A QUICK GUIDE TO CARD GAMES WITH UNITY

Get Started with Card Games in less than 60 minutes

Patrick Felicia

A QUICK GUIDE TO CARD GAMES WITH UNITY

First published: November 2017

Published by Patrick Felicia

CREDITS

Author: Patrick Felicia

ABOUT THE AUTHOR

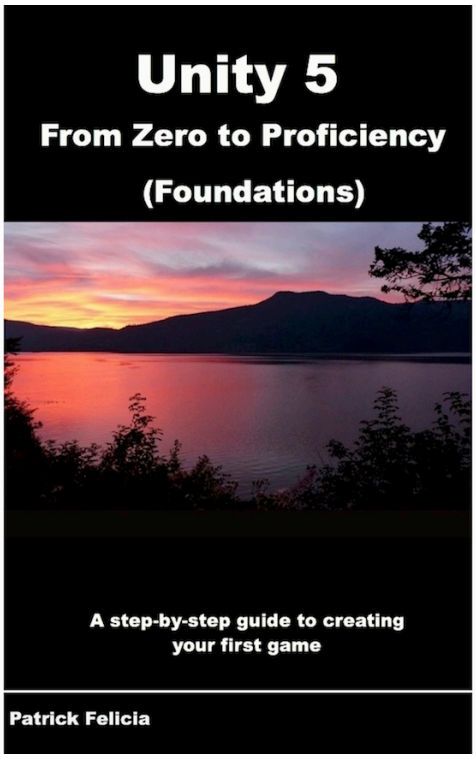
Patrick Felicia is a lecturer and researcher at Waterford Institute of Technology, where he teaches and supervises undergraduate and postgraduate students. He obtained his MSc in Multimedia Technology in 2003 and PhD in Computer Science in 2009 from University College Cork, Ireland.

He has published several books and articles on the use of video games for educational purposes, including the Handbook of Research on Improving Learning and Motivation through Educational Games: Multidisciplinary Approaches (published by IGI), and Digital Games in Schools: a Handbook for Teachers, published by European Schoolnet.

Patrick has published over 10 books on Unity, covering several key skills such as C# and JavaScript in Unity, 3D and 2D game development with Unity, as well as 3D Character Animation.

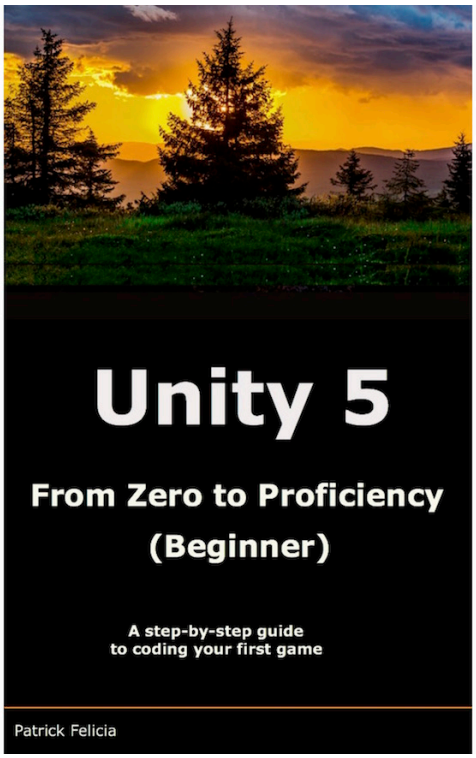
Patrick is also the Editor-in-chief of the International Journal of Game-Based Learning (IJGBL), and the Conference Director of the Irish Conference on Game-Based Learning, a popular conference on games and learning organized throughout Ireland.

BOOKS FROM THE SAME AUTHOR



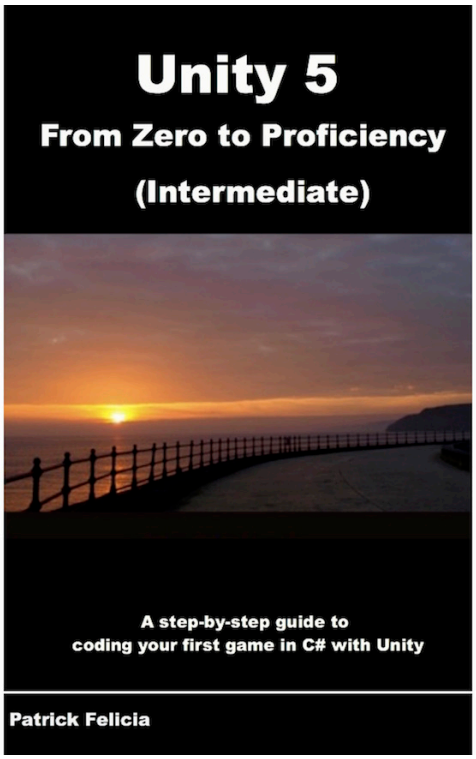
Unity 5 from Zero to Proficiency (Foundations)

In this book, you will become more comfortable with Unity's interface and its core features by creating a project that includes both an indoor and an outdoor environment. This book only covers drag and drop features, so that you are comfortable with Unity's interface before starting to code (in the next book). After completing this book, you will be able to create outdoors environments with terrains and include water, hills, valleys, sky-boxes, use built-in controllers (First- and Third-Person controllers) to walk around the 3D environment and also add and pilot a car and an aircraft.



Unity 5 from Zero to Proficiency (Beginner)

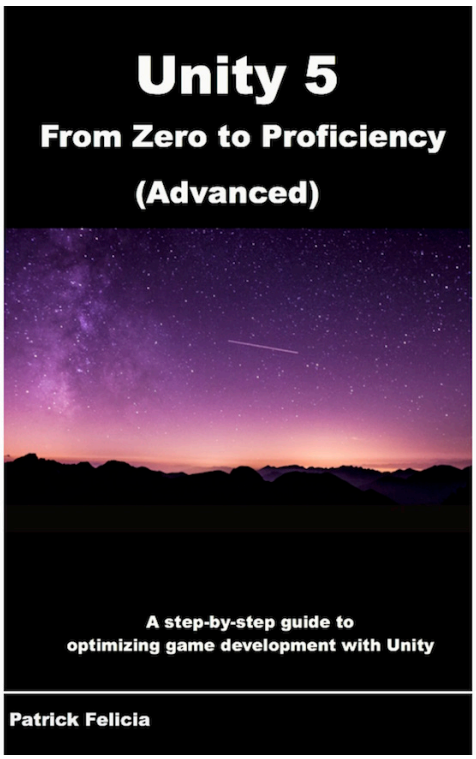
In this book, you will get started with coding using JavaScript. The book provides an introduction to coding for those with no previous programming experience, and it explains how to use JavaScript in order to create an interactive environment. Throughout the book, you will be creating a game, and also implementing the core mechanics through scripting. After completing this book you will be able to write code in JavaScript, understand and apply key programming principles, understand and avoid common coding mistakes, learn and apply best programming practices, and build solid programming skills.



Unity 5 from Zero to Proficiency (Intermediate)

In this book, you improve your coding skills and learn more programming concepts to add more activity to your game while optimizing your code. The book provides an introduction to coding in C# t. Throughout the book, you will be creating a game, and also implementing the core mechanics through scripting.

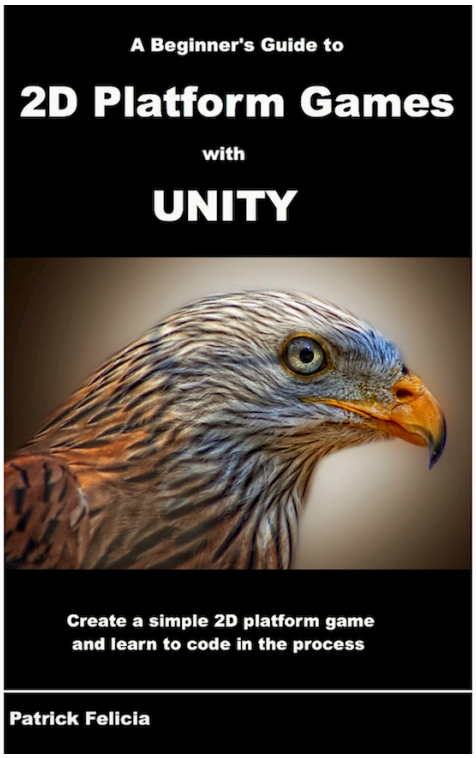
After completing this book you will be able to write code in C#, understand and apply Object-Oriented Programming techniques in C#, create and use your own classes, use Unity's Finite State Machines, and apply intermediate Artificial Intelligence.



Unity 5 from Zero to Proficiency (Advanced)

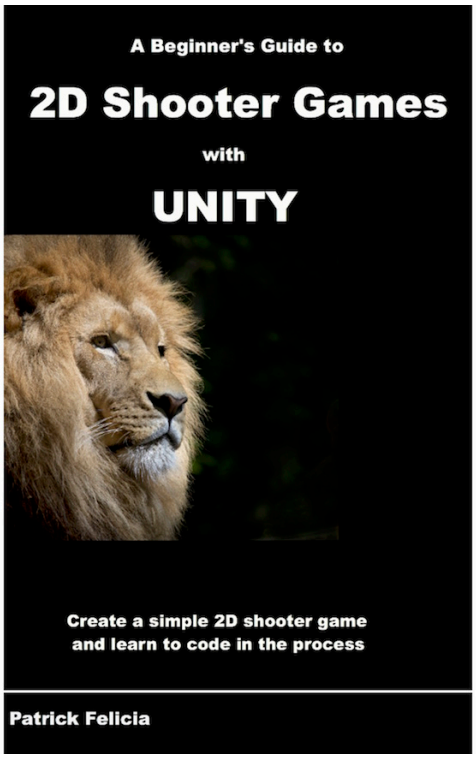
In this book, which is the last in the series, you will go from Intermediate to Advanced and get to work on more specific topics to improve your games and their performances.

After completing this book, you will be able to create a (networked) multi-player game, access Databases from Unity, understand and apply key design, patterns for game development, use your time more efficiently to create games, structure and manage a Unity project efficiently, optimize game performances, optimize the structure of your game, and create levels procedurally.



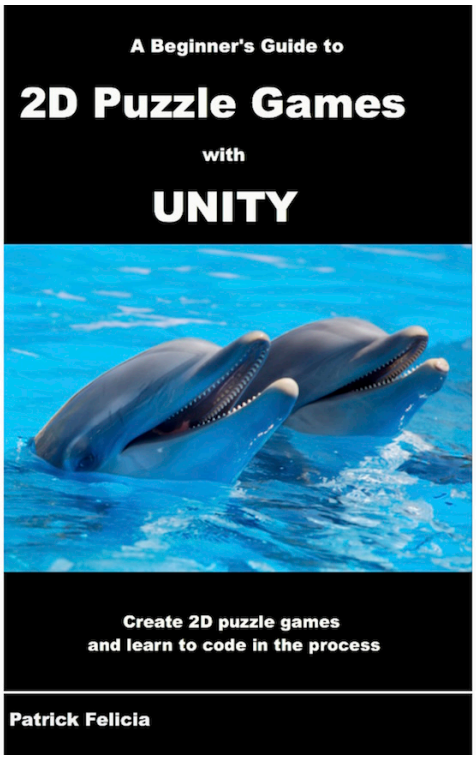
A Beginner's Guide to 2D Platform Games with Unity

In this book, you will get started with creating a simple 2D platform game. The book provides an introduction to platform games , and it explains how to use C# in order to create an interactive environment.



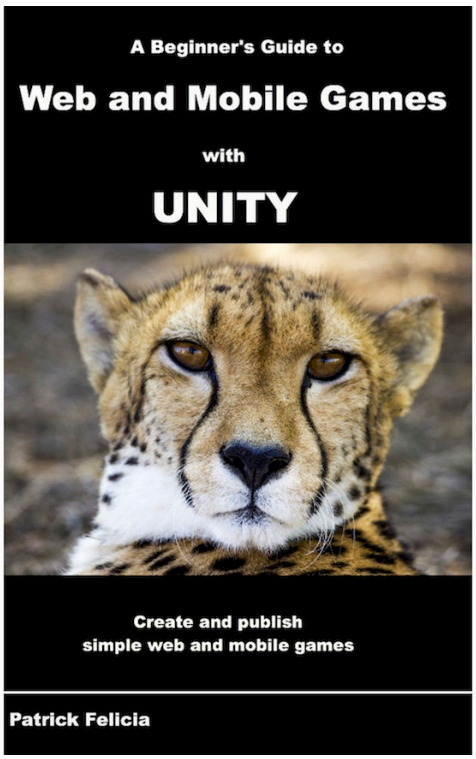
A Beginner's Guide to 2D Shooter Games with Unity

In this book, you will get started with creating a simple 2D shooter game. The book provides an introduction to 2D shooter games, and it explains how to use C# in order to create an interactive environment



A Beginner's Guide to 2D Puzzle Games with Unity

In this book, you will get started with creating four different types of puzzle games. The book provides an introduction to 2D puzzle games , and it explains how to use C# in order to create four addictive types of puzzle games including: word games (i.e., hangman), memory game (i.e., simon game), card matching game, and a puzzle.



