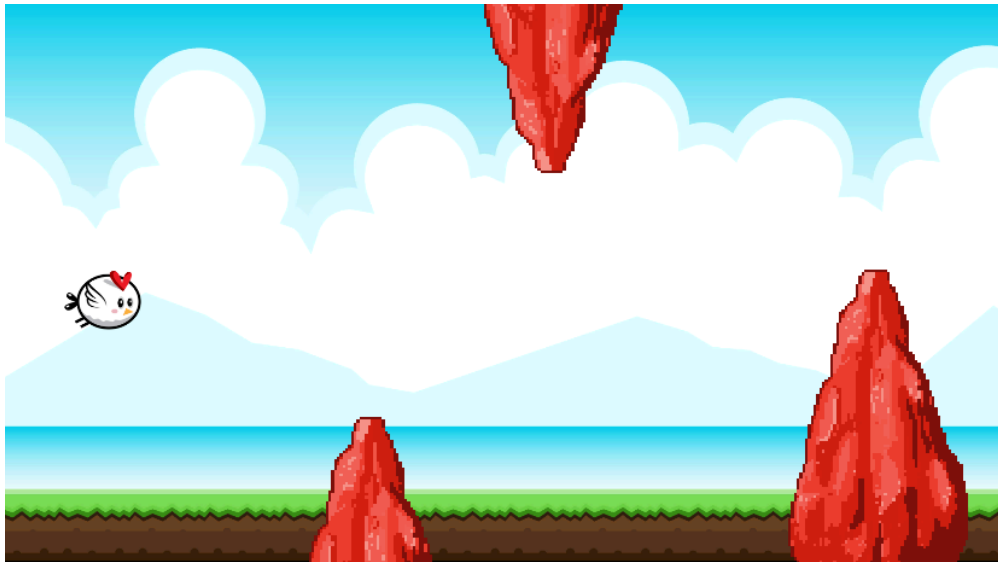


# Flappy Chicken

Totally unrelated to any probably similar mobile games

In this exercise you will create the “Flappy Chicken” game, a flappy bird clone. You will learn basic Unity scripting and create the game completely bottom up yourself.

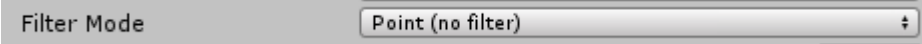


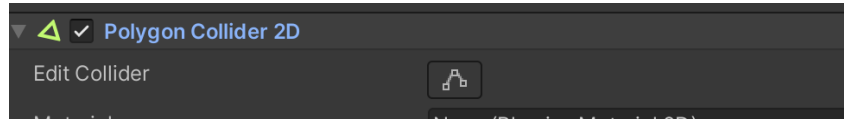
## Part 1: Setup

We will do this **together**, follow along in the class. For people at home:

1. Create a new Unity project (built-in 2D) with the name “FlappyChicken”.
2. Import the assets found on canvas by extracting the zip “FlappyChicken\_Assets.zip” and dragging the folders into the Project window (or copying the folders into the “assets” folder of your project in the explorer).
3. In the “Assets/Scenes” folder there is a scene file called “SampleScene”. Rename that to Game and save.  
\*If there is no “Scenes” folder, create one and also create a new scene called “Game” and save it in there. Open this scene to create the game in.\*
4. Put the “chicken” into the scene as a *GameObject* from the “chicken-fly-down” sprite. Rename the *GameObject* to “Chicken”
5. Add a *PolygonCollider2D* to the Chicken *GameObject* and fit it to the sprite if Unity doesn’t do it automatically.
6. Create the obstacle prefabs:
  - a. Put a red crystal into the scene and name it “ObstacleDown”.
  - b. See that it is way too small, correct the import settings for “crystal-red” as following:



- ii. 
- iii. Read more on how to import sprite properly in the appendix.
- c. Add a *PolygonCollider* to it and fit it properly. It will fit sort-of ok-ish by itself, but you can adjust by using the Edit Collider Tool:



- d. Right click the prefab that you just created in the Project view. Select Create->Scene->Prefab Variant.
- e. Rename the prefab variant to *ObstacleUp*
- f. With the prefab selected, modify it's Transform's scale in the inspector in the Y axis to -1. This will flip the object upside down.
- g. Drag the *ObstacleUp* into the level.

