# Smoothing Voronoi-based Path with Minimized Length and Visibility using Composite Bezier Curves

Ilya Makarov [1] and Pavel Polyakov [2]

[1] National Research University Higher School of Economics, Department of Data Analysis and Artificial Intelligence, 3 Kochnovskiy Proezd, 125319 Moscow, Russia, iamakarov@hse.ru, revan1986@mail.ru http://www.hse.ru/en/staff/iamakarov

[2] National Research University Higher School of Economics, 3 Kochnovskiy Proezd, 125319 Moscow, Russia, polyakovpavel96@gmail.com

**Abstract.** We present an obstacle avoiding path planning method based on a Voronoi diagram. We use a tactical visibility measure to obtain the shortest path length with the lowest local probability to be discovered based on the map topology. A Voronoi-based navigation mesh for finding the shortest smooth path with the lowest visibility along the path is used. The piecewise linear rough path in the Voronoi diagram is compared with collision free composite Bezier curves with shortest curve length. Whether we use visibility component or not, the smooth path length does not differ more than 12%. This allows us to use tactical information from map geometry without significant loss in path length.

**Keywords:** Path Planning, Voronoi Diagram, Bezier Curve

## 1 Introduction

Path planning and path finding problems play one of the main topics in robotic and automation fields, especially for dynamically changing environments. There are a large variety of algorithms for different tasks, such as in [1], [2], [3], [4], [5].

In [6] J. Reif proved that the path planning problem is PSPACE-complete. In addition to the storage problem, Reif also proved that the path planning problem is NP-complete. The exponential time and storage requirements combined together create an open research problem. The path planning is represented by two diverse strategies, separating offline computing of initially known BOT and global characteristics from constructing sensor-based local environment.

Voronoi diagrams are the simplest case of a $k$-nearest neighbour classification rule with $k = 1$. There are certain types of continuous locational optimization problems that can be solved through modifications of the Voronoi diagram (see [7]). In game programming, Voronoi diagrams are used to make a partition of a navigation mesh to find a collision free path in both global [1] and local environments [4,8]. The path is a piecewise linear path or a curve smoothed with the help

of splines. In the first case, the actor movements look bad when programming BOT with human-like behavior. The smooth path is constructed by connecting splines with some functional property, such as continuity of the first or second derivatives. The resulting path should be continuous itself. The authors of [9,10] make a path by splines through the way points on a map. A rather different approach is presented in the works by [2,4,11,12], in which authors use Bezier curves to construct the path based on obstacle vertices' points.

We combine these two approaches by choosing composite Bezier curves to implement the second strategy of writing obstacle-avoiding path. In [2] the author started to use a set of control points for one Bezier curve instead of using the reference points as way points. In [13] the connection of reference points by taking each point as a way point leads to longer path length, by comparing to their connection by taking each point as a control point of Bezier curves. Using a composite Bezier curve they obtain a reduction of 8.33% of the path length.

In this paper, we propose an obstacle avoiding smooth path planning algorithm with visibility component. We increase the average path length without visibility in comparison to the works [13,14] to obtain more realistic trajectories.

## 2 Voronoi-based navigation mesh

### 2.1 Definition

Regular navigation meshes consisting of convex polygons without any additional constraints seem to be perfect for path finding only until we need to calculate navigation characteristics using tactical properties of a map. Different approaches can be used to achieve this goal. However, without applying structural changes to the navigation mesh itself they are either not efficient enough or too difficult to implement. In this paper, we study a special navigation mesh type called *Voronoi-based navigation mesh*, which is designed to find paths considering properties such as visibility/cover, curvature of path trajectory and general penalties for traveling in particular areas. Let $P = \{p_0, p_1, \ldots, p_n\}$ be a set of points (called *sites*) placed on a map manually or automatically with precomputed tactical properties. Let

$$VD(p_i) = \{x :\ |p_i - x| \leq |p_j - x|,\ \forall j \neq i, x \in \mathbb{R}^2\} \tag{1}$$

be polygons of a navigation mesh. To support multi-layered environments, several surfaces (simply connected closed surfaces with $Z = const$) are also placed on a map. Each surface determines an area where a Voronoi diagram should be constructed, before being projected on a level geometry. A union of diagrams constructed in all surfaces is called a Voronoi-based navigation mesh. Examples of such map partitions are shown on Figure 1 and Figure 2. The navigation area is similar to Riemann's variety and is homeomorphic to the plane $\mathbb{R}^2$.

