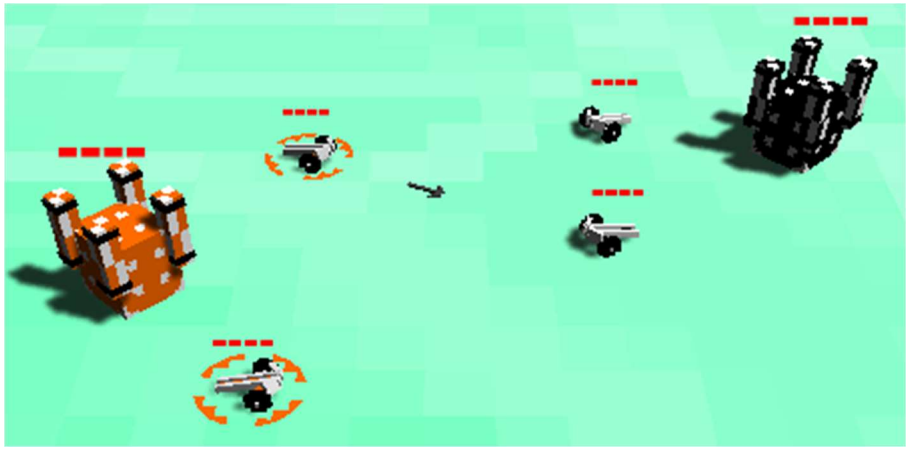
Unity Real Time Strategy (RTS) Game



Intro

In this Workshop, we will focus on RTS *(**R**eal **T**ime **S**trategy)* genre fundamentals. We will cover:
Part 1:

- Creating 3D objects in unity
- Adding Textures
- GUI
- Basic Scripting
    - Animations
    - Raycasts
    - Spawning Units

Part 1:

- Tagging
- More UI
- Advanced Scripting
    - Unit Selection
    - Unit Movement
    - Enemy AI & Navigation

Project & Scene Setup

Creating the Project

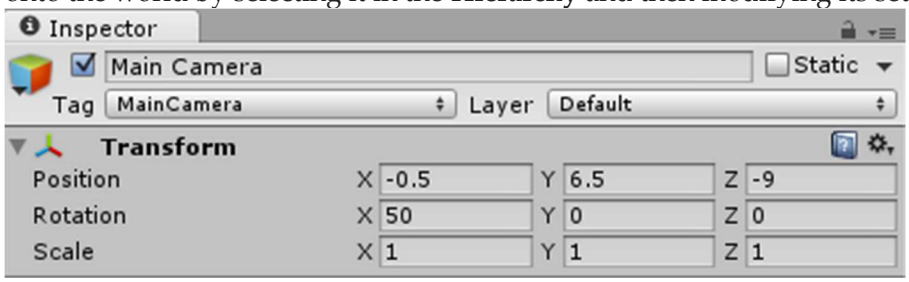
Start Unity and select **Open**:

Navigate to the location where you saved the Workshop-2 folder, inside the folder select the RTS-Game folder.

Note: To see the finished game, open a new Unity project and open the RTS-Game-FINISHED folder.

Camera

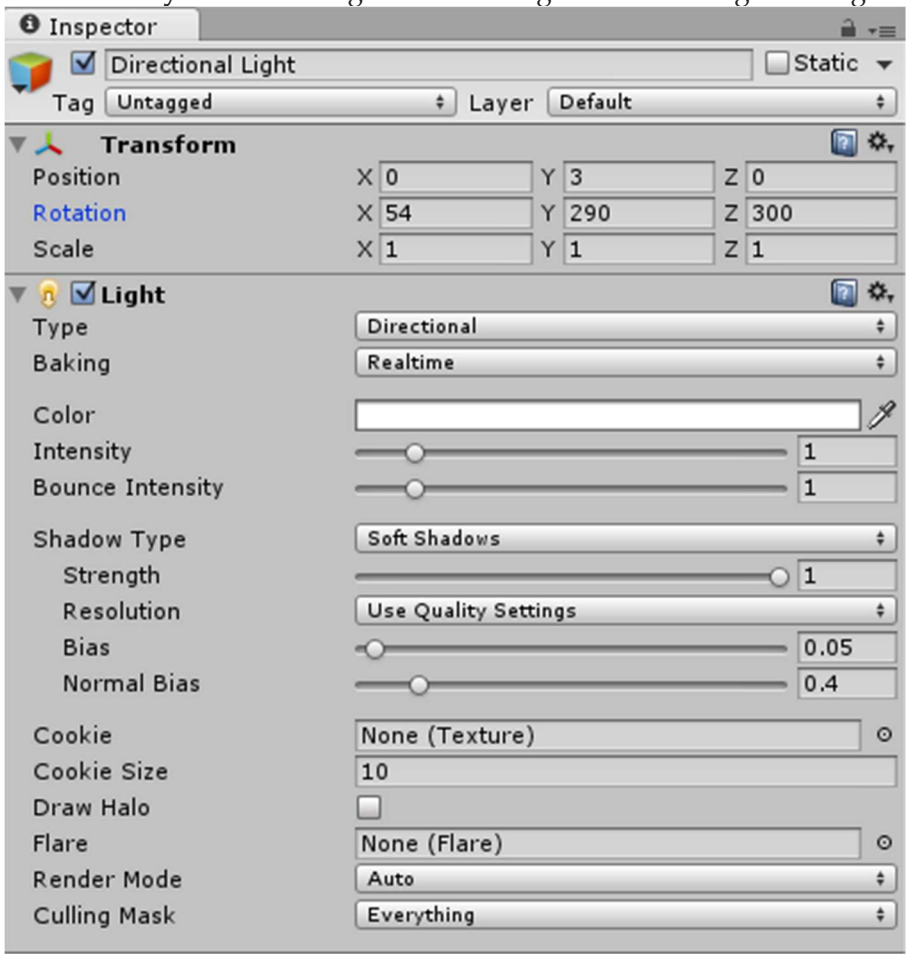
Unity automatically adds a Camera to the project. Let's position it in a way that makes it look down onto the world by selecting it in the **Hierarchy** and then modifying its settings in the **Inspector**:



Note: we don't see a difference yet because our scene is still empty.

Light

Unity already added a **Directional Light** to the scene when we created it. Let's select it in the **Hierarchy** and then assign the following rotation and light settings to it:



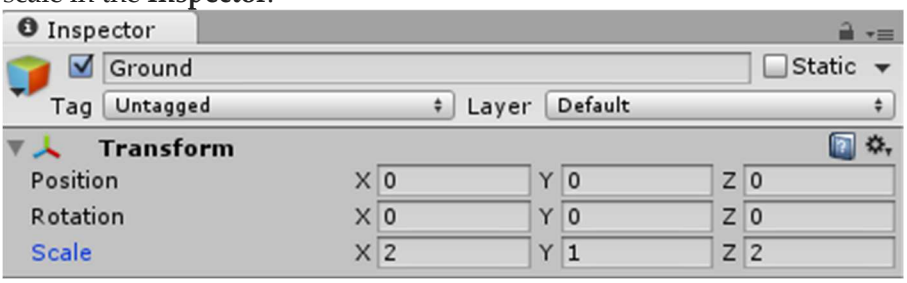
Note: we can use any settings that we want, but the ones above will look good later on.

Filling the World

Now that everything is set up, we can fill the scene with all kinds of cool things!

The Ground

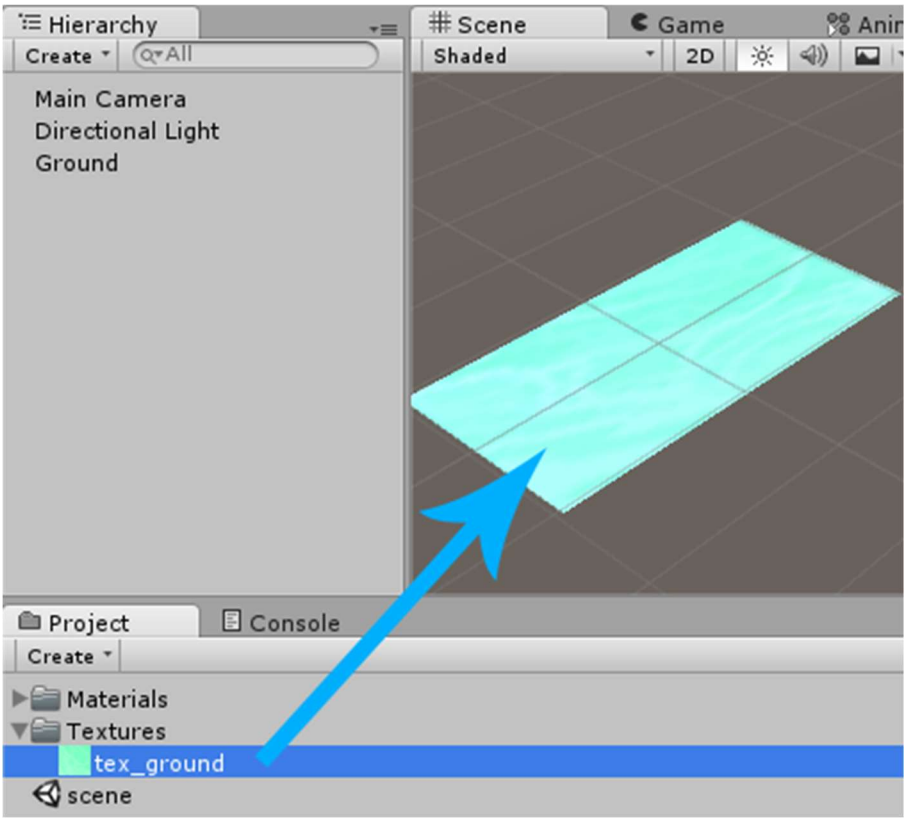
The ground will just be a simple Unity plane. We can create it by selecting **GameObject** -> **3D Object** -> **Plane** from the top menu. We will re-name it **Ground** and give it the following scale in the **Inspector**:



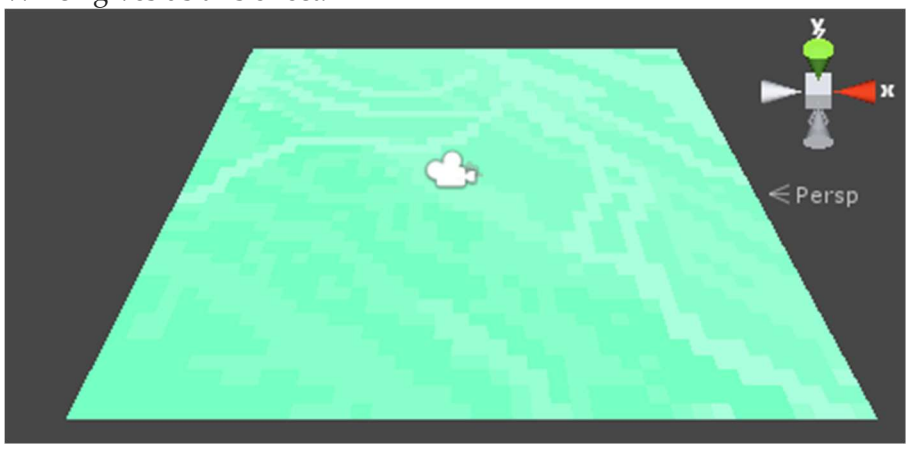
Note: it's important to give it exactly the name "Ground" as we will use this name later in our scripts.

Let's add some color.
Find the Textures Folder in Assets and drag **tex_ground** onto the **Ground** object we just created.

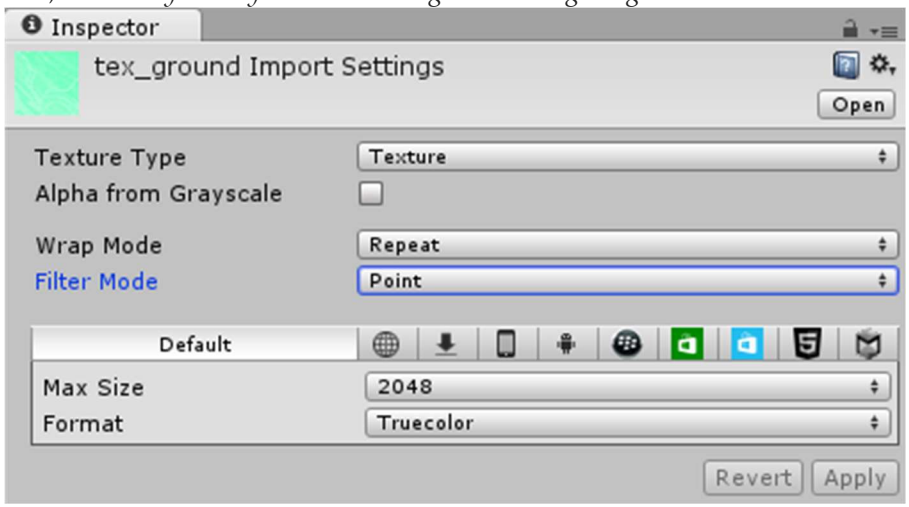
*Note: We could create a very simple pixelated **40x40 px** texture like this with any drawing tool we want. (e.g. Paint.NET):*



Which gives us this effect:



Note: We have set the Filter Mode to Point and the Format to Truecolor to give the texture a stylistic pixel look, however you may choose to change the settings to give the texture a smoother look.

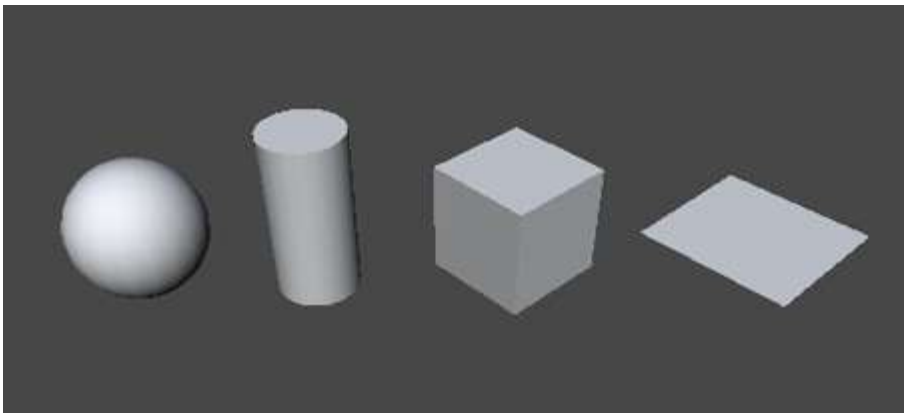


Creating the 3D-Models

It's time to create the 3D models for our game. We will do this use only Unity basic objects, no 3D modeling skills needed.

The Technique

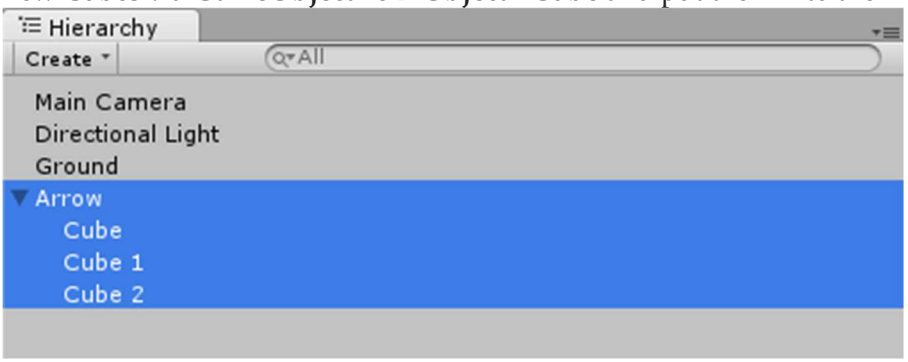
In Unity's **GameObject->3D Object** menu we can create basic geometry like Spheres, Cubes, Planes and Cylinders like those:



Then we put them all into an empty game object, name it and save it as a prefab by simply dragging it into the Project area *(preferably into a new folder called "Prefabs")*.

Creating the Arrow

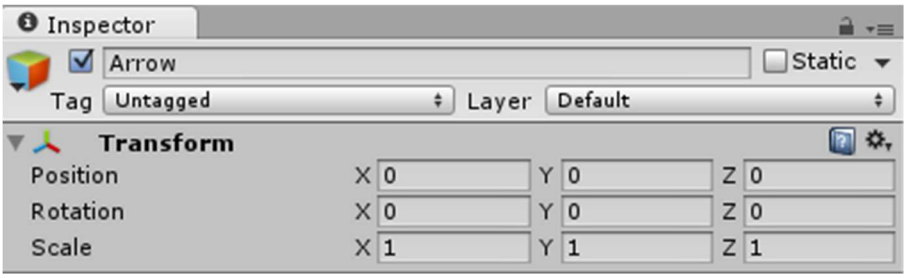
So to create an arrow that can be used to shoot things, we will first create an empty GameObject that holds the whole thing via **GameObject->Create Empty** and we name it Arrow. Then we create three new Cubes via **GameObject->3D Object->Cube** and put them into the Arrow GameObject like this:



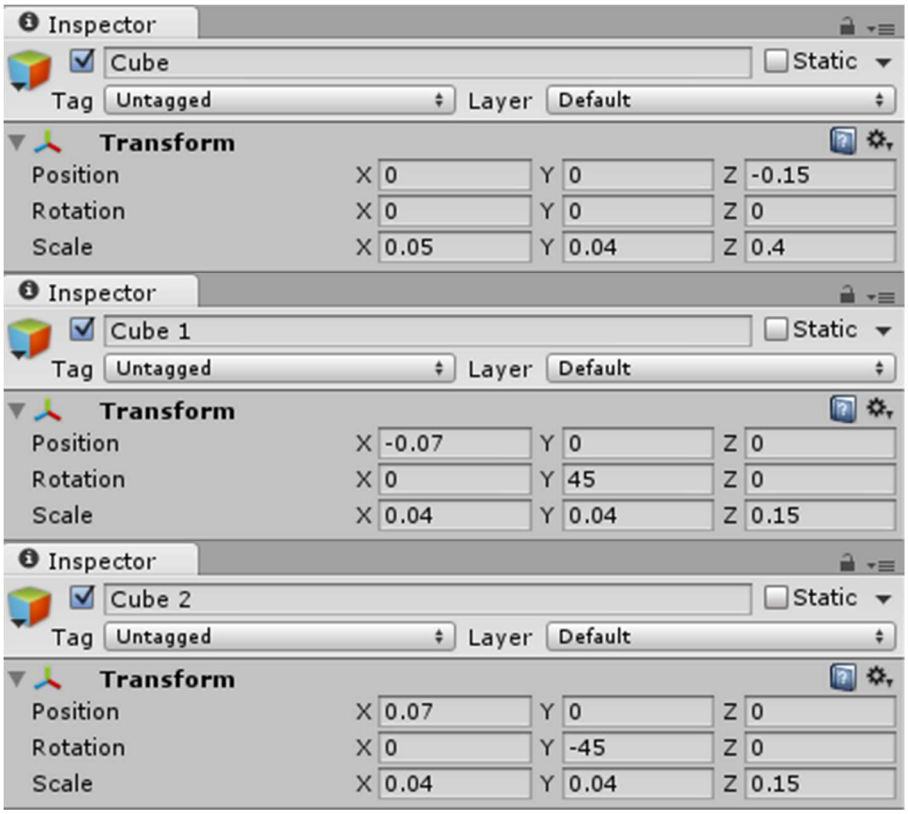
*Note: the Arrow is called the **root** object, and the Cubes are the **child** objects.*

