8
Developing Glow Hockey from Scratch in Unity 5

This chapter will show you how easy it is to develop a prototype of the most popular game on Android Market—*Glow Hockey*, right from scratch in Unity 5. You will learn how to create a camera for any screen resolutions and any screen sizes. Also, there you will see in practice how easy it is to use physics. You will learn in practice to design beautiful effects, animations, physical behaviors, and other different real-world features and techniques for your Android games and applications. Also, you will learn how to optimize your project and any other real-world projects for Android devices. Much more useful things and features will be covered in this chapter.

Game architecture

In the beginning of development, it is very useful to understand exactly what we need as the end result. Let's start our project from game concept, its features, and basic ideas. We have the following states in our game example:

- `AskToBuyFullVersion`
- `MainMenu`
- `About`
- `Settings`
- `SettingsChangePaddles`
- `SettingsChangePuck`
- `Difficulty`
- `GameStarted`
- `GamePaused`
- `GameFinished`

Glow Hockey's game states, from the first to the seventh, are implemented as different menu screens. The last three states—`GameStarted`, `GamePaused`, and `GameFinished`—are implemented on one single game screen. The `GamePaused` and `GameFinished` states are shown as an overlay for the main game screen. Also, in our *Glow Hockey* game clone we have different game modes:

- `Easy`
- `Medium`
- `Hard`
- `Insane`
- `TwoPlayers`

The first four game modes (`Easy`, `Medium`, `Hard`, and `Insane`) is where a player plays against the computer's AI. Also, in our game example, there are four different colors that you can choose from for the two paddles and the puck: yellow, red, green, and blue. In the game settings, you can enable or disable next options, such as sounds, particle effects, and vibration.

The screens workflow and core game mechanics

Now, it's time to understand the main idea behind screen transitions for different game states. This idea is very simple because all we need to do is just to enable or disable different game objects for different game states. Of course, we should prepare different game objects to be children for different or same parents, just as you wish. In our game example, we grouped game objects that are tied logically to the same game state.

The AskToBuyFullVersion screen

Let's start with reviewing our first game state, known as `AskToBuyFullVersion`, and let's look at its object hierarchy view, as listed here.

Let's look at the `Ask To Buy Full Version` game object's hierarchy:

- `Background` (a sprite image)
- The **Buy** button opens the Packt Publishing web URL, as an example, in our game clone based on original *Glow Hockey* from the Android market.
- The **Continue** button changes the game state from `AskToBuyFullVersion` to `MainMenu`.

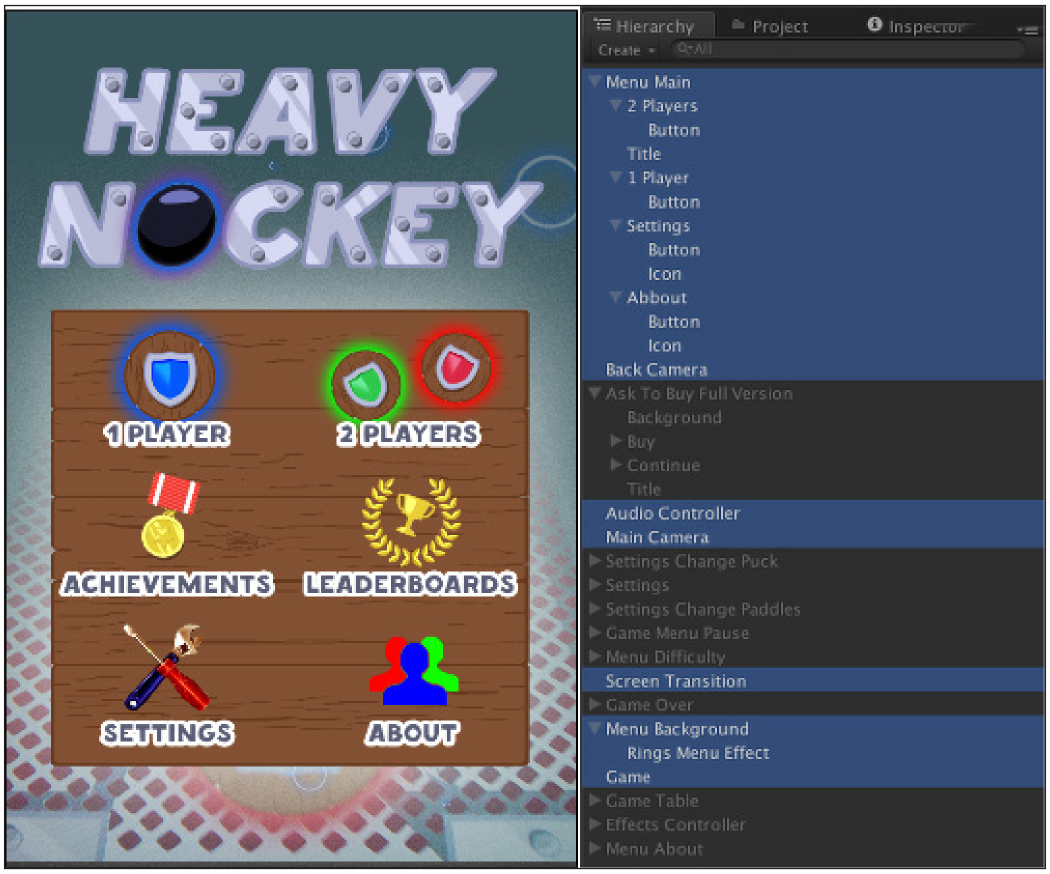
The following game objects are always active in our game:

- `Back Camera`
- `Audio Controller`
- `Main Camera`
- `Game`
- `Screen Transition`

All other game objects are enabled or disabled based on the current game state.

The MainMenu screen

The `MainMenu` screen is based on the same game state and it is shown in the following screenshot:



```
▼ Menu Main
   ▼ 2 Players
        Button
     Title
   ▼ 1 Player
        Button
   ▼ Settings
        Button
        Icon
   ▼ Abbout
        Button
        Icon
  Back Camera
▼ Ask To Buy Full Version
     Background
   ▶ Buy
   ▶ Continue
     Title
  Audio Controller
  Main Camera
▶ Settings Change Puck
▶ Settings
▶ Settings Change Paddles
▶ Game Menu Pause
▶ Menu Difficulty
  Screen Transition
▶ Game Over
▼ Menu Background
     Rings Menu Effect
  Game
▶ Game Table
▶ Effects Controller
▶ Menu About
```

In order to make a transition from the `AskToBuyFullVersion` game state to the `MenuMain` game state, we need to disable the `Ask To Buy Full Version` game object and enable the `Menu Main` and `Menu Background` game objects. The `Menu Background` game object must be disabled only for the following game states:

- `AskToBuyFullVersion`
- `GameStarted`
- `GamePaused`
- `GameFinished`

For all other game states (except the four states listed earlier in the text), the `Menu Background` game object must be activated. The `Menu Main` game screen has the following game object's hierarchy:

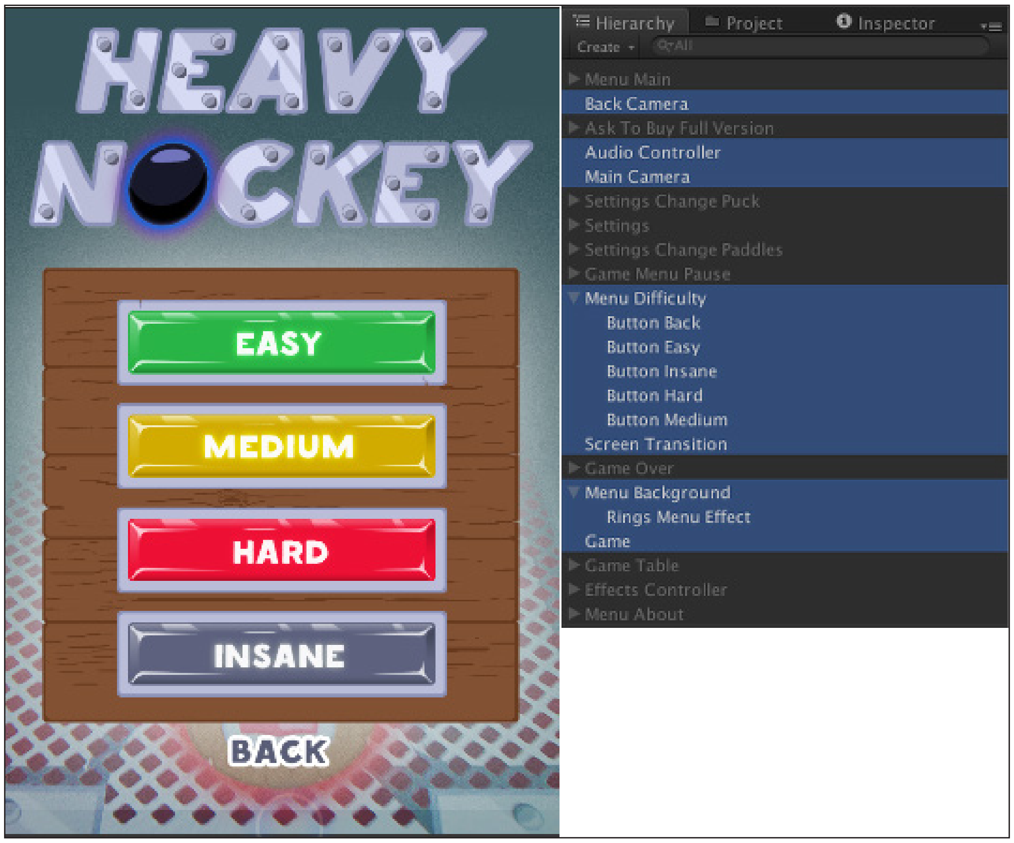
- `Title` (a sprite image)
- The **One Player** button changes the game state from `MainMenu` to `Difficulty`.
- The **Two Players** button changes the game state from `MainMenu` to `GameStarted`. Also, this button changes the game mode to `TwoPlayers`.
- The **Settings** button changes the game state from `MainMenu` to `Settings`.
- The **More Apps** button opens on the Android devices Packt Publishing market URL as an example in our game.
- The **About** button changes the game state from `MainMenu` to `About`.

