
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.

Content

1. Introduction.....	3
1.1 Motivation summary.....	3
1.2 Scope of the report.....	4
2. Background and Related Work.....	4
2.1 Background.....	4
2.2 Related Work.....	6
3. Implementation.....	8
3.1 Generate obstacle and boundary.....	8
3.2 Generate Ray Cast.....	12
3.3 Generate Mesh.....	15
3.4 Implement the partially viewing.....	17
3.5 Implement range limit.....	17
4. Test Case.....	19
4.1 Convex Polygon Obstacle.....	19

4.2 Concave Polygon.....	20
4.3 Partially View.....	21
4.4 Visibility With Range Limitation.....	22
5. Conclusion and Impact.....	22
6. Reference.....	22

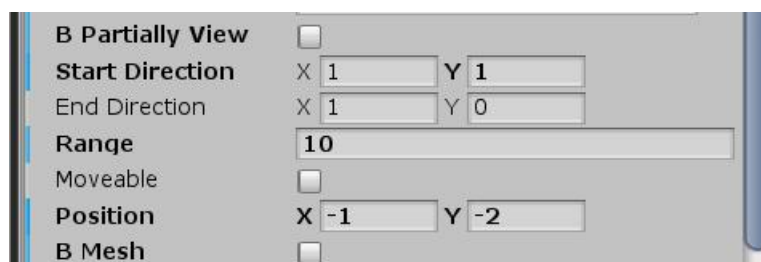
1. Introduction

1.1 Motivation summary

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

In addition to get the over all 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 screenshot shows a configuration panel for a game object named "InitialViewPoint". The panel has a sidebar on the left with a blue highlight on "B Partially View". The main area contains the following settings:

- B Partially View**: ☐
- Start Direction**: X Y
- End Direction**: X Y
- Range**:
- Moveable**: ☐
- Position**: X Y
- B Mesh**: ☐

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 determine 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 choose no, it will only generate the debug lines cast.

1.2 Scope of the report

The effect is implemented on the Unity 3.19 Platform.

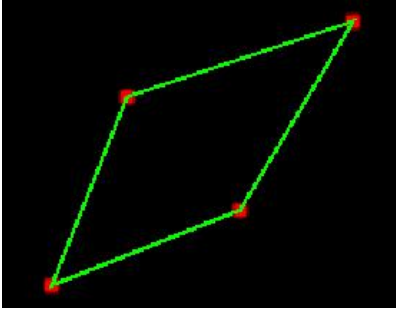
2. Background and Related Work

2.1 Background

2.1.1 Convex Polygon

(“Convex”, 2018)

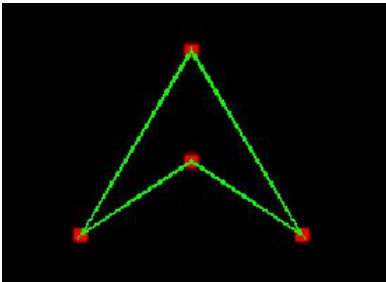
A polygon in which no line segment between two points on the boundary ever goes outside the polygon.



2.1.2 Concave Polygon

("Concave", 2018)

A polygon which is not convex.



2.1.3 Cross Product

("Cross Product", 2019)

The cross product of two vectors **a** and **b** is defined as $\mathbf{a}.x * \mathbf{b}.y - \mathbf{a}.y * \mathbf{b}.x$.

If the result is smaller than 0, which means the unsigned angle between **a** and **b** is clockwise from **a** to **b**.

For example,

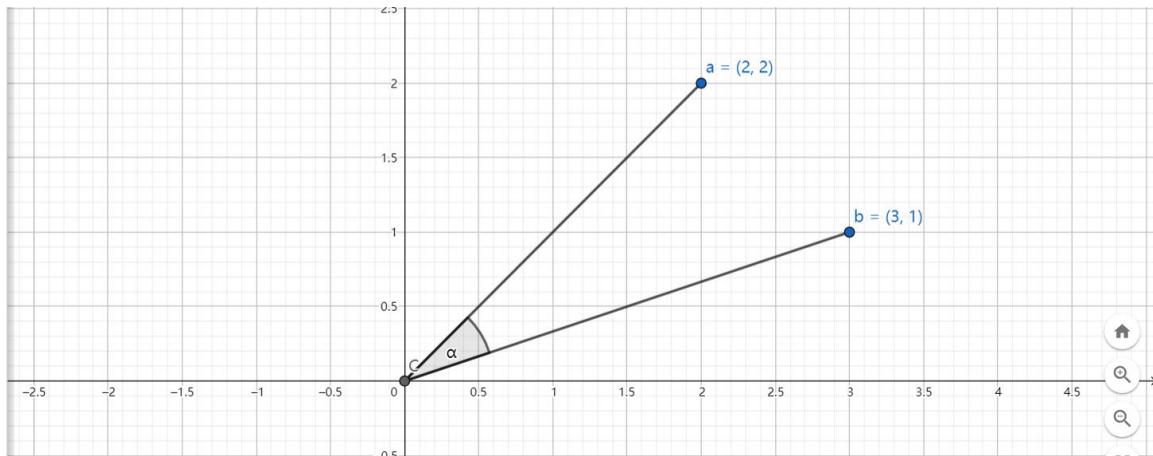
$$\mathbf{a} = (2, 2), \mathbf{b} = (3, 1)$$

