Grid Formation Behaviour

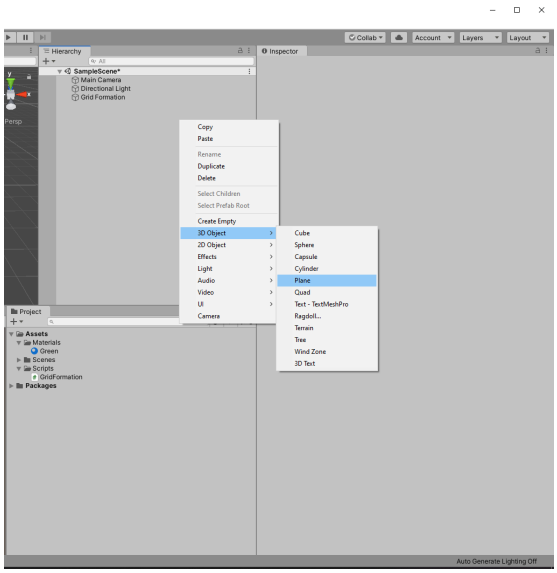
In Unity

Jonas Munoz

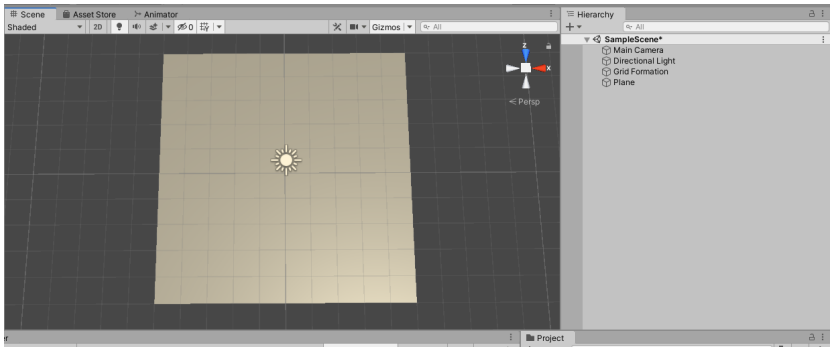
Contents

1 Setup

1. Create **GridFormation** Script
2. Create a plane at position 0.



(a) Select plane



(b) Create plane

3. Have *Main Camera* selected and align Camera with Editor camera to view from top

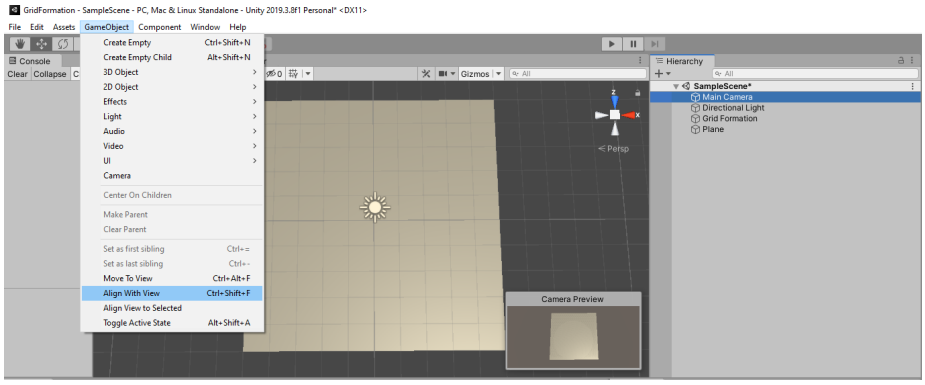
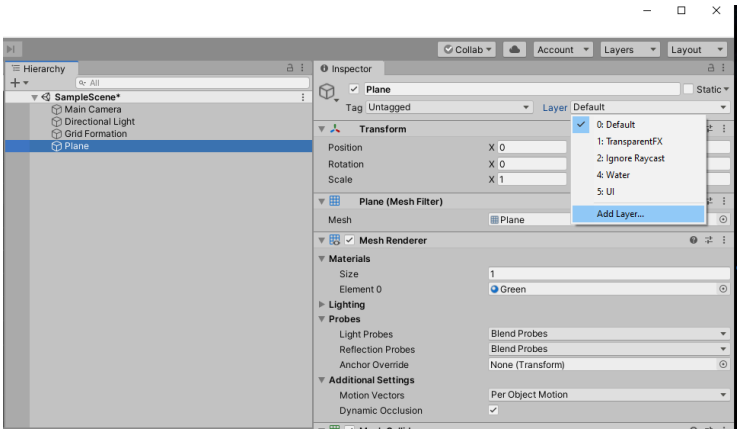
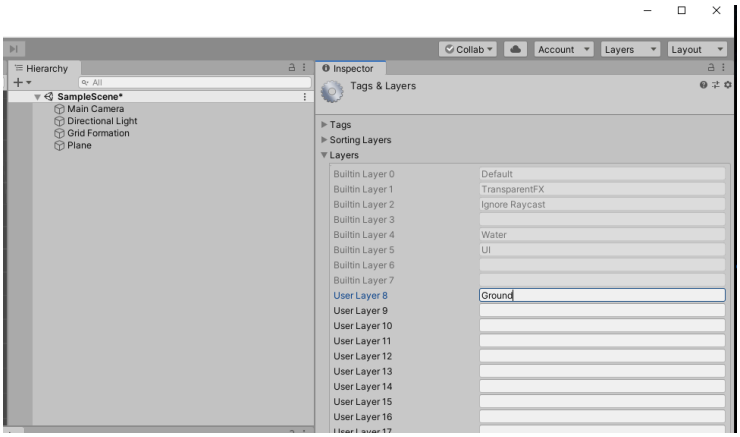


Figure 2: Align Camera

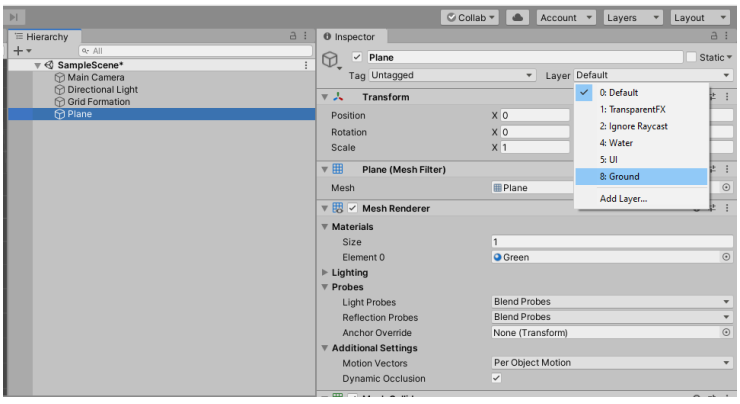
4. Click on Plane and add Ground Layer Mask



