Unity 5 is a revolution in developing great games for Android that provides a great integration platform that works seamlessly with Unity 5, which means that games can be developed quicker and easier than ever before.

Packed with a lot of examples, this book starts by helping you to understand all the great features that Unity 5 and Android have to offer. You will then create great games like Tic-Tac-Toe and the Monkey Ball game and also learn to enhance them. You will then expand the game’s environment with lights and a skybox and learn to create enemies in a tank battle game. You will then explore the touch and tilt controls with the creation of a Monkey Ball clone.

With the recreation of a game similar to Angry Birds, you will delve into configuring physics and options for a 2D game experience. Finally, you will get a complete experience by learning the optimization techniques needed to keep your games running smoothly.

Who this book is written for
If you are an Android developer who wants to learn how to build games with Unity for the Android platform, then this book is ideal for you. Some prior knowledge of C# and JavaScript would be helpful.

What you will learn from this book
- Set up the development environment when working with both Unity and Android
- Import and work with the basic blocks of a game such as meshes, materials, and animations
- Utilize particles and sound effects to provide feedback to the player
- Adjust camera effects and game logic to create 2D games
- Interface with touch and tilt logic to create custom control systems
- Create custom graphical interfaces and control elements
- Leverage pathfinding to get intelligently moving characters
- Illuminate scenes dynamically using lightmaps

Learn to create stunning Android games using Unity
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 7 'Throwing Your Weight Around – Physics and a 2D Camera'
- A synopsis of the book’s content
- More information on Learning Unity Android Game Development

About the Author

Thomas Finnegan graduated from Brown College in 2010, and he now works as a freelance game developer. Since 2010, he has worked on everything from mobile platforms to web development, and he has even worked with experimental devices. His past clients include Carmichael Lynch, Coleco, and Subaru. His most recent project is Battle Box 3D, a virtual tabletop. Currently, he teaches game development at the Minneapolis Media Institute in Minnesota.
Learning Unity Android Game Development

In this book, we explore the ever-expanding world of mobile game development. Using Unity 3D and Android SDK, we learn how to create every aspect of a mobile game while leveraging the new features of Unity 5.0 and Android L. Every chapter explores a new piece of the development puzzle. By exploring the special features of development with mobile platforms, every game in the book is designed to increase your understanding of these features. We will finish the book with a total of four complete games and all of the tools that you need to create many more.

The first game that we will make is Tic-Tac-Toe. This game functions just like the classic paper version. Two players take turns filling a grid with their symbols and the first to make a line of the same three symbols wins. This is the perfect game for us to explore the graphical interface options that we have in Unity. By learning how to add buttons, text, and pictures to the screen, you will have all the understanding and the tools that are needed to add any interface that you might want to any game.

The next game that we will create is the Tank Battle game. In this game, the player takes control of a tank to drive around a small city and shoot targets and enemies. This game spans three chapters, allowing us to explore the many key points of creating games for the Android platform. We start it by creating a city and making the player's tank move around by using the controls about which we learned when we made the Tic-Tac-Toe game. We also create and animate the targets at which a player will shoot. In the second part of this game, we add some lighting and special camera effects. By the end of the chapter, the environment looks great. In the third part of the game's creation, we create some enemies. Using the power of Unity, these enemies chase the player around the city and attack them when they are close.

The third game is a simple clone of a popular mobile game. Using the power of Unity's physics system, we are able to create structures and throw birds at them. Knock down the structures to gain points and destroy the target pigs to win the level. We also explore some of the specific features of a 2D game and Unity's 2D pipeline, such as a parallax scrolling background and the use of sprites. We complete the chapter and the game with the creation of a level-selection menu and the saving of high scores.
Finally, we will create a Monkey Ball-style game. This game involves using the special inputs of a mobile device to control the movement of the ball and the player's interaction with the world. When a player's device is tilted, they will be able to guide the monkey around the level. When they touch the screen, they can do damage and eventually collect bananas that are scattered throughout the game. This game also shows you how to include the special effects that are necessary to complete the look of every game. We create explosions when bananas are collected and dust trails when our monkey moves around. We also add in sound effects for touching and exploding.

We wrap up the book by taking a look at optimization. We explore all the great features of Unity and even create a few of our own to make our game run as best it can. We also take a little bit of time to understand some things that we can do to minimize the file size of our assets while maximizing their look and effect in the game. At this point, our journey ends, but we will have four great games that are just about ready for the market.

What This Book Covers

Chapter 1, Saying Hello to Unity and Android, explores the feature lists of the Android platform and the Unity 3D game engine and explains why they are great choices for development. We also cover the setting up of the development environment and create a simple Hello World application for your device and emulators.

Chapter 2, Looking Good – The Graphical Interface, takes a detailed look at the graphical user interface. By creating a Tic-Tac-Toe game, you learn about the user interface while you make it pleasing to look at.

Chapter 3, The Backbone of Any Game – Meshes, Materials, and Animations, explores how you can utilize meshes, materials, and animations in Unity. Through the creation of a Tank Battle game, we cover the core of what players will see when they play the game.

Chapter 4, Setting the Stage – Camera Effects and Lighting, explains the camera effects and lighting options that are available in Unity. With the addition of shadows, lightmaps, distance fog, and a skybox, our Tank Battle environment becomes more dynamic. By utilizing special camera effects, we create extra feedback for players.

Chapter 5, Getting Around – Pathfinding and AI, shows the creation of bile enemies in our Tank Battle game. We explore pathfinding and AI to give players a target that is more meaningful than a stationary dummy.

Chapter 6, Specialities of the Mobile Device – Touch and Tilt, covers the features that make the modern mobile device special. We create a Monkey Ball-style game to understand the touch interface and tilt controls.
Chapter 7, *Throwing Your Weight Around – Physics and a 2D Camera*, shows you how to create a clone of the Angry Birds game while taking a short break from the Monkey Ball game. Physics and Unity's 2D pipeline are also explored here.

Chapter 8, *Special Effects – Sound and Particles*, returns us to the Monkey Ball game to add special effects. The inclusion of sound effects and particles allows us to create a more complete game experience.

Chapter 9, *Optimization*, covers optimization in Unity 3D. We cover the benefits and costs of making our Tank Battle and Monkey Ball games as efficient as possible.
In the previous chapter, you learned about the special features of a mobile device and how to create touch and tilt controls. We also created a Monkey Ball game to use these new controls. The steering of the ball was done by tilting the device and collecting bananas by touching the screen. We also gave it some win and lose conditions by creating a timer and finish line.

In this chapter, will we take a short break from the Monkey Ball game to explore Unity's physics engine. We will also take a look at the options available for creating a 2D game experience. To do all of this, we will be recreating one of the most popular mobile games on the market, Angry Birds. We will use physics to throw birds and destroy structures. We will also take a look at the creation of a level-selection screen.

In this chapter, we will cover the following topics:

- Unity physics
- Parallax scrolling
- 2D pipelines
- Level selection

We will be creating a new project for this chapter, so start up Unity and let's begin!
2D games in a 3D world

Perhaps the most little-known thing when developing games is the fact that it's possible to create 2D-style games in a 3D game engine, such as Unity. As with everything else, it comes with its own set of advantages and disadvantages, but the choice can be well worth it for generating a pleasing game experience. The foremost advantage is that you can use 3D assets for the game. This allows dynamic lighting and shadows to be easily included. However, when using a 2D engine, any shadow will need to be painted directly into the assets and you will be hard-pressed to make it dynamic. On the side of disadvantages is the use of 2D assets in the 3D world. It is possible to use them, but large file sizes become necessary to achieve the desired detail and to keep it from appearing pixelated. Most 2D engines, however, make use of vector art that will keep the image's lines smooth as it is scaled up and down. Also, one is able to use normal animations for the 3D assets, but frame-by-frame animation is generally required for any 2D asset. Altogether, the advantages have outweighed the disadvantages for many developers, creating a large selection of great looking 2D games that you may never realize were actually made in a 3D game engine.

To address the growing demand from developers for 2D game support, the Unity team has been additionally been working long and hard on creating an optimized 2D pipeline for the 3D engine. When creating your project, you have the option to select 2D defaults, optimizing assets for use in a 2D game. While there is still no direct vector graphics support from Unity, many other features have been optimized to work better in a 2D world. One of the biggest features is the 2D optimization of the physics engine, which we will be focusing on in this chapter. All the principles that we will use will transfer over to 3D physics, which will save some trouble when setting up and working with it.

Setting up the development environment

To explore making a 2D game in a primarily 3D engine, and the use of physics, we will be recreating a highly popular 2D game, Angry Birds. However, before we can dive into the meat of the game, we need to set up our development environment so that we are optimized for 2D game creation. Let's use these steps to do this:

1. To begin with, we need to create a new project in Unity. Naming it Ch7_AngryBirds will work well. We also need to select 2D under Templates, so all the defaults are set for our 2D game.

