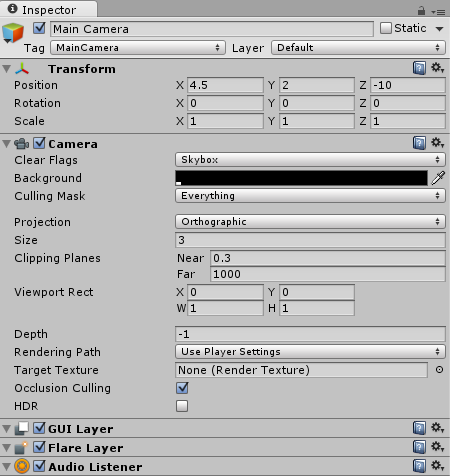
Unity 2D Plants vs. Zombies Tutorial



Today we will make a [Plants vs. Zombies](http://en.wikipedia.org/wiki/Plants_vs._Zombies) clone with Unity's 2D features. To keep things simple, we will ignore all the fancy things like the menu, multiple levels or cut-scenes and focus on fighting Zombies in the backyard.

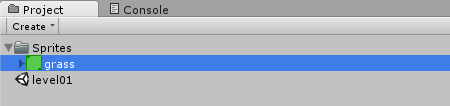
You will learn to create a project, import assets, create animations, create UI elements, and write some simple scripts to tie everything together with less than 130 lines of code!

Project Setup

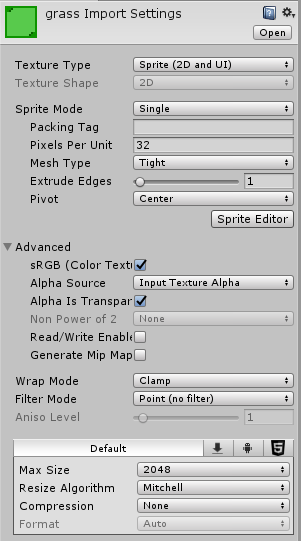
If we select the **Main Camera** in the **Hierarchy** then we can set the **Background Color** to black, adjust the **Size** and the **Position** like shown in the following image:  


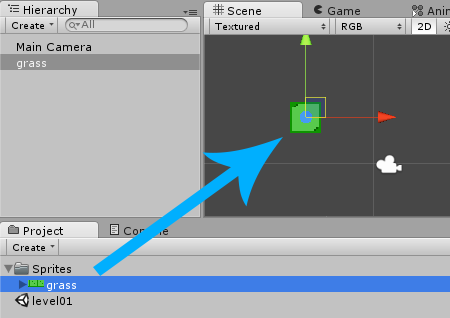
Creating the Grass

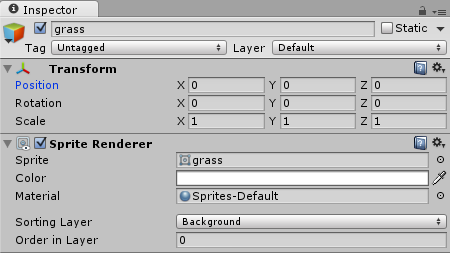
We will use a tool like GIMP or Photoshop to create a grass tile:  
Grass Tiles  
*Note: right click on the image, select****Save As...****, navigate to the project's****Assets****folder and save it in a new****Sprites****folder as “grass.png”.*

Let's select the grass tile in the **Project Area**:  


And then modify the **Import Settings** in the **Inspector**:

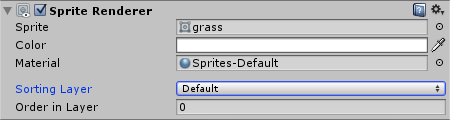
  
*Note: a****Pixels to Unit****value of****32****means that 32 x 32 pixels will fit into one unit in the game world. We will use this value for all our textures.* ***Filter Mode****: Point and* ***Compression****: None means our image will appear exactly as is without being compressed.*

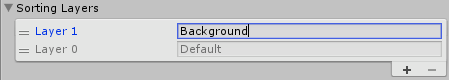
We add the image to the game by dragging it from the **Project Area** into the **Scene**:  


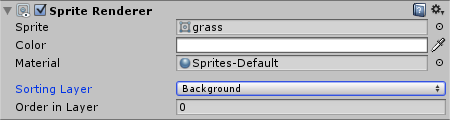
We will position it at **(0, 0)** so it's at the bottom left of our game:  


**Adding a Sorting Layer**

We are making a 2D game, so we can't use the third dimension as depth effect, or to distinguish between background and foreground easily. Instead we will use a **Sorting Layer** to tell Unity which elements it should draw first. For example, our background should be drawn first and the plants should be drawn afterwards *(hence on top of the background)*.

We can change the grass **Sorting Layer** if we take a look at the Sprite Renderer component in the Inspector:  


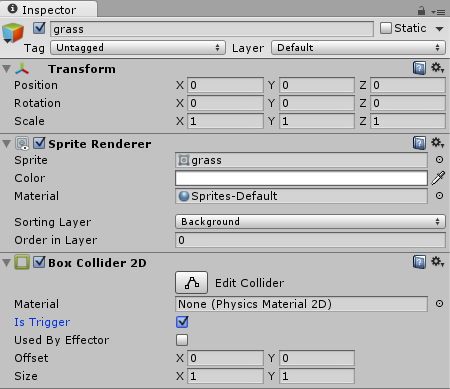
Let's select **Add Sorting Layer..** from the **Sorting Layer** list, add a **Background** layer and move it to the top like shown below:  


Afterwards we select the grass again and assign the previously created **Background** Sorting Layer:  
  
*Note: Unity draws the layers from top to bottom, hence whatever should be in the background will be at the top of the list.*

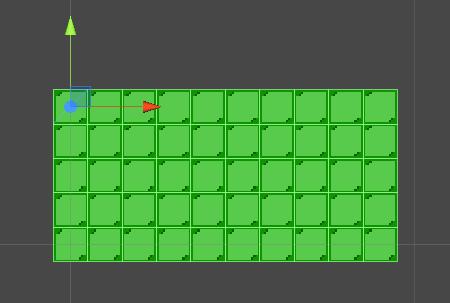
Now the grass will always be drawn behind the plants and the zombies.

**Making the Grass clickable**

Later on when working on the build menu we will need a way to find out if a grass tile was clicked. There are all kinds of different ways to do this in Unity, but the easiest one is to just use the **OnMouseUpAsButton** function that is automatically called by Unity if the grass was clicked.

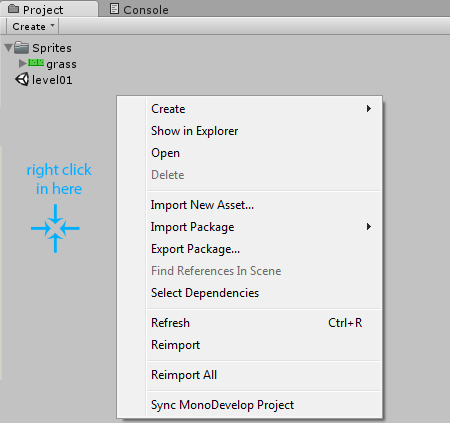
Now there is one thing to keep in mind here: Unity only does this if the GameObject has a Collider. So let's select **Add Component**->**Physics 2D**->**Box Collider 2D** in the **Inspector** and enable the **Is Trigger** option:  
  
*Note: enabling****Is Trigger****means that the grass will receive all kinds of collision information, but things won't actually collide with. So if a Zombie walks into a grass tile, it won't collide with it.*

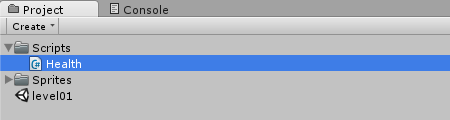
**Duplicating the Grass**

Right now we only have one grass tile in our game. Let's right click it in the **Hierarchy**, select **Duplicate** and then position it at **(1, 0)**. Then we will duplicate it again and position it at **(2, 0)**. We will keep duplicating it until we have **10 \* 5** grass tiles where the bottom-left tile is at **(0, 0)** and the top-right tile is at **(9, 4)**:  
  
*Note: it's important that all the tiles have rounded positions like****(2, 3)****and never****(2.01, 3.023)****.*

The Health Script

The Zombies should be able to attack the Plants, and the Firing Plant should be able to deal damage to the Zombies. We will stick with Unity's component based nature and create only one **Health** script for all entities.