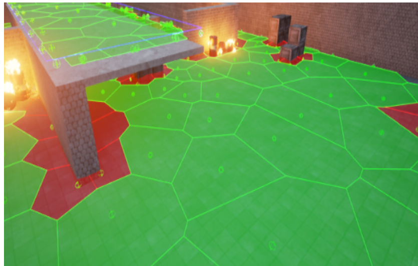

**Fig. 1.** VD-based Navigation Mesh

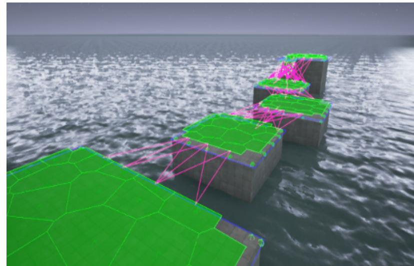

**Fig. 2.** Navigation Links

# 3 Construction and Memory Storage of Voronoi Diagram

## 3.1 Voronoi diagram

The Voronoi diagrams (VD) in each surface are independently computed in $\Theta(n \log n)$ time using Fortune's algorithm presented in [15]. The polygons lying outside an area determined by the corresponding surface are discarded. A diagram itself is stored in *DCEL* format, which guarantee $\Theta(n)$ memory footprint. The number of vertices and edges of VD are not greater than $2n$ and $3n$, respectively. In addition, a quad tree for each diagram is built so that the solution to the point location problem can be found in short time.

In [16] authors give a new randomized incremental algorithm for the construction of planar Voronoi diagrams in $O(n \log n)$ time and $O(n)$ space. Improvements of this work can also be found in [17].

## 3.2 Projection and obstacle detection

First, the polygon vertices are projected from surface's $Z$ onto the geometry. It is important to move vertices at some $\varepsilon$−height above the ground so that obstacle detection will work correctly for non-flat areas, such as stairs or hills. After that, the polygon's Z is calculated as an average of its vertexes.

Once navigation mesh is properly located in 3D space, a number of geometry flags for each polygon can be computed. We iterate through all edges of a mesh (there are $\Theta(n)$ edges) and perform the following: obstacle collision check along an edge; ground collision check under an edge; height above an edge evaluation for both crouch and jump. These flags are then copied from edges to adjacent polygons using logical OR. That is a way to gather information about the geometry of each polygon. This step requires $\Theta(n)$ ray casts.

### 3.3 Building navigation links

This step requires the exploration of jump-points on a navigation mesh and $O(n^2)$ ray casts in worst case. In practice, this number can be greatly reduced to $O(n)$ on real maps with bounded gravitation fields. For each of the polygons lying near the surface border, we use the following strategy to find link candidates: polygons should belong to different surfaces, both be navigable and the distance between the Voronoi sites inside them should not exceed a constant $d$ derived from gravity field. A set of polygons lying within a range $d$ from a given one in each surface can be found using a corresponding quad tree. In addition, link candidate is eliminated if a segment of polygons' sites intersects the edge of the border face which is not near the border.

Each link candidate is tested with two possible scenarios: jump (Figure 2) and walking off a ledge. The first one can be checked with a few ray casts and physics calculations. The second one minimizes unnecessary jumping and requires a number of ground height checks between two polygons.

### 3.4 Calculating tactical properties

We introduce a method for calculating visibility as a characteristic of Voronoi regions areas. All the values are computed offline as parts of map tactical properties and can be combined with on-line computed enemies' locations frag map.

