Learn 2D game development with Unity

By Rasmus Møller Selsmark

NOTE: New revisited version is available here: 2D-Unity-cannon-game

License for this book

Prerequisites

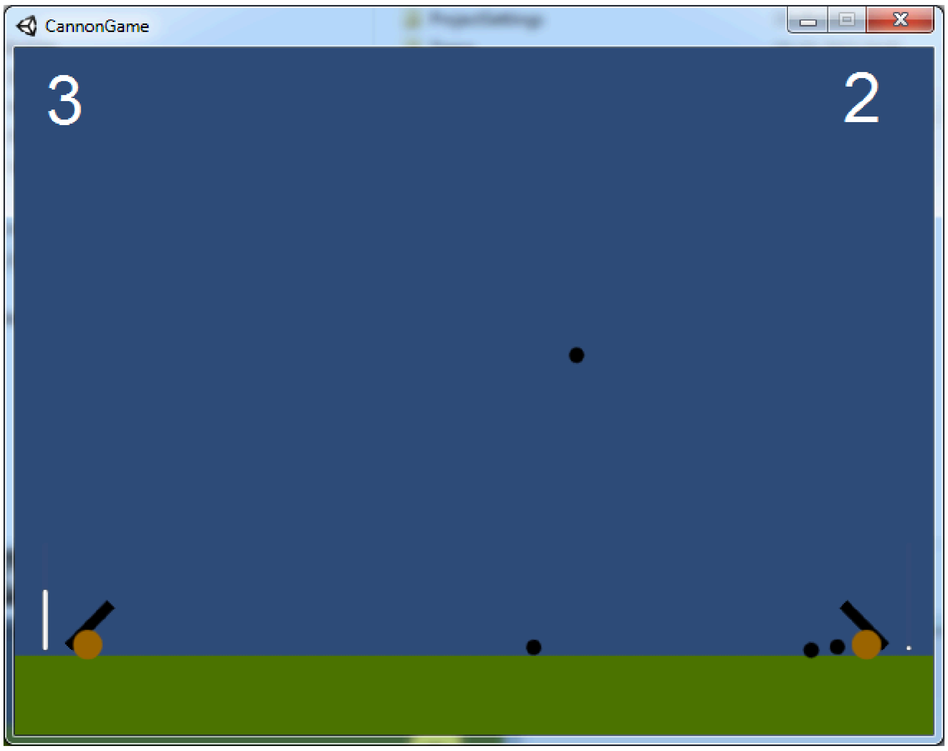
In order to develop games using Unity, you need a newer PC or Mac computer, Unity does e.g. not run on iPad. See “For development” section on https://docs.unity3d.com/Manual/system-requirements.html for additional information.

About this book

> **Tip:** Code including graphics for the game can be found at https://github.com/rasmusselsmark/CannonGame-Basic
>
> Github is a website, where you can store and version code, and is used by many open-source projects. By using the link above you can download a zip-file with latest version of game.
>
> For this book, Unity 2017.1 has been used for building the game, but you can also use other versions of Unity, however it’s normally recommended just to use latest available version of Unity, which can be downloaded at (TODO: link)

Purpose of this book is to give an introduction to developing 2D games with Unity. The game being used as example throughout the material is being developed for PC/Mac, but can later be extended to also support mobile.

At the end of this book, the game will look like this:



Note that the game developed in this book is kept relatively simple, however the purpose is to be able to create a game completely from scratch, i.e. not importing pre-made models or already partly finished games. This way the intention is to learn Unity better, and give better foundation when continuing with more advanced games later.

In this material, Unity 2017.1 on Windows 10 is used, however other versions of Unity can be used as well, and except for the installation of Unity (which are different on Mac and Windows), all steps and instructions apply to Mac users as well.

Unity also offers tutorials at https://unity3d.com/learn and documentation at https://docs.unity3d.com, both which are good sites for getting started with game development with Unity. You can also find lots of material for Unity on e.g. YouTube.

Despite lots of other Unity-based learning material, based on my experiences teaching kids programming at the Coding Pirates organization (http://codingpirates.dk), I saw a need for a more beginner/entry-level book for learning Unity and to some extend programming. This book is the first in a (still planned) series of books based on the same cannon shooter game.

Introduction to Unity

Unity is a tool specifically aimed at developing games, and is used by more than 1 million game developers worldwide, from small indie game developers to large game studios and productions. Developing a game using Unity combines designing the game graphics with programming the gameplay. In this book, the C# programming language is used, and to beginners within programming, especially the programming part can be challenging and take some time to learn, but learning how to program makes it possible to freely define the gameplay of your game, and allows you to create any type of game you would like, for any platform ranging from PC/Mac desktop to mobiles and consoles like Xbox and Playstation.

As introduction to programming, Unity and game development in general, provides a good way into the concept of "objectoriented programming" as a game consists of tangible objects, e.g. a player or enemy, so it's often natural where the logic/code "belongs" in a game.

The primary reason for choosing Unity (or other game engines) as development tool, is the possibility to develop your game for multiple platforms. Typically you can e.g. release a game for both Android and iOS without changing anything in the game code.

Unity is free to use as long as you are making less than 200,000 USD revenue per year, and you can ship games made with the free version of Unity.

Get started with Unity

Download and installation

1. Open https://unity.com/download
2. Click the "Download for Windows" (or Mac/Linux) link to install the Unity Hub
3. Choose "Download Installer"

TODO: Show installation flow using Unity Hub

For our purpose, the following list of components are installed. Actually it's just the "Unity" component which is required to make a game with Unity and play it locally on your computer. The other components are used if you want to distribute games to others or for other platforms than used in this book.

Component	Description
Unity 5.3.4f1 (TODO)	The Unity editor application. This is the only required component for developing games, e.g. if you just want to test Unity.
Documentation	Unity documentation installed on your PC. Useful if you are offline, otherwise all documentation can also be found at https://docs.unity3d.com
Standard Assets	Collection of objects, characters and scripts which can be used to quickly start building simple games.
Example Project	Example project using Standard Assets.

Windows Build Support (TODO: still required?)	For at kunne bygge spil som kan køre direkte på Windows, uden Unity installeret, f.eks. hvis du vil dele dit færdige spil. Hvis du er på Mac, bør du vælge “Mac Build Support” i stedet.
Android Build Support	If you want to build games for Android mobile devices. If you don’t have an Android device, you don’t need to install this component.
WebGL Build Support	Publishing your game as HTML5 for playing in browser.

This book uses MonoDevelop for developing code for the game, so Visual Studio hasn’t been selected above. If you want to use Visual Studio, just install this. MonoDevelop/Visual Studio is the tool used for writing the code for our game.

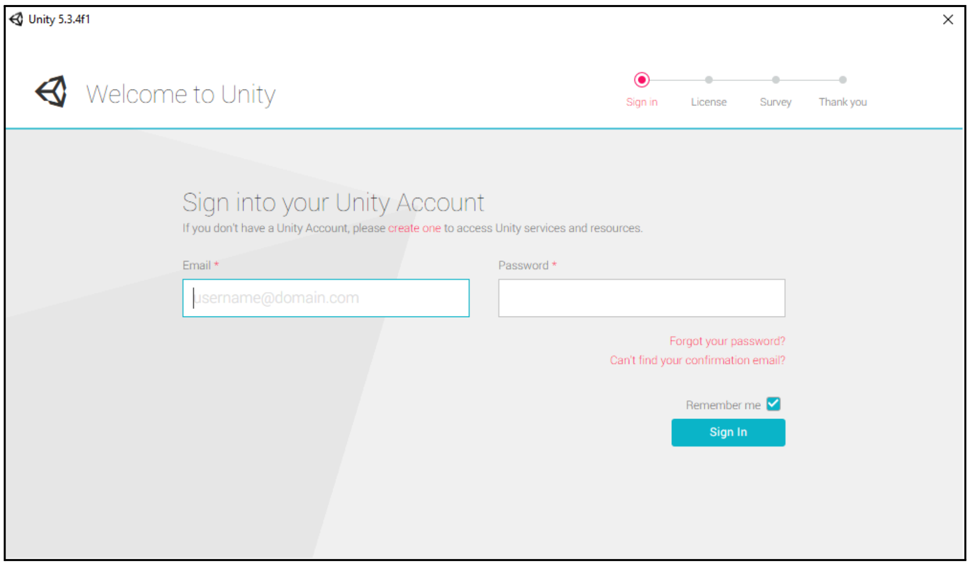
Press the “Next” button to choose where to install Unity on your computer. In this case I choose to name the folder with Unity version number, which makes it easier to install several versions of Unity on the same machine. Note, it’s not possible to choose folder name when installing on Mac, but you can simply rename the /Application/Unity folder after installation is done on Mac.

(TODO: new screenshot)
Wait until Unity has been downloaded and installed on your computer, which can take a while depending on which components you have selected, combined with speed of your internet connection. In some cases up to one hour (based on experience of installing many machines in same classroom over same internet connection, in these cases it can be a good idea to download the installers to a USB and share this)

When the installation is done, you can choose “Launch Unity” in last part of the installation guide: (TODO: new screenshot)

Activation

When you start Unity for the first time, you will be asked to register as user and activate Unity. This requires internet connection from the machine.



Sign into your Unity Account

If you don't have a Unity Account, please create one to access Unity services and resources.

Email *: username@domain.com

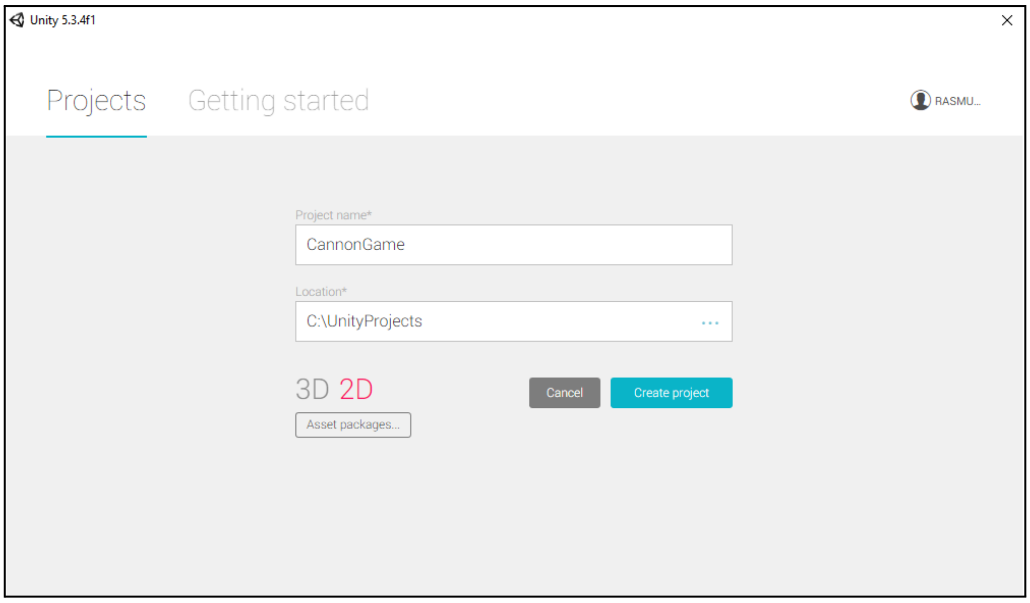
Password *:

(TODO: New screenshot)

Follow the instructions to sign in or create a new account and activate Unity.

Create 2D Unity project

In the projects list, click the “New Project” button to create the project, and select project type to be 2D:



Project name*: CannonGame

Location*: C:\UnityProjects

(TODO: new screenshot)

In this case we’ll name the game ”CannonGame” (you can of course freely choose another name) and store the project files under `C:\UnityProjects` folder, as well as specifying type as 2D.

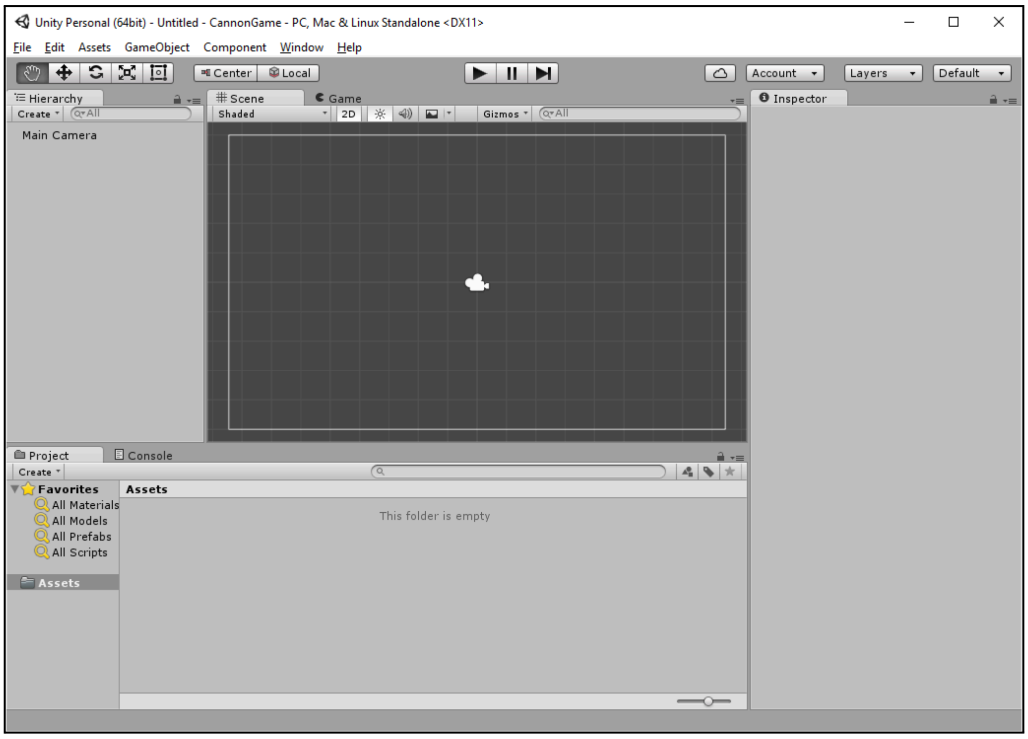
You can switch type between 2D and 3D later, but it's easiest if you choose the correct type from beginning, as it affects e.g. camera settings. Remember where you have stored the project files, e.g. if you need to make a backup or share with others.

Finally click the "Create project" button.

Unity user interface

Unity editor

When starting Unity editor, it will typically look like this:



The Unity editor is composed by windows/tabs, each which is used for editing or showing properties for a part of the game. It's possible to customize the layout of windows in the editor, which we will get back to later.

In this section the most common windows in Unity are covered.

Window: Scene

This is where you work on the current "scene", i.e. can place game characters and other objects. As scene is often the same as a level in your game. Later we'll show how to insert objects into the game.

Window: Game

The Game tab is where you can see how the game will look when played, and you can also play the game in this window by clicking the “Play” button in the editor.

Window: Hierarchy

A scene typically is composed by a number of objects, e.g. player characters, enemies, landscape etc. In the “Hierarchy” tab you can see a list of these game objects, which especially in large games is easier than trying to find the visual object within the scene.

The objects shown in the Hiearchy window/tab are called “Game Objects”

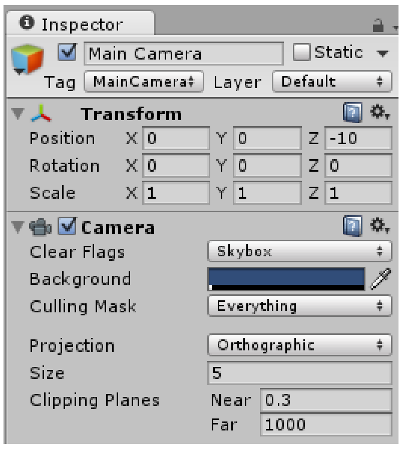
Window: Project

Game objects can be reused across different scenes, so the “Project” tab contains list of all objects/assets used in the game. When we later start to create graphics for our game, it will also be located here.

For the more technical people, the folder structure in “Project” window is the same as for the project files stored on disk.

Window: Inspector

This tab shows properties for the currently selected object in scene. If you e.g. select the “Main Camera” game object in the Hierarchy Window, the position of the camera will be shown with X, Y and Z coordinates:


Inspector

✓ Main Camera ☐ Static

Tag MainCamera Layer Default

Transform

Position	X	0	Y	0	Z	-10
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

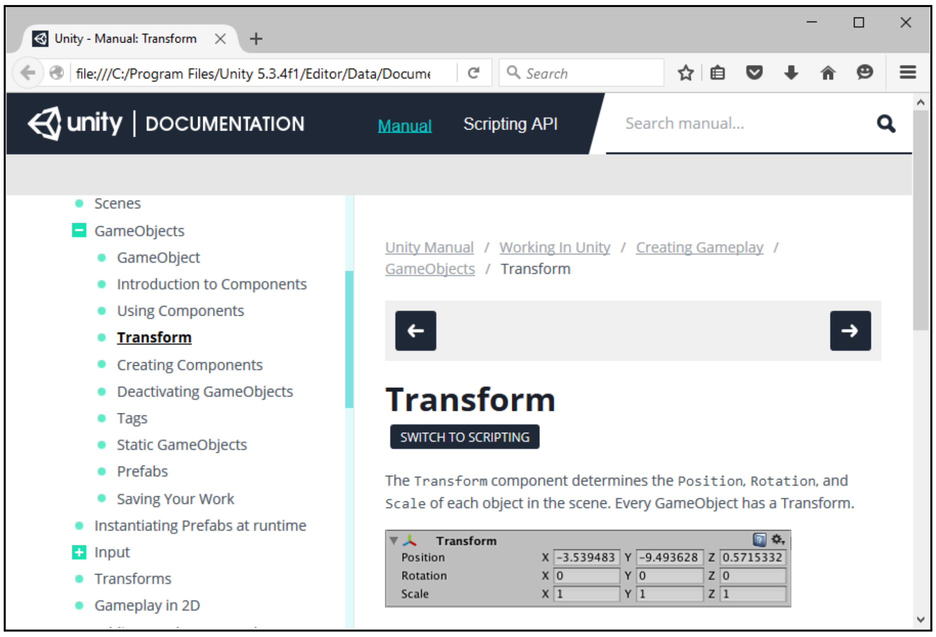
✓ Camera

Clear Flags	Skybox
Background	■
Culling Mask	Everything
Projection	Orthographic
Size	5
Clipping Planes	Near 0.3
	Far 1000

Each of these are called “Components”, in above example the Transform and Camera components, which each has their own “responsibility/capability” for the selected game object.

