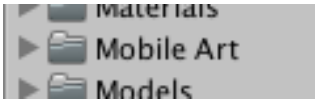
Unity3D - SpaceShooter 1.1

Moving to iOS mobile platform

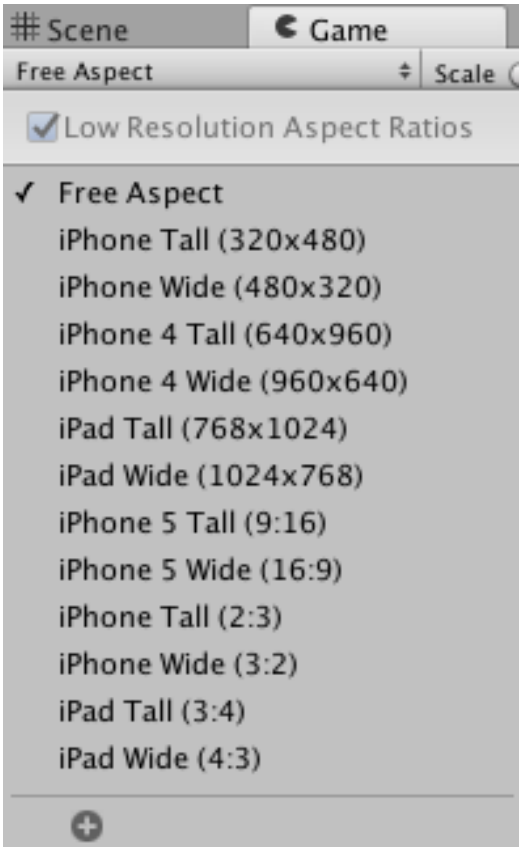
When you adapt a game - or create a brand new one - for mobile platform there are some benefits and some drawbacks. No physical keyboard, only touchscreen and accelerometer are the usual input interfaces.

- Get started with the SpaceShooter 1.0
- Download the mobile artwork assets from https://oc.unity3d.com/index.php/s/4c0a290a55fc27d2a6de8936cc4a66ed/download
- Drag the downloaded *Mobile Art* folder into the Project window

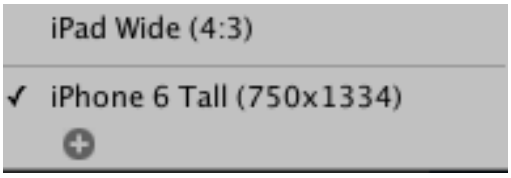


Build Settings

- From *File > Build Settings* switch the project to iOS platform
- From the Game view, click the “Free Aspect” button and choose a more appropriate aspect ratio for your target screen



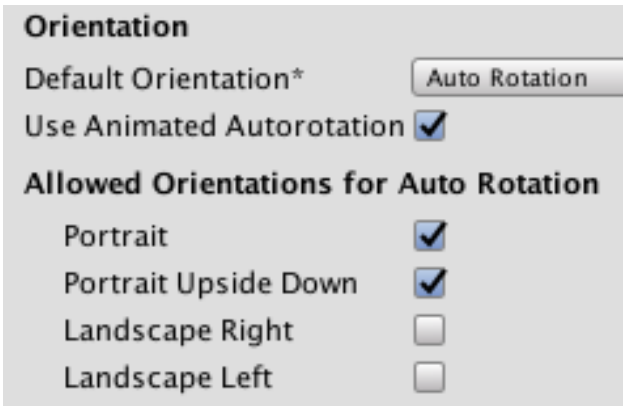
If needed, add a new configuration



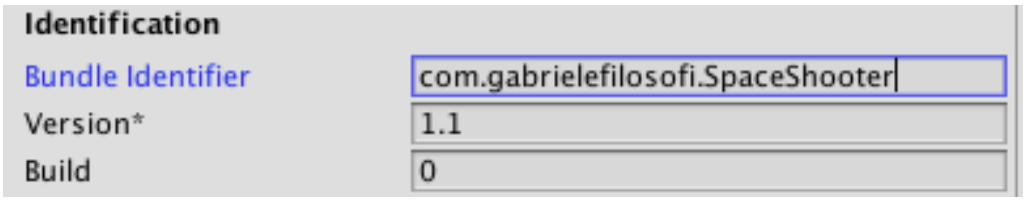
Note that switching the platform back to MacOS X the *Free Aspect* screen is restored automatically

