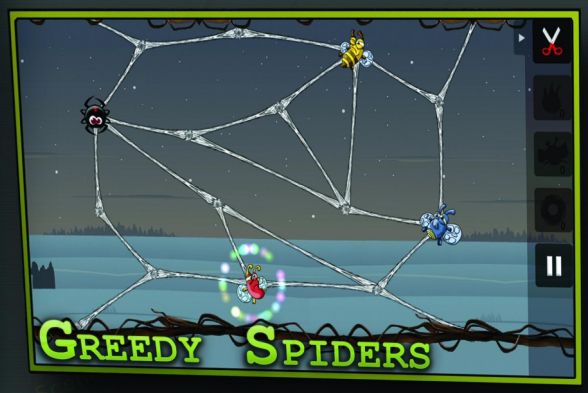
RAW RAW RAW



RAW RAW RAW

Quick answers to common problems

AndEngine for Android Game Development Cookbook: RAW

Over 90 highly effective recipes with real-world examples to get to grips with the powerful capabilities of AndEngine and GLES 2.

Jayme Schroeder
Brian Jamison Broyles

[PACKT] PUBLISHING open source* community experience distilled

AndEngine for Android Game Development Cookbook

RAW Book

Over 90 highly effective recipes with real-world examples to get to the grips with powerful capabilities of AndEngine and GLES.

Jayme Schroeder

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

AndEngine for Android Game Development Cookbook

Current RAW Publication: September 2011

RAW Production Reference: 1260912

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN

www.packtpub.com

Preface

Welcome to AndEngine for Android Game Development Cookbook, the RAW edition. A RAW (Read As we Write) book contains all the material written for the book so far, but available for you right now, before it's finished. As the author writes more, you will be invited to download the new material and continue reading, and learning. Chapters in a RAW book are not "work in progress", they are drafts ready for you to read, use, and learn from. They are not the finished article of course—they are RAW!

This book will show you exactly how to design video game environment environment from scratch and will explain the two primary purposes - providing player with a goal and providing player with enjoyable play experience. You will learn to strive and produce quality gameplay and provide an immersive experience. You will also learn to texture and use audio necessary to produce immersive player experience.

What's in This RAW Book

In this RAW book, you will find these chapters:

In *Chapter 1*, *AndEngine Game Structure*, will help readers understand the life cycle of a game. It will also teach readers to create game and resource managers as well as introduce them to differnet types of textures and graphics.

In *Chapter 2*, *Designing Your Menu*, we will take our first steps into creating a menu. This will include adding buttons, adding music, and button touch events as well as menu navigation. It will also teach readers how to create level titles and chapters.

What's Still to Come?

We mentioned before that the book isn't finished, and here is what we currently plan to include in the remainder of the book:

Chapter 3, Working with Cameras

Chapter 4, Working with Entities

Chapter 5, Scene and Layer Management

Chapter 6, Applications of Physics

Chapter 7, Working with Update Handlers

Chapter 8, Maximizing Performance

Chapter 9, Overview and Example of AndEngine Extensions

Chapter 10, Useful Recipes

Chapter 11, Complete Game with Class-By-Class Description

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: To do so we create a file '`index.html`' in a new '`no_signup_no_party`' folder containing this markup:

A block of code will be set as follows:

```
<!doctype html>
<html>
<head>
   <meta charset="utf-8">
   <title>No signup? No party!</title>
   <link rel="stylesheet" type="text/css" href="http://yui.yahooapis.
com/3.4.1/build/cssreset/cssreset-min.css">
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
::-webkit-validation-bubble-arrow{
     background: #000;
     border: none;
     box-shadow: 0px 0px 10px rgba(33,33,33,0.8);
```

New terms and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: “clicking the **Next** button moves you to the next screen”.

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

What Is a RAW Book?

Buying a Packt RAW book allows you to access Packt books before they’re published. A RAW (Read As we Write) book is an eBook available for immediate download, and containing all the material written for the book so far.

As the author writes more, you are invited to download the new material and continue reading, and learning. Chapters in a RAW book are not “work in progress”, they are drafts ready for you to read, use, and learn from. They are not the finished article of course—they are RAW! With a RAW book, you get immediate access, and the opportunity to participate in the development of the book, making sure that your voice is heard to get the kind of book that you want.

Is a RAW Book a Proper Book?

Yes, but it’s just not all there yet! RAW chapters will be released as soon as we are happy for them to go into your book—we want you to have material that you can read and use straightaway. However, they will not have been through the full editorial process yet. You are receiving RAW content, available as soon as it written. If you find errors or mistakes in the book, or you think there are things that could be done better, you can contact us and we will make sure to get these things right before the final versio0n is published.

When Do Chapters Become Available?

As soon as a chapter has been written and we are happy for it go into the RAW book, the new chapter will be added into the RAW eBook in your account. You will be notified that another chapter has become available and be invited to download it from your account. eBooks are licensed to you only; however, you are entitled to download them as often as you like and on as many different computers as you wish.

How Do I Know When New Chapters Are Released?

When new chapters are released all RAW customers will be notified by email with instructions on how to download their new eBook. Packt will also update the book's page on its website with a list of the available chapters.

Where Do I Get the Book From?

You download your RAW book much in the same way as any Packt eBook. In the download area of your Packt account, you will have a link to download the RAW book.

What Happens If I Have Problems with My RAW Book?

You are a Packt customer and as such, will be able to contact our dedicated Customer Service team. Therefore, if you experience any problems opening or downloading your RAW book, contact `service@packtpub.com` and they will reply to you quickly and courteously as they would to any Packt customer.

Is There Source Code Available During the RAW Phase?

Any source code for the RAW book can be downloaded from the **Support** page of our website (`http://www.packtpub.com/support`). Simply select the book from the list.

How Do I Post Feedback and Errata for a RAW Title?

If you find mistakes in this book, or things that you can think can be done better, let us know. You can contact us directly at `rawfeedback@packtpub.com` to discuss any concerns you may have with the book.

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of. To send us general feedback, simply drop an email to `rawfeedback@packtpub.com`, making sure to mention the book title in the subject of your message.

1

AndEngine Game Structure

In this chapter, we're going to take a look at the main components needed for structuring a game in AndEngine. The topics include:

- Understanding the life-cycle
- The engine object
- What are singletons?
- What are object factories?
- Creating the game manager
- Introducing sounds and music
- Different types of textures
- Applying options to our textures
- Introducing graphic-independent resolutions
- Introducing font resources
- Creating the resource manager
- Designing your splash screen
- Saving and loading game data

Introduction

One of the attractive features of AndEngine is the ease of creating games. The possibility of designing and coding a game in a matter of weeks after first looking into AndEngine is not too farfetched, but that's not to say it will be a perfect game. The coding process can be a tedious task when we do not understand how the engine works. It is a good idea to understand the main building blocks of AndEngine and game structure in order to create precise, organized and expandable projects.

In this chapter, we're going to go over a few of the most necessary components of AndEngine and general game programming. We're going to take a look at some classes that will aid us in quickly and efficiently create a foundation for all sorts of games. Additionally, we'll cover some of the differences between resources and object types which are extremely useful for creating diverse games.

It is encouraged to keep tabs on this chapter as reference if needed.

Understanding the life-cycle

It is important to understand the order of operations when it comes to the initialization of your game. The basic needs for a game include creating the engine, loading the games resources, and setting up the initial screen and settings. This is all it takes in order to create the foundation for an AndEngine game. However, if we plan on more diversity within our games it is wise to get to know the full life-cycle included in AndEngine.

Getting ready

Refer to the folder `ClassCollection01` (`PacktActivity.java`) for the working code for this topic.

How to do it...

Create a new AndEngine project with the following activity class:

```
public class PacktActivity extends BaseGameActivity {

    //=====================================================
    // CONSTANTS
    //=====================================================
    public static final int WIDTH = 800;
    public static final int HEIGHT = 480;

    //=====================================================
    // VARIABLES
    //=====================================================
    private Scene mScene;
    private Camera mCamera;

    //=====================================================
    // CREATE ENGINE OPTIONS
    //=====================================================
    @Override
```

```
    public EngineOptions onCreateEngineOptions() {
            // Create our game's camera (view)
            mCamera = new Camera(0, 0, WIDTH, HEIGHT);

// Setup our engine options. Including resolution policy, screen
orientation and full screen settings.
            EngineOptions engineOptions = new EngineOptions(true,
ScreenOrientation.LANDSCAPE_FIXED, new FillResolutionPolicy(),
mCamera);

            // Allow our engine to play sound and music
            engineOptions.getAudioOptions().setNeedsMusic(true);
            engineOptions.getAudioOptions().setNeedsSound(true);

            // Do not allow our game to sleep while it's running
            engineOptions.setWakeLockOptions(WakeLockOptions.SCREEN_ON);

            return engineOptions;
    }

    //=====================================================
    // CREATE RESOURCES
    //=====================================================
    @Override
    public void onCreateResources(
                OnCreateResourcesCallback pOnCreateResourcesCallback)
                throws Exception {

            pOnCreateResourcesCallback.onCreateResourcesFinished();
    }

    //=====================================================
    // CREATE SCENE
    //=====================================================
    @Override
    public void onCreateScene(OnCreateSceneCallback
pOnCreateSceneCallback)
                throws Exception {
            mScene = new Scene();


            pOnCreateSceneCallback.onCreateSceneFinished(mScene);
    }
```

```
    //=====================================================
    // POPULATE SCENE
    //=====================================================
    @Override
    public void onPopulateScene(Scene pScene,
                  OnPopulateSceneCallback pOnPopulateSceneCallback)
throws Exception {

          pOnPopulateSceneCallback.onPopulateSceneFinished();
    }}
```

How it works...

This class is the foundation for any AndEngine game. We've setup the project with all of the necessary methods and variables needed in order to start writing any type of game you desire. Keep this activity handy as we will be referencing its methods throughout various areas of the current and future chapters.

Below, we will go over the life-cycle methods in the order they are called from the startup of an activity to the time it is terminated.

The life-cycle calls during launch

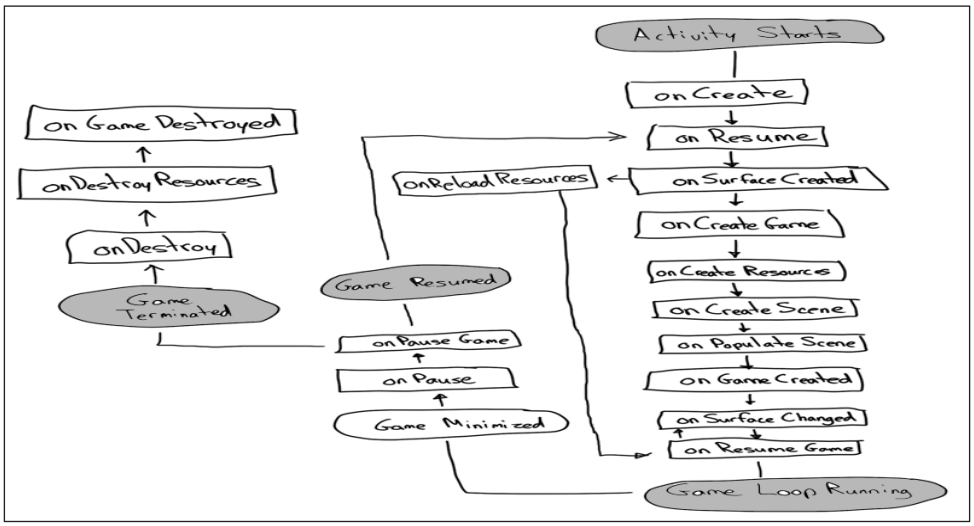
- **onCreate -** You should already be familiar with this method if you've worked with the Android SDK before. It's the entry point to any android application by default. In AndEngine development this method simply calls the `onCreateEngineOptions` method in your `BaseGameActivity` then applies the returned options to the game engine.
- **onResume** - Another Android SDK native method. Here we simply acquire the wake lock settings and continue on to call the `onResume` method for the engine's `RenderSurfaceView` object.
- **onSurfaceCreated -** `onSurfaceCreated` will either call `onCreateGame` for initial start-up or register a `boolean` variable true for resource reloading if the activity had previously been deployed.
- **onReloadResources –** Reload our game resources if our application is brought back into focus from minimization. *This method is not called on the initial execution of an application*.
- **onCreateGame** - `onCreateGame` is in place to handle the order of execution of the next three callbacks in the AndEngine life-cycle.
- **onCreateResources** - This method should be used to load the required assets for your initial scene if following the single activity/multiple scene approach. Otherwise you should setup all resources needed for the activity in this method.

- **onCreateScene** - Here, we handle the initialization of the scene objects you may be using in your game. You aren't restricted to only constructing your scenes here and there is no penalty, but for organization's sake we will be in this book.
- **onPopulateScene** - In the `onPopuplateScene` method of the life-cycle we are just about finished setting up the scene, though the rest of the life-cycle calls will be taken care of automatically. Here, we will populate our scene, generally applying the HUD (heads-up-display), background, and other various entities. This is where your games first scene will be setup and eventually, seen by the player.
- **onGameCreated** - Signals that the `onCreateGame` sequence has finished, reloading resources if necessary, otherwise doing nothing. Reloading resources depends on the Boolean variable briefly mentioned in the `onSurfaceCreated` method five life-cycle calls back.
- **onSurfaceChanged** - This method is called every time your applications orientation changes from landscape to portrait mode or vice-versa.
- **onResumeGame** - Here we have the final method call which takes place during an activity's start-up cycle. If your activity reaches this point without any problems, the engine's `start()` method is called, bringing the game to life.

The life-cycle calls during minimization/termination

- **onPause** - The first method call when an activity is minimized or finished. This is the native android pause method which calls the pause method for the `RenderSurfaceView` objects and reverts the wake lock settings applied by the game engine.
- **onPauseGame -** Secondly, the AndEngine implementation of onPause which simply calls the `stop()` method on the engine, causing all of the engine's update handlers to halt along with the update thread.
- **onDestroy -** In the `onDestroy` method AndEngine deals with the "destruction" of objects tied to all the main managers which deal with textures in one way or another. These managers include the vertex buffer object manager, the font manager, the shader program manager and finally the texture manager.
- **onDestroyResources –** This method name may be a little misleading since we've already unloaded the majority of resources in `onDestroy`. What this method really does is releases all of the sounds used in the activity, calling the `releaseAll()` methods in the sound manager and music manager.
- **onGameDestroyed** - Finally we reach the last method call required during a full AndEngine life-cycle. Not a whole lot of action takes place in this method. AndEngine simply sets a Boolean variable used in the engine to false, which states that the activity is no longer running.

Below we can see what the life-cycle looks like in action whether the game is created, minimized, or destroyed:



Due to the asynchronous nature of the AndEngine life-cycle, it is possible for some methods to be executed multiple times during a single startup instance. The occurrence of these events varies between devices. Take this into account if you decide to include certain functionalities into life-cycle methods.

There's more...

BaseGameActivity Class

BaseGameActivity is the AndEngine activity class which handles most of the life-cycle calls for us. Though as a developer we are required to initialize some objects used by the game engine on start-up. The life-cycle methods we are responsible for include `onCreateEngineOptions` (Technically, this is not part of the life-cycle but we must return the engine options to `onCreateGame`), `onCreateResources`, `onCreateScene` and `onPopulateScene` and are executed in that order. The `BaseGameActivity` class requires us to execute callbacks when we are finished with the life-cycle method in order for the engine to continue on to the next life-cycle method. Alternatively, if your game doesn't necessarily require control over the callbacks, you can use a **SimpleBaseGameActivity** which handles the callbacks for us.

LayoutGameActivity Class

There is also one other type of activity which AndEngine provides us with. The same life-cycle applies, but there is one main difference between this class and the `BaseGameActivity` class. This activity is called a **LayoutGameActivity**, whose purpose is to allow our activity to be applied to an ordinary android application as a view. The main benefit of using this type of activity is the fact that you can mix native android views (text boxes, on-screen keyboard, buttons, etc.) with the AndEngine game surface which is otherwise not possible. The most likely scenario for mixing AndEngine with native android views would have to be applying Ads to your game for some revenue generation.

The engine object

Before we start programming our game, it is a good idea to come up with the performance needs of the game. AndEngine includes a few different types of engine's we can choose to use, each with their own benefits. The benefits, of course, depend on the type of game we plan to create.

Getting ready

Locate the `onCreateEngineOptions()` method in the previous `PacktActivity.java` class.

How to do it...

If we override the `onCreateEngine()` method of our activity class, we can create the specific type of engine by simply changing the '**new**' object type:

```
@Override
public Engine onCreateEngine(EngineOptions engineOptions){
return new FixedStepEngine(engineOptions, 60);
}
```

How it works...

Choosing our engine

- **Engine** – First and foremost, we have the ordinary engine class. This engine is not ideal for game development as it has absolutely no limitations on our games in regards to frames per second. On two separate devices it is very likely that you will notice differences in the speed of the game. One way to think of this is if two separate devices watching a video which was started at the same time, the faster device is likely to finish the video first rather than them finishing at the same time.

- **FixedStepEngine** – The second type of engine we have is the fixed step engine. This is the ideal engine used in game development as it forces the game loop to update at a constant speed regardless of the device. This is done by updating the game based on time passed rather than device speed. The contents of this book's recipes will be based on the use of this engine.
- **LimitedFPSEngine –** The limited FPS engine allows us to set a preferred step count for the engine. The difference between this engine and fixed step engine is that if the time since the last update is longer than the preferred step count, AndEngine will call additional updates to the game loop in order to compensate.
- **SingleSceneSplitScreenEngine –** The single scene split screen engine allows us to create applications which have two separate camera views on the same scene. One of the official AndEngine examples utilizes this engine to show two separate views on the same sprite. One of the views is zoomed in on the sprite while the second shows a zoomed out view of the entire scene. The split screen engines do not have fixed step or limited step options.
- **DoubleSceneSplitScreenEngine –** Finally, we have the double scene split screen engine. This engine is the same idea as the previous engine, except we are able to set a separate scene for each side of the screen. This engine is likely to be used to allow a multiplayer component where each player controls half of the devices display.

What are singletons?

