Grid Formation Behaviour v1.0

Mathematics and Algorithms

Jonas Munoz Novelo

Contents

1 Introduction

This document goes step by step into implementing a grid formation behvaiour by clicking and dragging from point A to point B. First implementing it in the X axis only, and then being able to rotate the grid formation in any angle using the Unit circle. Focusing mostly on the theory and mathematics of the grid formation behaviour. Actual implementations will be discussed in another paper.

The goal is to show how mathematics can be utilised to create a system that is robust and easy to add/make changes without breaking everything. The outcome of this paper lays the foundation for future iterations and additions. Understanding the mathematics will allow one to implement the behaviour in any platform that support cartesian coordinates.

2 Dragging

Creating a function that detects when the cursor is being dragged across the screen.

```
DraggingMechanism:
{
    bool dragging = false;
    MouseButton button = RIGHT;

    void input() {
        if ButtonDown == button {
            dragging = true;
        }
        if ButtonReleased == button {
            dragging = false
        }
    }

}
```

3 Imaginary Line

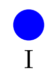
As units will be placed along a line, we can create and imaginary line from the moment the drag begins until the moment the drag stops.

Modifying the Dragging Mechanism from before

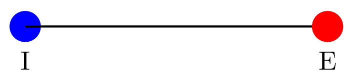
```
DraggingMechanism_V2:
{
    bool dragging = false;
    MouseButton button = RIGHT;
    Vector initCoordinate = Vector(0,0);
    Vector endCoordinate  = Vector(0,0);

    void input() {
        if ButtonDown == button {
            dragging = true;
            initCoordinate = mouseCoordinate();
        }
        if ButtonReleased == button {
            dragging = false;
            endCoordinate = mouseCoordinate();
        }
    }
}
```

3.0.1 Example

The mouse begins to drag at point I



The mouse is being dragged and stops on point E, creating an imaginary line from point I to point E.



3.1 Length of Imaginary Line

The length of the imaginary line will dictate the number of units that are able to be placed on the line.

Calculating the length of the line can be done in two ways:

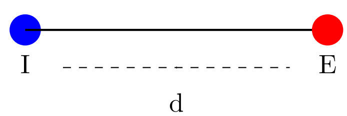


Figure 1: $d = E - I$

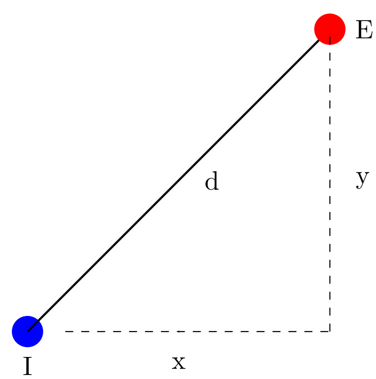


Figure 2: Pythagora's theorem

Using Pythagora's theorem allows one to calculate the length.

$$d^2 = x^2 + y^2$$
$$d = \sqrt{x^2 + y^2}$$

The approach in *Figure 1* only considers a line with no `y` value, making it very limited for our purposes later on. Because of this, the Pythagora's theorem approach will be used.

3.2 Placing Units on the Line

3.2.1 Number of Units on the Line

Calculating the number of units that can fit on a line requires knowing the *size* of the units that are going to be in formation. For the sake of this document,: $sizeOfUnit = 1$.

Given the size of the unit, the total number of units that lie on a line is as follows:

$$totalUnits = \frac{lengthOfLine}{sizeOfUnit}$$

This assumes that for every radius of integer value ¿= 1, a unit is placed, but this behaviour places the first unit on the initial coordinate. Resulting in *1* unit extra.

$$totalUnits = \frac{lengthOfLine}{sizeOfUnit} + 1$$

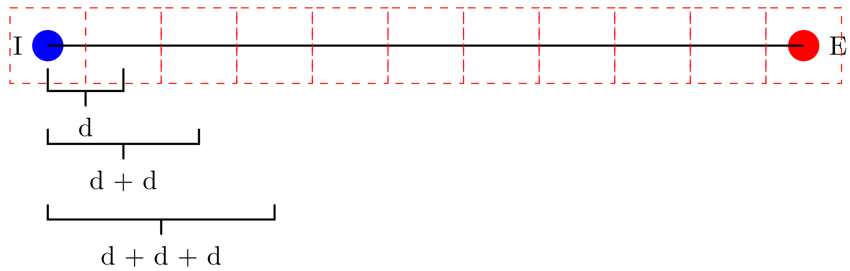
3.2.2 Positions of units on the line

To get the positions of each units on the line requires the following:

1. Initial Coordinate
2. End Coordinate
3. Size of unit
4. Total number of units that lie on the Line
5. Distance between each unit

The first unit will spawn at initial point I. Given that units can be d distance away from each other, the next unit will be located at $I + d$. The third unit will be located at $I + d + d$. The Fourth unit will be located at $I + d + d + d$ and so on.
This can be represented in the following diagram:

$$positionOnLine = I + (d * i)$$



3.2.3 Manipulating the Distance between each Unit

Changing the distance between each unit affects the total number of units that lie on the line.
This means that the distance between each unit must be taken into account when calculating the total numbers of units that lie on the line. The current equation calculates the number of units on a line with no gaps. To add gaps, the size of the unit + another value (g) must be taken into account. Giving the new equation:

$$totalUnits = \frac{lengthOfLine}{(sizeOfUnit + gap)} + 1$$

As *sizeOfUnit + gap* is the distance between each unit, this can be considered as: $distanceBetweenUnit = sizeOfUnit + gap$. Allowing for the following equation:

$$totalUnits = \frac{lengthOfLine}{distanceBetweenUnit} + 1$$

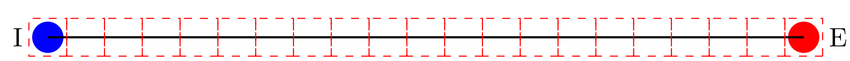
Using this equation and the previous example, the gap was *0*.

3.2.4 Increasing/Decreasing Distance between Units

Changing the size of the unit as well as the distance between each unit affects the total number of units on the line. Below are some scenarios that help visualize what is going on, and may help in fine tuning the variables to meet certain needs.

Secnario 1: Reducing Size of Unit Reducing the size by $\frac{1}{2}$ will give the total units to be 21.

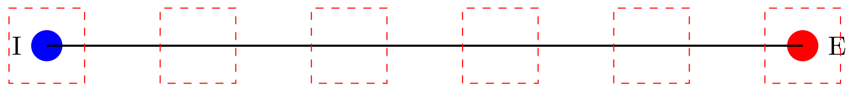
$$21 = \frac{10}{0.5 + 0} + 1$$



Allowing for more units to fit on the line. The inverse effect applies when increasing the size of the unit: **Greater Size, Less Units**.

Sencario 2: Adding a Gap Having a size of 1, and a gap of 1 increases the distance between each unit. While at the same time decreases the number of units on the line.

$$6 = \frac{10}{1+1} + 1$$



To calculate the position on the line:

```
List<Vector> PositionsOnLine(float distBtxt,
                             int totalUnits,
                             Vector I)
{
    List<Vector> positions = empty;

    for(int currentUnit = 0; currentUnit < totalUnits; currentUnit++) {
        float offsetOfCurrentUnit = distBtxt * currentUnit;
        Vector positionOfCurrentUnit = I + offsetOfCurrentUnit;
        positions.add(possitionOfCurrentUnit);
    }

    return positions;
}
```

The first position on the line will be located on index *0* in `positions` list.

3.3 Integrating

For the moment, the calculation of positions occurs after dragging has ended.

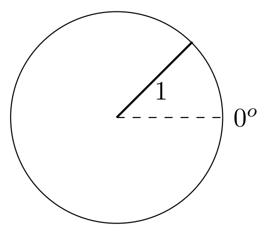
```
DraggingMechanism_V3:
{
    bool dragging = false;
    MouseButton button = RIGHT;
    Vector initCoordinate = Vector(0,0);
    Vector endCoordinate = Vector(0,0);

    float gap = g;
    float sizeOfUnit = s;
    List<Vector> positions = empty;

    void input() {
        if ButtonDown == button {
            dragging = true;
        initCoordinate = mouseCoordinate();
        }
        if ButtonReleased == button {
            dragging = false;
            endCoordinate = mouseCoordinate();

            float lengthOfLine = Pyhtagora(initCoordinate, endCoordinate);
            float distBtxt = sizeOfUnit + gap;

            // unit on I is counted, thus the '+1'
            int totalUnits = (lengthOfLine / distBtxt) + 1

            positions = PositionsOnLine(distBtxt, totalUnits, initCoordinate);
        }
    }
}
```

4 Applying Unit Circle

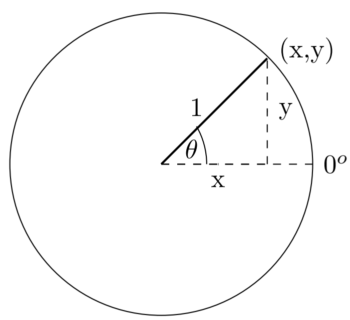
The current implementation only places units along the X-Axis. To overcome this limitation, the Unit circle will be used as a guide to place units along both the X-Axis and the Y-Axis. Having the angles be aligned onto the X-axis.

4.1 Quick summary on the Unit Circle

The unit circle consists of a circle with *radius* of `1`.

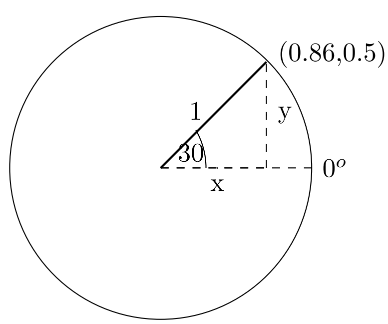


The coordinates from the center to the end of the radius can ba calculated using Sin and Cos:



$$\cos(\theta) = \text{x}$$
$$\sin(\theta) = \text{y}$$

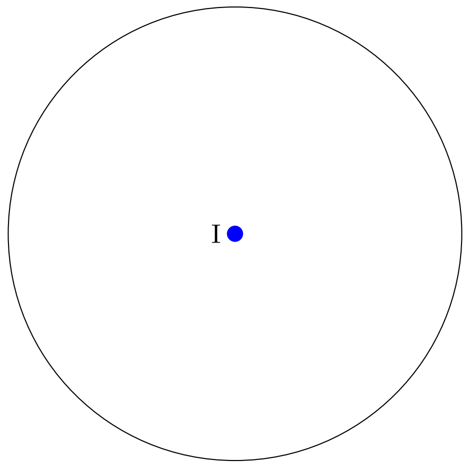
substituting the values, the coordinates are as follow:



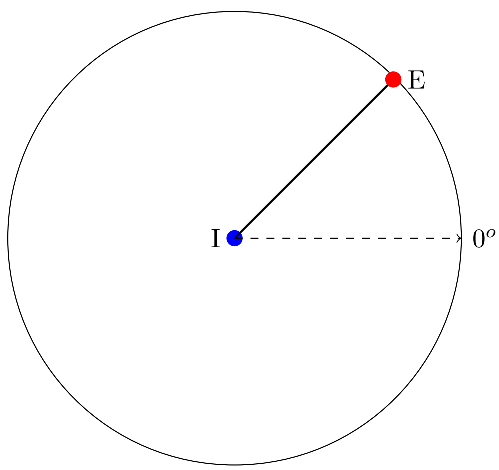
$$\cos(30) = 0.86$$
$$\sin(30) = 0.5$$

4.2 Applying Imaginary Line to Unit Circle

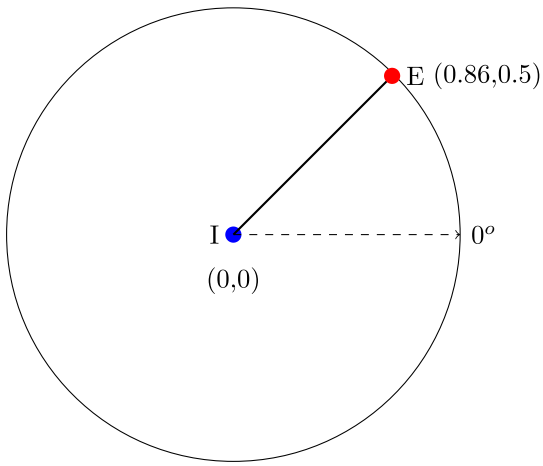
Imagine the center of the circle is the Initial coordinate I.



And E is located on the circumference at the end of the radius. This creates the imaginary line as before, with angles being aligned on the X-Axis.



For the sake of the example $I = (0, 0)$ and $E = (0.86, 0.5)$, as I and E will be known upon finishising the dragging mechanism.

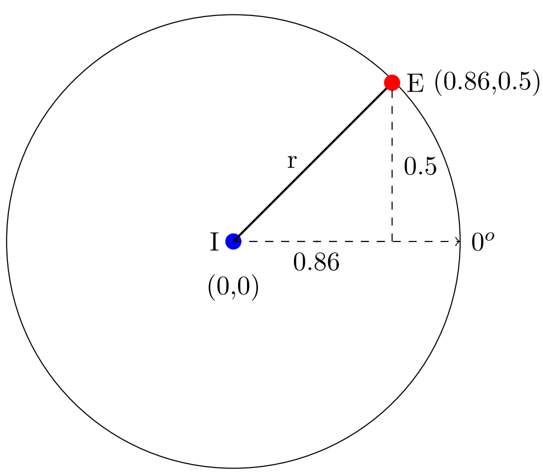


;

What is left is to calculate the length of the radius

4.2.1 Length of Radius/Line

Using Pythagora's theorem calculates the length of the given radius, r:



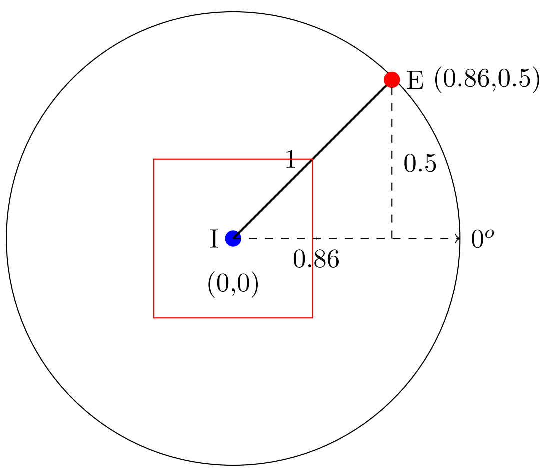
$$r = \sqrt{0.86^2 + 0.5^2}$$
$$r = 1$$

A radius of length 1 allows for the following number of units to lie on the radius with a unit size of 1 and no gaps.

$$
\begin{aligned}
totalUnits &= \frac{lengthOfRadius}{sizeOfUnit + gap} + 1 \\
&= \frac{1}{1+0} + 1 \\
&= \frac{1}{1} + 1 \\
&= 1 + 1 \\
totalUnits &= 2
\end{aligned}
$$

4.2.2 Positions on Radius/Line

The position of the first unit will be at I.



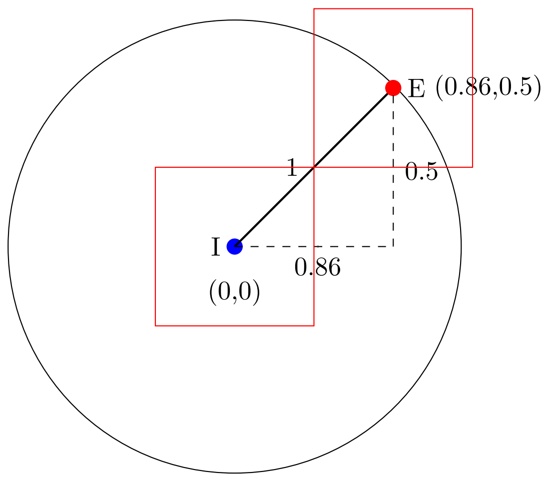
To calculate the position of the *next* unit requires finding the angle.

Acquiring Angle Using an equation derived from SOHCATOAH.

$$
\begin{aligned}
\tan(\theta) &= \frac{opposite}{adjacent} \\
&= \frac{0.5}{0.86} \\
\tan(\theta) &= 0.58 \\
\theta &= \tan^{-1}(0.58) \\
\theta &= 30
\end{aligned}
$$

Applying the Sin/Cos will give the position of the unit on the radius/line at an angle of *30*, with unit size = 1 and no gap.

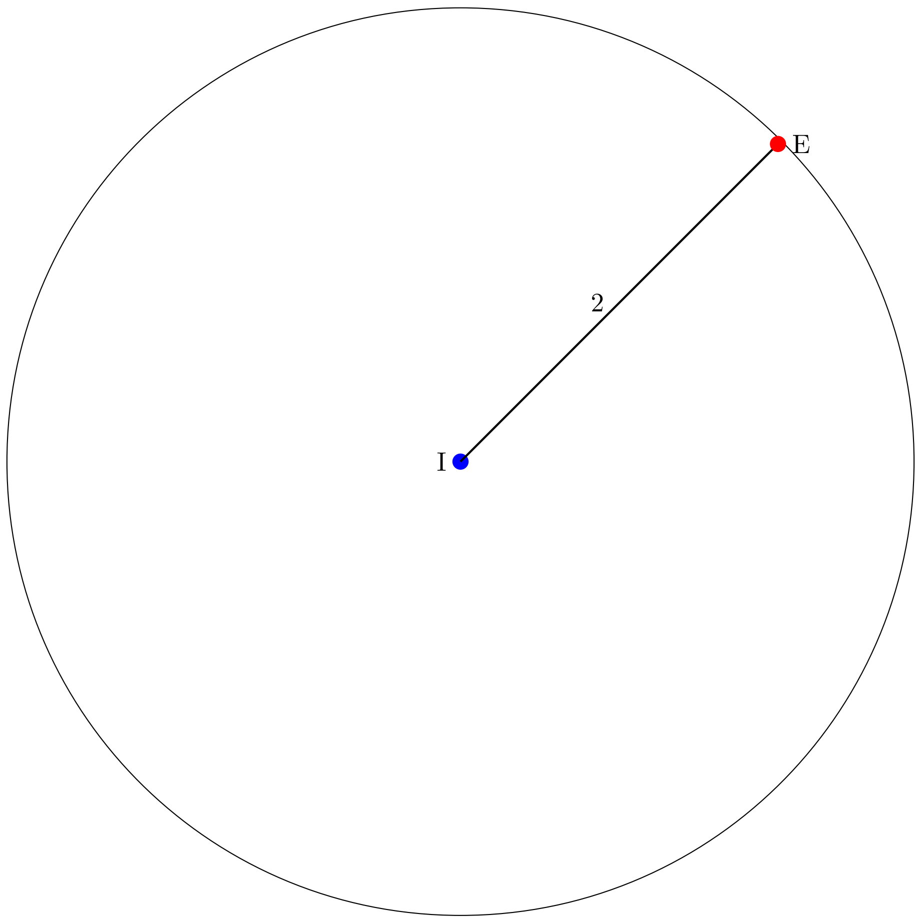
$$0.86 = \cos(30);$$
$$0.5 = \sin(30);$$



REMEMBER The distance between each unit starts from the center of one unit to the center of the next unit. **Notice** how the distance between each unit is the same as the radius, r.

