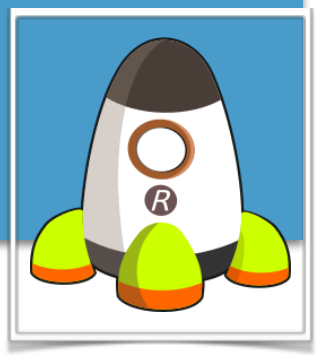
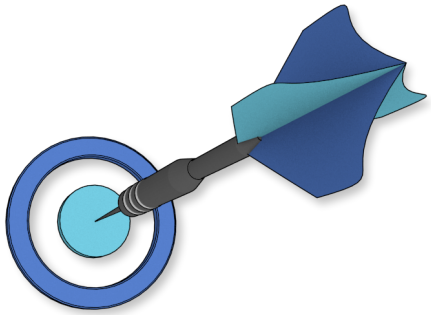




Beginning Ranger-Dart

A Hands-On Guide to Building Games with Ranger-Dart

William DeVore

Preface

Welcome to **Ranger-Dart** game development (or **Ranger** for short.)

Developing games in Ranger is easy once you learn the basic ins and outs. But there is much to learn starting with Dart the language all the way to the HTML5 Canvas and beyond.

Most developers, including me, look for ways to reduce and/or simplify our game coding experience. Game engines are one such approach. It is true you can write a tight game-loop, transformation stack and Canvas code but in the end you generally end up creating a ton of infrastructure code to ease the use of your own code.

With Ranger you can minimize your work by reducing the need to deal directly with timing, matrices and Canvas. Ranger's framework can help you focus your efforts as close to the game mechanics as possible—and it is free and open source as well.

This book's goal is to teach and guide you in creating Ranger based games. It takes you step by step in the process of creating a game called Moon Lander (ML). Along the way we cover topics designed to help you make decisions about game design, and in the process you gain real experience with the most important aspects of Ranger.

Ranger's goal isn't to be an enforcer but an enabler. At any time you can choose to use as much or as little of Ranger as you choose. As a matter of fact because Ranger is FOSS you can ultimately create your own game engine styled specifically to your own requirements.

What is Ranger?

Ranger is technically two Dart projects: **Ranger-Dart** and **Ranger-Sack** (https://github.com/wdevore) both of which are **FOSS** and written in the Dart language. When coding in Ranger you will always use the Ranger-Dart library but reference Ranger-Sack for examples and templates.

Originally Ranger was a partial port of Cocos2D-js 1.x but eventually was rewritten from scratch to take better advantage of Dart's programming patterns, out-of-the-box Pub packages and runtime framework.

Ranger current uses an **HTML5** Canvas as its rendering surface, but plans are afoot to include **WebGL** as a target as well, but Ranger is more than just rendering, it includes a game-loop based on updates-per-second, object pooling, audio, and a scene graph all of which you will learn as you progress through this book.

Ranger's Key features

Ranger comes with a host of features that can reduce your coding: tween animations, vector math, message system, particle systems just to name a few.

Tween animations

Animations give flare to any game. They help Scenes transitioning in and out, Nodes spinning and fading and more to punch it up a notch. Ranger's animations are provided by the **TweenEngine** library ported to Dart from the Java Universal Tween Engine by Xavier Guzman.

Lets take a quick look at an animation applied to a typical Node that is shaped like an airplane with landing gear. In just these few lines of code we extend the landing gear:

```
void extend() {
  Ranger.Application app = Ranger.Application.instance;
  app.animations.tweenMan.killTarget(this, TWEEN_ROTATE);
  parent.visible = true;
  UTE.Tween rotateTo = new UTE.Tween.to(this, TWEEN_ROTATE, 2.0)
    ..targetValues = [180.0]
    ..easing = UTE.Linear.INOUT;

  app.animations.add(rotateTo);
}
```

Not much code there. The animation engine rotates the “this” node 180 degrees at a linear rate and out pops the landing gear that was invisible while hiding behind the airplane Node. You will learn how to do this when we starting coding our space ship that is trying to land.

This is just a small sample of the many kinds of animations supported by Ranger.

Particle Systems

Ranger also comes with two built in Particle Systems (PS) that you can use to build complex particle effects. Each one supplies a variety of styles from Unidirectional (aka rocket thrusts) to Omnidirectional (aka explosions) plus a few more to have fun with.

Audio Effects

If you’re in a pinch and want sounds effects without having to create them then look no further. Ranger comes with not just one but two procedural sound effects generators. The first is a port of a popular 8bit sound effects generator **Sfxr** and the second, **RSfxr**, is a pure Web Audio creation with even more controls than the first. Each one allows you to create some really crazy/ funky sounds ranging from warblers and transporters to alarms and phasers including the good old classic Asteroids space ship shooter sound.

Important Concepts

The are a few concepts we should cover before we start along our journey, and it would be helpful if we were on the same track.

Dart

Ranger is written in the Dart programming language and as such it would be beneficial to learn the Dart language. Don’t worry though if you have experience with Java, C/C++ or Scala you shouldn’t have any problem picking up Dart whatsoever.

Editor (IDE)

This book will periodically refer to Dart’s older, and now deprecated, companion code editor called appropriately enough “Dart Editor”. The Dart team has now redirected their efforts to a

plugin for Jet Brains' WebStorm editor instead. However, keep in mind that any configuration and launching discussions will focus on the Dart Editor for simplicity and consistency.

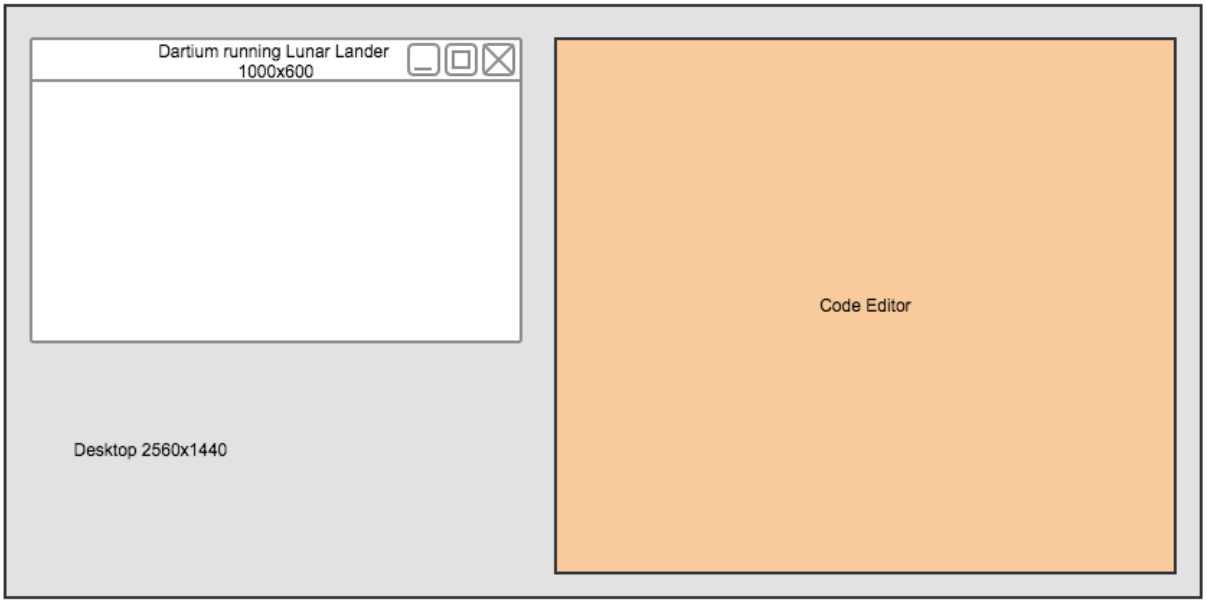
Design resolution

A core feature of Ranger is that is supports the concept of a Design resolution. Instead of thinking about the dimensions of a particular physical device, like a mobile phone, we think about a "virtual" resolution.

When designing any game we need to think about where it is going to be played. It could be played on a desktop or mobile device, we don't really know. Regardless, most of the time it helps to know what dimensions we need to design around. There really isn't a perfect solution but perhaps we can find a middle ground. This is where the Design-to-Device ratio comes into play.

Say for example we choose to design the game with the expectation that it will be played on a Nexus 7 tablet with physical dimensions of 1920x1200 pixels. But we find that it could be played on a Nexus 5, 6 or even a Nexus 9, where each one device has different devices dimensions. We don't want to write code that specifically targets any individual device. One way to minimize targeting device dimensions directly is to target "Design" dimensions instead. It could turn out that the design dimensions are equal to your a device pixel-for-pixel but that could be random happenstance.

Hence, when you are creating games in Ranger always think in terms of Design dimensions. With that being the case in this book I am going to target a Design dimension of 1920x1200, and because I want to view the game alongside my code editor I am going to choose a DIV dimension (i.e. the device dimensions) of 1000x600. This gives me plenty of horizontal room on my desktop where they both fit (See image below.)



HTML5 Canvas

Even though Ranger comes with many Nodes none of them are meant to be used in production code. Ranger's philosophy is to be an enabler and as such you are strongly encouraged to

create your own Nodes using the Ranger Nodes as examples on how to make your own. Knowing how to use the Canvas API will go along way in creating efficient custom Nodes.

Of course if your game is graphically very simple then you could probably get away with using Ranger's Nodes. Just be conscious of performance when deciding which path to take. For our Moon Lander game we will be creating custom Nodes the whole way.

The Source Code

If you would like to see the Moon Lander game before departing on this book's journey you can pull down the code from **(**https://github.com/wdevore/Ranger-MoonLander**)**.

As a minimum read Chapter 1. It shows how to setup the environment so you can load the project and run it. However, I encourage you to have fun building the game step-by-step as you follow along, and in doing so this book will teach you how to use all the features and capabilities of Ranger. When you're done you will have enough knowledge to take your own awesome game design and bring it to life in Ranger.

Moon Lander

This book's journey into the land of Ranger is based upon a classic arcade game called Lunaer Lander. For those of you who might be raising an eyebrow in confusion don't worry it is a fun game and easy to play.

The original game is simple, you try and land a spaceship with a limited amount of fuel. Upon entering the planet's outer "area" you are suddenly presented with several landing sites. Based on the amount of fuel you have and your entrance velocity you need to quickly choose a site that is within in range. Some sites are very simple to land on because they are wide and close, others are very difficult because they are narrow and far away. Our game will be a bit more simplified so as not to stray from the goal which is to learn Ranger.

Graphically the original was pretty stark. A black background with vector lines representing the lunar landscape and space ship. Our game is going to have a few customizations to add a bit of flare, both in game play and graphics. First we will be using SVG where possible, instead of a lander ship it will be an alien saucer, multi-landing pickup and deliver, caves, dangerous asteroids. In addition, we will introduce the concept of levels.

In the end the goal of the game is to encourage and inspire you to write games in Ranger. Along the way I hope you find the code useful enough to fork into your own games.

Book Organization

First we hit the ground running by building the ubiquitous Hello World example. This will be our first exposure to the framework of Ranger. Part II dives into further details on Ranger's framework, Part III takes it even further and Part IV goes beyond.

- **Part I: Getting started with Ranger**
  - **Chapter 1: Hello World**
    How to setup a Ranger development environment including the Dart Editor plus pulling down the Ranger library. Dart and it's SDK are all freely available and used by a large development community. We then run a simple Hello World template to get a good idea of what Ranger is and how it works.
  - **Chapter 2: Moon Lander**
    Here we define and create the underpinnings of the Moon Lander game while covering some of the basics of Ranger such as creating a custom Node.
  - **Chapter 3: Introduction to Ranger**
    In this chapter we formally introduce Ranger and its capabilities and how it will help us create our awesome game. Topics range from Scenes and Layers to SceneManager. After this we should have a much better idea of how Ranger plays a role in games.

- **Part II: UFOs**
  In this part we focus on the beginnings of Moon Lander while covering more advanced features of Ranger.
  - **Chapter 4: Under Construction**
    We begin construction of the Moon Lander game using the Level3 template. In addition, we will look at Scenes and Layers in more depth and how to use them to layout a flow between various other Scenes.
  - **Chapter 5: Asset loading**
    This chapter covers loading SVG assets complete with an animated loading animation. We also introduce keyboard input for controlling the landing saucer's orientation. We get our first opportunity to work with Group Nodes to add landing gear.
  - **Chapter 6: Menus and the Main Course**
    This chapter covers setting up a first draft of the main menu and general flow of Moon Lander. This is where the player can choose to play or adjust settings.
  - **Chapter 7: Nodes**
    In this chapter we design and construct a basic first level shell and populate it with a lander (aka flying saucer Actor). We get our first experience with the scene graph using relative Nodes including animating the landing gear of the flying saucer.
  - **Chapter 8: Particle Systems**
    Our lander is a bit old fashioned and still uses thrust in high gravity fields. In this chapter we outfit our lander with not one but three thrusters! We also learn about two Particle Systems supplied with Ranger.

- **Part III: Leveling Up**
  In Part III we design a more formal structure of our Moon Lander game. This includes things such as menus, popups, storage, audio.
  - **Chapter 9: Dialogs**
    Up until now we haven't really had a complete sense of overall game flow. In this chapter we begin by adding Scenes that cater to dialogs. These dialogs provide access to Settings and Score Scenes.

- **Chapter 10: Pop-ups**
  What happens when the player finishes a level or perhaps fails to land? Here we create pop-ups dialogs using custom Nodes and Animations.
- **Chapter 11: Audio Effects**
  Adding sound effects to games can add a whole new dimension. In this chapter we introduce sound effects by how they are created and implemented. Ranger includes two sound effects generators and we will cover both, with them we can create all the sounds for the game, Alarms, Bonus, Level-Complete just to name a few.
- **Chapter 12: Physics**
  Moon Lander at it core is an example of simple physics, gravity and collision. Rather than use a full-on physics engine we will hand code these right in. In Chapter 7 we began our venture into physics by integrating Gravity. In this chapter we add collision.

- **Part IV: Bonus round**
  This part isn't so much about Ranger per say as much as it is about game development in general. But there will still some Ranger relevance where appropriate. In this part we finish off the Moon Lander.
- **Chapter 13: Scores and More**
  It is the challenge in life that gives meaning to life itself and as such our challenges in the game are along the lines of goals that the player can achieve. We also add the finishing touches and closeout Moon Lander.

- **Part V: Ranger Reference**
  This part is a reference guide that includes deeper coverage of Ranger.
- **Chapter 14: Reference I**
  In this chapter we focus on the higher level aspects such as Scenes, Layers and Nodes.

- **Part VI: Extras**
  This part focuses on examples that touch on specific features or techniques. The key to using Ranger is knowing how to use the Scene Graph and Nodes effectively. We start with very simple examples and progressively work our way to more complex examples each with goal of showing how to accomplish useful tasks for games.
- **Chapter 15: Simplicity**
  In this chapter we create a series of projects each focused on a specific technique.
- **Chapter 16: Transitions**
  In this chapter we create a new transition called Chomp.
- **Chapter 17: Tween Animations**
  Animations can be quite complex. This chapter dives deeper into the TweenEngine library that Ranger uses.
- **Chapter 18: Scene Graphs**
  To fully understand how to create Nodes in Ranger it helps to understand how Scene Graphs work in detail. In this chapter we cover the workings of Ranger's scene graph.
- **Chapter 19: Mobile first**
  In progress in relation to Sky.

- **Conclusion**

Acknowledgments

This book would not have been possible without the kindness and support of my lovely wife. I would also like to thank the following people for their efforts and contributions to the world of Dart.

- The designers of Dart, Lars Bak and Kasper Lund. They created an awesome language that —in my opinion—is a great replacement for javascript.
- The Dart team advocates. Their continued efforts show Dart's strength inside and outside the community.
- The Pub community for creating and supporting Dart through their awesome libraries. More specifically I would like to thank Xavier Guzman for creating and porting the Universal Tween Engine, Marco Jakob for creating EventBus, Dennis Kaselow for creating Dartemis and its Pooling system and Seth Ladd (who by the way is now Dart's project manager) for creating LawnDart and A* pubs.

About the Author

William DeVore is a long time C/C++ developer with roots in BitSliced processors surrounding computer graphics who also still owns his giant yellow bulletproof TMS 34010 manual. ;-)

My journey started with Z80 assembly and the 68000. I then hopped aboard the college train working with the Athena widget set under Solaris. I then hopped off into Qt under linux then swerved into D programming language.

Eventually I began working with Cocos2d-c++ until one day when I stumbled upon Dart. As for javascript, to me (in it current incarnation) it is a ticking time bomb of bugs and infinite frameworks. Languages like Dart represent what javascript should have evolved into some time ago.

My web site: https://sites.google.com/site/williamquartz/

Part I

Getting Started with Ranger

How to setup a Ranger development environment including the Dart Editor and pulling down the Ranger library.

Chapter 1 Hello World

Ranger is really easy to use. In this chapter we learn how to install Ranger, the Dart SDK and run a simple Hello World template application using Ranger's Templates. The application uses a TextNode, Scene and Layer Nodes in just a few lines of code.

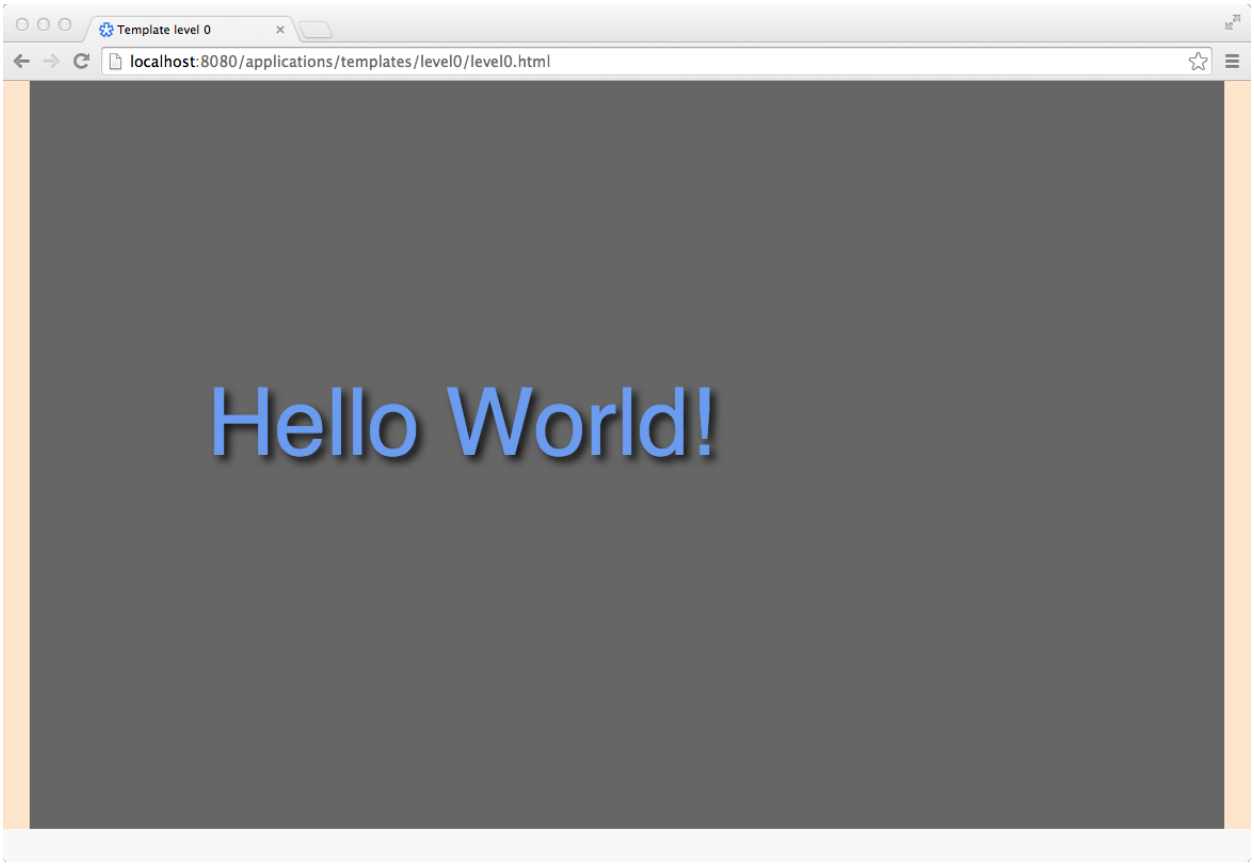


Figure 1.0 Hello World!