The Difficulty screen

The `Difficulty` screen is based on the same game state and it is shown in the following screenshot. In order to make the transition from the `MainMenu` game state to the `Difficulty` game state (by pressing the **1 Player** button), we need to disable the `Menu Main` game object and to enable the `Menu Difficulty` game object which has the following structure:

- The **Easy** button changes the game state from `Difficulty` to `GameStarted`. Also, this button changes the game mode to `Easy`.
- The **Medium** button changes the game state from `Difficulty` to `GameStarted`. Also, this button changes the game mode to `Medium`.
- The **Hard** button changes the game state from `Difficulty` to `GameStarted`. Also, this button changes the game mode to `Hard`.
- The **Insane** button changes the game state from `Difficulty` to `GameStarted`. Also, this button changes the game mode to `Insane`.

- The **Back** button changes the game state from `Difficulty` to `MainMenu`.

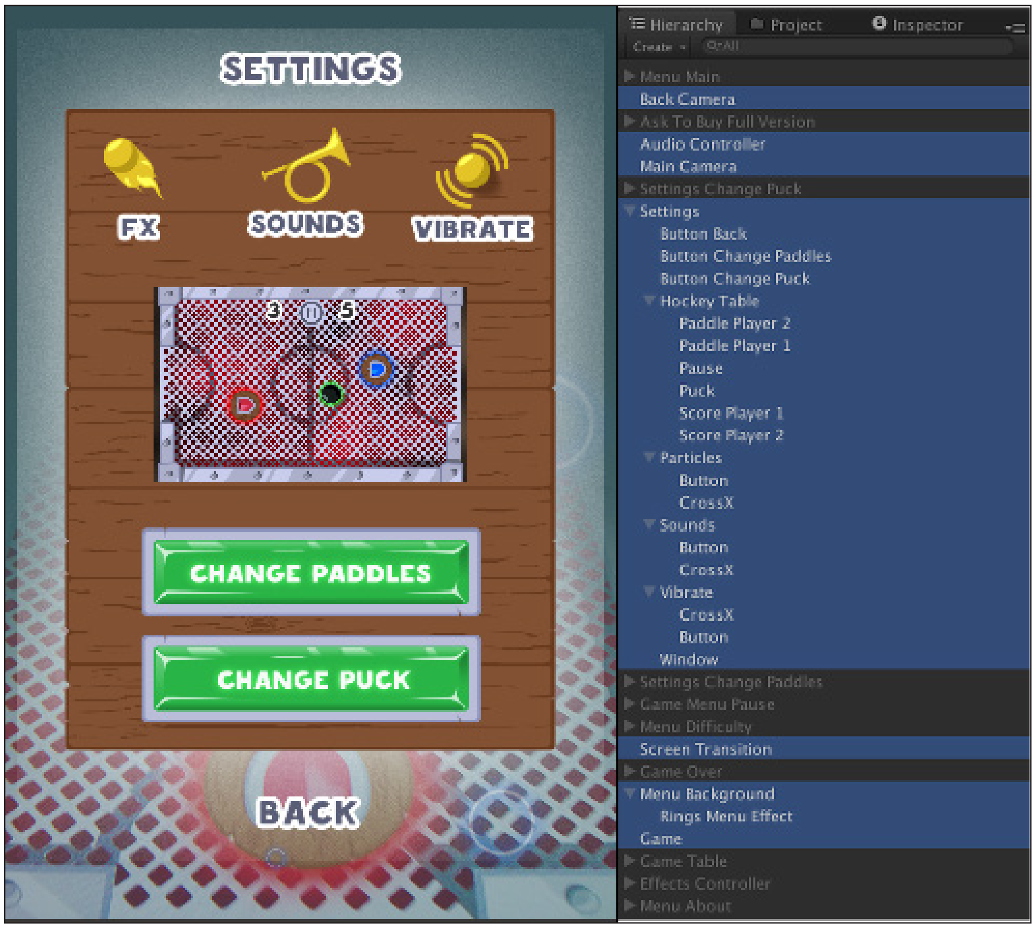


The Settings screen

The `Settings` screen is based on the same game state and it is shown in the following figure. In order to make the transition from the `MainMenu` game state to the `Settings` game state (by pressing the **Settings** button), we need to disable the `Menu Main` game object and to enable the `Settings` game object. The `Settings` game object's hierarchy is listed here:

- The window (a sprite image)
- The hockey table (two paddles, one puck, a pause icon, and two scores for both players)
- The **Particles** button (with cross image to enable and disable this setting) disables and enables particle effects.

- The **Sounds** button (with cross image to enable and disable this setting) disables and enables sounds.
- The **Vibrate** button (with cross image to enable and disable this setting) disables and enables vibration.
- The **Change Paddles** button changes the game state from `Difficulty` to `SettingsChangePaddles`.
- The **Change Puck** button changes the game state from `Difficulty` to `SettingsChangePuck`.
- The **Back** button changes the game state from `Difficulty` to `MainMenu`.
- The preview label is just an empty game object (it is not visible). Therefore, you can completely remove it or you can make `MeshText` from it as an example to show any desired label for your game.



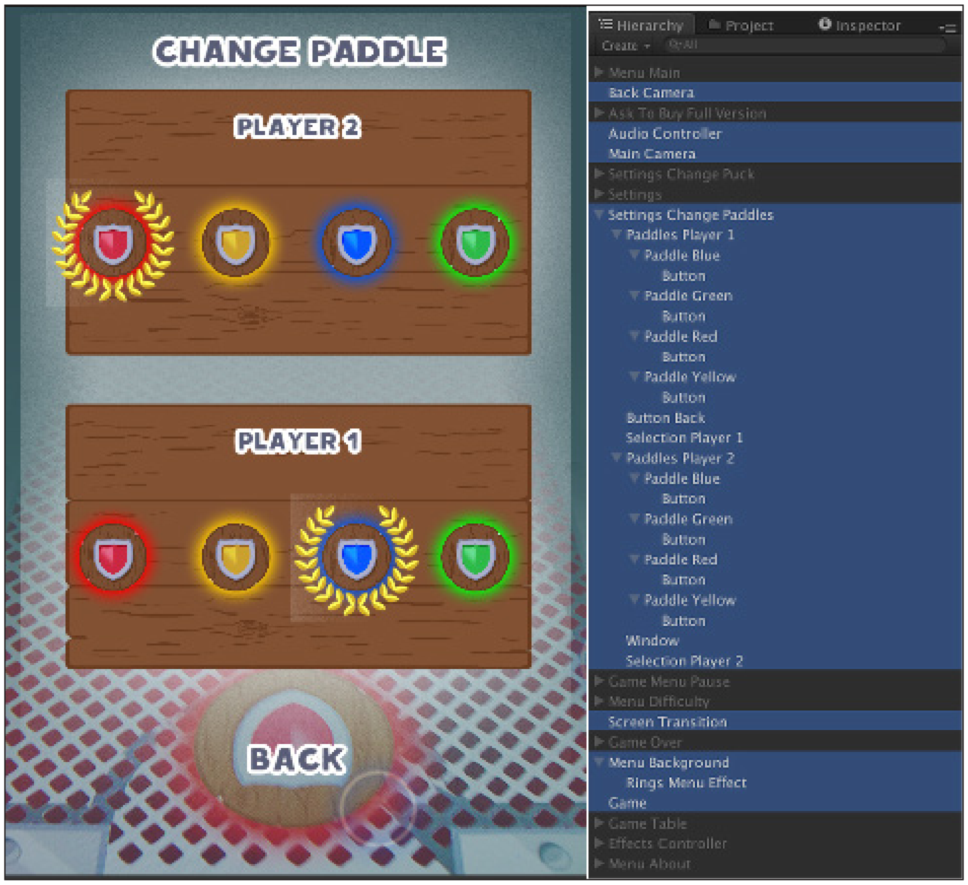
```
▶ Menu Main
  Back Camera
▶ Ask To Buy Full Version
  Audio Controller
  Main Camera
▶ Settings Change Puck
▼ Settings
    Button Back
    Button Change Paddles
    Button Change Puck
  ▼ Hockey Table
      Paddle Player 2
      Paddle Player 1
      Pause
      Puck
      Score Player 1
      Score Player 2
  ▼ Particles
      Button
      CrossX
  ▼ Sounds
      Button
      CrossX
  ▼ Vibrate
      CrossX
      Button
    Window
▶ Settings Change Paddles
▶ Game Menu Pause
▶ Menu Difficulty
  Screen Transition
▶ Game Over
▼ Menu Background
    Rings Menu Effect
  Game
▶ Game Table
▶ Effects Controller
▶ Menu About
```