Unity documentation

It's possible to get context-specific help directly from the Unity editor, by clicking the small blue book-icon next to each component. If you select Main Camera and click the help icon for Transform component, you will see the documentation for Transform:


Unity - Manual: Transform
file:///C:/Program Files/Unity 5.3.4f1/Editor/Data/Docume
Search
unity | DOCUMENTATION
Manual
Scripting API
Search manual...

- Scenes
- GameObjects
  - GameObject
  - Introduction to Components
  - Using Components
  - Transform
  - Creating Components
  - Deactivating GameObjects
  - Tags
  - Static GameObjects
  - Prefabs
  - Saving Your Work
- Instantiating Prefabs at runtime
- Input
- Transforms
- Gameplay in 2D

Unity Manual / Working In Unity / Creating Gameplay / GameObjects / Transform

Transform

SWITCH TO SCRIPTING

The `Transform` component determines the `Position`, `Rotation`, and `Scale` of each object in the scene. Every GameObject has a Transform.

Transform						
Position	X	-3.539483	Y	-9.493628	Z	0.5715332
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

(TODO: new screenshot)
Likewise it's possible to open documentation by clicking the menu item Help → Unity Manual or Scripting Reference.

Window: Console

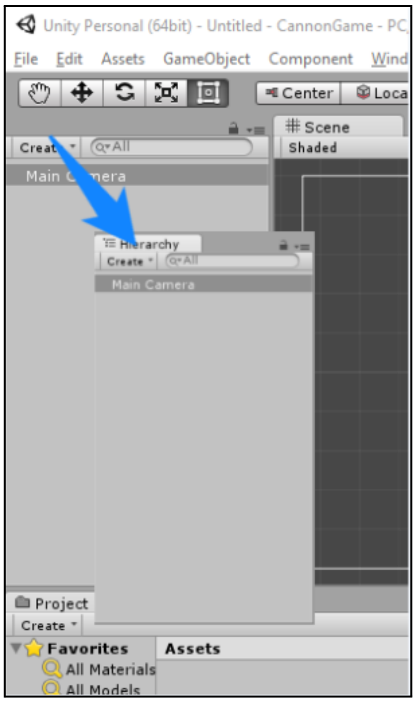
While developing a game (or any other software) you will often need to get additional information about the game, in order to locate any errors in the game. The Console tab shows this kind of information, and will also report if there are problems building the game code.

Customizing the editor layout

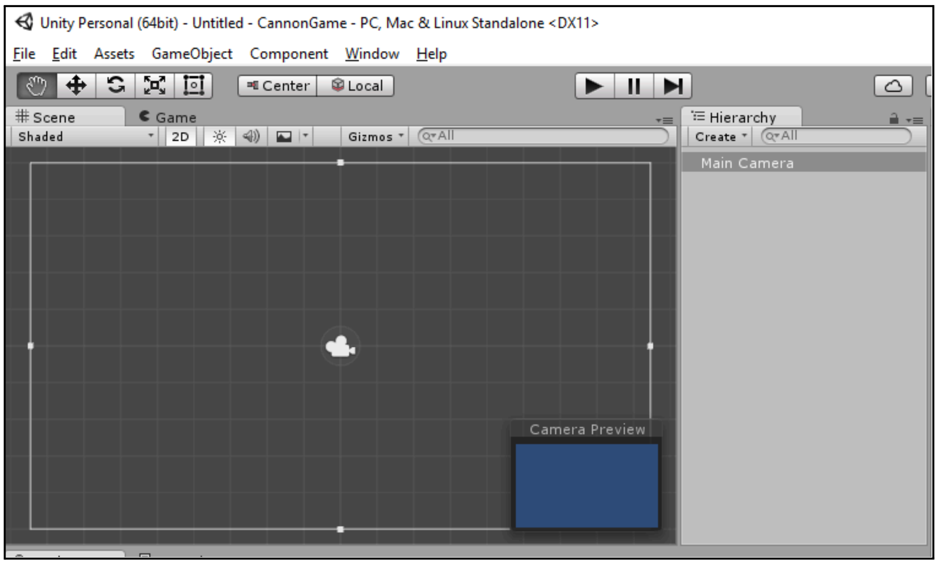
Previously we saw how the Unity editor looks by default, however it's possible to rearrange, and I prefer to always have the Console window visible, in order to see any issues in game reported here, instead of having it hidden behind the Project tab, which is the default layout.

The suggestion for layout presented here, is purely my personal preference. You can choose to use it, or a different layout. And you can always change back to one of the built-in layouts. The most important is that you find a layout which works for you. However in this material, I will use the layout shown in this section.

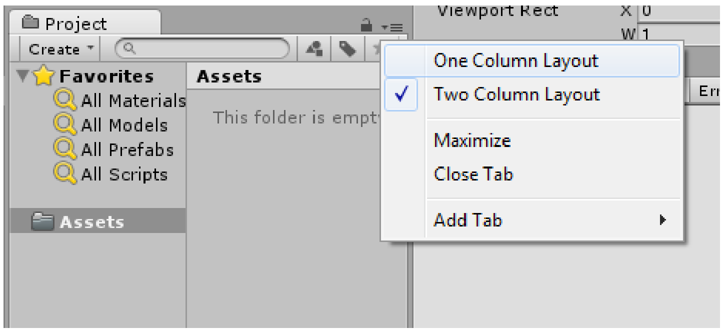
Start by moving the Hierarchy tab to right side of Scene and Game tabs. This is done by pressing the Hierarchy tab with the mouse, and dragging it while holding the mouse button down.



(TODO: Update screenshot?)
The Hierarchy tab is now located to the right of Scene and Game tabs, and should look like this:


Unity Personal (64bit) - Untitled - CannonGame - PC, Mac & Linux Standalone <DX11>
File Edit Assets GameObject Component Window Help
Center
Local
Scene
Game
Shaded
2D
Gizmos
Hierarchy
Create
Main Camera
Camera Preview

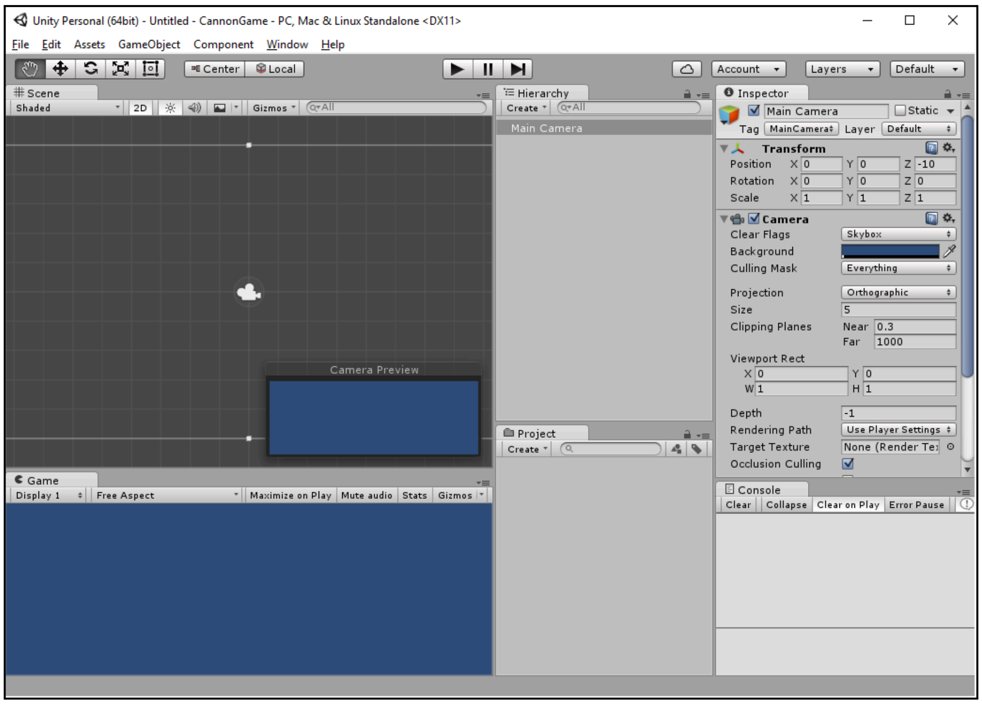
In order to have better overview of the content of the Project tab, I prefer to show this as “One Column Layout”, which is done by clicking the three horizontal lines next to the Project tab.



Finally move following tabs:

1. Project to lower part of Hierachy tab (so Hierarchy stays above)
2. Console below Inspector
3. Game below Scene window

The Unity editor layout should then look like the following:

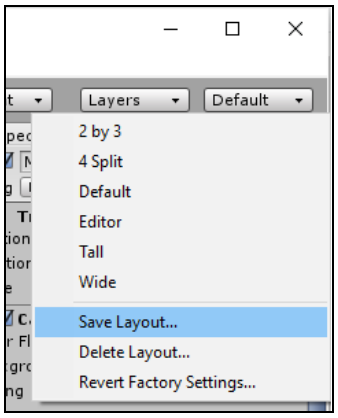

Unity Personal (64bit) - Untitled - CannonGame - PC, Mac & Linux Standalone <DX11>
File Edit Assets GameObject Component Window Help
Center Local
Account Layers Default
Scene
Shaded 2D Gizmos All
Camera Preview
Game
Display 1 Free Aspect Maximize on Play Mute audio Stats Gizmos
Hierarchy
Create All
Main Camera
Project
Create
Inspector
Console
Clear Collapse Clear on Play Error Pause

Main Camera Static
Tag MainCamera Layer Default

Transform
Position X 0 Y 0 Z -10
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Camera
Clear Flags Skybox
Background
Culling Mask Everything
Projection Orthographic
Size 5
Clipping Planes Near 0.3 Far 1000
Viewport Rect X 0 Y 0 W 1 H 1
Depth -1
Rendering Path Use Player Settings
Target Texture None (Render Te
Occlusion Culling ✓

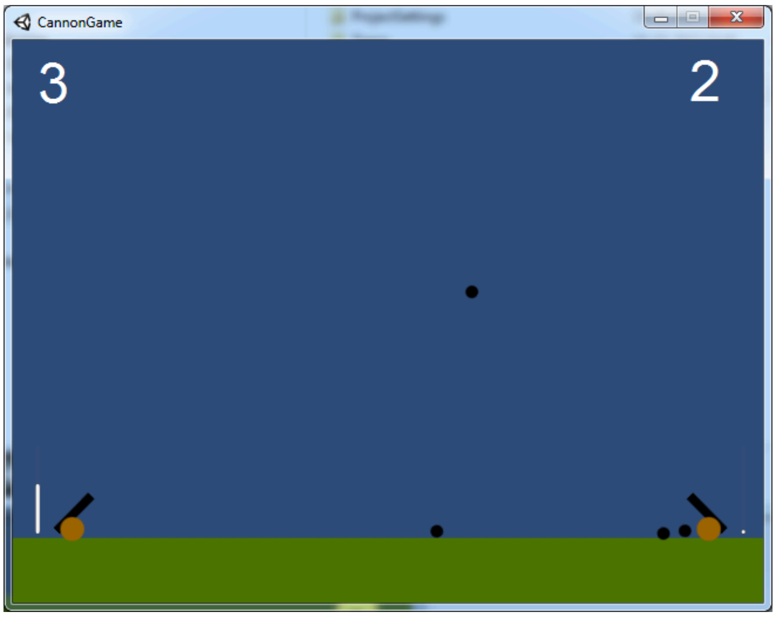
You can choose to save this layout, by clicking the upper-right button in the editor (says ”Default”), choose ”Save Layout” and specify a name for the layout (in my case saved it as “Editor” which then shows up in the list)



The CannonGame game

We're now almost ready to start developing our game. The game we're going to develop is a "Artillery game" style with two cannons, where goal is to hit the other player. As such a relatively simple game, however still will let us work with e.g. controlling players, physics and gravity, prefabs, colliders and programming C# scripts, which all are essential parts of game development.

At the end of this book, the game will look as follows:



Graphics and introduction to GameObjects

For our game we will need the following graphics/art:

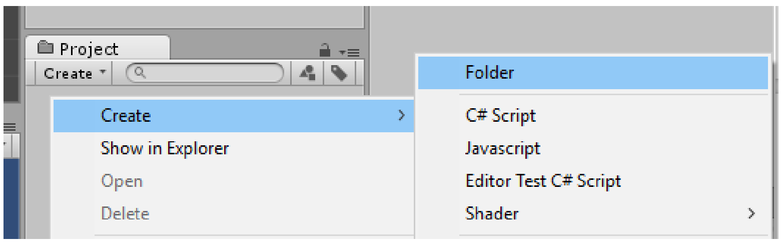
- Grass
- Cannon, composed by a wheel and barrel
- Cannonball

Often it’s possible to game art online for most types of games, either free or reasonable priced. In this case however we will draw all the graphics for the game ourself, partly because the graphics is relatively simple, but also in order to learn how to import and adjust graphics settings in Unity.

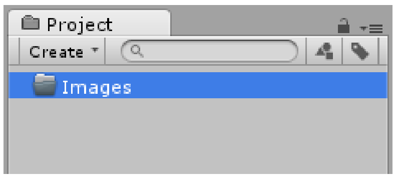
Unity supports importing different kinds of assets from 3D models to simpler 2D graphics, in this case we’ll be working with 2D images in PNG format. Here we’ll be using an online paint tool, however you can freely choose any other tool, as long as it supports transparent background.

First create a folder named ”Images”

1. Right-click in Project window and select **Create→Folder**



2. Name the folder “Images”



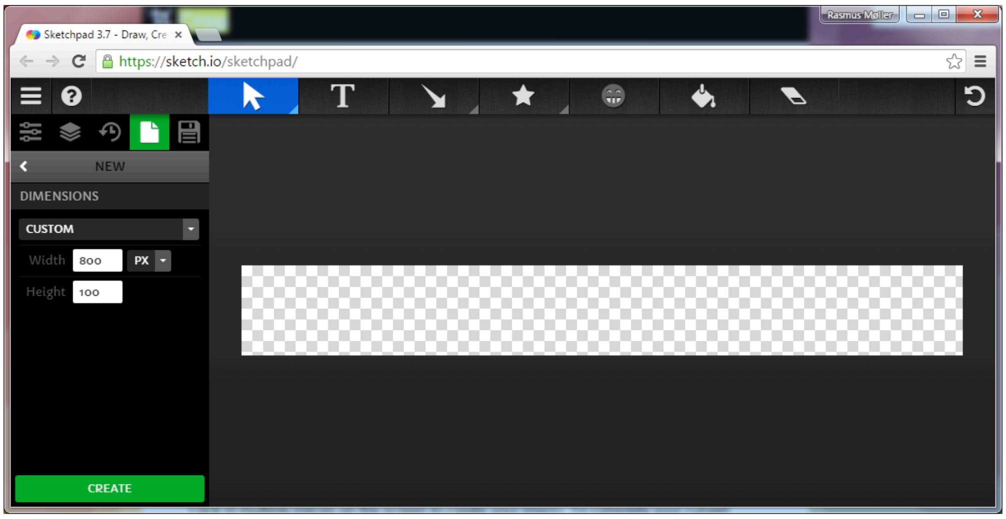
Later we will create similar folders for e.g. scripts, in order to keep files for our game well structured. You can see which files are in your game by right-clicking in the Project window and selecting “Show in Explorer” (“Show in Finder” on Mac)

Grass

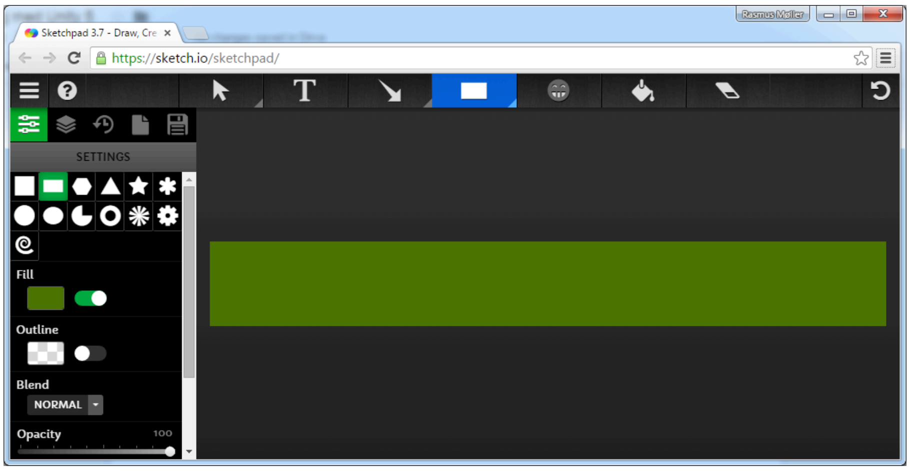
1. Open https://sketch.io/sketchpad/ (or your favorite paint application)

