

Unity 2D Platform Game

By: Michael Hansen, Coding Pirates Furesø, 2022-23, version 1.52

Document and code available here: <https://github.com/mhfalken/unity/>



This is a guide on how to create a small 2D platform game in Unity, as shown in the image. It includes everything from a starter package to a complete walkthrough of how to do it.

There is no specific C# teaching, but most of the code you need to use is shown with examples. On my github lies the complete “solution” as a packaged file

https://github.com/mhfalken/unity/blob/main/2dplatform_final.zip

There is also a list of tips, if you already know a bit about Unity.

Documents also contain some links to good videos at the bottom.

The document and code were made in Unity version 2020.3. It should also work in other versions, but there may be small differences. (The document has been updated to also fit 2021.3).

Contents

Contents

Unity 2D Platform Game	1
Contents	2
C# editor mini guide (Visual Studio)	3
2D Platform Game	4
Draw Level	5
Add Player	6
Move Player	7
Flip Player	8
Jump Player	9
Ground Detect (advanced – can be done later)	10
Fruits	11
PreFabs	12
Game text and points	13
Points	15
Simple obstacles	15
Restart game	16
Moving objects	16
Platform	18
Sound	18
Expand level to a wider screen	19
Background image	20
2D Platform Game - Advanced Features	20
Turn on/off obstacles	20
Shooting	22
Fan booster	23
Player animations	25
Run animation (advanced)	26
Collect fruit animation	29
Trampoline	30
Multiple levels	32
Transfer points	33
Inspiration for further development	34
Unity Reference	34
Assets Folders	34
External Fields	34
Components	35
Keys	36

Movement	36
Jump	37
Collision	38
Animation	39
Multiple Animations (Advanced)	40
Sound	40
Music	40
Game Text	41
Fonts	41
Static Text	42
Prefabs	43
Particles	43
Dynamically Create and Delete an Object	43
Script Communication	44
Utility	44
Time Handling	44
Debugging	45
Interval check	45
Delay	45
Invoke	45
Coroutine	46
Import Unity Package	46
Special Scripts	46
Moving Objects	46
Stick to Moving Platforms	48
Special setup	49
Move camera to follow player	49
Smooth surface	50
WEB release	50
Android support (advanced)	52
Game over to Android device	54
Links	55

C# editor mini guide (Visual Studio)

The C# editor (the program you write your C# code in) is called MVS in the following. This also applies if you use Visual Studio Code, as they look the same.

It can do many different things, where a few of them are described here.

MVS “communicates” with Unity and knows its syntax. It can therefore help you when writing your code.

- If you start by writing a name of a function it knows, it shows ongoing suggestions. Here you can choose from a list or just press *Enter* if the ‘right one’ is already selected.
- If you need to make e.g. a for loop, you can write for and press the *TAB* key, then MVS automatically creates a for loop template. This works for: for/while/if/foreach/switch. This makes it easier, if you are not completely sure about the syntax.
- You can get help with a Unity function by holding the mouse pointer over the name.
- MVS colors everything depending on what it is, so look at the colors to see if it looks right.
- If something is underlined in red, it’s because MVS thinks there is something wrong.
- Unity GUI doesn’t ‘see’ the correction until the file is saved.

Good keyboard shortcuts: (CTRL = CMD on MAC)

Key	Function
CTRL-S	Save file
CTRL-Z	Undo
CTRL-C	Copy
CTRL-V	Paste
CTRL-H	Search/replace

Special MAC keys:

Key	Function
OPTION-8	[
OPTION-9]
SHIFT-OPTION-8	{
SHIFT-OPTION-9	}

2D Platform Game

This tutorial will walk through how to create a 2D Platform game – see the front page of this document.

Generally, all *C#* code is marked with this font, and all references to **Unity GUI are marked with this font**. Let’s get started.

1. Open Unity HUB and create a new project by clicking **New Project** in the upper right corner. Choose **2D** and give it a name, e.g. **2dplatform_game** and click **Create project**. It takes some time... The program that appears will be called **the GUI** from now on.
2. The first thing to do is import the starter package we need to use. It's located here: <https://github.com/mhfalken/unity/blob/main/2dplatform.unitypackage> Choose download and save the file somewhere you can find it again.
3. In the GUI: **Assets->Import Package->Custom package** and select the above file. In the window that appears, click **Import** in the bottom right corner. The package is now imported and is located under **Assets**.
4. Select **Assets/Levels** and double click on **Level1**. Delete the **Assets/Scenes** folder.

Right now there should be 6 green squares in the **Scene** window.

Draw Level

Draw a small simple level with at least one platform you can jump up on (it doesn't have to look like mine!). You can always draw more later...

1. Click on *Terrain* in the **Hierarchy** (under *Grid*) and then on **Open Tile Palette** in the **Scene** window.

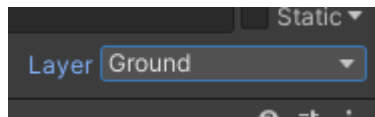
By selecting the images in the **Tile Palette** window you can draw your level in the **Scene** window. You can also delete, copy etc. by selecting in this menu (at the top of the **Tile Palette** window).



2. Draw a bit on the level. Not too much at this point, as you need a Player to find the right dimensions for jumping etc.

When you're done drawing, you need to set the **Layer** to *Ground* in the **Terrain Inspector**.

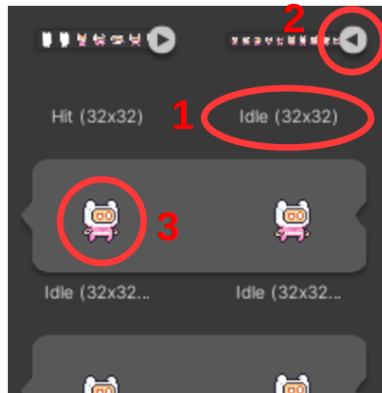
3. First *Ground* must be created, this is done by clicking on the small down arrow in the **Inspector** next to **Layer** and selecting **Add Layer** at the bottom of the list. Write *Ground* next to **User Layer 3**.
4. Click on *Terrain* in the **Hierarchy** again and now select *Ground* in the **Inspector**.



Add Player

We need to select the character we want to use as our Player in the game. There are 4 different ones to choose from and they are all in the folder **Pixel Adventures 1/Assets/Main Characters/**. Look through them and find the one you like best.

1. Select a character type under **Main Characters**. Click on the image of **idle** (1) version's arrow to the right (2).
2. Click on the first image (3) that appears and drag it into the **Scene**.



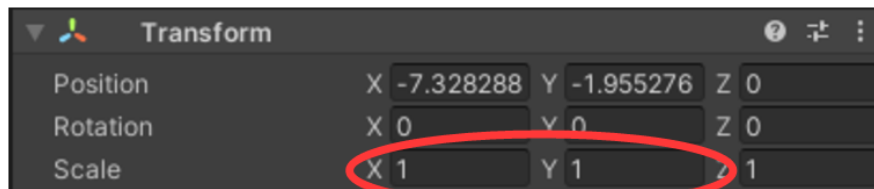
It should now look like this.



A new object also appears in the **Hierarchy** called something like “Idle ...”.

3. Rename ‘Idle ...’ to *Player*. Remember capital P!

You can change the size of the character in the **Inspector** by changing the **Scale** values.





Click **Play** at the top of the screen and see how it looks. Nothing happens, *Player* becomes ‘floating’ in the air.


4. Select *Player* in the **Hierarchy** and in the **Inspector** (out to the right) click **Add Component** and select **Rigidbody 2D**.

The Rigidbody component makes *Player* handled by the physics engine, which among other things means it will be affected by gravity.

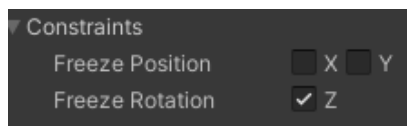
Click **Play** and see what happens! Now *Player* falls through the platform. This is because it doesn’t have a Collider.

5. In the **Inspector** now add a **Box Collider 2D**

Click **Play** again. Now the character should fall down and stand on the platform.

6. Adjust the **Collider** so it better fits the size of the character by clicking **Edit Collider**  and then adjusting the ‘rectangle’ in the **Scene**.

7. Under **Rigidbody 2D** you must set the following (otherwise the character can tip over):



Move Player

To get *Player* to move we need to create our first script.

1. In the folder **Assets/Scripts/** right-click and create a **Create->C# Script** and call it *PlayerController*. (It’s important to do this in one flow – you must not rename the file...!)
2. Now drag this script up over *Player* in the **Hierarchy**.
3. Double-click on the script to open it in the MVS editor.

The code here is written in C# and it’s intended that you should add the code that’s missing depending on what you want to achieve.

If you want a walkthrough of C# here’s a link to a good guide in Danish, made specifically for Unity: <https://github.com/Grailas/CodingPiratesAalborg/blob/master/Guides/HjÅlpeguide.pdf>

4. Write the following lines in the Update() function (the new lines are shown in **bold**):

```

void Update()
{
    float dir = Input.GetAxisRaw("Horizontal");
    Debug.Log(dir);
}

```

5. Look at the **Console** window and click **Play**.

Now press the *arrow keys* and see what happens. The Debug line writes out every time you press an arrow key. (Also works with A and D). It's the Debug.Log() line that prints to the **Console** and is an important tool when you need to debug your code.

6. Now change the file to the following (you're supposed to change the file yourself so it matches the below):

```

public class PlayerController : MonoBehaviour
{
    [SerializeField] float speed = 10;

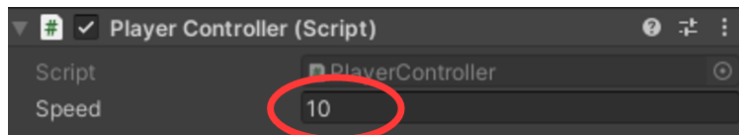
    // Start is called before the first frame update
    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {
        float dir = Input.GetAxisRaw("Horizontal");
        transform.Translate(dir * speed * Time.deltaTime, 0, 0);
    }
}

```

Click **Play** and see if it works.

7. You can change the speed in the **Inspector** by changing the **Speed** value.



Flip Player

The character should now look in the direction it's moving. This is done by using the **Flip** field in the **Sprite Renderer**. This should of course be done automatically from the *PlayerController* script.

8. Add the missing code:


```

public class PlayerController : MonoBehaviour
{
    [SerializeField] float speed = 10;
    SpriteRenderer sr;

    // Start is called before the first frame update
    void Start()
    {
        sr = GetComponent<SpriteRenderer>();
    }

    // Update is called once per frame
    void Update()
    {
        float dir = Input.GetAxisRaw("Horizontal");
        transform.Translate(dir * speed * Time.deltaTime, 0, 0);
        if (dir < 0)
            sr.flipX = true;
    }
}

```

Now the character can turn left but not back to the right again.