(a) Ground Layer



(b) Adding Layer



(c) Apply Ground mask

5. Add Green material

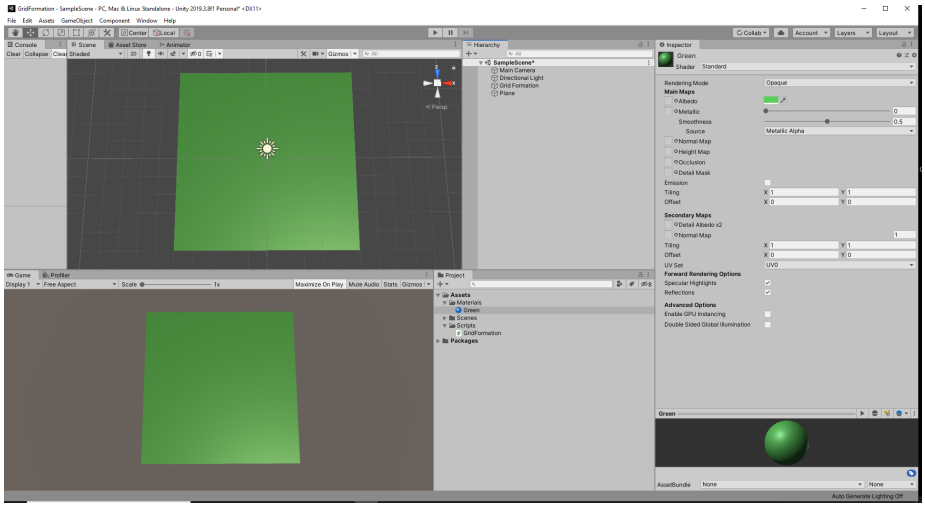


Figure 4: Green Material

6. Create Empty object, name it **GridFormation** and attach the script.

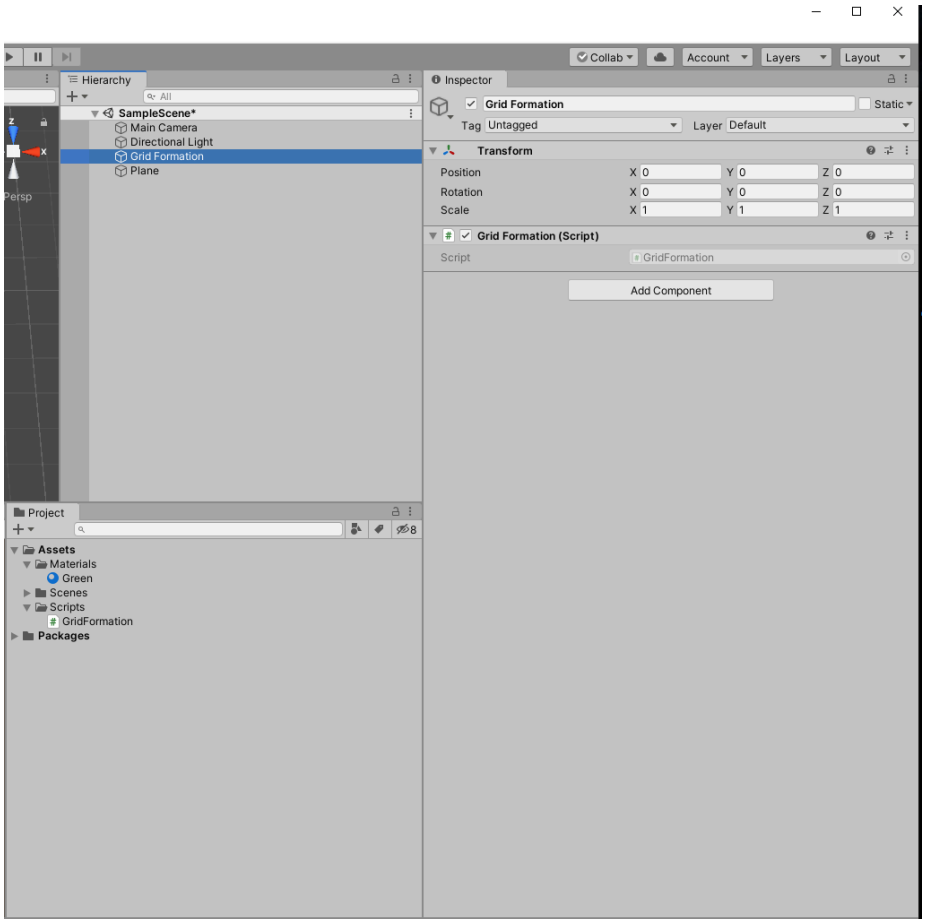


Figure 5: Grid Formation Empty object

2 Dragging

In Grid Formation, the dragging mechanism is implemented as follows:

```
1  public class GridFormation : Monobehaviour
2  {
3      private bool dragging;
4      int rightMButton = 1;
5
6      private void Dragging() {
7          if (Input.GetMouseButtonDown(rightMButton))
8          {
9              dragging = true;
10         }
11         if (Input.GetMouseButtonUp(rightMButton))
12         {
13             dragging = false;
14         }
15     }
16 }
```

Implementing the drag mechanic in Update:

```
1  public class GridFormation : MonoBehaviour
2  {
3      private bool dragging;
4      int rightMButton = 1;

6      private void Update()
7      {
8          Dragging();
9          if (dragging)
10         {
11             // Do stuff
12         }
13     }

14     ...
15 }
```

3 Imginary Line

The dragging mechanism requires the position at which the mouse was initially clicked. To do so, there must be a *LayerMask* and a *Camera* in **Grid Formation**:

```
public class GridFormation : MonoBehaviour
{

  public Camera mainCam;
  public LayerMask groundMask;


  private bool dragging;
  ...
}
```

1. Drag *Main Camera* onto the `mainCam` slot
2. Select *Ground* to be in Layer Mask

Then Initialize I, E variables:

```
public class GridFormation : MonoBehavior {
  ...
  private bool dragging;

  Vector3 I;
  Vector3 E;

  ...
}
```

As the mouse clicked coordinate will be used multiple times, it is best to create its own function that returns the clicked coordinate.

```
public class GridFormation : MonoBehavior {
    ...

  private Vector3 ClickedCoord() {
    Ray ray = mainCam.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;

    Vector3 clickedCoord = Vector3.zero;

    if (Physics.Raycast(ray, out hit, 100.0f, groundMask)) {
      clickedCoord = hit.point;
    }
    return clickedCoord;
 }
```

```
14 }
```

Now add the coordinates of both **I** and **E** in the `Dragging` method.

```
1  private void Dragging()
2      {
3          if (Input.GetMouseButtonDown(rightMButton))
4          {
5              dragging = true;

6              I = ClickedCoord();

7          }
8          if (Input.GetMouseButtonUp(rightMButton))
9          {
10             dragging = false;

11             E = ClickedCoord();

12         }
13     }
```

4 Length of Imaginary Line

A static class will be created to hold both equations and procedures.

4.1 GFequation

Create a new script called **GFequation** and make it static.

```
public static class GFequation
{

}
```

Add the *TotalUnits* method:

```
public static GFequation {

  public static int TotalUnits(float lengthOfLine,
                               float sizeOfUnit)
  {
      int units = (int) (lengthOfLine / sizeOfUnit);
      return units + 1;
  }

}
```

The **int** cast is used to convert the float to int as there will never be half a unit unless it is dead.
Create a new script called **GFprocedure** and make it static.

```
public static class GFprocedure
{

}
```

Add the *PositionOnLine* method:

```

public static class GFprocedure {

    public static Vector3 PositionOnLine(Vector3 l,
                                         float d,
                                         int i)
    {
        float offset = d * i;

        Vector3 position = l;
        position.x = l.x + offset;

        return position;
    }
```

```
}
```

As the line only moves along the X-Axis, onlt the x value is affected starting from I.

4.2 GFprocedure

Create a new script called **GFprocedure** and make it static.

```
public static class GFprocedure
{

}
```

Add the *PositionsOnLine* method:

```
public static class GFprocedure {

  public static List<Vector3> PositionsOnLine(float distBtxtUnits,
                                              int totalUnits,
                                              Vector3 I)
  {
    List<Vector3> positions = new List<Vector3>();

    for (int currIndexUnit = 0;
         currIndexUnit < totalUnits;
         currIndexUnit++)
      {
        float offset = distBtxtUnits * currIndexUnit;
        Vector3 offsetVec = new Vector3(offset, 0, 0);
        Vector3 curPos = I + offsetVec;

        positions.Add(curPos);
      }
    return positions;
  }

}
```

4.3 Integrating

Add the following variables onto **GridFormation**

```
public static class GridFormation {
  ...
  Vector3 E;

  [SerializeField]
  float gap;
  [SerializeField]
  float sizeOfUnit;

  ...
}
```

4.3.1 Dragging

```
private void Dragging()
  {
    if (Input.GetMouseButtonDown(rightMButton))
    {
      dragging = true;
      I = ClickedCoord();
    }

    if (Input.GetMouseButtonUp(rightMButton))
    {
      dragging = false;
      E = ClickedCoord();

      float lengthOfLine = Vector3.Magnitude(E - I);
      float distBtxtUnit = sizeOfUnit + gap;
      int totalUnits = GFequation.TotalUnits(lengthOfLine,
                                             sizeOfUnit);

      List<Vector3> positions =
                          GFprocedure.PositionsOnLine(
                                        distBtxtUnit,
                                        totalUnits,
                                        I);

        }
    }
```