2. Select **File→New** to create a new drawing

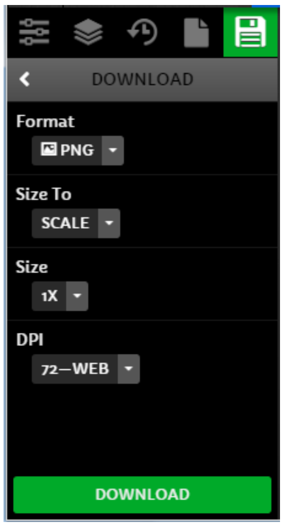
3. Specify size as 800 x 100 pixels and click Create



4. Now draw a green box, by selecting the middle button in the top (likely shows a star symbol), then select “Rectangle” and draw the entire area green:

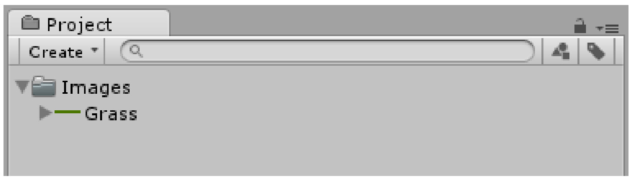


5. Download the picture to your computer, by selecting “Export” (the floppy-disc icon), choose PNG format and click “Download”

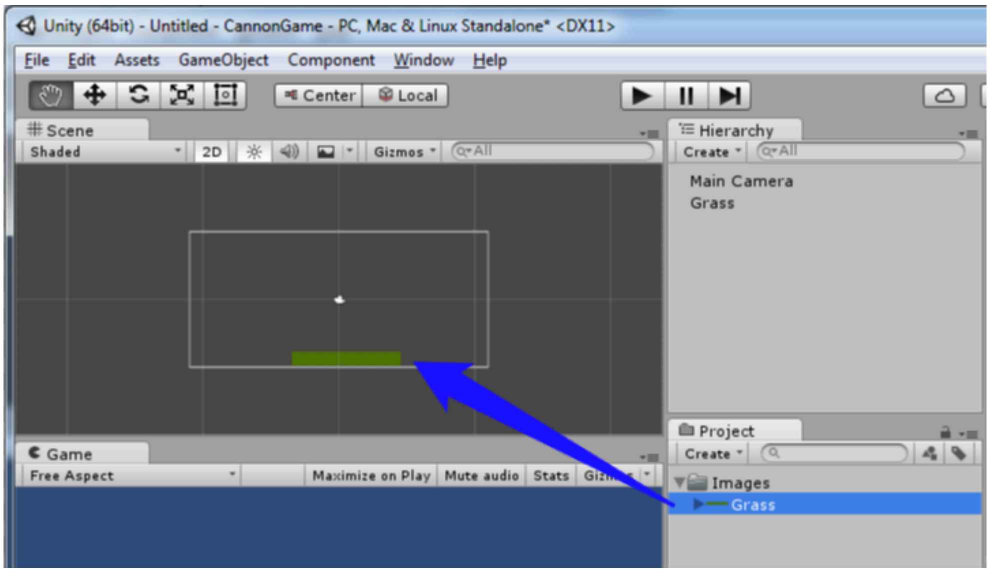


6. The image is then stored on your computer. Open the folder containing the downloaded image file (in Firefox by pressing Ctrl/Cmd+J, in Chrome by by clicking the arrow next to the file, and select “Show in folder”)

7. Use the mouse to drag the file to the “Images” folder in Unity (requires you to have both Unity and Windows Explorer/Finder windows visible), og rename the image to “Grass”:



8. Now drag and drop “Grass” into the Scene window above:



We have now created the first game object in our game(!), so let’s take a look at properties for this, including location and scale/size.

Zoom in on game object

When working with a single game object in the scene, it’s a often good idea to zoom in on that object, which can simply be done by double-clicking on the game object (named “Grass” in this case) in the Hierarchy tab, or alternatively:

1. Select the Grass object
2. Move mouse cursor to Scene view (don’t click)
3. Press the “F”-key on keyboard

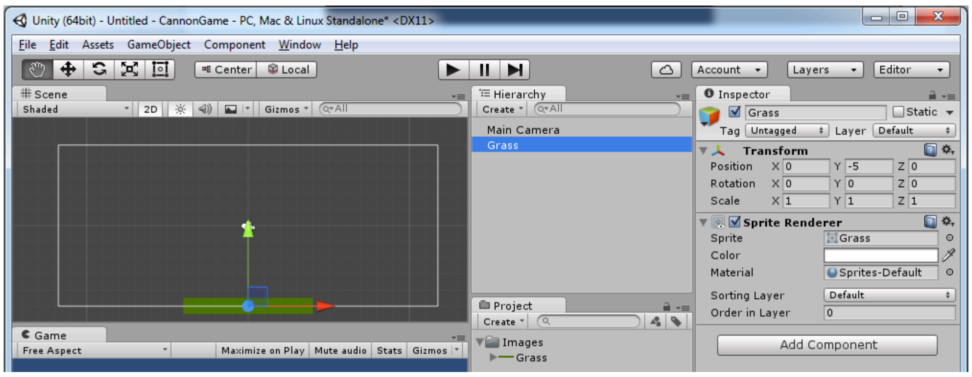
I you want to zoom additionally, this can be done by either

1. Use the scroll-wheel on your mouse (if you have)

2. Hold Alt-key on keyboard, right-click on mouse. You can now zoom in and out by moving the mouse up and down

Game Objects and Components

Select the “Grass” game object in Hierarchy window:



Grass ☑ Static
Tag Untagged Layer Default

Transform

Position	X	0	Y	-5	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

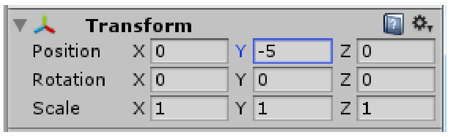
Sprite Renderer

Sprite	Grass
Color	
Material	Sprites-Default
Sorting Layer	Default
Order in Layer	0

Properties for the selected game object are now displayed in the the Inspector window, e.g. X and Y coordinates of position. Z coordinate is also shown, although not used in a 2D game. Reason for showing the Z coordinate is that Unity also is used for building 3D games, and a 2D game as such is just a special projection/“subset” of a 3D game.

This game object currently has just the following two components:

- **Transform**

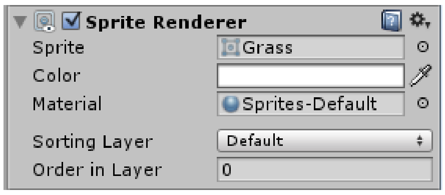


Transform

Position	X	0	Y	-5	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

All game objects in Unity has a Transform component attached, which has information about position, rotation and size/scale

- **Sprite Renderer**



☑ Sprite Renderer

Sprite	Grass
Color	
Material	Sprites-Default
Sorting Layer	Default
Order in Layer	0

This component draws the image in the game. Note the “Order in Layer” field, which controls the order of images drawn, and can in a 2D game be used to control which game objects are drawn on top of others, e.g. that player is drawn on top of background

Later we will add additional components to Grass, e.g. to ensure that the cannonball will stop when it hits the ground.

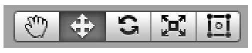
Position and size of grass

So far the grass is just a small box in lower part of the screen, so we need to place it as well as change the size.

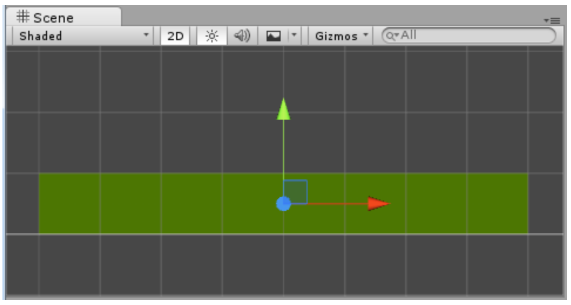
The position of a game element can be set in two different ways:

- **Use the "gizmo"**

  Select the "Pan" tool in toolbar at top of Unity editor window:

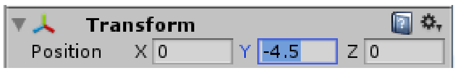
  This will show a "gizmo" with arrows in X and Y directions.

  By dragging the arrows, you can move the game object.

- **Enter X and Y coordinates**

  Another way to change the position, is to directly enter the X and Y coordinates for the game object in the Transform component, in this case changing Y coordinate to -4.5:

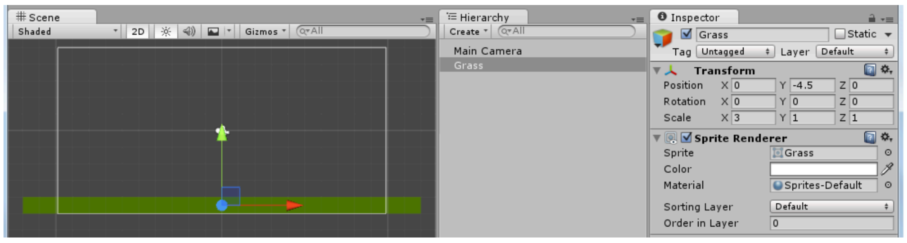
  

  Transform
  Position X 0 Y -4.5 Z 0

  This method is typically best when you are fine-tuning location of an object, after having placed it with the gizmo, as described above

The other buttons in the toolbar is used to rotate, scale and change size of objects, and generally works in the same way as positioning the object.

In similar way we can change size of the grass to fill width of screen, by changing Scale to 3 for X-axis:



Inspector
Grass Static
Tag Untagged Layer Default

Transform

Position	X	0	Y	-4.5	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	3	Y	1	Z	1

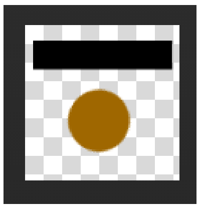
Sprite Renderer

Sprite	Grass
Color	
Material	Sprites-Default
Sorting Layer	Default
Order in Layer	0

Cannon graphics

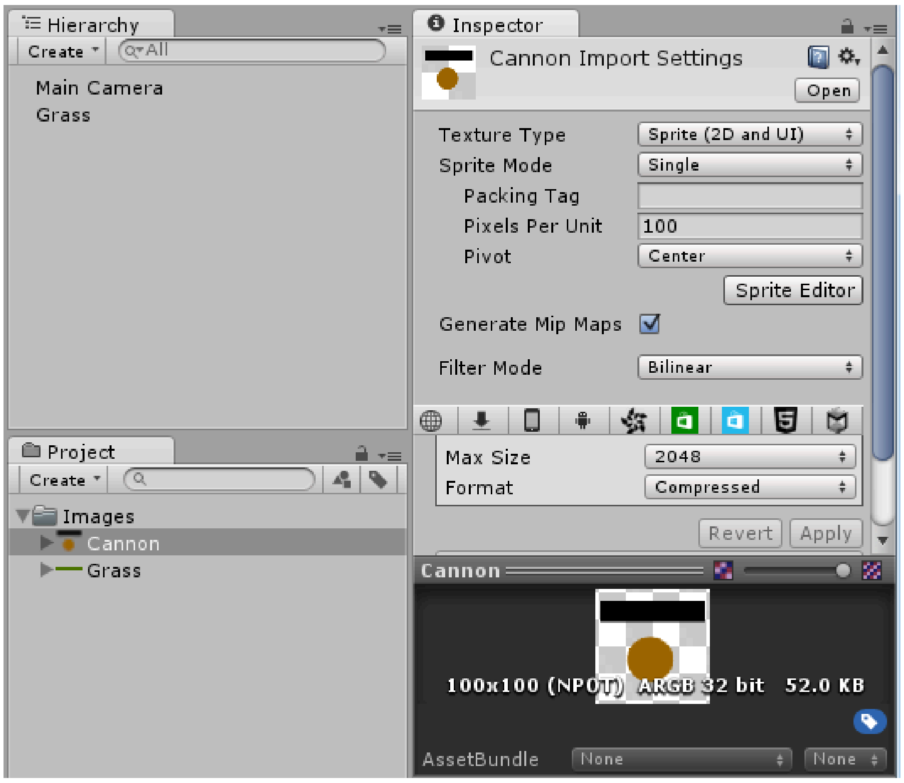
For the cannon, we need slightly more advanced graphics, as the cannon barrel needs to move independently of its wheels. In your paint application, create a new image of size 100x100 pixels, and draw a barrel (90x20 pixels) and wheel (diameter 40 pixels).

Note that it’s important that barrel and wheel are drawn separately, i.e. not touching each other.



The light chess background indicates that the background is transparent, which is important for Unity to identify the two separate images. Above cannon is kept very simple, you are of course welcome to add more details.

1. Import the image file into the Unity project, and name the imported asset “Cannon”:

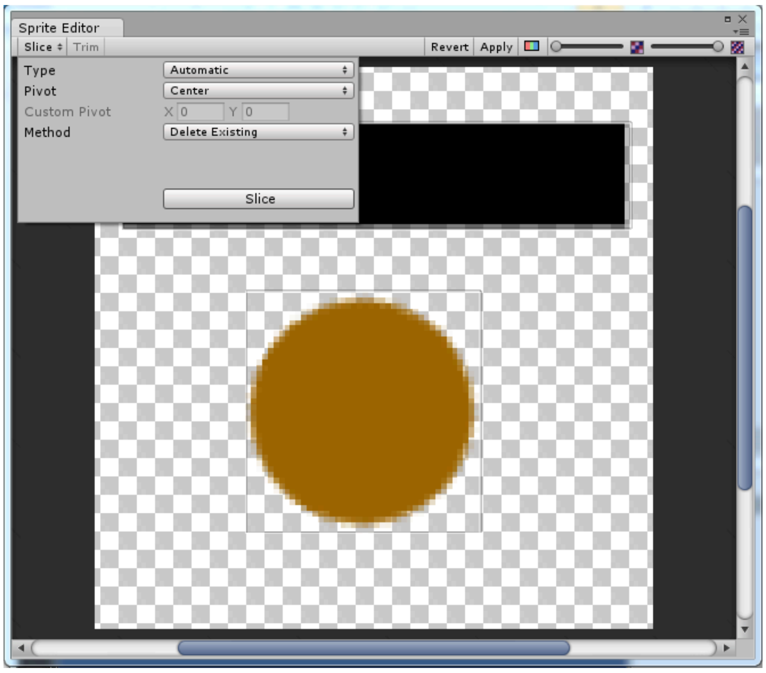


Cannon Import Settings

Texture Type: Sprite (2D and UI)
Sprite Mode: Single
Packing Tag:
Pixels Per Unit: 100
Pivot: Center
Generate Mip Maps: ✓
Filter Mode: Bilinear
Max Size: 2048
Format: Compressed

Cannon
100x100 (NPOT) ARGB 32 bit 52.0 KB
AssetBundle None None

2. As this image consists of two separate objects, select “Multiple” as “Sprite Mode” and click the “Apply” button.

3. Now click the “Sprite Editor” button, followed by “Slice”, to indicate which “slices” the image consists of:
   (TODO: image of showing where the Sprite Editor button is)
   

   Type: Automatic
   Pivot: Center
   Custom Pivot: X 0 Y 0
   Method: Delete Existing

   Just select type “Automatic” to have Unity automatically detect the cannon barrel and wheel in the image.

4. Click “Apply” button and close the Sprite Editor window.

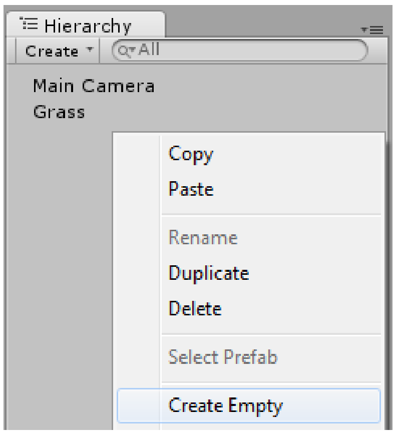
5. Select “Cannon” in the Project window, and expand it by clicking the small triangle on its left side. Unity has detected the two different objects in the image, which we will use soon to compose the cannon game object.

We could have chosen to make the cannon barrel and wheels as two separate images, however as it’s the same object, it makes sense to keep them in a single file. And then to show that Unity can automatically slice an image consisting of multiple objects, which is often the case e.g. with animated characters.

Cannon Game Object

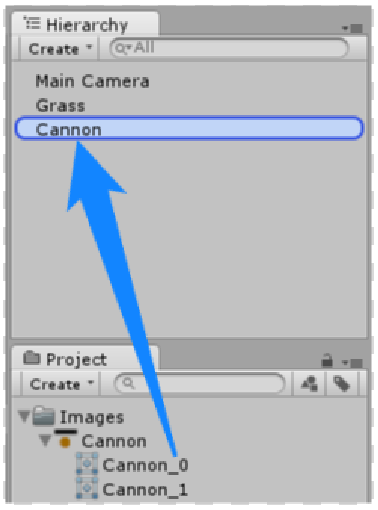
As the cannon consists of multiple objects, the process of creating the game object is different from when we created the grass.