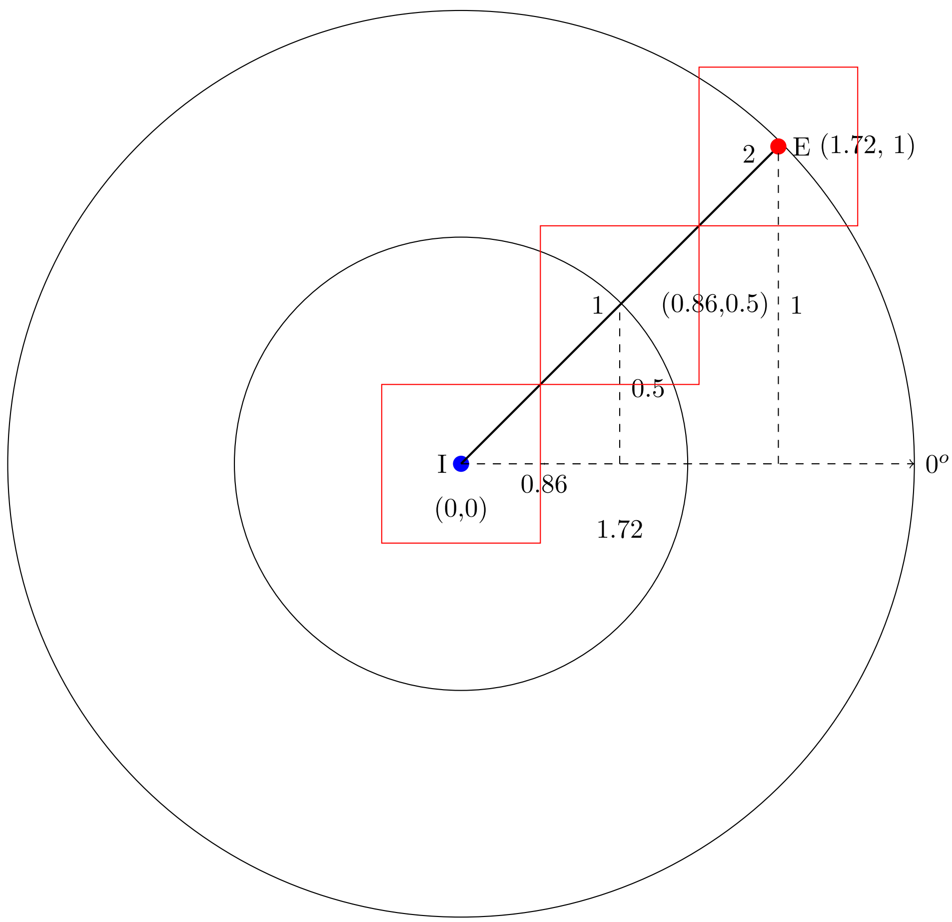
4.2.3 Adding more units

Increasing the radius to 2 will allow to place more units on the line.

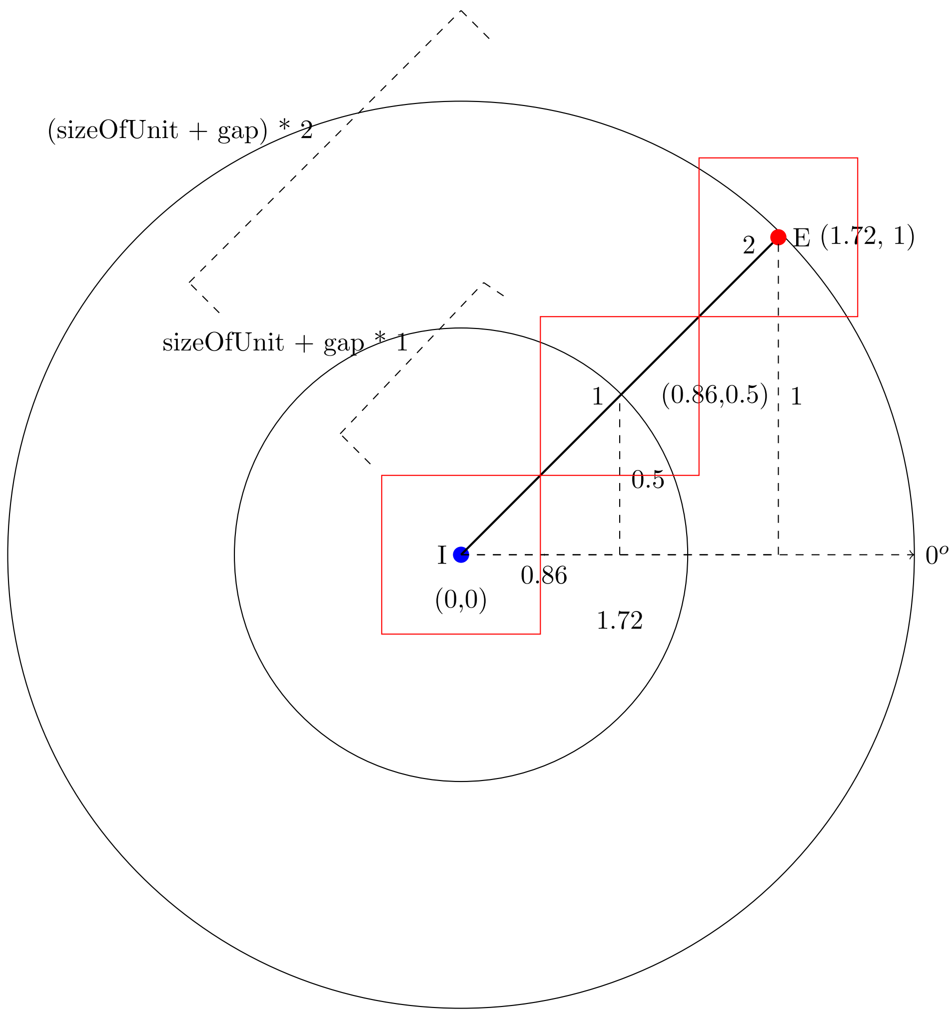


Based on the equation, there should be three units.

Once can imagine as having two circles of same center I and same degrees but of varying radii (In this case, 1 and 2).



Allowing for three units of size 1 with no gaps to lie in the radius.



Based on the imagine above, one can derive the following equation:

$$positionOnRadius = I + ((sizeOfUnit + gap) * (cos(\theta), sin(\theta)) * i)$$

where:

sizeOfUnit + gap is the Radius of the circle *Or* the distance between units

(cos(θ), sin(θ)) aligns the radius by a degree of θ

i is the current unit *Or* the *nth* circle, starting from *0*.

Using the above example, the following coordinates on the line are:

`First Unit`

$$
\begin{aligned}
positionOnRadius &= I + ((1+0) * (cos(30), sin(30)) * 0) \\
&= I + (1 * (0.86, 0.5) * 0) \\
&= I
\end{aligned}
$$

`Second Unit`

$$
\begin{aligned}
positionOnRadius &= I + ((1+0) * (cos(30), sin(30)) * 1) \\
&= I + (1 * (0.86, 0.5) * 1) \\
&= I + ((0.86, 0.5) * 1) \\
&= I + (0.86, 0.5)
\end{aligned}
$$

`Third Unit`

$$
\begin{aligned}
positionOnRadius &= I + ((1+0) * (cos(30), sin(30)) * 2) \\
&= I + (1 * (0.86, 0.5) * 2) \\
&= I + ((0.86, 0.5) * 2) \\
&= I + (1.72, 1.0)
\end{aligned}
$$

4.2.4 Deriving the function

Caulculating the position on the radius requires the following variables:

$$CalculatePositionOnRadius(I, E, sizeOfUnit, gap)$$

where:

I and E give the *Radius* of the circle.

sizeOfUnit and gap give the distance between each unit.

NOTE subtracting $E - I$ give the x and y or *adjacent* and *opposite* values respectively. Allowing one to calculate the angle of the radius.

4.2.5 Algorithm

```
List<Vector> PositionsOnRadius(Vector I,
                               Vector E,
                               float sizeOfUnit,
                               float gap)
{
    float radius = Pythagora(I, E);
    int totalUnits = (radius/ (sizeOfUnit + gap)) + 1;
    float angle = arctan(E-I);

    List<Vector> positions = empty;

    for(int currentUnit = 0; currentUnit < totalUnits; currentUnit++) {
        Vector currentPosition = positionOnRadius(I,
                                                  sizeOfUnit,
                                                  gap,
                                                  angle,
                                                  currentUnit
                                 );
        positions.add(currentPosition);
    }
    return positions;
}
```

4.3 Integrating

The following algorithm *DraggingMechanism* is integrated with *PositionsOnRadius*:

```
DraggingMechanism_V4:
{
    bool dragging = true;
    MouseButton button = RIGHT;

    Vector initCoordinate = Vector(0,0);
    Vector endCoordinate = Vector(0,0);

    float gap = g;
    float sizeOfUnit = s;
    List<Vector> positions = empty;

    void input() {
        if ButtonDown == button {
            dragging = true;
            initCoordinate = mouseCoordinate();
        }
            if ButtonReleased == button {
            dragging = false;
            endCoordinate = mouseCoordinate;

            positions = PositionsOnRadius(initCoordinate,
                                          endCoordinate,
                                          sizeOfUnit,
                                          gap
                                         );
        }
    }
}
```

5 Dynamic Positions

Currently the positions are calculated after dragging ends. What if the positions can be calculated while dragging?

To achieve this behaviour, the calculation of positions must be actievely done *during* dragging.

5.1 Re-arranging Calls

Certain changes must be done in **Dragging Mechanism**. A new Procedure `Grid Formation` will be in charge of initializing the varaibles.

Have *Dragging Mechanism* only return whether dragging is occuring or not.

5.1.1 Dragging Mechanism

```
DraggingMechanism_V5:
{
    bool dragging_mechanism(Vector I, Vector E, MouseButton button) {
        if ButtonPressed == button {
            dragging = true;
            I = mouseCoordinate();
        }
        if ButtonReleased == button {
            dragging = false;
            E = mouseCoordinate();
        }
        return dragging;
    }
}
```

5.1.2 Grid Formation

A new coordinate C will store the current coordinate that the mouse is located at

```
GridFormation:
{
bool dragging = false;
MouseButton button = RIGHT;

Vector I = Vector(0,0);
Vector E = Vector(0,0);
Vector C = Vector(0,0);

float sizeOfUnit = s;
float gap = g;
List<Vector> positions = empty;

loop {
    dragging = Dragging(I, E, button);

    if dragging {
        C = mousePosition();
        Vector position = PositionsOnRadius(I, C, sizeOfUnit, gap);
    }
}
}
```

6 Fixed Number of Units

Currently the number of units on the line is dependent on the size of the line, and not on a fixed value. If one wants to display a fixed number of units, regardless of the size of the radius, then the following changes must be made in *PositionsOnRadius*

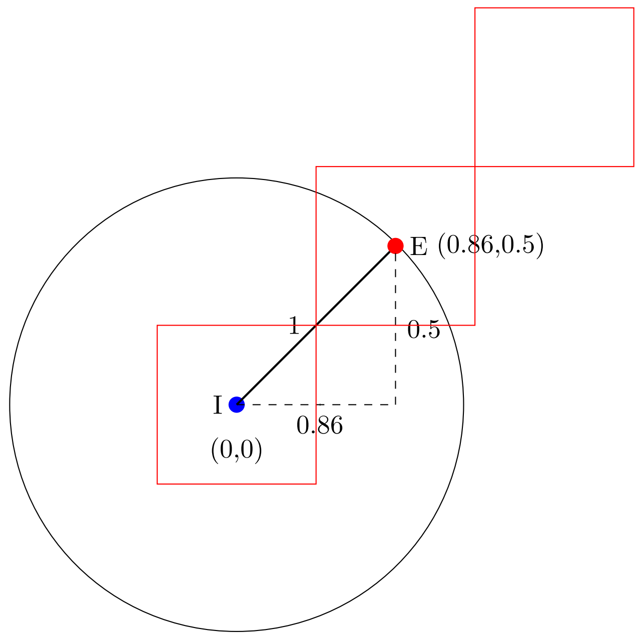
6.1 Adding limit to Number of Units

The *PositionsOnRadius* will be given a limit to the number of position to calculate based on the size of the Radius.

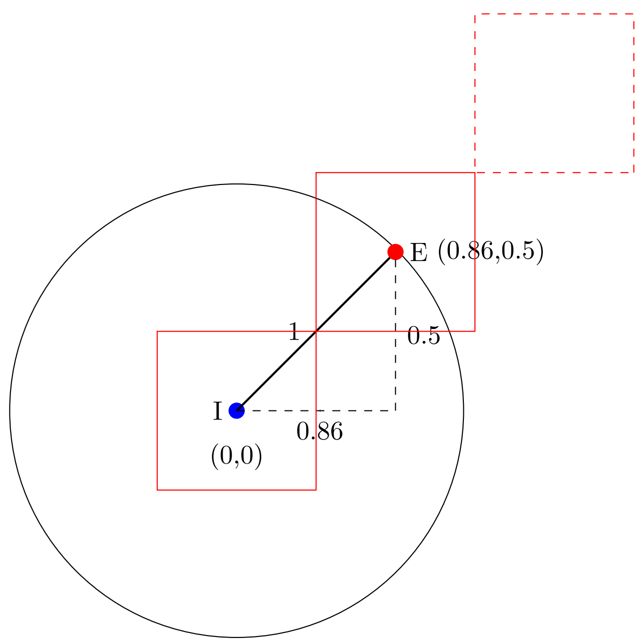
```
List<Vector> PositionsOnRadius(Vector I,
                               Vector E,
                               float sizeOfUnit,
                               float gap,
                               int maxUnits)
{
    float radius = Pythagora(I, E);
    int totalUnits = (radius/ (sizeOfUnit + gap)) + 1;
    float angle = arctan(E-I);

    List<Vector> positions = empty;

    for(int currentUnit = 0; currentUnit < totalUnits; currentUnit++) {
        if currentUnit >= maxUnits {
            return positions;
        }
        Vector currentPosition = positionOnRadius(I,
                                                  sizeOfUnit,
                                                  gap,
                                                  angle,
                                                  currentUnit
                                   );
    positions.add(currentPosition);
    }
    return positions;
}
```

6.2 Unit Left Hanging

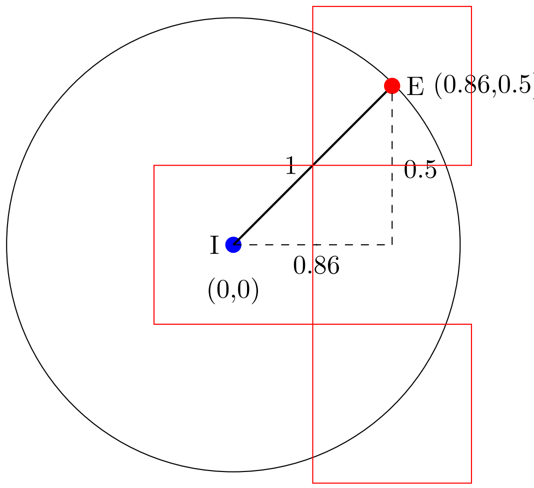
An issue arises when the length of the radius can not accomodate all the units that are available to be placed on the radius. The unit that does not fit in the radius will be left hanging.



One can decided to ignore the hanging unit and discard it



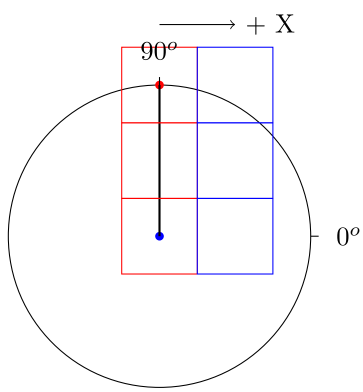
or create depth in the formation. Making hanging unit be placed directly behind the first unit located at I.

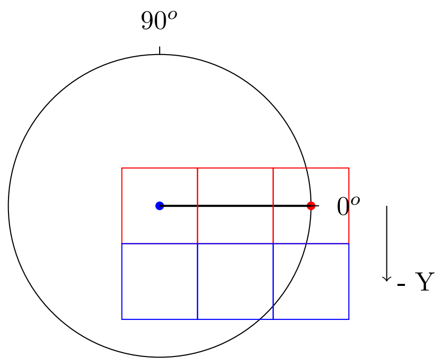


7 Depth in Formation

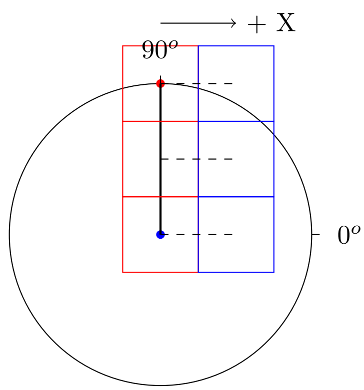
Two scenarios will help derive the pattern to implement the depth formation behaviour.