When it comes to game development, it is important to keep your projects organized and efficient. One of the most practical scenarios is to implement **design patterns**. The singleton is a design pattern which allows us to access specific methods for our game on a global level.

Getting ready

Refer to the folder ClassCollection01 (`Singleton.java`) for the working code for this topic.

How to do it...

Import the following code:

```
public class Singleton {

/* Construct a singleton object within its own class, ensuring
that we'll only have one instance across the entire game */
private static final Singleton INSTANCE = new Singleton();

// Empty Constructor
Singleton(){
}
```

```
/* Since the singleton is a private object, we will be using a
"getter" in order to obtain the singleton */
public static Singleton getInstance(){
    return INSTANCE;
}
}
```

How it works...

The above implementation of a design pattern includes the very basic necessities in order to create a design pattern of the singleton type. From here, we can add methods to perform various tasks throughout our game which we can call from anywhere in our project.

Singletons in game development are typically used for object management of some form. These types of singletons are generally used to make method calls on specific types of objects or make general modifications of game data. A resource manager would be one example of a singleton which handles specific objects (textures, sounds, fonts, and more). On the other end, we could have a game manager which deals with a more broad set of methods, such as changing text on a screen and resetting levels.

A singleton's constructor is usually pretty bare. The reason for this is the singleton's constructor is called a maximum of one time throughout the entire life-cycle of an application. Keep this in mind when deciding to use a singleton for any sort of initialization.

What are object factories?

Another type of design pattern we tend to focus on in game development is the object factory.

Getting ready

Refer to the folder ClassCollection01 (`ObjectFactory.java`) for the working code for this topic.

How to do it...

Refer to the project ObjectFactory in the code bundle

How it works...

A factory is another design pattern that is used regularly in game development. The purpose of a factory class is to create a subtype object from a base object, usually incorporated with a set of parameters in order to setup the objects properties. This type of design pattern allows for ease of object creation while helping to keep object creation organized. One scenario where a factory might be used would be creating various enemies to be spawned in our game.

This particular factory class consists of two methods which allow a user to create subtype objects of types `LargeObject` and `SmallObject`. To create a new `LargeObject` via the `ObjectFactory`, the code would be as simple as `BaseObject object = ObjectFactory.createLargeObject(x, y);` where x and y would be the position of the object.

An alternative approach for setting up factory classes is to use a single method with a switch statement based on object id's which would be attributed to each object-type.

Creating the game manager

The game manager is a singleton which is used for general modification to the game-state.

Getting ready

Refer to the project ClassCollection01 (`GameManager.java`) for the working code for this topic.

How to do it...

Import the following code:

```
public class GameManager {

    //=====================================================
    // CONSTANTS
    //=====================================================
    private static final GameManager INSTANCE       = new
GameManager();

    //=====================================================
    // VARIABLES
    //=====================================================
```

```
    private int mCurrentScore;
    private int mBirdCount;
    private int mEnemyCount;

    private Text mScoreText;

    //=====================================================
    // CONSTRUCTOR
    //=====================================================
    GameManager(){
        // This constructor is of no use to us
    }

    //=====================================================
    // GETTERS & SETTERS
    //=====================================================
    public static GameManager getInstance(){
        return INSTANCE;
    }

    public void incrementScore(int incrementBy){
        // incrementBy would be determined by the object or enemy
destroyed
        mCurrentScore += incrementBy;

        // Convert the mCurrentScore value to a string and apply it
to our text object
        mScoreText.setText(String.valueOf(mCurrentScore));
    }

    // Any time a bird is launched, this method will be called
    public void decrementBirdCount(){
        mBirdCount -= 1;
    }

    // Any time an enemy is hit, this method will be called
    public void decrementEnemyCount(){
        mEnemyCount -= 1;
    }

    //=====================================================
    // INITIALIZATION
    //=====================================================
```

```
    // This method would be called when entering a level
    public void initializeGameManager(final Font font,final HUD
hud,final  Engine engine){
          /* Setup a simple text in the top left corner of the screen
which will be applied
             to the game's heads-up-display */
          mScoreText = new Text(15, 15, font, "0", 5, engine.
getVertexBufferObjectManager());
          hud.attachChild(mScoreText);

          // Reset game variables when a level is loaded
          resetGame();
    }

    //=====================================================
    // RESET GAME
    //=====================================================
    public void resetGame(){
          // We're simply reverting our counts back to their original
values
          this.mCurrentScore = 0;
          this.mBirdCount = 3;
          this.mEnemyCount = 5;
    }
}
```

How it works...

The game manager is bound to have different tasks depending on the type of game you are creating. We're going to create a game manager based on a game similar to Angry Birds. The tasks involved for a game manager in this case wouldn't be too demanding, but it will give you an understanding of the use of a game manager.

Game manager tasks

1. Keeping track of current score
2. Keeping track of available birds
3. Keeping track of available enemies
4. Resetting game data

The first three tasks listed will be stored in integer variables; each provided their own 'setter' methods. The fourth task will handle resetting the game data simply reverting all of our scores and bird/enemy counts back to default.

That is all it really takes when it comes to setting up a game manager. Obviously the larger the game, the more complex the game manager will be. This should give you an understanding of how much more organized you can be by bundling variables and methods related to the game in general.

See also

- What are singletons?

Introducing sounds and music

Including sounds and music into our games with the help of AndEngine is a simple task.

Getting ready

Refer to the project ClassCollection01 (`SoundsAndMusic.java`) for the working code for this topic.

How to do it...

Refer to the project SoundsAndMusic in the code bundle

How it works...

Unless the game being developed is highly focused on sounds and music, AndEngine should have no problem handling your needs. Because of the simplicity of AndEngine's sound and music objects, we are limited in some ways in terms of applying modifiers to the sounds. However AndEngine does let us play around with the sounds a bit. Some methods included for adjusting the properties of sound/music objects are listed below:

1. Playing and pausing sound files with `play()` and `pause()` methods
2. Controlling the frequency at which a sound file plays with `setRate()` method
3. `setLooping()` and `setLoopCount()`
4. `setVolume()` – With this method, assuming the device playing the sounds allows for stereo, we can actually adjust the left/right volume of the sound being played

Additionally, there are methods included in the sound objects that allow the developer to handle unloading of the resources, but keep in mind that AndEngine makes use of the `SoundPool` class which handles the loading/unloading of audio-related objects for the most part. In future chapters, we will take a deeper look into handling the loading and unloading of audio-related objects as we transition between different scenes in our projects.

There's more...

The resources used in game development play a big role in its success on the mobile platform, much like in PC or console games. It is usually emphasized that the graphics should look very nice and polished, which is true, but many indie game developers forget that attractive sounds go hand-in-hand with attractive graphics.

Free sound resources

For those of us who are not interested in creating our own sound files to add sound effects and music to our games, there are plenty of resources out there which are free to use. One of my favorite free sound effect collections can be found at `http://www.soundjay.com/`. Keep in mind, they do require you give all credits to them for sound files used if you plan to release a product with their resources.

See also

- Creating the resource manager

Different types of textures

Creating sprites and textures is relatively easy in AndEngine, but are we doing more harm than good? Let's go over textures in AndEngine and find out how they really work.

Getting ready

For this topic, we can test the various ways of creating textures via use of the `PacktActivity.java` class. Create a few sample images named `rectangle_one.png`, `rectangle_two.png`, and `rectangle_three.png` and include them in the `assets` folder of a testing project.

How it works...

In AndEngine development, there are two main components we will be using in order to apply graphics (sprites) to our projects. These objects can be thought of much like a world map. The first component is known as a **BitmapTextureAtlas**, which is basically just a layout we will use to store sub-textures. These sub-textures are called **TextureRegions**. You can think of these two components similar to a real-world map. The texture atlas would be the map itself if you removed all cities. You'd be left with a blank page, which is what a texture atlas is, initially. Once we've created our texture atlas, we can create and apply our texture regions to our map to populate it; these texture regions can be thought of as cities, if you will.

BitmapTextureAtlas

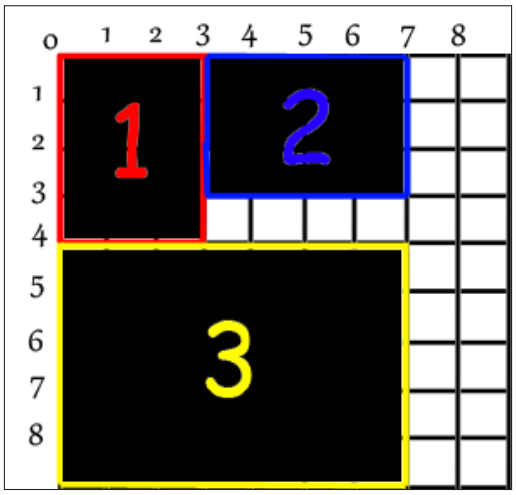
A texture region must be supplied width and height values on creation in order to set its size. Additional parameters for a texture atlas include texture format and texture options, which tell the atlas how do handle the quality of the texture regions.

The purpose of a texture atlas is to allow bundling of many smaller textures in order to reduce the amount of draw calls OpenGL must make to the canvas. For example, if you had 5 separate textures opposed to one texture atlas, you are performing 5 times the tasks needed in order to display those images on a device. It is important to remember that in game development, overhead can build up quickly. Cut out any unnecessary tasks where possible.

Creating a texture atlas only takes a few lines of code at most.

```
BitmapTextureAtlas mBitmapTextureAtlas = new
BitmapTextureAtlas(mEngine.getTextureManager(), 1024, 1024);
```

With this line of code, we create a texture atlas named texture. First parameter we pass is our engine's texture manager which we will use to load the texture atlas into memory. The second two parameters are max width and height of the texture atlas, respectively. This is all it takes in order to create a texture atlas. Once we've created a texture atlas and added texture regions to it, we'll simply call `mBitmapTextureAtlas.load()` in order to load the textures to our engine.



We will use this image to represent a texture atlas for the next few topics. The grid is a basic representation of the available area in our `BitmapTextureAtlas` object. We can read this texture atlas at 9x9 pixels in dimension.

Android devices are fairly restricted in terms of texture sizes. It is safe to create texture atlases up to 1024x1024 pixels as most Android 2.2+ devices can safely create textures of that size and lower. If you plan to create textures larger than that, keep in mind you may be restricting the number of devices who can successfully run your game.

TextureRegion

A texture region holds more specific data about the various textures within the texture atlas. It contains its coordinates in the texture atlas, the size of the texture and the graphic it will contain for the most part. With this knowledge, we can create a sprite simply by passing our texture region to it.

Looking back at the `BitmapTextureAtlas` image, we can see 3 rectangles numbered from 1 to 3. These areas each represent a texture region. The first rectangle is 3x4 pixels in dimension; the second is 4x3 pixels in dimension, and the third is7x5 pixels in dimension. The `BitmapTextureAtlas` class requires us to specify the coordinates we apply the texture region to. In the following code, we will create a texture atlas based on the previous image.

```
// Create texture region number 1 at position 0, 0
TextureRegion mTextureRegionOne =
BitmapTextureAtlasTextureRegionFactory.createFromAsset(mEngine.
getTextureManager(), mBitmapTextureAtlas, context, "rectangle_one.
png", 0, 0);
// Create texture region number 2 at position 4, 0
TextureRegion mTextureRegionTwo =
BitmapTextureAtlasTextureRegionFactory.createFromAsset(mBitmapTextureA
tlas, context, "rectangle_two.png", 4, 0);
// Create texture region number 3 at position 0, 5
TextureRegion mTextureRegionThree =
BitmapTextureAtlasTextureRegionFactory.createFromAsset(mBitmapTextureA
tlas, context, "rectangle_three.png", 0, 5);
```

You might notice that in the `BitmapTextureAtlas.png` image, texture region number 2 starts at 3 pixels but in code we are applying the texture region to the atlas 4 pixels in width. The purpose for this is to apply **texture source padding** to the texture region. What this does is mitigate the chance that our sprites will appear with texture bleeding (overlapping of texture regions).

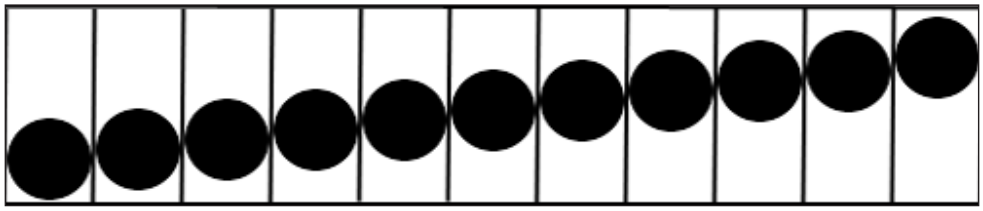
There are different types of routes we can take in order to create our texture regions. We can create texture regions from android resources, from images in our `assets` folder and additionally, we can create tiled texture regions which allow use of sprite sheets for animated sprites.

Careful!

Naturally, when applying texture regions to a texture atlas, we must manually define its position on the atlas. It is good practice to keep your texture regions as close to each other as possible without exceeding the bounds of your texture atlas in order to limit the number of texture atlases needed.

TiledTextureRegion

A tiled texture region is essentially the same object as a normal texture region. The difference between the two is that a tiled texture region allows you pass to it a single image file and create a sprite sheet out of it. This is done by specifying the number of columns and rows within your sprite sheet. From there, AndEngine will automatically divide the tiled texture region into smaller segments. This will allow you to navigate through each segment via the `TiledTextureRegion` object. This is how the tiled texture region will appear to create a sprite with animation.



A real sprite sheet should not have outlines around each column or row. They are in place to give you a visualization of how a sprite sheet is divided up.

Let's assume this image is 165 pixels wide by 50 pixels high. Since we have 11 individual columns and a single row, we could create the tiled texture region like so:

```
TiledTextureRegion mTiledTextureRegion =
BitmapTextureAtlasTextureRegionFactory.createTiledFromAsset(mBitmapTex
tureAtlas, context,"sprite_sheet.png",11,1);
```

What this code does is tell AndEngine to divide the `sprite_sheet.png` image into 11 individual segments, each 15 pixels wide (165 pixel wide image divided by 11 segments). We can now use this tiled texture region object to instantiate a sprite with animation.

There's more...

BuildableBitmapTextureAtlas

The buildable bitmap texture atlas is a great way to implement texture regions into your atlas without having to manually define its position on the atlas. The purpose of this type of texture atlas is to automatically package its texture regions onto the atlas by applying them to the most convenient coordinates. I find this approach to creating textures the easiest and most efficient as it can become time-consuming (and error-prone) when building large games with many texture atlases. In addition to the buildable bitmap texture atlas being automated, it also allows for the developer to define padding between texture regions to decrease texture bleeding.

The creation of a `BuildableBitmapTextureAtlas` is no different than creating a `BitmapTextureAtlas`, aside from the naming. However, the buildable texture atlas requires a few extra lines of code before we can load the atlas into memory.

```
try {
    mBitmapTextureAtlas.build(new BlackPawnTextureAtlasBuilder<IBitmap
TextureAtlasSource, BitmapTextureAtlas>(0, 1, 4));
    mBitmapTextureAtlas.load();
} catch (TextureAtlasBuilderException e) {
    Debug.e(e);
}
```

In order to allow us to use the buildable texture atlas, we must first tell AndEngine to build it. AndEngine builds these texture atlases automatically through the use of a class called **BlackPawnTextureAtlasBuilder**. The parameters in the `BlackPawnTextureAtlasBuilder` are in place to allow us to add spacing or padding to the texture atlas border as well as the source (in between each texture region). The above implementation is what I use in terms of spacing and padding as it gives sufficient space to mitigate chances of texture bleeding. Additionally, the build method requires us to surround the code with a try/catch clause.

Compressed textures

Additional to the more common image types (`.bmp`, `.jpeg` and `.png`), AndEngine also has built-in support for PVR and ETC1 compressed textures. The main benefit to using compressed textures is the impact it has on reducing loading times and possibly increasing frame rates during gameplay. On that note, there are also down-sides to compressed textures. ETC1, for example doesn't allow for an alpha channel to be used in its textures. Compressed textures may also cause a noticeable loss of quality in your textures. The use of these types of textures should be relevant to the significance of the object being represented by the compressed texture. You most likely wouldn't want to base your entire games texture format on compressed textures, but for large quantities of subtle images, using compressed textures can add noticeable performance to your game.

See also

- Creating the resource manager
- Applying options to our textures

Applying options to our textures

We've discussed the different types of textures AndEngine provides; now let's go over the options we can supply our textures with. The contents in this topic tend to have noticeable effects on quality and performance of our games.

Getting ready

In the previous topic, we had setup for testing various texture atlases and texture regions. We can continue to test the texture options through the use of that class.

How to do it...

Applying texture options and formats is simple to do. All we need is to add a parameter or two to the `BitmapTextureAtlas` constructor depending on the intended quality of the sprites using the texture atlas.

```
BitmapTextureAtlas mBitmapTextureAtlas      = new
BitmapTextureAtlas(mEngine.getTextureManager(), 1024, 1024,
BitmapTextureFormat.RGB_565, TextureOptions.BILINEAR);
```

You can see here that the constructor is the same aside from the two additional parameters. From here on out, all texture regions placed on this specific texture atlas will have this format and option applied to it.

How it works...

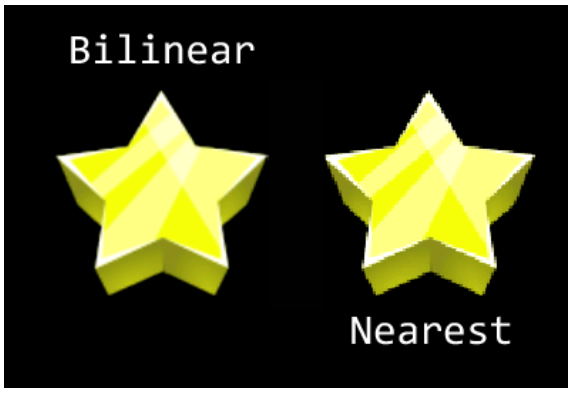
AndEngine allows us to apply texture options and formats to our texture atlases. The various combinations of options and formats applied will affect the overall quality and performance of the sprites while displayed on-screen.