9. Add the two lines of code yourself that are missing so that *Player* can also turn right again.

Jump Player

We need to get *Player* to be able to jump and this is done by adding some more code to the *PlayerController* script. The jump itself works by adding a force to the **Rigidbody** component:

```
rb.AddForce(Vector3.up * jumpPower);
```

The Rigidbody component must first be ‘retrieved’ (just like *SpriteRenderer* was done) and this is done in the following way (they should be placed in the same place in the code as for *SpriteRenderer*):

```

Rigidbody2D rb;

...

rb = GetComponent<Rigidbody2D>();

```

jumpPower should be a visible variable in the **Inspector**, so you can later adjust it:

```
[SerializeField] float jumpPower = 200;
```

The Jump key is read in the following way:

```
bool jump = Input.GetButtonDown("Jump");
```

1. Now put all the above lines in the “right” places in the script. Look at the script to see where the corresponding lines are.
2. Add an if statement to perform the jump when you press the *Space* key.
3. Adjust the jump height in the **Inspector** with the **Jump Power** value.

If you think *Player* can jump a bit too far, you can increase the **Rigidbody 2D->Gravity Scale** value to e.g. *1.5* in the **Inspector**, which will then require you to increase the **Jump Power** field to achieve the same jump height. This way *Player* can jump just as high, but shorter.

Ground Detect (advanced – can be done later)

You can jump while you’re in the air, which is not the intention. You can avoid this by checking if you’re standing on the ground before jumping. The following function checks if you’re standing on the ground.

```
[SerializeField] LayerMask groundLayer;
BoxCollider2D bc;

void Start()
{
    bc = GetComponent<BoxCollider2D>();
}

bool GroundCheck()
{
    return Physics2D.CapsuleCast(
        bc.bounds.center,
        bc.bounds.size,
        0f, 0f,
        Vector2.down,
        0.5f,
        groundLayer
    );
}
```

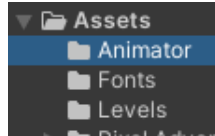
1. Now put all the above lines in the “right” places in the script.
2. The **Ground Layer** field in the **Inspector** should be set to *Ground*.

Use the function (GroundCheck()) to make an extra check before jumping.

3. You need to ‘extend’ if (jump) with an extra condition, and this can be done by writing if (jump && GroundCheck()). This means that both conditions must be met before you can jump.

Click **Play** and see that everything works.

Fruits




Now we need some items that we can ‘collect’. There are a series of fruits that can be used for this. The fruits should also be animated.

1. Start by creating a folder under **Assets** called **Animator**.
2. In the folder **Pixel Adventures 1/Assets/Items/Fruits** select *Cherries* and set **Pixels Per Unit** to 16 in the **Inspector**. Then press **Apply** in the bottom right corner. This should generally be done for all the graphic images you want to use. I have done it for a few of them, the rest you must handle yourself. If you ‘forget’ this, the images will just become very small!



3. Click on *Cherries* and drag it into the **Scene**. In the dialog box that appears, save the file in the *Animator* folder and call it *Cherries*.
4. Change the name in the **Hierarchy** to *Cherries* (remove _0).

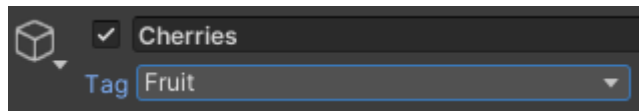
Try running it and see how the berry moves.

5. Add a **Box Collider 2D** to *Cherries* and adjust the size so it fits .
6. Then set **Is Trigger**.



A *trigger* collision means you don’t get a real collision, but only a trigger that the two figures overlap.

7. In the **Inspector** add a **Tag** called *Fruit* and select it for *Cherries* (**Add Tag**, click on ‘+’ and write *Fruit*. Then select *Fruit* for *Cherries*).

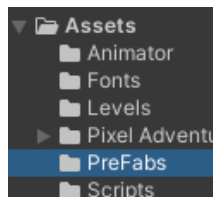


8. Add the following lines at the bottom of the *PlayerController* script (before the last '}').

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.CompareTag("Fruit"))
    {
        Destroy(collision.gameObject);
    }
}
```

Test and see what happens when the *Player* hits the fruit.

PreFabs



When you have finished making your fruit, you should create a *PreFab*. A *PreFab* is a template that you can use again and again.

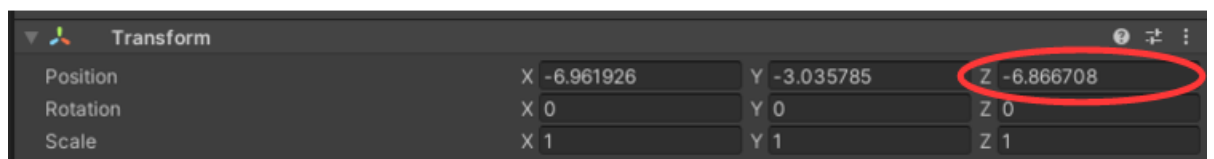
1. First create a folder under **Assets** called **PreFabs**.
2. Now drag *Cherries* from the **Hierarchy** down into the **PreFabs** folder.

Now make some more different fruits and create **PreFabs** for each of them, so you get a small collection in **PreFabs**.

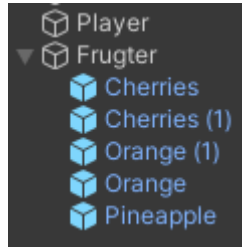
When you later want to use a fruit in your game, you just drag one from the **PreFabs** folder directly into the **Scene** where you want it and then they work. You just need to move them to the right place in the game.

When you get many fruits, your **Hierarchy** becomes a bit messy.

3. Create a **Create Empty** object in the **Hierarchy** and call it *Fruits*. Here Unity does something a bit annoying – it sets the Z position to something other than 0! Fix this in the **Inspector** by setting it to 0 – see image.




4. Drag all the fruits under this new object to get a bit more order.

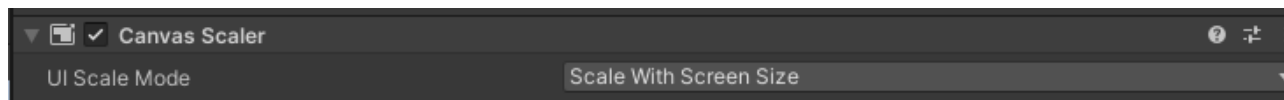


Game text and points


When you collect fruits and other items in the game, you should get points for it. These points should of course be displayed on the screen.

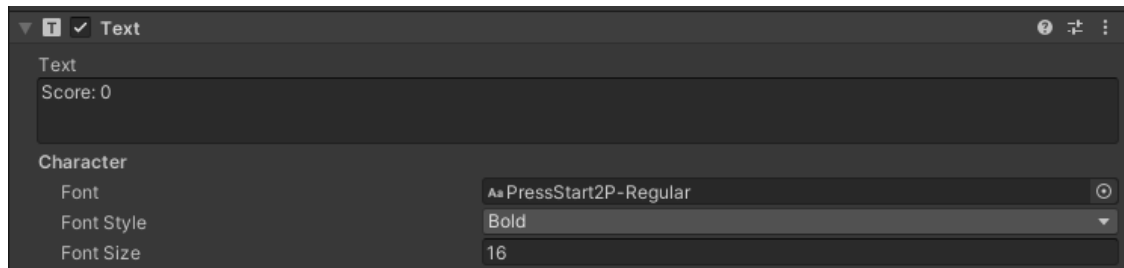
You start by inserting text into the **Scene** in the following way:

1.  Create an object in the **Hierarchy** and select **UI->Text** and call it *Score*. (If you are running Unity version 2021.3 it is located in **UI->Legacy->Text**.) This will automatically create two objects: **Canvas** and **Score**.
2. Start by selecting *Canvas* and set the following in the **Inspector**:



After you have created *Score*, it should be configured in the **Inspector** (see the image below).

3. Change **Text** to *Score: 0*.
4. Next to **Font**, click on  and select *PressStart2P-Regular*.
5. Set **Font Style** to *Bold*
6. Set **Font Size** to 16.
7. Choose a good color in **Color**.



Look now in the **Game** window and see that the text is located all the way down in the left corner. This is a bit strange (but that's just how it is).

8. Now move the text up to the top of the screen (**Scene** window), by dragging it up (here you should look at the **Game** window, but move the text in the **Scene** window!).

When you zoom out in the **Scene** window you can see that a white rectangle appears – it should be at the top of this rectangle! (Your entire game is in the red circle down in the left corner – you have to zoom out that far!)



9. Now place the text in a good location and increase the size so it is easier to see.

When you make the size larger, the text can suddenly “disappear”. This is because the box that the text is in is too small. This is fixed by selecting the text and then adjusting the box size in the **Scene**. Remember to also make the box long enough for the text we will write later. (Drag the blue dots to change the box size).



10. In the *PlayerController* script now add the following lines ‘in the right places in the code’.

```
using UnityEngine.UI;

[SerializeField] Text scoreText;

scoreText.text = "Score: ";
```

The last line should be at the bottom of the `Update()` function.

11. Drag the *Score* object in the **Hierarchy** over to the **Inspector: Player Controller->Score Text** field.

Run the code and see that it works.

Points

Right now no points are displayed, as we are not counting any points!

12. Now create a variable called *points* of type *int*. Count it up each time you take a fruit. (Use the following lines)

```
int points;
```

13. You display the points by updating the line with the Score text to the following:

```
scoreText.text = "Score: " + points;
```

Try and see how it works.

14. If you want the different fruits to give different points, it can be done in the following way:

```
if (collision.gameObject.name.Contains("Cherries"))
    points += 20;
```

Simple obstacles

Until now the game has been quite peaceful, so it's time to add some obstacles. There is a list of obstacles and traps under the **Pixel Adventures 1/Assets/Traps** folder.

1. Under *Spikes* select *Idle* and drag it into the **Scene** at a good location.
2. In the **Hierarchy** rename it to *Spikes*.
3. Select *Spikes* and add a **Collider** and adjust the size

4. Set **Is Trigger**.
5. Also add a **Tag** called *Trap*.
6. Now add the following lines in the *PlayerController* script, in the function `private void OnTriggerEnter2D(Collider2D collision)`

```
if (collision.gameObject.CompareTag("Trap"))
{
    Destroy(gameObject);
}
```

7. Put several different ‘obstacles’ in, like the saw (**Saw**), and make sure it is animated (so it looks like it’s running).
- (Pro tip: If you want only part of the saw to be visible, like having it stick up from the ground, you should set **Position Z** to 1.)
8. Remember to create **PreFabs** of the different obstacles, so they are easier to reuse.

Restart game

When we die, we need a way to restart the game.

1. Add the following lines to the code:

using UnityEngine.SceneManagement;