The ChangePaddles screen

The `ChangePaddles` screen is based on the same game state and it is shown in the following screenshot. In order to make a transition from the `Settings` game state to the `ChangePaddles` game state (by pressing the **Change Paddles** button), we need to disable the `Settings` game object and to enable the `Change Paddles` game object. The `Change Paddles` game object's hierarchy is listed as follows:

- The window (a sprite image)
- `Selection Player 1` (a sprite image)
- `Selection Player 2` (a sprite image)
- The red paddle button (player 1) changes the `Selection Player 1` game object's position to be equal to its own position. Also, it saves the current chosen paddle type, which is displayed in the `Settings` and `Game` screens on their hockey tables.
- The yellow paddle button (player 1) changes the `Selection Player 1` game object's position to be equal to its own position. Also, it saves the current chosen paddle type, which is displayed in the **Settings** and **Game** screens on their hockey tables.
- The blue paddle button (player 1) changes the `Selection Player 1` game object's position to be equal to its own position. Also, it saves the current chosen paddle type, which is displayed in the **Settings** and **Game** screens on their hockey tables.
- The green paddle button (player 1) changes the `Selection Player 1` game object's position to be equal to its own position. Also, it saves the current chosen paddle type, which is displayed in the **Settings** and **Game** screens on their hockey tables.
- The red paddle button (player 2) changes the `Selection Player 2` game object's position to be equal to its own position. Also, it saves the current chosen paddle type, which is displayed in the **Settings** and **Game** screens on their hockey tables.
- The yellow paddle button (player 2) changes the `Selection Player 2` game object's position to be equal to its own position. Also, it saves the current chosen paddle type, which is displayed in the **Settings** and **Game** screens on their hockey tables.
- The blue paddle button (player 2) changes the `Selection Player 2` game object's position to be equal to its own position. Also, it saves the current chosen paddle type, which is displayed in the **Settings** and **Game** screens on their hockey tables.

- The green paddle button (player 2) changes the `Selection Player 2` game object's position to be equal to its own position. Also, it saves the current chosen paddle type, which is displayed in the **Settings** and **Game** screens on their hockey tables.
- The **Back** button changes the game state from `ChangePaddles` to `Settings`.

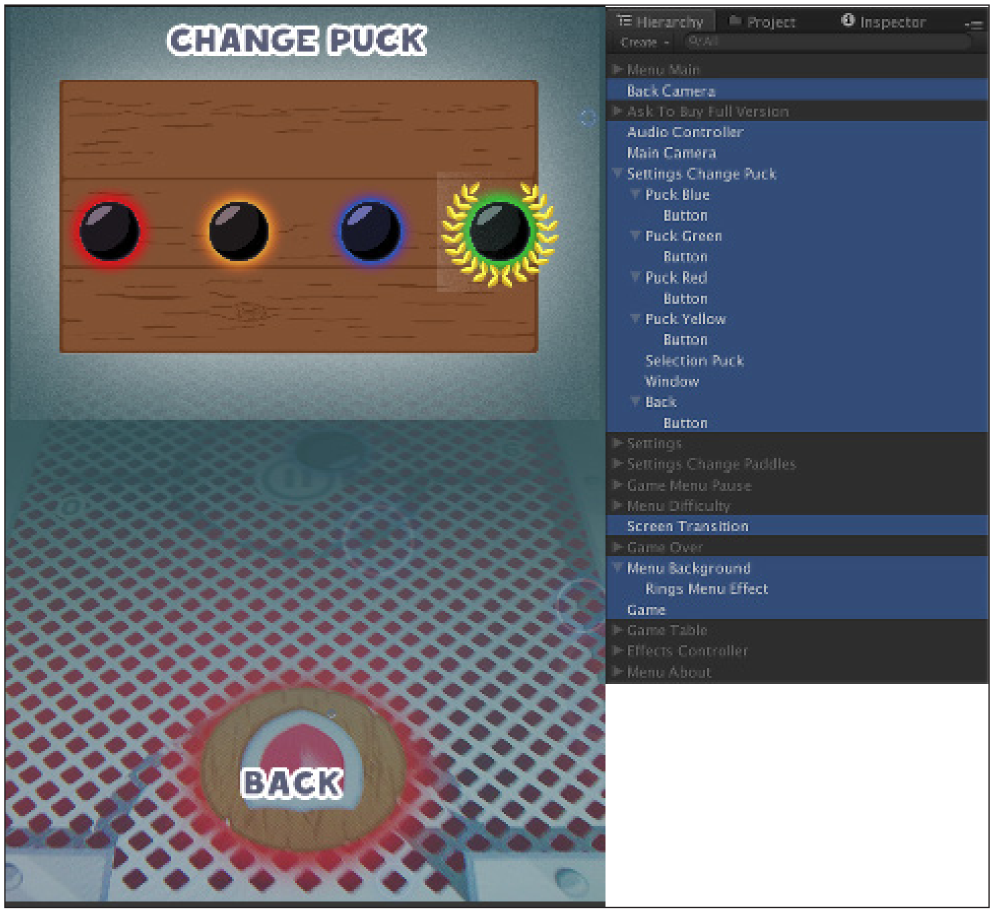


The ChangePuck screen

The `ChangePuck` screen is based on the same game state and is shown in the following screenshot. In order to make the transition from the `Settings` game state to the `ChangePuck` game state (by pressing the **Change Puck** button), we need to disable the `Settings` game object and enable the `Change Puck` game object. The `Change Puck` game object's hierarchy is listed here:

- The window (a sprite image)
- The selection puck (a sprite image)

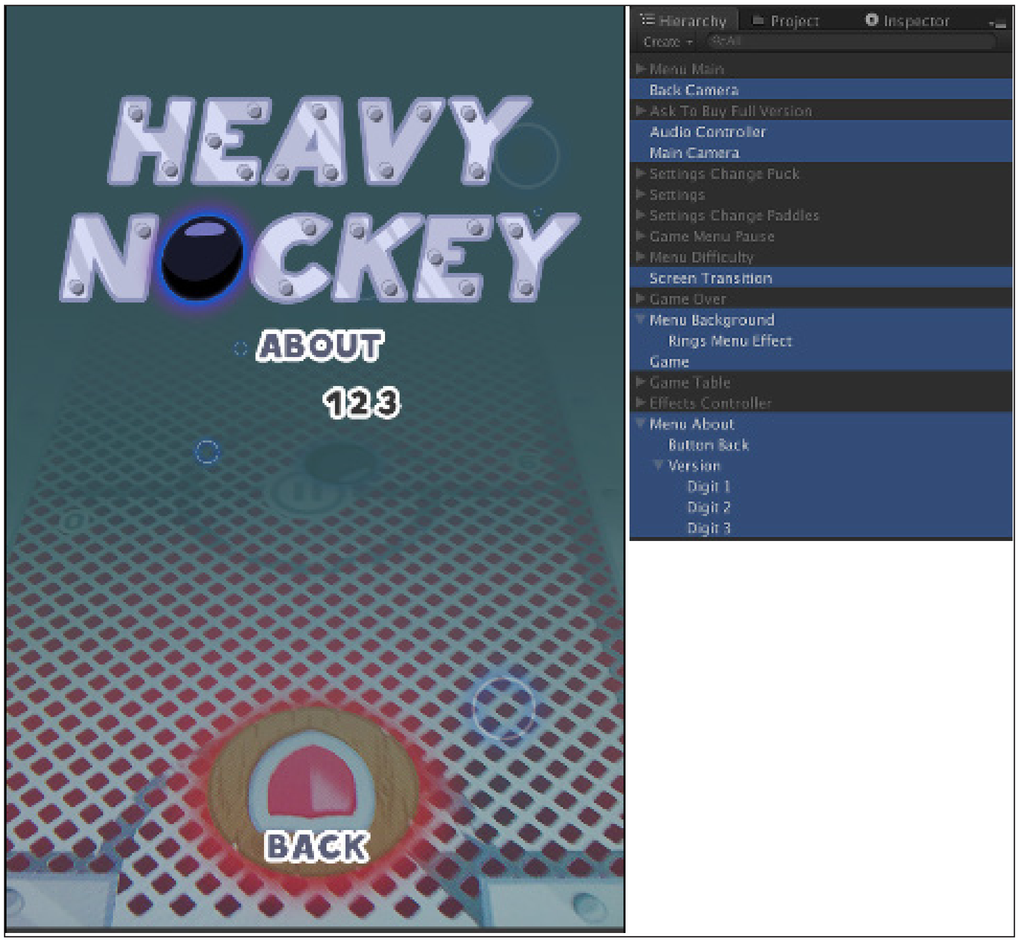
- The red puck button changes the `Selection Puck` game object's position to be equal to its own position. Also, it saves the current chosen puck type, which is displayed in the **Settings** and **Game** screens on their hockey tables.
- The yellow puck button changes the `Selection Puck` game object's position to be equal to its own position. Also, it saves the current chosen puck type, which is displayed in the **Settings** and **Game** screens on their hockey tables.
- The blue puck button changes the `Selection Puck` game object's position to be equal to its own position. Also, it saves the current chosen puck type, which is displayed in the **Settings** and **Game** screens on their hockey tables.
- The green puck button changes the `Selection Puck` game object's position to be equal to its own position. Also, it saves the current chosen puck type, which is displayed in the **Settings** and **Game** screens on their hockey tables.
- The **Back** button changes the game state from `ChangePaddles` to `Settings`.