AndEngine texture options

Base Texture Options

- **Nearest –** The 'nearest' texture option is applied by default to texture atlases unless manually configured. This is the fastest performing texture option applied to sprites, but also the poorest quality. The 'nearest' option simply means that the texture will apply blending of pixels, grabbing only the nearest (1 texel) neighbor in order to attempt smoothing of scaled pixels. The end-result usually appears more pixelated or blocky.

Bilinear – The second main texture filtering option in AndEngine is called bilinear texture filtering. This approach adds a bigger hit to performance but the quality of scaled sprites will increase. Bilinear filtering, opposed to nearest filtering, obtains the nearest four texels in order to gather a smoother blend per pixel.



These two images are rendered at the highest bitmap format. The difference between the nearest and bilinear filtering is very clear in this case. The bilinear star has almost no jagged edges and the colors are very smooth. On the right-hand side, we've got a star rendered with 'nearest' filtering. The quality level suffers as jagged edges are more apparent and if you look closely, the colors aren't as smooth.

Additional Texture Options

- **Repeating** – Repeating texture option allows the sprite to 'repeat' the texture assuming the width and height of the image has been exceeded by the size of the sprite. In games, terrain is usually created by using a repeating texture and stretching the size of the sprite rather than creating many separate sprites to cover the ground.

Repeating texture example

Let's take a look at how to work with repeating textures:

```
            /* Create our repeating texture. Repeating textures require
width/height which are a power of two */
```

```
            BuildableBitmapTextureAtlas texture = new BuildableBitm
apTextureAtlas(engine.getTextureManager(), 32, 32, TextureOptions.
REPEATING_BILINEAR);

            // Create our texture region - nothing new here
            mSquareTextureRegion =
BitmapTextureAtlasTextureRegionFactory.createFromAsset(texture,
context, "square.png");

            try {
                  // Repeating textures should not have padding
                  texture.build(new BlackPawnTextureAtlasBuilder<IBitma
pTextureAtlasSource, BitmapTextureAtlas>(0, 0, 0));
                  texture.load();

            } catch (TextureAtlasBuilderException e) {
                  Debug.e(e);
            }
```

The above code is based on a square image which is 32 by 32 pixels in dimension. Two things to keep in mind when creating repeating textures:

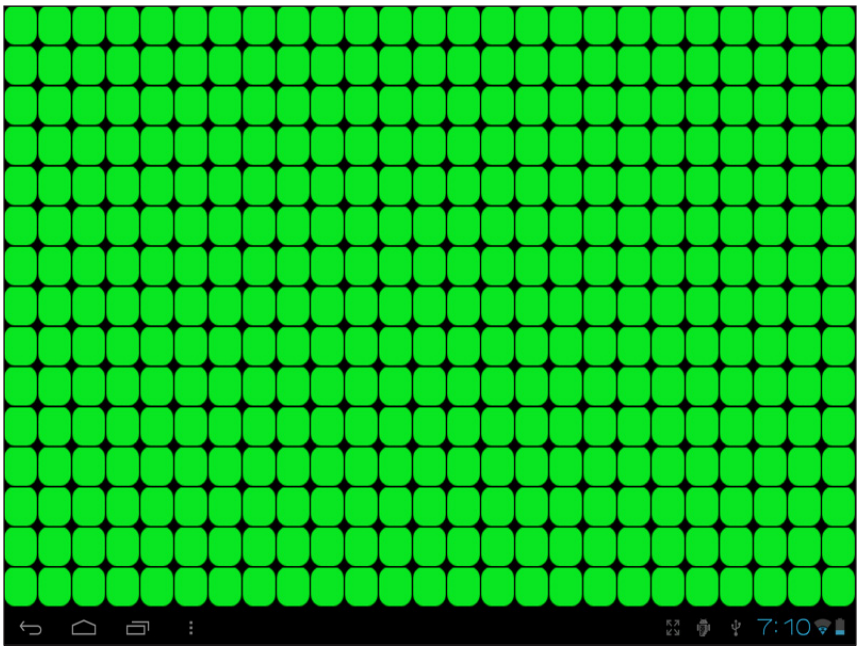
1. *Texture atlases using the repeating texture option format require power of two dimensions (2, 4, 8, 16, etc.).*
2. *If using a buildable texture atlas, do not apply padding or spacing during the build() method as it will be taken into account in the repeating of the texture.*

Next, we have to create a sprite which uses this repeated texture:

```
/* Increase the texture region's size, allowing repeating textures to
stretch up to 800x480 */
ResourceManager.getInstance().mSquareTextureRegion.setTextureSize(800,
480);
// Create a sprite which stretches across the full screen
Sprite sprite = new Sprite(0, 0, 800, 480, ResourceManager.
getInstance().mSquareTextureRegion, mEngine.
getVertexBufferObjectManager());
```

What we're doing here is increasing the texture region's size to 800x480 pixels in dimension. This doesn't alter the size of the image while the repeating option is applied to a texture, rather it allows the image to be repeated up to 800x480 pixels. Additionally, we're creating a sprite which overlays the entire screen.

Here's the outcome taken from a device screenshot:



- **Pre-multiply Alpha** – Lastly, we have the option to add the pre-multiply alpha texture option to our textures. What this option does is multiply each of the RGB values by the specified alpha channel and then apply the alpha channel in the end. The main purpose of this option is to allow us to modify the opacity of the colors without loss of color. Keep in mind, modifying the alpha value of a sprite directly may introduce unwanted effects when using pre-multiplied alpha values. *Sprites will most likely not appear fully transparent when this option is applied to sprites with an alpha value of 0.*

When applying texture options to our texture atlases, we can choose either nearest or bilinear texture filtering options. On top of these texture filtering options, we can include either the repeating option, the pre-multiply alpha option or even both.

There's more...

Aside from texture options, AndEngine also allows us to set the texture format of each of our texture atlases. Texture formats, similar to texture options, are often decided upon depending on its purpose. The format of a texture can greatly affect both performance and quality of an image even more noticeably than the texture options. Texture formats allow for us to choose the available color ranges of the RGB values in a texture atlas. Depending on the texture format being used, we may also allow or disallow sprite from having any alpha value which affects the transparency of the textures.

The texture format naming conventions are not very complicated. All formats have a name similar to '`RGBA_8888`' where the left-hand side of the underscore refers to the color or alpha channels available to the texture. The right-hand side of the underscore refers to the bits available to each of the color channels.

Texture formats

- **RGBA_8888** – Allow the texture to use red, green, blue and alpha channels, assigned 8 bits each. Since we have 4 channels each assigned 8 bits (4x8), we're left with a 32-bit texture format. This is the slowest texture format of the four.
- **RGBA_4444** – Allow the texture to use red, green, blue and alpha channels, assigned 4 bits each. Following the same rule as the previous format, we're left with a 16-bit texture format. You will notice an improvement with this format over `RGBA_8888` as we're only saving half as much information as the 32-bit format. The quality will suffer noticeably.

In this image we compare the difference between two texture formats. The stars are both rendered with the default texture option (nearest), which has nothing to do with the RGBA format of the image. What we're more interested here is the color quality of the two stars. The left star is rendered with full 32-bit color capabilities, the right with 16-bit. The difference between the two stars is rather apparent.

- **RGB_565** – Another 16-bit texture format, though this one does not include an alpha channel Textures using this texture format will not allow for transparency. Due to the lack of transparency, the need for this format is limited, but it still valuable. One example of this texture format being used would be to display a full-screen image such as a background. Backgrounds don't require transparency so it is wise to keep this format in mind when introducing a background. The performance saved is fairly noticeable.

The `RGB_565` format color quality is more or less the same as you would expect from the `RGBA_4444` star image above.

- **A_8** – Finally we have the last texture format, which is an 8-bit alpha channel (does not support colors). Another limited-use format; the `A_8` format is generally used as an alpha mask (overlay) for sprites who do have color. One example of a use for this format is screen-fading in or out by simply overlaying a sprite with this texture, then altering the transparency as time passes.

When creating your texture atlases, it is a good idea to think about which types of sprites will use which type of texture regions and pack them into texture atlases accordingly. For more important sprites, we'll most likely want to use `RGBA_8888` texture format since these sprites will be the main focus of our games. These objects might include the foreground sprites, main character sprites, or anything on the screen that would be more visually prominent. Backgrounds underlay the entire surface area of the device, so we most likely have no use for transparency. We will use `RGB_565` for these sprites in order to remove the alpha channel, which will help improve performance. Finally we have objects which might not be very colorful, might be small, or simply may not need as much visual appeal. We can use the texture format `RGBA_4444` for these types of sprites in order to cut the memory needed for these textures in half.

See also

- Understanding the life-cycle
- Different types of textures

Introducing graphic-independent resolutions

Graphics are obviously very important when it comes to programming for games. This includes both the drawing stage performed by the artist as well as the implementation of the graphic into the game by the programmer. We've already gone over texture options, texture formats, and the various ways of getting the image itself from the `assets` folder into our devices memory. However, the process of importing graphics is rather trivial compared to designing the artwork itself.

Getting ready

Apply the following code to the `onCreateEngineOptions()` method in the `PacktActivity.java` class.

How to do it...

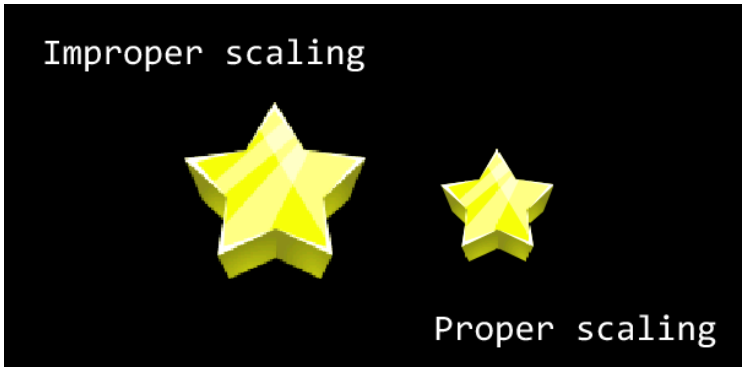
```
// Get the window manager from android system services
WindowManager windowManager = (WindowManager) getSystemService
(WINDOW_SERVICE);

// Apply the default display (devices display) to a display object
Display display = windowManager.getDefaultDisplay();

// Apply the devices width/height to our respective constants
WIDTH = display.getWidth();
HEIGHT = display.getHeight();

// Create our camera object with the width/height of the devices
display
mCamera = new Camera(0, 0, WIDTH, HEIGHT);
```

How it works...

The design of the artwork can be very a tedious task when first starting out with game development on a mobile platform (especially Android!). You will most likely want to be able to scale your graphics accordingly on both small displays as well as large displays. This is where the problem lies for most people because of the fact that resolutions can vary greatly from one device to the next. There are two obvious "solutions" here, the first being draw your art for a single display size and let the rest of the devices deal with the improper scaling. The second obvious approach would be to draw your images at dimensions which would best fit the largest of displays, then scale them down for smaller displays. This "solution" would technically work, as we wouldn't be losing any quality in our images. But then again, it's not exactly fair to make smaller devices suffer during load times simply because you have to create all of your images at 2 to 3 times the size.

Let's take a look at how we can easily handle the design of our images at lower resolutions while not causing improper scaling of images. Rather, we will base the width/height of our game surface depending on the size of the devices display.

With these few lines of code, instead of setting a hard-coded width and height of our camera's surface area we base those two variables off of the actual pixel dimensions of the devices display. In order to setup display-density scaling, these few lines should simply be included *before* the creation of the camera in our `onCreateEngineOptions()` method.



You've most likely played a game on your computer where after changing to a higher resolution, entities on the screen appear smaller. This is done to mitigate scaling of the images on-screen which causes the large, pixelated graphics. In this image, both stars have the same texture option and formats but we can clearly see that the star on the right (based off the display width/height) looks much smoother. In fact, both of these images are rendered with nearest texture filtering, yet the properly scaled star 'almost' looks as if it is using the bilinear format. Display scaling does not affect the performance of the game.

AndEngine font resources

AndEngine fonts are simple to setup and include for use in our text objects to be displayed on screen. We can choose from preset fonts or we can add our own via the `assets` folder.

Getting ready

Apply the following code to the `onCreateResources()` method in the `PacktActivity.java` class.

How to do it...

create() method for preset fonts

```
mFont = FontFactory.create(mEngine.getFontManager(), mEngine.
getTextureManager(), 256, 256, Typeface.create(Typeface.DEFAULT,
Typeface.NORMAL),  32f, true, Color.WHITE)

mFont.load();
```

createFromAsset() method for custom fonts

```
mFont = FontFactory.createFromAsset(engine.getFontManager(), engine.
getTextureManager(), 256, 256, this.getAssets(), "Arial.ttf", 32f,
true, Color.WHITE);

mFont.load();
```

createStroke() and createStrokeFromAsset() methods

```
BitmapTextureAtlas mFontTexture = new BitmapTextureAtlas(engine.
getTextureManager(), 256, 256);

mFont = FontFactory.createStroke(engine.getFontManager(),
mFontTexture, Typeface.create(Typeface.DEFAULT, Typeface.BOLD), 32,
true, Color.WHITE, 3, Color.BLACK);

mFontTexture.load();
mFont.load();
```

How it works...

create() method explanation

The `create()` method doesn't allow for too much customizability. This method's parameters (starting at the fifth parameter) include supplying a typeface, font size, anti-aliasing, and a color. We're using the Android native typeface class which only supports a few different fonts and styles.

createFromAsset() method explanation

We can use this method in order to introduce custom fonts into our project via our `assets` folder. Let's assume we have a true-type font called `Arial.ttf` located in our project `assets` folder.

We can see that the general creation is the same. In this method, we must pass the activities **AssetManager** which can be obtained through our activity's `getAssets()` method. The parameter following that is the true type font we would like to import.

createStroke() and createStrokeFromAsset() method explanations

Finally we have our stroke fonts. The stroke font gives us the ability to add outlines to the characters in our font. These fonts are useful in situations where we would like our text to "pop."

For creating stroke fonts, we'll need to supply a texture atlas as the second parameter rather than passing the engine's texture manager. From this point we can either create the stroke font via a typeface or through our assets folder. Additionally we're given the option to adjust 2 new parameters (the last 2 parameters in the method call). With these new parameters, we are able to adjust the thickness of the outline as well as the color. Don't forget that we must call the `load()` methods on both our texture and our font.

There's more...

FontUtils class

AndEngine includes a class called **FontUtils** which allows us to retrieve information regarding a text object's width on the screen via the `measureText()` method. This is important when dealing with dynamically changing strings as it gives you the option to relocate your text object, assuming the width or height of the string (in pixels) has changed. The `FontUtils` class also includes 2 other methods, one of which allows us to split lines depending on the width of the text and another that allows us to cut off any text larger than a specified width. These two methods are called `splitLines()` and `breakText()`.

See also

- Understanding the life-cycle
- Different types of textures

Creating the resource manager

Getting ready

Refer to the project ClassCollection01 (`ResourceManager.java`) for the working code for this topic.

How to do it...

Due to the efficiency of working with `BuildableBitmapTextureAtlases` opposed to `BitmapTextureAtlases`, we're going to continue from this point on, working with the `BuildableBitmapTextureAtlas` class.

How it works...

With this class imported to our project, we can now easily load our various scenes resources completely independent of one-another. On top of that, we now only require one line of code when it comes to loading our scene resources which helps greatly in keeping our main activity class more organized. Here is what our `onCreateResources()` methods should look like with the use of a resource manager:

```
//=====================================================
// CREATE RESOURCES
//=====================================================
@Override
```

```
    public void onCreateResources(
                OnCreateResourcesCallback pOnCreateResourcesCallback)
                throws Exception {

            // ResourceManager requires us to pass our engine and
application's context
            // in order to load resources for textures and/or fonts
            ResourceManager.getInstance().loadMenu(mEngine,
getApplicationContext());

            ResourceManager.getInstance().loadFonts(mEngine,
getAssets());

            pOnCreateResourcesCallback.onCreateResourcesFinished();
    }
```

If we take a look back at the resource manager class file, we can see that we must pass the engine as well as a context for loading textures and an asset manager for loading stroke fonts. The `BaseGameActivity` class we are using provides us with an Engine object (`mEngine`) as well as methods to retrieve the activity's context and assets manager.

The resource manager's structure is similar to that of the game manager, though the purpose is different. The resource manager will provide loading methods for all of the required texture regions, sounds, and fonts that are to be used throughout our project. Since each scene in a game will require a different set of texture regions, we'll create a few methods such as `loadMenuResources()`, `loadGameResources()` and so on in that fashion for all additional scenes in our game.

On top of using our resource manager to initially load the required texture atlases for our scenes, we'll have access to all of the texture regions in our game through this class. With this implementation, any time we need to obtain a texture region from the resource manager, the call would be as simple as `ResourceManager.getInstance().mTextureRegion`. Where `mTextureRegion` is the specific texture region we're attempting to apply to a sprite.

There's more...

In larger projects, sometimes we may find ourselves passing main objects to classes very frequently. Another use for the resource manager is to store some of the more important game objects such as the engine or camera. This way we no longer have to continuously pass these objects as parameters, rather we can call respective 'get' methods in order to get the game's camera, engine or any other specific object we'll need to reference throughout the classes. To do this, we can call the `ResourceManager.getInstance().setEngine()` method from our resource manager in the `onCreateResources()` method. From this point on, we can simply call `ResourceManager.getInstance().getEngine()` to obtain our engine object. Ultimately reducing some of the more general objects we'd otherwise find ourselves passing as parameters relatively frequently.

Designing a splash screen

Splash screens are vital when it comes to letting the users of our applications know who is behind the development of the project. A splash screen is usually a simple logo which is displayed before or during the loading process. Generally a splash screen is shared across all applications by the same company. The purpose of this is to allow users to become familiarized with our company more efficiently, as opposed to having a different style for each logo and application. It also makes it easier using a single logo for all applications in a case where our company has a project website. Instead of having a number of logo's tied to the website, we'll have a single universal logo wherever our company name is found.

Getting ready

Refer to the project `ClassCollection01` (`SplashSceneActivity.java`) for the working code for this topic.