```
Invoke("RestartLevel", 1);

private void RestartLevel()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}
```

The second line ‘Invoke ...’ should be where we die. It is important to remove `Destroy(gameObject)`; as all the code in the script will otherwise stop!

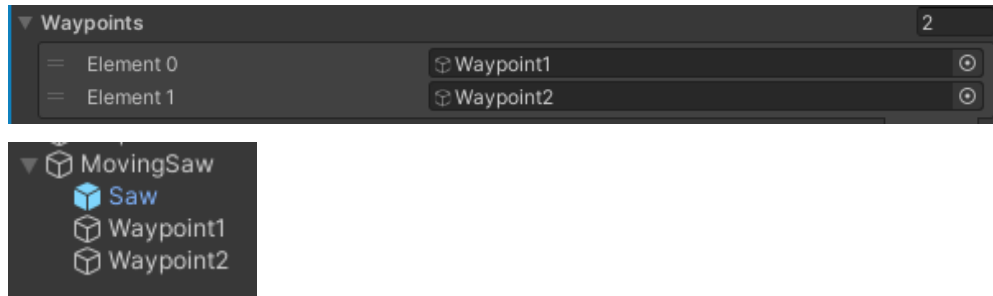
When you die, the game should restart after one second.

Moving objects

We now want some of our obstacles to move. The script we need for this is located in **Assets/Scripts** and is called *WaypointFollower*. The script can be used both to get platforms to move, but also enemies to walk in a pattern, i.e. between a list of points.

In the following we will get the circular saw to move to the sides, i.e. between two points.

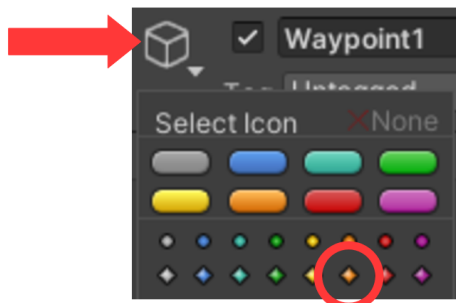
1. Create an empty object and call it *MovingSaw*. (Remember to set **Transform->Position->Z** = 0.)
2. Add *Saw* (use Prefab) under *MovingSaw*
3. Add the script *WaypointFollower* to *Saw*.
4. In the **Inspector** for *Saw* create the number of Waypoints you need – in this case 2. Open **Waypoints** and click on ‘+’ to add Waypoints.
5. Create a corresponding number of *Waypoints* (**Create Empty**) in the **Hierarchy** under *MovingSaw* and call them *Waypoint1*, *Waypoint2*, ...
6. The *Waypoints* are dragged over to the **Inspector** for *Saw* one by one. It should now look like this:



7. Set **Speed** to **5** in the **Inspector** for *Saw*. This is the speed at which it should move and should be trimmed later when everything is running.

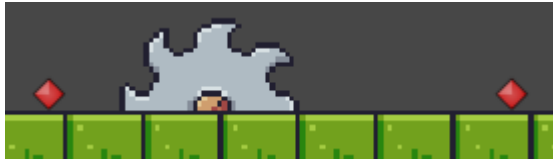
Now everything is prepared, but we still need to place the two *Waypoints* in the right places.

8. Select *Waypoint1* and in the **Inspector** click on the small cube in the top left corner and select one of the colors in the bottom row.



The waypoint now becomes visible in the **Scene** window.

9. Move it to one endpoint.
10. Do the same for *Waypoint2*, but place it as the other endpoint.



Now everything should work. Remember that you can change the speed with **Speed**. Note that the *Waypoints* are not visible in the **Game** window.

11. Remember to create a **PreFab**, so it's easy to reuse it.
12. Select another object and get it to move, e.g. *Spike Head*.

It can also be an advantage to have a few different Prefabs of e.g. the saw, one for left/right and one for up/down. This makes it easier to use later when you want to create more levels.

Platform

If it is a platform that should move, it is important that you have set **Layer** to *Ground* for the platform and added a **Collider**. If the platform runs to the sides, the *Player* falls off, as the *Player* does not automatically follow the platform. This can be solved by adding the script *StickyPlatform* to the *Platform*.

Sound

The game is a bit quiet, so it's time to add some sound. There are already some sound files in the **Assets/Sounds/** folder. By double-clicking on them, you can hear how they sound.

For the *object* that you want sound support for, you must first add an **AudioSource** component.

1. Select *Player* and add **AudioSource**.
2. In the *PlayerController* script, insert these lines in the 'right' places in the code (hint: the lines are in the correct order).

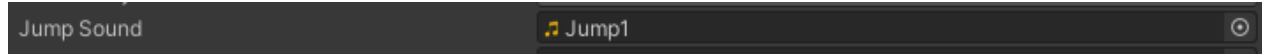
```
[SerializeField] AudioClip jumpSound;

AudioSource audioPlayer;

void Start()
{
    audioPlayer = GetComponent<AudioSource>();
}

// When jumping:
audioPlayer.PlayOneShot(jumpSound, 1);
```

3. The first line creates a field in the **Inspector** that you need to drag the sound you want to use for jumping into.



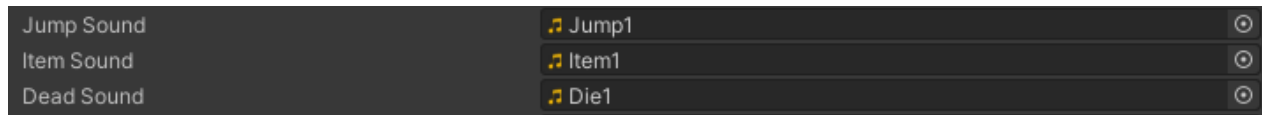
The last line plays the actual sound file, and should be placed where you jump.

Try if the sound works when you jump.

You can add more sounds by creating these two lines for each sound.

```
[SerializeField] AudioClip itemSound;  
...  
audioPlayer.PlayOneShot(itemSound, 1);
```

The first line is used to select which sound file you want and the second line plays the sound file. It's important to remember to drag a sound file into the **Inspector** for each sound.



In the case where we die, we need to wait a bit before destroying our *Player*, as the sound would otherwise disappear too.

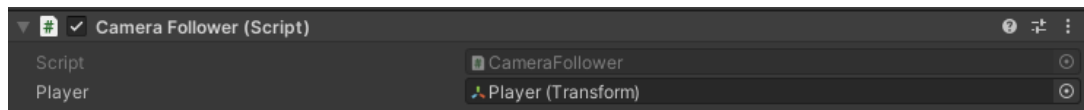
4. This is done by adding an extra parameter to `Destroy(...)`, which tells how long to wait. Below we wait 1 second, which should be enough time to play a small sound.

```
Destroy(gameObject, 1);
```

Expand level to a wider screen

We're starting to run out of space in our game, so it's time to make the level wider. This requires the camera to follow the *Player*, so we can still see what's happening.

1. Find the *CameraFollower* script and drag it over the *Main Camera* in the **Hierarchy**.
2. Select *Main Camera* and drag the *Player* object into **Player** in the **Inspector**.

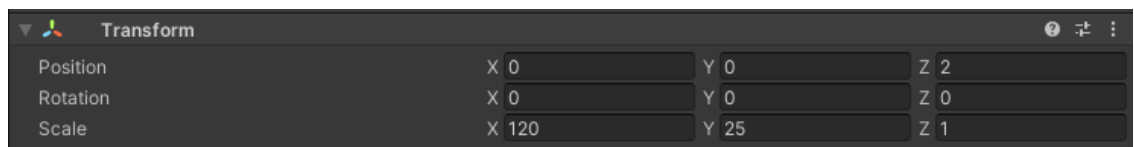


Now the camera should follow the *Player* and you can therefore expand your level in width and height.

Background image

There are some background images in the **Background** folder.

1. Select one of them and drag it into the **Scene** and call it *Background*.
2. Set **Position Z** to 2 (in the **Inspector**).
3. Adjust **Scale** so it fills everything.



2D Platform Game - Advanced Features

Here comes a list of more advanced features that can mostly be made in any order. They require that you have completed all the previous steps and therefore have some routine in using Unity.

Turn on/off obstacles

This describes how to create a ‘switch’ that can turn an obstacle on or off. When the obstacle is turned off, it is completely gone (disabled).

1. Find a good animation or a couple of still images and drag it into the **Scene** as you did when creating a fruit.
2. Call it *Switch* and save it in the *Animator* folder. (If using still images, there won’t be an animation – you’ll need to add one yourself – see later section. If you choose not to have an animation, you must delete the code lines that control the animation.).
3. Add **AudioSource**
4. Create a C# script and call it *Switch*.
5. Drag it over the *Switch* object in the **Hierarchy**.
6. Open the script and insert the following code:

```
[SerializeField] GameObject gameobj;  
[SerializeField] bool ModeSet;  
[SerializeField] AudioClip clickSound;  
  
private void OnTriggerEnter2D(Collider2D collision)
```

```

{
    if (collision.gameObject.CompareTag("Player"))
    {
        AudioSource audioPlayer = GetComponent<AudioSource>();
        audioPlayer.PlayOneShot(clickSound, 1);

        BoxCollider2D coll = GetComponent<BoxCollider2D>();
        coll.enabled = false;

        Animator animC = gameObject.GetComponent<Animator>();
        animC.enabled = false;

        gameobj.SetActive(ModeSet);
    }
}

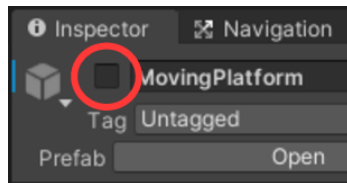
```

Fill in:

Gameobj	The object to be controlled (on/off). Drag the object into this field.
Mode Set	If set, the object is turned on, otherwise it's turned off.
Click Sound	The sound when touching the switch



This will turn on *MovingPlatform* when touching the Switch. This requires that *MovingPlatform* is turned off before starting the game. This is done by removing the checkmark in this field in the **Inspector** for *MovingPlatform*:



8. Remember to create a Prefab.

You can easily create multiple variants – some that only turn on/off the script part. This way you can, for example, see a platform, but it doesn't move.

Shooting



This describes how to create something that can shoot. We start by creating the fireball itself. Under the **Assets/Graphics** folder, there are some images that can be used for it.

1. Drag the *Fireball1* image into the **Scene**.
2. Add a **Rigidbody 2D**.
3. Add a collider and set **Is Trigger**.
4. Set **Tag** to *Trap*.
5. Create a script and call it *Fireball* and drag it over the *Fireball* object in the **Hierarchy**.
6. Open the script and add the following code:

```
float maxLifeTimeS = 6;

void Update()
{
    maxLifeTimeS -= Time.deltaTime;

    if (maxLifeTimeS <= 0)
        Destroy(gameObject);

    transform.Translate(0, -5 * Time.deltaTime, 0);
}

private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.CompareTag("Player") ||
        (collision.gameObject.layer == LayerMask.NameToLayer("Ground")))
    {
        Destroy(gameObject);
    }
}
```

7. Create a Prefab of it.

We now need to create the part that will shoot the fireball.

8. Under **Traps/Fire** take the image file *Off* and drag it into the **Scene**.
9. Rename the object to *Fire_shoot*.
10. Create a script called *Fire_shoot* and drag it over the *Fire_shoot* object
11. Add the following code to the file:

```
[SerializeField] GameObject fireball;
[SerializeField] float fireRateSec = 2;

Quaternion rotation;
Vector3 pos;
float rateTimeS;

void Start()
{
    rateTimeS = 0;
    rotation = transform.rotation * Quaternion.Euler(0, 0, 180);
    pos = new Vector3(transform.position.x, transform.position.y, transform.position.z + 0.1f);
}

void Update()
{
    rateTimeS += Time.deltaTime;
    if (rateTimeS > fireRateSec)
    {
        // Shoot
        Instantiate(fireball, pos, rotation);
        rateTimeS = 0;
    }
}
```

PreFabs folder into the *Fireball* field.

The shooting speed is controlled with the **Fire Rate Sec** field.

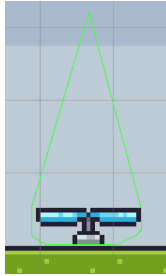
13. When everything works, create a Prefab.

Fan booster

This describes how to create a fan (blower) that can blow you up into the air.

1. Find the image of the *Fan* (On) and drag it into the **Scene** as you did for the fruits and call it *Fan* and save it in the *Animator* folder.
2. In the **Hierarchy** rename the object to *Fan*.
3. Select the *Fan* object and give it a **Tag** called *Fan* (must be created first).

4. Add a **Polygon Collider 2D** and make it look like in the image below (the green line). (The height of the triangle is not that important – just something that resembles the image).
5. Set **Is Trigger**.



6. Add the following code to the *PlayerController* script (at the bottom of the file before the last '}').