- `Front Units - F`: Red
- `Depth Units`: Blue





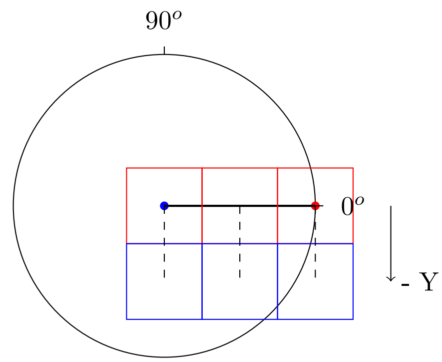
In the first scenario, the distance between the front unit and the depth unit is equal to the *distanceBetweenEachUnit* along the positive X-Axis.



Here $\sin(90) = 1$ and $\cos(90) = 0$, and $distanceBetweenEachUnit = 1$. As the depth unit must travel along the positive X-Axis by a distance equal to the *distanceBetweenEachUnit* from its front unit, the following equation is derived: where d = *distanceBetweenEachUnit*

$$\begin{aligned} depthUnitPosition &= Fn + ((\sin(90), \cos(90)) * d) \\ &= Fn + ((1, 0) * d) \\ &= Fn + (d, 0) \end{aligned}$$

In the second scenario, the distance between the front unit and the depth unit is equal to the *distanceBetweenEachUnit* along the negative Y-Axis.



Here $\sin(0) = 0$ and $\cos(0) = 1$, and $distanceBetweenEachUnit = 1$. As the depth unit must travel along the negative Y-Axis ba a distance equal to the *distanceBetweenEachUnit* from its front unit, the following equation is derived:

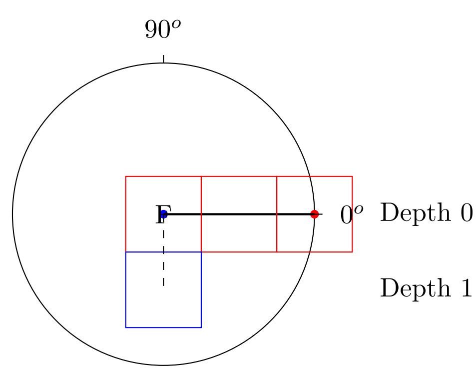
$$\begin{aligned} depthUnitPosition &= Fn + ((\sin(0), -\cos(0)) * d) \\ &= Fn + ((0, -1) * d) \\ &= Fn + (0, -d) \end{aligned}$$

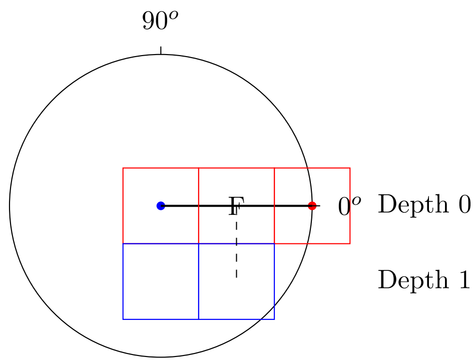
The general equation for the position of a *depth unit*:

$$depthUnitPosition = F + ((\sin(\theta), -\cos(\theta)) * d)$$

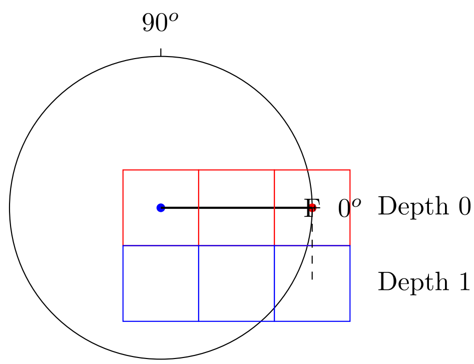
where: `F` is the unit's position that is in front of the *depth unit*. `d` is the *distanceBetweenEachUnit*.
NOTICE how the front units are on a depth of layer 0, and the units behind depth0 are on a depth of layer 1. This means the units on depth n will have units infront at depth layer *n - 1*.

This behaviour is visualized in the following figures.

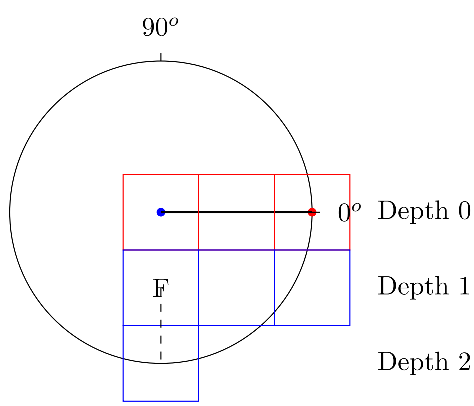




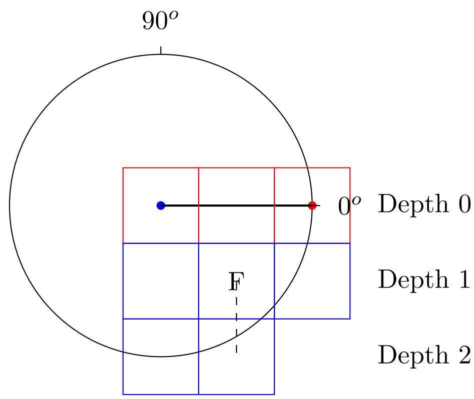
90^o
F
0^o
Depth 0
Depth 1



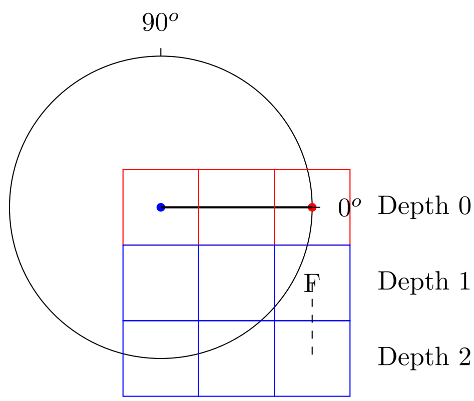
90^o
F
0^o
Depth 0
Depth 1



90^o
0^o
F
Depth 0
Depth 1
Depth 2



90^o
0^o
Depth 0
F
Depth 1
Depth 2



90^o
0^o
Depth 0
F
Depth 1
Depth 2

7.1 Integrating Depth

Check Appendix B to ensure that the coordinate system that is being used is the correct one.

The algorithm for calculating the depth positions requires the following variables:

- `Front Unit positions`
- `distanceBetweenEachUnit`
- `unitLimit`
- `Number of front units`
- `angle`

```
DepthPositions(List<Vector> positions,
               float distBtxt,
               int totalFrontUnits,
               int maxUnits,
               float angle);
{
    for(int currentDepthUnit = totalFrontUnits, currentFrontUnitIndex = 0;
        currentDepthUnit < maxUnits;
        currentDepthUnit++, currentFrontUnitIndex++
       )
    {
        Vector currentFrontUnit = positions[currentFrontUnitIndex];
        Vector depthPosition = depthUnitPosition(currentFrontUnit,
                                                  angle,
                                                  distBtxt
                              );
        positions.add(depthPosition);
    }
    return positions;
}
```

Integrate it with the `CalculatePositionsOnRadius`:

```
List<Vector> PositionsOnRadius_V3_Limit_Depth(Vector I,
                                              Vector E,
                                              float sizeOfUnit,
                                              float gap,
                                              int maxUnits);
{
    float radius = Pythagora(I, E);
    int totalFrontUnits = (radius/ (sizeOfUnit + gap)) + 1;
    float angle = arctan(E-I);

    List<Vector> positions = empty;

    for(int currentUnit = 0; currentUnit < totalFrontUnits; currentUnit++) {
        Vector currentPosition = positionOnRadius(I,
                                                  sizeOfUnit,
                                                  gap,
                                                  angle,
                                                  currentUnit
                                 );
        positions.add(currentPosition);
    }

    float distBtxt = sizeOfUnit + gap;
    positions = DepthPositions(
                   positions,
                   distBtxt,
                   totalFrontUnits,
                   maxUnits,
                   angle
               );

    return positions;
}
```

8 Refactoring Procedures

The *PositionsOnRadius* is doing the following:

1. Initializing unit data
2. Gathering front row positions
3. Gathering depth positions

A procedure called *GridFormationPositions* will in charge of initializing the data, and deciding when to gather front positions and depth positions. Allowing *PositionsOnRadius* to solely focus on calculating the front unit positions.

8.1 Grid Formation Positions

```
List<Vector> GridFormationPositions(Vector I,
                                    Vector E,
                                    float sizeOfUnit,
                                    float gap,
                                    int maxUnits)
{
    float radius = Pythagora(I, E);
    float distBtxt = sizeOfUnit + gap;
    int totalFrontUnits = (radius/distBtxt) + 1;
    float angle = arctan(E-I);

    List<Vector> frontPositions = empty;
    List<Vector> depthPositions = empty;

    List<Vector> gridPositions = empty;

    gridPositions = PositionsOnRadius(
                        I,
                        sizeOfUnit,
                        gap,
                        angle,
                        totalFrontUnits
                  );
    gridPositions = DepthPositions(
                        gridPositions,
                        distBtxt,
                        totalFrontUnits,
                        maxUnits,
                        angle
                  );

    return gridPositions;
}
```

8.2 Front Positions (Version 4)

```
List<Vector> PositionsOnRadius(Vector I,
                               Vector E,
                               float sizeOfUnit,
                               float gap,
                               float angle,
                               int totalFrontUnits)
{
    List<Vector> positions = empty;

    for(int currentUnit = 0; currentUnit < totalFrontUnits; currentUnit++) {
        Vector currentPosition = positionOnRadius(
                                       I,
                                       sizeOfUnit,
                                       gap,
                                       angle,
                                       currentUnit
                                 );
    positions.add(currentPosition);
    }

    return positions;
}
```

8.3 Depth Positions (Version 2)

```
List<Vector> DepthPositions(List<Vector> Ps,
                            float distBtxt,
                            int totalFrontUnits,
                            int maxUnits,
                            float angle)
{

    for(int currentDepthUnit = totalFrontUnits, currentFrontUnitIndex = 0;
        currentDepthUnit < maxUnits;
        currentDepthUnit++, currentFrontUnitIndex++
       )
    {
        Vector currentFrontUnit = positions[currentFrontUnitIndex];
        Vector depthPosition = depthUnitPosition(currentFrontUnit,
                                                  angle,
                                                      distBtxt
                                );
        positions.add(depthPosition);
    }
    return positions;
}
```

9 Main Procedure

```
GridFormation_V2:
{
bool dragging = false;
MouseButton button = RIGHT;

Vector I = Vector(0,0);
Vector E = Vector(0,0);
Vector C = Vector(0,0);

float sizeOfUnit = s;
float gap = g;
int maxUnits = totalUnits;
List<Vector> positions = empty;

loop {
    dragging = Dragging(I, E, button);

    if dragging {
        C = mousePosition();
    Vector position = GridFormationPositions(I, C, sizeOfUnit, gap, maxUnits);
    }
}
}
```

10 Conclusion

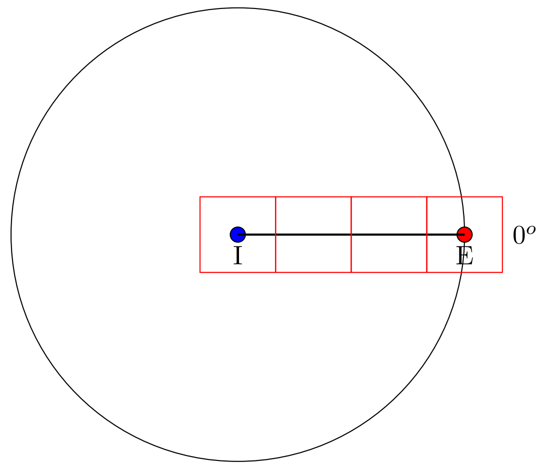
What has been implemented lays the foundation for the Grid Formation Behaviour. From here one can add, modify, or experiment with the behaviour as one desires. All new additions to the Base behaviour will be included in the `Appendix A`.

A General Procedure for Formation Positions

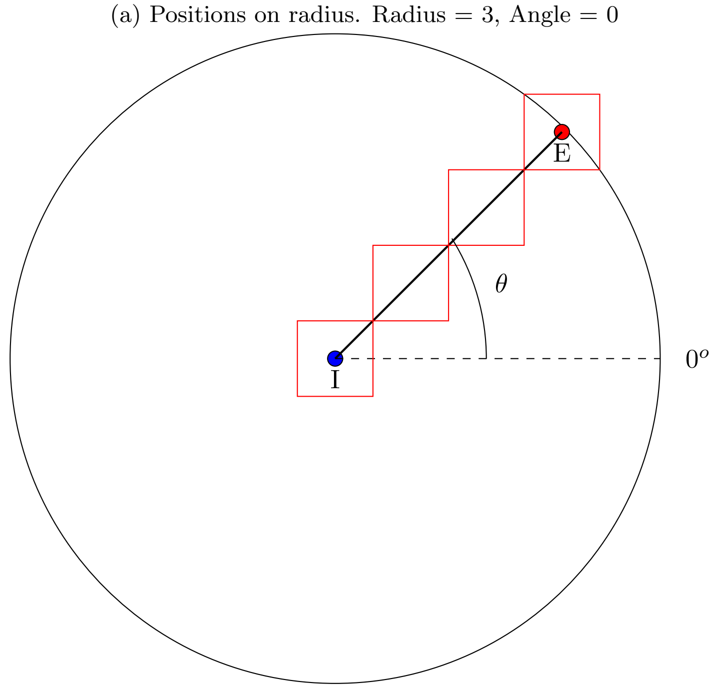
At the moment two methods are utilized to calculate the positions on the radius, and the positions behind the units on the radius. This approach is limiting when wanting to affect a certain depth, as doing so would require one to juggle with various variables.
To affect a specific depth and not have to juggle with various variables, a general procedure will be implemented that will allow more control on the grid positions.

A.1 Scenarios

The following scenarios will help derive the procedure. Currently the positions on the radius are calculated starting from I and moving up to E



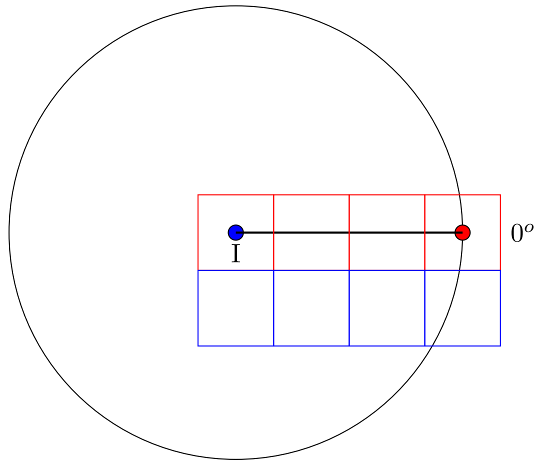
(a) Positions on radius. Radius = 3, Angle = 0



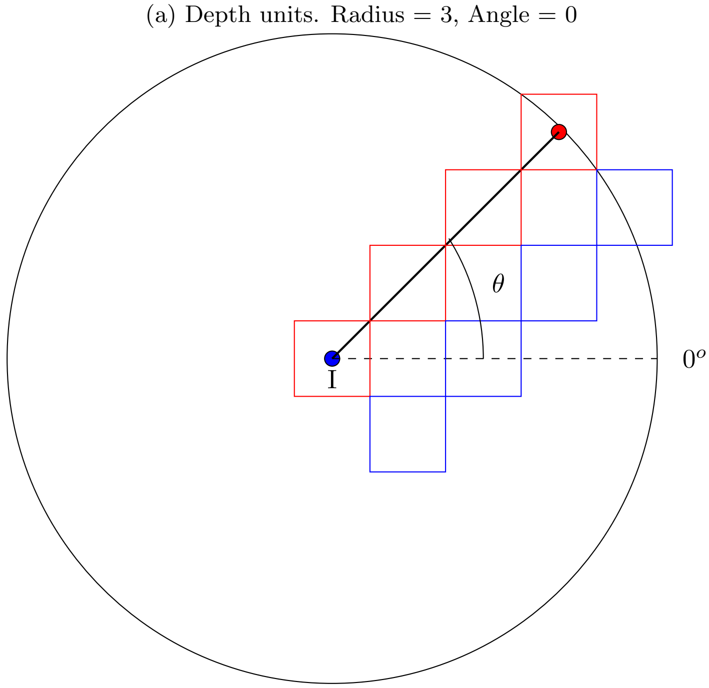
(b) Positions on radius. Radius = 3, Angle = θ

Figure 3: $positionOnRadius = I + (\cos(\theta), \sin(\theta)) * d * i$

The units currently lie on positions located on the radius, while the depth units will lie behind the front units.



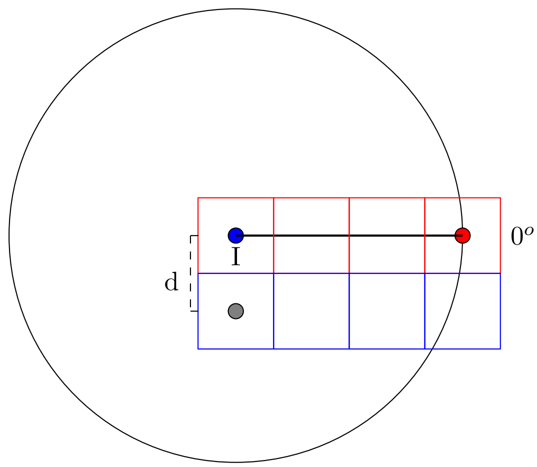
(a) Depth units. Radius = 3, Angle = 0



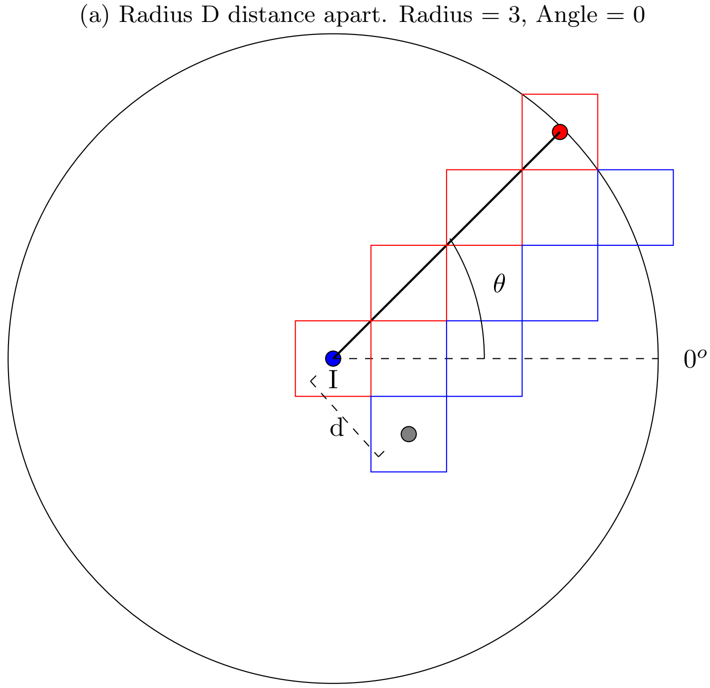
(b) Depth units. Radius = 3, Angle = θ

Figure 4: $depthUnitPosition = F + (\sin(\theta), -\cos(\theta)) * d$

Notice how the beginning of the depth position is exactly `d` distance away from `I`.