Now to make the whole thing an arrow, we just move, scale and rotate each cube until it looks like one. Copy the following settings for each cube to create the arrow:

The Arrow:

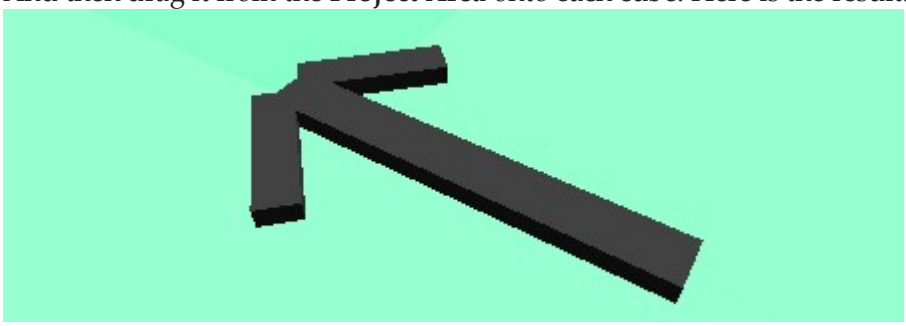


The Cubes:



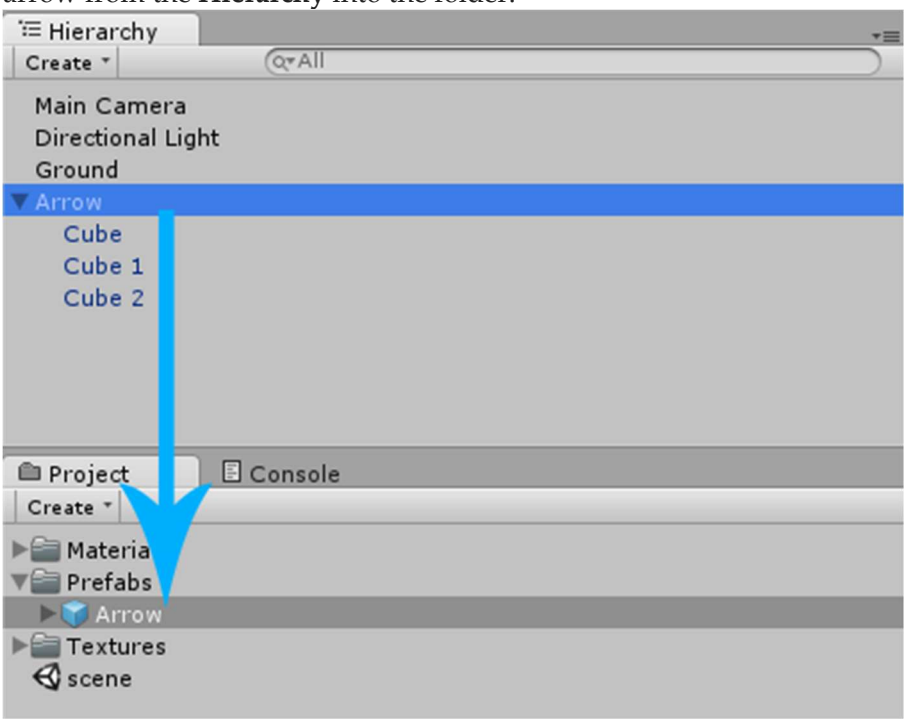
Alright, let's add some color to it. We will use a simple texture fount in our **Assets** folder: **tex_arrow**

And then drag it from the **Project Area** onto each cube. Here is the result:



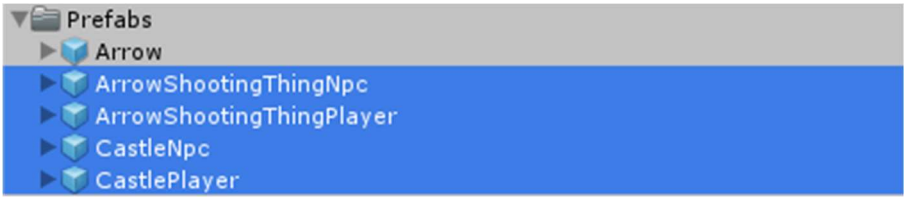
Now we don't want the arrow to be in the Scene all the time. Instead we will save our Arrow GameObject as an Arrow Prefab.

To create the Prefab, we will create a new "Prefabs" Folder in our **Project Area** and then drag the arrow from the **Hierarchy** into the folder:



Now we can delete it from the Hierarchy.

Here is what our Prefabs folder looks like now:



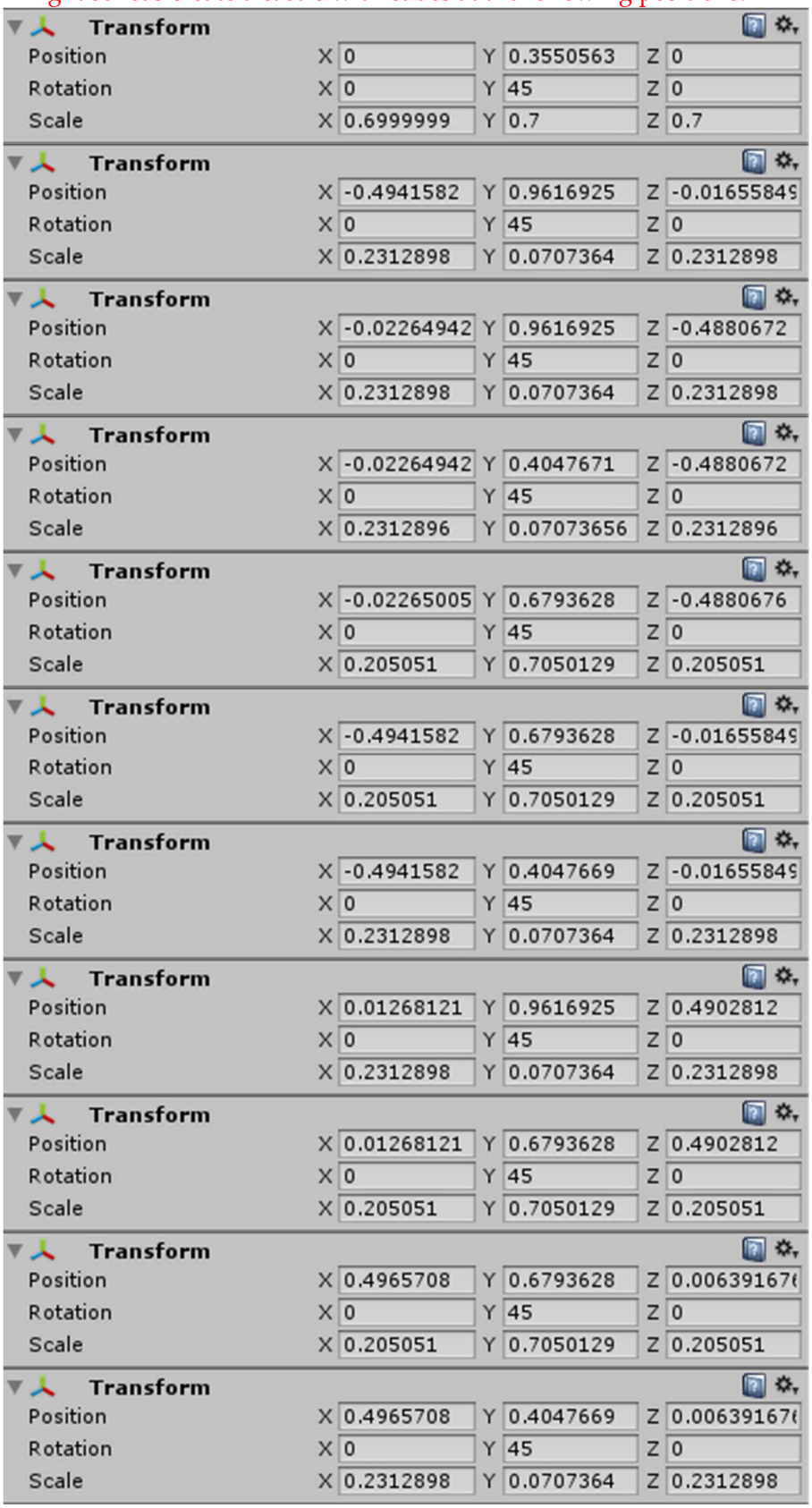
Optional: To save time the remaining Prefabs have been provided for you. If you would like to try creating them on your own, delete them from the Prefabs folder and follow these steps.

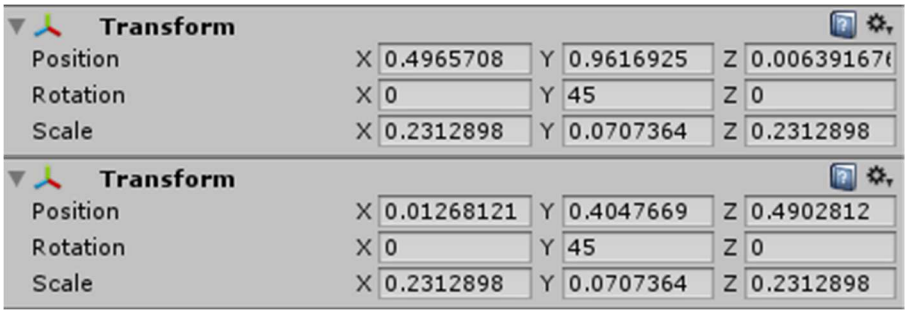
Castles & Arrow Shooting Things

We will use this same method to create all the remaining 3D models for our game.