```
private void OnTriggerStay2D(Collider2D collision)
{
    if (collision.gameObject.CompareTag("Fan"))
    {
        float distX = collision.gameObject.transform.position.x - gameObject.transform.position.x;
        if (Mathf.Abs(distX) < 0.5f)
            distX = Mathf.Sign(distX) * 0.5f;

        float distY = Mathf.Abs(collision.gameObject.transform.position.y - gameObject.transform.position.y);
        float forceY = jumpPower / (7 * distY * distY);
        float forceX = -jumpPower / (4 * distX);

        if ((distY < 3) && (rb.velocity.y < 0))
            forceY *= 3;

        if (Mathf.Abs(forceY) > 450)
            forceY = Mathf.Sign(forceY) * 450;

        rb.AddForce(new Vector2(forceX, forceY));
    }
}
```

7. You also need to add a line to the *Update()* function (it should be placed right after the *transform.Translate* call - only the line in **bold** needs to be added):

```
transform.Translate(dir * speed * Time.deltaTime, 0, 0);
```

```
rb.velocity = new Vector2(0, rb.velocity.y);
```

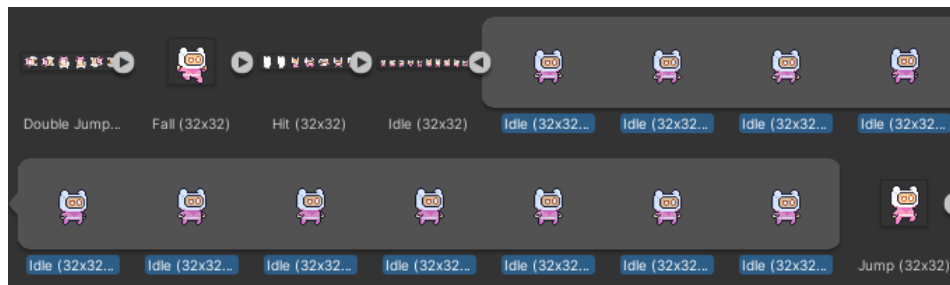

- Remember to create a Prefab when it works.

Player animations




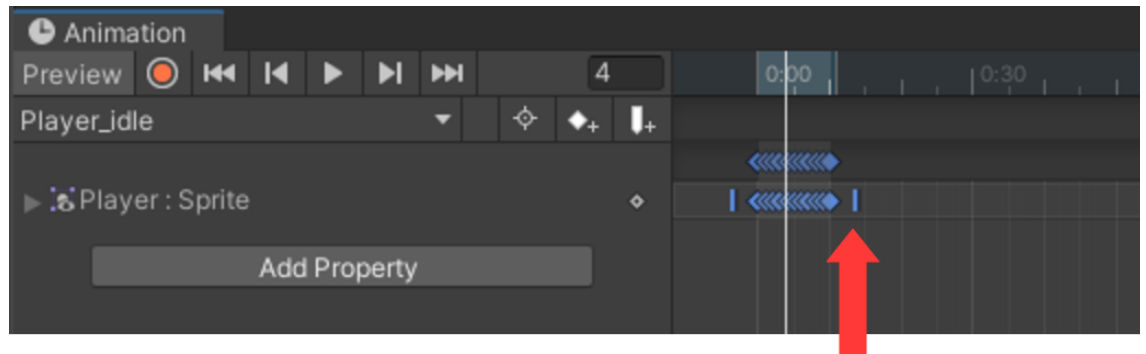
Now we need to animate our *Player*. Under the graphics folder where you found your *Player* (**Main Characters/**), there are some different “animations”, i.e. series of images that, when shown at the right speed, create an animation. We’ll start by creating an animation when the *Player* stands still (idle).

- In the *Animator* folder, **Create->Animation** and call the file *Player_idle*.
- Then drag the *Player_idle* file over the *Player* object in the **Hierarchy**. (This also creates a *Player controller* the first time you do it – see image on the right.)
- Then you need to open the animation window: **Window->Animation->Animation**.
- Then select the *Player* object in the **Hierarchy**.
- Now select all the images that make up the animation and drag them into the **Animator**. (This is done by finding the *Idle* animation file you want to use and clicking on the small arrow so you can see all the images, select them all and drag them into the animator with the mouse).

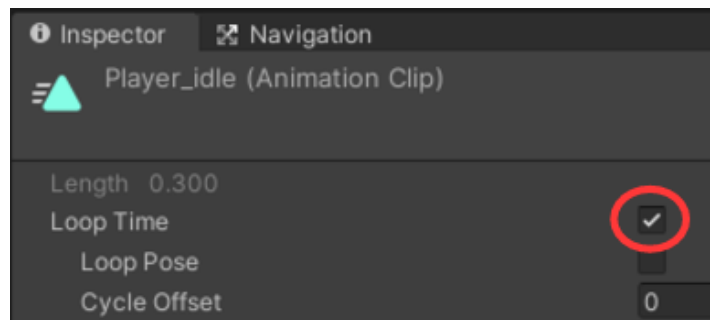


It should look like the image below.

- You can now see how it looks by pressing the play  button, and adjust the speed by moving the blue line to the sides (see red arrow below).



7. Since the animation should continue, remember to set *Loop Time* in the **Inspector** for *Player_idle*.



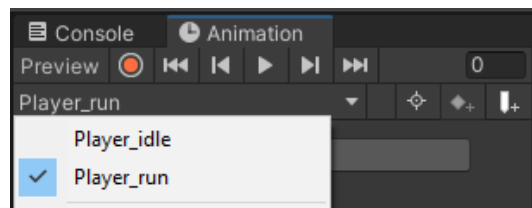
When you're done with the animation window, you can advantageously dock it to one of the other windows you have, e.g. the **Console** window, so it's easier to find next time.

Try the game and see if it works.

Run animation (advanced)

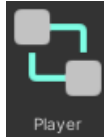
Our *Player* needs more than one animation, as he also needs a run animation.

1. For each additional animation you need, repeat the steps above. Remember to select the correct animation in the **Animation** window before copying the images over.

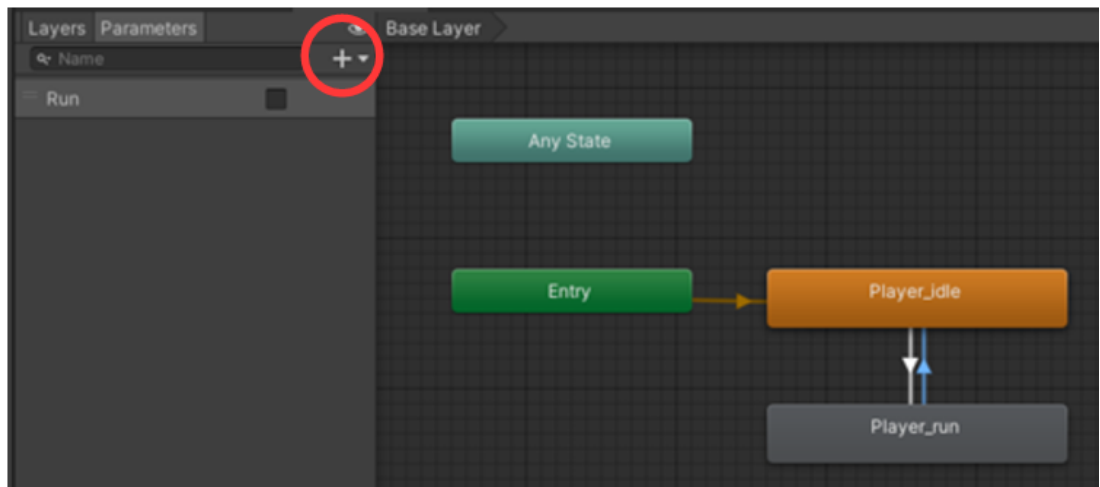


Since you now have multiple animations, you need to tell Unity which animation to use when.

2. Open the Player **Animator** – double click on:

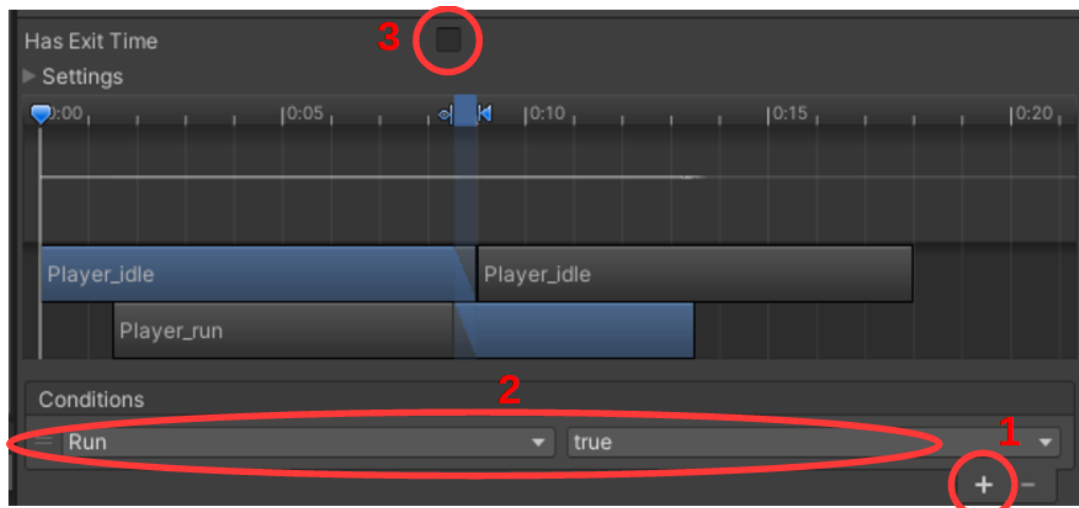


An image appears that looks a bit like this:



This is a diagram showing the different animation states and how you can go from one to another.

3. Create the two vertical arrows by right-clicking on *Player_idle*, select **Make Transition** and drag it to *Player_run*.
4. Do the same in the opposite direction too, so there are two arrows – one in each direction – see image.
5. Add a parameter by clicking on '+' at the top left (red circle), select **Bool** and call it *Run*.
6. Now click on the arrow that goes from *Player_idle*->*Player_run* and look at the **Inspector** on the right.



7. Press '+' [1], which automatically creates [2]. Remove the checkmark in **Has Exit Time** [3].

8. Do the same for the other arrow, but this time choose **false** in [2].

It is now set up so that when *Run* is **true**, it switches to the *Player_run* state and when *Run* is **false**, it switches back to *Player_idle*. This way *Run* determines which animation the player has.

Now we need to get our Player script to control this *Run* parameter.

9. Add the following lines to the *PlayerController* script (in the right places).