The About screen

The **About** screen is based on the same game state and it is shown in screenshot here. In order to make the transition from the `MainMenu` game state to the `About` game state (by pressing the **About** button), we need to disable the `Menu Main` game object and enable the `About` game object. The `About` game object's hierarchy is listed here:

- The version (a root game object of sprite images) shows the current game version as an example version equal to **123**.
- The **Back** button changes the game state from `About` to `MainMenu`.



The Game screen

The **Game** screen is based on the main game state known as `GameStarted`, which is shown in the following screenshot. In order to make the transition from the `MainMenu` game state to the `GameStarted` game state (by pressing the **2 Players** button), we need to disable the `Menu Main` and the `Menu Background` game objects and enable the `Game Table` game object. Also, there is a second way to make the transition from another game state known as the `Difficulty` game state to the `GameStarted` game state by pressing one of the following buttons:

- **Easy**
- **Medium**
- **Hard**
- **Insane**

Thus, we have to disable the `Difficulty` and `Menu Background` game objects and enable the `Game Table` game object. Also, you should remember that the `Game Table` game object will only be active on the following game states and not on others:

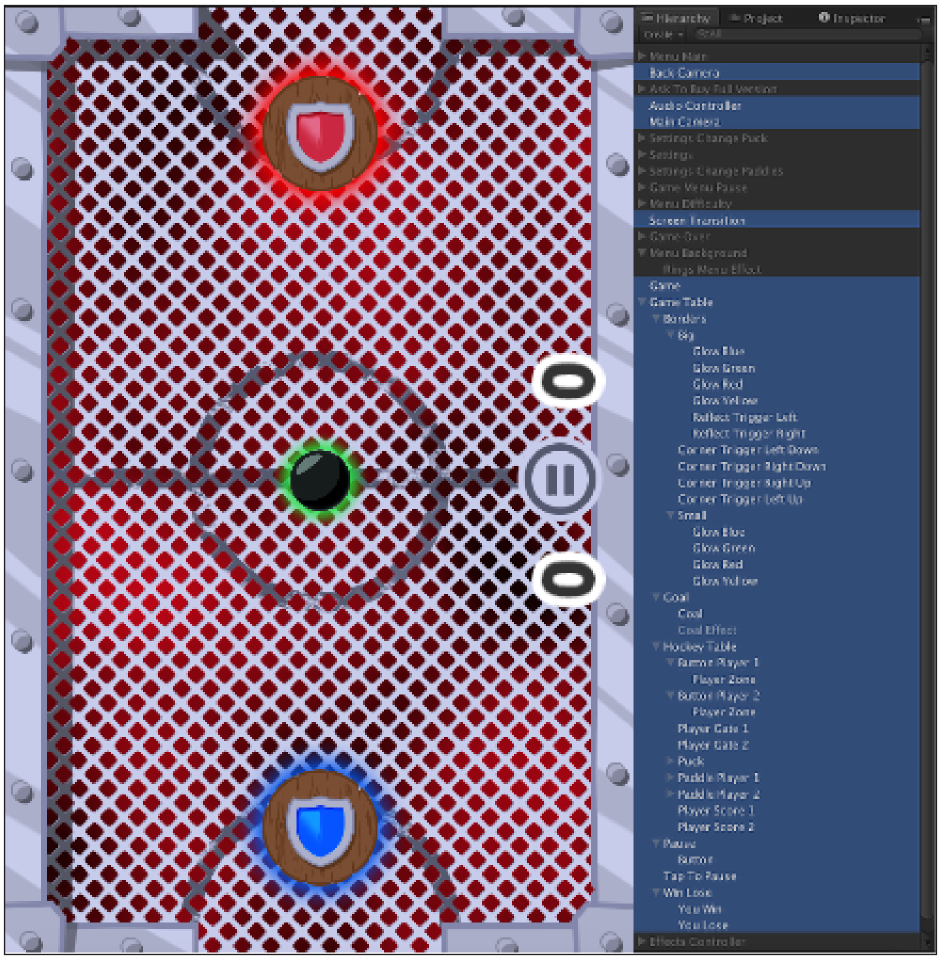
- `GameStarted`
- `GamePaused`
- `GameFinished`

The `Game Menu Pause` game object and the `Game Over` game object will be active only on the states `GamePaused` and `GameFinished`, respectively. The `Game Table` game object's hierarchy is listed here:

- The borders (2D physics for `Hockey Table` in conjuction with sprite images)
- The goal (a sprite image in conjuction with animation)
- The hockey table (two paddles, one puck, two scores, two gates, and two player buttons) game object's hierarchy is listed here:
    - The Player 1 button: This is the first player button with its limits defined by the child game object's (Player Zone) world scale values:
      ```
      limit width = PlayerZone.transform.lossyScale.x
      limit height = PlayerZone.transform.lossyScale.y
      ```
    - The Player 2 button: This is the second player button with its limits defined by the child game object's (Player Zone) world scale values:
      ```
      limit width = PlayerZone.transform.lossyScale.x
      limit height = PlayerZone.transform.lossyScale.y
      ```
    - Player Gate 1: This is the first player gate (hidden physics)
    - Player Gate 2: This is the second player gate (hidden physics)

- Paddle Player 1: This is the first player paddle (with four glow overlay images for highlighting)
- Paddle Player 2: This is the second player paddle (with four glow overlay images for highlighting)
- The puck (a sprite image and 2D physics with four glow overlay images for highlighting)
- Player Score 1 (a sprite image)
- **Player Score 2** (sprite image)

- The **Pause** button changes the game state from `GameStarted` to `GamePaused`. Notice that this button has to be touchable only on the `GameStarted` state.
- Tap To Pause (a sprite image with animation)
- Win/Lose (a sprite images with animation)

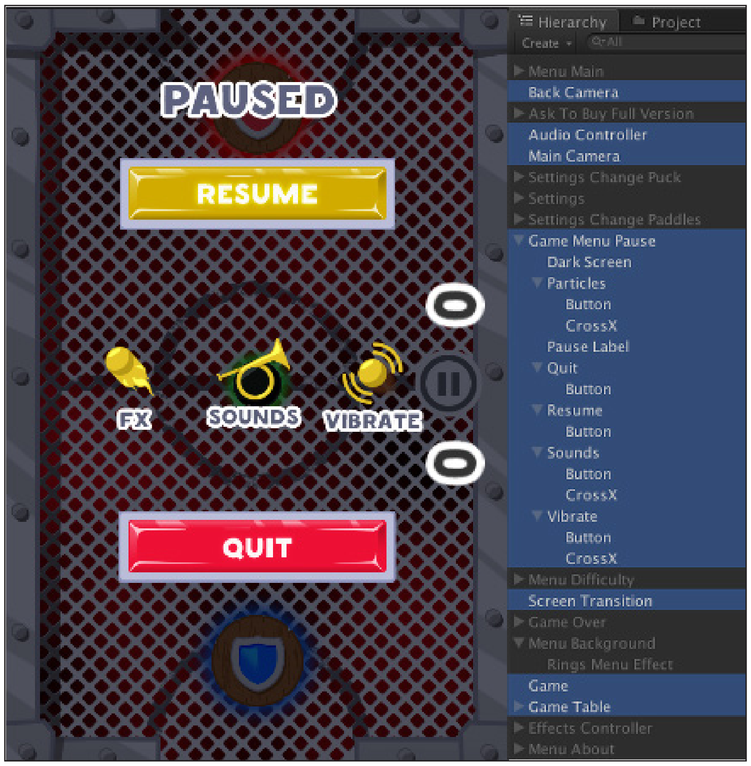


The GamePaused screen

The **GamePaused** screen is based on the same game state and it is shown in the following screenshot. In order to make the transition from the `GameStarted` game state to the `GamePaused` game state, by pressing the **Pause** button, we have to enable the `Game Menu Pause` game object over the `Game Table` game object, without any deactivation for the `Game Table` game object. We did the same thing many times earlier in this chapter. The `Game Menu Pause` game object's hierarchy is listed here:

- A dark screen (a black transparent image for a darkness effect while game is paused)
- The **Pause** label (a sprite image for title)
- The **Resume** button changes the game state from `GamePaused` to `GameStarted` by disabling the `Game Menu Pause` game object.
- The **Particles** button (with a cross image to enable and disable this setting) disables and enables particle effects.
- The **Sounds** button (with a cross image to enable and disable this setting) disables and enables sounds.
- The **Vibrate** button (with a cross image to enable and disable this setting) disables and enables vibration.

- The **Quit** button changes the game state from `GamePaused` to `MainMenu` by disabling the `Game Table` and `Game Menu Pause` game objects and by enabling the **Menu Main** and **Menu Background** game objects.