After we run the app we will go behind the code and see how Ranger works from a high level. Don't worry all the concepts will be covered in detail later throughout the book.

Downloading and installing the Dart SDK

We will begin by downloading Dart followed by creating a project and pulling the Ranger library.

In order to run our Hello World template application in Ranger we first need to install the Dart SDK. Installing the SDK will get you the Dart Language VM, an Editor and a custom browser (called Dartium) for running actual Dart code.

The Dart SDK is located at (dartlang.org). Navigate to the website and click the “Download Dart + Editor” button. See figure 1.1.

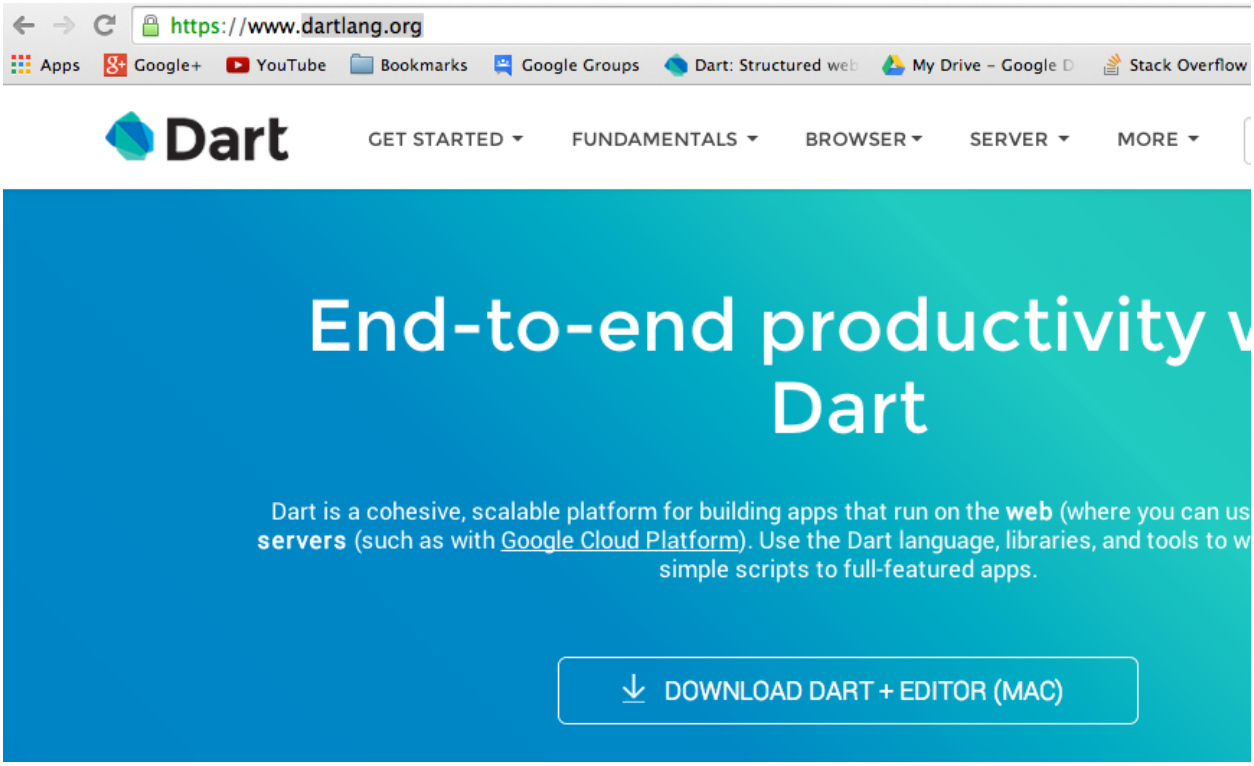


Figure 1.1 Dart website

It is pretty simple to install, just run the .dmg or .exe file and you are done.

Installing Ranger

Ranger as a whole is conceptually split up into two parts: Ranger-Dart (**Ranger** for short) and Ranger-Sack (**Sack** for short). Ranger is the core piece that your game references. Sack is a sibling project (that also references Ranger) and is chuck full of templates, unit tests and examples that you can use to help guide and inspire you.

The Hello World application is part of Sack as a template, so lets pull down Sack from GitHub. To do this we navigate to Ranger-Sack’s GitHub location:

Figure 1.3 Ranger-Sack GitHub: (https://github.com/wdevore/Ranger-Sack)

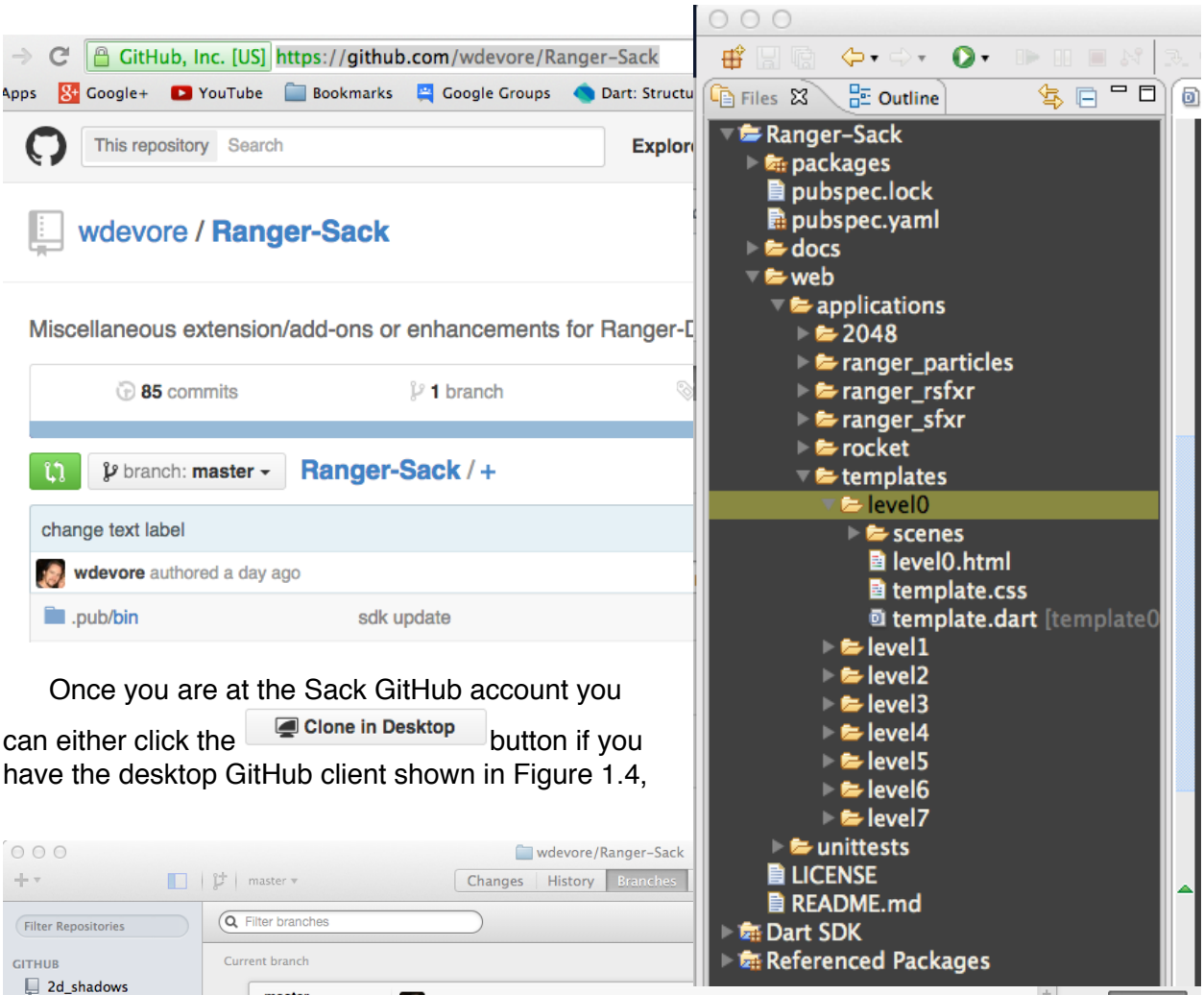


Figure 1.5 Files View

Once you are at the Sack GitHub account you can either click the Clone in Desktop button if you have the desktop GitHub client shown in Figure 1.4,

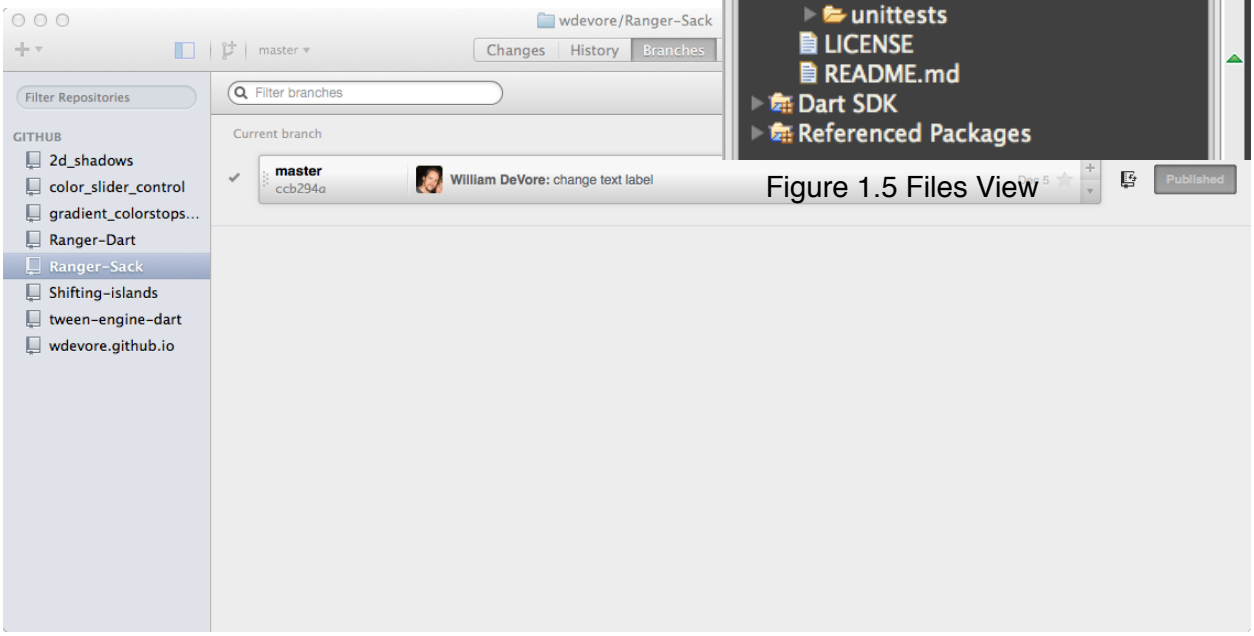


Figure 1.4 GitHub client for OS X Mac.

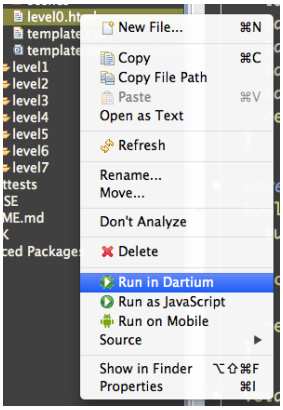
or click Download ZIP button to get a zip file to uncompress.

For now click the “Download Zip”—you can always install the GitHub client at your leisure. This will cause GitHub to generate a zip file from the source code and begin downloading a file

called “**Ranger-Sack-master.zip**.” Once the zip has finished downloading uncompress it to a known location, then change the name of the uncompressed folder to just “Ranger-Sack”.

Now launch the Dart Editor and select the file menu option “Open Existing Folder…” and navigate to where you uncompressed Ranger-Sack and select “Open”.

Expand the Ranger-Sack project and you should now see a *Files* view as shown in Figure 1.5.



Now lets see if everything went according to plan. Ctrl or Right click *level0.html* file and select “Run in Dartium”. First Dartium will launch and then level0 will run.

For about 2 seconds you will see the template’s splash Scene where upon it instantly switch to another Scene showing the ubiquitous “Hello World”! See Figure 1.0.

Congratulations, you just successfully ran your first Ranger application. Sweet!

A tour of Ranger-Sack

Perusing the folder you can see there are many templates under the templates directory. Each one takes it up a notch starting from Asset loading to Input. But wait there is more!

Sack also has a suite of Unit Tests designed not only to verify Ranger but also to showcase Ranger’s features. There is quire a bit:

- Audio
- Colors
- Fonts
- Input
- Particle Systems
- Space mapping
- Sprites
- Transforms
- Transitions

To run them simply launch *unittests.html* and browse through the different tests. Figure 1.6 shows the Particle System test in action.

But wait! There is *even* more! Sack has an application folder that has several games and applications: 2048, Ranger-Rocket, Ranger-Sfxr, Ranger-RSfxr and Ranger-Particles.

Template Level 0

Level 0 is as basic as it gets. At the application root you have the standard Dart file layout, an HTML file, CSS file and a Dart file. For a Ranger application this is all there will be in your application root—and they rarely change aside from file renaming.

You will always create a sub folder that contains your entire game. Looking at Level0 you can see there is a sub folder called *scenes*, and within that folder is a *game* and *splash* folder. There isn’t any rules on folder structure. That is entirely up to you.

The HTML and CSS files are going to be the same from game to game. Starting with the HTML file (Code 1.7) we can see that it is pretty minimal. There are two important lines: the DIV

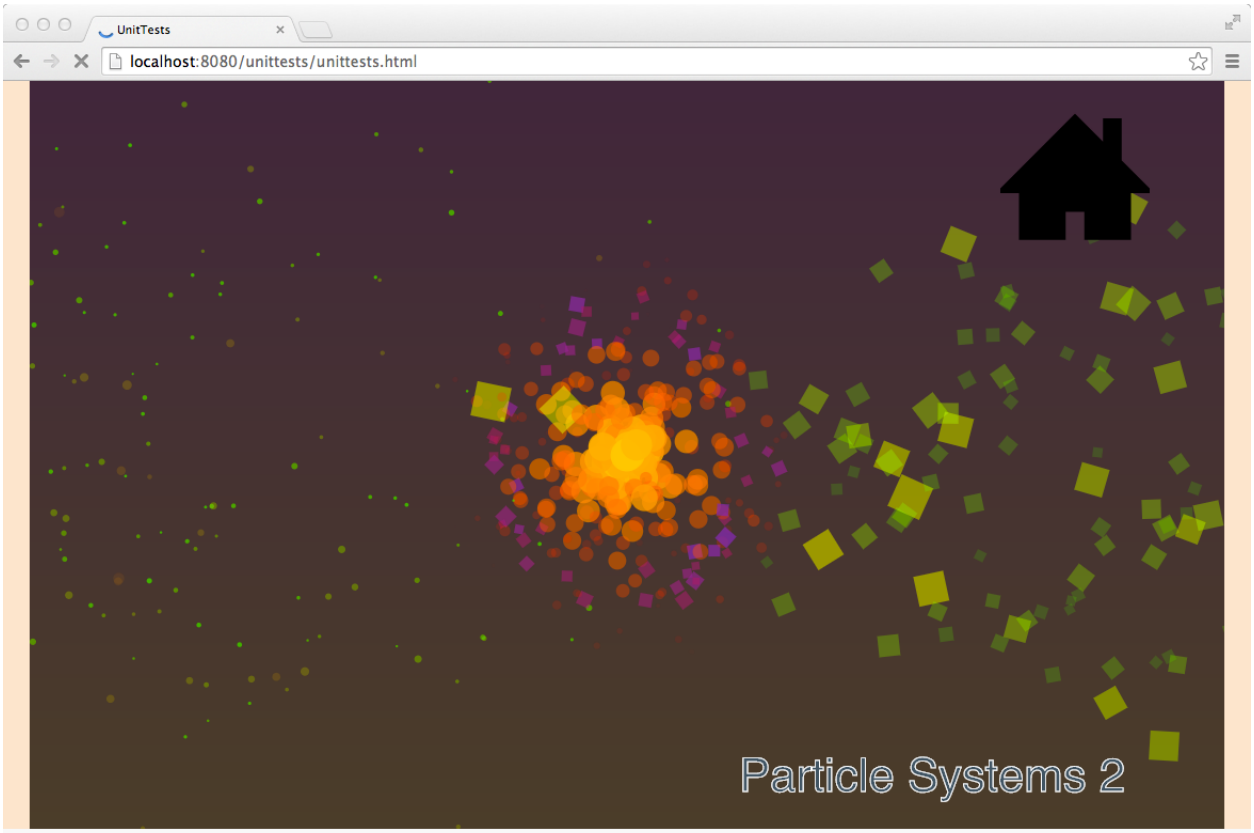


Figure 1.6 Particle Systems test 2

element with id="*gameSurface*" and the script element that specifies a file that contains the *main* entry point (src="*template.dart*").

The script element is standard Dart and won't change unless you rename the Dart file. But the most important line relative to Ranger is the DIV element.