```
Animator animator;

...

animator = GetComponent<Animator>();

...

if (dir == 0)
    animator.SetBool("Run", false);
else
    animator.SetBool("Run", true);
```

If *dir* (our movement) is 0 (we're standing still), then we set *Run* to **false** (i.e. don't run), otherwise set *Run* to **true** (run).

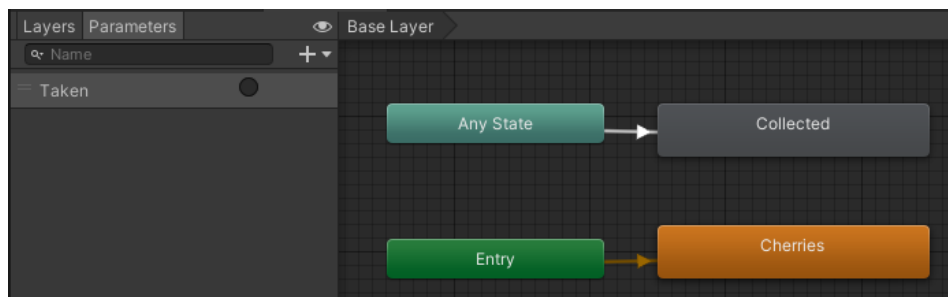
Try to see how it works.

Collect fruit animation

We need to create an animation when collecting a fruit. (It's an advantage if you've created Player animations first).

Down in the **Assets** folder under **Fruits** there's an animation called *Collected* which should be used when collecting a fruit.

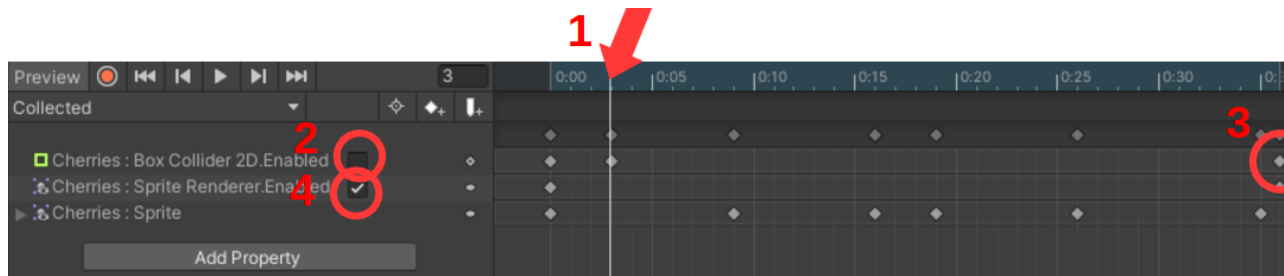
1. Under the **Animator** folder right-click and **Create->Animation** and call it *Collected*.
2. Now drag it over one of the fruits in the **Hierarchy**.
3. Open the **Animator** window and then select this fruit. There's now a new state called *Collected*.
4. Create an arrow between **Any State** and the **Collected** state.
5. Create a **Parameter (Trigger)** and call it *Taken* and select it for the arrow in the **Inspector**.



6. In the **Animation** window select *Collected* and drag the images from the *Collected* file into the **Animation** window and trim the speed so it looks good.

Then we need to add some special states so we can control that you only get points once and that the object becomes disabled after the animation.

7. Press **Add Property** and then press the arrow next to **Box Collider 2D** and then '+' next to *Enable*.
8. Do the same for **Sprite Renderer**. See image below.
9. Then press in the blue area ([1] in the image) and then press [2] so the checkmark disappears (this also creates a small diamond).
10. Then remove the 'diamond' at [3] (right-click **Delete Key**).
11. Now click in the blue area above [3] and remove the checkmark at [4].



12. In the *PlayerController* script now change the `OnTriggerEnter2D` to the following:

```
Animator animC = collision.gameObject.GetComponent<Animator>();
animC.SetTrigger("Taken");
//Destroy(collision.gameObject);
```

Now it should work.

It's important to do this for all fruits, but it's much easier now, as you only need to do a couple of the steps.

1. (2) From the **Assets/Animator** folder drag *Collected* over the fruit in the **Hierarchy**.
2. (4-5) Select the fruit and in the **Animator** window, create the arrow, create the parameter *Taken* and select it for the arrow.
3. This is done for all fruits.

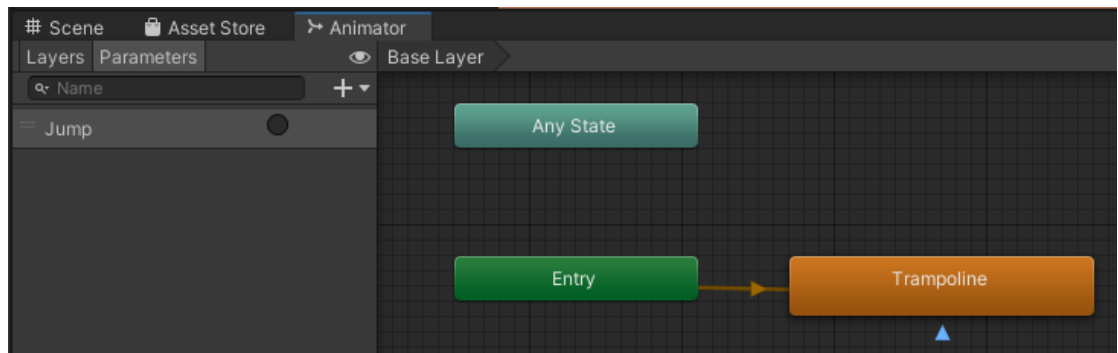
The same method can be used to create an animation when the *Player* dies. There are some special graphics images for this under the *Main Characters* folder.

Trampoline

This describes how to create a trampoline. The code is made so that for the trampoline to 'work' you need to jump on it. (It's an advantage if you've created *Player* animations first).

1. Find the image of *Trampoline* (Jump) and drag it into the **Scene** as you did for the fruits and call it *Trampoline* and save it in the *Animator* folder.
2. In the **Hierarchy** rename the object to *Trampoline*.
3. Select the *Trampoline* object in the **Hierarchy** and add a **Box Collider 2D** and adjust it.
4. Also set **Is Trigger**
5. Give it a **Tag** called *Trampoline* (must be created first).

6. Open the **Animator** window (Trampoline object must be selected) and add a *trigger* **Parameter** called *Jump*.
7. Create a **Make Transition** from the *Trampoline* state to itself – see blue arrow in the image below in the bottom right corner:



8. Select the new *Transition* (the arrow) and add a **Conditions** and set it to *Jump*.
9. Also remove the checkmark next to **Has Exit Time**.



10. Select *Trampoline* in the *Animator* folder and remove the checkmark in **Loop Time** (Inspector):



11. Add the following lines to the *PlayerController* script (at the bottom of `OnTriggerEnter2D(Collider2D collision)`)

```
if (collision.gameObject.CompareTag("Trampoline"))
{
    Animator animC = collision.gameObject.GetComponent<Animator>();
    float forceY = Mathf.Abs(rb.velocity.y * jumpPower / 8);

    if (forceY > 100)
    {
        if (forceY > 800)
            forceY = 800;
    }
}
```

```

        animC.SetTrigger("Jump");
        rb.velocity = new Vector2(rb.velocity.x, 0);
        rb.AddForce(new Vector2(0, forceY));
    }
}

```

12. Remember to create a Prefab when it works.

Multiple levels

We need to create support for multiple levels. For this purpose we'll use a flag as a goal. It's located under **Checkpoints/Checkpoint** and is called *Checkpoint (Flag Idle)*.

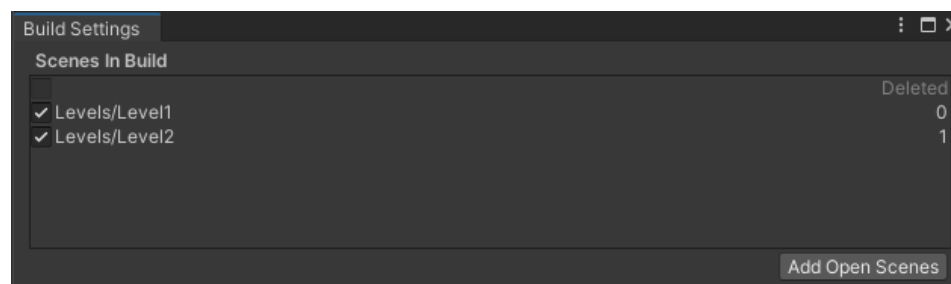
1. Drag the flag into the Scene at the end of Level1 and call it *FlagIdle* and place it in the Animator folder.
2. In the Hierarchy call it *FlagEnd*.
3. Add a **Box Collider 2D**
4. Set **Is Trigger**.
5. Create a new script and call it *LevelEnd* and drag it over *FlagEnd*.

We now need to add all our levels to the build settings.

6. Save (Ctrl-S) and switch to the **Levels** folder.
7. Select *Level1* and press Ctrl-D and call the new one *Level2*.

We now have two identical levels.

8. Double click on *Level1* (to open it)
9. Open **File->Build Settings**.
10. In **Build Settings** press **Add Open Scenes** (bottom right corner).
11. Repeat for *Level2* so you get the following image.



12. Close the window and open *Level1* again (double click).
13. Open the *LevelEnd* script and add the following lines.


```
private void OnTriggerEnter2D(Collider2D collision)
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
}
```

14. The levels are now identical, but this is easily fixed by deleting everything you don't need from *Level2* and then drawing something new (Remember to select *Level2* first...).
15. You can also add a sound in the same way as described earlier. However, there's the problem that you don't get to hear the sound before the new level is loaded. This is solved by changing the code to:

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.name == "Player")
    {
        audioPlayer.PlayOneShot(finishSound, 1);
        Invoke("CompleteLevel", 2f);
    }
}

private void CompleteLevel()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
}
```

Transfer points

When you start the next level, you've 'lost' all your points. This is because everything gets reset. This is solved in the following way.

1. For *Level1* add an empty object, which you call *Init*
2. Create a C# script called *Init* and drag it over the *Init* object.
3. In the *Init* script write the following (this deletes 'everything' the first time):

```
private void Awake()
{
    PlayerPrefs.DeleteAll();
}
```

4. In the *PlayerController* script write the following:

```
// in Start()
point = PlayerPrefs.GetInt("Score");
```

```
// Every time you update points  
PlayerPrefs.SetInt("Score", point);
```

The last line saves *Score* and the first line retrieves *Score*.

Now the points should be transferred from level to level.

Inspiration for further development

Player jump animation

Multiple lives, save points (so you don't have to start completely over each time)

Better sound – near/far

Boxes that can be moved and climbed on

Rolling stone ball like Indiana Jones

Climbing up plants on walls, beanstalk

Build – release on the web

Unity Reference

This is a small Unity C# (C sharp) reference that describes some of the code snippets you will need. If you need a guide to basic C#, there is a good link at the bottom of this document.