2. We also need to be sure to change the target platform in the Build Settings field to Android and set Bundle Identifier to an appropriate value. We don't want to have to worry about this later.
3. There are a few differences that you will notice right away. First, you can only pan from side to side and up and down when moving around in the scene. This is a setting that can be toggled in the top-middle of the Scene view, by clicking on the little 2D button. Also, if you select the camera in the Hierarchy window, you can see that it simply appears as a white box in the Scene view. This is because it has been defaulted to use the Orthographic mode for its Projection setting, which you can see in the Inspector panel.

Every camera has two options for how to render the game. A perspective renders everything by utilizing their distance from the camera, imitating the real world; objects that are farther away from the camera are drawn smaller than objects that are closer. An orthographic camera renders everything without this consideration; objects are not scaled based on their distance from the camera.

4. Next, we are going to need a ground. So, go to the menu bar of Unity and navigate to GameObject | 3D Object | Cube. This will work well as a simple ground.

5. To make it look a little like a ground, create a green material and apply it to the Cube GameObject.

6. The ground cube needs to be large enough to cover the whole of our field of play. To do this, set the cube's Scale attribute to 100 for the X axis, 10 for the Y axis, and 5 for the Z axis. Also, set its Position attribute to 30 for the X axis, -5 for the Y axis, and 0 for the Z axis. Since nothing will be moving along the x axis, the ground only needs to be large enough for the other objects that will be in our scene to land on. It does, however, need to be wide and tall enough to keep the camera from seeing the edges.

7. To optimize our ground cube for use in our 2D game, we need to change its collider. Select the Cube GameObject in the Hierarchy window and take a look at it in the Inspector panel. Right-click on the Box Collider component and select Remove Component. Next, at the top of Unity, navigate to Component | Physics 2D | Box Collider 2D. This component works just like a normal Box Collider component, except that it does not have limited depth.

8. Right now, the ground looks rather dark due to the lack of light. From the menu bar of Unity, navigate to GameObject | Light | Directional Light in order to add some brightness to the scene.
9. Next, we need to keep all the objects that will be flying around the scene from straying too far and causing problems. To do this, we need to create some trigger volumes. The simplest way to do this is to create three empty GameObjects and give each a Box Collider 2D component. Be sure to check the Is Trigger checkbox in order to change them into trigger volumes.

10. Position one at each end of the ground object and the last GameObject at about 50 units above. Then, scale them to form a box with the ground. Each should be no thicker than a single unit.

11. To make the volumes actually keep objects from straying too far, we need to create a new script. Create a new script and name it GoneTooFar.

12. This script has a single, short function, OnTriggerEnter2D. We use this function to destroy any object that might enter the volume. This function is used by Unity's physics system to detect when an object has entered a trigger volume. We will go into more detail regarding this later, but for now, know that one of the two objects, either the volume or the object entering it, needs a Rigidbody component. In our case, everything that we might want to remove when they enter the trigger will have a Rigidbody component:

   ```csharp
   public void OnTriggerEnter2D(Collider2D other) {
       Destroy(other.gameObject);
   }
   ```

13. Finally, return to Unity and add the script to the three trigger-volume objects.
We have done the initial setup for our 2D game. By changing the project type from 3D to 2D, defaults in Unity are changed to be optimized for 2D game creation. The most immediately noticeable thing is that the camera is now in the Orthographic view, making everything appear flattened. We also created a ground and some trigger volumes for our scene. Together, these will keep our birds and anything else from straying too far.

Physics
In Unity, physics simulation primarily focuses on the use of the Rigidbody component. When the Rigidbody component is attached to any object, it will be taken over by the physics engine. The object will fall with gravity and bump into any object that has a collider. In our scripts, making use of the OnCollision group of functions and the OnTrigger group of functions requires a Rigidbody component to be attached to at least one of the two interacting objects. However, a Rigidbody component can interfere with any specific movement we might cause the object to take. But the Rigidbody component can be marked as kinematic, which means that the physics engine will not move it, but it will only move when our script moves it. The CharacterController component that we used for our tank is a special, modified Rigidbody. In this chapter, we will be making heavy use of the Rigidbody component to tie all our birds, blocks, and pigs into the physics engine.

Building blocks
For our first physics objects, we will create the blocks that the pig castles are built out of. We will be creating three types of blocks: wood, glass, and rubber. With these few simple blocks, we will be able to easily create a large variety of levels and structures to be smashed with birds.

Each of the blocks we will be creating will be largely similar. So, we will start with the basic one, the wooden plank, and expand upon it to create the others. Let's use these steps to create the blocks:

1. First, we will create the plank of wood. To do this, we need another cube. Rename it Plank_Wood.
2. Set the value of the plank's Scale to 0.25 for the X axis and 2 for both the Y and Z axes. Its scale on the x and y axes defines its size as seen by the player. The scale on the z axis helps ensure that it will be hit by other physics objects in the scene.
3. Next, create a new material using the plank_wood texture and apply it to the cube.
4. To make this new wooden plank into a physics object suitable for our game, we need to remove the cube's **Box Collider** component and replace it with a **Box Collider 2D** component. Also, add a **Rigidbody** component. Make sure that your plank is selected; go to the menu bar of Unity and navigate to **Component | Physics 2D | Rigidbody 2D**.

5. Next, we need to make the plank function properly within our game; we need to create a new script and name it **Plank**.

6. This script begins with a bunch of variables. The first two variables are used to track the health of the plank. We need to separate the total amount of health from the current health, so that we will be able to detect when the object has been reduced to its half-health. At this point, we will make use of our next three variables to change the object's material to one that shows damage. The last variable is used when the object runs out of health and is destroyed. We will use it to increase the player's score:

   ```csharp
   public float totalHealth = 100f;
   private float health = 100f;
   public Material damageMaterial;
   public Renderer plankRenderer;
   private bool didSwap = false;
   public int scoreValue = 100;
   ```

7. For the script's first function, we use **Awake** for initialization. We make sure that the object's current health is the same as its total health and the **didSwap** flag is set to **false**:

   ```csharp
   public void Awake() {
       health = totalHealth;
       didSwap = false;
   }
   ```

8. Next, we make use of the **OnCollisionEnter2D** function, which is just the 2D optimized version of the normal **OnCollisionEnter** function used in 3D. This is a special function, triggered by the **Rigidbody** component, that gives us information about what the object collided with and how. We use this information to find **collision.relativeVelocity.magnitude**. This is the speed at which the objects collided, and we use this as damage in order to reduce the current health. Next, the function checks to see whether the health has been reduced to half and calls the **SwapToDamaged** function if it has. By using the **didSwap** flag, we make sure that the function is only called once.
Finally, the function checks to see whether the health has dropped below zero. If it has, the object is destroyed and we call the LevelTracker script, which we will soon be making, to add to the player's score:

```csharp
public void OnCollisionEnter2D(Collision2D collision) {
    health -= collision.relativeVelocity.magnitude;

    if (!didSwap && health < totalHealth / 2f) {
        SwapToDamaged();
    }

    if (health <= 0) {
        Destroy(gameObject);
        LevelTracker.AddScore(scoreValue);
    }
}
```

9. Finally, for the script, we have the SwapToDamaged function. It starts by setting the didSwap flag to true. Next, it checks to make sure that the plankRenderer and damageMaterial variables have references to other objects. Ultimately, it uses the plankRenderer.sharedMaterial value to change the material to the damaged-looking material:

```csharp
public void SwapToDamaged() {
    didSwap = true;
    if (plankRenderer == null) return;

    if (damageMaterial != null) {
        plankRenderer.sharedMaterial = damageMaterial;
    }
}
```

10. Before we can add our Plank script to our objects, we need to create the LevelTracker script that was mentioned earlier. Create it now.

11. This script is fairly short and starts with a single variable. The variable will track the player's score for the level and is static, so it can easily be changed as objects are destroyed, for points:

```csharp
private static int score = 0;
```

12. Next, we use the Awake function to make sure the player starts at zero when beginning a level:

```csharp
public void Awake() {
    score = 0;
}
```
13. Finally, for the script, we add the AddScore function. This function simply takes the amount of points passed to it and increases the player's score. It is also static, so it can be called by any object in the scene without needing a reference to the script:

```csharp
public static void AddScore(int amount) {
    score += amount;
}
```

14. Back in Unity, we need to create a new material using the plank_wood_damaged texture. This will be the material that the script will swap to.

15. We need to add the Plank script to our Plank_Wood object. Connect the Damaged Material reference to the new material and the Plank Renderer reference to the object's Mesh Renderer component.

16. As we create different types of planks, we can adjust the value of Total Health to give them different strengths. A value of 25 works pretty well for the wood planks.

17. Next, create an empty GameObject and rename it LevelTracker.

18. Add the LevelTracker script to the object, and it will begin to track the player's score.

19. If you want to see the wood plank in action, position it above the ground and hit the play button. As soon as the game starts, Unity's physics will take over and drop the plank with gravity. If it started out high enough, you will be able to see it switch textures as it loses health.

20. To make the other two planks that we need, select the Plank_Wood object and press Ctrl + D twice to duplicate it. Rename one plank to Plank_Glass and the other to Plank_Rubber.