How to do it...

```
public class SplashSceneActivity extends BaseGameActivity {

    //=====================================================
    // CONSTANTS
    //=====================================================
    public static int WIDTH = 800;
    public static int HEIGHT = 480;

    // Instead of graphics, we'll be using these strings which will
    // represent our splash scene and menu scene
    public static final String SPLASH_STRING = "HELLO SPLASH SCREEN!";
    public static final String MENU_STRING = "HELLO MENU SCREEN!";

    //=====================================================
    // VARIABLES
    //=====================================================
    // We'll be creating 1 scene for our main menu and one for our
splash image
    private Scene mMenuScene;
    private Scene mSplashScene;

    private Camera mCamera;

    // These two text objects will represent each of our two scenes to
be displayed
```

```
    Text mSplashSceneText;
    Text mMenuSceneText;

    //=====================================================
    // CREATE ENGINE OPTIONS
    //=====================================================
    @Override
    public EngineOptions onCreateEngineOptions() {
          mCamera = new Camera(0, 0, WIDTH, HEIGHT);

          EngineOptions engineOptions = new EngineOptions(true,
ScreenOrientation.LANDSCAPE_FIXED, new FillResolutionPolicy(),
mCamera);

          engineOptions.setWakeLockOptions(WakeLockOptions.SCREEN_ON);

          mEngine = new FixedStepEngine(engineOptions, 60);

          return engineOptions;
    }

    //=====================================================
    // CREATE RESOURCES
    //=====================================================
    @Override
    public void onCreateResources(
                OnCreateResourcesCallback pOnCreateResourcesCallback)
                throws Exception {

          // Load our fonts.
          ResourceManager.getInstance().loadFonts(mEngine,
getAssets());


          pOnCreateResourcesCallback.onCreateResourcesFinished();
    }

    //=====================================================
    // CREATE SCENE
    //=====================================================
    @Override
    public void onCreateScene(OnCreateSceneCallback
pOnCreateSceneCallback)
                throws Exception {
```

```
            // Retrieve our font from the resource manager
            Font font = ResourceManager.getInstance().mFont;

            // Set the location of our splash 'image' (text object in
this case).
            // We can use FontUtils.measureText to retrieve the width of
our text
            // object in order to properly format its position
            float x = WIDTH / 2 - FontUtils.measureText(font, SPLASH_
STRING) / 2;
            float y = HEIGHT / 2 - font.getLineHeight() / 2;

            // Create our splash scene object
            mSplashScene = new Scene();
            // Create our splash screen text object
            mSplashSceneText = new Text(x, y, font, SPLASH_STRING,
SPLASH_STRING.length(), mEngine.getVertexBufferObjectManager());
            // Attach the text object to our splash scene
            mSplashScene.attachChild(mSplashSceneText);

            // We must change the value of x depending on the length of
our menu
            // string now in order to keep the text centered on-screen
            x = WIDTH / 2 - FontUtils.measureText(font, MENU_STRING) /
2;

            // Create our main menu scene
            mMenuScene = new Scene();
            // Create our menu screen text object
            mMenuSceneText = new Text(x, y, font, MENU_STRING, MENU_
STRING.length(), mEngine.getVertexBufferObjectManager());
            // Attach the text object to our menu scene
            mMenuScene.attachChild(mMenuSceneText);

            // Finally, we must callback the initial scene to be loaded
(splash scene)
            pOnCreateSceneCallback.onCreateSceneFinished(mSplashScene);
        }

        //=====================================================
        // POPULATE SCENE
        //=====================================================
        @Override
        public void onPopulateScene(Scene pScene,
                OnPopulateSceneCallback pOnPopulateSceneCallback)
```

```
    throws Exception {

              // We will create a timer-handler to handle the duration
              // in which the splash screen is shown
              mEngine.registerUpdateHandler(new TimerHandler(4, new
    ITimerCallback(){

              // Override ITimerCallback's 'onTimePassed' method to allow
              // us to control what happens after the timer duration ends
              @Override
              public void onTimePassed(TimerHandler pTimerHandler) {
                    // When 4 seconds is up, switch to our menu scene
                    mEngine.setScene(mMenuScene);
                    }
              }));

              pOnPopulateSceneCallback.onPopulateSceneFinished();
        }
    }
```

How it works...

The general process of application start up in proper order starts with the splash screen, then the loading screen and finally the main menu. In some cases it may be alright to incorporate the splash screen into the loading screen, but in mobile development that can be a good idea or a bad idea depending on the scenario. If your game is set to load a menu screen with a maximum of 10 textures, a few sounds and a few sprites, the loading process may take only 0.5 seconds depending on the device. The splash screen would disappear before the user can even see it. On the other end, if we're using a splash screen which has a constant timer, we may be causing the user to wait longer than they have to if their device loads quickly. It is a good idea to base your plan for loading and splash screens around the resources needed for the main entry to your application. We can usually follow the rule; for long load times, incorporate the splash screen into the load time. For short load times, use a timer of about 2-4 seconds for displaying the splash screen.

Splash screen design

As mentioned before, unless we have plans for solely creating a specific genre and theme for our games it is a good idea to create a universal logo. We may be interested in creating games at this point in time, but in the future if we were offered a job in which we were given the option to include our own logo into the end-product, we might receive some looks when applying our 'hack n slash' styled logo into a business app related to food products (this is pure example, but the possibility is real). We never know which projects we will be working on a few months or years down the road, so choose carefully.

I'll now take this opportunity to include the independent company logo which I work for as an example:

The logo is small, simple and it gets the point across. All the splash screen really needs to include is a company name. The rest of the artwork is completely optional. There's no harm in designing a fancy looking splash screen, but in reality the users of the application are going to base the quality of the company on the application itself and not the splash screen.

There's more...

Timers and simple transition effects

In the `SplashSceneActivity.java` class we introduced a simple splash screen to our startup process. The splash scene was shown, then in the blink of an eye we were brought to the menu scene. In reality there's nothing wrong with that approach, but adding simple effects can help to make our splash screen more appealing than a static image. When it comes to designing a game, try to keep the full-screen inanimate images to a minimum. They are boring and unnecessary.

We're going to add one of the most basic, yet widely used effects for a logo screen, which is the fade-in/fade-out transition. The effect is simple, giving the user visualization that there is actually something happening and that the game is not frozen. A user should always feel as though one scene is taking them to another, rather than scenes feeling as though they're randomly disappearing and reappearing.

Splash screen transition effect

The following code is based on the `SplashSceneActivity.java` class. The only changes here reside in the `onPopulateScene()` method. We will no longer need a timer-handler as we will be using a modifier to create the fading effect and a modifier listener to switch our scene when the modifier ends its sequence.

```
@Override
public void onPopulateScene(Scene pScene,
          OnPopulateSceneCallback pOnPopulateSceneCallback) throws
Exception {
```

```
        // Loop through scene children
        for(int i = 0; i < mSplashScene.getChildCount(); i++){
                // Register our fade effect to each of the scenes children
                mSplashScene.getChildByIndex(i).registerEntityModifier(new
    SequenceEntityModifier(new IEntityModifierListener(){

                        @Override
                        public void onModifierStarted(IModifier<IEntity>
    pModifier,
                                        IEntity pItem) {
                                // TODO Auto-generated method stub
                        }

                        @Override
                        public void onModifierFinished(IModifier<IEntity>
    pModifier,
                                        IEntity pItem) {
                                // Switch to our menu scene when the sequence
    modifier ends
                                mEngine.setScene(mMenuScene);
                        }
                        // Add alpha modifiers of 2 seconds each, one fades
    in, one fades out
                }, new AlphaModifier(2, 0, 1), new AlphaModifier(2, 1, 0)));
        }

        pOnPopulateSceneCallback.onPopulateSceneFinished();
    }
```

What we're doing here is looping through all of the `mSplashScene` children and applying an **entity modifier** which will handle the fade-in/fade-out via **alpha modifiers**. In this particular instance, we're including a listener in our modifier which will call `mEngine.setScene(mMenuScene)` in order to switch over to our menu scene when the transition effect is complete.

See also

- Creating the resource manager
- Andengine font resources

Saving and loading game data

In the final topic for game structure we're going to set up a class which can be used in our project to manage data and settings. The more obvious game data we must save should include character stats, high scores and other various data we may have included in our game. We should also keep track of certain options a game might have, such as whether the user has sounds muted or not, gore effects, and more. In a large game with a large amount of data to keep track of, it might seem like a tedious task to come up with a way to save and load all of the different settings. Thankfully, the Android SDK provides us with a solution to this called the **SharedPreferences** class.

Getting ready

Refer to the project `ClassCollection01` (`UserData.java`) for the working code for this topic.

How to do it...

```
public class UserData {

    //=====================================================
    // FIELDS
    //=====================================================
    private static final UserData INSTANCE = new UserData();

    // These keys will tell the shared preferences editor which
    // data we're trying to access
    private static final String UNLOCKED_LEVEL_KEY = "unlockedLevels";
    private static final String SOUND_KEY = "soundKey";

    //=====================================================
    // FIELDS
    //=====================================================
    // Include a 'filename' for our shared preferences
    private static final String PREFS_NAME = "GAME_USERDATA";

    // Create our shared preferences object & editor which will
    // be used to save and load data
    private SharedPreferences mSettings;
    private SharedPreferences.Editor mEditor;
```

```
    //=====================================================
    // VARIABLES
    //=====================================================
    // keep track of our max unlocked level
    private int   mUnlockedLevels;

    // keep track of whether or not sound is enabled
    private boolean     mSoundEnabled;

    //=====================================================
    // SINGLETON
    //=====================================================
    UserData(){   }
    public static UserData getInstance(){ return INSTANCE;      }


    //=====================================================
    // INIT
    //=====================================================
    public void init(Context context){
          if(mSettings == null){
                // Retrieve our game's shared preferences options
                mSettings = context.getSharedPreferences(PREFS_NAME,
0);
                // Setup our editor which will be used to save data
                mEditor = mSettings.edit();

                //-------------------
                //-- DEFAULT VALUES--
                //-------------------
                // Retrieve our current unlocked levels. if the
UNLOCKED_LEVEL_KEY
                // does not currently exist in our shared
preferences, we'll create
                // the data to unlock level 1 by default
                mUnlockedLevels = mSettings.getInt(UNLOCKED_LEVEL_
KEY, 1);
                // Same idea as above, except we'll set the sound
boolean to true
                // if the setting does not currently exist
                mSoundEnabled = mSettings.getBoolean(SOUND_KEY,
true);
          }
    }
```

```
    //=====================================================
    // GET AND SET
    //=====================================================
    // Retrieve our settings for loading game data
    public int getMaxUnlockedLevel() { return mUnlockedLevels; }
    public boolean isSoundMuted() { return mSoundEnabled; }

    //=====================================================
    // UNLOCK NEXT LEVEL
    //=====================================================
    // This method would generally be called after completing the last
    // level as user currently has available
    public void unlockNextLevel(){
           // Increase the max level by 1
           mUnlockedLevels++;
           // Edit our shared preferences unlockedLevels key, setting
its
           // value our new mUnlockedLevels value
           mEditor.putInt(UNLOCKED_LEVEL_KEY, mUnlockedLevels);
           // commit() *must* be called after changing any preferences
           mEditor.commit();
    }

    //=====================================================
    // CYCLE SOUND MUTE
    //=====================================================
    // For this method, we're following the same idea as the
unlockNextLevel
    // method, except we're saving a boolean value rather than an
integer
    public void setSoundMuted(final boolean enableSound){
           mSoundEnabled = enableSound;
           mEditor.putBoolean(SOUND_KEY, mSoundEnabled);
           mEditor.commit();
    }
}
```

How it works...

This class demonstrates just how easily we are able to store and retrieve a games data and options through the use of shared preferences. The structure of the `UserData` class is fairly straight-forward and can be used in this same fashion in order to adapt to various other options you may include in a game. Before doing that, we should take a look at the structure of this class, split into the initialization method and the getter/setter methods (per key-value).

Shared preference initialization

The `UserData` class only requires one call to the `init()` method on activity startup. The purpose of this method is to create a new shared preferences file for our application, creating each field our game requires in terms of data storage. If our application already has a shared preference file, we simply load our settings and apply them to our `UserData` class variables through the use of our keys (the constant strings).

Getting and setting data

Each and every setting we choose to save into our shared preference file should have a 'getter' and 'setter' method related to it. This will allow us to easily manipulate the state of our shared preferences throughout our game's class files in order to save and load game data.

Setting

Additionally, for all of our game data variables we will have a corresponding method in order to modify it. The `setSoundMuted()` method is an example of a 'setter' for game data. The purpose of these setters is to provide updates to our shared preference file through the use of our keys. A situation in which this method would be used is after clicking a mute button in your options menu, we'd send a call to `setSoundMuted(true)` which will save the '`true`' variable in our shared preferences.

Getting

Assuming a user has turned off the sound, before playing our theme music we may check for `UserData.getInstance().getIsSoundMuted()`. If the returning value is equal to '`true`', we can allow our application to skip the process of playing sounds. This concept is transferable to many of the other forms of loading data as well.

See also

- What are singletons?

2
Working with entities

In this chapter, we're going to start getting into displaying objects on the screen and various ways we can work with these objects. The topics include:

- Understanding AndEngine entities
- Setting up nested layers
- Applying primitives to a layer
- Applying text to a layer
- Applying an FPS counter for development
- Adding a screen capture function to your game
- Creating a sprite group
- Applying culling to entities
- Using Relative rotation
- Working with OpenGL
- Overriding onManagedUpdate
- Using entity modifiers
- Advanced entity modifiers
- Working with particle systems
- Creating Advanced particle systems

Introduction

In this chapter, we're going to start working with all of the wonderful entities included in AndEngine. Entities in a game and play the biggest role in the overall look and feel of the final product. They include everything from particles, sprites, text, and even the layers that all of our different entities are attached to.

Understanding AndEngine entities

Aside from layers, entities have many, many uses. Though, even these other various entities *can* play the role of a layer if necessary (we can attach a sprite to a sprite, for example). In AndEngine, entities make up the entire list of objects which are physically displayed on the device. Text, sprites, particles, primitive shapes and everything else you can think of are all sub-types of entities. In future topics and chapters, we will be going over the different aspects of `Entity` sub-types individually and see how we can manipulate those sub-types to create completely customized user-interfaces and game-play.

How to do it...

One of the first tasks that should take place within the creation of our scene is the creation/attachment of our Entity layers.

1. This can be done in only two lines per layer:

```
Entity layer = new Entity();
mScene.attachChild(layer);
```

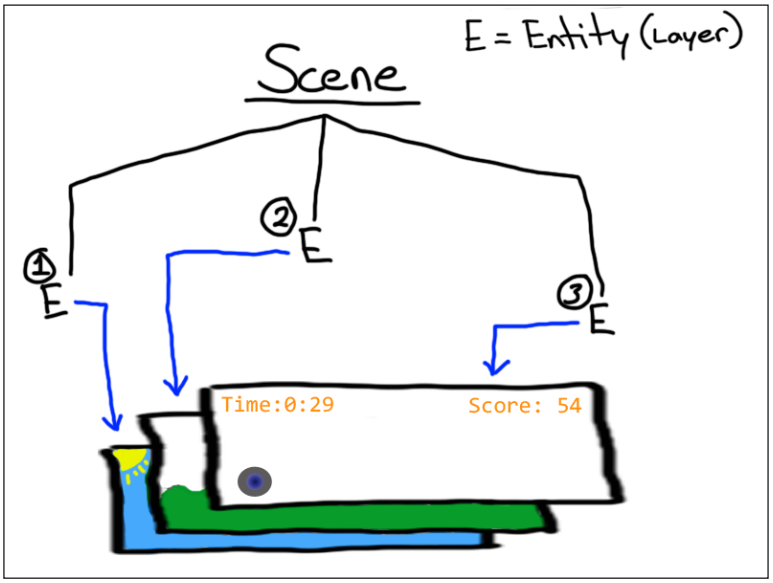
How it works...

The two lines of code in the step above allow us to attach an empty entity, or a layer in this case, to our `Scene` object. The first line simply creates the `Entity` object for us to use as a layer, while the second line applies it to the scene. From here on, we can continue to attach any other variation of an `Entity` to the layer in order to properly organize the objects in a first attached/last visible scenario.

Entities are very important when developing games. In AndEngine, the fact of the matter is that all objects displayed on our scenes are derived from entities (including the `Scene` object itself!). When working with entities, in most cases we can assume that the entity is either a display object (sprite, text, etc.) or a layer, which use is to contain and organize children. Seeing as how broad the entity class is, we're going to talk about each of the two uses for entities as if they were separate objects.

The first and arguably most important aspect of an Entity is the layering capabilities. A layer is a very simple concept in game design, however due to the amount of entities games tend to support during game-play, things can quickly become confusing when first getting to know them. We must think of a layer as an object which has one parent and an unlimited amount of children (unless otherwise defined). As the name suggests, the purpose of a layer is to apply our various entity objects on our scene in an organized fashion. We can assume that if we have a background image, a foreground image, and a `HUD` (heads-up display), that our game will have three separate layers. These three layers would appear in a specific order,

as if stacking pieces of paper on top of each other. The last piece of paper added to the stack will appear in its full-form. See the image below:

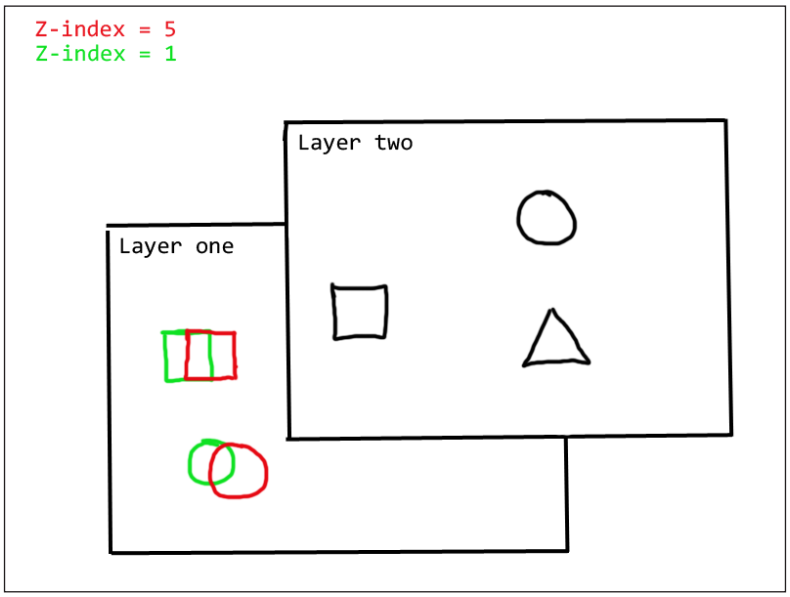


The image above depicts a basic game scene consisting of three layers. Each of the three layers has a specific purpose, which is to store all relative entities in terms of depth. The first layer applied to the scene is the background layer, including a blue sky and a sun. The second layer applied to the scene is the middle layer. On this layer we would find objects which are relative to the player, including our landscape, collectable items, enemies and more. The final layer is the `HUD`, used to display the front-most entities on the devices display. The `HUD` object in AndEngine will automatically appear as if it were the last layer added, by default. This is to always have them available and visible to the player as a heads-up display should be.