Code 1.7 Level0.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Template level 0</title>
    <link rel="stylesheet" href="template.css">
  </head>
  <body>
    <Div id="gameSurface" width="720" height="700"></Div>
    <script type="application/dart" src="template.dart"></script>
    <script src="packages/browser/dart.js"></script>
  </body>
</html>
```

This is the location where Ranger will “inject” an HTML Canvas element. We will go into more detail in Chapter 3 so stay tuned.

The CSS File is also pretty minimalistic and won’t change either. The most important selector is *#gameSurface*. This style configures the DIV for better alignment during construction.

Code 1.8 template.dart

```
library template0;
import 'dart:html';
import 'package:ranger/ranger.dart' as Ranger;
part '../level0/scenes/game/game_scene.dart';

Ranger.Application ranger;

void main() {
  ranger = new Ranger.Application.fitDesignToWindow(
      window,
      Ranger.CONFIG.surfaceTag,
      preConfigure,
      1280, 800
      );
}
void preConfigure() {
}
```

Template.dart is where all the bootstrap action occurs and will continue to grow as more and more Dart files are added. Code 1.8 is an abbreviated version that does the following:

- Give our application a library name (all Dart apps are libraries, by default they have none).
- Import a Dart specific package/library.

- Import Ranger and give it a prefix to help with clashes with class names.
- Specify the “parts” that make up the app (i.e. Library)
- And finally create a Ranger Application using one of the Application’s factories supplying a callback.

We will go into details in Chapter 3 as well, especially the callback. For now the callback is our first opportunity to create actual Ranger Nodes. Template Level0 uses three of them: TextNode, Scene and Layer.

Well that should give you a taste of Ranger and what it can do. Now that we have a better idea of how to use Ranger we can create our Moon Lander game. So how about we brainstorm a bit.

Chapter 2 Moon Lander Design

In this chapter we layout our plans for building a fun game that involves some simple physics with gravity and collision detection.

Figure 2.0 Classic Lunar Lander

The original game (Lunar Lander) consisted of what you see in Figure 2.0. Our version will be a bit more modern in the graphics department plus we will add a few more pieces of functionality.

BrainStorming

Lets list what the original game had both visually and functionally:

- A single score
- Time
- Fuel
- Altitude
- Horizontal speed
- Vertical speed
- Black and White
- Vector rendering
- Basic Audio

Not bad for the day, but we can add a bit more sparkle to it—literally. Lets list out what we would like to add:

- Color both for text, landscape
- Translucent bars for Fuel, Altitude and Speeds
- Twinkling stars with Parallax (Optional)
- More advanced audio effects
- Dangerous meteors (Optional)
- Use SVG instead of Vectors (the irony is that SVG *is* vectors…but with curves!)
- Interstellar backgrounds
- Multiple landers with different characteristics. (Optional
- Caves
- Doors to allow for sequencing/puzzles
- Multiple scores
- Achievements
- GUI (menus, popsups etc.)

Challenges for yourself:

- Twinkling stars with Parallax
- Dangerous meteors
- Multiple landers with different characteristics. (Optional
- Caves
- Doors to allow for sequencing/puzzles

That is quite a list but it's easily doable using Ranger. Lets review these and see where Ranger will fit in. We will start with Color. Ranger's rendering is currently based on the HTML5 Canvas which supports much more than just colors.

Actually everything on the list, minus the scores, achievements and audio will be custom Ranger Nodes (we cover Nodes in the next chapter.)

For audio we will use Ranger's AudioEffects object and feed it with sounds created in Ranger-RSfxr generator application (see Figure 2.1) This application's sole purpose is to create classic arcade sound effects like Coin-Pickup, Explosions, Jumping plus

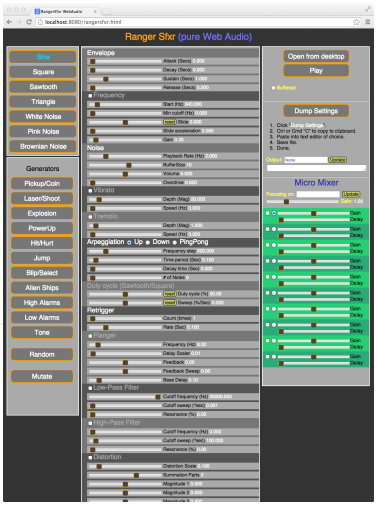


Figure 2.1 Ranger-RSfxr

many more. It does this using pure Web Audio nodes. Its output is **JSON** maps that you copy and paste into a text file. This text file can then be loaded and passed to the AudioEffects class. It is quite fun to play with. You can find it under the *Applications/ranger_rsfxr* folder of Sack.

Now if RSfxr isn't creating the kinds of sounds you like—perhaps they are a bit to "pure" bell like you can always switch to using another Ranger sound effects application called Ranger-Sfxr. This app is a port of a javascript app and it tends to generate sounds that are more "bit-crushed". Don't worry, the AudioEffects class can load either one.

For Score and Achievements we need to be able to store them somewhere, specifically for cases where the player decides to pause or quit. We can do this with the help of a Pub library called Lawndart. This library hides the details of dealing with a browser's local storage and will even interface to the appropriate type.

Pubs

Speaking of Pubs. Dart has the concept of packages, and includes a repository (as shown in Figure 2.2) that anyone can add to. These packages contain one or more Libraries. However,

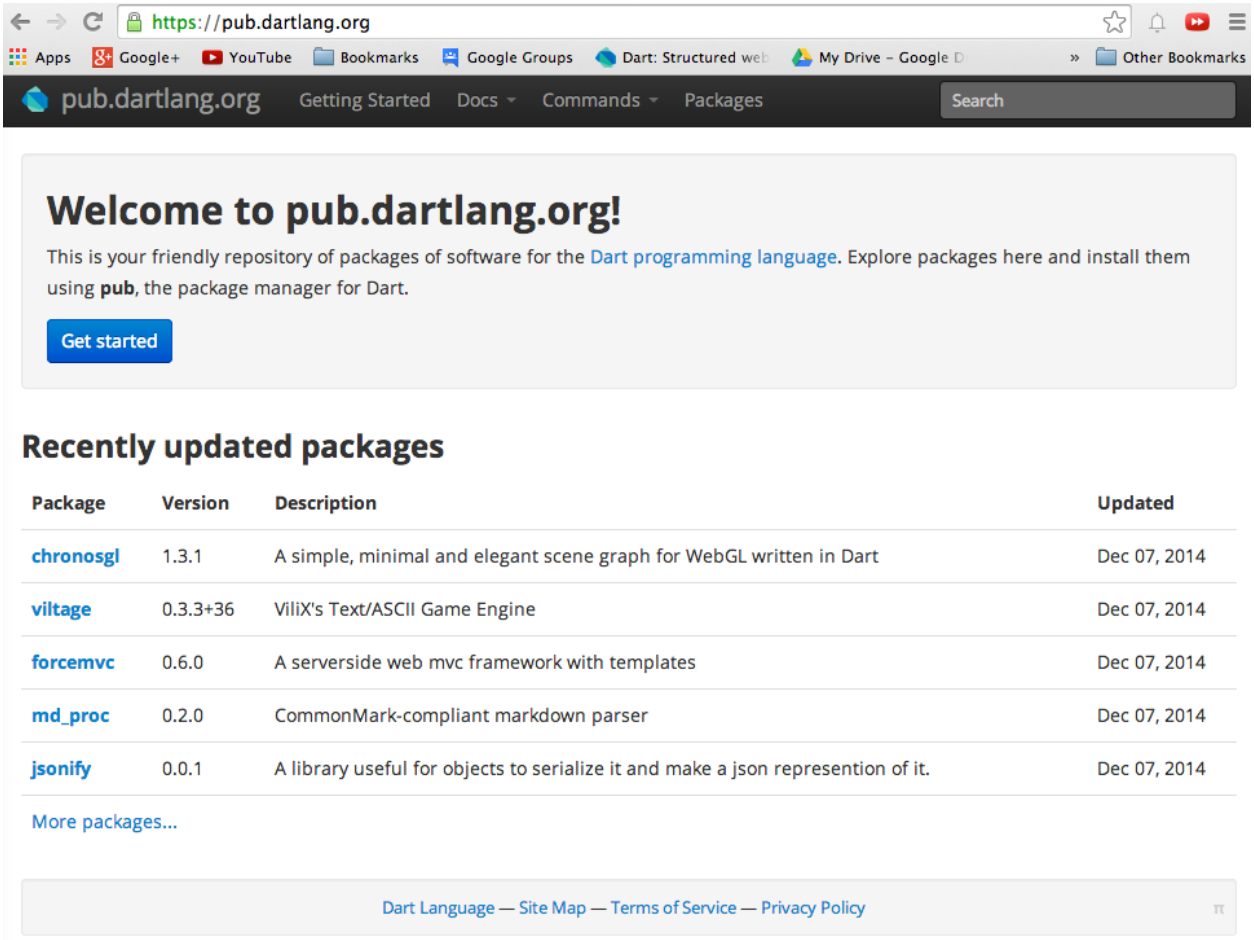


Figure 2.2 Dart's Pub

not all packages are kept on Pub, some (like Ranger) are kept on GitHub. On average packages and libraries are one-to-one but not always.

For the GUI we will roll our own using yet more custom Nodes, but this doesn't mean you can't use HTML/CSS to design your GUI, for example, the Unit tests' main menu is built with HTML/CSS.

Physics will be custom built as well. The game's physics are very trivial and doesn't require a full fledge physics engine like dart-box2d (https://github.com/dominichamon/dart-box2d).

By now we should have a pretty good idea of what our goals are and how Ranger and Dart can help. Along the way we may chuck a few things out but we will do our best to follow through. Up to this point we have been mostly playing with Ranger-Sack, but now it is time for a formal introduction to Ranger-Dart, the core of Ranger.

Chapter 3 Introduction to Ranger

Ranger is a package written in Dart that encapsulates several different concepts into one library. Timing, Transformations, Pooling, Audio just to name a few. However, Ranger doesn't have a formal face until a coder puts forth the effort.

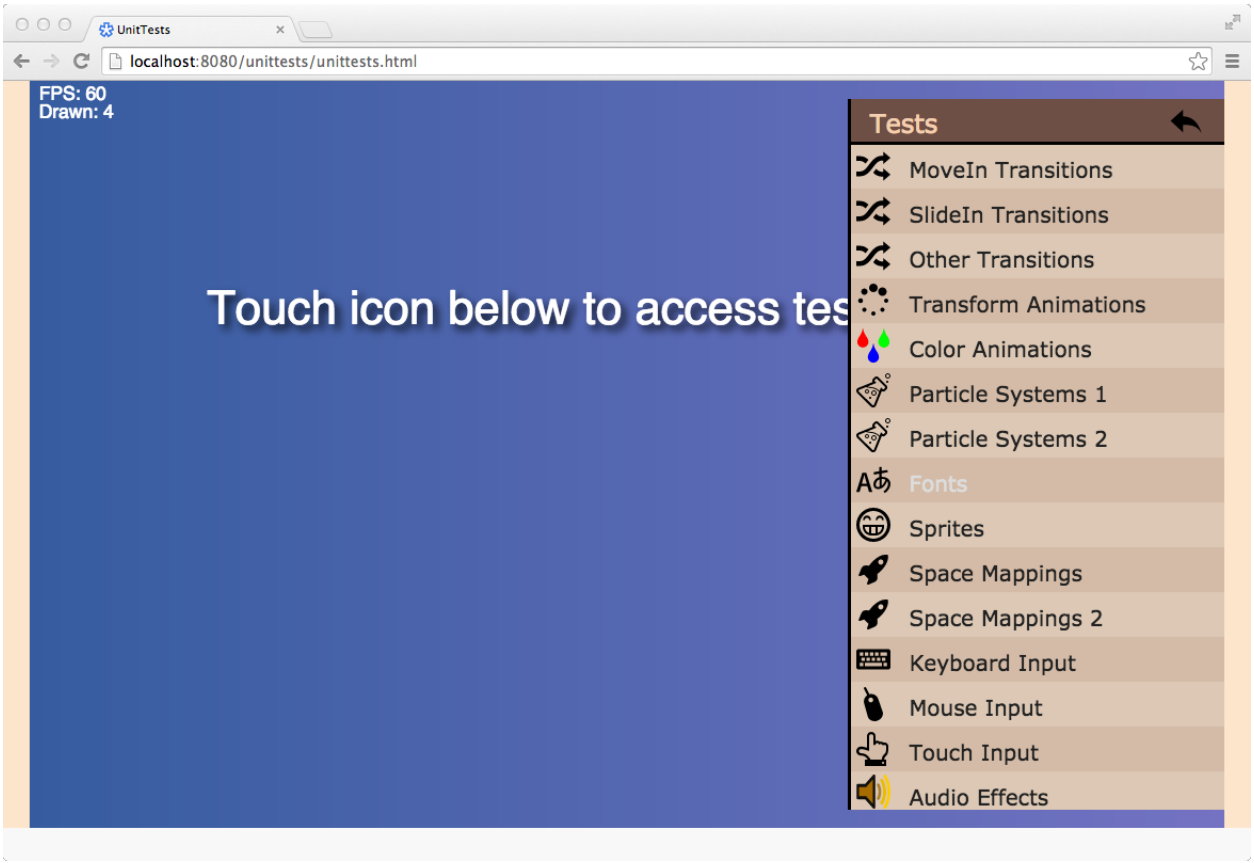


Figure 3.0 Ranger's default face—Unit tests

In this chapter we will cover the basic essentials of Ranger by looking at template/level0 as an example. This template contains Scenes, Layers and TextNodes. Along the way we will introduce the SceneManager.

Bootstrapping Ranger

Before we can dive into Scenes and Layers and all those cool Nodes we need to "bootstrap" Ranger. Ranger's visual component is an HTML5 Canvas element. All you need to do is specify a DIV element with a specific id attribute and use one of Ranger's Application factories to get an instance of the Ranger API.

For Dart applications the entry point is the *main* function. Whatever file has this method will be the file referenced in the script *src* attribute in your html file. We saw this in Figure 1.7 and 1.8. It is in the *main* function where we will bootstrap Ranger.

Go ahead and open up *template.dart* underneath the *applications/templates/level0* folder inside of the Sack project. The first thing you will see is a **library** statement.

Libraries

All Dart projects are actually libraries, even if you don't explicitly indicate it using the *library* statement. I tend to "name" my projects (aka libraries), so for level0 I named the "library" **template0**.

```
library template0;
```

This name will be used throughout other files in the project using the sibling statement *part of*.

```
part of template0;
```

Imports

Next you will see import statements. The most important *import* is Ranger itself--assuming we have referenced the Ranger-Dart Package first.

```
import 'package:ranger/ranger.dart' as Ranger;
```

Notice that the line has the "**as**" keyword. This is a prefix that allow us to "isolate" the Ranger library into its own space within our code. This helps reduce collisions with other libraries including Dart's libraries.

> Note
>
> You may be wondering why you didn't have to add a dependency entry to Moon Lander's *pubspec.yaml* file before importing, that is because Ranger's *pubspec.yaml* has this entry and by extension now our Moon Lander project has this dependency.

Parts

Because we decided to build Moon Lander in a modular fashion we are going to break the code up into separate files. We do this by supplying a library name for our project (which we did above) and by including "parts" from different files (i.e. the **part** keyword).

```
part '../level0/scenes/game/game_scene.dart';
part '../level0/scenes/game/game_layer.dart';
```

In each of those files their will be a matching "**part of**" statement that corresponds to our library name.

Application

Ranger's API is accessed through the **Application** class, and we can obtain an instance using one of the factories provided. Continuing to look at the *template.dart* file we can finally see Dart's entry point *main* and there doesn't seem to be much there (comments have been removed for brevity):

```
Ranger.Application ranger;
void main() {
  ranger = new Ranger.Application.fitDesignToWindow(
      window,
      Ranger.CONFIG.surfaceTag,
      preConfigure,
      1280, 800
      );
}
```

The first parameter to the factory is a ***window*** object. Ranger uses that for subscribing to browser events.

The second parameter is an id that we specified in our html file. By default that value should be "gameSurface". You can use any value you want just be sure to specify it here instead of using the CONFIG value.

The third parameter is a callback which is referred here as the *preconfigure* callback. It will be our very first opportunity to create Ranger Nodes—the essence of a Ranger game. See the section "Preconfigure Callback" below.

The final parameters are the Design dimensions. For a review of what Design dimensions are see the Preface chapter on "Design and Device". But to reassert, these dimensions are a type of virtual dimension, and Ranger will endeavor to fit those dimensions into the HTML Canvas element (aka simulated physical device.) So lets talk about fitting.

Fitting policy

Thinking in terms of Design dimensions I would like to setup a desktop arrangement that will give me the ability to view the game alongside my code editor. For this I am going to choose a DIV dimension (i.e. the device dimensions) of 1000x600. This gives me plenty of horizontal room on my desktop for them to both fit (See Figure 3.1 Desktop arrangement.) With these Design and DIV dimensions Ranger can *almost* create a Canvas and inject it into our DIV. There is no guarantee that our DIV and the Design width-to-height ratios are the same. So we need to tell Ranger what policy/strategy to use just in case.

The **Fitting policy** does that, and it is what the *fitDesignToWindow* factory does for you.

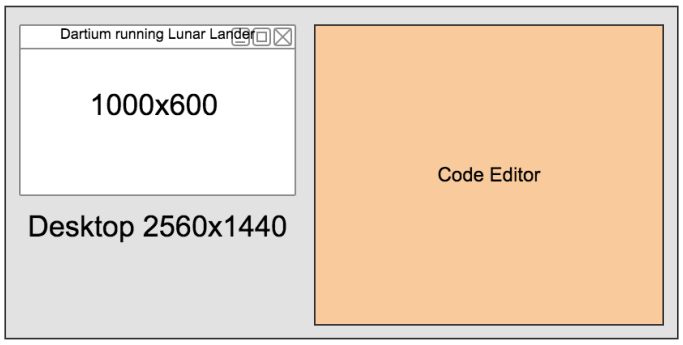


Figure 3.1 Desktop arrangement

As a matter of fact the fitting strategy for this factory is such that Ranger will determine a ratio that will make sure the Canvas fills to extent vertically with a horizontal center alignment. But this factory isn't the only one, there are several others, for example, *fitDesignToContainer* is a factory with a policy that ignores your Design and simply fits the Canvas into the DIV as is. This type of factory/policy is for applications that need to "host" Ranger in a window-like area without any scaling or skewing—Ranger-Particles application is an example of such a use.

No matter what factory you use Ranger can now calculate a scaling transformation matrix that properly fits a Canvas into our DIV element.

But what about the callback?

Preconfigure Callback

Once the *fitDesignToWindow* factory has finished constructing Ranger's core components it waits for a page-show event. When the event fires it completes the Canvas construction phase and finally calls our callback.

This callback (called **preConfigure** in this book) is our very first chance to begin construction of our game—literally our first opportunity to create actual Ranger Nodes!

What kinds of Nodes are we itching to create?

Nodes

Ranger is a Scene Graph (SG) at its core. An SG is a way to hierarchically organize data structures and manage the contents of spatially oriented scene data. SGs shield you from the details of rendering and let you focus on what to render rather than how to render it. They are an abstraction between the graphics subsystems (i.e. Canvas) and your code.

Although there are many types of SGs, Ranger's is purposefully constrained as a Directed Acyclic Graph (DAG.) All Nodes in the graph have a directed parent-child relationship in which no cycles are allowed—Nodes cannot be their own parent.

Depending on how the SG is traversed rendering Nodes can look significantly different from one traversal style to another. Ranger's traversal is always Top to Bottom, Left to Right (see Figure 3.2 Traversal). However, each Node in the SG has a Z-order value that, if negative, can allow a child to be visited *before* a parent thus allowing a child to visually appear *above* a parent. This effectively reverses the Top to Bottom traversal locally between the parent and child.

Ranger comes with a set of prebuilt Nodes. Some assist in flow (Scene), some assist in visual layout (Layer) and some assist with organization (GroupNode, EmptyNode.)

We will cover Scene Nodes first as they are absolutely required otherwise you wouldn't see anything.

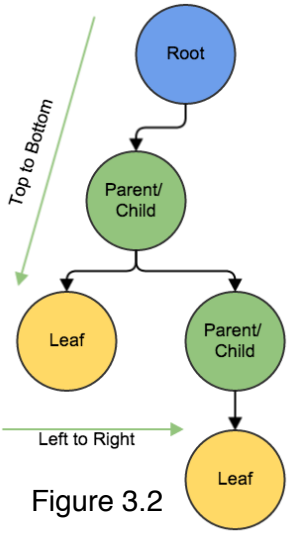


Figure 3.2

Scene

Ranger game flow is centric around a Node called **Scene**. As your game flows from one intention to another your code is actually moving between Scenes. Here is a basic set of scenes a typical game may have:

- Splash screen
- Main interaction
- Configuration
- Social interaction
- Scores
- Pause/Resume
- Levels

From a high level these are the most common. Every game is different of course and some could be missing a few or even have more.

In Ranger a Scene is designed to be both a collection of other Nodes and as a target of the **SceneManager** (we will cover the SceneManager shortly.) A Scene can collect Nodes because it "mixes" in a **GroupingBehavior** (see Figure 3.3). This makes for two types of Nodes: Leafs (those that *do not* mix in the grouping behavior) and those that do.

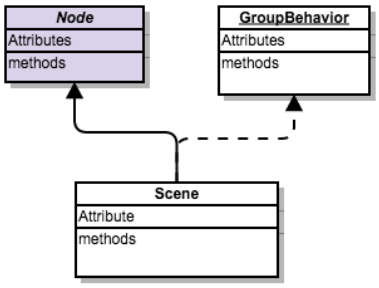


Figure 3.3 Scene

A Scene, in and of itself, is nothing more than any other Node. The real functionality comes from extending Scenes, and Ranger comes with three such class extensions:

- BasicScene (rarely used)
- BootScene (required)
- AnchoredScene (recommended)

BasicScene

BasicScene is mostly used internally but it can be used as a base for any custom Scenes you create.

BootScene

This Scene is used solely for handling the boot sequence of Dartium/Browser and Ranger. While Dartium is still launching and stabilizing we can't really use Ranger predictably because the **SceneManager** (SM) hasn't been started. Therefore the BootScene simply waits for the SceneManager to begin functioning. Once stable the SM will call BootScene's *onEnter*(), upon which BootScene will promptly replace itself with a "real" Scene. In most cases this "real" Scene is a Splash Scene.

You will always use BootScene as the very first Scene otherwise you will experience unpredictable results.

AnchoredScene

AnchoredScene is the work horse of Scenes and is pretty much the only Scene you will use. Advanced developers may find that they need more capability and are encouraged to adapt this Scene for their own use—or even the BasicScene for that matter.

We almost have enough knowledge to begin looking at *template.dart*'s *preConfigure* method. What we lack is knowledge about the SceneManager mentioned above.

SceneManager

From a top level Ranger functions or flows around **Scene**s, and the class that manages them is the SceneManager (SM). Its sole job is to "run" the scenes through a lifecycle. Internally the SM is a stack where the top of the stack is the current or active running scene. If it isn't on the top it isn't running.

In Figure 3.4 we can see that there are three scenes on the SM's stack. The top most is a

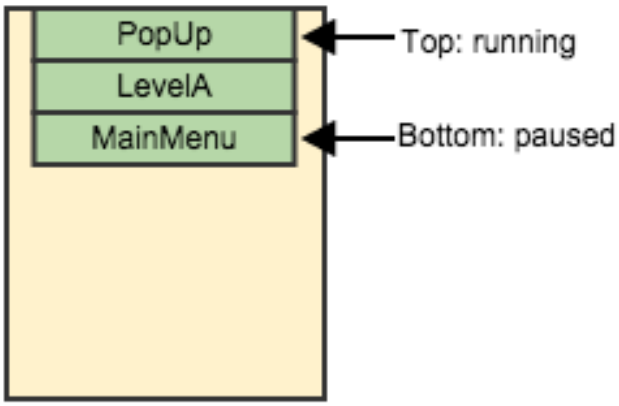


Figure 3.4 Scene Stack

PopUp scene that could be notifying the player of an achievement. When they dismiss the popup Scene the *PopUp* scene is popped off the top. *LevelA* now becomes the top Scene and begins running again.

During all these stack manipulations each running scene is moving through a lifecycle:

- **onEnter**. Called when a scene starts running.
- **onEnterTransitionDidFinish** (only Transition Scenes understand this event)
- **visit** (called with a rendering context)
- **completeVisit** (called at the end of a complete visit of the scene graph)
- **onExitTransitionDidStart** (only Transition Scenes understand this event)
- **onExit**. The last event a Scene when it is removed from the stack.

If your Scene is a TransitionScene class then your scene will get two extra events dealing with the beginning and ending of the transition: *onExitTransitionDidStart* and *onEnterTransitionDidFinish*.

In each of these events you get opportunities to manage your scene's resources. For example if your scene is paused because another Scene was pushed onto the stack above yours, then your Scene will receive exit events giving you an opportunity to stop any animations perhaps. Later when your Scene is resumed by appearing at the top of the stack again you can resume your animations.

Now we can finally look at the *template.dart*'s *preConfigure* method with a bit more understanding (see Code 3.5.)

Code 3.5

