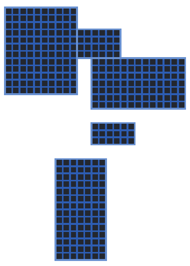
Lab - Procedural



Goals

- To get acquainted with some of Unity's features that allow you, as a programmer, to make the development process more robust.
- To get introduced to various methods for procedural creation of content.

Resources

- Unity scripting tutorials (https://unity3d.com/learn/tutorials/s/scripting)
- Line renderer (https://docs.unity3d.com/Manual/class-LineRenderer.html)
- Scripting concepts (https://docs.unity3d.com/Manual/ScriptingConcepts.html)

- Procedural Mesh Geometry (https://docs.unity3d.com/Manual/GeneratingMeshGeometryProcedurally.html)

Basic Challenge

Broadly speaking, there are two fundamental aspects to programming for game development. One is to produce the code that makes the game, and the other is to produce tools for other members of the team (who may be non-programmers) in order to make it easier for them to build the game. In this lab we will mostly address the former, focusing on few basic elements of procedural programming in Unity. However, we will begin with a brief exercise that highlights some aspects of programming for other users.

Programming for others

A programmer in a game development team is a specialist. Often their job is to abstract away the technical details of what's under the hood, so that non-programmers can get on with their work without needing to read and/or understand the code. In this exercise you will be producing a script intended to be used by non-programmers, so you need to reduce the number of potential errors that these other users can make when using your script.

- Fork the Procedural project (https://altitude.otago.ac.nz/cosc360/Procedural) on GitLab, then clone your fork to your hard-drive.

- Open the *Procedural* project, the directory of the cloned repository, in Unity.

- In the *Assets/Scripts* you'll find "Animate.cs" script. This is a generic script which produces basic animation by changing the sprite that gets rendered over a particular game object. The code is given below. Read it carefully and make sure you understand what it does.

Animate.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Animate : MonoBehaviour {

   // An array with the sprites used for animation
   public Sprite[] animSprites;

   // How fast to change frame in the animation
   public int framesPerSec = 20;

   // Reference to the renderer of the sprite
   // game object
   private SpriteRenderer sr;

   // Index of the next frame to show in the animation
   private int nextFrameIndex = 0;

   // Variable for keeping track of time since
   // last frame change
   private float timeSinceLastFrame = 0.0f;


   // Use this for initialization
   void Start () {
      // Get a reference to game object renderer and
      // cast it to Sprite Render
      sr = GetComponent<Renderer>() as SpriteRenderer;
   }

   void Update () {
      // Add time since last game frame to time since last
      // animation frame change
      timeSinceLastFrame += Time.deltaTime;

      //Check if it's time for animation frame change
      if(timeSinceLastFrame > 1f/(float) framesPerSec) {

         //Change the sprite
         sr.sprite = animSprites[nextFrameIndex];
         //Increment frame index and
         //reset to 0 if it's larger than the number
         //of frames in animSprites
         nextFrameIndex++;
         nextFrameIndex %= animSprites.Length;

         //Reset time since last animation frame
         timeSinceLastFrame -= 1f/(float) framesPerSec;
      }
   }
}
```

```
}

```

The script expects the user to supply the sprites for animation in the *animSprites* array. On Start() it grabs the reference to the *Sprite Renderer* component of the game object. Inside the *Update()* method it keeps track of time since last sprite change and once enough time has passed (to produce the required frames per second) the next frame from the *animSprites* array is provided to the *SpriteRenderer* to render from now on.

- Let's try to build an animated game object that will be using this script. In the **Hierarchy panel** create a new empty object. Add "Animate.cs" as its component. Play the scene.
- You should get "IndexOutOfRangeException: Array index is out of range" error (look for it at the bottom of the Unity Editor window). Of course, you haven't initialised the *animSprites* array, and the script is crashing when it tries to reference the first element in that array.

When creating a generic script, which might be used by someone who won't necessarily understand the logic of your code, you should handle potential problems like that and let the user know what is wrong. Remember, often people will have many game objects in the scene using the same script, and so error messages that don't point out exactly where the problem might lie are of little help. You should handle potential misconfigurations and give as much information to the user as possible about the error.

- Below are three possible solutions to handling this problem. Test them one by one and explain what is different about the way they handle the problem of no sprites in the *animSprites* array. Think about which way might be more appropriate in which situation.

Code diffs

In this lab, in order to encourage reading and understanding the code, as opposed to passive copying and pasting, the changes to the script will be given as difference outputs between the previous version and the new version of the code. In those difference outputs the blue/underlined code is the code you need to add and the red/crossed-out code is the code you need to remove. There should be enough in the context of the changes to let you know where to make them. Remember, this whole exercise will be pointless if you don't make an effort to understand what the code does and how.

Differences between two versions of Animate.cs

```
.
.
.
   void Update () {
      // Add time since last game frame to time since last
      // animation frame change
           timeSinceLastFrame += Time.deltaTime;

      //Check if it's time for animation frame change
      if(animSprites.Length > 0 && timeSinceLastFrame > 1f/(float) framesPerSec) {

         //Change the sprite
         sr.sprite = animSprites[nextFrameIndex];
         //Increment frame index and
         //reset to 0 if it's larger than the number
         //of frames in animSprites
         nextFrameIndex++;
         nextFrameIndex %= animSprites.Length;

         //Reset time since last animation frame
         timeSinceLastFrame -= 1f/(float) framesPerSec;
      }
   }
}
```

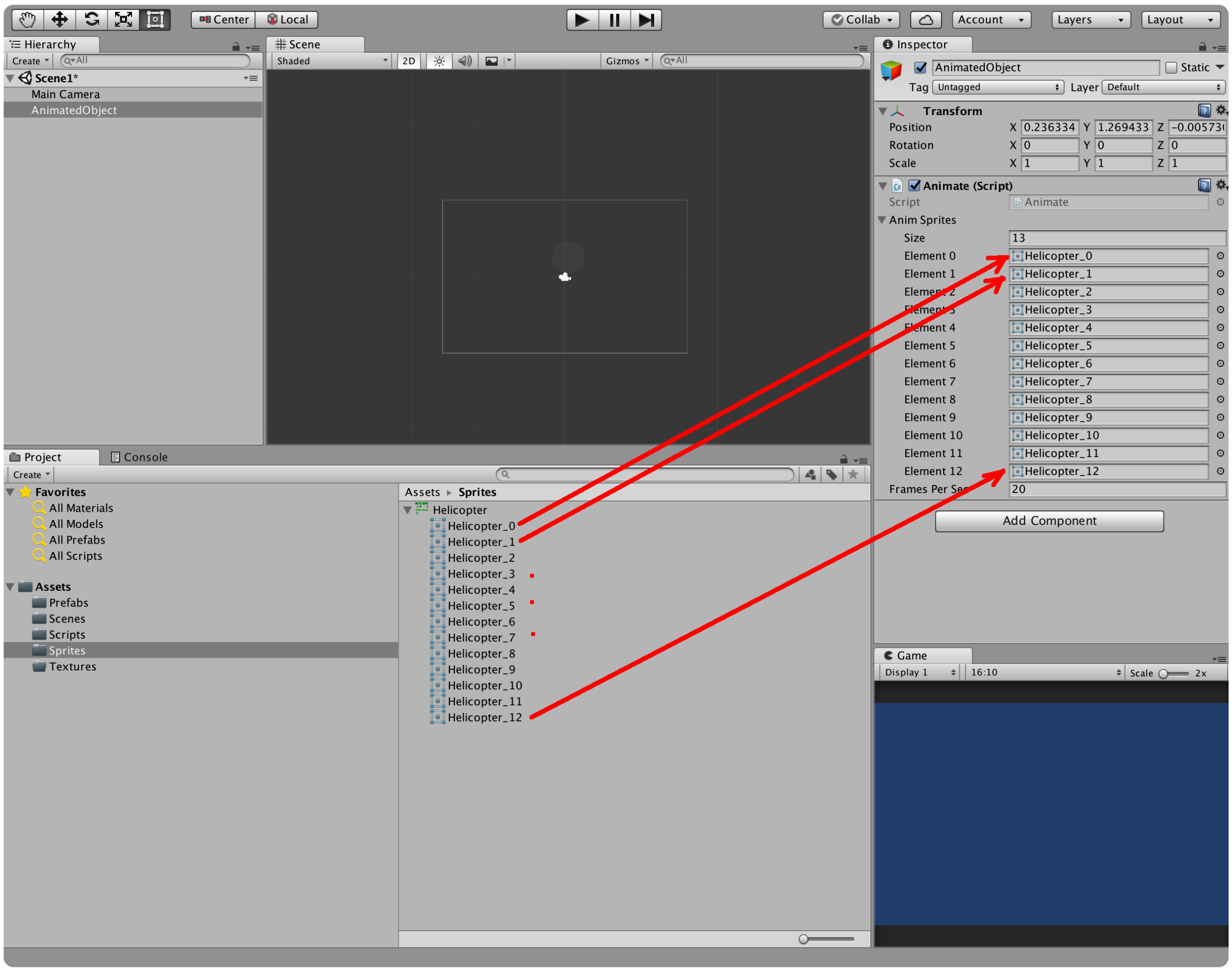
Differences between two versions of Animate.cs

```
.
.
.
   // Use this for initialization
   void Start () {
      // Get a reference to game object renderer and
      // cast it to Sprite Render
      sr = GetComponent<Renderer>() as SpriteRenderer;
      if(animSprites.Length == 0) {
         Debug.LogError("Missing sprites in game object '" + gameObject.name + "'!  Deact
         gameObject.SetActive(false);
      }
   }
.
.
.
```

Differences between two versions of Animate.cs

```
.
.
.
   // Use this for initialization
   void Start () {
      // Get a reference to game object renderer and
      // cast it to Sprite Render
      sr = GetComponent<Renderer>() as SpriteRenderer;
      if(animSprites.Length == 0) {
         Debug.LogError("Missing sprites in game object '" + gameObject.name + "'!  Pausi
         Debug.Break();
      }
   }
.
.
.
```

- Decide which error handling you prefer and use that one in "Animate.cs".
- Let's initialise the *animSprites* array. Expand the "Anim Sprites" property of the *Animate (Script)* component of the object you've been working with. Set its size to 13 (for there are 13 sprites). In *Assets/Sprites* there is a sprite sheet, already spliced, depicting a helicopter with a rotating blade. Expand the sprite sheet and drag the images, one by one, over to consecutive positions in the "Anim Sprites" array as shown in the screenshot below.



AnimatedObject — Static
Tag: Untagged — Layer: Default

Transform
Position X 0.236334 Y 1.269433 Z -0.00573
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Animate (Script)
Script: Animate
Anim Sprites
Size: 13
Element 0: Helicopter_0
Element 1: Helicopter_1
Element 2: Helicopter_2
Element 3: Helicopter_3
Element 4: Helicopter_4
Element 5: Helicopter_5
Element 6: Helicopter_6
Element 7: Helicopter_7
Element 8: Helicopter_8
Element 9: Helicopter_9
Element 10: Helicopter_10
Element 11: Helicopter_11
Element 12: Helicopter_12
Frames Per Se: 20

- OK, the sprite animations are set. Play the scene.

0. There still should be no animation and you should see the "NullReferenceException: Object reference not set to an instance of an object" error. This error is triggered by the line that tries to change the sprite by referencing the variable *sr*, which is supposed to reference the *SpriteRenderer* component of the game object...except the game object doesn't have the *SpriteRenderer* component, and the call to *GetComponent* in *Start()* in the "Animate.cs" script returns null.

1. You could handle this error in a similar way as you handled the problem of empty *animSprites* array and keep checking if *sr* is null. Sometimes however, it's just easier to ensure that the error cannot occur. In this particular situation, this means ensuring that *sr* is never null.

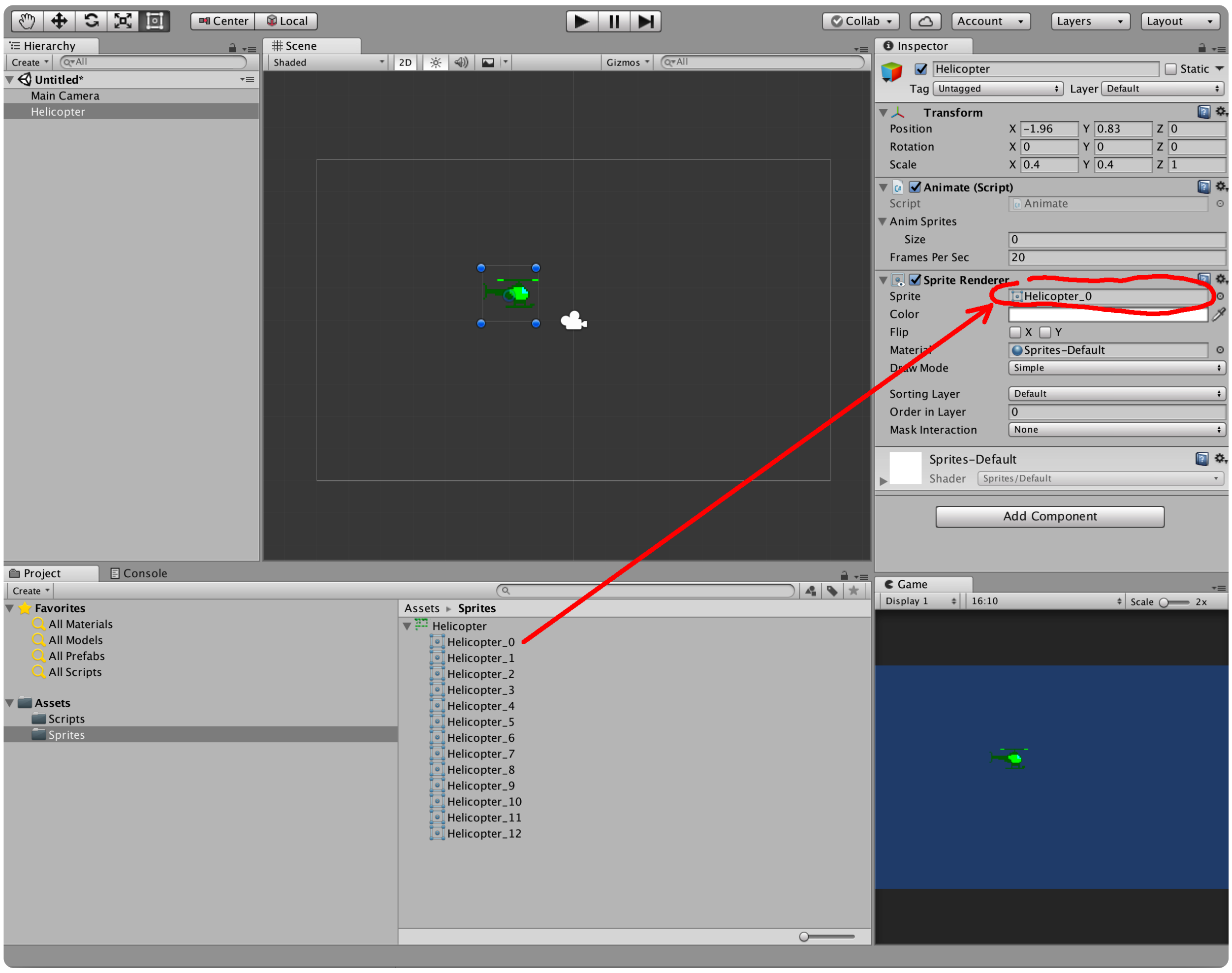
2. In your script you can include directives to the Unity Editor about required components (https://docs.unity3d.com/ScriptReference/RequireComponent.html). This means that a game object that adds this particular script as a component must also have these required components. Go ahead and add the requirement for the *Sprite Render*.

Differences between two versions of Animate.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(SpriteRenderer))]
public class Animate : MonoBehaviour {
.
.
.
```

3. The *RequireComponent* directive doesn't update game objects that already have "Animate.cs" as their script component. The check for requirements occurs when the script is added to the game object. To see how it works, delete the current game object from the hierarchy and crate a new empty object. Name it "Helicopter" and add the "Animate.cs" script component. The *SpriteRenderer* component should be added automatically. You don't need to check the *sr* for null, because it should never be null now.