There's more...

We should take some time to discuss the z-index functionality of our entity objects. If you are familiar with the coordinate system in OpenGL, you will already know that the z-index is the 'depth' value of an entity. In AndEngine, since we're working with a 2d engine we don't work with depth in the conventional sense. Entities allow for defining a z-index which is useful for organizing the hierarchy of the entities which share the same parent, altering the way we see objects on the screen relevant to being in front of or behind other entities. Manually setting an entities z-index will no longer take into account the order that the entities were attached to the scene.

Let's see what the effect of setting the z-index of an object will do:



In this image, we have two layers. Let's assume these layers were attached to a scene in their respective order (layer one first attached to the scene, causing its entities to appear behind layer two's entities by default). Now let's assume that the two red shapes were attached to layer one first, then the green shapes. Originally, the red shapes would have appeared behind the green shapes due to the order in which the children (shapes) were attached. We can see that we've manually set the shape z-indexes, red shapes being 5, green being 1. Entities with a higher z-index will *always* appear to be in front of entities with a lesser z-index, regardless of the order of attachment. That being said, this is only taken into account for entities with a relative parent. In this case, the red shapes with a z-index of 5 will still appear behind layer two's shapes because layer two does not belong to the entity hierarchy of layer one.

The following code shows how we can define layering based on the z-index:

```
// Create a new layer
Entity layer = new Entity();

// Create two 'open' entities
Entity childOne = new Entity();
Entity childTwo = new Entity();

// Attach children to our layer
layer.attachChild(childOne);
layer.attachChild(childTwo);
// Manually define z-index of our children
```

```
childOne.setZIndex(5);
childTwo.setZIndex(1);

// Sort children
layer.sortChildren();
```

This code relates to the previous image. With this code, since our children are base-entities we won't actually be able to see that the first child is attached to the scene, followed by the second child. This causes the first child to appear behind the second by default. We continue on to manually set the first child's z-index to a higher value than that of the second child, then call the `sortChildren()` method on the child's parent in order to cause the z-index changes to take effect. In order to visualize these changes, feel free to substitute the `childOne` and `childTwo` objects for sprite or rectangle entities.

The entity class in AndEngine is responsible for positioning, rotating, skewing, scaling and almost every other visual object modification we can make. Want to cause our game to 'flip' out of camera view when moving to the main menu? We can do that by applying certain methods to an entire layer!

Setting up nested layers

We've discussed the uses of layers in the previous topic, now let's find out how we can setup our game to work with layers rather than planting all of our entities directly on the scene. This will help to add organization in our game.

Getting started..

Refer to the `ClassCollection02` (`NestedLayers.java`) for the working code for this topic.

How to do it...

Setting up the layers for the scene is one of the first tasks that should be handled when starting the development of a game. Nested layers are a key component of keeping the various entities on a scene organized and more importantly, visible when they need to be.

1. The layers must first be declared. The layers will be divided up into master layers and sub-layers:

```
// Master layers
Entity mBackgroundMasterLayer;
Entity mPlayerMasterLayer;
Entity mHudMasterLayer;
```

```
// Background sub-layers
Entity mLayerBackgroundFurthest;
Entity mLayerBackgroundMid;
Entity mLayerBackgroundClosest;

// Character and item sub-layers
Entity mLayerPlayerFurthest;
Entity mLayerPlayerClosest;

// HUD sub-layers
Entity mLayerHudFurthest;
Entity mLayerHudClosest;
```

2. The second step simply involves attaching the layers to the scene in the order we wish for them to group their children, in terms of depth:

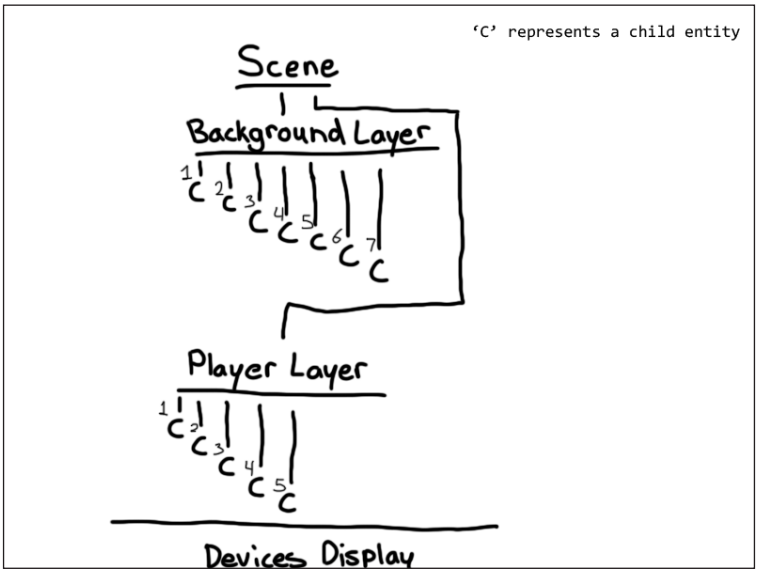
```
// Attach our master layers directly to the scene (in proper
order)
mScene.attachChild(mBackgroundMasterLayer);
mScene.attachChild(mPlayerMasterLayer);
mScene.attachChild(mHudMasterLayer);

// Attach our background sub-layers to the master background layer
mBackgroundMasterLayer.attachChild(mLayerBackgroundFurthest);
mBackgroundMasterLayer.attachChild(mLayerBackgroundMid);
mBackgroundMasterLayer.attachChild(mLayerBackgroundClosest);

// Attach our player sub-layers to the master player layer
mPlayerMasterLayer.attachChild(mLayerPlayerFurthest);
mPlayerMasterLayer.attachChild(mLayerPlayerClosest);

// Attach our HUD sub-layers to the master HUD layer
mHudMasterLayer.attachChild(mLayerHudFurthest);
mHudMasterLayer.attachChild(mLayerHudClosest);
```

How it works...

In step one we are simply declaring the layers which will allow us to group on-screen entities more easily. The first three layers declared are what we will call a master layer, whose purpose is to solely contain sub-layers. For each master layer, we may have a background, mid-ground or foreground sub-layer whose purpose is to contain sprites, text, or other entities which will be physically displayed on the device.

Step two contains the code required in order to apply the layers to the scene in an organized manner. The first three master layers will be attached directly to the scene. Once the master layers have been attached to the scene, we can continue on to attach the sub-layers in regards to how they should be seen on the device. Take the `mBackgroundMasterLayer` as an example; `mLayerBackgroundFurthest` is attached first. Any other entity attached to the master layer after that will appear in front of the `mLayerBackgroundFurthest` object. Once all of necessary master layers and sub-layers have been attached in their proper order, we can continue on to attach objects to the background or the to `HUD` without disturbing the order of any other entities.

Let's take a look at this diagram to get an idea of what's happening:

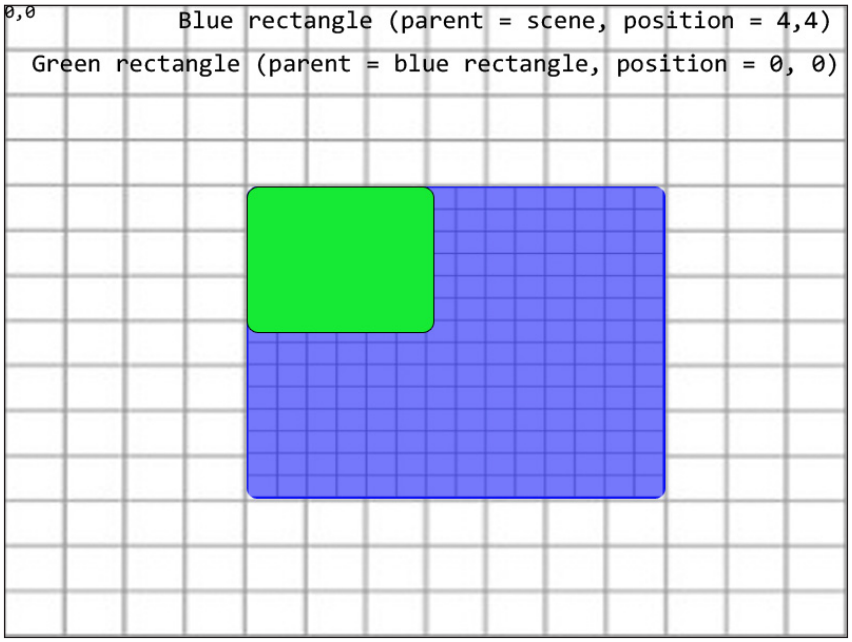


This image represents the background and player layer of a game assuming they were attached in correct order. The numbers beside each child represent the order in which they are attached to their parent. First of all, we must understand that the scene is the furthest point of depth in our game world. The devices display would be the closest. Knowing this, we can see that children on the background layer will never appear in front of the children on the player layer.

The depth of an entity in AndEngine does not affect its size. Depth will only affect whether or not one entity will appear in front or behind others.

There's more...

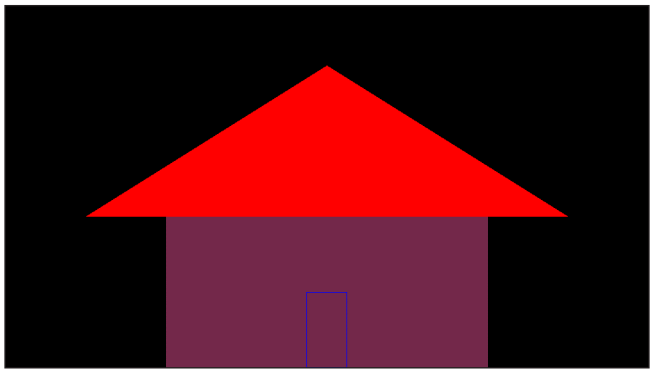
One thing we should go over before moving on is the fact that *children inherit parent values*! This is a common mistake that many AndEngine developers run into when setting up multiple layers in their games. Everything from skew, scale, position, rotation, visibility, and more are all taken into account by child entities when their parent's properties change. However, a child *does not* inherit colors and alpha values from their parent. See the image below for an example:



In this image, we've got two separate rectangles. The large blue rectangle is attached directly to the scene, with a defined position of (x = 4, y = 4). The green rectangle is attached directly to the blue rectangle, making the blue rectangle green's parent. Note that the "initial" (0,0) coordinates of an entity will always remain at the exact top/left corner of its parent, following even if the parent moves. A child will always follow its parents inherited values unless otherwise defined.

Applying primitives to a layer

AndEngine's primitive types include lines, rectangles, and meshes. In this topic we're going to focus on the mesh class. Meshes are useful for creating more complex shapes in our games which can have an unlimited amount of uses. We're going to draw a basic house in this topic out of meshes.



Getting started..

Import the `ApplyingPrimitives.java` code into the `onPopulateScene()` method of your current project. Refer to the `ClassCollection02` (`ApplyingPrimitives.java`) for the working code for this topic.

How to do it...

In order to create a mesh object, we need to do a little bit more work than what's required for a typical `Rectangle` or `Line`.

1. Create the buffer data float arrays:

```
// Set the raw points for our rectangle mesh
float baseBuffer[] = {
            // Triangle one
            -200, -100, 0, // point one
            200, -100, 0, // point two
            200, 100, 0, // point three

            // Triangle two
            200, 100, 0, // point one
            -200, 100, 0, // point two
            -200, -100, 0, // point three
};
```

2. Create and attach the `Mesh` object:

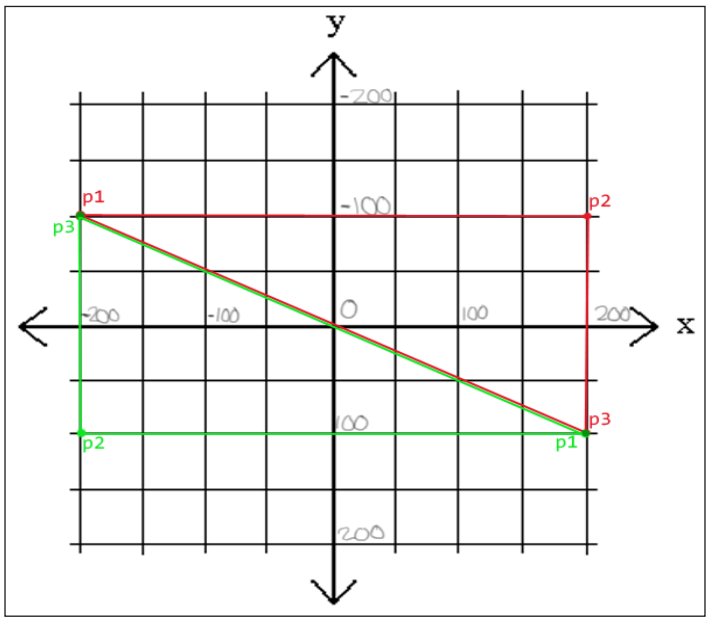
```
// Create the base mesh at the bottom of the screen
Mesh meshBase = new Mesh(400, 480 - 100, baseBuffer, 6,
      DrawMode.TRIANGLES, mEngine.getVertexBufferObjectManager());
// Attach base mesh to the scene
mScene.attachChild(meshBase);
```

How it works...

Let's break down the process a little bit more in order to explain how to find out how we came up with a "house."

In step one, we can see from the comments that each triangle is made up of three points and each point made up of three values. The first value per point is the x-coordinate, second is the y-coordinate, and the third value is the z-coordinate. Since working with a 2d engine, we won't need to set the z-coordinate.

See the image below for a graph which we will use to draw the triangles based on the points in the above code snippet:



The second step involved in applying a mesh to the screen is creating the object. The mesh constructor requires a few parameters to be passed to it. The first two parameters are the `x` and `y` coordinates for the entity. The third parameter is the `float` array that was created in the first step, followed by an integer specifying the number of vertices in our buffer data (each triangle is made up of three vertices). The fifth parameter is where we pass the `DrawMode` defining which type of mesh we will be drawing.

Depending on the `DrawMode` for our mesh, we will need to adjust the amount of points needed. For `DrawMode.TRIANGLES` we'll need to include points in multiples of three (three points for each of the triangles our mesh is made up of), for `DrawMode.LINE_STRIP` we'll need at least two points, and for `DrawMode.POINTS` we can pass single index float arrays as a parameter for our mesh..

Applying text to a layer

In this topic we're going to take a look at applying `Text` objects to our layers. `Text` objects are an important part of game development as they can be use to dynamically display point systems, tutorials, descriptions and more. AndEngine also allows us to create text styles which suit individual game types better by specifying customized `Font` objects.

Getting started..

Refer to the `ClassCollection02` (`ApplyingText.java`) for the working code for this topic.

How to do it...

AndEngine's `Text` objects allow a fair amount of customization through the use of `Font` objects, the `FontUtils` class, and the `TextOptions` object.

1. First, we must create the `Font` object in the `onCreateResources()` method of our activity.

```
@Override
public void onCreateResources(
       OnCreateResourcesCallback pOnCreateResourcesCallback)
       throws Exception {

// Load our font
mFont = FontFactory.create(mEngine.getFontManager(), mEngine.
getTextureManager(), 256, 256, Typeface.create(Typeface.DEFAULT,
Typeface.NORMAL),  32f, true, Color.WHITE);
mFont.load();

pOnCreateResourcesCallback.onCreateResourcesFinished();
}
The next step is to obtain the correct position of the text. In
this case, we're obtaining the center screen position through the
use of the FontUtils.measureText() method.
// Measure the width our string will appear on the screen in
pixels
final float textLength = FontUtils.measureText(mFont, TEST_
STRING);

// Set our text to appear in the center of the screen based on its
width and height
final float x = (WIDTH / 2) - (textLength / 2);
final float y = (HEIGHT / 2) - (mFont.getLineHeight() / 2);
```

2. For better alignment on the `Text` object, we can apply a `TextOptions` object to the text.

```
// Create TextOptions for our text
final TextOptions textOptions = new TextOptions();

textOptions.setHorizontalAlign(HorizontalAlign.CENTER);
```

3. Finally, we can continue on to creating the `Text` object with the previously defined coordinates, `Font`, and `TextOptions`.

```
mText = new Text(x, y, mFont, TEST_STRING, TEST_STRING.length(),
textOptions, mEngine.getVertexBufferObjectManager());
```

How it works...

In step one we are creating a generic font-type to be used later by the `Text` object. Don't forget to set the `Font` color to white in order to allow colors to be properly applied to the `Text` object via the `setColor()` method.

In step two, we are obtaining the text's position in the center of the device's display through the use of the `FontUtils.measureText()` method. By passing a string to the `measureText()` method, we can obtain the width of the string in pixels. We can successfully obtain the center coordinates of the display for the text object by dividing half of the text's width and height by half of the display's width and height.

In step three we are creating a `TextOptions` object to be passed to the `Text` object as a parameter. The `TextOptions` allows us to specify auto-wrap width's and alignment options. However, the `TextOptions` object is not a necessity when creating `Text` objects.

Once we've created our `Font` object and obtained the proper coordinates for the `Text`, we can go ahead and create the object. As per all entities, the first two parameters are the `x/y` coordinates of the entity. The third is the font that our text will be using. The fourth and fifth parameters is the initial string that the text entity will display followed by the maximum character length of this text entity. The sixth parameter is the previously created `TextOptions` object.