```
void preConfigure() {
 GameScene gameScene = new GameScene(2001);

 SplashScene splashScene = new SplashScene.withReplacementScene(gameScene);
 splashScene.pauseFor = 3.0;

 Ranger.BootScene bootScene = new Ranger.BootScene(splashScene);

 ranger.sceneManager.pushScene(bootScene);

 ranger.gameConfigured();
}
```

The first thing to notice is that three Scenes are being created. The BootScene we recognize as being required in order to wait for Ranger to stabilize. The GameScene is the Scene where the main action is—in this case it is the awesome “Hello World!” text. The SplashScene is pretty much what it says it is a Splash scene, and it is where things such as resource loading may occur.

Pushing the BootScene is what keeps Ranger from simply stopping and shutting down:

```
  ranger.sceneManager.pushScene(bootScene);
```

Note

Ranger will not run unless there is at least one Scene placed on the stack. You may see Ranger start briefly, but as soon as the SceneManager inspects the stack and sees there are no Scenes to run it signals Ranger to simply stop and shutdown.

When we construct the BootScene we give it another Scene that is destined to replace BootScene once the onEnter event is sent. If you don’t provide a replacement scene BootScene will wait forever being unable to transition. BootScene’s job is to throw itself off the stack by replacing itself with the replacement scene—this makes it the shortest living Scene ever. Goodbye BootScene hello SplashScene.

A lot of games have a welcome screen called a splash screen. These types of Scenes are where you introduce your application during launch. At times launching can entail some time consuming things like connecting to a social network or downloading resources. Splash scenes are completely optional but they do add a bit of polish, our Moon Lander game will have one.

Once the SplashScene has finished it too replaces itself making it the second shortest living Scene ever. Goodbye SplashScene hello GameScene.

GameScene is the essence of your game—you could think of it as the main entry point to your game. You aren’t required to name your main scene GameScene but the templates do for consistency. As a matter of fact none of the Scenes discussed so far need to be named as they are. Of course giving them meaningful names helps you as a developer.

What does the SM provide for controlling our scenes, Push, Pop and Replace commands of course.

SceneManager Commands

- **pushScene**: places a Scene on the stack and starts running it by sending the onEnter event.
- **popScene**: pulls a Scene off the top of the stack (sends an exit event to it) and begins running the scene on the top by sending it enter events.
- **replaceScene**: pulls a Scene off the top of the stack (send exit event to it) then places a new Scene on top and begins running the new scene by sending an enter event to it.
- **popToRootScene**: pulls all Scenes from stack except last scene at the bottom. The bottom of the stack is considered the root.
- **popToSceneStackLevel**: pops all Scenes right up to a particular ordinal position. The position is relative to the top where the top is 0 and the bottom in N.
- **popToSceneTag**: similar to the previous pop but works by Tag id.

Once the BootScene has been pushed onto the scene stack we are ready to signal Ranger that we have finished configuring. This is done by call the Application's *gameConfigure* method:

```
  ranger.gameConfigured();
```

This tells Ranger to finish constructing and start the SM. From this point on Ranger is up and running. Now lets take a look inside the SplashScene and see what is going on—feel free to inspect BootScene but there really isn't anything to see.

SplashScene

Open the *splash_scene.dart* file under the *splash* folder. Looking at the code we can see that SplashScene inherits from **AnchorScene**, think of the AnchorScene as the work horse of

```
class SplashScene extends Ranger.AnchoredScene {
```

Scenes. It's design includes both an "anchor" Node and Animation capabilities, these two things really define what Scenes are designed for. We will cover AnchorScene later, but for now it is enough to know that AnchorScenes are the key to Scenes.

Looking at SplashScene we can see there are two constructors:

```
SplashScene.withPrimary(Ranger.Layer primary, Ranger.Scene replacementScene,…
SplashScene.withReplacementScene(Ranger.Scene replacementScene, [Function…
```

- *withPrimary*: allows you to specify a primary Layer immediately during construction.
- w*ithReplacementScene*: allows you to postpone specifying a primary Layer.

The *preConfigure* method uses *withReplacementScene* because SplashScene is going to supply the primary Layer during the *onEnter* event.

Code 3.6

```
@override
 void onEnter() {
   super.onEnter();

   SplashLayer splashLayer = new SplashLayer.withColor(Ranger.color4IFromHex("#aa8888"), true);
  initWithPrimary(splashLayer);

   transitionEnabled = true;
 }
```

Speaking of *onEnter*, recall when we covered the lifecycle of Scenes, this is one of the first lifecycle events sent to our Scene (see Code 3.6.) It is *a* place to create Nodes that your Scene needs. Here we are creating a single Layer Node then setting it as the primary Layer Node (we will cover primary Layers in the AnchorScene section.) Finally we enable transitions so our Scene automatically transitions after a pause delay—which default to 0 seconds. By setting it to **true** the *transition* method is called after the delay.

Once the SplashScene's delay has timed out, the *transition* method (Code 3.7) is called and a **TransitionScene** is created and configured to transition "instantly" to the replacement scene.

Code 3.7

```
@override
void transition() {
  Ranger.TransitionScene transition = new Ranger.TransitionInstant.initWithScene(_replacementScene);

  Ranger.SceneManager sm = Ranger.Application.instance.sceneManager;
  sm.replaceScene(transition);
}
```

TransitionScene

TransitionScenes are a transient Scene in that they last only as long as the duration of their transition. Their type is recognized by the SM and treated differently that standard Scenes, meaning they don't actually appear on the Scene stack during their lifetime, and this is what makes them transient. They are covered in greater detail in chapter "Transitions".

Once the transition is finished, your main Scene will finally rise to the top of the stack and begin running.

Now what? Glad you asked because Scenes aren't the only Node in town. As mentioned previously Scenes are designed for game flow, and it is true they could contain visual Nodes, for example some of the custom transitions do. Instead, there is another Node specifically designed to handle visual Nodes as well as Input—the Layer Node.

Beware

The incoming Scene begins running immediately at the *beginning* of the transition. This means your custom transition should make sure the Scene is either invisible or out of view by translating it off screen. Ranger's transitions do this for you during the *onEnter* event. For example, **TransitionInstant** sets the incoming scene to invisible: inScene.visible = false;

Layers

Layers can be thought of as the main companion Node to Scenes. Layers provide a pathway for Input (keyboards, mouse, touch etc.), as a foreground or background for your Scenes and as mix-in GroupingBehavior so they can collect other Nodes.

Ranger comes with two Layers ready for you to extend: **BackgroundLayer** and **OverlayLayer**.

BackgroundLayer

This is the most common Layer Node used. It retains all the features of a typical layer but adds an additional trick, it always keeps a background over the Canvas even if the layer is translated. This insures that the background always fills the viewport no matter where the Layer is translated to.

Setting this layer's ***constrainBackground*** flag to **false** will cause the Layer's background to "stick" with the Layer's origin which isn't good if the Layer has been moved (for example, centered), in addition, the actual Canvas surface would be exposed (it is **Orange** by default). So if you are seeing a lot of Orange then you have exposed the surface—not really good. Take a look at Figure 3.5.

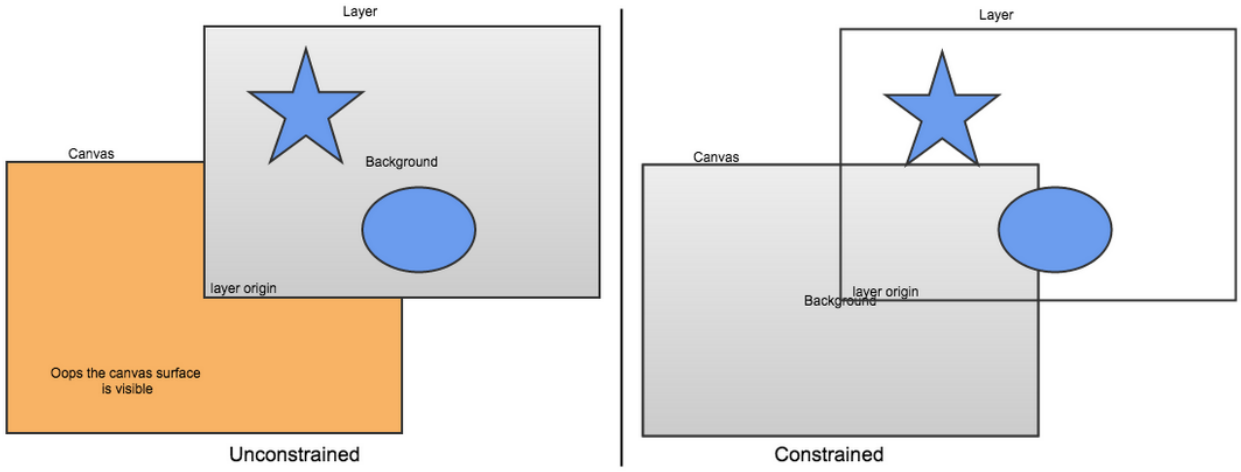


Figure 3.5 Background constraint

On the left the background *is not* constrained so when the Layer moves so does the background, which exposes the surface. Most of the time you don't want to expose the surface which is why the default is to enable constraining.

The alternative is to create a separate Layer that sticks to a covering position while another transparent layer translates. Again Ranger is an enabler so you are not forced to use the BackgroundLayer, you can simply fork it and create your own custom Layer— and you are encouraged to do so.

Hint

With multiple layers only the bottom most layer should to be constrained.

OverlayLayer

This Layer is almost identical to the BackgroundLayer however it doesn't use a constrained background and defaults to transparent. You would typically use this as a Head-Up-Display (HUD).

Lets take a peek at the *splash_layer.dart* file. We can see that it is a BackgroundLayer:

```
class SplashLayer extends Ranger.BackgroundLayer {
```

It has a factory that takes a background color, a centering flag and width and height:

```
factory SplashLayer.withColor(Ranger.Color4<int> backgroundColor, [bool centered = true, int width, int
height]) {
```

The onEnter event calls a private method that adds two **TextNode**s:

```
    Ranger.TextNode title = new Ranger.TextNode.initWith(Ranger.Color4IOrange);
    title.text = "Splash Screen";
    title.setPosition(-350.0, 50.0);
    title.uniformScale = 10.0;
    title.shadows = true;
    addChild(title, 10, 701);

    Ranger.TextNode version = new
Ranger.TextNode.initWith(Ranger.Color4IDartBlue);
    version.text = "${Ranger.CONFIG.ENGINE_NAME} ${Ranger.CONFIG.ENGINE_VERSION}";
    version.strokeColor = Ranger.Color4IWhite;
    version.strokeWidth = 1.0;
    version.shadows = true;
    version.setPosition(-600.0, -150.0);
    version.uniformScale = 15.0;
    addChild(version, 10, 702);
```

That is it. There isn't much more to the SplashLayer. The layer shows some text for 3 seconds as specified back in *template.dart* file:

```
  SplashScene splashScene = new
SplashScene.withReplacementScene(gameScene);
  splashScene.pauseFor = 3.0;
```

Note

Ranger comes with a few prebuilt Leaf Nodes, TextNode being one of them, that are designed mostly for the Unit tests and templates. They are not optimized for everyday usage, but they may still be useful as either an example or temporary Node. It is highly encouraged that you create your own Nodes that suite your particular game.

And then instantly transitions to the next Scene (GameScene in the template's case.)

At this point we have a pretty good idea of what Ranger is all about. There is certainly a lot more to Ranger and we will be covering those in due time of course. But I think it is time we have some fun and kickoff our Moon Lander game and in the process learn some new aspects of Ranger as well game design.

So Lets get started shall we!

Part II

UFOs

In Part II we began putting together the Moon Lander game. We now cover Scene flow, proper asset loading, animation and super awesome fun particle systems.

- Chapter 4: Under construction
- Chapter 5: Asset loading
- Chapter 6: Menus and the Main Course
- Chapter 7: Particle Systems

Chapter 4 Under Construction

As you have seen in the previous chapters Scenes and Layers are the foundation for game flow. Without Scenes Ranger won't even run.

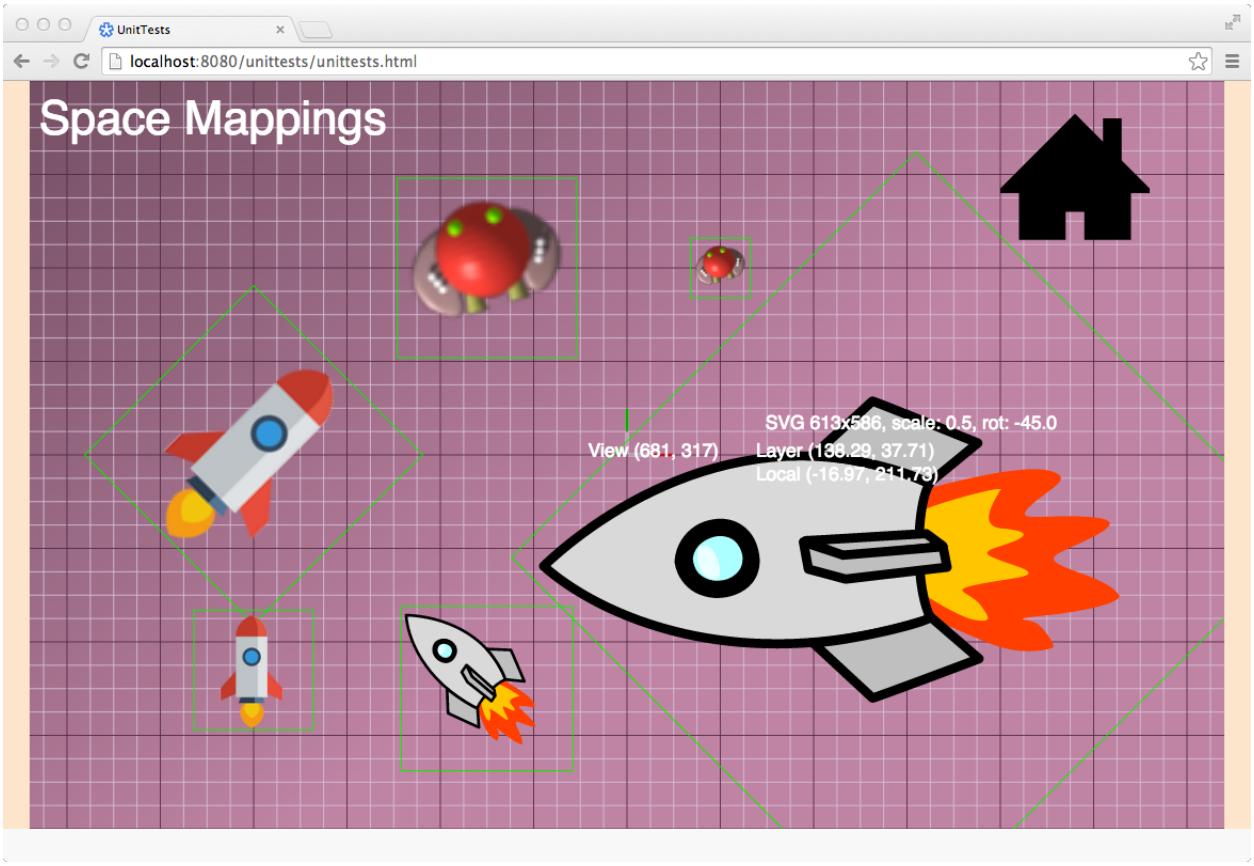


Figure 4.0 Space mappings Unit test

Above in Figure 4.0 is one of the Space mapping unit tests. As the cursor moves the mapping changes showing both local and layer space coordinates. The green boxes are the local bounding boxes.

In this chapter we will begin the construction of the Moon Lander game. To accomplish this we start with template level 3 and modify it to suit our needs. We also get our first taste at Animations and AnchorScenes.

Laying the foundation

Lets get started by first creating an application shell. There are several ways to go about it, but in this book will use Dart Editor as our tool. We are going to create a Web Application to be more specific.

Once the editor is running go to the “File/New Project” menu and choose “**web-app**” (Figure 4.1).

Enter “**MoonLander**” for the project name. Once Stagehand completes you will have a new Dart web application, and with it you will get a ton-o-files.

These files are not really suited for a self contained game environment like Ranger.

We are going to delete everything except the favicon.ico  .

Go ahead and remove the following items:

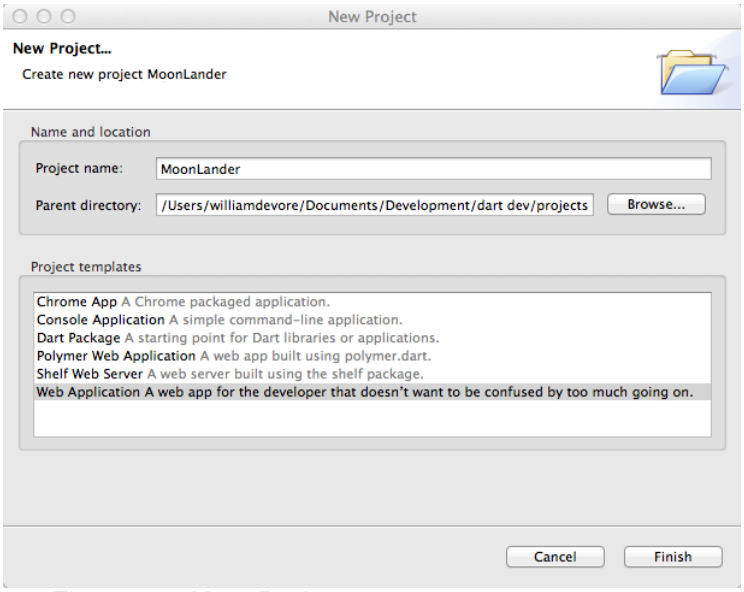


Figure 4.1 New Project

- Delete the styles folder
- Delete apple-touch-icon-precomposed.png
- Delete the “images” folder
- Delete the robots.txt file
- You can also clean out the README.md file and populate it later.
- Delete the “lib” folder

Now remove some Package dependencies that Stagehand put in the pubspec.yaml file:

- Remove the sass entry
- Remove the script_inliner entry
- Remove the route_hiearchical entry
- Remove the transformers section that includes the inliner.
- And change the description to whatever you like.

What you should be left with are three things:

- pubspec.lock
- pubspec.yaml
- the *web* folder with the favicon.ico.

Now we copy a starter shell set from **Sack**. This set consists of three files and they can be found in the *web/applications/shell* folder:

- index.html <— Rename this to *moonlander.html*
- main.css
- main.dart

Rename *index.html* to ***moonlander.html***. That takes care of the base set of code. Our shell is still missing one thing that keeps us from running it—the Ranger library.

> Note
>
> As of version 1.8, Dart's Editor now uses Stagehand project generator to create new projects.

Where is Ranger?

Ranger is a Dart library that is bundled into a Dart Package. To get Ranger we need to fetch it from **GitHub**. This can done by simply adding an entry to our project's *pubspec.yaml* file. Lets go ahead and do that now.

Double click on the pubspec.yaml file. This will open up an overview form. From here click the "**Add…**" button in the "Dependencies" area. Notice that Ranger isn't listed in the large list box. That is okay, just type "ranger" and click "OK". Why? Because we are adding a new reference not referencing an existing one from Dart's Pub. The list box shows Packages from Dart's Pub and Ranger isn't in Dart's Pub it's in located on GitHub so we must create a new entry and configure this entry in the Dependency Details area (Figure 4.2.)

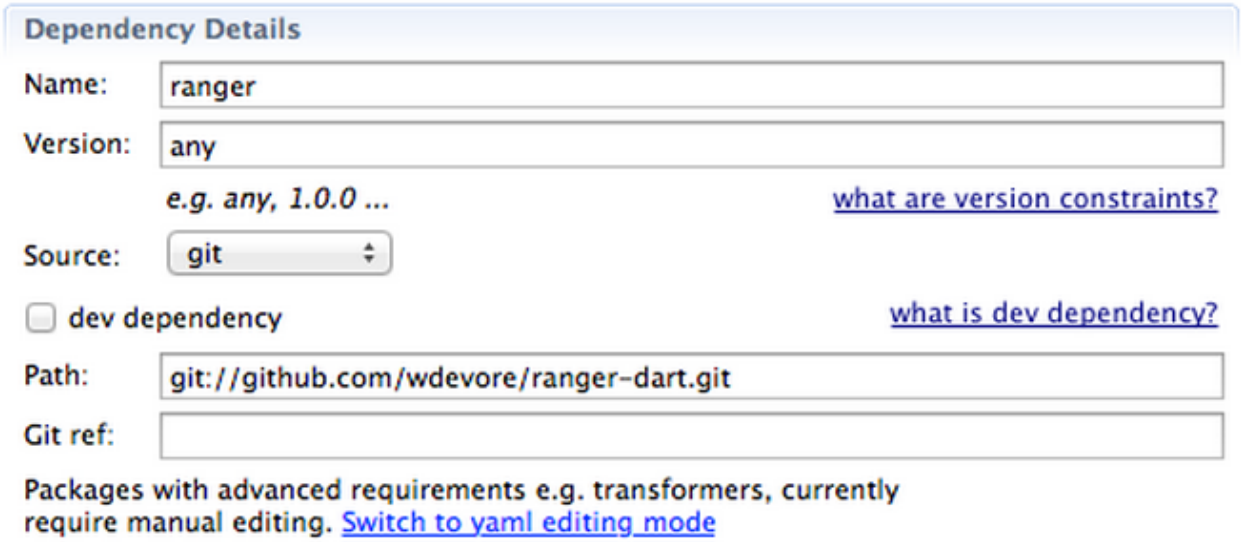


Figure 4.2 Dependency Details - Overview

Select as the source "**git**" and set the "**Path**" to:

```
git://github.com/wdevore/ranger-dart.git
```

Who is [wdevore]? That is me the creator of Ranger, the same person who created the GitHub project. ;-)

As soon as you save the yaml file you will see several libraries being downloaded including Ranger from Github.

```
Resolving dependencies...
+ event_bus 0.3.0+2
+ ranger 0.9.5 from git git://github.com/wdevore/ranger-dart.git
+ tweenengine 0.11.1
+ vector_math 1.4.3
Changed 4 dependencies!
```

Ranger uses these libraries to extend or provide core features.

- **event_bus** is for message passing between disparate objects. For example, a menu item sending a message to a Layer object.
- **tweenengine** supplies all the animation features. For example, sliding a Node into view.
- **vector_math** provides vectors for transforms.

At this point we now have a simple shell. Lets go ahead and run it see what we have. Ctrl or Right click the *index.html* file and select “Run in Dartium”. Once Dartium has booted it launches our web-app (see Figure 4.2.)

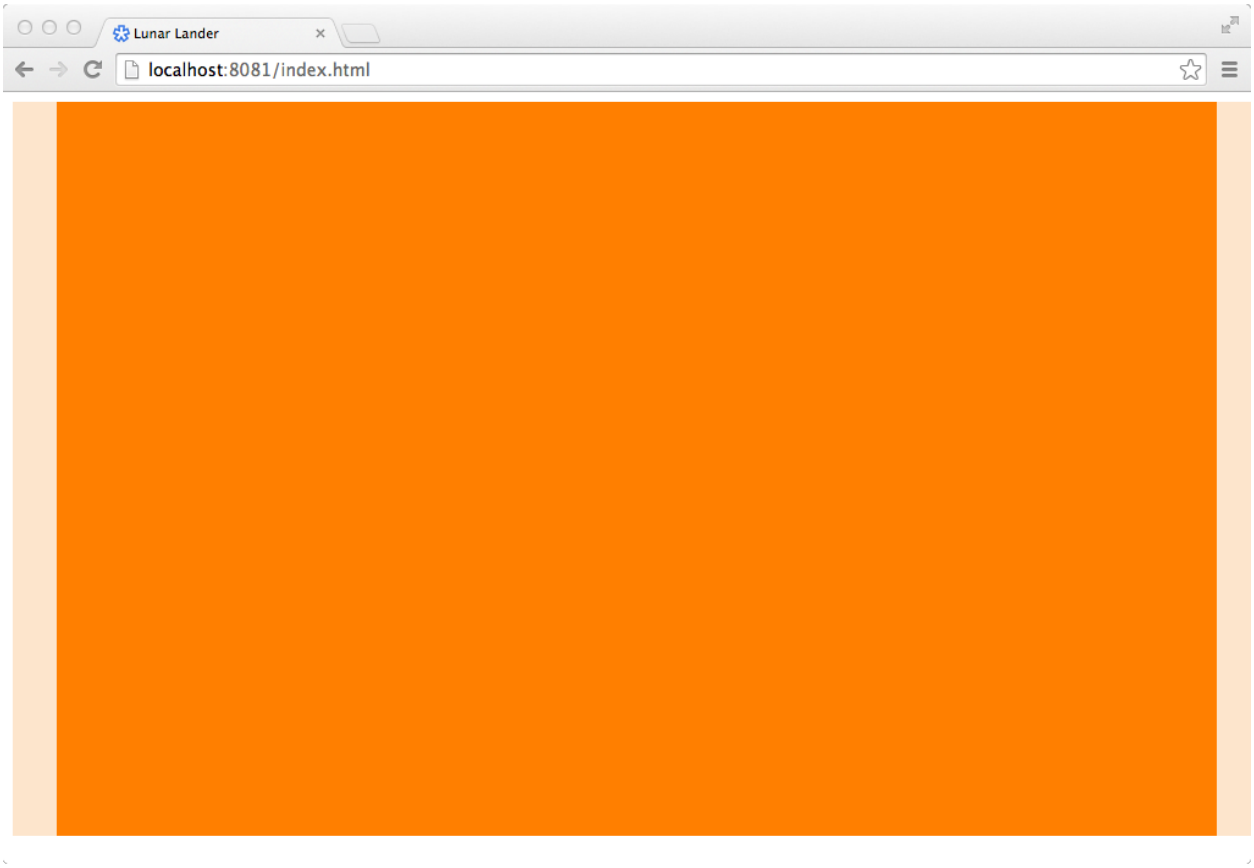


Figure 4.2 Simple shell

Not much to see except the Canvas surface—the **Orange** color. If you look at the output tab you will see that Ranger is

“stuck” in a booting sequence and that is because we haven’t coded the *preConfigure* callback.

But before we can code the callback we need some Scenes. How about we setup the Splash scene next.

Chapter 5 Asset Loading

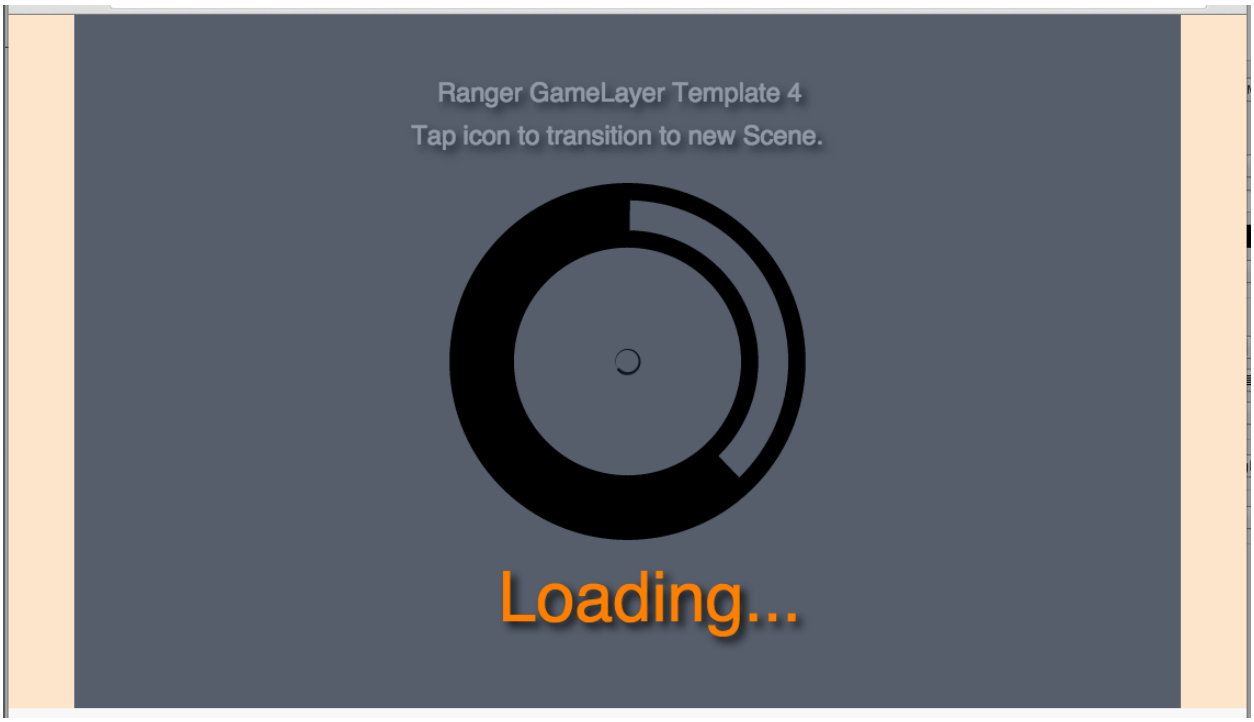


Figure 5.0 Template 4's asset loading

Above, in Figure 5.0, is template 4's asset loading example. We will be modifying template 3, which is bit simpler, to include a more formal resource loading construct.

In chapter 4 we built a simple shell and learned how to pulldown the Ranger library. In this chapter we code up an asset loading splash scene with a matching Layer. Then we setup a second Scene and Layer combo to provide for our entry point. All of these Scenes will be used inside the preConfigure method.

Goal

We want to be able to launch MoonLander and have it run a splash scene while loading an asset then transition to a main menu scene using a Slide-In transition (see Figure 5.1.) This will all happen when we add code to the preConfigure callback. The **BootScene** is part of Ranger so we only need to code the SplashScene/SplashLayer and MainScene/MainLayer combos.

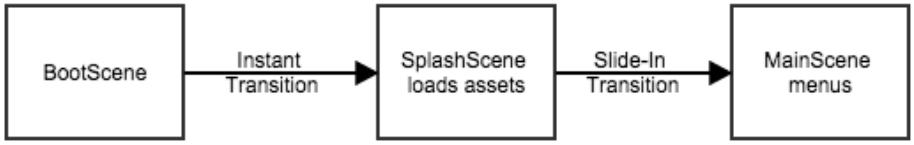


Figure 5.1 Scene flow

The SplashScene will show an animated icon while it loads a background logo, and once that is complete the scene will hang around for a another second or two and “slide in” the MainScene. Out of the three scenes the only scene that isn’t an AnchoredScene is the BootScene.

AnchoredScene

The **AnchoredScene** (AS) is the work horse of Scenes. It is designed specifically for transition animations. Think of AS as a mini scene graph all packaged up into one Node.

It works by taking a primary Layer (in our case the SplashLayer) and internally attaching it as a child Node underneath a **SceneAnchor** Node. The SceneAnchor itself is a child of the AS. And there is a reason for this, animations.

The AS is designed to work with specific types of animations, for example, Slide-In and Move-In types. For now we are going to cover AS from a surface level. For a complete understanding of why and how the AS works refer to chapter “Scene Graph”.

The AS works by using a parent/child combination to control both translations, rotations and scaling (see upper part of Figure 5.2.) This effectively isolates the primary layer (aka SplashLayer) from direct transforms on the primary layer, the primary has no idea it is being transformed.

The current animation system expects this relationship. For example, the **SlideIn** transition animates the AS’s position property.

But there is more to it than that. The SceneAnchor can be moved relative to the primary

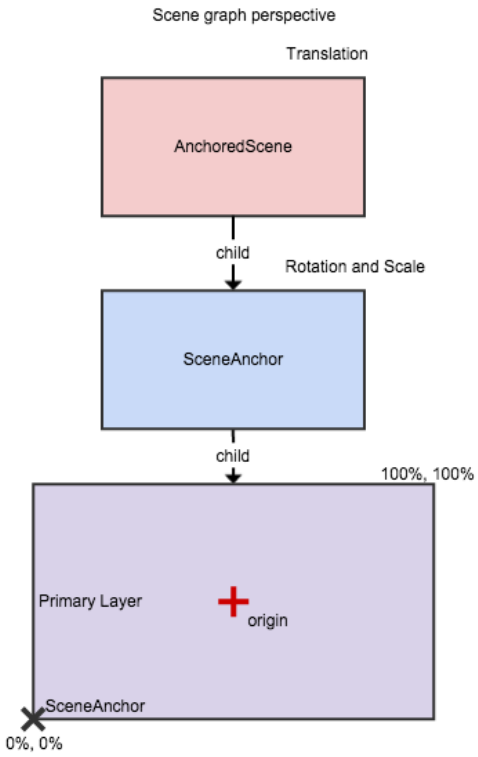


Figure 5.2 AnchoredScene

layer so that you can control where the Scale and Rotation epicenter occurs. Internally AS is translating the primary layer in the opposite direction of the anchor to create the effect.

You can specify the anchor's absolute location—including out of view—or you can specify the location in percentages as defined by the view. This later control is the most common way to set the location. The percentages (as shown in lower part of Figure 5.2) are based in the lower left corner.

For example, to place the anchor location in the lower-right corner you would specify (100%, 0%). Any rotations would cause the primary layer to appear to rotate about the lower-right corner. A really good example of setting the anchor location can be seen in the **TransitionFanInFanOut** transition, in this transition the "incoming" Scene's anchor is set to the upper-right (100%, 100%) and the "outgoing" Scene's anchor is set to the lower-left (default).

Remember back in "Introduction to Ranger" chapter the "beware" note? Not only is the "incoming" Scene's anchor set to the upper-right but the anchor's rotation is set to -90.0 degrees. This forces the incoming Scene out-of-view which is just we what want because the animation is going to rotate the Scene ***back into view***!

Hint

As with most Nodes in Ranger the AnchoredScene is also an example. You are encouraged to create your own AnchoredScene Nodes that are likely to be more complex.

Splash Scene

Now that we know a bit about AnchoredScenes how about we create our SplashScene scene. Lets take an easer route and copy it from Sack's template #3. Create a new folder under the *web* folder called *scenes* and copy *templates/level3/splash* folder into our new *scenes* folder—included in the copy is the splash layer too. Open *splash_scene.dart* and change the **part of** statement from "template3" to "moonlander".

```
part of moonlander;
```

Copying them isn't enough, we need to include them as well. Open *main.dart* and add a **part** statement to match our SplashScene, and while we are at it add a **part** statement for the copied SplashLayer too.

```
part 'scenes/splash/splash_scene.dart';
part 'scenes/splash/splash_layer.dart';
```

We should now have a library "moonlander" made up of two parts: SplashScene and SplashLayer (Figure 5.3.)

Note

Looking at the code for SplashScene you may notice a "**tag**" property being set with what appears to be random numbers. They aren't random at all. Tags are a simple way to help uniquely identify your Nodes during debugging, but they can also be used to *find* nodes using the **getChildByTag** method.

Looking at the copied we see the order to meet our the *transition* **TransitionInstant**:

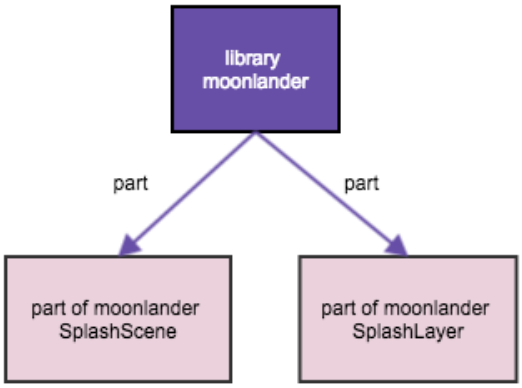


Figure 5.3 Library layout.

Scene and Layers we just need for modifications in goals; first, the transition in method is an

```
Ranger.TransitionScene transition = new
Ranger.TransitionInstant.initWithScene(_replacementScene);
```

Second the SplashLayer creates the wrong text and we need to remove the Ranger version.