1. Create an empty game object:



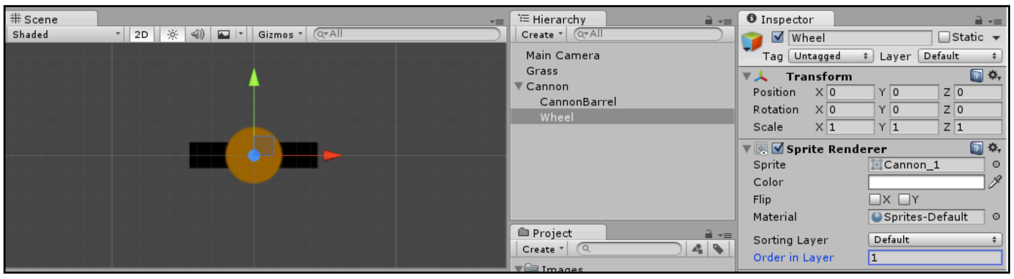
2. Select the created object in the Hierarchy window, press F2 (Windows) / Enter (Mac) and change the name to “Cannon”

3. Now drag and drop the “Cannon_0” image under the create “Cannon” game object



4. Also drag the “Cannon_1” image

5. Rename “Cannon_0” to “CannonBarrel” and “Cannon_1” to “Wheel”, so our cannon looks like this (doubleclick on “Cannon” to zoom in)

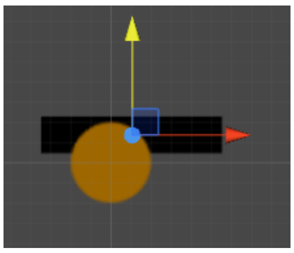


Wheel Static
Tag Untagged Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Sprite Renderer
Sprite Cannon_1
Color
Flip X Y
Material Sprites-Default
Sorting Layer Default
Order in Layer 1

If you cannot see the images for cannon, check that the coordinates are correct, i.e. (0,0) for both Cannon, CannonBarrel and Wheel

6. Set “Order in Layer” to “1” for Wheel, to ensure it’s drawn in front of CannonBarrel

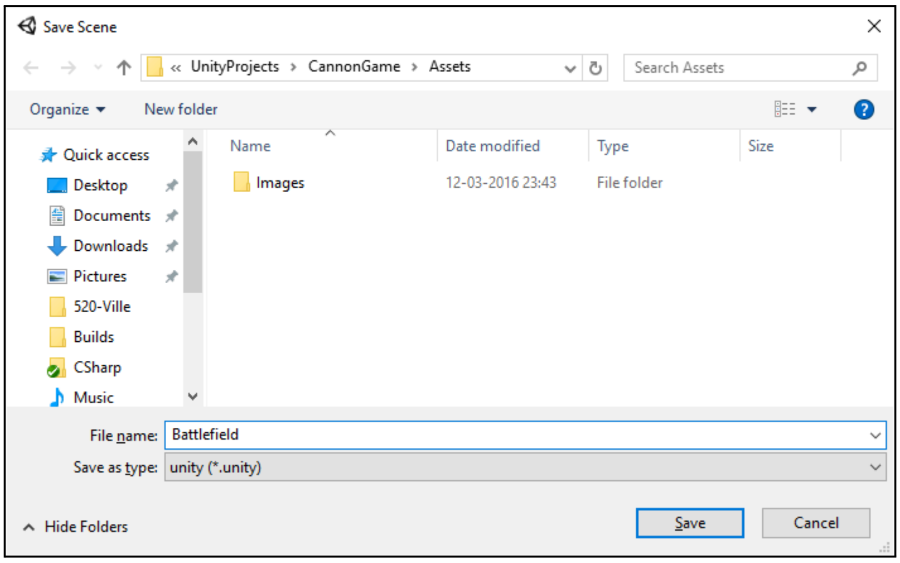
7. Move the cannon barrel to make it look like a cannon:



As such we’re done designing the cannon, but still need to place it correctly in the scene, to control it and not least be able to shoot cannonballs, which we will look closer at in next chapters.

Save project/scene

As the scene now contains a few objects, let’s use the opportunity to save it, by selecting **File→Save Scene** menu item, or simply press Ctrl+S (Windows) or Cmd+S (Mac), and then specify scene filename, in this case “Battlefield”:



Name	Date modified	Type	Size
Images	12-03-2016 23:43	File folder	

File name: Battlefield

Save as type: unity (*.unity)

Just save the scene file in “root” of the project. If working on larger projects with many scenes, it will be a good idea to structure scenes into subfolders.

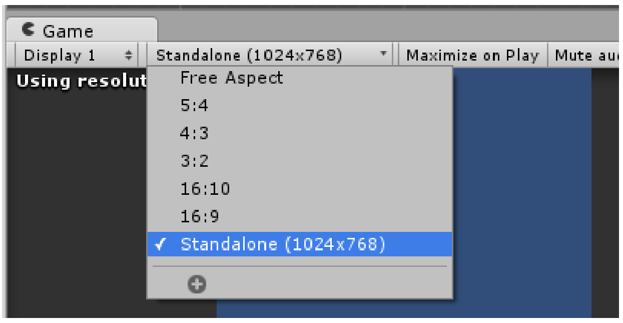
“Grouped” gameobjects

The cannon we have designed, consists of two individual parts, wheel and barrel. As they are part of the same object in game, we have chosen to make them sub-objects. This has e.g. the advantage that we can move the entire object (with wheel and barrel) at the same time.

Screen resolution of game in editor

When developing games, especially for mobile devices, you usually don't know the resolution of the screen where game is being played, which needs to be taken into consideration, e.g. by dynamically with code positioning objects based on the actual screen resolution.

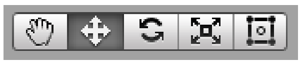
Unity allows you to set the screen resolution in editor, to see how the game looks in different resolutions, using the "Aspect" menu of the Game window. To keep it simple for now, just select "Standalone", as we currently are focusing on PC/Mac version of the game.



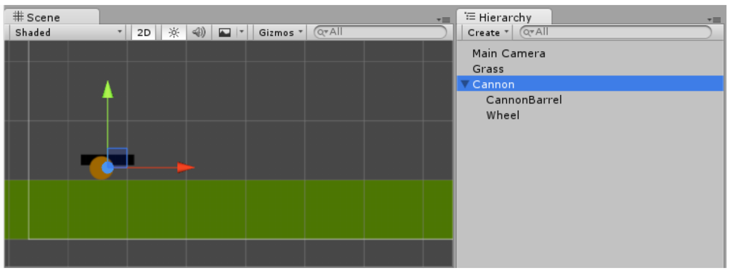
Position of the cannon

To place the cannon on the grass, do the following:

1. Select the toolbar button for moving objects.

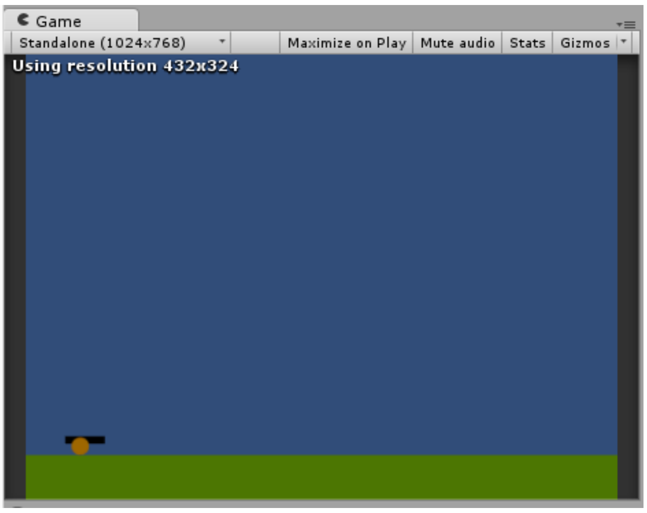


2. Select the Cannon gameobject in Hierarchy window and move it to lower left corner.



Note: move the "Cannon" gameobject, and not the wheel or barrel individually.

The cannon should now be located in lower left corner in the Game window:



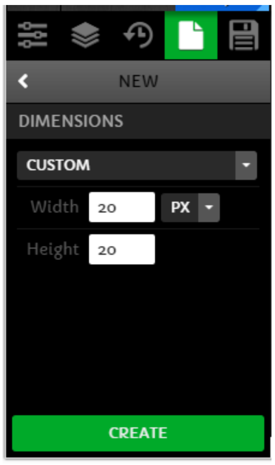
Cannonball

Before starting programming the game, we just need to add the cannonball. After all, the game will be slightly more fun if we actually can shoot something...

Start by drawing a cannonball in a size which fits the barrel, approx. 18 pixels in diameter.

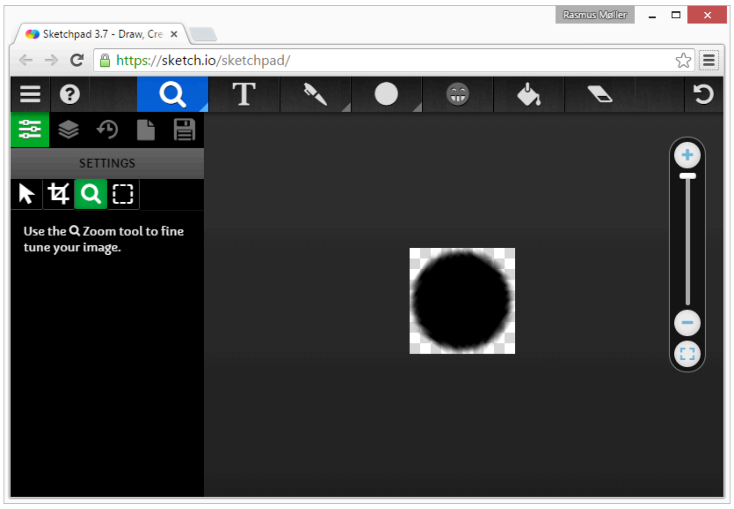
1. Open https://sketch.io/sketchpad/ (or a similar drawing tool), to draw a black circle.

2. Create a new image

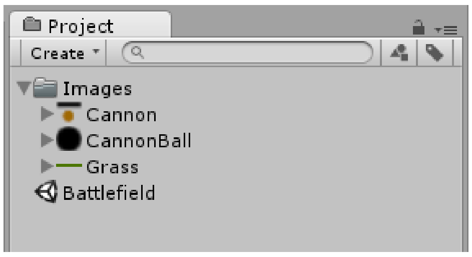


Width 20
Height 20

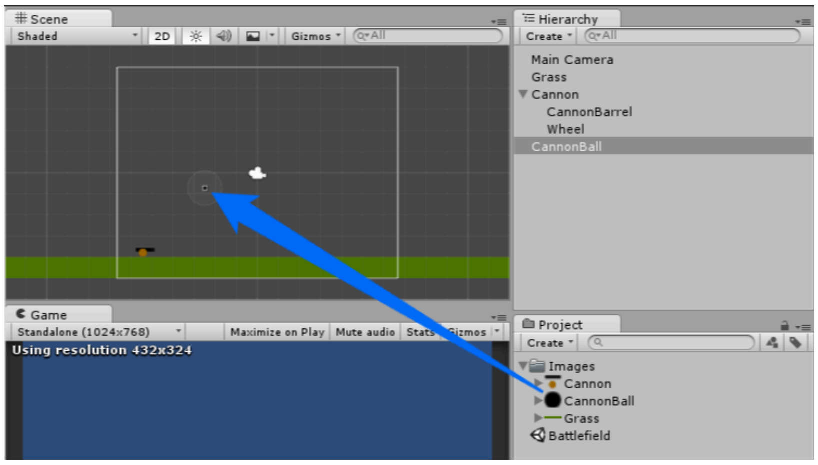
3. Draw a black circle, taking up almost entire image


Sketchpad 3.7 - Draw, Cre
https://sketch.io/sketchpad/
SETTINGS
Use the Zoom tool to fine tune your image.

4. Save the image as CannonBall.png, and drag it to the Images folder in Unity:



5. Drag the cannonball to the Scene window:



Test the ”game” in Play-mode

We have now added all the elements needed for the game, i.e. simple landscape, a cannon plus cannonball. Admittedly not a very interesting game yet, as a matter of fact, not even playable, but we’ll get to in a minute.

In order to test our game so far, press the ”Play”-button in the toolbar, alternatively press Ctrl+P/Cmd+P on keyboard.

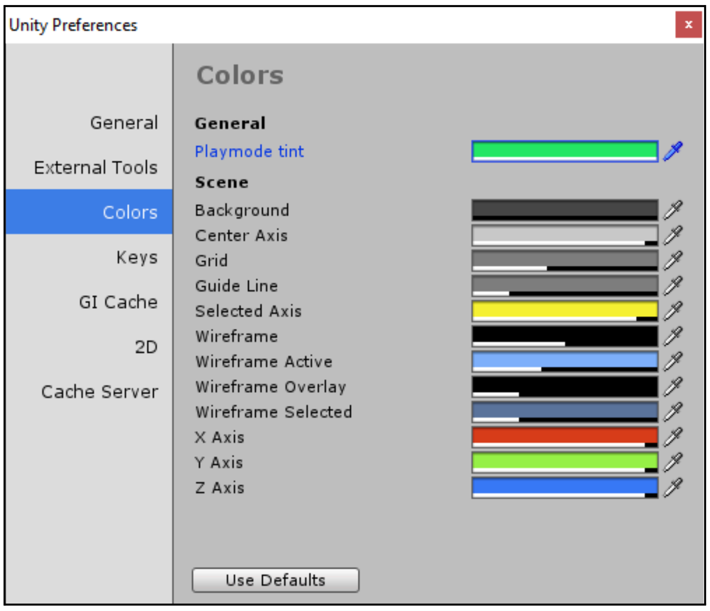
As our game objects at this state only consists of images, nothing will happen when playing the game, except you will notice that the Unity editor turns into a slightly darker color, which indicates that the application is now in “Play-mode”.

Next we’ll apply physics to the cannonball, so it will be affected by gravity.

Press the Play-mode button again to get back into edit mode.

Tip: Properties, e.g. position, size etc, can still be changed for game objects in game while in playmode, however when you leave playmode, objects will restore their original values. This can be useful for testing changes to gameplay while playing, however you need to remember which changes were done, if you want to apply some of them.

To make sure you are not by accident making changes while in playmode, it’s possible to modify the “Playmode tint” color in **Edit→Preferences...**


Unity Preferences

General | External Tools | Colors | Keys | GI Cache | 2D | Cache Server

Colors

General	
Playmode tint	🟩
Scene	
Background	⬛
Center Axis	⬜
Grid	▨
Guide Line	▨
Selected Axis	🟨
Wireframe	⬛
Wireframe Active	🟦
Wireframe Overlay	⬛
Wireframe Selected	🟦
X Axis	🟥
Y Axis	🟩
Z Axis	🟦

Use Defaults

My personal preference is a light green color, which clearly indicates that you are in playmode

Rigidbody - physical capabilities for game objects

One of the advantages of using a game engine like Unity for building your games, is the built-in “physics engine”, which simlutes physical capabilities as known from real life, e.g. that objects are affected by gravity. By applying physics to the cannonball, it will follow a more realistic (TODO: bane) when fired.

1. Select “CannonBall” in Hierachy window
2. Click “Add Component” button at the bottom of the Inspector window
3. Add “Rigidbody 2D” component

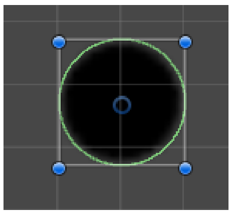
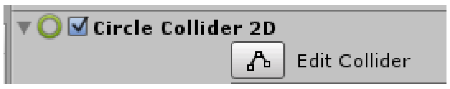
4. Enter Play-mode

The cannonball will now fall to the ground (and actually continue through the ground, as we haven't told it to stop…)

Colliders

In same way as physics capabilities were applied by adding the Rigidbody component, a "Collider" component will tell Unity to register when an object collides with other objects in the game, in this case the cannonball and ground, which will both have colliders attached.

1. Select the "CannonBall" game object

2. Click "Add Component" and select "Circle Collider 2D". If zooming in on the cannonball in Scene window, a green circle is visible around the cannonball. This is the collider boundary, i.e. where Unity will detect then the object touches/collides with other objects:
   
   In order to change the size of the collider, click the "Edit Collider" button for game object in the Inspector window:
   

3. Next select the "Grass" game object

4. Click "Add Component" and select "Box Collider 2D"

When entering playmode now, the cannonball will fall, but stop when it hits the ground.

Scripting / programming in C#

An important aspect of a game, beside the graphics, is the gameplay, i.e. the logic behind the game, story, how controls are working etc, which is where programming applies. In Unity it's possible to program the game logic using either C# or UnityScript (a variant of the JavaScript programming language).

More than 80% of all games developed using Unity are using C# as programming language, and documentation and support is better for C# compared to UnityScript. For these reasons the code in this book is written using C#.

The code in the book will be described briefly as we go along, and should be possible for beginners to understand. However this book is not a full introduction to programming, and I also believe that programming is best learnt by actually trying, and looking up the errors on internet as they happen, and incrementally improve this way.

Some notes about programming

This will be a very brief introduction to programming for beginners. If you are already familiar with programming, you can skip to the next section.

Programming is the way to describe for a computer how to solve a given task, which can be performing calculations, e.g. in financial systems/calculating capabilities for constructions/buildings, or in our case describing how the game should react to keypresses in order to control our cannons.

