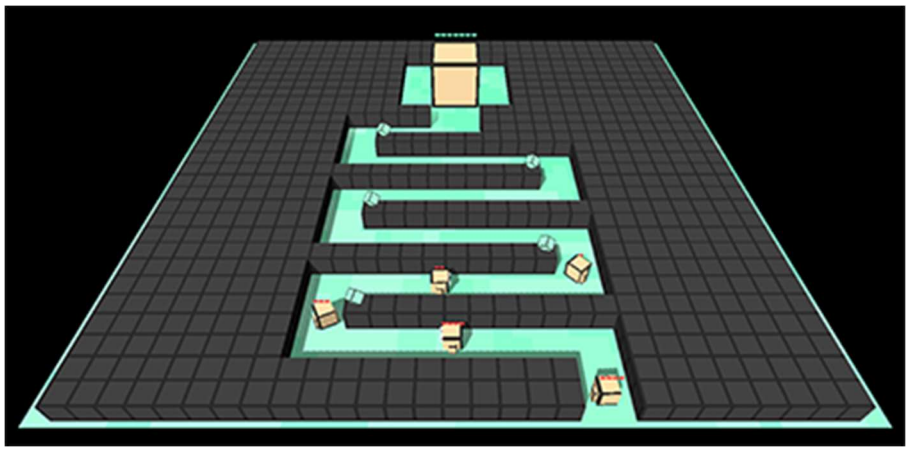
Unity Tower Defense Tutorial



Intro

In this Tutorial we will create a three-dimensional Tower Defense game. The game will be very easy to make, you won't need any 3D modeling skills, animations, or anything very complicated. We will focus on the programming portion of the game in order to learn more about Unity's scripting API.

After opening the project folder you will notice two more folders (these are the things with the unity icon), one called "**Finished Tower Defense**" another called "**Tower Defense Tutorial**".

"**Finished Tower Defense**" is the completed game with all the objects placed, components added, scripts attached, etc… You can select "**Finished Tower Defense**" to view the final project and see how things should be set up, but we will be recreating the game from scratch in "**Tower Defense Tutorial**".

"**Tower Defense Tutorial**" is a project that contains all objects placed, but is missing the scripts. If you want to practice recreating the entire game from scratch, you can create a brand new empty project in Unity or start a new scene in "**Tower Defense Tutorial.**" This walkthrough will assume you are recreating the entire game from scratch, but during the live workshop we will only focus on scripting.

Game Mechanics

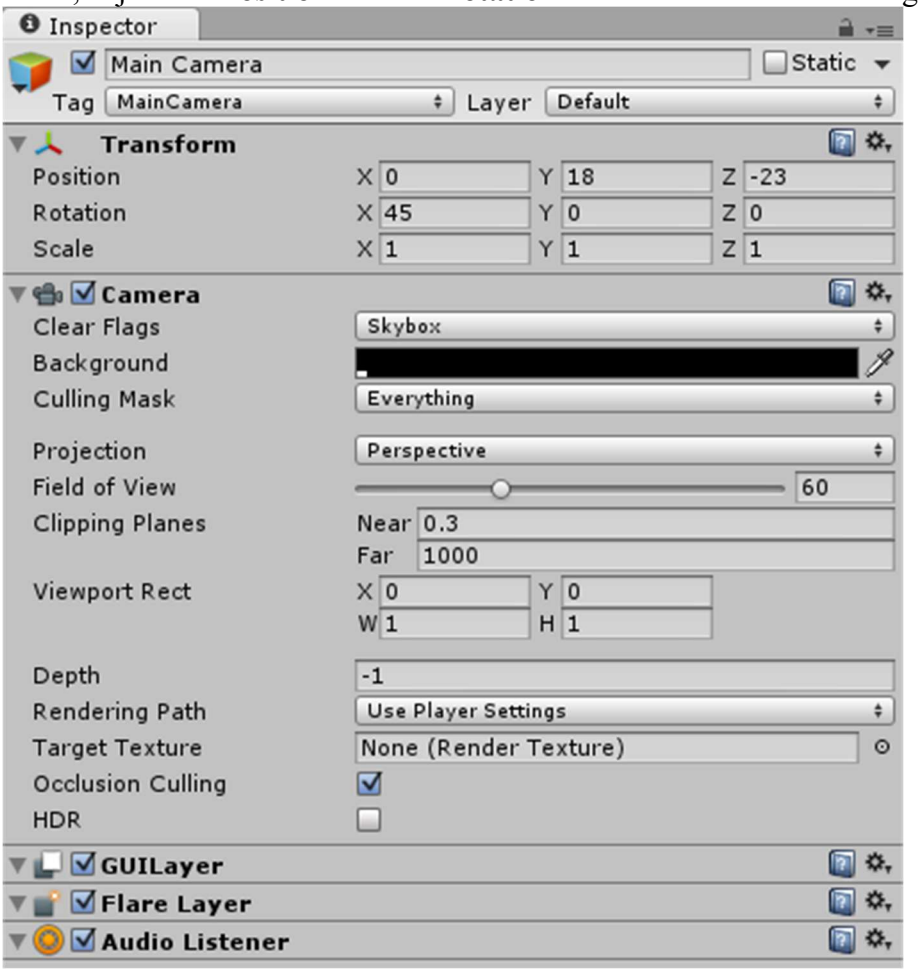
For those who never heard about Tower Defense games before, here is what basically happens:

- There is a point where Monsters spawn each few seconds
- The Monsters run through the world towards a Castle that they want to destroy
- The player can build Towers that attack the Monsters

In addition, we will use **Buildplaces**. Buildplaces are predefined places on the map where the player can build Towers on. We will use them because it will make our lives much easier later on *(in compare to being able to build towers everywhere on the map)*.

Project Setup

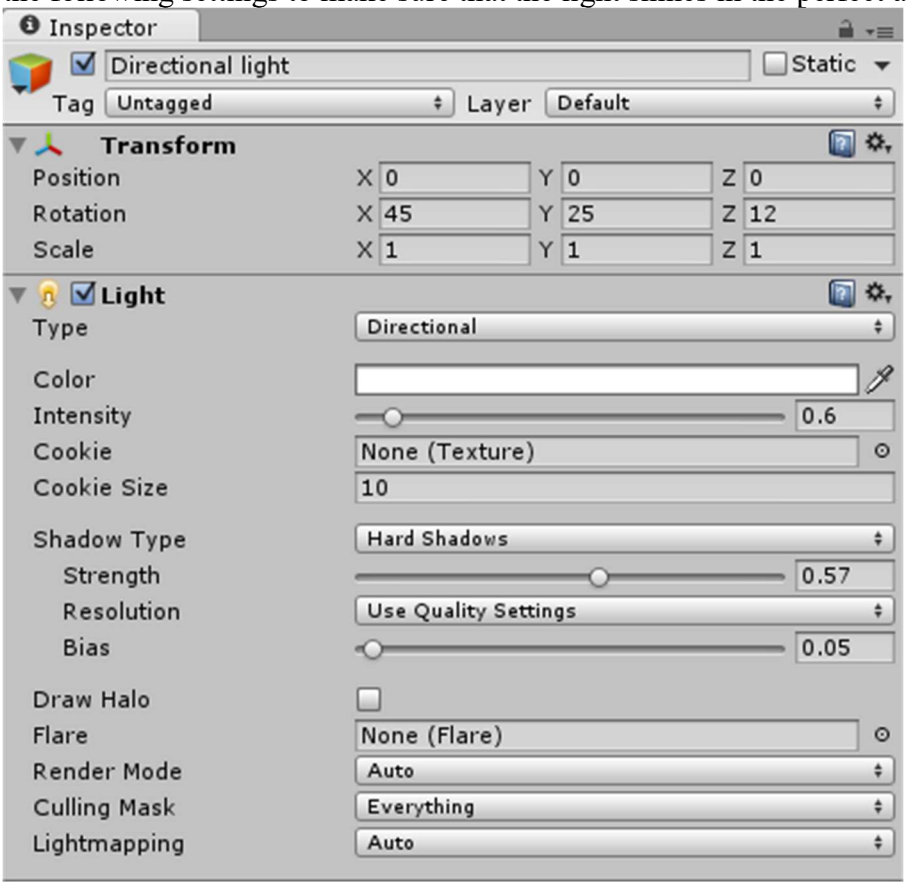
If we select the **Main Camera** in the **Hierarchy** then we can set the **Background Color** to black, adjust the **Position** and the **Rotation** like shown in the following image:



Note: this will make the camera look down onto the game world in a 45° angle later.

The Light

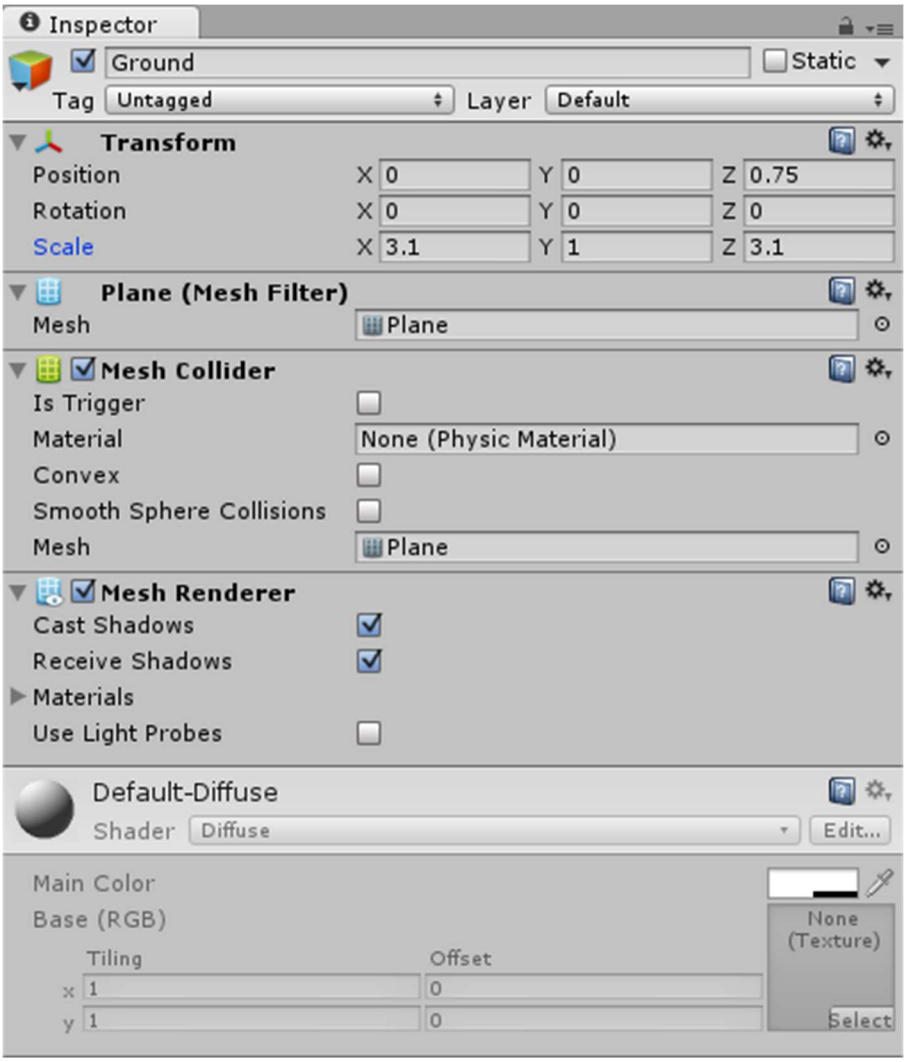
Let's add some light to our game, so that the world won't be too dark later on. We can add a light source by selecting **GameObject->Light->Directional Light** in the top menu. We will then use the following settings to make sure that the light shines in the perfect angle onto our scene:



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

The Ground

Alright so we will need some kind of Ground for the Monsters to walk on. Let's add a Plane by selecting **GameObject->3D Object->Plane** from the top menu. We will name it **Ground** and assign the following **Position** and **Scale** in the Inspector:

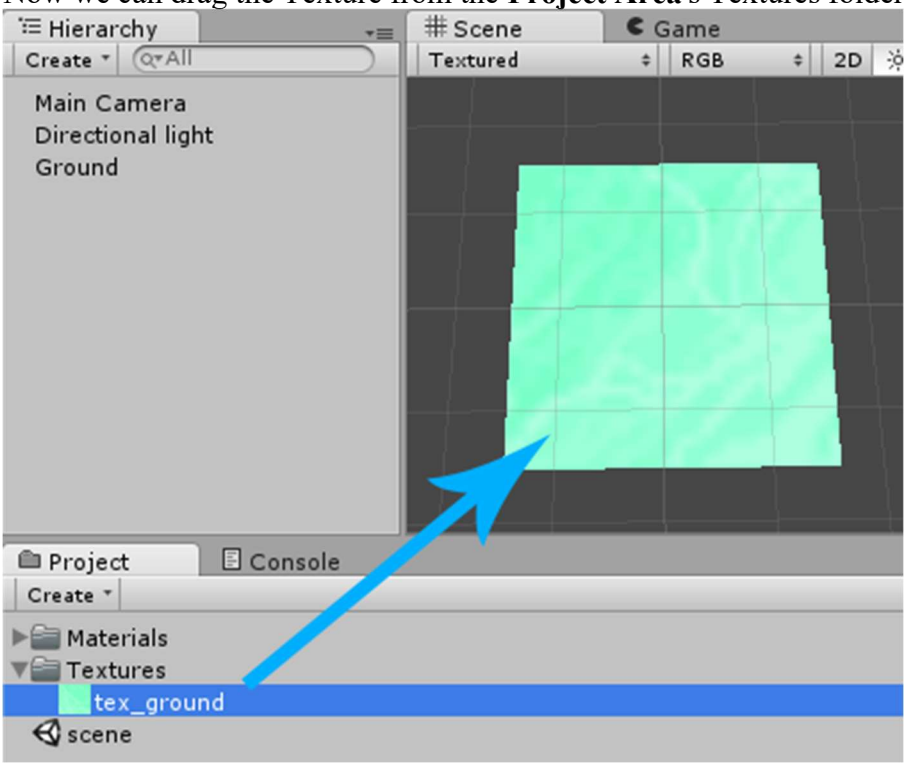


Note: this will be the perfect size to fit all the Buildplaces and the castle.

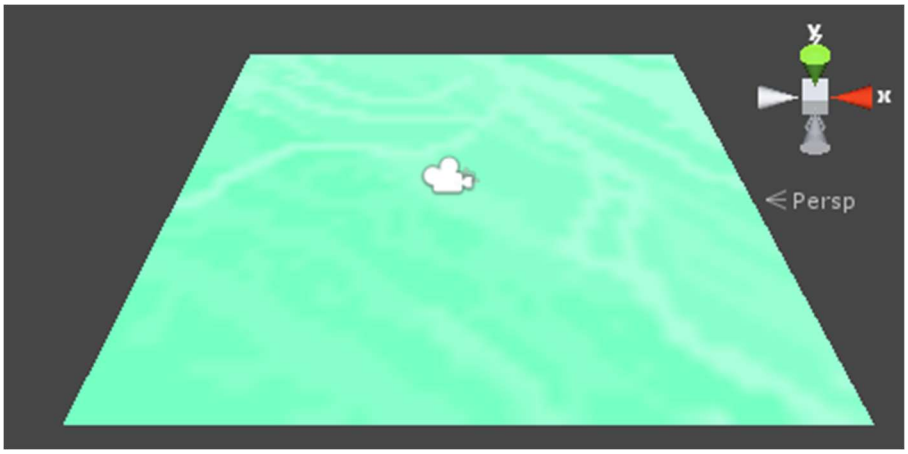
Open your drawing tool of choice and then create a small 40 x 40 px texture with just some basic green tones. Feel free to create your own texture by filling it with some base color and then drawing in a few random lines with a slightly brighter or darker green tone. Or you use this one:

*Note: right click on the image, select **Save As...**, navigate to the project's **Assets** folder and save it in a new **Textures** folder "tex_ground.png"..*

Now we can drag the Texture from the **Project Area**'s Textures folder onto the **Ground** plane:

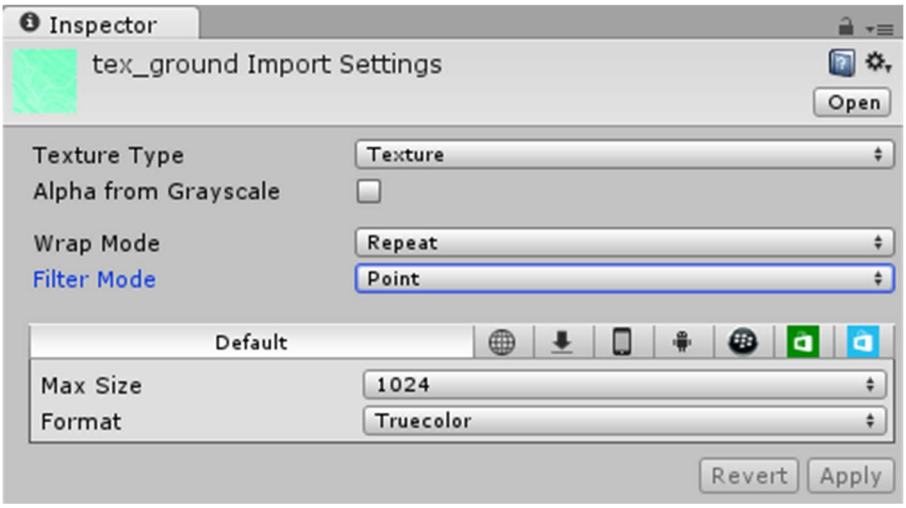


If we take a closer look at the Ground plane now, then we can see how the Texture looks really smooth:



This is fine for most games, but for our Tower Defense Tutorial we want the Texture to look pixelated to achieve a more unique style.