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

$\text{crossProduct}(\mathbf{a}, \mathbf{b}) = 2 - 6 < 0$, which means **a** gets angle α by go clockwise to **b**.

$\text{crossProduct}(\mathbf{b}, \mathbf{a}) = 6 - 2 > 0$, which means **b** gets angle α by go

anticlockwise to **a**.

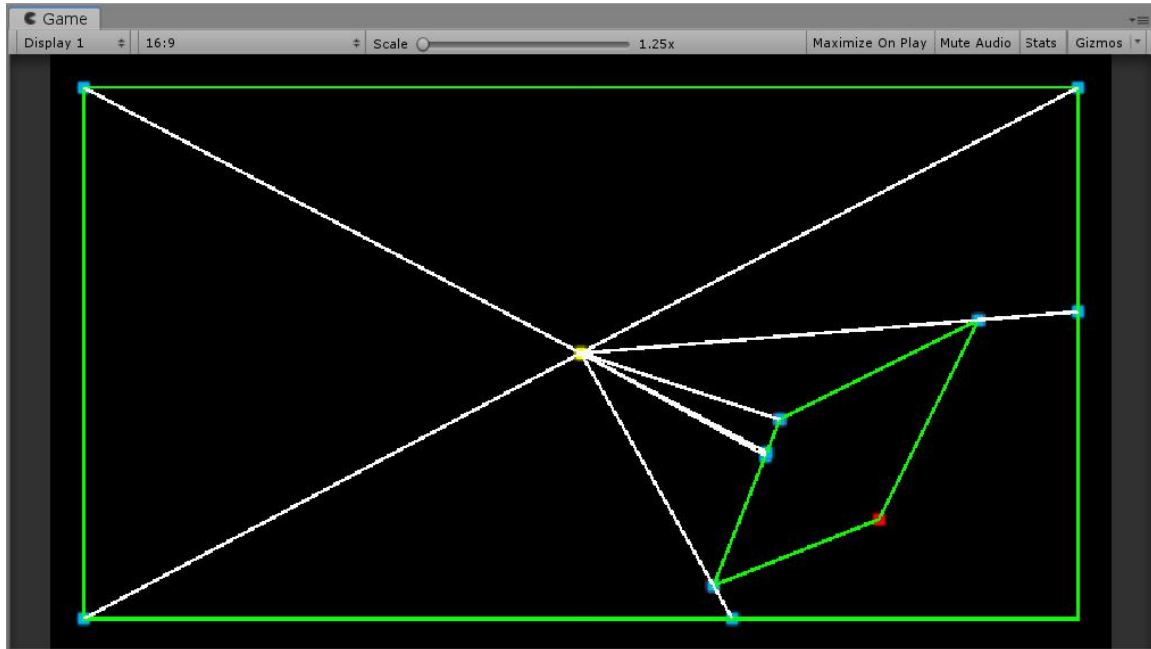


2.2 Related Work

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.



2.1.4 Sight Range

In the real world, it is 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.

2.1.5 Sight Angle

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

2.1.6 View Point

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

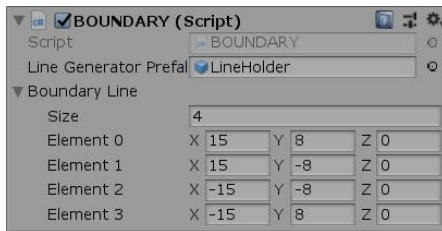
3. Implementation

The realization is based on Art Gallery Algorithm(*"Art Gallery Theorems and Algorithms"*, 1987).