This only gets the positions, does not display any units on screen. A prefab must be created and added onto the Grid Formation script, which would then instantiate the prefabs onto their corresponding positions.

```
public classGridFormation : MonoBehaviour {
    ...

    public GameObject prefab

    ...
}
```

4.3.2 Creating Prefab

1. Create cube
2. Create folder called **Prefabs**
3. Drag cube into Prefabs folder
4. Drag prefab cube onto **GridFormation**'s prefab variable

4.3.3 Place Units

Displaying the units requires a method that instantiates the prefabs given a position in the game:
Iterating through all positions and giving them to an instantiated prefab.

```
public class GridFormation {
    ...

    private void PlaceUnits(List<Vector3> positions)
    {
        for(int i = 0; i < positions.Count; i++)
        {
            GameObject unit = Instantiate(prefab);
            unit.transform.position = positions[i];
        }
    }

}
```

Units are placed after the positions have been calculated.

```
private void Dragging() {
    ...
    List<Vector3> positions = GFprocedure.PositionsOnLine(
    distBtxtUnit, totalUnits, 1);

    PlaceUnits(positions);

}
```

4.4 Going both directions

Currently the line only goes from left to right. This is because the offset from I is always positive. To make it negative, the **distBtxtUnit** must be negative when the X- value is negative.

```
1  private void Dragging()
2      {
3          if (Input.GetMouseButtonDown(rightMButton))
4          {
5              dragging = true;
6              I = ClickedCoord();
7          }
8
9          if (Input.GetMouseButtonUp(rightMButton))
10         {
11             dragging = false;
12             E = ClickedCoord();
13             float lengthOfLine = Vector3.Magnitude(E - I);
14             float distBtxtUnit = sizeOfUnit + gap;

15             if (E.x < I.x)
16             {
17                 distBtxtUnit = -distBtxtUnit;
18             }

19
20             int totalUnits = GFequation.TotalUnits(
       lengthOfLine, sizeOfUnit, gap);
21
22             List<Vector3> positions = GFprocedure.
       PositionsOnLine(distBtxtUnit, totalUnits, I);
23             PlaceUnits(positions);
24
25         }
26     }
```

5 Applying Unit Circle

5.1 Setup

Add the equation **PositionOnRadius** inside `GFequation`.

```
1  public static class GFequation {
2    ...

3    public static Vector3 PositionOnRadius(Vector3 l,
4                                           float sizeOfUnit,
5                                           float gap,
6                                           float angle,
7                                           int index)
8    {
9        float distBtxtUnit = sizeOfUnit + gap;
10       Vector3 offset = new Vector3(distBtxtUnit * Mathf.Cos(
     angle * Mathf.Deg2Rad), 0f, distBtxtUnit * Mathf.Sin(angle
      * Mathf.Deg2Rad));
11
12       Vector3 position = l + (offset * index);
13
14       return position;
15   }

16 }
```

Add the procedure **PositionsOnRadius** inside `GFprocedure`.

```
public static class GFprocedure {
  ...

  public static List<Vector3> PositionsOnRadius(Vector3 I,
                                                Vector3 E,
                                          float sizeOfUnit,
                                                float gap)
    {
        float radius = Vector3.Magnitude(E - I);
        int totalUnits = GFequation.TotalUnits(radius,
    sizeOfUnit, gap);
        float angleRad = Mathf.Atan((E.x - I.x) / (E.z - I.z))
    ;
        float angle = angleRad * Mathf.Rad2Deg;

        List<Vector3> positions = new List<Vector3>();

        for (int i = 0; i < totalUnits; i++)
        {
            Vector3 currentPosition =
                        GFequation.PositionOnRadius(I,
    sizeOfUnit, gap, angle, i);
            positions.Add(currentPosition);
        }

        return positions;
    }

}
```

5.2 Guide

At the moment, there is no way to orientate the Unit Circle onto an Axis. A guide will be placed at I upon dragging. This guide will have the Z-axis as its base.

5.2.1 Creating the Guide

1. Create a cube and place it at the center. Name it *Guide*.

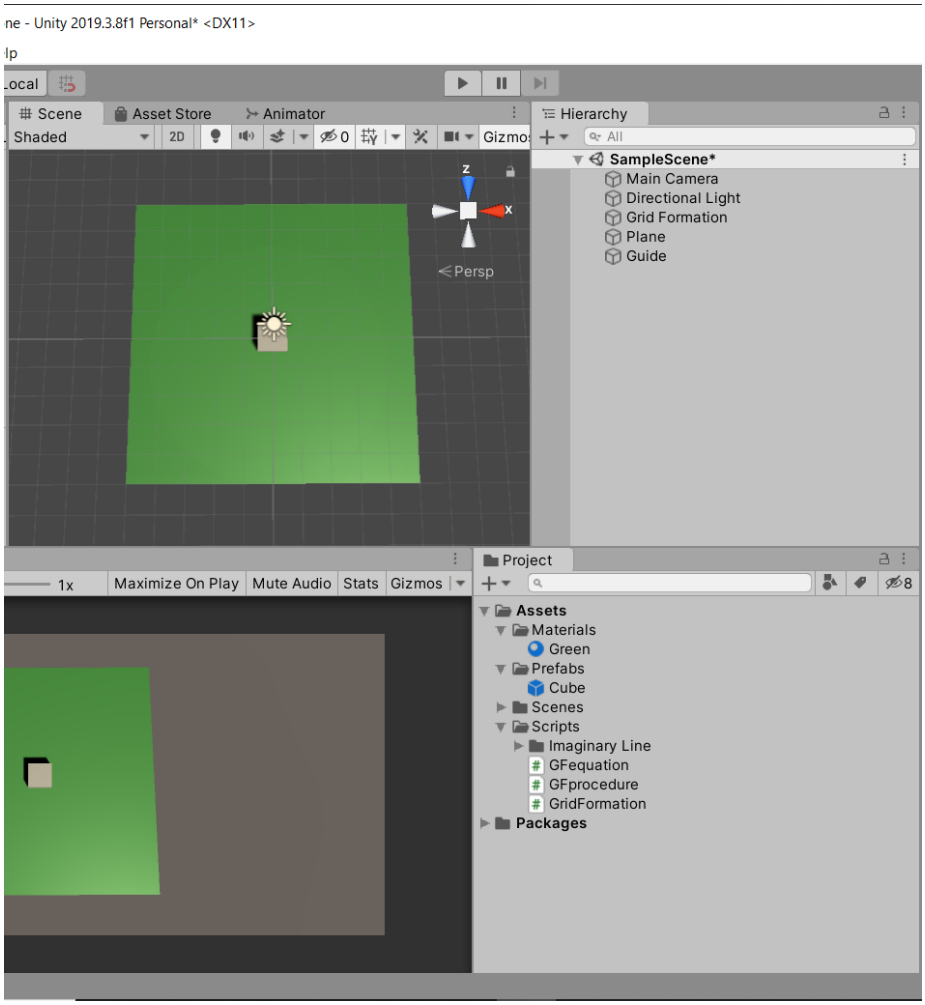


Figure 6: Guide

2. Scale it to the following dimensions: X:0.2, Y: 0.2, Z:0.2

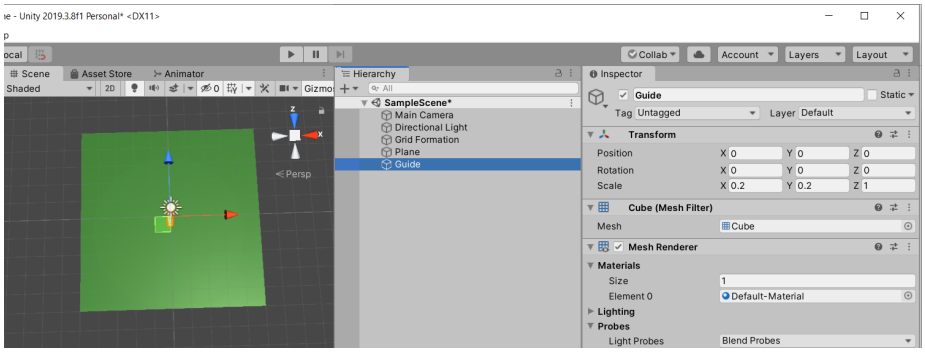


Figure 7: Scaled

3. Make a public variable in **GridFormation** calling it guide:

```
1        public class GridFormation : MonoBehaviour {
2            ....
3            public GameObject prefab;
4
5            public GameObject guide;
6
7            private bool dragging;
8            ....
9
```