Assets Folders

It is a good idea to organize all your files as you go, as it can otherwise be difficult to find them later. It is therefore a good idea to start by creating the following folders under **Assets**:

- *Scripts* (for all C# scripts)
- *Animator* (for all animations)
- *PreFabs* (for all PreFabs)

There will be a need for more folders as you develop your game. In addition, the assets you download from the *Asset Store* will automatically come in separate folders.

External Fields

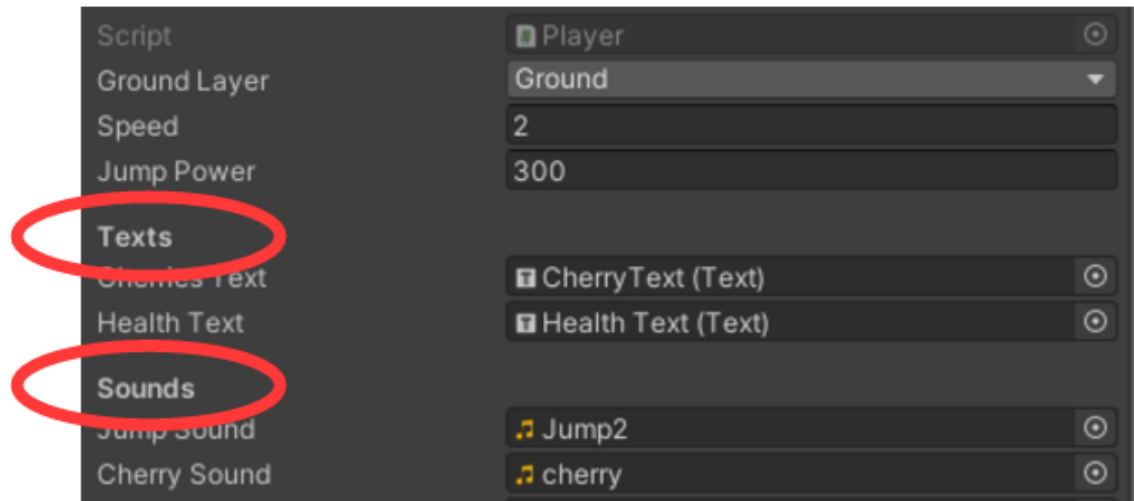
In many cases, you need to be able to see or access script variables from the Unity GUI. By writing [SerializeField] in front of a variable, it becomes visible in the **Inspector**.

```
[SerializeField] Text xxText;
[SerializeField] GameObject[] xxList;
```

The first line creates a single field, while the second creates a list of fields.

If you have many different fields, you can insert some cosmetic texts to make it more organized in the **Inspector**.

```
[Header("Texts")] // Creates a text box
[Space] // Creates an empty line (spacing)
[Tooltip("tip")] // Creates a tooltip for the next line, if you
hover the mouse over the field
```



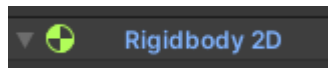
Here **Texts** and **Sounds** are examples of Header fields.

Components

Components are all the ‘properties’ that you have attached to your object, such as *Sprite Renderer*, *Colliders*, etc.

You often need to be able to access these components and they can be ‘retrieved’ in the following way:

```
xxComp = GetComponent<Rigidbody2D>();
```



The individual component names can be found in the **Inspector**, where you just need to remove any spaces from the name. For example, Rigidbody 2D -> Rigidbody2D

The **Transform** component is always there and is accessed directly with transform.

If you have multiple of the same component, you should use:

```
xxCompList = GetComponents<CapsuleCollider2D>();
```

The function has gotten an 's' and xxCompList is a list of components.

Keys

When you need to read which keys are being pressed, you can advantageously use GetAxisRaw(), which unlike a specific key is a collection of keys that are more generic. For example, if you want to move a character to the sides, then both the A, D keys, left and right arrow keys will work as well as a joystick (Horizontal).

The individual combinations can be found in **Edit->Project Settings->Input Manager**.

```
float dir;  
  
dir = Input.GetAxisRaw("Horizontal"); // [-1, 0, 1]
```

For jumping, you should use this function, as it requires that you release the key before you can jump again.

```
float jumpKey;  
  
jumpKey = Input.GetButtonDown("Jump");
```

Movement

When you have figured out which movement of the character you need, the actual movement is done in the following way:

```
transform.Translate(xSpeed, ySpeed, 0); // Move  
  
rb = GetComponent<Rigidbody2D>();  
  
rb.AddForce(new Vector2(xForce, yForce)); // Jump
```

If you need to mirror your character so it looks in the right direction, it can be done like this:

```
sprite = GetComponent<SpriteRenderer>();  
  
sprite.flipX = true; // Flip image
```

A complete example could look like this:

```
[SerializeField] float speed;

Rigidbody2D rb;
SpriteRenderer sprite;

void Start()
{
    rb = GetComponent<Rigidbody2D>();
    sprite = GetComponent<SpriteRenderer>();
}

void Move(float dir)
{
    transform.Translate(dir * speed * Time.deltaTime, 0, 0);

    if (dir > 0)
        sprite.flipX = false;
    else if (dir < 0)
        sprite.flipX = true;
}
```

Instead of using `transform.Translate(...)`, you can also write:

```
rb.velocity = new Vector2(dir * speed, rb.velocity.y);
```

However, this can cause some problems with moving the *Player* sometimes (it ‘gets stuck’ in the ground). This is solved by changing the **Box Collider 2D** to **Capsule Collider 2D**.

Jump

Before jumping, it is important to check if you are standing on the ‘ground’, as you could otherwise jump in the air! Here is shown one of the ways this can be solved. You move your ‘collision box’ slightly down and see if it overlaps with the ground.

```
[SerializeField] LayerMask groundLayer;

void Update()
{
    if (jumpKey && GroundCheck())
        Jump();
}

bool GroundCheck()
{
}
```

```

    var coll = GetComponent<CapsuleCollider2D>();
    return Physics2D.CapsuleCast(coll.bounds.center, coll.bounds.size, CapsuleDirection2D.V
    );
}

```

Ground Layer must be set in the **Inspector** to the layer that is *Ground*.

Here a *CapsuleCollider* is used, but it works with other colliders as well.

The number 0.1f indicates how much the collider is moved down. You can adjust this number a bit to get the best effect.

Collision

When two objects, both having a **Collider** component, touch each other, you have a collision. A collision can either be a ‘normal’ collision or a ‘trigger’ collision. A ‘normal’ collision will prevent the two objects from overlapping each other, for example like a person walking on a floor, while a ‘trigger’ collision is merely an indication that two objects overlap, like when a person should pick up an item. If one of the two objects has the ‘trigger’ set, it becomes a ‘trigger’ collision. The ‘trigger’ is set in the **Inspector** under the **Collider**.



In the script, you can then create an action on the collision in the following way:

Normal collision:

```

// Normal collision:
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.name == "Player")
    {
        // ...
    }
}

// Trigger collision:
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.CompareTag("Cherry"))
    {
        // ...
    }
}

```

For this to work, at least one object must also have a *Rigidbody* component. Note that two different ways to ‘see’ which object you hit are shown – this is

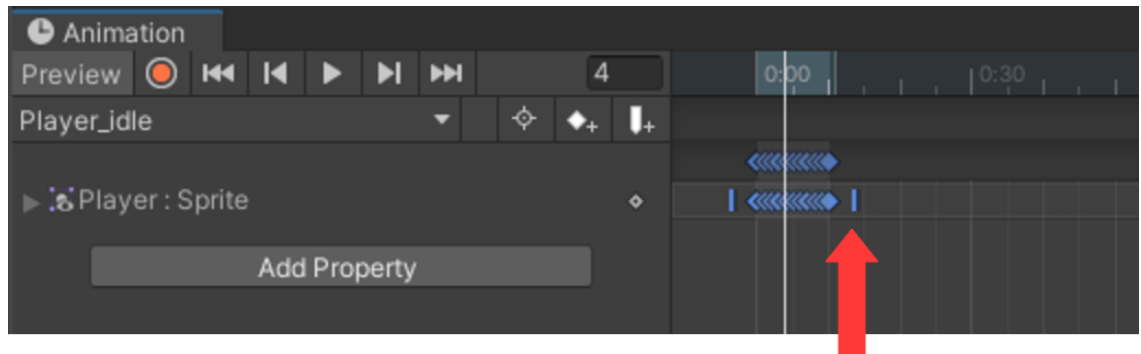
not dependent on the trigger method, but only shown so there is something to choose from.

Animation

When you want your character and objects' graphics to move, you need to use an Animator. This is described in this video: <https://youtu.be/GChUpPnOSkg?list=PLrnPJCHvNZuCVTz6lvhR81nnafla-b67U&t=14>

Here is how you create your first animation for your *Player* (requires you to have watched the video...). If you don't have a folder called *Animator* under *Assets*, it should be created first.

1. In the *Animator* folder, **Create->Animation** and call the file something 'sensible', for example *Player_idle*.
2. Then drag the *Player_idle* file to the *Player* object in **Hierarchy**. (This also creates a *Player controller* the first time you do it.)
3. Then you need to open the animation window: **Window->Animation->Animation**.
4. Then select the *Player* object in **Hierarchy**.
5. Now select all the images that make up the animation and drag them over into the **Animator**.
6. You can now see how it looks by pressing *play*, and adjust the speed by moving the blue line to the sides (see image below).
7. If the animation should continue, you must remember to set *Loop Time* in the **Inspector**.



When you are done with the animation window, you can advantageously dock it to one of the other windows you have, so it is easier to find next time.

Multiple Animations (Advanced)

If you need several different animations for the same character, like standing and running, you repeat the steps above for each animation.

Now you need to add some transitions and parameters to control it with (see video). These parameters can then be accessed from the script in the following way:

```
animator = GetComponent<Animator>();

animator.SetTrigger("Die");

animator.SetFloat("Velocity", speed);
```

Remember that the parameters must be spelled exactly the same as in the **Animator** (Die/Velocity)!

Sound

To get sound in your games, you must first find some sound files and place them in **Assets/Sounds/**. For the *object* that you want sound support for, you must first add an **AudioSource** component. In the script for that *object*, you do the following:

```
[SerializeField] AudioClip jumpSound;

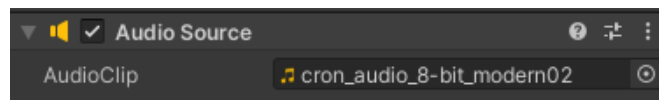
AudioSource audioPlayer;

void Start()
{
    audioPlayer = GetComponent<AudioSource>();
}

void Update()
{
    if (jump)
        audioPlayer.PlayOneShot(jumpSound, 1.0f);
}
```

You must then drag the sound file over to the **Jump Sound** field in the **Inspector** that you want to use.

Music



If you want background music, you should add an **AudioSource** component to the **Main**

Camera object and then drag a sound file over to the **AudioClip** field and set the **Loop** flag.

Game Text



You insert text on the screen in the game by creating an object with **UI->Text**. (2021.3 **UI->Legacy->Text**) This will automatically create two objects: **Canvas** and **the object**. This object must then be 'linked' to the script in the score field – see below.


After you have created the object, it can be configured in the **Inspector**. When you make the size larger, the text can suddenly “disappear”. This is because the box that the text is in is too small. This is fixed by selecting the text and then adjusting the box size in the **Scene**.