Create a new Script by right clicking in the **Project Area** and then selecting **Create**->**C# Script**:  


We will name it **Health** and then move it into a new **Scripts** folder:  


We can open and then modify the Script by double clicking it:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** Health : MonoBehaviour {  
  
    *// Use this for initialization*  
    **void** Start () {  
      
    }  
      
    *// Update is called once per frame*  
    **void** Update () {  
      
    }  
}

We won't need the **Start** or the **Update** function, so let's remove both of them. Instead we will add a **int** variable that keeps track of the current health and add a function that decreases that variable. If the current health drops below **0** then the entity should be destroyed:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** Health : MonoBehaviour {  
    *// Current Health*  
    [SerializeField]  
    **int** cur = 5;  
  
    **public** **void** doDamage(**int** n) {  
        *// Subtract damage from current health*  
        cur -= n;  
  
        *// Destroy if died*  
        **if** (cur <= 0)  
            Destroy(gameObject);  
    }  
}

*Note: we used****SerializeField****to let Unity know that we want to be able to modify the****cur****variable in the****Inspector****. Usually this is done by making it****public****, but in this case we don't want other Scripts to be able to access the****cur*** *variable. Instead they should always use the****doDamage****function.*

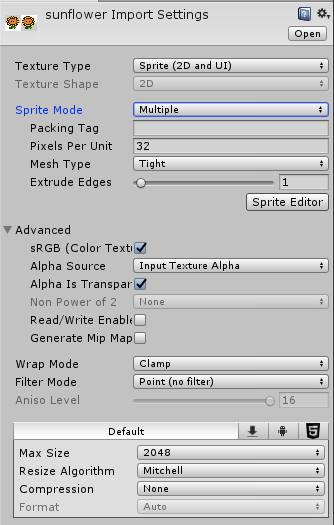
This Script will be added to all Plants and Zombies later on.

Creating a Sunflower Plant

**Drawing the Animation**

Since we are developing a 2D game, animations are very easy to make with our drawing tool of choice and a few hours of time.

For our sunflower we will only need an idle animation where its head slightly moves. Here is what we came up with:  
Pixel Art Sunflower Plant  
*Note: right click on the image, select****Save As...****and save it in the project's****Assets/Sprites****folder as “sunflower.png”.*

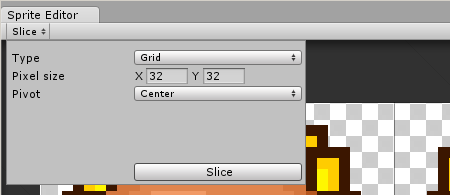
Let's select the sunflower image in the **Project Area** and then modify the **Import Settings** in the Inspector:  


Setting the **Sprite Mode** to **Multiple** tells Unity that there are several sunflower parts *(also known as****Tiles****)* in one image.

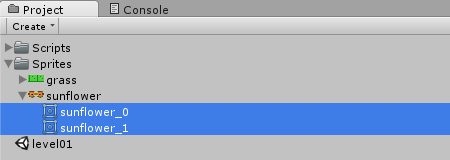
**Importing the Animation**

Now we have to tell Unity where those tiles are in the image. We can open the **Sprite Editor** by pressing the **Sprite Editor** button in the Inspector *(it can be seen in the above image)*.

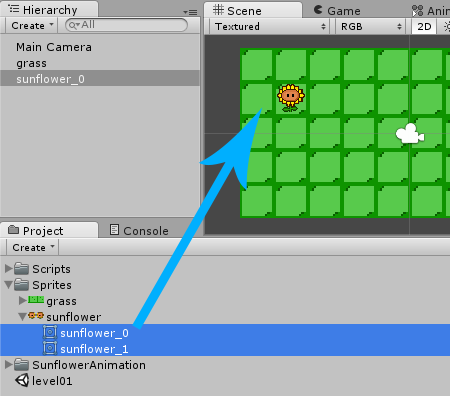
Afterwards we can see our Sunflower in the Sprite Editor:  


All we have to do here is open the **Slice** menu, set the type to **Grid By Cell Size** and the pixel size to **32 x 32**. Afterwards we press the **Slice** button:  


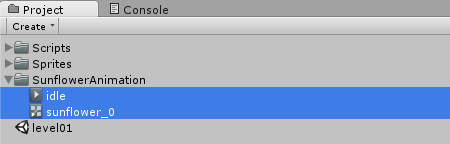
Make sure to press **Apply** and then close the Sprite Editor.

If we take a look in the **Project Area** then we can see that our Sunflower now has 2 children *(the slices)*:  


**Creating the Animation**

Creating a Unity Animation from those 2 slices is incredibly easy. All we have to do is select them in the **Project Area** and then drag them right into the Scene:  


Once we drag it into the scene, Unity asks us where to save the Animation. We can create a new **SunflowerAnimation** folder in our Project Area and then save it as **idle.anim**.

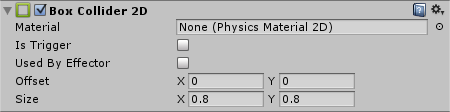
Unity then creates two files for us:  


Press **Play** to see what the default animation looks like:

We can modify the animation speed by double clicking the **sunflower\_0** file in the **SunflowerAnimation** folder, selecting the **idle** state and then changing the **speed** in the **Inspector (**from 1 to 0.1).

**Physics, Tags and Health**

Our plant should be part of the physics world, which means that we have to assign a **Collider** to it. A **Collider** makes sure that things will collide with the plant and that the plant will collide with other things.

Let's select the plant and then press **Add Component**->**Physics 2D**->**Box Collider 2D** in the **Inspector**:  


We will also need to find out if a certain GameObject is a Plant, a Zombie or something completely different. This is what Unity's **Tag** system is for. If we take a look in the **Inspector**, we can see that the current tag is **Untagged**:  
Plants untagged

We can assign the **Plant** tag by selecting **Add Tag...** in the Tag list, then adding the **Plant** tag to the list of available tags, selecting the plant again and then assigning the Tag from the Tag list:  
Plants Tagged

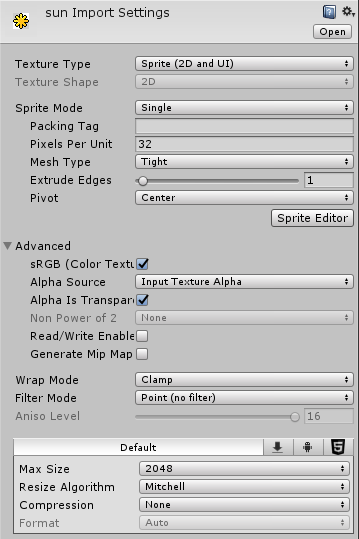
Finally we will select **Add Component**->**Scripts**->**Health** in the Inspector, so the Zombies can deal damage to the plant later on:  
Health Script in Inspector

*Note: our Sunflower was just placed somewhere randomly in the Scene. We will keep it in there for now as long as we are working on the sun spawns in the next step. Proper positioning on a grass tile will be implemented later on.*