Select the Texture in our **Project Area** and then change the **Import Settings** in the **Inspector** like shown below:



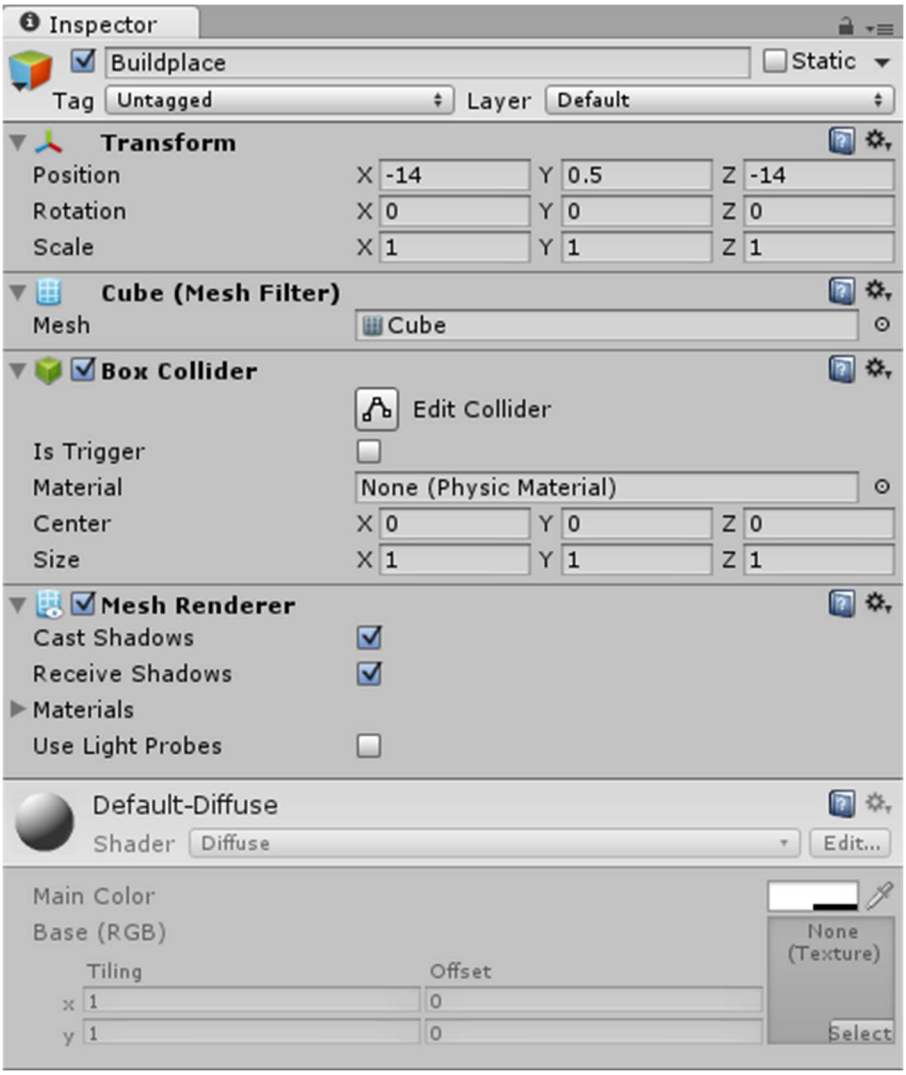
It looks completely pixel exact now:

Note: we will use this style for all the Textures in our game.

The Buildplaces

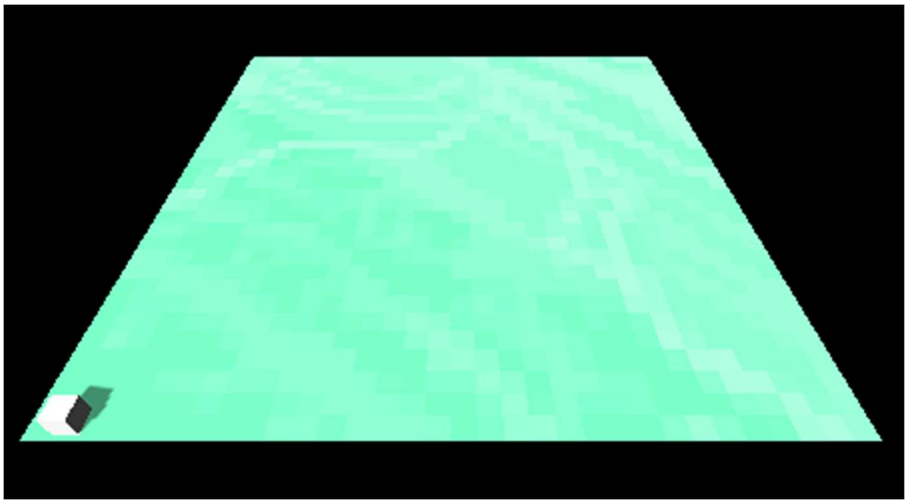
The Buildplace Cube

Alright, we will need some kind of Buildplace where the player can build a Tower on *(instead of being able to build it anywhere in the world, which would be much more complicated)*. We will begin by creating one Buildplace and then duplicate it a few times so that we can use the Buildplaces to design some kind of maze for the monsters to walk in.
Let's select **GameObject->3D Object->Cube** from the top menu. We will position it at **(-14, 0.5, -14)** and name it **Buildplace**:



*Note: the **x** and **z** coordinates should always be rounded like **-14** (instead of something like -13.99). The **y** position is **0.5** so that the cube stands exactly on top of the Ground plane, instead of being half way inside it.*

Here is how it looks in the Scene:

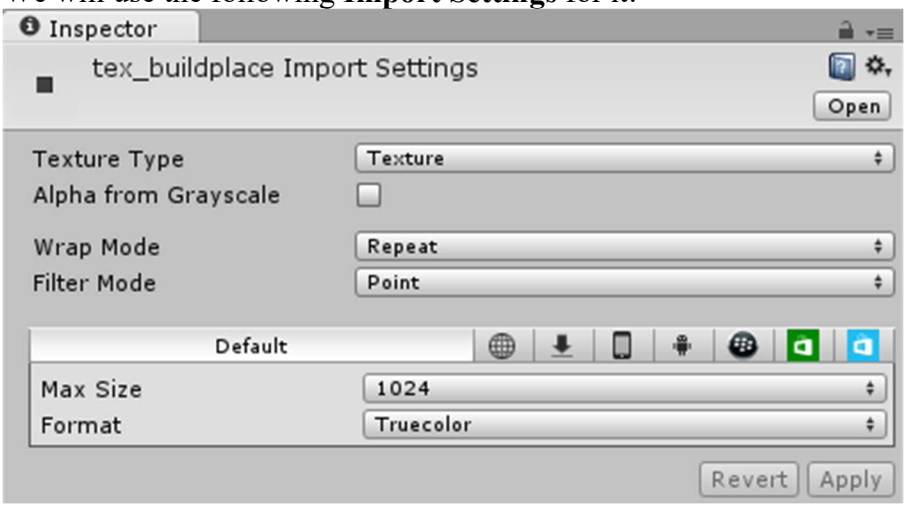


The Buildplace Texture

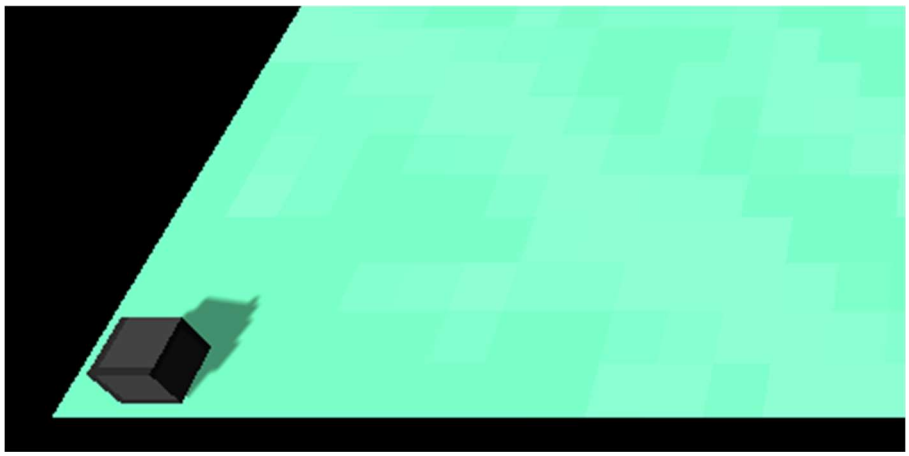
Let's also draw a very simple 8 x 8 px Texture that we can put on it:

*Note: right click on the image above, select **Save As...** and save it in the project's **Assets/Textures** folder "tex_buildplace.png"..*

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

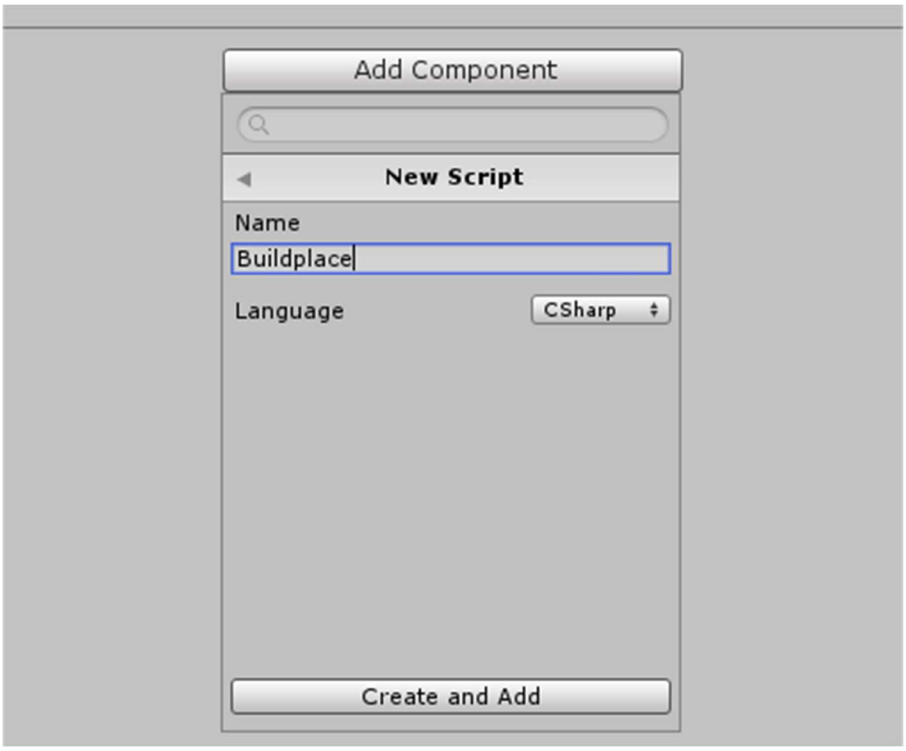


Afterwards we can drag it from the **Project Area** onto the **Buildplace** so that it looks nicely textured:

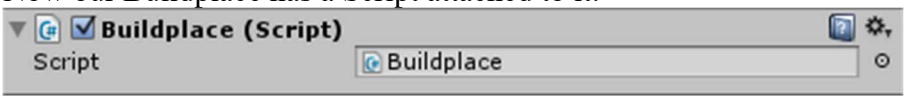


The Buildplace Script

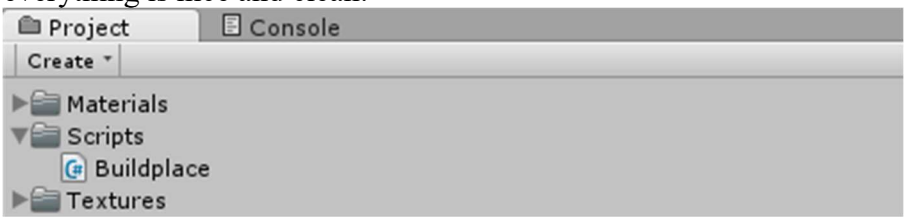
We want to build a Tower on top of a Buildplace as soon as the user clicks on it. This kind of behavior is always implemented with **Scripting**. We can create a new Script by selecting **Add Component->New Script**in the **Inspector**. We will name it **Buildplace** and select **CSharp** as the language:



Now our Buildplace has a Script attached to it:



Let's also create a new **Scripts** folder in the **Project Area** and move the Script into it, so that everything is nice and clean:



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

```
using UnityEngine;
using System.Collections;

public class Buildplace : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

First of all, we won't need the **Start** or the **Update** methods, so let's remove them:

```
using UnityEngine;
using System.Collections;

public class Buildplace : MonoBehaviour {

}
```

We will however need some kind of public variable to specify the Tower that should be built later on:

```
using UnityEngine;
using System.Collections;

public class Buildplace : MonoBehaviour {
    // The Tower that should be built
    public GameObject towerPrefab;

}
```

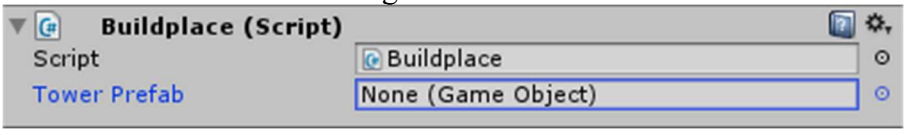
*Note: because the **towerPrefab** variable is public, we can specify it in the **Inspector** later on.*
The next step is to use the Instantiate function to build the Tower after the Buildplace was clicked. As usual, Unity makes our lives very easy here because it already offers a **OnMouseUpAsButton** function that we can make use of:

```
using UnityEngine;
using System.Collections;

public class Buildplace : MonoBehaviour {
    // The Tower that should be built
    public GameObject towerPrefab;

    void OnMouseUpAsButton() {
        // TODO build stuff...
    }
}
```

Now all we have to do is build the Tower above the Buildplace:

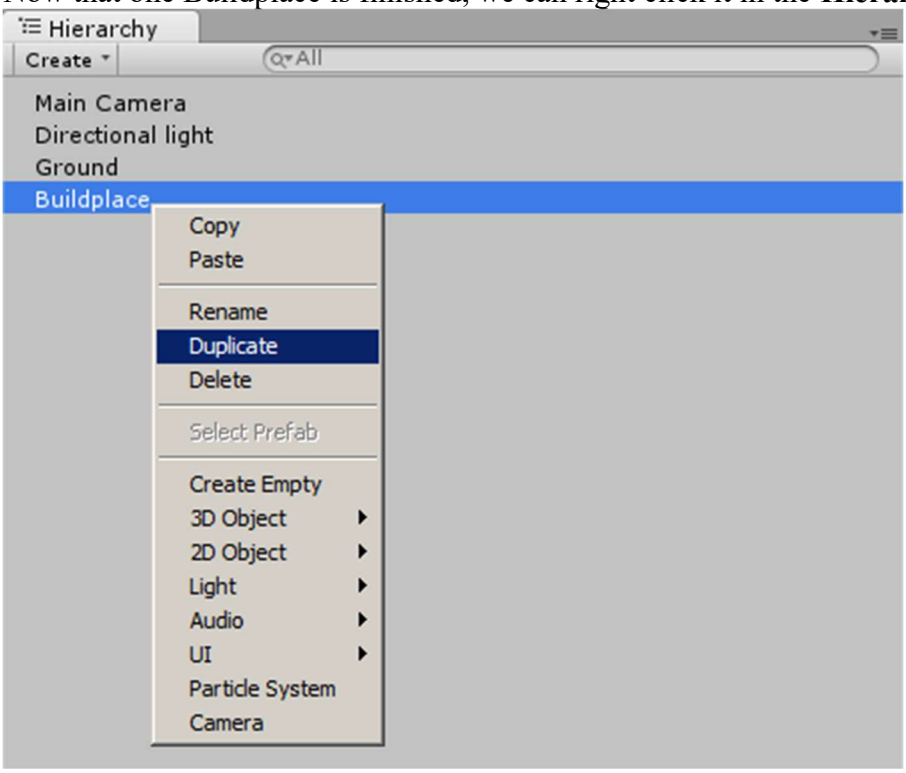
```
void OnMouseUpAsButton() {
    // Build Tower above Buildplace
    GameObject g = (GameObject)Instantiate(towerPrefab);
    g.transform.position = transform.position + Vector3.up;
}
```

*Note: at first we load the towerPrefab into the game by using **Instantiate**. Afterwards we modify the position to be the current Buildplace's position + one unit upwards.*
If we save the Script and take a look at the **Inspector**, then we can also see the **Tower Prefab** slot where we can drag a Tower into later on:

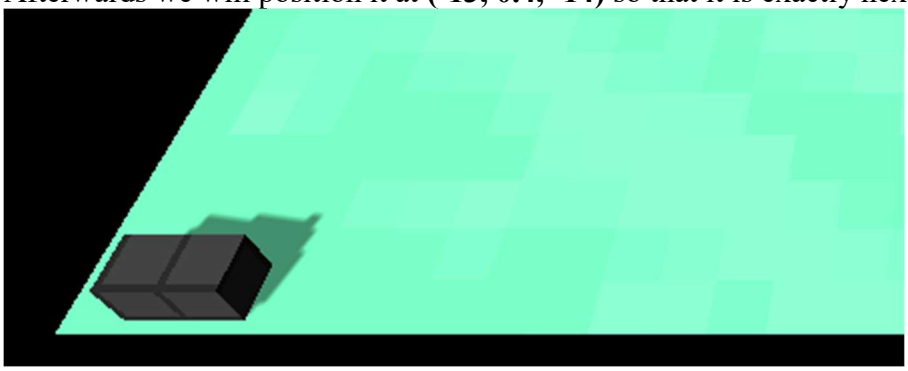


Creating more Buildplaces

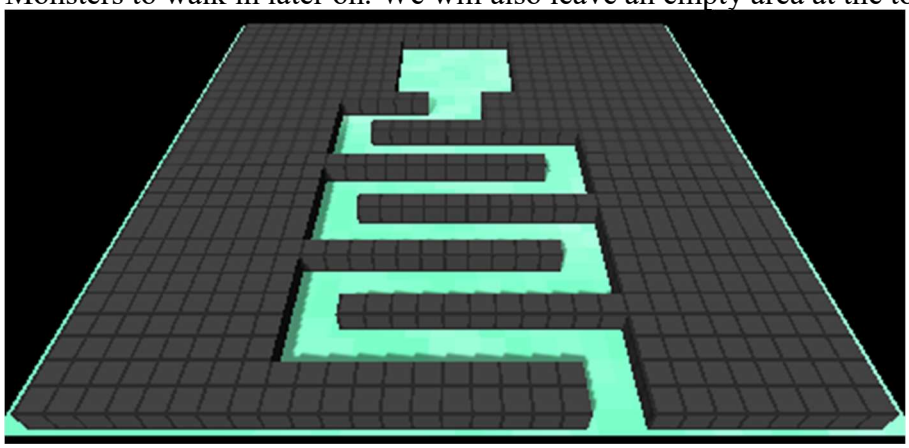
Now that one Buildplace is finished, we can right click it in the **Hierarchy** and select **Duplicate**:



Afterwards we will position it at **(-13, 0.4, -14)** so that it is exactly next to the previous one:



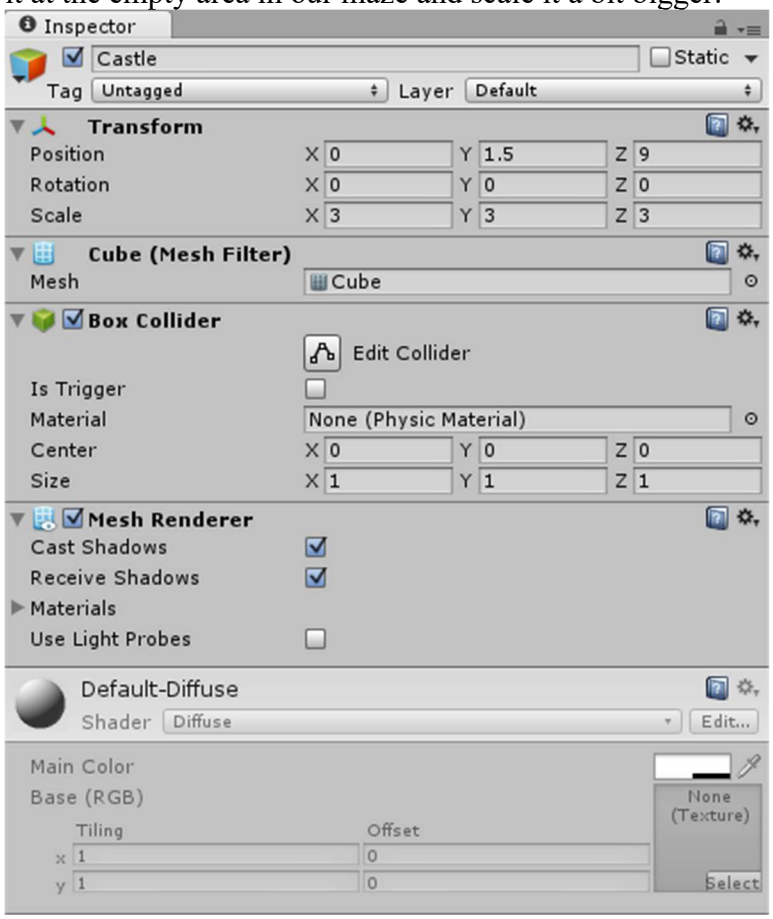
We will repeat this process quite a few times in order to build some kind of maze for the Monsters to walk in later on. We will also leave an empty area at the top for the Castle:



The Castle

The Castle Cube

Alright, let's create a Castle so that the Monsters have something to destroy. We will begin by selecting **GameObject->3D Object->Cube**from the top menu. We will name it **Castle**, position it at the empty area in our maze and scale it a bit bigger:

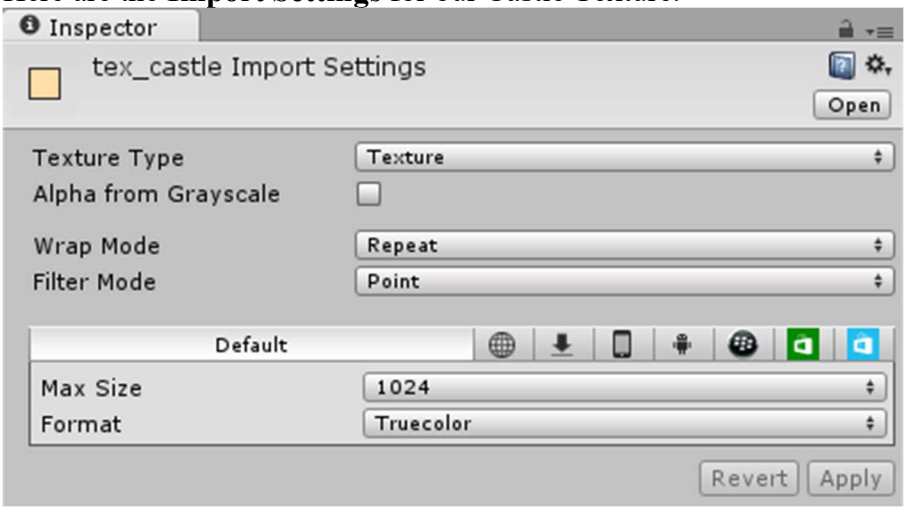


The Castle Texture

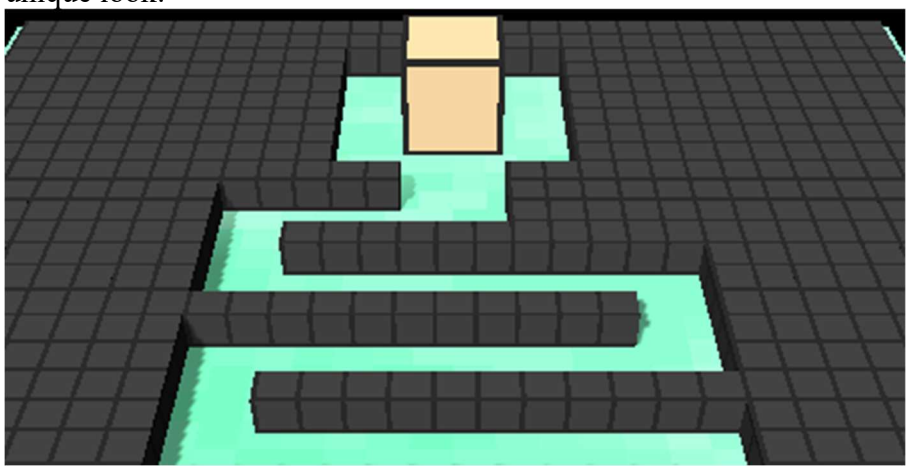
We will also create a very simple 16 x 16 px Texture again:

*Note: right click on the image, select **Save As...** and save it in the project's **Assets/Textures** folder "tex_castle.png"..*

Here are the **Import Settings** for our Castle Texture:



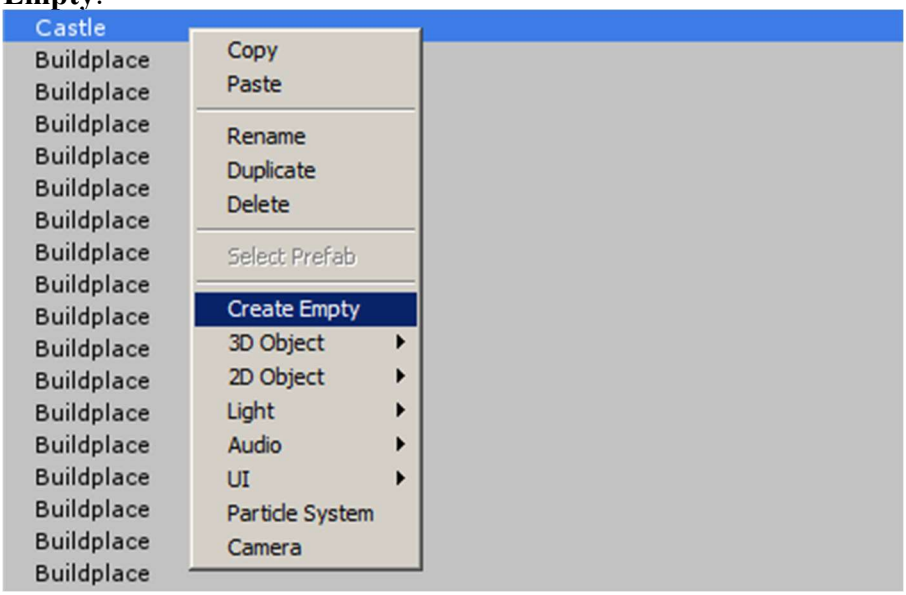
Let's drag the Texture from the **Project Area** onto the Castle to achieve a pretty simple, yet unique look:



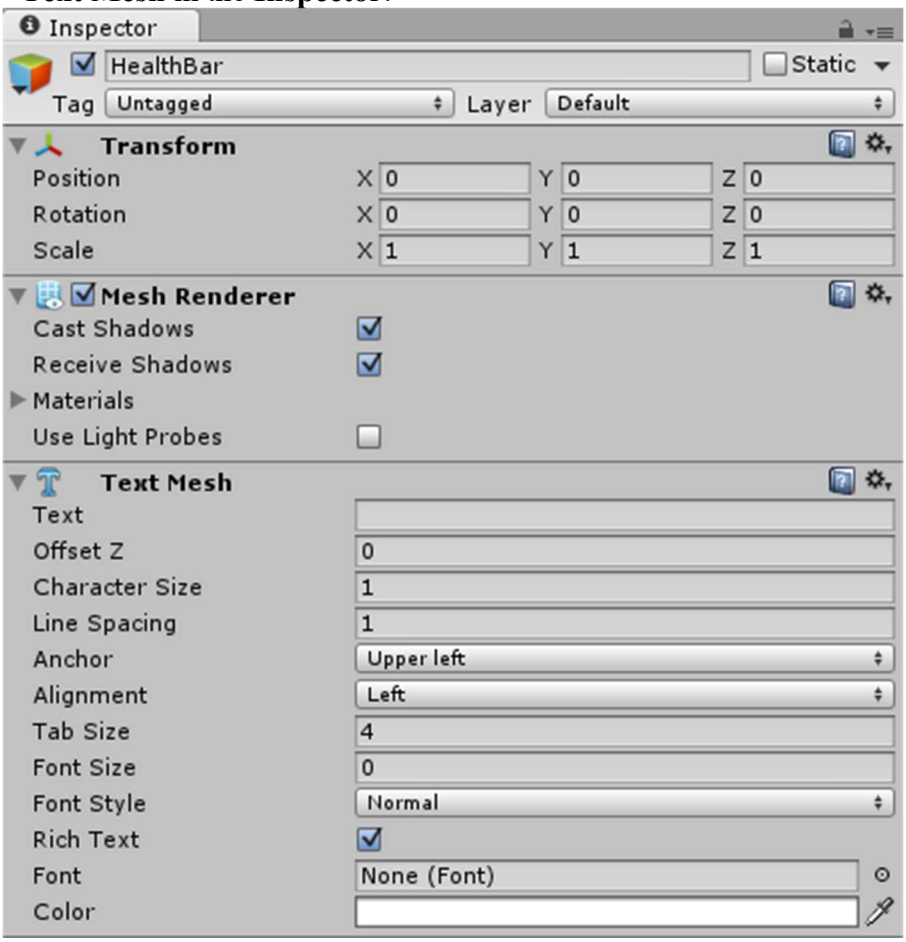
The Health Bar

Alright, let's add some kind of Health Bar to our Castle. There are a lot of different ways to do this, and we will pick the most simple one. We will add a 3D text above the castle that consists of a text like '-' for 1 health, '--' for two health, '---' for three health and so on.

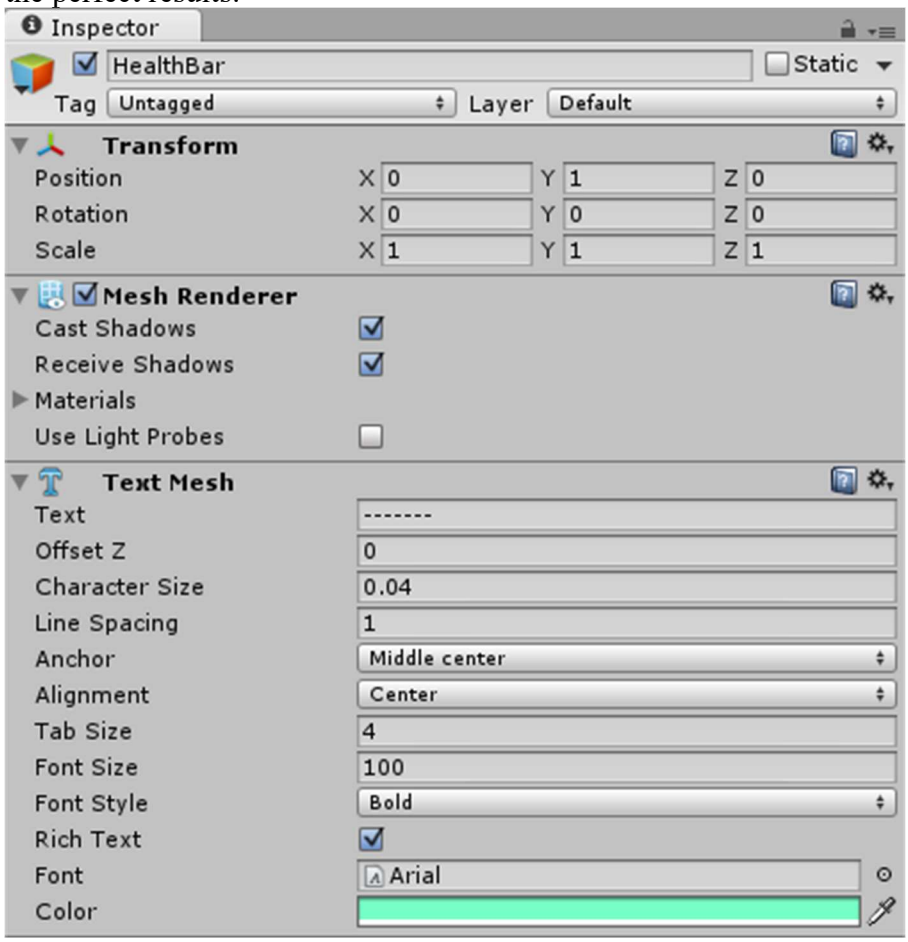
We can add a 3D text to the Castle by right clicking it in the **Hierarchy** and selecting **Create Empty**:



Afterwards we rename the new GameObject to **HealthBar** and select **Add Component->Mesh->Text Mesh** in the **Inspector**:

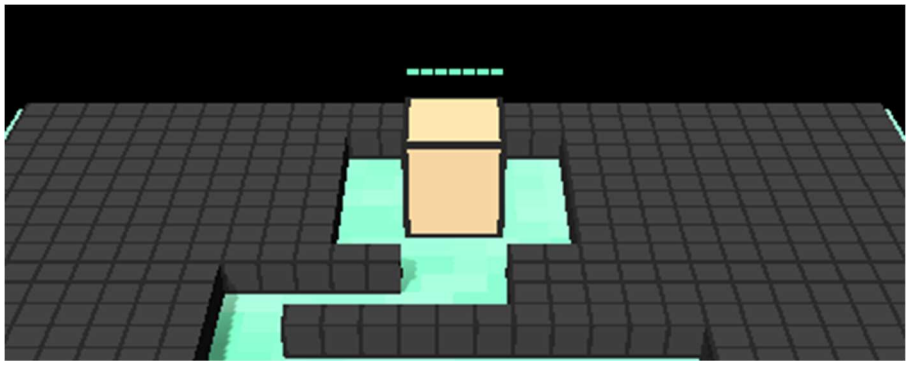


We will position it one unit above the Castle and then modify the TextMesh component to get the perfect results:



*Note: the **Text** property already contains 7 x '-', which equals 7 health. Most of the other properties are changed to modify the font size and sharpness. It's important to also put a **Arial** (or any other font) into the **Font** property. We can do this by clicking the little circle on the right of the Font property. Afterwards Unity shows a list of currently available fonts, where we can select one.*

Here is how it looks in the Scene:



It's really that simple. Now we just need a little Health Script that does exactly three things:

- Return the current health by counting the '-'
- Decrease the current health by removing one '-'
- Make the TextMesh face the Camera at all times

We want the TextMesh to always face the Camera in order to avoid weird angles like this one:

Alright, let's select **Add Component->New Script**, name it **Health** and select **CSharp** for the language. We will also move it into our Scripts folder in the **Project Area** and then open it:

```
using UnityEngine;
using System.Collections;

public class Health : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

First of all we will need to get access to the TextMesh component:

```
using UnityEngine;
using System.Collections;

public class Health : MonoBehaviour {
    // The TextMesh Component
    TextMesh tm;

    // Use this for initialization
    void Start () {
        tm = GetComponent<TextMesh>();
    }

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

    }
}
```

Now we can make it face the Camera at all times by using the **Update** function:

```
using UnityEngine;
using System.Collections;

public class Health : MonoBehaviour {
    // The TextMesh Component
    TextMesh tm;

    // Use this for initialization
    void Start () {
        tm = GetComponent<TextMesh>();
    }

    // Update is called once per frame
    void Update () {
        // Face the Camera
        transform.forward = Camera.main.transform.forward;
    }
}
```

Note: there is no complicated math going on here. All we do is make our GameObject look into the exact same direction that the camera looks at.

Now we can add our **current()** and **decrease()** functions:

```
// Return the current Health by counting the '-'
public int current() {
    return tm.text.Length;
}

// Decrease the current Health by removing one '-'
public void decrease() {
    if (current() > 1)
        tm.text = tm.text.Remove(tm.text.Length - 1);
    else
        Destroy(transform.parent.gameObject);
}
```

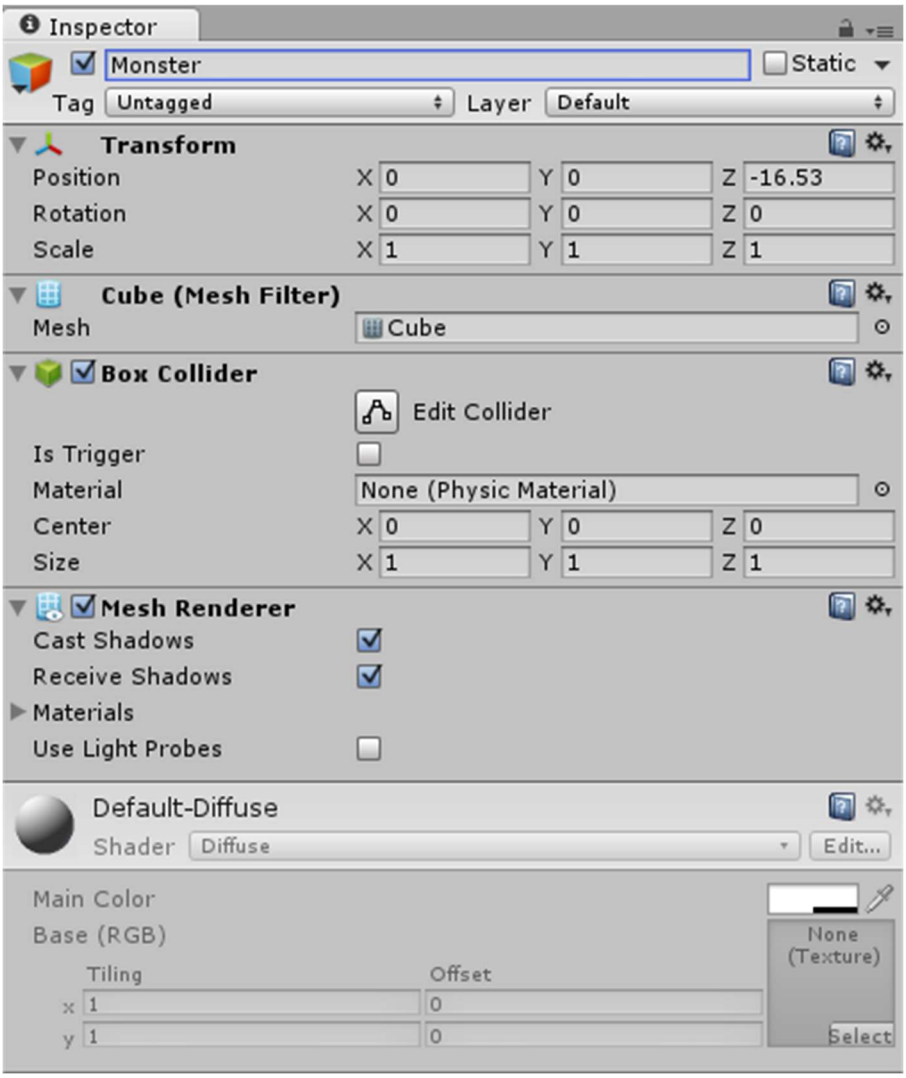
*Note: this is the only thing that looks a bit weird. Usually we would have some kind of integer variable to represent our current health. Instead we made use of the TextMesh's **Text** component that consists of those '-' strings. We also automatically Destroy the Health Bar's parent object (which is the Castle) as soon as the health becomes **0**. We made those functions **public** so that other Scripts can use them.*

That's it, the simplest HealthBar for our game. And the good thing is that we will be able to use it for the Monsters, too.

The Monsters

Let's add a Monster to our game. Usually we would create a Monster with any CAD tool like Blender, Maya or 3DS Max. But since we want to keep things simple, we will use Unity's primitives to create some kind of cube that looks a bit like a monster.

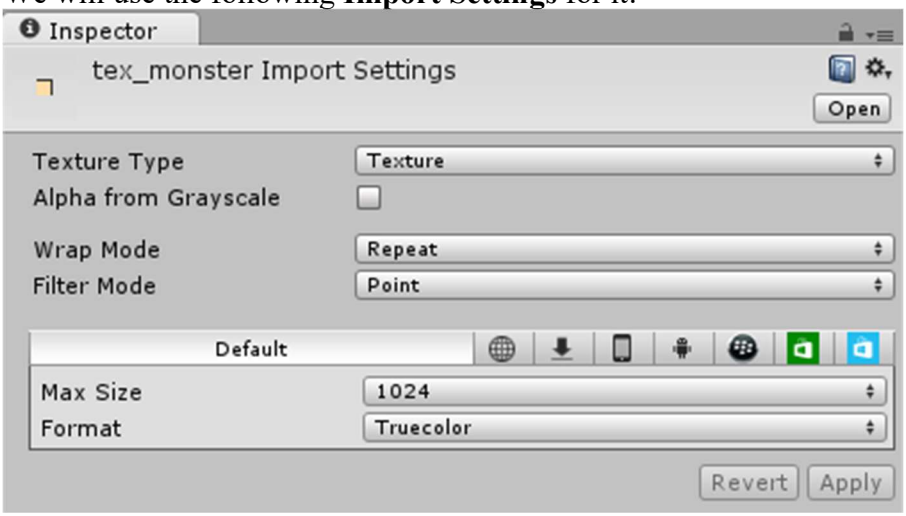
We will begin by selecting **GameObject->3D Object->Cube** from the top menu. We will rename it to **Monster** and check the Collider's **Is Trigger** option so that the Monsters won't collide with each other later on:



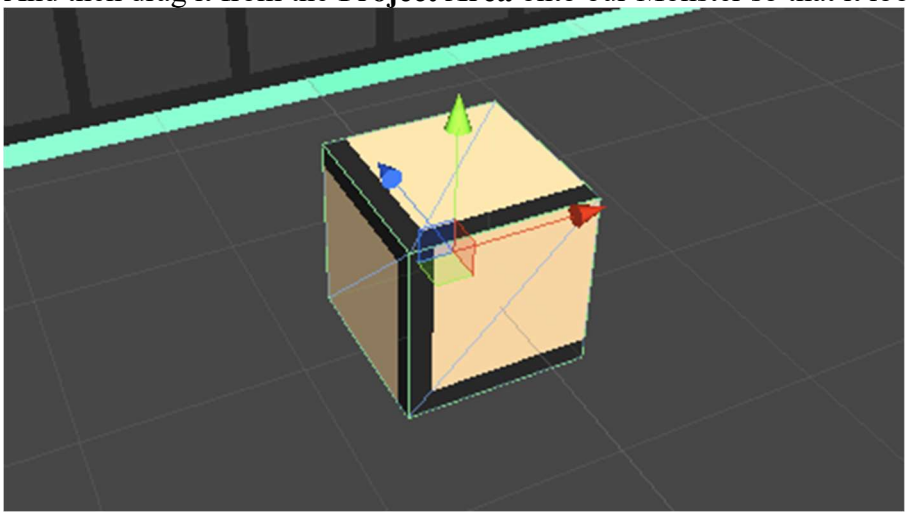
Let's also draw a little 8 x 8 px Texture to give it some color:

*Note: right click on the image, select **Save As...** and save it in the project's **Assets/Textures** folder as "tex_monster.png".*

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

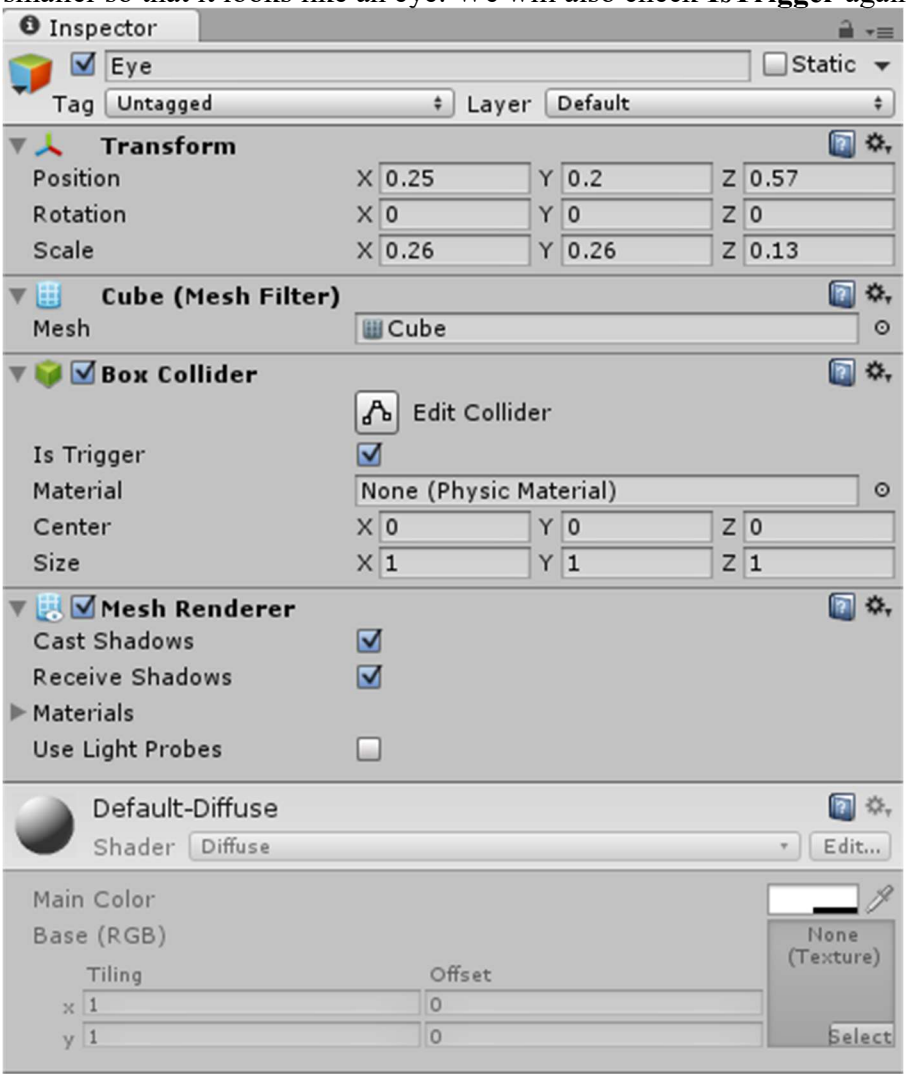


And then drag it from the **Project Area** onto our Monster so that it looks like this:

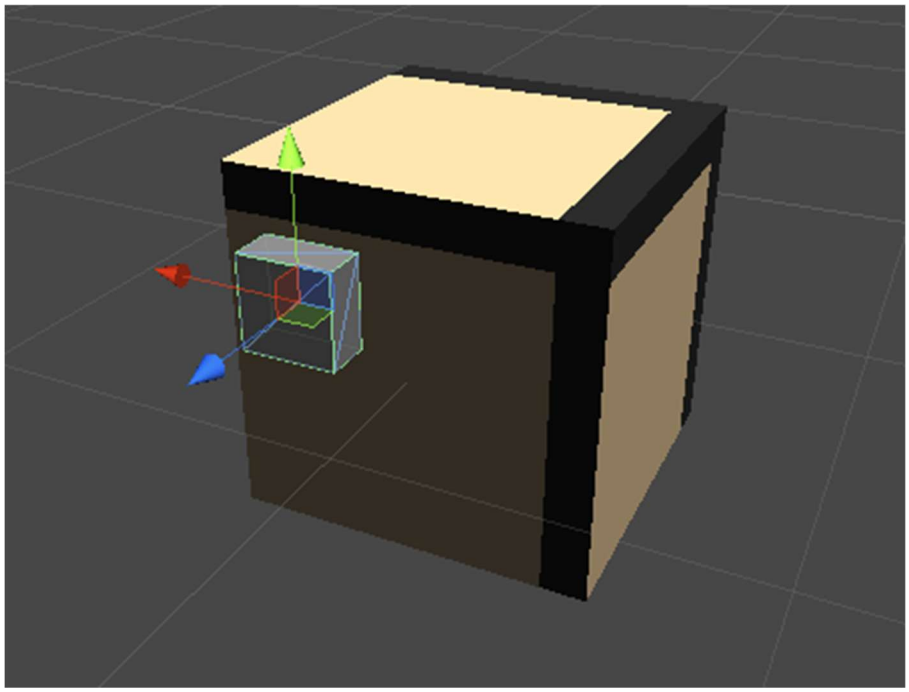


The Details

Let's right click the Monster in the **Hierarchy** and select **3D Object->Cube**. We will then rename the new Cube to **Eye**, position it somewhere at the left center of our Cube and make it smaller so that it looks like an eye. We will also check **IsTrigger** again:



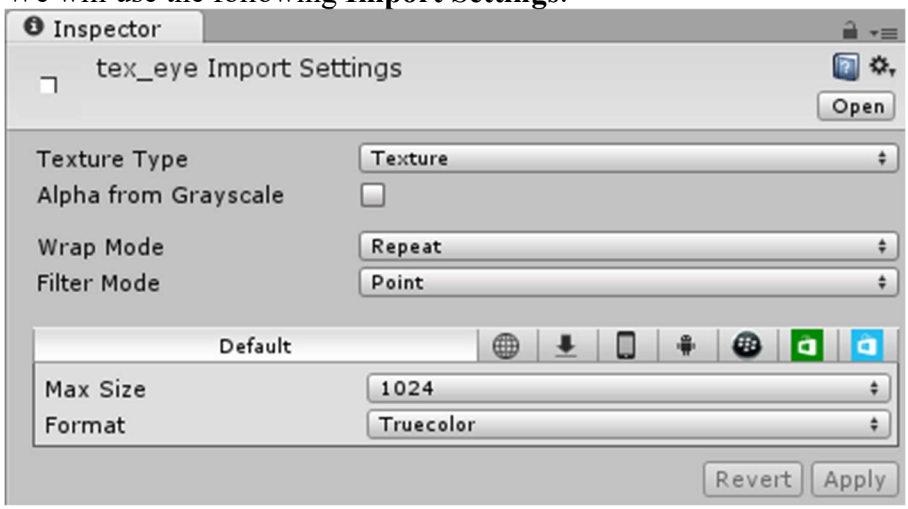
Here is how it looks in the Scene:



Let's also draw a very simple white Texture for the eye:

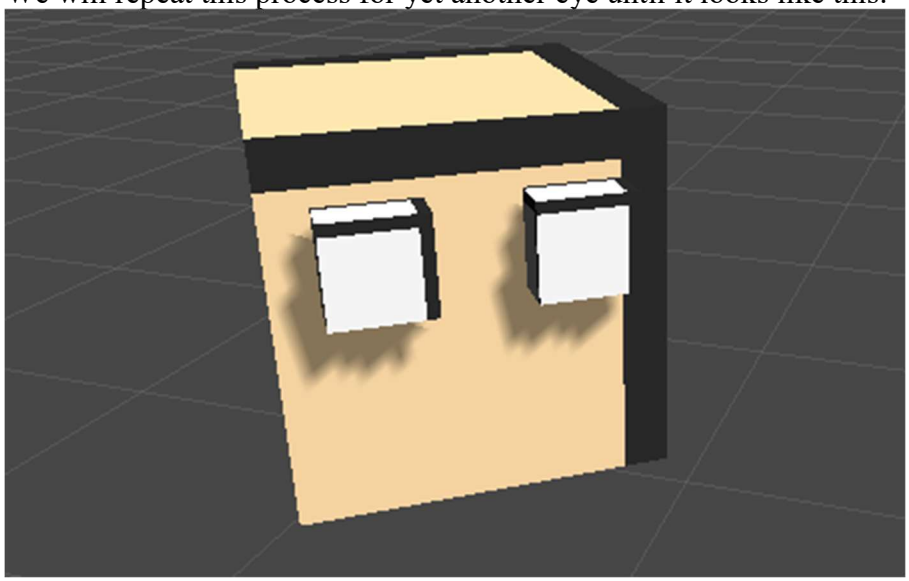
*Note: right click on the image above, select **Save As...** and save it in the project's **Assets/Textures** folder as "tex_eye.png".*

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

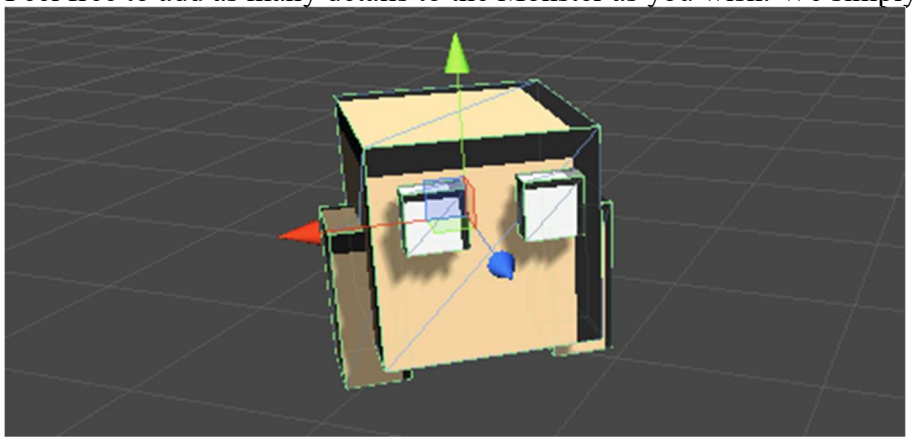


And then drag it onto our Eye GameObject.

We will repeat this process for yet another eye until it looks like this:



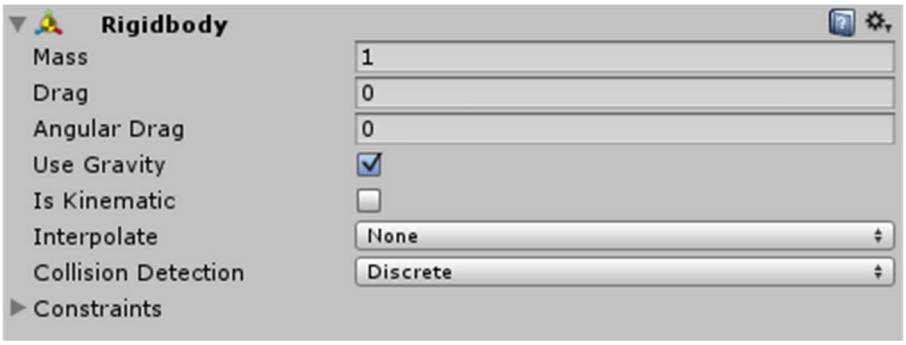
Feel free to add as many details to the Monster as you wish. We simply added two more feet:



Monster Physics

Our Monster already has a Collider, which pretty much makes it part of the physics world already. There is one more thing to add here though. Everything that is supposed to move through the physics world should have a Rigidbody attached to it. A Rigidbody takes care of stuff like gravity, velocity and other forces that make things move.

We can add a Rigidbody by first selecting the **Monster** GameObject *(not the eye or the foot)* and then clicking on **Add Component->Physics->Rigidbody** in the **Inspector**. We will use the following settings for it:



Note: we pretty much used the default settings here.

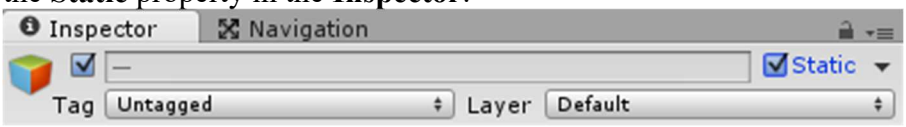
Pathfinding

We want the Monsters to be able to run through the maze and into the Castle. There are exactly two things that we have to do to make this possible:

- Bake a Navigation Mesh *(to tell Unity which area is walkable)*
- Add a NavMeshAgent to the Monster

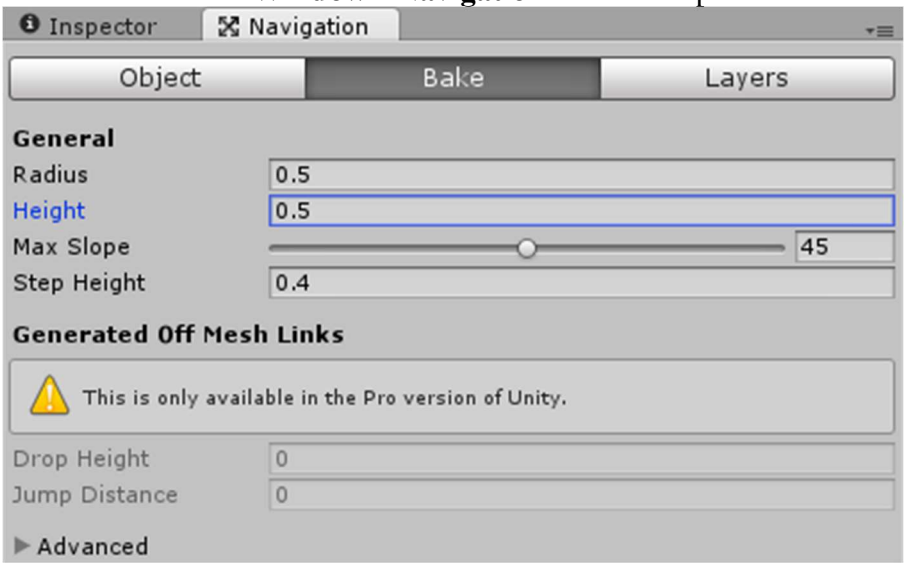
So let's begin by telling Unity which parts of our world are walkable. The good news is that Unity actually figures this out on its own. All we really have to do is tell Unity which parts of our world are **static** *(as in: never moving)*.

Let's select the **Ground** and all the **BuildPlaces** in the **Hierarchy** and then enable the **Static** property in the **Inspector**:

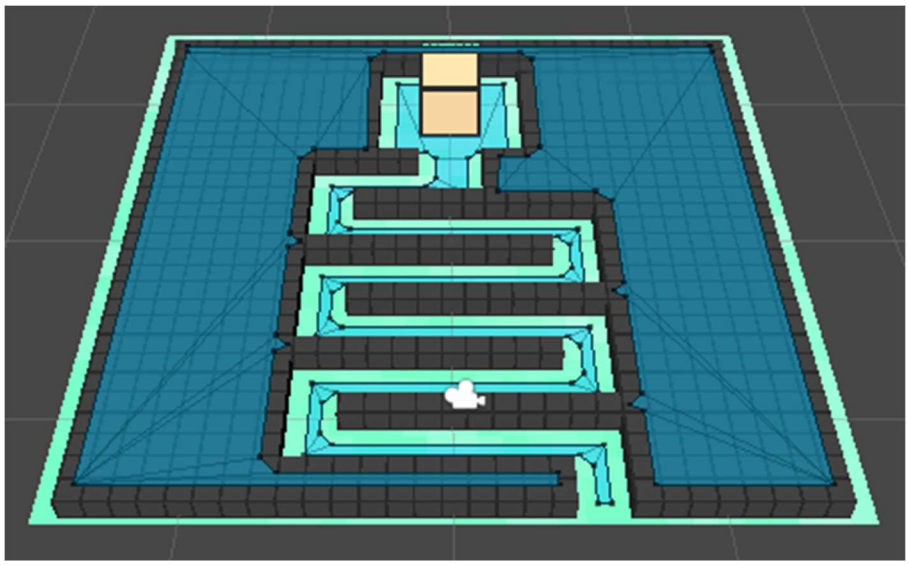


Now Unity knows that those things will never move, hence it will use them to calculate the walkable areas.

Now we can select **Window->Navigation** from the top menu and use the following properties:

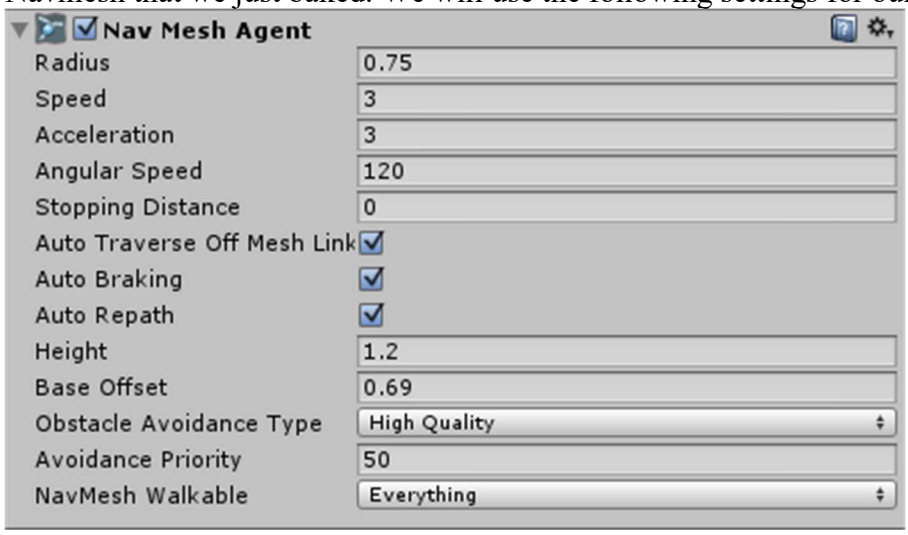


Afterwards we click the **Bake** button at the bottom. After a few seconds we can see how Unity calculated the walkable areas in the Scene:



Note: the important part is the Path through the maze. Unity also thinks that the Buildplaces itself are walkable, but since there are no stairs for the monsters to climb up there, it doesn't really matter.

Alright, now we can select the **Monster** again and click on **Add Component->Navigation->Nav Mesh Agent**. This component is needed for all things that want to walk along the Navmesh that we just baked. We will use the following settings for our agent:



Note: we simply modified the movement speed and the size to fit our monster. Feel free to play around with those settings later on if you are not satisfied with them yet.