21. Next, create three new materials. One should be purple in color for the rubber plank, one should use the plank_glass texture for the glass plank, and the last material should use the plank_glass_damaged texture for when the glass plank is damaged. Apply the new materials to the proper locations for the new planks.

22. As for the health of the new planks, a value of 15 for the glass and 100 for the rubber will work well.
23. Finally, turn your three planks into prefabs and use them to build a structure for you to knock down. Feel free to scale them in order to make differently sized blocks, but leave the $z$ axis alone. Also, all of the blocks should be positioned at 0 on the $z$ axis and your structure should be centered around about 30 on the $x$ axis.

We have created the building blocks we needed for the structures that are going to be knocked down in our game. We used a **Rigidbody** component to tie them into the physics engine. Also, we created a script that keeps track of their health and swaps to damaged materials when it drops below half. For this game, we are sticking to the 2D optimized versions of all the physics components. They work in exactly the same way as the 3D versions, just without the third axis.

Wood and glass work well as basic blocks. However, if we are going to make harder levels, we need something a little stronger. Try your hand at making a stone block. Create two textures and materials for it to show its pristine and damaged states.
Physics materials

Physics materials are special types of materials that specifically tell the physics engine how two objects should interact. This does not affect the appearance of an object. It defines the friction and bounciness of a collider. We will use them to give our rubber plank some bounce and the glass plank some slide. With these few steps, we can quickly implement physics materials to create a pleasing effect:

1. Physics materials are created in the same way as everything else, in the Project panel. Right-click inside the Project panel and navigate to Create | Physics2D Material. Create two physics materials and name one of them Glass and the other Rubber.

2. Select one of them and take a look at it in the Inspector window. The 2D version has only two values (the 3D version has a few extra values, but they are only used in more complex situations):
   - **Friction**: This property controls the amount of movement lost when sliding along a surface. A value of zero denotes no friction, such as ice, and a value of one denotes a lot of friction, such as rubber.
   - **Bounciness**: This property is how much of an object's energy is reflected when it hits something or is hit by something. Zero means none of the energy is reflected, while a value of one means the object will reflect all of it.

3. For the Glass material, set the Friction value to 0.1 and Bounciness to 0. For the Rubber material, set the Friction to 1 and Bounciness to 0.8.

4. Next, select your Plank_Glass prefab and take a look at its Box Collider 2D component. To apply your new physics materials, simply drag and drop them one by one from the Project panel to the Material slot. Do the same for your Plank_Rubber prefab, and any time an object hits one of them, the materials will be used to control their interaction.

We have created a pair of physics materials. They control how two colliders interact when they run into each other. Using these, we are given control over the amount of friction and bounciness that is possessed by any collider.

Characters

Having a bunch of generic blocks is just the beginning of this game. Next, we are going to create a few characters to add some life to the game. We are going to need some evil pigs to destroy and some good birds to throw at them.
Creating the enemy

Our first character will be the enemy pig. On their own, they don't actually do anything. So, they are really just the wooden blocks we made earlier that happen to look like pigs. To make their destruction the goal of the game, however, we are going to expand our LevelTracker script to watch them and trigger a Game Over event if they are all destroyed. We will also expand the script to update the score on the screen and make it save the score for later use. Unlike our planks, which are cubes that we can only see one side of, pigs are created as flat textures and are used as sprites by Unity's 2D pipeline. Let's get started with these steps to create the pigs for our Angry Birds game:

1. The pigs are created in a manner similar to that of the wood planks; however, they use a special 2D object called a sprite. A sprite is really just a flat object that always looks at the screen. Most 2D games are made with just a series of sprites for all the objects. You can create one by navigating to GameObject | 2D Object | Sprite. Name it Pig.

2. To make your new sprite look like a pig, drag the pig_fresh image from the Project panel and drop it into the Sprite slot of the Sprite Renderer component.

3. Next, add a Circle Collider 2D component and a Rigidbody 2D component. The Circle Collider 2D component works just like the Sphere Collider components we have used previously but is optimized for working in a 2D game.

4. Before we can use our pigs in the game, we need to update the Plank script so that it can handle the changing of sprite images as well as materials. So, we open it up and add a variable at the beginning. This variable simply keeps track of which sprite to change to:
   ```
   public Sprite damageSprite;
   ```

5. Next, we need to add a small part to the end of our SwapToDamaged function. This if statement checks whether a sprite is available to change into. If it is, we convert our generic renderer variable into SpriteRenderer so that we can get access to the sprite variable on it, and update to our new image:
   ```
   if(damageSprite != null) {
       SpriteRenderer spriteRend = plankRenderer as SpriteRenderer;
       spriteRend.sprite = damageSprite;
   }
   ```
6. Add the Plank script to the pig and fill in the Plank Renderer slot with the Sprite Renderer component. Also, put the pig_damage image in the Damage Sprite slot. By changing this script a little, we will be able to save ourselves a lot of trouble later, when we may perhaps want to track the destruction of more than just pigs.

7. Now, turn the pig into a prefab and add it to your structure. Remember that you need to leave them at zero on the z axis, but feel free to adjust their size, health and score values to give them some variety.

8. Next, we need to expand the LevelTracker script. Open it up and we can add some more code.

9. First, we need to add a line at the very beginning of the script, so we can edit the text displayed in our GUI. Just like we have done previously, add this line at the very top of the script, where the other two lines that begin with using are:

```csharp
using UnityEngine.UI;
```

10. We will next add some more variables at the beginning of the script. The first one, as its name suggests, will hold a list of all the pigs in our scene. The next is a flag for signaling that the game has ended. We also have three Text variables, so we can update the player's score while they are playing, tell them why the game ended, and what their final score was. The last variable will allow you to turn on and turn off the final screen, where we tell the player whether or not they won:

```csharp
public Transform[] pigs = new Transform[0];
private gameOver = false;
public Text scoreBox; 
public Text finalMessage;
public Text finalScore;
public GameObject finalGroup;
```

11. Next, we need to add a line to the Awake function. This simply makes sure that the group of GUI objects that tell the player how the game ended are turned off when the game starts:

```csharp
FinalGroup.SetActive(false);
```
12. In the `LateUpdate` function, we first check whether the game has ended. If it hasn't, we call another function to check whether all the pigs have been destroyed. We also update the display of the player's score, both while they are playing and for the game over screen:

```csharp
public void LateUpdate() {
    if (!gameOver) {
        CheckPigs();

        scoreBox.text = "Score: " + score;
        finalScore.text = "Score: " + score;
    }
}
```

13. Next, we add the `CheckPigs` function. This function loops through the list of pigs to see whether they are all destroyed. Should it find one that hasn't been destroyed, it exits the function. Otherwise, the game is flagged as being over and the player is given a message. We also turn off the in-game score and turn on the game over a group of GUI objects:

```csharp
private void CheckPigs() {
    for (int i = 0; i < pigs.Length; i++) {
        if (pigs[i] != null) return;
    }

    gameOver = true;
    finalMessage.text = "You destroyed the pigs!";

    scoreBox.gameObject.SetActive(false);
    finalGroup.SetActive(true);
}
```

14. The `OutOfBirds` function will be called by the slingshot we are going to create later, when the player runs out of birds to launch at the pigs. If the game has not yet ended, the function ends the game and sets an appropriate message for the player. It also turns off the in-game score and turns on the game over a group of GUI objects, just like the previous function:

```csharp
public void OutOfBirds() {
    if (gameOver) return;

    gameOver = true;
    finalMessage.text = "You ran out of birds!";

    scoreBox.gameObject.SetActive(false);
    finalGroup.SetActive(true);
}
```
15. Finally, we have the `SaveScore` function. Here, we use the `PlayerPrefs` class. It lets you easily store and retrieve small amounts of data, perfect for our current needs. We just need to provide it with a unique key to save the data under. For this, we use a short string combined with the level's index, as provided by `Application.loadedLevel`. Next, we use `PlayerPrefs.GetInt` to retrieve the last score that was saved. If there isn't one, the zero that we passed to the function is returned as a default value. We compare the new score with the old score and use `PlayerPrefs.SetInt` to save the new score, if it is higher. Finally, the `Application.LoadLevel` function can be used to load any other scene in our game. All the scenes you intend to load have to be added to the `Build Settings` window, found in the `File` menu, and can be loaded by using either their name or their index, as shown here:

```csharp
public void SaveScore() {
    string key = "LevelScore" + Application.loadedLevel;
    int previousScore = PlayerPrefs.GetInt(key, 0);
    if(previousScore < score) {
        PlayerPrefs.SetInt(key, score);
    }

    Application.LoadLevel(0);
}
```

Note that using `PlayerPrefs` is by far the easiest method of storing saved information in Unity. However, it is not the most secure. If you have experience changing values in the registry of your computer, you can easily find and make changes to these `PlayerPrefs` values from outside the game. This by no means makes it a bad path for storing game information. You should just be aware of it in case you ever make a game and wish to prevent the player from hacking and changing values in their game saves.