A Beginner's Guide to Web and Mobile Games with Unity

In this book, you will get started with exporting a simple infinite runner to the web and Android. The book provides an introduction to how to export and share your game with friends on the Web and on Android Play. It provides step-by-step instructions and explains how to easily share a simple game with your friends so that they can play it on your site or an Android device including: processing taps, exporting the game to a web page, debugging your app, signing your app, and much more.

SUPPORT AND RESOURCES FOR THIS BOOK

You can download the resource pack for this book; it includes solutions scripts for some of the sections in this book, as well as some of the files needed to complete some of the activities presented in this book.

To download these resources, please do the following.

If you are already a member of my list, you can just go to the member area (**http://learntocreategames.com/member/**) using the usual password and you will gain access to all the resources for this book.

If you are not yet on my list, you can do the following:

- Open the following link: **http://learntocreategames.com/books/**
- Select this book ("**A Quick Guide to Card Games with Unity**").
- On the new page, click on the link labelled "**Book Files**", or scroll down to the bottom of the page.
- In the section called "**Download your Free Resource Pack**", enter your email address and your first name, and click on the button labeled "**Yes, I want to receive my bonus pack**".
- After a few seconds, you should receive a link to your free start-up pack.
- When you receive the link, you can download all the resources to your computer.

This book is dedicated to Helena

TABLE OF CONTENTS

PREFACE

To be able to help people like you, I have designed and published more than 8 books on Unity; these books are in-depth and really provide a significant amount of information on a wide range of topics related to Unity, including 2D/3D game development, Artificial Intelligence, Animation, and much more…

This being said, while these books are comprehensive, many readers, like you, may just want to focus on a particular topic and get started fast.

This book is part of a series entitled **A Quick Guide To**, and does just this. In this book series, you have the opportunity to get started on a particular topic in less than 60 minutes, delving right into the information that you really need. Of course, you can, after reading this book, move-on to more comprehensive books; however, I understand that sometimes you may have little time to complete a project and that you need to get comfortable with a topic fast.

In this book entitled "**A Quick Guide to Card Games with Unity**" you will discover how to quickly create a simple card guessing game.

WHAT YOU NEED TO USE THIS BOOK

To complete the project presented in this book, you only need Unity 2017 (or a more recent version) and to also ensure that your computer and its operating system comply with Unity's requirements. Unity can be downloaded from the official website (**http://www.unity3d.com/download**), and before downloading, you can check that your computer is up to scratch on the following page: **http://www.unity3d.com/unity/system-requirements**. At the time of writing this book, the following operating systems are supported by Unity for development: Windows XP (i.e., SP2+, 7 SP1+), Windows 8, and Mac OS X 10.6+. In terms of graphics card, most cards produced after 2004 should be suitable.

In terms of computer skills, all knowledge introduced in this book will assume no prior programming experience from the reader. This book does not include any scripting. So for now, you only need to be able to perform common computer tasks such as downloading items, opening and saving files, and be comfortable with dragging and dropping items and typing.

WHO THIS BOOK IS FOR

If you can answer **yes** to all these questions, then this book is for you:

1. Are you a total beginner in Unity?
2. Would you like to become proficient in creating card games?
3. Would you like to start creating great 2D games?
4. Although you may have had some prior exposure to Unity, would you like to delve more into the creation of card games and C#?

If you can answer yes to all these questions, then this book is **not** for you:

1. Can you already easily create a card game with Unity?
2. Are you looking for a reference book on Unity programming?
3. Are you an experienced (or at least advanced) Unity user?

If you can answer yes to all three questions, you may instead look for the next books in the series. To see the content and topics covered by these books, you can check the official website (**www.learntocreategames.com/books**).

IMPROVING THE BOOK

Although great care was taken in checking the content of this book, I am human, and some errors could remain in the book. As a result, it would be great if you could let me know of any issue or error you may have come across in this book, so that it can be solved and the book updated accordingly. To report an error, you can email me (**learntocreategames@gmail.com**) with the following information:

- Name of the book.
- The page where the error was detected.
- Describe the error and also what you think the correction should be.

Once your email is received, the error will be checked, and, in the case of a valid error, it will be corrected and the book page will be updated to reflect the changes accordingly.

SUPPORTING THE AUTHOR

A lot of work has gone into this book and it is the fruit of long hours of preparation, brainstorming, and finally writing. As a result, I would ask that you do not distribute any illegal copies of this book.

This means that if a friend wants a copy of this book, s/he will have to buy it through the official channels (i.e., through Amazon or the book's official website: **http://www.learntocreategames.com/books)**.

If some of your friends are interested in the book, you can refer them to the book's official website **(http://www.learntocreategames.com/books)** where they can either buy the book, or join the mailing list to be notified of future promotional offers or enter a monthly draw and be in for a chance to receive a free copy of the book.

1
CREATING A CARD GUESSING GAME

In this section, we will create a simple card guessing game where the player has to remember a set of 20 cards and to match these cards based on their value.

After completing this chapter, you will be able to:

- Create a card game.
- Change the sprite of an object at run-time.
- Check when two cards picked by the player have the same value.
- Shuffle the cards.

INTRODUCTION

In this chapter, we will create a new card game as follows:

- The deck of cards will be shuffled.
- There will be two rows of cards (i.e., 10 cards in each row).
- All cards are initially hidden.
- The player needs to pick one card from the first row and then one card from the second row.
- If the cards are identical, it's a match, and they are then both removed.
- Otherwise both cards are hidden again.
- The player wins when s/he has managed to match (and subsequently remove) all the cards.

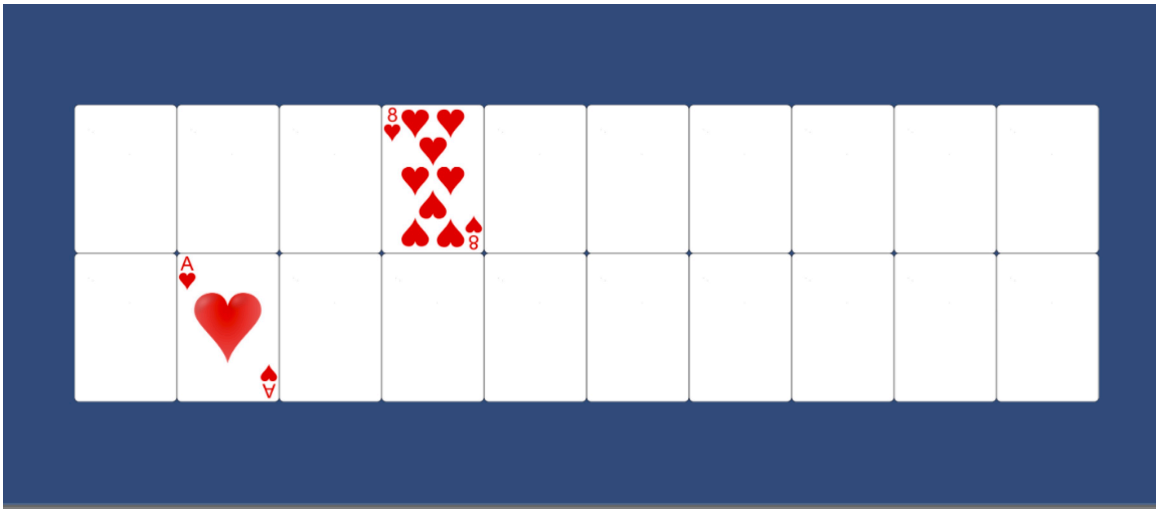


Figure 1: An example of the game completed

SETTING-UP THE INTERFACE

First, we will import the new deck of cards:

- Please save your current scene (**File | Save Scene**).
- Create a new scene (**File | New Scene**).
- Save this new scene as **chapter3**.
- As we have done before, you can remove the skybox used for the background (if any), using the menu **Window | Lighting**.
- Please locate the resource pack that you have downloaded in your file system, and import (i.e., drag and drop) the folder called **cards** to the **Project** window in Unity.
- As you will see, this folder includes a set of 68 cards that we will be using for our game, as illustrated in the next figure.

Figure 2: Importing the cards

- Once this is done, we just need to change some of the properties of these images so that they can be used as sprites in our game.
- In the **Project** window, and in the folder called **cards**, that contains the different images imported, please select all the cards (i.e., click on one card and then select **CTRL + A**).

- Using the **Inspector**, change the **Texture Type** of these images to **Sprite (2D and UI)** and leave the other options as default, as described on the next figure.

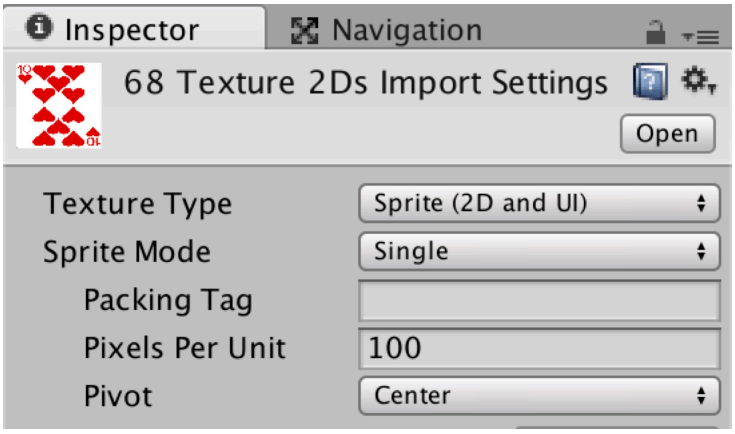


Figure 3: Changing the attribute Texture Type

- You can then press the **Apply** button located in the bottom-right corner of the window, as described on the next figure.

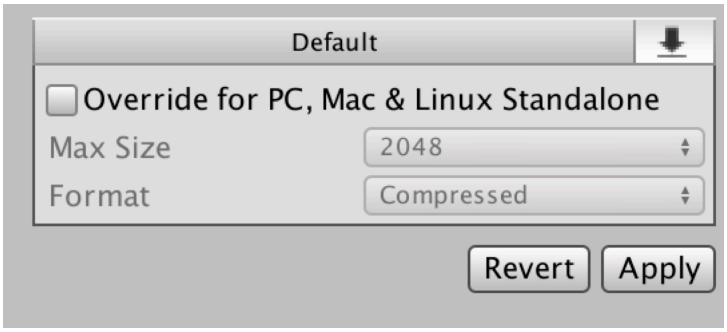
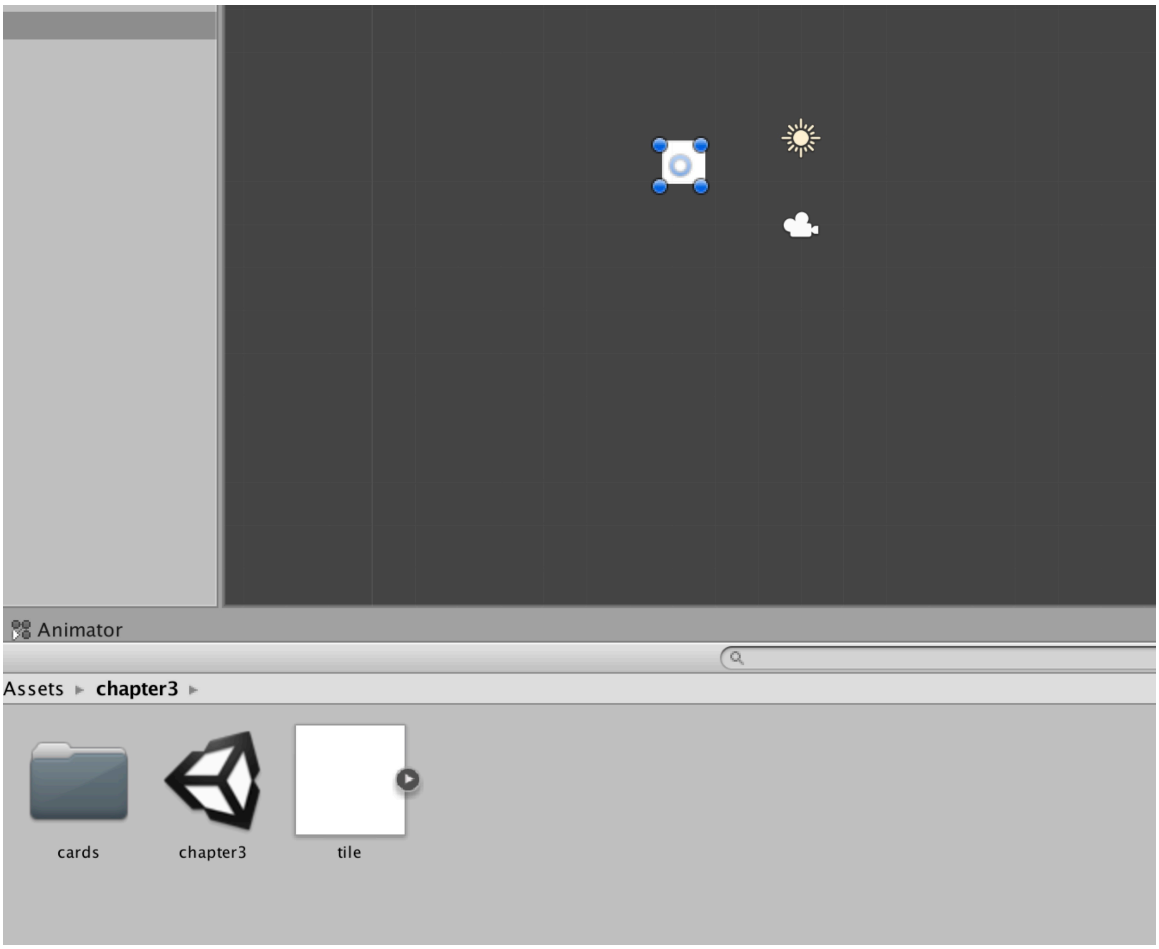


Figure 4: Applying changes

- It should take Unity a few seconds to convert these images.

