# 1 Basics of Scene Construction

Introduction

In this chapter, we are going to learn about the basic objects that construct a 3D scene, most important properties of these objects, and relations between these objects. If you are already familiar with 3D engines or 3D design software, you might find the subjects of this chapter familiar; since there are similarities between Unity's way of constructing 3D scenes and other 3D software/engines. In this case, you can safely skip this chapter without worrying about missing important topics. In this chapter, we are not going to cover advanced subjects regarding scene construction, but rather stick to basics that allow us to carry on in our journey towards the development of a 3D computer game.

After completing this chapter, you are expected to:

- Be able to construct a scene using basic 3D shapes.
- Understand the properties of objects in 3D scene (position, rotation, scale).
- Use relations between different objects in 3D space and their effects on the objects to accelerate and facilitate scene construction.
- Understand and use rendering properties (textures, materials, shaders) to enrich your 3D objects in the scene.
- Use different types of lights and understand their properties.
- Use camera to render the scene for the player.
- Write simple scripts that modify the properties of the objects at run time to achieve desired effects.

## 1.1 Basic shapes and their properties

In addition to its capability of importing 3D models from most known 3D design software, Unity provides us with a collection of game objects that represent basic 3D shapes. These shapes include cube, sphere, plane, cylinder, and many others. These objects make it possible to construct a basic scene and interact with it. Scene construction is done by simply adding a number of these shapes and modifying their properties; such as position, rotation, and scale.

To add a new game object to the scene, go to *Game Object* menu, then select *Create Other*. You'll find the basic shapes in the third section of the menu starting from *Cube* and ending with *Quad*

Once you add a new shape game object to the scene, it appears in the scene window and the hierarchy. If you can't see the object in the scene, you can simply select it from the hierarchy and then press F key on the keyboard while the mouse pointer is inside the scene window.

Download free eBooks at bookboon.com

Initially, the hierarchy contains only one object, which is the main camera. This camera is responsible for rendering the scene for the player. In other words, it is the player eye on the game world. Let's now start with a small scene that consists of a number basic shapes. Try to construct a scene similar to the one in Illustration 1 by yourself. If you find this difficult, you can follow the steps after the Illustration.

To modify the position, rotation, or scale of a game object; use the buttons at the top left of Unity's main window. Alternatively, you can use the properties of *Transform* component in the inspector window (see Illustration 2)

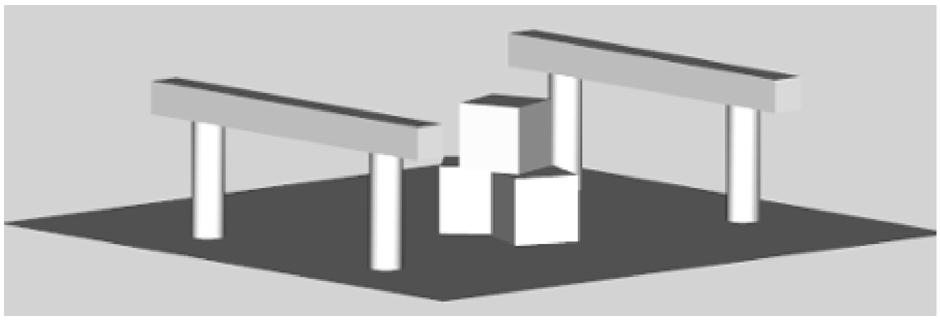


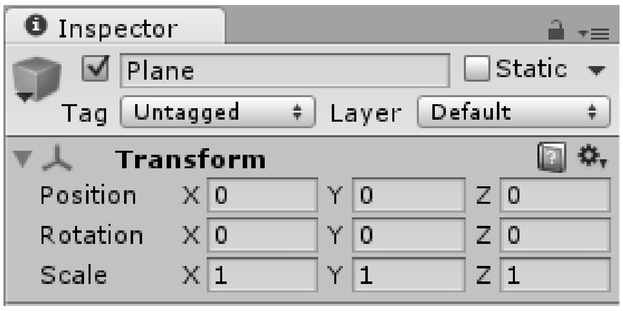
**Illustration 1:** Simple scene constructed using the basic 3D shapes provided by Unity

To construct the scene we see in Illustration 1, we need first to know the type and the properties of each shape we are going to add. The values of *position*, *rotation*, and *scale* are determined by the three axes of the 3D space, which are x (+right, -left), y (+up, -down), and z (+inside screen, -outside screen). The 3D Coordinate system used in Unity follows the left-hand rule. To remember this rule, hold your left hand with your index pointing forward, your thumb pointing up, and your middle pointing right. Your three fingers represent the positive directions of the axes in the 3D space, where the middle finger represents the x-axis, the thumb represents the y-axis, and the index represents the z-axis.

Follow these steps to construct a scene similar to Illustration 1:

1. Create a plane and position it at the center of the 3D space (0, 0, 0). The plane is a 2D shape that covers a 10*10 area on the xz plane, supposing that we use the default scale (1, 1, 1). This plane represents the ground in our scene.
2. Create 4 cylinders and position them at 1 on the y axis, so they sit on the top of the ground plane. Now we need to distribute them uniformly around the origin. For example, we can use the values of (2, 3.5), (-2, 3.5), (-2, -3.5), and (2, -3.5) as the values of (x, z) position for the each one of these cylinders. Finally, scale the cylinders to 0.5 on x and z axes to make them thinner. By completing this we have successfully added the pillars of the arcs to the scene.
3. Now we need to add the two bars at the top of the arcs. These bars are two cubes positioned at (0, 2, 3.5) and (0, 2, -3.5). To extend these cubes, scale them to 6 on x axis and to 0.5 on both y and z axes.

4. Finally, add 3 cubes and position them near the center of the scene by using the 3D gizmo, and rotate them with different angles. Set the y position for two cubes to 0.5, and the third to 1.5, which will make it sit on top of the other two. By adding these boxes we have completed the construction of our simple scene.



Inspector

Plane ☑ · Static ☐

Tag: Untagged · Layer: Default

Transform

| | X | Y | Z |
|---|---|---|---|
| Position | 0 | 0 | 0 |
| Rotation | 0 | 0 | 0 |
| Scale | 1 | 1 | 1 |

**Illustration 2:** Transform component

You can see the final result in *scene1* in the accompanying project. You might have noticed that we did not need more than the alternation of the positions, scales, and rotations of the basic shapes to construct our scene. You might have also noticed the existence of several components in the inspector window that are added to the game object. One of these components is *Transform* which we've just used. Each one of the components has its unique function and plays specific role in the look or the behavior of the game object. For instance, *Mesh Renderer* component is responsible for rendering the object. Try to disable this component and see what happens.