Let *visibility* be a value from 0 to 1 indicating the amount of area visible from a polygon within a given range. One may argue how visibility should be converted into a number, but in fact, it gives us believable tactical results. Similar algorithm of calculating tactical map properties are presented in [18].

```
for Polygon in Polygons do
    if Polygon.IsNavigable() then
        Polygon.Visibility = 0
        for Other in Polygons do
            if Other.IsNavigable() then
                Hits = 0
                for (H1, H2) in { CrouchHeight, FullHeight} do
                    Hits +=
                    CheckCollision(Polygon.Location+ H1 ,Other.Location+ H2 )
                if Distance(Polygon,Other) < d then
                    Polygon.Visibility += Area(Other) * Hits / 4
        Polygon.Visibility = Min(1,Polygon.Visibility / c)
```

**Algorithm 1:** Visibility Computation

The threshold parameters $d$ and $c$ should be chosen experimentally depending on the game engine and the empirical estimation of the visibility map (see [19]). It is clear that $c$ should be about $\pi \cdot d^2$ but it is not necessary.

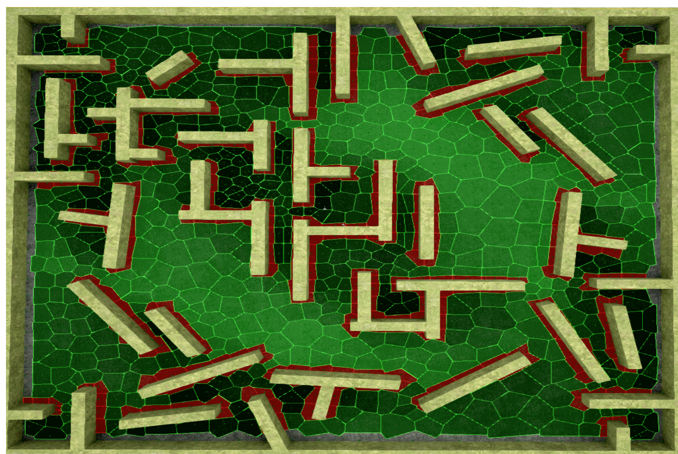

**Fig. 3.** Visibility Based on Voronoi Diagram

The result of the Algorithm 1 is shown on Figure 3. The darker the polygon is, the worse the visibility is in it. All non navigable areas have red colour.

We will further improve the representation of visibility and distance structures with BSP-trees.

## 4 Path planning

### 4.1 Path finding

The work with navigation proceeds in the following steps:

1. BOT makes a query to navigation system;
2. Navigation system uses $A^*$ (or I–ARA$^*$ anytime algorithm from [20] for large maps) finding a sequence of adjacent polygons on navigation mesh;
3. A sequence of polygons is converted into a sequence of points;
4. BOT receives a sequence of points and build a collision free path to walk.

### 4.2 Weight function and heuristics

We design the interface for interaction between querier and navigation system at each iteration of $A^*$ algorithm as follows:

```
float GetPenaltyForRotation();
float GetPenaltyMultiplierForCrouch();
float GetPenaltyForJump();
float GetAdditionalPenalty(PreviousPolygon,NextPolygon);
FVector2D GetInitialRotation();
```

Using the first three methods, a querier can manage penalties for path's curvature, crouching and jumping. The last function is a method for querier to influence navigation with respect to previous movement direction. GetAdditionalPenalty is used by querier to modify path penalty according to tactical properties of "previous" and "next" polygons and depending on its current preferences, similar to Markov's chains. There are also so-called general penalties, such as base cost, base enter cost and no way flag, which can be dynamically modified by any game event.

```
// Zeroth is Polygon visited before First
// Negative distance means no way
def Distance(Querier,Zeroth,First,Second,bJumpRequired):
if not Second.IsNavigable() or Second.bNoWay then
    return -1
Penalty = Querier.GetAdditionalPenalty(First,Second)
if Penalty < 0 then
    return -1
Direction ← Calculate direction from First to Second
if ∃ Zeroth then
    PreviousDirection ← Calculate direction from Zeroth to First
else
    PreviousDirection ← Querier.GetInitialRotation()
// <a,b> stands for dot product of normalized vectors
Rotation = 1 - <Direction,PreviousDirection>
Penalty += Rotation ∗ Max(0,Querier.GetPenaltyForRotation())
BaseCost = (First.BaseCost + Second.BaseCost) / 2
if First.bCrouchedOnly or Second.bCrouchedOnly then
    BaseCost *= Max(1,Querier.GetPenaltyMultiplierForCrouch())
Penalty += Distance(First,Second) * BaseCost + Second.BaseEnterCost
if bJumpRequired then
    Penalty += Max(0,Querier.GetPenaltyForJump()
return Penalty
```

**Algorithm 2:** Calculating Weight Function

For proper work of A* algorithm, each penalty is jammed to a limited range, so the resulting penalty is not less than the Euclidean distance, which is used as heuristics in our implementation.