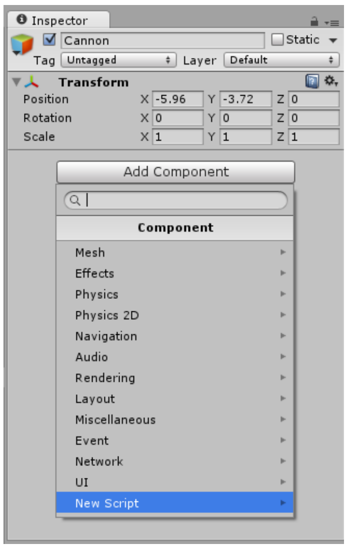
By being able to describe a problem in code, usually also means that you also understand the problem, so in my view learning to program also improves a person's problem-solving skills.

Control the cannon using script

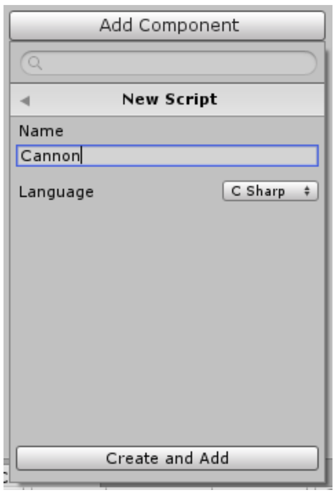
For our game, we will control the cannon using arrow keys on the keyboard.

1. Select "Cannon" game object in Hierachy window

2. Click the "Add Component" button, and choose "New Script" (at bottom):



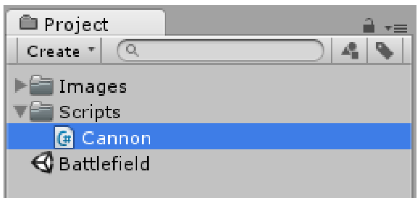
3. Name script “Cannon”, select “C Sharp” as programming language and click the “Create and Add” button:



Name
Cannon

Language C Sharp

The script has now been added to the project and can be found in the Projects window:



In order to keep the files easier maintainable, especially when working on larger projects, it’s recommended to create a folder structure for the files in project, in this case a “Scripts” folder for scripts in project. Move the created script into the folder, as shown in screenshot above.

Double-click on the Cannon script to open it in MonoDevelop/Visual Studio, and write following code (don’t write line numbers, will be used later to describe the code):

```
1: using UnityEngine;
2:
3: public class Cannon : MonoBehaviour
4: {
5:     public GameObject CannonBarrel;
6:
7:     void RotateBarrel (int degrees)
8:     {
9:             CannonBarrel.transform.Rotate (new Vector3 (0, 0, degrees));
10:    }
11:
12:    // Use this for initialization
13:    void Start ()
14:    {
15:
16:    }
17:
18:    // Update is called once per frame
19:    void Update ()
20:    {
21:            if (Input.GetKey (KeyCode.UpArrow))
22:          {
23:                  RotateBarrel (1);
24:          }
25:
26:          else if (Input.GetKey (KeyCode.DownArrow))
27:          {
28:                  RotateBarrel (-1);
29:          }
30:     }
31: }
```

A brief explanation of the code follows.

Important: When programming, it's necessary to follow some rules for the "syntax"/structure of the code. For beginners in C#, especially the "curly brackets" { and }, which in C# are used to describe a "scope" of code. If not writing these curly brackets correctly, the computer won't be able to correctly interpret the code, and will report an error when trying to play the game.

- *Line 1:* By including `using UnityEngine` in the code, we tell the computer to use elements from the UnityEngine "namespace". A namespace is used in C#, and other programming languages, to ensure unique naming. In this case we want to use elements/classes from Unity, which all are located under the UnityEngine namespace.

- *Line 3:* All classes in Unity which contains logic for a gameobject, must "inherit" from the `MonoBehaviour` class, which contains declarations for e.g. the Update() method, which we use in the example.

  Note: Give the C# class same name as the file, with same capitalization (lowercase/uppercase) of characters.

- *Line 5:* As we want to turn the cannon barrel (and not the entire cannon), we are creating a reference to this, by creating a public variable, which will be available from the editor.

- *Line 8:* So far the `Start()` method is empty, but it will later contain code which will be run when game starts, so we leave the method in the code for now. If you don't need some code, you should normally remove the code, as it can have impact on the performance of the game.

- *Line 14:* The method `Update()` is being called by Unity for each frame, i.e. each time the game updates the graphics, normally at least 50 times per second. For this example, we check if the player has pressed one of the arrow keys on keyboard, and call `RotateBarrel()` method in that case.

- *Line 27:* Here the RotateBarrel method is defined, which rotates the cannon barrel around the Z-axis, i.e. rotating either up or down. `CannonBarrel.transform` returns a reference to the "Transform" component of the CannonBarrel object, i.e. where we can control position, rotation etc.

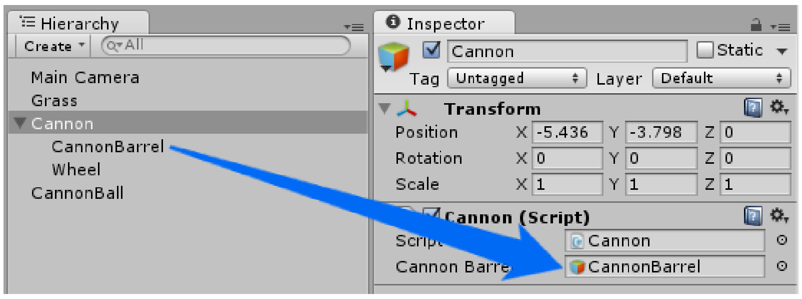
  By defining the method RotateBarrel, we can both reuse the code, and also makes the code easier to read, when using meaningful names for the methods, describing the purpose of the method, in this case rotating the barrel.

Switch back to Unity editor again and select the Cannon object. The public variable CannonBarrel from the code, is now available in the Inspector window for the gameobject, so we can tell Unity to use the CannonBarrel object.

Do this by

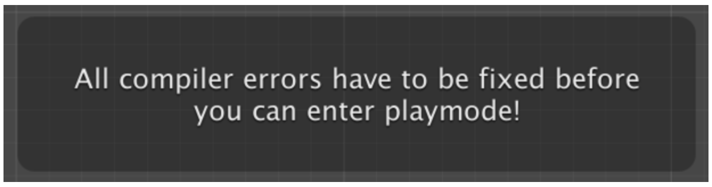
1. Selecting Cannon objektet

2. Using mouse, drag CannonBarrel object to the "Cannon Barrel" field in Inspector window, as shown in the figure below. It is important not to release the mouse button until you have dragged the object all the way. This might cause a little trouble the first couple of times (and likely also later…), alternatively click the small circle to the right of the "Cannon Barrel" field in the inspector window, to select the object.



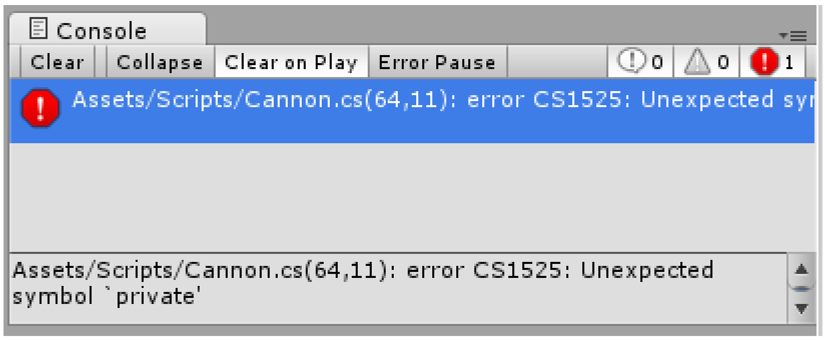
Start the game again in playmode, and raise/lower the cannon using arrowkeys on keyboard.

Tip: If the computer cannot "understand" the code, Unity will show an error similar to this:



All compiler errors have to be fixed before you can enter playmode!

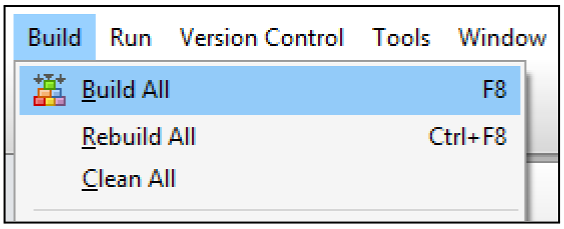
At the same time, the Console window will show additional information about the error, e.g.:



```
Assets/Scripts/Cannon.cs(64,11): error CS1525: Unexpected sy
Assets/Scripts/Cannon.cs(64,11): error CS1525: Unexpected
symbol `private'
```

Double-click on the error message in the Console-window to show the code in either MonoDevelop or Visual Studio.

It's also possible in MonoDevelop to check your code for "syntax errors" (spelling errors), by choosing menu item **Build**→**Build All**:



This will highlight any errors in the code.

Some notes on code, errors and tools

Contrary to natural/spoken languages, where it's normally possible to understand the meaning, even if there are spelling error, programming languages must be written precisely, otherwise the computer will report errors. Writing correct programs means following the syntax/structure of language, as well as not having any typos in the words used for programming.

Typical compiler-errors when using the C# programming language:

Error	**Example**

Curly brackets doesn't match	`void Update ()`
	`{`
	`    if (Input.GetKey (KeyUp))`
	`    {`
	`        RotateBarrel (1);`
	`    }`
	`    else if (Input.GetKey (KeyDown))`
	`    {`
	`        RotateBarrel (-1);`
	`    ← ERROR!`

	`}` In this case, the KeyDown block of code hasn't been properly closed with a curly bracket.

Normal parentheses doesn't match	`if (Input.GetKey (KeyUp)  ← ERROR!`
	`{`
	`   RotateBarrel (1);`
	`}`
	
	The first line has two left parentheses, but only one right, which means that we need to add one additional right parenthesis at the end of line 1, to make the code compile.

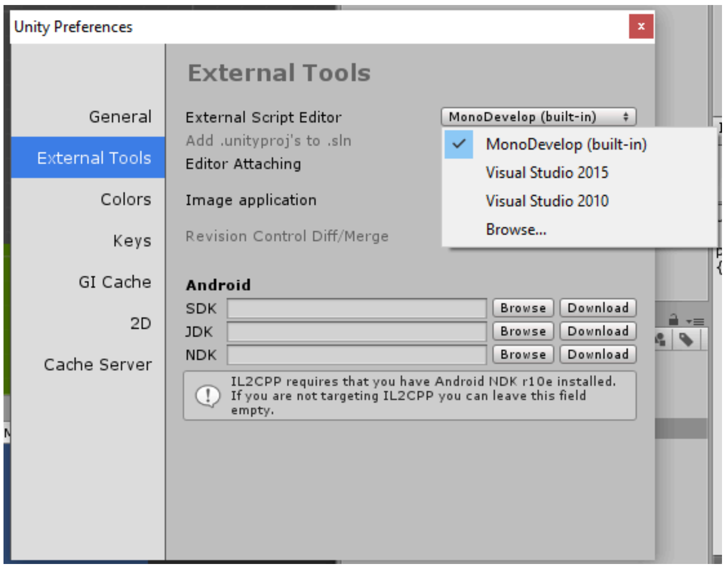
Missing semicolon	`if (Input.GetKey (KeyUp))`
	`{`
	`   RotateBarrel (1)  ← ERROR!`
	`}`
	
	Similar to sentences in natural languages are ending with a period ".", statements in C# must end with a semicolon ";" to tell the computer that we're starting a new statement.
	
	In this example, we therefore must add the missing semicolon to end of line 3.

Incorrect use of lower- and uppercase letters	`void update ()  ← ERROR!`
	`{`
	`   if (input.GetKey (KeyUp))  ← ERROR!`
	`   {`
	`      Rotatebarrel (1);  ← ERROR!`
	`   }`
	`}`
	
	C# programming language is "case-sensitive", i.e. lower- and uppercase letters are different. In this example "update" should have been "Update", "input" -> "Input" and "Rotatebarrel" -> "RotateBarrel".
	
	In this case Unity would have been able to detect some of the errors, however not "update" as this is not a syntax error. But since Unity will call the "Update" method (with uppercase beginning letter), the code wouldn't get executed.

MonoDevelop, Visual Studio or another editor

When installing Unity on Windows you can choose between MonoDevelop (installed by default with Unity) and Visual Studio. On Mac it's only possible to install MonoDevelop.

However it’s possible to use your favorite editor with Unity, which you can specify using the menu item **Edit → Preferences...**


Unity Preferences

General
External Tools
Colors
Keys
GI Cache
2D
Cache Server

External Tools

External Script Editor: MonoDevelop (built-in)
Add .unityproj's to .sln
Editor Attaching
Image application
Revision Control Diff/Merge

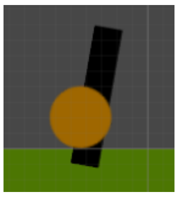
✓ MonoDevelop (built-in)
Visual Studio 2015
Visual Studio 2010
Browse...

Android
SDK [Browse] [Download]
JDK [Browse] [Download]
NDK [Browse] [Download]

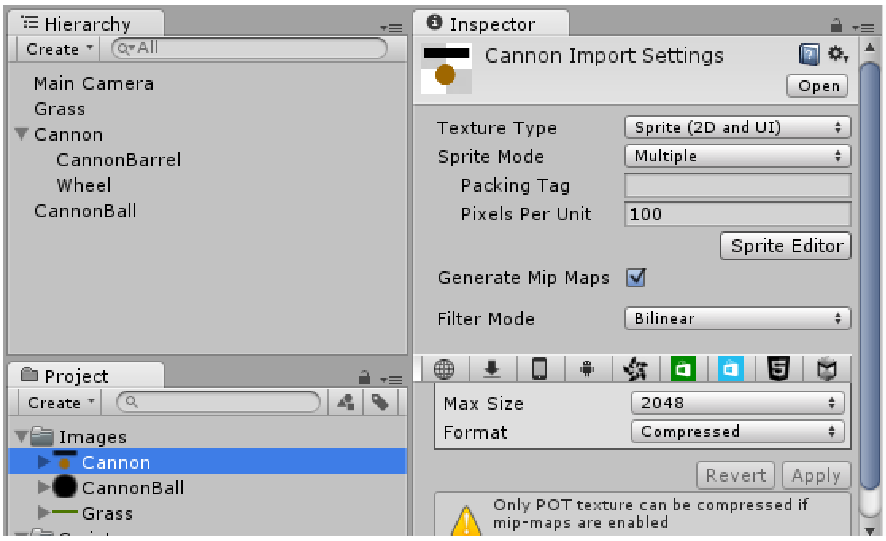
IL2CPP requires that you have Android NDK r10e installed. If you are not targeting IL2CPP you can leave this field empty.

Testing and fixing first issue in game

When now playing the game in playmode, you should be able to control the rotation of the barrel using up and down arrow keys, however you’ll see that we need to adjust the “pivot point” for the barrel backwards, so it rotates where the wheel is attached. Currently it will look something like this when rotating:



1. Select the Cannon sprite in the Project window:



Hierarchy: Main Camera, Grass, Cannon, CannonBarrel, Wheel, CannonBall

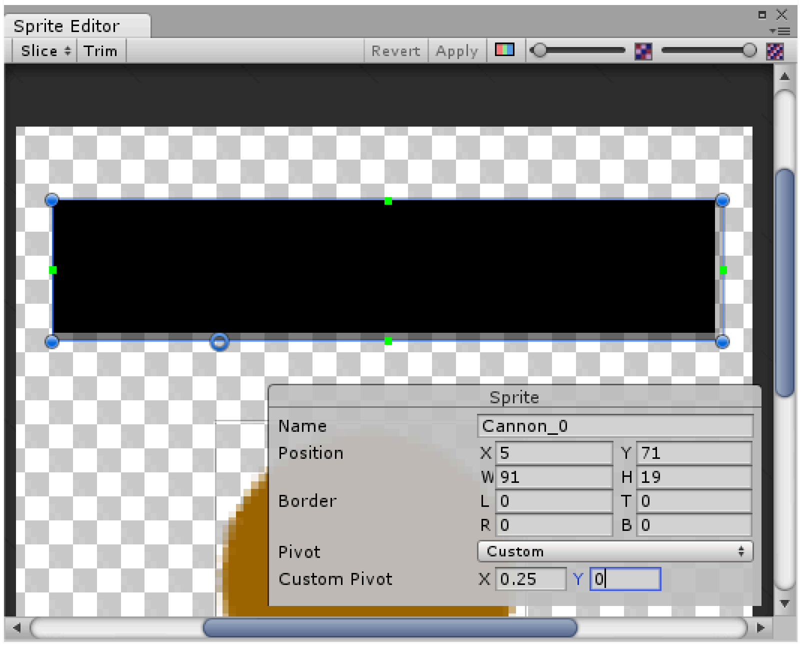
Project: Images, Cannon, CannonBall, Grass

Inspector — Cannon Import Settings

Texture Type: Sprite (2D and UI)
Sprite Mode: Multiple
Packing Tag:
Pixels Per Unit: 100
Generate Mip Maps: ✓
Filter Mode: Bilinear
Max Size: 2048
Format: Compressed

Only POT texture can be compressed if mip-maps are enabled

2. Click the “Sprite Editor” button and set Pivot to 0.25 on the X-axis and 0 on the Y-axis, click Apply and close the Sprite Editor window.