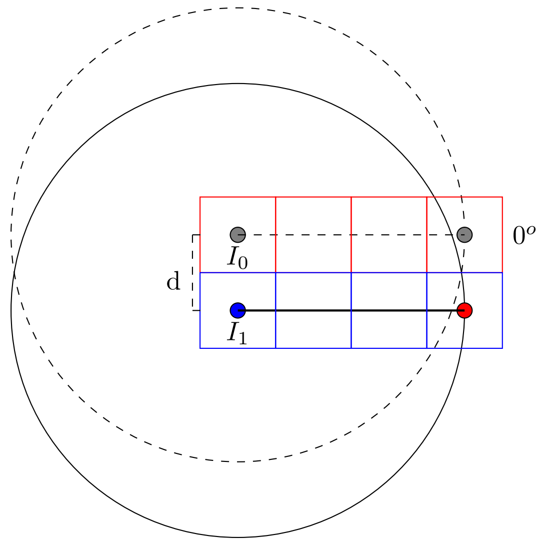
(a) Radius D distance apart. Radius = 3, Angle = 0



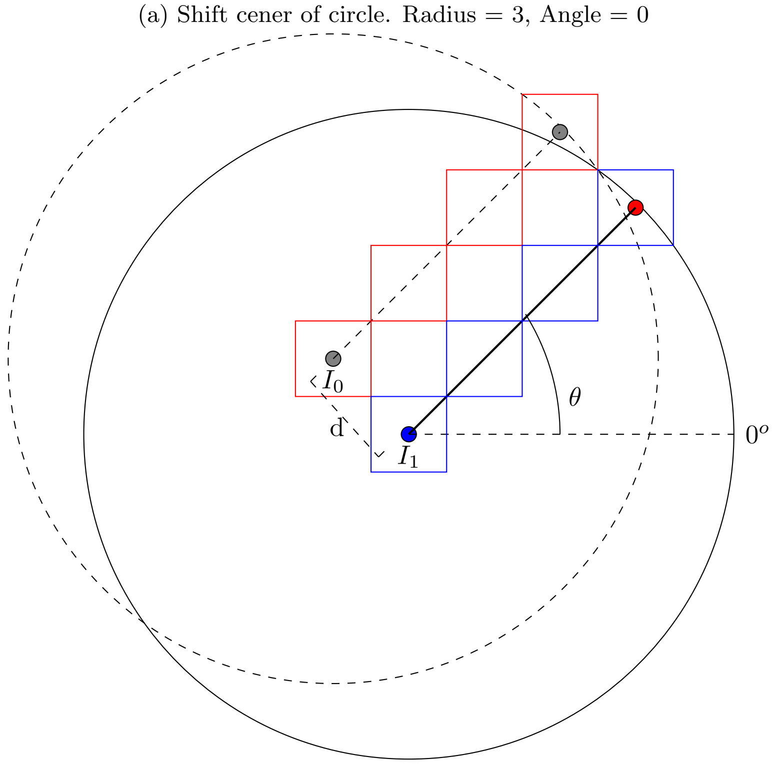
(b) Radius D distance apart. Radius = 3, Angle = θ

Figure 5: d = distBtxtUnit

One can imagine moving the center of the circle to the beginning of the depth, I_d. Where I_d is the intial position, `I`, on the depth d.



(a) Shift cener of circle. Radius = 3, Angle = 0



(b) Shift center of circle. Radius = 3, Angle = θ

Figure 6: Shifting center of circle

Since this is the same radius but in different position, the same equation `positionsOnRadius` can be utilized to calculate the depth positions. The I_d position must be found in order to calculate the positions on the current depth radius.

A.2 I Depth

From the diagram above, I_1 is d distance away from I. This is the exact calculation as acquiring a depth unit from the front position, `F`. Simply replace `F` with $\mathtt{I}_0$ for the example above:

$$I_1 = I_0 + (d * (\sin(\theta), -\cos(\theta)))$$

The general equation for gathering the I_{d+1} given I_d:

$$I_{d+1} = I_d + (d * (\sin(\theta), -\cos(\theta)))$$

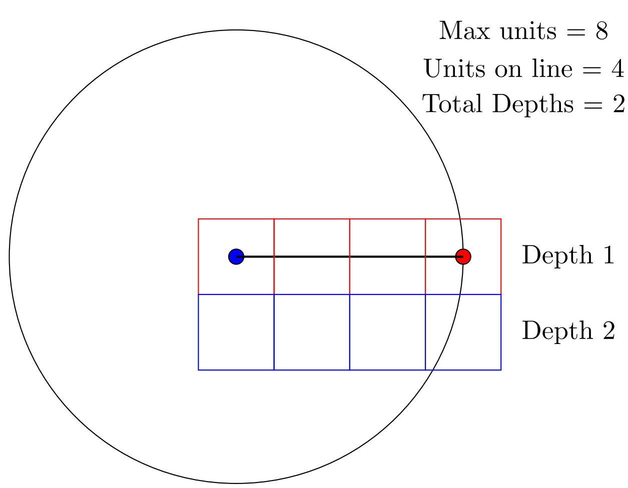
A.3 Number of Depths

The number of depths is determined by how many rows of units are in the formation. One must first calculate the:

- Number of rows in formation, given max units
- Number of units on a row/radius

$$totalDepths = \frac{maxUnits}{unitsOnRadius}$$

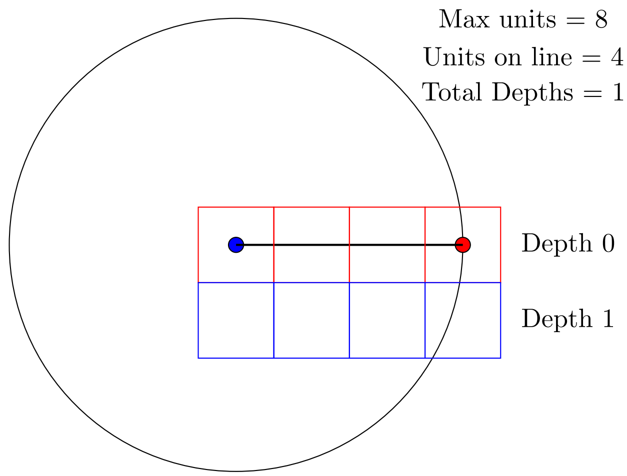
A.3.1 Example



This considers the front row a depth row.
Where $\frac{10}{10} = 1$. Instead of treating the first row as a depth row of one, it will be better to treat it as the 0th depth row. resulting in the following equation

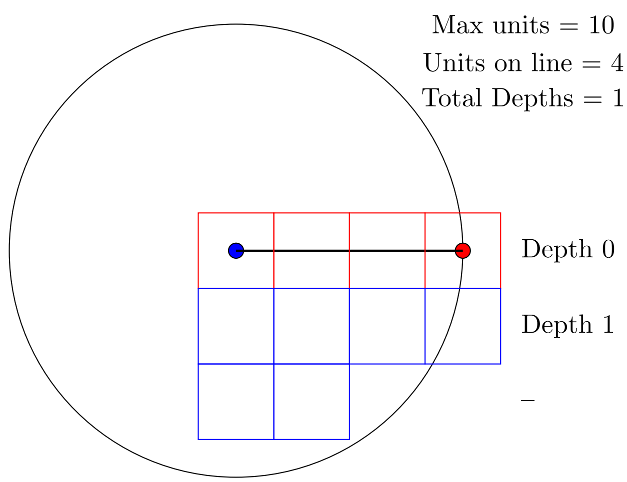
$$totalDepth = (maxUnits/unitsOnRadius) - 1$$

Depicting the 0th depth row, otherwise known as the front row.



A.3.2 Leftover units

The equation currently calculates the number of full rows, but does not take into consideration the rows that are not full. In other words, the last row, if not full, will not be considered in the totalDepth equation.



Leftover units are calculated by taking the remainder of total depth:

$$\texttt{maxUnits} \bmod \texttt{unitsOnRadius}$$

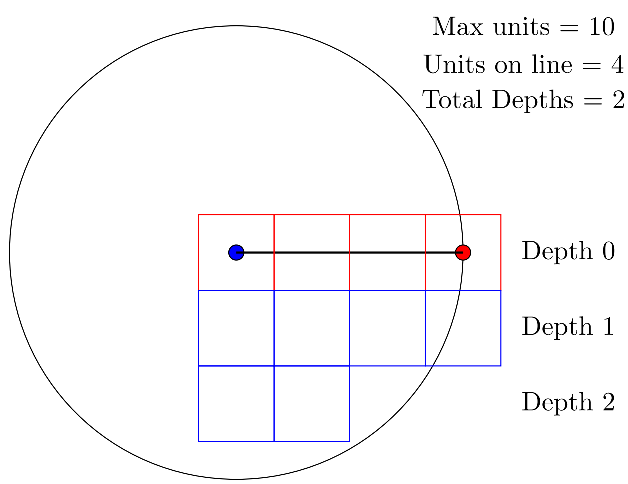
When left-over units is greater than zero, then the last depth contains leftover units and a `1` is added onto `total depths`. If not then the last row is full, and `0` is added onto `total depths`.

This behaviour is abstracted into the function below.

$$LeftOverCheck(maxUnits, unitsOnRadius)$$

Having *Left-over* check be added onto the *totalDepth*:

$$totalDepth = (\frac{maxUnits}{unitsOnRadius} - 1) + LeftOverCheck(maxUnits, unitsOnRadius)$$



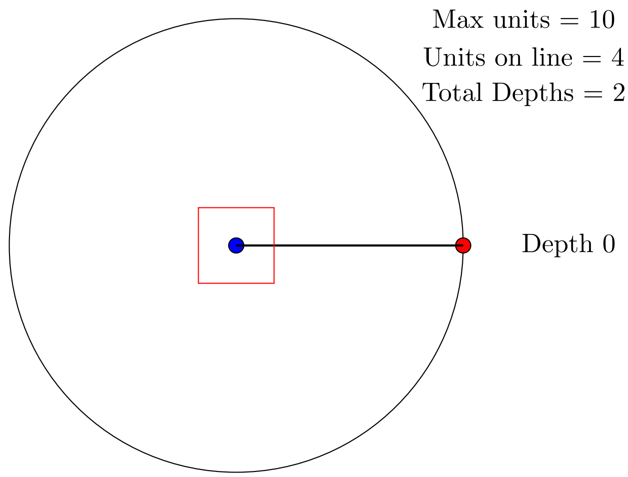
A.4 Deriving the procedure

The general algorithm goes as follows:

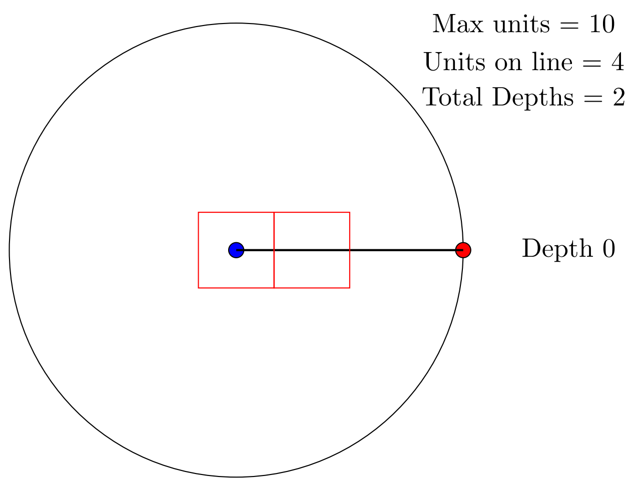
1. Given $I_d = I_0$
2. Calculate positions on I_d
3. No more units? Go to 7
4. Calculate I_{d+1}.
5. $I_d = I_{d+1}$.
6. Repeat 2
7. Return positions.

A.5 Visual Representation

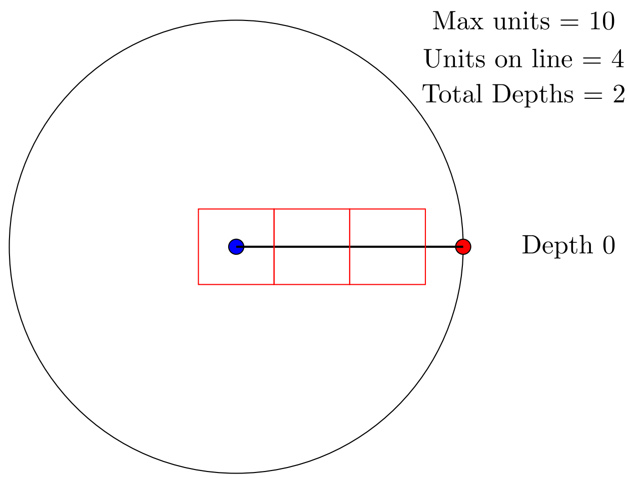
A.5.1 Calculating positions on I_0



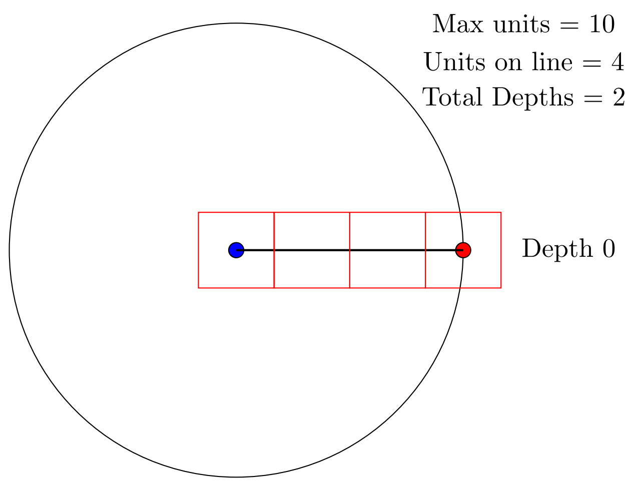
(a) position = I_0 + (d * 0 * $(\cos(0), \sin(0))$)



(b) position = I_0 + (d * 1 * $(\cos(0), \sin(0))$)



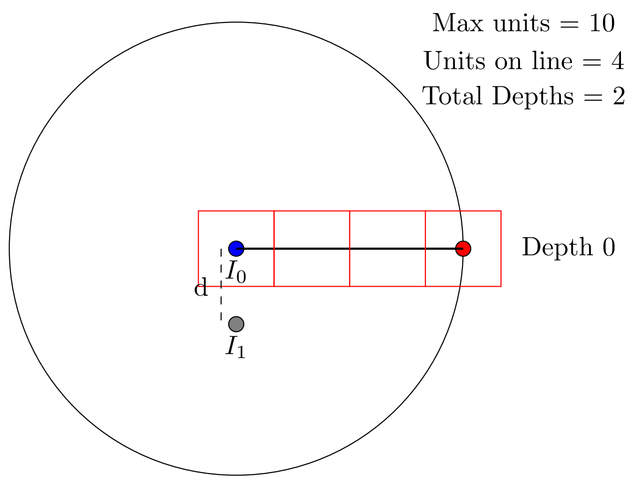
(a) position = I_0 + (d * 2 * $(\cos(0), \sin(0))$)

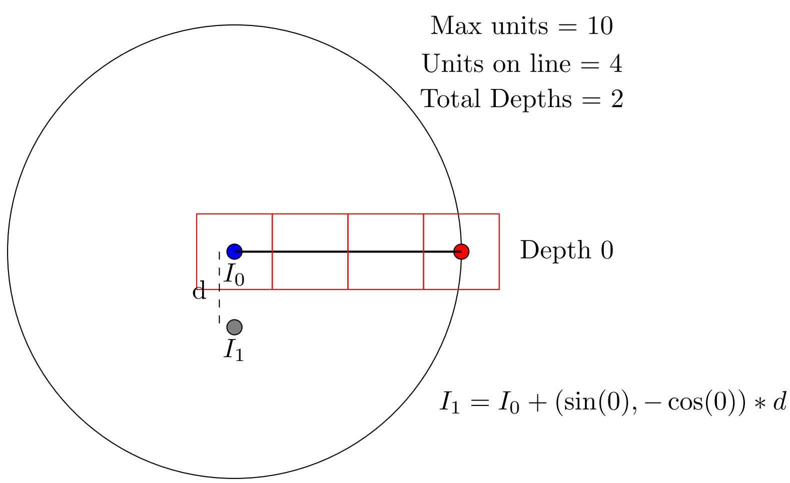


(b) position = I_0 + (d * 3 * $(\cos(0), \sin(0))$)