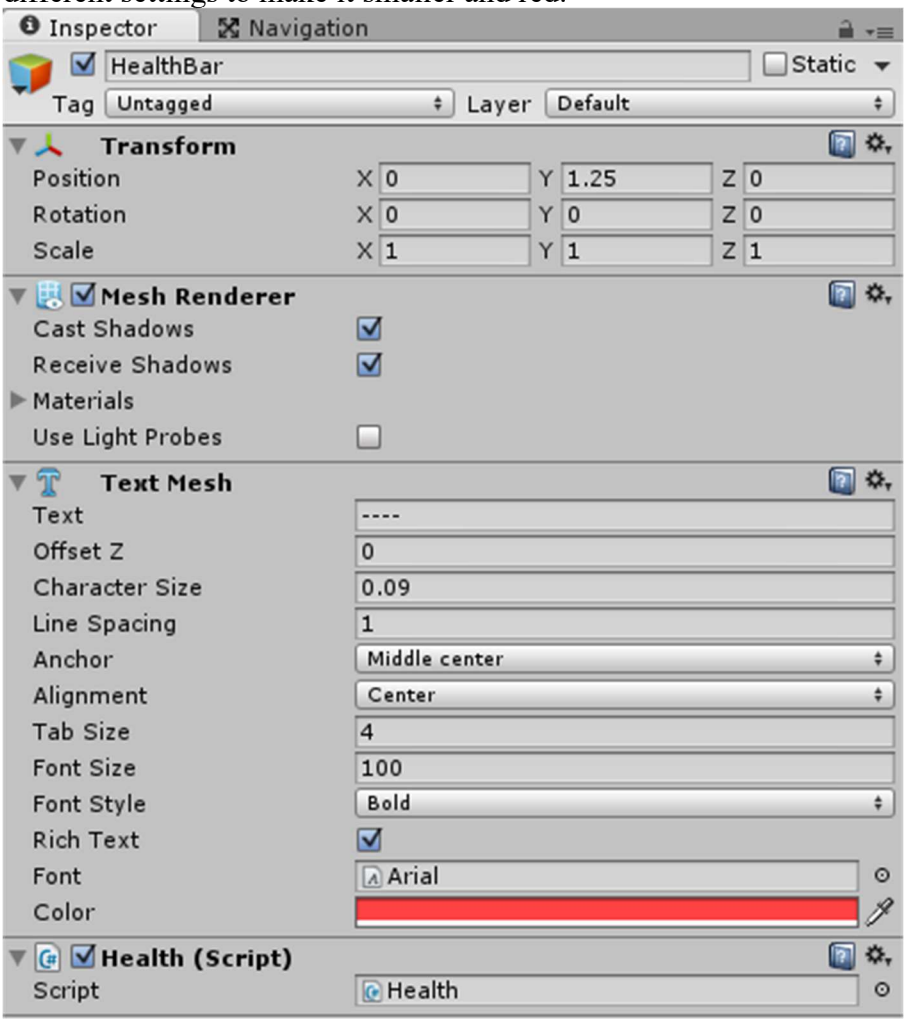
Now that our NavMesh was baked and now that our Monster has a NavMeshAgent, all we have to do to make it move somewhere is something like this:

```
GetComponent<NavMeshAgent>.destination = new Vector3(1, 2, 3);
```

Note: we will do this when we create the Monster Script later.

The Health Bar

Let's also add a Health Bar to our Monster, so that the Towers can attack it later on. We will use the exact same workflow that we used for our Castle's HealthBar, just with some slightly different settings to make it smaller and red:



*Note: we also changed the **Text** property to 4 x '-' so that the monsters only resist 4 tower attacks.*

Here is how the Monster's HealthBar looks in the Scene:

The Monster Script

Okay so as mentioned before, we want to create some kind of Monster Script that makes it move to the Castle and then deal one damage to it as soon as it reached the Castle.

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

```
using UnityEngine;
using System.Collections;

public class Monster : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

Let's remove the **Update** method because we won't need it:

```
using UnityEngine;
using System.Collections;

public class Monster : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }
}
```

We will use the **Start** function to find the Castle GameObject in the Scene and then use the Monster's **NavMeshAgent** to move to it:

```
// Use this for initialization
void Start () {
    // Navigate to Castle
    GameObject castle = GameObject.Find("Castle");
    if (castle)
        GetComponent<NavMeshAgent>().destination = castle.transform.position;
}
```

Now we want to decrease the Castle's Health as soon as the Monster walks into it. Our Monster's Collider has **Is Trigger** enabled, which means that Unity automatically notifies us about all kinds of collisions in the **OnTriggerEnter** function. All we have to do is find out if whatever we collided with was the Castle, and then decrease its health:

```
void OnTriggerEnter(Collider co) {
    // If castle then deal Damage
    if (co.name == "Castle") {
        co.GetComponentInChildren<Health>().decrease();
    }
}
```

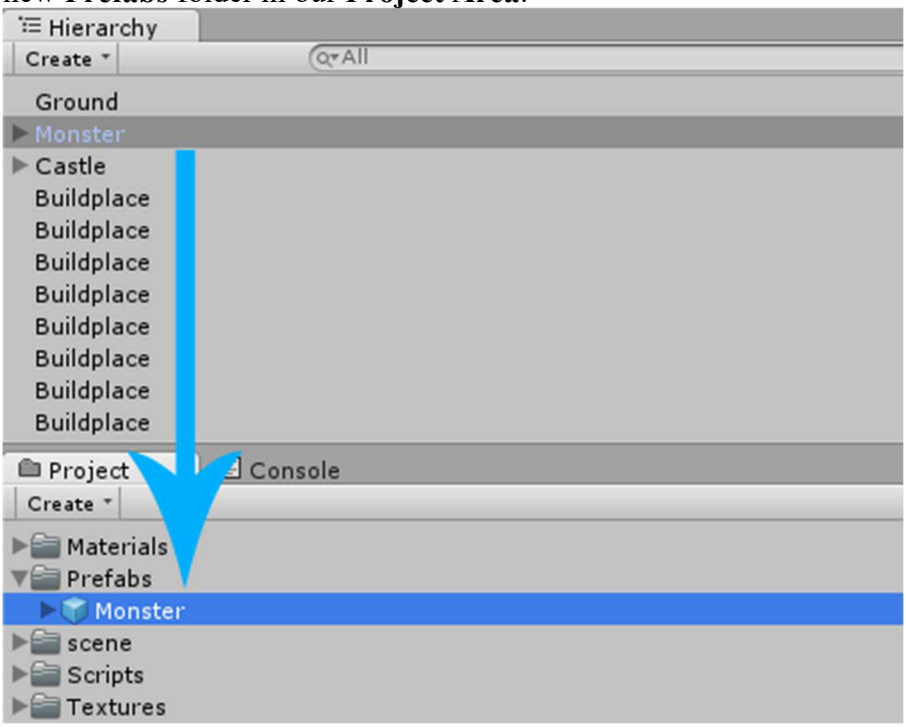
We won't need the Monster anymore after it dealt damage to the Castle, so let's destroy it here, too:

```
void OnTriggerEnter(Collider co) {
    // If castle then deal Damage, destroy self
    if (co.name == "Castle") {
        co.GetComponentInChildren<Health>().decrease();
        Destroy(gameObject);
    }
}
```

And that was our Monster Script, as usual Unity made our lives really easy here.
Note: don't forget to save the Script.

The Monster Prefab

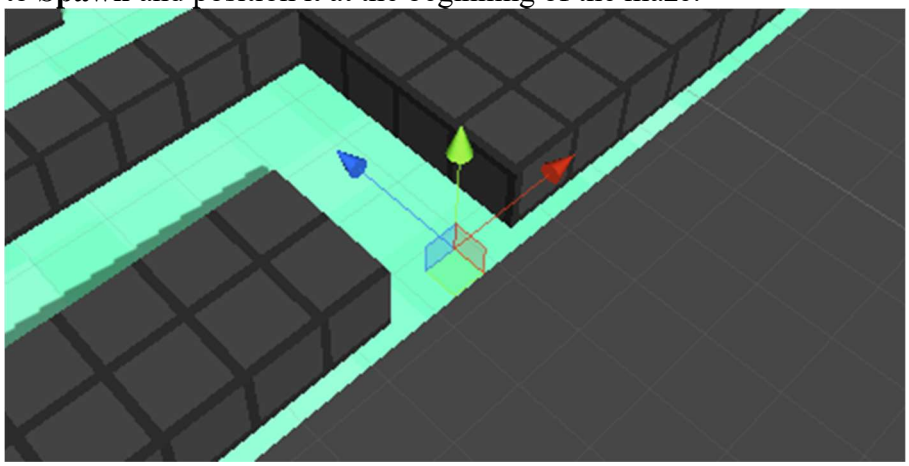
Now we don't want our Monster to be in the game from the very beginning. Instead we want to save it as a Prefab so that we can load it into the game any time we want.
All we have to do to create a Prefab is drag our Monster from the **Hierarchy** into a new **Prefabs** folder in our **Project Area**:



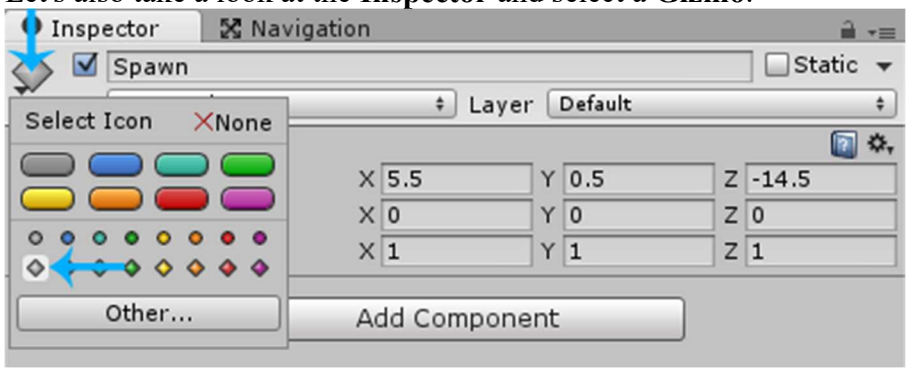
Now we can load it into the Scene any time that we want by either just dragging it in there from the Project Area, or by using the **Instantiate**function in a Script *(which we will do soon)*.
Since we have a Prefab now, we can right click the Monster in the **Hierarchy** and select **Delete**.

The Spawn

Alright, let's create a Spawn that loads new Monsters into the game world every few seconds. We will begin by selecting **GameObject->Create Empty** from the top menu. We will rename it to **Spawn** and position it at the beginning of the maze:

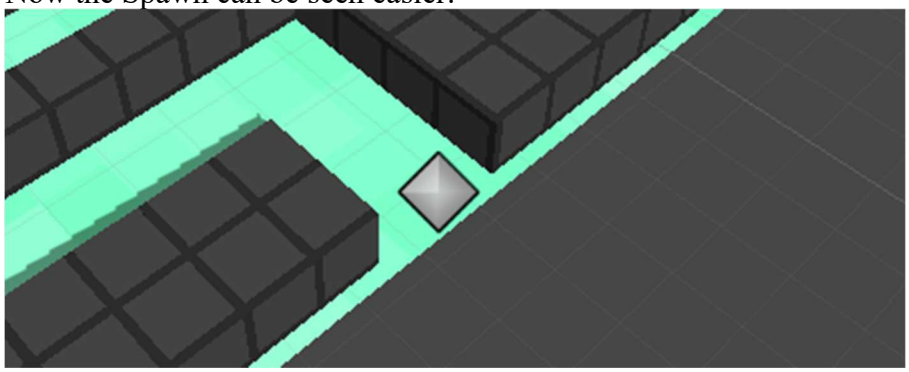


Let's also take a look at the **Inspector** and select a **Gizmo**:



Note: A Gizmo is a visual indicator, it can be used for empty GameObjects so that they can be seen easier. Gizmos are only shown in the Scene view, not in the final game.

Now the Spawn can be seen easier:



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

```
using UnityEngine;
using System.Collections;

public class Spawn : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

We won't need the **Update** method, so let's remove it:

```
using UnityEngine;
using System.Collections;

public class Spawn : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }
}
```

Now we can add a public **MonsterPrefab** variable so that we can specify the Monster that should be spawned:

```
using UnityEngine;
using System.Collections;

public class Spawn : MonoBehaviour {
    // The Monster that should be spawned
    public GameObject monsterPrefab;

    // Use this for initialization
    void Start () {

    }
}
```

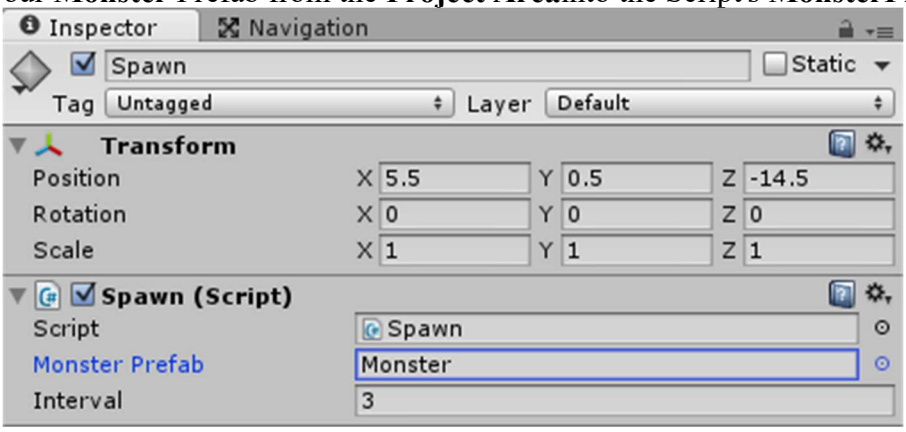
We will add another variable for the **interval**, which is the delay between spawning the monsters:

```
using UnityEngine;
using System.Collections;

public class Spawn : MonoBehaviour {
    // The Monster that should be spawned
    public GameObject monsterPrefab;

    // Spawn Delay in seconds
    public float interval = 3;

    // Use this for initialization
    void Start () {

    }
}
```

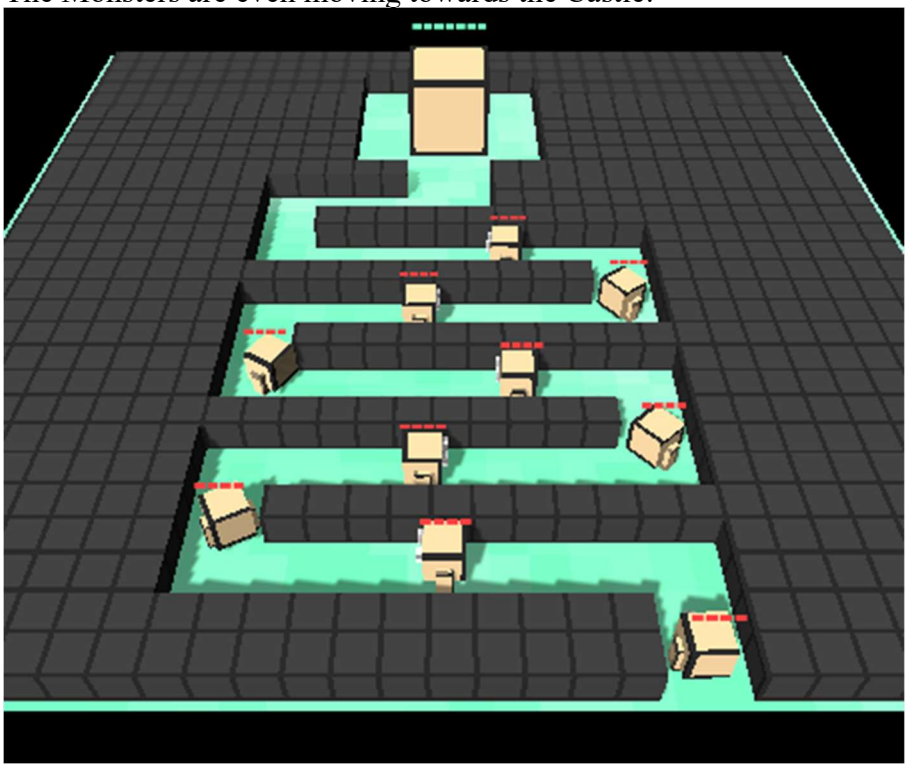
Now all that we have to do is create a function that spawns a new monster, and then call that function every few seconds by using InvokeRepeating:

```
using UnityEngine;
using System.Collections;

public class Spawn : MonoBehaviour {
    // The Monster that should be spawned
    public GameObject monsterPrefab;

    // Spawn Delay in seconds
    public float interval = 3;

    // Use this for initialization
    void Start() {
        InvokeRepeating("SpawnNext", interval, interval);
    }

    void SpawnNext() {
        Instantiate(monsterPrefab, transform.position, Quaternion.identity);
    }
}
```

It's really that easy. We can save the Script and take a look at the **Inspector**, where we can drag our **Monster** Prefab from the **Project Area**into the Script's **MonsterPrefab** slot:

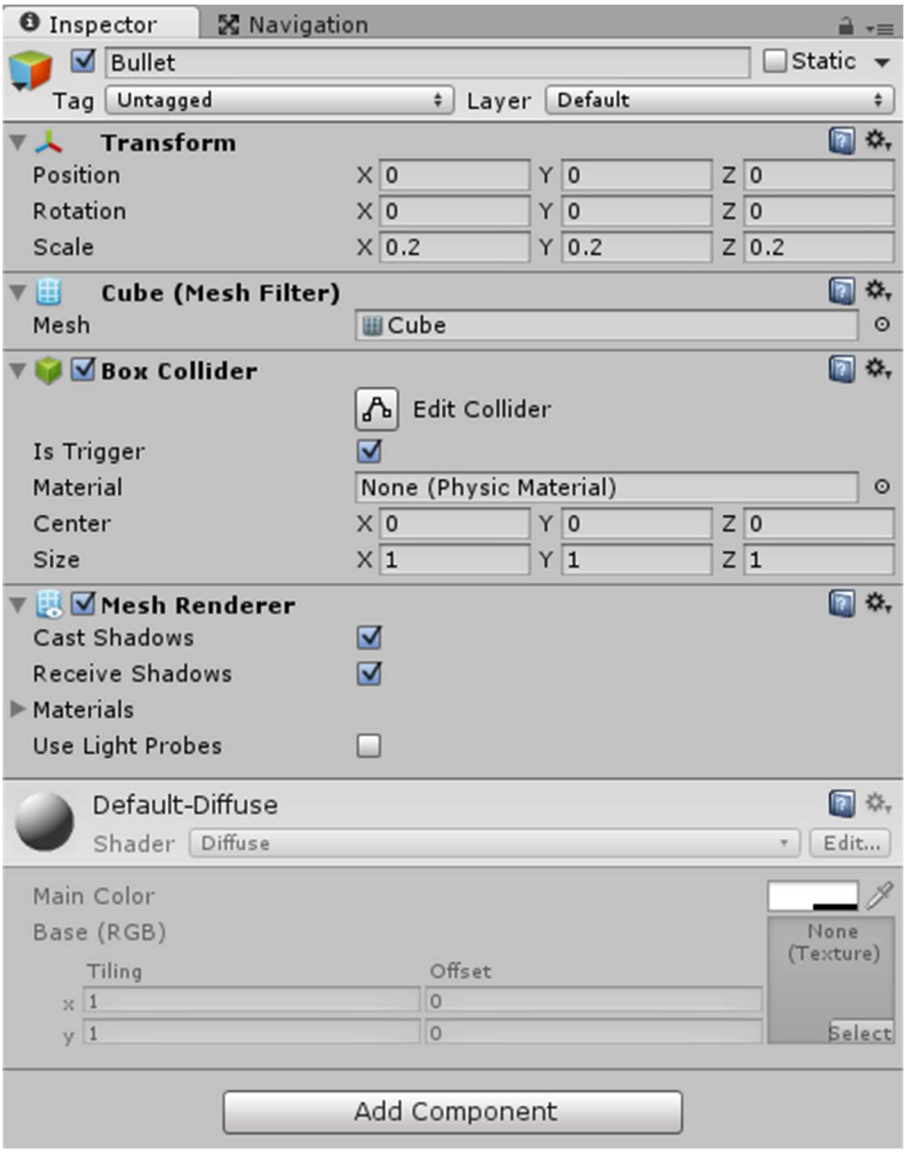


If we press **Play** then we can now see how a new Monster is being spawned every 3 seconds. The Monsters are even moving towards the Castle:



The Bullet

Let's create a simple Bullet Prefab, so that we can create a Tower afterwards. We will begin by selecting **GameObject->3D Object->Cube**from the top menu. We will rename it to **Bullet**, scale it a bit smaller and enable **IsTrigger** again:

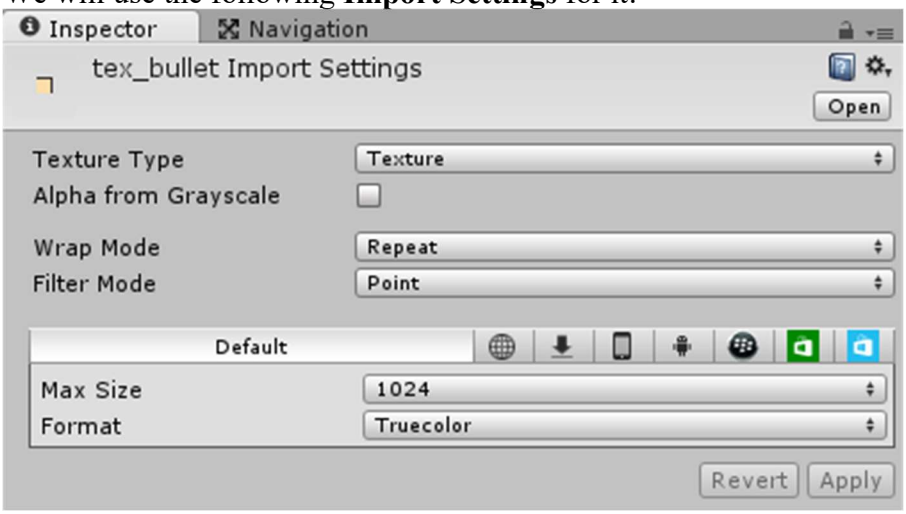


Note: we enabled IsTrigger so that the bullet is able to fly through walls and hit the monster in any case.

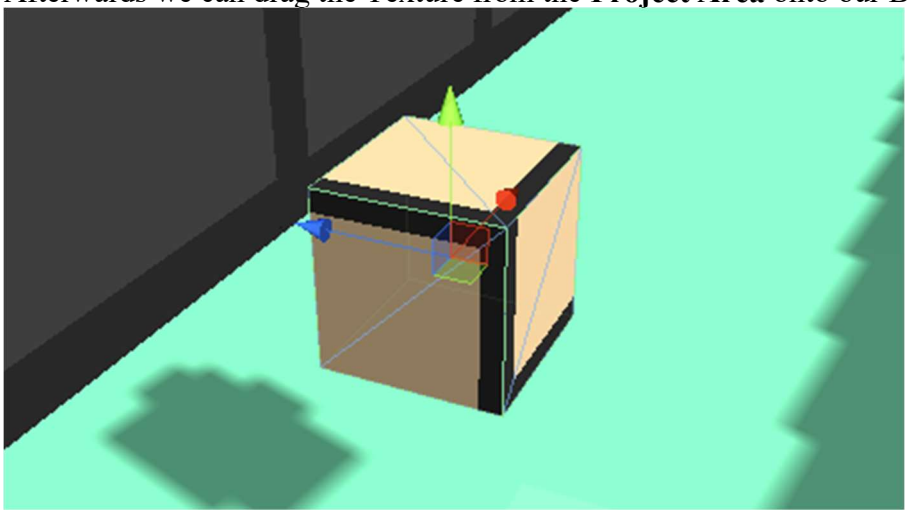
Let's draw a simple 8 x 8 px Texture that we can use for our bullet:

*Note: right click on the image, select **Save As...** and save it in the project's **Assets/Textures** folder as "tex_bullet.png"..*

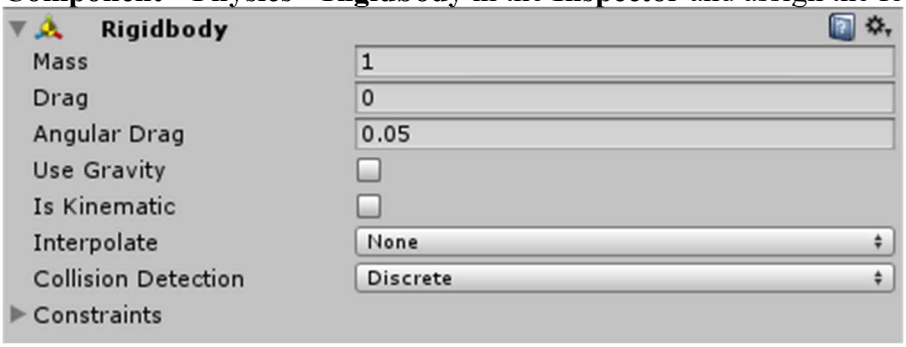
We will use the following **Import Settings** for it:



Afterwards we can drag the Texture from the **Project Area** onto our Bullet:



Alright, the Bullet is supposed to fly through the game world until it hits a Monster. And as we know, everything in the physics world that wants to move needs a Rigidbody. Let's select **Add Component->Physics->Rigidbody** in the **Inspector** and assign the following properties:



Now we can use the Rigidbody to fly to the Monster. Let's select **Add Component->New Script** in the **Inspector**, name it **Bullet** and select **CSharp** for the language. We will then move the Script into the **Scripts**folder and open it:

```
using UnityEngine;
using System.Collections;

public class Bullet : 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 **FixedUpdate**, which is kinda like Update, just that it is being called in the same interval where physics calculations are made. And since the Bullet is part of the physics world, we will use **FixedUpdate**:

```
using UnityEngine;
using System.Collections;

public class Bullet : MonoBehaviour {

    void FixedUpdate() {

    }
}
```

The Bullet will need a **target** to fly to, and a **speed** variable:

```
using UnityEngine;
using System.Collections;

public class Bullet : MonoBehaviour {
    // Speed
    public float speed = 10;

    // Target (set by Tower)
    public Transform target;

    void FixedUpdate() {

    }
}
```

Note: the target will be the Monster that it wants to hit. The target will be set by the Tower that fires the bullet.

Let's modify our **FixedUpdate** function to make the Bullet fly towards the target. We will simply use the Rigidbody's velocity property. The velocity is defined as the movement direction multiplied by the speed, so let's do just that:

```
void FixedUpdate() {
    // Fly towards the target
    Vector3 dir = target.position - transform.position;
    GetComponent<Rigidbody>().velocity = dir.normalized * speed;
}
```

*Note: we calculated the direction by subtracting the current position from the Monster's position, which is basic vector math. Afterwards we multiply that direction by the speed. We also used the **normalized** direction to make sure that it has the length of one.*

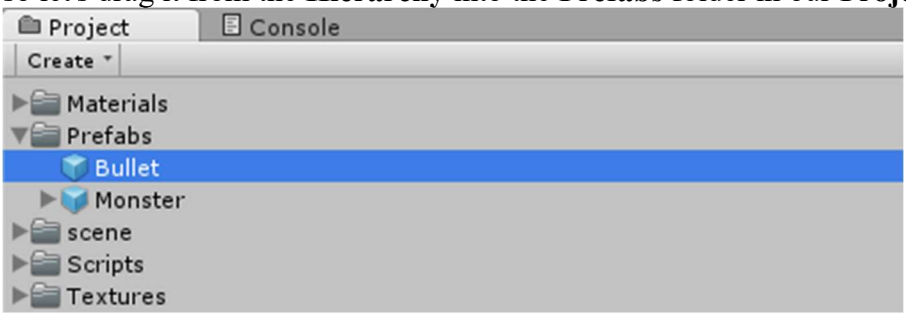
Now it might happen that a Monster runs into the Castle and then dies, while a Bullet still tries to reach it. Let's prepare for that:

```
void FixedUpdate() {
    // Still has a Target?
    if (target) {
        // Fly towards the target
        Vector3 dir = target.position - transform.position;
        GetComponent<Rigidbody>().velocity = dir.normalized * speed;
    } else {
        // Otherwise destroy self
        Destroy(gameObject);
    }
}
```

Great, there is one more thing to do here. We want the Bullet to deal damage at the Monster as soon as it reached the Monster. The Bullet should then destroy itself after dealing damage. We will simply make use of Unity's **OnTriggerEnter** function to know when it reached the Monster:

```
void OnTriggerEnter(Collider co) {
    Health health = co.GetComponentInChildren<Health>();
    if (health) {
        health.decrease();
        Destroy(gameObject);
    }
}
```

And that's our Bullet. Yet again we don't actually want it to be in the Scene from the beginning, so let's drag it from the **Hierarchy** into the **Prefabs** folder in our **Project Area**:

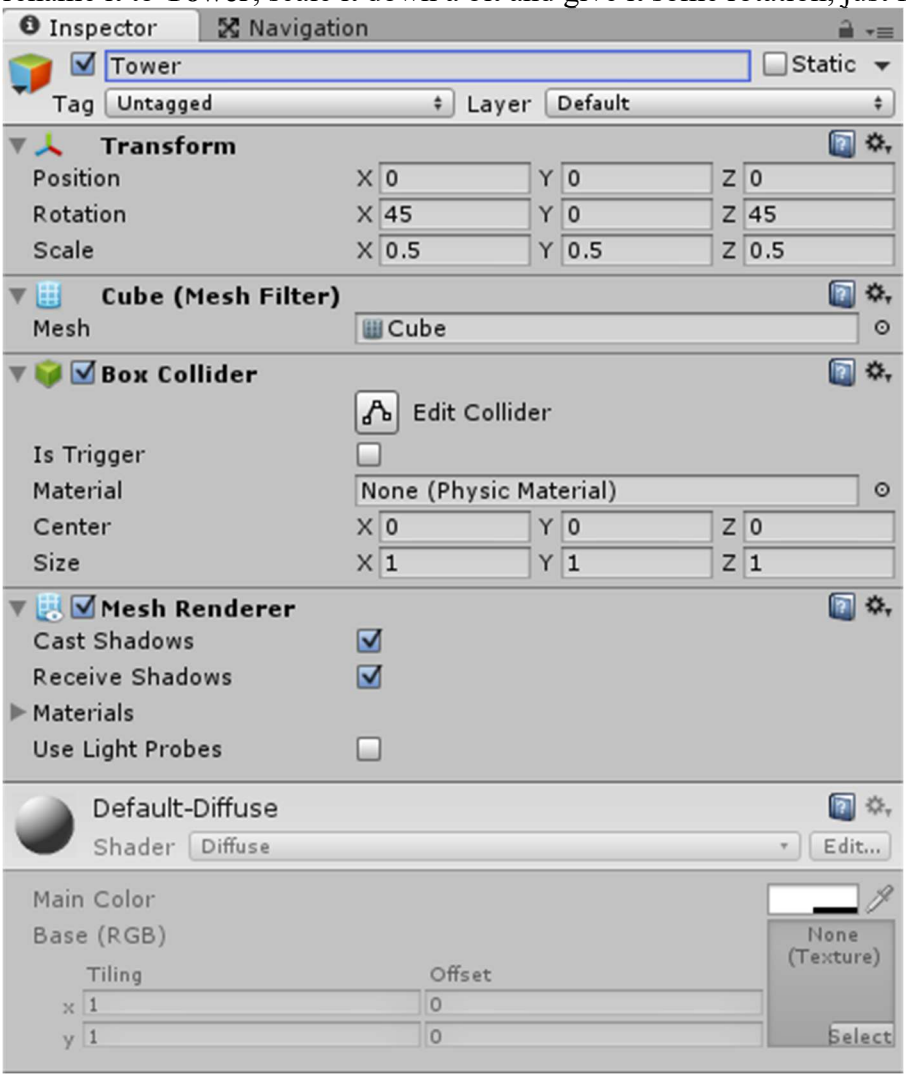


Afterwards we can delete it from the **Hierarchy**, because we don't need it just yet.

The Tower

We already created many Buildplaces where we can put Towers onto, and now it's time to create a Tower.

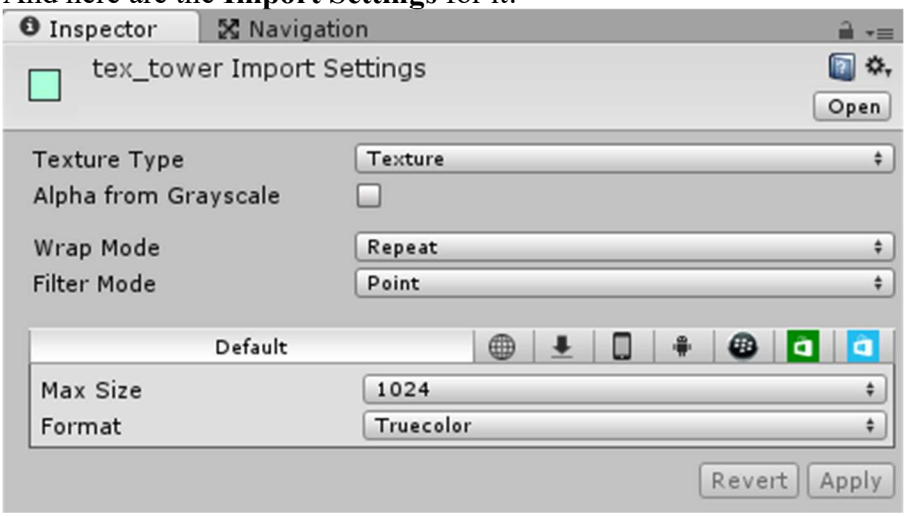
We will begin by selecting **GameObject->3D Object->Cube** from the top menu again. We will rename it to **Tower**, scale it down a bit and give it some rotation, just for the better looks:



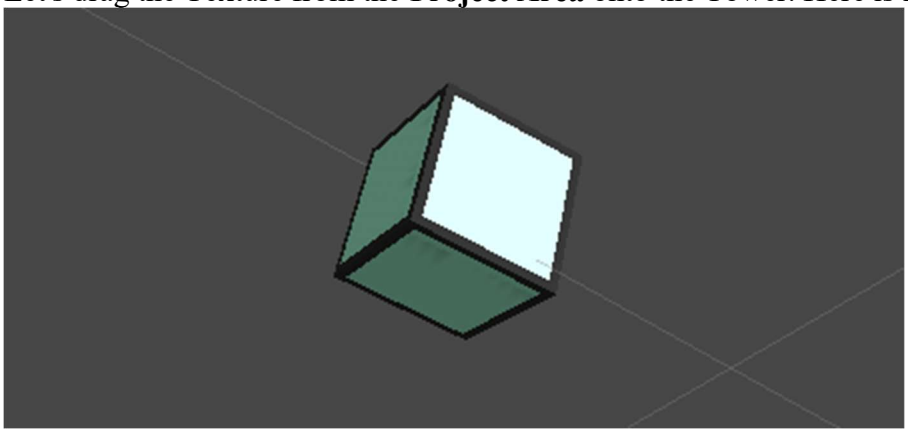
We will use the following 16 x 16 px Texture for our Tower:

*Note: right click on the image, select **Save As...** and save it in the project's **Assets/Textures** folder "tex_tower.png".*

And here are the **Import Settings** for it:

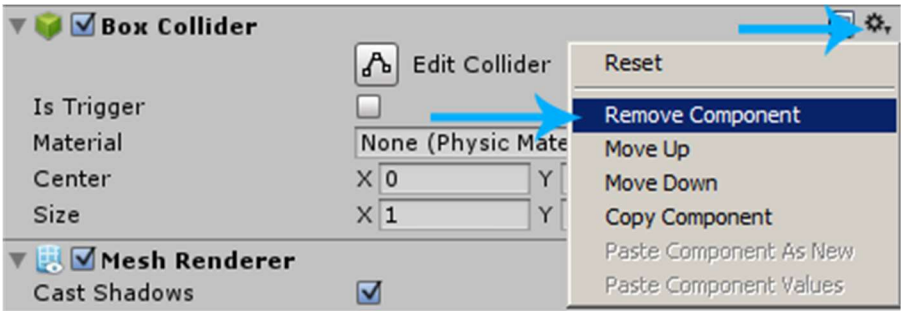


Let's drag the Texture from the **Project Area** onto the Tower. Here is how it looks now:

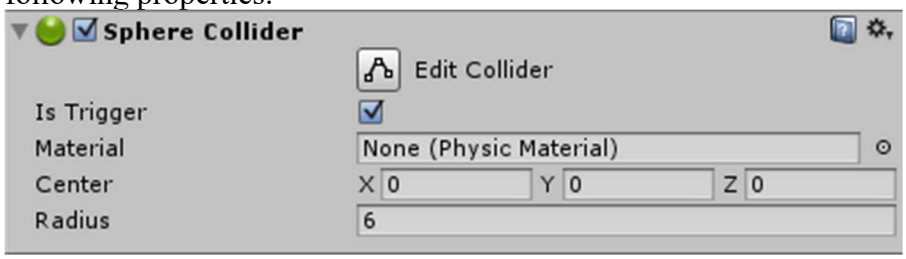


Our Tower model is finished. Now we still have to add the logic that finds a monster that is close to the Tower, and then shoots a Bullet at it. There are many different ways to do this. The obvious way would be a Script that finds all Monsters and then shoots a Bullet at the closest one. While this would work just fine, it would also be really computationally expensive.

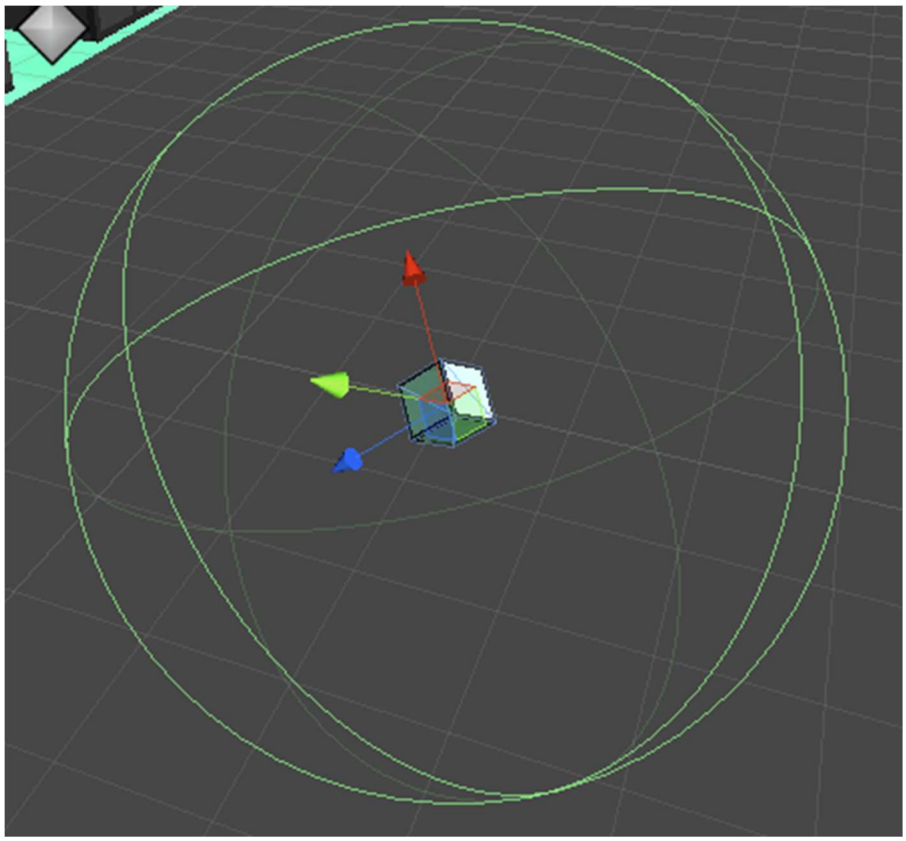
It's much smarter to add a big sphere Trigger around the Tower and use
Unity's **OnTriggerEnter** function. This way Unity will automatically notify us whenever a Monster walks into the Tower's 'awareness' area.
Let's remove the Tower's Box Collider first:



Afterwards we will select **Add Component->Physics->Sphere Collider** and assign the following properties:



If we take a look at the **Scene**, then we can see the Sphere. This is the area in which the Tower sees and attacks the Monsters:



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

```
using UnityEngine;
using System.Collections;

public class Tower : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

We won't need the **Start** function, so let's remove it:

```
using UnityEngine;
using System.Collections;

public class Tower : MonoBehaviour {

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

    }
}
```

Let's add a public **BulletPrefab** variable that specifies which Bullet Prefab to use:

```
using UnityEngine;
using System.Collections;

public class Tower : MonoBehaviour {
    // The Bullet
    public GameObject bulletPrefab;

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

    }
}
```

Now we can use the **OnTriggerEnter** function and find out if whatever walked into the Trigger was a Monster, in which case we fire a Bullet at it:

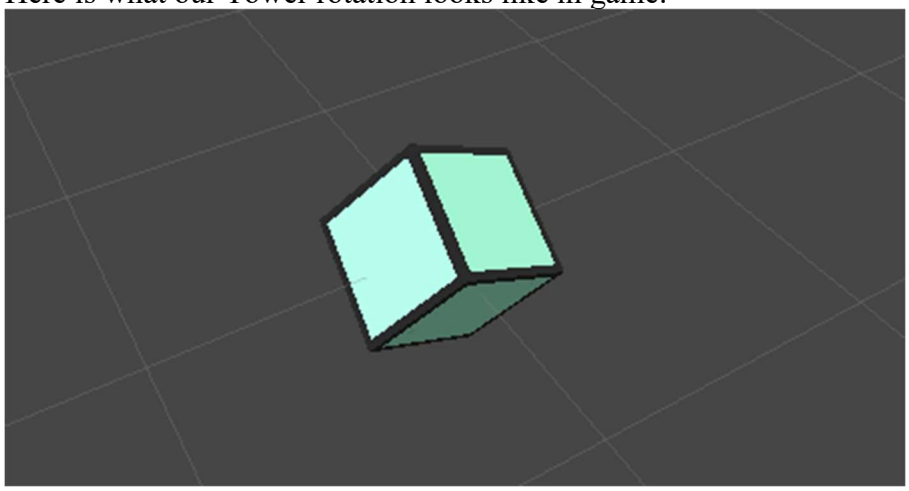
```
void OnTriggerEnter(Collider co) {
    // Was it a Monster? Then Shoot it
    if (co.GetComponent<Monster>()) {
        GameObject g = (GameObject)Instantiate(bulletPrefab,
transform.position, Quaternion.identity);
        g.GetComponent<Bullet>().target = co.transform;
    }
}
```

*Note: we use **GetComponent** to find out if whatever walked into the Trigger has a **Monster** Script attached to it. If this is the case then we use **Instantiate**to load the Bullet into the game world at the Tower's current position and with the default rotation (Quaternion.identity). Afterwards we access the Bullet component and set the **target** to the Monster.*

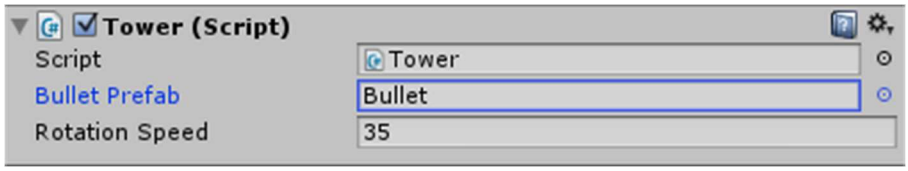
Let's also use our **Update** function to add a very simple rotation to our Tower, just so that it looks better:

```
using UnityEngine;
using System.Collections;

public class Tower : MonoBehaviour {
    // The Bullet
    public GameObject bulletPrefab;

    // Rotation Speed
    public float rotationSpeed = 35;

    void Update() {
        transform.Rotate(Vector3.up * Time.deltaTime * rotationSpeed,
Space.World);
    }

    void OnTriggerEnter(Collider co) {
        // Was it a Monster? Then Shoot it
        if (co.GetComponent<Monster>()) {
            GameObject g = (GameObject)Instantiate(bulletPrefab,
transform.position, Quaternion.identity);
            g.GetComponent<Bullet>().target = co.transform;
        }
    }
}
```

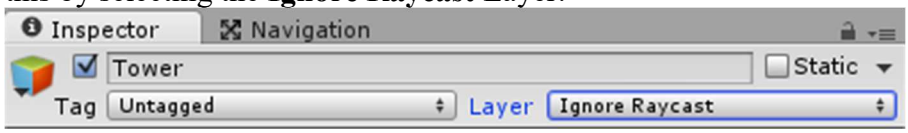
Here is what our Tower rotation looks like in game:



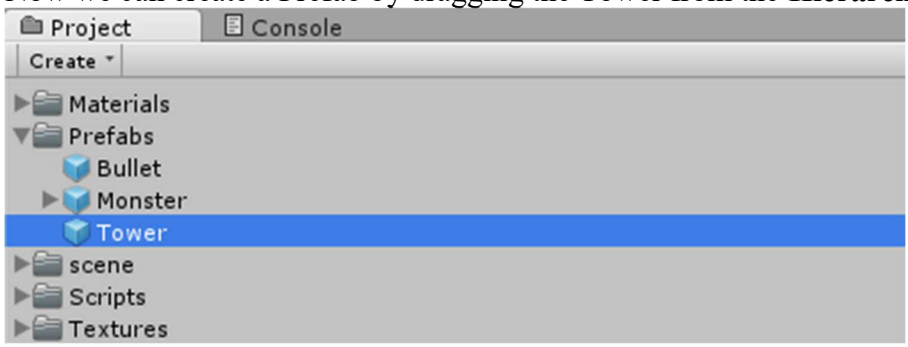
Let's not forget to drag our Bullet Prefab from the **Project Area** into the Tower's **Bullet Prefab** slot:



There is one little modification that's left to do here. The Tower has a pretty big Sphere Collider wrapped around it. This Collider would overlap a Buildplace's Collider, preventing the **OnMouseUpAsButton** function from being called. All we have to do to make it work properly is tell Unity to ignore the Tower's Collider when checking for mouseclicks. We can do this by selecting the **Ignore Raycast** Layer:

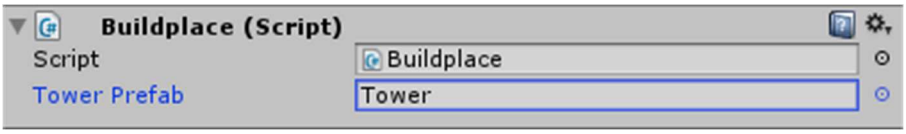


Now we can create a Prefab by dragging the Tower from the **Hierarchy** into our **Prefabs** folder:

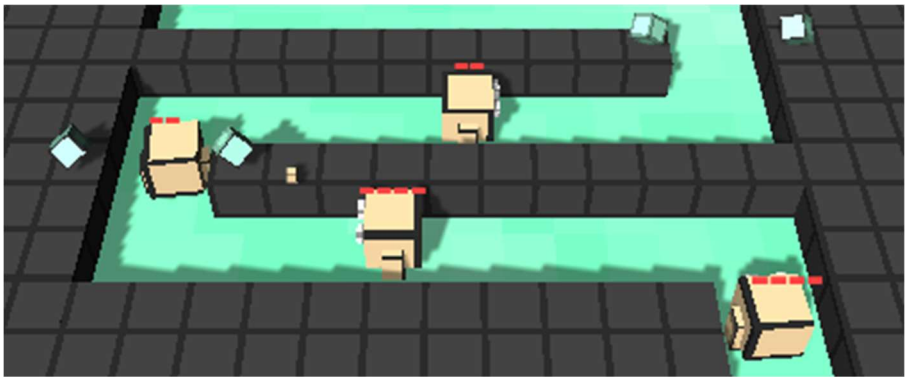


Afterwards we can delete it from the **Hierarchy**.

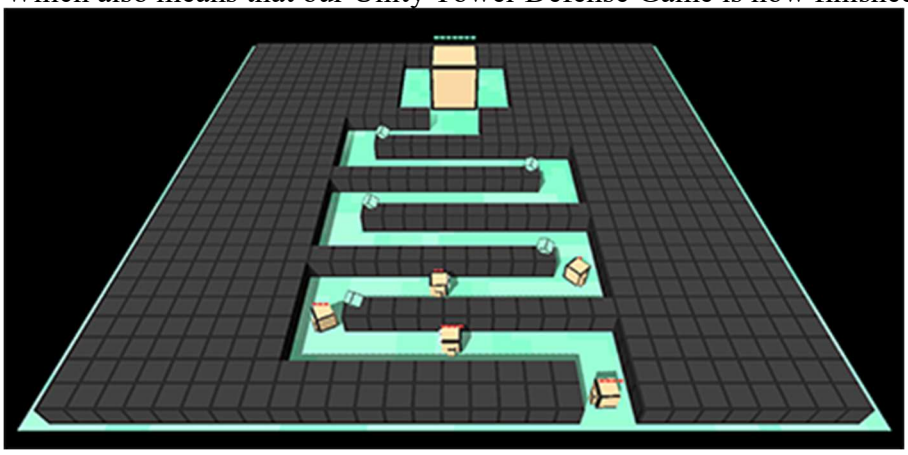
Let's select all the Buildplaces in the **Hierarchy**, take a look at the **Inspector** and then drag our Tower Prefab into the **Tower Prefab** slot:



If we press **Play** then we can now build Towers by clicking on the Buildplaces. The Towers will even attack attack the Monsters:



Which also means that our Unity Tower Defense Game is now finished:



Summary

This was our very long Unity Tower Defense Tutorial. Now we challenge you to add more stuff to the game to make it even more fun! Here are some ideas for things you could add:

- A Score
- Resources/Gold
- Different Monsters
- Different Towers
- Different Bullets
- Different Levels
- Background Music
- Better 3D Models with Animations
- A Menu
- Savegames
- And more...