Sprite

Name: Cannon_0
Position: X 5, Y 71, W 91, H 19
Border: L 0, T 0, R 0, B 0
Pivot: Custom
Custom Pivot: X 0.25, Y 0

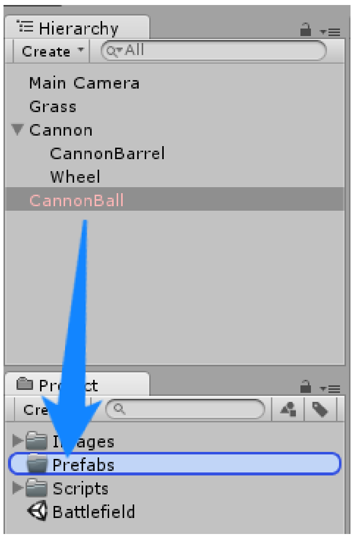
3. This however means that the cannon barrel will be moved a bit forward, so we need to adjust the position for it.

Fire cannonball - “Prefabs”

Next step in our game is to be able to fire cannonballs. This is done by creating multiple instances/copies of the same object. In Unity this is done by making a gameobject into a “prefab”.

1. Create a new folder “Prefabs” in the Project window.

2. Drag the CannonBall object to the created folder:



3. The CannonBall gameobject in the scene is now written in blue text, which indidates that it's an instance of a prefab. In practice this means that all gameobjects based on the same prefab, will share the settings from the prefab, e.g. size.

Now add the following code to the Cannon.cs file, to fire the cannonball.

Add the following lines to the beginning of the Cannon class, to make a reference to the cannonball prefab and also set the default force used for shooting the cannonball:

```
public GameObject CannonBallPrefab;
private float _firePower = 10f;
```

Plus a FireCannon method, which will instantiate a cannonball and apply a physical force to give it speed:

```
private void FireCannon (float power)
{
    Vector2 position =
        CannonBarrel.transform.position + CannonBarrel.transform.right * 0.8f;

    GameObject cannonBall = Instantiate (
        CannonBallPrefab,
        position,
        CannonBarrel.transform.rotation) as GameObject;

    cannonBall.GetComponent<Rigidbody2D> ().AddForce (
        CannonBarrel.transform.right * power,
        ForceMode2D.Impulse);
    Destroy (cannonBall.gameObject, 30);
}
```

Brief description of the code:

- Find the position where the cannonball should be instantiated, in this case the pivot position of the cannonbarrel (`CannonBarrel.transform.position`) plus 0.8 x length of barrel. Although the position is of type `Vector2` (which normally would indicate a direction), we're in this case just using it as position.
- Next we call `Instantiate()` to create a copy of the cannonball prefab as a gameobject.
- Shooting the cannonball is done by the `AddForce()` method, which is a part of the physics engine in Unity. As the method name indicates, `AddForce()` will apply a physical force to an object, which results in the cannonball flying towards `CannonBarrel.transform.right` (relative to the current rotation of the cannon barrel).
- Finally the `Destroy()` method is called, which will remove the cannon ball after 30 seconds. Note that this is not the most effective way to clean up objects, as this can result in memory fragmentation. The solution to this is using an "object pool" for game objects which are created and destroyed often, however this won't be covered in this book (just search internet for more information on object pooling in Unity)

As it can be seen from even this relatively simple game, we're using both coordinate system, vectors and physics. So developing games can be a way to practice what you have learned in math and physics in school :)

Above we created the `FireCannon()` method, which we just need to call, using some additional mode in `Update()`

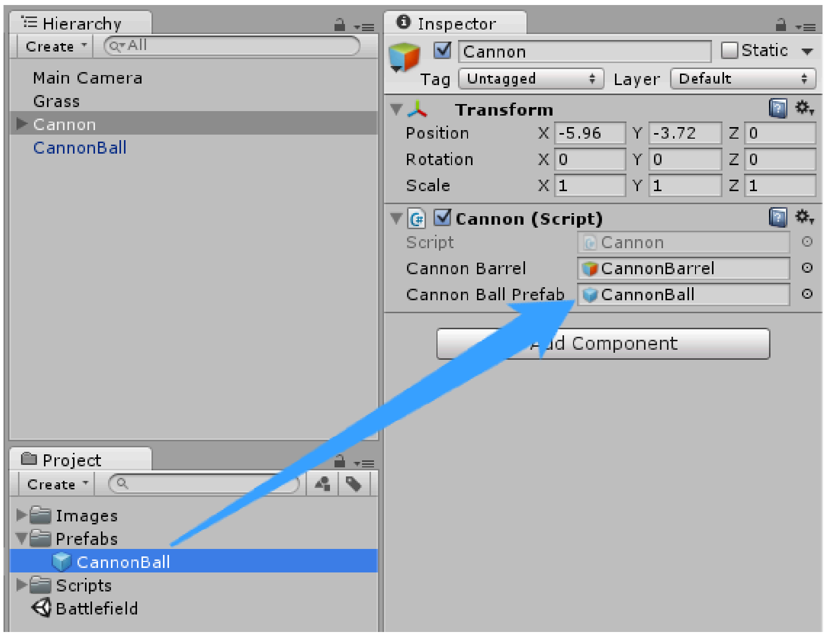
```
else if (Input.GetKeyUp (KeyCode.Space))
{
    FireCannon (_firePower);
}
```

Note that we use Input.GetKeyUp() instead of Input.GetKey(). This results in the cannon ball being fired when the spacebar key is released, and not constantly while being pressed down.

Switch back to the Unity editor, in order to specify our prefab.

1. Select Cannon object in the Hierachy window

2. Drag the “CannonBall” prefab to the “Cannon Ball Prefab” field



Main Camera
Grass
Cannon
CannonBall

Cannon
Tag Untagged Layer Default

Transform

Position	X -5.96	Y -3.72	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Cannon (Script)

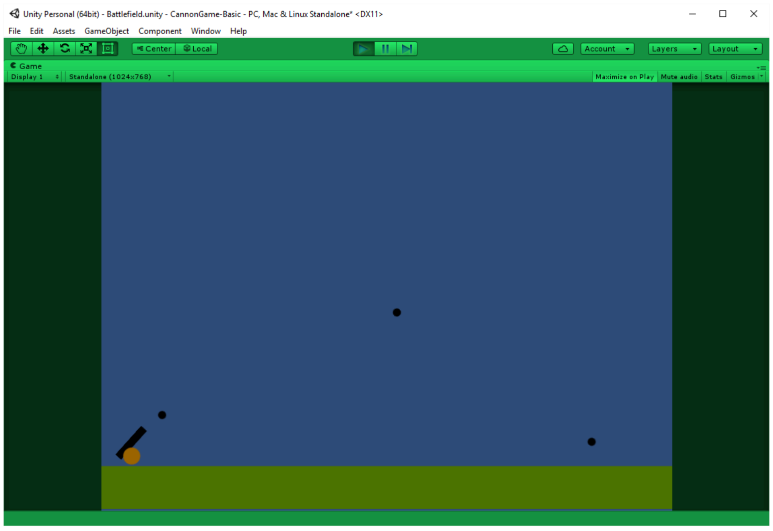
Script	Cannon
Cannon Barrel	CannonBarrel
Cannon Ball Prefab	CannonBall

Images
Prefabs
CannonBall
Scripts
Battlefield

(Alternatively click the small circle to the right of the “Cannon Ball Prefab” field and select the prefab)

3. Delete the existing CannonBall object in the Hierarchy window, as new instances will be created from code.

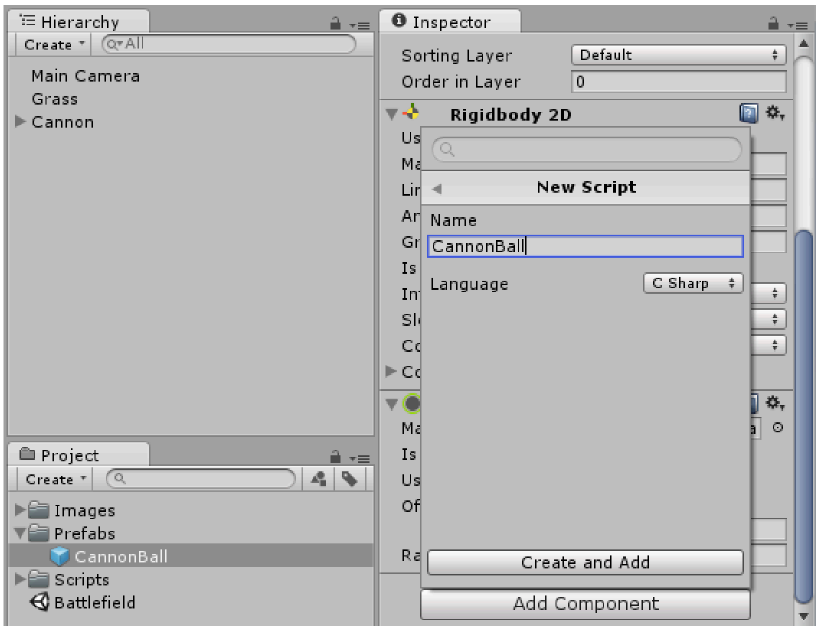
When playing the game, it’s now possible to fire cannonballs by pressing spacebar on the keyboard.


Unity Personal (64bit) - Battlefield.unity - CannonGame-Basic - PC, Mac & Linux Standalone* <DX11>
File Edit Assets GameObject Component Window Help
Center
Local
Account
Layers
Layout
Game
Display 1
Standalone (1024x768)
Maximize on Play
Mute audio
Stats
Gizmos

Cannonball physics

As you can see, the cannonball gets fired and it’s trajectory is affected by gravity so it will eventually hit the ground, which gives our game a realistic feeling. However when the cannonball hits the ground, it will continue rolling, which on the other hand is both unrealistic and inconvenient, as it means we would be able to simply hit somewhere in front of the other player, and let the cannonball roll forward to hit the other players cannon.

Unity allows us to adjust the physical capabilities of objects using “PhysicsMaterial”, however the simplest way to stop the cannonball when it hits the ground, is using code. Add a CannonBall script to the CannonBall prefab:



Sorting Layer: Default
Order in Layer: 0
Rigidbody 2D
New Script
Name: CannonBall
Language: C Sharp
Create and Add

And insert the following code:

```
using UnityEngine;

public class CannonBall : MonoBehaviour
{
    void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.name == "Grass")
        {
            GetComponent<Rigidbody2D>().isKinematic = true;
            GetComponent<Rigidbody2D>().velocity = Vector3.zero;
        }
    }
}
```

Brief explanation of the code:

- OnCollisionEnter2D is the method which Unity calls when our cannonball hits another “collider”. In this case we’re simply checking by name “Grass”

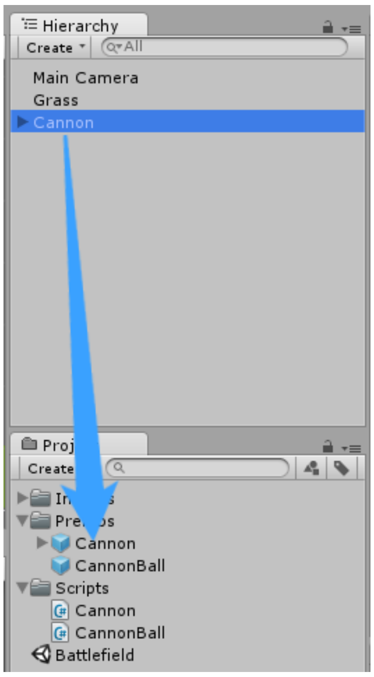
- Then physics for the cannonball is disabled by setting `isKinematic = true` and setting`velocity = Vector3.zero` to stop the cannonball. The reason for disabling physics, is to avoid other cannonballs hitting this one, and making it move.

Again, remember to move the CannonBall.cs script to the Scripts folder, to keep the structure for the files in your project.

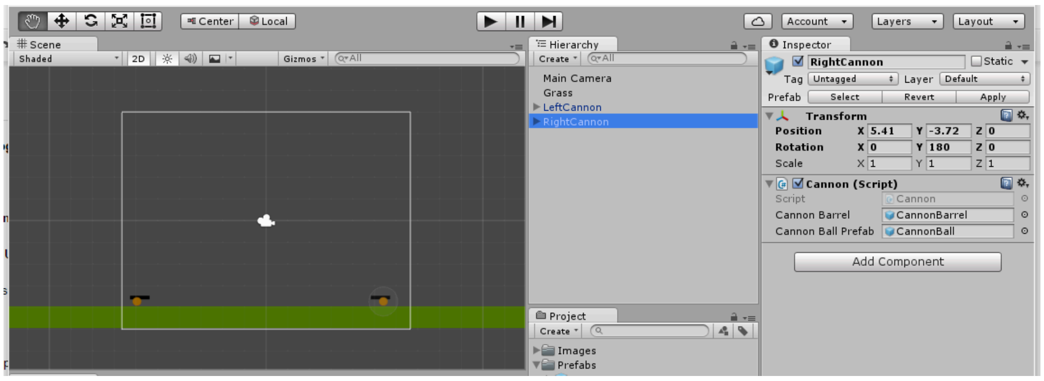
Two players, points and simple UI

The game now consists of the basic elements needed, i.e. a cannon with ability to control and fire a cannonball. To make the game slightly more challenging and interesting, we would like to allow two players to play against each other, which requires some modifications to the game:

1. Turn the Cannon into a prefab:



2. Insert another instance of the cannon, by dragging Cannon from the Project window to the scene and place it in the right side. Rename the cannon gameobjects to “LeftCannon” and “RightCannon”. The cannon on right side should be rotated 180 degrees around the Y-axis:



RightCannon Static
Tag Untagged Layer Default
Transform
Position X 5.41 Y -3.72 Z 0
Rotation X 0 Y 180 Z 0
Scale X 1 Y 1 Z 1
Cannon (Script)
Script Cannon
Cannon Barrel CannonBarrel
Cannon Ball Prefab CannonBall

It's possible to play the game now with both cannons, however they will react to the same keypresses on keyboard, so we need to configure them individually. So far we have simply checked for hardcoded keys, so we need to modify the Cannon.cs script.

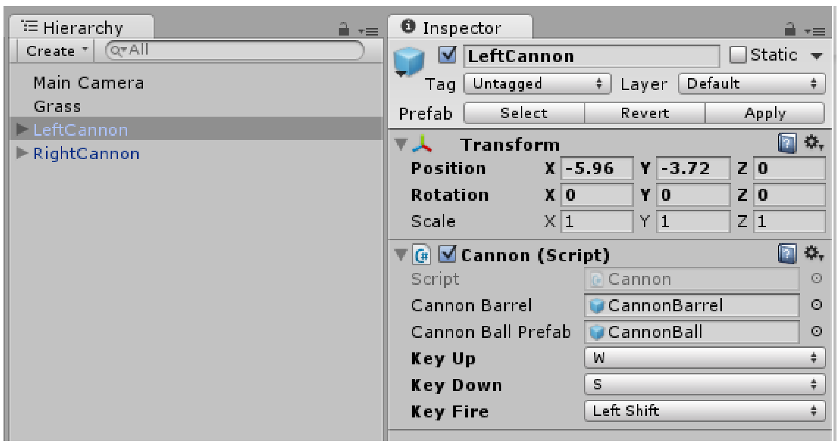
First insert following fields in the beginning of the Cannon class:

```
public KeyCode KeyUp;
public KeyCode KeyDown;
public KeyCode KeyFire;
```

This allows us from the editor to configure keys per cannon. Next modify the Update() method to use these configured keys:

```
// Update is called once per frame
void Update ()
{
    if (Input.GetKey (KeyUp))
    {
        RotateBarrel (1);
    }
    else if (Input.GetKey (KeyDown))
    {
        RotateBarrel (-1);
    }
    else if (Input.GetKeyUp (KeyFire))
    {
        FireCannon (_firePower);
    }
}
```

Lastly we just need to choose the keys in the editor.


Hierarchy: Main Camera, Grass, LeftCannon, RightCannon

Inspector: LeftCannon
Tag: Untagged, Layer: Default
Transform
Position X -5.96 Y -3.72 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Cannon (Script)
Script: Cannon
Cannon Barrel: CannonBarrel
Cannon Ball Prefab: CannonBall
Key Up: W
Key Down: S
Key Fire: Left Shift

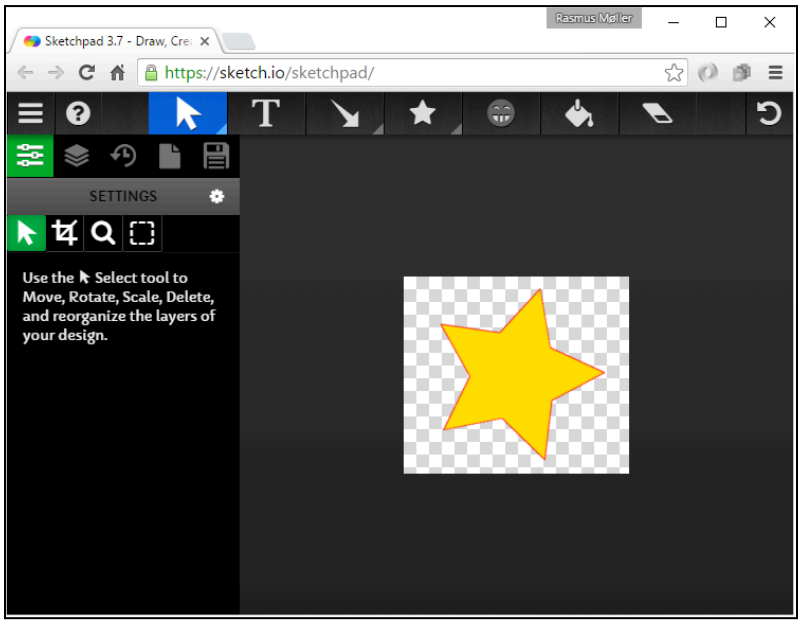
Suggestion for keys:

	LeftCannon	**RightCannon**
Key Up	W	Up Arrow

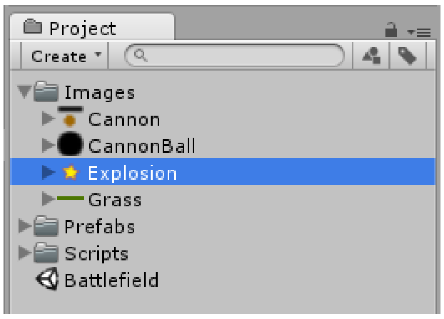
Key Down	S	Down Arrow
Key Fire	Left Shift	Right Shift

Explosions (!)