To disable a specific component, simply uncheck the check box at the top of the component in the inspector window

If you still unsure about the difference between scene, game object, and component; refer to Illustration 3, which summarizes scene construction in Unity.

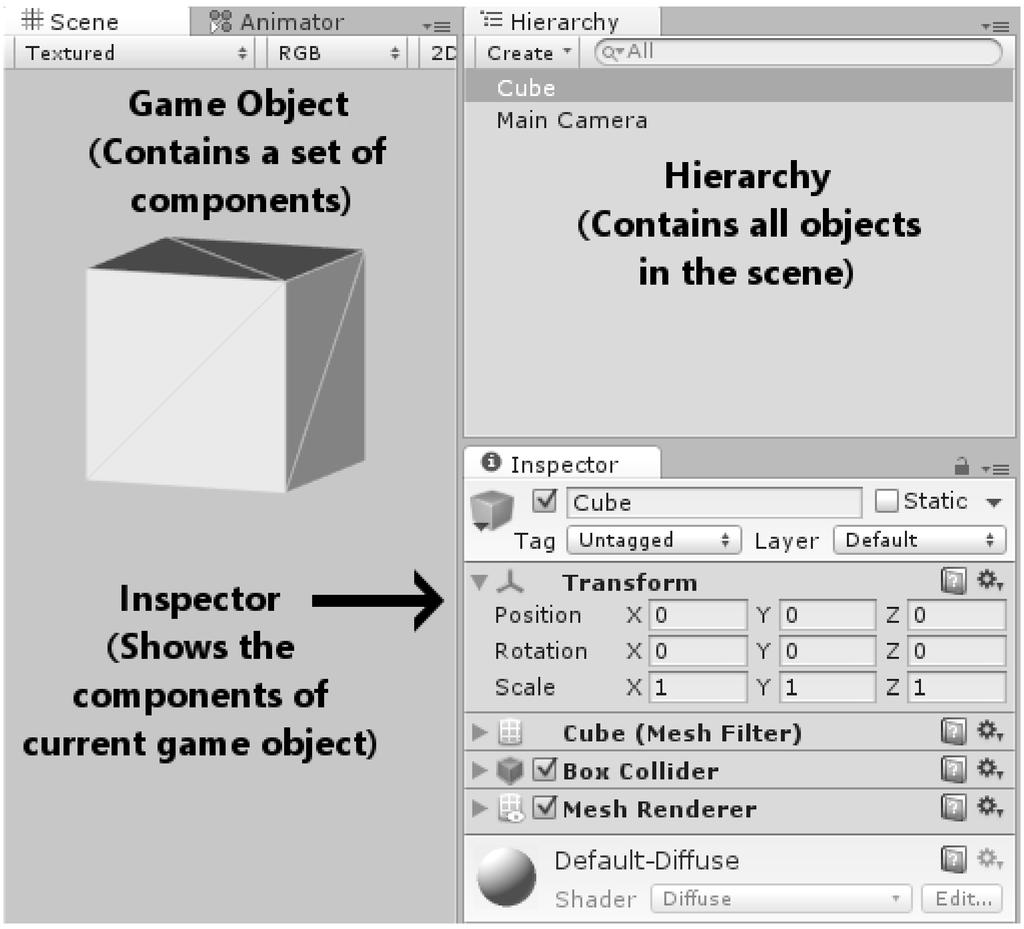


**Illustration 3:** The relation between the objects and the components in Unity

## 1.2 Relations between game objects

You might have noticed during your work in the last section that each game object can be handled independently without affecting the other objects. You might need, however, to move these objects as whole to another place, or make several copies of them. In either case, you need first to select all objects together.

A better approach will be to handle all related scene elements as one unit by adding logical relations between them. In most 3D programs and game engines, objects can be connected together using child/parent relations. In such relations, changes applied to the parent object affect its children but not vice-versa. However, children can override the values resulted from changes to parent.

In Illustration 1, we can identify different building blocks that construct the scene: the ground, the two arcs, and the three boxes. Supposing that the ground is the root of the scene (so that all objects move together if we move the ground), it is reasonable to make it the parent of all other objects in the scene, excluding the main camera.

To build a parent-child relation between two objects in Unity, simply drag the child into the parent inside the hierarchy. To unparent, drag the child out to an empty position in the hierarchy.

It is also reasonable to have each one of the two arcs as a whole unit, even it is not easy to determine which part of the arc must be the parent. In such case, we can simply add a logical parent, which is an empty object that is used as the root of other arc objects.

To add an empty game object in Unity, go to *Game Object* menu and select *Create Empty*

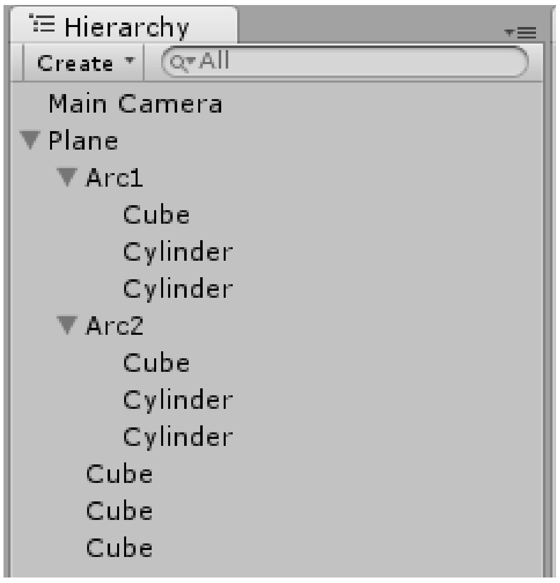
Let's add two empty objects and name them *arc 1* and *arc 2*. It is always a good idea to give meaningful names to game objects in the scene, in order to make dealing with scene elements easier. This is especially useful in large scenes that have many objects.