- From *Edit* > *Project Settings* > *Player*
- Check Company and Product names

- Check *Resolution and Presentation* options for a iPad or iPhone, only portrait mode



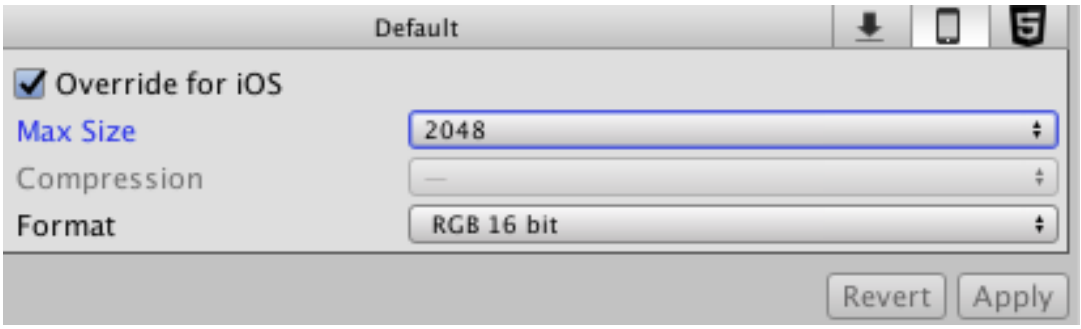
- Check *Other Settings* options



Optimise texture size and format settings

Texture assets are relatively small size, the only exception being represented by the background. In mobile applications the compression settings are generally

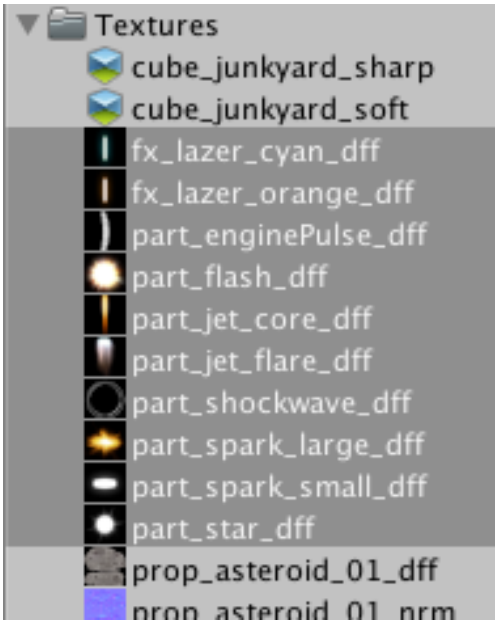
more harsh.
Select *Assets/Textures/tile_nebula_green_dff*
For the iOS target platform select this option



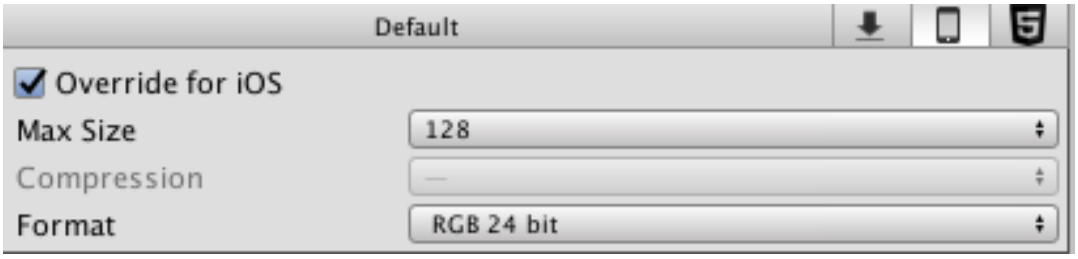
which will result in a reduced file size

Note that you can set different format and Max size for each target platform. When you switch the platform, the correct settings override the default settings

For the smaller textures, we can improve the quality without a significative overhead. Then select the following textures



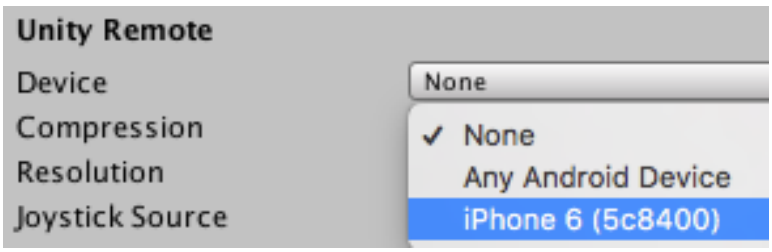
and choose uncompressed truecolor



Unity Remote

Unity Remote is a downloadable app designed to help with Android, iOS and tvOS development. The app connects with Unity while you are running your project in Play Mode from the editor. The visual output from the editor is sent to the device's screen, and the live inputs are sent back to the running project in Unity. This allows you to get a good impression of how your game really looks and handles on the target device, without the hassle of a full build for each test.