4. You can drag "Helicopter_0" sprite over its "Sprite" property to render the first frame over the game object. Play the game.



Inspector

Helicopter

Tag Untagged Layer Default

Transform

	X	Y	Z
Position	-1.96	0.83	0
Rotation	0	0	0
Scale	0.4	0.4	1

Animate (Script)

Script	Animate
Anim Sprites	
Size	0
Frames Per Sec	20

Sprite Renderer

Sprite	Helicopter_0
Color	
Flip	X Y
Material	Sprites-Default
Draw Mode	Simple
Sorting Layer	Default
Order in Layer	0
Mask Interaction	None

Sprites-Default
Shader Sprites/Default

Assets ▸ Sprites

Helicopter
Helicopter_0
Helicopter_1
Helicopter_2
Helicopter_3
Helicopter_4
Helicopter_5
Helicopter_6
Helicopter_7
Helicopter_8
Helicopter_9
Helicopter_10
Helicopter_11
Helicopter_12

5. If you followed the instructions faithfully, you haven't initialised the *animSprites* array. Did your method for handling the empty "animSprites" array work? Go ahead and initialise the 13 sprite frames again.

6. If you play the scene now, the helicopter blades should be spinning. If you don't see the sprites, perhaps you need to change game object's Z-position (to 0). Set the "Helicopter" game object's scale (0.4,0.4,1).

7. Save the scene.

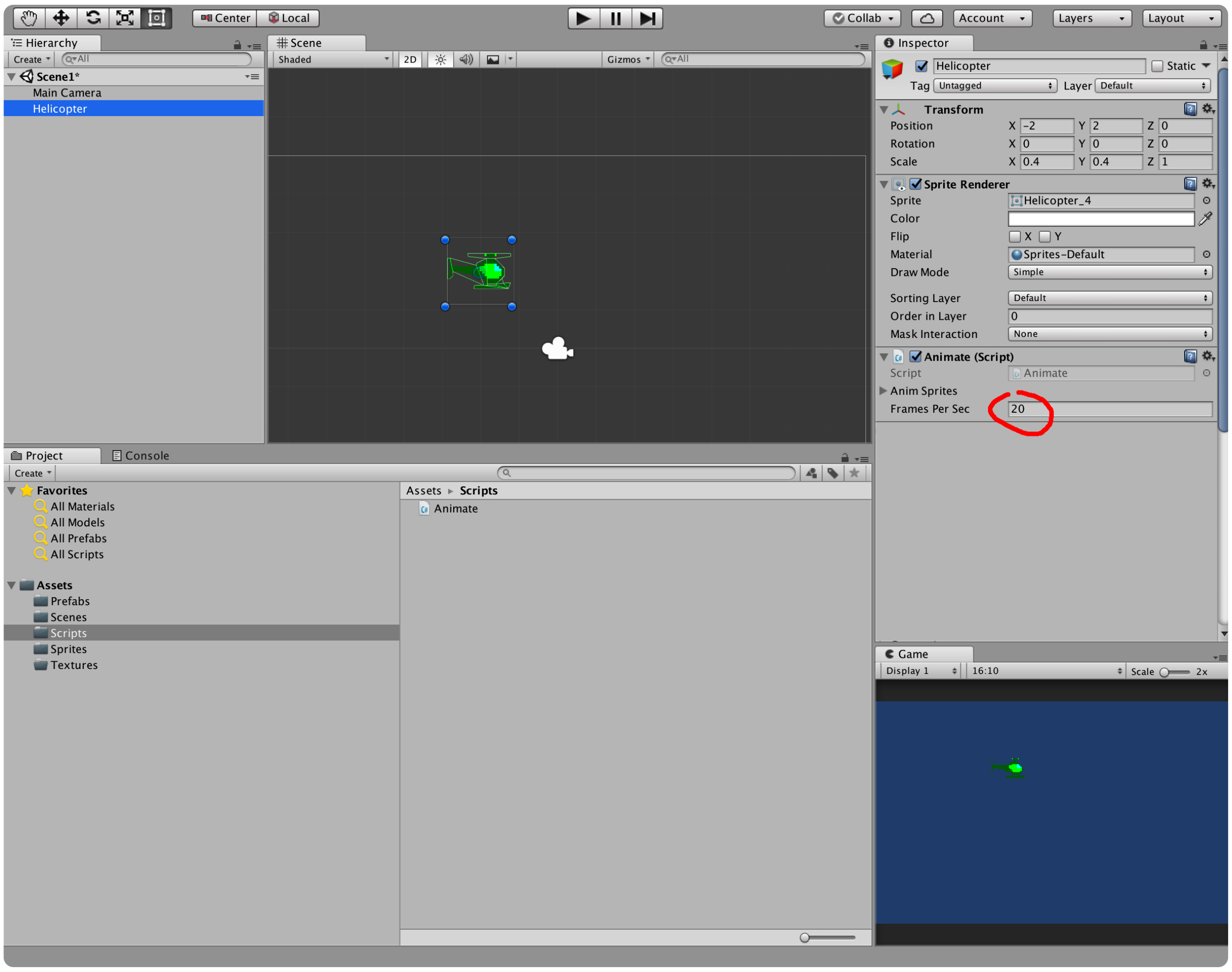
8. Sometimes it might be useful, for the purposes of development, to have a script executing in edit mode (https://docs.unity3d.com/ScriptReference/ExecuteInEditMode.html) - that is executing in the editor when the game is not playing. This is a little bit dangerous, because you have to be careful that the script doesn't depend on objects that are not there when the scene is in edit mode. However, in the case of our "Animate.cs" script, executing in edit mode is relatively safe - all it does is change the sprites. Add the following line to "Animate.cs":

Differences between two versions of Animate.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(SpriteRenderer))]
[ExecuteInEditMode]
public class Animate : MonoBehaviour {
.
.
.
```

Now, if you select the "Helicopter" game object in the **Hierarchy panel** you should see the animation even when the scene is not playing. This might be quite useful in various circumstances...for instance when procedurally generating content. You can test scripts quickly without running the whole scene. But we won't be needing this feature any more - you can comment out the *[ExecuteInEditMode]* line from "Animate.cs".

Did you notice the "Frame Per Sec" property of the "Animate" script component? It controls the speed of the animations. It's a great way to give another user the ability to customise how the script works without looking inside.



Helicopter Static
Tag Untagged Layer Default

Transform
Position X -2 Y 2 Z 0
Rotation X 0 Y 0 Z 0
Scale X 0.4 Y 0.4 Z 1

Sprite Renderer
Sprite Helicopter_4
Color
Flip X Y
Material Sprites-Default
Draw Mode Simple
Sorting Layer Default
Order in Layer 0
Mask Interaction None

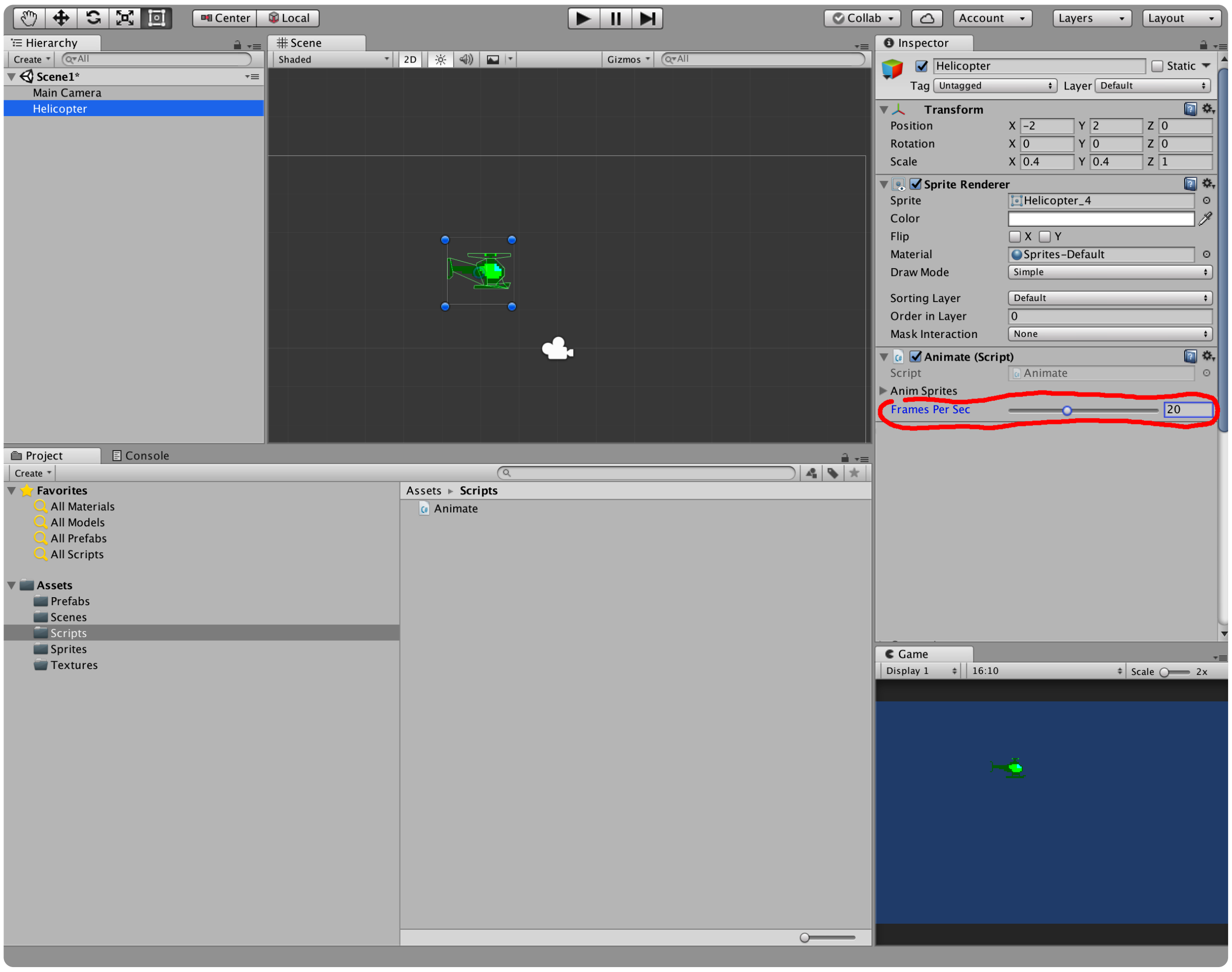
Animate (Script)
Script Animate
Anim Sprites
Frames Per Sec 20

However, not knowing how the numbers are used inside the script, the user might set them incorrectly. In this case, for instance, it's best if the "Frame Per Sec" is never 0 (because of the division inside the script) and not much point if it's more than 50....because at some point the animation cannot go faster than the game frame rate. One thing you can do is to provide a range of values that the number is expected to be in. Inside "Animate.cs" before declaring the *framesPerSec* variable add the following directive:

Differences between two versions of Animate.cs

```
.
.
.
   // How fast to change frame in the animation
   [Range (1, 50)]
   public int framesPerSec = 20;
.
.
.
```

2. Take a look at the "Frames Per Sec" attribute of "Animate" script component - it's now a slider, which will only allow the user to set the values of "Frames Per Sec" in the specified range.


Helicopter
Tag Untagged, Layer Default
Transform
Position X -2, Y 2, Z 0
Rotation X 0, Y 0, Z 0
Scale X 0.4, Y 0.4, Z 1
Sprite Renderer
Sprite Helicopter_4
Material Sprites-Default
Draw Mode Simple
Sorting Layer Default
Order in Layer 0
Mask Interaction None
Animate (Script)
Script Animate
Anim Sprites
Frames Per Sec 20

3. Commit all your work to git.

Drawing lines

The previous section tried to demonstrate some tools at the programmer's disposal and a bit of thinking about to make your script easy to use and robust to various errors. Next, we'll be looking at some ways of generating content procedurally. There are various reasons why you might want to generate game content directly from a script. Perhaps you want some randomisation, or need a bit of finer control, or perhaps aspects of the content are dynamic and thus need to be under control of a script.

4. The easiest thing to create procedurally in Unity is a line. Create a new game object and call it "Ground". Add the "Grassland.cs" script component, which can be found in *Assets/Scripts*. Notice the *LineRenderer* component that gets added automatically.

5. At the moment, the "Grassland.cs" script should have the following code:

Grassland.cs

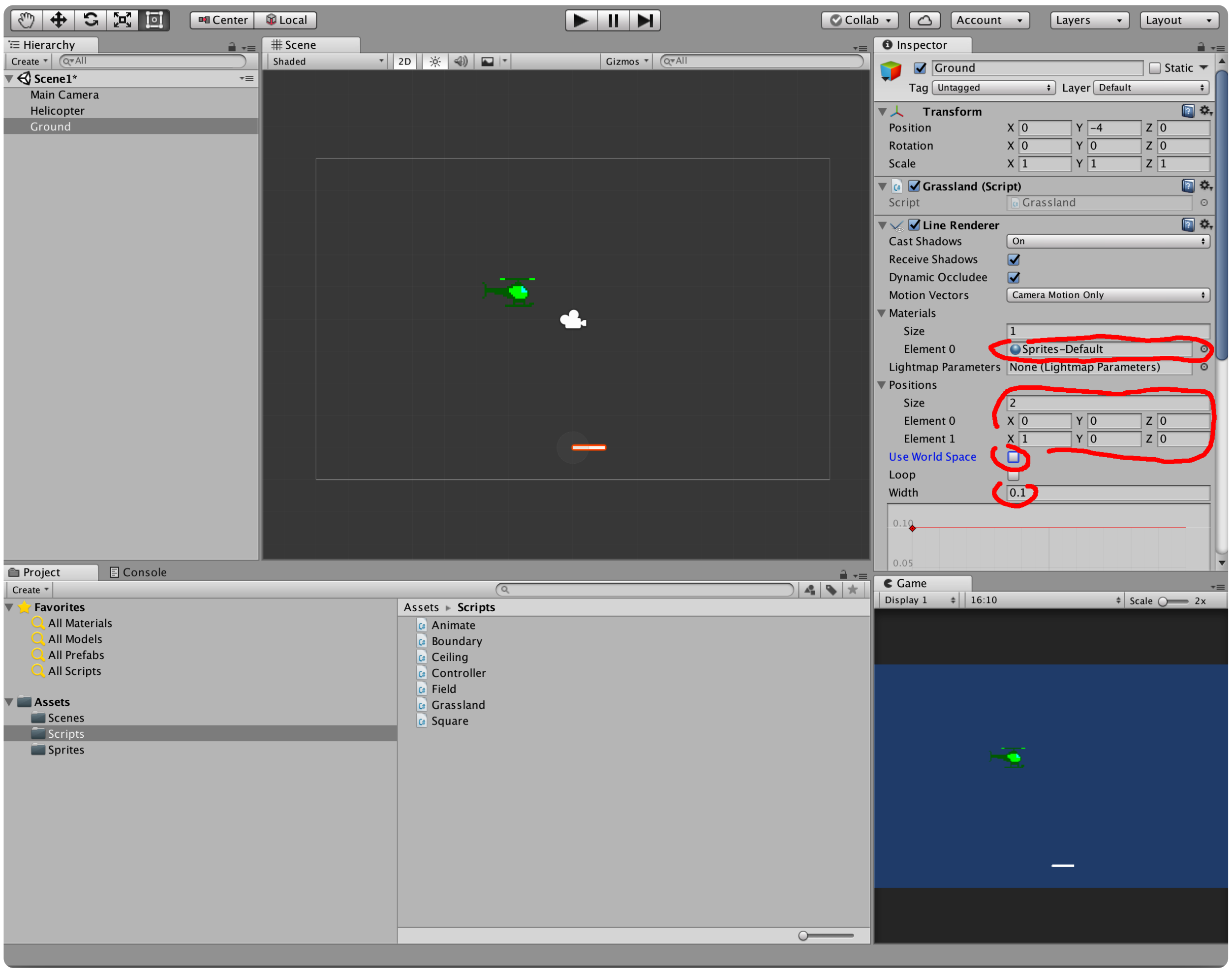
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(LineRenderer))]
public class Grassland : MonoBehaviour {

   // Reference to line renderer component
   private LineRenderer   lr;

   // Use this for initialization
   void Start () {
      // Fetch the reference to the line renderer component
      lr = GetComponent<LineRenderer>();
   }

   // Update is called once per frame
   void Update () {

   }
}
```

   All that the script does at the moment is to fetch a reference to the *Line Renderer* component of its game object.