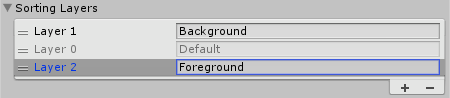
Spawning Suns

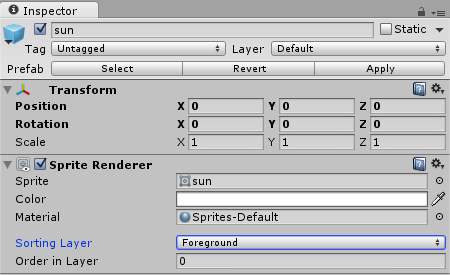
**Drawing the Sun**

The Sunflower plant is supposed to spawn a new sun every few seconds, so let's start by drawing one:  
Pixel Art Sun  
*Note: right click on the image, select****Save As...****and save it in the project's****Assets/Sprites****folder as “sun.png”.*

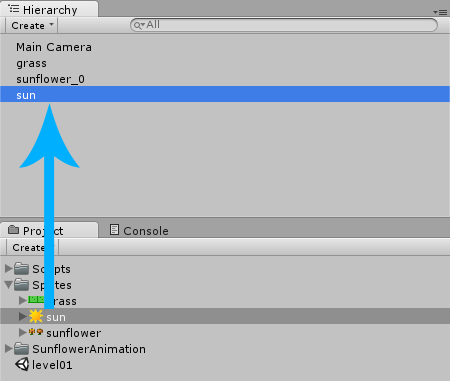
We will use the following **Import Settings**:  


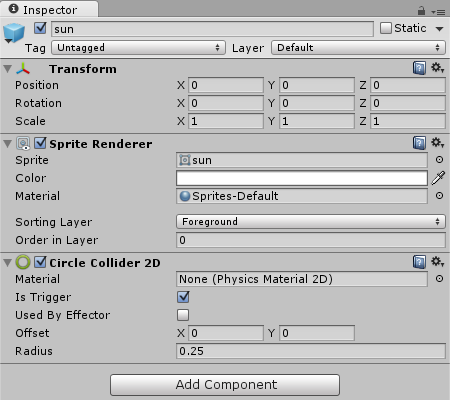
**The Foreground Layer**

We want the sun to be drawn in front of the background and in front of the plants. Let's click on **Add Sorting Layer…** again and create another **Sorting Layer**, name it **Foreground** and move it to the bottom of the list:  


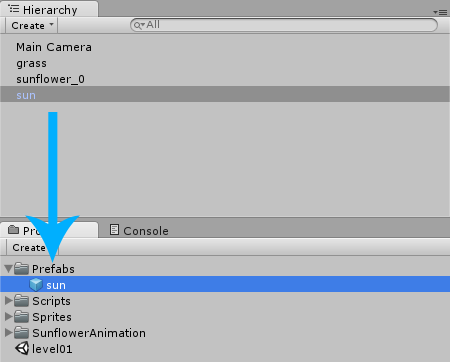
Afterwards we can select the sun again and assign the **Foreground** Sorting Layer to it:  


**Creating the Prefab**

Let's drag the sun into the **Hierarchy** *(or into the Scene)* once in order to create a GameObject from our texture:  


Now we take a look at the **Inspector** where we rename it to **SunPrefab** and press the **Add Component** button where we select **Physics 2D**->**Circle Collider 2D**. A Collider allows us to do some physics stuff with the Sun later on:  


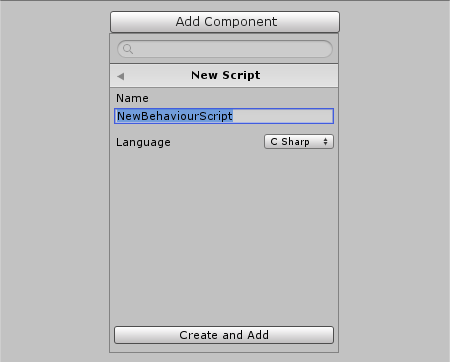
*Note: In the original Plants vs. Zombies game the Sun doesn't really collide with the plants or the zombies. We can achieve this behavior by selecting****Is Trigger****in the Circle Collider 2D. This means that the sun receives collision information, but never really collides with anything. The zombies will be able to walk right through it, instead of colliding with it.*

Afterwards we drag it from the **Hierarchy** into a new **Prefabs** folder in the **Project Area** to create a **Prefab** (an object that can be reused multiple times in our game):  


Now we can delete it from the **Hierarchy**.

*Note: the sun will later be loaded into the Scene by using the****Instantiate*** *function. We can't just use the Sun Texture because we really do need a GameObject with Colliders and everything.*

**Creating the Spawn Script**

We want the sunflower to spawn a new sun every few seconds. This kind of behavior can be implemented with Scripting. Let's select our sunflower and then click on **Add Component**->**New Script**:  


We will name it **SunSpawn**, select **CSharp** as the language and then move it into our Scripts folder in the Project Area. Afterwards we will open it:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** SunSpawn : MonoBehaviour {  
  
    *// Use this for initialization*  
    **void** Start () {  
      
    }  
      
    *// Update is called once per frame*  
    **void** Update () {  
      
    }  
}

We will add a public GameObject variable that allows us to specify the Prefab later on in the Inspector:

**public** **class** SunSpawn : MonoBehaviour {  
    **public** GameObject prefab;  
    ...

The [Instantiate](http://docs.unity3d.com/ScriptReference/Object.Instantiate.html) function allows us to load a Prefab into the Scene. The [InvokeRepeating](http://docs.unity3d.com/ScriptReference/MonoBehaviour.InvokeRepeating.html) function allows us to call a certain function repeatedly, starting at some time in the future. For example, if we would want to spawn something the first time in 1 second and then repeat that every 2 seconds, we would use **InvokeRepeating("Spawn", 1, 2)**. We want the first sun to be spawned in 10 seconds and then keep spawning more suns every 10 seconds, so here is our code:

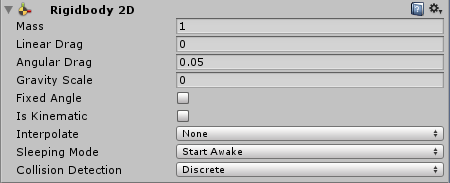
**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** SunSpawn : MonoBehaviour {  
    **public** GameObject prefab;  
  
    *// Use this for initialization*  
    **void** Start() {  
        *// Spawn first Sun in 10 seconds, repeat every 10 seconds*  
        InvokeRepeating("Spawn", 10, 10);  
    }  
      
    **void** Spawn() {  
        *// Load prefab into the Scene*  
        *// -> transform.position means current position*  
        *// -> Quaternion.identity means default rotation*  
        Instantiate(prefab,  
                    transform.position,  
                    Quaternion.identity);  
    }  
}

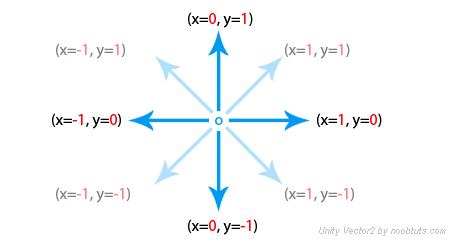
Now we can save the Script and take a look at the **Inspector** again. The Script has a **Prefab** property because our **prefab** variable was public. Let's drag our **SunPrefab** from the Project Area into the **Prefab** slot of the Script:  
Sunflower SunSpawn Script Prefab

If we press **Play** then we can see a new Sun spawn every 10 seconds.

**Sun Movement**

After a sun spawned it should slowly fade away towards the top of the screen. We will use a **Rigidbody2D** for that. A Rigidbody is usually used for everything in the physics world that is supposed to move around.