Next, we will use one of these sprites to start to implement basic functionalities for our game.

- Please create a new square sprite: from the **Project** window select **Create | Sprites | Square**.
- Rename this new asset **tile** and drag and drop it to the **Scene** view, as illustrated on the next figure.



- This will create a new object called **tile** in the **Hierarchy**.
- You can change its position to **(0, 0, 0)** and add a **2D Box Collider** component to it: from the top menu select **Component | Physics2D | Box 2D Collider**.
- This collider is needed so that we can detect clicks on this sprite.

Next, we will create a script that will process clicks on this sprite.

- Please create a new C# script called **Tile** (i.e., from the **Project** window, select **Create | C# Script**).
- Open this script.
- Add the following function to it.

```
public void OnMouseDown()
{
      print ("You pressed on tile");
}
```

- Save your script, and drag and drop it to the object called **tile** in the **Hierarchy**.

Once this is done, you can test the scene by playing the scene and by then clicking on the white rectangle (tile) that you have created; a message should appear in the **Console** window.

Next, we will just change the appearance of our tile by using one of the sprites that we have imported.

- Please select the object called **tile** in the **Hierarchy**.
- Using the **Inspector**, you will see that it has a component called **Sprite Renderer**, with an attribute called **Sprite**, as described on the next figure.

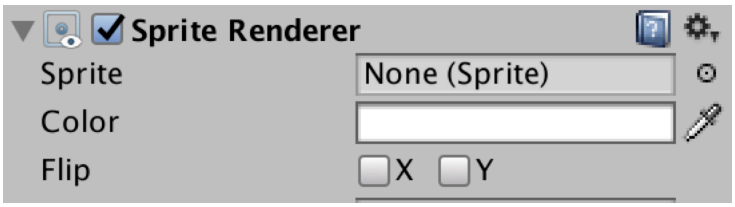
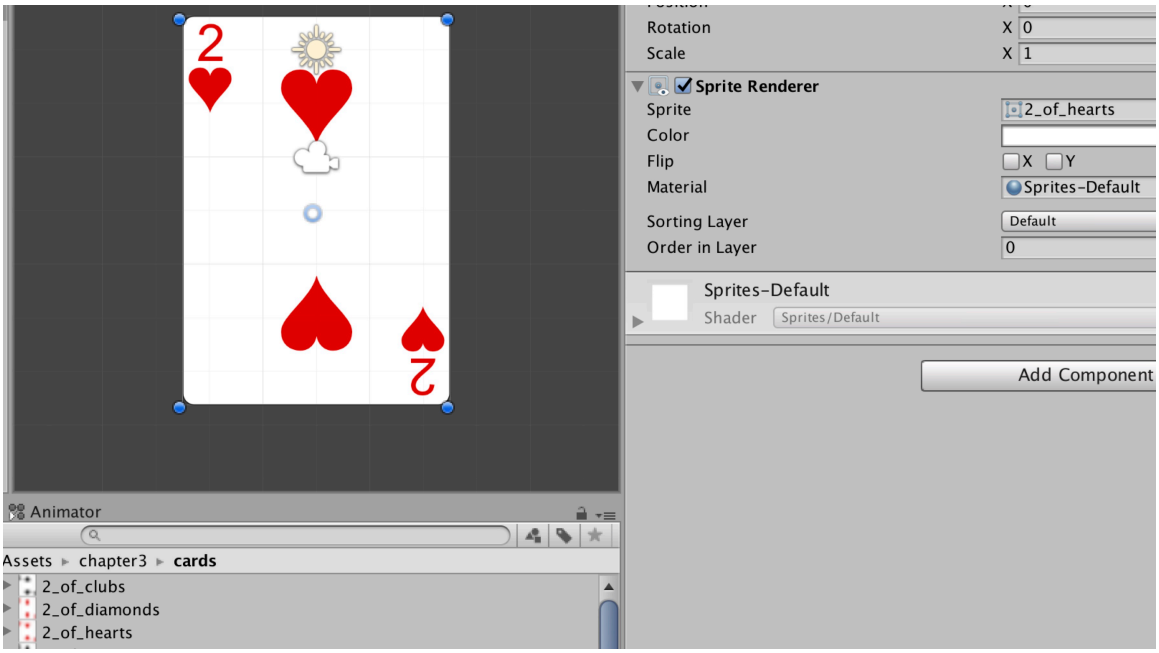


Figure 5: Identifying the Sprite Renderer component

- Drag and drop one of the images that you have imported from the **Project** window (i.e., from the folder called **cards**) to the attribute called **Sprite**, for example the **two of hearts**, as illustrated in the next figure.



- Once this is done, you should see that the tile has now turned into a card, as per the previous figure.

Next, we will create the code that either hides or displays a card; for this purpose, we will be using two different sprites: the sprite for the card that is supposed to be displayed, and a blank sprite that symbolizes the back of each card, for when a card is supposed to be hidden.

- Please add the following code at the beginning of the class **Tile**.

```
private bool tileRevealed = false;
public Sprite originalSprite;
public Sprite hiddenSprite;
```

- Because the last two variables (i.e., **originalSprite** and **hiddenSprite**) are public, if you select the object called **tile** in the **Hierarchy** and look at the **Inspector**, you should now see that its component called **Tile** has two empty fields (or placeholders) called **originalSprite** and **hiddenSprite**.

- Please drag and drop the sprite called **2_of_hearts** from the **Project** window to the field **originalSprite**, and the sprite called **back_of_cards** to the field called **hiddenSprite**.

- The component should then look as in the next figure.

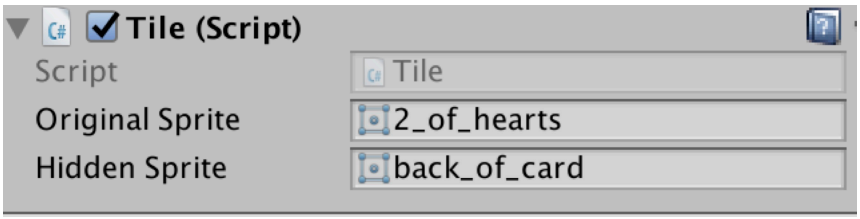


Figure 6: Setting the appearance of the card.

Next, we will create two functions called **hideCard** and **displayCard** that will either display or hide a card.

- Please open the script called **Tile**.
- Add the following code to it.

```
public void hideCard()
{
      GetComponent<SpriteRenderer> ().sprite = hiddenSprite;
      tileRevealed = false;
}
public void revealCard()
{
      GetComponent<SpriteRenderer> ().sprite = originalSprite;
      tileRevealed = true;
}
```

In the previous code, we create two functions that either display or hide a card by setting the sprite for this particular card to the **hiddenSprite** or the **originalSprite**. The variable **tileRevealed** is also amended to indicate whether the card is displayed or hidden.

- Next, add the following code to the **Start** function, so that all cards are initially hidden at the start of the game.

```
hideCard();
```

- Last, we can modify the function **OnMouseDown** as follows:

```
public void OnMouseDown()
{
      print ("You pressed on tile");
      if (tileRevealed)
            hideCard ();
      else
            revealCard ();
}
```

In the previous code:

- When the mouse is clicked, we check whether the card is currently revealed or hidden.
- If it is revealed, then we call the function **hideCard**.
- Otherwise, we call the function **revealCard**.

You can now save your code, and test the scene; you should see that the card is hidden at the beginning (i.e., the back of the card is displayed); then, as you click several times on the card, it should subsequently be hidden or revealed.

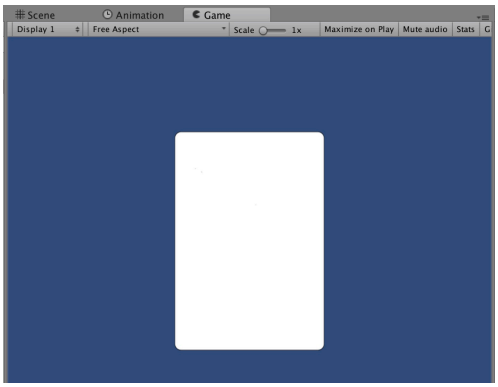


Figure 7: The card is hidden

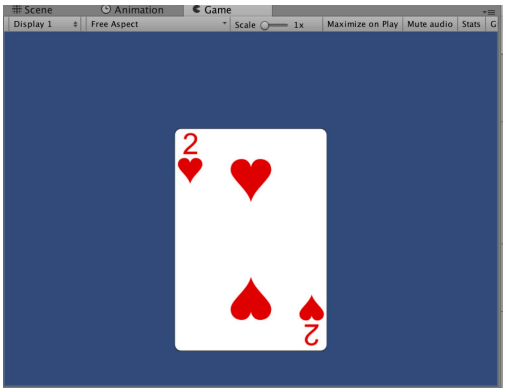


Figure 8: The card is displayed

- Lastly, we will scale down the card: using the **Inspector**, modify the scale property of the object **tile** to (0.5, 0.5, 1).

Now that the interaction with the card works, we can create a prefab from it, so that this prefab can be used to generate several similar cards.

- Please select the object **tile** in the **Hierarchy**.
- Drag and drop it to the **Project** window.
- This should create a new **prefab** called **tile**, as illustrated on the next figure.

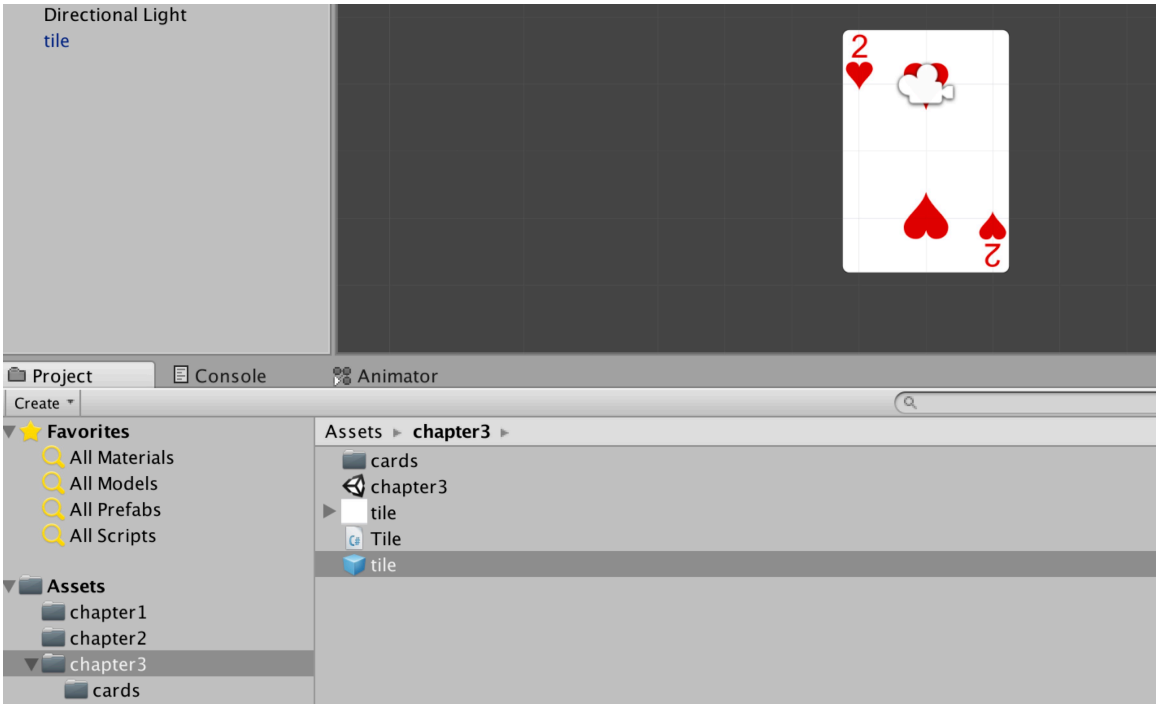


Figure 9: Creating a prefab from the card

- You can then delete or de-activate the object called **tile** in the **Hierarchy**.