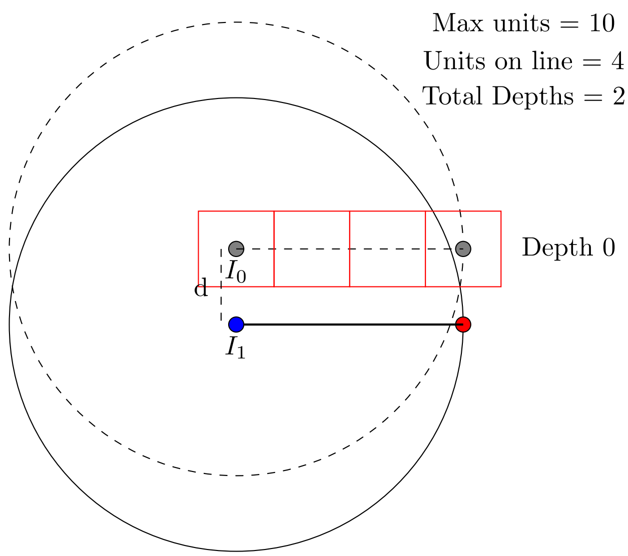
Figure 8: $I_d = I_0$

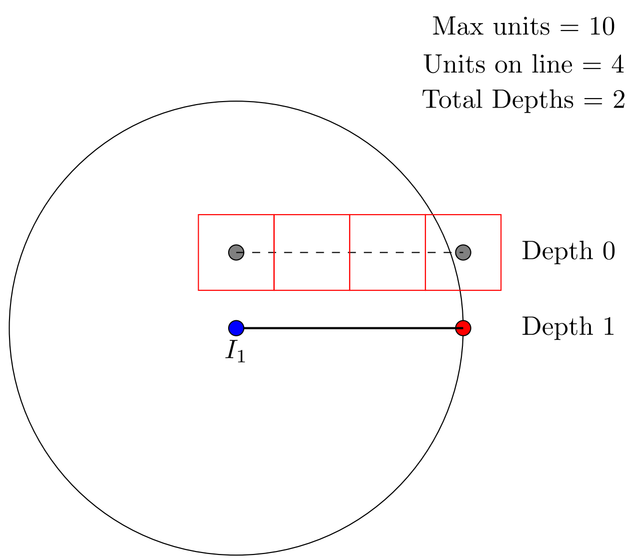
A.5.2 Calculating I_1



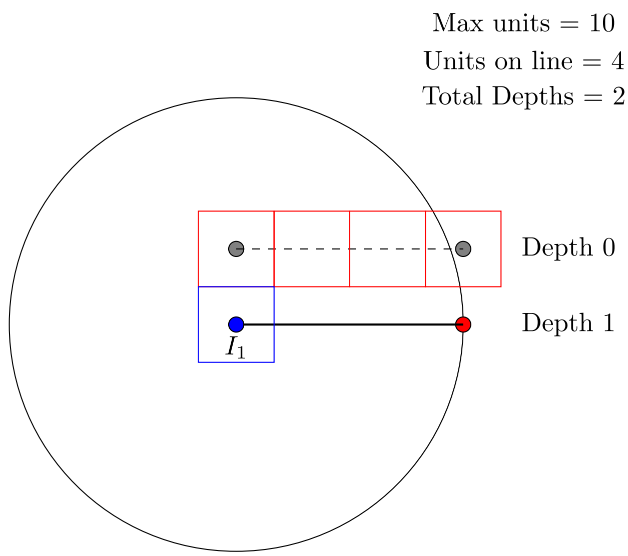


A.5.3 Shifting circle

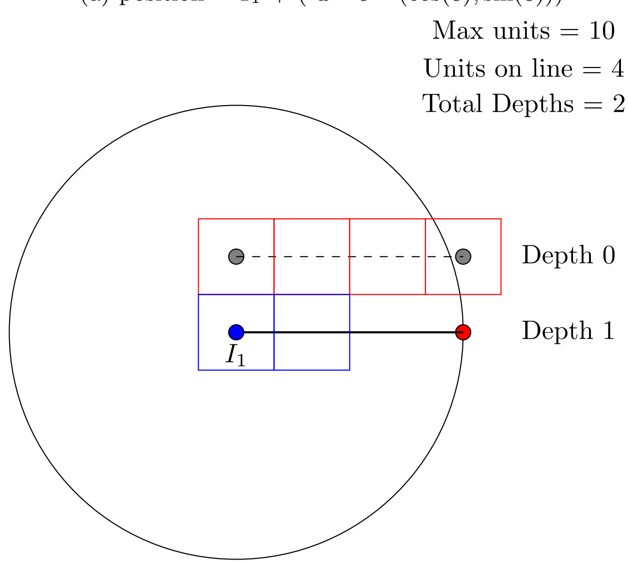




A.5.4 Calculating positions on I_1

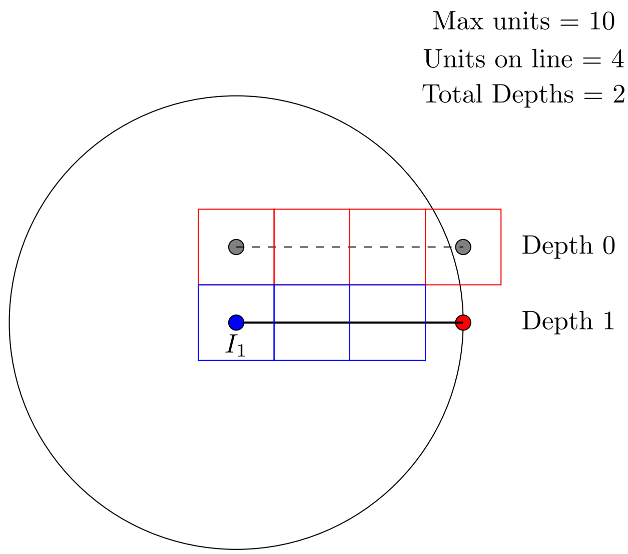


(a) position = I_1 + (d * 0 * $(\cos(0), \sin(0))$)

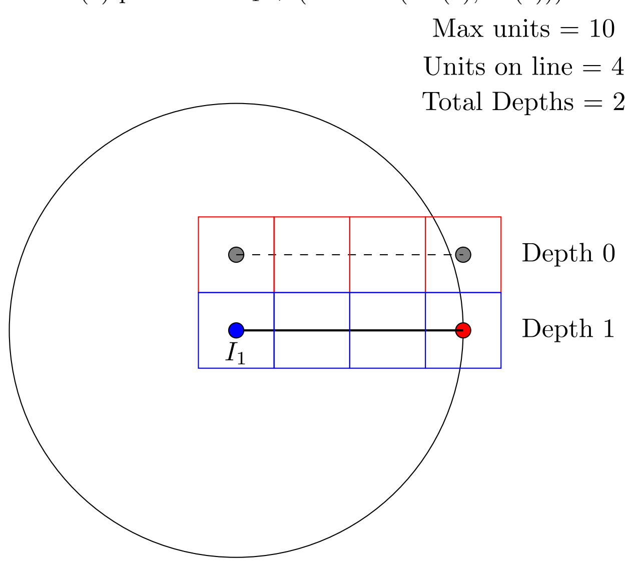


(b) position = I_1 + (d * 1 * $(\cos(0), \sin(0))$)

Figure 11: $I_d = I_1$



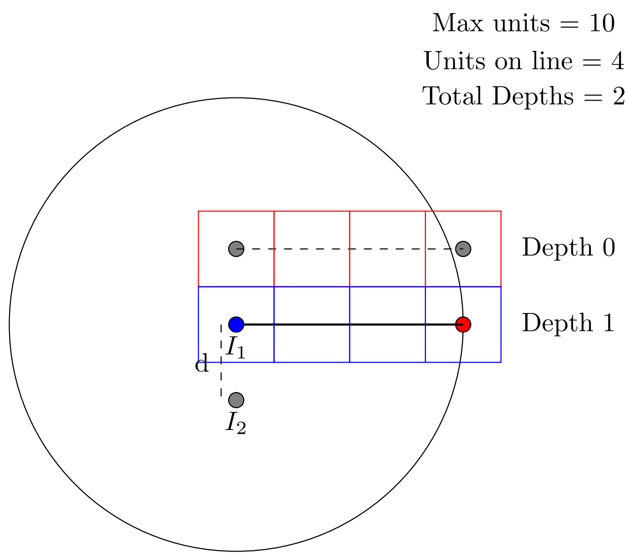
(a) position = I_1 + (d * 2 * $(\cos(0), \sin(0))$)

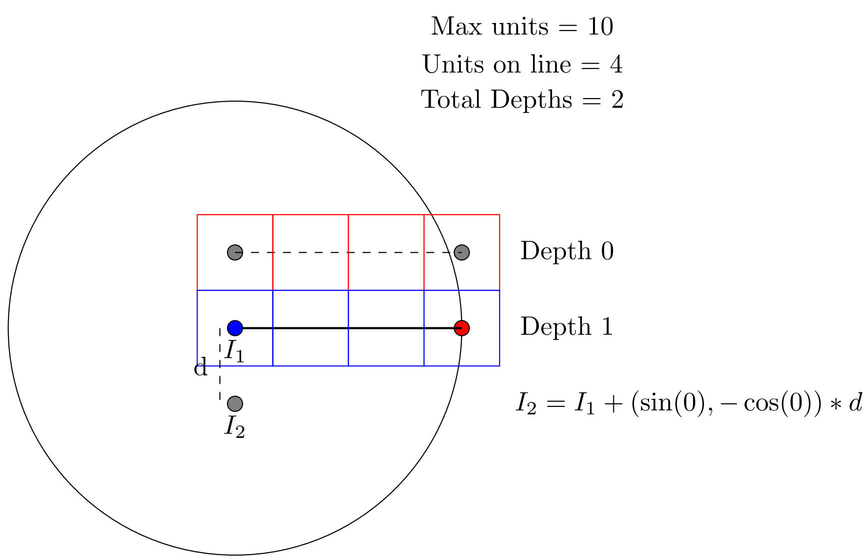


(b) position = I_1 + (d * 3 * $(\cos(0), \sin(0))$)

Figure 12: $I_d = I_1$

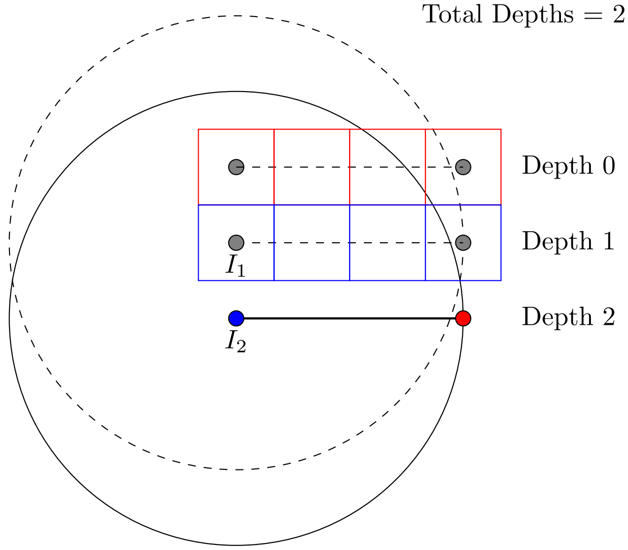
A.5.5 Calculating I_2



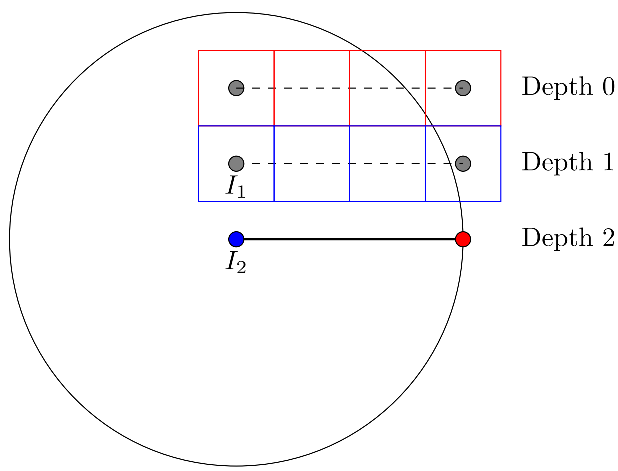


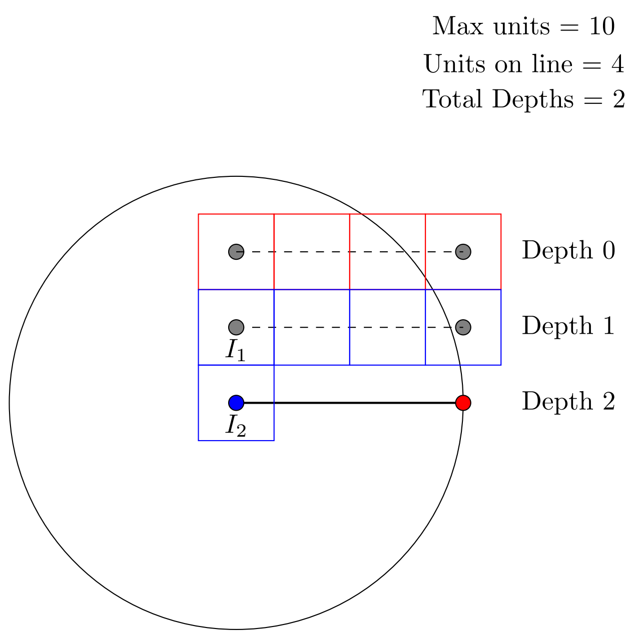
A.5.6 Shifting circle

Max units = 10
Units on line = 4
Total Depths = 2

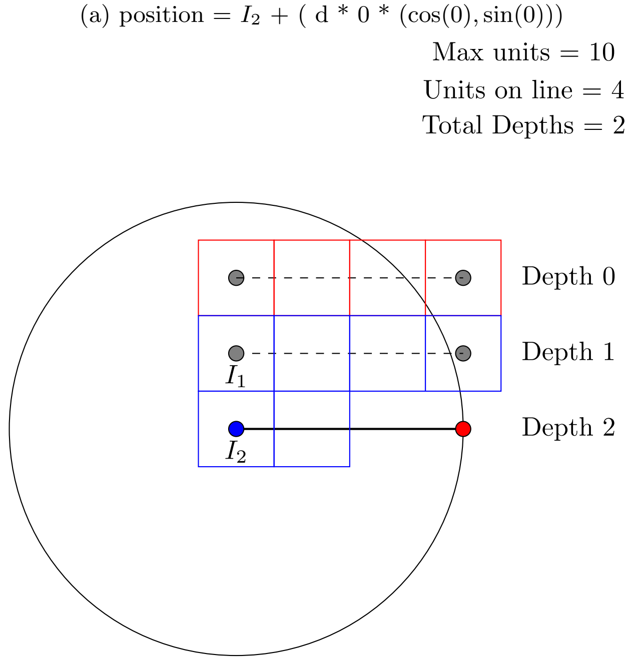


Max units = 10
Units on line = 4
Total Depths = 2





(a) position = I_2 + (d * 0 * $(\cos(0), \sin(0))$)



(b) position = I_2 + (d * 1 * $(\cos(0), \sin(0))$)

Figure 15: $I_d = I_2$

A.5.7 Calculating Positions on single line

This is the same procedure as calculating the positions of the `Front Units`.

```
List<Vector> PositionsOnSingleRow(Vector I,
                                  Vector E,
                                  float sizeOfUnit,
                                  float gap,
                                  float theta,
                                  int maxUnits
                     )
{
    float radius = Pythagora(I, E);
    float distBtxt = sizeOfUnit + gap;
    int unitsOnRadius = (radius/distBtxt) + 1;

    List<Vector> positions = empty;

    for(int currentUnit = 0; currentUnit <= maxUnits; currentUnit++) {
        if currentUnit > maxUnits {
            Vector position = positionsOnRadius(I,
                                                distBtxt,
                                                currentUnit,
                                                angle
                                               );
            positions.add(position);
        }
    }
    return positions;
}
```

Minor changes were added to accomodate for the general behaviour.

A.5.8 Moving along Depths

For every new depth, the positions on the line must be calculated.

A.6 Final Procedure

Combining both steps, gives the following procedure:

```
GeneralPositions:
{
    List<Vector> GeneralPositions(Vector I,
                                  Vector E,
                                  float sizeOfUnit,
                                  float gap,
                                  float angle,
                                  int totalFrontUnits,
                                  int maxUnits
                                 )
    {
        float radius = Pythagora(I, E);
        int totalDepth = TotalDepth(maxUnits, totalFrontUnits);
        float distBtxt = sizeOfUnit + gap;

        Vector currentI = I;

        List<Vector> positions = empty;

        for (int curDepth = 0, currentUnit = 0;
             curDepth <= totalDepth;
             curDepth++)
        {
            for(unitInDepth = 0;
                unitInDepth < totalFrontUnits;
                currentUnit++, unitInDepth++
               )
            {
                if currentUnit > maxUnits {
                    return positions;
                }

                Vector position = positionOnRadius(I,
                                                   sizeOfUnit,
                                                   gap,
                                                   angle,
                                                   unitInDepth
                                                  );
                positions.add(positions)
            }
            currentI = currentI + (distBtxt * (sin(angle), -cos(angle))
        }
        return positions;
    }
}
```

B Coordinate System Accomodation

The coordinate system being used is a **left-handed** coordinate system.

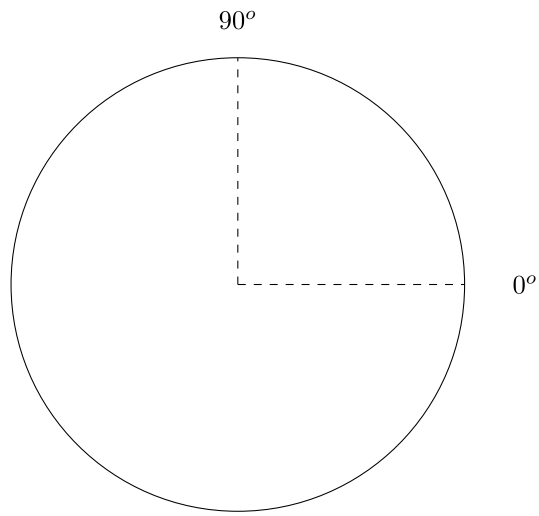


Figure 16: Current LHS

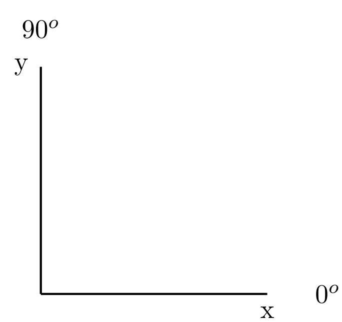


Figure 17: LHS Axis

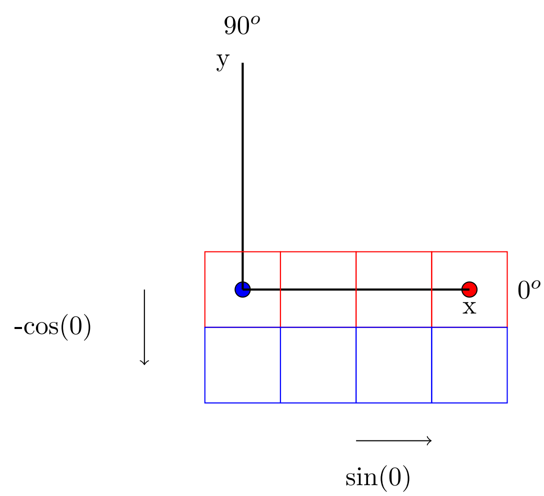


Figure 18: Depth position below X-axis

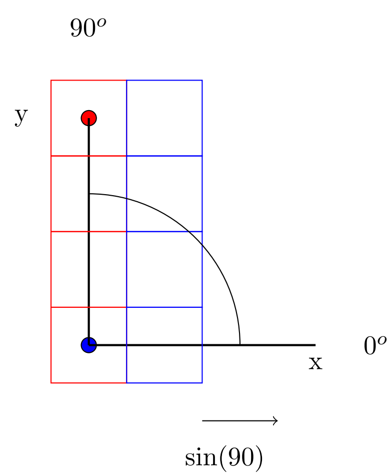


Figure 19: Depth position to the right of Y-axis

From these diagrams one can see that the depth positions is calculated with the following equation:

$$depthPosition = (\sin(\theta), -\cos(\theta))$$

The current implementation will work if a **left-handed** coordinate system is being used, with the *up vector* being $(0, 1)$.