Let's select the sun in the **Project Area** and then click on **Add Component**->**Physics2D**->**Rigidbody2D** in the Inspector. We will assign the following properties to it, making sure that **Gravity Scale** is set to 0:  


Now all we have to do is give the Rigidbody some velocity *(which is****movement direction \* speed****)*. The following image shows the different vectors needed for certain movement directions:  


We will create another C# Script, name it **DefaultVelocity** and then use it to set the velocity all the time:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** DefaultVelocity : MonoBehaviour {  
    *// The Velocity (can be set from Inspector)*  
    **public** Vector2 velocity;  
  
    **void** FixedUpdate () {  
        GetComponent<Rigidbody2D>().velocity = velocity;  
    }  
}

*Note: we set it in each****FixedUpdate****call so that whatever has this script attached to it will try to keep moving no matter what happens. This is useful for the Zombies that might not be able to keep moving when running into a plant, but our Script will make sure that they start moving again as soon as the plant was destroyed.*

Afterwards we select the sun prefab again and then add the Script to it by clicking on **Add Component**->**Scripts**->**Default Velocity**. We will also assign a velocity that goes **upwards** *(into the****y****direction)*:  
Sun DefaultVelocity Script

If we press **Play** then we can now see the sun spawn and move upwards afterwards:

**Collecting Sun**

The last thing we have to do when it comes to the sun is to allow the player to collect it. We will need one global score that keeps track of how many suns were collected, and we will need to increase that score whenever the player clicks on a sun.

Unity comes with a [OnMouseDown](http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnMouseDown.html) function that allows us to find out if our GameObject was clicked. So let's create a new **C# Script**, name it **SunCollect** and then use the **OnMouseDown** function:

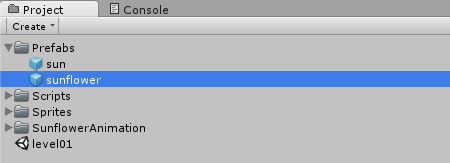
**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** SunCollect : MonoBehaviour {  
    *// Global score*  
    **public** **static** **int** score = 100;  
  
    **void** OnMouseDown() {  
        *// Increase Score*  
        score += 20;  
  
        *// Destroy Sun*  
        Destroy(gameObject);  
    }  
}

*Note: we have to use****static****in order to make the score variable global. This means that it can be accessed by other scripts with****SunCollect.score****any time. The initial score value is 100 so the player has some sun to start building plants with.*

Once we added the Script to the sun Prefab we can press **Play**, wait for a sun to spawn and then

collect it by clicking on it.

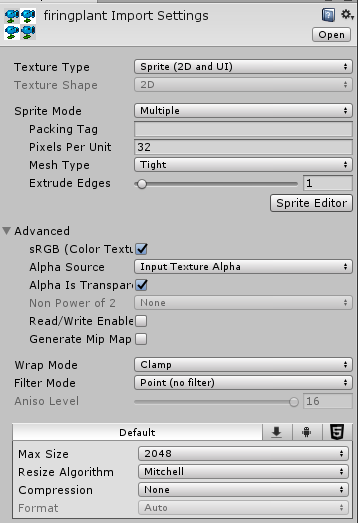
**Cleaning Up the Hierarchy**

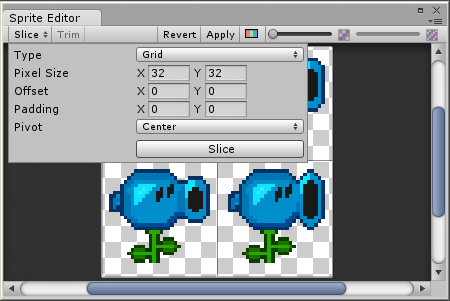
Right now the Sunflower is still in the Hierarchy, but we don't want it to be in there from the beginning. The player is supposed to build it manually later on. So let's create a Prefab from it by dragging it into the **Project Area**:  


The Firing Plant

**The Image**

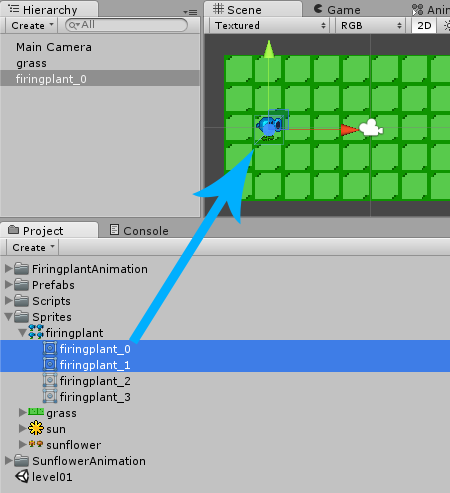
Alright let's create one more plant, the type that is able to shoot at the Zombies. As usual we will begin by drawing all the animations. The two slices at the top are the idle animation, the two slices at the bottom are the firing animation:  
  
*Note: right click on the image, select****Save As...****and save it in the project's****Assets/Sprites****folder as “firingplant.png”.*

We will use the following **Import Settings** for our firing plant:  


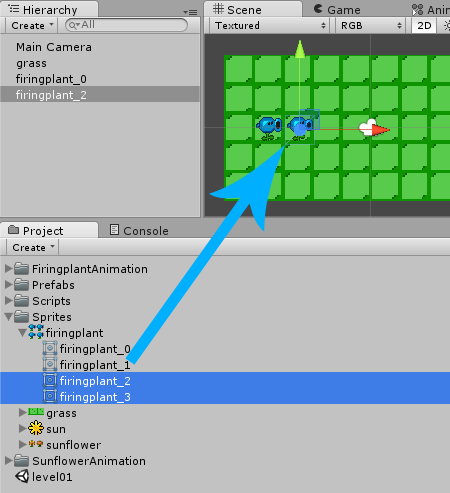
Let's click on the **Slice Editor** button and then slice it with the following settings:  


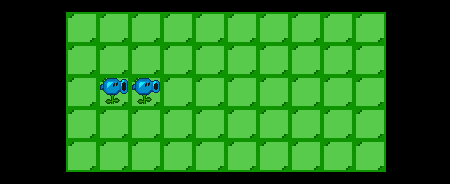
**Creating the Animations**

This time we have two animations in our image. The first two slices are the idle animation and the last two slices are the firing animation.

The idle animation can be created by selecting the first two slices in the **Project Area** and dragging them into the scene:  


When Unity asks us where to save it, we name it **idle.anim** and save it in a new folder like **FiringplantAnimation** inside our Project Area.

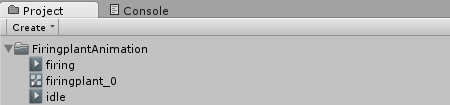
It works the same way for the firing animation *(the last two slices)*:  


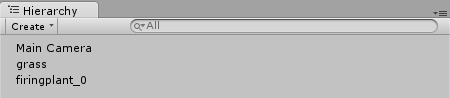
If we press **Play** then we can see our two animations in the game:  


*Note: they are still two different GameObjects, but we will take care of that soon.*

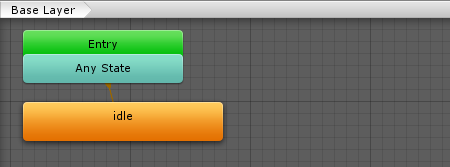
Just like the sunflower idle animation, let’s change the speed of the animation from 1 to 0.1 in the Animator.

