B
Additional Recipes

Appendix B, *Additional recipes*, includes a select few recipes that take advantage of some of the features of AndEngine which are less likely to be included in the average development process. These recipes include:

- Applying an FPS counter for development
- Adding a screen capture function to our games
- Working with OpenGL

Applying an FPS Counter for development

Obtaining the frames-per-second that our applications are achieving is an important part of the development process in order to accurately gauge performance. By applying an `AverageFPSCounter` object to our engine, we can output an accurate FPS count to the LogCat. This is useful during the optimization process of applications in order to make the most of our games for a wider range of devices.

How to do it...

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

Create the `AverageFPSCounter` object:

```
        /* Create the AverageFPSCounter with a duration of 1 second */
        AverageFPSCounter mAverageFPSCounter = new AverageFPSCounter(1) {

          /*
           * This method will be called every x amount of seconds,
    depending
```

```
         * on the value passed within the AverageFPSCounter's
constructor
         */
        @Override
        protected void onHandleAverageDurationElapsed(float pFPS) {

          /* Output the application's FPS to LogCat */
          Log.i("FRAMES PER SECOND", "FPS: " + pFPS);
        }

    };
Register the AverageFPSCounter to our mEngine object:
    mEngine.registerUpdateHandler(mAverageFPSCounter);
```

How it works...

As we can see, setting up our game to monitor how many frames per second we are achieving is a simple task.

In the first step, we must create the `AverageFPSCounter` object. The parameter will define how often the `onHandledAverageDurationElapsed(pFPS)` method receives the updated value of our game's FPS. Within this method, we're simply posting the obtained `pFPS` value to the LogCat.

In the second step, we are required to register the `AverageFPSCounter` as an update handler in order for the `mEngine` object to notify the `AverageFPSCounter` of frame updates as well as how many seconds have passed after each update. The FPS is then calculated by dividing the the number of frames passed by the number of seconds it took to process those frames. For example, if our application receives 37 frame updates which take 1.6 seconds to complete, we'd be running at about 23 frames per second.

Adding a screen capture function to our games

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. Let's find out how we can incorporate screen capture functionality into our own games.

Getting ready...

In order to allow a device to write the images captured to the same user's device, we must include the necessary permission in the `AndroidManifest.xml` of our project. Add the following code snippet to the `AndroidManifest.xml` of the project you wish to add screen capturing to, then refer to the class named `ScreenCap` in the code bundle :

```
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_
STORAGE" />
```

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. Next, we must obtain the device's display size in order to tell the `ScreenCapture` object the proper width and height values to capture. Create two global `int` variables called `mDisplayWidth` and `mDisplayHeight`, then import the following code into the `onCreateEngineOptions()` method of the `BaseGameActivity`:

```
    WindowManager windowManager = (WindowManager)
getSystemService(Context.WINDOW_SERVICE);
    Display display = windowManager.getDefaultDisplay();

    /* Obtain the API level of the device running the game */
    int api = android.os.Build.VERSION.SDK_INT;

    /*
     * We're dealing with deprecated methods, so we filter older
devices
     * (less than api 13) to use the older methods, while the new
API levels
     * will use the non-deprecated methods in ord to obtain the
device's
     * display size
     */
    if (api >= 13) {
      Point point = new Point();

      /* Pass the display size to the point object */
      display.getSize(point);

      /*
```

```
        * pass the device's display size to our width/height
variables to
        * capture
        */
       mDisplayWidth = point.x;
       mDisplayHeight = point.y;
     } else {

       /*
        * If API level is less than 13, revert to using the
deprecated
        * methods used for grabbing the device's display size
        */
       mDisplayWidth = display.getWidth();
       mDisplayHeight = display.getHeight();
     }
```