To change the name of a game object, you can either select it in the hierarchy and press F2, or change the name directly from the name field at the top of the inspector

Position these two empty objects at the center of the scene and add them as children for the ground. Now add the objects of each arc as children to one of these empty parents. Finally, add the three boxes as children for the ground. The final hierarchy should look like Illustration 4.

I joined MITAS because
I wanted **real responsibility**
The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com
Month 16
I was a construction supervisor in the North Sea advising and helping foremen solve problems
Real work
International opportunities
Three work placements
MAERSK

Click on the ad to read more



```
Main Camera
▼ Plane
  ▼ Arc1
      Cube
      Cylinder
      Cylinder
  ▼ Arc2
      Cube
      Cylinder
      Cylinder
    Cube
    Cube
    Cube
```

**Illustration 4:** The hierarchy of the scene with the relations between the objects

If you move the ground object to another position in the scene, you'll notice that its children move with it. However, examining position values in the inspector reveals that only the values of the ground object position change, while the position values of the children remain the same. Although this is unexpected, it can be justified by recognizing that the position values of the children are not absolute, but rather relative to the position of their parent. In other words, what you see in the inspector is the position of the child inside the *local space* of its parent, rather than the *world space*. The same applies to scale and rotation of the child. We are going to discuss these concepts later on, so don't worry about them at the moment.

## 1.3 Rendering properties

In this section, I am going to introduce to you the basic properties of object rendering in Unity. As I have mentioned earlier, *Mesh Renderer* component is responsible for rendering the object. Therefore, it can be only found in objects that are visible in the game. Therefore, it is missing in *arc 1* and *arc 2* as well as the main camera. It is not, however, the only component related to rendering; and we are going to deal with other components later on.

First element to discuss in this section is *texture*. A textures a 2 dimensional image painted over the surface(s) of a 3D shape to give it unique look. For example, we can assign a sand texture to the ground, a brick texture to the arcs, and a wood texture to the boxes. The textures we are going to use are shows in Illustration 5. The files containing these textures must be added to the project before they can be assigned to game objects.

Download free eBooks at bookboon.com

To add texture files to the project, drag them from their current position inside Unity's project explorer. The best practice is to create a new folder for texture files and place the files inside it. To create a new folder, right click the root folder *Assets,* then select *Create > Folder.* Finally, to assign a specific texture to an object, drag the texture from the project explorer to the target object inside the scene window

Once you assign the texture to an object, Unity automatically creates a new material for that texture and adds the material to the renderer of the target object. Materials are automatically added to *Materials* folder, which Unity also automatically creates in the same location of *Textures* folder. Practically, the texture cannot be directly assigned to a game object. Alternatively, the material that is added to the renderer of the object has a shader, and the texture is set as a property of that shader. Shaders specify the final form of the material after applying all effects. Unity uses *Diffuse* shader as default. To fully understand the relation between textures, materials, shaders, and the renderer; see Illustration 6.

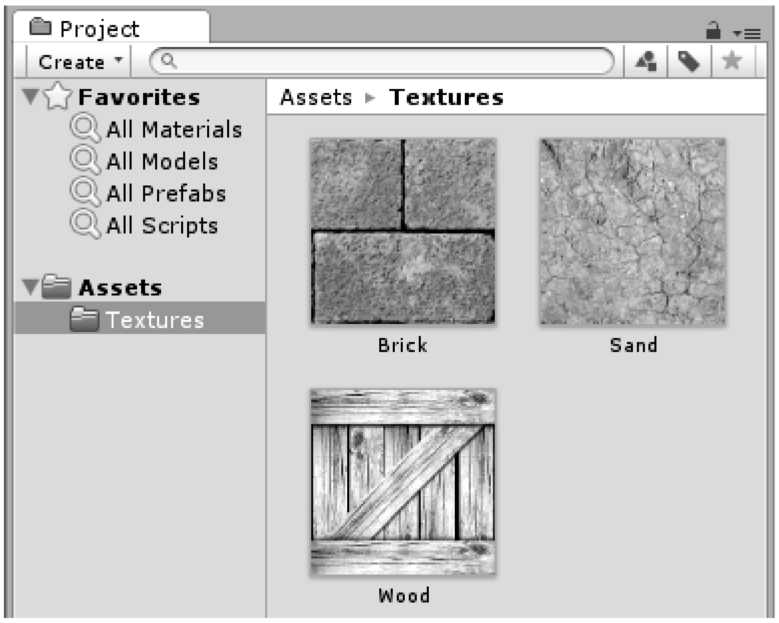
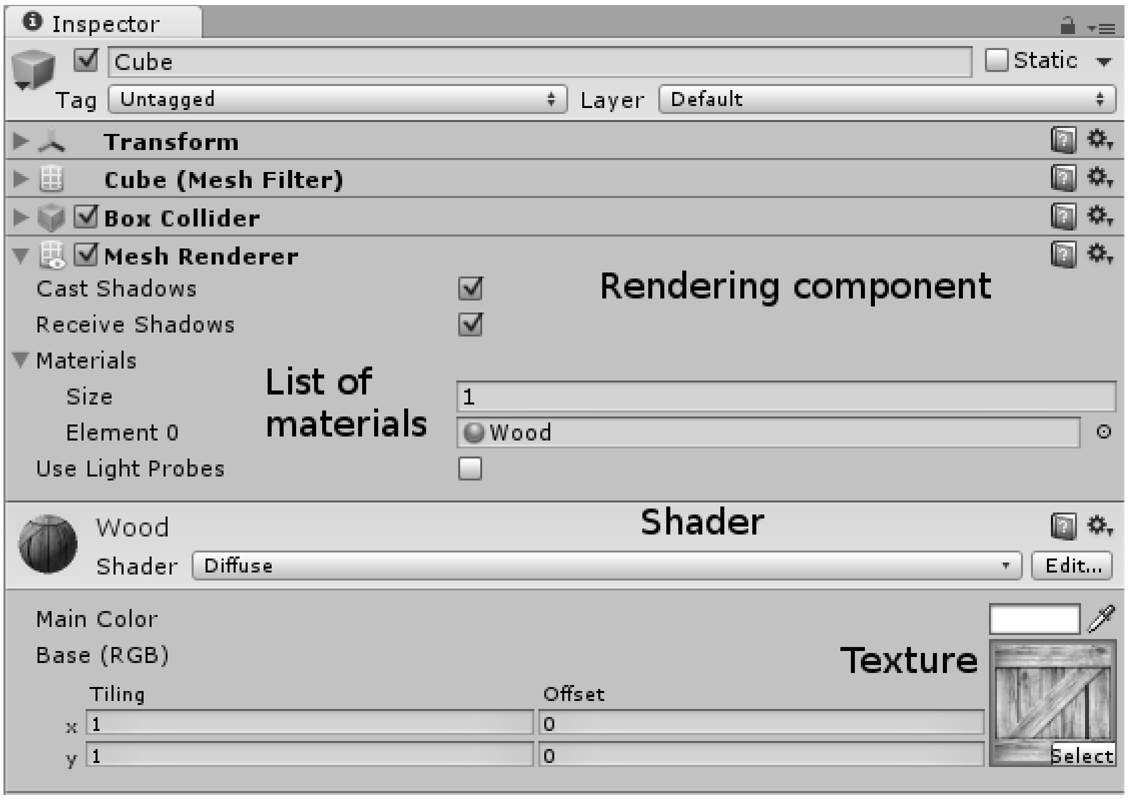


