Spherical Maze Generation

Xue Li[1] and David Mount[2]

1 **Department of Computer Science, University of Maryland, College Park**
`echolixue@gmail.com`

2 **Department of Computer Science, University of Maryland, College Park**
`mount@cs.umd.edu`

Abstract

In this paper, we proposed an automatic spherical maze generation framework. Traditional mazes are generated on two or three dimensional spaces, with regular grids as walls and paths. Instead of pure two or three dimensional mazes, we automatically generate maze on the surface of a sphere, which provides a unique and fresh maze solving experience. Because regular grids on spherical surface will result in nonuniform room spaces, we randomly generate uniformly distributed points on the sphere. Then we automatically create the wall mesh structures by triangulation and Voronoi graph generation. Gift-wrapping algorithm and incremental construction are both conducted for the spherical triangulation. To create the spherical maze, union-find algorithm is applied to remove parts of the walls, build the paths and join all the rooms.

1 Introduction

Maze, also known as labyrinths, is a puzzle consisting of multiple valid and invalid paths from a start position to a target position. Many algorithms for automatic maze generation have been proposed, classified by different criteria such as dimension, topology, tessellation and routing. A comprehensive list of the classification and algorithms can be found in website Think Labyrinth [1].

Traditional two dimensional mazes are created based on regular grids with one corner as the starting position and the corner on the diagonal as the target position or exit. There are also some three dimensional maze games providing some basic operations moving upwards and downwards. Instead of pure two or three dimensional mazes, we build mazes on the surface of a sphere, which makes gamers feel like wandering on a planet and provides a unique and fresh maze solving experience.

In general, there are two strategies to generate the mazes. One is by adding the walls of the maze during the algorithm. The other is by carving the paths, which is to remove some of the walls[1]. In this paper, path carving idea is utilized. In both of the strategies, we should define the positions of the walls in the maze. For 2D maze, a common way to define the walls is to utilize uniform grids in x and y directions. For spherical maze, we could also use equal-divided grids in longitude and latitude as the walls to add or remove. However, the areas of the rooms (spaces surrounded by walls) near tow polars will be more smaller than the areas near the equator. The non-uniform areas result from the non-uniform distributed grid points on the sphere. An alternate method to create rooms with uniform areas is to generate uniform spherical points(See Section 2), and to build walls based on the uniform distributed points.

After generating the points, we conduct spherical triangulation(See Section 3), and Voronoi diagram(See Section 4) to build the rooms. For spherical surfaces, triangulation

[1] http://www.astrolog.org/labyrnth/algrithm.htm

means to find the convex hull of the points. Incremental construction, gift wrapping, divide-and-conquer and graham scan are four commonly used convex hull construction algorithms[2], which can be extended from 2D cases to 3D cases. In this paper, we conduct incremental and gift wrapping for the spherical triangulation. To remove some of the walls and connected the exit room with the starting room, union-find algorithm is implemented by randomly and recursively remove walls that block different joint areas(See Section 5).

2 Uniform Spherical Points Generation

To generate points uniformly on the surface of a sphere, we randomly distribute points on the 3D spaces. To distribute the points evenly on the surface of a sphere, we utilize **Archimedes' Hat-Box Theorem**[3] as the generation strategy.

▶ **Theorem 1** (Archimedes' Hat-Box Theorem[3]). *Enclose a sphere in a cylinder and cut out a spherical segment by slicing twice perpendicularly to the cylinder's axis. Then the lateral surface area of the spherical segment S_1 is equal to the lateral surface area cut out of the cylinder S_2 by the same slicing planes, i.e.,*

$$S = S_1 = S_2 = 2\pi Rh, \tag{1}$$

where R is the radius of the cylinder (and tangent sphere) and h is the height of the cylindrical (and spherical) segment.

Therefore, our strategy to generate uniformly distributed points on an unite sphere is as follows,

- Randomly generate N points on a cylinder, with height ranging from -1 to 1.
- Project these N points to the surface of the unit sphere.