16. Next, we need to create some GUI objects so that our player can see how they are doing in the game. Remember that you can find them by navigating to `GameObject | UI`. We are going to need three text objects, a button, and a panel.

17. The first text object should be named `Score`. It will display the player’s points while the level is in progress. Anchor and position it in the top-left corner of the `Canvas` area.

18. The button needs to be a child of the panel. It should be anchored to the center of the screen and positioned just below it. Also, change the text of the button to something meaningful; `Return to Level Select` will work well here.
19. For **On Click**, we need to click on the plus sign to add a new event. Select the `SaveScore` function of the `LevelTracker` script. Otherwise, we will not be able to record the player’s high score and leave the level.

20. The last two text objects should also be made children of the panel. Name one of them `Message`; it will tell our player why the level ended. The other should be named `FinalScore`, displaying the player’s score when they are finished. They both need to be anchored to the center of the screen as well. Position the `FinalScore` object above the button, and the message above that.

21. Finally, all the pig objects in our scene need to be added to the `LevelTracker` script’s list by dragging and dropping each pig in the `Pigs` value under the `Inspector` window. Also, put each text object into its slot and the panel into the `Final Group` slot.

We created the pigs and updated our `LevelTracker` script to track them. The pigs are really just like the planks of wood, but they are circles instead of boxes. The updated `LevelTracker` script watches for the instance when all the pigs are destroyed and triggers a **Game Over** screen when they are. It also draws the score while the game is being played and saves this score when the level is over.
Our game doesn't quite work yet, but that doesn't mean it has to look like the defaults that Unity provides. Use your skills from the previous chapters to make the interface elements that you have look better. Even just a change in the font will make a world of difference to how our game looks. Perhaps even try changing the background image of Panel, to add that last bit of flare to our game over screen.

Creating the ally

Next, we need something to throw at the pigs and their fortifications. Here, we will create the simplest of birds. The red bird is essentially just a rock. It has no special powers and there is nothing particularly special about its code, besides health. You will also notice that the bird is a 3D model, giving it the shadows that the pigs are missing. Let's use these steps to create the red bird:

1. The red bird is another 3D model, so it is set up in a manner similar to that of the planks. Create an empty GameObject, naming it Bird_Red, and add the appropriate model from the birds model as a child, zeroing out its position and scaling it as needed to make it about a single unit across. The model should be rotated to align it along the x axis. If turned a little more toward the camera, the player is able to see the bird's face while still giving the impression of looking down the field of play.

2. Next, give it a Circle Collider 2D component and a Rigidbody 2D component.

3. Now, we need to create a new script named Bird. This script will be a base for all our birds, tracking their health and triggering their special powers when appropriate.

4. The script starts with three variables. The first will keep track of the bird's current health. The second is a flag, so the bird will only use its special power once. It is marked as protected so that all our birds can use it while protecting it from interference from outside sources. The last will hold a reference to our Rigidbody component:

   ```csharp
   public float health = 50;
   protected bool didSpecial = false;
   public Rigidbody2D body;
   ```

5. The Update function does three checks before activating the bird's special power. First, it checks whether it has already been done and then whether the screen has been touched. We can easily check whether any amount of touching has been done in this frame by checking the left mouse button, which Unity triggers if we touch our screen. Finally, it checks whether the bird has a Rigidbody component and whether it is being controlled by another script:

   ```csharp
   public void Update() {
       if(didSpecial) return;
   }
   ```
if(!Input.GetMouseButtonDown(0)) return;
if(body == null || body.isKinematic) return;

DoSpecial();
}

6. In the case of the red bird, the DoSpecial function only sets its flag to true. It
is marked as virtual so that we can override the function for the other birds
and make them do some fancy things:
protected virtual void DoSpecial() {
    didSpecial = true;
}

7. The OnCollisionEnter2D function works in a similar way to that of
the planks, subtracting health based on the strength of the collision and
destroying the bird if it runs out of health:
public void OnCollisionEnter2D(Collision2D collision) {
    health -= collision.relativeVelocity.magnitude;
    if(health < 0)
        Destroy(gameObject);
}

8. Return to Unity and add the script to the Bird_Red object.
9. Complete the bird's creation by turning it into a prefab and deleting it from
the scene. The slingshot we will be creating next will handle the creation of
the birds when the game starts.

We created the red bird. It is set up just like our other physics objects. We also
created a script to handle the bird's health. This script will be expanded later when
we create the other birds for our game.

Controls
Next, we are going to give the player the ability to interact with the game. First,
we will create a slingshot to throw the birds. Following that we will create the
camera controls. We will even create a nice background effect to round out the
look of our game.
### Attacking with a slingshot

To attack the pig fortress, we have our basic bird ammo. We need to create a slingshot to hurl this ammo at the pigs. It will also handle the spawning of the birds at the beginning of the level and automatically reload as birds are used. When the slingshot runs out of birds, it will notify the `LevelTracker` script and the game will end. Finally, we will create a script that will keep the physics simulation from going on for too long. We don't want to force the player to sit and watch a pig slowly roll across the screen. So, the script will, after a little while, start damping the movement of the `Rigidbody` components to make them stop rather than keep rolling. To do all of this, we are going to follow these steps:

1. To start off with the creation of the slingshot, add the slingshot model to the scene and position it at the origin. Scale it, as necessary, to make it about four units tall. Apply a light brown material to the `Fork` model and a dark brown one to the `Pouch` model.

2. Next, we need four empty GameObjects. Make them all the children of the `Slingshot` object. Name the first GameObject `FocalPoint` and center it between the forked prongs of the slingshot. This will be the point through which we fire all the birds.

   The second GameObject is `Pouch`. First, set its rotation to 0 for the X axis, 90 for the Y axis, and 0 for the Z axis, making the blue arrow point forward along our field of play. Next, make the pouch model a child of this object, setting its position to 0 on the X and Y axes and -0.5 on the Z axis and its rotation to 270 for X, 90 for Y, and 0 for Z. This will make the pouch appear in front of the current bird without having to make a complete pouch model.

   The third GameObject is `BirdPoint`; this will position the bird that is being fired. Make it a child of the `Pouch` point and set its position to 0.3 on the X axis and 0 for the Y and Z axes.

   The last GameObject is `WaitPoint`; the birds waiting to be fired will be positioned behind this point. Set its position to -4 for the X axis, 0.5 for the Y axis, and 0 for the Z axis.

3. Next, rotate the `Fork` model so that we can see both prongs of the fork while it appears to be pointing forward. The values of 270 for the X axis, 290 for the Y axis, and 0 for the Z axis will work well.

4. The `Slingshot` script will provide most of the interaction for the player. Create it now.
5. We start this script with a group of variables. The first group will keep a reference to the damper that was mentioned earlier. The second group will keep track of the birds that will be used in the level. Next is a group of variables that will track the current bird that is ready to be fired. Fourth, we have some variables to hold references to the points we created a moment ago. The maxRange variable is the distance from the focal point to which the player can drag the pouch. The last two variables define how powerfully the bird will be launched:

public RigidbodyDamper rigidbodyDamper;

public GameObject[] levelBirds = new GameObject[0];
private Rigidbody2D[] currentBirds;
private int nextIndex = 0;
public Transform waitPoint;
public Rigidbody2D toFireBird;
public bool didFire = false;
public bool isAiming = false;

public Transform pouch;
public Transform focalPoint;
public Transform pouchBirdPoint;

public float maxRange = 3;

public float maxFireStrength = 25;
public float minFireStrength = 5;

6. As with the other scripts, we use the Awake function for initialization. The levelBirds variable will hold references to all the bird prefabs that will be used in the level. We start by creating an instance of each one and storing its Rigidbody in the currentBirds variable. The isKinematic variable is set to true on each bird's Rigidbody component so that it does not move when it is not in use. Next, it readies the first bird to be fired and, finally, it positions the remaining birds behind waitPoint:

public void Awake() {
    currentBirds = new Rigidbody2D[levelBirds.Length];
    for (int i = 0; i < levelBirds.Length; i++) {
        GameObject nextBird = Instantiate(levelBirds[i]) as GameObject;
        Rigidbody2D rigidbody2D = nextBird.GetComponent<Rigidbody2D>();
        currentBirds[i] = rigidbody2D;
        currentBirds[i].isKinematic = true;
    }
}
7. The `ReadyNextBird` function first checks whether we have run out of birds. If so, it finds the `LevelTracker` script to tell it that there are no birds left to fire. The `nextIndex` variable tracks the current location of the birds in the list to be fired by the player. Next, the function stores the next bird in the `toFireBird` variable and makes it a child of the `BirdPoint` object we created; its position and rotation are zeroed out. Finally, the fire and aim flags are reset:

```csharp
public void ReadyNextBird() {
    if(currentBirds.Length <= nextIndex) {
        LevelTracker tracker = FindObjectOfType(typeof(LevelTracker)) as LevelTracker;
        tracker.OutOfBirds();
        return;
    }

    toFireBird = currentBirds[nextIndex];
    nextIndex++;

    toFireBird.transform.parent = pouchBirdPoint;
    toFireBird.transform.localPosition = Vector3.zero;
    toFireBird.transform.localRotation = Quaternion.identity;

    didFire = false;
    isAiming = false;
}
```