**Illustration 5:** Adding texture files to a Unity project



Cube — Static unchecked
Tag: Untagged — Layer: Default
Transform
Cube (Mesh Filter)
Box Collider ✓
Mesh Renderer ✓
Cast Shadows ✓
Receive Shadows ✓
Materials
Size 1
Element 0 Wood
Use Light Probes ☐
Wood
Shader Diffuse
Main Color
Base (RGB)
Tiling x 1, y 1
Offset x 0, y 0

**Illustration 6:** Renderer component and its relation with materials, shaders, and textures

ie business school

#1 EUROPEAN BUSINESS SCHOOL FINANCIAL TIMES 2013

#gobeyond

MASTER IN MANAGEMENT

**Because achieving your dreams is your greatest challenge.** IE Business School's Master in Management taught in English, Spanish or bilingually, trains young high performance professionals at the beginning of their career through an innovative and stimulating program that will help them reach their full potential.

- Choose your area of specialization.
- Customize your master through the different options offered.
- Global Immersion Weeks in locations such as London, Silicon Valley or Shanghai.

*Because you change, we change with you.*

www.ie.edu/master-management | mim.admissions@ie.edu

Click on the ad to read more

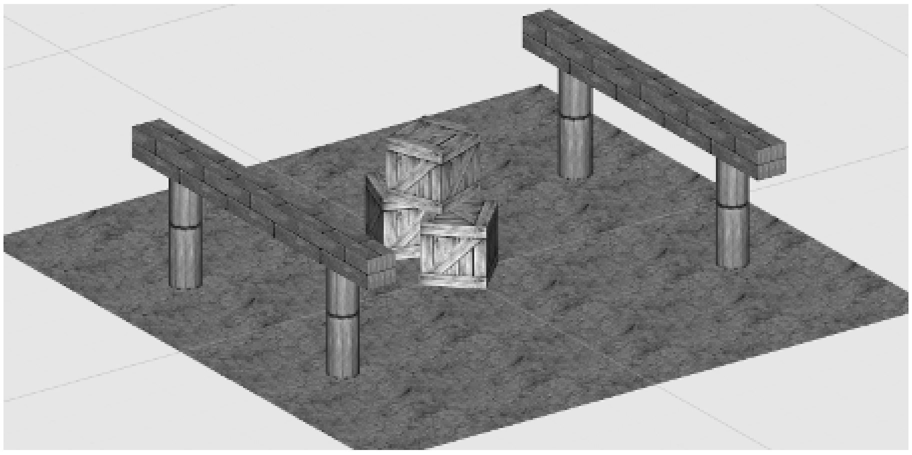


**Illustration 7:** The textured version of the scene

One of the interesting properties of shaders is *Tiling*. Tiling specifies how many times the texture should be repeated on each surface of the object on both x and y coordinates of the image of the texture. *Offset*, on the other hand, specifies whether the texture should be shifted vertically or horizontally. We can make use of tiling to enhance the view of the objects. Let's set tiling values for *Sand* to 5 on both x and y axes, and for *Brick* to 5 on x axis only. As for *Wood*, each box surface should show the texture only once; therefore we keep the tiling values at 1. You can see the final result in Illustration 7, the result can also be found in the accompanying project in *scene1 textured*.

## 1.4 Light types and properties

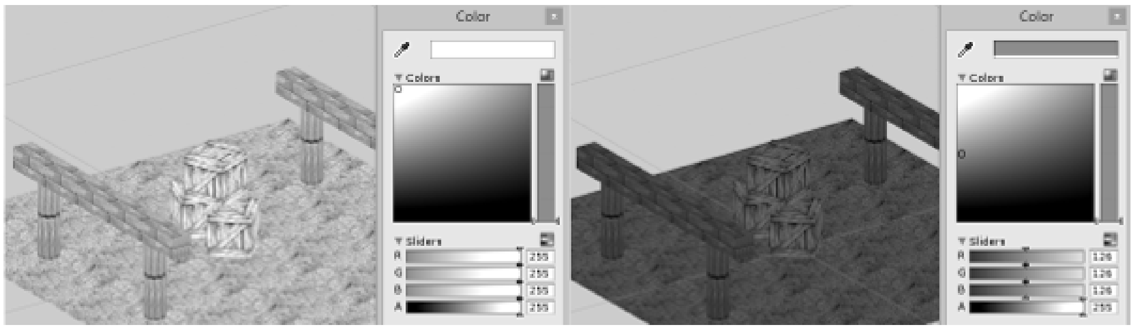
Lighting is an essential element that contributes in scene building. By using lighting, illuminated areas can be distinguished from dark ones, and shadows can be generated. Additionally, light can be used to focus player attention to a specific part of the scene, or even to design puzzles.

In this section, our aim is to introduce types of lights that Unity provides, and how they can be used in scene construction. First type to discuss is *ambient light*, which represents default lighting color on scene-wide level, without adding any other light source. Therefore, it can be used to simulate day/night, sunrise/sunset times, and so on. These effects can be achieved by changing the color of the ambient light.

Because ambient light is a scene-wide property, it is not bound to a particular game object in the scene, but can rather be adjusted from *Render Settings* window, which contains all scene-wide properties for the current scene.