## Part 2: Make a level

Create a small level out of 3-5 obstacles in front of the chicken. (see page 1 picture, background will be added later)

## Part 3: Chicken Controller, Movement

The "chicken" needs to fly left to right at a constant speed and "flap", or receive an upward push, when the spacebar is pressed.

1. Create a script named *ChickenController* and add it as a component to the "chicken".
2. **Continuous Movement:**

- a. To create movement of objects, you have several options. We will go with a change of `transform.position` first. Create the field

```
public float speed;
```

in *ChickenController*

- b. Declare a private void method *MoveChickenForward()*

```
private void MoveChickenForward()
{
    ...
}
```

- c. Add the following line to *MoveChickenForward()*:

```
transform.position += Vector3.right * speed * Time.deltaTime;
```

Try to understand the line, what exactly is happening here? The formula of `direction * speed * 1/fps (deltaTime)` is very important to understand. Ask me if you have trouble with it.

- d. Call `MoveChickenForward()` from the `Update()` method

```
private void Update()
{
    MoveChickenForward();
}
```

- e. Save, enter a proper value for speed in the inspector (start with 1) and hit play. Discuss with your neighbour why this line makes the “chicken” move.

Read more on `Time.deltaTime` in the Appendix.

3. Adding a “flap”:

- a. The “chicken” needs to have **gravity** and fall down:

- i. Add a *Rigidbody2D* component to the “chicken”.

- b. In `ChickenController` we now need to access this *Rigidbody2D*, to tell it to bounce up when hitting space:

- i. Add the following field to `ChickenController`, to hold a reference to the *Rigidbody2D* component:

```
private Rigidbody2D chickenRigidbody;
```

- ii. In `Start()`, add the following line to grab the *Rigidbody2D* and save the reference:

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

Read more on how `GetComponent<>()` works in the appendix.

- iii. Add the following field to `ChickenController` to save how much we push the “chicken” up when hitting space:

```
public float flapForce;
```

- iv. For readability in the inspector, add Headers to create two sections: “Movement” and “Jumping”

```
[Header("Movement")]
public float speed;
[Header("Jumping")]
public float flapForce;
```

- v. Create a private void method `HandleJump()`

```
private void HandleJump()
{
    ...
}
```

- vi. Add this to `HandleJump()` to handle the keypress and push the chicken upwards:

```
Vector2 vel = myRb2D.velocity;
if (Input.GetKeyDown(KeyCode.Space))
{
    vel.y = flapForce;
    myRb2D.velocity = vel;
}
```

- vii. Call `HandleJump()` in `Update()` before `MoveChickenForward()`

```
private void Update()
{
    HandleJump();
    MoveChickenForward();
}
```

4. There are 2 main ways to move objects in Unity: **Position** based and **Velocity(Physics)** based. Above we implemented:

```
transform.position += Vector3.right * speed * Time.deltaTime;
```

This is **Position** based movement: We change `transform.position` every frame. To be consistent we should only use one type of movement in the controller. We will change the modification of `transform.position`, in `MoveChickenForward()` to also have the *Rigidbody2D* velocity take care of horizontal movement. We will use **Velocity(Physics)** based movement:

- a. In `MoveChickenForward()` remove:

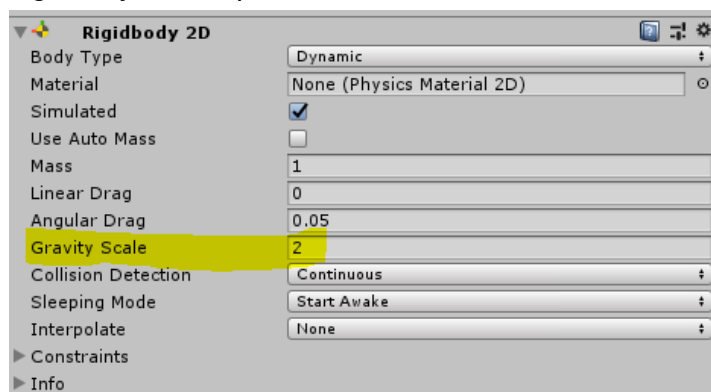
```
transform.position += Vector3.right * speed * Time.deltaTime;
```

- b. Change `MoveChickenForward()` to:

```
Vector2 velocity = chickenRigidbody.linearVelocity;
velocity.x = speed;
chickenRigidbody.linearVelocity = velocity;
```

Discuss with your neighbour: Why do you not need `Time.deltaTime` now for the horizontal movement?