```
Ranger.TextNode title = new
Ranger.TextNode.initWith(Ranger.Color4IOrange);
title.text = "Splash Screen";
title.setPosition(-350.0, 50.0);
title.uniformScale = 10.0;
title.shadows = true;
addChild(title, 10, 701);
```

Finally we need to add a logo asset.
Putting it all together here is what we want our splash scene to look and act like:

- Change text to "Moon Lander" and have it animate into view from the top.
- The text color should be chalk white on a dark gray background with a shadow.
- A loading spinner should animate while our logo is retrieved. Also the text "Loading" should be placed next to the spinner. Once it is retrieved we start animating the moon text into view and hide the spinner and loading text.
- Pause for 2 seconds and attempt to transition to the next scene. We haven't created the MainScene yet so it will simply stay on the SplashScene from a thrown exception.

Beware

TransitionScene will throw an exception if you attempt to create a transition without providing an incoming scene to transition to.

Lets get started. First we change the text to “Moon Lander” and change the color using the helper function *color4IFromHex* to create a pantone color of “#e5e1e6” and make sure shadows are turned on. Also we increase the text size and recenter it:

```
Ranger.TextNode title = new
Ranger.TextNode.initWith(Ranger.color4IFromHex("#e5e1e6"));
title.text = "Moon Lander";
title.setPosition(-550.0, 75.0);
title.uniformScale = 20.0;
title.shadows = true;
addChild(title);
```

Because SplashLayer inherits from **BackgroundLayer** we get grouping behavior mixed in

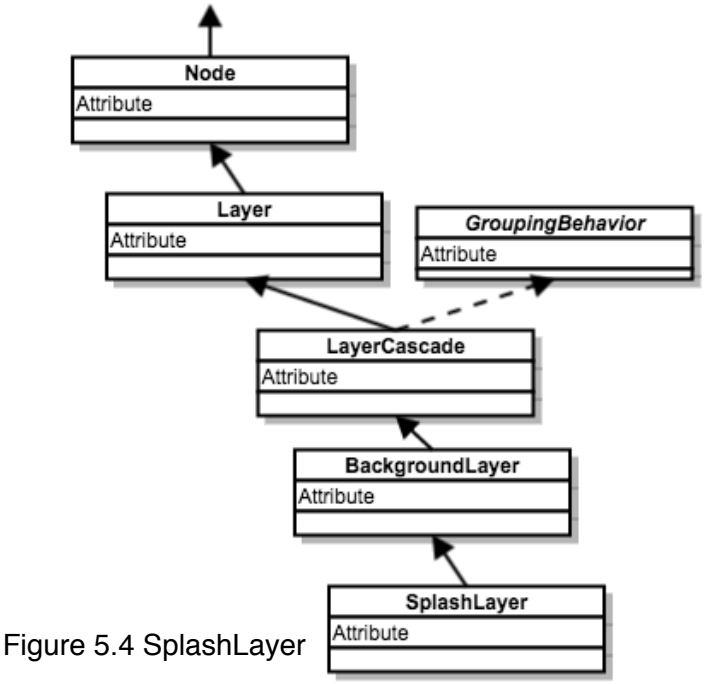


Figure 5.4 SplashLayer

Figure 5.5 Splash half baked

via a **GroupingBehavior** mix-in (see Figure 5.4.) This allows us to add the **TextNode** as a child.

Lets change the background color next. Go to the *onEnter* method inside SplashScene and change the color passed to the factory to a pantone color of “#4b4f54”.

If we run it now you will see Figure 5.5. Looks good. The text appears correctly centered horizontally, but we still need to move it off screen so that we can animate it back onto the screen while the asset is being loaded.

Go ahead and change the current title position code from:

```
title.setPosition(-350.0, 50.0);
```

to:

```
double h = contentSize.height;
title.setPosition(-550.0, h / 2.0 + 50.0);
```

The layer has a size and that size is specified by the *contentSize* object. But how do we now where the text will move to? For that we need to know the orientation of the coordinate system.

Coordinate System

By default Ranger's coordinate system is configured where the +Y axis is progressing from bottom to top (i.e. your thumb is upward, see Figure 5.6.) Ranger calls this orientation the left-handed system. The other orientation is +Y is down—the right-handed system. Knowing which one is active is important for predicting vertical translations and rotations.

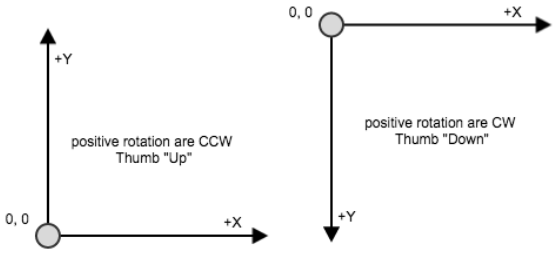


Figure 5.6 Coordinate Systems.

> Note
>
> You can change this orientation by setting the **base_configuration_system** property of the **Config** class. It is pretty rare that you would do this.

Because the +Y axis is upward that means we need to "shift" the text upward by half the screen height plus some text height offset. This will put the text *just* hanging above the screen out of view, perfect for animating the text downward from above. Keep in mind that we won't animate it until the asset is loaded. So lets work on that now.

Asset Loading

Our goal with asset loading is to display an animated icon while resources are being loaded. This is a bit of a chicken-and-egg situation here. The animated icon is an asset too, so how do we deal with that?

There are several ways to handle it but for now we will use an embedded Base64 encoded asset that travels along with the application. Ranger comes with two small Base64 resources: **spinner** (32x32) and **spinner2** (512x512).

No loading is required, just simply pass one of them to an HTML **ImageElement.** Here is an example of using spinner2:

```
spinner2 = new ImageElement(
    src: Ranger.BaseResources.svgMimeHeader +
Ranger.BaseResources.spinner2,
    width: 512, height: 512);
```

With this ImageElement you can create a sprite using **SpriteImage**.*withElement* factory:

```
Ranger.SpriteImage si = new Ranger.SpriteImage.withElement(spinner2);
```

Now we are ready to create a resource manager to handle our spinner:

1. Create a folder under the *web* folder called ***resources***.
2. Create a file called *resources.dart*.
3. Include this in our application by adding a **part** entry to *main.dart* —> part 'resources/resources.dart';
4. In the resources.dart file create a new class called **Resources**.

The Resources class will contain all the resources for MoonLander, and our first resource is the spinner. So lets do that now.

> Note
>
> If you are wondering how these spinners where created there are websites that will take any text and Base64 encode it. The website used to create the embedded spinners was MobileFish. Just paste your SVG text and convert, then embed the encoded text as a Dart **String**.

Futures

A goal for any game engine is to support async asset loading—we don't want the display paused while assets are being loaded. Dart has async programming built right in under the nomenclature of **Futures** or Async/Await.

When we need to load an asset we create an object called **Completer** which in turn provides a Future that we can return while the loading process does its work. In order to use Futures we need to import the Dart **async** package.

Go ahead and add a new **import** statement to *main.dart*:

```
import 'dart:async';
```

Now return to the Resource class and add a class-scoped Completer variable:

```
class Resources {
  Completer _loadingWorker;
…
```

We will use this Completer in a new method called *load*. The method will return the Completer's Future:

```
Future load() {
  _loadingWorker = new Completer();
  // DO WORK HERE…
  return _loadingWorker.future;
}
```

Add a constructor for the Resource class that builds the spinner SpriteImage:

```
Resources() {
  ImageElement spinner2 = new ImageElement(
    src: Ranger.BaseResources.svgMimeHeader + Ranger.BaseResources.spinner2,
    width: 512, height: 512);
  spriteSpinner2 = new Ranger.SpriteImage.withElement(spinner2);
}
```

Remember we don't have to do any asynchronous work because the spinner is already part of Ranger's code-base as a **String** inside **BaseResources**. After each asset is loaded we need a away to track when all assets have been loaded. One way is to use a counter. For each asset needed we increment a counter and then decrement once the asset has loaded.

We will add two counters, an *_updateLoadStatus* method and a property to monitor loading state. Add the counters first:

```
  int _resourceCount = 0;
  int _resourceTotal = 0;
```

Add the boolean property next:

```
bool get isLoaded => _resourceCount == _resourceTotal;
```

Now add the update method:

```
  void _updateLoadStatus() {
    _resourceCount++;

    if (isLoaded) {
      _loadingWorker.complete();
    }
  }
```

Every time *_updateLoadStatus* is called we increment the resource counter, and once the counter has reached the max total we know that all assets have loaded.

We now have the basic infrastructure to handle async loading of assets, but we need another method for performing the actual loading of images (our logo being one of them.) Add a *loadImage* method that returns a Future using Ranger's **ImageLoader**:

```
Future<ImageElement> loadImage(String source, int iWidth, int iHeight, [bool
simulateLoadingDelay = false])
{
    Ranger.ImageLoader loader = new Ranger.ImageLoader.withResource(source);
    loader.simulateLoadingDelay = simulateLoadingDelay;
    return loader.load(iWidth, iHeight);
}
```

Before we can finish coding the *load* method we need our logo asset in our project's path.

The logo SVG image, Lander1.svg  , was created using InkScape. You can download it from this book's source-code repository located on GitHub under the folder *BookAssets* (see Source code section.)

Once you have copied Lander1.svg into the project's **resources** folder go ahead and add a line that increments the total resource counter:

```
_loadingWorker = new Completer();

_resourceTotal++; // Lander1
```

For every resource that we are loading we increment the counter.

With the counter incremented we can now load the asset. When I created the SVG image I noted its size. We need that size for properly setting the ImageElement dimensions. Once it is bound to a SpriteImage Node we can scale it accordingly.

This is the nice feature of SVG based ImageElements is that HTML will treat the image as a vector graphic meaning it is scaled without pixelation as apposed to bitmapped graphics that pixelate. Lets go ahead and load the asset using our *loadImage* method. Note that we pass a **true** as the last parameter to activate a simulated network delay:

```
loadImage("resources/Lander1.svg", 295, 466, true).then((ImageElement ime)
{
  logo = ime;
  _updateLoadStatus();
});
```

Above we are loading an image using the Future's **then** statement to "synchronize" and update the loading status. The *_updateLoadStatus* checks the counts and if they are equal it signals the **_loadingWorker** to **complete**. In turn our SplashLayer's **then** statement will be invoked on the *load* method.

Hint

When developing on a desktop the network delay will be near zero. To simulate a network delay you can set the ImageLoader's ***simulateLoadingDelay*** property to True prior to calling the ImageLoader's *load* method, the loader will then generate a random delay.

Recap

Using **Future**s we created a class that can asynchronously load assets (see Code 5.8). All that is required to load more assets, for example JSON settings and Scores while incrementing the counter for each asset and make sure to call *_updateLoadStatus* when the asset is loaded.

Now we can return to SplashScene and integrate our asset loading code.

Code 5.8

```
part of moonlander;

class Resources {
  Completer _loadingWorker;

  int _resourceCount = 0;
  int _resourceTotal = 0;

  Ranger.SpriteImage spriteSpinner2;
  ImageElement logo;

  Resources() {
    ImageElement spinner2 = new ImageElement(
        src: Ranger.BaseResources.svgMimeHeader + Ranger.BaseResources.spinner2,
        width: 512, height: 512);
    spriteSpinner2 = new Ranger.SpriteImage.withElement(spinner2);
  }

  Future<ImageElement> loadImage(String source, int iWidth, int iHeight, [bool
simulateLoadingDelay = false]) {
    Ranger.ImageLoader loader = new Ranger.ImageLoader.withResource(source);
    loader.simulateLoadingDelay = simulateLoadingDelay;
    return loader.load(iWidth, iHeight);
  }

  bool get isLoaded => _resourceCount == _resourceTotal;

  Future load() {
    _loadingWorker = new Completer();

    _resourceTotal++; // Lander1

    loadImage("resources/Lander1.svg", 295, 466, true).then((ImageElement ime) {
      logo = ime;
      _updateLoadStatus();
    });

    return _loadingWorker.future;
  }

  void _updateLoadStatus() {
    _resourceCount++;

    if (isLoaded) {
      _loadingWorker.complete();
    }
  }
}
```

SplashScene continued

Two things we can do prior to loading assets is to show a spinner that spins while animating a title. Lets create a *beforeResourcesLoaded* method inside SplashLayer that is called just before assets begin loading.

The most appropriate place to call this method is during the *onEnter* method inside SplashScene because it is the SplashScene that will create SplashLayer which just happens to the place were we will create and add the spinner Node to the SplashLayer. However, there is one more thing we need to cover in order to fulfill our goal of an animated spinner—Animations.

Animations

We want our spinner to spin and the way to do that is by using the TweenEngine (TE). At first TE can be a bit of a challenge to use so Ranger provides an optional helper wrapper called **TweenAnimation** that acts both as a source for learning TE and as manager for simple animations. In UTE there are two ways to animate your objects:

1. Implementing the **TweenAccessor** generic or
2. inheriting from **Tweenable**.

Both accomplish the same thing but from two different perspectives. The choice depends on what kind of access you have to code. For example, if you don't have the freedom of changing your object (i.e. it's part of a library) then the TweenAccessor is your only choice. However, if you do have access to the code then Tweenable is your "optional" choice.

The real difference is how you "register" them with TE. TweenAccessor requires you to register a relationship between a Class that you will animate and a TweenAccessor object that will do the work on the Class, doing this allows you to animate literally any object. It is for this feature alone as to why I Ranger chose TE as its animation engine. We will see an example of this usage when we animate the "Moon Lander" text.

Using the Tweenable approach doesn't require registration because TE already understands objects that inherit from Tweenable. For Moon Lander we will use both depending on what kind of object and how simple the animation is.

Just about anything can be animated, but the two most common are Transformations and Color or use the TweenAnimation class to rotate a SpriteImage Node simply by registering the SpriteImage class in relation to a TweenAccessor (hint the TweenAnimation class):

```
UTE.Tween.registerAccessor(Ranger.SpriteImage, ranger.animations);
```

Once registered we can apply an animation by using the TweenAnimation class (aka ranger.animations.) For example, if we wanted to rotate a SpriteImage by a relative amount we could use TweenAnimation's *rotateBy* helper method:

```
    UTE.Tween rot = app.animations.rotateBy(
        si, lapTime, deltaDegrees,
        UTE.Linear.INOUT, null, false);
```

And this is precisely what we are going to do. However, there is certainly more to animations than what we just covered. For an in depth coverage of Animations be sure to read Chapter 16 on “Tweening Animations.” In there we dive deeper into Tweenable(s) by looking at how they are created and what possibilities they have. For now we have enough surface knowledge to add code for our Resources class and *beforeResourcesLoaded* method.

Lets think about something for a moment. Our game is going to load assets at various stages of the game—probably between levels. It would advantageous to place the creation code inside our Resources class such that all levels have access to the same Node. We need to add a new method to the Resource class called *getSpinnerRing*. This method will create a rotation animation and apply it to our spinner sprite Node that we created in the constructor.

In order to use Animations in our project we need to include the TE library. Go back *main.dart* and **import** the library:

```
import 'package:tweenengine/tweenengine.dart' as UTE;
```

Return to Resources class and add the *getSpinnerRing* method along with some animation code (see Code 5.9.)

Code 5.9

```
Ranger.SpriteImage getSpinnerRing(double lapTime, double deltaDegrees, int
tag) {
  spriteSpinner2.tag = tag;

  Ranger.Application ranger = Ranger.Application.instance;
  UTE.Tween rot = ranger.animations.rotateBy(
      spriteSpinner2,
      lapTime,
      deltaDegrees,
      UTE.Linear.INOUT, null, false);

  rot..repeat(UTE.Tween.INFINITY, 0.0)
     ..start();

  return spriteSpinner2;
}
```

The main area of interest in Code 5.9 is the **app.animations.rotateBy** call. This one call creates an animation which targets our sprite (spriteSpinner2). Recall that the **Application** class is Ranger’s main API. The Application exposes the TweenAnimation object as a property called “**animations**”.

The TweenAnimation class has a huge laundry list of animation builder methods. They range from Tinting, Fading, Color, Rotation, Scaling, Moving etc., and each on them properly constructs a TE Tween animation plus automatically adds the tween to a TE TweenManager. The only thing we would need to do is make sure the SpriteImage class is registered, and good place to do that is in the *init* method of our SplashLayer.

The *rotateBy* method, much like all the other animation methods, takes parameters that describe what to animate and how the animation is to function. Lets look at the *rotateBy* method signature (see Code 5.10.)

Code 5.10

```
UTE.BaseTween rotateBy(
            Node node,
            double duration,
            double deltaDegrees,
            UTE.TweenEquation easeEquation,
            [Object userData,
             bool autoStart = true]) {
```

- **Node**: This is the Ranger Node that will the target of the animation.
- **duration**: This is how long the animation will last.
- **deltaDegrees**: This animation is relative so the angle is a delta from the current orientation.
- **easeEquation**: What type of easing to against the linear animation. Ex: ease in or or out of. Without easing all animations would start into and stop out-of the animation in a linear manner.
- **userData**: Any type of data that needs to be associated with the animation, handy for callbacks.
- **autoStart**: Should the animation start immediately or let the caller start it. This is typically set to false for Sequences, Parallels and Repeats, basically situations where you need to do work prior to the animation starting. For example, the repeat value can't be set after the animation has started.

The last line prior to the **return** statement in code 5.9 is an example of needing **autoStart** = **false**. This is because we need to set a repeat value (**Tween**.INFINITY in this case.) Once it is set we can finally start the animation:

```
rot..repeat(UTE.Tween.INFINITY, 0.0)
   ..start();
```

That takes care of the creating and configuring the spinner. Now lets put it to the test.

First we override Node's init method inside our SplashLayer. This isn't required but it does help distribute code better (Code 5.11), and then we register the SpriteImage class in association with Ranger's TweenAccessor (aka **ranger.animations**).

Code 5.11

```
@override
bool init([int width, int height]) {
  if (super.init(width, height)) {
    // We need to register the SpriteImage class so that the
    // Tween Engine (TE) recognizes the class.
    UTE.Tween.registerAccessor(Ranger.SpriteImage, ranger.animations);

    _configure();
  }

  return true;
}
```

Next we return to the SplashLayer's *beforeResourcesLoaded* method and retrieve the spinner from the Resources class and add it as a child to the layer. But before we do this lets establish a global object for common access throughout our game, we will call it **GameManager**.

GameManager

There are certain types of data that need to be accessed by all objects. You may think of using a Singleton object but they are considered an Anti-pattern and rarely used. You can also use messages or tightly couple by passing an object to all constructors and factories.

For Moon Lander we are going use a simple global-scoped object defined in *main.dart*. For now it will contain only the Resources object. Go ahead and create a new file called *game_manager.dart* in the main *web* folder, and remember to include this in the *main.dart* file:

```
part 'game_manager.dart';
```

Create a new Class called **GameManager** and add our Resources class to it. Be sure to add the **part of** statement (Code 5.12.)

Code 5.12

```
part of moonlander;

class GameManager {
  Resources resources = new Resources();
}
```

Return to *main.dart* and create the GameManager (Code 5.13)

Code 5.13

```
GameManager gm = new GameManager();

// Ranger application access
Ranger.Application ranger;
```

With the Resources and GameManager objects created we can now finish coding SplashLayer's *beforeResourcesLoaded*. Recall that the goal of this method is to configure a spinner and add it as a child to the Layer prior to async asset loading. We have several things we need to do:

1. Get the spinner and configure it.
2. Add it as a child so that it becomes visible.
3. Add an orange "Loading" text Node.

First we get the spinner from **Resources** and configure it:

```
Ranger.SpriteImage spinner = gm.resources.getSpinnerRing(1.5, -360.0,
7001);
    spinner.uniformScale = 0.5;
    spinner.setPosition(0.0, 0.0);
```

Setting the position of the spinner to (0.0, 0.0) puts it right in the middle of the view, this is because the Layer was centered during the *withColor* factory call, also, the SVG image is kind of big so we scale it down while were at it.

The spinner we get back from Resources is animated forever so we need a way to cancel it when the SplashScene exits. Ranger's TweenAnimation can do this using the *track* method. So lets add a line that will track the animation:

```
ranger.animations.track(spinner, Ranger.TweenAnimation.ROTATE);
```

Now we add it to the layer:

```
addChild(spinner);
```

If you run Moon Lander now you should see a slowly spinning ring and after about 2 seconds it stops (see Figure 5.14), again this is because we haven't coded the MainScene yet so the game is still trying to transition into oblivion.

That spinner looks awfully lonely. How about adding some accompanying text just below it (Code 5.15)

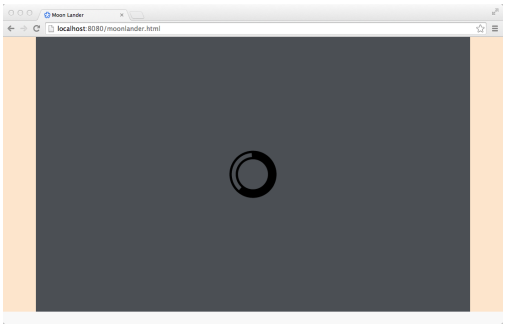


Figure 5.14 spinner animating.

Code 5.15

```
    Ranger.TextNode loading = new
Ranger.TextNode.initWith(Ranger.Color4IOrange);
    loading.text = "Loading";

    double h = contentSize.height;
    loading.setPosition(-280.0, -230.0);
    loading.uniformScale = 15.0;
    loading.shadows = true;
    addChild(loading);
```

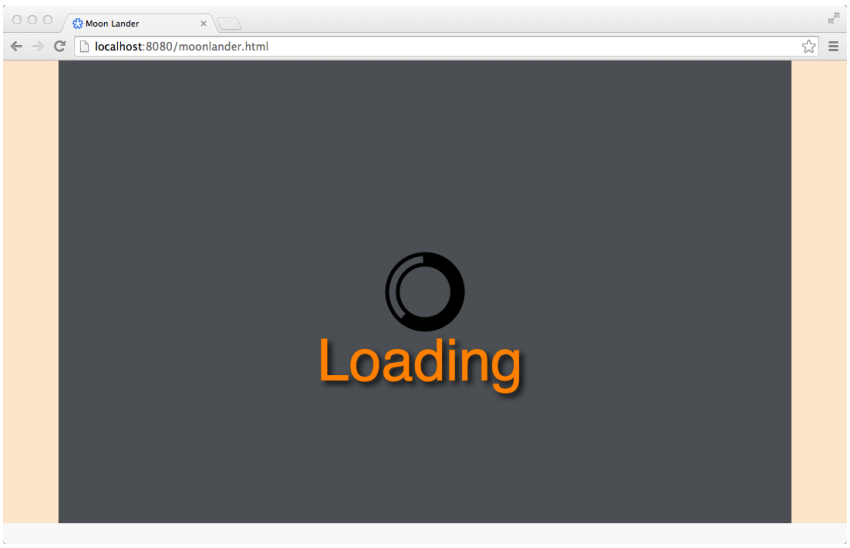


Figure 5.16 Loading

That looks better (Figure 5.16.) Now we animate the Title into view. First register the TextNode with the animation system:

```
UTE.Tween.registerAccessor(Ranger.TextNode, ranger.animations);
```

Now we create our first animation using Ranger's **TweenAnimation** helper.