### 4.3 Post processing path

Once a path is found, it should be converted into a point sequence. Apart from how this is done in regular navigation meshes, potential field or Funnel algorithms (see [19,21]) are not of any use because they can not smooth the paths furhter.

There are two ways polygons in navigation mesh can be connected: by link and by edge. In the first case, positions of points are chosen as coordinates of sites. In the second one, point can take different positions on an edge connecting two sequential polygons in a path. We use a weighted approach to choose this point in order to produce believable (in the sense of [22]) and natural looking paths.

Let $p_{i-1}$ be a point already chosen, and $p_{i+1}$ be a site position of the next polygon. Then $p_i$ is a point to be chosen. Also let $v_1$ and $v_2$ be vertices of an edge connecting polygons $i$ and $i+1$. We have:

$$d_j = distance(v_j, line(p_{i-1}, p_{i+1})), \tag{2}$$

$$w_j = rand(1, b)/d_j,\ j = 1,\ 2, \tag{3}$$

$$p_i = \frac{v_1 * w_1 + v_2 * w_2}{w_1 + w_2}, \tag{4}$$

where $b$ stands for the amount of randomness. We recommend to set $b$ around the value 2 so that it does not overweight distance modifier, and at the same time produce paths that appear to be “unique” (it looks unnatural when BOT is travelling along the same trajectory all the time, especially when there are several BOTs going one after another). The distance to the line modifier is used for removing zigzag effects and as a first attempt to smooth the path. Figure 4 shows how a typical path may look after this step.

**Fig. 4.** Post Processed Random Path

**Fig. 5.** Smooth Path

### 4.4 Building Bezier curve

When developing a BOT navigation, *smoothing* is one of the key steps. It is the first thing for a human to distinguish a BOT from a human player. Several approaches can be used to smooth movements. For example, a potential method when a force proportional to distance from obstacle is applied to a BOT, limiting rotation rate of a BOT when moving between points, and splines. *Bezier curves* seem to be the most suitable because they lie strictly inside the convex hull of way points[1]. This fact guarantees that the smoothed path is collision-free, and that a BOT will not be stuck into an obstacle. In addition, the path does not pass through a majority of points it is built on, and that is actually why we do not eliminate the points with a Funnel or another algorithm. We use all these points to smooth paths.

In this section we present how to build a composite Bezier curve, as shown on Figure 5. We also improve the results of [13,14], obtaining worse length path for realistic motion and detour around the obstacles.

The first step is an insertion of additional points into a path so that the distance between sequential ones is greater than some predetermined constant. In practice, it greatly enhances results of the further step where we split a path into several pieces to build a Bezier curve in each one. Points of each piece of the path should lie in a collision-free convex hull so we have to choose the subsequences of points with no obstacles in their convex hull and ensure that there are no any holes under it. We want to minimize over all partitions:

$$\sum distance(Piece.FirstPoint, Piece.LastPoint) \to \min \tag{5}$$

Also if two partitions lengths differs on a small $\varepsilon$ then we choose the one with less parts as a composite Bezier curve produces worse smoothing when it consists of a big number of small curves. This strategy roughly minimizes a length of the resulting curve and allows better smoothing behavior on turns.

[1] http://www.ams.org/samplings/feature-column/fcarc-bezier

The following algorithm for path subdivision is used:

```
CollideArray[N][N] ← initialized by false
Partition[N][N] ← struct { Value,Parts,End }
for k in range(1,N) do
    for i in range(0,N - k) do
        j = i + k
        if CollideArray[i][j - 1] or CollideArray[i + 1][j] or
         CheckCollision(PathPoints[i],PathPoints[j]) then
            CollideArray[i][j] = true
        if CollideArray[i][j] then
            Partition[i][j].Value = +∞
        else
            Partition[i][j].Value = distance(PathPoints[i],PathPoints[j])
            Partition[i][j].Parts = 1
            Partition[i][j].End = j
        for l in range(i + 1,j) do
            ValueAlternative = Partition[i][l].Value + Partition[l][j].Value
            PartsAlternative = Partition[i][l].Parts + Partition[l][j].Parts
            if Partition[i][j].Value > ValueAlternative + ε or
             (abs(Partition[i][j].Value - ValueAlternative) < ε and
             PartsAlternative < Partition[i][l].Parts) then
                Partition[i][j].Value = ValueAlternative
                Partition[i][j].Parts = PartsAlternative
                Partition[i][j].End = l
```