6. Set the position of the "Ground" game object to (0,-4,0). Change the "Materials->Element 0" property of the *Line Renderer* component to "Sprite-default". That means that the line will be fully illuminated. Line renderer, as any other rendering object in Unity, can be set to use different materials. This allows you to give your lines different textures, have them react to light, etc. For now, fully illuminated, like the default sprite, should be fine. The default colour for "Sprite-default" is white.

7. Make sure that the "Positions->Size" attribute of the *Line Renderer* is set to 2, and change the "Element0" and "Element1" coordinates to go from point (0,0,0) to (1,0,0) respectively. Line Render will join the points in the "Positions" array...in this case just two points spanning 1 horizontal unit.

8. Make sure the "Use World Space" attribute is turned off, and the "Width" is set to 0.1 on the *Line Renderer* component. You should now see a thin white line near the bottom of your scene.



Ground — Static
Tag Untagged — Layer Default

Transform
Position X 0 Y -4 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Grassland (Script)
Script Grassland

Line Renderer
Cast Shadows On
Receive Shadows ✓
Dynamic Occludee ✓
Motion Vectors Camera Motion Only
Materials
Size 1
Element 0 Sprites-Default
Lightmap Parameters None (Lightmap Parameters)
Positions
Size 2
Element 0 X 0 Y 0 Z 0
Element 1 X 1 Y 0 Z 0
Use World Space ☐
Loop ☐
Width 0.1

9. You can control where and how a line is drawn through a script. We'll need more points - change the "Positions->Size" property of the *Line Render* component on the "Ground" game object to 200. Don't worry about the actual positions values - they will be set properly in the script.

0. Make the following changes to the "Grassland.cs":

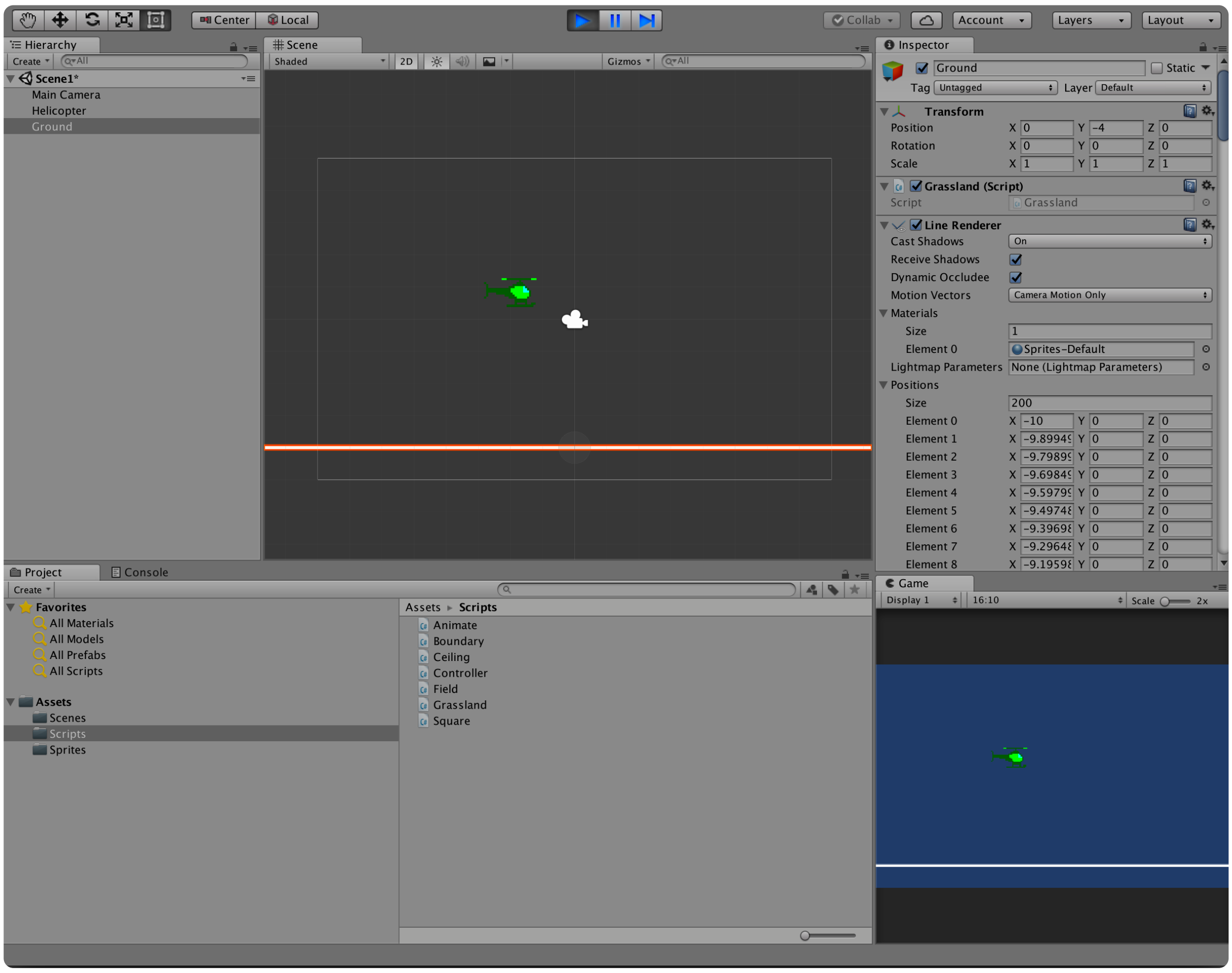
Differences between two versions of Grassland.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(LineRenderer))]
public class Grassland : MonoBehaviour {

   // Reference to line renderer component
   private LineRenderer   lr;

   //Array of point positions
   private Vector3[] linePoints;

   // Use this for initialization
   void Start () {
      // Fetch the reference to the line renderer component
      lr = GetComponent<LineRenderer>();

      // Get the number of line points
      int numLinePoints = lr.positionCount;
      // Initialise the points position array
      linePoints = new Vector3[numLinePoints];

      // Left-most and right-most points (in local coordinates),
      // between which line is rendered
      float xLeft = -10f;
      float xRight = 10f;

      //Distance between points on the line (calculated to fit exactly
      //numPoints from xLeft to xRight)
      float dx = (xRight-xLeft)/(float) (numLinePoints-1);
      for(int i=0;i<numLinePoints;i++) {
         //Horizontal location of point i
         linePoints[i].x = xLeft+i*dx;
         //Vertical location of point i
         linePoints[i].y = 0;
         // Z-coordinate of point i (taken from game object's
         // z coordinate)
         linePoints[i].z = transform.position.z;

      }

      // Set the line renderer points to the positions from point array
      lr.SetPositions(linePoints);

   }

   // Update is called once per frame
   void Update () {

   }
}
```

```
}
```

The script now defines an array of *linePositions*. On *Start()* it fetches the number of points of the *Line Renderer* and spaces all the positions uniformly from x=-10 to x=10. The height of the position point is kept at 0 and the z coordinate of the line is kept the same as the z position of the game object. The positions of the liner renderer are then taken from the *linePositions* array.

1. If you play the game now, you should get a white line spanning the screen near the bottom. Recall that the y positions of the "Ground" game object has been set to -4, and you have instructed the *Line Renderer* to not to use "World Space Coordinates", and so the point positions of the line are all relative to the location of the game object.



Ground — Static
Tag Untagged — Layer Default

Transform
Position X 0 Y -4 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Grassland (Script)
Script Grassland

Line Renderer
Cast Shadows On
Receive Shadows ✓
Dynamic Occludee ✓
Motion Vectors Camera Motion Only
Materials
Size 1
Element 0 Sprites-Default
Lightmap Parameters None (Lightmap Parameters)
Positions
Size 200
Element 0 X -10 Y 0 Z 0
Element 1 X -9.8994 Y 0 Z 0
Element 2 X -9.7989 Y 0 Z 0
Element 3 X -9.6984 Y 0 Z 0
Element 4 X -9.5979 Y 0 Z 0
Element 5 X -9.4974 Y 0 Z 0
Element 6 X -9.3969 Y 0 Z 0
Element 7 X -9.2964 Y 0 Z 0
Element 8 X -9.1959 Y 0 Z 0

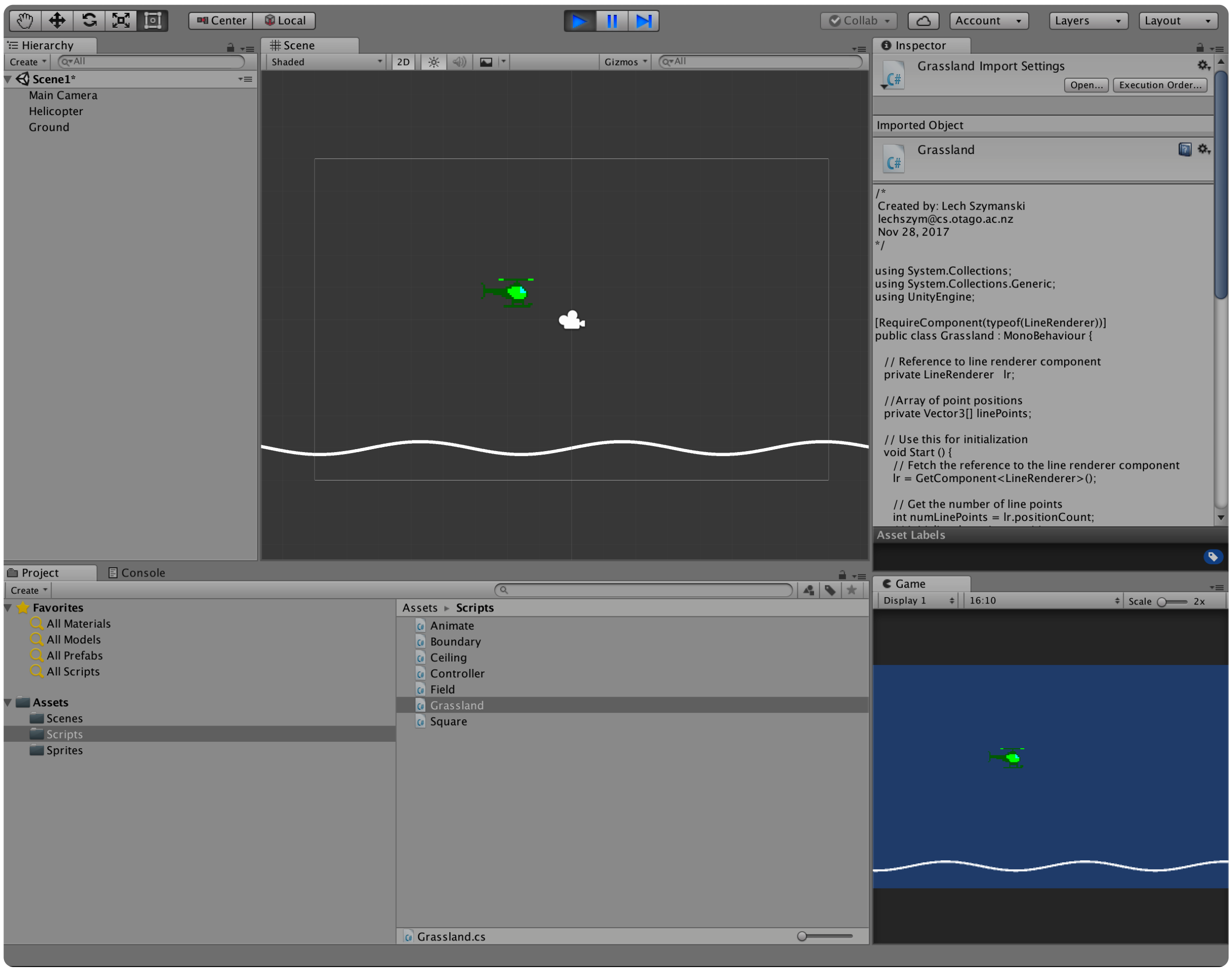
2. Why do wee need 200 hundred points to plot a straight line? Indeed we don't need more than two point for a straight line...but of course we won't make our ground straight. Make the following change to the way the vertical position of the point is computed in the "Grassland.cs" script:

Differences between two versions of Grassland.cs

```
.
.
.
        linePoints[i].y = 0.2f*Mathf.Sin(linePoints[i].x);
.
.
.
```

Instead of keeping the points of the line at the same height, we use a sinusoid to change the height over the range of x values. The range over which the y position varies between the negative and positive value of the amplitude of the sinusoid - in this case from -0.2 to 0.2. The frequency of how fast the y position changes is given by the frequency, which is the multiple of x inside the Mathf.Sin() function - in this case it's 1.

3. If you play the scene you should see few rolling hills.