CREATING A GAME MANAGER

So at this stage, we have managed to create a card (and a corresponding prefab) that we can either hide or reveal. So the next step will be to display several cards and make it possible for the player to match them. For this purpose, we will create an empty object **gameManager** that will manage the game, including adding cards to the game.

- Please create a new empty object (**GameObject | Create Empty**), and rename it **gameManager**.
- Using the **Project** view, create a new script called **ManageCards** (i.e., from the **Project** window, select **Create | C# Script**), and drag and drop it on the object called **gameManager**.
- Open the script called **ManageCards** (i.e., double-click on it in the **Project** window).
- Add this code at the beginning of the script (new code in bold).

```
public GameObject card;
void Start ()
{
     displayCards();
}
public void displayCards()
{
     Instantiate    (card,    new    Vector3    (0,    0,    0),
Quaternion.identity);
}
```

In the previous code:

- We declare a new variable called **card**, that is public (hence accessible from the **Inspector** window), and that will be used as a template for all the cards to be added to the game (i.e., it will be based on the template called **tile**).
- We also create a function called **displayCards**.
- This function instantiates a new card.

Before we can test this code, we just need to initialize the variable **card**, as follows:

- Please select the object **gameManager** in the **Hierarchy**.
- Drag and drop the prefab called **tile** to the field called **card**, in the component called **ManageCards**, as illustrated on the next figure.

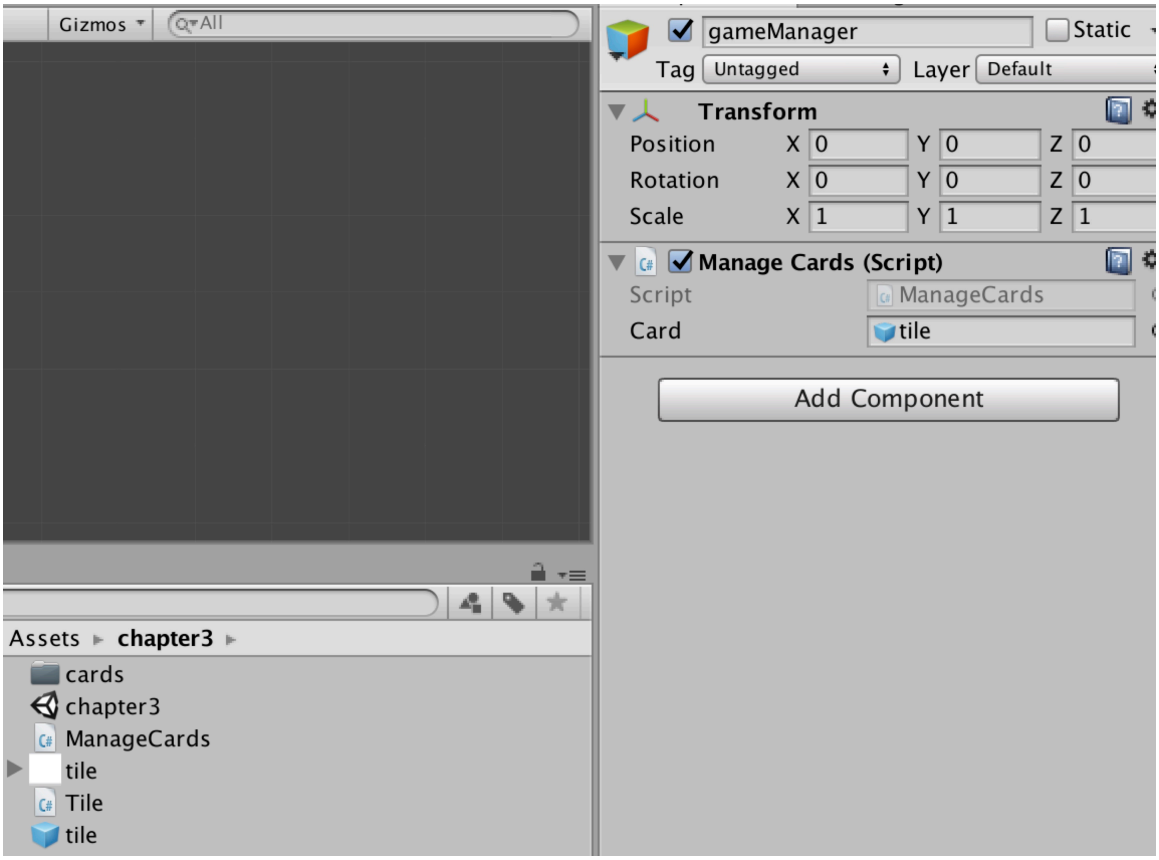


Figure 10: Initializing the variable called card

- Once this is done, you can test the game by playing the scene; you should see that a card has been added to the game, as described in the next figure.

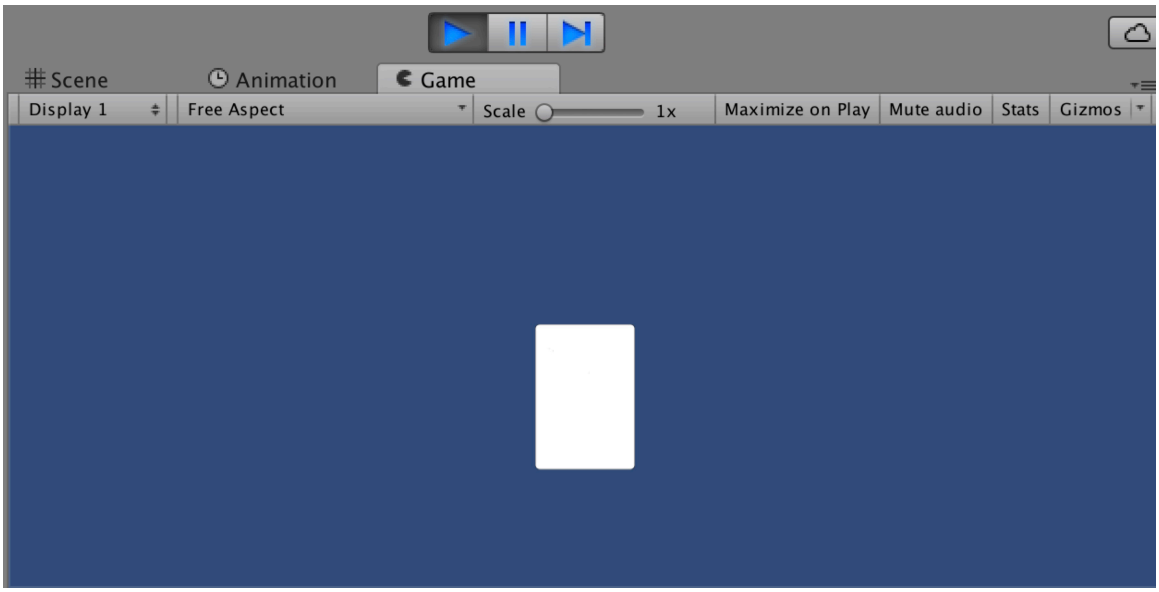


Figure 11: Testing the game

If you find it difficult to select (i.e., click on) some of the cards, it may be because their collider is too small and needs to be resized, hence collision and clicks might only be detected on a portion of the card rather than on the entire card.

Next, we will modify the script called **Tile**, to add a function that will set the sprite (or image) that should be displayed when this card is revealed; this is because in the next section, the cards will be allocated randomly; so we will no longer set the sprite for each card manually; instead, this will be done through our code.

- Please open the script called **Tile**.
- Add the following function to it.

```
public void setOriginalSprite(Sprite newSprite)
{
      originalSprite = newSprite;
}
```

In the previous code, we create a public function that will be accessible from outside this class and that will change the original sprite for this card to a sprite that is passed as a parameter to the function.

- Please save this script (i.e., the script called **Tile**).
- Please open the script **ManageCards**.

- Add the following function to the class.

```
void addACard(int rank)
{
     GameObject c = (GameObject)(Instantiate (card, new Vector3
(0, 0, 0), Quaternion.identity));
}
```

In the previous code:

- We declare a function called **addACard.**
- This function creates a new card.
- We can now modify the function called **displayCards** as follows:

```
public void displayCards()
{
     //Instantiate    (card,    new    Vector3    (0,    0,    0),
Quaternion.identity);
     addACard(0);
}
```

ADDING MULTIPLE CARDS AUTOMATICALLY

We will now create several cards on the go. The idea will be to display two rows of 10 cards each. The player will then need to match cards from the first row to cards on the second row. So the idea will be to:

- Create and add new cards based on the prefab created earlier.
- Arrange the cards so that they are aligned around the center of the screen and all visible onscreen.
- For each card, set the default sprite that should be displayed when the card is revealed.
- Shuffle the cards.

So the first step will be to add all of these cards.

If you look at the sprites that we have imported in the **Project** window, you will notice that their size is 500 by 725 pixels; we have also scaled-down these images using the **Inspector**, so their actual width in the game is 250 pixels; if you remember well, we used an import setting of 100 pixel per units pixels; this means that our cards will have a size of 2.5 (i.e., 250/100) in the game's units. So when creating our cards, we will need to make sure that their origins (or center) is at least 2.5 units apart (we will choose 3 units to be safe).

- In the script called **ManageCards**, please modify the function **AddAcard** as follows (new code in bold):

```
void addACard(int rank)
{
     //GameObject c = (GameObject)(Instantiate (card, new Vector3
(0, 0, 0), Quaternion.identity));
     GameObject c = (GameObject)(Instantiate (card, new Vector3
(rank*3.0f, 0, 0), Quaternion.identity));
}
```

In the previous code, we make sure that the x coordinate of the new card will be based on its rank; as we will be adding other cards, the first card's x coordinate will be 3 (i.e., 1 x 3), the second card's x coordinate will be 6 (i.e., 2 x 3), and so on.

- Next, we can modify the function called **displayCards** so that we can add a row of 10 cards, with the following code (new code in bold).

```
public void displayCards()
{
     //Instantiate    (card,    new    Vector3    (0,    0,    0),
Quaternion.identity);
     //addACard(0);
     for (int i = 0; i < 10; i++)
     {
          addACard (i);
     }
}
```

- Please save your code and play the scene; you will notice that 10 cards have been created; however, some of them are outside the screen; in other words, the cards need to be centered around the center of the screen.

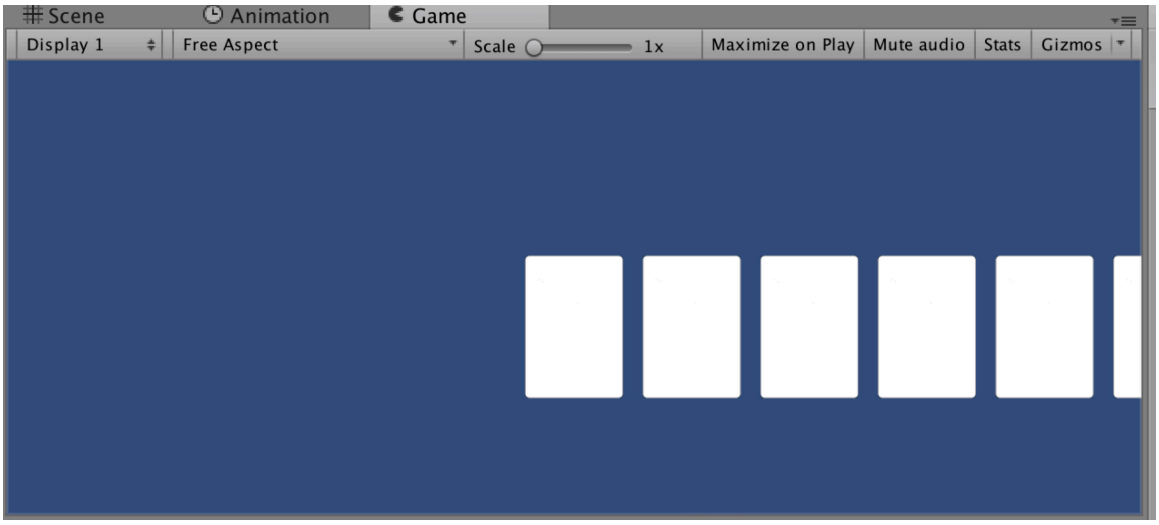


Figure 12: Displaying 10 cards

So, to solve this issue, we will do the following:

- Create an empty object that will be used as an anchor for the cards.
- Move this object to the center of the screen.
- Center the cards horizontally around this object.