To change the ambient light color, open *Edit* menu and select *Render Settings*. You can then see these settings in the inspector and adjust the *Ambient Light* property. To make the changes in lighting visible in Unity editor, you need first to switch lighting on by using ☼ button.

From an artistic point of view, the importance of ambient light lies in the general feeling it gives to the player. For example, lava environments use warm light colors such as orange and red, while cold environments use blue. Green ambient color is used to give the feeling of a humid environment. Illustration 8 shows how changing the color of ambient light affects the scene.



R 255
G 255
B 255
A 255

R 126
G 126
B 126
A 255

**Illustration 8:** Effect of ambient light on the scene

The second type of lights is *Directional Light*, which can exist once per scene. Directional light represents the major light source that has the largest effect outdoors. Therefore, it can represent the sun in day scenes, or the full moon in night scenes. Directional light, as the name suggest, has a specific emitting direction, unlike the ambient light. It has also other properties such as *Intensity*. Let's now add a directional light to the scene and adjust its properties.

To add a directional light to the scene, go to *Game Object* > *Create Other* > *Directional Light* then select it from the hierarchy to adjust its properties

The directional light affects all areas of the scene equally. Therefore, it does not matter where it is positioned in the space, or what its scale is. However, altering the rotation of the directional light will change the its emitting angle. For example, setting the rotation to (90, 0, 0) will make the light emit vertically from above; just like the noon sun. To make the effect of the directional light easily visible in our scene, it is better to set the rotation of the directional light object to another value, such as (50, -45, 0). Unity draws short beams in the emitting direction of the directional light to make it easier for us to recognize its rotation, like the ones in Illustration 9.

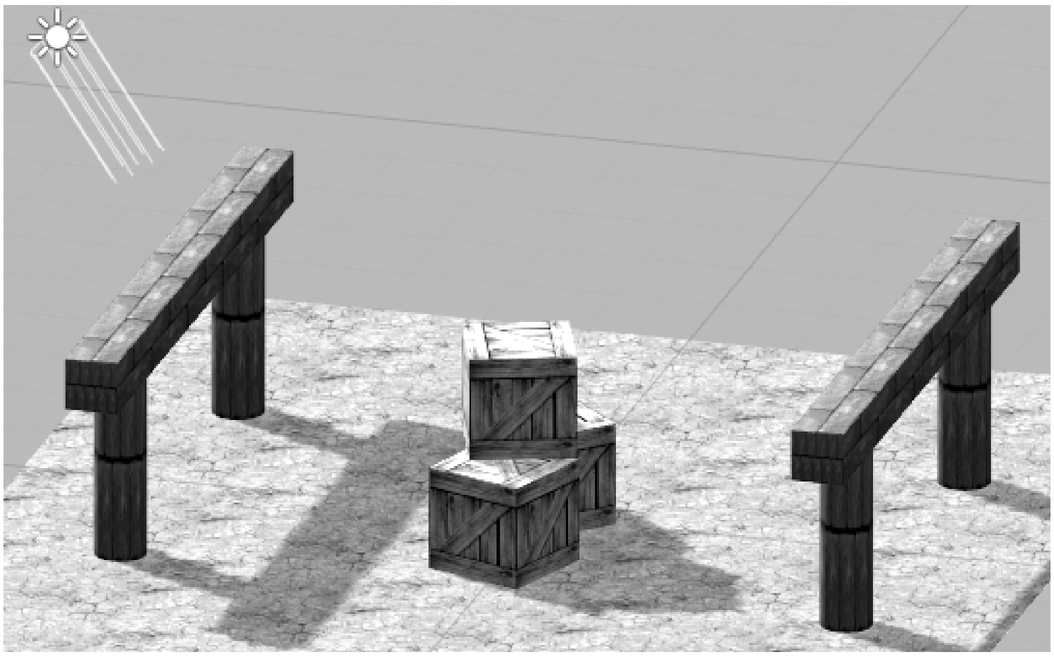


**Illustration 9:** The scene after adding the directional light

Another important property of the directional light is *Intensity*, which controls how strong is the effect of the light on the objects in the scene. A very high intensity will minimize the effect of textures and make the surfaces of the objects look like flat surfaces that have the color of the directional light.

"I studied English for 16 years but... ...I finally learned to speak it in just six lessons"
Jane, Chinese architect
ENGLISH OUT THERE
Click to hear me talking before and after my unique course download

The other two types of light available in Unity are *Point Light* and *Spot Light*. Even we are not going to use them in the current scene, it is a good idea to know how they work. Point light emits light equally in all directions, just like an ordinary electric light bulb. Spot light, on the other hand, emits beams in one direction, forming a spot of light on the surface of the target. You can think of search lights and car lights as examples of spot light. Illustration 10 shows examples of point and spot light. Each one of these lights has its unique properties that you may discover yourself.

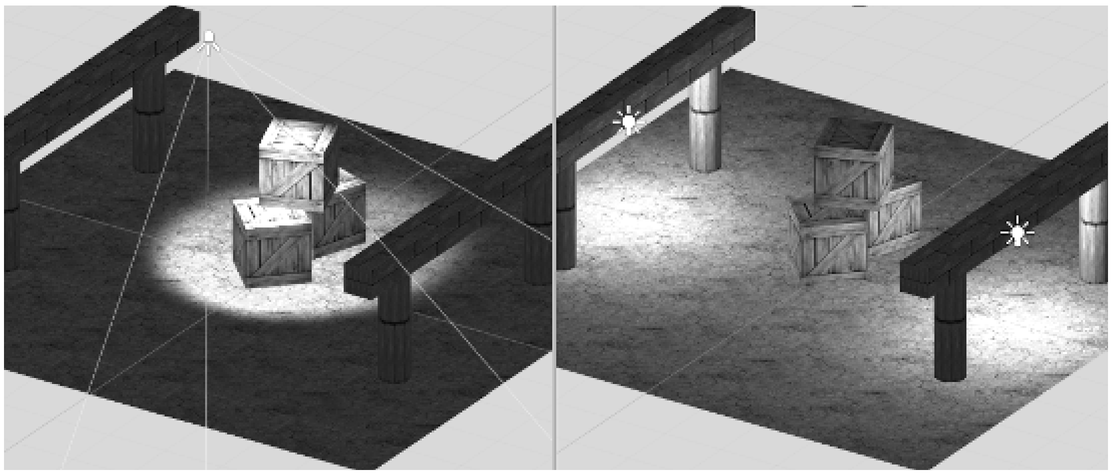
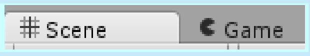


**Illustration 10:** Spot light (left) and point light (right) and their effect on the scene

## 1.5 Camera

When we are done constructing the scene, it is time to know how it going to look for the player when the game starts. Until now, we have been dealing with the scene from the *Scene* window. The other window (*Game*) displays the scene as it appears to player when the game starts. While Unity switches between these two windows automatically when you start or stop the game, you may switch between them at any time to see the game from the player perspective.