As a comparison, we also tried an intuitive method by randomly generating the three coordinates of the points on the unit sphere. To generate N uniformly distributed random points on the surface of a unit sphere, vectors X, Y and Z are randomly generated as three coordinates, whose distributions are uniform. To adjust each point to the unit sphere surface, the coordinates are calculated as,

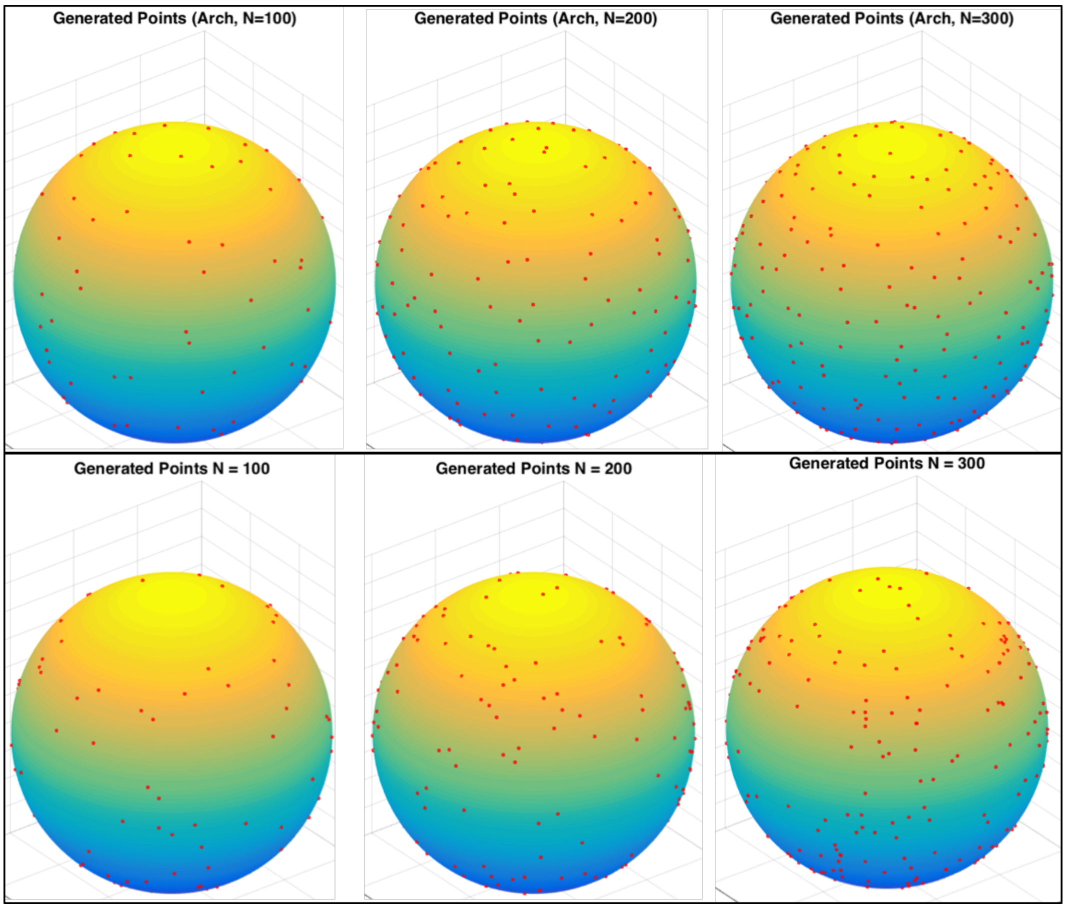
$$[X_i', Y_i', Z_i'] = \frac{X_i, Y_i, Z_i}{\sqrt{X_i^2 + Y_i^2 + Z_i^2}}, i = 1, ..., N$$

Figure 1 shows the comparison between the generated points according to Archimedes' Theorem method (above) and random 3 dimensional coordinates on the sphere (below), with the number of points N to be 100, 200 and 300. As shown in the figures, Archimedes' Theorem method generates more uniform distributed points than direct random methods. Considering different difficulties and visualizations of maze game, point number is recommended to be set from 50 to 200.

3 Spherical Triangulation

In this section, we applied two methods of Triangulation on sphere surface space, gift wrapping method and incremental construction.

Figure 1 Randomly generated points with Archimede method(above) and with direct random 3d coordinates(below), when N is set to be 100, 200 and 300.



3.1 Gift Wrapping

In computational geometry, the gift wrapping algorithm is to compute the convex hull of a given set of points. To find the convex hull is a process of triangulation.

Convex Hull. In our case of spherical surface, a *convex hull* constraint should be satisfied. All the generated triangles should be in the convex hull of all the points. To test whether a triangle in the convex hull, we should make sure there is no point outside the plane of the triangle from the sphere center. Therefore, any other point P and sphere center O should be on the same side of the plane defined by triangle ABC. Figure2 shows an example of a triangle failing to be a convex hull triangle.

Orientation of Four Points. Given four points A, B, C, and D in three dimensional space, $Ori_3(A, B, C, D)$ stands for the orientation of four points in three dimension. The expression is the signed volume of the parallelepiped determined by the vectors $t = A - B$, $u = B - C$, and $v = C - D$. By applying a right-hand rule, orient your right hand with fingers curled to follow the circular sequence BCD, it is positive if the thumb points towards A, negative if they occur in the mirror-image orientation, and zero if the four points are coplanar.