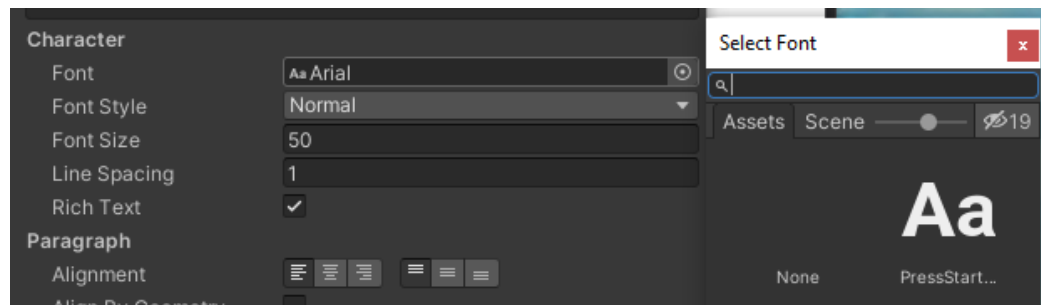
```
using UnityEngine.UI;  
  
[SerializeField] Text score;  
  
score.text = "Score: " + point;
```

Link: <https://youtu.be/pXn9icmreXE?list=PLrnPJCHvNZuCVTz6lvhR81nnaf1a-b67U&t=966>

Fonts

By default, Unity only has one font. You can easily install more fonts and get a cooler look that way. The fonts can be found, among other places, at the link at the bottom of this document.

Create a folder under **Assets** and call it **Fonts**. The fonts you want to use should then be dragged into this folder. When this is done, the font can be used in the text object by clicking on  next to **Font**.



Static Text

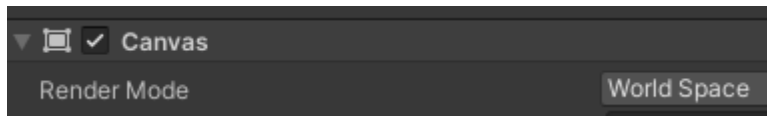
If you want to create text that doesn't follow the camera, like a sign, you can do the following.

1. Create a square object (**2D Object->Sprites->Square**) and call it for example *Sign*.



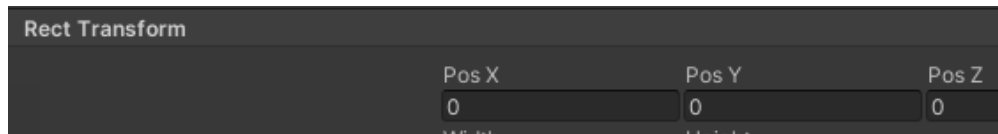
2. Adjust **Scale** so the square gets the right size.

3. Select *Sign* and insert text (**UI->Text** or **UI->Legacy->Text**).



4. In **Hierarchy** under *Sign* select *Canvas* and in the **Inspector** under *Canvas* set **Render Mode** to **World Space**.

Still in the **Inspector** under **Rect Transform** set **Pos X**, **Pos Y** and **Pos Z** to 0.



In **Hierarchy** select *Text* and set **Scale** to 0.02 for both X and Y and change **Text** to what it should be.



Try the game and it should look like this now (my text is “<- This way”):



The color of the sign is adjusted under the *Sign* object, while the text is formatted under the *Text* object.

Prefabs

Prefabs are copies of entire objects (templates), which can then be inserted again and again. This is the way you can copy an object many times in the same game, such as enemies and coins.

They are created by using the mouse to click on the desired object and then dragging it down into the **Prefabs** folder under **Assets**. (If the folder doesn't exist, it should be created first). After that, they can be inserted into the game by dragging them from the **Prefabs** folder into the game (**Scene**).

Particles

Particles are set up in the GUI and can then be controlled from a script in the following way.

```
[SerializeField] ParticleSystem explosionParticle;

explosionParticle.Play();

explosionParticle.Stop();
```

Dynamically Create and Delete an Object

If you want to dynamically create an object, for example for a shot, you should do the following. Create a Prefab of the object in the GUI.

In the script, create a variable to hold the Prefab that you want to create.

```
[SerializeField] GameObject objPrefab;
```

You must then go into the GUI and 'drag' the Prefab you want to use into the field that has appeared in the script (**objPrefab**).

After that, the Prefab can be generated in the following way:

```
Instantiate(objPrefab, pos, objPrefab.transform.rotation);
```

Where pos is a vector that specifies where the object should be generated.

If a dynamic object 'disappears' off the screen or is otherwise not active anymore, it is important that it is deleted again, as there will otherwise be far too many 'dead' objects over time. You create a script that should be attached to the object that should expire. The script can look like this – here you examine whether the object is outside the area, and if so it is deleted.

```
void Update()
{
    // Destroy object if x position is less than leftLimit
    if (transform.position.x < leftLimit)
    {
```

```

        Destroy(gameObject);
    }
}

```

Script Communication

If you need a script to be able to ‘see’ a state or variable in another script, it can be done in the following way.

In this example, there is an object called ‘**Player**’ that has a script called **PlayerController** and this script has a *public* variable called `gameOver`;

If you want to access this `gameOver` variable from another script, you should do the following:

```

PlayerController playerControllerScript;

void Start()
{
    playerControllerScript = GameObject.Find("Player").GetComponent<PlayerController>();
}

void Update()
{
    if (playerControllerScript.gameOver == false)
    {
        // do something
    }
}

```

Utility

Time Handling

For each frame in the game, time has passed since the last frame. This time is useful when you want to move a character in each loop, for example. The time is in seconds (i.e. a decimal number):

`Time.deltaTime;`

So if you want to create a speed that is independent of the frame rate, it can be done in the following way:

```

move = speed * Time.deltaTime;

```

where `speed` specifies a factor for how fast something should move.

When you want to create a delay between, for example, two shots, you can do the following:

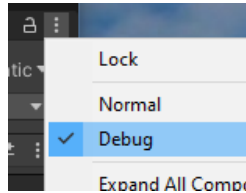
```
timeWait += Time.deltaTime;

if (timeWait > 2) { // 2 seconds
    // <shot>
    timeWait = 0;
}
```

Debugging

When you need to debug your C# scripts, you can advantageously add some print statements to the code to see how different values change. You can add debug text in the following way, which can be seen in the **Console** when the program runs.

```
Debug.Log("Text" + value);
```



You can also click on the three dots in the right corner of the **Inspector** GUI and select Debug. This way you can see all variables in the program while it's running in the **Inspector** under **Script**.

Interval check

If you have a value that should stay within an interval, for example number of lives, you can use a Clamp function.

```
newLevel = Mathf.Clamp(level - 1, 0, 3);
```

The first parameter is the new value to be tested, next is min and then max. The function returns a valid value as close to the new value as possible.

Delay

Sometimes you need to create a delay between two operations. This can be done in several different ways.

Invoke

Invoke takes two parameters: A function name and a number that specifies how long it waits before calling the function.

```

Invoke("RestartLevel", 2);

...

private void RestartLevel()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}

```

Coroutine

This requires creating a new function for this and calling it in a ‘special’ way.

```

StartCoroutine(xxDelay());

...

IEnumerator xxDelay()
{
    // <do something>
    yield return new WaitForSeconds(2); // Wait 2 seconds
    // <do something>
}

```

Where xxDelay is the name of the new function.

Import Unity Package

Import a Unity package.

In the GUI select: **Assets->Import Package->Custom package** and select the file. In the window that appears, click **Import** in the bottom right corner. The package is now imported and located under **Assets**.

Special Scripts

Moving Objects

Here is a script that can make an object move between a series of points. It can be used to create a moving platform, but also enemies that move in a pattern. The idea is described in this video:

Link: <https://youtu.be/UlEE6wjWuCY?list=PLrnPJCHvNZuCVTz6lvhR81nnaf1a-b67U&t=198>

The script is created in **Assets/Scripts/** and the following content is added: (The script is part of the *2dplatform* starter package)

WaypointFollower.cs

```
[SerializeField] GameObject[] waypoints;

int currentWaypointIndex = 0;

[SerializeField] float speed;

SpriteRenderer sr;

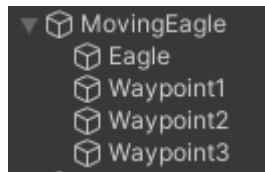
private void Start()
{
    sr = GetComponent<SpriteRenderer>();
}

void Update()
{
    int prevWaypointIndex;
    if (Vector2.Distance(waypoints[currentWaypointIndex].transform.position, transform.position) > speed * Time.deltaTime)
    {
        prevWaypointIndex = currentWaypointIndex;
        currentWaypointIndex++;

        if (currentWaypointIndex >= waypoints.Length)
            currentWaypointIndex = 0;

        // Change direction for default layer only
        if (gameObject.layer != LayerMask.NameToLayer("Ground"))
        {
            if (waypoints[currentWaypointIndex].transform.position.x > waypoints[prevWaypointIndex].transform.position.x)
                sr.flipX = true;
            else
                sr.flipX = false;
        }
    }

    transform.position = Vector2.MoveTowards(
        transform.position,
        waypoints[currentWaypointIndex].transform.position,
        Time.deltaTime * speed
    );
}
```



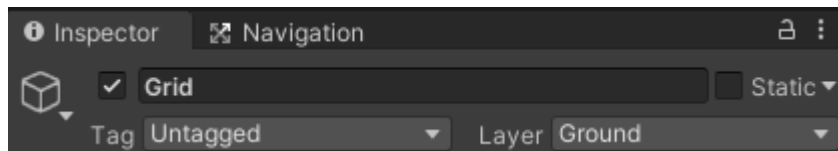
The script automatically turns the object so it points in “the right direction”, except if it is on the **Ground** layer. This way the script can be used for both enemies that move and platforms that should move.

It can be advantageous to create an object group in the **Hierarchy**, so the object and waypoints are collected together.

You add the script to the object that should move. In the **Inspector** you create the number of Waypoints you need – typically 2. Next, you create a corresponding number of Waypoints in the **Hierarchy** and call them *Waypoint1*, *Waypoint2*, ... They should then be dragged into the **Inspector** one by one.



If it is a platform that should move, it is important that you have set the **Layer** to *Ground*. If it doesn’t exist, it should just be created.



Stick to Moving Platforms

If you stand on a platform that moves, it is important that the character moves together with the platform, otherwise the character will fall off. The script is created in **Assets/Scripts/** and added to the platforms that should have this property. It is important that your character is named *Player*, otherwise you can just change it in the script.

(The script is part of the *2dplatform* starter package)

StickyPlatform.cs

```
private void OnCollisionEnter2D(Collision2D collision)
{
```



```

        if (collision.gameObject.name == "Player")
        {
            collision.gameObject.transform.SetParent(transform);
        }
    }

    private void OnCollisionExit2D(Collision2D collision)
    {
        if (collision.gameObject.name == "Player")
        {
            collision.gameObject.transform.SetParent(null);
        }
    }
}

```

Special setup

Move camera to follow player

If you want the camera to follow the character, you need to create the following script and add it to the **Main Camera** object. (Remember to set the character in the **player** field in the **Inspector**.)