So let's proceed.

- Please create an empty object and call it **centerOfScreen**.
- Using the **Inspector**, change its position to **(0, 0, 0)**.
- Open the script **ManageCards** and modify the function **addACard** as follows (new code in bold):

```
void addACard(int rank)
{

     GameObject cen = GameObject.Find("centerOfScreen");
     Vector3 newPosition = new Vector3 (cen.transform.position.x
+       ((rank-10/2)       *3),       cen.transform.position.y,
cen.transform.position.z);
     GameObject c = (GameObject)(Instantiate (card, newPosition,
Quaternion.identity));

     //GameObject c = (GameObject)(Instantiate (card, new Vector3
(0, 0, 0), Quaternion.identity));
     //GameObject c = (GameObject)(Instantiate (card, new Vector3
(rank*3.0f, 0, 0), Quaternion.identity));
}
```

As you save the code and play the scene, you should see that the cards are now centered; however, their size is too large for all of them to fit onscreen; so we will need to resize the cards accordingly.

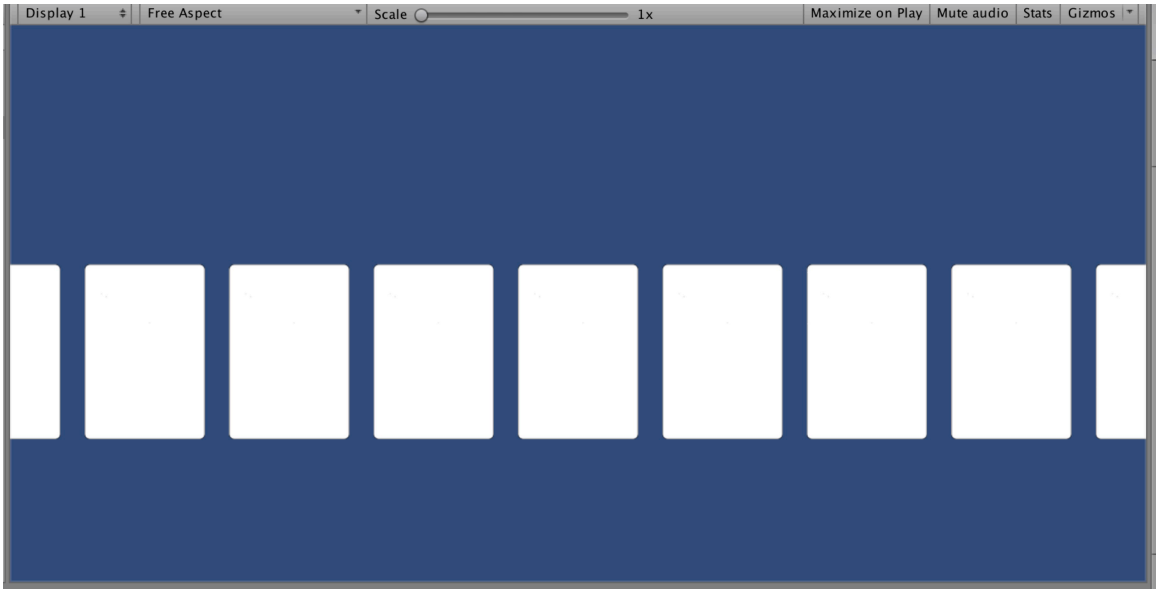


Figure 13: Displaying 10 aligned cards

- Using the **Inspector**, change the scale of the **tile** prefab to **(0.3, 0.3, 0.3)**.
- Open the script called **ManageCards** and modify it as follows (new code in bold):

```
void addACard(int rank)
{

     float cardOriginalScale = card.transform.localScale.x;
     float scaleFactor = (500 * cardOriginalScale) / 100.0f;

     GameObject cen = GameObject.Find("centerOfScreen");
     //Vector3 newPosition = new Vector3
(cen.transform.position.x + ((rank-10/2) *3),
cen.transform.position.y, cen.transform.position.z);
     Vector3 newPosition = new Vector3 (cen.transform.position.x
+    ((rank-10/2)    *scaleFactor),    cen.transform.position.y,
cen.transform.position.z);

     GameObject c = (GameObject)(Instantiate (card, newPosition,
Quaternion.identity));
```

In the previous code:

- We save the initial scale of the card.
- We create a variable called **scaleFactor** that takes into account the original width of the card (i.e., **500**) as well as its original scale.
- This **scaleFactor** variable is taken into account when defining the x coordinate of each card.

Please save your code, and play the scene; you should see that now all cards are displayed within the camera's field of view, as illustrated in the next figure.

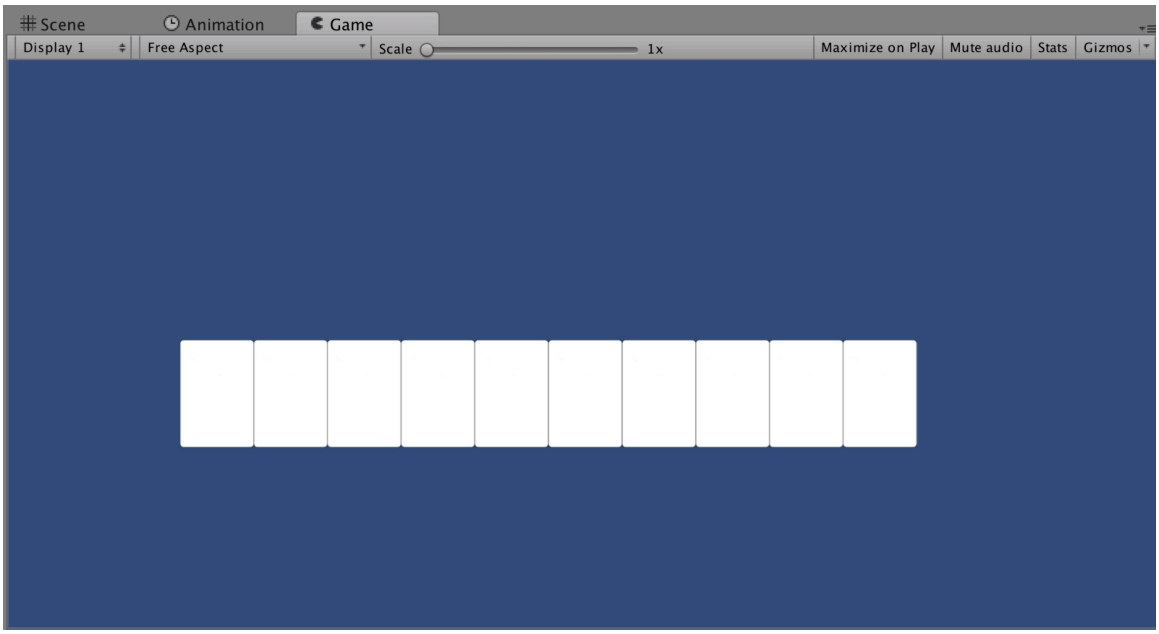


Figure 14: Displaying and scaling the cards

ASSOCIATING THE CORRECT IMAGE TO EACH CARD

At this stage, we can display the hidden cards, however, their value is the same (i.e., they are all showing the same sprite); if you play the scene and click on each of the images, they will all display the **2 of hearts**, as illustrated on the next figure.

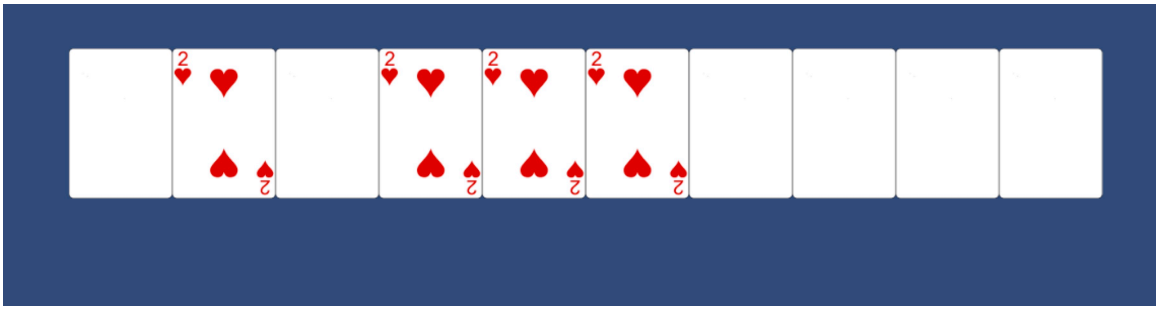


Figure 15: Displaying cards with a similar image

So, the idea now is to specify a corresponding sprite for each card; so we will do the following:

- Create 10 tags.
- Assign a tag to each of these cards based on their rank (e.g., first card from the left will use the tag called **1**, the second card from the left will use the tag called **2**, and so on).
- Associate an image to each card based on its tag (e.g., **ace for tag 1**, **two for tag 2**, and so on).

So let's proceed:

- Please select the prefab called **tile** in the **Project** window.
- Using the **Inspector**, click to the right of the label called **Tag**.

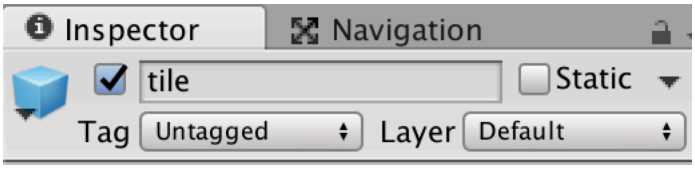


Figure 16: Checking tags already created

If you find it difficult to select (i.e., click on) some of the cards, it may be because their collider is too small and needs to be resized, hence collision and clicks might only be detected on a portion of the card rather than on the entire card.

- If you have already completed the previous chapters, you should see that the tags **1**, **2**, **3**, and **4** have already been created; if not, we can create these in the next steps.
- Please click on **Add Tag**.

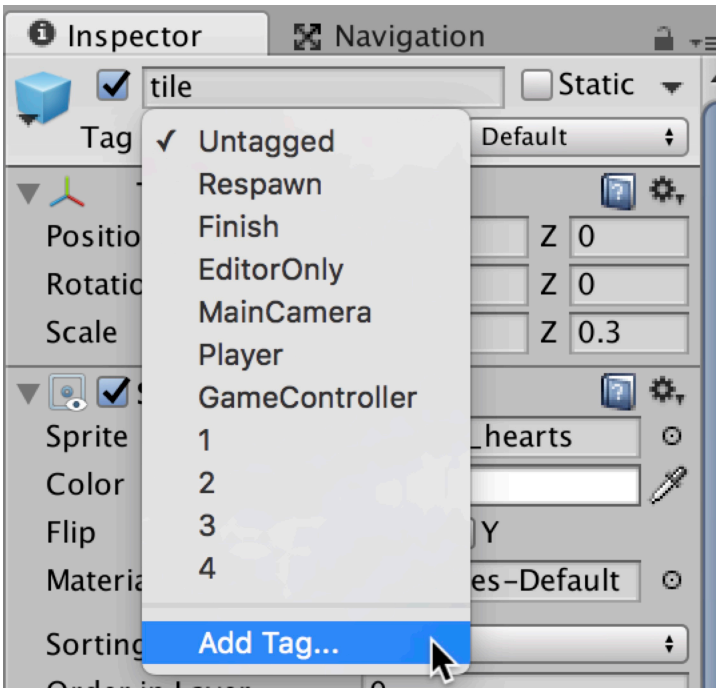
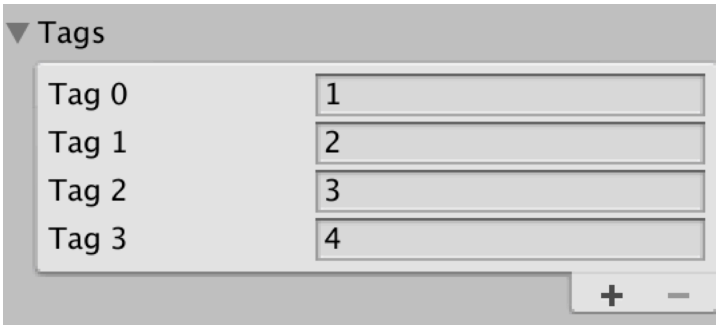


Figure 17: Creating tags

- In the next window, please click several times on the button +, as illustrated in the next figure.