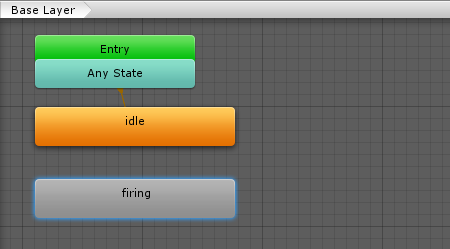
**Cleaning up after Unity**

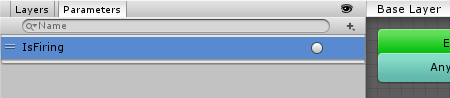
Every time we drag slices into the scene, Unity creates an animation and a controller in our Project Area, and adds an animated GameObject to the Hierarchy. Right now we have 2 controllers and 2 GameObjects, but we only need one of each so let's delete the **firingplant\_1** controller from the Project Area:  


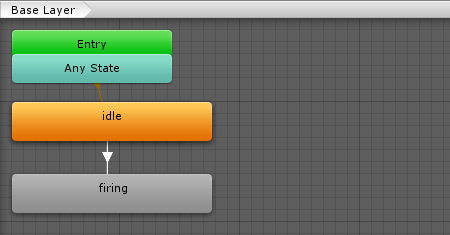
And then we delete the **firingplant\_1** GameObject from the Hierarchy:  


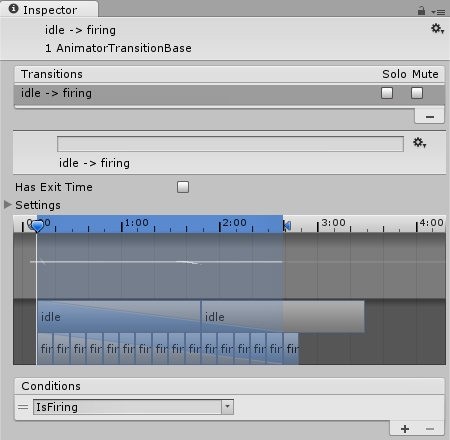
**Modifying the Controller**

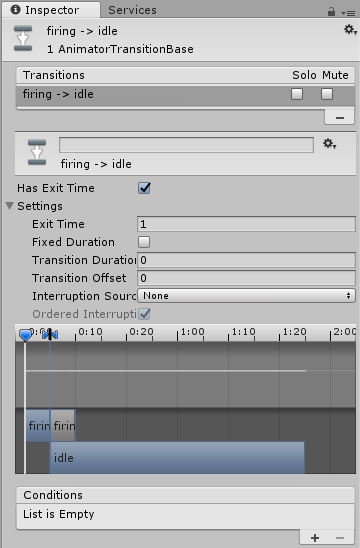
There is a trick to get both animations into one GameObject, it's called a **animation state machine**. We can see our Firing Plant's current state machine by double clicking the controller **firingplant\_0** in the Project Area:  


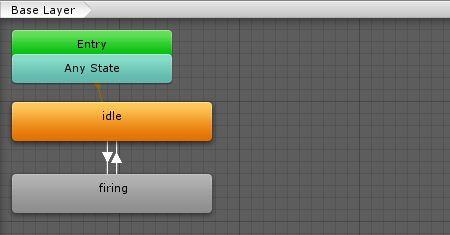
Right now all we see is the **idle** state. This is why we only see the **firing** animation when we press **Play**. If we want it to play the **firing** animation at some point, then we will have to create a state for it, too. We can do this by simply dragging the **firing** animation from the **Project Area** into the **Animator**:  


We need some kind of Parameter that helps Unity to decide when to go to the firing state. We will select the **Parameters** tab and then click the **+** on the top right, add a Parameter of type **Trigger** and name it **IsFiring**:  
  
*Note: A Trigger is a Paramater that just fires once. We will set this parameter from within a Script later on.*

So far so good. Now we just have to tell Unity to go to the **firing** state when the **IsFiring** Parameter is triggered. Let's right click the **idle** state, select **Make Transition** and then drag the white Arrow onto the **firing** state:  


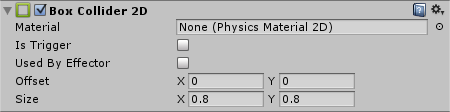
Afterwards we select the new **Transition** *(the white arrow)* and take a look in the Inspector. We will disable the **Has Exit Time** property and give it the Condition **IsFiring**:  
  
*Note: we disabled****Has Exit Time****because the Transition shouldn't happen after a certain time automatically.*

Now we add one more transition from **firing** to **idle**. This time we will keep the **Has Exit Time** enabled because we want the animation state machine to switch back from **firing** to **idle** after one second. We will also set the **Exit Time** to **1** and **Transition Duration** to **0**:  
 

And here is our final state machine:  


*Note: the animator will automatically switch between the****idle****and****firing*** *animations now. It will play them and loop them fully automatically, all we have to do is set the****IsFiring****parameter from a Script.*

**Physics and Tags**

We will also add a **Box Collider 2D** to the plant:  


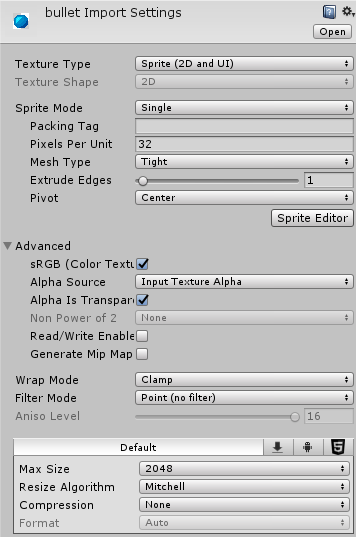
And assign the **Plant** tag to it so we can find out what type it is later on:  
Plants Tagged

Finally we will select **Add Component**->**Scripts**->**Health** in the Inspector, so that the Zombies can deal damage to the plant later on:  
Health Script in Inspector

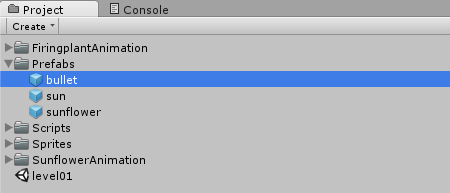
Giving the Plant something to Shoot

**Drawing the Bullet**

We will use the a 16 x 16 px texture for a bullet:  
Plants Bullet  
*Right click the image, select****Save As...****and save it in the project's****Assets/Sprites****folder as “bullet.png”.*

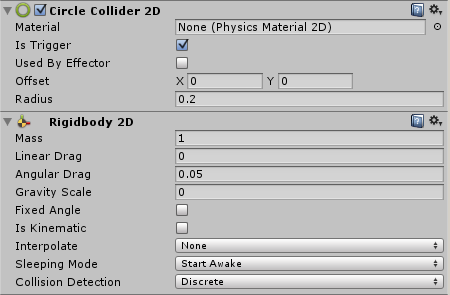
With the following **Import Settings**:  


**Creating the Bullet Prefab**

Now we have to create a Prefab from it again. Therefore we will drag the bullet texture into the **Scene** once *(which creates a GameObject)* and then drag it back into the **Project Area**, which creates a Prefab:  


Afterwards we can delete it from the Hierarchy again.

**Adding Physics to the Bullet**

Our bullet is supposed to fly towards the right side of the screen and collide with Zombies if necessary. So let's select it in the **Project Area** and then click on **Add Component**->**Physics 2D** where we add a **Circle Collider 2D** and a **Rigidbody 2D** with the following properties:  
  
*Note: we enabled****Is Trigger****so the bullet will fly through plants instead of colliding with them.*

**Letting it fly**

The bullet should fly towards the right as soon as it was Instantiated. We already have a Script that makes something move into a direction by using the Rigidbody's **velocity**. So let's select **Add Component**->**Scripts**->**DefaultVelocity** and assign a velocity that makes it fly towards the right:  
Bullet with Default Velocity Script  
*Note: this is the beauty of component based development. We can write a Script once and then re-use it for all kinds of entities in our game.*

**Dealing Damage on Collision**

To check for collisions we will make use of Unity's [OnTriggerEnter2D](http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnTriggerEnter2D.html) function that will be called as soon as the bullet flies into another GameObject, in which case we will check if it was a Zombie *(by comparing the tag)* and then decrease its health.

Let's create a new **C# Script**, name it **BulletDamage** and then open it:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** BulletDamage : MonoBehaviour {  
  
    *// Use this for initialization*  
    **void** Start () {  
      
    }  
      
    *// Update is called once per frame*  
    **void** Update () {  
      
    }  
}

We won't need the **Start** or the **Update** function. Instead we will use the **OnTriggerEnter2D** function as explained above. We don't really know what the Zombie will look like yet, but we do know that we will add "Zombie" **Tag** and a **Health** component to it.

