CREATING A 2D VISIBILITY SHADOW EFFECT ON UNITY PLATFORM WITH ROBUST C# CODE

Prepared by: Chicheng Zheng

Supervisor: Prof. Clark Verbrugge

Preparation for COMP396 Undergraduate Research Project

McGill University

April 2019

Abstraction

Guards are very common in game, therefore it is important to implement the visibility of the guard in a map with obstacles. This project aims to create a 2D visibility shadow effort and apply it on real scenario.

目录

1. Introduction

1.1 Motivation summary

The visibility effect of a guard is a core part when implementing a role play games. It is important to compute the visibility polygon corresponding to different locations of view point.

1.2 Implementation summary

The realization is based on Art Gallery Algorithm ("*Art Gallery Theorems and Algorithms",* 1987). Basically, the idea is to find the all the triangles that form the visibility polygon, and we can get those triangles by emitting the ray from the view point.

1.3 Test case summary

The project will test different shape of obstacles, since whether an obstacle concave or not may cause a bit different. Basically, there are 4 situations need to be considered, which are concave polygon, convex polygon, concave boundary and convex boundary.

1.4 Supported Platform

The effect is implemented on the Unity 3.19 Platform.

2. Background and Related Work

2.1 Background

2.1.1 Visibility Polygon

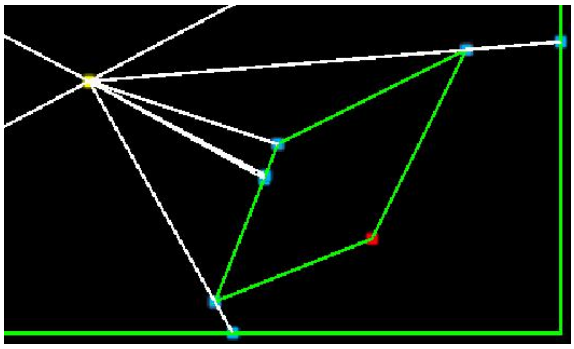
In computational geometry, the visibility polygon or visibility region for a point p in the plane among obstacles is the possibly unbounded polygonal region of all points of the plane visible from p. The visibility polygon can also be defined for visibility from a segment, or a polygon. ("Visibility Polygon", 2019)

2.2 Related Work

In order to understand meaning of some variable, I self-defined some terminology.

2.2.1 Hit Point

All the hit points generated by the ray cast from the view point, the visibility polygon is surrounded by the hit points. For example, all the blue points are hit points.



2.2.2 Sight Range

In the real world, it impossible for a guard to see infinity far away. Therefore, he has a range of visibility. Sight Range is the max value it could see from the view point.

2.2.3 Sight Angle

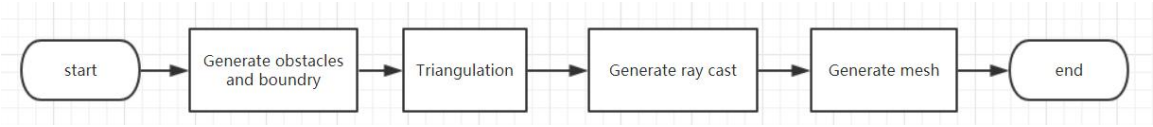
In the real world, it impossible for a guard to see all the things around it. Therefore, the sight angle defines the max range of perspective.

2.2.4 View Point

It is the guard position, and this project is focusing the visibility effect of related to this view point.

3. Implementation

The realization is based on Art Gallery Algorithm ("*Art Gallery Theorems and Algorithms",* 1987). The detailed process flow chart is as following.



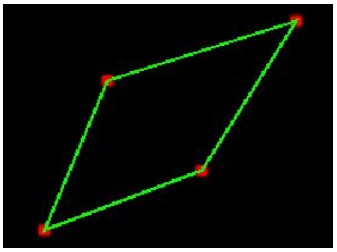
Before introducing the detailed implementation, there exists some preparatory work to do.

3.1 preparatory work

The following part is going to introduce some basic math and graphic concept in advance including the definition of cross product as well as

concave and convex polygon.

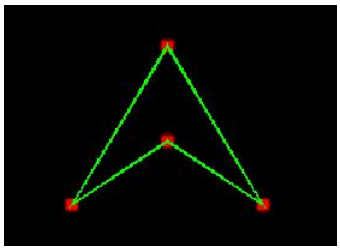
3.1.1 Convex Polygon



("Convex", 2018)

A polygon in which no line segment between two points on the boundary ever goes outside the polygon.