Remember to include the absolute maximum character count that the text entity will encounter! If this rule is not followed, it is very likely that the entity will cause a force-closure of the application due to `IndexOutOfBoundsException`.

Applying an FPS Counter for development

Obtaining the frames-per-second that our applications are achieving is an important part of the development process of large-scale games. By applying an FPSCounter to our engine, we can output an accurate FPS count to the LogCat. This is useful during the optimization of applications as a whole, or even during more specific method executions, depending on what we're looking to optimize.

Getting started..

Refer to the `ClassCollection02` (`FPSCounterDevelopment.java`) for the working code for this topic.

How to do it...

In order to obtain the FPS of our application, we must create an object called an FPSCounter and register it as an update handler with our engine.

1. Create the `FPSCounter` object:

```
FPSCounter mFpsCounter = new FPSCounter();
```

2. Register the `FPSCounter` as an update handler:

```
mEngine.registerUpdateHandler(mFpsCounter);
```

3. Output FPS logs after every five second via updates to the Scene object:

```
mScene = new Scene(){
@Override
protected void onManagedUpdate(float pSecondsElapsed) {

       // Accumulate current time passed
       mCurrentTime += pSecondsElapsed;

       // If current time passed greater or equal to timer limit (5
seconds)
       if(mCurrentTime >= FPS_TIMER){
              // Display our FPS in logcat every 5 seconds
              Log.i("FPS", String.valueOf(mFpsCounter.getFPS()));

              // Reset our current time passed variable
              mCurrentTime = 0;
       }
       super.onManagedUpdate(pSecondsElapsed);
}
};
```

How it works...

In the first and second steps, we're creating an `FPSCounter` and registering it to the engine as an update handler. What this allows the `FPSCounter` to do is obtain a constant update of frames passed as well as how many seconds elapsed before the next frame is called. With this info, the `FPSCounter` is able to tell us how much FPS the application is running at by dividing the frames passed by the seconds elapsed.

In the third step, we're overriding the scene's `onManagedUpdate()` method in order to print FPS logs. After every update of our main thread, we are passing the time passed since last update to our `mCurrentTime` variable. Next we have an 'if' statement, which checks if the `mCurrentTime` value is equal to or greater than the `FPS_TIMER` constant; if so, the FPS is outputted to the logcat and we reset the `mCurrentTime` variable back to 0 and start the process over again.

Adding a screen capture function to your game

In today's mobile games, a game is just not complete without the ability to capture that perfect moment. Users love being able to share their experiences while playing games, but at the same time it can be useful to developers as well. Let's find out how we can incorporate screen capture functionality into our own games.

Modify the AndroidManifest.xml!

In order to allow a device to write the image file to a user's device, we must include the necessary permission in the AndroidManifest.xml of our project. Simply add '`<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>`' to the manifest file.

Getting started..

Refer to the `ClassCollection02` (`ScreenCap.java`) for the working code for this topic.

How to do it...

Adding screen capture functionality to an AndEngine game can be accomplished in just a few steps.

1. The first step is to create the ScreenCapture object:

```
ScreenCapture mScreenCapture = new ScreenCapture();
```

2. The second step is to attach the ScreenCapture object to our scene:

```
mScene.attachChild(mScreenCapture);
```

3. Step two requires us to ensure that we've got the necessary folder to write the captured image files to:

```
FileUtils.ensureDirectoriesExistOnExternalStorage(this, "");
```

4. Once we've made sure we've got the necessary folder to write to, we can call the `capture()` method on the `ScreenCapture` object:

```
mScreenCapture.capture(800, 480, FileUtils.getAbsolutePathOnExtern
alStorage(this, "name" + ".png"),new IScreenCaptureCallback(){

@Override
public void onScreenCaptured(String pFilePath) {

       // ScreenCap Successful!
       Log.i(TAG, "Successfully saved to: " + pFilePath);
}

@Override
public void onScreenCaptureFailed(String pFilePath,
              Exception pException) {

       // ScreenCap Failed!
       Log.e(TAG, pFilePath + " : " + pException.
getLocalizedMessage());
}
});
```

How it works...

In the first step, as with all `Entity` objects, we must declare and instantiate the `ScreenCapture` entity. Since we are dealing with an entity, the second step requires us to attach the object to the scene or a layer. It is important to remember that if the `ScreenCapture` object should capture all entities on the display, it must be on the highest level layer. Attaching the `ScreenCapture` object to the scene before anything else will most likely capture an image of whatever the scene's background color is set to and nothing else.

Step three is in place to assure the application that the required folder is available to write images to. Without with step, it is likely that the capture will fail. The same goes for if the application is not able to write to the external SD drive, declared within the project's manifest file.

The final step requires us to call the `capture()` method on the `ScreenCapture` object in order to capture an image of the screen. The first two parameters specify the area in pixels that we wish to capture. The second parameter we are specifying the path via '`this`' which basically tells the `FileUtils` class to set the path to the location of our application directory along with the specified file name. Lastly, we have to include an `IScreenCaptureCallback()`. We don't have to include any code in the callback, but in this case we're sending the info to LogCat specifying whether or not the image was captured properly.

See also

- Setting up nested layers – Chapter 2

Creating a sprite group

Sprite groups allow us to reduce the overhead of our child sprites by 'grouping' together many sprites which share similar properties. This is done my combining these relative properties into a single buffer (via the sprite group) rather than each sprite producing its own buffer for OpenGL.

Getting started...

Refer to the `ClassCollection02` (`ApplyingSpriteGroups.java`) for the working code for this topic.

How to do it...

A `SpriteGroup` can be treated somewhat as an optimized Entity layer which can only contain sprites. The `SpriteGroup` will render large amounts of sprites faster, but it has a maximum capacity.

1. Create the `SpriteGroup`:

```
mSpriteGroup = new SpriteGroup(0, 0, mBitmapTextureAtlas, 500,
mEngine.getVertexBufferObjectManager());
mScene.attachChild(mSpriteGroup);
```

2. Attaching sprites to the `SpriteGroup`:

```
Sprite sprite = new Sprite(tempX, tempY,
spriteWidth, spriteHeight, mTextureRegion, mEngine.
getVertexBufferObjectManager());

mSpriteGroup.attachChild(sprite);
```

How it works...

In step one we are creating the `SpriteGroup` layer. The `SpriteGroup` requires two new parameters that we've not dealt with yet. The `SpriteGroup` is an entity, so we can assume that the the first two parameters are the `x` and `y` positions. The third parameter, we're passing a `BitmapTextureAtlas`. The sprite group can only contain sprites which share the same texture atlas as the sprite group! The fourth parameter is the maximum capacity of sprites that our sprite group is able to draw. It is important to limit the capacity to the maximum size of sprites you wish to draw. *Exceeding the limit will cause the application to crash*.

When attaching sprites to the sprite group layer, we can create our sprite as we would any other sprite. We can set the position, scale, and texture region as usual. Each sprite we attach to the sprite group will increase the child count of the sprite group. Be careful that the capacity is not exceeded.

Following these steps, we can successfully allow our sprite groups to draw many, many sprites onto the screen before even noticing a drop in performance within our application. The difference is huge compared to individually drawing sprites with separate buffers. In many cases, users have claimed to achieve an improvement of up to %50 when working with games which include large amounts of entities on the scene at one time.

There's more...

It's not time to run off and convert all of your projects to use sprite groups just yet! The benefits to using sprite groups speak for themselves, but that's not to say there are no negative side-effects either. The sprite batch is not supported directly by OpenGL. The class is more or less a 'hack' which allows us to save some calls. Setting up sprite groups in more complex projects can be a hassle due to the 'side-effects'.

Due to the fact that the sprite group is technically a 'hack', there are situations where the outcome may not be warranted. Beware that after attaching and detaching many sprites which take advantage of alpha modifiers and modified visibility, some of the sprites on the sprite group may start to 'flicker'. This outcome is most noticeable after more and more sprites have been attached and detached or set to invisible/visible multiple times. There are ways around this that will not hurt performance too much, which involves moving sprites off the screen rather than detaching them from the layer or setting them to invisible. However, for larger games that only take advantage of one activity and swap scenes based on the current level, moving the sprites off the screen might only lead to future problems.

Take this into account and plan wisely before deciding to use a sprite group. It might also help to test the sprite group in terms of how you plan to use your sprites before incorporating it into your game. The sprite group is not guaranteed to cause problems, but it's something to keep in mind.

Applying culling to entities

Culling entities allows us to disallow certain entities from being rendered depending on their position. This is useful when we have many sprites on a scene that might occasionally move out of view of the camera. With culling enabled, those entities which are outside of the camera view will stop being rendered in order to save us some overhead.

How to do it...

Culling can be enabled on any type of Entity or Entity sub-type, including sprites, shapes, and text objects.

1. Attach the following method to any pre-existing entity:

```
entity.setCullingEnabled(true);
```

How it works...

Step one shows how we can apply culling to any AndEngine Entity in order to disable specific entities from being rendered while off camera. Depending on the number of entities active within a game scene, this application can improve performance quite significantly.

There's more...

Culling will only stop rendering entities which move out of visibility of the camera when attached to the scene. Because of this, it is not a bad idea to enable culling on all game objects (boxes, items, enemies, etc.) in games with a constantly moving camera or scene. For instances with large backgrounds made up of smaller textures, culling can also greatly improve performance, especially considering the size of background images.

Culling can really help us save some rendering time on entities, but that doesn't automatically mean that we should enable it on all entities. After-all, there's a reason why it's not enabled by default. It is a bad idea to enable culling on HUD entities. It might seem like a viable option to include for pause menus or other large entities which might transition in and out of the camera view, but this can lead to problems when moving the camera. AndEngine works in a way that the HUD never really moves with the camera, so if we enable culling on HUD entities, then move our camera 800 pixels to the right (assuming our camera width is 800 pixels) our HUD entities would still physically respond as if there were in the proper position on our screen but they will not render. They would still react with touch events and other various scenarios, but we simply won't see them on the screen.

Using Relative rotation

Rotating entities relative to the position of other entities in 2d space is a great function to know. The use for relative rotation is limitless and always seems to be a 'hot topic' for newer mobile game developers. One of the most prominent examples of this type of rotation is in tower defense games, when the tower's turret rotates in order to follow the position of the enemies.

Getting started..

Refer to the `ClassCollection02` (`RelativeRotation.java`) for the working code for this topic. We'll need to include two images called '`marble.png`' and '`arrow.png`' included in our project's "`assets/gfx/`" folder.

How to do it...

Follow the steps below:

1. Implement the scene touch listener in the activity:

```
public class RelativeRotation extends BaseGameActivity implements
IOnSceneTouchListener{
```

2. Set the scene's `onSceneTouchListener` in the activity's `onCreateScene()` method:

```
mScene.setOnSceneTouchListener(this);
```

3. Populate the scene with the marble and arrow sprites. The arrow sprite is positioned in the center of the scene, whiling the marble's position is updated to wherever a touch event occurs.
4. This step introduces the `onSceneTouchEvent()` method which handles the movement of the marble sprite via finger movement on the device's display:

```
@Override
      public boolean onSceneTouchEvent(Scene pScene, TouchEvent
pSceneTouchEvent) {
            // If a user moves their finger on the device
            if(pSceneTouchEvent.getAction() == TouchEvent.ACTION_
MOVE){

// Set our marble's position to the touched area on the screen
                  mMarbleSprite.setPosition(pSceneTouchEvent.
getX(), pSceneTouchEvent.getY());
```

```
// Calculate the difference between the sprites x and y values.
                    final float dX = (mArrowSprite.getX()
+ (mArrowSprite.getWidth() / 2)) - (mMarbleSprite.getX() +
(mMarbleSprite.getWidth() / 2));
                    final float dY = (mArrowSprite.getY()
+ (mArrowSprite.getHeight() / 2)) - (mMarbleSprite.getY() +
(mMarbleSprite.getHeight() / 2)) ;

// We can use the atan2 function to find the angle
                    // Additionally, OpenGL works with degrees so
we must convert
// from radians
                    final float rotation = MathUtils.
radToDeg((float) Math.atan2(dY, dX));

// Set the new rotation for the arrow
                    mArrowSprite.setRotation(rotation);
                }
                return false;
        }
```

How it works...

In this class, we're creating an arrow (sprite) in the direct center of the screen which will automatically point to a marble (sprite). The marble is moveable via touch through the use of a touch scene listener registered to the `mScene` object. In situations where an entity rotates according to another entity's position, we'll have to include the rotation functionality in some method that is consistently updated otherwise our arrow would not continuously react. We can do this through update threads, but in this instance we'll include that functionality in the `onSceneTouchEvent()` overridden method.

In the first step, we're allowing our activity to override the `onSceneTouchEvent()` by implementing the `IonSceneTouchListener` interface. Once we've implemented the touch listener, we can allow the `Scene` object to receive touch events and respond according to the code situated inside the activity's overriden `onSceneTouchEvent()`.

In step four, the 'if' statement is comparing the current touch event via `pSceneTouchEvent.getAction()` with the constant `TouchEvent.ACTION_MOVE`. If `getAction()` returns the same value which corresponds to `ACTION_MOVE`, the marble's position is updated to the touch event coordinates and the arrow's rotation is calculated.

We first start by updating the marbles position to the location of touch via:

```
mMarbleSprite.setPosition(pSceneTouchEvent.getX(), pSceneTouchEvent.
getY());
```

Next we update the arrow's rotation by first calculating the difference in position via:

```
// Calculate the difference between the sprites x and y values.
final float dX = (mArrowSprite.getX() + (mArrowSprite.getWidth() / 2))
- (mMarbleSprite.getX() + (mMarbleSprite.getWidth() / 2));
final float dY = (mArrowSprite.getY() + (mArrowSprite.getHeight() /
2)) - (mMarbleSprite.getY() + (mMarbleSprite.getHeight() / 2)) ;
```

Notice that we are adding half our sprite's width and height. This is because AndEngine's `x` and `y` coordinates are positioned at the top left corner of our entities. Adding half the width and height of our entity will set the arrow to point to the center of our marble instead of the top left corner.

Lastly we must convert from radians to degrees due to the fact that OpenGL works with degrees. We will couple the conversion with the `Math.atan2()` method to calculate the angle (based on the previously gathered position differences) then continue to call `setRotation()` on our arrow:

```
final float rotation = MathUtils.radToDeg((float) Math.atan2(dY, dX));
mArrowSprite.setRotation(rotation);
```

Working with OpenGL

AndEngine provides its developers most of the functionality we need for 2d game development. Unfortunately, the fact is that we can't expect one general game engine to satisfy every game developer's needs for their designs. For those of us that are feeling a little bit more adventurous, we are fully capable of applying OpenGL capabilities to our AndEngine projects.

How to do it...

This code is a reference of the `Entity` classes `GLState` methods:

```
Rectangle rectangle = new Rectangle(WIDTH / 2 - 50, HEIGHT / 2 - 50,
          100, 100, mEngine.getVertexBufferObjectManager()) {

    @Override
    protected void preDraw(GLState pGLState, Camera pCamera) {

          super.preDraw(pGLState, pCamera);
    }

    @Override
    protected void draw(GLState pGLState, Camera pCamera) {
```

```
            super.draw(pGLState, pCamera);

        }

        @Override
        protected void postDraw(GLState pGLState, Camera pCamera) {

            super.postDraw(pGLState, pCamera);
        }

        @Override
        protected void applyTranslation(GLState pGLState) {

            super.applyTranslation(pGLState);
        }

        @Override
        protected void applyRotation(GLState pGLState) {

            super.applyRotation(pGLState);
        }

        @Override
        protected void applySkew(GLState pGLState) {

            super.applySkew(pGLState);
        }

        @Override
        protected void applyScale(GLState pGLState) {

            super.applyScale(pGLState);
        }
    };
```

How it works...

We're not going to go too much into detail of the inner-workings of OpenGL as it is outside the scope of this book. However, if you already have some knowledge on how to work with OpenGL or have future plans to learn, it will help to know where and when to access OpenGL's state.

In the above methods, we can use the `pGLState` object to apply more technical modifications to our entities. For example, the following code shows us how we can rotate an entity on the y-axis rather than the z-axis in order to apply a flipping effect to the entity:

```
@Override
protected void applyRotation(GLState pGLState) {

    // If rotation reaches 360 degrees, reset to 0
    if (this.mRotation >= 360)
           this.mRotation = 0;

    // Accumulate rotation
    this.mRotation = this.mRotation += 0.5f;

    if (this.mRotation != 0) {

           // Set the rotation center
           pGLState.translateModelViewGLMatrixf(this.mRotationCenterX,
                                                this.mRotationCenterY, 0);

           // Apply rotation to the entity on the x axis
           pGLState.rotateModelViewGLMatrixf(this.mRotation, 1, 0, 0);

           // Reset the entity to its proper position
           pGLState.translateModelViewGLMatrixf(-this.mRotationCenterX,
                                                -this.mRotationCenterY,
0);
    }
}
```

This code will be executed each time the entity is redrawn, increasing the rotation of the entity on the x-axis by `0.5f`. The two `translateModelViewGLMatrixf()` methods are in place to allow for the entity to rotate around a defined point in space. By default, the rotation center for our entities is the direct center of the object. The `rotateModelViewGLMatrixf()` method allows us to define the angle (in degrees) to rotate as well as set the axis to rotate on (x,y,z). In this snippet, we're choosing to rotate on the x-axis, which will appear as though the entity is flipping up/down. Applying the rotation on the y-axis will cause the entity to flip left/right. By default, rotation is applied on the z-axis which causes the rotation effect that we'd expect to see, much like turning a steering wheel.