8. The `SetWaitPositions` function uses the position of `waitPoint` to position all the remaining birds behind the slingshot:

```csharp
public void SetWaitPositions() {
    for(int i=nextIndex;i<currentBirds.Length;i++) {
        if(currentBirds[i] == null) continue;
        Vector3 offset = Vector3.right * (i – nextIndex) * 2;
        currentBirds[i].transform.position = waitPoint.position – offset;
    }
}
```
9. The `Update` function starts by checking whether the player has fired a bird, and watches the `rigidbodyDamper.allSleeping` variable to see whether all the physics objects have stopped moving. Once they do, the next bird is readied to be fired. If we have not fired, the aiming flag is checked and the `DoAiming` function is called to handle the aiming. If the player is neither aiming nor has just fired a bird, we check for touch input. If the player touches close enough to the focal point, we flag that the player has started aiming:

```csharp
public void Update() {
    if(didFire) {
        if(rigidbodyDamper.allSleeping) {
            ReadyNextBird();
            SetWaitPositions();
        }
        return;
    }
    else if(isAiming) {
        DoAiming();
    } else {
        if(Input.touchCount <= 0) return;
        Vector3 touchPoint = GetTouchPoint();
        isAiming = Vector3.Distance(touchPoint, focalPoint.position) < maxRange / 2f;
    }
}
```

10. The `DoAiming` function checks whether the player has stopped touching the screen and fires the current bird when they have. If they have not, we position the pouch at the current touch point. Finally, the pouch's position is limited to keep it within the maximum range:

```csharp
private void DoAiming() {
    if(Input.touchCount <= 0) {
        FIREBIRD();
        return;
    }

    Vector3 touchPoint = GetTouchPoint();

    pouch.position = touchPoint;
    pouch.LookAt(focalPoint);
```
float distance = Vector3.Distance(focalPoint.position, pouch.position);
if (distance > maxRange) {
    pouch.position = focalPoint.position - (pouch.forward * maxRange);
}
}

11. The GetTouchPoint function uses ScreenPointToRay to find out where the player is touching in 3D space. This is similar to when we were touching bananas; however, because this game is 2D, we can just look at the ray's origin and return a zero for its z axis value:

private Vector3 GetTouchPoint() {
    Ray touchRay = Camera.main.ScreenPointToRay(Input.GetTouch(0).position);
    Vector3 touchPoint = touchRay.origin;
    touchPoint.z = 0;
    return touchPoint;
}

12. Finally, for this script, we have the FireBird function. This function starts by setting our didFire flag to true. Next, it finds out the direction in which the bird needs to be fired by finding the direction from the pouch's position to focalPoint. It also uses the distance between them to determine the power with which the bird needs to be fired, clamping it between our minimum and maximum strengths. Then, it releases the bird by clearing its parent and setting its isKinematic flag to false, after finding its Rigidbody component. To launch it, we use the AddForce function and pass it the direction multiplied by the power. ForceMode2D.Impulse is also passed to make that the force applied happens once and is immediate. Next, the pouch is positioned at focalPoint, as if it were actually under tension. Finally, we call rigidbodyDamper.ReadyDamp to start the damping of the Rigidbody component's movement:

private void FireBird() {
    didFire = true;

    Vector3 direction = (focalPoint.position - pouch.position).normalized;
    float distance = Vector3.Distance(focalPoint.position, pouch.position);
    float power = distance <= 0 ? 0 : distance / maxRange;
    power *= maxFireStrength;
    power = Mathf.Clamp(power, minFireStrength, maxFireStrength);
toFireBird.transform.parent = null;
toFireBird.isKinematic = false;
toFireBird.AddForce(new Vector2(direction.x, direction.y) *
power, ForceMode2D.Impulse);

pouch.position = focalPoint.position;

rigidbodyDamper.ReadyDamp();

13. Before we can make use of the Slingshot script, we need to create the RigidbodyDamper script.

14. This script starts with the following six variables. The first two define how long you need to wait before damping movement and how much you need to damp it by. The next two track whether damping can be applied and when it will start. The next is a variable that will be filled with a list of all the rigidbodies that are currently in the scene. Finally, it has the allSleeping flag that will be set to true when the movement has stopped:

```csharp
public float dampWaitLength = 10f;
public float dampAmount = 0.9f;
private float dampTime = -1f;
private bool canDamp = false;
private Rigidbody2D[] rigidbodies = new Rigidbody2D[0];

public bool allSleeping = false;
```

15. The ReadyDamp function starts by using FindObjectsOfType to fill the list with all the rigidbodies. The dampTime flag is set when you need to start damping as the sum of the current time and the wait length. It marks that the script can do its damping and resets the allSleeping flag. Finally, it uses StartCoroutine to call the CheckSleepingRigidbodies function. This is a special way of calling functions to make them run in the background without blocking the rest of the game from running:

```csharp
public void ReadyDamp() {
    rigidbodies = FindObjectsOfType(typeof(Rigidbody2D)) as Rigidbody2D[];
    dampTime = Time.time + dampWaitLength;
    canDamp = true;
    allSleeping = false;

    StartCoroutine(CheckSleepingRigidbodies());
}
```
16. In the `FixedUpdate` function, we first check whether we can damp the movement and whether it is time to do it. If it is, we loop through all the rigidbodies, applying our damp to each one’s rotational and linear velocity. Those that are kinematic, controlled by scripts, and already sleeping—meaning that they have stopped moving—are skipped:

```csharp
public void FixedUpdate() {
    if (!canDamp || dampTime > Time.time) return;

    foreach (Rigidbody2D next in rigidbodies) {
        if (next != null && !next.isKinematic && !next.isSleeping()) {
            next.angularVelocity *= dampAmount;
            next.velocity *= dampAmount;
        }
    }
}
```

17. The `CheckSleepingRigidbody` function is special and will run in the background. This is made possible by the `IEnumerator` flag at the beginning of the function and the `yield return null` line in the middle. Together, these allow the function to pause regularly and keep the rest of the game from freezing while it waits for the function to complete. The function starts by creating a check flag and using it to check whether all the rigidbodies have stopped moving. If one is still found to be moving, the flag is set to `false` and the function pauses until the next frame, when it will try again. When it reaches the end, because all the rigidbodies are sleeping, it sets the `allSleeping` flag to `true` so that the slingshot can be made ready for the next bird. It also stops itself from damping while the player is getting ready to fire the next bird:

```csharp
private IEnumerator CheckSleepingRigidbody() {
    bool sleepCheck = false;

    while (!sleepCheck) {
        sleepCheck = true;

        foreach (Rigidbody2D next in rigidbodies) {
            if (next != null && !next.isKinematic && !next.isSleeping()) {
                sleepCheck = false;
                yield return null;
                break;
            }
        }
    }
}
```
18. Finally, we have the `AddBodiesToCheck` function. This function will be used by anything that spawns new physics objects after the player has fired the bird. It starts by creating a temporary list and expanding the current one. Next, it adds all the values from the temporary list to the expanded one. Finally, the list of rigidbodies are added after those of the temporary list:

```csharp
public void AddBodiesToCheck(Rigidbody2D[] toAdd) {
    Rigidbody2D[] temp = rigidbodies;
    rigidbodies = new Rigidbody2D[temp.Length + toAdd.Length];
    for (int i = 0; i < temp.Length; i++) {
        rigidbodies[i] = temp[i];
    }
    for (int i = 0; i < toAdd.Length; i++) {
        rigidbodies[i + temp.Length] = toAdd[i];
    }
}
```

19. Return to Unity and add the two scripts to the `Slingshot` object. In the `Slingshot` script component, connect the references to the `Rigidbody Damper` script component and to each of the points. Also, add as many references to the red bird prefab to the `Level Birds` list as you want for the level.

20. To keep objects from rolling back and through the slingshot, add a `Box Collider 2D` component to `Slingshot` and position it at the stock of the `Fork` model.

21. To finish off the look of the slingshot, we need to create the elastic bands that tie the pouch to the fork. We will do this by first creating the `SlingshotBand` script.