4. Drag guide onto `prefab`

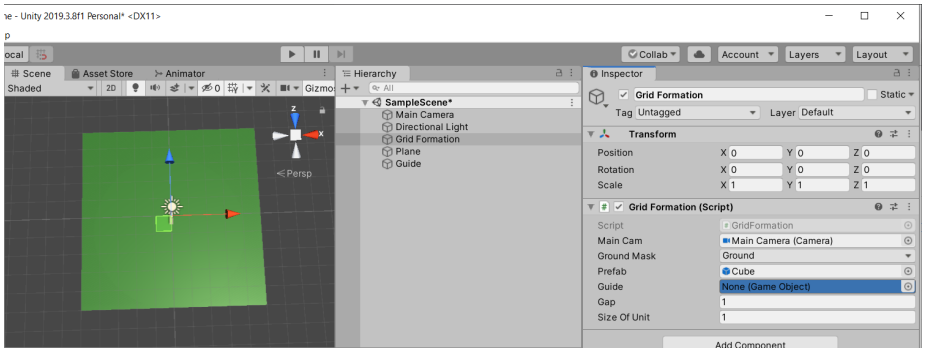


Figure 8: Add to variable

5.2.2 Rotating the Guide

In **GridFormation**, add the rotating method that rotates the guide according to the mouse location.

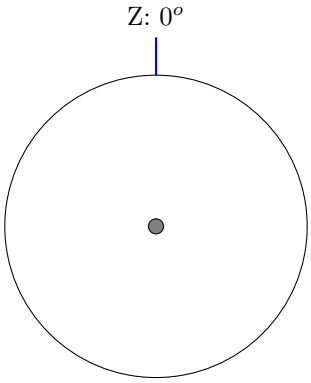
```
1  private void RotateGuide() {
2      Ray cameraRay = mainCam.ScreenPointToRay(Input.
       mousePosition);
3          RaycastHit hit;
4
5          //  Coordinate
6          if (Physics.Raycast(cameraRay, out hit, 100.0f,
       groundMask))
7          {
8
9              Vector3 pointToLook = cameraRay.GetPoint(hit.
       distance);
10             // Rotate parent Object
11             guide.transform.LookAt(new Vector3(0f, transform.
       position.y, pointToLook.z));
12
13
14         }
15 }
```

Have the Method be called inside the `dragging` if statement

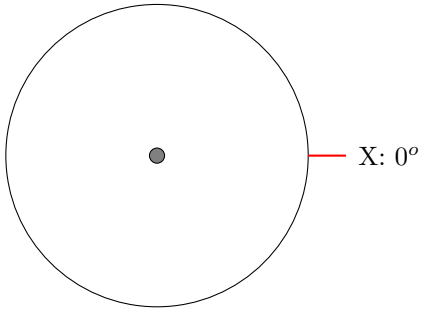
```
1  public class GridFormation : MonoBehaviour {
2    void Update() {
3      ...
4      Dragging();
5          if (dragging)
6          {
7              RotateGuide();
8          }
9      ...
10   }
11
12 }
```

5.2.3 Aligning the Z-axis

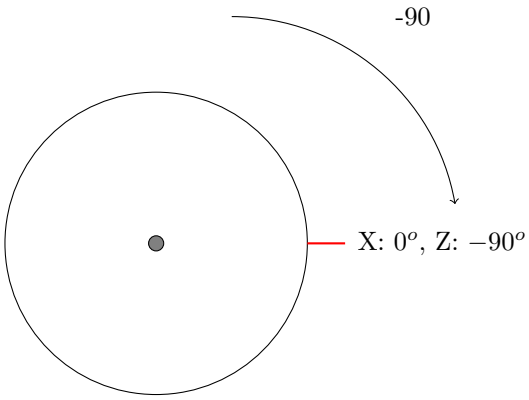
Currently Unity's unit circle is aligned with the Z-axis due to our guide.



The guide must be aligned with the X-axis according to the Unit Circle.



To do so, one can subtact 90 from the guide angle to shift the angles by -90.

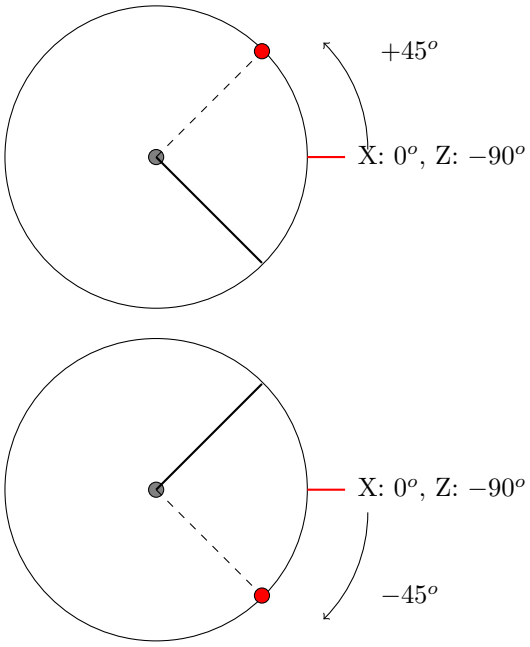


```
1  private void Dragging()
2    {
3       ...
4       if (Input.GetMouseButtonUp(rightMButton))
5       {
6          ...

7          float angledAligned =
8                      guide.transform.rotation.eulerAngles.y - 90;

9          List<Vector3> positions = GFprocedure.PositionsOnRadius(
       I, E, sizeOfUnit, gap, angledAligned);
10         int totalUnitsPlaced = PlaceUnits(positions);
11      }
12 }
```

5.2.4 Inverting the Z value

The X-Axis is now aligned, but the Z-axis is inverting the Unit Circle.

When the mouse is moved up by θ the units go down, and vice versa.



Inverse the Z value to at `PositionOnRadius`properly align the units.

```
1  public static Vector3 PositionOnRadius(Vector3 l,
2                                         float sizeOfUnit,
3                                         float gap,
4                                         float angle,
5                                         int index)
6    {
7      float distBtxtUnit = sizeOfUnit + gap;
8      Vector3 offset =
9              new Vector3(distBtxtUnit *
10                         Mathf.Cos(angle * Mathf.Deg2Rad),
11                         0f,
12                         -distBtxtUnit *
13                               Mathf.Sin(angle * Mathf.Deg2Rad));
14
15     Vector3 position = l + (offset * index);
16
17     return position;
18   }
```

6 Dynamic Unit Positions

Before continuing, there is a need to refactor the code in order to ahieve the *Dynamic Unit Positions* with ease.
Instead of the positions being placed upon finishing dragging, positions will now be calculated while dragging is occuring. Meaning functionalities form `Dragging` will be placed in the `Update` function under `if (dragging) ...`
Below is the refactored code.

6.1 Refactoring

`GridFormation`

```
1  public class GridFormation : MonoBehavior {
2      Vector3 I;
3      Vector3 E;
4      Vector3 C;
5
6      private void Update()
7      {
8          Dragging(I, C);
9          if (dragging)
10         {
11             RotateGuide();
12             C = ClickedCoord();
13
14             float lengthOfLine = Vector3.Magnitude(C - I);
15             Debug.Log(lengthOfLine);
16             float distBtxtUnit = sizeOfUnit + gap;
17
18             int totalUnits = GFequation.TotalUnits(
       lengthOfLine, sizeOfUnit, gap);
19             float angledAligned = guide.transform.rotation.
       eulerAngles.y - 90;
20             List<Vector3> positions = GFprocedure.
       PositionsOnRadius(I, C, sizeOfUnit, gap, angledAligned);
21             int totalUnitsPlaced = PlaceUnits(positions);
22
23         }
24     }
25
26     /** Dragging Mechanisms**/
27     private void Dragging(Vector3 I, Vector3 E)
28     {
29         if (Input.GetMouseButtonDown(rightMButton))
30         {
31             dragging = true;
32             this.I = ClickedCoord();
33             guide.transform.localPosition = this.I;
```

```
34          }
35
36          if (Input.GetMouseButtonUp(rightMButton))
37          {
38              dragging = false;
39              this.E = ClickedCoord();
40          }
41      }
42  }
```

Playing around will now draw a circle.