3.1 Generate obstacle and boundary

3.1.1 Boundary

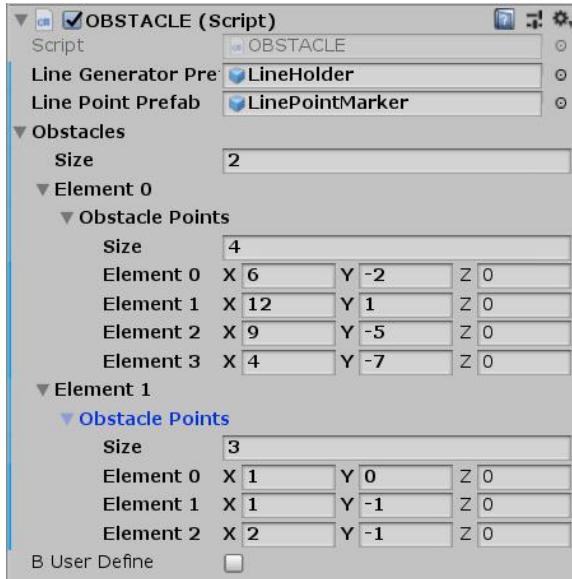
The Boundary Manager receive a list of Vector2D points denoted the vertex of the boundary,



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 obstacles. 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

```
bool isClockWise(Vector2 a, Vector2 b, Vector2 c) {
    if ((a.x - c.x) * (b.y - c.y) - (b.x - c.x) * (a.y - c.y) < 0){
        return true;
    } else {
        return false;
    }
}
```

```
}  
}
```

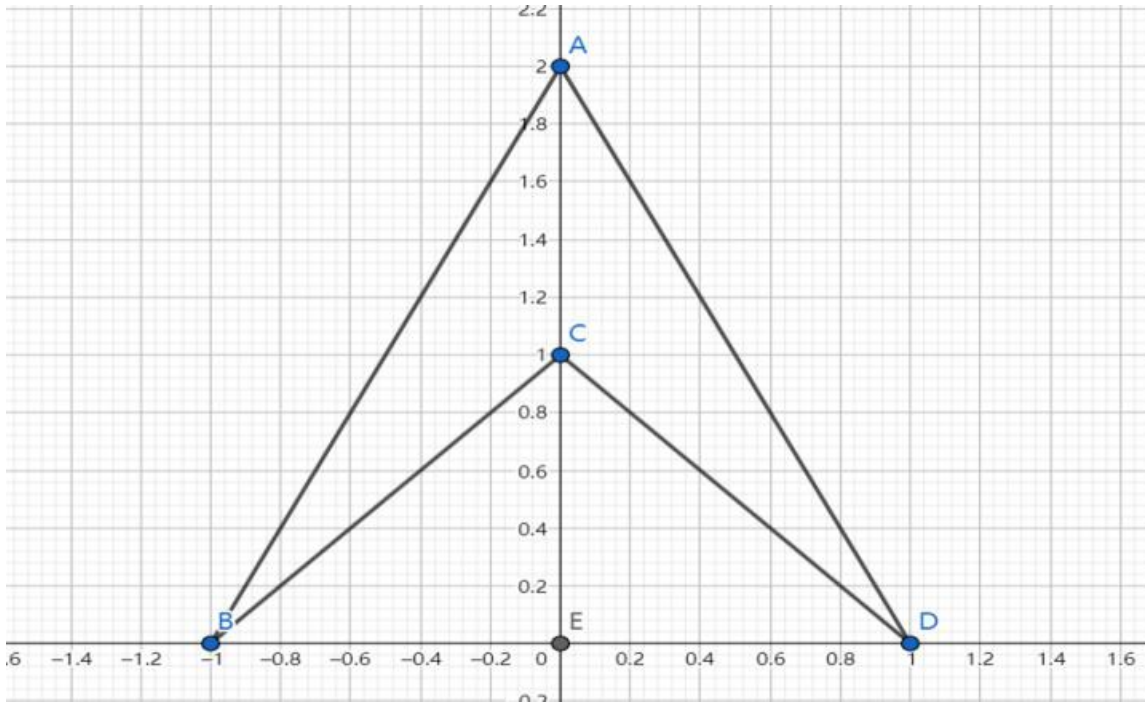
3.1.3.2 Determine whether it is a concave polygon or not.

bool flag; // Randomly choose two adjacent edge, determine whether it is clockwise or not.

```
foreach (AdjacentEdge e in polygon) {  
    // all the adjacent edge in convex polygon must have a consecutive  
    result  
    if (e.isClockwise() == flag){ Continue;}  
    else { return false;}  
}
```