- Download and install Unity Remote 5 from the Apple Store on your iOS device
- Connect the iOS device to the Mac via Lightning USB cable
- From E*dit > Project Settings > Editor* select the device



Now we can enter play mode and see the game view on the device's screen, however we need to change the PlayerController script in order to accept input control from the device

PlayerController script

- Edit the PlayerController script and change the FixedUpdate function this way

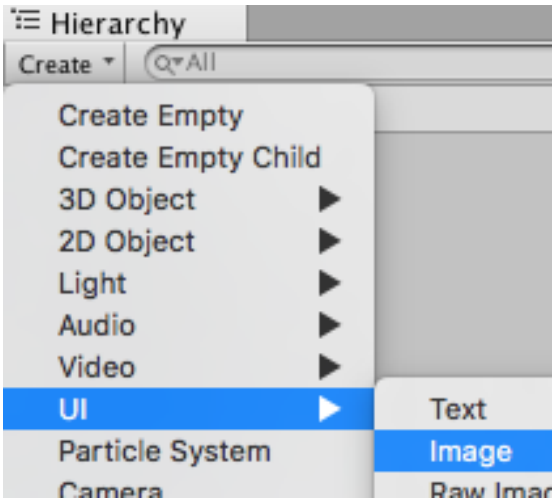
```
68      void FixedUpdate () {
69          #if UNITY_IOS
70          Vector3 acceleration = Input.acceleration;
71          float horizontal = acceleration.x;
72          float vertical = acceleration.y;
73          #else
74          float horizontal = Input.GetAxis ("Horizontal");
75          float vertical = Input.GetAxis ("Vertical");
76          #endif
```

- Taking into account non horizontal device orientations require to call a calibration routine in the Start function

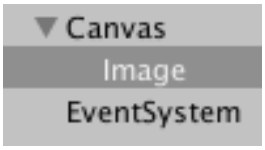
```
53      #if UNITY_IOS
54      //Used to calibrate the Iput.acceleration input
55      void CalibrateAccelerometer () {
56          Vector3 accelerationSnapshot = Input.acceleration;
57          Quaternion rotateQuaternion = Quaternion.FromToRotation (new Vector3 (0.0f, 0.0f, -1.0f), accelerationSnapshot);
58          calibrationQuaternion = Quaternion.Inverse (rotateQuaternion);
59      }
60
61      //Get the 'calibrated' value from the Input
62      Vector3 FixAcceleration (Vector3 acceleration) {
63          Vector3 fixedAcceleration = calibrationQuaternion * acceleration;
64          return fixedAcceleration;
65      }
66      #endif
67
68      void FixedUpdate () {
69          #if UNITY_IOS
70          Vector3 acceleration = FixAcceleration(Input.acceleration);
71          float horizontal = acceleration.x;
72          float vertical = acceleration.y;
73          #else
74          float horizontal = Input.GetAxis ("Horizontal");
75          float vertical = Input.GetAxis ("Vertical");
76          #endif
```