- This will create new fields that you can use to create additional tags, by typing a number (for a new tag) to the right of the fields which name starts with **Tag,** as illustrated on the next figure.

Figure 18: Creating the 11 tags

- Once this is done, you should have **11** tags ranging from **0** to **10**, as per the previous figure.

We can then use these tags from our code.

- Please open the script called **ManageCards**.
- Add the following code at the end of the function **addACard**.

```
c.tag = ""+rank;
```

- Please save your code.

As you play the scene, and if you click on one of the cards that has been created in the **Hierarchy**, you will see, using the **Inspector** that each card has a tag that ranges from **0** to **9**.

Note that tags cannot be created from a script; they have to have already been defined in the editor before they can be applied to objects in a script.

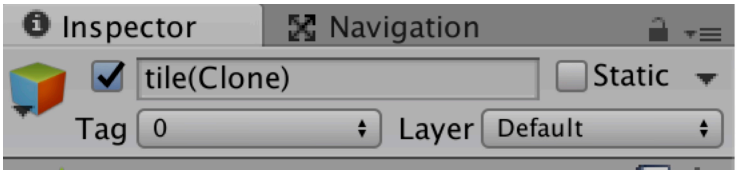


Figure 19:Checking the new tags

Next, we will work on the images that need to be displayed when a card is revealed. For this, we will access the images that we have imported in our project directly from our script; so the next steps will consist in:

- Moving all the images that we have imported to a "recognized" or "standard" folder that we can access from our script.
- Accessing the images from this folder.
- Associating a corresponding image based on the rank of the card created.

So let's proceed:

If you have already completed the previous chapters, then you would already have created a **Resources** folder (within the **Assets** folder), as illustrated in the next figure.

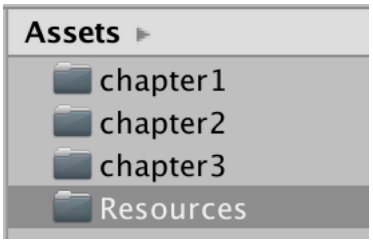


Figure 20: Checking the folder Resources

If this is not the case, we can create this folder as follows:

- In the **Project** window, select the folder called **Assets**.

- Then, from the **Project** window, select **Create | Folder**.

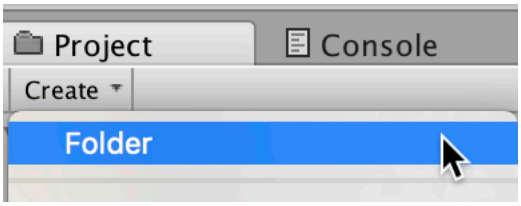


Figure 21: Creating a new folder

- Rename the new folder **Resources** (i.e., right-click on the folder and then select the option **Rename**).

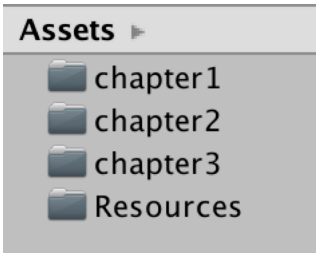


Figure 22: Renaming the folder

Now that this folder has been created, you can move the images that you have imported inside this folder, within Unity, as follows:

- Within Unity, navigate to the folder called **cards** where all the cards have been stored previously.
- Select all the cards (i.e., **CTRL + A**).
- Move these cards (i.e., drag and drop them) to the folder called **Resources**, as illustrated in the next figure.

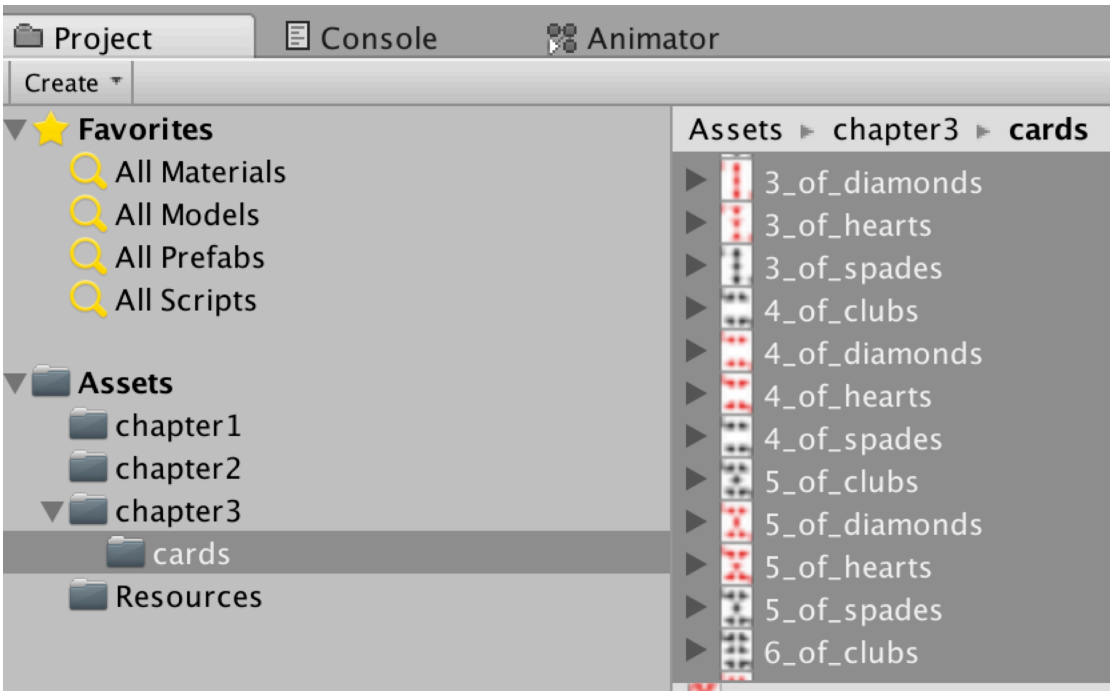


Figure 23: Moving the images to the Resources folder

- If you then check the content of the folder called **Resources**, you should see that the cards were moved successfully, as illustrated in the next figure.

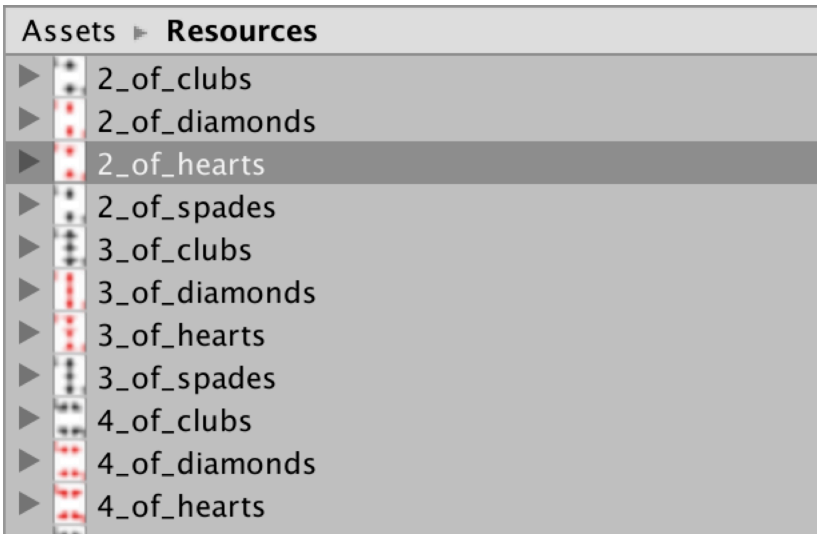


Figure 24: Checking that the cards have been copied properly

Next, we will modify the script **ManageCards** so that the correct images from the **Resources** folder are assigned to each card.

- Please open the script **ManageCards**.
- Add this code at the end of the function **addACard**.

```
c.name = "" + rank;
string nameOfCard = "";
string cardNumber = "";
if (rank == 0)
     cardNumber = "ace";
else
     cardNumber = "" + (rank+1);
nameOfCard = cardNumber + "_of_hearts";
Sprite s1 = (Sprite) (Resources.Load<Sprite>(nameOfCard));
print ("S1:" + s1);
GameObject.Find(""+rank).GetComponent<Tile>  ().setOriginalSprite
(s1);
```

In the previous code:

- The name of the sprite to be selected for a particular card will be in the form: **XX_of_hearts**; where **XX** can be a number (e.g., 1, 2, 3, 4, 5, 6, etc.) or a string (e.g., ace). So the purpose of this code is to form this word based on the card that we have just added and its rank; the first card should be the **ace of hearts**, the second card a **2 of heart**s, and so on.
- We set the name of the new card.
- We declare a variable called **nameOfCard** that will be used to save the name of the corresponding sprite for a particular card.
- We form the first part of the name of the card based on its rank.
- Once the name of the sprite to be used is formed properly and stored in the variable **nameOfCard**, we access the corresponding sprite and save it in the variable called **s1**.
- We then set the **originalSprite** variable for this card to the sprite **s1**.

You can now save your code, and test the scene. You should see that if you click on each of the cards, they will show the correct image.

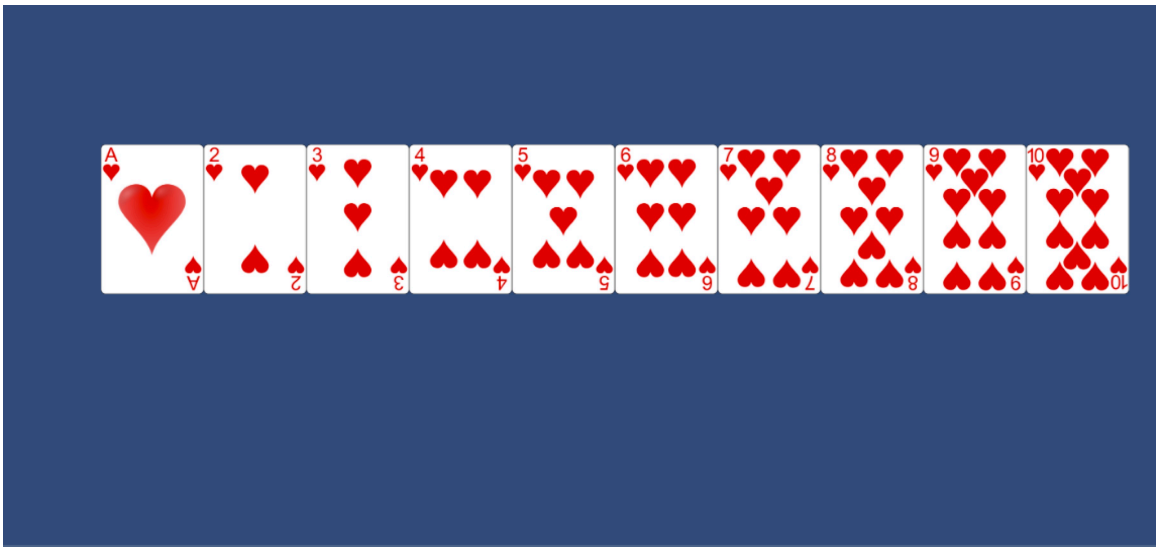


Figure 25: Displaying all the cards.

SHUFFLING THE CARDS

Now we just need to be able to shuffle the cards; the idea will be to create two rows of cards with identical sets of cards but shuffled. So we will create a new function that will shuffle the cards for us, and then call it from the **Start** function, after the cards have been added to the game view.

To do so, we need to differentiate between the order in which the cards are picked (this will determine their position) and their value (this will determine the value of the card and the corresponding sprite).

- Please create the following function in the class **ManageCards**.

```
public int []  createShuffledArray()
{
      int [] newArray = new int [] {0,1,2,3,4,5,6,7,8,9};
      int tmp;
      for (int t = 0; t < 10; t++ )
      {
            tmp = newArray[t];
            int r = Random.Range(t, 10);
            newArray[t] = newArray[r];
            newArray[r] = tmp;
      }
      return newArray;
}
```

In the previous code:

- We declare a new function called **createShuffledArray**; it returns an array of integers for which the values are shuffled.
- We declare a new array of integers that includes the labels of all 10 cards; these values are ordered in ascending order.
- We then loop through this array and shuffle its content.
- Once this is done, we return an array that includes these values in a random order.