22. The script starts with two variables, one for the point that the band will end at and one to reference the `LineRenderer` variable that will draw it:

```csharp
public Transform endPoint;
public LineRenderer lineRenderer;
```

23. The `Awake` function ensures that the `lineRenderer` variable has only two points and sets their initial positions:

```csharp
public void Awake() {
    if (lineRenderer == null) return;
    if (endPoint == null) return;
    ```
lineRenderer.SetVertexCount(2);
lineRenderer.SetPosition(0, transform.position);
lineRenderer.SetPosition(1, endPoint.position);
}

24. In the LateUpdate function, we set the lineRenderer variable's end position to the endPoint value. This point will move around with the pouch, so we need to constantly update the renderer:
public void LateUpdate() {
    if(endPoint == null) return;
    if(lineRenderer == null) return;

    lineRenderer.SetPosition(1, endPoint.position);
}

25. Return to Unity and create an empty GameObject. Name it Band_Near and make it a child of the Slingshot object.

26. As children of this new point, create a cylinder and a second empty GameObject, named Band.

27. Give the cylinder a brown material and position it around the near prong of the slingshot fork. Be sure to remove the Capsule Collider component so that it doesn't get in the way. Also, don't be afraid to scale it in order to make it fit the look of the slingshot better.

28. To the Band object, add a Line Renderer component found under Effects in the Component menu. After positioning it in the center of the cylinder, add the SlingshotBand script to the object.

29. To the Line Renderer component under Materials, you can put your brown material in the slot to color the band. Under Parameters, set the Start Width to 0.5 and the End Width to 0.2 in order to set the size of the line.

30. Next, create another empty GameObject and name it BandEnd_Near. Make it a child of the Pouch object and position it inside the pouch.

31. Now, connect the script's references to its line renderer and end point.

32. To make the second band, duplicate the four objects we just created and position them according to the other prong of the fork. The end point for this band can just be moved back along the z axis to keep it out of the way of the birds.
33. Finally, turn the whole thing into a prefab so that it can be easily reused in other levels.

We created the slingshot that will be used to fire birds. We used techniques that we learned in the previous chapter to handle touch input and to track the player's finger while they aim and shoot. If you save your scene and position the camera to look at the slingshot, you will notice that it is complete, if not entirely playable. Birds can be fired at the pig fortress, although we can only see the destruction from within Unity's Scene view.

**Watching with the camera**

The game is technically playable at this point, but it is kind of hard to see what is going on. Next, we will create a system to control the camera. The system will allow the player to drag the camera to the left and right, follow the bird when it is launched, and return to the slingshot when everything stops moving. There will also be a set of limits to keep the camera from going too far and viewing things we do not want the player to see, such as beyond the edge of the ground or sky we have created for the level. We will only need one, fairly short, script to control and manage our camera. Let's create it with these steps:

1. To start and to keep everything organized, create a new empty GameObject and name it CameraRig. Also, to keep it simple, set its position to zero on each axis.
2. Next, create three more empty GameObjects and name them LeftPoint, RightPoint, and TopPoint. Set their Z axis positions to -5. Position the LeftPoint object to be in front of the slingshot and 3 on the Y axis. The RightPoint object needs to be positioned in front of the pig structure you created. The TopPoint object can be over the slingshot but needs to be set to 8 on the Y axis. These three points will define the limits of where our camera can move when being dragged and following the birds.

3. Make all the three points and the Main Camera object children of the CameraRig object.

4. Now, we create the CameraControl script. This script will control all the movement and interaction with the camera.

5. Our variables for this script start with a reference to the slingshot; we need this so that we can follow the current bird when it is fired. The next are the references to the points we just created. The next group of variables control for how long the camera will sit without input before returning to take a look at the slingshot and how fast it will return. The dragScale variable controls how fast the camera actually moves when the player drags their finger across the screen, allowing you to keep the scene moving with the finger. The last group controls whether the camera can follow the current bird and how fast it can do so:

```csharp
public Slingshot slingshot;
public Transform rightPoint;
public Transform leftPoint;
public Transform topPoint;

public float waitTime = 3f;
private float headBackTime = -1f;
private Vector3 waitPosition;
private float headBackDuration = 3f;

public float dragScale = 0.075f;

private bool followBird = false;
private Vector3 followVelocity = Vector3.zero;
public float followSmoothTime = 0.1f;
```
6. In the Awake function, we first make certain that the camera is not following a bird and make it wait before heading to take a look at the slingshot. This allows you to initially point the camera to the pig fortress when the level starts and move to the slingshot after giving the player a chance to see what they are up against:

```java
public void Awake() {
    followBird = false;
    StartWait();
}
```

7. The StartWait function sets the time when it will start to head back to the slingshot and records the position that it is heading back from. This allows you to create a smooth transition:

```java
public void StartWait() {
    headBackTime = Time.time + waitTime;
    waitPosition = transform.position;
}
```

8. Next, we have the Update function. This function starts by checking whether the slingshot has fired. If it hasn't, it checks whether the player has started aiming, signaling that the bird should be followed and zeroing out the velocity if they have. If they have not started aiming, the followBird flag is cleared. Next, the function checks whether it should follow and does so if it should, also calling the StartWait function—in case this is the frame in which the bird is destroyed. If it should not follow the bird, it checks for touch input and drags the camera if it finds any. The wait is again started in case the player removes their finger from this frame. Finally, it checks to see whether the slingshot is done firing the current bird and whether it is time to head back. Should both be true, the camera moves back to point at the slingshot:

```java
public void Update() {
    if(!slingshot.didFire) {
        if(slingshot.isAiming) {
            followBird = true;
            followVelocity = Vector3.zero;
        } else {
            followBird = false;
        }
    }

    if(followBird) {
        FollowBird();
    }
}
```
9. The FollowBird function starts by making sure that there is a bird to follow, by checking the toFireBird variable on the Slingshot script, and stops following if a bird is not found. Should there be a bird, the function then determines a new point to move to, which will look directly at the bird. It then uses the Vector3.SmoothDamp function to smoothly follow the bird. This function works similar to a spring—the farther away it is from its target position, the faster it moves the object. The followVelocity variable is used to keep it moving smoothly. Finally, it calls another function to limit the camera's position within the bounding points we set up earlier:

```csharp
private void FollowBird() {
    if(slingshot.toFireBird == null) {
        followBird = false;
        return;
    }

    Vector3 targetPoint = slingshot.toFireBird.transform.position;
    targetPoint.z = transform.position.z;

    transform.position = Vector3.SmoothDamp(transform.position,
        targetPoint, ref followVelocity, followSmoothTime);
    ClampPosition();
}
```

10. In the DragCamera function, we use the deltaPosition value of the current touch to determine how far it has moved since the last frame. By scaling this value and subtracting the vector from the camera's position, the function moves the camera as the player drags across the screen. This function also calls upon the ClampPosition function to keep the camera's position within the field of play:

```csharp
private void DragCamera() {
    transform.position -= new Vector3(Input.GetTouch(0).deltaPosition.x, Input.GetTouch(0).deltaPosition.y, 0) *
11. The ClampPosition function starts by taking the camera's current position. It then clamps the x position to be between those of the leftPoint and rightPoint variables' x positions. Next, the y position is clamped between the leftPoint and topPoint variables' y positions. Finally, the new position is reapplied to the camera's transform:

```csharp
private void ClampPosition() {
    Vector3 clamped = transform.position;
    clamped.x = Mathf.Clamp(clamped.x, leftPoint.position.x, rightPoint.position.x);
    clamped.y = Mathf.Clamp(clamped.y, leftPoint.position.y, topPoint.position.y);
    transform.position = clamped;
}
```

12. Finally, we have the BackToLeft function. It starts by using the time and our duration variable to determine how much progress the camera will have made to return to the slingshot. It records the camera's current position and uses Mathf.SmoothStep on both the x and y axes to find a new position that is at an appropriate distance between the waitPosition variable and the leftPoint variable. Finally, the new position is applied:

```csharp
private void BackToLeft() {
    float progress = (Time.time - headBackTime) / headBackDuration;
    Vector3 newPosition = transform.position;
    newPosition.x = Mathf.SmoothStep(waitPosition.x, leftPoint.position.x, progress);
    newPosition.y = Mathf.SmoothStep(waitPosition.y, leftPoint.position.y, progress);
    transform.position = newPosition;
}
```

13. Next, return to Unity and add the new script to the Main Camera object. Connect the references to the slingshot and each of the points to finish it off.

14. Position the camera to point at your pig fortress and turn the whole rig into a prefab.
We created a camera rig that will let the player watch all the action as they play the
game. The camera will now follow the birds as they are fired from the slingshot and
can now be dragged by the player. By keying off the positions of a few objects, this
movement is limited to keep the player from seeing things we don't want them to; if
the camera is left idle for long enough, it will also return to look at the slingshot.

Another function of the camera, common to many mobile games, is the pinch-to-
zoom gesture. It is such a simple gesture for the user to expect, but it can be complex
for us to implement well. Try your hand at implementing it here. You can use
Input.touchCount to detect whether there are two fingers touching the screen.
Then, using the Vector2.Distance function, if you have recorded the distance from
the last frame, it is possible to determine whether they are moving toward or away
from each other. Once you have determined your zoom direction, just change the
camera's orthographicSize variable to change how much can be seen; be sure to
include some limits so that the player can't zoom in or out forever.

Now that we have all the pieces needed to make a complete level, we need some
more levels. We need at least two more levels. You can use the blocks and pigs to
create any level you might want. It is a good idea to keep structures centered around
the same spot as our first level, giving the player an easier time dealing with them.
Also, think about the difficulty of the level while making it so that you end up with
an easy, medium, and hard level.