3.1.2 Concave Polygon



("Concave", 2018)

A polygon which is not convex.

3.1.3 Cross Product

("Cross Product", 2019)

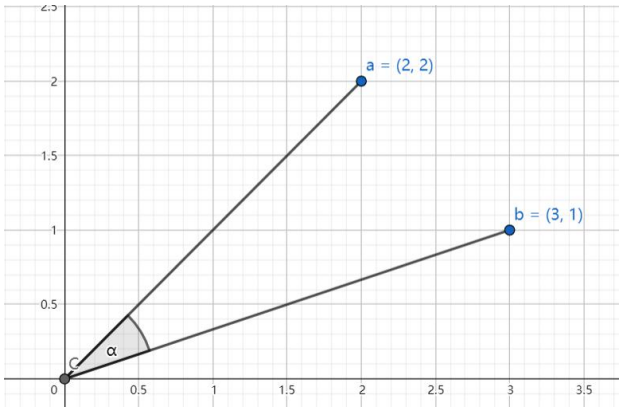
The cross product of two vectors **a** and **b** is defined as **a**.x*__b__.y-__a__.y*__b__.x. If the result is smaller than 0, which means the unsigned angle between **a** and **b** is clockwise from **a** to **b**.

For example,

a = (2, 2), **b** = (3, 1)

α is the unsigned angle between **a** to **b**.

crossProduct(**a**,**b**) = 2 - 6 < 0, which means **a** gets angle α by go clockwise to **b**.

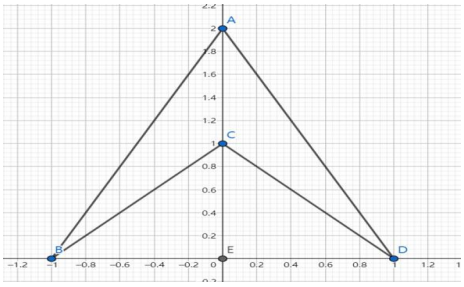


crossProduct(**b**,**a**) = 6 - 2 > 0, which means **b** gets angle α by go

anticlockwise to **a**.

3.1.4 Split-able Point

If the polygon is a concave polygon, then we need to split it into a series convex piece. First, we need to find the split-able point. Basically, the split-able point is point that has different clockwise or anticlockwise direction from another vertex.



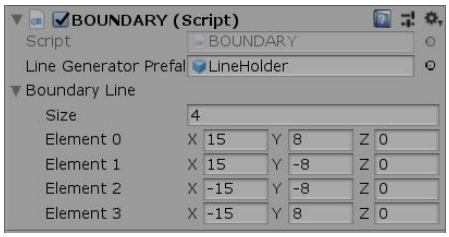
For example,

Point C can be a split-able point, cause CDA, DAB, ABC are all anticlockwise, only BCD is clockwise.

3.1 Generate obstacle and boundary

3.1.1 Boundary

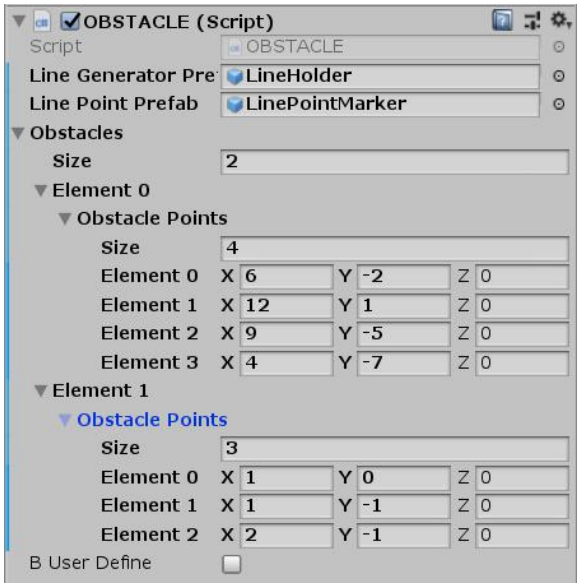
The Boundary Manager receive a list of Vector2D points denoted the vertex of the boundary,



it generates the edge collider 2D by loop through all the vertex and choose the adjacent pairs. We will have a list of edge collider before we generate the visibility area.