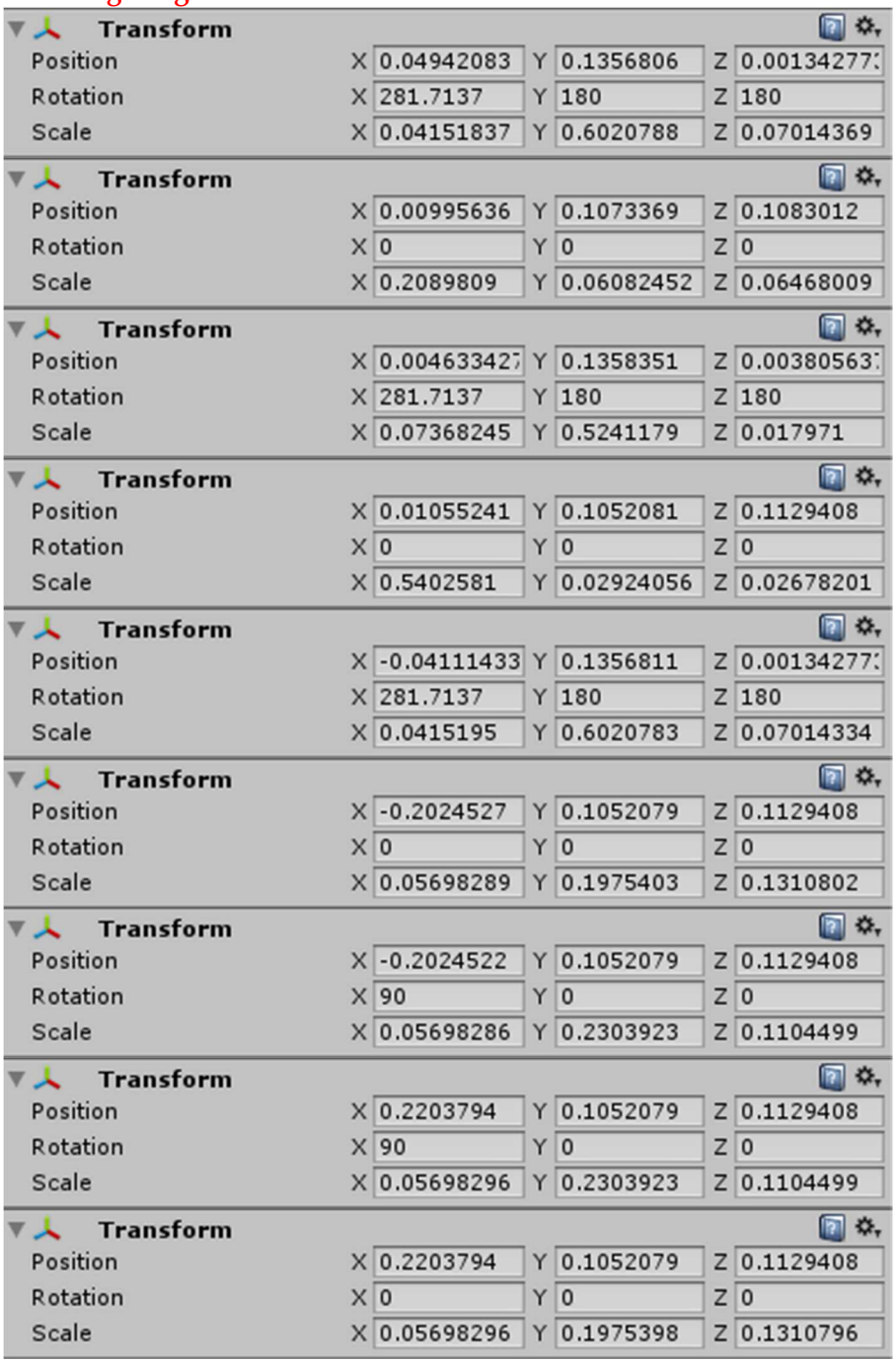
The most important thing to keep in mind is that the child GameObjects should always have a position somewhere close to **(0, 0, 0)**, because being a child, their position is always relative to the parent. Or in other words: if a child has a X-coordinate of 1, it means that it's 'one meter in x-direction away from the parent', no matter where the parent is. Whenever we move the parent, all the child GameObjects move along with it.

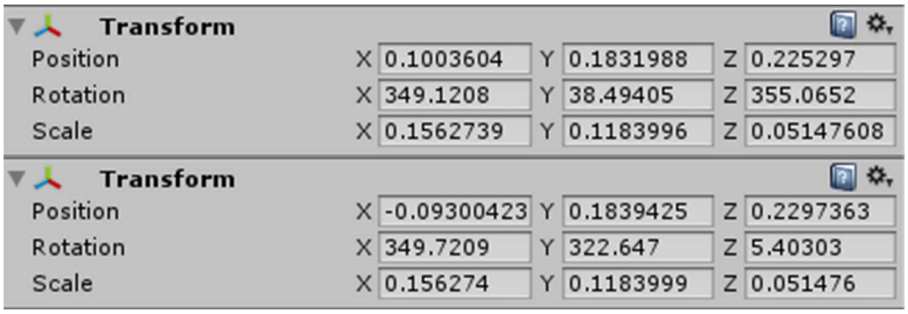
Alright so let's create a **castle** with cubes at the following positions:





And a **fighting unit** with cubes at:





We can then duplicate the castle and name one '**CastlePlayer**' and the other one '**CastleNpc**'. We will also duplicate the fighting unit and name one of it '**ArrowShootingThingPlayer**' and the other one '**ArrowShootingThingNpc**'.

Feel free to also put some simple colored textures onto them from the Textures folder:

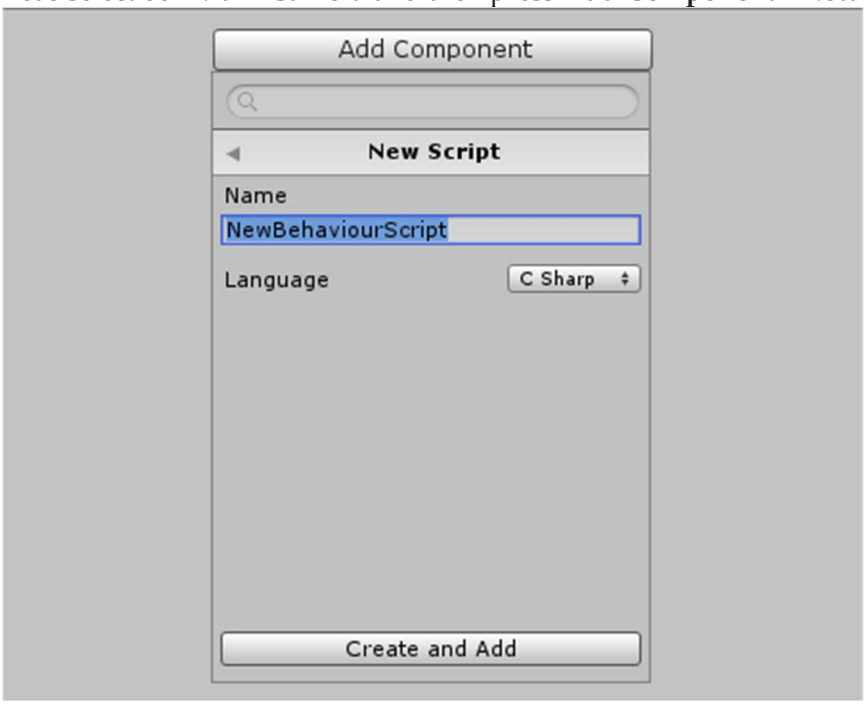
Note: we used orange and white for all Player models and black and white for all Npc models.

End of Optional.

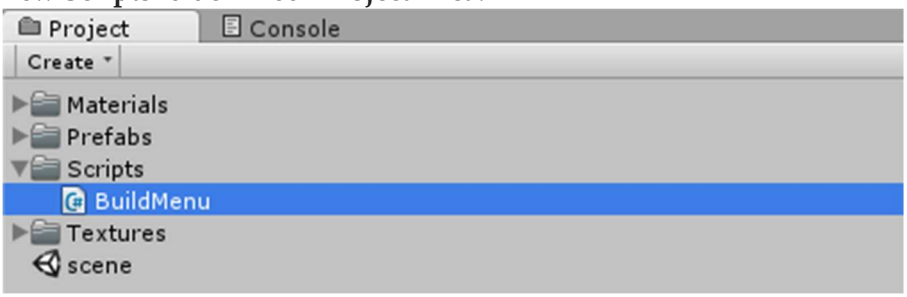
The Build Menu

Creating the Script

Let's select our **Main Camera** and then press **Add Component**->**New Script** in the **Inspector**:



We will name it **BuildMenu**, select **CSharp** as the language and then move it into a new **Scripts** folder in our **Project Area**:



Afterwards we can double click the Script in order to open it:

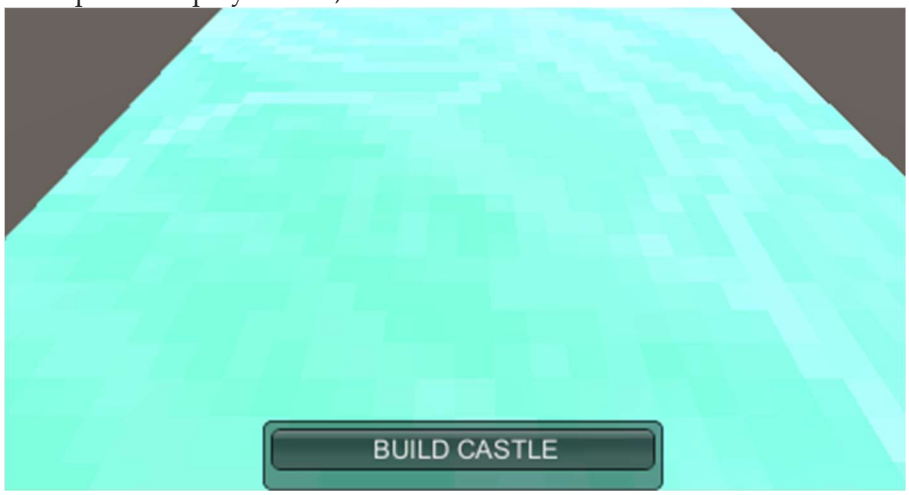
```
using UnityEngine;
using System.Collections;

public class BuildMenu : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

The GUI Code

Let's add some simple **GUI** code to our Script:

```
using UnityEngine;
using System.Collections;

public class BuildMenu : MonoBehaviour {
    public int width = 200;
    public int height = 35;

    void Update() {

    }

    void OnGUI() {
        GUILayout.BeginArea(new Rect(Screen.width/2 - width/2,
                                     Screen.height - height,
                                     width,
                                     height), "", "box");

        if (GUILayout.Button("BUILD CASTLE")) {
            // ToDo things...
        }
        GUILayout.EndArea();
    }
}
```

If we press the play button, we can see a GUI menu with a "**BUILD CASTLE**" button on it:



The Plan

Let's give the button some functionality. We want it to work like this:

- The Player presses the Button
- Now the Castle appears under the cursor
- The Player can move the cursor to position the Castle
- The Castle will be built when the Player does a mouse click

Instantiating the Castle

In order to make the castle appear under the cursor, we will add two variables. The first variable will allow us to specify which prefab to use. The second variable will keep track of the instantiated prefab *(when the prefab was copied into the game world)*:

```
...
    // This is the GUI size
    public int width = 200;
    public int height = 35;

    // This is the castle prefab, to be set in the inspector
    public GameObject prefab;

    // This holds the game-world instance of the prefab
    GameObject instance;
...
```

Okay so let's instantiate the prefab as soon as the player presses the button:

```
...
    if (GUILayout.Button("BUILD CASTLE")) {
        // Instantiate the prefab and keep track of it by assigning it to
        // our instance variable.
        instance = (GameObject)GameObject.Instantiate(prefab);
    }
...
```

This will put our castle into the game world.

Positioning the Castle at the Cursor Position

Now the player should be able to position it by moving the cursor. So whenever the player moves the cursor, we will have to find out the 3D world coordinates under the cursor, and then move our castle there.

To do this, we will use **Input.mousePosition**, then **ScreenPointToRay** to convert the mouse position to a ray that goes from the camera into the game world, and then find out where that ray hits the Ground via **Physics.Raycast**:

```
...
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;
    if (Physics.Raycast(ray, out hit)) {
        if (hit.transform.name == "Ground") {
            // Refresh the instance position
            instance.transform.position = hit.point;
        }
    }
...
```

Note: this is pretty much the default way to find out where the user clicked in the 3D world.

The Mouse Click to build the Castle

Now the last thing to do is build the castle as soon as the player does a mouse click. Let's think about this for a second. Right now, we already have some kind of castle in the game world, which is currently being positioned whenever the player moves the cursor. So all we really have to do is stop the player from being able to position the castle by clearing our instance variable. Hence we can't change its position anymore:

```
...
    if (Input.GetMouseButton(0)) {
        instance = null;
    }
...
```

The Final Script

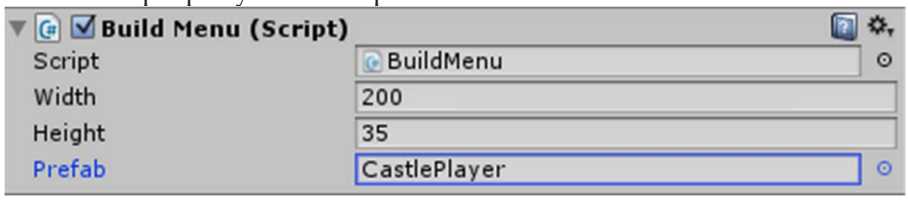
Here is how all those things work together in the final Script:

```
using UnityEngine;
using System.Collections;

public class BuildMenu : MonoBehaviour {
    // This is the GUI size
    public int width = 200;
    public int height = 35;

    // This is the castle prefab, to be set in the inspector
    public GameObject prefab;

    // This holds the game-world instance of the prefab
    GameObject instance;

    void Update() {
        // Is the player currently selecting a place to build the castle? Or
in
        // other words, was the instance variable set?
        if (instance != null) {
            // Find out the 3D world coordinates under the cursor
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast(ray, out hit)) {
                if (hit.transform.name == "Ground") {
                    // Refresh the instance position
                    instance.transform.position = hit.point;

                    // Note: if your castle appears to be 'in' the Ground
                    //       instead of 'on' the ground, you may have to
adjust
                    //       the y coordinate like so:
                    //instance.transform.position += new Vector3(0, 1.23f,
0);
                }
            }

            // Player clicked? Then stop positioning the castle by simply
            // loosing track of our instance. It's still in the game world
after-
```

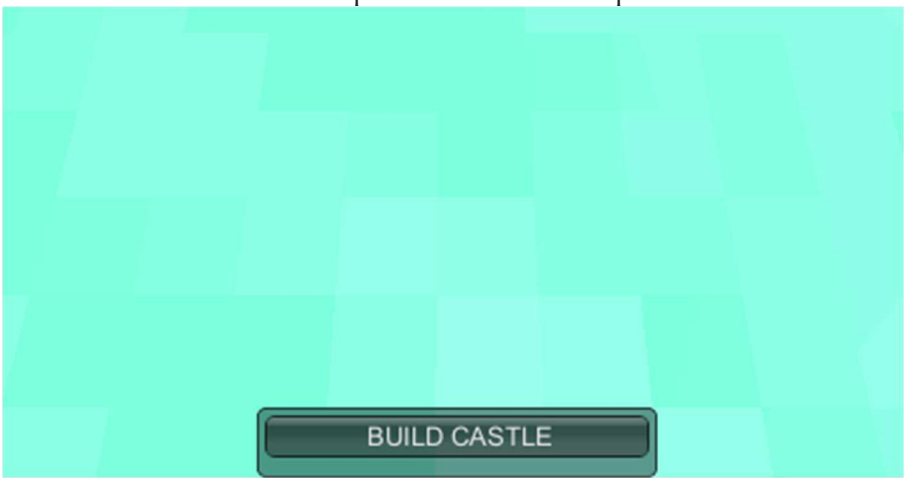
```
            // wards, but we just can't position it anymore.
            if (Input.GetMouseButton(0)) {
                instance = null;
            }
        }
    }

    void OnGUI() {
        GUILayout.BeginArea(new Rect(Screen.width/2 - width/2,
                                     Screen.height - height,
                                     width,
                                     height), "", "box");