This shuffling function is based on the Fisher-Yates algorithm.

Next, we will need to modify the function **addACard** so that it accounts for a card's rank and its value:

- Please modify the definition of the function **addACard** as follows:

```
void addACard(int rank, int value)
```

- Modify the function **addACard** as follows (new code in bold):

```
c.tag = ""+(value+1);
c.name = "" + value;
string nameOfCard = "";
string cardNumber = "";
if (value == 0)
     cardNumber = "ace";
else
     cardNumber = "" + (value+1);
nameOfCard = cardNumber + "_of_hearts";
Sprite s1 = (Sprite) (Resources.Load<Sprite>(nameOfCard));
GameObject.Find(""+value).GetComponent<Tile> ().setOriginalSprite
(s1);
```

- Last, we will modify the function **displayCards** as follows (new code in bold):

```
int [] shuffledArray = createShuffledArray();
for (int i = 0; i < 10; i++)
{
     //addACard (i);
     addACard (i,shuffledArray[i]);
}
```

- Please save your script and test the game; you should see that, after clicking on some of the cards, that these cards have been shuffled, as described on the next figure.

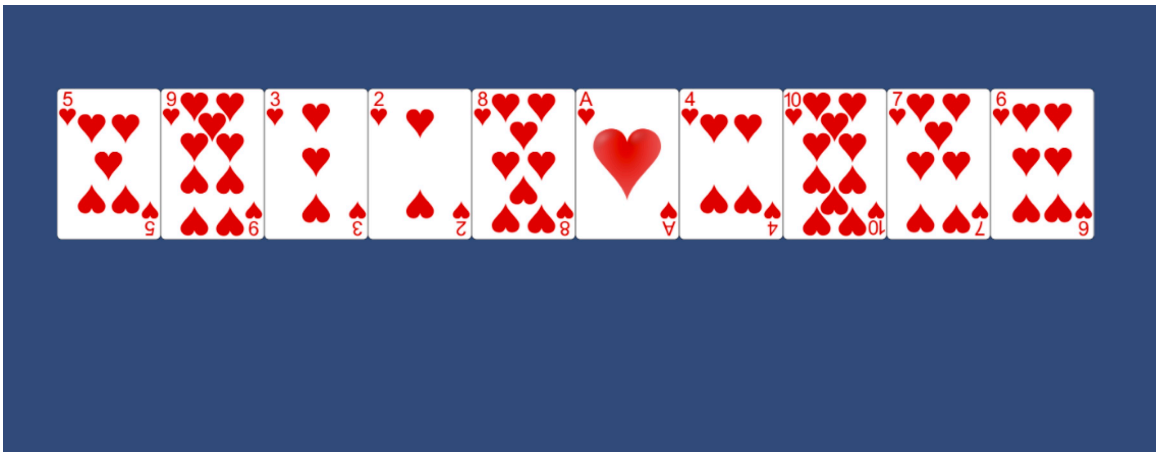


Figure 26: Displaying the card (after shuffling)

If you find it difficult to select (i.e., click on) some of the cards, it may be because their collider is too small and needs to be resized, hence collision and clicks might only be detected on a portion of the card rather than on the entire card.

Last but not least, we want to create two rows of cards. So it will be the same as we have done so far, except that we will modify the function **addACard** so that we can specify the row where the card should be added (i.e., first or second row).

- Please change the definition of the function **addACard** as follows (new code in bold):

```
void addACard(int row, int rank, int value)
{
```

In the previous code, we have modified the function so that it takes a third parameter named **row**.

- Next, please modify the function **displayCards** as follows (new code in bold):

```
int [] shuffledArray = createShuffledArray();
int [] shuffledArray2 = createShuffledArray();
for (int i = 0; i < 10; i++)
{
//addACard (i);
      addACard (0, i, shuffledArray[i]);
      addACard (1, i, shuffledArray2[i]);

}
```

In the previous code:

- We declare another array of integers called **shuffledArray2**, that will be used for the second row of cards.
- In the **for** loop, we then call the function **addACard** to display both the first and the second row.
- When the function **addACard** is called, three parameters are passed: the **row** (0 or 1), the **rank** of the card (i.e., its **position** in the row), and its value (i.e., ace, 1, or 2, etc.).

Next, we can modify the function **addACard** as follows:

```
//Vector3 newPosition = new Vector3 (cen.transform.position.x +
((rank-10/2)       *scaleFactor),       cen.transform.position.y,
cen.transform.position.z);
float yScaleFactor = (725 * cardOriginalScale) / 100.0f;
Vector3 newPosition = new Vector3 (cen.transform.position.x +
((rank-10/2) *scaleFactor), cen.transform.position.y + ((row-2/2)
*yScaleFactor), cen.transform.position.z);
```

In the previous code:

- We comment the previous line that was used to set the new position of the card.
- We define a new variable called **yScaleFactor** that will be used to calculate the position of the card (especially its y coordinate).
- We then define the new position of the card using the parameter called **row** and the variable called **yScaleFactor** to centre it properly.

Next, we just need to change the naming of the new card.

- Please replace this code:

```
c.name = "" + value;
```

- ...with this code...

```
c.name = ""+row+"_"+value;
```

In the previous code, we specify that the cards from the first row have a name starting with 0 **(row = 0)** and that the cards from the second row have a name starting with 1 **(**i.e., **row = 1)**.

- Please replace this code...

```
GameObject.Find(""+value).GetComponent<Tile>  ().setOriginalSprite
(s1);
```

- ... with this code

```
GameObject.Find(""+row+"_"+value).GetComponent<Tile>
().setOriginalSprite (s1);
```

- Please save your code.

As you play the scene, you should now have two rows of cards, as illustrated on the next figure.

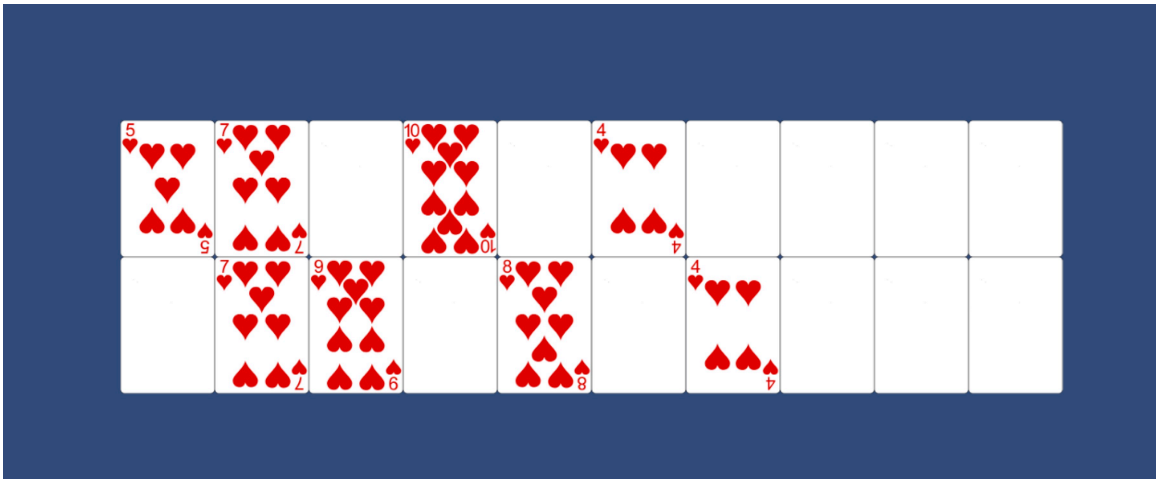


Figure 27: Displaying two rows of cards

If you find it difficult to select (i.e., click on) some of the cards, it may be because their collider is too small and needs to be resized, hence collision and clicks might only be detected on a portion of the card rather than on the entire card.

So, in this case you can resize the collider as follows:

- Select the prefab called **tile** in the **Project** window.
- Scroll down to the component called **Box Collider 2D**.
- Change the size of the collider to (**x=5**, **y = 7**).

ALLOWING THE PLAYER TO CHOOSE CARDS FROM EACH ROW

So now that we can shuffle and display two rows of 10 cards, we just need to make it possible for the player to select a card from the first row, then a card from the second row, and then check if these cards are similar (i.e., have the same value) by comparing their tags and value.

Based on our code, we know that the cards from the first row have a name starting with **0** and that cards from the second row have a name starting with **1**, as illustrated in the next figure.

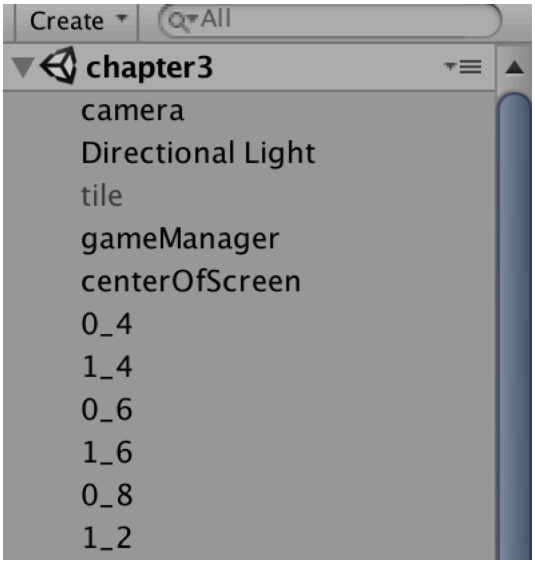


Figure 28: Naming the cards

If you find it difficult to select (i.e., click on) some of the cards, it may be because their collider is too small and needs to be resized, hence collision and clicks might only be detected on a portion of the card rather than on the entire card.

So we will proceed as follows:

- Display the cards (i.e., two rows).
- Allow the player to successively pick (i.e., click on) two cards.
- We will then check that whenever the player clicks on a card for the first time, that the name of this card includes **0** (i.e. that it belongs to the first row).

- We will also check that whenever the player clicks on a second card, that the name of this card includes **1** (i.e. that it belongs to the second row).
- After that we will compare their tags.
- If these cards have the same tag (i.e., the same value) then both will be destroyed.

Next, we can implement the code that checks how we handle clicks on cards.

- Please add the following code at the start of the script **ManageCards**.

```
private bool firstCardSelected, secondCardSelected;
private GameObject card1, card2;
```

- Please add the following code just before the end of the class.

```
public void cardSelected(GameObject card)
{
      if (!firstCardSelected)
      {
            firstCardSelected = true;
            card1 = card;
            card1.GetComponent<Tile> ().revealCard ();
      }
}
```

In the previous code:

- We declare a new function called **cardSelected**; this function will be used to monitor whether we have already selected the first card.
- If this is not the case, then the card that was passed as a parameter (i.e., the card currently selected – the first card -) is saved in the variable **card1**.
- We also display (i.e., reveal) the value of this card.

The hiding and revealing of the cards will now be handled by the game manager; so we will modify the function called **OnMouseDown** in the **Tile** class, as follows:

```
public void OnMouseDown()
{
      print ("You pressed on tile");
      /*if (tileRevealed)
            hideCard ();
      else
            revealCard ();*/
      GameObject.Find    ("gameManager").GetComponent<ManageCards>
().cardSelected (gameObject);
}
```

In the previous code:

- We comment the previous code.
- Now, when the player clicks on a card, the function called **cardSelected** (that we have defined earlier), is called.
- The card that the player has just selected is also passed as a parameter.

Next, we will further code the overall management of the game.

- Please add the following code at the beginning of the class called **ManageCards**.

```
private string rowForCard1, rowForCard2;
```

- Modify the function **cardSelected** as follows:

```
public void cardSelected(GameObject card)
{
      if (!firstCardSelected)
      {
            string row = card.name.Substring (0, 1);
            rowForCard1 = row;
            firstCardSelected = true;
            card1 = card;
            card1.GetComponent<Tile> ().revealCard ();
      }
      else if (firstCardSelected && !secondCardSelected)
      {
            string row = card.name.Substring (0, 1);
            rowForCard2 = row;
            if (rowForCard2 != rowForCard1)
            {
                  card2 = card;
                  secondCardSelected = true;
                  card2.GetComponent<Tile> ().revealCard ();
            }
      }
}
```

In the previous code:

- If the player selects the first card, we record the name of the row for this card, we save this card in the variable **card1**, and we also reveal the card.
- Then, if the player is selecting the second card, we record the name of the row for this card, and check that it is a different row than the first card selected.
- If this is the case, then we save this card in the variable **card2**, and reveal this card also.
- We also check whether we have a match, using the function **checkCard** that we yet have to create.