Figure 2 A triangle failing to be a convex hull triangle with a point P outside its plane.

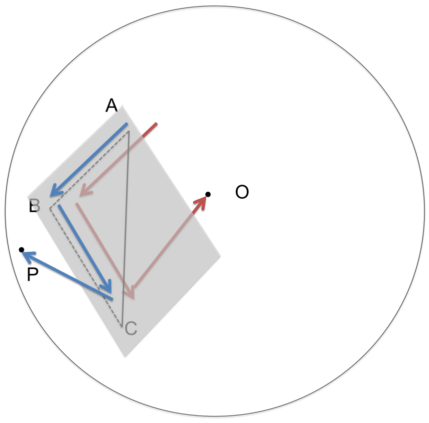
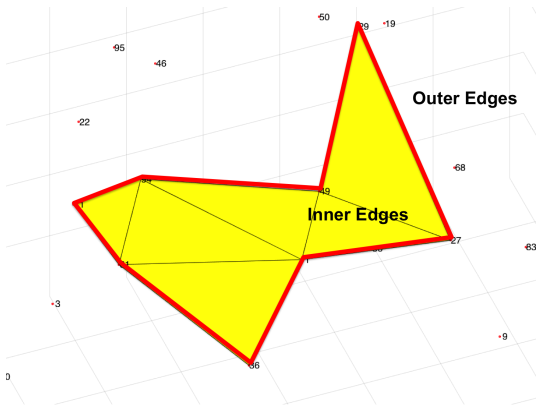


Figure 3 Outer Edges and Inner Edges during the insertion.



$$Ori_3(A,B,C,D) = sign(\begin{vmatrix} A_x & A_y & A_z & 1 \\ B_x & B_y & B_z & 1 \\ C_x & C_y & C_z & 1 \\ D_x & D_y & D_z & 1 \end{vmatrix}) \tag{2}$$

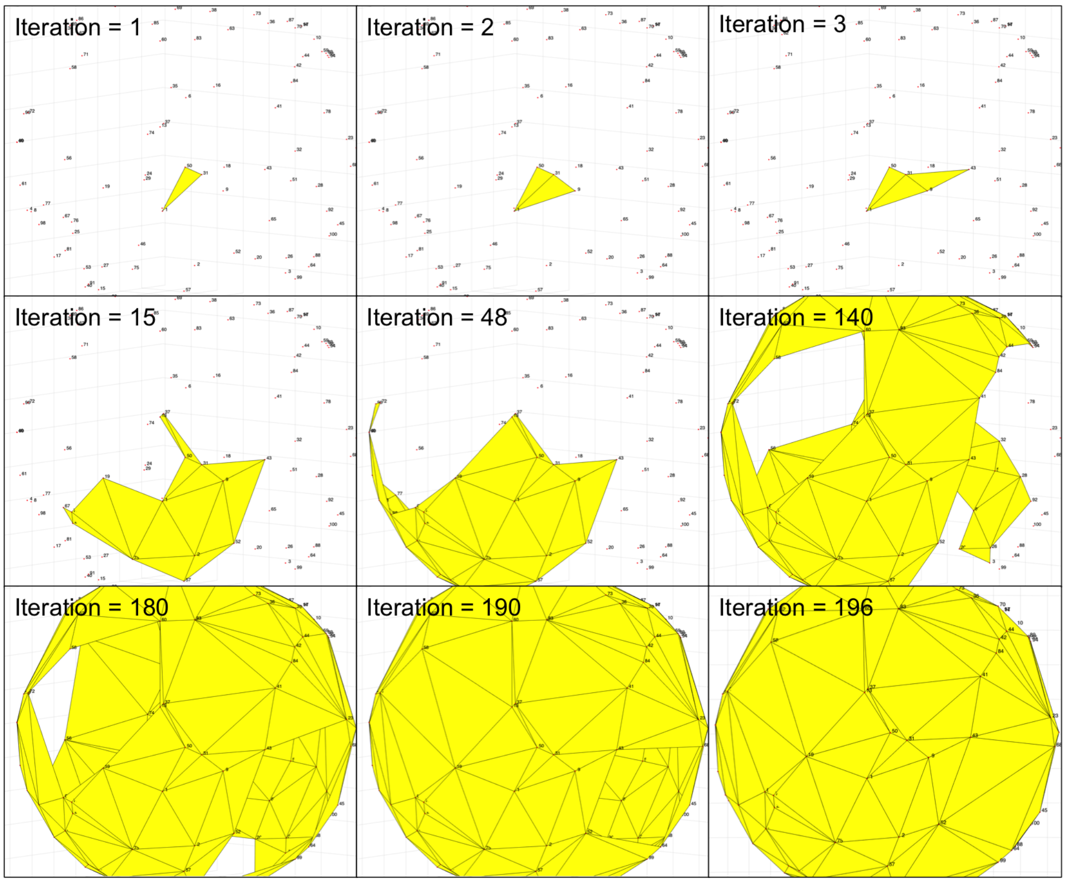
$$= sign(\begin{vmatrix} A_x - D_x & A_y - D_y & A_z - D_z \\ B_x - D_x & B_y - D_y & B_z - D_z \\ C_x - D_x & C_y - D_y & C_z - D_z \end{vmatrix}) \tag{3}$$