To switch between *Scene* view and *Game* view, use 

tabs at the top of the scene view

The main difference between *Scene* and *Game* views is that the latter does not allow you to surf freely in the scene and limits you to the view of the main camera, from which the player observes the game world. If you select the *Main Camera* game object from the hierarchy, you will see a number of properties including:

1. *Background*: the color that fills the horizon of the scene. Empty areas that are not covered by any visible game objects are going to appear in this color.
2. *Projection*: determines whether the distance between an object and the camera affects the size of that object when rendering it. In the default *Perspective* projection, far objects are rendered smaller than near ones. This behavior is similar to human vision system. In the *Orthographic* projection, all objects are rendered with their original size, regardless of how near or far they are from the camera. Orthographic projection is useful in some cases, like 2D games.
3. *Field of View*
4. *Clipping Planes (Far and Near)*

Field of view option is only available in the perspective projection. In this projection, the field of view takes a shape of a frustum. If you complete this frustum to a pyramid, the head of the pyramid is at the position of the camera, and the base of the pyramid is the far clipping plane. The near clipping plane is what turns the pyramid into a frustum, while the field of view is the angle between left and right sides of the frustum. Illustration 11 shows how does the camera use the view frustum to determine the visible objects in the scene. Illustration 12 shows the same scene from the perspective of the camera, and the player as well.

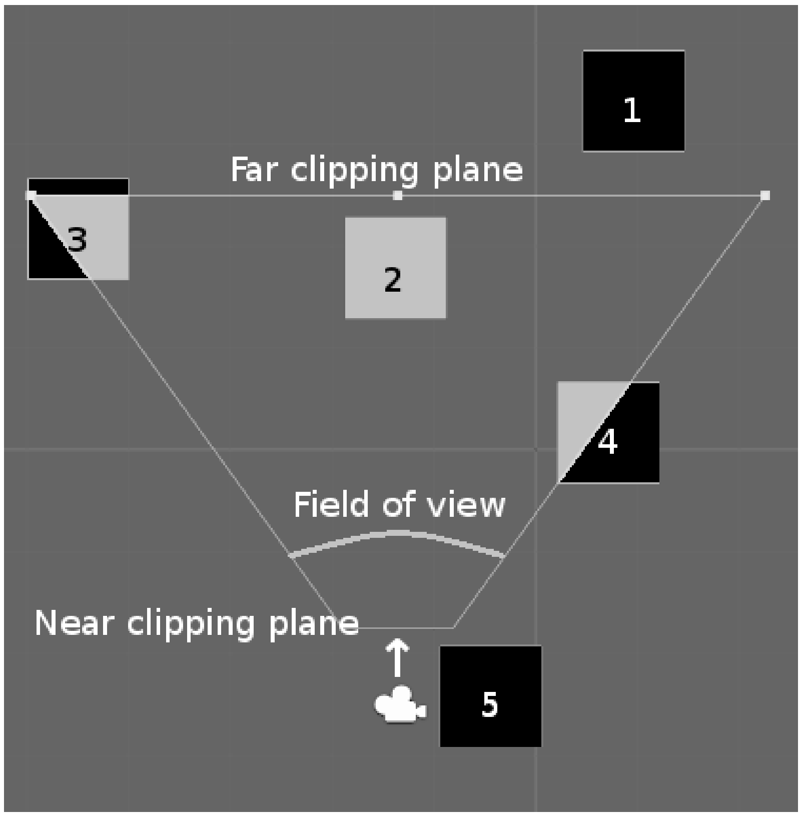


**Illustration 11:** The view frustum of the camera. The black shaded areas mark the invisible parts of the objects

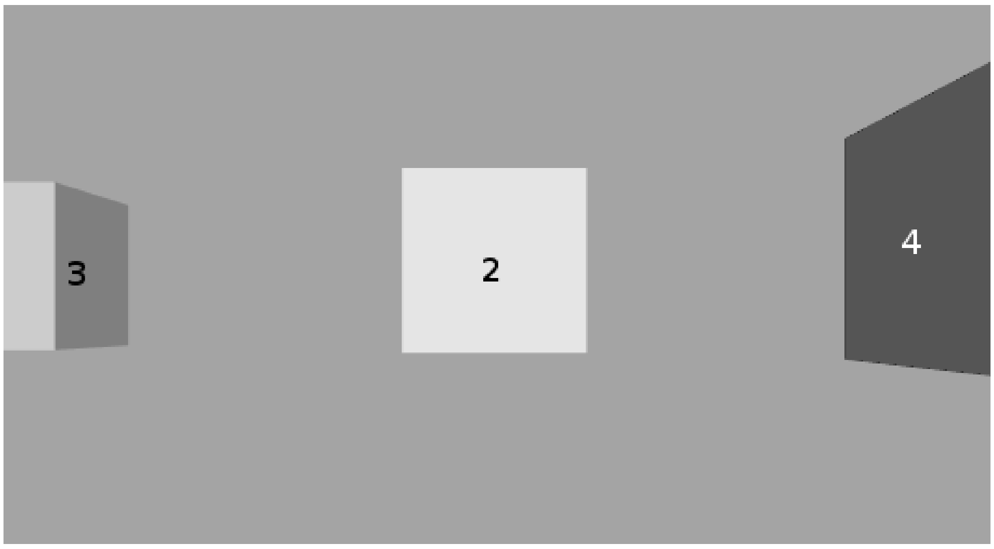


**Illustration 12:** The scene of Illustration 11 as seen by the camera

## 1.6 Controlling objects properties

After we have constructed the scene and had an idea about how it is going to look like for the player, it is time to do some programming. Programming scrips is a core element in game development, since scripts define the behavior of game objects during play time. Let's begin with script that has a simple function: displacement of the objects. Let's also take the camera as the first object to add scripts to.