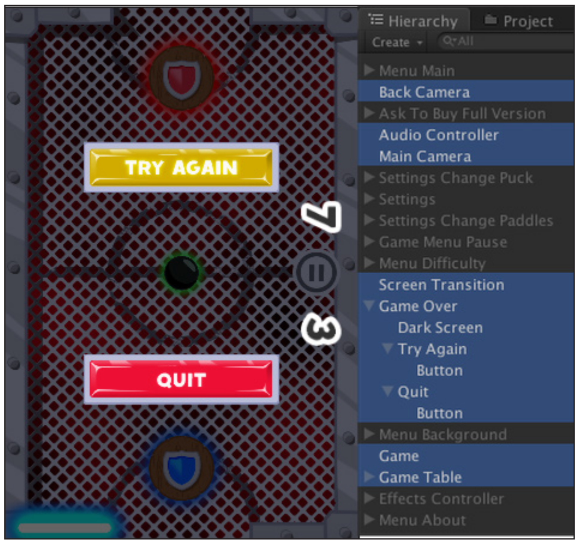


The GameFinished screen

The **GameFinished** screen is based on the same game state and it is shown in the following screenshot. In order to make the transition from the `GameStarted` game state to the `GameFinished` game state, by achieving the upper score limit by one of two available player scores, we have to enable the `Game Over` game object over the `Game Table` game object, without any deactivation of the `Game Table` game object just as in the `GamePaused` state. The `Game Over` game object's hierarchy is listed here:

- A dark screen (a black transparent image for darkness effect while the game is over)
- The **Try Again** button changes the game state from `GameFinished` to `GameStarted` by initializing the `Game Table` game object as you will see in our C# code-bundle available to download from the Packt Publishing website (here is a little bit different action than it was in the `GamePaused` state).

- The **Quit** button changes the game state from `GameFinished` to `MainMenu` by disabling the `Game Table` and `Game Over` game objects and by enabling the `Menu Main` and `Menu Background` game objects as in the `GamePaused` state.



The project settings

The first step is to set up our project settings. **Quality Settings** and **Player Settings** are the most important and useful settings for any project are. For the current *Glow Hockey* game clone example, it will be enough to have the fastest preset values for Android and all other platforms too. Choose the right values for your game by playing and testing a little bit around them. We do not need to touch all other project settings (except **Quality Settings** and **Player Settings**), and they can have all have default values. Also, most options have default values in **Quality Settings** and in **Player Settings** too.

An overview of the folder hierarchy

It is very useful first to prepare global project settings and, after that, to prepare, folder hierarchy for future images, fonts, sounds, scripts, prefabs, materials, physic materials, and all other types of assets. For huge projects, it is very useful to separate different textures from different modules in their own directories. Therefore, it is a very good idea for huge projects to imagine different modules as completely standalone projects with their own folder hierarchy, sounds, textures, scripts, and many other assets.

The famous five

In this section, you will discover in great details about five game objects that are always active in our game example, as we mentioned them all in the previous section. Let's list these famous five game objects that must be always active in our game:

- `Back Camera`
- `Audio Controller`
- `Main Camera`
- `Game`
- `Screen Transition`

All other game objects are enabled or disabled based on the current game state.

The Back Camera object

Now, it is time to look into Unity Inspector, starting with the `Back Camera` game object components and its parameters. In this game object, only the `Camera` component attached. Therefore, this camera just renders nothing except black color, instead of any other background image, for example, or instead of maybe some cool and fantastic effects in conjunction with 3D models and animations, just as you wish and as you imagine it.

The Audio Controller object

Let's look closely at Unity Inspector, starting with the `Audio Controller` game object's components and its parameters. This game object is attached to the `AudioController` and `AudioSource` components. The `AudioController` component is our custom script, which has 11 different sounds attached to its list. The `AudioSource` component is built-in the `Unity` component. The `AudioController` component is a singleton class, which is implemented by a static instance variable as described here:

```
// First of all we need to define using UnityEngine namespace,
// which is very similar to import packages in Java language.
using UnityEngine;

// Define your namespace name, as an example GlowHockey is good
enough.
namespace GlowHockey {

// Tell Unity that this script is based on AudioSource component.
// So Unity will care instead of you about checking and attaching
// this component if required. Thus you can be sure that static
// AudioSource component anytime is not equal to null value,
// with the least efforts provided by you.
[RequireComponent(typeof(AudioSource))]

// Let's define AudioController class which inherits from
// MonoBehaviour base component. MonoBehaviour class allows us to
// attach our custom script on any game object in any scene,
// as all other built-in components. Also MonoBehaviour provides
// many different and very useful callbacks,
// or in other words functions,
// which Unity calls at specified time points.
// We need just a few basic callbacks from MonoBehaviour class.
public class AudioController : MonoBehaviour {

// Let's define our public list of sounds we need to use.
// In our game example we need just
// eleven sounds as is shown in figure below.
  public AudioClip[] audioClips;

// We need this private temporary sound variable,
// which will be used a bit later.
  private AudioClip _audioClip;

// Private static instance variable to implement singleton class.
  private static AudioController _instance;

// Public static instance property in order to get
// this variable from other scripts as AudioController.Instance
// call. It is a very good style to make public get/set methods
// for private variables. So you can control limits or any other
// required conditions for your variables.
// In our example we have to control nothing.
// All we need do is just to return private instance.
  public static AudioController Instance {
    get {
      return _instance;
    }
  }

// Unity calls this callback/function only once when the script
// instance is being loaded, before any other callbacks
// on the same script.
  void Awake() {

// First of all we should check if our game scene
// has already AudioController instance. If it has,
// then we should destroy this game object immediately
// and also immediately return from this function, because
// we must implement singleton pattern for this class.
    if (null != _instance) {
      DestroyImmediate(gameObject);
      return;
    }

// Otherwise if current game scene has not any instance of
// AudioController class then we should save this object
// in our private static instance variable,
// in order to access it further from different scripts.
    _instance = this;
  }

// Following function FindAudioClipByName finds sound by name,
// and returns that sound from our public list,
// which was defined a bit earlier. If function cannot find
// desired sound by name then it simply returns null value.
  AudioClip FindAudioClipByName(string name) {
    if (null == audioClips) return null;

    foreach (AudioClip ac in audioClips) {
      if (ac.name.Equals(name)) {
        return ac;
      }
    }

    return null;
  }

// Next public function PlaySound plays sound by name.
  public void PlaySound(string name) {

// First of all we must to check if sounds are disabled,
// then we should return immediately from this function.
// Game class also implements singleton
// pattern by static instance property.
// Game class will be described a bit later.
// It is also located in this Famous Five section,
// as was mentioned earlier.
    if (false == Game.Instance.areSoundsEnabled) return;

// Now we can use our FindAudioClipByName function,
// which was described above, in order to find sound by name.
    _audioClip = FindAudioClipByName(name);

// If we cannot find sound by name in our public list,
// then our private variable will be equal to null value,
// and we will have to return from this function immediately.
    if (null == _audioClip) return;

// Static audio component variable was removed in version 5.0.0
// Property audio has been deprecated.
// So you should use GetComponent<AudioSource>() instead.
// We know that there must be attached AudioSource,
// it cannot be equal to null value, thus we can
// play sound that was found by given name as a parameter.
    GetComponent<AudioSource>().PlayOneShot(_audioClip);
  }
}
```

The `AudioController` class finds a sound by name, and returns that sound from our public list in order to play it. If the function cannot find the desired sound by name, then it simply returns a `null` value which means that it cannot play the sound.

After the `AudioController` game object is ready and all its component values have the correct values, we can play any desired sound by name in any of our scripts, as shown here:

```
AudioController.Instance.PlaySound("Your Sound Name");
```

As you can see, it is very fast and useful to write one line of code, in order to call a function, which plays a sound by your given name. As a very convinient example here, you should know that, for all our many different buttons and other game objects, we play any desired sound at the required time with no effort at all and just a single line of code.

The Main Camera object

Let's look closely at the `Main Camera` game object components and its parameters.

The following components are attached to this game object:

- `Camera`: This component is a built-in Unity component
- `AudioListener`: This component is a built-in Unity component
- `CameraGrid`: This is our custom component (this class inherits from `MonoBehaviour`)
- `CameraRatio`: This is our custom component (this class inherits from `MonoBehaviour`)

You can see our values for all components by downloading our C# code-bundle available from the Packt Publishing website, therefore, it should be easy enough to recreate the whole `Main Camera` game object needed our game example. Do not forget to set the correct tag (`MainCamera`), so we can use this camera instance using the public static variable from the `Camera` class (`Camera.main`) in our different custom scripts.