**Algorithm 3:** Composite Bezier Curve Construction

Once array Partition is computed, we can get indexes of the path subdivision using a simple recursive backtracking algorithm.

```
def Backtracking(Partition,i,j,Indexes):
if j == Partition[i][j].End then
    Indexes.Add(j)
else
    Backtracking(Partition,i,Partition[i][j].End,Indexes)
    Backtracking(Partition,Partition[i][j].End,j,Indexes)
```

**Algorithm 4:** Backtracking

Our next step is removal of crowded control points in each part of the path, therefore decreasing the curvature and the length of a curve. This can be done directly by verifying that points are not closer than some $e > 0$ and that there are not too many control points.

The final step of building a composite Bezier curve is the “repairing” of a derivative at parts’ connection locations. This can be achieved by inserting additional control points at these locations in order to ensure that we have at least continuously differentiable trajectories. Complete algorithms for constructing composite Bezier curves can be found in [13,14].

Now, a BOT can start moving along it. In general, there may be difficulties with the correct calculation of the current and the next positions on the curve, so we use the simplest prediction for small time interval and correct path finder.

## 5 Experiment and Conclusion

In practice, the contribution of visibility component to remain undetected during BOT motion is very low if we are not taking into account the enemies’ movements. We consider the relative dependence of the smooth low-visibility path length with the length of the shortest path obtained by Recast navigation mesh.

We take 1000 different start and end locations on the map shown on Figure 3. The average difference (AD, %) and variance of difference (VD, %) from the shortest path length for the next paths were calculated during the experiment: [Case 1 & 2] Piecewise path with visibility penalty set to 0 and 10, respectively; [Case 3 & 4] Smoothed path with visibility penalty set to 0 and 10, respectively. We perform the exepriment in Unreal Engine 4 [2]. Results are shown in Table 1.

**Table 1.** Comparison with the Shortest Path Length

| \ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| AD, % | 5.9 | 12.5 | 27 | 37.2 |
| VD, % | 0.3 | 1 | 0.7 | 2.1 |

We can see that the resulting difference between the smooth paths with and without a visibility component does not exceed 10–12%, so taking into account tactical information seems to be a useful decision.

The difference in 15–25% between smooth path length from our algorithm and the results from [13,14] is not too significant because we mainly focus on constructing realistic randomized paths for BOTs. When implementing such an algorithm in 3D first-person shooter, our algorithm provides more realistic motion behaviours than the minimized CBR-based path, while saving the property of the path to be suboptimal.

[2] https://www.unrealengine.com

Figure 6 illustrates the visual comparison between piecewise linear path, smoothed path and the shortest path drawn in yellow(1), pink(2) and green(3) color respectively.

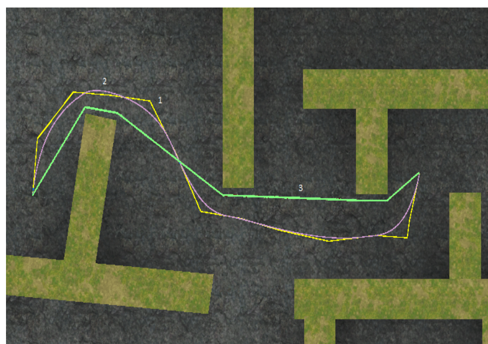


**Fig. 6.** Path Comparision

We will continue the evaluation of this method by applying it to BOTs' formations and by simplifying the computational part of the visibility measure, expanding Markov's processes based approach for BOT motion from [23].