```
/*
 Created by: Lech Szymanski
 lechszym@cs.otago.ac.nz
 Nov 28, 2017
*/

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(LineRenderer))]
public class Grassland : MonoBehaviour {

  // Reference to line renderer component
  private LineRenderer   lr;

  //Array of point positions
  private Vector3[] linePoints;

  // Use this for initialization
  void Start () {
    // Fetch the reference to the line renderer component
    lr = GetComponent<LineRenderer>();

    // Get the number of line points
    int numLinePoints = lr.positionCount;
```

4. That was simple enough. With few tiny changes we can get the rolling hills to move underneath the helicopter - as if the helicopter was flying over the hills (or wavy body of water, depending how you look at it). Make the following changes to "Grassland.cs":

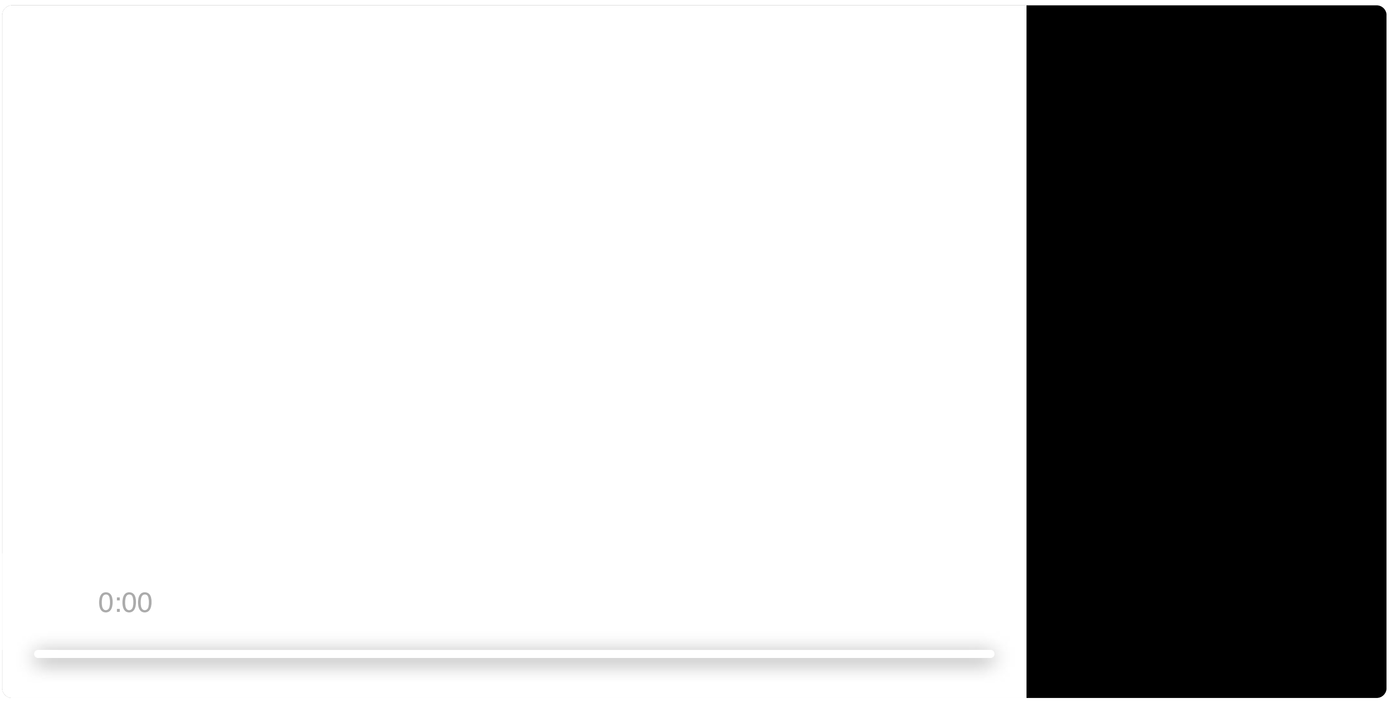
Differences between two versions of Grassland.cs

```
.
.
.
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(LineRenderer))]
public class Grassland : MonoBehaviour {

   // Reference to line renderer component
   private LineRenderer   lr;

   //Array of point positions
   private Vector3[] linePoints;

   // Use this for initialization
   void Start () {
      // Fetch the reference to the line renderer component
      lr = GetComponent<LineRenderer>();

      // Get the number of line points
      int numLinePoints = lr.positionCount;
      // Initialise the points position array
      linePoints = new Vector3[numLinePoints];
   }

   // Update is called once per frame
   void Update () {

      // Get the number of line points
      int numLinePoints = lr.positionCount;
      // Left-most and right-most points (in local coordinates),
      // between which line is rendered
      float xLeft = -10f;
      float xRight = 10f;

      //Distance between points on the line (calculated to fit exactly
      //numPoints from xLeft to xRight)
      float dx = (xRight-xLeft)/(float) (numLinePoints-1);
      for(int i=0;i<numLinePoints;i++) {
         //Horizontal location of point i
         linePoints[i].x = xLeft+i*dx;
         //Vertical location of point i is changing with time
         linePoints[i].y = 0.2f*Mathf.Sin(linePoints[i].x/20f+Time.timeSinceLevelLoad);

         // Z-coordinate of point i (taken from game object's
         // z coordinate)
         linePoints[i].z = transform.position.z;

      }

      // Set the line renderer points to the positions from point arra
```

```
        // Set the line renderer points to the positions from point array
        lr.SetPositions(linePoints);
~~    }~~

~~    // Update is called once per frame~~
~~    void Update () {~~

    }
}
```

The code for setting the points of the line is now in the Update() method, meaning the positions can change from frame to frame. The make the sinusoid move left and right you change it's phase. In this particular example, the phase is set to Time.timeSinceLevelLoad variable...which keeps changing over time. The frequency was also changed, reduced 20 times, by dividing x by 20.

5. Play the game now, the terrain should move underneath the helicopter. It should look like the ground in the video below:



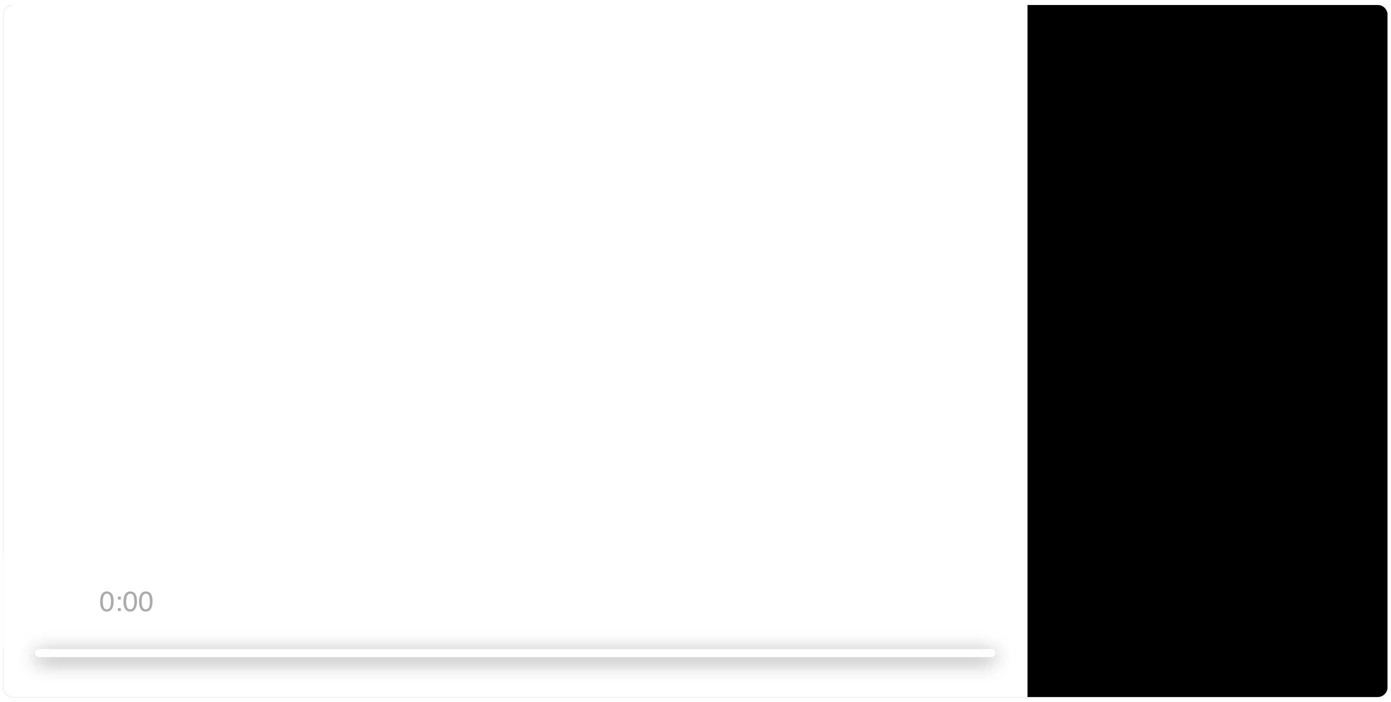
6. Save the scene.
7. Let's see what happens when we make gravity act on the helicopter. Add the *Rigidbody 2D* and *Polygon Collider 2D* component to the "Helicopter" game object.
8. If you play the game now, the helicopter should fall through the ground line, because there is no collider associated with the "Ground" object. Since the line is generated procedurally, and changing over time, so will the collider have to be.
9. Add the *Edge Collider 2D* component to the "Ground" game object.
10. Make the following changes to "Grassland.cs" script:

Differences between two versions of Grassland.cs

```
.
.
.
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(LineRenderer))]
[RequireComponent(typeof(EdgeCollider2D))]
public class Grassland : MonoBehaviour {

   // Reference to line renderer component
   private LineRenderer   lr;

   //Array of point positions
   private Vector3[] linePoints;

   // Reference to the edge collider component
   private EdgeCollider2D ec;
   // Array of edge collider points
   private Vector2[] edgePoints; 

   // Use this for initialization
   void Start () {
      // Fetch the reference to the line renderer component
      lr = GetComponent<LineRenderer>();

      // Get the number of line points
      int numLinePoints = lr.positionCount;
      // Initialise the points position array
      linePoints = new Vector3[numLinePoints];

      // Fetch the reference to the edge collider component
      ec = GetComponent<EdgeCollider2D>();
      // Initialise the edge points array
      edgePoints = new Vector2[numLinePoints];
   }

   // Update is called once per frame
   void Update () {

      // Get the number of line points
      int numLinePoints = lr.positionCount;
      // Left-most and right-most points (in local coordinates),
      // between which line is rendered
      float xLeft = -10f;
      float xRight = 10f;


      //Distance between points on the line (calculated to fit exactly
      //numPoints from xLeft to xRight)
      float dx = (xRight-xLeft)/(float) (numLinePoints-1);
      for(int i=0;i<numLinePoints;i++) {
         //Horizontal location of point i
         linePoints[i].x = xLeft+i*dx;
         //Vertical location of point i is changing with time
         linePoints[i].y = 0.2f*Mathf.Sin(linePoints[i].x/20f+Time.timeSinceLevelLoad);
         // Z-coordinate of point i (taken from game object's
         // z coordinate)
         linePoints[i].z = transform.position.z;

         //Horizontal location of edge point i follows the line
         edgePoints[i].x = linePoints[i].x;
         //Vertical location of edge point i follows the line
         edgePoints[i].y = linePoints[i].y;

      }

      // Set the line renderer points to the positions from point array
      lr.SetPositions(linePoints);

      // Set the edge point to the poisitions from edge point array
      ec.points = edgePoints;
   }
}
```

The edge collider is in essence a line set up in a similar way to the line render, except that it uses 2D coordinates instead of 3D coordinates. The script initialises an array of *Vector2* objects of the same size the points on the line renderer. On Update() very point of the edge collider is made to matches a point of the liner renderer. The z-coordinate is not necessary, because *Edge Collider 2D* only operates in 2D.

1. Play the game now, the helicopter should stay on the ground line and move up and down with it.



2. Save the scene and commit your changes to git.

Inheritance

Next, we'll be putting in a ceiling. Just like the ground it will be made from a line render with an edge collider on top. It will be slightly more complex in terms of the shape. Now, we could just make a copy of the "Grassland.cs" script and put in the required changes. That's not an ideal solution, because this creates multiple copies of the same code. If you decide to change one aspect of the code, you have to do it in many places and that's error prone. It's better to use inheritance. In this exercise you will create a parent (or base) class which provides common functionality for boundaries (such as ground or ceiling) and then have "Grassland.cs", and "Celining.cs" inherit from the base class and add their own specific code.

3. Create a new script and call it "Boundary.cs". From "Grassland.cs" take out (and paste in) the logic that initialises the position arrays for line renderer and the edge collider, and the logic that makes the edge collider follow the line render in the Update(). The script should look like this:

Boundary.cs

```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(LineRenderer))]
[RequireComponent(typeof(EdgeCollider2D))]
public class Boundary : MonoBehaviour {

   // Reference to line renderer component
   protected LineRenderer   lr;
   //Array of point positions
   protected Vector3[] linePoints;


   // Reference to the edge collider component
   private EdgeCollider2D ec;
   // Array of edge collider points
   private Vector2[] edgePoints;

   // Use this for initialization
   protected void Start () {
      // Fetch the reference to the line renderer component
      lr = GetComponent<LineRenderer>();

      // Get the number of line points
      int numLinePoints = lr.positionCount;
      // Initialise the points position array
      linePoints = new Vector3[numLinePoints];

      // Fetch the reference to the edge collider component
      ec = GetComponent<EdgeCollider2D>();
      // Initialise the edge points array
      edgePoints = new Vector2[numLinePoints];
   }

   // Update is called once per frame
   protected void Update () {

      // Get the number of line points
      int numLinePoints = lr.positionCount;

      for(int i=0;i<numLinePoints;i++) {
         //Horizontal location of edge point i follows the line
         edgePoints[i].x = linePoints[i].x;
         //Vertical location of edge point i follows the line
         edgePoints[i].y = linePoints[i].y;

      }
      // Set the edge point to the poisitions from edge point array
      ec.points = edgePoints;
   }
}
```

Note that some of the *private* directives have changed to *protected*. The *private* directive specifies things (methods and internal variables) that are accessible only within the class they have been defined in. And so, anything to do with edge collider remains private, since subclasses are not meant to touch the collider. The *protected* makes the methods and variables accessible from the subclass without making them *public*. Anything to do with line renderer is protected, so that a subclass can specify where the line is, but the parent class will worry about the collider.

4. Next, change "Grassland.cs" to inherit from the *Boundary* class like this:

Differences between two versions of Grassland.cs

```
.
.
.
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

~~[RequireComponent(typeof(LineRenderer))]~~

~~[RequireComponent(typeof(EdgeCollider2D))]~~
~~public class Grassland : MonoBehaviour {~~

    ~~// Reference to line renderer component~~
    ~~private LineRenderer   lr;~~
    ~~//Array of point positions~~
    ~~private Vector3[] linePoints;~~

    ~~// Reference to the edge collider component~~
    ~~private EdgeCollider2D ec;~~
    ~~// Array of edge collider points~~
    ~~private Vector2[] edgePoints;~~

    ~~// Use this for initialization~~
    ~~void Start () {~~
        ~~// Fetch the reference to the line renderer component~~
        ~~lr = GetComponent<LineRenderer>();~~

        ~~// Get the number of line points~~
        ~~int numLinePoints = lr.positionCount;~~
        ~~// Initialise the points position array~~
        ~~linePoints = new Vector3[numLinePoints];~~

        ~~// Fetch the reference to the edge collider component~~
        ~~ec = GetComponent<EdgeCollider2D>();~~
        ~~// Initialise the edge points array~~
        ~~edgePoints = new Vector2[numLinePoints];~~
    ~~}~~

    ~~// Update is called once per frame~~
    ~~void Update () {~~

// Grassland class inherits from the Boundary class
public class Grassland : Boundary {


    // Update() overrides the Update() from the parent class
    new void Update () {

        // Get the number of line points
        int numLinePoints = lr.positionCount;
        // Left-most and right-most points (in local coordinates),
        // between which line is rendered
        float xLeft = -10f;
```

```
      float xRight = 10f;

      //Distance between points on the line (calculated to fit exactly
      //numPoints from xLeft to xRight)
      float dx = (xRight-xLeft)/(float) (numLinePoints-1);
      for(int i=0;i<numLinePoints;i++) {
         //Horizontal location of point i
         linePoints[i].x = xLeft+i*dx;
         //Vertical location of point i is changing with time
         linePoints[i].y = 0.2f*Mathf.Sin(linePoints[i].x/20f+Time.timeSinceLevelLoad);
         // Z-coordinate of point i (taken from game object's
         // z coordinate)
         linePoints[i].z = transform.position.z;