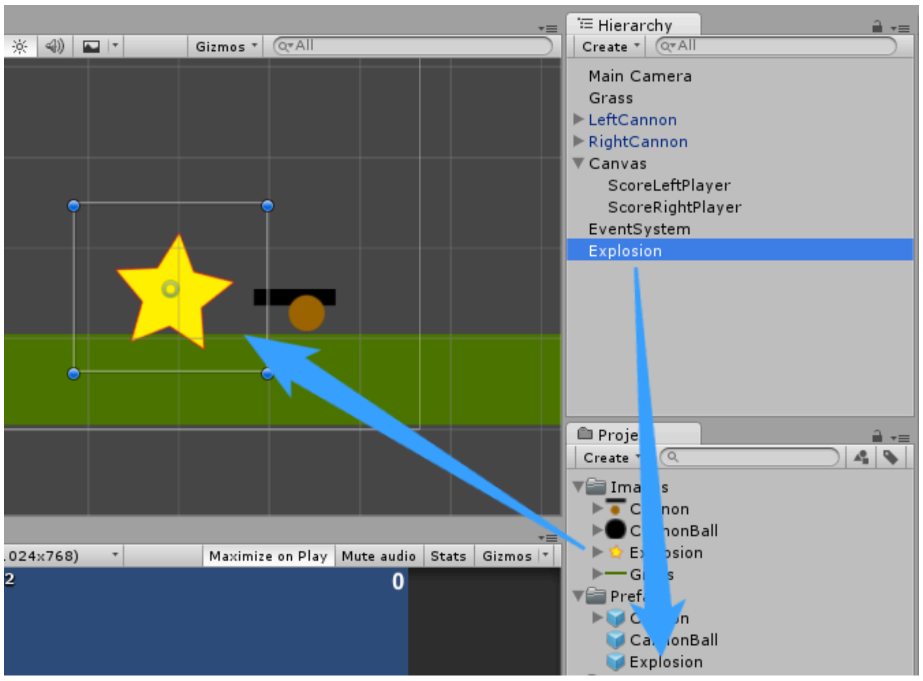
And of course we need explosions, although relatively simple ones. Draw a star in your paint application, of approx. size 210 x 180 pixels:


Sketchpad 3.7 - Draw, Cre
https://sketch.io/sketchpad/
SETTINGS
Use the Select tool to Move, Rotate, Scale, Delete, and reorganize the layers of your design.

Save the file and drag it to the Images folder:



As we will show an explosion for every time we hit the opponent cannon, we should make a prefab for the explosion image. Insert the image into the Scene window, and adjust the size of it according to your cannon. Then drag it from Scene window to the Prefabs folder in Projects window:



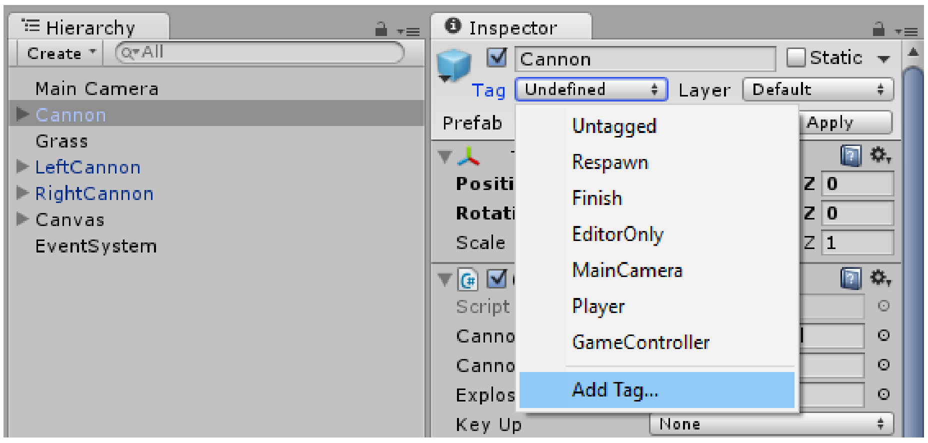
You can now delete the explosion gameobject from the scene, as we will use the prefab from now on.

Collider and "tag" the cannon

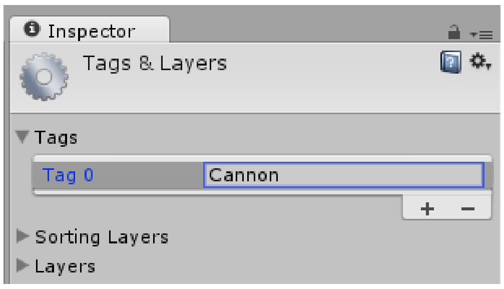
To have Unity detect that the cannonball has hit a cannon, we add a collider to the cannon.

1. Start by dragging the cannon prefab into the scene, as we need to update the prefab. This is how you currently modify a prefab in Unity, and we cannot use one of the existing cannon game objects already in the scene, as we have e.g. set keys for those, which would override the values in prefab.

2. In order to easily "recognize" a cannon in code, we add a "tag" on the Cannon prefab. Select the Cannon gameobject in Hierarchy window, click on the "Tag" dropdown list and select "Add Tag...":

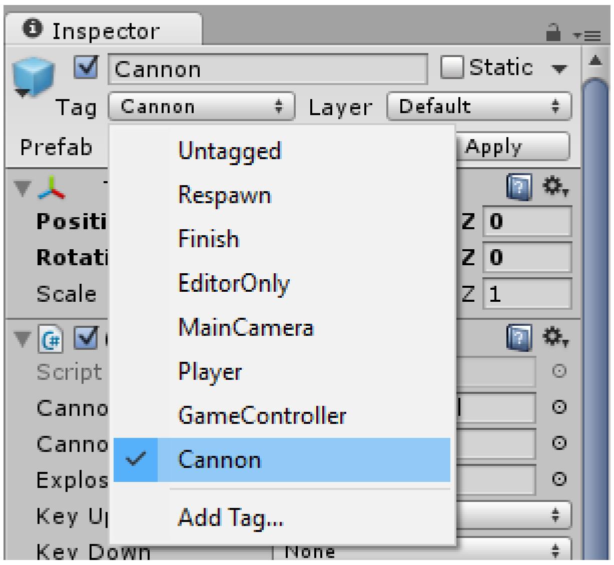


3. Add “Cannon” tag:



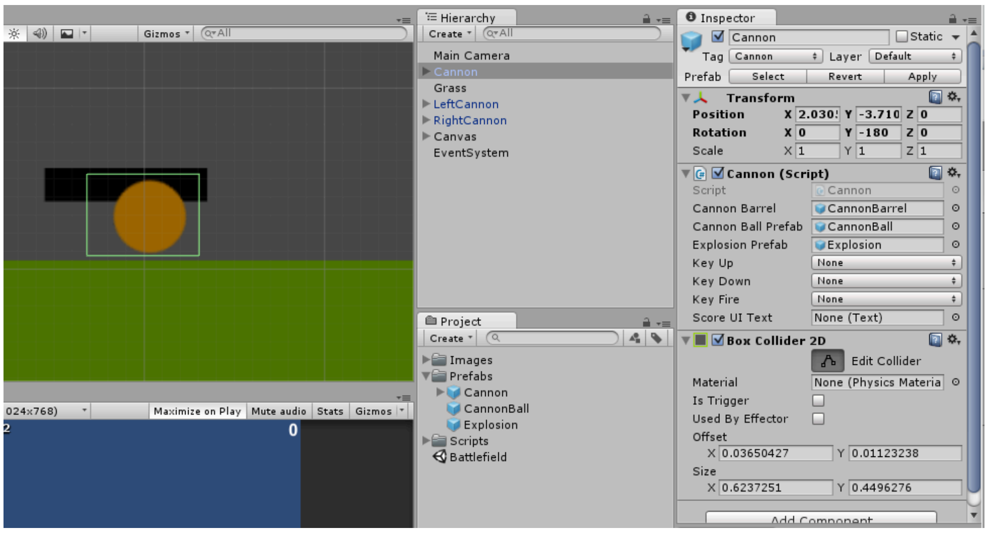
Tag 0: Cannon

4. And choose this for the Cannon object:



Cannon
Tag: Cannon

5. Now add a “Collider 2D” component to the cannon, click the “Edit Collider” button to specify the size of the area it’s possible to hit for the cannon:



Cannon
Tag: Cannon Layer: Default
Prefab: Select Revert Apply

Transform
Position X 2.030 Y -3.710 Z 0
Rotation X 0 Y -180 Z 0
Scale X 1 Y 1 Z 1

Cannon (Script)
Script: Cannon
Cannon Barrel: CannonBarrel
Cannon Ball Prefab: CannonBall
Explosion Prefab: Explosion
Key Up: None
Key Down: None
Key Fire: None
Score UI Text: None (Text)

Box Collider 2D
Edit Collider
Material: None (Physics Materia
Is Trigger: ☐
Used By Effector: ☐
Offset X 0.03650427 Y 0.01123238
Size X 0.6237251 Y 0.4496276

The green “collider box” controls when Unity detects the cannon as being hit, and our code gets called. This way you as game developer/designer can control how easy/hard it should be to hit the opponent.

6. (Cannot do this yet, as the ExplosionPrefab field isn't defined until page 49)
   At the same time update the "Explosion Prefab" field on the Cannon, so we can use this from code.

7. Click the"Apply" button to update the prefab, which then automatically also updates the two existing cannons.

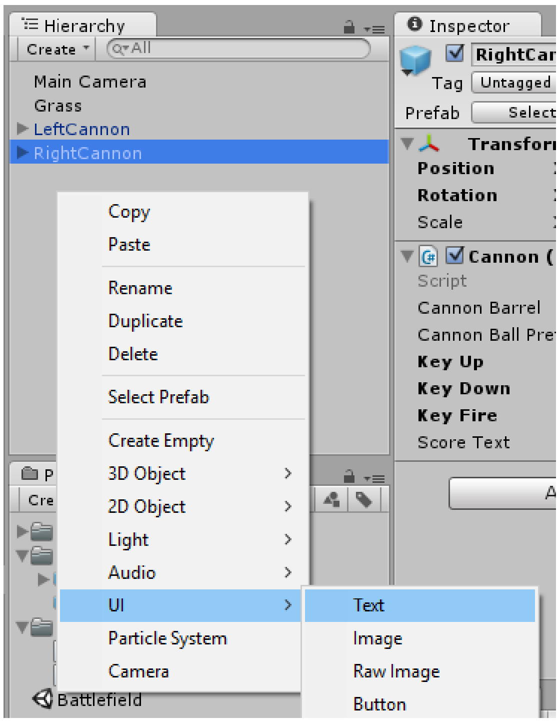
8. Delete the temporary Cannon object again from the scene, as we now have updated the prefab.

When selecting one of the existing Cannon objects ("LeftCannon" or "RightCannon"), you can see that they now have the "tag" value updated and both have a collider added from the prefab.

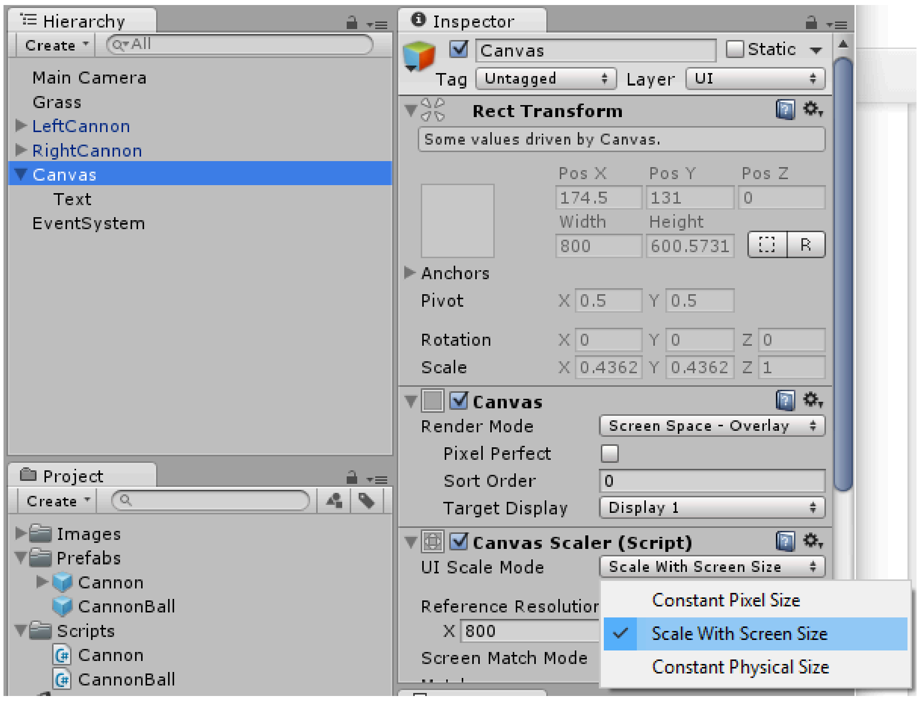
UI

Yet another important element of a game is to score points when hitting the other player. Showing the scores is done by implementing UI (= "User Interface") text elements to show the scores for each player.

1. Right-click in the Hiearchy window and create **UI**→**Text** element



2. Besides the text element, this will also create a UI canvas, which will contain all UI elements for the game. To make it easier to support different screen resolutions, set the "UI Scale Mode" field to "Scale With Screen Size" for the canvas.

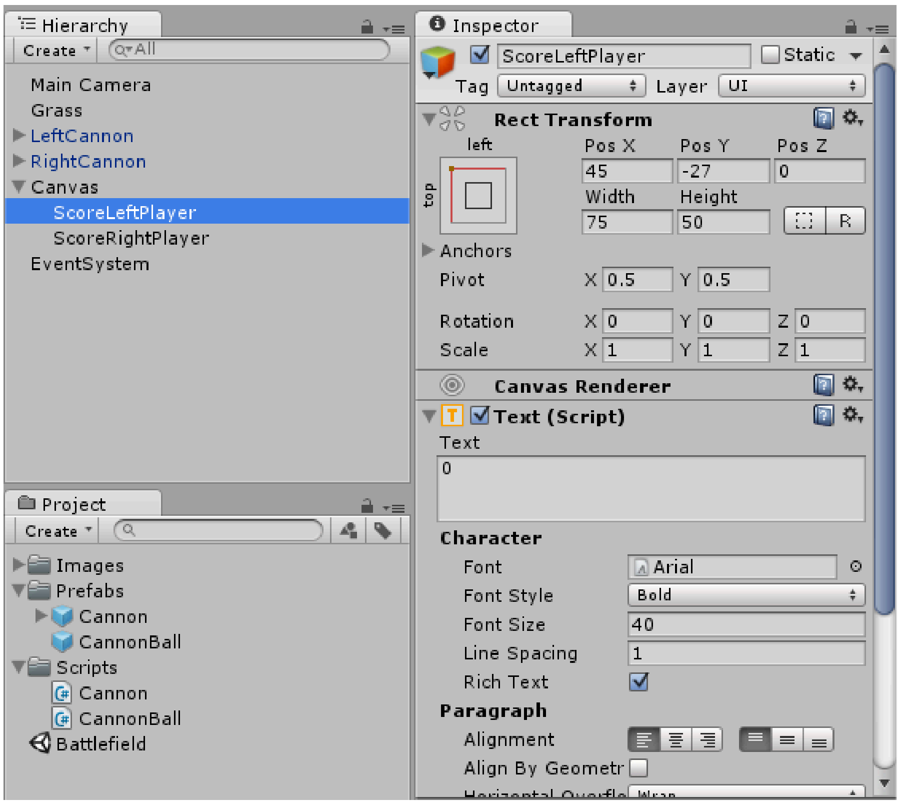

Hierarchy
Main Camera
Grass
LeftCannon
RightCannon
Canvas
Text
EventSystem

Project
Images
Prefabs
Cannon
CannonBall
Scripts
Cannon
CannonBall

Inspector
Canvas Static
Tag Untagged Layer UI
Rect Transform
Some values driven by Canvas.
Pos X 174.5 Pos Y 131 Pos Z 0
Width 800 Height 600.5731
Anchors
Pivot X 0.5 Y 0.5
Rotation X 0 Y 0 Z 0
Scale X 0.4362 Y 0.4362 Z 1
Canvas
Render Mode: Screen Space - Overlay
Pixel Perfect ☐
Sort Order 0
Target Display: Display 1
Canvas Scaler (Script)
UI Scale Mode: Scale With Screen Size
Reference Resolution X 800
Screen Match Mode

Constant Pixel Size
✓ Scale With Screen Size
Constant Physical Size

3. Create UI text element for both left and right player. It's simplest to create the left player score UI element and then copy it by right-clicking and selecting "Duplicate"


Hierarchy
Main Camera
Grass
LeftCannon
RightCannon
Canvas
ScoreLeftPlayer
ScoreRightPlayer
EventSystem

Project
Images
Prefabs
Cannon
CannonBall
Scripts
Cannon
CannonBall
Battlefield

Inspector
ScoreLeftPlayer Static
Tag Untagged Layer UI
Rect Transform
left / top
Pos X 45 Pos Y -27 Pos Z 0
Width 75 Height 50
Anchors
Pivot X 0.5 Y 0.5
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Canvas Renderer
Text (Script)
Text
0
Character
Font: Arial
Font Style: Bold
Font Size: 40
Line Spacing: 1
Rich Text ✓
Paragraph
Alignment
Align By Geometry ☐
Horizontal Overflow: Wrap

Implementing score and explosions

Last part of this book is to implement the code changes in order to update score when the opponent is hit as well as showing the explosion. This requires a number of changes to code. If you have problems getting the game working, you can compare with the final code listed in Appendix A

In order to know which player has scored a point, we need to know who fired the cannonball, so first change is in the CannonBall.cs script, which will look like following (most of the file is changing, so it's easiest just to show the entire file):

```
using UnityEngine;

public class CannonBall : MonoBehaviour
{
    public Cannon FiringPlayer;

    void OnCollisionEnter2D (Collision2D collision)
    {
        if (collision.gameObject.name == "Grass")
        {
            GetComponent<Rigidbody2D> ().isKinematic = true;
            GetComponent<Rigidbody2D> ().velocity = Vector3.zero;
        }
        else if (collision.gameObject.tag == "Cannon")
        {
            Cannon cannon = collision.gameObject.GetComponent<Cannon> ();
            cannon.GotHit ();

            if (collision.gameObject.name != FiringPlayer.name)
            {
                FiringPlayer.Score += 1;
                FiringPlayer.Reset ();
            }

            Destroy (this.gameObject);
        }
    }
}
```

The code uses "Cannon" tag, to check that the cannonball actually hit cannon, as well as we didn't hit ourself. In this case, we give one point to the player who fired the cannonball.