Now, let's discover the next component that helps us fit our game for any screen size and resolution correctly by keeping your target aspect ratio, so your images will not be broken. The aspect ratio will always be the same as your desired target ratio. You may use this script for almost all your projects, especially for all your menu interfaces:

```
// As always we need to define using UnityEngine namespace,
// which is very similar to import packages in Java language.
using UnityEngine;

namespace CameraTools {
//[ExecuteInEditMode]
  public class CameraRatio : MonoBehaviour {

// Public float values by which we can control our target
// aspect ratio and how camera should render its graphics.
  public float heightKoefficient = 0.5f, widthKoefficient = 0.5f,
WIDTH = 640.0f, HEIGHT = 960.0f;

// Attached camera on this public variable.
// If there is attached no camera, then this value
// will be equal to null value, and in this case
// we have to use Camera.main static reference.
```

```
  public Camera cam;

// We can tell Unity to hide this public variables.
// These four public float variables will be adjusted
// automatically right in the following code.
// But it is very useful for debug purposes to see its
// current values right in Unity Inspector window.
//[HideInInspector]
  public float targetAspect, windowAspect, scaleHeight, scaleWidth;

// Private rectangle variable we will use as a temporarily.
    private Rect _rect;

// Let's initialize our public camera variable. It was
// specially created to save reference for required camera.
// Unity calls this function just only once before
// Update function and after Awake callback.
// Unity calls Start function on the frame
// when a script is enabled.

// Awake and Start methods are called exactly
// once during their lifetime. Awake function is called only
// in event of script instance is already initialized.
// Awake is triggered when the script is enabled and disabled,
// in both situations. However, Start method is called only when
// the script is enabled, and of course when the script
// is initialized, cause initialization comes by first step.

// First of all, as was mentioned a bit earlier, will be called
// Awake method on all objects in our scene and only after that
// will be called Start method on all those objects.
// Where objects are instantiated during gameplay,
// their Awake function will naturally be called after
// the Start functions of scene objects have already completed.
  void Start() {
    if (null == cam) {
      cam = gameObject.GetComponent<Camera>();
      if (null == cam) cam = Camera.main;
    }
  }

// Unity calls Update function every frame,
// in event of the MonoBehaviour component is enabled.
// This function Unity calls as fast as possible.
// So on slow devices Unity calls this function less times, than
// Unity calls this function on more powerfull devices.
// FixedUpdate Unity's callback is called always the same amount
// of times on any device, without any dependence from slow or
// powerfull devices. FixedUpdate function is very useful if
// you are required to move for example your game object
// with the constant speed at any time and on any devices,
// so the speed will not depend on device's performance.
  void Update()
  {
// Let's determine our target aspect ratio, which
// is controlled by two public float values: WIDTH & HEIGHT.
    targetAspect = WIDTH / HEIGHT;

// Determine the game window's current aspect ratio.
    windowAspect = (float)Screen.width / (float)Screen.height;

// Current viewport height should be scaled by this amount.
    scaleHeight = windowAspect / targetAspect;

// If scaled height is less than current height, add letterbox
    if (scaleHeight < 1.0f)
    {
      _rect = cam.rect;

// Width is less than Height, that's why
// Width can be visible 100%, but Height is
// only visible partly, because it is bigger than
// our Width, and we can see only scaleHeight * 100%
// in percents.
      _rect.width = 1.0f;
          _rect.height = scaleHeight;

// Let's define how should be adjusted our camera's
// viewport. It can be positioned right in the center, if
// heightKoefficient will be between 0 and 1, so if it
// will be equal to 0.5 float value. 0 value means
// that camera should stay on the bottom side, and 1 value
// means that camera should stay on the top side.
// You should just to play a little bit around these values
// in real-time and this will help you to understand how
// things are going behind it and how do they must work.
      _rect.x = 0;
      _rect.y = (1.0f - scaleHeight) * heightKoefficient;

           cam.rect = _rect;
        }
        else // add pillarbox
        {
// If scaleHeight is greater or equal to 1.0 float value,
// then scaleWidth float value is inversely proportional
// to scaleHeight float value, then it must be less
// than 1.0 float value.
      scaleWidth = 1.0f / scaleHeight;

          _rect = cam.rect;

// Height is less than Width, that's why
// Height can be visible 100%, but Width is
// only visible partly, because it is bigger than
// our Height, and we can see only scaleWidth * 100%
// in percents.
      _rect.width = scaleWidth;
          _rect.height = 1.0f;

// Let's define how should be adjusted our camera's
// viewport. It can be positioned right in the center, if
// widthKoefficient will be between 0 and 1, so if it
// will be equal to 0.5 float value. 0 value means
// that camera should stay on the left side, and 1 value
// means that camera should stay on the right side.
// You should just to play a little bit around these values
// in real-time and this will help you to understand how
// things are going behind it and how do they must work.
        _rect.x = (1.0f - scaleWidth) *
widthKoefficient;
              _rect.y = 0;

        cam.rect = _rect;
      }
    }
  }
}
```

The Core Game objects

As mentioned earlier, the `Core Game` objects are listed here. If you have any ideas on how to improve this game or how to make any other type of game from this project, then you can easily adjust and edit all the required states, as well as all the other properties:

```
public enum State {
  AskToBuyFullVersion,
  MainMenu,
  About,
  Settings,
  SettingsChangePaddles,
  SettingsChangePuck,
  Difficulty,
  GameStarted,
  GamePaused,
  GameFinished
}
```

In order to determine when our game's state was changed, we need to have one private variable known as `_currGameState`. Also, we need another private variable known as `_prevGameState` in order to save the previous game state, which will be used later in this script to define if it is required to initialize the `Game Table` instance.

As mentioned earlier, there are also different game modes:

```
public enum Mode {
  Easy, Medium, Hard, Insane, TwoPlayers
}
```

Also, we have just four different types of colour for the two paddles and puck:

```
public enum PaddlePuckType {
  Yellow, Red, Green, Blue
}
```

In our game, we have three different settings that can be enabled or disabled in the `Settings` or `GamePaused` game states:

```
public enum Settings {
  Particles, Sounds, Vibrate
}
```

We made a `Game` singleton implementation via a public static instance getter property, as you will see in our C# codebundle available to download from Packt Publishing website. We need a sprite renderer's reference to use animation with its alpha channel in co-routines, a float value to control its speed, and just a temporary private color variable. This sprite renderer's reference must be attached to the `Screen Transition` game object from our game scene. There are different possible solutions and the simplest way is just to drag and drop the required game object to the right property slot in the Unity editor.

The Screen Transition object

We use screen transition each time the game state is changed. As you can see in the C# code bundle available to download from the Packt Publishing website, this screen transition is needed to smoothly show the menu background.

The buttons implementation

The first two scripts are core for all buttons. You can use them in any project. Also, it is not a problem to extend next scripts because they are pretty simple and very useful at the same time. We hope that the following scripts, like any other code examples from this book, will be very helpful for your projects.

The InputController.cs script

The following script/component is as simple as the others, as the `InputController` component calls the button's functions, and after that, the `Button` component calls the `ButtonAction` function. Only when it is required, you will see our C# codebundle available to download from Packt Publishing website.

The Button.cs script

The following script/component is as simple as the others, as the `InputController` component calls the button's functions, and after that, the `Button` component calls the `ButtonAction` function when it is required (as we discovered one step ago).

The EnableInputOnState.cs script

Next, we need a public game state variable for this component in order to adjust it with the correct value in Unity Inspector without extending the code for many different buttons. The private variable is a reference to the instance of your `InputController` component attached to the same game object, where this component is attached too. Anytime, Unity will care for us, that this instance of `InputController` will not be equal to a `null` value. We should use this script just for the players' buttons and the **Pause** button, as you can see in the C# code bundle available to download from Packt Publishing website.

The ButtonBuyFullVersion.cs script

The following script/component is as simple as many others, as the `InputController` component calls the button's functions, and after that the `Button` component calls the `ButtonAction` function, only when it is required. We can adjust different URLs for different buttons in Unity Inspector without having to write new code or to extend current code. The `Button` component calls the next function shown in the following code snippet via the `SendMessage` method:

```
void ButtonAction() {
  // The main idea of this button is just to open url.
  Application.OpenURL(openUrl);
}
```

We should use this script just for one button, as you will see in the current Unity C# project available for downloading from Packt Publishing website.

The ButtonChangeGameState.cs script

We can adjust different states for different buttons in Unity Inspector without having to write the new code or to extend current code by defining the next public variable with different values for different buttons as we made for many of our buttons:

```
public Game.State changeGameState = Game.State.MainMenu;
```

The `Button` component calls this function via the `SendMessage` method:

```
void ButtonAction() {
  // The main idea behind this button is just to change current game
state.
  Game.Instance.gameState = changeGameState;
}
```

We should use this script for 21 different buttons as listed here:

- The menu main screen shows the following buttons:
    - The **1 Player** button
    - The **2 Players** button
    - The **Settings** button
    - The **About** button
- The `Ask to buy full version` screen:
    - The **Continue** button
- The **Settings** screen:
    - The **Change paddles** button
    - The **Change puck** button
    - The **Back** button
- The `settings change paddles` screen:
    - The **Back** button
- The `settings change puck` screen:
    - The **Back** button
- The `game menu pause` screen:
    - The **Resume** button
    - The **Quit** button
- The `Menu difficulty` screen:
    - The **Easy** button
    - The **Medium** button
    - The **Hard** button
    - The **Insane** button
    - The **Back** button
- The `Game over` screen:
    - The **Try again** button
    - The **Quit** button

- The `game` screen
    - The **Pause** button
- The `menu about` screen
    - The **Back** button

The ButtonChangeGameMode.cs script

We can adjust different modes for different buttons in Unity Inspector without having to write the new code or to extend current code:

```
public Game.Mode changeGameMode = Game.Mode.Easy;
```

The `Button` component calls this function via the `SendMessage` method:

```
void ButtonAction() {
  // The main idea of this button is just to change current game mode.
  Game.Instance.gameMode = changeGameMode;
}
```

We have just five buttons, where this script is attached. Let's list all them:

- The `menu main` screen:
    - The **2 Players** button
- The `menu difficulty` screen:
    - The **Easy** button
    - The **Medium** button
    - The **Hard** button
    - The **Insane** button