~~         //Horizontal location of edge point i follows the line~~
~~         edgePoints[i].x = linePoints[i].x;~~
~~         //Vertical location of edge point i follows the line~~
~~         edgePoints[i].y = linePoints[i].y;~~

~~      }~~

      }

      // Set the line renderer points to the positions from point array
      lr.SetPositions(linePoints);

~~      // Set the edge point to the poisitions from edge point array~~
~~      ec.points = edgePoints;~~
      // Call to the Update from the parent class
      base.Update();
   }
}
```

There is no need to declare the point arrays in "Grassland.cs", because these variables are inherited from *Boundary*. There is also no need for the Start() method, because it is inherited as well. The *Grassland* class also inherits *Boundary*'s Update() method. However we need specify the positions of the points, which are to be specific to "Grassland.cs", and so we need to override the Update() method (notice the keyword *new* in front of Update(), which confirms that you intend to override). The "Grassland.cs"-specific Update() positions and moves the points of the line renderer. It does not touch the edge collider - all the functionality of the edge collider following the line is in the *base.Update()*, which is the version of the Update() in the base class. Notice that this version is evoked at the end of the new Update(), and so the new Update() does some new stuff and then executes the code in the base Update().

5. Run the scene and make sure that everything still works as it did before.

6. Now, given the *Boundary*, it's relatively easy to create a new script for a different type of boundary. Create a new script and call it "Ceiling.cs". Add the following code:

Ceiling.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Ceiling class inherits from the Boundary class
public class Ceiling : Boundary {

   // Number of sinusoid waves to add up to a
   // single random-looking curve
   private int numWaves=10;

   // Paremeters for each of the sinusoids
   private float[] A;   // Amplitude
   private float[] f;   // Frequence
   private float[] phi; // Phase

   // Start() overrides the Start() from the parent class
   new void Start() {
      // First call the Start() of the parent class to initialise
      // all the inherited internals
      base.Start();

      // Allocate space for a set of numWaves parameters for
      // amplitude, frequence and phase of different sinusoids
      A = new float[numWaves];
      f = new float[numWaves];
      phi = new float[numWaves];

      // Choose each parameter at random from a range of numbers - this
      // will make the composition of numWaves look different each time
      // the scene is run
      for(int i=0;i<numWaves;i++) {
         A[i] = Random.Range(.1f, 5f)/(float) numWaves;
         f[i] = Random.Range(.1f, 5f);
         phi[i] = Random.Range(0f, Mathf.PI);
      }

   }


   // Update is called once per frame
   new void Update () {

      // Get the number of line points
      int numLinePoints = lr.positionCount;
      // Left-most and right-most points (in local coordinates),
      // between which line is rendered
      float xLeft = -10f;
      float xRight = 10f;

      //Distance between point on the line (calculated to fit exactly
      //numPoints from xLeft to xRight)
      float dx = (xRight-xLeft)/(float) (numLinePoints-1);
      for(int i=0;i<numLinePoints;i++) {
         //Horizontal location of point i
         linePoints[i].x = xLeft+i*dx;
         //Vertical location of point i is changing with time, which is a sum of
         //ten sinuosoid of various amlitude, frequency and phase
         linePoints[i].y = 0;
         for(int j=0;j<numWaves;j++) {
             linePoints[i].y += A[j]*Mathf.Sin(f[j]*(linePoints[i].x+Time.timeSince
         }

         // Z-coordinate of point i (taken from game object's
         // z coordinate)
         linePoints[i].z = transform.position.z;
      }
      // Set the line renderer points to the positions from point array
      lr.SetPositions(linePoints);

      // Call to the Update from the parent class
      base.Update();
   }
}
```

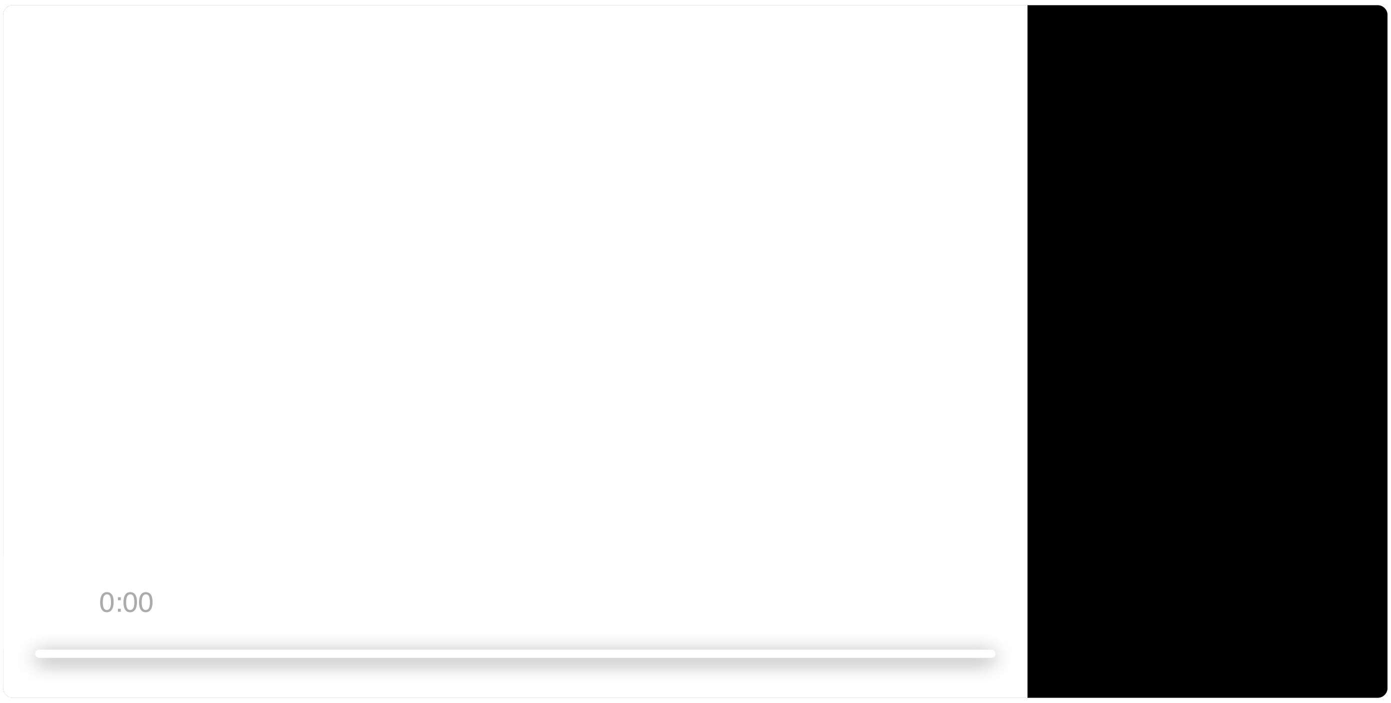
```
























```

The *Ceiling* class, like the *Grassland* class, inherits variables and methods from the *Boundary* class. The only thing that it needs to do is to specify where the line goes. This is still done in the Update() method, with a call to base class Update() at the end. However, this time, to produce a random pattern on the ceiling, the position of points is computed from the composition of a random set of 10 waves. For this the class needs 10 instances of amplitudes, frequencies and phases chosen at random from a certain range. Since this has to happen on Start(), the *Ceiling* class overrides the Start() method of the *Boundary*. It still calls the *Boundary*'s Start() to initialise all the inherited variables, then adding initialisation for the 10 instances of the amplitude, frequency and phase of different sinusoid.

7. Create a new game object in the scene and call it "Ceiling". Set its position to (0,3,0). Add the "Ceiling.cs" script component. Note that the *Line Renderer* and *Edge Collider 2D* get added automatically (due to directives inherited from "Boundary.cs".

8. Make sure to set the *Line Render*'s properties to Sprites-Default material, 200 points, 0.1 width, and to use local space coordinates (that is, disable the "Use World Space" option).

9. Run the scene - you should get a nice random pattern as the ceiling, moving along with the ground.

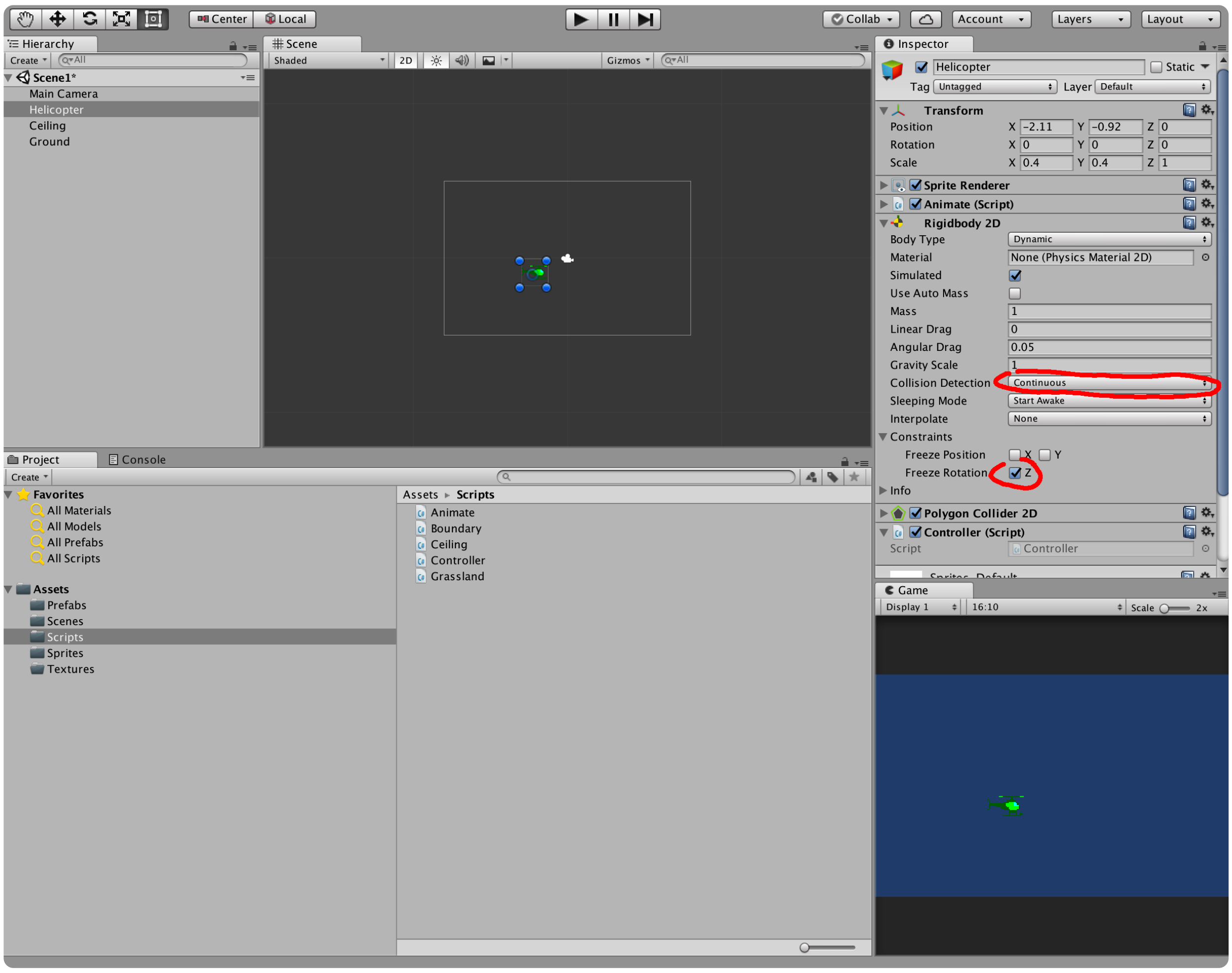


In order to test that the edge collider works over the ceiling, you need to add controls to the "Helicopter" to make it move up. In *Assets/Scripts* you'll find "Controller.cs" script - add it to the the "Helicopter" game object. Now, when playing, you can press Space to make the helicopter go up.

Controller.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
public class Controller : MonoBehaviour {

   // Reference to the rigid body component
   private Rigidbody2D rb;

   private float lift = 20f;

   void Start () {
      // Get reference to the rigid body component
      rb = GetComponent<Rigidbody2D>();
   }

   void FixedUpdate() {

      // Check whether jump button has been pressed
      if(Input.GetButton("Jump")) {
         // Add current lft as a force (will push the gameobject up)
         rb.AddForce(Vector2.up*lift);
      }
   }
}
```

1. Does the helicopter roll when you hit the ceiling at an angle? Does it (if you come up fast) seem to get over the ceiling collider a bit? To fix the roll problem expand the "Constraints" property of "Helicopter"'s *Rigidbody 2D* component and put a check the "Freeze Rotation" option. The collision problem can be fixed by changing the "Collision Detection" setting of the *Rigidbody 2D* from "Discrete" to "Continuous" (https://docs.unity3d.com/ScriptReference/Rigidbody-collisionDetectionMode.html).



```
Helicopter                          Static
Tag Untagged          Layer Default
Transform
  Position   X -2.11  Y -0.92  Z 0
  Rotation   X 0      Y 0      Z 0
  Scale      X 0.4    Y 0.4    Z 1
Sprite Renderer
Animate (Script)
Rigidbody 2D
  Body Type            Dynamic
  Material             None (Physics Material 2D)
  Simulated            ✓
  Use Auto Mass        ☐
  Mass                 1
  Linear Drag          0
  Angular Drag         0.05
  Gravity Scale        1
  Collision Detection  Continuous
  Sleeping Mode        Start Awake
  Interpolate          None
  Constraints
    Freeze Position    ☐ X  ☐ Y
    Freeze Rotation    ✓ Z
  Info
Polygon Collider 2D
Controller (Script)
  Script               Controller
```

2. Save the scene and commit all changes to git.

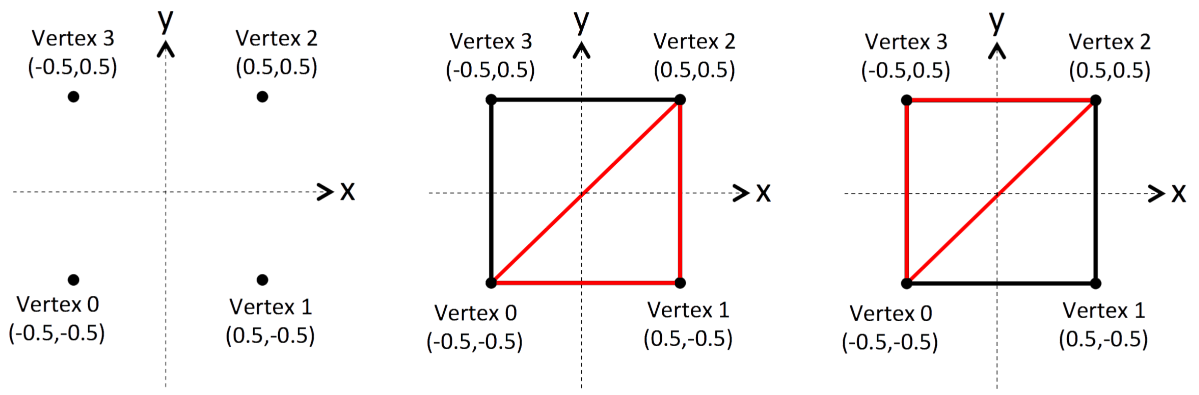
Drawing meshes

Drawing lines is nice, and hopefully you got a bit of sense of how to get a bit of procedural content into your game. Next, we'll go over how to draw solid objects, which in Unity you accomplish with meshes.

Mesh is the next step up from the line renderer. You specify points coordinates, as you did for the line, but this time you also specify which of the points are to be treated as triangles, which Unity will fill out with a material (https://docs.unity3d.com/Manual/Materials.html). Different materials can give all kinds of different looks to the objects created with a mesh. In fact, a Sprite game object in Unity is just a mesh, where the material is the image you intend to display. In this exercise you will create a circular mesh, which is meant to serve as a shield/field effect for the helicopter.

3. Let's start with something very simple - let's draw a square. The Ffigure below shows how to construct a square from triangles. We start by specifying the positions of 4 points in a 2D-coordinate system, call them vertices and index them from 0 to 3. Then, to define two triangles, we specify two sets of 3 points by their

index: in this case we will want triangles made from vertices 0,1,2 and 0,2,3. That's pretty much all it takes to specify a mesh.



1. You could use a tool like Blender to create the mesh and then port it to Unity. But we want to do this with code. The script below shows how the above mesh procedurally:

Square.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(MeshFilter))]
[RequireComponent(typeof(MeshRenderer))]
public class Square : MonoBehaviour {

   // References to mesh filter and renderer components
   MeshFilter  mf;
   MeshRenderer mr;

   // Array specifying mesh vertices and triangles
   private Vector3[] vertices;
   private int[] triangles;

   // Use this for initialization
   void Start () {
      // Get references to the mesh filter and renderer components
      mf = GetComponent<MeshFilter>();
      mr = GetComponent<MeshRenderer>();

      // Create new mesh and attach it to the mesh filter
      Mesh mesh = new Mesh();
      mf.mesh = mesh;

      // Specify the cooridinates (local space) of the four vertices
      // of a square
      vertices = new Vector3[4];
      vertices[0] = new Vector3(-0.5f,-0.5f,transform.position.z);   //Vertex 0
      vertices[1] = new Vector3(0.5f,-0.5f,transform.position.z);    //Vertex 1
      vertices[2] = new Vector3(0.5f,0.5f,transform.position.z);     //Vertex 2
      vertices[3] = new Vector3(-0.5f,0.5f,transform.position.z);    //Vertex 3

      // Specify 2 triangles that will make up the square
      int numTriangles = 2;
      triangles = new int[3*numTriangles];

      // First triangle with vertices 0,1,2
      triangles[0] = 0;
      triangles[1] = 1;
      triangles[2] = 2;

      // Second triangle with vertices 0,2,3
      triangles[3] = 0;
      triangles[4] = 2;
      triangles[5] = 3;

      // Set the vertices and triangles in the mesh
      mesh.vertices = vertices;
      mesh.triangles = triangles;
```

```
        // Set the colour of the square to red
        mr.material.color = new Color(1f,0f,0f);
    }
}