$$= sign(\begin{vmatrix} t & u & v \end{vmatrix}) \tag{4}$$

We can use the orientations of the point and the center regarding to the plane defined by the triangle as the convex hull test. If for any other point P, $Ori_3(A,B,C,P) \cdot Ori_3(A,B,C,O)$ is positive, triangle ABC is a part of the convex hull; If negative, triangle ABC fails the convex hull test, and should not be considered as the generating candidates.

Outer Edge. We start with a random point, find its minimum triangle as the initialized triangle, and iteratively add new vertex. Because each edge should connect two triangles at last, we define the edge connecting only one triangle as an outer (boundary) edge of the partial convex hull during the iteration, as shown in Figure3. In each iteration, we pick one of the boundary edges as an edge of the new triangle, along with the new point as the third

Figure 4 Gift-wrapping Spherical Triangulation when $N = 100$



vertex. After the new vertex is added, this edge is no longer an boundary edge.

The **end condition** of the iteration is that,

- all the points have been added,
- there are no outer edges needed to be connected.

The whole spherical gift wrapping algorithm is designed as Algorithm 1.

Algorithm 1 Gift Wrapping Algorithm

```
Input: OuterEdges = {}, AddedPoints = {}, N, AllPoints

Initialize a triangle P1P2P3,
    OuterEdges = {P1P2, P1P3, P2P3},
    AddedPoints = {P1, P2, P3}

while(AddedPoints.size != N AND OuterEdges.size != 0)
    CandidatePoints = AllPoints - AddedPoints
    Pi, Pj, P = PickOneOuterEdge(OuterEdges, CandidatePoints)
    AddedPoints += P
    AddTriangle(PiPjP)
    update OuterEdges
end while
```

Figure 5 A ray shooting from a outer point P to origin O, which hits a triangle on the previous convex hull.

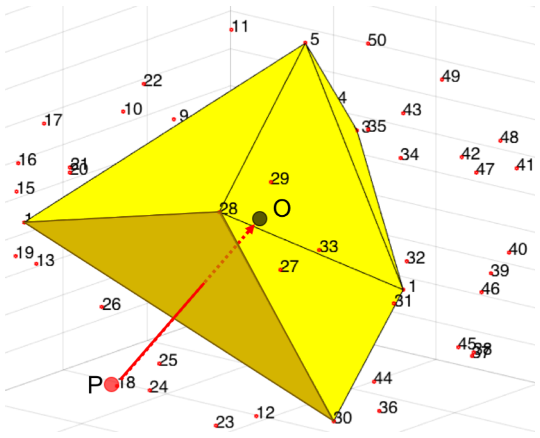


Figure 4 shows the gift-wrapping spherical triangulation results after different iterations when N = 100.

3.2 Incremental Construction

To compute the convex hull of the given points, gift wrapping method iteratively inserting new triangle to the existed partial convex hull. Instead of building parts of the final convex hull, incremental algorithm iteratively adds new points and creates the temporary convex hull from the previous the points. Incremental construction algorithm keeps maintaining the convex hull from existed points, and the convex hull is expanding during the iteration.

Ray Shooting. Since we are adding new points in the iteration, and the convex hull will grow to fit the new point, we need to decide which part of the previous convex hull should be updated. Ray shooting from the new point to the origin is applied to determine the infected triangle. The origin is maintained to be inside the convex hull. In practice, we use line segment instead of ray to avoid finding the triangle on the other side of the convex hull. Each existed triangle is checked whether it intersects with the shooting line segment until the intersecting triangle is found. To be noticed, because the intersecting target is a convex hull, every line passing the origin and a point outside of the convex hull will intersect with a triangle of the convex hull. Therefore, one and only one of the previous triangles exists to be found. Figure 5 illustrates the ray shooting from a outer point P to origin O, which hits a triangle on the previous convex hull.