5. Test your controller and change `speed`, `flapForce` and the “Gravity Scale” of the *Rigidbody2D* component until it feels nice.



## Part 4: Chicken Controller, Collision

We now need to have the "chicken" collide with the obstacles and let the game restart when they are hit.

1. Prepare the obstacles: Just like with the asteroids from the last lab, we need a way to identify what we are colliding with. Add the "Obstacle" **Tag** to the project and give it to the obstacles. Since ObstacleDown is the source Prefab, changing the tag on it will also change it on the Prefab Variant ObstacleUp.
6. Create an empty method in your ChickenController class called `ReloadScene()`

```
private void ReloadScene()
{
}
```

2. The function `OnCollisionEnter2D(Collision2D col)` can accept **collision events** from the Unity physics engine. `Collision2D col` will hold data on the collision, for example the *GameObject* we collided with. Add the following function to `ChickenController`:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Obstacle"))
        ReloadScene();
}
```

3. Add the following code to `ReloadScene()`

```
SceneManager.LoadScene(SceneManager.GetActiveScene().name);
```

`SceneManager` can load us scenes and handle scene transition. But it needs to be imported for the file, add this to the top of `ChickenController`:

```
using UnityEngine.SceneManagement;
```

4. Test if the "chicken" collides with the obstacles and the scene gets restarted.

## Part 5: Camera movement

The camera should follow the chicken, when it is progressing to the right. But only horizontally! This means we have to **restrict** movement of the camera to the **X-Axis**.

1. Create a script called `CameraController` and attach it to the "Main Camera" *GameObject*.
2. Add the following fields to the script:

```
public Transform cameraTarget;
```

```
public float followOffsetX;
```

This is the target to follow (as transform, since we are only interested in the position, and the offset to follow it on the X-Axis).

3. Create an private void method `FollowCameraTargetHorizontally()`

```
public void FollowCameraTargetHorizontally()
{
    ...
}
```

4. Call `FollowCameraTargetHorizontally()` inside the `Update()` method

```
private void Update()
{
    ...
    FollowCameraTargetHorizontally();
}
```

5. You can let the camera follow, by using this code in the `FollowCameraTargetHorizontally()` function:

```
Vector3 targetPosition = transform.position;
targetPosition.x = cameraTarget.position.x + followOffsetX;
transform.position = targetPosition;
```

Because of the way, `transform.position` is implemented in the engine, you can not modify `transform.position.x` directly. So we have to grab it and store it first in `targetPosition`.

6. Save your script. Then select the "Main Camera" object in the scene (where the `CameraController` script should be attached) drag the "chicken" (Player) object from the scene into the `player` field in the inspector. Then find a good value for `offsetX` by trying different values in the inspector.

## Part 6: Background music

Now it's time to add some nice music to the game.

1. Create an empty *GameObject Audio*
2. Inside *Audio*, create an empty *GameObject* and call it "Background Music", attach it as a child to the camera with no offset (position: x=0, y=0. y=0).
3. Add an *AudioSource* component to "Background Music".
4. Drag the Sounds/music sound asset into the "AudioClip" field.
5. Check "Play On Awake" and "Loop".
6. Listen to some sweet tunes.

## Part 7: Chicken Sounds

The "chicken" should have some **sounds** as well, when we are "flapping".

1. Add an *AudioSource* component to "chicken". Make sure "Play On Awake" and "Loop" are NOT checked.
2. Add the chickenSound-1 sound as *AudioClip*.
3. In *ChickenController* add the field (*Note: Adding "\_" before your private variables will help you identify them when receiving suggestions by the code editor while working on your Unity scripts*)