```

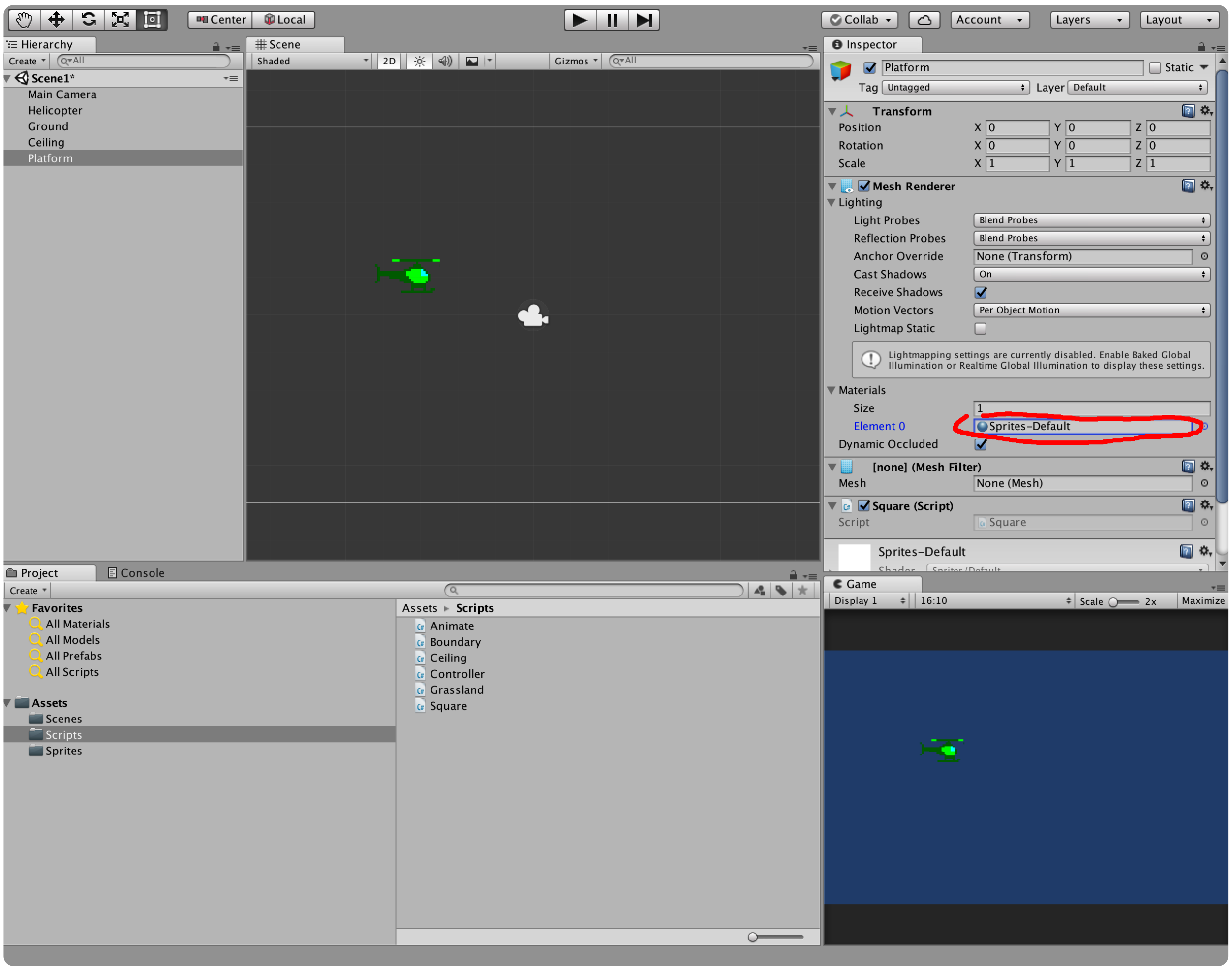
In order to draw a mesh, a game object needs need a *Mesh Renderer* and a *Mesh Filter* components - which are specified as required before the definition of the *Square* class. Aside from references to those two components, the class also defines two arrays - one fro storing the point coordinates of the vertices of the mesh, the other for vertex indexes that define the triangles.

On Start(), the script fetches the references to the mesh components, and creates an instance of a Mesh class that the mesh filter needs in order to render a mesh. The vertices array is initialised to hold 4 vertices. In Unity, mesh vertices are specified in 3D, but since we are creating a 2D component, the z-coordinates will be inherited from game objects z position. The four vertices get assigned the coordinates as specified in the figure above.

The triangles array is just an array of indexes - this array must be 3 times as big as there are triangles, since each triangle is made up of three vertices. That's why the array is initialised to 3*numTriangles. The first set of 3 indexes makes a triangle from vertices 0,1,2 and the second set makes a triangle from vertices 0,2,3. Then the mesh object's vertices and triangles variables (which we mirrored with the vertices and triangles arrays), are set to point to our arrays. Finally, the colour of the material is specified as (1,0,0) RGB value, with is a code for red.

5. In the **Hierarchy panel**, create an empty game object and name it "Platform". Create the "Square.cs" script using the code as given above and add it as a component to the "Platform" object.

6. Set the material of the *Mesh Renderer* to "Sprite-default".



Platform — Static
Tag Untagged — Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Mesh Renderer
Lighting
Light Probes: Blend Probes
Reflection Probes: Blend Probes
Anchor Override: None (Transform)
Cast Shadows: On
Receive Shadows: ✓
Motion Vectors: Per Object Motion
Lightmap Static: ☐

Lightmapping settings are currently disabled. Enable Baked Global Illumination or Realtime Global Illumination to display these settings.

Materials
Size: 1
Element 0: Sprites-Default
Dynamic Occluded: ✓

[none] (Mesh Filter)
Mesh: None (Mesh)

Square (Script)
Script: Square

Sprites-Default
Shader: Sprites/Default

7. Play the game. You should see a red square somewhere on the screen (depending where you specified the position of the "Platform" game object).

8. Specifying a 2D collider over a mesh can be done in an almost identical way to how you did it with the *Line Renderer*, except it makes more sense to use a *Polygon Collider 2D* as opposed to the *Edge Collider 2D*. For the *Polygon Collider 2D* you also specify set of points, but the polygon collider joins the first and last points to close its geometry, thus also creating a notion of *inside* and *outside*, in addition to crossing the collider line. Modify the "Square.cs" script as shown below to lay the collider out exactly over the generated square. Don't forget to add *Polygon Collider 2D* component to the "Platform" game object.

Differences between two versions of Square.cs

```
.
.
.
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(MeshFilter))]
[RequireComponent(typeof(MeshRenderer))]
[RequireComponent(typeof(PolygonCollider2D))]
public class Square : MonoBehaviour {

   // References to mesh filter and renderer components
   MeshFilter  mf;
   MeshRenderer mr;

   // Array specifying mesh vertices and triangles
   private Vector3[] vertices;
   private int[] triangles;

   // Reference to the polygon collider
   PolygonCollider2D    pc;

   // Array of polygon points
   private Vector2[] points;

   // Use this for initialization
   void Start () {
      // Get references to the mesh filter and renderer components
      mf = GetComponent<MeshFilter>();
      mr = GetComponent<MeshRenderer>();

      // Create new mesh and attach it to the mesh filter
        Mesh mesh = new Mesh();
      mf.mesh = mesh;


      // Specify the cooridinates (local space) of the four vertices
      // of a square
      vertices = new Vector3[4];
      vertices[0] = new Vector3(-0.5f,-0.5f,transform.position.z);   //Vertex 0
      vertices[1] = new Vector3(0.5f,-0.5f,transform.position.z);    //Vertex 1
      vertices[2] = new Vector3(0.5f,0.5f,transform.position.z);     //Vertex 2
      vertices[3] = new Vector3(-0.5f,0.5f,transform.position.z);    //Vertex 3

      // Specify 2 triangles that will make up the square
      int numTriangles = 2;
      triangles = new int[3*numTriangles];

      // First triangle with vertices 0,1,2
      triangles[0] = 0;
      triangles[1] = 1;
```

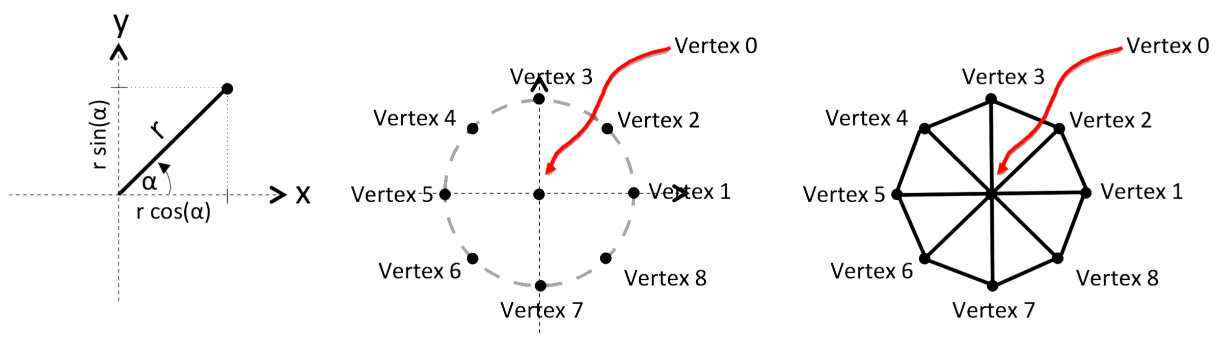
```
        triangles[2] = 2;

        // Second triangle with vertices 0,2,3
        triangles[3] = 0;
        triangles[4] = 2;
        triangles[5] = 3;

        // Set the vertices and triangles in the mesh
        mesh.vertices = vertices;
        mesh.triangles = triangles;

        // Set the colour of the square to red
        mr.material.color = new Color(1f,0f,0f);

        // Get the reference to the polygon collider
        pc = GetComponent<PolygonCollider2D>();

        // Set the polygon point of its path to
        // the same coordinates as those of the mesh vertices
        points = new Vector2[vertices.Length];
        for(int i=0;i<vertices.Length;i++) {
            points[i].x = vertices[i].x;
            points[i].y = vertices[i].y;
        }
        pc.SetPath(0, points);
    }
}
```

Polygon collider is allowed to have open spaces within its collider, and so rather than just assigning a set of points to it at the end, you assign a path of points that will be joined into a closed figure. In the script above we need only one path made up of the points that fall on the vertices of the square.

9. Position the platform somewhere underneath the helicopter and run the scene to check if the collider works.

0. Disable the "Platform" game object. We're going to create a bit more complicated mesh that traces out a circle.

1. Probably the easiest way to triangulate the circle is to put one vertex in the centre, use the remaining points to trace out the circumference at equally spaced intervals, and then define triangle joining each pair of consecutive vertices on the circumference to the centre. It should works as shown in the figure below.



The first diagram in the figure shows how to compute the x and y coordinates of a vertex on a circle's circumference. Given a circle's radius *r* and angle ***α*** (which is the angle between the horizontal axis and the line joining the origin and the point on the circumference) the x and y co-ordinates of that points are given by *r*cos**α*** and *r*sin **α*** respectively. This angle ***α*** must be given in radians. In order to trace out the set of points around the circle's circumference at evenly space intervals you can divide *2 **α*** (recall, *2 **α*** radians=360 degrees) by the number of points and then take different multiples of ***α*** to compute the co-ordinates of the points. The last image in the figure shows the triangulation.