6.1.1 Placing on New Radius

Currently units are being placed when the mouse is stationary. To fix this, a check will be placed that holds the current angle and compares it with the new angle. As well as checking if a change in radius occurs

```
1  if (angle != newAngle || radius != newRadius) {
2      angle = newAngle;
3      radius = newRadius;
4      // Calculate Units
5  }
```

Create a variable in `GridFormation`:

```
1  public class GridFormation : MonoBehaviour {
2  ...
3  Vector3 C;

4  float angle = 0;
5  float radius = 0;

1  ...
2  }
```

Implement the check inside `update`

```
1  public class GridFormation : MonoBehaviour {
2    void Update() {
3      float angledAligned = guide.transform.rotation.eulerAngles.y - 90;
4      float newRadius = Vector3.Magnitude(C - I);

5      if (angle != angledAligned || radius != newRadius)
6      {

7          angle = angledAligned;
8          radius = newRadius;
9
10         float distBtxtUnit = sizeOfUnit + gap;
11
```

```
12          int totalUnits = GFequation.TotalUnits(radius,
        sizeOfUnit, gap);
13
14          List<Vector3> positions = GFprocedure.
        PositionsOnRadius(I, C, sizeOfUnit, gap, angle);
15          int totalUnitsPlaced = PlaceUnits(positions);

16      }

17    }
18  }
```

7 Fixed Number of Units

Storing certain number of units to be displayed:

```
1   public class GridFormation : MonoBehaviour {
2   ...

3   [SerializeField]
4   private int maxUnits;
5   List<Transform> units;
6
7     void Start() {
8       units  = new List<Transform>
9     }

10  ...
11  }
```

Have `PositionsOnRadius` stop calculating positions once it reaches the *max units*.

```
1   public static List<Vector3> PositionsOnRadius(Vector3 I,
2                                                  Vector3 E,
3                                                  float sizeOfUnit,
4                                                  float gap,
5                                                  float angle,
6                                                  int maxUnits)
7     {
8       ...
9       for (int i = 0; i < totalUnits; i++)
10      {

11          if (i >= maxUnits)
12          {
13              return positions;
14          }
```

```
15          Vector3 currentPosition = GFequation.PositionOnRadius(
    I, sizeOfUnit, gap, angle, i);
16          positions.Add(currentPosition);
17      }
18
19      return positions;
20  }
```

Have a method that fills the number of available positions based on the *maxUnits* value given.

```
1  public class GridFormation : MonoBehaviour {
2    ...

3    private void FillUnits(int amount)
4    {
5      for(int i = 0; i < amount; i++)
6      {
7          GameObject unit = Instantiate(prefab);
8          unit.SetActive(false);
9          units.Add(unit.transform);
10     }
11   }

12 }
```

Call this method upon starting the `GridFormation`

```
1  public class GridFormation : MonoBehaviour {
2    ...
3    [SerializeField]
4    private int maxUnits;
5    List<Transform> units;
6
7    void Start() {
8      units  = new List<Transform>

9      FillUnits(maxUnits);
10   }

11 ...
12 }
```

Refactor `PlaceUnits` to use the given units in **units** list.

```
1      private int PlaceUnits(List<Vector3> positions)
2      {
3        int unitIndex = 0;
4        for (int i = 0; i < positions.Count; i++)   // use
     positions count for safety
5        {
```

```
6             Transform unit = units[i];
7
8             unit.position = positions[i];
9             unit.gameObject.SetActive(true);
10            unitIndex++;
11        }
12        return unitIndex;
13    }
```

7.1 Hanging Units

When the radius can not accomodate all units, some units will be left hanging in their original position.

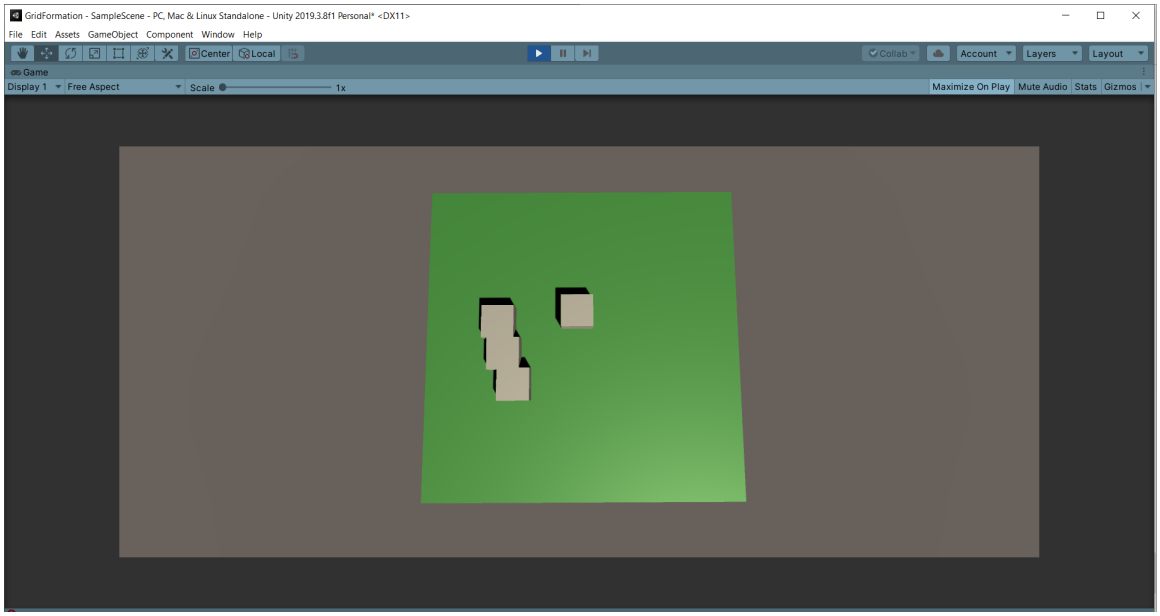


Figure 9: Hanging Unit

One way to fix this is to have a method that deactivates the hanging units.

```
1   public class GridFormation : MonoBehaviour {
2     ...
3     private void DeactivateUnits(int index)
4     {
5       for (int i = index; i < units.Count; i++)
6       {
7           units[i].gameObject.SetActive(false);
```

```
8       }
9     }

10  }
```

Have it be called after placing the units:

```
11  private int PlaceUnits(List<Vector3> positions)
12      {
13          int unitIndex = 0;
14          for (int i = 0; i < positions.Count; i++)   // use positions count for safety
15          {
16
17              Transform unit = units[i];
18              unit.position = positions[i];
19              unit.gameObject.SetActive(true);
20              unitIndex++;
21          }

22          DeactivateUnits(unitIndex);

23          return unitIndex;
24      }
```

8 Adding Depth

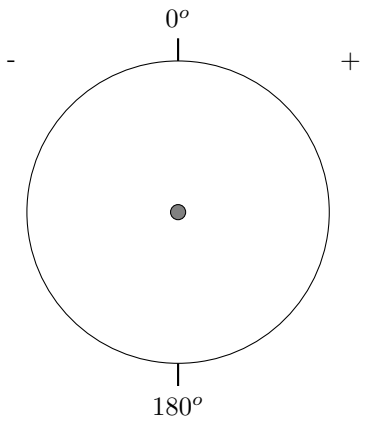
Prepare the `GFequation` with the required equations.

```
1  public static class GFequation {
2      ...

3      public static Vector3 DepthUnitPosition(Vector3 F,
4                                              float angle,
5                                              float distBtxtUnit
       )
6      {
7          Vector3 offset = new Vector3 (-Mathf.Sin(angle * Mathf
       .Deg2Rad), 0f, -Mathf.Cos((angle * Mathf.Deg2Rad))) *
       distBtxtUnit;
8          Vector3 position = F + offset;
9
10         return position;
11     }

12 }
```

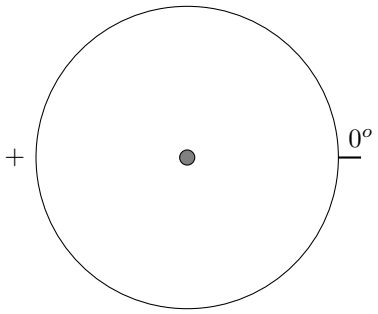
8.1 Changes to Depth Equation

NOTICE how the x value in `offset` is given a negative sign instead of being positive. This is to align the coordinates with Unity's rotation.