        // Disable the building button if we are currently building
something.
        // Note: this enables GUIs if we have no instance at the moment, and
        //       it disables GUIs if we currently have one. Its just written
in
        //       a fancy way. (it can also be done with a if-else construct)
        GUI.enabled = (instance == null);
        if (GUILayout.Button("BUILD CASTLE")) {
            // Instantiate the prefab and keep track of it by assigning it to
            // our instance variable.
            instance = (GameObject)GameObject.Instantiate(prefab);
        }
        GUILayout.EndArea();
    }
}
```

Using the Script

Now we just select the Main Camera and drag our **CastlePlayer** Prefab from the Project Area into the **Prefab** property in the Inspector:



We can test it by pressing **Play** and then clicking on the **BUILD CASTLE** button. Afterwards we can move the castle around and position it with a simple mouse click:



Sinus Animation

The Concept

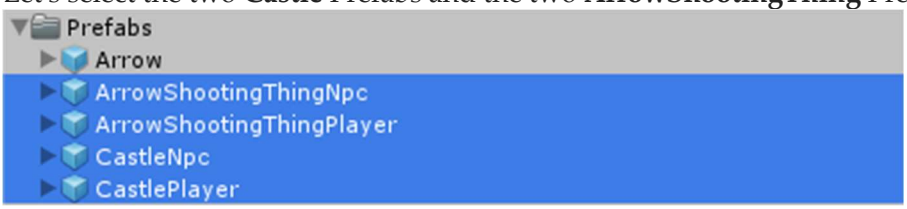
Before continuing with the fun stuff like spawning units, we will need a way to give some visual feedback by using animations.

The common way of animating things is by using tools like Blender. However, we can also animate objects using just Unity itself. We will create a new Animation script that simply resizes the castle to make it smaller, bigger and then back to the original size again.

There are lots of ways to implement this kind of behavior. The obvious way would be to use a Script's **Update** function and then modify the **transform.localScale** proper to go higher and higher until it reached a certain point, at which it starts to go lower and lower again, with the same speed all the time. This approach is not that good for performance reasons and because the movement won't look very smooth.

Instead we want the animation to look really smooth by slowly accelerating, then going faster upwards, downwards again and then slowing down before reaching the bottom again. This kind of movement can be implemented by using an AnimationCurve.

Let's select the two **Castle** Prefabs and the two **ArrowShootingThing** Prefabs in our **Project Area**:

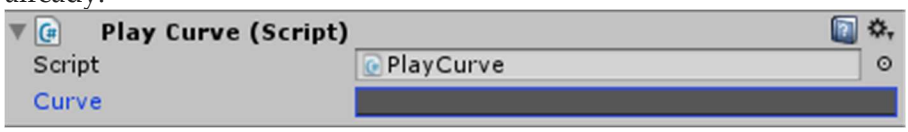


They will all need the growth animation, so let's select **Add Component->New Script** in the **Inspector**. We will name it **PlayCurve**, select **CSharp** as the language and then move the Script into our **Scripts** folder again.

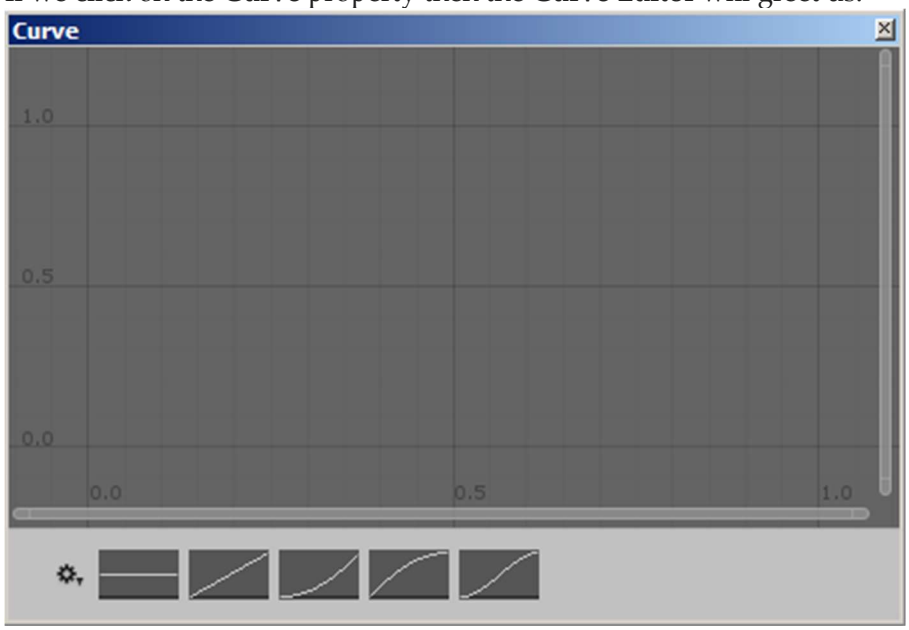
Let's double click the Script in order to open it. At first we will remove the **Start** and **Update** functions and give it a public **AnimationCurve** variable:

```
using UnityEngine;
using System.Collections;

public class PlayCurve : MonoBehaviour {

    // The Curve
    public AnimationCurve curve;
}
```

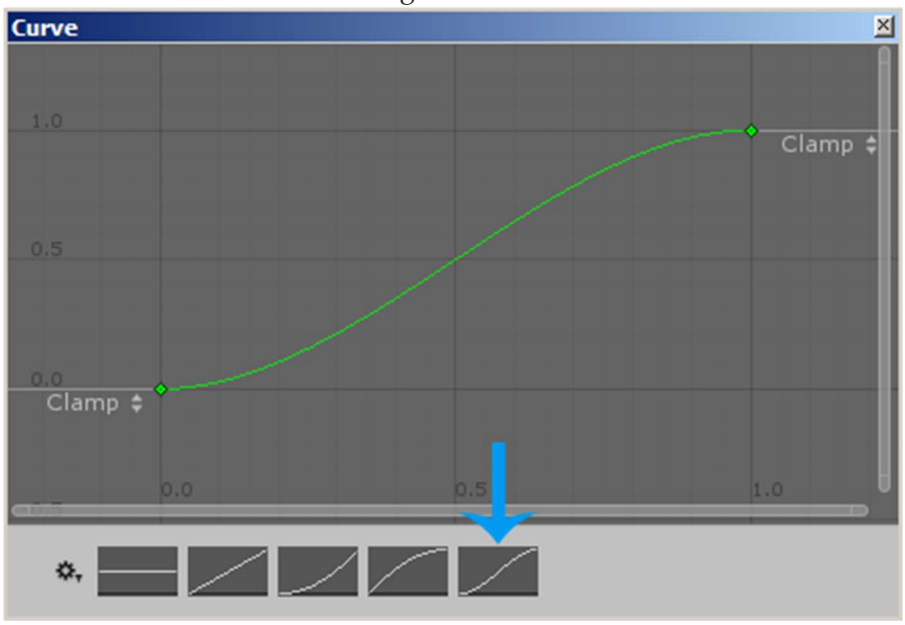
If we save the Script and take a look at the **Inspector** then we can see the Script's **Curve** variable already:



If we click on the **Curve** property then the **Curve Editor** will greet us:

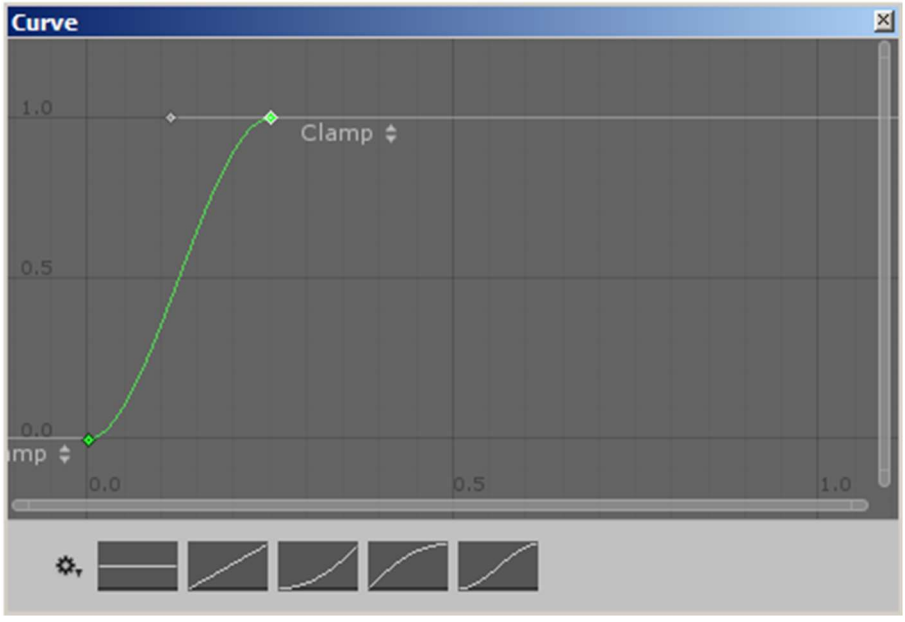


We will select the last curve image at the bottom in order to create the first half of our curve:



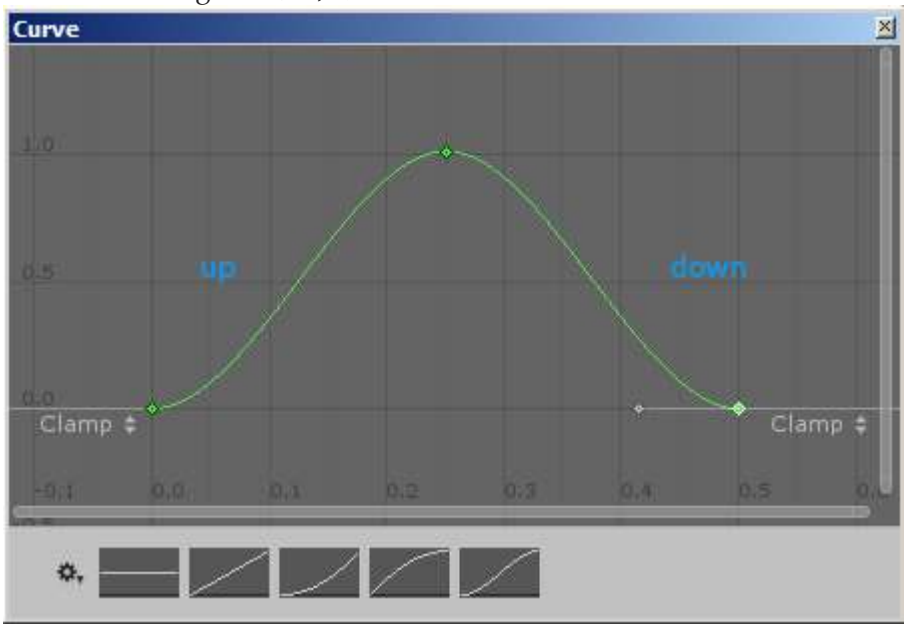
*Note: the horizontal axis will be the **time** and the vertical axis will be the **y** coordinate. In other words, the curve shows the **y coordinate over time**. We can also see that the curve slowly rises at first but then rises quicker later on. This is exactly the kind of movement that we are looking for.*

Now we can move the right green point much further to the left at **(0.25, 1)**:



Note: the wider the curve, the slower the Animation will look later on. If we squeeze it together like this then it will move faster.

Afterwards we can double click the right point to create a new one. We will then drag the new one to the bottom right at **(0.5, 0)**. Now we have the final curve that describes the up and down scaling:



Let's get back to our Script and make use of the curve. We will create a **Toggle** function that starts the whole animation:

```
using UnityEngine;
using System.Collections;

public class PlayCurve : MonoBehaviour {

    // The Curve
    public AnimationCurve curve;

    public void Toggle() {

    }
}
```

Now, we need to change the curve's position over time in a performance efficient way. The downside of using the **Update** function here is that it would still do work even after the movement was finished *(because things are checked in each Update call)*.

The solution is a Coroutine. In simplified terms, we will create a function that is called in the same interval as **Update**, and that function will be removed automatically as soon as we are done *(hence why there are no performance issues)*.

The Unity documentation describe a **coroutine** as: a function that has the ability to pause execution and return control to **Unity** but then to continue where it left off on the following frame.

Here is how our Coroutine looks:

```
IEnumerator sample() {
    // go through curve time
    for (float t=0; t < curve.keys[curve.length-1].time; t+=Time.deltaTime) {
        // do stuff...