Touch areas

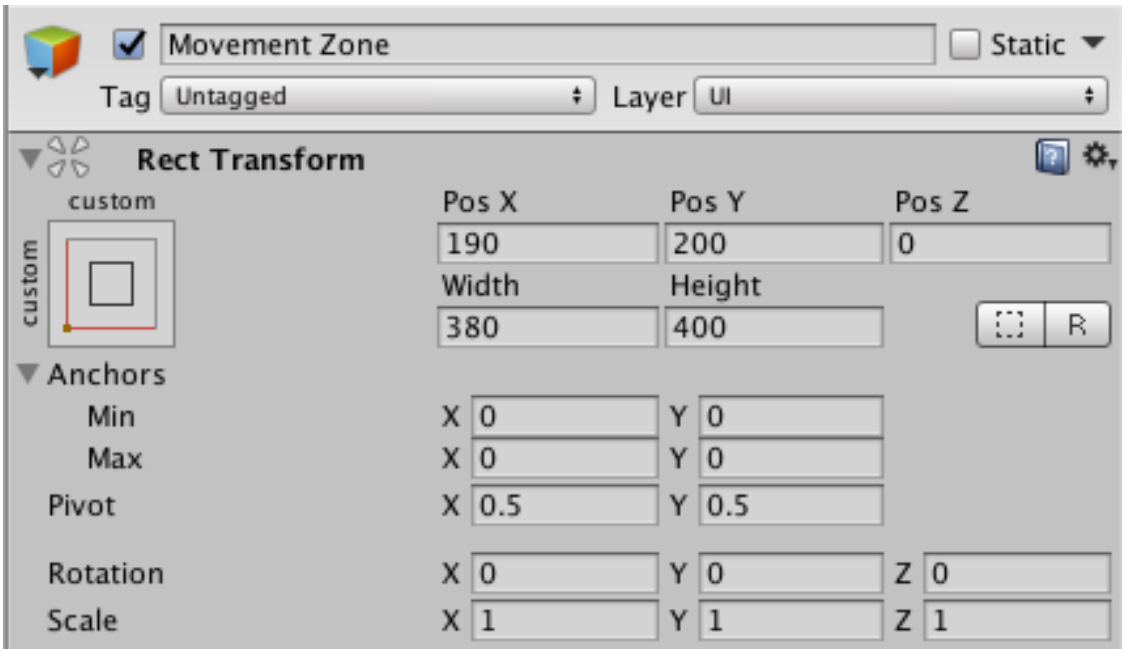
- We want to add touch areas.
- switch to the Scene view and click the 2D button
- In the Hierarchy window create a UI Image

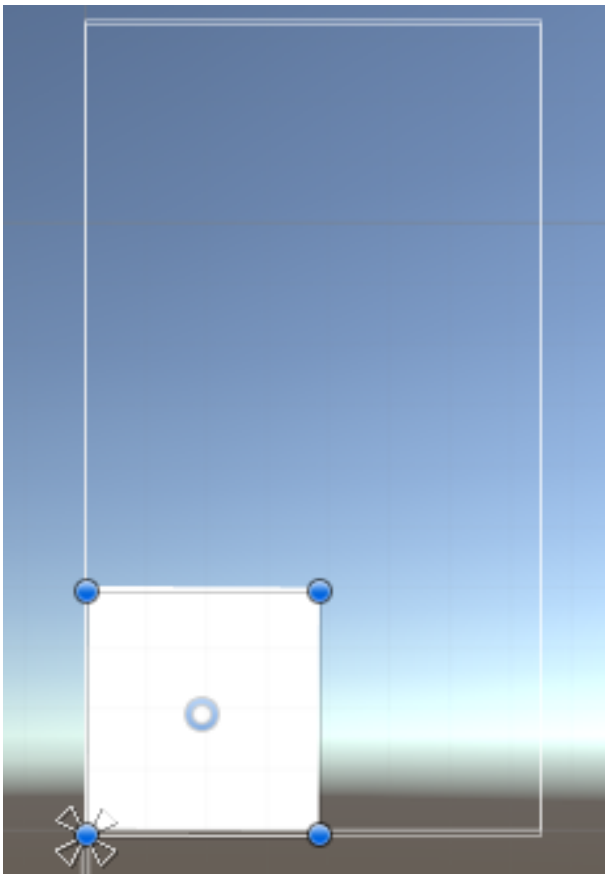


- Select the image and double click on it to zoom in



- Anchor the image area to the left bottom corner of the canvas. Eventually the touch area Rect Transform now look like that





- From the Inspector window, set to zero the alpha value of the image. Now it is invisible, but it is still here.

- Rename the image as “Movement Zone”
- In the *Assets/_Scripts* folder create a new C# script “SimpleTouchPad”
- Edit the file and include support for *UI* and *EventSystem* namespace

- Add to the script class the *IPointerDownHandler*, the *IDragHandler* and the *IPointerUPHandler* interfaces

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;

public class SimpleTouchPad : MonoBehaviour, IPointerUpHandler, IDragHandler, IPointerUpHandler
{

}
```

- Now let's write three delegate methods which allow us to do something when the thumb touches, drags and leave the screen.