For Cannon.cs script we'll add a reference to the explosion prefab and the score UI text element for the cannon:

```
    public GameObject ExplosionPrefab;
    public UnityEngine.UI.Text ScoreUIText;
```

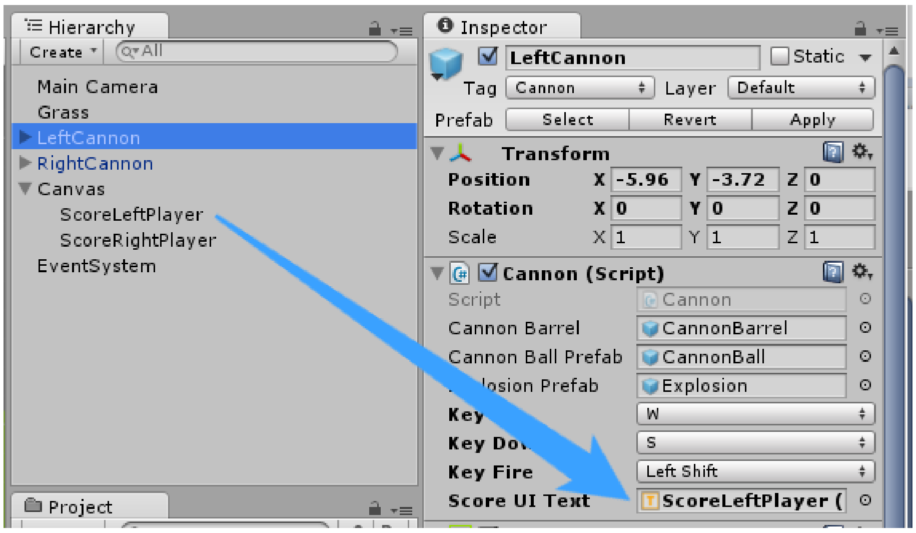
And a variable for keeping track of points for the player:

```
private int _score;

public int Score
{
    get { return _score; }
    set
    {
        _score = value;

        if (ScoreUIText)
            ScoreUIText.text = _score.ToString();
    }
}
```

As the cannonball now needs to know which player fired it, we're setting the `FiringPlayer` property:

```
        cannonBall.GetComponent<CannonBall> ().FiringPlayer = this;
```

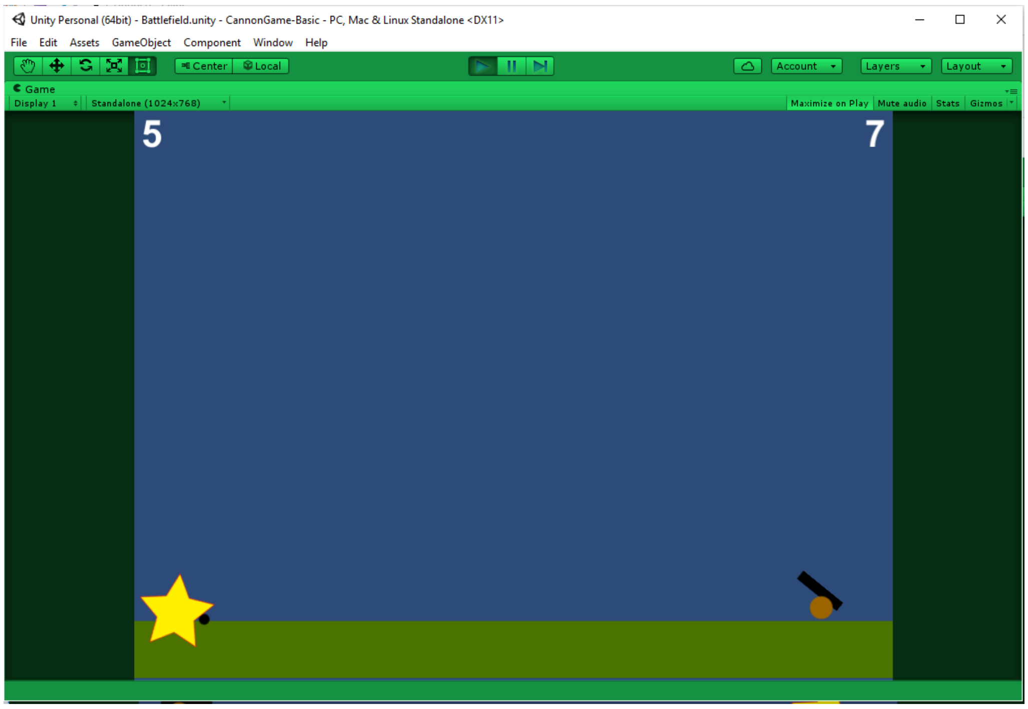
In the editor we assign the UI text element for each cannon, i.e. "ScoreLeftPlayer" for "LeftCannon" and similar for the right cannon:



Hierarchy
Create
All
Main Camera
Grass
LeftCannon
RightCannon
Canvas
ScoreLeftPlayer
ScoreRightPlayer
EventSystem
Project

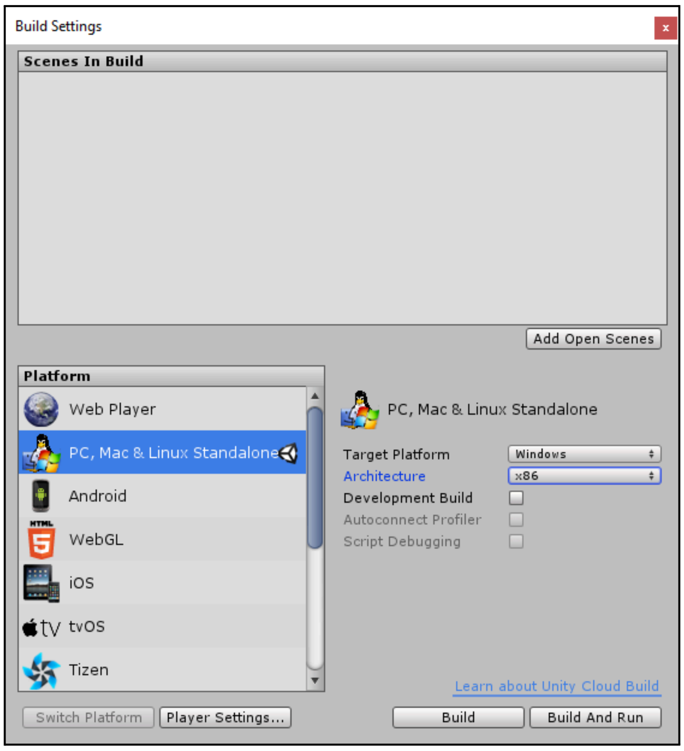
Inspector
LeftCannon
Static
Tag Cannon
Layer Default
Prefab Select Revert Apply
Transform
Position X -5.96 Y -3.72 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Cannon (Script)
Script Cannon
Cannon Barrel CannonBarrel
Cannon Ball Prefab CannonBall
losion Prefab Explosion
Key W
Key Do S
Key Fire Left Shift
Score UI Text ScoreLeftPlayer (

At this point we actually do have a game which can be played by two players in the editor.


Unity Personal (64bit) - Battlefield.unity - CannonGame-Basic - PC, Mac & Linux Standalone <DX11>
File Edit Assets GameObject Component Window Help
Center Local
Account Layers Layout
Game
Display 1 Standalone (1024x768)
Maximize on Play Mute audio Stats Gizmos
5
7

Build

Until now we have only tested the game inside Unity editor using playmode, but in order to let others play the game on their computers, it has to be “build” using **File→Build Settings...** menu:


Build Settings

Scenes In Build

Add Open Scenes

Platform
- Web Player
- PC, Mac & Linux Standalone
- Android
- WebGL
- iOS
- tvOS
- Tizen

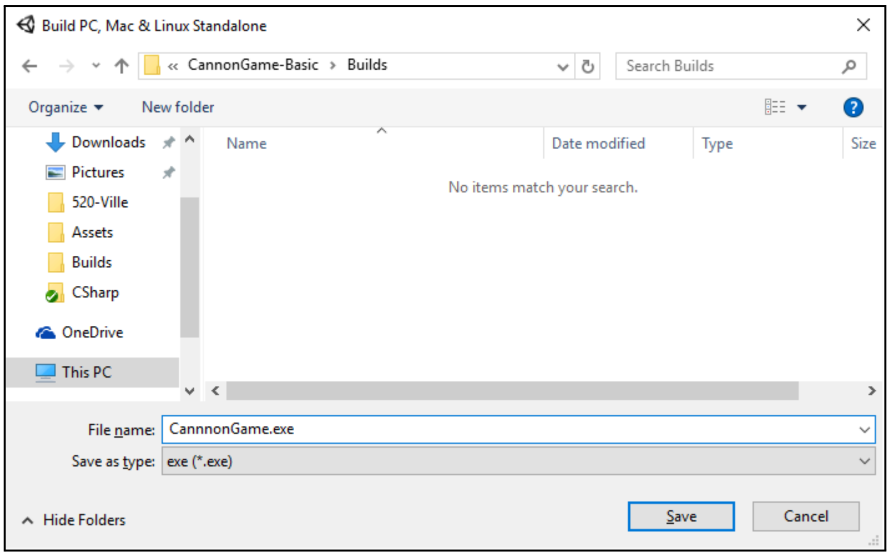
PC, Mac & Linux Standalone

Target Platform	Windows
Architecture	x86
Development Build	☐
Autoconnect Profiler	☐
Script Debugging	☐

Learn about Unity Cloud Build

Switch Platform | Player Settings... | Build | Build And Run

Choose “PC, Mac & Linux Standalone”, click “Build And Run” and specify a filename for the game:

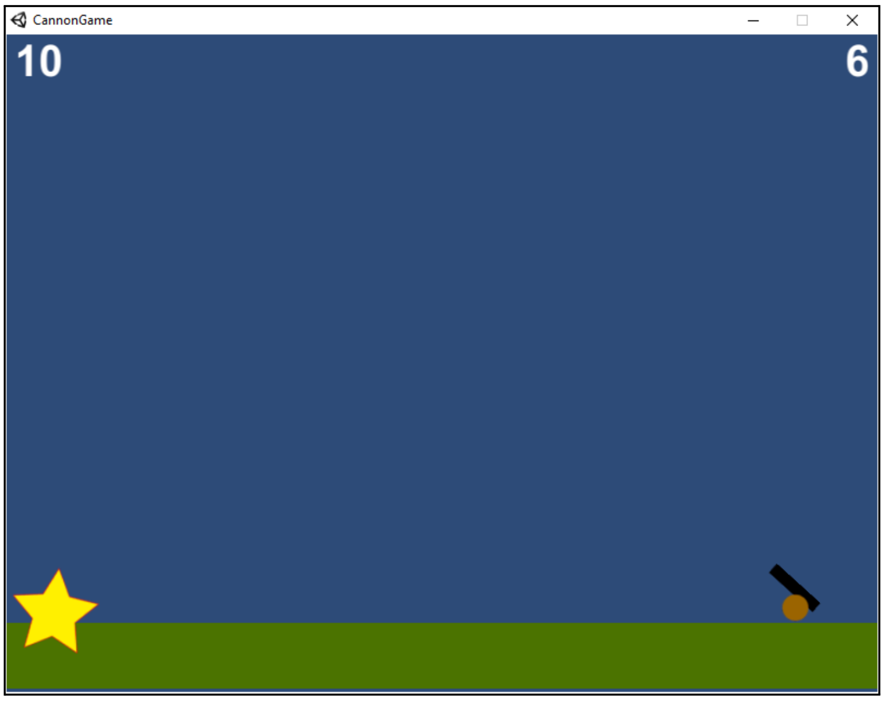


No items match your search.

File name: CannnonGame.exe

Save as type: exe (*.exe)

The game files can now e.g. be copied to a USB-stick and be played on another computer which doesn’t need to have Unity installed.



It’s also possible to build the game for WebGL, i.e. HTML, which makes it possible to upload the game to a website and play through a browser.

Ideas for improving game

If you would like to improve the game, here are some ideas for getting started:

- Turn-based, i.e. wait for other player to shoot
- Control the firepower, e.g. by how long the fire key is pressed
- Create an obstacle (hill, buildings etc) between players, which you need to shoot across
- Be able to move cannon forwards/backwards
- Implement a computer/AI player, so game can be played alone against computer

Appendix A: The final code

Cannon.cs

```
using UnityEngine;
using System.Collections;

public class Cannon : MonoBehaviour
{
    public GameObject CannonBarrel;
    public GameObject CannonBallPrefab;
    public GameObject ExplosionPrefab;

    public KeyCode KeyUp;
    public KeyCode KeyDown;
    public KeyCode KeyFire;

    public UnityEngine.UI.Text ScoreUIText;

    private float _firePower = 10f;
    private int _score;

    public int Score
    {
        get { return _score; }
        set
        {
            _score = value;

            if (ScoreUIText)
                ScoreUIText.text = _score.ToString();
        }
    }

    // Use this for initialization
    void Start ()
```

```
{
    Reset ();
}

public void Reset()
{
    gameObject.SetActive (true);
    CannonBarrel.transform.localRotation = Quaternion.identity;
}

// Update is called once per frame
void Update ()
{
    if (Input.GetKey (KeyUp))
    {
        RotateBarrel (1);
    }
    else if (Input.GetKey (KeyDown))
    {
        RotateBarrel (-1);
    }
    else if (Input.GetKeyUp (KeyFire))
    {
        FireCannon (_firePower);
    }
}

void RotateBarrel (int degrees)
{
    CannonBarrel.transform.Rotate (new Vector3 (0, 0, degrees));
}

private void FireCannon (float power)
{
    Vector2 position =
        CannonBarrel.transform.position + CannonBarrel.transform.right * 0.8f;

    GameObject cannonBall = Instantiate (
                          CannonBallPrefab,
                          position,
                          CannonBarrel.transform.rotation) as GameObject;

    cannonBall.GetComponent<CannonBall> ().FiringPlayer = this;
    cannonBall.GetComponent<Rigidbody2D> ().AddForce (
        CannonBarrel.transform.right * power,
        ForceMode2D.Impulse);
    Destroy (cannonBall.gameObject, 30);
}

public void GotHit ()
{
    if (ExplosionPrefab)
    {
        GameObject explosion = Instantiate (
                ExplosionPrefab,
                this.transform.position,
                Quaternion.identity) as GameObject;
            Destroy (explosion, 1f);
        }

        this.gameObject.SetActive (false);
        Invoke ("Start", 2);
    }
}
```

CannonBall.cs

```
using UnityEngine;

public class CannonBall : MonoBehaviour
{
    public Cannon FiringPlayer;

    void OnCollisionEnter2D (Collision2D collision)
    {
        if (collision.gameObject.name == "Grass")
        {
            GetComponent<Rigidbody2D> ().isKinematic = true;
            GetComponent<Rigidbody2D> ().velocity = Vector3.zero;
        }
        else if (collision.gameObject.tag == "Cannon")
        {
            Cannon cannon = collision.gameObject.GetComponent<Cannon> ();
            cannon.GotHit ();

            if (collision.gameObject.name != FiringPlayer.name)
            {
                FiringPlayer.Score += 1;
                FiringPlayer.Reset ();
            }

            Destroy (this.gameObject);
        }
    }
}
```