So let's just work with that:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** BulletDamage : MonoBehaviour {  
  
    **void** OnTriggerEnter2D(Collider2D co) {  
        *// Zombie?*  
        **if** (co.tag == "Zombie") {  
            *// Deal Damage, destroy Bullet*  
            co.GetComponent<Health>().doDamage(1);  
            Destroy(gameObject);  
        }  
    }  
}

*Note: we used****GetComponent****to find the****Health****component on the Zombie. We then deal damage by calling the****doDamage****function. Afterwards we remove the bullet from the game by using****Destroy****.*

We can add the Script to the bullet Prefab by selecting **Add Component**->**Scripts**->**Bullet Damage** in the Inspector:  
Bullet Damage Script in Inspector

Shooting the Bullets

Now that we have a bullet, we can make our Firing Plant shoot it whenever it sees a Zombie. Let's select **Add Component**->**New Script** in the **Inspector**. We will name it **Firing**, select **CSharp** as the language, move it into our Scripts folder and then open it:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** Firing : MonoBehaviour {  
  
    *// Use this for initialization*  
    **void** Start () {  
      
    }  
      
    *// Update is called once per frame*  
    **void** Update () {  
      
    }  
}

First of all we will have to find out if there are any Zombies at the right of the plant. We can do this by using the [RaycastAll](http://docs.unity3d.com/ScriptReference/Physics2D.RaycastAll.html) function. A Raycast shoots a ray through the world and finds out if it hit anything. So if we shoot a ray from the very right of the screen towards the Plant and if it collides with something that has the tag "Zombie" then the plant should be firing.

Here is our function for it:

**bool** zombieInFront() {  
    *// Raycast from the right of the game to the plant*  
    Vector2 origin = new Vector2(9.5f, transform.position.y);  
    RaycastHit2D[] hits = Physics2D.RaycastAll(origin, -Vector2.right);  
  
    *// Find out if any Zombie was hit*  
    **foreach** (RaycastHit2D hit **in** hits) {  
        **if** (hit.collider != **null** &&  
            hit.collider.gameObject.tag == "Zombie")  
            **return** **true**;  
    }  
    **return** **false**;  
}

*Note: the****RaycastAll****function simply returns****all****the hits instead of just the first one like the****Raycast****function would do. We could also shoot a Ray from the Plant towards the right of the screen. But this way it makes more sense because we won't run into trouble if we add a Collider to the plant, in which case the Ray would just hit the plant itself. We used****-Vector2.right****for the direction because it's the left direction. We used the vector****(9.5f, pos.y)****as the Raycast's position because this is the right border of the grass at the height of the plant.*

We will also need a public Prefab variable again, which will be set to the **bullet Prefab** in the Inspector later:

**public** **class** Firing : MonoBehaviour {  
    *// The Bullet Prefab*  
    **public** GameObject bulletPrefab;      
    ...  
}

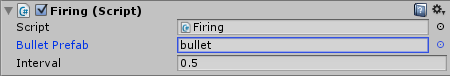
Now we can write our **Shoot** function that uses **Instantiate** to spawn the bullet and sets the **IsFiring** parameter in our animation state machine whenever it sees a zombie:

**void** Shoot() {  
    **if** (zombieInFront()) {  
        *// Animation*  
        GetComponent<Animator>().SetTrigger("IsFiring");  
  
        *// Instantiate Bullet*  
        Instantiate(bulletPrefab, transform.position, Quaternion.identity);  
    }  
}

Here is how it looks if we put it all together:

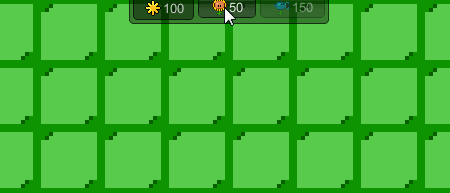
**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** Firing : MonoBehaviour {  
    *// The Bullet Prefab*  
    **public** GameObject bulletPrefab;  
      
    *// Shooting Interval*  
    **public** **float** interval = 0.5f;  
  
    *// Use this for initialization*  
    **void** Start () {  
        *// Try to shoot every few seconds*  
        InvokeRepeating("Shoot", 0, interval);  
    }  
  
    **bool** zombieInFront() {  
        *// Raycast from the right of the game to the plant*  
        Vector2 origin = new Vector2(9.5f, transform.position.y);  
        RaycastHit2D[] hits = Physics2D.RaycastAll(origin, -Vector2.right);  
  
        *// Find out if any Zombie was hit*  
        **foreach** (RaycastHit2D hit **in** hits) {  
            **if** (hit.collider != **null** &&  
                hit.collider.gameObject.tag == "Zombie")  
                **return** **true**;  
        }  
        **return** **false**;  
    }  
      
    **void** Shoot() {  
        **if** (zombieInFront()) {  
            *// Animation*  
            GetComponent<Animator>().SetTrigger("IsFiring");  
  
            *// Instantiate Bullet*  
            Instantiate(bulletPrefab, transform.position, Quaternion.identity);  
        }  
    }  
}

*Note: the Shoot function is called every few seconds thanks to****InvokeRepeating****.*

If we save the Script and take a look at the **Inspector** then we can drag the bullet Prefab from the Project Area into the **Prefab** slot:  
  
*Note: we will be able to try out the firing mechanics as soon as we added Zombies.*

We don't have a **firingplant Prefab** yet, so let's select the Firing Plant in our **Hierarchy**, drag it into our **Prefabs** folder and then delete it from the Hierarchy.

The Build Menu

Here is a preview of our build menu:  


We will now implement every single feature that is needed to create the menu.

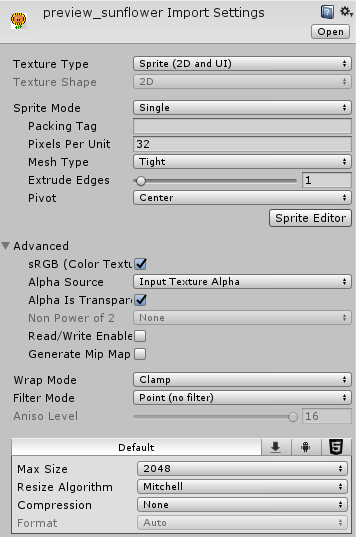
**The Build Info Component**

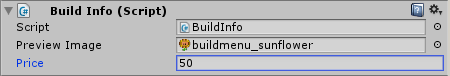
Our Build Menu will need the price and a preview image for each plant. Now that kind of data has to be stored somewhere. The most elegant way to do this is to store it in each plant prefab.

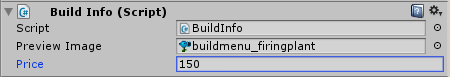
Let's select our two plant prefabs and then click on **Add Component**->**New Script**, name it **BuildInfo**, select **CSharp** as the language, move it into our Scripts folder and then open it. We will give it the previously mentioned properties:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** BuildInfo : MonoBehaviour {  
    **public** Texture previewImage;  
    **public** **int** price;  
}

We will use the following preview images:  
Buildmenu Sunflower Buildmenu Firingplant  
*Note: right click each image, select****Save As...****and save them in the project's****Assets/Sprites****folder as “preview\_sunflower.png” and “preview\_firingplant.png”.*

Here are the **Import Settings** for both images:  
 

Afterwards we will select our plant prefabs one after another and then assign the following settings to the **Build Info** Scripts:  
  
*The Sunflower*

  
*The Firing Plant*

**Drawing the BuildMenu**

In order to show the menu ingame, we will use the [OnGUI](http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnGUI.html) function to:

* draw the sun and the sun score
* for each plant:
  + draw the plant image and the sun price

Our GUI code will do the following:

* Begin a **GUILayout** at the top center
* Begin a **Horizontal Area** *(with Box style)*
  + Draw the **Sun** Image and the **CollectSun.score**
  + For each Plant:
    - Draw a Button with the **previewImage** and the **Price**

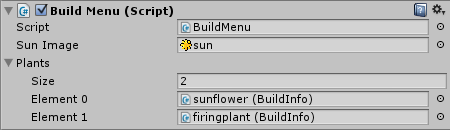
One thing to remember is that by default a [GUILayout.Button](http://docs.unity3d.com/ScriptReference/GUILayout.Button.html) can only have an image or a text, never both. But since we want it to show the previewImage and the price, we will have to use an additional [GUIContent](http://docs.unity3d.com/ScriptReference/GUIContent.html), which is capable of showing both.