On top of modifying the scale, skew, position and other general properties of our shapes, OpenGL allows us to enable capabilities for the `PGLState` which can help improve performance or improve quality. We're going to take a look at how to enable and use `GL_SCISSOR_TEST` in order to restrict rendering to specific coordinates and dimensions. One situation where this is useful is to disallow rendering of parts of a game background which might be covered by a mini-map.

```
@Override
protected void preDraw(GLState pGLState, Camera pCamera) {
```

```
        // Enable scissor test
        pGLState.enableScissorTest();

        // Restrict the entity from rendering outside the defined area
        GLES20.glScissor(0, 0, 100, 100);

        super.preDraw(pGLState, pCamera);
    }

    @Override
    protected void postDraw(GLState pGLState, Camera pCamera) {

        // We should disable GLStates we are finished using
        pGLState.disableScissorTest();

        super.postDraw(pGLState, pCamera);
    }
```

The scissor capability must be enabled before we can apply it to our entity. This should be done in the `preDraw()` method, followed by the defined position and area that should be *allowed* rendering. Calling `GLES20.glScissor(0, 0, 100, 100)` causes our entity to only render in the bottom left corner of the screen, up to 100 pixels wide and 100 pixels high. It should also be noted that OpenGL does not match our defined `WIDTH` and `HEIGHT` in terms of screen size. In order to scissor the full screen, it helps to know the physical display's full display size (see topic: "Introducing graphic-independent resolutions" in the first chapter).

When working with raw OpenGL, remember that the coordinate system is not the same as that in AndEngine. The 0,0 position in OpenGL is the bottom/left corner of the screen as opposed to AndEngine's top/left.

When accessing the `GLState` to make changes, it is good practice to enable capabilities in the `preDraw()` method and disable them in the `postDraw()` method. Make this a habit as it can otherwise lead to issues when OpenGL proceeds to draw another entity which might not reset the state automatically. As mentioned before, OpenGL is a state machine. Everything we enable when drawing one entity will remain that way (and apply to other entities!) until disabled.

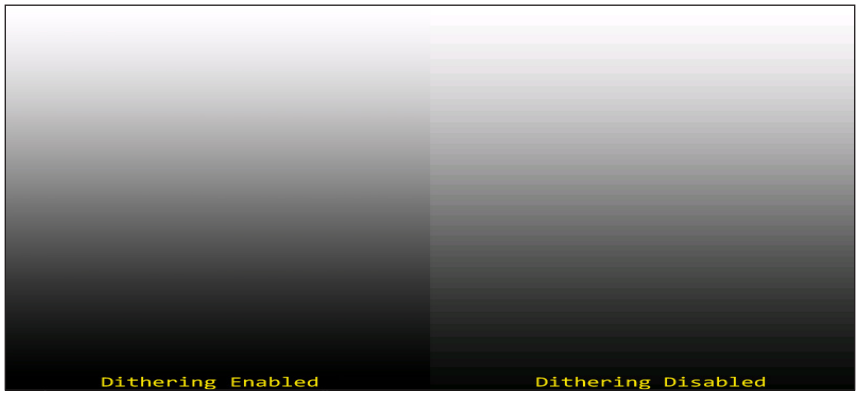
There's more...

When developing visually appealing games on the mobile platform, it is a likely scenario that we'll want to include some gradients in our images, especially when dealing with 2d graphics. Gradients are great for creating lighting effects, shadows, and many other objects we'd otherwise not be able to apply to a full 2d world. The problem lies in the fact that we're

developing for mobile devices which are unfortunately do not have an unlimited amount of resources at our disposal. Because of this, AndEngine down-samples the surface view color format to `RGB_565` by default. Regardless of the texture format we define within our textures, they will always be down-sampled before being displayed on the device. We could alter the color format that the surface view uses, but it's likely that the performance-hit will not be worth it when developing larger games with many sprites.

Using OpenGL's dithering capability

Below, we have two separate screen-shots; both textures are using the `RGBA_8888` texture format and `BILINEAR` texture filtering (the highest of quality).



The image on the right is applied to the scene without any modifications, while the left side has OpenGL's dithering capability enabled. The difference between the two otherwise identical images is immediately noticeable. Dithering is a great way for us to combat the down-sampling applied by the surface view without us having to rely on maximum color quality formats. In short, dithering low-levels of randomized noise within our images colors, resulting in the smoother finish which is found in the left image.

Enabling dithering is simple to apply to our entities in AndEngine, but as with everything, it's wise to pick and choose which textures apply dithering. The algorithm does add extra overhead, where if used too often could result in a larger performance loss than simply reverting our surface view to `RGBA_8888`. Below, we are enabling dithering in our preDraw() method and disabling it in our postDraw() method:

```
@Override
protected void preDraw(GLState pGLState, Camera pCamera) {

   // Enable dithering
   pGLState.enableDither();
```

```
        super.preDraw(pGLState, pCamera);
    }

    @Override
    protected void postDraw(GLState pGLState, Camera pCamera) {

        // Disable dithering
        pGLState.disableDither();

        super.postDraw(pGLState, pCamera);
    }
```

Dithering can be applied to any sub-type of AndEngine's `Shape` class (`Sprites`, `Text`, primitives, etc.).

For more information about OpenGL ES 2.0 and how to work with all of the different functions, visit the following link: `http://www.khronos.org/opengles/sdk/docs/man/`.

Overriding onManagedUpdate

Overriding our entity's `onManagedUpdate()` method can be extremely useful in all types of situations. By doing so, we can allow our entity to execute code every time the entity is updated through our update thread. There are so many possibilities including animating our entity, checking for collisions, producing timer events, and much more. Using our entity's `onManagedUpdate()` method also saves us from having to create and register new timer handlers for time/frame based events for a single entity.

Getting started..

`ClassCollection02` (`OverridingUpdates.java`) for the working code for this topic.

How to do it...

In this recipe, we're working with two different rectangle's in order to manipulate position, rotation, and apply collision-checking through the use of their update methods.

1. Overriding `onMangedUpdate()` for rotation (`rectangleOne`):

```
@Override
protected void onManagedUpdate(float pSecondsElapsed) {
Log.d("TIME", String.valueOf(pSecondsElapsed));
// Calculate rotation offset based on time passed
final float rotationOffset = pSecondsElapsed * 25;
```

```
// Adjust this rectangle's rotation
this.setRotation(this.getRotation() + rotationOffset);

// Pass the seconds elapsed to our update thread
super.onManagedUpdate(pSecondsElapsed);
}
```

2. Overriding `onManagedUpdate()` for position (`rectangleTwo`):

```
	@Override
protected void onManagedUpdate(float pSecondsElapsed) {

	// Adjust our rectangles position
	if(this.getX() < (WIDTH)){
		// Increase the position by 5 pixels per update
		this.setPosition(this.getX() + 5f, this.getY());
	} else {
		// Reset the rectangles X position and slightly
increase the Y position
		// If the rectangle exits camera view (width)
		this.setPosition(-RECTANGLE_WIDTH, this.getY() +
(RECTANGLE_HEIGHT / 2));
	}

	// Reset to initial position if the rectangle exits camera
view (height)
	if(this.getY() > HEIGHT){
		this.setPosition(0, 0);
	}
```

3. Shape collision checking:

```
if(this.collidesWith(rectangleOne) && this.getColor() != Color.
GREEN){
	this.setColor(Color.GREEN);

// If the shape is not colliding and not already red,
// reset the rectangle to red
} else if(this.getColor() != Color.RED){
	this.setColor(Color.RED);
}

// Pass the seconds elapsed to our update thread
super.onManagedUpdate(pSecondsElapsed);
}

};
```

How it works...

The first rectangle we create, we're overriding its update method to continuously update the rotation to a new value. The second rectangle, we're moving it from the far left side of the screen to the far right side of the screen. Once the second rectangle leaves the camera view, it is sent back to the left side and we lower the rectangle on the screen by 40 pixels. Additionally, when the two rectangles overlap, the moving rectangle will change colours until they are no longer touching.

The code in step one allows us to create events based on our entities update calls. In this specific overridden method, we're calculating a rotation based on the time elapsed since last update of our game occurred. Because the update thread fires so quickly (fractions of a second), we multiple `pSecondsElapsed` by 25 in order to increase the rotation speed a bit. Otherwise, we'd be rotating our entity about 0.01 degrees every update which would take quite a while. Use the `pSecondsElapsed` variable to your advantage when dealing with updates.

In step two, all we're doing is moving the rectangle from one side of the screen to the other, then once it reaches the opposite side, revert it back to its the initial `x` position and slightly increase the `y` position. This time around we are creating our new offset with a defined value of 5 pixels each update. This approach is considered bad practice as in most scenarios; offsets are based on time passed as they do not cause 'syncing' issues when a game slows down for whatever reason. Imagine having two rectangles changing positions every update, one via time passed since last update and one with a fixed value (Assuming at 60 FPS these two rectangles were moving at the same speed). A few hiccups in the game could spell disaster.

In the third and final step, we are checking to see on every update whether or not `rectangleTwo` is colliding with `rectangleOne` through the use of the `Shape` classes `collidesWith(shape)` method, where '`shape`' is the shape to check for collisions against. Once collision is detected, we check to see whether or not the color is already set to green. If it's already green, we let the update method finish running its course, otherwise we set the color to green.

There's more...

As briefly mentioned before, *all children receive the update call from their parent*. Child entities in this case also inherit the modified `pSecondsElapsed` value of the parent. We can even go as far as slowing our entire scene (including all of its children) down by overriding its `onManagedUpdate()` method and reducing the `pSecondsElapsed` value like so:

```
super.onManagedUpdate(pSecondsElapsed * 0.5f);
```

If we override our scene's `onManagedUpdate()` method, this would cause all entities attached to that scene to slow down by half in all aspects.

Using entity modifiers

AndEngine provides us with what are known as modifiers. Through the use of these modifiers (as briefly seen in the previous chapter's `SplashSceneActivity` class) we can apply neat effects and movements to our entities with ease. On top of that, we can include listeners and ease functions to these modifiers for almost full control over how they work, making them great for their purpose.

Getting started..

Refer to the `ClassCollection02` (`ApplyingEntityModifiers.java`) for the working code for this topic.

How to do it...

In this recipe, we're covering a range of different modifiers which can be used to modify the properties of our entities.

1. Setting up the parallel entity modifier:

```
ParallelEntityModifier parallelModifierA = new
ParallelEntityModifier(
       new QuadraticBezierCurveMoveModifier(4, 0, 0, WIDTH / 2,
HEIGHT / 2, WIDTH - RECTANGLE_WIDTH, 0),
       new SequenceEntityModifier(
       new RotationModifier(2, 0, -360),
       new RotationModifier(2, -360, 720)));
```

2. Setting up the sequence entity modifier:

```
SequenceEntityModifier sequenceModifier = new
SequenceEntityModifier(
       parallelModifierA,
       parallelModifierB,
       parallelModifierC,
       ModifierD);
```

3. Setting up the loop entity modifier:

```
LoopEntityModifier loopModifier = new LoopEntityModifier(sequence
Modifier);
```

4. Registering the modifiers to an entity:

```
rectangleOne.registerEntityModifier(loopModifier);
```

How it works...

In step one, we're creating a `ParallelEntityModifier` which contains a movement modifier running parallel to a `SequenceEntityModifier` containing two `RotationModifier's`. This recipe includes two other parallel entity modifiers similar to this one, as well as a `CardinalSplineMoveModifier`. These four modifiers handle the position, rotation, scale, and visibility of the rectangle applied in this recipe.

In step two, we're creating a `SequenceEntityModifier` which contains all of the `ParallelEntityModifiers`, causing them to run in sequence.

In step three, a `LoopEntityModifier` is created which will hold the `SequenceEntityModifier` created in step two. The purpose of this modifier is to continuously 'playback' the modifier sequence which ultimately causes the `ParallelEntityModifiers` in step one to continuously 'playback' as well. The reason for this is because we have setup the modifiers in a nested approach.

In this recipe we are setting up four rectangles in each corner of the screen which will continuously move from their own start position to their own end position through the use of the `LoopEntityModifier`. There are many different combinations of entity modifiers which we can apply to our entities. For the sake of keeping things simple, we're going to only explain the main modifiers which will give a general understanding of how to use all of them. One thing to know before we get into talking about the modifiers is that in order for our sprite to use them, we must call `entity.registerEntityModifier(modifier)` as mentioned in step four, where `entity` would be any shape, text, sprite, etc. and `modifier` would be the entity modifier we wish to apply. When we are finished with the modifier, we call `entity.unregisterEntityModifier()`.

In this recipe we are applying modifiers to our entities through the use of 'modifier containers' (this is not an official term, it just helps to explain their purpose). These modifier containers are known as the `LoopEntityModifier`, `SequenceEntityModifier` and `ParallelEntityModifier`. The purpose of these 'container modifiers' is to handle the more general aspects of the entity modifiers. The names of these modifiers explain their purpose well, but for the sake of being thorough let's go over them individually.

- The loop entity modifier simply allows our entity modifiers to loop 'playback'. We can allow this modifier to loop forever by solely passing the entity modifier as a parameter, or we can specify a maximum loop count via the `LoopEntityModifier`'s constructor like so:

```
// Create a loop modifier for our quadratic curve modifier
LoopEntityModifier loopModifierD = new LoopEntityModifier
(moveModifier, 5);
```

The first parameter is always the modifier to loop, while the second is the amount of times to loop. This code will execute `moveModifier` five consecutive times.

- The sequence modifier allows us to pass multiple entity modifiers to it, allowing them to play in sequence. This modifier has all sorts of uses, but one example is animation. In order for us to have `modifierA` play, then directly after `modifierA` finishes, play `modifierB`, we would use a `SequenceEntityModifier`. We can include as many different modifiers in this sequence as we'd like to:

```
// Create a sequence with 5 entity modifiers
SequenceEntityModifier sequenceModifier = new
SequenceEntityModifier(
      new MoveYModifier(1.5f, 0, HEIGHT - RECTANGLE_HEIGHT),
      new MoveYModifier(1.5f, HEIGHT - RECTANGLE_HEIGHT, 0),
      new RotationModifier(1.5f, 0, 360),
      new AlphaModifier(1.5f, 1, 0),
      new AlphaModifier(1.5f, 0, 1));
```

In this snippet, we're creating a sequence of five modifiers. The sequence would play from the first modifier added through to the last (the first being the move modifier, the last being alpha modifier).

- The parallel modifier is very much like the sequence modifier. It allows us to provide a list of entity modifiers that will play, except in this case rather than running in consecutive order, they will run parallel with each other (they will all run at the same time). Apply the sequence modifier above to an entity and run the application, then change the word "sequence" to "parallel" and see what happens.

Now that we've listed the main entity modifiers, let's talk about the modifiers which actually affect the entity. The list of modifiers that physically manipulate the entities is quite large so we won't cover all of them in detail. Thankfully, once you are aware of the way a few of them work, the knowledge is pretty much transferable to other modifiers of different types. Modifiers mostly follow the rule, `Modifier(duration, from, to)`, where `duration` is the length it takes to achieve the full modifier effect, `from` is the initial value of the property being modified, and `to` is the end value. In some cases, the modifier might allow for 'pathing' which allows us to link multiple from-to coordinates in a single modifier (such as `QuadraticBezierCurvedMoveModifier`).

- The first and probably most widely used modifier is the `MoveModifier`:

```
MoveModifier moveModifier = new MoveModifier(5, 0, 200, 0, 300);
```

This snippet will create a movement modifier which moves the entity from coordinates 0 to 200 on the a-axis and 0 to 300 on the y-axis. Note that the parameters follow the "duration-from-to" pattern even after adding a second position (for the y-axis).

- Next up is the `RotationModifier`:

```
RotationModifier rotationModifier = new RotationModifier(3,
0, 360);
```

Here, we're creating a rotation modifier which will rotate the entity from 0 (default) to 360 degrees in three seconds.

- The final modifier we are going to include in this topic is to give an understanding of how to allow our entities to follow a set path. This approach is a little bit different in terms of creating our 'from-to' points. This modifier requires us to pass an object called a 'Path' which stores x and y coordinates for the entity to follow. Let's see how we can create a path modifier, starting an entity at the top left of the screen and moving clockwise, touching the other three corners of the screen and returning to the initial position(assuming our camera width and height is 800(w) by 480(h)):

```
// Path X coordinates
float xCoords[] = {
             0, // x1
             800 - RECTANGLE_WIDTH, // x2
             800 - RECTANGLE_WIDTH, //x3
             0, //x4
             0}; //x5

// Path Y coordinates
float yCoords[] = {
             0, //y1
             0, //y2
             480 - RECTANGLE_HEIGHT, //y3
             480 - RECTANGLE_HEIGHT, //y4
             0};//y5

// Create our path object with coordinates
Path path = new Path(xCoords, yCoords);

// Create the modifier with our new path
// and a duration of 10 seconds
PathModifier pathModifier = new PathModifier(10, path);
```

We can see here that we're creating two float arrays, one for x coordinates and one for y coordinates. I've numbered the specific coordinates each x1, y1, x2, y2, etc. The x and y coordinates with matching numbers would be read together via the path modifier. Additionally, *every x coordinate needs a corresponding y coordinate and vice versa*! If we have six floats for x and only five for y, our application will crash with an `IllegalArgumentException`.

Once we plot our points, we simply create a new `Path` object and pass our new coordinates. Finally we pass our `path` to the `PathModifier` object with our defined duration (10 seconds in this case) and register it to our entity. We now have an entity which follows a defined path around the screen.

There's more...

Before moving on to the next topic, we should take a look at the extra features included for entity modifiers. There are two more parameters that we can pass to our entity modifiers which we haven't discussed yet; those being modifiers listeners and ease functions. These two classes can help to make our modifiers more customized than simply applying general modifiers to our entities.

The `IEntityModifierListener` can be used in order to fire events when a modifier starts and when a modifier finishes. In the following snippet, we're simply printing logs to logcat which notify us when the modifier has started and finished.

```
IEntityModifierListener listener = new IEntityModifierListener(){

    // When the modifier starts, this method is called
    @Override
    public void onModifierStarted(IModifier<IEntity> pModifier,
                IEntity pItem) {
          Log.i("MODIFIER", "Modifier started!");
    }

    // When the modifier finishes, this method is called
    @Override
    public void onModifierFinished(final IModifier<IEntity> pModifier,
                final IEntity pItem) {
          Log.i("MODIFIER", "Modifier started!");
    }
};

moveModifier.addModifierListener(listener);
```

The above code shows the skeleton of a modifier listener with basic log outputs. In more relative scenarios, we could call `pItem.detachSelf()` in order to remove the object from the game. This could be useful for handling subtle falling leaves or raindrops in a game scene. Obviously the usages are not limited to this, though.