3.1.2 Obstacle

The Obstacle Manager receive a list of obstacles. Each obstacle is a list of Vector2D points.



Based on the list of obstacles, I generated a list of Polygon Collider 2D by loop through all the obstacles. This is basically same as generating the edge Collider 2D.

3.1.3 Triangulation

After we generate all the Polygon Collider 2D for each obstacle. We need to detect whether it is a concave obstacle or convex obstacle. If it is a concave obstacle, then we need to do triangulation. This mainly because when the obstacle is a concave, it may have two hit point with one line. However, the mechanism of Collider in Unity can only detect one collision. (“The triangulation in Unity”, 2014)

3.1.3.1 Determine whether three point is clock wise or not

Algorithm: isClockWise

Input: Vector2 a, Vector2 b, Vector2 c

Output: true if clockwise, otherwise false.

```
return (a.x - c.x) * (b.y - c.y) - (b.x - c.x) * (a.y - c.y) < 0);
```

3.1.3.2 Determine whether it is a concave polygon or not.

Algorithm: isConcave

Input: polygon

Output: true if the polygon is concave, otherwise false.

```
foreach (AdjacentEdge e in polygon)
    Compute e.isClockwise(), all the adjacent edge must have
    same result if it is a concave polygon, otherwise it is a convex
    polygon.
endfor
```

3.1.3.3 Find the split-able point

This algorithm is to determine the point of given index can split or not.

Algorithm: IsSplitIndex

Input: vertices,

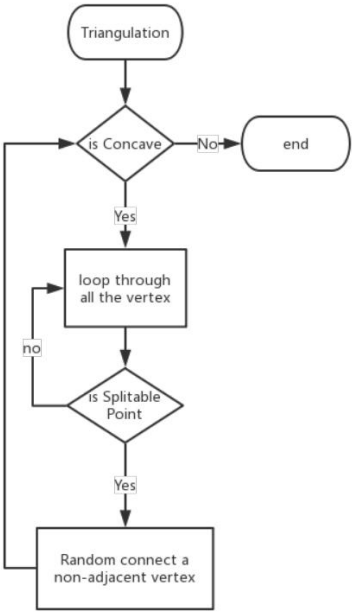
index,

Output: true if the point of given index can split, otherwise false.

```
Generate a temporary triangle polygon with vertices[previous],
vertices [index], vertices [next] three points.
```

```
foreach (Vertex v in vertices)
    // if the point is inside the polygon
    if (IsPointInsidePolygon(v, temp)) return true;
endfor
```

3.1.3.4 Triangulation



Randomly choose a point connected with the split-able point. Repeat the split process, until all the polygon is convex.

3.2 Generate Ray Cast

Unity has already implemented a function named Physics2D.RaycastAll(). Basically, this function casts a ray against colliders in the Scene, returning all colliders that contact with it.(“Unity Documentation”, 2018)

3.2.1 Stable view point

Loop through all the vertex of obstacles in the map, generates the ray cast from the view point position to the vertex, use the given function RaycastAll() to get the cast result which is an array. Then we loop through the array to check verify each hit result.

3.2.1.1 Generate Rays Cast

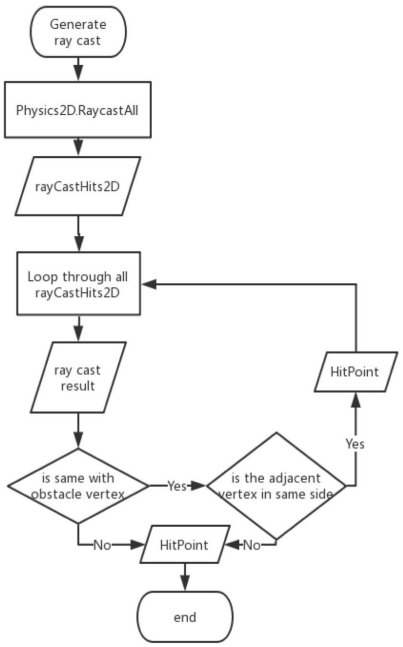
Loop through all the vertex in obstacles and generate ray cast corresponding to this vertex.