Tetrahedron Initialization. During the iterative construction, a base-case convex hull should be initialized, which is a tetrahedron. Because in later ray shooting procedure, we need to maintain the origin inside the convex hull to make sure the rays to shoot insight, a tetrahedron containing the origin should be initialized.

There are two options to generate the initialized tetrahedron from the randomly distributed points. One is to randomly select four points from all the candidate points iteratively until one satisfying tetrahedron is found; The other option is to fix a tetrahedron at first and adjust one of the vertices iteratively until the simplex contain the origin. In this paper, since we have abundant candidate points, we choose the first method the randomly initialize the simplex. Figure 6 shows an initialized tetrahedron.

To check whether the origin O is within a tetrahedron $ABCD$, we use O to replace

Figure 6 An initialized tetrahedron

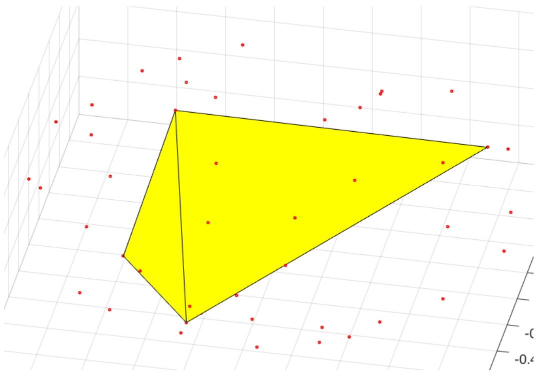
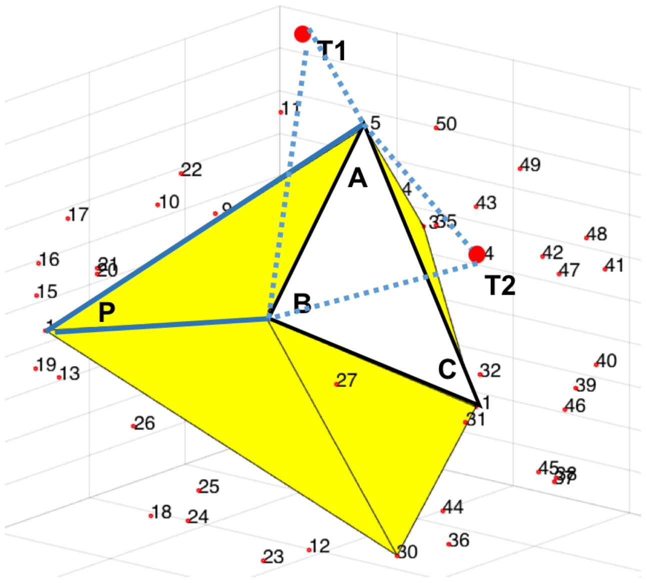


Figure 7 An illustration for checking edge AB, when adding two new points T_1(fail) and T_2(pass).



the vertex of $ABCD$ one by one and check whether the orientations of the tetrahedrons change. When O is inside $ABCD$, replacing each vertex of the simplex would not change the orders of the four points. If their orientations remain the same, which is $Ori_3(A, B, C, D) = Ori_3(O, B, C, D) = Ori_3(A, O, C, D) = Ori_3(A, B, O, D) = Ori_3(A, B, C, O)$, the tetrahedron $ABCD$ is ensured to contain the origin O.

Edge Checking. When adding a new point T to a triangle ΔABC to construct the new convex hull, we cannot directly remove triangle ΔABC and replace it with triangles ΔABT, ΔBCT, ΔACT, because the new added triangles may not satisfy the convex hull condition. When we are adding triangle ΔABT to the previous edge AB, the position of the other triangle containing AB should be considered. The other neighbor vertex is represented as P and the neighbor triangle as ΔABP. If the point to be added T and the origin O are on different sides of the plane defined by ΔABP, the plane fails the convex hull condition. Then we need to remove triangle ΔABP, and recursively check edges AP and BP until the convex hull conditions are satisfied and new triangles are added. Figure 7 is an illustration for checking edge AB, when adding two new points T_1 and T_2. T_1 will fail the edge checking and need to recursively check BP and AP. T_2 will pass the edge checking and we can directly add edges BT_2 and AT_2.

The whole incremental construction algorithm and the recursive edge updating algorithm are shown in listing 2 and 3 respectively.