```
public class SimpleTouchPad : MonoBehaviour, IPointerDownHandler, IDragHandler, IPointerUpHandler
{
    private Vector2 origin;
    private Vector2 direction;

    public void OnPointerDown (PointerEventData data)
    {
        origin = data.position;
    }

    public void OnDrag (PointerEventData data)
    {
        direction = data.position - origin;
        direction = direction.normalized;
        Debug.Log (direction);
    }

    public void OnPointerUp (PointerEventData data)
    {
        direction = Vector2.zero;
    }
}
```

- Drag the *SimpleTouchPad* script onto the *Movement Zone* Image
- Save and see the direction vector components printed out in the Console window.
- Now we want the direction value get sampled by the PlayerController script with the purpose of moving the Player ship. So, let's add a public method GetDirection in the the SimpleTouchPad script

```
    public Vector2 GetDirection ()
    {
        return direction;
    }
```

- Edit the PlayerController and declare a public reference to the simpleTouchPad

```
    public SimpleTouchPad simpleTouchPad;
```

then use this reference to retrieve the touchpad direction and use it to move the ship

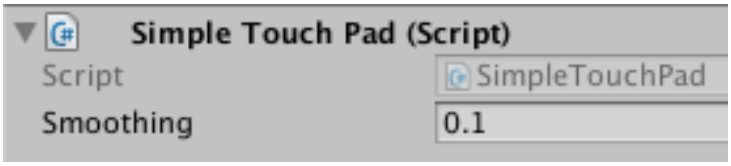
```
        Vector2 direction = simpleTouchPad.GetDirection();
        Vector3 movement = new Vector3 (direction.x, 0.0f, direction.y);
```

- Save and test. The touchpad works, however the ship changes direction too abruptly. Let's add some smoothing. In the SimpleTouchPad script add the following code

```
11      public float smoothing;
12      private Vector2 smoothedDirection;
```

```
30      public Vector2 GetDirection ()
31      {
32          smoothedDirection = Vector2.MoveTowards (smoothedDirection, direction, smoothing);
33          return smoothedDirection;
34      }
```

and from the Inspector Window, set the smoothing coefficient.



- Another improvement will prevent a touch manoeuvre initiated with a finger could be completed by another finger by accident. To filter out this situation we exploit the touchID information obtained by the touch data.

```
14      private bool touched;
15      private int pointerID;
16
17      public void OnPointerDown (PointerEventData data)
18      {
19          if (touched) {
20              pointerID = data.pointerId;
21              origin = data.position;
22              touched = true;
23          }
24      }
25
26      public void OnDrag (PointerEventData data)
27      {
28          if (data.pointerId == pointerID) {
29              direction = data.position - origin;
30              direction = direction.normalized;
31          }
32      }
33
34      public void OnPointerUp (PointerEventData data)
35      {
36          if (data.pointerId == pointerID) {
37              direction = Vector2.zero;
38              touched = false;
39          }
40      }
```

You may notice that any touch causes the gun to fire. This is not acceptable Create the *Fire Zone* object by duplicating and renaming the *Movement Zone*.

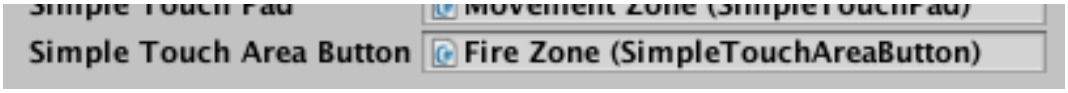
- In the *Assets/_Scripts* folder create a new C# script "SimpleTouchAreaButton". The new script will look like this

```
 7 public class SimpleTouchAreaButton : MonoBehaviour, IPointerDownHandler, IPointerUpHandler {
 8
 9     private bool touched;
10     private int pointerID;
11
12     void Awake ()
13     {
14         touched = false;
15     }
16
17     public void OnPointerDown (PointerEventData data)
18     {
19         if (!touched) {
20             pointerID = data.pointerId;
21             touched = true;
22         }
23     }
24
25     public void OnPointerUp (PointerEventData data)
26     {
27         if (data.pointerId == pointerID) {
28             touched = false;
29         }
30     }
31
32     public bool GetStatus ()
33     {
34         return touched;
35     }
36 }
```

In the *PlayerController* you will get the reference and make use the GetStatus method

```
45     void Update () {
46         if (simpleTouchAreaButton.GetStatus()) {
47             if (Time.time > nextFire) {
48                 Instantiate (shot, shotSpawn.position, shotSpawn.rotation);
49                 audio.Play ();
50                 nextFire = Time.time + fireRate;
51             }
52         }
53     }
```

- From the Inspector window drag *SimpleTouchAreaButton* into the *Fire Zone* object, and drag the Fire Zone into the empty slot reference of the *PlayerController*



Text UIs

sd