You can save both scripts now (i.e., **Tile** and **ManageCards**), and test the scene; as you try to select two cards, you should only be able to choose one card from the first row and a second card from the second row (or vice versa).

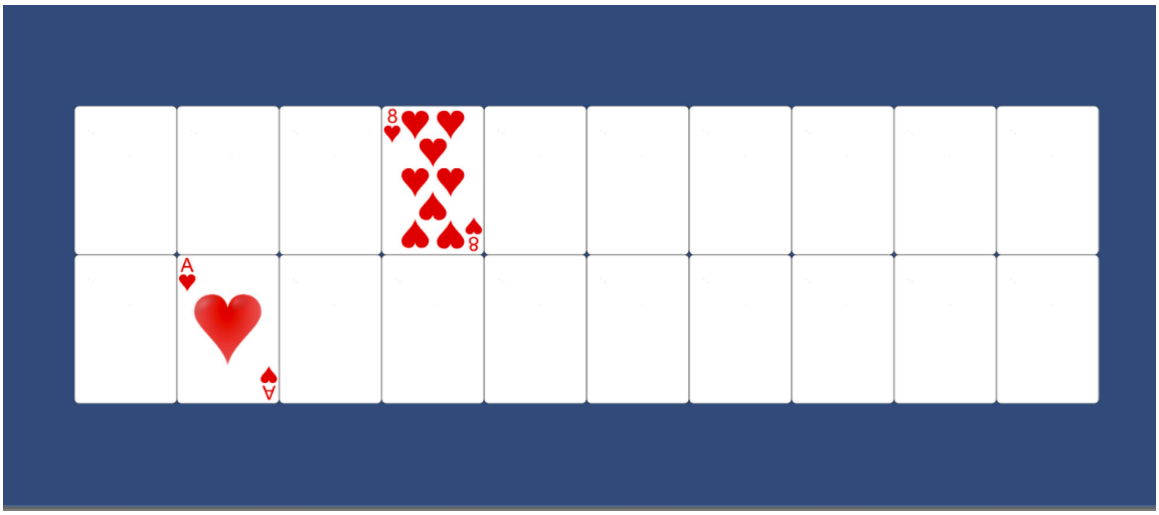


Figure 29: Picking one card from each row

CHECKING FOR A MATCH

Now we just need to determine when there is a match between the two cards selected by the player. In this case, we will delete both cards.

First let's create a function called **checkCards**; this function will check whether the two cards selected by the player are identical; if this is the case, they will be destroyed; otherwise, they will be hidden again; the checking will happen after a slight pause, so that the player can see the cards that have been selected before they are hidden again (if this is the case).

- Please add the following function to the class **ManageCards**:

```
public void checkCards()
{
      runTimer ();
}
```

In this function we call another function called **runTimer** (that we yet have to create).

- Please add the following code at the beginning of the class.

```
bool timerHasElapsed, timerHasStarted;
float timer;
```

In the previous code:

- The new variables defined will be employed to create a timer that will be used to pause after the second card has been collected.
- **timerHasElapsed** will be used to know whether the **pause time** has elapsed.
- **timerHasStarted** is used to determine whether the timer has already been started.
- The variable **timer** will be used to count the number of seconds between when the second card has been selected and when we start to compare the two cards.

Please add the following function to the class:

```
public void runTimer()
{
      timerHasElapsed = false;
      timerHasStarted = true;
}
```

In the previous code, we just initialise the timer and the associated variables.

Next, we will create and implement the timer; it will be done in the **Update** function; the timer will increase until it reaches **2** seconds; after two seconds, we will start to compare the two cards.

- Please add the following code to the **Update** function.

```
void Update ()
{
      if (timerHasStarted)
      {
            timer += Time.deltaTime;
            print (timer);
            if (timer >= 1) {
                  timerHasElapsed = true;
                  timerHasStarted = false;
                  if (card1.tag == card2.tag) {
                        Destroy (card1);
                        Destroy (card2);
                  } else {
                        card1.GetComponent<Tile> ().hideCard ();
                        card2.GetComponent<Tile> ().hideCard ();
                  }
                  firstCardSelected = false;
                  secondCardSelected = false;
                  card1 = null;
                  card2 = null;
                  rowForCard1 = "";
                  rowForCard2 = "";
                  timer = 0;
            }
      }
}
```

In the previous code

- We check if the timer has started.
- We then check whether the timer has started and that we have reached the end of the pause.
- In this case, we reinitialize the timer (using the variables t**imerHasElapsed** and **timerHasStarted**).
- We check if the cards match and destroy (or hide) them both if we have a match.
- We then initialise the variable linked to the card selection, so that the player can restart the process of selecting two cards.

- Last, please check that the function **cardSelected** includes a call to the function **checkCards**, as highlighted in the next code (in bold).

```
secondCardSelected = true;
card2 = card;
card2.GetComponent<Tile> ().revealCard ();
checkCards ();
```

- Please save the code and test the game.

You should see that as you select one card from each row: if they match, then they should be destroyed after a few seconds.

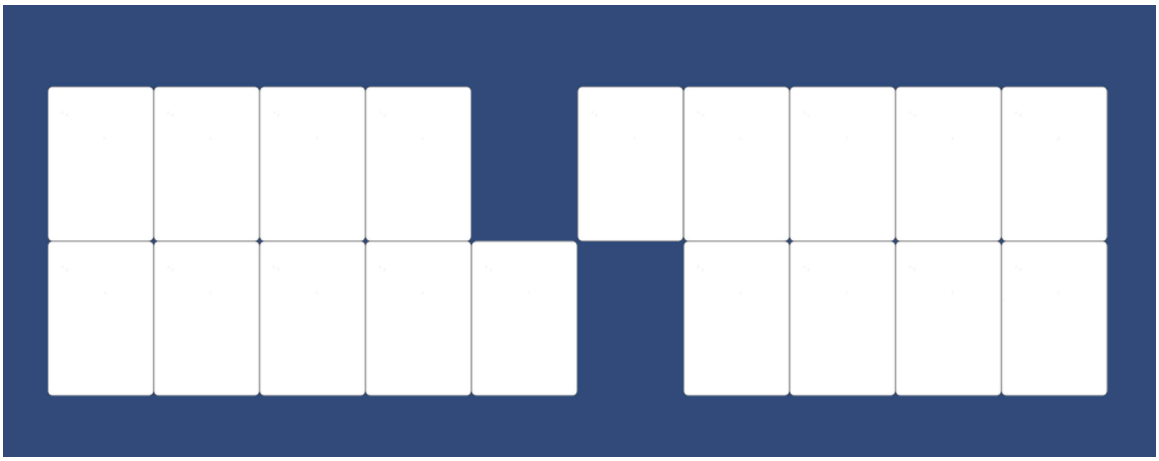


Figure 30: The game after two cards were matched

Next we will add a sound when the player has managed to match two cards:

- Please add an **AudioSource** component to the object **gameManager** (i.e., select: **Component | Audio | AudioSource**).
- Import the audio file called **ok.mp3** from the resource pack to the Unity **Project** window.
- Drag and drop this audio clip from the **Project** window to the **AudioClip** attribute of the component **Audio Source** for the object **gameManager**, as illustrated in the next figure.

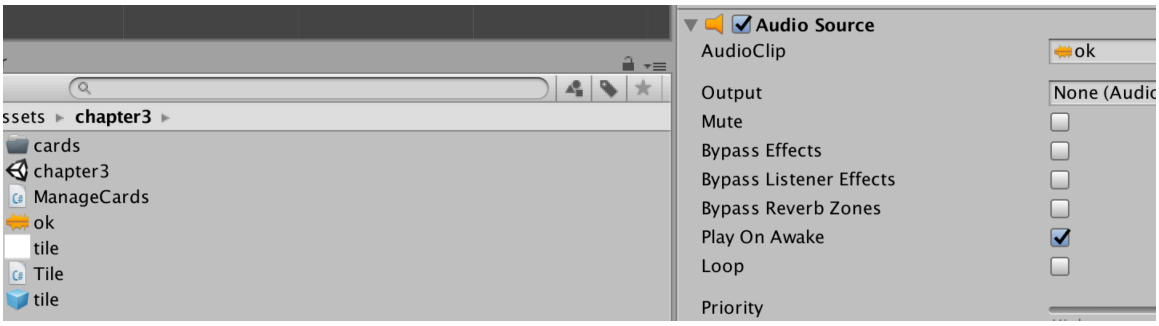


Figure 31: Adding an Audio Clip to the Audio Source

- Set the attribute **Play on Awake** to false.

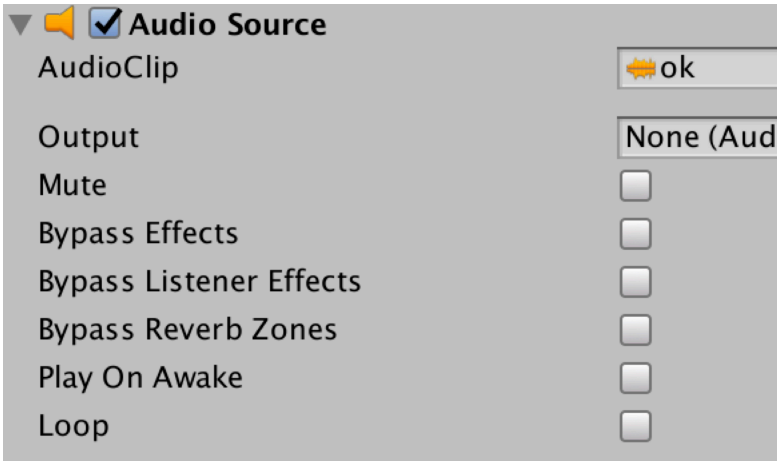


Figure 32: Setting the attribute Play on Awake

Next, we need to check how many cards the player has managed to match.

- Please, add this code at the beginning of the class **ManageCards**:

```
int nbMatch = 0;
```

- Add this code at the beginning of the script.

```
using UnityEngine.SceneManagement;
```

- Modify the function **Update** as follows (new code in bold).

```
Destroy (card1);
Destroy (card2);
nbMatch++;
if (nbMatch == 10)
     SceneManager.LoadScene (SceneManager.GetActiveScene().name);
```

In the previous code, we reload the current scene if the player has managed to find the 10 sets of identical cards.

2
FREQUENTLY ASKED QUESTIONS

This chapter provides answers to the most frequently asked questions about the features that we have covered in this book. Please also note that some videos are also available on the companion site to help you with some of the concepts covered in this section, including networking, performances, database access, or procedural level creation.

READING FILES

How can I access a text file that I have saved in the Resources folder?

To access this file (and the text within) you can use the following code snippet.

```
TextAsset      t1      =      (TextAsset)Resources.Load("words",
typeof(TextAsset));
string s = t1.text;
```

How can I access an image file that I have saved in the Resources folder?

To access an individual sprite from the **Resources** folder, you can use the following code snippet.

```
Sprite s1 = (Sprite) (Resources.Load<Sprite>(nameOfCard));
```

However, if you'd prefer to access and save several sprites at once, then you can use the following snippet:

```
Sprite[] allSprites = Resources.LoadAll<Sprite> ("lion");
```

DETECTING USER INPUTS

How can I detect keystrokes?

You can detect keystrokes by using the function **Input.GetKey**. For example, the following code detects when the key **E** is pressed; this code should be added to the **Update** function.

```
If (Input.GetKey(KeyCode.E)){...}
```

How can I detect a click on a button?

To detect clicks on a button, you can do the following:

- Create an empty object.
- Create a new script and link it to this object.
- Select the button in the **Hierarchy**.
- Using the **Inspector**, click on the button located below the label called **OnClick**.
- Drag and drop the empty object to the field that just appeared.
- From the **Inspector**, you should then be able to select the function that should be called in case the button is clicked.

How can I detect drag and drop events on an image?

To detect drag and drop events or actions on an image:

- Select the image from the **Hierarchy**.
- Using the **Inspector**, add a component called **Event Trigger** (**Component | Event | Event Trigger**) to this image.

In the component **Event Trigger**, click on the button called **Add New Event Type** (as illustrated in the next figure) and then choose the option called **Drag or End Drag** from the drop-down menu.

3
THANK YOU

I would like to thank you for completing this book; I trust that you are now comfortable with the creation of simple card games. This book is of course only a quick guide on this topic; if you would like to know more about Unity, you may try some of my other books available from the official page: **http://www.learntocreategames.com/books**.

So that the book can be constantly improved, I would really appreciate your feedback and hear what you have to say. So, please leave me a helpful review on Amazon letting me know what you thought of the book and also send me an email (**learntocreategames@gmail.com**) with any suggestion you may have. I read and reply to every email.

Thanks so much!!