```
private AudioSource _chickenAudioSource;
```

4. You can grab the *AudioSource* component like you did the *Rigidbody2D* in `Start()` with `GetComponent<AudioSource>()` ;
5. You can make the *AudioSource* play by using `myAudioSource.Play()` ; Do that in the place where the "flapping" happens. Try to do it with a new method `PlayFlappingSound()`
6. Test your chicken sounds.
7. *AudioSources* can also play back different *AudioClips* with `myAudioSource.PlayOneShot(AudioClip clip)` ;
8. Create the functionality to play back a **random** chicken sounds when "flapping"
  - a. First create a (public) array of type *AudioClip* as a field in *ChickenController*.
  - b. Put the 5 different chicken sounds in the array in the inspector.
  - c. Then pick a random *AudioClip* from it when "flapping" using `Random.Range(int min, int max)` where min is 0 and max is the arrays length.
  - d. Instead of using `myAudioSource.Play()` ; , play back the randomly chosen clip with `myAudioSource.PlayOneShot(AudioClip clip)` ;

## Part 8: Chicken Animations

The chicken should have some flapping animation. In our case we do this very simply by just changing the sprite when moving up or down.

1. Add the two `public` fields `spriteUp` and `spriteDown` of type `Sprite` to `ChickenController`. If you want, wrap them under a Header “Animation” in the Inspector.
2. Drag in the respective sprites `chicken-fly-down` and `chicken-fly-up`.
3. Add a field `_chickenSpriteRenderer` for the `SpriteRenderer` component and grab the reference in `Start()` just like with the `AudioSource`.
4. In `Update()` you can test if we are moving up or down by asking if the `Rigidbody2Ds` velocity on the Y-Axis is bigger or smaller than 0.
5. Change the sprite accordingly by setting `mySpriteRenderer.sprite` to `spriteUp` or `spriteDown`. You can try to do this wrapped under a method `AnimateChickenSprite()`

## Part 9: Spawning Obstacles

*GameObjects* can be created on Runtime from *Prefabs* using `Instantiate(GameObject original, Vector3 position, Quaternion rotation);`. Where `original` should be a *Prefab* reference, `position` is the point where to spawn and `rotation` is an orientation using Quaternion notation (We will get what that is later, for now just use `Quaternion.Identity`, which is the 0/0/0 rotation).

1. Create a new empty *GameObject* “Obstacle Spawner” and attach it to the “Main Camera” as a child. Position it so it is slightly out the camera's view to the right, ahead of the game field.
2. Add a new script called `ObstacleSpawner` and add it to “Obstacle Spawner”.
3. Add two new `public` fields of type `GameObject` to `ObstacleSpawner`. One for the bottom obstacle and one for the top obstacle prefab. Assign the prefab references in the inspector.
4. Add the field `private float _currentTimer;` and `public float spawnTimer;` to `ObstacleSpawner`, `spawnTimer` is the time in between obstacle spawn and `_currentTimer` will keep track of the countdown.
5. Set the Spawn Timer to 2 in the inspector. You can change this later to a fitting value with testing.
6. The following code will count down a timer and call `SpawnNewObstacle()`; when it expires, then reset it, when placed in `Update()`. Add it to `Update()` in `ObstacleSpawn`:

```
_currentTimer += Time.deltaTime;
if(_currentTimer >= spawnTimer)
{
    SpawnNewObstacles();
    _currentTimer = 0;
}
```



7. Create the function `SpawnNewObstacles()` :