Finally, we'll quickly discuss the ease functions in AndEngine. Ease functions are a great way to add an extra "transitioning" layer to our entity modifiers. After having used many modifiers in my time with AndEngine, I can't find many situations where I don't like to add ease functions to my modifiers. The best way to explain an ease function is to think about a game where the menu buttons fall from the top of the screen and 'bounce' in to place. The bounce in this case would be our ease function.

```
// Create a loop modifier
LoopEntityModifier loopModifierA = new LoopEntityModifier(new
MoveXModifier(3, 0, WIDTH, EaseElasticIn.getInstance()));
```

The code above is stolen from the `ApplyingEntityModifiers` class so that you can simply plug-n-play this code snippet. As we can see here, using ease functions is very easy . Often the hardest part is choosing which one to use as the list of ease functions is quite large. Take some time to look through the various ease functions provided by locating the `org.andengine.util.modifier.ease` package. Simply replace `EaseElasticIn` from the code above with the ease function you'd like to test out!

Ease Function Reference

Download the "**AndEngine – Examples**" application from Google Play to your device. Open the application and locate the "**Using EaseFunctions**" example. To this day, I still use this example app in order to decide best which ease function to apply to my modifiers.

Advanced entity modifiers

This topic, similar to the previous topic, discusses the use of entity modifiers in order to manipulate the movement and appearance of entities. However, in this topic things get a little more tricky as a relatively large list of modifiers are being applied to a single rectangle in a synchronized fashion in order to provide an enhanced movement effect, looping continuously.

Getting started..

Refer to the previous `ApplyingEntityModifiers.java` class and overwrite the `onPopulateScene()` method with the code below.

How to do it...

1. First sequence:

```
// Movement from top left to top right
ParallelEntityModifier parallelModifierA = new
ParallelEntityModifier(
             new QuadraticBezierCurveMoveModifier(4, 0, 0, WIDTH /
2, HEIGHT / 2, WIDTH - RECTANGLE_WIDTH, 0),
             new SequenceEntityModifier(
             new RotationModifier(2, 0, -360),
             new RotationModifier(2, -360, 720)));
```

2. Second sequence:

```
// Movement from top right to bottom right
ParallelEntityModifier parallelModifierB = new
ParallelEntityModifier(
```

```
                new MoveYModifier(3, 0, HEIGHT - RECTANGLE_HEIGHT,
EaseBounceOut.getInstance()),
                new SequenceEntityModifier(
                new ScaleModifier(0.7f, 1, 0.3f),
                new DelayModifier(2f),
                new ScaleModifier(0.7f, 0.3f, 1, EaseQuartIn.
getInstance())));
```

3. Third sequence:

```
// Movement from bottom right to bottom left
ParallelEntityModifier parallelModifierC = new
ParallelEntityModifier(
                new JumpModifier(1, WIDTH - RECTANGLE_WIDTH, 0,
HEIGHT - RECTANGLE_HEIGHT, HEIGHT - RECTANGLE_HEIGHT, 100),
                new SequenceEntityModifier(
                new FadeOutModifier(0.3f),
                new FadeInModifier(0.7f)));
```

How it works...

The code above demonstrates the more diverse applications of entity modifiers. Rather than creating four separate rectangles which each have simple modifiers, we're creating a single rectangle which moves around the screen in four different sequences with multiple modifiers per sequence. The code might look confusing if we're looking at everything as one big modifier, but the trick is to always break them down into smaller categories. Let's go ahead and do that now by visiting each sequence individually.

Step one introduces a sequence modifier within a parallel modifier. This can be read similar to the move modifier, where quadratic will move from x1 to x2. Parallel to that movement, the sequence modifier will be able to move from modifier1 to modifier2. All modifiers which are contained as sequence modifier parameters will not be affected by parallelism, following the "from-to" pattern so to speak.

Step two introduces ease functions to a couple of the modifiers. The ease functions can be placed as the final parameter when creating a new modifier. Additionally, we're applying a sequence of scale modifiers which will run parallel to the movement modifier, shrinking the rectangle then enlarging it with a 2 second delay in between.

In the third sequence, we're implementing a jump modifier which has made life a lot easier for development of 2d platform developers. We can set the starting x and y coordinates, the end x and y coordinates, and the height that the entity will jump. We've included a fade out/fade in modifier to run parallel to the jump for a neat looking 'ninja' effect.

Due to the spline modifiers being a little more complex, I've included this sequence as a single modifier (`CardinalSplineMoveModifier`). The setup for this modifier is similar to the `PathModifier` discussed in the previous topic except this modifier allows us to add tension to the movement, resulting in smooth curves upon reaching a waypoint. The first thing we must do is create a configuration for this type of modifier:

```
// Create our spline movement config
CardinalSplineMoveModifierConfig config = new
CardinalSplineMoveModifierConfig(4, -1);
```

The first parameter in the snippet above defines the number of waypoints for the path. The second allows us to apply tension to the entity's movement. The outcome of this tension will be either a straight motion from one waypoint to another (tension set to 1), or curved movement (tension set to -1). We can set the tension anywhere between those two values.

The second step for this type of modifier is to setup the waypoints. We can do this the same way we'd created waypoints for the `PathModifier` in the previous topic.

```
// Setup the x coords for spline movement
final float xCoords[] =
{
          0, // x1
          WIDTH /2, // x2
          WIDTH - RECTANGLE_WIDTH, // x3
          0 // x4
};

// Setup the y coords for spline movement
final float yCoords[] =
{
   HEIGHT - RECTANGLE_HEIGHT, // y1
   0, // y2
   HEIGHT - RECTANGLE_HEIGHT, // y3
   0 // y4
};
```

Once we have our waypoint float arrays, we need to take an additional step. We need to plug these waypoints into our `CardinalSplineMoveModifierConfig` object (config) via:

```
// Apply the x and y coords to the config
for(int i = 0; i < xCoords.length; i++){
   config.setControlPoint(i, xCoords[i], yCoords[i]);
}
```

We can use a loop in order to apply the array's coordinates via the `setControlPoint()` method, where `i` is the index.

Finally, we create the modifier with a duration of 10 seconds and include our new `config` object for our waypoints.

```
// Setup the cardinal spline modifier with the above config
CardinalSplineMoveModifier ModifierD = new
CardinalSplineMoveModifier(10, config);
```

4. Creating the main sequence modifier:

```
// Setup our modifiers to fire in sequence
SequenceEntityModifier sequenceModifier = new
SequenceEntityModifier(
            parallelModifierA,
            parallelModifierB,
            parallelModifierC,
            ModifierD);

// Setup the modifier sequence to continuously loop
LoopEntityModifier loopModifier = new LoopEntityModifier(sequenceM
odifier);

// Register the modifier loop to the rectangle
rectangleOne.registerEntityModifier(loopModifier);
```

This step is pretty straight-forward. In order to play all of the previous sequences in order, we must add them to a `SequenceEntityModifier`. Since we want them to continuously loop once the final sequence has ended, we will then add the `sequenceModifier` object to a `LoopEntityModifier` and we can finish by registering the loop modifier to our rectangle.

See also

- Using entity modifiers – Chapter 2

Working with particle systems

Particle systems can provide our games with very attractive effects for many different events in our games. In this chapter we're going to take a look at how we can setup our particle's with some of the more basic features of AndEngine's particle system.

Getting started..

Refer to the `ClassCollection02` (`WorkingWithParticles.java`) for the working code for this topic.

How to do it...

Creating a particle system with AndEngine requires us to create a few separate objects which work together. These objects include the `ParticleSystem`, `ParticleEmitter`, and various particle initializers and particle modifiers.

1. Create the particle emitter:

```
PointParticleEmitter particleEmitter = new PointParticleEmitter(0,
0)
```

2. Create the particle system:

```
SpriteParticleSystem particleSystem = new SpriteParticleSystem
(particleEmitter, 10, 20, 200, mParticleTextureRegion, mEngine.
getVertexBufferObjectManager());
```

3. Adding particle initializers and modifiers:

```
particleSystem.addParticleInitializer(new VelocityParticleInitiali
zer<Sprite>(-50, 50, -10, -400));
particleSystem.addParticleModifier(new ScaleParticleModifier<Spri
te>(3, 6, 0.2f, 2f, 0.2f, 2f));
```

4. Positioning and attaching the particle system:

```
particleEmitter.setCenter(WIDTH / 2, HEIGHT);
mScene.attachChild(particleSystem);
```

How it works...

Particle systems in AndEngine include a wide range of customizable initializers and modifiers which we can apply to each particle, allowing us full control over the particles spawned. However, AndEngine's particle systems also allow us to set minimum and maximum values in situations where we wish for more randomized particle effects.. In this activity, we're creating a simple particle system with a couple of particle initializers and a single particle modifier. The end result involves particles being thrown into the air in random directions, growing as time passes. A texture is needed for this particular particle system, but we're going to exclude that step from the explanation (see the code snippet above for the texture implementation):

The first step involves creating our particle emitter. The particle emitter's main purpose is to handle the location that the particles will spawn, as well as the pattern. AndEngine includes a few different patterns which can allow us to spawn particles in circular shapes, rectangular shapes, and single points. In this activity we are creating a `PointParticleEmitter`, where the parameters are the coordinates for particle spawning (top left initially):

Step two is to create the `ParticleSystem`. When creating a sprite particle system, the first parameter we'll pass to it is the particle emitter. The second and third parameters define the minimum and maximum spawn rate (how slow/fast the particles will spawn), followed by the maximum particle count that our system will be allowed to display at one time. The final two parameters are similar to a sprite, where we pass a texture region for the particle system to use as well as our vertex buffer object manager

The third step is to decorate the particle system with various particle initializers and particle modifiers. Particle initializers and modifiers play the most vital role in how our particles will act once they are spawned. With these objects, we can define our particles scale, color, alpha, rotation, expiration time, velocity, acceleration and a few more properties. Initializers handle the properties of the particle when it is first spawned, while modifiers handle particles effects over a period of time. We can define strict values for these modifiers in order to apply a consistent look to our particles. Otherwise we can choose to take advantage of 'from-to' constructors while creating our initializers/modifiers in order to create more random particle effects.

We can call the `addParticleInitializer()` method on a particle system in order to add new initializers. In this recipe, we're setting up initializers to deal with the particles movements as well as the expire time. The velocity initializer takes advantage of randomized values. The first and second parameters are min/max x-axis values, the third and fourth are min/max y-axis values, in that order. The coordinates of the movement initializers/modifiers on particles is relative to *its own position, not world coordinates*. This means that if we create a particle modifier with an x value of -1, the particle will move to the left. If we create the modifier with a value of 1, the particle will move to the right.

The second initializers is an expire initializer, used to handle the lifetime of each individual particle. In this case, the particles will last anywhere between 10 and 15 seconds.

We can call the `addParticleModifier()` method on a particle system in order to add new modifiers. Much like entity modifiers, particle modifiers deal with parameters on a 'from-to' basis. However, unlike entity modifiers, with particle modifiers we can also define a 'from-to' duration in which the modifier will take effect. In the scale modifier we are using for this particle system, we can read it as "3 seconds after the particle is spawned, the particle will scale from 0.2f to 2f in 3 seconds". The particle modifier's duration can be found by subtracting the "to" time from the "from" time (six minus three in this case).

The final step involved in setting up the particle system is to set the position and attach it to the scene. We set the center to the bottom/center location of our device's display by calling `setCenter()` on the particle emitter. Finally, we attach the particle system to the scene as we would an ordinary entity. We're now ready to watch our particles come to life on-screen!

There's more...

Below is a list of the different types of particle initializers and modifiers we can add to our particle systems. Feel free to replace the current setup with different combinations of initializers/modifiers.

The particle initializers include:

- AccelerationParticleInitializer
- AlphaParticleInitializer
- BlendFunctionParticleInitializer
- ColorParticleInitializer
- GravityParticleInitializer
- RotationParticleInitializer
- ScaleParticleInitializer
- VelocityParticleInitializer

Additionally, we can add a class called `IParticleInitializer` to our particle systems. These allow us to apply our own custom changes to the particles on time passed, or on initialization:

```
particleSystem.addParticleInitializer(new
IParticleInitializer<Sprite>(){

    @Override
    public void onInitializeParticle(Particle<Sprite> pParticle) {
           // We can add our own code here
    }
});
```

The particle modifiers include:

- AlphaParticleModifier
- ColorParticleModifier
- ExpireParticleModifier
- OffCameraExpireParticleModifier
- RotationParticleModifier
- ScaleParticleModifier

Once again, we can add our own custom particle modifiers by adding an IParticleModifier to the system:

```
particleSystem.addParticleModifier(new IParticleModifier<Sprite>(){

    @Override
    public void onInitializeParticle(Particle<Sprite> pParticle) {
          // We can add our own code here which runs on initialization
    }

    @Override
    public void onUpdateParticle(Particle<Sprite> pParticle) {
          // We can add our own code here which will run constantly

    }

});
```

Creating Advanced particle systems

When creating more complex particle systems, it's likely that the pre-built particle initializers and modifiers won't quite cut it. In this topic, we're going to take a look at creating our customizable particle modifiers through the use of the particle modifier interface (`IparticleModfier`). With this interface, we can add more detailed color modifiers (randomly changing colors), more diverse movement effects, and even add physics to our particles! These are only a few of the modifications we can make to AndEngine's particle systems.

Getting started..

Refer to the `ClassCollection02` (`AdvancedParticleSystems.java`) for the working code for this topic.

How to do it...

This recipe is going to cover how we can make custom modifications to each particle as it spawns, as well as allow us to make modifications to it on each update.

1. Creating the movement modifier:

```
public static final float HEIGHT_OFFSET = HEIGHT / 8;

// Modifier x coordinates
private final float pointsX[] = {
             WIDTH / 2 - 90, // x1
             WIDTH / 2 + 90, // x2
             WIDTH / 2 - 180, // x3
             WIDTH / 2 + 180, // x4
```

```
                WIDTH / 2 - 40, // x5
                WIDTH / 2 + 40, // x6
                WIDTH / 2 - 100, // x7
                WIDTH / 2 + 100  // x8
                };

// Modifier y coordinates
private final float pointsY[] = {
                HEIGHT - (HEIGHT_OFFSET * 1), // y1
                HEIGHT - (HEIGHT_OFFSET * 2), // y2
                HEIGHT - (HEIGHT_OFFSET * 3), // y3
                HEIGHT - (HEIGHT_OFFSET * 4), // y4
                HEIGHT - (HEIGHT_OFFSET * 5), // y5
                HEIGHT - (HEIGHT_OFFSET * 6), // y6
                HEIGHT - (HEIGHT_OFFSET * 7), // y7
                HEIGHT - (HEIGHT_OFFSET * 8), // y8
                };
```

2. Creating a BatchedSpriteParticleSystem:

```
BatchedSpriteParticleSystem particleSystem = new
BatchedSpriteParticleSystem(
                particleEmitter, 1, 2, 20, mParticleTextureRegion,
                mEngine.getVertexBufferObjectManager());
```

3. Applying particle initializers and modifiers to a batch particle system:

```
particleSystem.addParticleInitializer(new ColorParticleInitializer
<UncoloredSprite>(0, 1, 0, 1, 0, 1));
```

4. Adding custom particle modifiers:

```
particleSystem.addParticleModifier(new IParticleModifier<Uncolored
Sprite>() {

      @Override
      public void onInitializeParticle(Particle<UncoloredSprite>
pParticle) {
                // Do something when new particle is spawned
      }

      @Override
      public void onUpdateParticle(Particle<UncoloredSprite>
pParticle) {
                // Do something every time a particle is updated
      }

});
```

How it works...

In this particular recipe, we're creating a batch particle system which follows a modifier path while continuously changing the colors of each individual particle to pseudo-random color values. The final outcome appears to be a spiraling, rainbow colored particle system which appears to be 'floating' toward the top of the screen. On top of that, we're creating a sequence of scale modifiers which shrink and grow the particles, allowing for an illusion of depth to our system.

The `BatchedSpriteParticleSystem` is equivalent to the `SpriteGroup` class. Rather than drawing each individual particle to the scene, AndEngine 'batches' the entire particle count into a single OpenGL draw call, resulting in a (roughly) %50 performance increase, as stated by Nicolas Gramlich, the lead developer of AndEngine. There *may* be some noticeable color differences from the original `SpriteParticleSystem` when using the batched particle system.

The first step we take into setting up this particle system is plotting our modifier points. These modifier points will be applied as a path for each particle that spawns. The `HEIGHT_OFFSET` variable is in place to divide our maximum screen height into 8 different segments. We'll use these segments when plotting the y coordinates, incrementing the multiplication by 1. This will set each coordinate 1/8th (in pixels) higher on the screen for each subsequent point. For the x movement, we're simply moving left to right (from the center of the screen).

In step two we're going to be creating a `BatchedSpriteParticleSystem`. Creating a `BatchedSpriteParticleSystem` is no different than creating a `SpriteParticleSystem`, aside from the class name. The parameters remain the same. For a roughly %50 improvement in particle performance, it's a fair trade.

In the third step, we're setting up an initializer for our `BatchedSpriteParticleSystem`. One thing you may notice that we do differently in the batch particle system's case, is that we must supply the `UncoloredSprite` type rather than a `Sprite`. We won't be able to build our project otherwise.

In the final step for this recipe, we're applying a customizable particle modifier to the particle system by passing a new `IParticleModifier` interface as a `ParticleModifier` parameter. The possibilities in this type of situation are endless, from applying entity modifiers to each individual particle, to even being able to apply physical Box2d bodies to the particles. In this recipe, we're choosing to apply a CardinalSplineMoveModifier to the particles as they spawn, causing them to float upward while quickly flashing random rainbow colors. The onInitializeParticle() method might look something like this:

```
@Override
public void onInitializeParticle(Particle<UncoloredSprite> pParticle)
{
// Create our movement modifier
CardinalSplineMoveModifier moveModifier = new
```

```
CardinalSplineMoveModifier(10, mConfig);

// Register our modifier to each individual particle
pParticle.getEntity().registerEntityModifier(moveModifier);
}
```

See also

- Working with particle systems – Chapter 2