We will begin by selecting our **Main Camera** and then clicking on **Add Component**->**New Script** in the **Inspector**. We will name it **BuildMenu**, select **CSharp** as the language, move it into our Scripts folder and then open it.  
*Note: We added our script to the Main Camera because it will always be there.*

Here is our **BuildMenu** script:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** BuildMenu : MonoBehaviour {  
    *// Sun Image*  
    **public** Texture sunImage;  
  
    *// Plant Prefabs*  
    **public** BuildInfo[] plants;  
  
    **void** OnGUI() {  
        *// Begin Gui*  
        GUILayout.BeginArea(new Rect(Screen.width/2 - 100, -7, 200, 100));  
        GUILayout.BeginHorizontal("box");  
  
        *// Draw the Sun*  
        GUILayout.Box(new GUIContent(SunCollect.score.ToString(), sunImage));  
  
        *// Draw each Plant's BuildInfo*  
        **foreach** (BuildInfo bi **in** plants) {  
            GUILayout.Button(new GUIContent(bi.price.ToString(), bi.previewImage));  
        }  
  
        *// End Gui*  
        GUILayout.EndHorizontal();  
        GUILayout.EndArea();  
    }  
}

*Note: the plant Prefabs are an array (hence****[ ]****). It means that it's not just one but a whole bunch of them.*

After saving the Script we will drag our sun image and the plant Prefabs from our **Project Area** into the Script's slots:  


If we press **Play** then we can already see our Build Menu:  


**Enabling and Disabling the Buttons**

Now there are a few more things to implement here. First of all, each button should only be enabled if the player collected enough sun to pay for it. This can be done very easily by using **GUI.enabled**:

*// Draw each Plant's BuildInfo*  
**foreach** (BuildInfo bi **in** plants) {  
    GUI.enabled = SunCollect.score >= bi.price;  
    GUILayout.Button(new GUIContent(bi.price.ToString(), bi.previewImage));

GUI.enabled = true;  
}

*Note: this assigns a boolean value (true or false) to****GUI.enabled*** *before the button is created. The boolean value is calculated by testing if the****SunCollect.score****is bigger than the plant's price. Afterwards, we set* ***GUI.enabled*** *to true again to prevent unintentionally disabling any other UI elements.*

If we press Play then we can see our Buttons being disabled unless we have enough sun:  


**Building the Plant**

Let's take care of the most important part: building a plant. The player should first click the plant button in the build menu and then click on a grass tile. Afterwards the plant should be built on that grass tile.

So first of all we will need to keep track of the plant that the player wants to build:

*// Currently building...*  
**public** **static** BuildInfo cur;

*Note: it's****public static****because we want to be able to access it from another Script later on.*

And as soon as the player clicks on a button we will set the **cur** variable to the plant that the player wants to build:

*// Draw each Plant's BuildInfo*  
**foreach** (BuildInfo bi **in** plants) {  
    GUI.enabled = SunCollect.score >= bi.price;  
    **if** (GUILayout.Button(new GUIContent(bi.price.ToString(),

bi.previewImage))){  
        cur = bi;

}

GUI.enabled = true;  
}

**The Grass Tile Script**

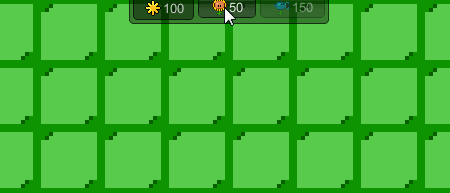
Now we will have to wait for the player to click on a grass tile. And if he clicked on one then we want to build the **BuildInfo.cur** plant. Let's select all the grass tiles in the Scene and then click on **Add Component**->**New Script** in the **Inspector**. We will name it **Grass**, select **CSharp** as the language, move it into our Scripts folder again and then open it. We will use the **OnMouseUpAsButton** function to find out if a grass tile was clicked:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** Grass : MonoBehaviour {  
  
    **void** OnMouseUpAsButton() {  
        *// Build Stuff*  
    }  
}

Now we will find out if there is something to build, and if so we will build it and make the player pay the price:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** Grass : MonoBehaviour {  
  
    **void** OnMouseUpAsButton() {  
        *// Is there something to build?*  
        **if** (BuildMenu.cur != **null**) {  
            *// Build it*  
            Instantiate(BuildMenu.cur.gameObject, transform.position, Quaternion.identity);  
            SunCollect.score -= BuildMenu.cur.price;  
            BuildMenu.cur = **null**;  
        }  
    }  
}

*Note: we can access the BuildMenu's****cur****variable from anywhere by using****BuildMenu.cur****. This works because it's****public static****. We load the plant into the game by using****Instantiate****at the grass tile's position (transform.position) with the default rotation (Quaternion.identity). Afterwards we clear the****cur*** *variable because we are done building.*

If we press **Play** then we can now build some plants:  


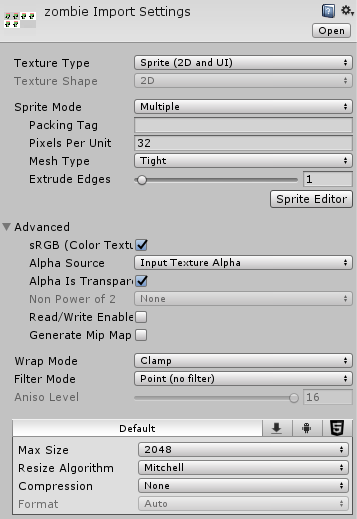
What we saw here was the full power of the Unity engine. With only a few lines of code we were able to solve the relatively hard problem of creating a build menu that works for arbitrary amounts of plants.

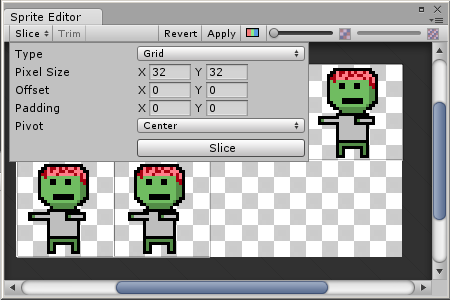
Zombies

**Drawing the Zombie**

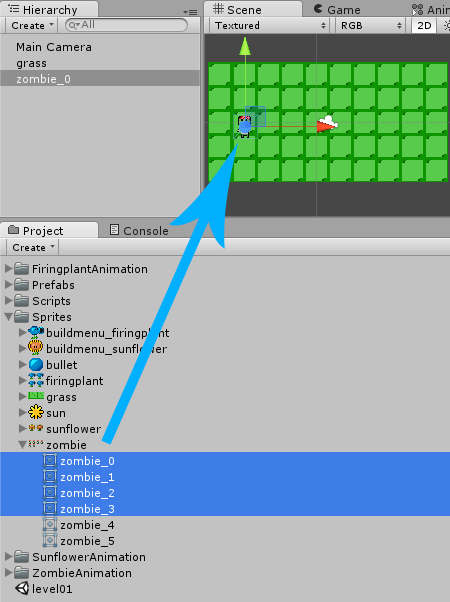
Now the part that we have all been waiting for: **Z o m b i e s**!