Excellent Economics and Business programmes at:

university of groningen

"The perfect start of a successful, international career."

CLICK HERE to discover why both socially and academically the University of Groningen is one of the best places for a student to be

www.rug.nl/feb/education

Click on the ad to read more

It is recommended that you create a new folder called *scripts* inside the root folder of the project *Assets* to save our scripts in. After creating the folder, we are going to create our first script *CameraMover* into it.

To create a new script, right click *scripts* folder and select Create > C# Script then name the file *CameraMover.cs*

Unity supports three different programming languages, but I will stick to *C#* in this book. However, if you are familiar with *Javascript*, you may use it instead of *C#* by changing the syntax of the scripts listed in the book. It is also advisable to use *MonoDevelop* development environment included with Unity instead of *Microsoft Visual Studio*. Listing 1 shows the default template for all *C#* scripts created in Unity.

```
using UnityEngine;
using System.Collections;

public class CameraMover : MonoBehaviour {

	// Use this for initialization
	void Start () {

	}

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

	}
}
```

**Listing 1:** Default script template in Unity

Unity helps us by adding the most common functions, *Start()* and *Update()*. The first one is important to initialize the script (i.e. to set the default values of the variables), and is called once at the beginning of the script life cycle. The second function *Update()* is called once per frame, in order to perform the required changes on the properties of the object over time.

Unity handles scripts like other components we've seen so far, such as *Renderer* and *Transform*. Therefore, the scripts are not active unless they are added to game objects. Scripts must inherit from *MonoBehavior* class in order to be recognized by Unity as components. If you don't have much experience in object-oriented programming, you might not be sure what does inheritance mean in this context. But that's fine, since all you need is to keep the structure of the default template. Next step is to add our newly created script to the camera. Once we do this, all behavior we code in the script applies to the camera game object.

To add a script to a game object, select the target game object from the hierarchy, and then drag the script inside the inspector. Alternatively, you may click *Add Component* button at the bottom of the inspector and type the name of the script in the search box then hit Enter.

Download free eBooks at bookboon.com

We are now ready to code the behavior in our script. In the cut-scenes of some video games, the camera moves around slowly and shows the player the scene from different angles. Why don't we try something like this? Assuming that we want the camera to keep moving upwards slowly, we need to move it by constant speed in the positive direction of the y axis. Additionally, we can add a variable that controls movement speed. Listing 2 illustrates necessary code for camera movement. Double clicking a script file loads the default script editor (*MonoDevelop* in our case) and opens the script for you to edit the code.

```
1. using UnityEngine;
2. using System.Collections;
3.
4. public class CameraMover : MonoBehaviour {
5.
6.       public float speed = 1;
7.
8.       // Use this for initialization
9.       void Start () {
10.
11.      }
12.
13.      // Update is called once per frame
14.      void Update () {
15.           transform.Translate(0, speed * Time.deltaTime, 0);
16.      }
17. }
```

**Listing 2:** Camera moving script

It is a good idea now to describe the mechanism that Unity uses to run the scripts. When the game starts, Unity calls *Start()* function from all active scripts in the scene. By doing this, Unity makes sure that all scripts are initialized and ready to enter the game update loop. In this loop, frames are constructed and rendered through various steps. These steps include reading user input, moving and animating objects, running the physics simulation and Artificial Intelligence (AI) algorithms, executing the game logic, and rendering the frame. In order to have an acceptable play experience for the player, at least 25 to 30 frames must be rendered every second. In each iteration of this loop, Unity passes through all active scripts in the scene and calls *Update()* function from them. This procedure continues as long as the game is running.

In line 6 of Listing 2, we declare a floating point number with the value of 1. This will be the speed of camera movement. The value of *speed* is multiplied by the *delta time* in line 15 to perform a translation on the y axis using *Translate()* function. *Translate()* takes the displacement values on x, y, and z axes. Notice that we provided a non-zero value for the y axis only, since we don't want to move the camera on neither x nor z axes.

Download free eBooks at bookboon.com

In line 15, we need to move the object upwards with specific distance every frame, and we need to compute this distance. As we know from physics, the speed of an object equals the distance the object moves in the time unit. Therefore, we need to multiply the speed by the time unit to compute the distance we need to move the object with. But how to get this time? Since translation is performed once every frame, the time we need to know is the time passed since the rendering of the previous frame (i.e. since the last time the object moved). This value is given to us by Unity in the variable *Time.deltaTime*. So, when we multiply this value by the movement speed, we get the distance we need to move the object with. All you have to do now is to save the script and start the game to see the result.

To start / stop the game, click on ▶ button

One interesting feature of Unity is the ability to modify the default values of the public variables directly from the inspector, so we do not have to change the code and recompile it after every modification. Illustration 13 shows how do public variables appear in the inspector. You may try to change the speed to a negative value and see the result you expect.

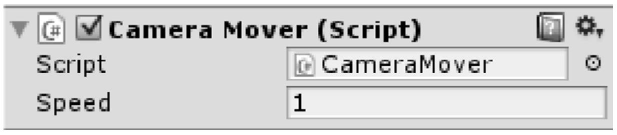

Camera Mover (Script)
Script CameraMover
Speed 1

**Illustration 13:** The script component as it appears in the inspector

LIGS University
based in Hawaii, USA
is currently enrolling in the Interactive Online BBA, MBA, MSc, DBA and PhD programs:
- enroll by October 31st, 2014 and
- save up to 11% on the tuition!
- pay in 10 installments / 2 years
- Interactive Online education
- visit www.ligsuniversity.com to find out more!

Note: LIGS University is not accredited by any nationally recognized accrediting agency listed by the US Secretary of Education.
More info here.

Click on the ad to read more

After running the game for a while, you are going to see that the scene you have constructed is no more visible to the camera as it moves higher and higher. Let's try to fix this little problem by forcing the camera to always look at the center of the scene, regardless of its current position. This is achieved by altering the rotation of the camera downwards as it goes higher. Fortunately, we don't need to bother ourselves with the dirty details of this rotation, since Unity provides this functionality directly through *transform.LookAt()* function. Simply add the following line after line 15 in Listing 2.

```
transform.LookAt(Vector3.zero);
```

What we expect to see now is vertical movement of the camera and rotation towards the origin of the scene, which keeps the scene visible. If you start the game now, you are going to get a change in the z position of the camera with the time. As the camera gets higher on the y axis, it gets also closer to the center of the xz plane. This unexpected movement along z axis can be justified by understanding the difference between translation in the *local space* and translation in the *world space*.

