and key control system to translate my main character when the up key is toggled. In unity3d every object has a „transform" which I may use to translate, rotate, or scale an object, along with many other functions.

So, unity3d provides a pipeline for importing objects, and also many functions to access and manipulate such objects, as well as primitive objects already available in the engine such as particles, sphere, cube, lights, and physics routines such as collision detection.

Where my work begins, is writing the series of classes, responsible for generating the complex flattened quasicrystalline pattern, which my test scenario agents utilize, so as to adequately navigating my 3d world. (which amounts to about 3 of the 64 classes written *See Contents, 4.1.3, & 4.1.3.1 to 4.1.3.3*) I indeed freshly wrote 64 classes, however most of those classes concern accessing and, manipulating objects in the world, so though I wrote 64 classes, all that was required for most of such was patience, and an understanding of the unity3d pipeline, for accessing and manipulating objects amidst my 3d world. Differently though, it was the series of 3 morphing somatic quasicrystal neural network algorithm classes, which provided elevated degrees of difficulties. Though I indeed used unity3d to manipulate the positions of the objects involved amidst my flattened quasicrystal field, developing the algorithms to adequately tender my flattened quasicrystalline pattern was the truly difficult, arduous task.

This involved understanding vector projection mathematics, caley graphs, and coxeter dynkin diagrams, all confluencing into calculus II, and slightly beyond.



Section VI – References

References

6.0 Ryder Jack, 2012; "Basic Search Pattern Algorithms for Search and Rescue Operations"

i. http://citation.allacademic.com/meta/p_mla_apa_research_citation/5/8/5/4/6/p585460_index.html

6.1 Craig Reynolds, 1986; "Boids"

i. http://www.red3d.com/cwr/boids/

6.2 Claudio Rochhini, Tom Ruen, Jgmoxness, via Wikipedia user talks, 2014;

i.http://en.wikipedia.org/wiki/User_talk:Jgmoxness#Hypercube_projection_vector_scalar_qualms,

ii.http://en.wikipedia.org/wiki/User_talk:Tomruen#polygon_orthogonal_hypercube_basic_vector_scalar &

iii. http://en.wikipedia.org/wiki/User_talk:Rocchini#Good_day_--_Kaleidoscope_Algorithm_under_petrie_polygon_orthogonal_hypercube_vector_multiplier_PX_and_PY

6.3 Unity3d 4, 2014

i. http://unity3d.com/unity/

6.4 Gimp 2.6, 2014

i. http://www.gimp.org/

6.5 Audacity 2.0

i. http://unity3d.com/unity/

6.6 lSpeech

i. http://unity3d.com/unity/

6.7 Make Human 1.0 Alpha 8 i. http://unity3d.com/unity/

6.8 Unicortex/illumium/brain universe synonymous interface i. www.illumium.tk