(The script is part of the *2dplatform* starter package)

CameraFollower.cs

```

[SerializeField] Transform player;

void Update()
{
    transform.position = new Vector3(player.position.x, transform.position.y, transform.position.z);

    if (player.position.y > 5)
    {
        transform.position = new Vector3(transform.position.x, player.position.y - 5, transform.position.z);
    }

    if (player.position.y < -3)
    {
        transform.position = new Vector3(transform.position.x, player.position.y + 3, transform.position.z);
    }
}


```

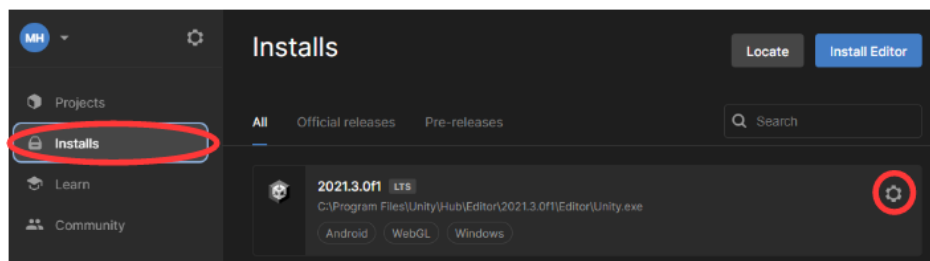
Smooth surface

If you don't want to be able to "stick" to walls and platform sides, then they need to have a smooth surface. This is done by creating a physical surface under the **Assets** folder (**Create->2D->Physics Material 2D**) and setting **Friction** to 0. Then select it in the **Collider** under **Material** for the surface you want to be 'smooth'.

WEB release

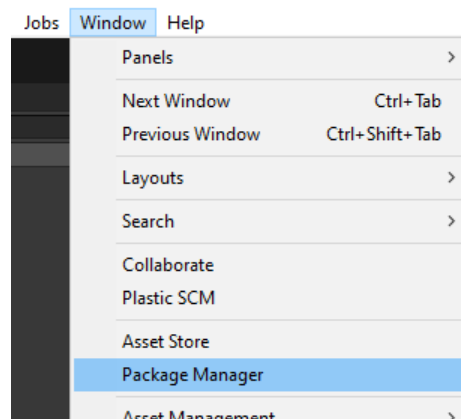
You can get your game released on the web, so it can be played from a browser without installation. That way others can try your game.

1. Open Unity HUB and click on **Installs** and then on .
2. Select *Add modules*
3. Select *WebGL Build Support*.

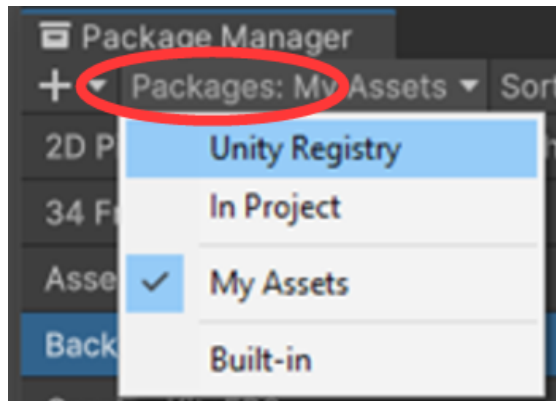


4. Click **Install** and wait for the installation to complete.
5. If Unity GUI is open it must be restarted after the installation

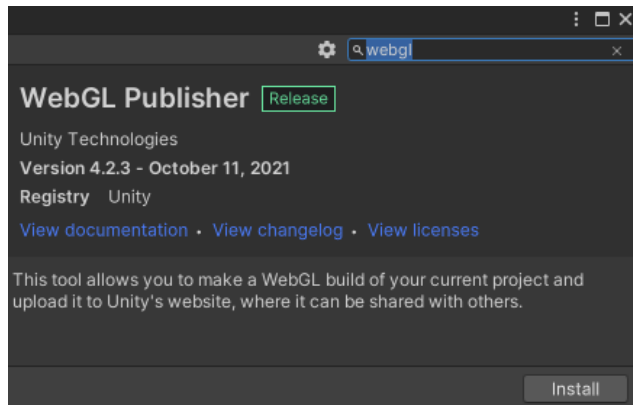
In Unity GUI: **Window->Package Manager**.



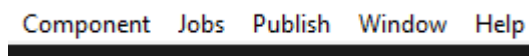
Click on **Packages** and select **Unity Registry**.



In the search field on the right, enter *webgl* and click **Install** in the bottom right corner.



When the installation is complete, a new menu item has appeared at the top of the screen next to **Window** called **Publish**.



1. Click on **Publish->WebGL Project**.
2. Select **Build and Publish** and then **Switch to WebGL**.
3. Click on **Select Folder** (without selecting anything).

Now the entire project needs to be compiled and uploaded to the web. This takes some time. When it's done, it opens a window in the browser. Here you can enter a name for the game and add an icon. Remember to click **Save**. Now the game can be played by clicking **Play**. The link to the game is in the address field at the top of the browser (as usual).

If you make changes to your game, you only need to repeat the process from clicking on **Publish**.

Android support (advanced)

Here is a general (brief) description of how you can get the game to run on an Android device (phone or tablet). **This will require that you fight with it a bit yourself to get it working!**

An Android device has no buttons, so the first step is to create some buttons on the screen, so you have something to press.



There is a Unity package that contains these buttons and the logic needed to read them here:

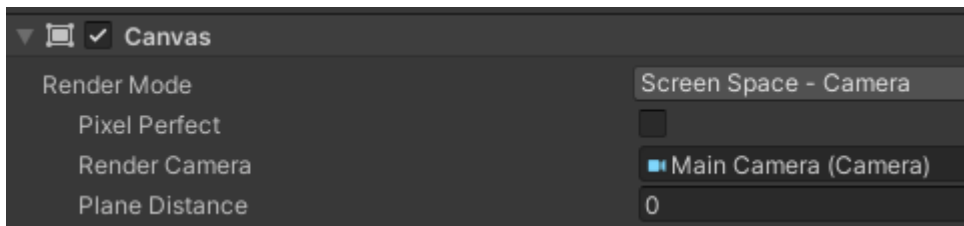
<https://github.com/mhfalken/unity/blob/main/android-knapper.unitypackage>

Start by importing this package.

The buttons must ‘lie’ under *Canvas* in the **Hierarchy**, but before they can do that, it needs to be changed a bit.

For *Canvas* you need to correct the following in the **Inspector**:

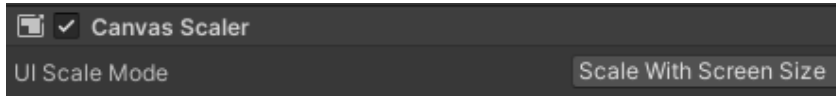
1. **Render Mode:** Screen Space – Camera
2. **Render Camera:** Drag *Main Camera* into this field.
3. **Plane Distance:** 0



Now the current texts (Score etc.) will most likely become ‘invisible’. This must be fixed by messing with the **Pos Z** values for the texts in the **Inspector**.

Remember they must be visible in both **Scene** view and **Game** view to work! Test that everything works as before.

Before continuing, it's important that the following option is set, as described earlier (if not, there's a bit of extra work here...).



Under *PreFabs* find *Knapper* and drag it up over *Canvas* in the **Hierarchy**, so it looks like this:



Now some buttons should appear. If they are not visible in **Game** view, then **Pos Z** needs to be fixed. It should now look like the above image of the game with the buttons.

We now need to add some lines to our *PlayerController* script. The lines are 'wrapped' in compiler options, so they are only active when you are in Android mode. Where the individual lines should go depends a bit on how the code already looks, so it may well be that the guide doesn't fit perfectly with your code!

At the very top of the file (line 1) should be the following (it's to make it easier to test and debug):

```
#if UNITY_ANDROID

#define SCREEN_BUTTONS

#endif
```

Under 'global' variables should be:

```
#if SCREEN_BUTTONS

Knapper knapperObj;

#endif
```

In *Start()* should be (it doesn't matter where):

```
#if SCREEN_BUTTONS

Debug.Log("SCREEN_BUTTONS");
```

```
knapperObj = GameObject.Find("Knapper").GetComponent<Knapper>();

#else

GameObject.Find("Knapper").SetActive(false);

#endif
```

These lines should be in Update() immediately after you have read *dir* and *jump* with Input.GetAxisraw().

```
#if SCREEN_BUTTONS

dir = knapperObj.GetDirX();

jump = knapperObj.GetJump();

#endif
```

To test if it works, you should put the following line in line 4 (it forces the code to activate the buttons even if you are not in Android mode):


```
#define SCREEN_BUTTONS
```

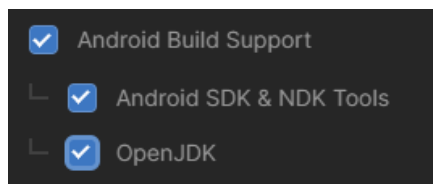
Run the game and click with the mouse on the buttons and see that they work (the keys don't work in Android mode). When it works, delete the #define SCREEN_BUTTONS line again.

When you are not in Android mode (or debug) the buttons disappear automatically when you run the game. Try and see that they are gone now when the game runs and that it otherwise works normally.

Game over to Android device

Now we need to get the game over to the Android device.

1. Open Unity HUB and install Android support (Install->>Add modules). Unity GUI must be restarted afterwards for it to work.

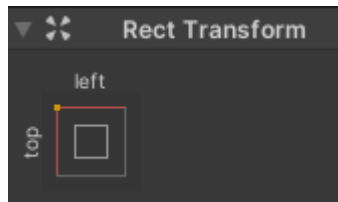


2. Put the Android device in *Developer mode* (search online – it's a bit surprising what you have to do and can depend on the model). This adds a new menu item.

3. Open **Developer options** (Developer mode) and select **USB debugging** (USB-debug mode)
4. Connect the Android device to the computer with a USB data cable and give it permission to debug the device.
5. In Unity GUI select **File->Build Settings** and select *Android* and click **Switch Platform**.
6. Then click **Build And Run**, give the file a name and cross your fingers. With a bit of luck, the game should start up on the Android device after some time.

The game is now on the Android device (look under programs) and can be played 'offline'. Name etc. can be set in **Player Settings** under **Build Settings** (bottom left corner).

If the texts (Score etc.) are not positioned correctly on the screen, you can fix it by using *anchors* for the individual texts.



Links

Good starting series

<https://www.youtube.com/playlist?list=PLrnPJCHvNZuCVTz6lvhR81nnafla-b67U>

Shooting example

<https://www.youtube.com/watch?v=PUpC44Q64zY&list=PLgOEwFbvGm5o8hayFB6skAfa8Z-mw4dPV&index=4>

Tips videos

TOP 10 UNITY TIPS - 2017 TOP 10 UNITY TIPS #2

Unity manual

<https://docs.unity3d.com/2020.2/Documentation/ScriptReference/index.html>

Unity assets store

<https://assetstore.unity.com>

Fonts

<https://fonts.google.com/> (E.g. 'Press Start 2D')

Danish C#/Unity document

<https://github.com/Grailas/CodingPiratesAalborg/blob/master/Guides/Hj%C3%A6lpeguide.pdf>