By default, *Translate()* function is applied in the local space of the object, which is affected by its position and rotation. World space, on the other hand, has fixed x, y, and z axes that are constant among all game objects all the time. To understand the concept of local space, consider an airplane object like the one in Illustration 14. This plane has a local space that is different from the global world space. The right wing of the plane, which points to the positive direction of the x axis of the local space. The plane front points towards the positive direction of the z axis of the local space, and the positive y axis of the local space is perpendicular to the upper surface of the plane and points upwards. (As a sort of motivation, we are going to build this plane model and fly with it in the next chapter).

Back to our camera, when it rotates to look at the center of the scene, its local y axis becomes no more parallel to the y axis of the world space as it originally was. Therefore, when you move the camera along its local y axis, it going actually to move along the z axis of the world space as well. Illustration 15 shows the local space axes of the camera after the rotation.

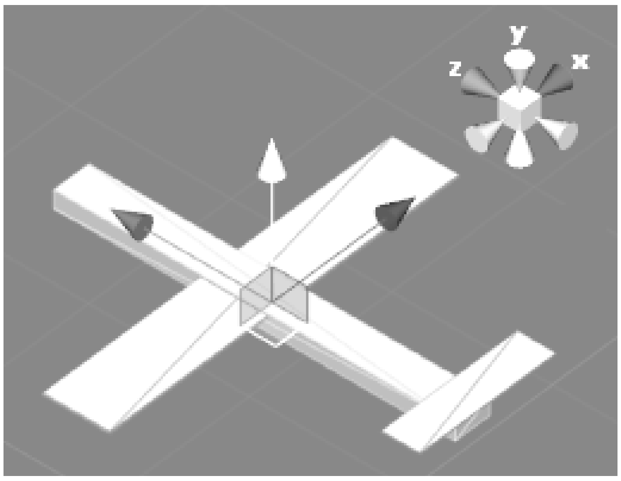


**Illustration 14:** Local space axes of a plane model

Download free eBooks at bookboon.com

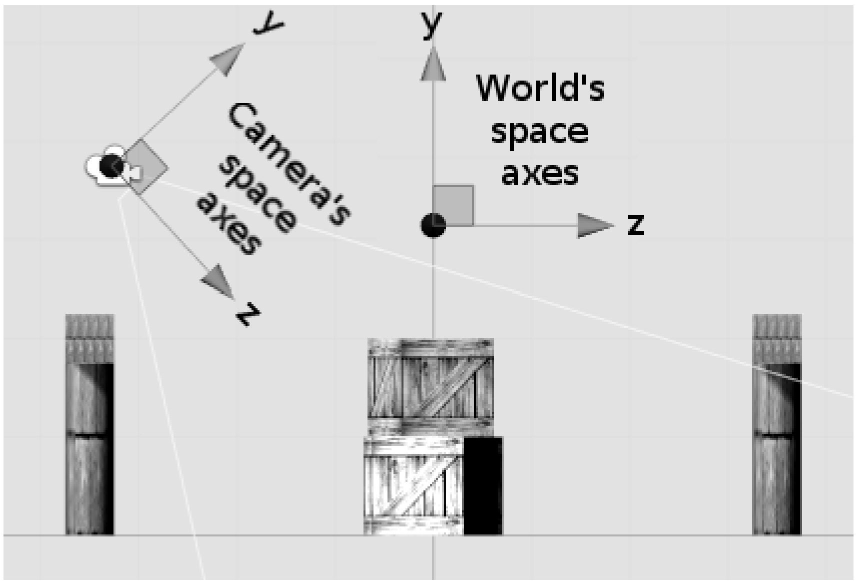


**Illustration 15:** The difference between world axes and camera's local space axes

To fix this unwanted behavior, we need to tell Unity to move the camera on the axes of the world space rather than the axes of its local space, which changes when camera rotates. So our new line 15 in Listing 2 is

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

Start the game now to see the difference. Let's now do something that is more exciting. What if we change the rotation of our directional light over the time? How will this rotation affect the shadows of scene elements? To do this we need to create a new script called *LightRotator* and add it to the directional light. This script is shown in Listing 3.

```
using UnityEngine;
public class LightRotator : MonoBehaviour {

    public float speed = 10;//10 degrees per second

    void Update () {
         transform.Rotate(speed * Time.deltaTime, 0, 0);
    }
}
```

**Listing 3**: Light rotator script

This script rotates the directional light around its local x axis. Since the directional light emits in the positive direction of its local z axis, the rotation changes the angle between the emitted light beam and the horizon. This gives an effect similar to sunrise and sunset (for better understanding of this rotation, use the left-hand rule, rotate your hand around the middle finger, and see how the direction of your index changes). The speed used here is a little bit higher than the speed of camera movement, because we want to see the effect light rotation before the camera goes far away from scene elements.

In this chapter we have learned how to construct a simple scene using basic shapes with different positions, rotations and scales. We have also learned how to use 2D images as textures to give details to the shapes. We have introduced different types of lighting sources and discussed their interesting properties. We have also written some scripts that change the properties of the objects during game execution.

Notice that there are more advanced topics regarding scene construction that were not covered in this chapter. The aim of this chapter was to provide a quick introduction to scene construction, so that we understand the structure of the scene and be able to interact with it. This was important because interaction is a core element in game development. There will be more on textures, materials, lighting, and shaders in the coming chapters.

## Exercises

1. We've discussed the use of basic shapes to construct a scene. Use them to construct a more complex scene. For example, draw a house with a garden surrounded by a wall. You can look up in the internet for free textures to use.
2. Add point lights to the scene you have constructed in exercise 1. Select appropriate positions for these lights, and adjust their properties to make a night scene with electric lights. Remember to choose a dark color for the ambient light.
3. Modify *CameraMover* script to make the camera rotate around the scene horizontally, while keeps looking at the center of the scene. Axes of local space can help you to achieve this behavior. Add the modified version to the camera in the scene you have constructed in exercise 1.
4. Try to make use of relations between objects to add a spot light to the camera. This spot light must move along with the camera and focus on the position where the camera looks.

Download free eBooks at bookboon.com