Unity's rotation goes from 0 to 180 or 0 to -180.

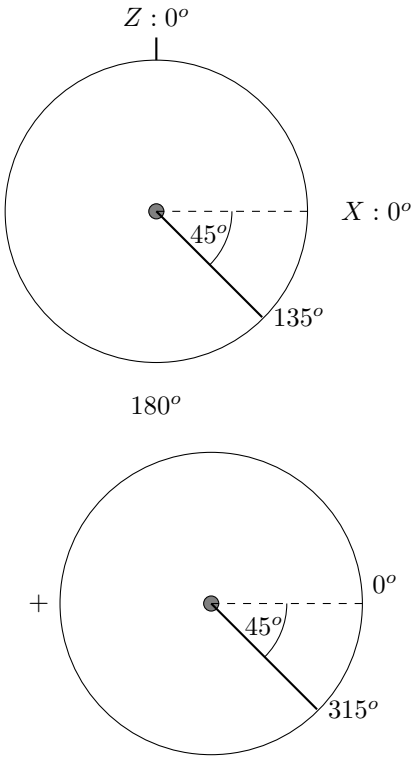


This poses an issue when using the Unit Circle as it that goes from 0 to 360.



8.1.1 Example

When moving down to the right by 45^o; in the Unit Circle the degree of would be `315`, but in Unity's alignment the angle would actually be `135`.



Substract 135 by 90, due to aligning the Z-axis with the X-axis, one would get the angle `45`.
Where $\sin(45) = 0.707$. It is a positive value which would make the depth units go up when dragging down instead of going down. Which is why the negative value is placed giving -sin(*angle*). As this produces the desired behaviour that is wanted when dragging up or down.

8.2 Adding Procedure

```
1  public static class GFprocedure {
2    ....

3    public static List<Vector3> DepthPositions(List<Vector3>
       positions,
4                                               float distBtxtUnit,
5                                               int totalFrontUnits
       ,
6                                               int totalUnits,
7                                               float angle)
8      {
9          int firstUnitOnDepth = totalFrontUnits;
10         for(int frontI = 0, depthI = firstUnitOnDepth; depthI
       < totalUnits; frontI++, depthI++)
11         {
12             Vector3 F = positions[frontI];
13             Vector3 depthPosition = GFequation.
       DepthUnitPosition(F, angle, distBtxtUnit);
14             positions.Add(depthPosition);
15         }
16
17         return positions;
18     }

19 }
```

`DepthPositions` is called inside **PositionsOnRadius**.

```
1     public static List<Vector3> PositionsOnRadius(Vector3 I,
2                                                    Vector3 E,
3                                                    float
      sizeOfUnit,
4                                                    float gap,
5                                                    float angle,
6                                                    int maxUnits
      )
7     {
8         ...
9         float distBtxtUnit = sizeOfUnit + gap;

10        positions = DepthPositions(positions, distBtxtUnit,
      totalFrontUnits, maxUnits, angle);

11        return positions;
12    }
```

8.3 Toggling Depth

Add a boolean to be able to toggle depth.

```
1  public class GridFormation : MonoBehaviour {
2    ...

3    bool depth;
```

private void Start()

```
5      depth = true;

6    }
7    ...
8  }
```

Now modify `PlaceUnits` to toggle between depth and not depth.

```
1  private int PlaceUnits(List<Vector3> positions)
2      {
3          int unitIndex = 0;
4          for (int i = 0; i < positions.Count; i++)   // use positions count for safety
5          {
6              ...
7          }

8          if (!depth)
9          {
10             DeactivateUnits(unitIndex);
11         }
```

```
12
13          return unitIndex;
14      }
```

9 Refactored

GFprocedure

```
1   public static List<Vector3> GridFormationPositions(Vector3 I,
2                                                       Vector3 E,
3                                                       float
        sizeOfUnit,
4                                                       float gap,
5                                                       int
        maxUnits,
6                                                       float angle
        )
7       {
8           float radius = Vector3.Magnitude(E - I);
9           int totalFrontUnits = GFequation.TotalUnits(radius,
        sizeOfUnit, gap);
10          float distBtxtUnit = sizeOfUnit + gap;
11
12          List<Vector3> gridPositions = new List<Vector3>();
13          gridPositions = PositionsOnRadius(I, E, sizeOfUnit,
        gap, angle, maxUnits);
14          gridPositions = DepthPositions(gridPositions,
        distBtxtUnit, totalFrontUnits, maxUnits, angle);
15
16          return gridPositions;
17      }
```

GFprocedure

```
1   public static List<Vector3> PositionsOnRadius(Vector3 I,
2                                                  Vector3 E,
3                                                  float sizeOfUnit
        ,
4                                                  float gap,
5                                                  float angle,
6                                                  int maxUnits)
7       {
8           float radius = Vector3.Magnitude(E - I);
9           int totalFrontUnits = GFequation.TotalUnits(radius,
        sizeOfUnit, gap);
10
11          List<Vector3> positions = new List<Vector3>();
12
13          for (int i = 0; i < totalFrontUnits; i++)
```

```
14          {
15              if (i >= maxUnits)
16              {
17                  return positions;
18              }
19              Vector3 currentPosition = GFequation.
    PositionOnRadius(I, sizeOfUnit, gap, angle, i);
20              positions.Add(currentPosition);
21          }
22
23          return positions;
24      }
```

10 Main Procedure

```
1   public class GridFormation : MonoBehaviour {
2     ...
3     private void Update()
4     {
5       Dragging(I, C);
6       if (dragging)
7       {
8         RotateGuide();
9         C = ClickedCoord();
10        float angledAligned = guide.transform.rotation.
      eulerAngles.y - 90;
11        float newRadius = Vector3.Magnitude(C - I);
12        if (angle != angledAligned || radius != newRadius)
13        {
14          angle = angledAligned;
15          radius = newRadius;
16
17          List<Vector3> positions = GFprocedure.
      GridFormationPositions(I, C, sizeOfUnit, gap, maxUnits,
      angle);
18          int totalUnitsPlaced = PlaceUnits(positions);
19        }
20      }
21    }
22    ...
23  }
```

11 Integrating with Units

At the moment only the positions in the grid formation are calculated and displayed. There are no units moving towards those positions. To have units move towards the positions in the grid formation, certain additions and changes must be made.

11.1 Unit Manager

Create a new script called **UnitManager**. The Unit Manager will contain:

- Grid Formation
- Units to move
- Move units

```
1  public class UnitManager : MonoBehaviour {
2    GridFormation gridFormation;
3    List<GameObject> units;
4
5    void Start() {
6      gridFormation = GetComponent<GridFormation>();
7    }
8
9    void MoveUnits(List<Vector3> positions) {
10     for(int i = 0; i < positions.Count; i++){
11       units[i].transform.position = positions[i];    // <-- replace
12       // Movement behaviour goes here
13     }
14   }
15 }
```

11.2 Changes to GridFormation

Units begin to move after dragging ended and positions have been calculated. From here Grid Formation calls **MoveUnits** from `UnitManager` to move units to calculated behaviour

```
1  public class GridFormation : MonoBehaviour {
2    ...

3    UnitManager unitManager;

4    void Start() {
5      ...

7      unitManager = GetComponent<UnitManager>();
```

```
8     }
9
10    private void Dragging(...)
11    {
12      ...
13      if (Input.GetMouseBottomUp(rightMButton) {
14        dragging=false;
15        this.E = ClickedCoord();

16        unitManager.MoveUnits(positions);

1         }
2       }
3     }
4     ...
5   }
```

11.3 Passing data between Unit Manager and Grid Formation

Have the Unit Manager pass the total units avaialble + the size of the units. This is to have both Unit Manager and Grid Formation to be in sync, and not have Grid Formation calculate unecessary positions.