Algorithm 2 Incremental Construction Algorithm

```
Input: AddedPoints = {}, AddedTriangles = {}, N, AllPoints

Initialize a tetrahedron $ABCD$ containing origin $O$,
    AddedPoints = {$A$, $B$, $C$, $D$}
    AddedTriangles = {$ABC$, $ACD$, $BCD$, $ABD$}

for each point $T$ to be added
    triangle Tri = IntersectTriangle(AddedTriangles, $T$, $O$)
    AddedTriangles -= Tri
    for each edge e in Tri
        updateEdge($e$, $T$)
    end for
end for
```

Algorithm 3 UpdateEdge

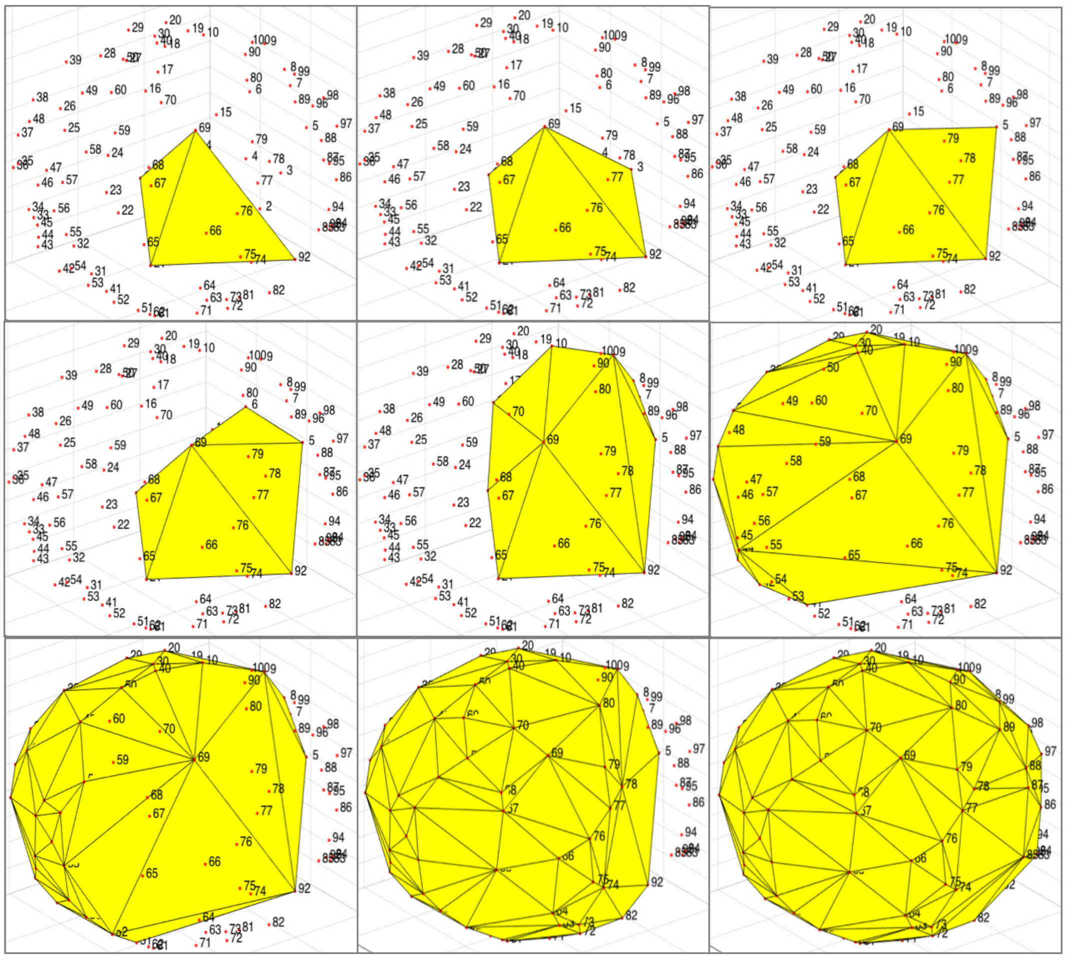
```
Input:  AddedTriangles, edge $e = AB$ to be updated,
        point $T$ to add, origin $O$

P = findNeighborPointOfEdge($A$, $B$, AddedTriangles)

if Orientation($A$, $B$, $P$, $O$) != Orientation($A$, $B$, $P$, $T$)
    AddedTriangles -= neighborTriangle $\Delta ABP$
    updateEdge($AP$, $T$)
    updateEdge($BP$, $T$)
else
    AddedTriangles += current triangle $ABT$
end if
```

Figure 8 shows the incremental construction method results after different iterations when N = 100.