Algorithm: GenerateRaysCast
Input: All the obstacles

```
foreach vertex in the obstacle
    GenerateRayCast ();
endfor
```

3.2.1.1 Generate Ray Cast

Generate ray cast for a specific vertex in an obstacle and get it hit points.



Algorithm: GenerateRayCast

Input: viewpoint,

obstacle,

index, (indicate which point in the obstacle)

direction, (the direction from view point to the vertex of the obstacle)

```
RaycastHit2D[] rayCastHits2D
                    = Physics2D.RaycastAll(viewpoint, direction);
foreach rayCastHitResult in rayCastHits2D
    // if the hit result is the same position as obstacle position
    if (rayCastHitResult == obstacle[index])
        // If the neighbor vertices of the hitting result are both in
        the one side, keep the hitting result
        if (isSameSide()) { continue;}
        else { break;}
```

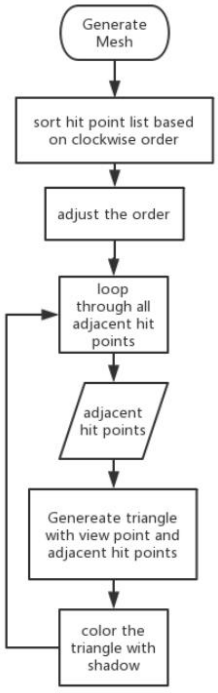
```
        else { break;}
    endfor
```

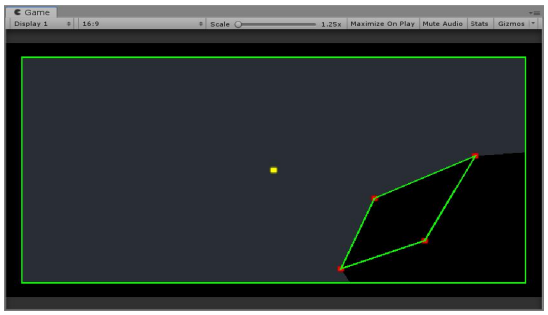
3.2.2 Movable view point

We want to implement a moveable view point. It can be implemented by connected the viewpoint location and the mouse location. We can generate ray cast in every frame, and then destroy it in a short delay.

3.3 Generate Mesh

In order to generate the following visibility effect from the hit point given by the ray cast.





3.3.1 Sort the hit point in a clockwise order

First, we need to sort all the hit point in a clockwise order (start from base vector (1, 0) by default). I use the cross product to determine the unsigned angle between hit point and base vector is clockwise or not.

3.3.2 Adjust the order

For those points are on the same line with view point, we need to determine it order, from inside to outside or not. Basically, I implement an algorithm to test whether on the same obstacles or not.

This algorithm is used to compare two vectors based on their angle in clockwise direction.

Algorithm: compareByAngle
Input: v1, v2
Output: 0 if the two vectors are the same
1 if v1 larger than v2
-1 if v1 smaller than v2

```
float angle1 = Vector2.Angle(v1, new Vector2(1, 0));
float angle2 = Vector2.Angle(v2, new Vector2(1, 0));
if (v1.y > 0) { angle1 = 360 - angle1; }
if (v2.y > 0) { angle2 = 360 - angle2; }
if (angle1 == angle2)) return 0;
else return angle1 > angle2 ? 1 : -1;
```

This algorithm is to determine whether a point is inside a line segment or

not.

Algorithm: isInsideSegement

Input: Vector point,

Vector endPoint1,

Vector endPoint2

Output: true if the point is inside the line segment, otherwise, false.

```
if ((point == endPoint1) || (point == endPoint2)) return true;
if ((point.x < endPoint1.x) && (point.x > endPoint2.x)
  || ((point.x < endPoint2.x) && (point.x > endPoint1.x))) {
        return (point - endPoint2).normalized
                == (endPoint1 - point).normalized);
}
return false;
```

This algorithm is to determine whether two points v1, v2 are on the same edge of an obstacle.

Algorithm: isInSameObstaclesEdge

Input: Vector v1, Vector v2

obstacle

Output: true if the two points are in the same edge of an obstacle, otherwise, false.

```
    bool result = false;
    for i = 0 to obstaclePoints.Length
        result |= (isInsideSegement(v1, obstacle[i], obstacle [i + 1])
            && isInsideSegement(v2, obstacle[i], obstacle [i + 1]))
    endfor
```

```
    return result;
```

This is the main part of how to adjust the order of the list.