2. Create "Field.cs" script and type in the following code:

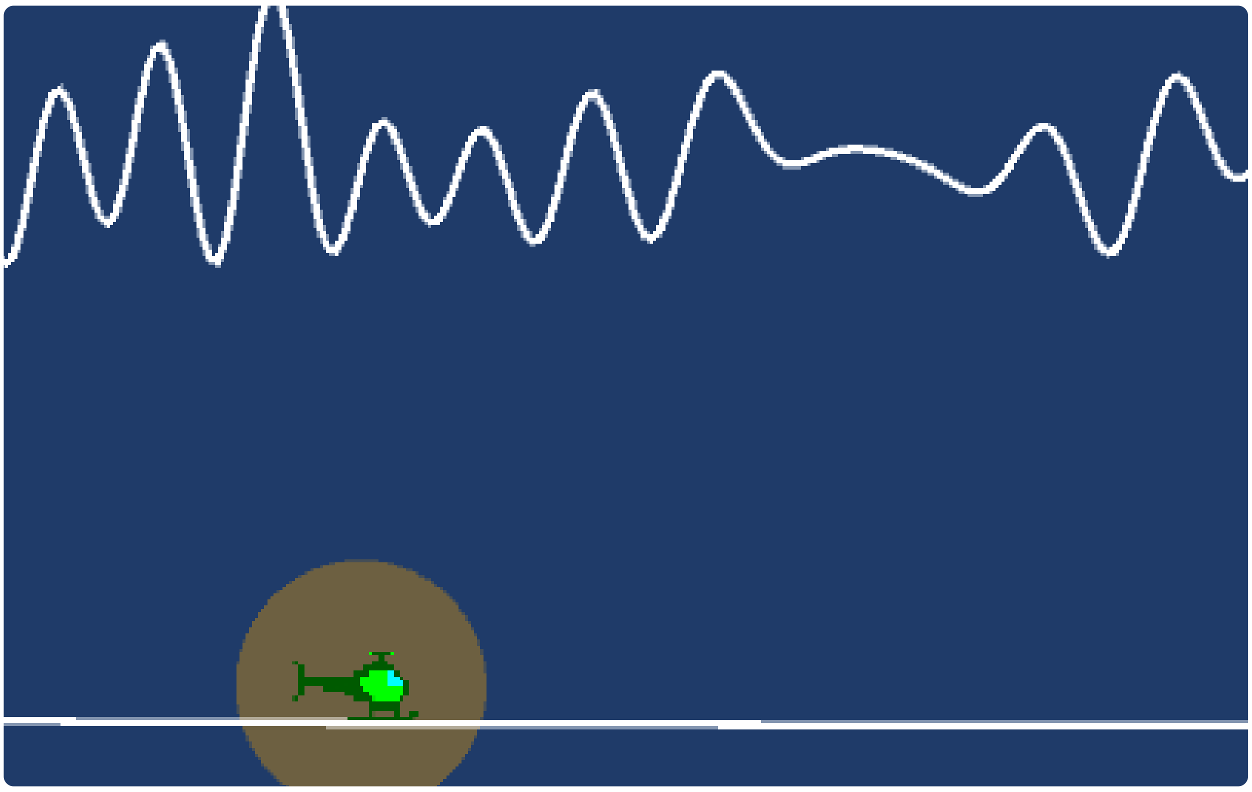
Field.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(MeshFilter))]
[RequireComponent(typeof(MeshRenderer))]
public class Field : MonoBehaviour {

   // References to mesh filter and renderer components
   MeshFilter  mf;
   MeshRenderer mr;

   // Array specifying mesh vertices and triangles
   private Vector3[] vertices;
   private int[] triangles;

   // Reference to mesh
   private Mesh mesh;

   // Radius of the circle
   float radius = 4f;
   // Number of points on circle's circumference
   int numPoints = 64;

   // Use this for initialization
   void Start () {
      // Get references to the mesh filter and renderer components
      mf = GetComponent<MeshFilter>();
      mr = GetComponent<MeshRenderer>();

      // Create new mesh and attach it to the mesh filter
      mesh = new Mesh();
      mf.mesh = mesh;

      // Set circle colour (and make it a bit transparent)
      mr.material.color = new Color(0.8f,0.6f,0.2f,0.5f);

      // Initialise array for vertices and triangles (one extra
      // vertex is taken for the centre of the circle)
      vertices = new Vector3[numPoints+1];
      triangles = new int[numPoints*3];
   }

   void Update() {

      // Compute the angle between two triangles in the cricle
      float delta = 2f*Mathf.PI/(float) (numPoints-1);
      // Stat with angle of 0
      float alpha = 0f;

      // The center vertex (index 0) is at location (0,0)
      vertices[0].x = 0;
      vertices[0].x = 0;
      vertices[0].y = 0;
      vertices[0].z = transform.position.z;

      //Other vertices will be positioned evenly around the circle
      for(int i=1;i<=numPoints;i++) {
         //Radius and alpha give a position of a point around
         //the circle in spherical coordinates

         // Compute position x from spherical coordinates
         float x = radius*Mathf.Cos(alpha);
         // Compute position y from spherical coordinates
         float y = radius*Mathf.Sin(alpha);

         // Set the vertex values
         vertices[i].x = x;
         vertices[i].y = y;
         vertices[i].z = transform.position.z;

         //Specify the triangle going from 0 vertex (centre) to
         //the i vertex and the previous vertex on the circle
         triangles[(i-1)*3] = 0;
         if(i==1) {
            // If current vertex is 1, then previous vertex is the
            // last vertex (to close the cricle)
            triangles[(i-1)*3+1] = numPoints;
         } else {
            triangles[(i-1)*3+1] = i-1;
         }
         triangles[(i-1)*3+2] = i;

         // Increase the angle to get the next positon around the circle
         alpha += delta;
      }
      // Set the new vertices and triangles in the mesh
      mesh.vertices = vertices;
      mesh.triangles = triangles;
   }
}
```

This script creates a circular mesh as shown in the diagram above. However, it uses 64 points around the circumference. Note that the position of points is done in the Update() method. That's because we will be changing these co-ordinates on the fly.

3. Create a child game object under the "Helicopter" and name it "Field". Add the "Field.cs" script component to it. Don't forget to set the "Material" of the *Mesh Renderer* to "Sprite-default". Set the position of the "Field" game object to (0,0,1) (to make it appear behind the helicopter).

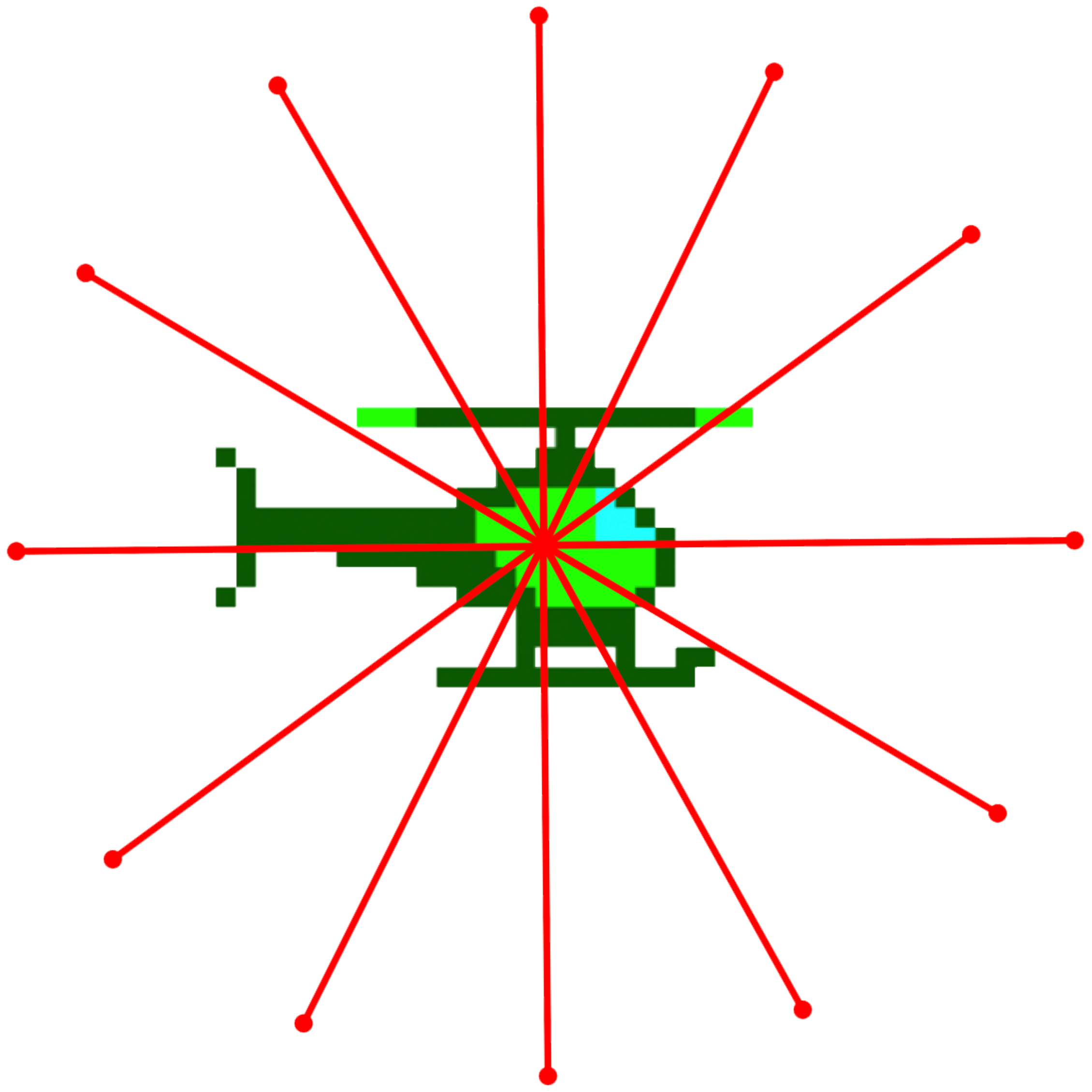
4. Run the game. There should be now a light-brown, semi-transparent field behind the helicopter - that's our new created circular mesh.



5. Now why go through all this trouble of creating a circular mesh, when we could have just use a sprite instead? Well, the main advantage of procedurally generated content is that you can change it dynamically. Let's suppose we wanted the field to interact with the colliders of the floor and the ceiling. Perhaps it's meant to work kind of like a field of view, and it cannot pass through solid object. It would be very hard to do this with sprites, but is relatively easy with a mesh object whose vertices are under the control of a script. However, before altering the shape of the mesh, we need to detect when it collides with other objects. Using a *Polygon Collider 2D* with its points changing along with the mesh on very Update() would potentially work...but in similar situations ray-casting is the method of choice for collision detection. Save the scene and commit all your work to git.

Ray-casting

Ray-casting is a rather fast (computationally) method of collision detection. It comes in handy when you need collisions over objects that change shape. In ray-casting you define a line (or a ray) by specifying it's origin point, direction, and length. If that line hits a collider, you get an event. For our helicopter, we will be shooting out several rays from the centre of the helicopter in the direction of the points on the circumference of the mesh of our circular field. If a ray hits a collider associated with a ceiling or ground the corresponding mesh point will be adjusted to not go beyond the collider in question.



5. Unity provides a ray casting API (https://docs.unity3d.com/ScriptReference/Physics2D.Raycast.html). In Unity the 3D and 2D colliders are incompatible - that is, a 3D collider doesn't interact with a 2D collider and vice versa. Hence, there are two versions of ray casting API - one for 3D and the other for 2D collisions. Change the code of "Field.cs" as follows:

Differences between two versions of Field.cs

```
.
.
.
      // The center vertex (index 0) is at location (0,0)
      vertices[0].x = 0;
      vertices[0].y = 0;
      vertices[0].z = transform.position.z;

      // Specify the layer mast for ray casting - ray casting will
      // only interact with layer 8
      int layerMask = 1 << 8;

      //Other vertices will be positioned evenly around the circle
      for(int i=1;i<=numPoints;i++) {
         //Radius and alpha give a position of a point around
         //the circle in spherical coordinates

         // Compute position x from spherical coordinates
         float x = radius*Mathf.Cos(alpha);
         // Compute position y from spherical coordinates
         float y = radius*Mathf.Sin(alpha);

         // Create a ray
         Vector2 ray = new Vector2(x,y);
         // Cast the ray
         RaycastHit2D hit = Physics2D.Raycast(transform.position,ray,ray.magnitude,layerM
         // Check if ray has hit something, if yes, check how far from the ray's origin p
         // and adjust the distance of where the mesh point is going to be located.
         if (hit.collider != null) {
            float distance = hit.distance;
            x = distance*Mathf.Cos(alpha);
            y = distance*Mathf.Sin(alpha);
         }

         // Set the vertex values
         vertices[i].x = x;
         vertices[i].y = y;
         vertices[i].z = transform.position.z;