4 Voronoi Graph Generation

In this section, our goal is to separate the spherical surface into different connected regions, which will be all the rooms in the maze. Given the triangulation mesh on the spherical surface computed in previous sections, we convert it into a connected Voronoi Graph. Each center point of all the triangles is considered as a corner of a wall. For each adjacent triangles pair, the two centers are connected together as a line segment, which constructs a wall of the maze. After the generation, all the triangles containing one vertex define a room area. Figure 9 shows a Voronoi graph generated from a triangulation mesh ($N = 100$).

Figure 8 Gift-wrapping Spherical Triangulation when $N = 100$

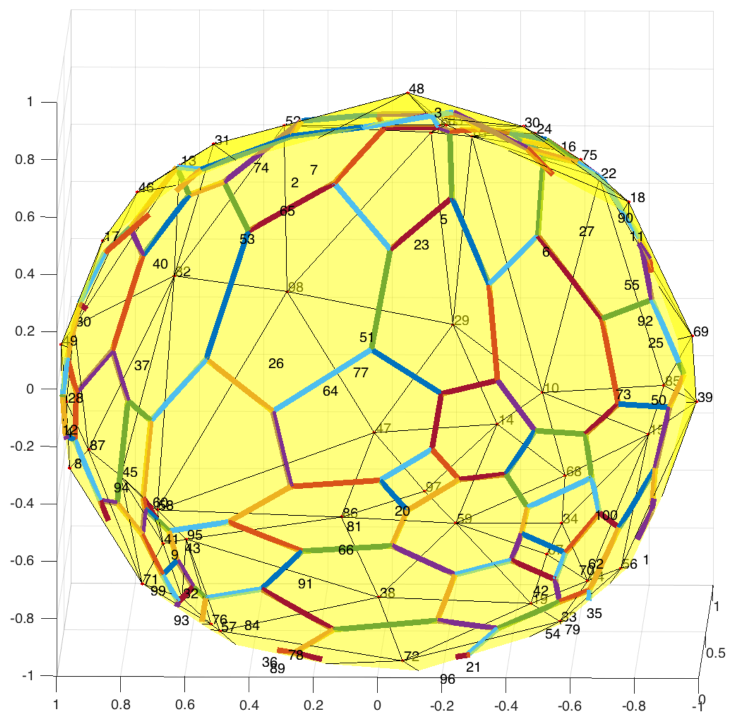


5 Spherical Maze Generation

After all the walls are created, *Union-Find* algorithm is utilized to remove some of the walls in order to connect all the rooms together.

The general idea of Union-Find algorithm is to iteratively select and remove a random wall to join its two connecting rooms until there is no isolate room. Each room is initialized with pointers to distinct labels. In the wall selection, the walls that connects different two labels are selected. After removing the selected wall, the neighbor two rooms are set to point to the same label. The iteration times is minimized as $R - 1$, where R is the number of rooms, because each room needs to be connected once to point to the final label, and the room initialized as final label does not need to be connected. The Union-Find algorithm is shown in Listing XX.

Figure 9 Voronoi graph generated from a triangulation mesh ($N = 100$)



Algorithm 4 Union-Find Maze Generation Algorithm

```
Input: Number of rooms $R$, $room_i, i = 1...R$, $Walls$

Initialize pointer of $room_i$: $pointer_i = i$

for k = 1 : $R-1$
    candidateWalls = findCandidateWalls(Walls)
    for each $wall$ in candidateWalls
        $i,j$ = findConnectingRooms($wall$)
        $label_i$ = findRoomLabel($pointer_i$, $i$)
        $label_j$ = findRoomLabel($pointer_j$, $j$)
        if $label_i$ == $label_j$
            continue
        else
            $pointer_i$ = $label_j$
            remove $wall$
            break
        end if
    end for
end for

//findRoomLabel()
$label_i$ = findRoomLabel($pointer_i$, $i$)
    while $pointer_i$ != $i$
        $i$ = $pointer_i$
    end while
    return $i$
```

Figure 10 shows the generated mazes setting the numbers of points as 50, 100, 150 and 200.

We can add more constrains to the wall selection algorithm to customize our mazes. For instance, we can choose the disconnected walls to remove in priority to make the maze more complicated in visualization. We can also stop the iteration as soon as there is a path between destination room and the starting room, which may create isolate rooms but simply the maze solving.