Algorithm: Adjust the hit point order

Input: list (all the hit points sorted by clockwise)

Output: list (a list after swapping some points order)

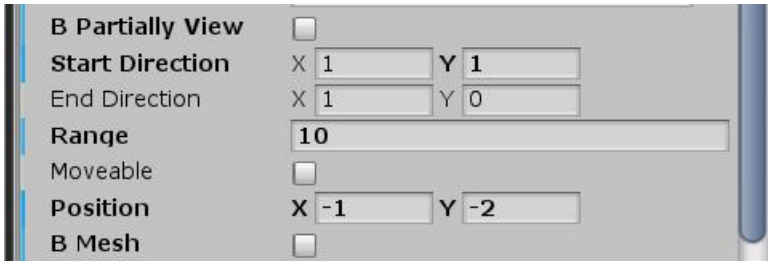
```
    int cur = 0;                // the index of current node
    int pre = list.Count - 1; // the index of previous node
    int next = 1;               // the index of next node
    while (cur < list.Count + 1) {
        int end = cur;
        while (end + 1 < list.Count
            && compareByAngle(list[end], list[end + 1]) == 0)
            end++;
            next++;
            next %= list.Count;
        endwhile
        // List with index from "cur" to "end" are all in the same line
        if (end > cur)
            HitPoint preNode = list[pre];
            HitPoint nextNode = list[next];

            if (!isInSameObstaclesEdge(list[end], nextNode)
                || !isInSameObstaclesEdge (list[cur].location, preNode)
                    // reverse the order from cur to end
                    reverseOrder(list, cur, end);
            endif
```

```
        endif
        cur = end + 1;
        pre = end;
        next = cur + 1;
        next %= list.Count
    endwhile
```

3.4 Implement the partially viewing && range limit

In addition to get the overall polygon with 360 degrees, in real scenario, instead of 360 degrees of view and infinity range, normally a character only has limited range of sight. Therefore, I further provide the interface for user to customize the parameter based on different purpose. Basically, now we have a relatively useful tool to handle the visibility problem.



The following panel is on the Game object named "InitialViewPoint". *"BPartiallyView"* is used for whether the user want to have a 360 degrees range of view or not. If user choose yes, then the next two "start direction" and "end direction" is used for determining the start direction and end direction in clockwise order.

"Range" is the parameter for user to specify the limit range for character, and it will consider infinity large if the value is set less or equal than 0.

"Moveable" is used for whether the user want the view point move with

the mouse or not. If choose no, then "Position" is used for the stable view position.

"BMesh" is used for whether the user want to see the shadow effect or not. If the user chooses no, it will only generate the debug lines cast.

3.4.1 partially viewing

3.4.1.1 Exclude the line cast

We need to exclude the line cast that out of the range.

This algorithm is to determine a vector is in the range of partially viewing or not.

Algorithm: isInsideClockRangeOfTwoVector

Input: Vector2 start,

Vector2 end,

Vector2 test

Output: true if inside the range, otherwise false

```
if (test.normalized == start.normalized)
    || test.normalized == end.normalized) return true;
// angle1 represent the angle from start to test in clockwise order
float angle1 = Vector2.Angle(start, test);

// angle2 represent the angle from test to end in clockwise order
 float angle2 = Vector2.Angle(test, end);

// angle3 represent the angle from start to end in clockwise order
float angle3 = Vector2.Angle(start, end);
```

```
    return float1 + float2 == float3;
```