Creating the parallax background
A great feature of many 2D games is a parallax scrolling background. This simply
means that the background is created in layers that scroll by at different speeds.
Think of it as if you are looking out the window of your car. The objects that are far
away appear to hardly move, while the ones that are near move by quickly. In a 2D
game, it gives the illusion of depth and adds a nice touch to the look of the game. For
this background, we will be layering several materials on a single plane. There are
several other methods to create this same effect, but ours will make use of a single
script that additionally allows you to control the speed of each layer. Let's create it
with these steps:

1. We will start this section with the creation of the ParallaxScroll script.
2. This script starts with three variables. The first two variables keep track of
each material and how fast they should scroll. The third keeps track of the
camera's last position, so we can track how far it moves in each frame:

   ```csharp
   public Material[] materials = new Material[0];
   public float[] speeds = new float[0];
   private Vector3 lastPosition = Vector3.zero;
   ```
3. In the Start function, we record the camera’s beginning position. We use Start instead of Awake here, in case the camera needs to do any special movement at the beginning of the game:

```csharp
public void Start() {
    lastPosition = Camera.main.transform.position;
}
```

4. Next, we use the LateUpdate function to make changes after the camera has moved about. It starts by finding the camera's new position and comparing the x axis values to determine how far it has moved. Next, it loops through the list of materials. The loop first gathers the current offset of its texture using mainTextureOffset. Next, the camera's movement multiplied by the material's speed is subtracted from the offset's x axis to find a new horizontal position. Then, the new offset is applied to the material. Finally, the function records the camera's last position for the next frame:

```csharp
public void LateUpdate() {
    Vector3 newPosition = Camera.main.transform.position;
    float move = newPosition.x - lastPosition.x;

    for(int i=0; i<materials.Length; i++) {
        Vector2 offset = materials[i].mainTextureOffset;
        offset.x -= move * speeds[i];
        materials[i].mainTextureOffset = offset;
    }

    lastPosition = newPosition;
}
```

5. Return to Unity and create six new materials. One for each background texture: sky, hills_tall, hills_short, grass_light, grass_dark, and fronds. All the materials, except for sky, need to use Transparent Render Mode. If they do not, we will not be able to see all the textures when they are layered.

6. Before we can tile the images in the background, we need to adjust their Import Settings. Select each in turn and take a look at the Inspector window. Because we chose to make a 2D game, Unity imports all the images as sprites by default, which clamps the edges of our images and keeps them from repeating. For all our background images, change the Texture Type option to Texture and the Wrap Mode option to Repeat. This will let us use them in a way that makes it look like an infinite scrolling background.
7. We also need to adjust the Tiling option for each of these new materials. For all of them, leave the Y axis as 1. For the X axis, set 5 for the sky, 6 for hills_tall, 7 for hills_shot, 8 for grass_dark, 9 for fronds, and 10 for grass_light. This will offset all the features of the textures, so a long pan does not see features regularly lining up.

8. Next, create a new plane. Name it Background and remove its Mesh Collider component. Also, attach our ParallaxScroll script.

9. Position it at 30 on the X axis, 7 on the Y axis, and 10 on the Z axis. Set its rotation to 90 for the X axis, 180 for the Y axis, and 0 for Z. Also, set the scale to 10 for the X axis, 1 for the Y axis, and 1.5 for the Z axis. Altogether, these position the plane to face the camera and fill the background.

10. In the plane’s Mesh Renderer component, expand the Materials list and set the value of Size to 6. Add each of our new materials to the list slots in the order of sky, hills_tall, hills_short, grass_dark, fronds, and grass_light. Do the same for the Materials list in the Parallax Scroll script component.

11. Finally, in the Parallax Scroll script component, set the value of Size of the Speeds list to 6 and input the following values in the order of 0.03, 0.024, 0.018, 0.012, 0.006, and 0. These values will move the materials gently and evenly.

12. At this point, turning the background into a prefab will make it easy to reuse later.
We created a parallax scroll effect. This effect will pan a series of background textures, giving the illusion of depth in our 2D game. To easily see it in action, press the play button and grab the camera in the Scene view, moving it from side to side in order to see the background change.

We have two other levels to add backgrounds to. Your challenge here is to create your own background. Use the techniques you learned in this section to create a night-style background. It can include a stationary moon, while everything else scrolls in the shot. For an added trick, create a cloud layer that slowly pans across the screen as well as with the camera and the rest of the background.

Adding more birds
There is one last set of assets that we need to create for our levels: the other birds. We will create three more birds that each have a unique special ability: a yellow bird that accelerates, a blue bird that splits into multiple birds, and a black bird that explodes. With these, our flock will be complete.

To make the creation of these birds easier, we will be making use of a concept called inheritance. Inheritance allows a script to expand upon the functions it is inheriting without the need to rewrite them. If used correctly, this can be very powerful and, in our case, will aid in the quick creation of multiple characters that are largely similar.

The yellow bird
First, we will create the yellow bird. Largely, this bird functions exactly as the red bird. However, when the player touches the screen a second time, the bird’s special ability is activated and its speed increases. By extending the Bird script that we created earlier, this bird’s creation becomes quite simple. Because of the power of inheritance, the script we are creating here consists of only a handful of lines of code. Let’s create it with these steps:

1. Start by creating the yellow bird in the same way as the red bird, using the YellowBird model instead.
2. Instead of using the Bird script, we will create the YellowBird script.
3. This script needs to extend the Bird script, so replace MonoBehavior with Bird on line four of our new script. It should look similar to the following code snippet:

```csharp
public class YellowBird : Bird {
```
4. This script adds a single variable that will be used to multiply the bird's current velocity:
   ```csharp
   public float multiplier = 2f;
   ```

5. Next, we override the `DoSpecial` function and multiply the bird's `body.velocity` variable when it is called:
   ```csharp
   protected override void DoSpecial() {
       didSpecial = true;
       body.velocity *= multiplier;
   }
   ```

6. Return to Unity, add the script to your new bird, connect the `Rigidbody` component reference, and turn it into a prefab. Add some to the list on your slingshot in order to use the bird in your level.

We created the yellow bird. This bird is simple. It directly modifies its velocity to suddenly gain a boost of speed when the player touches the screen. As you will soon see, we use this same style of script to create all our birds.

**The blue bird**

Next, we will create the blue bird. This bird splits into three birds when the player touches the screen. It will also extend the `Bird` script by using inheritance, reducing the amount of code that needs to be written to create the bird. Let's do it with these steps:

1. Again, start building your blue bird the same way as the previous two birds were built, substituting the appropriate model. You should also adjust the value of `Radius` of the `Circle Collider 2D` component to align appropriately with the small size of this bird.

2. Next, we create the `BlueBird` script.

3. Again, adjust line four so that the script extends `Bird` instead of `MonoBehaviour`:
   ```csharp
   public class BlueBird : Bird {
   ```

4. This script has three variables. The first variable is a list of prefabs to spawn when the bird splits. The next is the angle difference between each new bird that will be launched. The final variable is a value to spawn the birds a little ahead of their current position in order to keep them from getting stuck inside each other:
   ```csharp
   public GameObject[] splitBirds = new GameObject[0];
   public float launchAngle = 15f;
   public float spawnLead = 0.5f;
   ```
5. Next, we override the DoSpecial function and start, as with the others, by marking that we made our special move. Next, it calculates half of the number of birds to spawn and creates an empty list to store the rigidbodies of the newly spawned birds:

```csharp
protected override void DoSpecial() {
    didSpecial = true;

    int halfLength = splitBirds.Length / 2;
    Rigidbody2D[] newBodies = new Rigidbody2D[splitBirds.Length];
```

6. The function continues by looping through the list of birds, skipping the slots that are empty. It spawns the new birds at their position; after trying to store the object's Rigidbody, it goes on to the next one if it is missing. The new Rigidbody component is then stored in the list:

```csharp
for(int i=0;i<splitBirds.Length;i++) {
    if(splitBirds[i] == null) continue;

    GameObject next = Instantiate(splitBirds[i], transform.position, transform.rotation) as GameObject;
    Rigidbody2D nextBody = next.GetComponent<Rigidbody2D>();
    if(nextBody == null) continue;

    newBodies[i] = nextBody;
```

7. Using Quaternion.Euler, a new rotation is created that will angle the new bird along a path that is split off from the main path. The new bird's velocity is set to the rotated velocity of the current bird. An offset is calculated and it is then moved forward along its new path, so as to get out of the way of the other birds being spawned:

```csharp
Quaternion rotate = Quaternion.Euler(0, 0, launchAngle * (i - halfLength));
nextBody.velocity = rotate * nextBody.velocity;
Vector2 offset = nextBody.velocity.normalized * spawnLead;
next.transform.position += new Vector3(offset.x, offset.y, 0);
```

8. After the loop, the function uses FindObjectOfType to find the slingshot that is currently in the scene. If it is found, it is changed to track the first new bird spawned as the one that was fired. The new list of rigidbodies is also set to the rigidbodyDamper variable, in order to be added to its list of rigidbodies. Finally, the script destroys the bird it is attached to, completing the illusion that the bird has been split apart:

```csharp
Slingshot slingshot = FindObjectOfType(typeof(Slingshot)) as Slingshot;
if(slingshot != null) {
```
slingshot.toFireBird = newBodies[0];
slingshot.rigidbodyDamper.AddBodiesToCheck(newBodies);
}

Destroy(gameObject);
}

9. Before you add the script to your new bird, we actually need two blue birds: one that splits and one that does not. Duplicate your bird and name one Bird_Blue_Split and the other Bird_Blue_Normal. To the split bird, add the new script and to the normal bird, add the Bird script.