3.1.3.3 Find the split-able point

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



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

```
/// <summary>
```

```
/// Detect whether the point can split or not:
```

```
/// Detect whether new polygon have vertices in the splited triangle
```

```
/// </summary>
```

```
bool IsSplitIndex(int index, List<Vector3> verts) {
```

```
    List<Vector3> triangleVert = new List<Vector3>();
```

```
    triangleVert.Add(verts[prev]);
```

```
    triangleVert.Add(verts[index]);
```

```
    triangleVert.Add(verts[next]);
```

```
foreach(Vertex v in verts) {  
    // if the point is inside the polygon  
    if (IsPointInsidePolygon(verts[i], triangleVert)){  
        return false;  
    }  
}  
}
```

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 Lines Cast

```
Generate Lines Cast()
{
    foreach vertex in the obstacle {
        Generate Line Cast();
    }
}
```

3.2.1.1 Generate Line Cast

```
GenerateLineCast(
Vector viewpoint, // view point position
OBSTACLE endPoint, // an obstacle
Vector2 direction, // line direction
int endPointIndex // indicate which point in 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 == endPoint[endPointIndex ]) {
            // If the neighbour endpoints of the hitting result are both
            in the one side, keep the hitting result
            if (isSameSide()) {
                hit.add(rayCastHitResult )
                continue;
            }
        }
    }
}
```

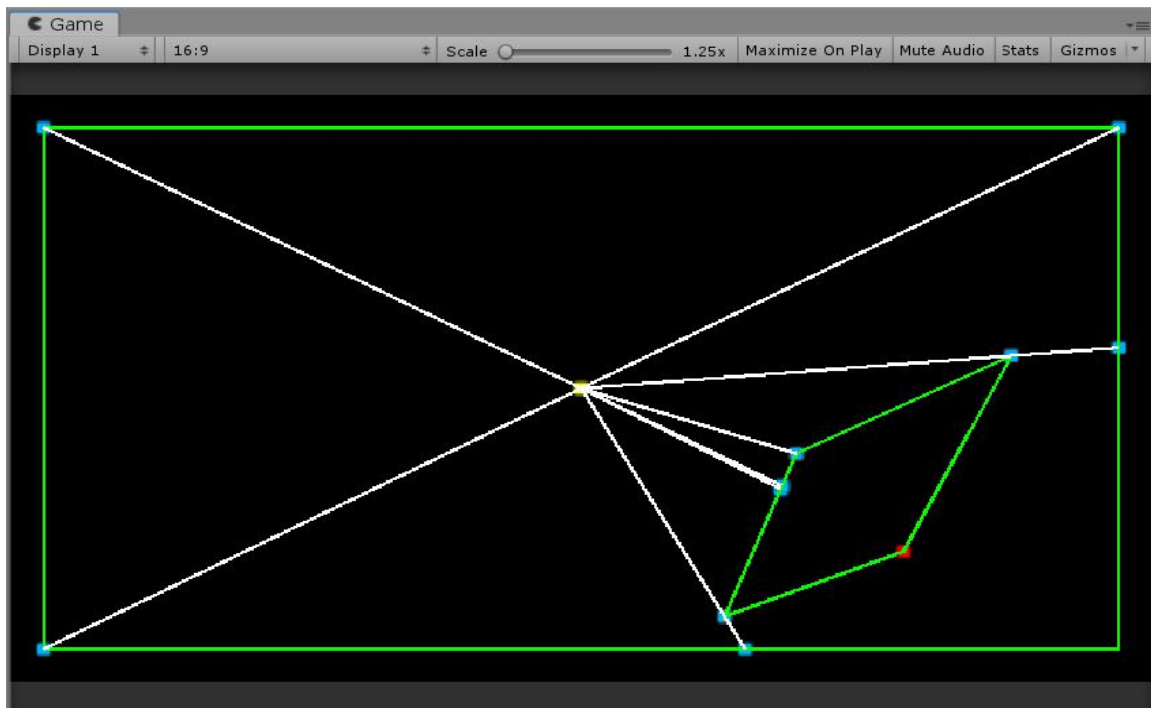
```

        else {
            hit.add(rayCastHitResult );
            break;
        }
    }
    else {
        hit.add(rayCastHitResult );
        break;
    }
}