        // come back next Update
        yield return null;
    }
}

public void Toggle() {
    StartCoroutine("sample");
}
```

*Note: while it looks really weird, it sure is not that hard to understand. We just created a **sample** function of type **IEnumerator**, because that's the type that Coroutines have. We then use a **for**-loop to step through our curve, starting at **0** and ending at the last value in the curve which is at **curve.keys[curve.length-1].time**. We increase our variable by **Time.deltaTime** in each step, just as we would do in the **Update** function. And the **yield** part simply tells Unity to **stop here and come back next Frame**. It's really as if we created our own Update function here.*

To make this more clear, here is what it would look like **if** we would use the **Update** function:

```
float t = 0;

void Update() {
    if (t < curve.keys[curve.length-1].time) {
        // do stuff..

        t += Time.deltaTime;
    }
}
```

We will stick to our Coroutine because there is **a lot** of value in understanding and using them properly.

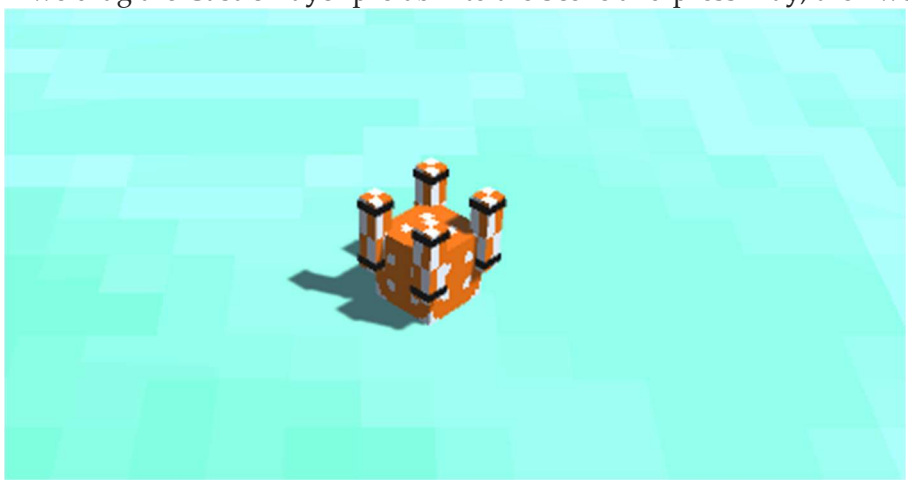
Now let's add some mechanics here. We want our object to grow/shrink a bit every time, and we will use the **curve.Evaluate** function to find out the growth at a given time:

```
IEnumerator sample() {
    // go through curve time
    for (float t=0; t<curve.keys[curve.length-1].time; t+=Time.deltaTime) {
        // scale a bit
        transform.localScale = Vector3.one * (1 + curve.Evaluate(t));

        // come back next Update
        yield return null;
    }
}
```

*Note: we use **1 + curve.Evaluate(t)** to make sure that the scale is always at least **1**.*
We can test our code by calling **Toggle** in a **Start** function once:

```
void Start() {
    Toggle(); // test
}
```

If we drag the **CastlePlayer** prefab into the Scene and press **Play**, then we can see that it works!



Now we can remove the **Start** function from our Script again, because it was just for testing. Here is our final Script:

```
using UnityEngine;
using System.Collections;

public class PlayCurve : MonoBehaviour {

    // The Curve
    public AnimationCurve curve;

    IEnumerator sample() {
        // go through curve time
        for (float t=0; t<curve.keys[curve.length-
1].time; t+=Time.deltaTime) {
            // scale a bit
            transform.localScale = Vector3.one * (1 + curve.Evaluate(t));

            // come back next Update
            yield return null;
        }
    }

    public void Toggle() {
        StartCoroutine("sample");
    }
}
```

We can also remove the **CastlePlayer** object from the **Hierarchy**.
Note: We will start making use of the Animation script in just a few minutes.

Unit Spawning

The Concept

Okay, let's spawn some ArrowFightingThing units for the Player and the Npc. At first we have to understand the difference between the Player and the Npc castles:

- The Player Castle spawns units after clicking on it
- The Npc Castle spawns units automatically every few seconds

But there is also a similarity:

- At some point, both should spawn a unit around the Castle.

Unity's component based nature is all about putting similarities into components that can be used by all kinds of things over and over again. Here is what we will do:

- Create a **UnitSpawner** Script that spawns units when asked to
- Create a **CastlePlayer** Script *(uses UnitSpawner when clicked)*
- Create a **CastleNpc** Script *(uses UnitSpawner every few seconds)*

Creating the Script

Let's select both castle prefabs in our **Project Area** and then click on **Add Component->New Script** in the **Inspector**. We will name it **UnitSpawner**, select **CSharp** as the language and then move it into our Scripts folder again.

Now we can open it:

```
using UnityEngine;
using System.Collections;

public class UnitSpawner : 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 them. We will create a **Spawn** function that plays our curve animation and then spawns a prefab somewhere around the castle. We will use **Random.Range** to get random x and z coordinates that are somewhere between -1 and 1 and to give each unit a different start rotation. We will also multiply x and z with the **spawnRange** variable to increase the distance a bit.

Here is our final Script:

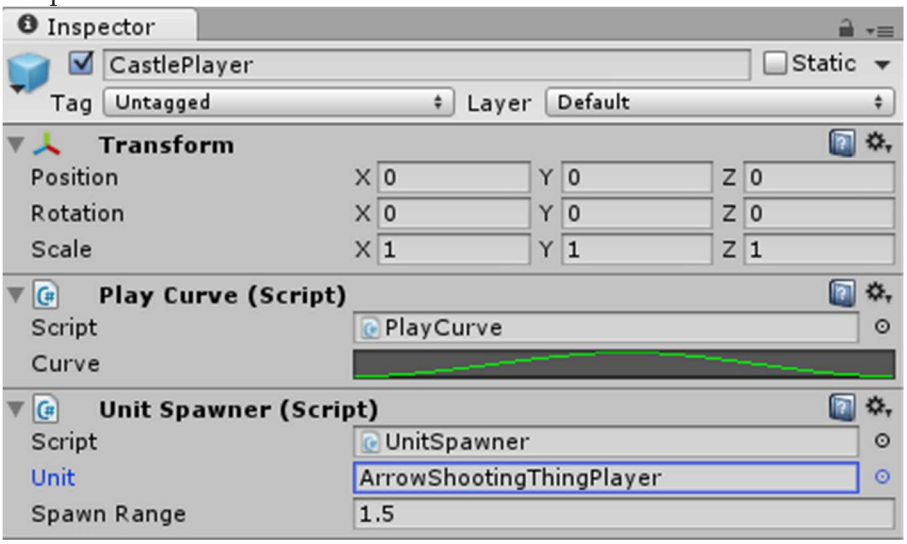
```
using UnityEngine;
using System.Collections;

public class UnitSpawner : MonoBehaviour {
    // unit prefab
    public GameObject unit;
    public float spawnRange = 1.5f;
```

```
    public void Spawn() {
        // start new animation
        GetComponent<PlayCurve>().Toggle();

        // create a new unit at some random position around this place
        Vector3 pos = transform.position;
        float x = pos.x + Random.Range(-1.0f, 1.0f) * spawnRange;
        float y = pos.y;
        float z = pos.z + Random.Range(-1.0f, 1.0f) * spawnRange;
        float angle = Random.Range(0.0f, 360.0f);
        Instantiate(unit, new Vector3(x, y, z), Quaternion.Euler(0.0f, angle,
0.0f));
    }
}
```

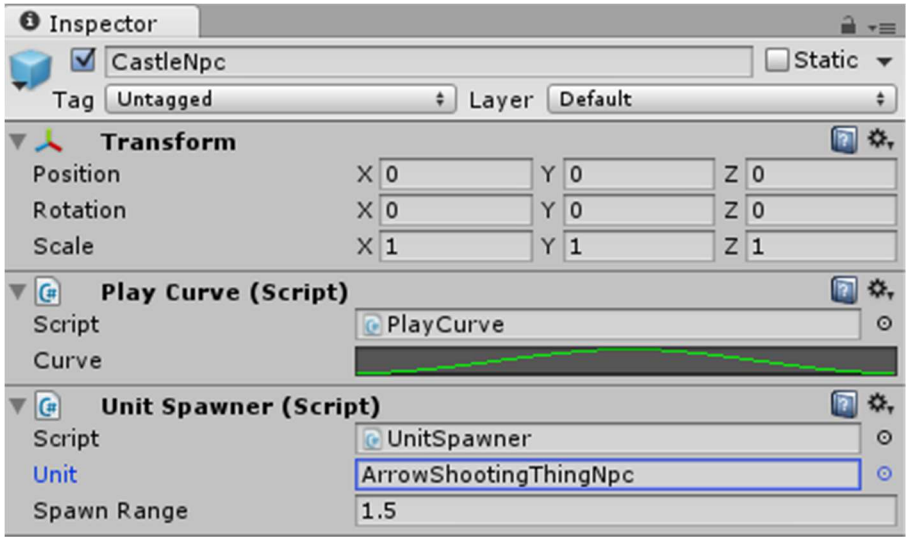
*Note: we used **GetComponent** to access the **PlayCurve** Script that is also attached to the castle. We then use **Instantiate** to load the **unit** prefab into the game world. We position it somewhere randomly around the castle and we also give it a random rotation by using **Quaternion.Euler**.*