```
    UTE.Tween move = ranger.animations.moveBy(
        _title, 1.0,
        -_deltaTitleY + 150.0, 0.0,
        UTE.Cubic.OUT, Ranger.TweenAnimation.TRANSLATE_Y, null);
```

Above we are animating for 1 second, translating by the distance the text is out view plus an offset. The offset helps the title stops just above the spinner and finally a cubicly easing out-of the animation. What you finally have is Figure 5.16a.

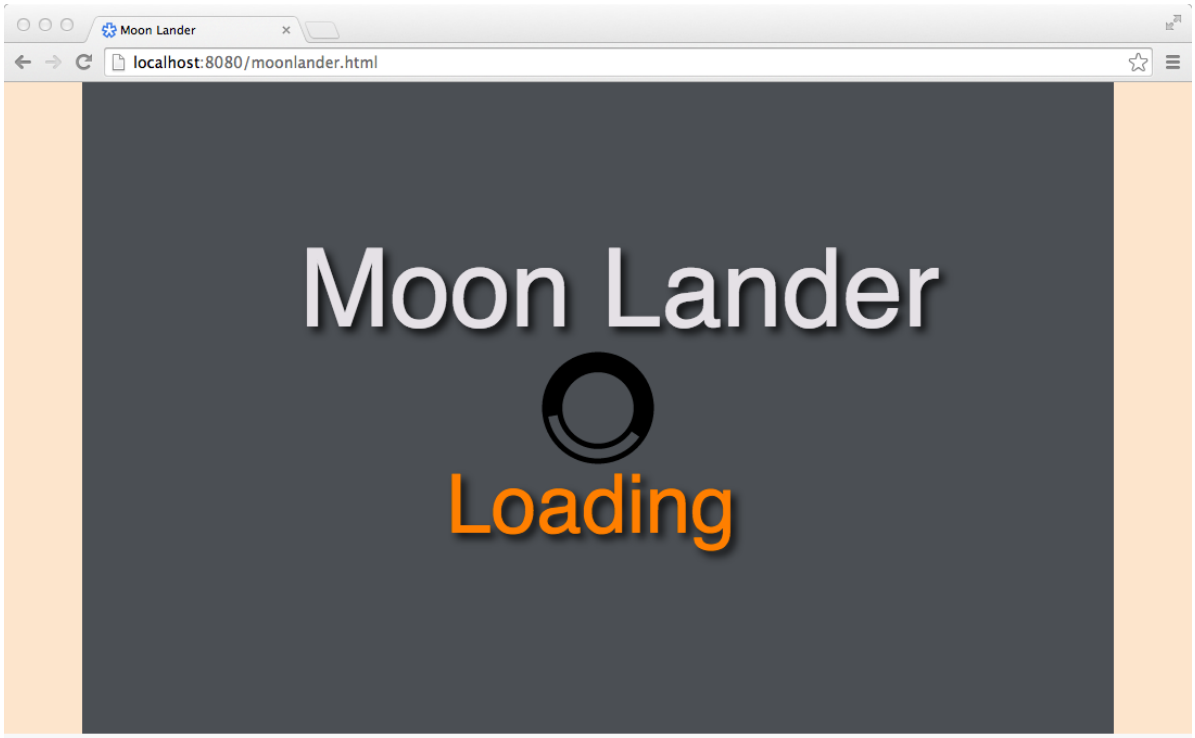


Figure 5.16a Loading complete

There is one more thing we should do before we return to SplashScene. We just added an infinite animation and began tracking it and we loaded an asset. We need to cover the other side of the "contract" by adding an *afterResourcesLoaded* method (see Code 5.17.) TweenAnimation has a *flushAll* method that literally stops all animations that you tracked. It is particularly useful for infinite animations, and while we are at it we should hide the spinner and loading text too.

Code 5.17

```
void afterResourcesLoaded() {
  // Stop any previous animations; especially infinite ones.
  ranger.animations.flushAll();

  _loading.visible = false;
  _spinner.visible = false;
}
```

Notice that I didn't remove the spinner and text Nodes. We don't have to because the default behavior of Scenes is to "clean" themselves up upon exiting "off stage". That clean up process involves removing any child Nodes.

With both before and after covered lets return to SplashScene's *onEnter*, where the spinner goes-to-town spinning, and finish the async loading process. Just below the call to *beforeResourcesLoaded* add a call to the Resources's *load* method:

Code 5.18

```
// Async load resources for the first level.
gm.resources.load().then((_) {
  splashLayer.afterResourcesLoaded();
  transitionEnabled = true;
});
```

Also, move the "transitionEnabled = **true**" line into the anonymous "**then**" statement. We don't want to transition until the assets have finished loading. Just as a reminder we enabled artificial network delay while loading the asset. If we didn't we wouldn't see the spin because of the desktop's near zero delay.

Congratulations! We just finished an asset loading sequence. Granted we can't "see" our asset yet but at least we loaded it. Lets recap.

Recap

So far the entire sequence starts from *main.dart*'s *preConfigure* method where the SplashScene is constructed. Once the SplashScene reaches the top of the scene stack the SceneManager calls the *onEnter* method (see Figure 5.19.)

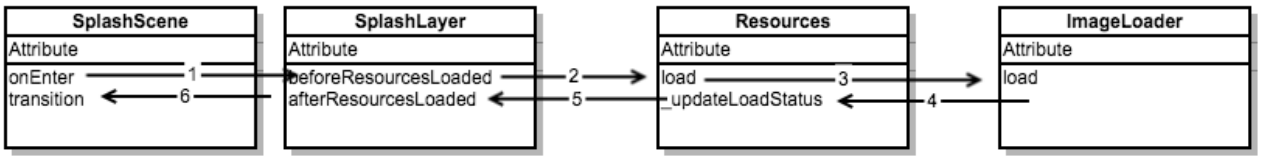


Figure 5.19 Asset loading sequence.

Here is how things play-out:

1. *onEnter* creates a primary layer (aka SplashLayer)
2. calls layer.*beforeResourcesLoaded*
3. *beforeResourcesLoaded*
    1. gets, configures and adds an animated spinner
    2. creates a TextNode and adds it too
    3. Spinner spins
4. Async loading begins
5. Asset is loaded
    1. *afterResourcesLoaded* is called
    2. transition is enabled
6. Transition finishes resulting in the next scene being placed on the stack.

Now that the assets are loaded we are ready to "transition" to the next scene, literally.

Chapter 6 Menus and the Main Course

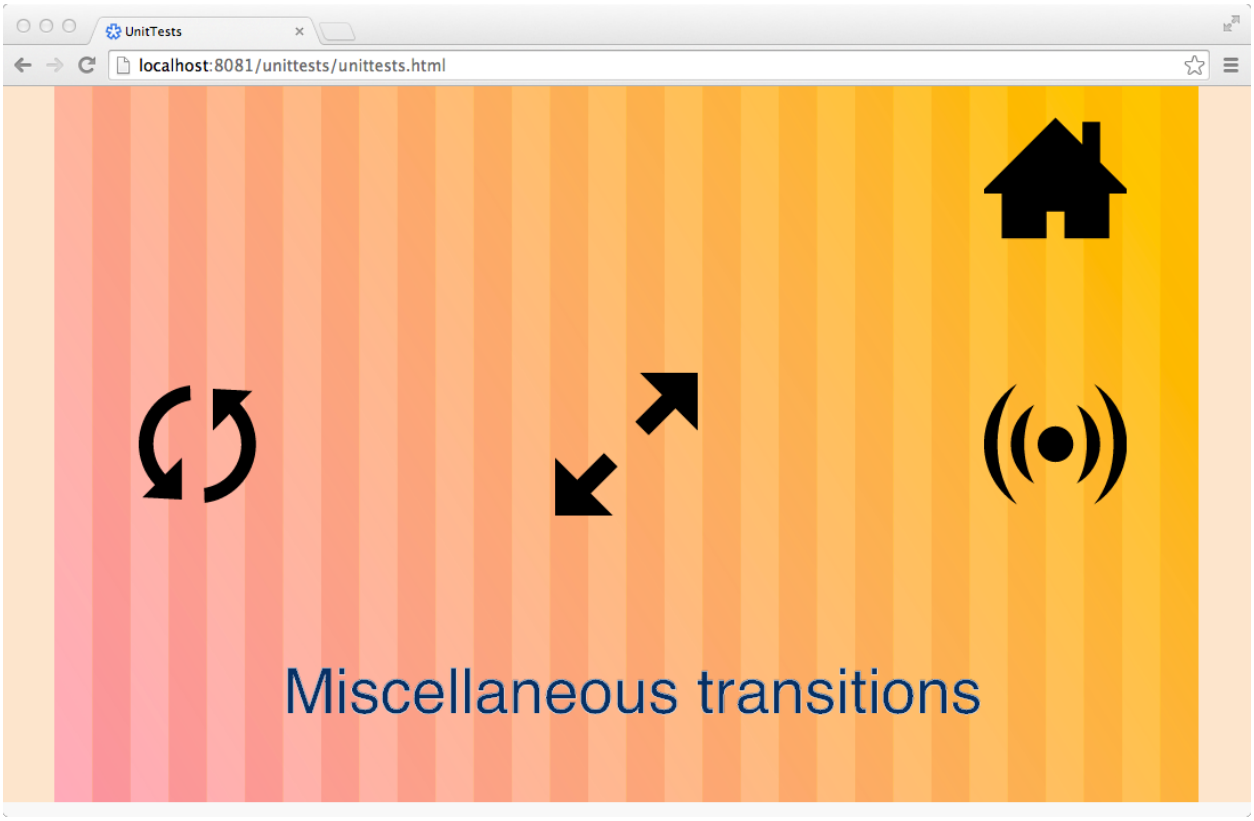


Figure 6.0 Another Ranger face.

Above in Figure 6.0 we see another one of Ranger's faces—Miscellaneous transitions.

In chapter 5 we setup the SplashScene to asynchronously load assets.

In this chapter we complete it by transitioning to a new Scene called MainScene. Instead of an Instant transition we will use a Slide-In transition. In addition, we will put together a beginner version of the MainScene that will allow the player to change settings and transition to a game level by clicking a play button. Through this we will get a better idea of how the SceneManager works with the scene stack.

Our goal is two fold in this chapter: create a scene for SplashScene to transition to and design and build a main entry scene along with some shell scenes for later use.

MainEntry Scene

Lets layout an agenda of what we need to do:

1. Create a new MainScene scene
2. Finish coding SplashScene's transition using MainScene
3. Build out MainScene with menu controls and navigation
4. Create a simple settings shell
5. Create a simple level selector shell

After we complete these goals we will have a basic flow of Moon Lander. It's just a start but good enough to know what lies ahead. Lets get started.

In the previous chapter once the assets were loaded the SplashScene would attempt to instantly transition to a nonexistent Scene. Lets fix that by creating an actual scene to transition to called "**MainScene**". Here are the abbreviated steps:

1. Create a folder named *game* under the *scenes* folder
2. Copy *splash_scene.dart* and *splash_layer.dart* in to the *game* folder
3. Rename both files to *main_scene.dart* and *main_layer.dart* respectfully
4. Rename *main_scene.dart*'s class to MainScene
5. Rename *main_layer.dart*'s class to MainLayer
6. Update MainScene to create a MainLayer instead of SplashLayer
7. Include both files in *main.dart*
8. Update *main.dart*'s *preConfigure* to create a MainScene and pass it to SplashScene's *withReplacementScene* factory as the replacement scene. Now the app will stop throwing an exception trying to transition into nowhere and instead transition to our new awesome MainScene.
9. Remove MainScene.*withReplacementScene* and *withPrimary* factories as they aren't needed. *onEnter* will create and assign a primary layer
10. Remove the asset loading code in MainLayer's *onEnter* method
11. Optionally you can remove all the .**tag** assignments sprinkled everywhere
12. Keep the **title** Node in MainLayer as we are going to animate it upward a little.

Once you have completed the steps above we can begin working on **MainScene**. To begin strip the MainScene class down to the bare bones by removing everything but the constructor and *onEnter* method. Leave just the creation of the primary MainLayer (see Code 6.1.) Because we know that MainScene will undergo a Transition animation we need to add a line that resets the Scene's position. If we didn't do this then when we return to it from the LevelSelectionScene we won't see anything but orange—the canvas surface.

Also we add a check for an existing primaryLayer such that we don't redundantly create another Layer when we return to this Scene via a stack "pop".

Code 6.1

```
class MainScene extends Ranger.AnchoredScene {
  MainScene([int tag = 0]) {
    this.tag = tag;
  }

  @override
  void onEnter() {
    super.onEnter();

    if (primaryLayer == null) {
      MainLayer layer = new
MainLayer.withColor(Ranger.color4IFromHex("#4b4f54"));
      initWithPrimary(layer);
    }

    // Reset Scene's position just incase it was animated by a
Transition
    setPosition(0.0, 0.0);
  }
}
```

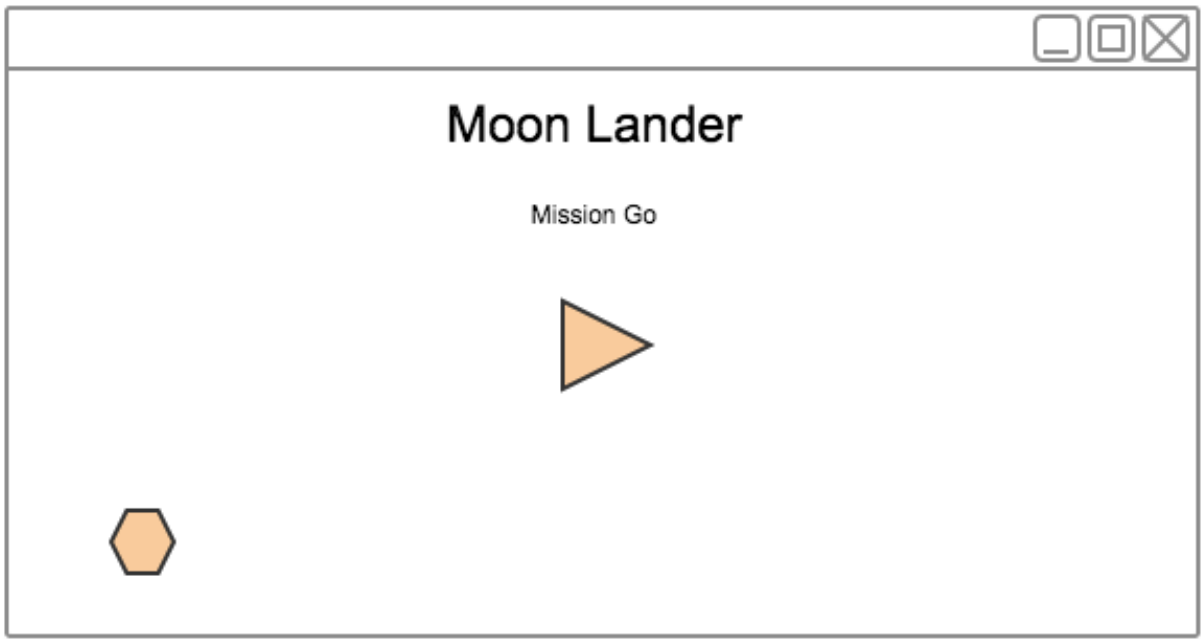


Figure 6.2 MainScene draft

Our next goal is to code the MainLayer where the user interaction is. To help guide us we put together a quick drawing in Gliffy as a guide (see Figure 6.2.)

The triangle represents the Lander1.svg asset  we loaded during the SplashScene. When the player clicks the icon we perform a rotation animation and then transition to the level selector scene. The hexagon will be a slowly rotating gear SVG icon  for activating the settings dialog.

This drawing suggests we need to load an additional asset (the gear icon) and enable mouse input. More over, both icons will send the player to a new Scene or dialog. The lander icon will send them to a level selector Scene and the gear will display a dialog. In both cases we want the player to have the ability to return to the main scene, and that sounds like another icon asset is need—perhaps a  return icon.

So lets head back to Resources class and add two more assets and copy the SVG files from this book's assets repository (https://github.com/wdevore/Ranger-MoonLander)—we will add the return icon later.

- Copy *gears.svg* and *back.svg* in to the resources folder.
- Add two class-scoped ImageElements to the Resources class named: **gear** and **back**.
- Inside Resources.*load* method
  - Increment **_resourceTotal** += 2;
  - Add two more *loadImage* statements (Code 6.2)

Code 6.2

```
    _resourceTotal += 2;    // gear and back icons
    loadImage("resources/gear.svg", 32, 32,
true).then((ImageElement ime) {
      gear = ime;
      _updateLoadStatus();
    });
    loadImage("resources/back.svg", 35, 32,
true).then((ImageElement ime) {
      back = ime;
      _updateLoadStatus();
    });
```

- Return to MainLayer's *_configure* method and add the **gear** and **back** icons then configure according to the draft (see Figure 6.2. and Code 6.4.)

MainLayer

We want the gear in the lower left corner so we use the contentSize as a guide and use a percentage to displace it from the edge. The gear is actually tiny compared to our design dimensions so we scale it up by 4 units.

Again, we use Ranger's TweenAnimation helper to create an infinite animation. Recall that we need to stop the animation when the scene exits. We can track the Nodes that have infinite animations ourselves or have TweenAnimation track it. Either way we need to stop them when

the scene exits. To do this we need to override the *onExit* method/signal sent by the SceneManager. Seeing as we know what Node is animated we use the *flush* method to target a specific Node:

```
ranger.animations.flush(_gear);
```

Code 6.3 shows the *onExit* method overridden. The method is typically used for pausing or cleaning resources. Note, this is slightly different from what we did with the SplashLayer in that the SplashScene explicitly called the *afterResourcesLoaded* method to flush any infinite animations. We could have just as easily added an *onExit* method to SplashLayer and flushed animations there.

Code 6.3

```
  @override
  void onExit() {
    super.onExit();

    // Stop animations when this scene leaves the stage.
    ranger.animations.flush(_gear);
  }
```

In Code 6.4 we create the gear sprite using the ImageElement resource, and track it for later when the scene exits.

Code 6.4

```
    _gear = new Ranger.SpriteImage.withElement(gm.resources.gear);
    _gear.uniformScale = 4.0;
    _gear.setPosition(-w + (w / 7.0), -h + (h / 5.0));
    addChild(_gear);

    // Lander
    _lander = new
Ranger.SpriteImage.withElement(gm.resources.lander1);
    addChild(_lander);
```

Recall that we are creating the animation in a yet-to-be-started state because we need to adjust the **repeat** value (Code 6.4b.) We separated the animation into it own method because we also need to call it in the *onEnter* method when we return from another Scene, and while we are at it we also reset the Lander's orientation too (see Code 6.6.)

Code 6.4b