```

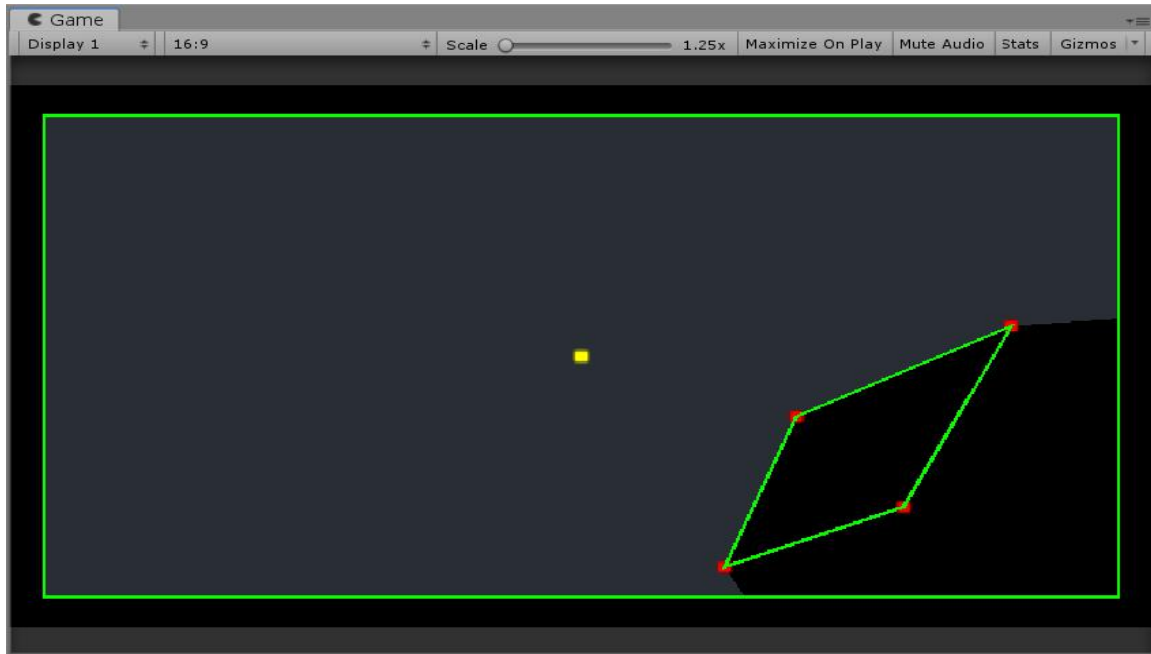
3.2.2 Movable view point

We want to have 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.

```

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;
    }
    // List with index from "cur" to "end" are all in the same line
    if (end > cur) {
        Vector2 preNode = list[pre].location;
        Vector2 nextNode = list[next].location;

        if (!isInSameObstaclesLine(list[end].location, nextNode
|| !isInSameObstaclesLine(list[cur].location, preNode) {
            // swap the order
            swapOrder(list, cur, end);
        }
    }
    cur = end + 1;
    pre = end;
    next = cur + 1;
    next %= list.Count

```


3.4 Implement the partially viewing

3.4.1 Exclude the line cast

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

```
public static bool isInsideClockRangeOfTwoVector(Vector2 start, Vector2
end, Vector2 test)
{
    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);
    // angle1 represent the angle from start to test in clockwise order
    float angle2 = Vector2.Angle(test, end);
    // angle1 represent the angle from start to test in clockwise order
    float angle3 = Vector2.Angle(start, end);
    return float1 + float2 == float3;
}
```