Setting the Prefabs

Let's make sure to set the unit that is supposed to be spawned by our **UnitSpawner** Script. At first we will select the **CastlePlayer** prefab and then drag the **ArrowShootingThingPlayer** prefab into the Script's **unit** slot:



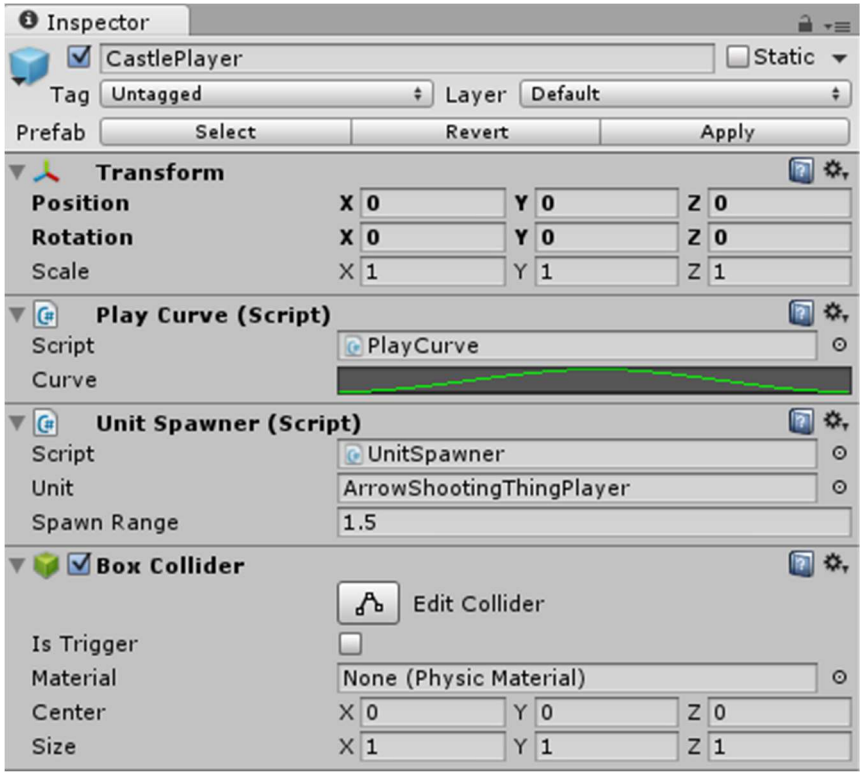
Afterwards we will select the **CastleNpc** prefab and then drag the **ArrowShootingThingNpc** prefab into the **unit** slot:



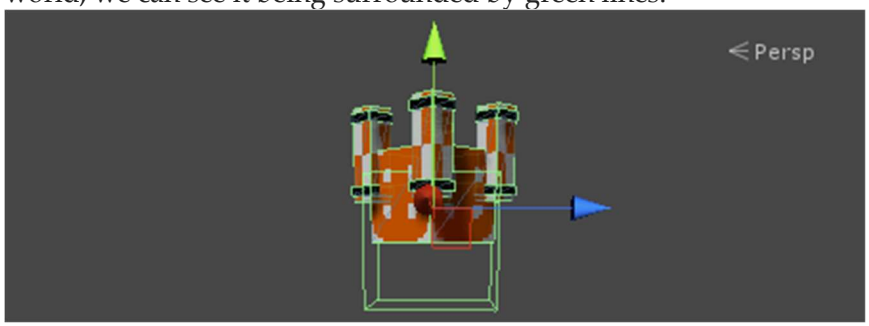
Now even though we just created the animation and unit spawning scripts, we still don't see any effect. Let's create the CastlePlayer and CastleNpc Scripts that will make use of the whole thing.

The CastlePlayer Script

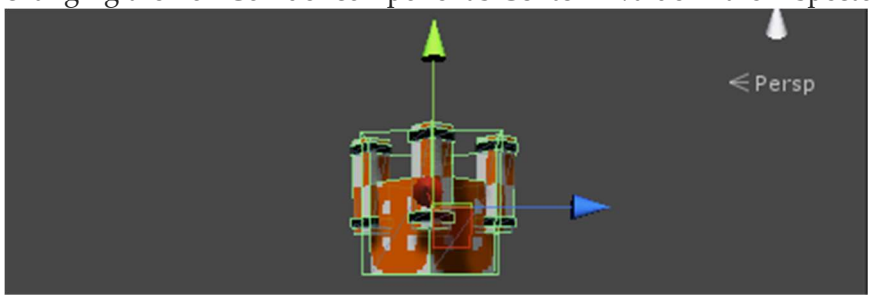
We will let the player create a new ArrowShootingThing unit by simply clicking on the castle. The whole thing only works if the Castle prefab has a collider on it. So let's select our CastlePlayer prefab in the project area and select **Add Component**->**Physics**->**BoxCollider** in the menu. This adds a box collider to our Prefab:



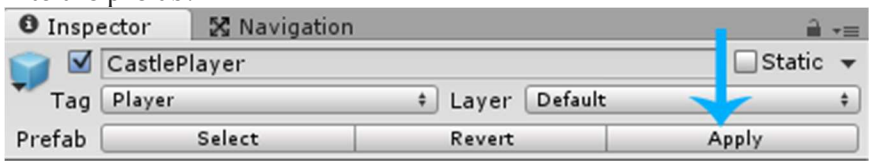
One more thing: the collider might not be adjusted properly. If we drag the prefab into the game world, we can see it being surrounded by green lines:



All the child cubes have a collider by default *(which is properly adjusted)*. But our recently added collider *(the biggest green box in the picture)* is clearly a bit below our castle. We can adjust it by changing the Box Collider component's **Center Y** value in the Inspector so it looks like this:



Afterwards we press the **Apply** button in the Inspector *(at the top right area)* to save the modifications into the prefab:



And then we can delete the **CastlePlayer** GameObject from the **Hierarchy** again.
Now that our Player castle has a collider, we can get to the **CastlePlayer** Script. Let's select the **CastlePlayer** prefab in our **Project Area** and then click on **Add Component->New Script**, name it **CastlePlayer**, select **CSharp** as a language, move it into our **Scripts** folder and then open it:

```
using UnityEngine;
using System.Collections;

public class CastlePlayer : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

We can remove the **Start** and the **Update** function, because we won't need them. Instead we will use Unity's OnMouseDown function and put our spawn code into it:

```
using UnityEngine;
using System.Collections;

public class CastlePlayer : MonoBehaviour {
    // Note: OnMouseDown only works if object has a collider
    void OnMouseDown() {
        // use UnitSpawner
        GetComponent<UnitSpawner>().Spawn();
    }
}
```

If we press the play button, build a castle and click on it then a unit appears:

The CastleNpc Script

Again we will begin by selecting our **CastleNpc** prefab in the **Project Area** and then we can click on **Add Component**->**New Script**, name it **CastleNpc**, select **CSharp** as the language, move it into our **Scripts** folder and then open it:

```
using UnityEngine;
using System.Collections;

public class CastleNpc : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

We won't need the **Update** function, so let's remove it. We will add a public variable for the build interval and then build a new unit every *'interval'* seconds. We can use Unity's InvokeRepeating to call a **Spawn** every few seconds.

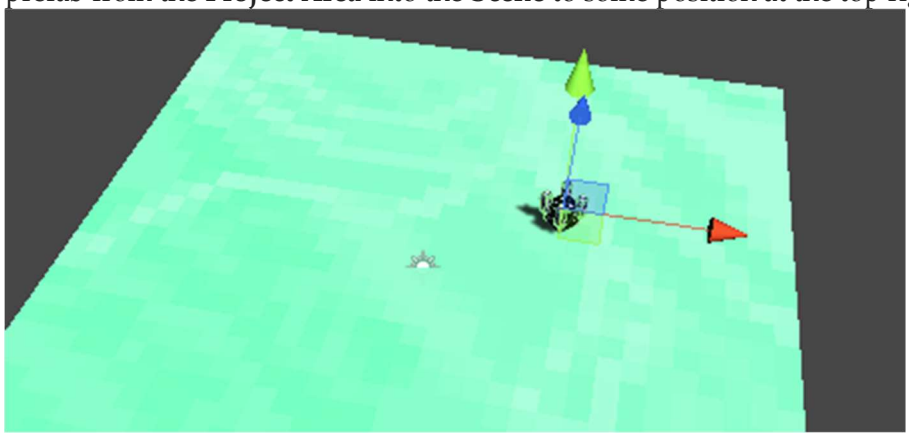
However, we need a little workaround because this doesn't work:

```
InvokeRepeating("GetComponent<UnitSpawner>().Spawn()", 1, 1);
```

Instead we will create a helper function that accesses the **UnitSpawner**'s Spawn function and then call that helper function with InvokeRepeating:

```
using UnityEngine;
using System.Collections;

public class CastleNpc : MonoBehaviour {
    // Parameters
    public float spawnInterval = 3.0f;

    // Use this for initialization
    void Start () {
        InvokeRepeating("SpawnHelp", spawnInterval, spawnInterval);
    }

    void SpawnHelp() {
        // this doesn't work directly with InvokeRepeating..
        GetComponent<UnitSpawner>().Spawn();
    }
}
```

So far, so good. We still have to place the Npc castle somewhere in the game world, so let's drag the prefab from the **Project Area** into the **Scene** to some position at the top right like **(3.5, 0, 2.5)**:



If we press play and wait a few seconds, then we can see the Npc building his units automatically:

Our project slowly starts to look like an actual Real Time Strategy game...