3.4.2 Range Limit

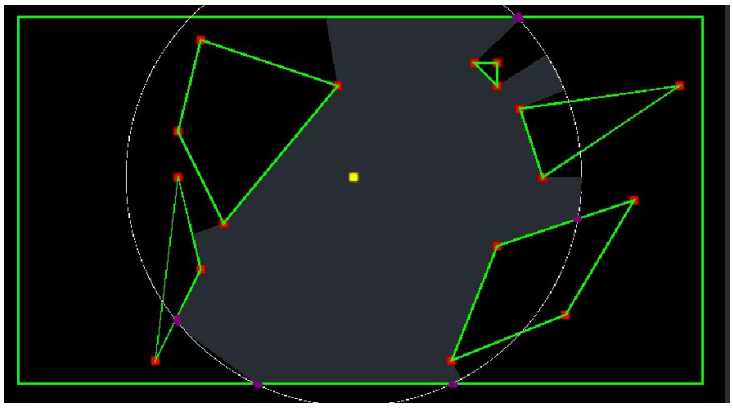
3.4.2.1 Truncation the ray cast

Now we add an additional feature, which means we need to make sure every point in the hit point list is not out of range. Otherwise, we truncation the ray cast.

```
Algorithm: truncate the ray cast
Input: Vector hitPoint
       Vector direction (the direction from view point to hit point)
    // If the hit point is out of the range
    if (hitpoint.magnitude > range)
        …
        // truncate the ray cast
        hitPoint = viewpoint + direction.normalized * range;
    endif
```

3.4.2.2 Add additional hit point

Implementing the range feature can cause additional hit points.
For example,



All the purple hit points are the extra hit point when implementing the range feature. We need to compute the intersection points between the circle of the range view and each obstacle polygon.

Algorithm: AddAdditionalHitPoint

Input: List<Obstacle> Obstacles,

Vector viewPoint,

int range

```
foreach (Obstacle obstacle in Obstacles)
    GenerateintersectionPoint(obstacle, viewPoint, range);
endfor
```

This algorithm is to generate hit points for a specific obstacle

---

Algorithm: GenerateintersectionPoint

Input: Obstacle obstacle

Vector viewPoint

int range

```
foreach (Vector2 edge in obstacle)
    // This method is getting a line segment and a circle, return the
```

intersection point. If don’t have intersection points, then return null.

ComputeIntersection(edge, viewPoint, range);

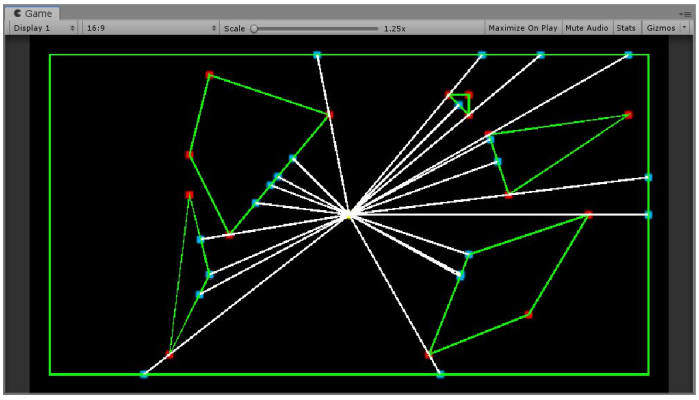
endfor

---

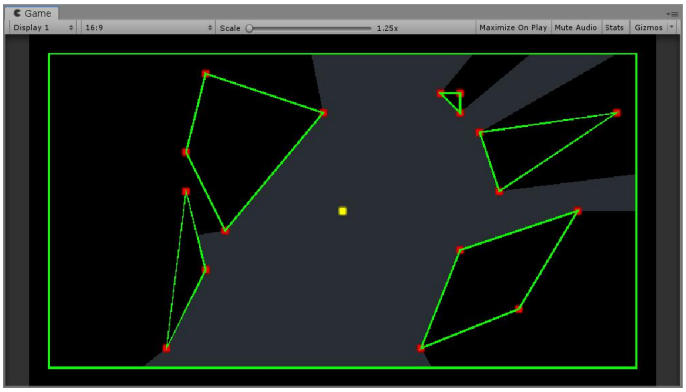
4. Test Case

4.1 Convex Polygon Obstacle

a)



The polygon surrounded by the red points are obstacle and the view point are denoted by yellow point. The blue points are the hit point.