However, in certain game engines a different coordinate system is used. This section will go over how to accomodate for the different coordinate systems.

B.1 Negative Y

In this situation, the *up vector* is now $(0, -1)$ so the following changes must be made.

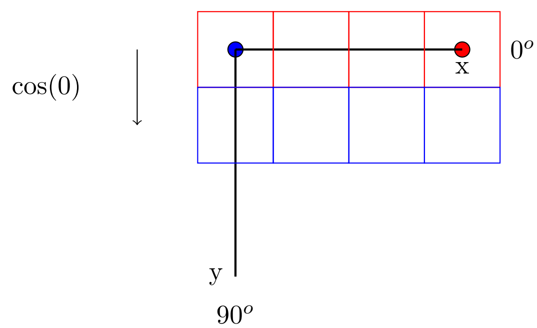


Figure 20: Negative up vector

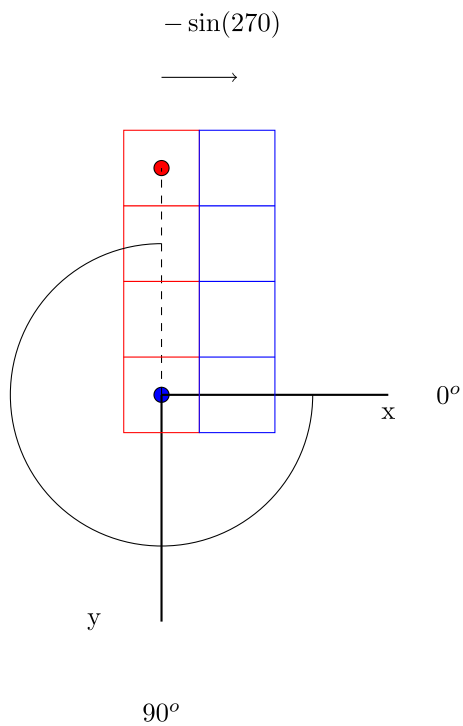


Figure 21: Depth position accomodation

The depth position in this coordinate system is based on the following equation:

$$depthPosition = (-\sin(\theta), \cos(\theta))$$

B.2 Right-Hand Coordinate System

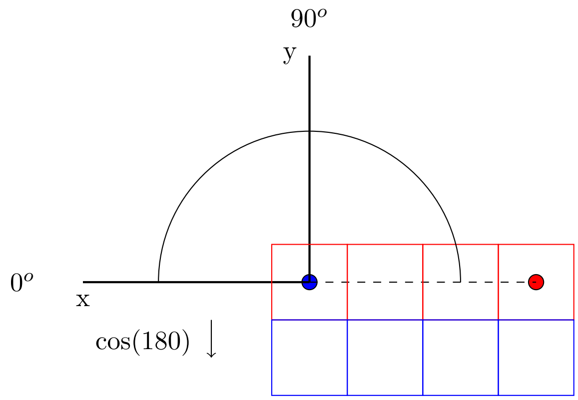


Figure 22: RHS units

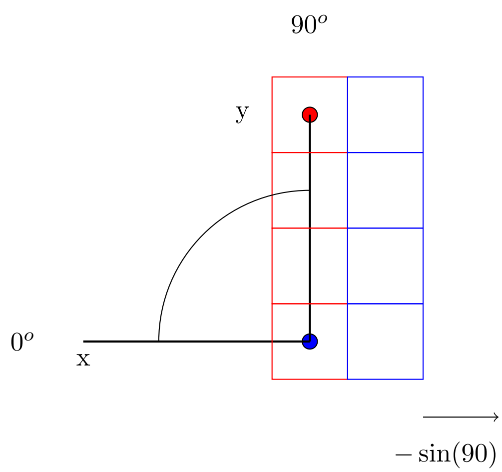


Figure 23

The depth position equation is as follows:

$$depthPosition = (-\sin(\theta), \cos(\theta))$$

B.2.1 Negative Y

The Right-Handed system with Y pointing downwards.

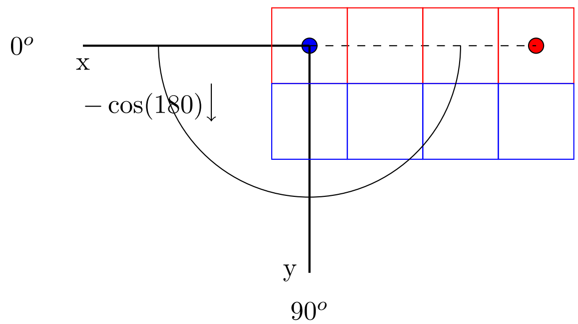


Figure 24: Negative Y

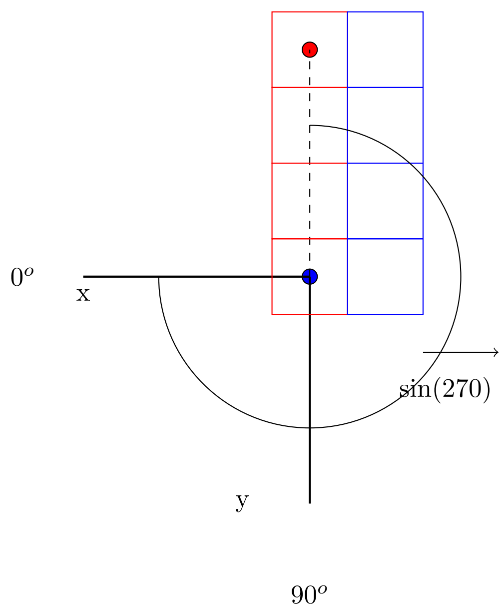


Figure 25

The depth positin is calculated as follows:

$$depthPosition = (\sin(\theta), -\cos(\theta))$$

These diagrams will help understand why some of the depth units may not be behaving as expected

C Pascal's Positions

Currently in the last row, the units go to the far left of the formation.

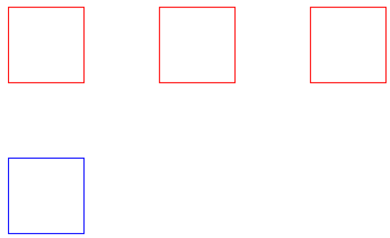


Figure 26: Current formation

What if one wants the units on the last row be placed in the middle of the formation as so:

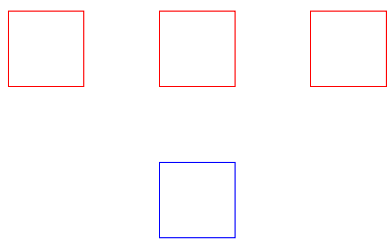


Figure 27: Desired formation

This behaviour can be implemented by observing Pascal's Triangle and deriving a pattern for the last row. This paper will not delve into the details as to how Pascal's Pattern is derived, rather it will focus on the essential properties that can be used to implement the behaviour.

C.0.1 Steps in Applying Pascal's Pattern Positions

1. Find mid point
2. Calculate Pascal's center from mid point
3. Calculate Pascal's end from center
4. Calculate positions starting from Pascal's end.

C.1 Scenarios

Given the mid point of the line, `m`.

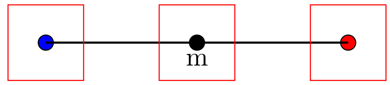


Figure 28: Mid point

C.1.1 Pascal's Center

When there is one unit, it will lie directly behind the mid point, `d` distance away.

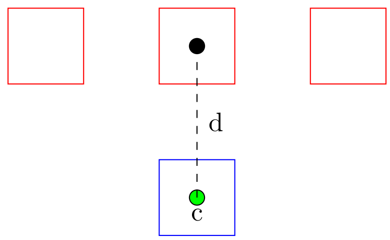


Figure 29: Distance to center

The center, `c`, is then calculated as follows:

$$c = mid - d$$

Apllying it to the Unit circle is the same as calculating the depth unit position from the front unit. Thi difference is that the front unit is the mid point, `m`, of the radius.

$$c = mid + (\sin(\theta), -\cos(\theta)) * d$$

This works for a formation of depth 1. To have it work with multiple depths, `c` must be multiplied by the depth it lies on to correctly get the `c` being at the last row.

Resulting in the equation:

$$c = mid + (\sin(\theta), \cos(\theta)) * d * curDepth$$

C.1.2 Pascal's End

When there is one unit, Pascal's End, `e` lies exactly on `c`.

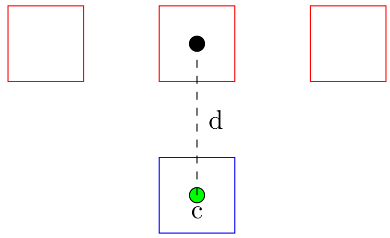


Figure 30: Single Unit

Given two units, Pascal's End lies $\frac{d}{2}$ distance away from `c`.

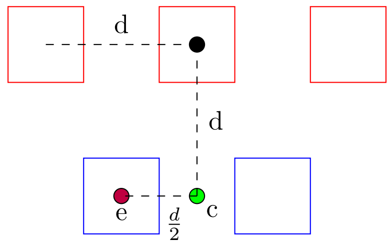


Figure 31: e = c - $\frac{d}{2}$

Given three units, Pascal's End lies d distance away from `c`.

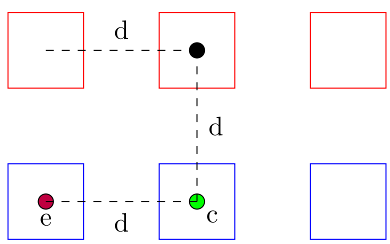


Figure 32: e = c - d

Allowing one to derive the following equation for Pascal's End:

$$e = c - \frac{d}{2} * (numUnits - 1)$$

Now applying it onto the Unit Circle:

$$e = c - \frac{d}{2} * (numUnits - 1) * (\cos(\theta), (\sin(\theta)))$$

c is subtracted from the coordinates because e lies below C in terms of radius given and angle θ. Acquiring the negative value of $(\cos(\theta), \sin(\theta))$ allows one to travel the radius in the opposite direction (i.e downwards).

C.2 Applying the Unit Circle

Given a circle of radius 6.

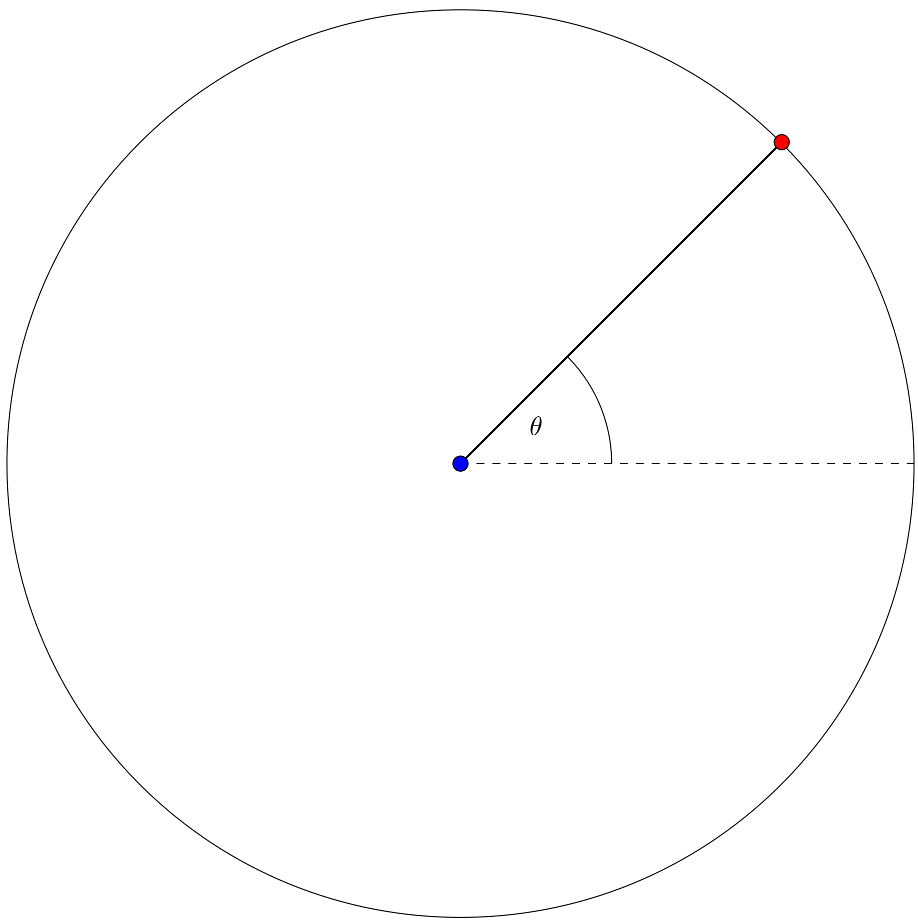


Figure 33: Circle of radius = 6

The following properties apply:

- Unit size = 1
- gap = 1.
- max units = 5.

- total front units = 3

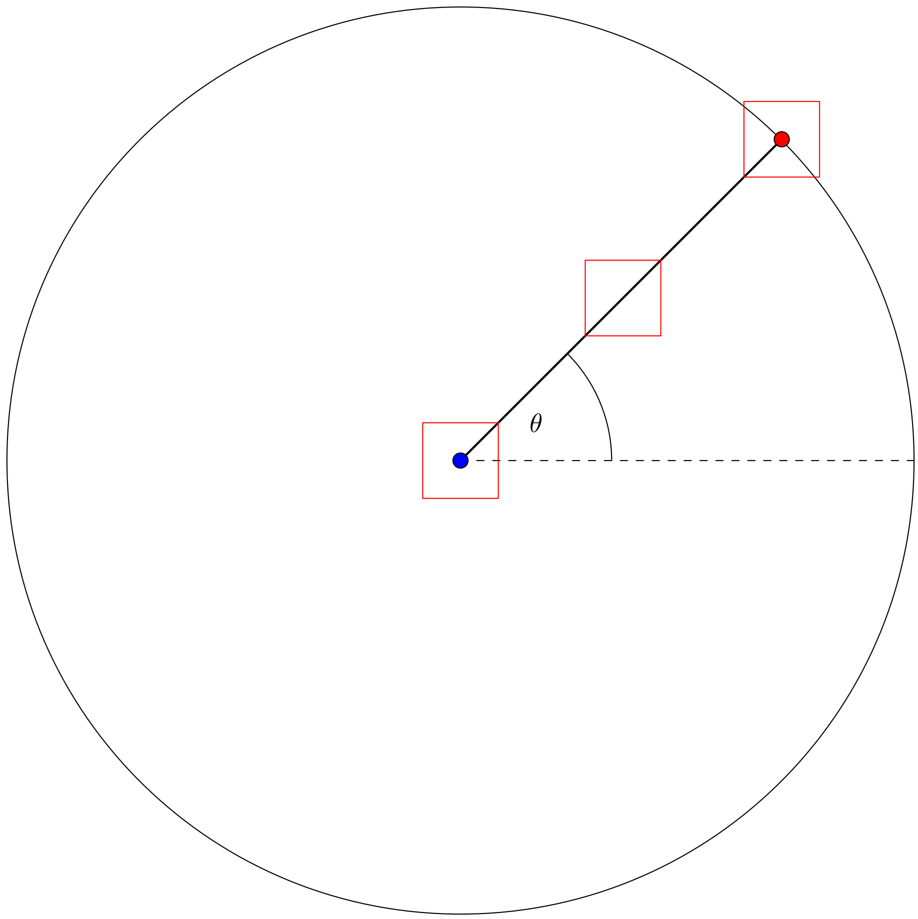


Figure 34: Units on Circle

C.2.1 Mid point

The mid point is the center of the radius. $mid = (E - I)/2$
This gives the vector of half the radius, `(rx, ry)`, but not the specific point at the mid point of th radius.
To align the mid point onto the current radius, simply add `(rx, ry)` to `I`. $midPoint = I + mid$