         //Specify the triangle going from 0 vertex (centre) to
.
.
.
```

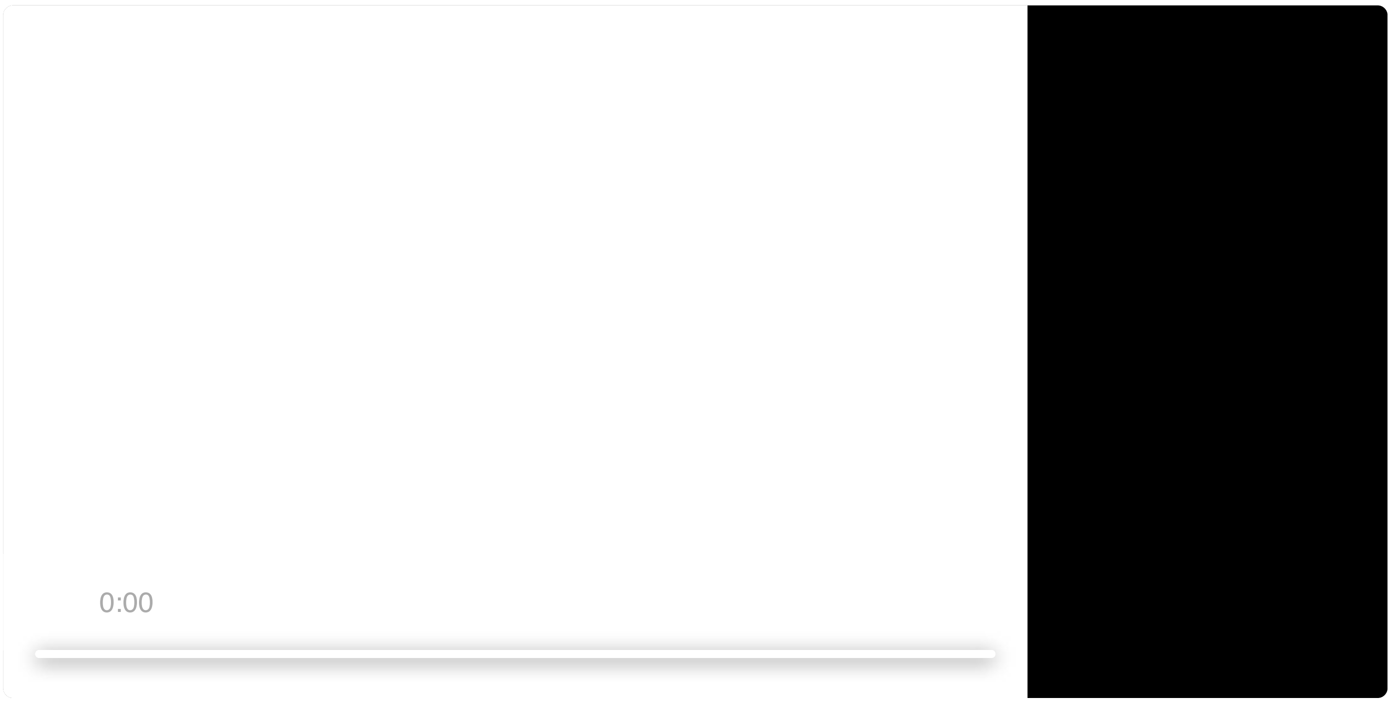
The ray-casting API allows you to specify a layer mask, which instructs which colliders to interact with. If you don't specify the mask, the rays will collide with the collider of the "Helicopter" game object and you will always get a hit. The masking is done by setting bits of a 32-bit integer. The line *int layerMask = 1 << 8;* clearing all the bits of a 32-bit integer to 0 except for the 8th bit. If you wanted to also set the 10th bit, you would do *int layerMask = 1 << 8 | 1 << 10;*.

The actual ray-casting takes place inside the for loop, because we need to cast a ray in every direction along a point on the circumference of the circular mesh. The vector direction of the ray takes exactly same co-ordinates as the co-ordinates of the vertex on the circle. We also want to ray to be only cast only up to the location of that vertex on the circle, but not farther. Ray casting is done in world co-ordinates, and so the starting position is the position of the game object (which is also the centre of the field circle), cast it in the direction of x and y co-ordinates as given for the mesh vertex, make it as long as the magnitude of that vector and include the layer mask defined previously. If the ray hits something it returns a non-null hit, which then can be queried for the distance where the hit occured. If that happens, the co-ordinates of that particular vertex get adjusted to go no further than that distance.

7. Next you need to assign the "Ground" and "Ceiling" game objects to be part of layer mask 8. Select one of those object and in the **Inspector panel** expand its "Layer" options (right next to the "Tag" option). Select "Add Layer..." and give a name to location 8 - I called it "Obstacles". The name is not important, but the number next to it, which specifies which bit needs to be set in the layer mask in order for ray cast to interact with that layer. In your code we used the 8th bit, so you want to give a meaningful name to the 8-th layer.

8. Make sure to set the "Layer" to "8: Obstacles" for both the "Ceiling" and the "Ground" game objects.

9. Play the game. Does it work? Does the mesh adjust itself to the contours of the ground and ceiling? It does...but something is not quite right. The deformation happens a bit too early, when the helicopter is still quite a distance from the collider.



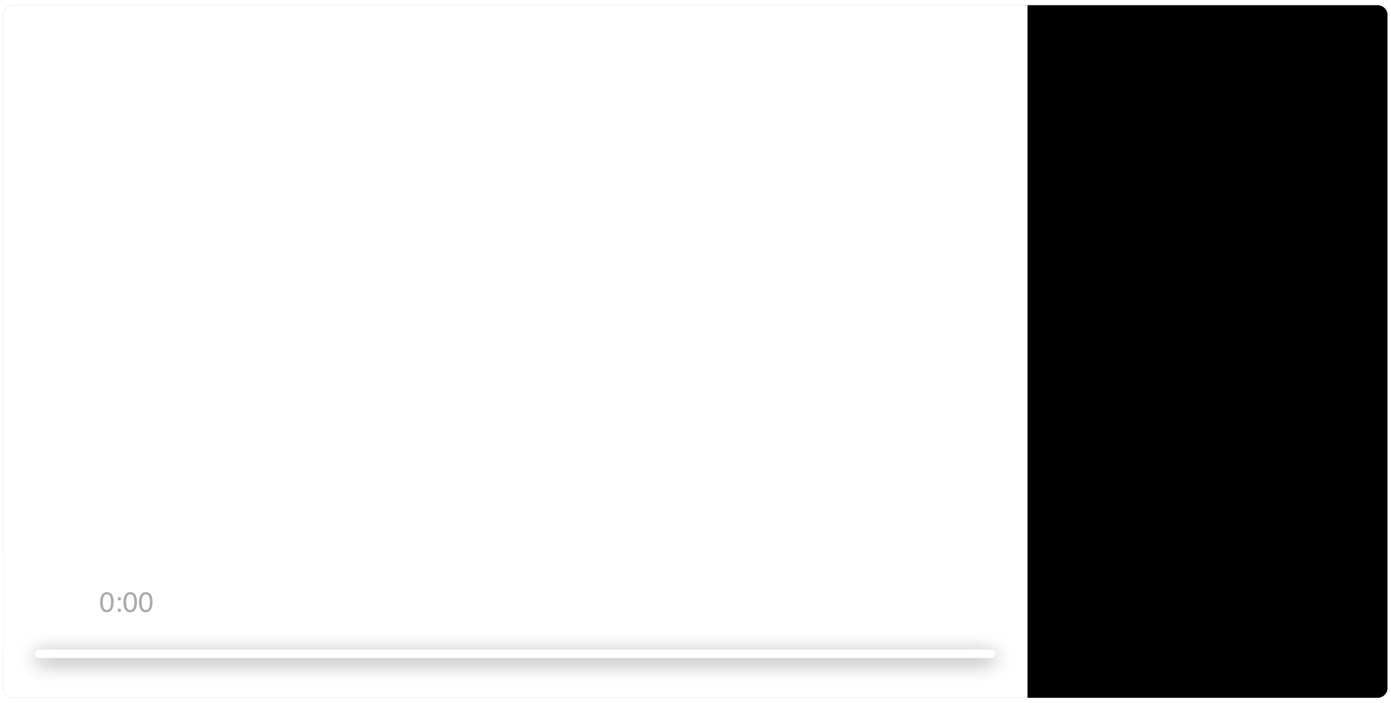
10. Well something is wrong, and it might be hard to figure out what. Perhaps something is not quite right with the ray casting. We need to see what the rays are, and for that you can use Gizmo (https://docs.unity3d.com/ScriptReference/Gizmos.html) visualisations. In Unity, Gizmos are things that give you powerful set of tools for visualisations in the Unity Editor. They only work in the editor, so when you build the game the Gizmos won't be there. Add the following function to "Field.cs":

Differences between two versions of Field.cs

```
.
.
.
    }

    void OnDrawGizmos() {
        Gizmos.color = Color.red;

        // Compute the angle between two triangles in the cricle
        float delta = 2f*Mathf.PI/(float) (numPoints-1);
        // Stat with angle of 0
        float alpha = 0f;

        //Other vertices will be positioned evenly around the circle
        for(int i=1;i<=numPoints;i++) {
            //Radius and alpha give a position of a point around
            //the circle in spherical coordinates 

            // Compute position x from spherical coordinates
            float x = radius*Mathf.Cos(alpha);
            // Compute position y from spherical coordinates
            float y = radius*Mathf.Sin(alpha);

            // Create a ray
            Vector3 ray = new Vector3(x,y, transform.position.z);

            Gizmos.DrawRay(transform.position, ray);

            // Increase the angle to get the next positon around the circle
            alpha += delta;
        }
    }
}
```

First of all, the OnDrawGizmos() function is kind of like the Update() function. It's called on every frame, but it's meant for adding visualisation code for Gizmos. Remember, OnDrawGizmos() will not be executed on the built game, and so you don't want to put any logic there that advances the game state - only things that help you with visualisation of the state for development purposes. Inside the OnDrawGizmos() we are making use of the DrawRay() function, which will draw a ray. The ray starting position and the direction vector are computed exactly as they are in the Update() function...except that DrawRay() expects a 3D vector for direction, and so specify the ray co-ordinates over *Vector3*.

1. Run the game. Can you see the red rays being cast in the Scene tab? Here's how it looked like for me at this stage:



It seems that the collision detection of the casted rays works perfectly...except that the rays are way longer than the radius of the mesh. How is that possible, when we are using the same co-ordinates as the vertices of the mesh? How can the rays be longer than the radius of the mesh? The answer lies in the fact that ray casting uses world co-ordinate system (independent of the game object scale) and the mesh points use local co-ordinates system. Our circular mesh gets scaled by the scale of its parent game object, and if you take a look at the scale setting of the "Helicopter" game object, it's X and Y scale is set to 0.4. The mesh is 60% smaller than its vertex co-ordinates suggest.

Having mesh vertices specified in the local co-ordinate system makes sense - when you scale the "Helicopter" usually you would want to scale its children objects. So, what you need to do in this case, is to take the account of the scale (as dictated by the parent object) when doing ray-casting

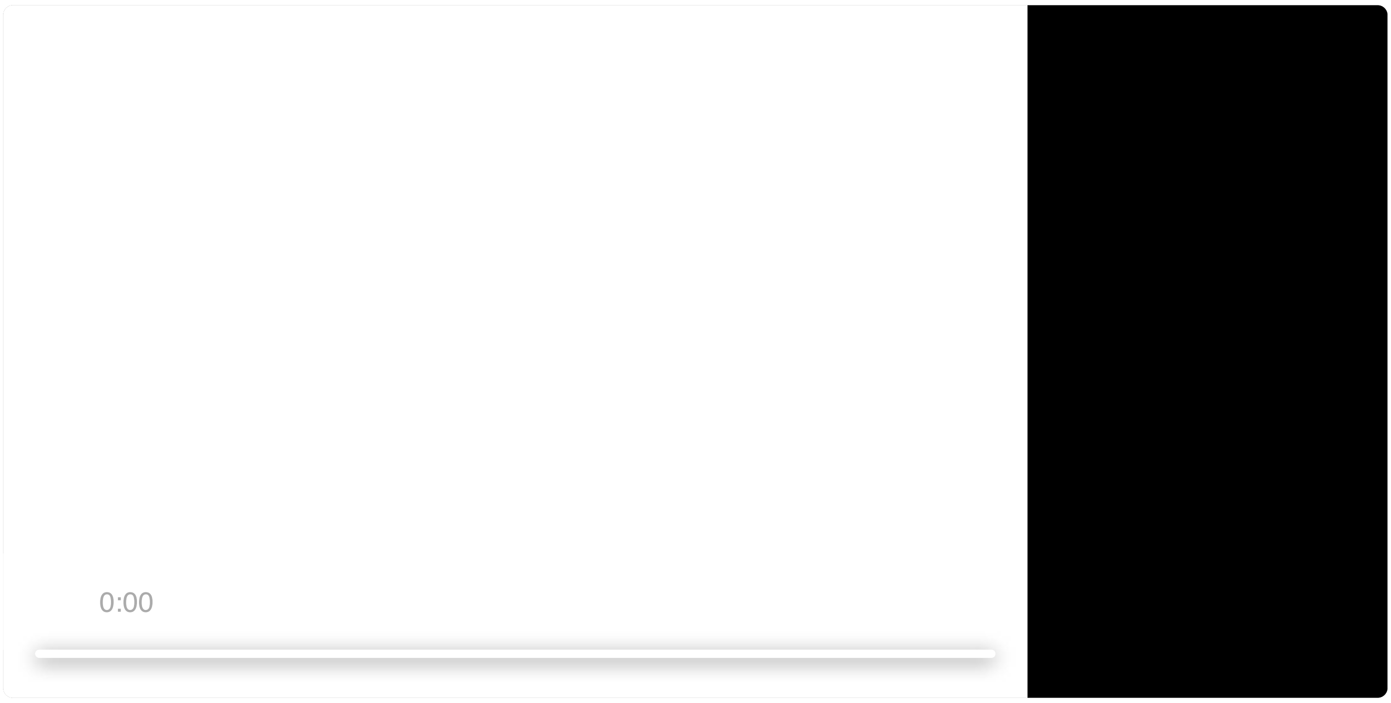
2. Here are the changes to "Field.cs" that will take into account the scale of the parent objects when casting the rays (in both Update() and OnDrawGizmos() methods):

Differences between two versions of Field.cs

```
.
.
.
         //the circle in spherical coordinates

         // Compute position x from spherical coordinates
         float x = radius*Mathf.Cos(alpha);
         // Compute position y from spherical coordinates
         float y = radius*Mathf.Sin(alpha);

         // Create a ray
         Vector2 ray = new Vector2(x,y);
         ray.x *= transform.lossyScale.x;
         ray.y *= transform.lossyScale.y;

         // Cast the ray
         RaycastHit2D hit = Physics2D.Raycast(transform.position,ray,ray.magnitude,layerM
         // Check if ray has hit something, if yes, check how far from the ray's origin p
         // and adjust the distance of where the mesh point is going to be located.
         if (hit.collider != null) {
            float distance = hit.distance;
            x = distance*Mathf.~~Cos(alpha);~~
Cos(alpha)/transform.lossyScale.x;
            y = distance*Mathf.~~Sin(alpha);~~
Sin(alpha)/transform.lossyScale.y;
         }

         // Set the vertex values
         vertices[i].x = x;
         vertices[i].y = y;
         vertices[i].z = transform.position.z;

         //Specify the triangle going from 0 vertex (centre) to
         //the i vertex and the previous vertex on the circle
         triangles[(i-1)*3] = 0;
         if(i==1) {
            // If current vertex is 1, then previous vertex is the
            // last vertex (to close the cricle)
            triangles[(i-1)*3+1] = numPoints;
         } else {
            triangles[(i-1)*3+1] = i-1;
         }
         triangles[(i-1)*3+2] = i;

         // Increase the angle to get the next positon around the circle
         alpha += delta;
      }
      // Set the new vertices and triangles in the mesh
      mesh.vertices = vertices;
      mesh.triangles = triangles;
   }
   void OnDrawGizmos() {
      Gizmos.color = Color.red;

      // Compute the angle between two triangles in the cricle
      float delta = 2f*Mathf.PI/(float) (numPoints-1);
      // Stat with angle of 0
      float alpha = 0f;

      //Other vertices will be positioned evenly around the circle
      for(int i=1;i<=numPoints;i++) {
         //Radius and alpha give a position of a point around
         //the circle in spherical coordinates

         // Compute position x from spherical coordinates
         float x = radius*Mathf.Cos(alpha);
         // Compute position y from spherical coordinates
         float y = radius*Mathf.Sin(alpha);

         // Create a ray
         Vector3 ray = new Vector3(x,y, transform.position.z);

         ray.x *= transform.lossyScale.x;
         ray.y *= transform.lossyScale.y;

         Gizmos.DrawRay(transform.position, ray);

         // Increase the angle to get the next positon around the circle
         alpha += delta;
      }
   }
}
```

The lossyScale method of the transform gets the overall scale as dictated by the game object hierarchy (not just the local scale of the mesh). So if the game object with this script happened to be a child of a object with 0.5 scale, which in turn is a child of an object with scale set to 3, the lossyScale would be 1.5. In this case, the lossyScale will be 0.4, as dictated by the parent object of the "Field" game object. Once a hit is detected, the new x and y co-ordinates are already scaled...but since mesh applies the parent's scale on top of that, we need to divide the vertex position by the lossyScale in order for the mesh vertex to be placed at the dcorrect distance from the "Helicopter".

3. Run the game. Take a look at the ray gizmos drawn in the Scene tab - it should be exactly the length of the circular filed. And in the Game window you should see the circular mesh to adapting itself nicely to the curves of the ceiling and the line barrier of the ground.



Save the scene and commit your work to git.

Assessment

Show your work to the demonstrator for assessment.

Intermediate Challenge

- Did you notice that the script used for the "Ground" object was called "Grassland.cs". Create grass procedurally on the ground. The grass should react to the blades of the helicopter - that is, if the helicopter gets close, the grass should be bent away as if being blown by the blades. If you get this working, it might be not all that much work to have a bit of wind simulation and have the grass blowing this and that way in the wind.

  or

- Devise and complete your own Procedural Programming Intermediate Challenge - just check with the lecturer or the demonstrator first, whether the scope of the work will be sufficient for the awarded skill points.

Assessment

Show your work to the demonstrator for assessment.

Master Challenge

- Write a proper procedural map generator for a game, and implement the layout using mesh renderers. The map has to have certain constraints, which will most likely depend on the mechanics of the game in question. The point is, there cannot be places where a character can get stuck or have no exit path, etc. A good place to start might be looking for some tutorials, and reading about the "Marching Squares" (https://en.wikipedia.org/wiki/Marching_squares) technique for drawing map contours.

  or

- Find a tutorial on Editor programming in Unity and learn how to customise the editor to make it easier to work in Unity without the need for programming. For instance, can you create a way where an artist can supply a sprite sheet, specify sprite sizes, and have the editor produce an entire game object that can run the animation? This would automate the whole process of sprite splicing, creating game object with a script component that can do animation, and automate the process of grabbing the references to all the sprites.

  or

- Learn how to use scripts to improve tiling using Unity's "Tile Palette", so that tiles are selected automatically based on their neighbours. For instance, if your level designer was to layout a road using "Tile Palette" it would be much easier for him/her if the right tile for an intersection or appropriate bend in the road was applied automatically based on what's around.

  or

- Devise and complete your own Procedural Programming master Challenge - just check with the lecturer or the demonstrator first, whether the scope of the work will be sufficient for the awarded skill points.

Assessment

Show your work to the demonstrator for assessment.