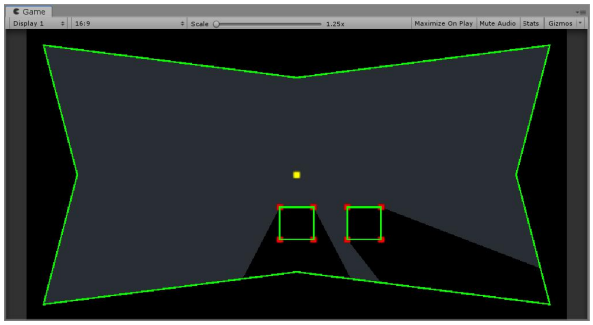


The gray area is the visibility effect of that view point without sight

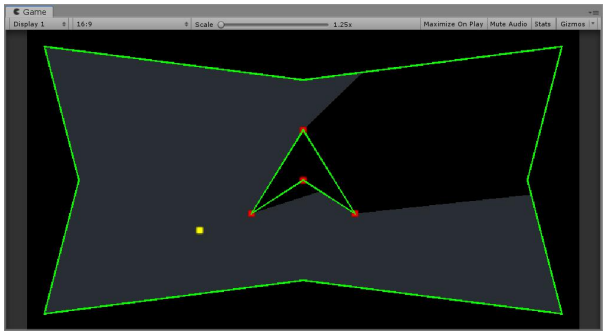
range or sight angle restrictions.

4.2 Concave Polygon

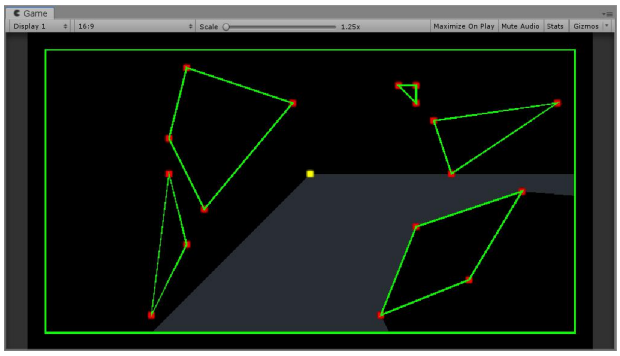
4.2.1 Concave Boundary



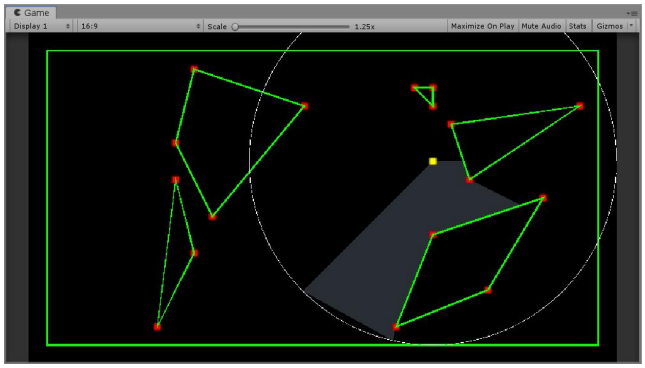
4.2.2 Concave Obstacle



4.3 Partially View



4.4 Visibility with Range Limitation



5. Conclusion and Impact

This project can be a very useful in real game design. For example, it can generate the smallest region for a guard to able look through all polygon. In addition, the basic attacking points and cover points in shooting game are also based on this visibility logic.

Cover point, obvious, can be placed where out of the visibility region. As for attacking spot, basically, it has both the property of cover point and some extra feature designed by the programmer. For example, the attacking spot need to switch the attacking mode and hiding mode easily. This varies from person to person. To sum, those are all the basic use of generate visibility polygon.

6. Reference

[1] In *Wikipedia, the free encyclopedia.* Retrieved May 16, 2018, from https://en.wikipedia.org/wiki/Concave (Accessed:18 March 2019)

[2] In *Wikipedia, the free encyclopedia.* Retrieved May 16, 2018,

from https://en.wikipedia.org/wiki/Convex (Accessed:18 March 2019)

[3] In *Wikipedia, the free encyclopedia.* Retrieved March 17, 2019, from https://en.wikipedia.org/wiki/Cross_product (Accessed:18 March 2019)

[4] In *Wikipedia, the free encyclopedia.* Retrieved October 20, 2018, from https://en.wikipedia.org/wiki/Visibility_polygon (Accessed: 18 March 2019)

[5] LAN_YT, (2014)*The triangulation in Unity.* https://www.cnblogs.com/lan-yt/p/9200621.html (Accessed:18 March 2019)

[6] John E. Hopcroft, & Gordon D. Plotkin, (1987). *Art Gallery Theorems and Algorithms.* NewYork, ON: Oxford University Press.

[7] Unity Documentation, (2018) https://docs.unity3d.com/ScriptReference/Physics2D.RaycastAll.html (Accessed: 18 March 2019)