As usual we will begin by drawing one image that contains all the animation frames. The first row contains the walking animation, the second row contains the attack animation:  
  
*Note: right click on the image, select****Save As...****and save it in the project's****Assets/Sprites****folder as “zombie.png”.*

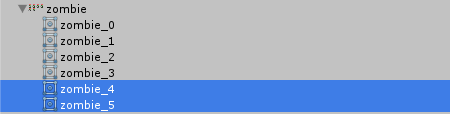
We will use the following **Import Settings**:  


The image will be sliced as a **32 x 32** Grid:  


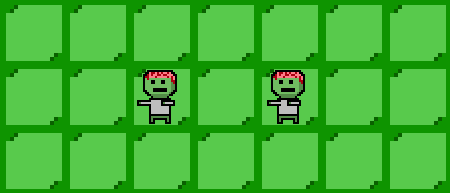
**Extracting the Zombie Animations**

Create the **moving** animation by selecting the first 4 slices and dragging them into the scene:  


We will save it as **moving.anim** in a new **ZombieAnimation** folder.

The next two slices will be the **attacking** animation:  


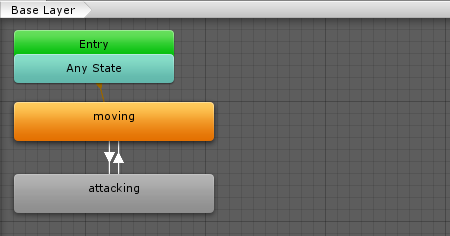
Which we will save as **attacking.anim** in the **ZombieAnimation** folder.

If we press **Play** then we can see our two animations:  


**Cleaning up after Unity**

As usual we will have to clean up after Unity by deleting the **zombie\_4** GameObject from the **Hierarchy** and the **zombie\_4** Controller from the **ZombieAnimation** folder.

**The Zombie Animation State Machine**

We already know our away around the **Animator** now, so this time we are able to create the state machine without much worrying:  


The Parameter is:

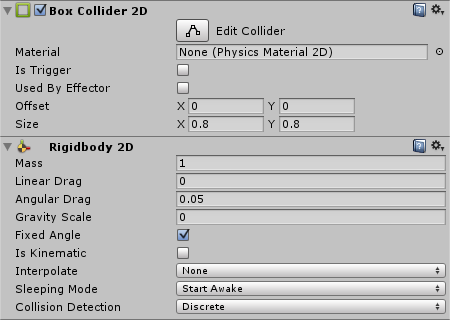
* **IsAttacking** of type **Trigger** *(only fires once)*

The Transitions are:

* moving to attacking: **= IsAttacking**
* attacking to moving: **= Exit Time 1.0**

Which means that as soon as we say **IsAttacking**, the animation state machine will play the **attacking** animation for one second. Afterwards it will go back to **moving** automatically.

**Zombie Physics**

Our Zombie should be part of the physics world, which means that it will need a **BoxCollider2D**. It should also move around, which means that it needs a **Rigidbody2D** too:  


**The Zombie Movement**

The Zombie should walk towards the left at all times. Of course, we can just use our **DefaultVelocity** Script once more. We can add it to the Zombie by selecting **Add Component**->**Scripts**->**Default Velocity** and then assigning a velocity that makes it slowly walk to the left:  
Zombie with DefaultVelocity Script

**The Zombie Attack Script**

As soon as the Zombie collides with a plant, it should start attacking it once a second. If the Zombie didn't attack anything for more than a second.

Let's select **Add Component**->**New Script** in the **Inspector**. We will name it **ZombieAttacking**, select **CSharp** as the language, move it into our Scripts folder and then open it:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** ZombieAttacking : MonoBehaviour {  
  
    *// Use this for initialization*  
    **void** Start () {  
      
    }  
      
    *// Update is called once per frame*  
    **void** Update () {  
      
    }  
}

We won't need the **Start** or the **Update** function. Instead we can use Unity's [OnCollisionStay2D](http://docs.unity3d.com/ScriptReference/Collider2D.OnCollisionStay2D.html) function which will keep notifying us as long as the Zombie collides with the plant:

**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** ZombieAttacking : MonoBehaviour {  
  
    **void** OnCollisionStay2D(Collision2D coll) {  
        *// Collided with a Plant?*  
        **if** (coll.gameObject.tag == "Plant") {  
            *// Do Stuff...*  
        }  
    }  
}

The Zombie should play the attack animation when standing in front of a plant:

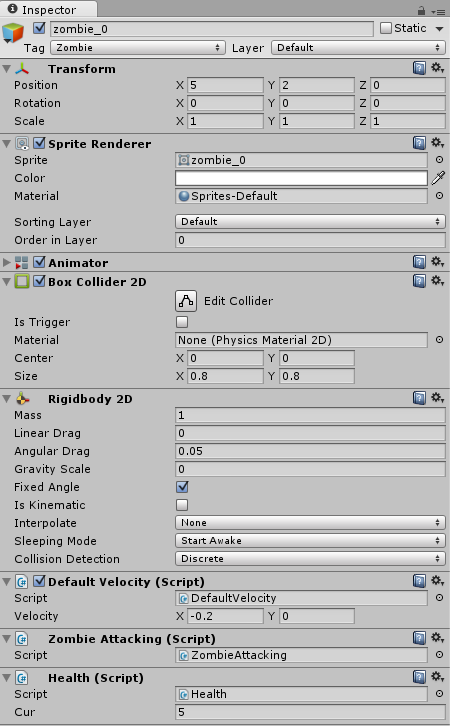
**void** OnCollisionStay2D(Collision2D coll) {  
    *// Collided with a Plant?*  
    **if** (coll.gameObject.tag == "Plant") {  
        *// Play Attack Animation*  
        GetComponent<Animator>().SetTrigger("IsAttacking");  
    }  
}

It should also deal damage to the plant, but only once a second:

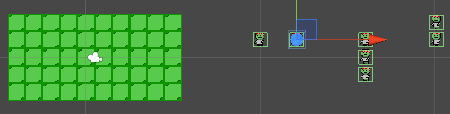
**using** UnityEngine;  
**using** System.Collections;  
  
**public** **class** ZombieAttacking : MonoBehaviour {  
    *// Last Attack Time*  
    **float** last = 0;  
  
    **void** OnCollisionStay2D(Collision2D coll) {  
        *// Collided with a Plant?*  
        **if** (coll.gameObject.tag == "Plant") {  
            *// Play Attack Animation*  
            GetComponent<Animator>().SetTrigger("IsAttacking");  
            *// Deal damage once a second*  
            **if** (Time.time - last >= 1) {  
                coll.gameObject.GetComponent<Health>().doDamage(1);  
                last = Time.time;  
            }  
        }  
    }  
}

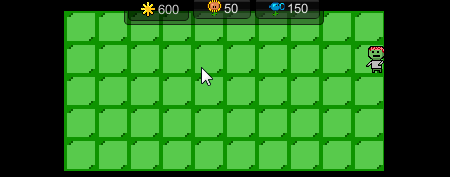
And that's all there is to it. We simply used **Time.time** to find out how much time was elapsed since **last** time, then get access to the plant's **Health** Script, call the **doDamage** function and reset the **last** time.

Remember how we checked for the **Zombie** tag in the Firing and Bullet Scripts? Let's add the **Zombie** tag to our Zombie to make sure that it can be shot by the bullet:  
Zombie Tag

We will also add a **Health** Script to it:  


If we press **Play** and build a Sunflower then we can see how the Zombie will try to attack it as soon as it reaches it. If we build a Firing Plant then we can see how it kills the Zombie after a few shots!

We will also duplicate the Zombie by right clicking it in the **Hierarchy** and then selecting **Duplicate**. We will repeat this a few times and them position the zombies on the right of the level so that they reach it only after a few seconds of walking:  


If we press **Play** then we can see the game in action:  


You could alternatively add spawn points and spawn zombies from them similarly to how we spawn the suns, but that is an additional you can try on your own!

Summary

Congratulations!!! You just created a clean and simple Plants vs. Zombies Clone in Unity.

Hopefully now you have a good idea of how to create a project, import assets, create animations, create UI elements, and write scripts. There are obviously many improvements and additions to be made to the game still. I encourage you to try and add some things on your own in the future!

The best way to learn Unity is to just keep practicing.