3.5 Implement range limit

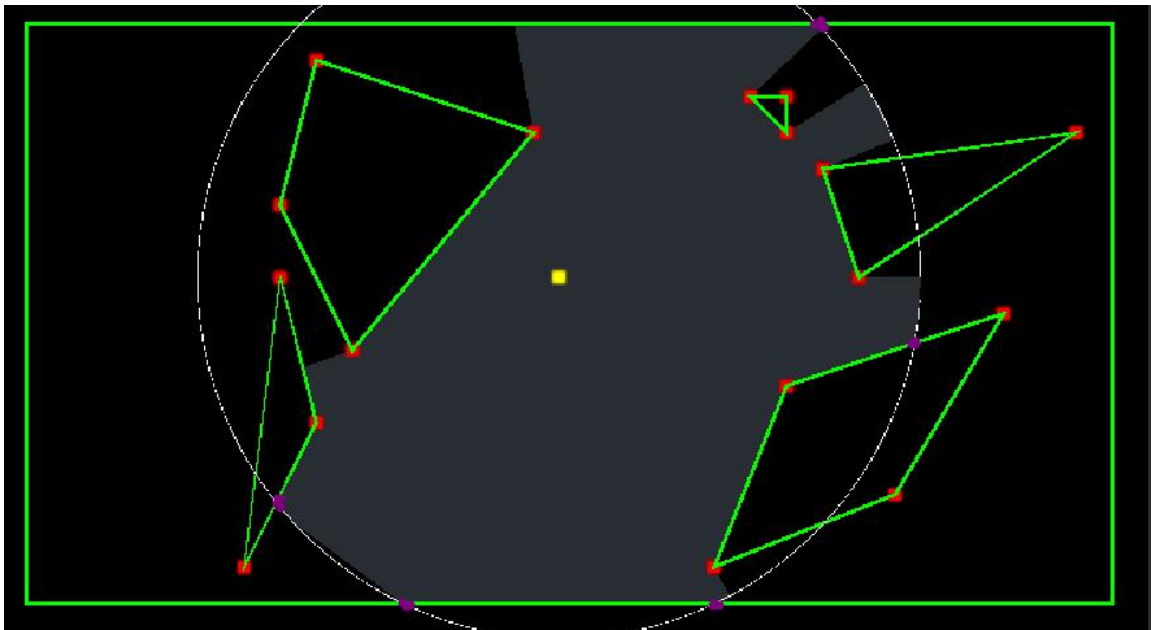
3.5.1 Truncation the ray cast

Now we add a 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.

```
// If the hit point is out of the range
if (hitpoint.magnitude > range)
{
    // truncate the ray cast
    hitPoint.location = viewpoint + direction.normalized * range;
    addPointTohitList(hitPoint);
}
```

3.5.2 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.

```
foreach (Obstacle obstacle in Obstacles) {
```

```

GenerateIntersectionPoint(obstacle, viewPoint, range);
}

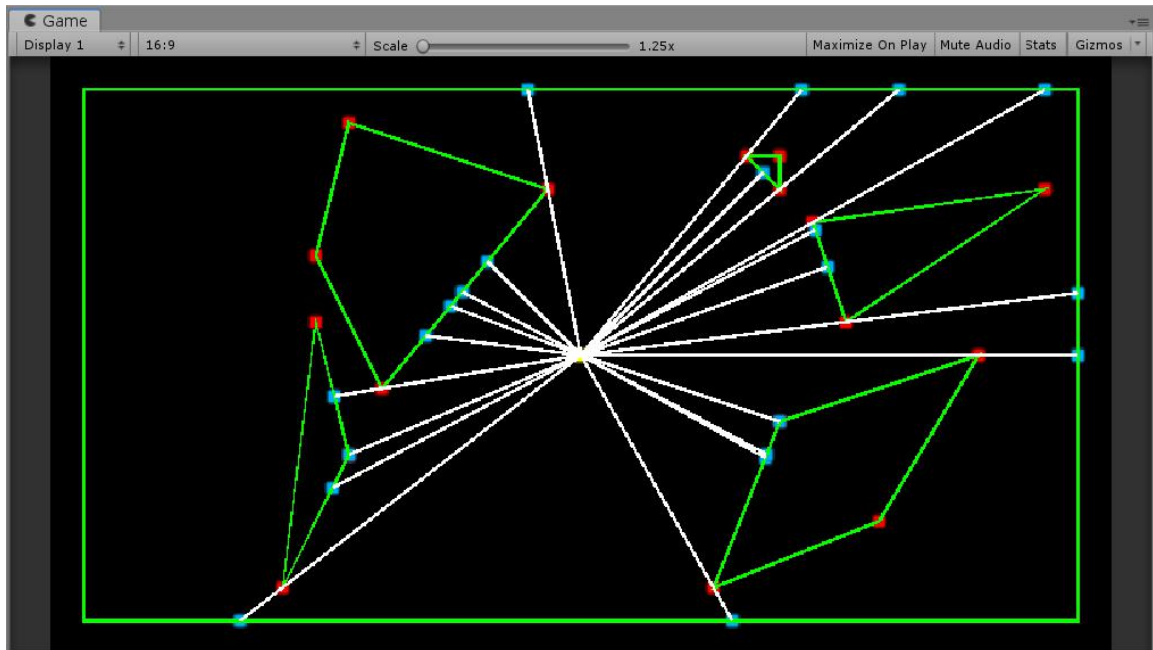
GenerateIntersectionPoint() {
    foreach (Vector2 edge in obstacle) {
        // This method is get a line segment and a circle, return the
        // intersection point. If don't have intersection points, then return null.
        ComputeIntersection(edge, viewPoint, range);
    }
}