```
Vector3 spawnPositionDown = transform.position +  
    Vector3.down * Random.Range(distanceMin, distanceMax);  
Vector3 spawnPositionUp = transform.position +  
    Vector3.up * Random.Range(distanceMin, distanceMax);  
Instantiate(obstaclePrefabDown, spawnPositionDown, Quaternion.identity);  
Instantiate(obstaclePrefabUp, spawnPositionUp, Quaternion.identity);
```

You will need to add the fields `distanceMin` and `distanceMax` to `SpawnNewObstacles` and give them appropriate values.

8. Test out your spawn and observe that you can not see the obstacles spawned. This is because they are spawned at `z = -10` (the camera's `z` position). So the camera can not see them, they are effectively behind.
9. Modify the code so `spawnPositionDown` and `spawnPositionUp` have their `z` value set to 0 before you call `Instantiate(...)`.
10. Search for good values for `distanceMin`, `distanceMax` and `spawnTimer` by testing.

## Part 10: Making the chicken face in the direction it is moving.

Rotations are usually pretty difficult to master in Unity because they are based on the fairly complex maths of Quaternions. Luckily there is a “hack” in 2D where we can just assign the proper Vector to `transform.up` or `transform.right` to make the object rotate that way.

1. Create a local variable for the normalized direction of movement from the *Rigidbody2Ds* velocity in the `Update()` function of `ChickenController` (normalized means of length 1):  
`Vector2 normalizedDir = myRb2D.velocity.normalized;`
2. Assign the direction using: `transform.right = normalizedDir;`
3. (Optional) Change the hack using proper math, rotations and Quaternions according to the appendix.

## Part 11: Polishing

1. Now add the following functionality yourself: The camera should only follow the player, if the player is less than `offsetX` units to the left of it. This means, if the player is further than that away, the camera will stay stationary. This gives us a nice effect at the start, where the bird flies into the view from the left.

## Part 12: Twist (optional)

Add a twist of your own design to the game! Be creative and see how far you can push Flappy Birds basic design. Anything goes!

# Appendix

## Importing Sprites

When importing sprites, you want to make sure of two things: That they have the correct imported size, and the correct filter mode. Scaling any kind of object in Unity can often cause unforeseen complications due to various Computer Graphics related issues, so if possible we want to avoid that and import the assets already with the right settings, so they will have the scale 1/1/1 in the scene.

### Importing Size:

Size for sprites is defined in Pixels per Unit, meaning how many pixels of the sprite will cover exactly one Unit in the scene. That means, to LOWER that number is, the BIGGER the sprite.

### Filter Mode:

Since one pixel on the texture almost never exactly fits one pixel on the screen, the texture has to be interpolated and filtered to fit the screen pixels. The method for this is called filter mode. Bilinear or Trilinear filter mode interpolate pixels and blend near ones together, creating a smooth texture, this is good for LARGE textures and sprites (high resolution). Low resolution sprites and texture should use point filter mode, this picks the closest pixel on the texture and does NOT interpolate, giving sharp edges. This is good for PIXEL GRAPHICS and very SMALL textures and sprites.

## Time.DeltaTime

### Why framerate independence is so important

The amount of FPS (frames per second) can vary, depending on how powerful the hardware you operate on is. This means, if you do not take the execution time of a frame (`Time.deltaTime`) into account, an object in a game running at 120 FPS will move twice as fast as an object in a game running at 60 FPS. The formula `direction * speed * Time.deltaTime` will move an object into `direction` for `speed` units per second, no matter the FPS. `Update()` is called every frame, meaning there are more `Update()` calls on a faster computer. That is why we need to use `Time.deltaTime` to adjust any kind of movement or timing when used in this function.