3. In the third step we will attach our `mScreenCapture` object to the `Scene` just as we would any other `Entity`:

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

4. Step four requires us to ensure that we've got a folder in place for us to write the captured image files to:

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

5. Once we've made sure that we've got the necessary folder to write to, we can call the `capture()` method on the `ScreenCapture` object. At this point, the screen shot will either be captured, or fail if there are any problems:

```
/* Capture the entire screen screen */
mScreenCapture.capture(mDisplayWidth, mDisplayHeight,
  FileUtils.getAbsolutePathOnExternalStorage(this, "name"
      + ".png"), new IScreenCaptureCallback() {

    /* This method is called if the screen capture was
     * successful */
    @Override
    public void onScreenCaptured(String pFilePath) {

    /* Display that the capture was successful in LogCat and
    * print the path to the file */
    Log.i(TAG, "Successfully saved to: " + pFilePath);
    }
```

```
            /* This method is called if the screen capture failed */
            @Override
            public void onScreenCaptureFailed(String pFilePath,
                Exception pException) {

             /* Display that the capture has failed in LogCat along
              * with the exception thrown */
            Log.e(TAG, pFilePath + " : " + pException.
getLocalizedMessage());
            }
          });
```

How it works...

In the first step, we are creating a global `ScreenCapture` object. The `ScreenCapture` object will be used to capture the screen in a later step.

In step two, we move to the `onCreateEngineOptions()` method of the `BaseGameActivity` class where we obtain the device's display size. To do this, we are using the `WindowManager` class to obtain the default display object, which we use to obtain the display size of the device. However, before we can obtain the display size we must determine which method to use depending on the API level of the device running our game. The `display.getWidth()` and height variant are now deprecated and we should use a `Point` object to obtain the width and height of the screen, passing the obtained values to our `mDisplayWidth` and `mDisplayHeight` variables through the use of `point.x` and `point.y`. If the device is running on API levels 8 through 12, then we revert back to using the deprecated methods.

In the third step we will visit the `onCreateScene()` method of the `BaseGameActivity`. In this method, we must attach our `Entity` objects to the `Scene` before we attach our `mScreenCapture` object to the `Scene`. The `ScreenCapture` object will only capture the objects which have been attached to the `Scene` before it. For example, if we happened to attach the `Rectangle` object to the `Scene` after the `ScreenCapture` object in this recipe, the captured image would appear as if it were an empty `Scene`.

Step four is in place to assure the application that the required folder is available to write images to. Without this step, it is very likely that the capture would fail as we would not have a properly designated folder to store the captured images. Additionally, we must declare the permission as stated in the *Getting started...* section of this recipe, otherwise we would not be able to create the necessary folder or save the image to the folder. The `this` parameter refers to the file path, `Android/data/com.application.package/files/`, where `com.application.package` would refer to the application's unique package name. The same rule applies for the `FileUtils` method call in the next step.

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 on the display that we wish to capture, which are defined via the `mDisplayWidth` and `mDisplayHeight` values that obtained the device-specific screen dimensions in step two. For the third parameter we are defining the location in which to store the image, as well as the file name of the image itself. Lastly, we have to include an `IScreenCaptureCallback()`. We aren't obligated to include any code in the callback methods, but it is recommended to at least notify the user as to whether the screen capture was a success or a failure.

Working with OpenGL

AndEngine provides its developers with a vast array of functionality needed for complex 2D game development. However, the fact remains 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 can apply OpenGL capabilities to our AndEngine projects. This recipe will explore some of the options we have for making lower level adjustments to certain aspects of our game with OpenGL.

How to do it...

This code is a reference of the `Entity` classes' `GLState` methods. In order to visualize the effects applied through the use of these methods, we can use a `Rectangle` object since it can be displayed on the `Scene`, unlike the average `Entity` object:

```
Rectangle rectangle = new Rectangle(WIDTH * 0.5f, HEIGHT * 0.5f,
    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 into too much detail concerning the inner-workings of OpenGL as it is outside the scope of this book. However, if you already have some knowledge of 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` object on the y-axis rather than the z-axis in order to apply a flipping effect to the `Entity` object:

```
  @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 += 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 `GLState` class, which can help improve performance or improve visual 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 to render. 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's coordinate system is based on the device's display and not the size of AndEngine's `Camera` object. For that reason, the `glScissor()` method's parameters should be based on the device's display size.

When accessing `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, applying the same capabilities for all other entities until manually disabled.