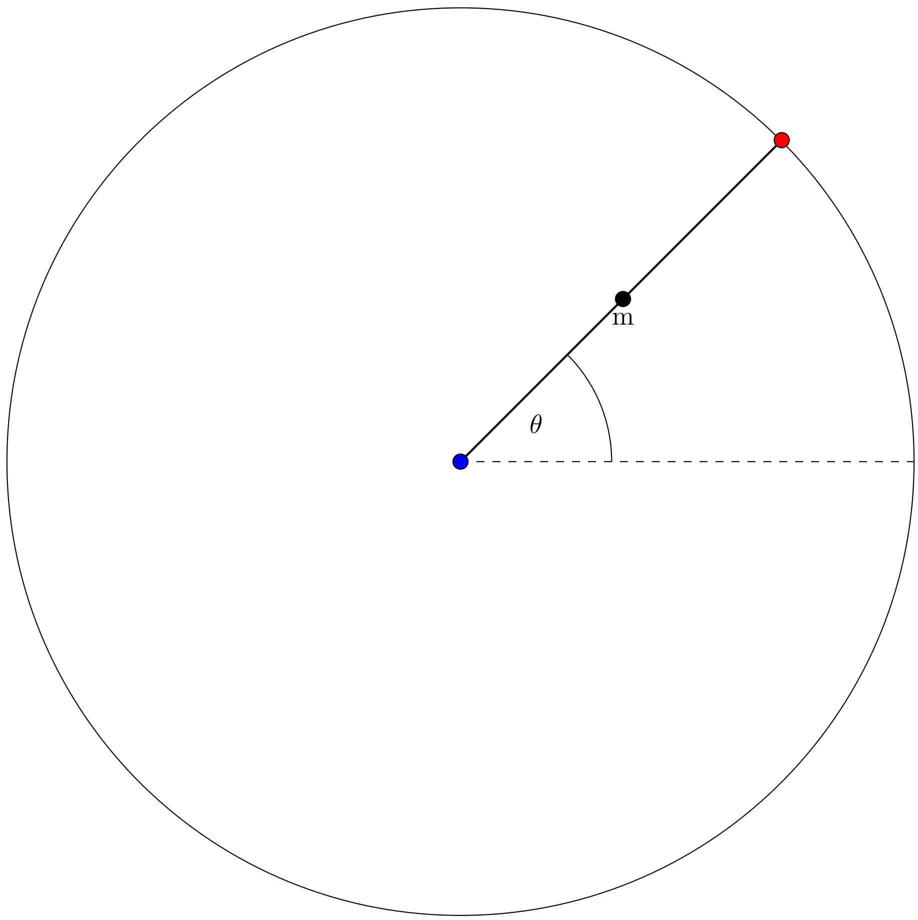


Figure 35: Mid

The equation for the mid point, `m`:

$$m = I + \frac{E - I}{2}$$

C.2.2 Center point

Apply the equation of the center point given the mid point

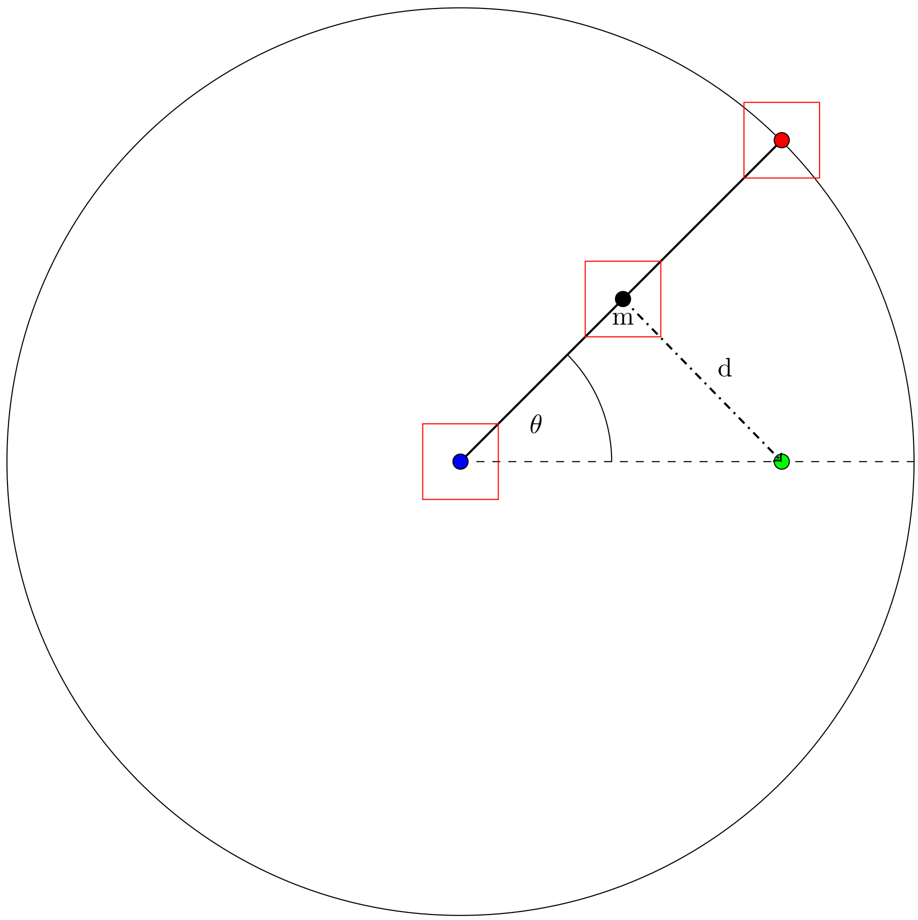


Figure 36: c = m + d * $(\sin(\theta), -\cos(\theta)) * 1$

NOTICE how this is the same behaviour as calculating the unit in $depth_n$ given the front unit at $depth_{n-1}$

C.2.3 End point

To find `e`, apply the equation:

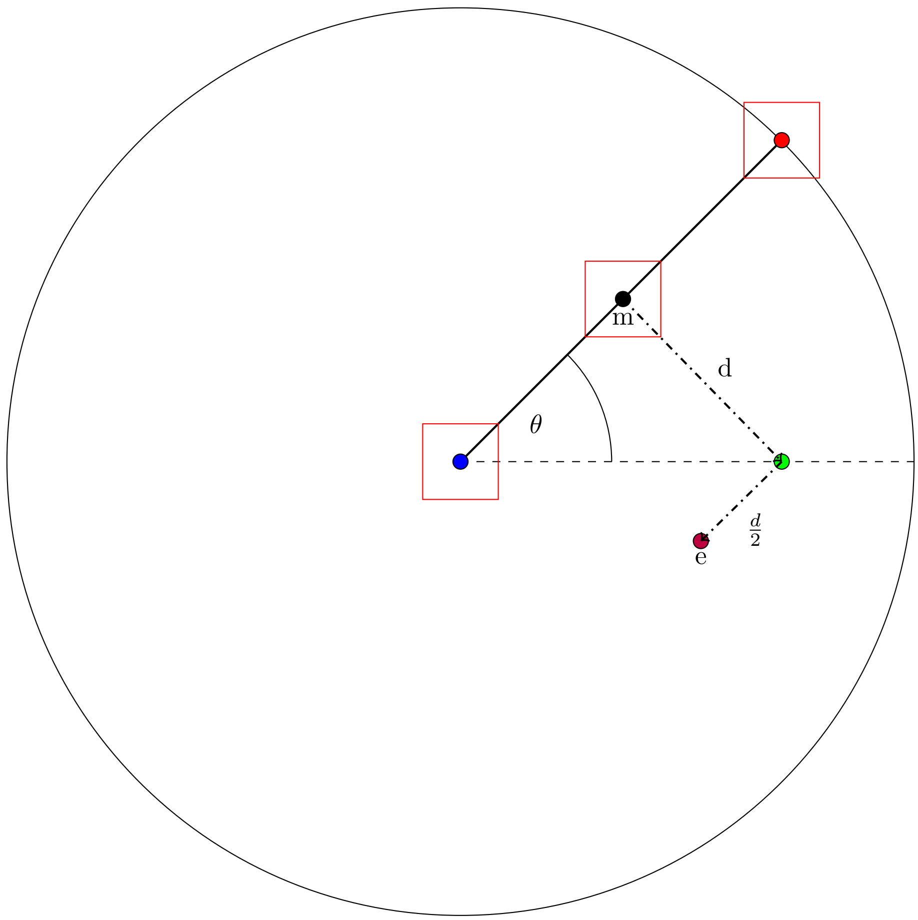


Figure 37: $e = c - \frac{d}{2} * (numUnits - 1) * (\cos(\theta), \sin(\theta))$

Imagine having another circle with `c` being the center of the circle. (fitting name).

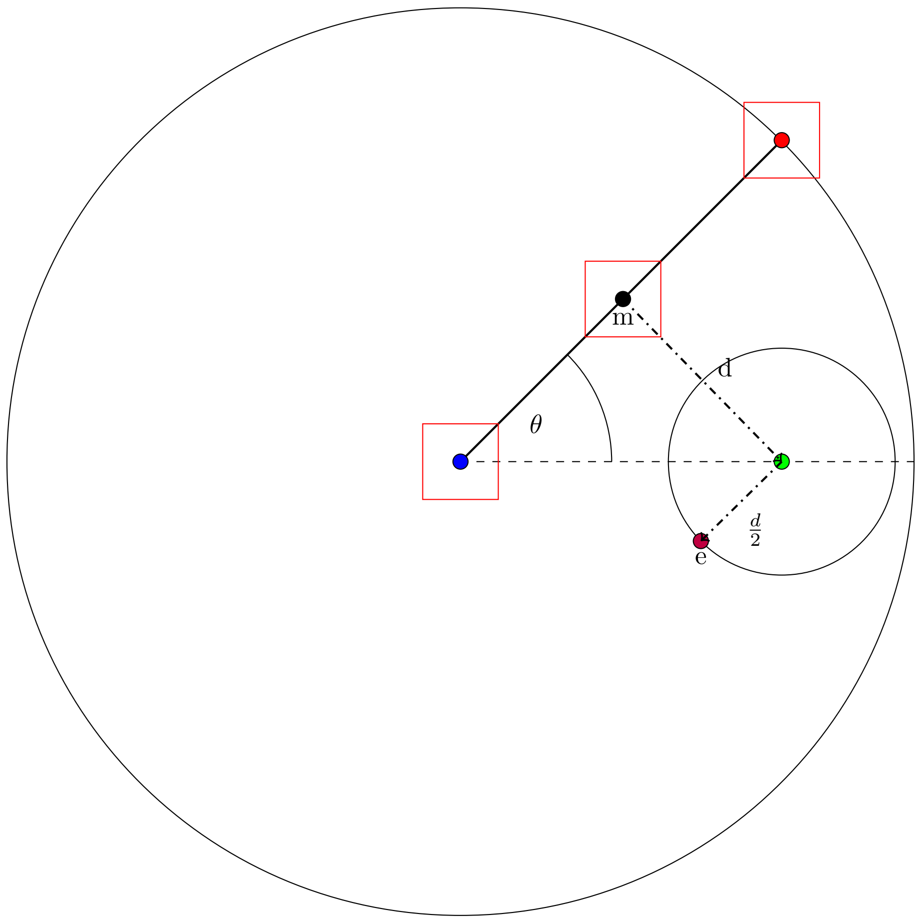


Figure 38: Imaginary circle on c

NOTICE how the diameter of the circle on `c` starting from `e` contains the line in which the units will be placed upon.

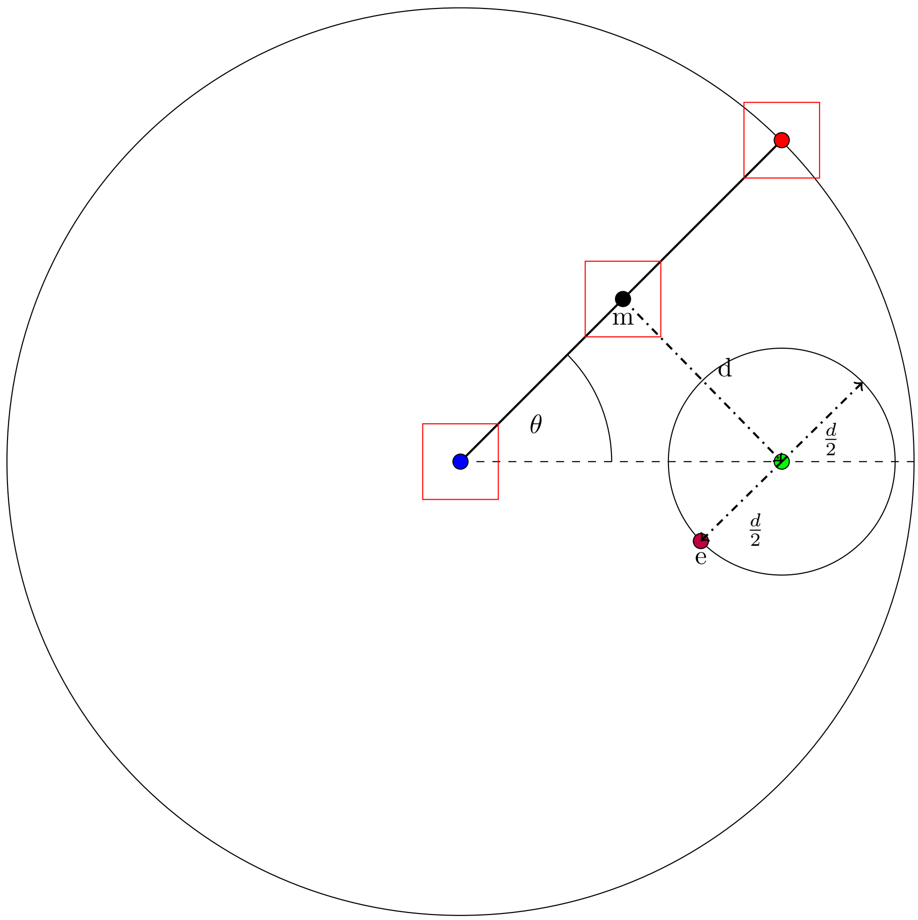


Figure 39: Diameter from e

One can then imagine another circle with center on `e`, and a radius with the diameter of the circle of `c`.

C.2.4 Circle on End point

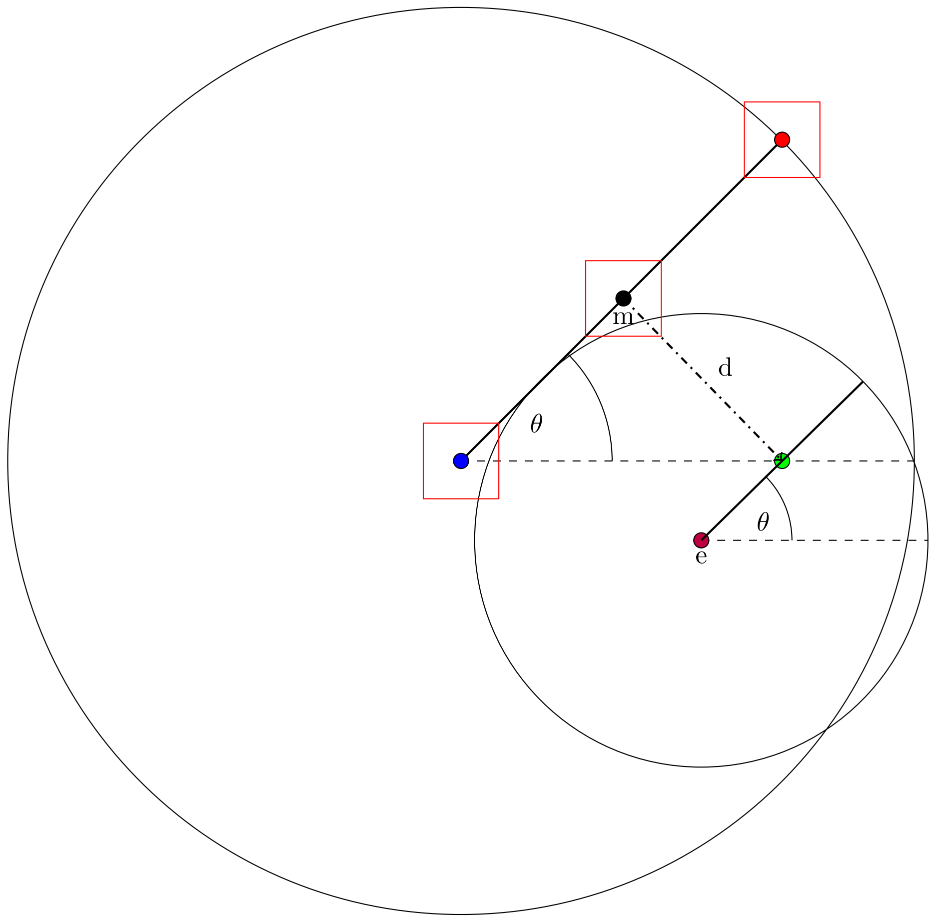


Figure 40: Circle on e

Given that the new radius on `e` has the same angle as the radius from circle `I`, the equation to calculate a position on a line is applied.