## References

1. Bhattacharya, P., Gavrilova, Marina L.: Voronoi diagram in optimal path planning. In: 4th IEEE International Symposium on Voronoi Diagrams in Science and Engineering. (2007) 38–47
2. Choi, J.w., Curry, Renwick E., Elkaim, Gabriel H.: Obstacle avoiding real-time trajectory generation and control of omnidirectional vehicles. In: American Control Conference. (2009)
3. Gulati, S., Kuipers, B.: High performance control for graceful motion of an intelligent wheelchair. In: IEEE International Conference on Robotics and Automation. (2008) 3932–3938
4. Guechi, E.H., Lauber, J., Dambrine, M.: On-line moving-obstacle avoidance using piecewise bezier curves with unknown obstacle trajectory. In: 16th Mediterranean Conference on Control and Automation. (2008) 505–510
5. Nagatani, K., Iwai, Y., Tanaka, Y.: Sensor based navigation for car-like mobile robots using generalized voronoi graph. In: IEEE International Conference on Intelligent Robots and Systems. (2001) 1017–1022
6. Reif, J. H.: Complexity of the mover's problem and generalizations. 20th Annual IEEE Conference on Foundations of Computer Science (1979) 421–427
7. Okabe, A., Suzuki, A.: Locational optimization problems solved through voronoi diagrams. European Journal of Operational Research **98**(3) (1997) 445–456
8. Mohammadi, S., Hazar, N.: A voronoi-based reactive approach for mobile robot navigation. Advances in Computer Science and Engineering **6** (2009) 901–904

9. Eren, H., Fung, C.C., Evans, J.: Implementation of the spline method for mobile robot path control. In: 16th IEEE Instrumentation and Measurement Technology Conference. Volume 2. (1999) 739–744
10. Magid, E., Keren, D., Rivlin, E., Yavneh, I.: Spline-based robot navigation. In: International Conference on Intelilgent Robots and Systems. (2006) 2296–2301
11. Hwang, J.H., Arkin, R.C., Kwon, D.S.: Mobile robots at your fingertip: Bezier curve on-line trajectory generation for supervisory control. In: IEEE International Conference on Intelligent Robots and Systems. Volume 2. (2003) 1444–1449
12. Škrjanc, I., Klančar, G.: Cooperative collision avoidance between multiple robots based on bezier curves. In: 29th International Conference on Information Technology Interfaces. (2007) 451–456
13. Ho, Y.J., Liu, J.S.: Smoothing voronoi-based obstacle-avoiding path by length-minimizing composite bezier curve. In: International Conference on Service and Interactive Robotics. (2009)
14. Ho, Y.J., Liu, J. S.: Collision-free curvature-bounded smooth path planning using composite bezier curve based on voronoi diagram. In: IEEE International Symposium on Computational Intelligence in Robotics and Automation. (2009) 463–468
15. Fortune, S.: A sweepline algorithm for voronoi diagrams. In: 2nd Annual Symposium on Computational geometry. (1986) 313–322
16. Guibas, Leonidas J., Knuth, Donald E., Sharir, M.: Randomized incremental construction of delaunay and voronoi diagrams. Algorithmica **7**(1) (1992) 381–413
17. van Toll, W.G., Cook, A.F., Geraerts, R.: A navigation mesh for dynamic environments. Comput. Animat. Virtual Worlds **23**(6) (November 2012) 535–546
18. Funge, J., Millington, I.: Artificial Intelligence for Games. M. Kaufmann (2009)
19. Barraquand, J., Latombe, J.: Robot motion planning: A distributed approach. International Journal of Robotics Research **10**(6) (1991) 628–649
20. Koenig, S., Sun, X., Uras, T., Yeoh, W.: Incremental ARA$^*$: An incremental anytime search algorithm for moving-target search. In: Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling. (2012)
21. Ish-Shalom, J.: The funnel algorithm and task level robot control. In: IEEE International Conference on Robotics and Automation. Volume 4. (1987) 25–32
22. Hingston, P.: Believable Bots: Can Computers Play Like People? Springer (2012)
23. Makarov, I., Tokmakov, M., Tokmakova, L.: Imitation of human behavior in 3D-shooter game. In: 4th International Conference on Analysis of Images, Social Networks and Texts. (2015) 64–77