```

4. Test Case

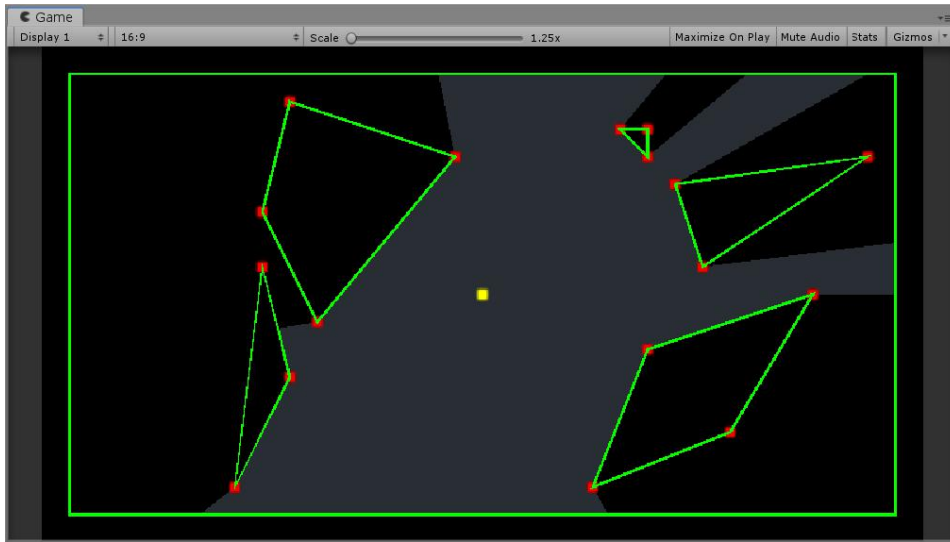
4.1 Convex Polygon Obstacle

a)



The polygon surrounded by the red points are obstacle and the view

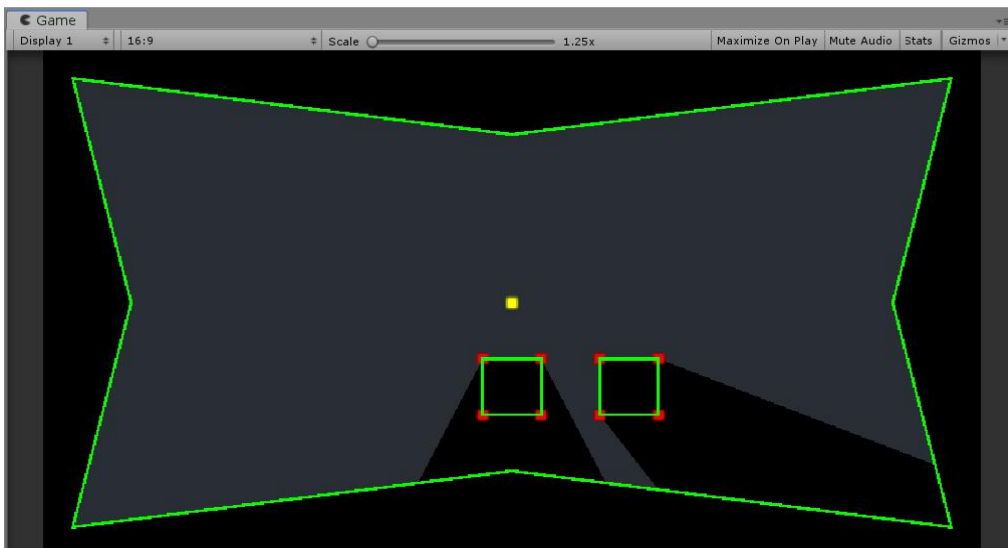
point is denoted by yellow point. The blue points are the hit point.



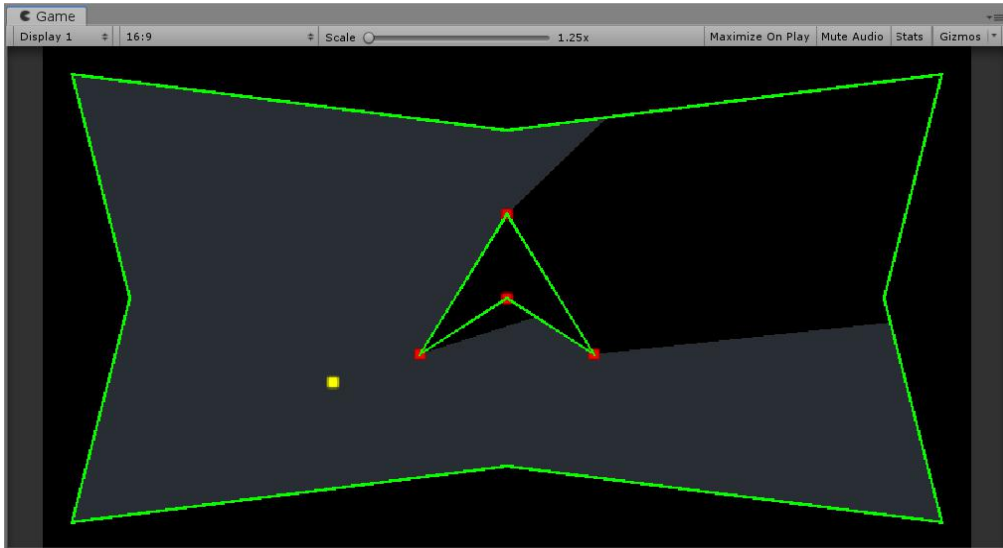
The gray area is the visibility effect of that view point without sight range or sight angle restrictions.