## GetComponent<>()

When adding a script to a *GameObject* you have access to the `Transform` component directly through the `transform` field. Any other component can always be accessed using the `GetComponent<>()` function by just adding the component type in the `<>` (this is also

called generic typing). This function can also be used on any other `GameObject` type object or reference that you have to get access to components on THAT *GameObject*. For example in `OnCollisionEnter2D(Collision2D col)` you can use `col.gameObject.GetComponent<SpriteRenderer>()` to get access to the `SpriteRenderer` component on the object you collide with.

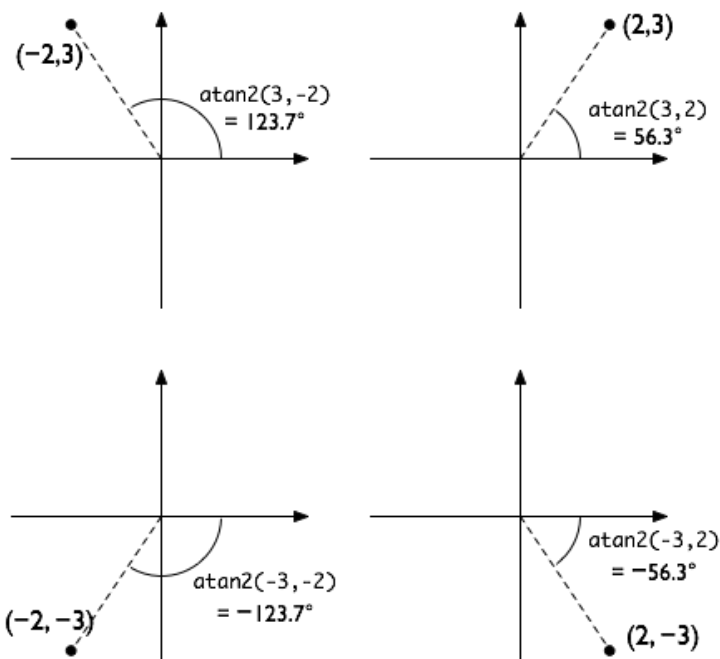
## Rotating Object in 2D with Quaternions and Angles

This is a task that is a bit more difficult than it sounds, as you have to deal with calculating angles and respecting the sprites default facing direction (up,down,left, right). Furthermore you have to set the rotation of an `GameObject` in Unity which requires work with Quaternions.

We need to convert a direction 2D vector into an angle first, then that angle into a quaternion. This is complicated to understand, but will actually result in only 2 lines of code:

1. **Calculating the correct facing angle:** Since we are in 2D we can use the very helpful function `Mathf.Atan2(float y, float x)` (yes y is first). `Atan2` is an extension of the tangents that allows conversion of a `Vector2(x,y)` into an angle, we can then use that angle to rotate the Chicken. `Atan2` is defined as:

$$\text{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{if } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{if } x < 0 \text{ and } y < 0 \\ +\frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0 \end{cases}$$



It returns the angle of a vector  $V = \text{Vector2}(x, y)$  to the positive x-axis (to the right) in **radians**. We can use the horizontal and vertical input values from earlier to create

the vector  $V$ . If the length of  $V$  is bigger than 0 (see that `atan2` is undefined if both  $x=0$  and  $y=0$ ), normalize the vector and calculate the angle with `Mathf.Atan2(float y, float x)`. Now we have to take the sprites default facing direction into account. The resulting angle is the inputs angle relative to the right facing direction ( $x$ -axis). If the sprite, for example, faces up, so we need to subtract `Mathf.pi * 0.5f` from the angle to align it with the sprite.

2. **Applying the angle to the GameObject.** In order to apply the angle you will need to convert it into a Quaternion. A Quaternion is a structure that represents a three dimensional rotation by rotating an object  $\theta$  degrees around an axis  $A$ . The `Quaternion` class has a range of helper functions to create Quaternions from different data. The best suited for us is `Quaternion.AngleAxis(float angle, Vector3 axis)` as we know both, the angle and the axis of rotation. The angle `Atan2` gives us is in **radians**, the angle we need is in **degrees**, so multiply it with `Mathf.Rad2Deg` to convert. The axis we want is the world's global  $z$ -axis, this equals to `Vector3.forward`, or new `Vector3(0, 0, 1)`. Just assign the resulting Quaternion to `transform.rotation`.

3. **The proper code to convert a direction into a 2D rotation is therefore:**

```
Vector2 dir; // get direction from anywhere
dir.Normalize();
float angle = Mathf.Atan2(dir.y, dir.x);
transform.rotation = Quaternion.AngleAxis(angle * Mathf.Rad2Deg, Vector3.forward);
```

This is the default for a sprite facing to the right. Any offset for sprites facing other directions need to be added in the last line.

#### *Tips for rotating objects in 2D:*

In 2D space you always want to rotate around the global  $z$ -axis. So

`Quaternion.AngleAxis(float angle, Vector3.forward)` is your friend all the way. Unity is an engine originally developed for 3D games. This means the 2D part of the engine is actually using 3D for all calculations, which is the reason we have to go through the pain of Quaternions for rotation in the first place.