Have `maxUnits` be private in *GridFormation*:

```
1   public class GridFormation : MonoBehaviour {
2     ...

3     private int maxUnits;

4     ...
5   }
```

In *UnitManager* create a property that returns the number of units currently in game:

```
1   public class UnitManager : MonoBehaviour {
2     GridFormation gridFormation;
3     List<GameObject> units;

4     public int TotalUnits { get {return units.Count; } }

5
6     ...
7   }
```

The amount of units to be displayed happens the moment dragging begins, and not in the beginning in start.

```
1  public class GridFormation : MonoBehaviour {
2    ...
3    private void Dragging(Vector3 I, Vector3 E) {
4    this.I = ClickedCoord();

5    maxUnits = unitManager.TotalUnits;
6    FillDisplayUnits(maxUnits);

7    ...
8  }
```

For the moment, new display units are being created upon dragging. It would be best to clear the display units and fill the units according to the max units.

Create a method that clears the display units in *GridFormation*

```
1  public class GridFormation : MonoBehaviour {
2    ...

3    private void DestroyUnitDisplay() {
4      foreach(Transform unit in displayUnits) {
5        Destroy(unit.gameObject);
6      }
7      displayUnits.Clear();
8    }

9  }
```

Add the method when dragging begins; before filling the `Display Units`.

```
1  public class GridFormation : MonoBehaviour {
2    ...
3    private void Dragging(Vector3 I, Vector3 E) {
4    this.I = ClickedCoord();

5    DestroyUnitDisplay();

6    maxUnits = unitManager.TotalUnits;
7    FillDisplayUnits(maxUnits);
8    ...
9  }
```

11.4 Orientating Units

Units do not face the direction of the formation when dragging finishes. A direction must be passed to the units that tells them the directoin they must face.

```
public class UnitManager : MonoBehaviour {
  ...
  public void MoveUnits(List<Transform> positions, Vector3
    direction)
  {...}
  ...
}
```

To avoid typing alot in the future in `MoveUnits`, the current unit and position will be stored in their seperate variable.

```
public class UnitManager : MonoBehaviour {
  ...
  public void MoveUnits(List<Transform> positions, Vector3 direction) {
    GameObject = unit = units[i];
    Vector3 pos = positions[i].position;

    unit.transform.position = pos;
  }
}
```

Rotating the unit requires gathering the angle between the `direction` vector and the `forward` vector of the unit.

```
public class UnitManager : MonoBehaviour {
  ...
  public void MoveUnits(List<Transform> positions, Vector3 direction) {
    ...
    unit.transform.position = pos;
    float angle = Vector3.SignedAngle(unit.transform.forward,
    direction, Vector3.up);
  }
}
```

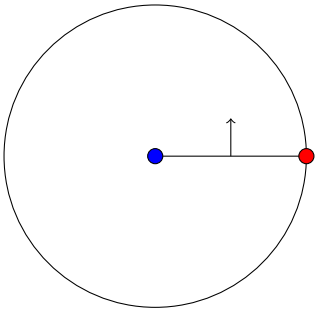
As the units rotation is on the `Y-Axis`, the angle will be applied as follows:

```
public class UnitManager : MonoBehaviour {
  ...
  public void MoveUnits(List<Transform> positions, Vector3 direction) {
    ...
    float angle = Vector3.SignedAngle(unit.transform.forward, direction, Vector3.up);
    unit.transform.Rotate(new Vector3(0, angle, 0));
  }
}
```

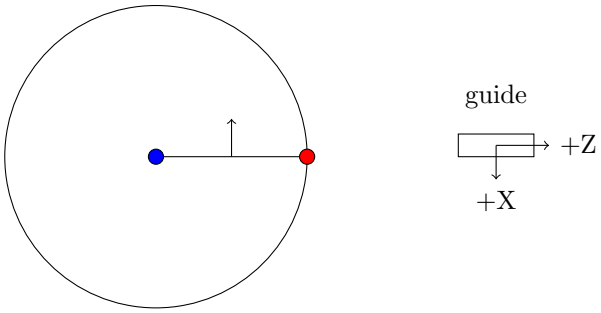
11.5 Passing Rotation from Grid Formation

The `guide` is responsible for acquiring the direction of the formation.

11.5.1 Direction to face



When dragging in the example above, the `guide's` local rotation will have its forward vector along the radius.



From observing the diagram, the correct direction is `-X` from the guide's local rotation.
In Unity, going along the X-Axis is depicted using `transform.right`. Acquiring the negative direction is simply negative the value to `-transform.right`.

```
1  public class GridFormation : MonoBehaviour {
2    ...
3    private void Dragging(Vector3 I, Vector3 E) {
4      ...
5      if (Input.GetMouseButtonUp(rightMButton) {
6        ...
7        this.E = ClickedCoord();

8      unitManager.MoveUnits(displayUnits, -guide.transform.right);

9    }
10   ...
11 }
```

11.6 Single unit upon clicking

When right clicking without dragging, there will be no units to display, despite there needing to be at least one unit on the origin. This is because the radius would have no size, meaning no units would fit when simply right clicking without dragging.
Fixing this requires to pass the size of the unit as the radius upon dragging. The size of the unit is acquired from `Unit Manager`. Make a property that returns the size of unit for grid formation to call.

```
1  public class UnitManager : MonoBehaviour {
2    ...
3    float sizeOfUnit;

4    public float SizeOfUnit { get { return sizeOfUnit; } }

5    ...
6  }
```

When beginning to drag, simply call the property and assign it to the radius in `GridFormation`.

```
12 public class GridFormation : MonoBehaviour {
13   ...
14   private void Dragging(Vector3 I, Vector3 E) {
15     if (Input.GetMouseButtonDown(rightMButton) {
16       ...
17       FillDisplayUnits(maxUnits);

18       radius = unitManager.SizeOfUnit;

19   }
20   ...
21 }
```

12 General Positions

12.1 Adding procedure

In `GFprocedure` add the *General Positions*:

```
1  public static class GFprocedure {
2    ...
3    public static List<Vector3> GeneralPositions(Vector3 I,
4                                                 Vector3 E,
5                                                 float
       sizeOfUnit,
6                                                 float gap,
7                                                 int maxUnits,
8                                                 float angle)
9    {
10     float radius = Vector3.Magnitude(E - I);
11     int unitsOnRadius = GFequation.TotalUnits(radius,
       sizeOfUnit, gap);
12     int totalDepths = GFequation.TotalDepth(maxUnits,
       unitsOnRadius);
13
14     float distBtxtUnit = sizeOfUnit + gap;
15
16     // No Depth
17     // ________
18     if (totalDepths == 0)
19     {
20         return PositionsOnRadius(I, E, sizeOfUnit, gap, angle,
        maxUnits);    // On the radius
21     }
22
23     // Depth
24     // _____
25     Vector3 curI = I;
26     List<Vector3> positions = new List<Vector3>();
27
28     // Main Procedure
29     // ______________
30     for (int curDepth = 0, numUnits = 0; curDepth <=
       totalDepths; curDepth++) {
31
32         for (int i = 0; i < unitsOnRadius && numUnits <
       maxUnits; i++, numUnits++)
33         {
34             Vector3 position = GFequation.PositionOnRadius(
       curI, sizeOfUnit, gap, angle, i);
35             positions.Add(position);
36         }
```

```
37          curI = GFequation.DepthUnitPosition(curI, angle,
        distBtxtUnit);      // Go down by to I depth
38      }
39
40      return positions;
41
42  }

44  }
```