$$position = e + d * (\cos(\theta), \sin(\theta)) * i$$

C.2.5 Calculating Positions

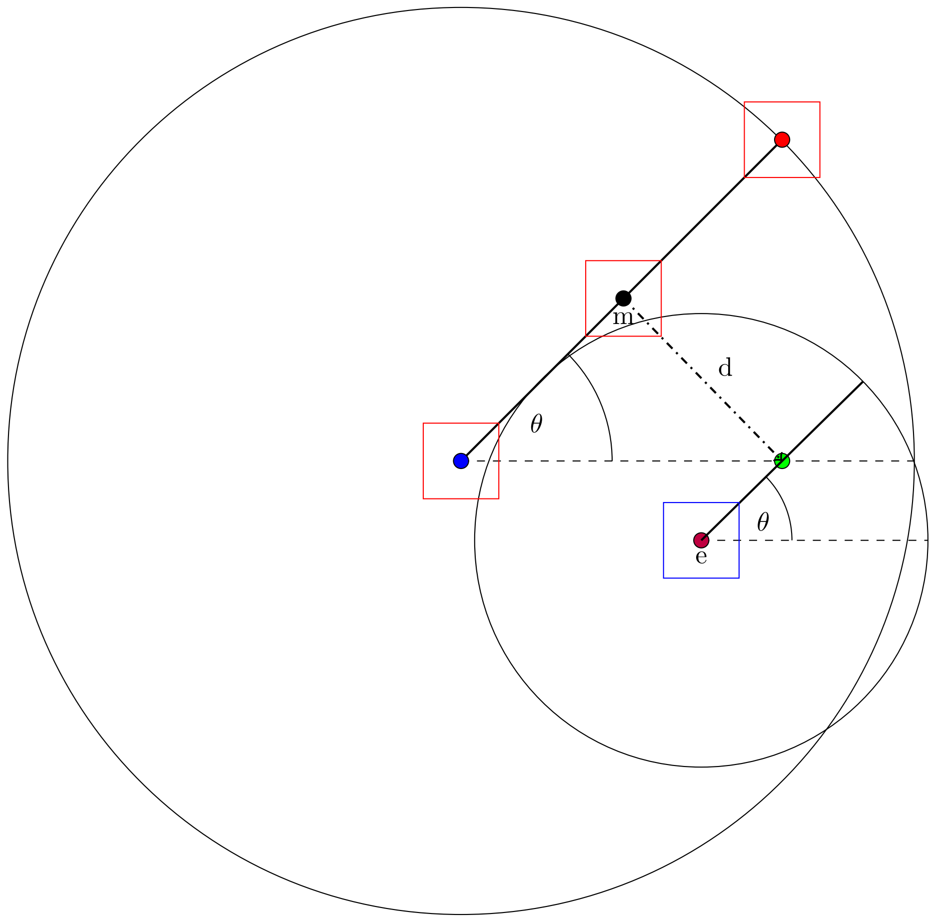


Figure 41: position = e + d * $(\cos(\theta), \sin(\theta))$ * 0

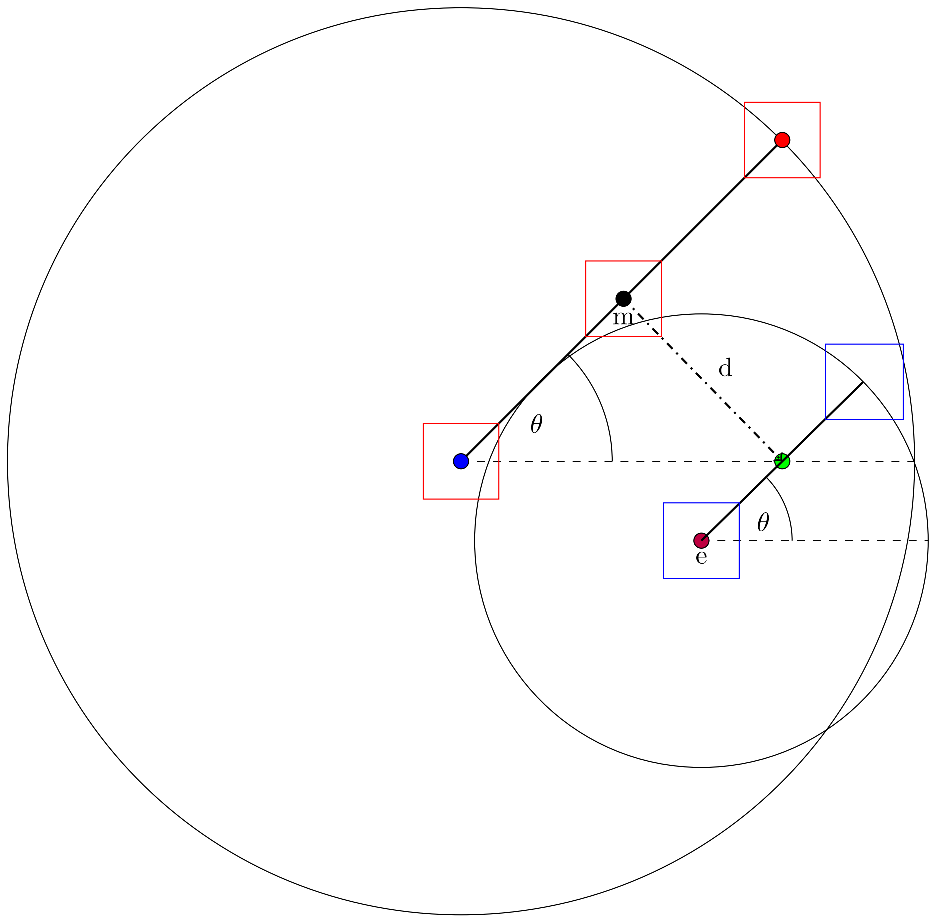


Figure 42: position = e + d * $(\cos(\theta), \sin(\theta))$ * 1

C.3 Pascal's Positions

To calculate the positions in the last row, the following values need to be known.

- mid point
- center
- end point
- remainder units on last row

```
List<Vector> PascalPositions(Vector I,
                             Vector E,
                             float depth,
                             float sizeOfUnit,
                             float gap,
                             float angle,
                             int remainingUnits
                            )
{
    float distBtxtUnit = sizeOfUnit + gap;
    Vector midPoint = I + (E - I)/2;
    Vector centerPoint = midPoint + distBtxtUnit
                    * (sin(angle),-cos(angle))
                    * depth;
    Vector endPoint = centerPoint
                         - ((distBtxtUnit/2) * (remainingUnits-1))
                         * (cos(angle), sin(angle));

    List<Vector> positions = empty;
    for(int currUnit = 0; currUnit < remainingUnits; currUnit++)
    {
        Vector currentPosition = PositionOnRadius(endPoint,
                                                  sizeOfUnit,
                                                  gap,
                                                  angle,
                                                  currUnit
                                                 );
    positions.add(currentPosition);
    }
    return positions;
}
```

C.4 Pascal's Procedure

Since applying Pasca's positions is on the last depth, a check can be implemented that checks whether the current depth is the last and if so apply the Pascal's position.

```
List<Vector> GeneralPascalPositions(Vector I,
                                    Vector E,
                                    float sizeOfUnit,
                                    float gap,
                                    float angle,
                                    int totalFrontUnits,
                                    int maxUnits
                                  )
{
    float radius = Pythagora(I, E);
    int totalDepth = TotalDepth(maxUnits, totalFrontUnits);
    float distBtxtUnits = sizeOfUnit + gap;
    Vector currentI = I;
    int remainder = maxUnits % totalFrontUnits

    List<Vector> positions = empty;

    for(int currDepth = 0; currDepth <= totalDepth; currDepth++)
    {
        if (curDepth == totalDepth) {
            List<Vector> pascalPositions = PascalPositions(I,
                                                           E,
                                                           sizeOfUnit,
                                                           gap,
                                                           angle,
                                                           remainder
                                                          );
            positions.Add(pascalPositions);
            return positions;
        }

        for(int unitInDepth = 0; unitInDepth < totalFrontUnits; unitInDepth++)
        {
            Vector position = PositionOnRadius(I,
                                               sizeOfUnit,
                                               gap,
                                               angle,
                                               unitInDepth
                                             );
            positions.Add(position);
        }
        currentI = currentI + (sin(angle), -cos(angle)) * distBtxtUnits;
    }
    return positions;
}
```

One can optimize the procedure by only applying pascal's positions if there are any remainder units.

```
List<Vector> GeneralPascalPositions(Vector I,
                                    Vector E,
                                    float sizeOfUnit,
                                    float gap,
                                    float angle,
                                    int totalFrontUnits,
                                    int maxUnits
                                  )
{
    float radius = Pythagora(I, E);
    int totalDepth = TotalDepth(maxUnits, totalFrontUnits);
    float distBtxtUnits = sizeOfUnit + gap;
    Vector currentI = I;
    int remainder = maxUnits % totalFrontUnits

    List<Vector> positions = empty;

    for(int currDepth = 0; currDepth <= totalDepth; currDepth++)
    {
        if (curDepth == totalDepth and remainder > 0) {
            List<Vector> pascalPositions = PascalPositions(I,
                                                           E,
                                                           sizeOfUnit,
                                                           gap,
                                                           angle,
                                                           remainder
                                                          );
            positions.Add(pascalPositions);
            return positions;
        }

        for(int unitInDepth = 0; unitInDepth < totalFrontUnits; unitInDepth++)
        {
            Vector position = PositionOnRadius(I,
                                               sizeOfUnit,
                                               gap,
                                               angle,
                                               unitInDepth
                                             );
            positions.Add(position);
        }
        currentI = currentI + (sin(angle), -cos(angle)) * distBtxtUnits;
    }
    return positions;
}
```

C.5 Sliding Row - Issue

When implementing the Pascal as is, one will encounter the last row to be sliding while dragging, even though the formation will be stationary. This is due to the End value that is sent to the grid formation.

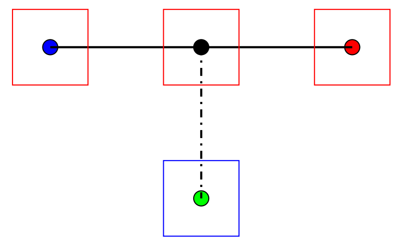


Figure 43: All units fit in line

When the radius fits exactly the amount of units, the last row will behave as desired.

But when the radius calculated upon dragging does not fit the exact number of units and is hanging to the right, the last row will move because the radius has increased, and thus the mid point, `m` has moved.

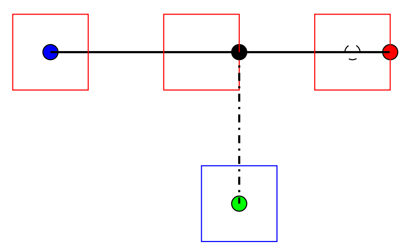


Figure 44: Shift of m

The solution is to pass the position of the last unit onto `Pascal's Positions`

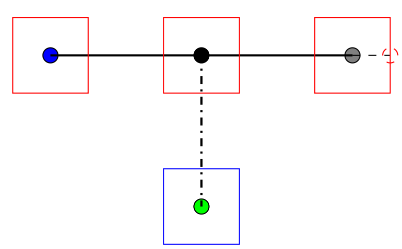


Figure 45: Pass position of last unit

Acquiring the last position of the unit on the radius:

$$end = I + ((sizeOfUnit + gap) * (\cos(\theta), \sin(\theta)) * totalFrontUnits - 1)$$

NOTE the index for the last unit on radius is: $totalFrontUnits - 1$.

C.6 Applying fix

The last position on the line must be calculated and passed onto Pascal's positions.

```
List<Vector> GeneralPascalPositions(Vector I,
                                    Vector E,
                                    float sizeOfUnit,
                                    float gap,
                                    float angle,
                                    int totalFrontUnits,
                                    int maxUnits
                                   )
{
    float radius = Pythagora(I, E);
    int totalDepth = TotalDepth(maxUnits, totalFrontUnits);
    float distBtxtUnits = sizeOfUnit + gap;
    Vector currentI = I;
    int remainder = maxUnits % totalFrontUnits;
    List<Vector> positions = empty;
    for(int currDepth = 0; currDepth <= totalDepth; currDepth)
    {
        if (currDepth == totalDepth and remainder > 0) {
            Vector end = positionOnRadius(currentI,
                                          sizeOfUnit,
                                          gap,
                                          angle,
                                          totalFrontUnits-1
                                       );
            List<Vector> pascalPositions = PascalPositions(currentI,
                                                           end,
                                                           sizeOfUnit,
                                                           gap,
                                                           angle,
                                                           remainder
                                                       );
            positions.Add(pascalPositions);
            return positions;
        }

        for(int unitInDepth = 0; unitInDepth < totalFrontUnits; unitInDepth++)
        {
            Vector position = PositionOnRadius(currentI,
                                               sizeOfUnit,
                                               gap,
                                               angle,
                                               unitInDepth
                                            );
            positions.Add(position);
        }
        currentI = currentI + (sin(angle), -cos(angle)) * distBtxtUnits;
    }
    return positions;
}
```

D Final Equations and Procedures

D.1 Equations

D.1.1 Base Equations

$$
\begin{aligned}
totalUnits &= \frac{lengthOfRadius}{sizeOfUnit + gap} + 1 \\
\theta &= \tan^{-1}(\frac{opposite}{adjacent}) \\
positionOnRadius &= I + ((sizeOfUnit + gap) * (\cos(\theta), \sin(\theta)) * i) \\
positionOnLine &= I + (d * i) \\
depthUnitPosition &= F + ((\sin(\theta), -\cos(\theta)) * d) \\
totalDepth &= (\frac{maxUnits}{unitsOnRadius} - 1) + (maxUnits\%unitsOnRadius == 0?0 : 1
\end{aligned}
$$

D.1.2 General Position Equation

$$I_{d+1} = I_d + d * (\sin(\theta), -\cos(\theta))$$

D.1.3 Pascal equations

$$
\begin{aligned}
m &= I + \frac{E - I}{2} \\
c &= m + distBtxt * (\sin(\theta), -\cos(\theta)) * depth \\
e &= c - \frac{d}{2} * (remainder - 1) * (\cos(\theta), \sin(\theta)) \\
endUnit &= I + distBtxt * (\cos(\theta), \sin(\theta)) * (totalFrontUnits - 1)
\end{aligned}
$$

D.2 Procedures

D.2.1 Main

```
GridFormation_V2:
{
bool dragging = false;
MouseButton button = RIGHT;

Vector I = Vector(0,0);
Vector E = Vector(0,0);
Vector C = Vector(0,0);

float sizeOfUnit = s;
float gap = g;
int maxUnits = totalUnits;
List<Vector> positions = empty;

loop {
    dragging = Dragging(I, E, button);

    if dragging {
        C = mousePosition();
    Vector position = GridFormationPositions(I, C, sizeOfUnit, gap, maxUnits);
    }
}
}
```

D.2.2 Position Calculaters

```
List<Vector> GridFormationPositions(Vector I,
                                    Vector E,
                                    float sizeOfUnit,
                                    float gap,
                                    int maxUnits)
{
    float radius = Pythagora(I, E);
    float distBtxt = sizeOfUnit + gap;
    int totalFrontUnits = (radius/distBtxt) + 1;
    float angle = arctan(E-I);

    List<Vector> frontPositions = empty;
    List<Vector> depthPositions = empty;

    List<Vector> gridPositions = empty;

    gridPositions = PositionsOnRadius(
                           I,
                           sizeOfUnit,
                           gap,
                           angle,
                           totalFrontUnits
                    );
    gridPositions = DepthPositions(
                          gridPositions,
                          distBtxt,
                          totalFrontUnits,
                          maxUnits,
                          angle
                     );

    return gridPositions;
}
```

```
List<Vector> PositionsOnRadius(Vector I,
                               Vector E,
                               float sizeOfUnit,
                               float gap,
                               float angle,
                               int totalFrontUnits)
{
    List<Vector> positions = empty;

    for(int currentUnit = 0; currentUnit < totalFrontUnits; currentUnit++) {
        Vector currentPosition = positionOnRadius(
                                      I,
                                      sizeOfUnit,
                                      gap,
                                      angle,
                                      currentUnit
                                );
    positions.add(currentPosition);
    }

    return positions;
}
```

```
List<Vector> DepthPositions(List<Vector> Ps,
                            float distBtxt,
                            int totalFrontUnits,
                            int maxUnits,
                            float angle)
{

    for(int currentDepthUnit = totalFrontUnits, currentFrontUnitIndex = 0;
        currentDepthUnit < maxUnits;
        currentDepthUnit++, currentFrontUnitIndex++
       )
    {
        Vector currentFrontUnit = positions[currentFrontUnitIndex];
        Vector depthPosition = depthUnitPosition(currentFrontUnit,
                                              angle,
                                                  distBtxt
                               );
        positions.add(depthPosition);
    }
    return positions;
}
```

D.2.3 General Position calculator

```
GeneralPositions:
{
    List<Vector> GeneralPositions(Vector I,
                                  Vector E,
                                  float sizeOfUnit,
                                  float gap,
                                  float angle,
                                  int totalFrontUnits,
                                  int maxUnits
                                 )
    {
        float radius = Pythagora(I, E);
        int totalDepth = TotalDepth(maxUnits, totalFrontUnits);
        float distBtxt = sizeOfUnit + gap;

        Vector currentI = I;

        List<Vector> positions = empty;

        for (int curDepth = 0, currentUnit = 0;
             curDepth <= totalDepth;
             curDepth++)
        {
            for(unitInDepth = 0;
                unitInDepth < totalFrontUnits;
                currentUnit++, unitInDepth++
               )
            {
                if currentUnit > maxUnits {
                    return positions;
                }

                Vector position = positionOnRadius(I,
                                                  sizeOfUnit,
                                                  gap,
                                                  angle,
                                                  unitInDepth
                                                 );
                positions.add(positions)
            }
            currentI = currentI + (distBtxt * (sin(angle), -cos(angle))
        }
        return positions;
    }
}
```

D.2.4 General + Pascal Positions

```
List<Vector> GeneralPascalPositions(Vector I,
                                    Vector E,
                                    float sizeOfUnit,
                                    float gap,
                                    float angle,
                                    int totalFrontUnits,
                                    int maxUnits
                                   )
{
    float radius = Pythagora(I, E);
    int totalDepth = TotalDepth(maxUnits, totalFrontUnits);
    float distBtxtUnits = sizeOfUnit + gap;
    Vector currentI = I;
    int remainder = maxUnits % totalFrontUnits;
    List<Vector> positions = empty;
    for(int currDepth = 0; currDepth <= totalDepth; currDepth)
    {
        if (currDepth == totalDepth and remainder > 0) {
            Vector end = positionOnRadius(currentI,
                                          sizeOfUnit,
                                          gap,
                                          angle,
                                          totalFrontUnits-1
                                       );
            List<Vector> pascalPositions = PascalPositions(currentI,
                                                           end,
                                                           sizeOfUnit,
                                                           gap,
                                                           angle,
                                                           remainder
                                                       );
            positions.Add(pascalPositions);
            return positions;
        }

        for(int unitInDepth = 0; unitInDepth < totalFrontUnits; unitInDepth++)
        {
            Vector position = PositionOnRadius(currentI,
                                               sizeOfUnit,
                                               gap,
                                               angle,
                                               unitInDepth
                                           );
            positions.Add(position);
        }
        currentI = currentI + (sin(angle), -cos(angle)) * distBtxtUnits;
    }
    return positions;
```

```
}
```

D.2.5 Pascal Positions

```
List<Vector> PascalPositions(Vector I,
                             Vector E,
                             float depth,
                             float sizeOfUnit,
                             float gap,
                             float angle,
                             int remainingUnits
                            )
{
    float distBtxtUnit = sizeOfUnit + gap;
    Vector midPoint = I + (E - I)/2;
    Vector centerPoint = midPoint + distBtxtUnit
                     * (sin(angle),-cos(angle))
                     * depth;
    Vector endPoint = centerPoint
                          - ((distBtxtUnit/2) * (remainingUnits-1))
                          * (cos(angle), sin(angle));

    List<Vector> positions = empty;
    for(int currUnit = 0; currUnit < remainingUnits; currUnit++)
    {
        Vector currentPosition = PositionOnRadius(endPoint,
                                                  sizeOfUnit,
                                                  gap,
                                                  angle,
                                                  currUnit
                                                 );
    positions.add(currentPosition);
    }
    return positions;
}
```