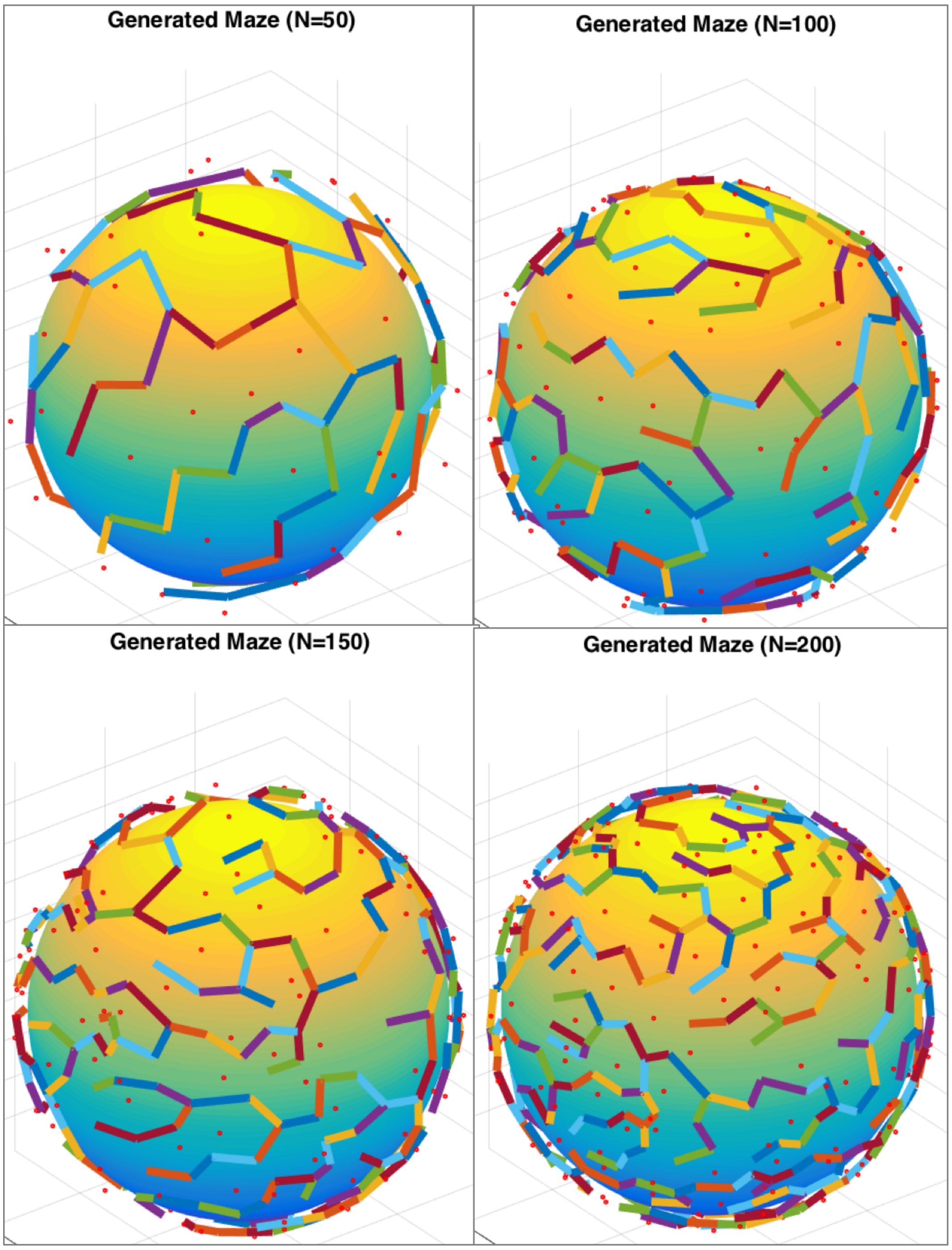
6 Conclusion

In this paper, an automatic spherical maze generation framework is proposed. Based on randomly generate uniformly distributed points on the sphere, we triangulate the points using gift-wrapping algorithm and incremental construction, and generate the walls based on Voronoi graph generation. During the spherical maze generation, we implement union-find algorithm to carve the paths and connect all the rooms.

References

1 Foltin, Martin. *Automated Maze Generation and Human Interaction.* Brno: Masaryk University Faculty Of Informatics (2011).
2 Devadoss, Satyan L., and Joseph O'Rourke. *Discrete and computational geometry.* Princeton University Press, 2011.
3 Cundy, H. and Rollett, A. "Sphere and Cylinder–Archimedes' Theorem." §4.3.4 in Mathematical Models, 3rd ed. Stradbroke, England: Tarquin Pub., pp. 172-173, 1989.

Figure 10 generated mazes with the numbers of points N as 50, 100, 150 and 200


Generated Maze (N=50)
Generated Maze (N=100)
Generated Maze (N=150)
Generated Maze (N=200)