4.2 Concave Polygon

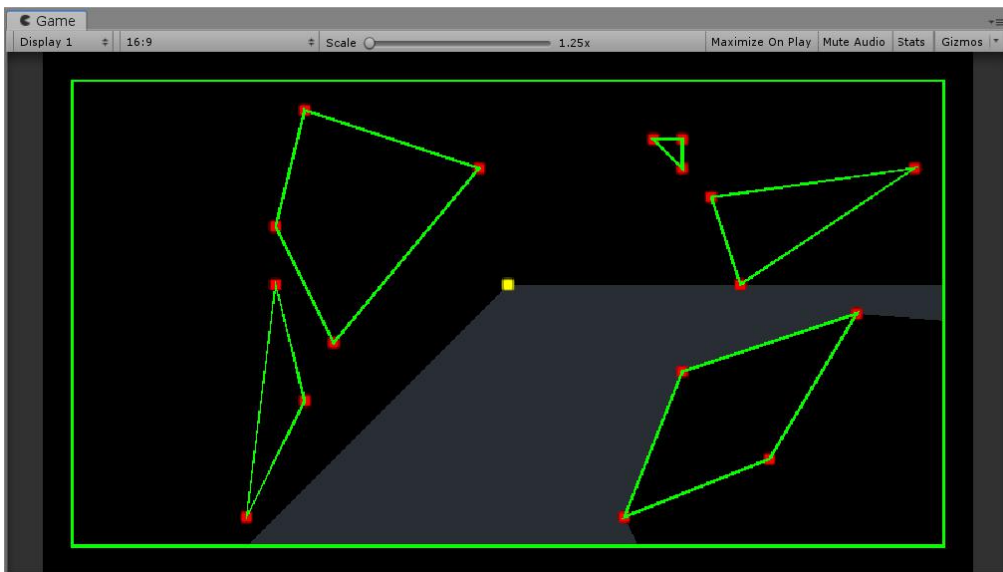
4.2.1 Concave Boundary



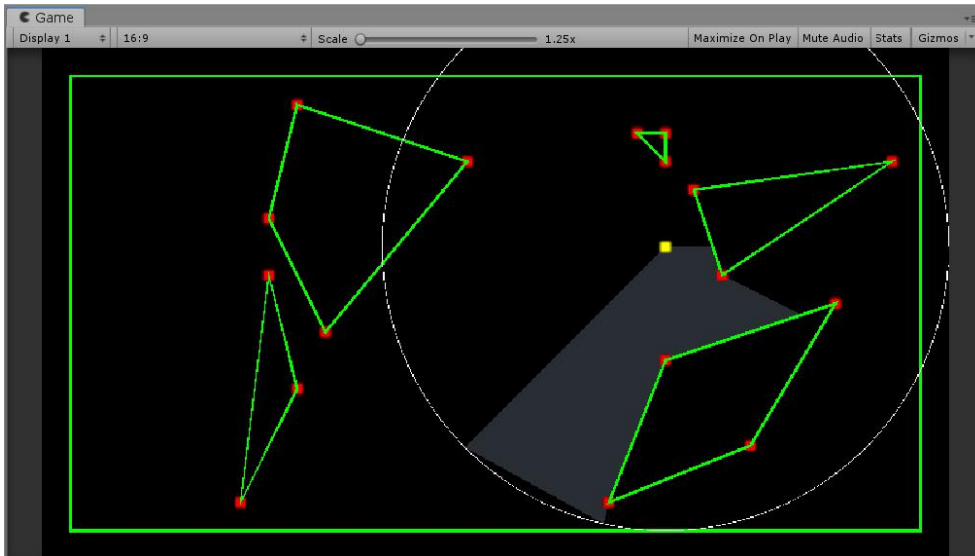
4.2.2 Concave Obstacle



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

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

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