The ButtonChangePaddle.cs script

The first public `transform` variable in this class must be attached to the Unity Inspector by `transform`, which will be positioned over the chosen paddle to show the current selection in **Settings**. The second variable will differentiate four paddle types; thus, we can attach this script to four objects at the same time and adjust different paddle types for them:

```
public Game.PaddlePuckType paddleType;
```

The next variable will differentiate two players; thus, we can attach this script to two objects at the same time and adjust different players for them. Therefore, we can choose two different paddles for two players in the **Settings** menu:

```
public Game.Player player;
```

The next variable's value controls how fast the selection transform has to move:

```
public float speed = 11.17f;
```

The following functions describe this component in more detail:

```
  void Start() {
    // Here we should check if attached player has this paddleType
    // then we have to be sure that the selection transform is
    // over this game object position, so players can see
    // which type of paddle is selected for first or second player.
    if (Game.Instance.GetPaddleType(player) == paddleType) {
      selection.position = transform.position;
    }
  }

  // Button component calls this function via SendMessage method.
  void ButtonAction() {
    // In event of pressed button we should save paddleType
    // for our player.
    Game.Instance.SetPaddleType(player, paddleType);
    // Next code line is commented to make smooth transition later.
    //selection.position = transform.position;
  }

  void FixedUpdate() {
    // Selection transform smoothly moves here to its target place.
    if (Game.Instance.GetPaddleType(player) == paddleType) {
      selection.position = Vector3.Lerp(selection.position,
transform.position, Time.fixedDeltaTime * speed); // with acceleration
    }
  }
```

There are eight buttons where this script is attached.

The ButtonChangePuck.cs script

The next public `transform` variable must be attached to the Unity Inspector by the `transform` component, which will be positioned over the chosen puck to show the current selection in **Settings**:

```
public Transform selection;
```

The following variable will differentiate the four types of puck; thus, we can attach this script to four objects at the same time and adjust different types of puck for them:

```
public Game.PaddlePuckType puckType;
```

The next variable's value controls how fast a selection transform has to move:

```
public float speed = 11.17f;
```

The following functions describe this component in more detail:

```
    void Start() {
      // Here we should check if puck has the same puckType
      // then we have to be sure that the selection transform is
      // over this game object position, so players can see
      // which type of puck is selected now.
      if (Game.Instance.puckType == puckType) {
        selection.position = transform.position;
      }
    }

    // Button component calls this function via SendMessage method.
    void ButtonAction() {

      // In event of pressed button we should save puckType.
      Game.Instance.SetPuckType(puckType);

      // Next code line is commented to make smooth transition later.
      //selection.position = transform.position;
    }

    void FixedUpdate() {
      // Selection transform smoothly moves here to its target place.
      if (Game.Instance.puckType == puckType) {
        selection.position = Vector3.Lerp(selection.position,
transform.position, Time.fixedDeltaTime * speed); // with acceleration
      }
    }
```

There are four buttons where this script is attached.

The ButtonChangeSettings.cs script

As we mentioned earlier, we have just three different settings: sounds, effects, and vibration. Thus, we can use this script for three different buttons with three different settings:

```
public Game.Settings settingType;
```

We have to initialize the following public variable in the Unity Inspector. We should disable this renderer when this setting is `enabled` and should enable this renderer when this setting is `disabled`:

```
public SpriteRenderer crossXrenderer;
```

The `private` flag is used to keep this value for the setting:

```
private bool _isEnabled;
```

The following functions describe this component in more detail:

```
  void OnEnable() {
    // Firstly we should get the latest
    // saved value for this setting and initialize our
    // private variable with returned value.
    _isEnabled = Game.Instance.GetSetting(settingType);
    // We should disable this renderer in event of this setting is
enabled.
    // We should enable this renderer in event of this setting is
disabled.
    crossXrenderer.enabled = !_isEnabled;
  }

  // Button component calls this function via SendMessage method.
  void ButtonAction() {
    // In event of pressed button we should revert
    // current setting value to the opposite Boolean value.
    _isEnabled = !_isEnabled;
    // Next step is to save current setting.
    Game.Instance.SetSetting(settingType, _isEnabled);
    // Following step is to visualize if current setting is enabled or
disabled.
    crossXrenderer.enabled = !_isEnabled;
  }
```

There are six buttons where this script is attached.

The ButtonMoreApps.cs script

The `Button` component calls this function via the `SendMessage` method:

```
void ButtonAction() {
  // The main idea of this button is just to open market url.
  Application.OpenURL("market://search?q=" + marketSearch);
}
```

There is just one single button where this script is attached.

The gameplay and game world implementation

Let's see in practice how easy it is to use physics for our game world. Also, you will see how easy it is in Unity 5 to design beautiful effects, animations, behaviors, and other very useful and effective features and techniques for our game world.

The GameTable.cs script

This value controls how fast the computer AI is:

```
public static float gameSpeedAI;
```

The following variables control AI speed for different game modes:

```
public float
  gameSpeedEasyAI = 5.0f,
  gameSpeedMediumAI = 7.0f,
  gameSpeedHardAI = 11.0f,
  gameSpeedInsaneAI = 17.0f;
```

Also, we need different animators to play them in the game at specified times. The `tap to pause` animation appears each time before a game starts. The `goal` animation appears after each goal event. The `you win` and `you lose` animations appear at the end of each game to declare the winner and the loser accordingly:

```
  public Animator tapToPauseAnimator, goalAnimator, youWinAnimator,
youLoseAnimator;
```

We have just two player buttons, which we need to initialize each time before the game starts:

```
private ButtonPlayer[] _playerButtons;
```

We need to be sure that the border's glowing effect is turned off by disabling the border's `SpriteRenderer` component:

```
private Border[] _borders;
```

The singleton implementation is as follows:

```
private static GameTable _instance;
public static GameTable Instance {
  get {
    return _instance;
  }
}
```

The next method is our `public` custom function. This method is called each time before a game starts. We call this function only from the `Game.cs` script and just from one single place in the `GameStartedState` function. If the previous game state was the `GamePaused` state, then we should not initialize our `Game Table` instance, since we want to continue playing after the `GamePaused` state. In all other circumstances (except the `GamePaused` state), we must initialize our `Game Table` instance, as shown here:

```
public void Init() {
  // Let's play sound here if sounds setting is enabled.
  AudioController.Instance.PlaySound("game_start");

  // Because this function is called each time before game starts
  // then here we need to reset to 0 values both player scores.
  Game.playerOneScore = Game.playerTwoScore = 0;

  // Also each time before game starts we need to
  // correctly place two paddles and one puck, the same is true
  // each time after goal event happens. Also we need to
  // initialize physics for two paddles and one puck.
  InitPaddlesAndPuck(true);

  // Let's disable glowing effect for all our borders.
  foreach (Border b in _borders) {
    b.SetSpriteRenderer(false);
  }

  // Depending on the current game mode we should
  // adjust right computer AI speed value.
  switch (Game.Instance.gameMode) {
    case Game.Mode.Easy:
      gameSpeedAI = gameSpeedEasyAI;
      break;
```

```
        case Game.Mode.Medium:
          gameSpeedAI = gameSpeedMediumAI;
          break;
        case Game.Mode.Hard:
          gameSpeedAI = gameSpeedHardAI;
          break;
        case Game.Mode.Insane:
          gameSpeedAI = gameSpeedInsaneAI;
          break;
        case Game.Mode.TwoPlayers:
          gameSpeedAI = 0.0f;
          break;
      }
      // Before starting tap to pause animation
      // we have to be sure it is not playing now.
      StopCoroutine("TapToPauseAnimation");
      // Here we can start our tap to pause animation.
      StartCoroutine(TapToPauseAnimation());
      // It's time to find one of two player gates and to
      // call its co-routine animation in event of placing puck
      // each time before game starts, or also each time after
      // goal happens as we will see just a bit later.
      PlayerGate _gate = GameObject.FindObjectOfType(typeof(PlayerGate))
    as PlayerGate;
      _gate.StartPlacingPuck();
    }
```

Each time before a game starts, we need to correctly place two paddles and one puck, the same is true each time after every `goal` event happens. Also, we need to initialize physics for two paddles and one puck. The following function does all the required initialization:

```
    public void InitPaddlesAndPuck(bool initButton) {
      // initButton equals to true each time before game starts.
      // initButton equals to false each time after goal occurs.
      Puck.Instance.Init();
      foreach (ButtonPlayer bp in _playerButtons) {
        // ButtonPlayer initialization happens here each time before game
    starts.
        if (initButton) bp.Init();
        else bp.InitPaddle();
      }
    }
```

Our co-routine `tap to pause` animation is where we only need to notify its animator to change the animator's state by adjusting its Boolean `CanPlayAnim` variable to a `true` value within a short period of time:

```
IEnumerator TapToPauseAnimation() {
  tapToPauseAnimator.SetBool("CanPlayAnim", true);
  yield return new WaitForSeconds(0.1f);
  tapToPauseAnimator.SetBool("CanPlayAnim", false);
}
```

Before playing our `goal` animation, we should rotate its animator's parent object to make it readable for the player who made this goal:

```
public void StartGoalAnimation(float degrees) {
  goalAnimator.transform.parent.rotation = Quaternion.Euler(new
Vector3(0, 0, degrees));
  StopCoroutine("GoalAnimation");
  StartCoroutine(GoalAnimation());
}
```

Our co-routine `goal` animation is where we need to notify its animator to change the animator's state by adjusting its Boolean `IsGoal` variable to a value of `true` within a short period of time:

```
IEnumerator GoalAnimation() {
  goalAnimator.SetBool("IsGoal", true);
  yield return new WaitForSeconds(0.1f);
  goalAnimator.SetBool("IsGoal", false);
}
```

There is just one single game object where this script is attached.