12.2 Integrating into Grid Formation

```
1   public class GridFormation : MonoBehaviour {
2     ...
3     void Update {
4       if (angle != angleAligned || radius != newRadius) {
5           ...

6           List<Vector3> positions = GFprocedure.GeneralPositions
        (I, C, sizeOfUnit, gap, maxUnits, angle);

7           int totalUnitsPlaced = PlaceUnits(positions);
8       }
9       ....
10  }
```

13 Pascal's position

13.1 Equations

Add the equations of pascal into `GFequation`.

```
1  public static class GFequation {
2    ...

3    public static Vector3 PascalsCenter(Vector3 midPoint, float
       distBtxtUnit, int currentDepth, float angle)
4    {
5      Vector3 offset = new Vector3(-Mathf.Sin(angle * Mathf.
       Deg2Rad), 0f, -Mathf.Cos(angle * Mathf.Deg2Rad));
6      offset *= distBtxtUnit * currentDepth;
7
8      Vector3 center = midPoint + offset;
9
10     return center;
11   }
12
13   public static Vector3 PascalsEnd(Vector3 center, float
       distBtxtUnit, int unitsRemaining, float angle)
14   {
15     float shift = distBtxtUnit * (unitsRemaining - 1);
16     shift = shift / 2;
17
18     Vector3 offset =  new Vector3(Mathf.Cos(angle * Mathf.
       Deg2Rad), 0f, -Mathf.Sin(angle * Mathf.Deg2Rad)); // must
       point opposit direction. Negative inverts the cubes
19     offset *= shift;
20
21     Vector3 end = center - offset;
22     return end;
23   }
24
25   public static Vector3 PascalsPosition(Vector3 end, float
       distBtxtUnit, float angle, int index)
26   {
27     Vector3 offset = new Vector3(Mathf.Cos(angle * Mathf.
       Deg2Rad), 0f, -Mathf.Sin(angle * Mathf.Deg2Rad)); //
       negate Z to inverse. Due to Unity implementation
28     offset *= distBtxtUnit;
29
30     return end + (offset * index);
31   }

33 }
```

13.2 Procedure

Add the Pascal procedure into `GFprocedure`.

13.2.1 Pascal's Positions

Create the function signature:

```
public statc List<Vector3> PascalsPositions(Vector3 I,
                                            Vector3 E,
                                            float distBtxtUnit,
                                            int curDepth,
                                            float angle,
                                            int unitsRem) {

}
```

Calculating the mid point:

```
public static List<Vector3> PascalsPositions(...) {

  Vector3 halfRadius = (E - I) / 2;
  Vector3 midPoint = I + halfRadius;

}
```

Once the mid point has been calculated, both the center and the end are to be calculated as well:

```
public static List<Vector3> PascalsPositions(...) {
  ...

  Vector3 center = GFequation.PascalsCenter(midPoint,
    distBtxtUnit, curDepth, angle);
  Vector3 end = GFequation.PascalsEnd(center, distBtxtUnit,
    unitsRemaining, angle);

}
```

Now it is time to calculate the positions on the last row:

```
public static List<Vector3> PascalsPositions(...) {
  ...

  List<Vector3> positions = new List<Vector3>();
  for (int i = 0; i <= unitsRemaining; i++)
  {
    Vector3 position = GFequation.PascalsPosition(end,
    distBtxtUnit, angle, i);
    positions.Add(position);
  }

  return positions;
```

```
}
```

13.2.2 Adding Pascal Positions in General Procedure

Create the function signature, the same as `GeneralPositions`, in *GFprocedure*.

```
public static List<Vector3> GeneralPositionsPascal(Vector3 I,
                                                   Vector3 E,
                                                   float sizeOfUnit,
                                                   float gap,
                                                   int maxUnits,
                                                   float angle) {

}
```

The contents from `General Positions` is the same for `GeneralPositionsPascal`

```
public static List<Vector3> GeneralPostisionsPascal(...){

    float radius = Vector3.Magnitude(E - I);
    int unitsOnRadius = GFequation.TotalUnits(radius,
    sizeOfUnit, gap);
    int totalDepths = GFequation.TotalDepth(maxUnits,
    unitsOnRadius);

    float distBtxtUnit = sizeOfUnit + gap;

    if (totalDepths == 0)
    {
    return PositionsOnRadius(I, E, sizeOfUnit, gap, angle,
    maxUnits);    // On the radius
    }
    Vector3 curI = I;
    List<Vector3> positions = new List<Vector3>();

    for (int curDepth = 0, numUnits = 0; curDepth <=
    totalDepths; curDepth++) {

      for (int i = 0; i < unitsOnRadius && numUnits < maxUnits
    ; i++, numUnits++)
      {
        Vector3 position = GFequation.PositionOnRadius(curI,
    sizeOfUnit, gap, angle, i);
        positions.Add(position);
      }
      curI = GFequation.DepthUnitPosition(curI, angle,
    distBtxtUnit);      // Go down by to I depth
    }

  return positions;
```

```
26 }
```

Since the Pascal Positions only occur at the last depth, a check is placed to apply pascal's positions upont the last depth.

```
1  public static List<Vector3> GeneralPositionsPascal(...) {
2    ...
3    for(int curDepth = 0, numUnits = 0; curDepth <= totalDepths; curDepth++) {

4      if (curDepth == totalDepths) {
5        int unitsRemaining = maxUnits % unitsOnRadius;
6        List<Vector3> pPascal = PascalsPositions(I, end,
       distBtxtUnit, curDepth, angle, unitsRemaining);
7        positions.AddRange(pPascal);
8
9        return positions;
10     }

11     ...
12 }
```

Pascal's positions is ony applied when there are reaminder units. $unitsRemaining > 0$. Move the *unitsRemainig* from Pascal's check onto the beginning, and inside the pascal's check.

```
1  public static List<Vector3> GeneralPositionsPascal(...) {
2    ...

1    int unitsRemaining = maxUnits % unitsOnRadius

...

2    if (curDepth == totalDepths && unitsRemaining > 0) {
3      // remove 'unitsRemainig'
4    }

5    ...
6  }
```

13.3 Integrating into Grid Formation

```
1  public class GridFormation : MonoBehaviour {
2    ...
3    void Update {
4      if (angle != angleAligned || radius != newRadius) {
5          ...

6      List<Vector3> positions = GFprocedure.
       GeneralPositionsPascal(I, C, sizeOfUnit, gap, maxUnits,
       angle);
```

```
7           int totalUnitsPlaced = PlaceUnits(positions);
8       }
9       ....
10  }
```

13.4 Fixing drag

At the moment the last row follows the middle of the radius. This creates a dragging effect. To fix this, simply pass the position of the last unit on the radius to `PascalsPositions`.

```
1   public static List<Vector3> GeneralPositionsPascal(...) {
2     ...
3     if (curDepth == totalDepths && unitsRemaining > 0) {

4       Vector3 end = GFequation.PositionOnRadius(I, sizeOfUnit,
        gap, angle, unitsOnRadius - 1);

5       ...
6     }
7     ...
8   }
```

13.5 Toggle

A boolean is added to toggle between `Pascal's Positions` or `Base Positions`.

```
1   public class GridFormation : MonoBehaviour {
2     ...

3     bool pascal;

4     ...
5   }
```

Switching between pascal or base:

```
1   public class GridFormation : MonoBehaviour {
2     ...
3     void Update {
4       if (angle != angleAligned || radius != newRadius) {
5           ...

6       List<Vector3> positions;
7           if (pascal) {
8               positions = GF.procedure(GeneralPositionsPascal(I,
        C, sizeOfUnit, gap, maxUnits, angle);
9           } else {
```

```
10              positions = GF.procedure(GeneralPositions(I, C,
    sizeOfUnit, gap, maxUnits, angle);
11          }

12      ...
13    }
14  }
```

14 Conclusion

This implements the Grid Formation behaviour only. It produces the coordinates to form the grid formation which can be integrated with other behaviours. There are many ways to improve or implement various new mechanics. This hopefully serves as foundation to implement a grid formation into a project.

15 Repository

Source Code

16 License