10. Turn both the birds into prefabs and add the normal bird to the other's list of birds to be split into.

We created the blue bird. This bird splits into multiple birds when the user taps the screen. The effect actually requires two birds that look identical, one that does the splitting and another that is split in two but does nothing special.

It is actually possible to add anything that we want to spawn to the blue bird's list of things to split into. Your challenge here is to create a rainbow bird. This bird can split into different types of birds, not just blue ones. Or, perhaps it is a stone bird that splits into stone blocks. For an extended challenge, create a mystery bird that randomly picks a bird from its list when it splits.

The black bird

Finally, we have the black bird. This bird explodes when the player touches the screen. As with all the birds discussed previously, it will extend the Bird script; inheriting from the red bird makes the black bird’s creation much easier. Let's use these steps to do it:

1. As with the others, this bird is initially created in the same way as the red bird, readjusting the value of Radius on your Circle Collider 2D component for its increased size.
2. Again, we create a new script to extend the Bird script. This time, it is called BlackBird.
3. Do not forget to adjust line four to extend the Bird script and not MonoBehaviour:

   public class BlackBird : Bird {

---------------------------------------------
4. This script has two variables. The first variable is the size of the explosion and the second is its strength:

```csharp
public float radius = 2.5f;
public float power = 25f;
```

5. Once more, we override the `DoSpecial` function, first marking that we did so. Next, we use `Physics2D.OverlapCircleAll` to acquire a list of all the objects that are within the range of the bird's explosion, the 3D version of which is `Physics.OverlapSphere`. Next, we calculate where the explosion is coming from, which is just our bird's position moved down three units. We move it down because explosions that throw debris up are more exciting than the ones that push debris out. The function then loops through the list, skipping any empty slots and those without rigidbodies:

```csharp
protected override void DoSpecial() {
    didSpecial = true;

    Collider2D[] colliders = Physics2D.OverlapCircleAll(transform.position, radius);

    Vector2 explosionPos = new Vector2(transform.position.x, transform.position.y) - (Vector2.up * 3);

    foreach(Collider2D hit in colliders) {
        if(hit == null) continue;
        if(hit.attachedRigidbody != null) {
```

6. If the object exists and has a `Rigidbody` component attached, we need to calculate how the explosion is going to affect this object, simulating the way an explosion's strength is reduced the further away you are from it. First, we save ourselves some typing by grabbing the other object's position. Next, we calculate where it is, relative to the position of the explosion. By dividing the magnitude or the length of the relative position by our `radius` variable, we can figure out how much force to apply to the object that was hit. Finally, we use `AddForceAtPosition` to give the object a kick as if the explosion was in a specific spot. The `ForceMode2D.Impulse` variable is used to apply the force immediately:

```csharp
        Vector3 hitPos = hit.attachedRigidbody.transform.position;
        Vector2 dir = new Vector2(hitPos.x, hitPos.y) - explosionPos;
        float wearoff = 1 - (dir.magnitude / radius);
        Vector2 force = dir.normalized * power * wearoff;
        hit.attachedRigidbody.AddForceAtPosition(force, explosionPos, ForceMode2D.Impulse);
    }
}
7. Finally, the function destroys the exploded bird:
   
   ```csharp
   Destroy(gameObject);
   ```

8. As with the last two, apply your new script to your new bird and turn it into a prefab. You now have four birds to choose from when selecting the slingshot arsenal for each level.

We created our fourth and last bird: the black bird. This bird explodes when the user touches the screen, throwing anything that might be near it into the sky. This can be a fun bird to play around with and is effective for destroying your pig forts.

The black bird from the game we are imitating has the additional ability of a timed explosion after it has hit something. Try creating a timer for our black bird to recreate this effect. You will have to override the `OnCollisionEnter` function to start your timer and use `LateUpdate` to count down. Once your timer runs out, you can just use our `DoSpecial` function to actually cause the explosion.

Now that you know how to cause explosions, we have another challenge: create an explosive crate. You need to extend the `Plank` script to make it, and when enough damage is done to the crate, trigger the explosion. For an additional challenge, instead of making the crate explode, configure it to throw out a few bombs that explode when they hit something.
Level selection

Finally, we need to create our level selection screen. From this scene, we will be able to access and start playing all the levels we created earlier. We will also display the current high scores for each level. A new scene and a single script will serve us well in managing our level selection. Let's use these steps to do it:

1. This last section begins by saving our current scene and pressing Ctrl + N to create a new one; we will name it LevelSelect.

2. For this scene, we need to create a single, short script also named LevelSelect.

3. This script is going to work with the buttons in the GUI to tell players about the high scores and load levels. However, before we can do this, we need to add a line at the very beginning of the script, along with the other using lines—just like the other scripts we have created that need to update the GUI:
   ```csharp
   using UnityEngine.UI;
   ```

4. The first and only variable is a list of all the button text that we want to update, with the high scores for the levels they are associated with:
   ```csharp
   public Text[] buttonText = new Text[0];
   ```

5. The first function is the Awake function. Here, it loops through all the buttons, finds the high score for it, and updates the text to display it. PlayerPrefs.GetInt is the opposite of the SetInt function we used earlier to save the high score:
   ```csharp
   public void Awake() {
       for(int i=0;i<buttonText.Length;i++) {
           int levelScore = PlayerPrefs.GetInt("LevelScore" + (i + 1), 0);
           buttonText[i].text = "Level " + (i + 1) + "\nScore: " + levelScore;
       }
   }
   ```

6. The second and last function for this script is LoadLevel. It will receive a number from the GUI button and use it to load the level that the players want to play:
   ```csharp
   public void LoadLevel(int lvl) {
       Application.LoadLevel(lvl);
   }
   ```
7. Return to Unity and add the script to the **Main Camera** object.

8. Next, we need to create three buttons. Without these, our player will not be able to select a level to play. Make each of them of **200 units square** and position them in a row in the center of the screen. Also, increase the value of **Font Size** to **25**, so that the text is easy to read.

9. Drag each of the buttons' **Text** children to the **Button Texts** list on the **Main Camera** component's **Level Select** script component. The way they are ordered in this list is the order in which they will get their text and high score information changed.

10. Also, each button needs a new **On Click** event. Select **Main Camera** for the object and then navigate to **LevelSelect | LoadLevel (int)** for the function. Then, each button needs a number. The button that has its **Text** child in the **Button Texts** list should have the number **1** since it will display the information for level one. The second has **2**, the third has **3**, and so on. Each button must have the same number as the order in the list, or they will cause a different level to load than what the player is expecting.

11. Finally, open **Build Settings** and add your scenes to the **Scenes in Build** list. Clicking and dragging on the scenes in the list will let you reorder them. Make sure that your **LevelSelect** scene is first and has an index of zero at the right-hand side. The rest of your scenes can appear in whatever order you desire. However, beware as they will be associated with the buttons in the same order.
We have created a level selection screen. It has a list of buttons associated with the levels in our game. When a button is pressed, `Application.LoadLevel` starts that level. We also made use of `PlayerPrefs.GetInt` to retrieve the high scores for each of the levels.

Here, the challenge is to style the GUI to make the screen look great. A logo and a background will help a lot. Additionally, take a look at the `Scrollbar` GUI object if you have more than three levels. This object will let you create a function that offsets the level buttons when the user scrolls through a list of levels that are far greater in size than can be easily seen on the screen.

**Summary**

In this chapter, we learned about physics in Unity and recreated the incredibly popular mobile game, *Angry Birds*. Using Unity's physics system, we are able to make all the levels that we will ever want to play. With this game, we also explored Unity's 2D pipeline for creating great 2D games. Our birds and slingshot are 3D assets, giving us the ability to light and shade them. The pigs and background, however, are 2D images, reducing our lighting options but allowing greater detail in the assets. The 2D images were also crucial in the creation of the parallax scrolling effect of the background. Finally, the blocks that make up the levels appear to be 2D but are actually 3D blocks. We also created a level-selection screen. From here, the player can see their high scores and pick any of the levels that we created.

In the next chapter, we return to the Monkey Ball game we started in the previous chapter. We are going to create and add all of the special effects that finish off a game. We will add the bouncing and popping sound effects that every Monkey Ball game needs. We will also add various particle effects. When the bananas are collected, they will create a small explosion, rather than just disappearing.
Where to buy this book

You can buy Learning Unity Android Game Development from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.