The ButtonPlayer.cs script

This 2D rigid body variable is initialized in the Unity Inspector. We need to use it to control the player's paddle 2D physics:

```
public Rigidbody2D rigidBodyPaddle;
```

The following variable controls how fast the paddle moves to a touch:

```
public float speedMoveTouch = 255.0f;
```

We need this private variable for our input touches:

```
private InputController.InputTouch _inputTouch;
```

The initial position is saved the first time only, after that, we will use this initial position in order to correctly place the paddle at the start of each game or after each goal. Also, we need temporary vectors `_v3` and `_v2`:

```
private Vector3 _initialPosition, _v3;
private Vector2 _v2;
```

Next, the `index` variable is for input: the touch finger ID or mouse button ID. This integer value keeps the current touch ID:

```
private int _index;
```

The paddle should be controlled by this script:

```
private Paddle _paddle;
```

Next, the `_isComputerAI` Boolean tells us if the computer AI should play instead of the second player or not:

```
private bool _isComputerAI;
```

This is the child game object as we saw earlier. We need to use this game object in order to have limits for two paddle movements, so they will not be placed outside our hockey table:

```
private Transform _playerZone;
```

Next, the following two variables control the width of our player zone which was discussed earlier in text:

```
private float _widthLeft, _widthRight;
```

There are two player buttons where this script is attached.

The Paddle.cs script

The `forceDelta` variable is used to control applied force to puck:

```
public static float forceDelta = 0.1f;
```

The following force value is applied to the puck:

```
public float force = 500.0f;
```

The next variable keeps the initial force:

```
public float forceInit;
```

We have to play sound on any paddle's collision with puck or with the border too, as shown here:

```
public string soundNameOnCollision = "paddle_hit";
```

This determines how long the glowing effect duration stays for the concrete paddle object where this script is attached:

```
public float secondsToWaitGlow = 0.17f;
```

Glowing childs for our paddle object:

```
public List<SpriteRenderer> glowOverlays;
```

The current sprite renderer where this script is attached:

```
private SpriteRenderer _spriteRenderer;
```

There are two game objects where this script is attached.

The PaddleTypeView.cs script

The following four sprites are used for the paddle's representation in menu settings and in gameplay too:

```
public Sprite yellowPaddle, redPaddle, greenPaddle, bluePaddle;
```

The attached player determines whether this is the first or second paddle:

```
public Game.Player player;
```

The current paddle's type is as follows:

```
private Game.PaddlePuckType _paddleType;
```

The next variable keeps reference for our sprite renderer component, which is attached to this game object; thus, we can change the paddle's view if its type was changed in menu settings:

```
private SpriteRenderer _spriteRenderer;
```

There are 10 game objects where this script is attached. Both are the game objects that we discovered earlier: `Paddle Player 1` and `Paddle Player 2` from `Game screen` (the `Game Table` object). All glow overlay renderers must be disabled at the start.

The Puck.cs script

We have to limit puck's value for the maximum velocity and also its position in order not to go further than it is allowed. The puck's `Y` coordinate in the middle zone (while its `X` coordinate is between the values of `leftCenterPosX` and `rightCenterPosX` as shown here in `Update` callback) can go further than in the right or in the left zones because the puck has to go till the gates:

```
public float
  velocityMaxLimit = 45.0f,
  leftLimitPosX = -2.59f,
  rightLimitPosX = 2.59f,
  leftCenterPosX = -1.5f,
  rightCenterPosX = 1.5f,
  downLimitPosY = -4.22f,
  upLimitPosY = 4.22f;
```

We have to play sound on any puck collision with the paddle or with the border too, as shown here:

```
public string soundNameOnCollision = "puck_hit";
```

This determines how long the glowing effect duration lasts for the puck object where this script is attached:

```
public float secondsToWaitGlow = 0.17f;
```

Glowing childs for our puck object:

```
public List<SpriteRenderer> glowOverlays;
```

The current sprite renderer where this script is attached:

```
private SpriteRenderer _spriteRenderer;
```

The puck's current velocity is:

```
private Vector2 _velocity;
```

The puck's current angular velocity and angular rotation values are:

```
private float _angularVelocity, _angularDrag;
```

This is our flag to save the current state of puck's 2D rigid body:

```
private bool _isSleeping;
```

The singleton pattern implementation:

```
private static Puck _instance;
public static Puck Instance {
  get {
    return _instance;
  }
}
```

There is just a single game object where this script is attached.

The PuckTypeView.cs script

The following four sprites are used for puck's representation in menu settings and in the gameplay too:

```
public Sprite yellowPuck, redPuck, greenPuck, bluePuck;
```

The following variable keeps reference for our `spriteRenderer` component, which is attached to this game object; thus, we can change the puck's view if its type was changed in the menu settings:

```
private SpriteRenderer _spriteRenderer;
```

There are six game objects where this script is attached. The puck object was shown a little earlier.

The PlayerGate.cs script

We need to use the next public variable just once in the `ButtonPlayer` script for its `TouchMove` function. The variable indicates whether the `placing puck` animation is currently playing. The `placing puck` animation should be played after each `goal` or each time before the game starts:

```
public static bool isPlacingPuckNow = false;
```

The attached player determines whether this is the first or second player gate:

```
public Game.Player player;
```

The game continues till this maximum limit of goals is reached:

```
public int maxGoals = 7;
```

This variable is used to control the initial paddle `Y` position coordinate after each `goal` and not if the `game starts` state is activated:

```
public float puckDeltaDistanceAfterGoal = 3.5f;
```

A temporary variable should keep puck's size:

```
private Vector3 _initPuckScale;
```

There are two game objects where this script is attached.

The PlayerScoreView.cs script

The next class' member player determines whether this is the first or second player's score:

```
public Game.Player player;
```

An array of sprites for digits from `0` to `7` numbers:

```
public Sprite[] spriteDigits;
```

We need the following two variables to control flash animation duration and interval for this player's score view:

```
public float totalSecondsFlash = 2.3f, deltaSecondsFlash = 0.3f;
```

The next two variables are temporary, which are used to manage flash animation for this player's score view:

```
private float _timeStartFlash, _timeDeltaFlash;
```

The following Boolean flag indicates whether the flash animation is playing or not for this player's score view:

```
private bool _isFlashing;
```

We use this variable further just to know when the current player's score was changed, it is a very basic example of logic optimization with the main idea to make any actions only when they are required:

```
private int _lastScore;
```

We need a variable to keep the current score view's sprite renderer component:

```
private SpriteRenderer _spriteRenderer;
```

There are two game objects where this script is attached.

The Border.cs script

This script uses trigger and collision callbacks in order to work for both options independently from how the trigger option was adjusted in the Unity Inspector. The next public variable determines the length of the glowing effect duration for this border where this script is attached:

```
public float secondsToWaitGlow = 0.17f;
```

We have to play sound on any border's collision with the puck or with paddles too, as shown here:

```
public string soundNameOnCollision = "border_hit";
```

The current border's sprite renderer component:

```
private SpriteRenderer _spriteRenderer;
```

There are eight game objects where this script is attached.

The BorderReflect.cs script

We need this script just to reflect puck from the hockey table borders. The `normal` variable is the first variable that we should adjust in the Unity Inspector:

```
public Vector2 normal;
```

The second variable keeps our calculation about reflected velocity:

```
private Vector2 _reflectedVelocity;
```

There are two game objects where this script is attached.

The CornerTrigger.cs script

We need this script just to know when puck is in the corner in order to avoid a bug while the computer AI can stop the puck in any corner. We just have four corners:

```
public enum Type {LeftDown, LeftUp, RightDown, RightUp};
public Type type;
```

We should have a Boolean flag for each corner:

```
public static bool
  isLeftDownTriggered,
  isLeftUpTriggered,
  isRightDownTriggered,
  isRightUpTriggered;
```

There are four game objects where this script is attached.

Effects and animations

It is not a problem to create very beautiful effects and animations in Unity 5. Therefore, here we will show you how to create some effects and animations for our Glow Hockey game. We will show you just a few effects and animations. It will be good homework for you to create your own effects and animations for this game. Also, we did not create vibration and effects to be played after each goal, thus we are sure that this homework will be very helpful for you to improve functionality. You can download this Unity 5 project (written in C#) from the Packt Publishing website and you will see which effects and animations are ready and which aren't yet. Also, you can see how to configure beautiful and very simple effects and animations right in Unity Editor.

Summary

In this chapter, you discovered how easy it is to develop a prototype of the game Glow Hockey in Unity 5 from scratch. The actual game Glow Hockey has about 100,000,000–500,000,000 downloads and can be downloaded from `https://play.google.com/store/apps/details?id=com.natenai.glowhockey&hl=en`. You saw how to create a camera for any screen resolution and any screen size. Also, you saw in practice how easy it is to use physics.