```
  void _rotateGear() {
    UTE.Tween rot = ranger.animations.rotateBy(
        _gear,
        12.0,
        360.0,
        UTE.Linear.INOUT, null, false);
    //                  v---------^
    // Above we set "autostart" to false in order to set the repeat
value
    // because you can't change the value after the tween has
started.
    rot..repeat(UTE.Tween.INFINITY, 0.0)
       ..start();

    ranger.animations.track(_gear, Ranger.TweenAnimation.ROTATE);
  }
```

Let see our handy-work in action. Go ahead Launch Moon Lander now. You should see a slowly spinning gear in the lower-left corner along with a rocket dead center (see Figure 7.5.)

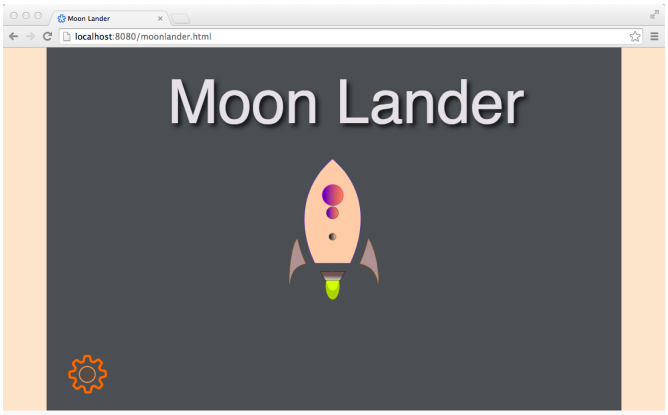


Figure 6.5 Gear and Lander

Well that is great but how to do we configure the icon for input sensitivity?

Mouse Input

Enabling mouse input is as simple as adding one line of code in the *onEnter* method (Code 6.6.)

Code 6.6

```
@override
void onEnter() {
  enableMouse = true;
  super.onEnter();

  // Reset Nodes in case we return to this Layer.
  _lander.rotationByDegrees = 0.0;

  _rotateGear();
}
```

Of course you will also want to disable any input, including mouse, in the *onExit* method respectfully. So add a *disableInputs* call in the *onExit* method (Code 6.7.)

Code 6.7

```
@override
void onExit() {
  disableInputs();
…
```

Why does this work? Because **BackgroundLayer** inherits from three **Mixins** (Figure 6.8.)

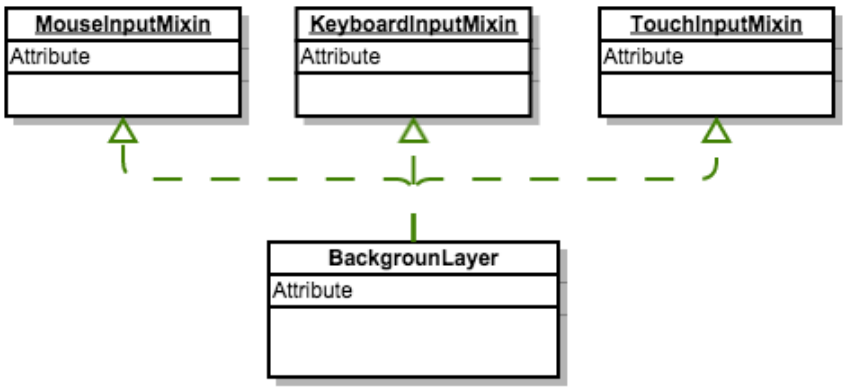


Figure 6.8 Background Layer

Input is one of the key features of Layers. The Mixins provide bindings to the window/browser events and overridable methods for receiving those events.

Having enabled mouse input we can now override the *onMouseDown* method to begin getting mouse down events. The event passed contains an **offset** object that represents the mouse coordinates relative to the window—not the desktop—which is why it is called offset. However, mouse-space is not synonymous with our game's world-space but it is with view-space. So how do we map from mouse-space (aka view-space) to a useful space? For that we need to revisit Scene Graphs.

Scene Graph

Internally Ranger manages Nodes in a Scene Graph (SG). We covered SGs briefly in the chapter “Introduction to Ranger”, and we are going to cover them a little more in depth here, but still not as deep as the chapter dedicated to Scene Graphs.

From our point of view SGs provide two main features:

1. Render order (later version of Ranger will most likely separate this away from the SG).
2. Space mapping

Render order is mostly top-to-bottom and left-to-right with the ability to locally override this using Z-order values. Space mapping is the process of transforming a point from one coordinate system into another. There are several Spaces present in Ranger—not including your device which has several others:

- **View-space**: Window offset space, typically mouse-space within the browser. You map to this space when you want to maintain a position relative to the other spaces. An example of its usage is AutoScrolling.
- **World-space**: The Root of the SG. An intermediate space used to transform between spaces within the SG itself. This is used quite often.
- **Parent-space**: A semi local space between two sibling Nodes. Used when you know the siblings’ parent is untransformed.
- **Local-space**: Any Node in the SG has a local-space. You most often want to map to it in order to perform things like collision tests.
- **Design-space**: A space that you specify during construction. It directly influences the pre transform matrix outside of the SG. There isn’t any real reason to map to it.

DrawContext

The space we want to map to is the Local-space of one of the icons—for example, the rocket. In order to map from view-space to the local-space of the rocket we use Ranger’s **DrawContext** (Figure 6.9.)

DrawContext is a thin wrapper around the HTML5 Canvas object. It main purpose is to provide basic state management and view-space mappings. One of the more common mappings is from View to Node space. In this case you want to map from a mouse click location into the equivalent Node location.

There are times when you may want to map from View to World space when you need to perform a series or sequence of tests against multiple Nodes. It saves computation to map only once to World-space.

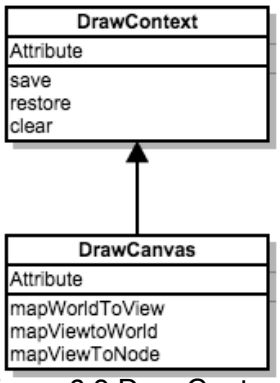


Figure 6.9 DrawContext

MainLayer Continued

In order to detect that the rocket icon was clicked/touched we have two things we need to do:

1. Map the click location to Local-space of rocket and
2. perform a collision check using local coordinates

Lets override **MouseInputMixin**'s *onMouseDown* method and add mapping and collision code that checks for a click on the rocket icon. First add the mapping code using Ranger's DrawContext:

```
Ranger.Vector2P nodeP = ranger.drawContext.mapViewToNode(_lander,
event.offset.x, event.offset.y);
nodeP.moveToPool();
```

mapViewToNode takes a View-space point (aka offset) and maps that to the Local-space of the Node provided. But what is that *moveToPool* method? That is Ranger's opt-in pooling.

Note

All of the mapping methods return a **Vector2P** object which is simply a pooled version of **Vector2**. Generally it is a good idea to move the object back to the pool after you are done with it and not before.

Pooling

Ranger's pooling is based on **Dartemis** and it is completely optional to participate in. At any time you can elect not to call a Node's *moveToPool*, and by *not* doing so you are basically saying "I don't care if the garbage collector (GC) collects it". But if you *do* call *moveToPool* then the pool will hold a reference to it thus it keeping from the prying eyes of the GC. Of course the flip-side is that you are consuming memory. So be conscious of how you use pooling. Some Scenes and Variables are temporary and likely never to be seen again during the lifetime of the game. In that case you can probably forgo pooling by simply forgetting to move the object back to the pool.

Here is a code snippet that occurs in the Ranger-Rocket app where we are performing collision detection between the ship's bullet and one of the shapes, (Code 6.10.)

Code 6.10

```
Ranger.Vector2P pw = p.node.convertToWorldSpace(_localOrigin);
Ranger.Vector2P nodeP = _squarePolyNode.convertWorldToNodeSpace(pw.v);
collide = _squarePolyNode.pointInside(nodeP.v);
if (collide) {
  _handleBulletToSquareCollide();
  pw.moveToPool();
  nodeP.moveToPool();
  return p;
}

pw.moveToPool();
nodeP.moveToPool();
```

In the Ranger-Rocket app we needed to map the bullet that is in GroupNode space into the Local-space of the Node we want to check for a collision. Once the check is complete we don't need the pooled versions of the **Vector2** objects any longer so we "move them" back to the pool. I chose to move them to the pool because on the very next "update" I am going to perform the very same check again. So it makes sense to "reuse" pooled objects rather than giving the GC a colossal migraine; and when the GC has a migraine you will feel the pain too as your game "pauses" while the GC takes its revenge.

Here is another scenario. Let's say that when your bullet hits a square Node it explodes and disappears never to be seen again. Your code will probably call *removeChild*(child, **true**). Pay attention to the last parameter. In this case the SG will detach the child *and* move the square Node back to the pool. The question is "when do you use **true** (default) or **false**"?. It all depends on your game. Perhaps squares will appear unending and your player's task is to destroy them forever; in this case it may be prudent to use **true**.

If it is a Node that will never appear again and memory is of no concern then **true** is also a valid choice, however, if resources are scarce perhaps **false** is better and you just have to deal with the GC. The less work the GC has to do the less noticeable the pause will be.

With basic knowledge of pooling and mapping we can code our *onMouseDown* like so (Code 6.11.)

Code 6.11

```
  @override
  bool onMouseDown(MouseEvent event) {
    Ranger.Vector2P nodeP =
ranger.drawContext.mapViewToNode(_lander, event.offset.x,
event.offset.y);

    if (_lander.pointInside(nodeP.v)) {
      nodeP.moveToPool();
      // Transition to level selector.
      return true;
    }

    nodeP.moveToPool();
    return false;
  }
```

Notice the usage of *mapViewToNode* to get the Local-space point of the rocket icon. With the the Local-space point we can check if it's inside the rocket's bounding box using SpriteImage's *pointInside* method. If the method returns **true** then a click occurred on top of the icon and in this case we would transition to the LevelSelectionScene. In this instance once we are done with the pooled Vector2P we return it back to the pool.

Our other requirement was that the rocket rotates prior to transitioning to the next scene. To do this we will use a new feature of TweenEngine—**TweenCallbackHandlers**.

TweenEngine's TweenCallbackhandler

TweenCallbackhandlers are called anytime a trigger filter is meet. There are eight events to filter on:

- **BEGIN**: right after the delay (if any)
- **START**: at each iteration beginning
- **END**: at each iteration ending, before the repeat delay
- **COMPLETE**: at last END event
- **BACK_BEGIN**: at the beginning of the first backward iteration
- **BACK_START**: at each backward iteration beginning, after the repeat delay
- **BACK_END**: at each backward iteration ending
- **BACK_COMPLETE**: at last BACK_END event

Once the filter is meet your callback is triggered which must have a signature type defined as:

```
void TweenCallbackHandler(int type, BaseTween source);
```

We want to filter on the **COMPLETE** event, hence when we create our rotation animation we immediately set a callback and trigger filter. For this we need to add several pieces of code:

1. Reset rocket's orientation
2. Create a callback method matching signature of TweenCallbackHandler
3. Create a transition method that handles the transition
4. Create a rotation animation using TweenAnimation helper class
5. Set the animation's callback
6. Set the animation trigger filter
7. Start the animation

Beware

TweenEngine animations are not aware of Ranger's pooling system. By default when a Scene exits it moves any pooled objects back to the pool. Upon entering a Scene some of those pooled objects could still be active in the animation system. It is recommend that any and all animations are stopped when a Scene exits. This way the next Scene won't allocate Nodes that may still be under animation. Consider using TweenAnimation's tracking system to help minimize collisions, and make sure you call ***ranger.animations.flushAll*** inside *onExit* if need be.

Code 6.12 shows the animation code inside of *onMouseDown* during rocket detection. I choose to use a Linear animation with a 0.25 second duration while rotating the rocket counter clockwise. A positive rotation is used because our game is using the default base coordinate system. Remember there are two systems; one with +Y upwards and the other with +Y downwards. The other system would have caused a clockwise rotation for a positive rotation value. Again, I set the AutoStart flag to **false** so I can set the callback and trigger filters before the animation starts.

Code 6.12

```
  _lander.rotationByDegrees = 0.0;

  UTE.Tween rot = ranger.animations.rotateTo(
      _lander,
      0.25,
      90.0,
      UTE.Linear.INOUT, null, false)
    ..callback = _rotationComplete
    ..callbackTriggers = UTE.TweenCallback.COMPLETE
    ..start();
```

Code 6.13 shows the two methods for handling the callback and transition.

Code 6.13

```
  void _rotationComplete(int type, UTE.BaseTween source) {
    switch(type) {
      case UTE.TweenCallback.COMPLETE:
        // Transition to selection scene
        _transitionToSelections();
        break;
    }
  }

  void _transitionToSelections() {
}
```

How about we add more code to *onMouseDown* to detect the Gear icon (Code 6.14.) This time we won't need a callback unless we decide to do something fancy like scale-up the icon rapidly to show selection feedback.

Code 6.14

```
nodeP = ranger.drawContext.mapViewToNode(_gear, event.offset.x,
event.offset.y);

if (_gear.pointInside(nodeP.v)) {
  nodeP.moveToPool();
  // Transition to settings scene
  return true;
}

nodeP.moveToPool();
```

For the moment that takes care of MainScene and MainLayer. Visually nothing has changed except now our layer recognizes when Nodes are clicked on. We will return to MainLayer once we have created the next Scene, **LevelSelectionScene**.

LevelSelectionScene

For now the LevelSelectionScene will be pretty simple with just a single level to select from and a Back icon. Once the left most icon is selected they we be routed to the first level—the other two are “locked”  . If they click the Back icon they are returned to the MainScene.

This is going to be are very first opportunity to suspend a Scene. This happens when we transition from MainScene to LevelSelectionScene by using the SceneManger’s *pushScene* method. LevelSelectionScene is placed at the top of the stack causing MainScene to move down thus suspending it, at which point the SceneManager sends the **onExit** signal to MainScene.

Lets put together a quick draft of what the scene should look like (Figure 6.15.)

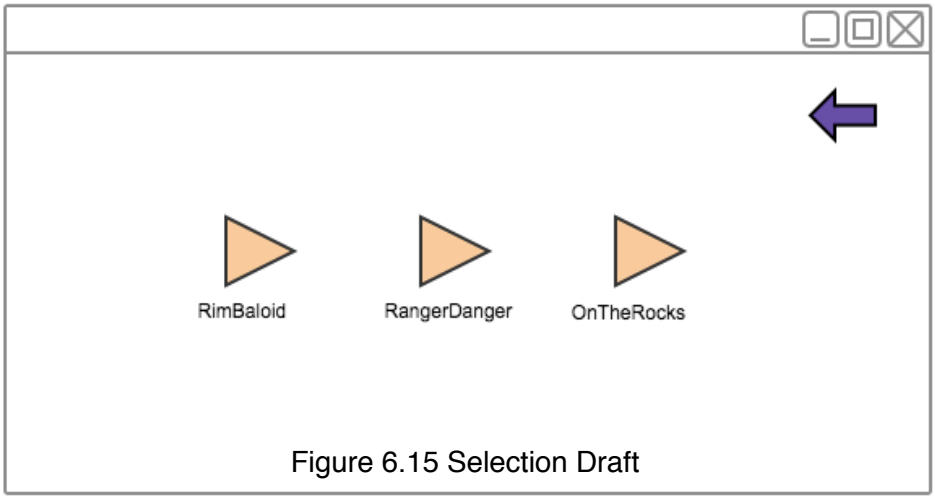


Figure 6.15 Selection Draft

From our design it appears we need to load at least four more assets, three rocket ships of different colors and the lock icon. The functionality will be pretty much the same as the MainScene other than different transition destinations.

So lets get coding. Perform these prep step first:

1. Copy three rocket ship SVG assets from the book's repository into the *resources* folder: rocket1.svg, rocket2.svg and rocket3.svg.
2. Copy lock.svg icon into the *resources* folder.
3. Update Resources class to increment and load the four new assets. Name their properties: **rocket1**, **rocket2**, **rocket3** and **lock** accordingly.
4. Copy *main_layer.dart* and *main_scene.dart* into the game folder and rename them *level_selection_scene.dart* and *level_selection_layer.dart* respectfully.
5. Update *main.dart* to include the two new Dart files in the game folder.
6. Rename the classes in each file to **LevelSelectionScene** and **LevelSelectionLayer** respectfully.

Now that the prep is done we begin by changing the background color of the LevelSelectionLayer in LevelSelectionScene's *onEnter*, for now we switch to a brownish color (#583d3e). Later we may implement something fancy like a star field.

Moving onto the LevelSelectionLayer we can remove the *flush* call from *onExit* because we don't have any infinite animations in progress:

```
~~ranger.animations.flush(_gear);~~
```

Change the **_title** Node from "Moon Lander" to "Mission Go!" and animate it into view from the top. To do that we set the **_title**'s position out of view just above the screen just like we did on the SplashLayer. Next ***override*** *onEnterTransitionDidFinish* method and add Code 6.16. This is our very first use of *onEnterTransitionDidFinish*.

Code 6.16

```
@override
void onEnterTransitionDidFinish() {
  super.onEnterTransitionDidFinish();

  UTE.Tween move = ranger.animations.moveBy(
      _title, 1.0,
      -_deltaTitleY + 200.0, 0.0,
      UTE.Cubic.OUT, Ranger.TweenAnimation.TRANSLATE_Y, null);
}
```

The *moveBy* method is nearly the same code as in SplashLayer but now the last parameter has been left off. This causes the animation to automatically start upon creation.

Return to the *_configure* method and create and position the three rocket sprite Nodes respectfully, and for each rocket we add a TextNode. For the last two rockets we create and add a **lock** sprite Node and place them relative to each locked rocket. The code is fairly large so Code 6.17 shows only one rocket setup. Each one follows the same pattern:

1. Add rocket SpriteImage, TextNode and Lock properties
2. Create rocket SpriteImage Node in *_configure* method
3. Configure rocket Node
4. Create match rocket title TextNode
5. Configure TextNode
6. Create lock Node
7. Configure lock Node

Code 6.17

```
  Ranger.SpriteImage _rocket2;
  Ranger.TextNode _rocket2Text;
  Ranger.SpriteImage _rocket2Lock;
  …
    _rocket2 = new
Ranger.SpriteImage.withElement(gm.resources.rocket2);
    _rocket2..uniformScale = 10.0
      ..setPosition(0.0, 0.0);
    addChild(_rocket2);
    _rocket2Text = new
Ranger.TextNode.initWith(Ranger.color4IFromHex("#e5e1e6"));
    _rocket2Text..uniformScale = 5.0
      ..text = "RangerDanger"
      ..setPosition(-160.0, -200.0)
      ..shadows = true;
    addChild(_rocket2Text);
    _rocket2Lock = new
Ranger.SpriteImage.withElement(gm.resources.lock);
    _rocket2Lock..uniformScale = 3.0
      ..setPosition(90.0, -80.0);
    addChild(_rocket2Lock);
```

In order to visually inspect what we have so far we need return to the MainLayer class and add code to the *_transitionToSelections* method that transitions to LevelSelectionScene (see Code 6.19,) this will also give us our first chance to use a slightly fancier transition effect—**TransitionSlideIn**.

If you look at the line that is "pushing" a Scene on the stack you may notice that it isn't our destination Scene but the **transition** scene instead. Why is this? Well transition scenes are Scenes in and of themselves too (See Figure 6.18.)

They too can be placed on the stack and "ran". Once they complete their transition the destination/incoming scene is placed on the stack and activated. Recall that the "incoming" Scene is activated—*onEnter* is called—***before*** the transition actually begins. The TransitionScene classes will make sure your destination scene isn't visible prior to starting the animation.

Code 6.19

```
  void _transitionToSelections() {
    // Transition to level selector.
    LevelSelectionScene inComingScene = new LevelSelectionScene();
    Ranger.TransitionSlideIn transition = new
Ranger.TransitionSlideIn.initWithDurationAndScene(0.5,
inComingScene, Ranger.TransitionSlideIn.FROM_RIGHT);

    ranger.sceneManager.pushScene(transition);
  }
```

Something to note, these transition scenes are transient meaning they're created and discarded on every use.

Run Moon Lander now. Click the rocket on the MainScene (see Figure 6.5). The MainScene should "slide" out the view to the left and at the same time LevelSelectionScene follows closely behind coming in from the right. When the transition finishes the "Mission Go!" drops down from the top (see Figure 6.20.)

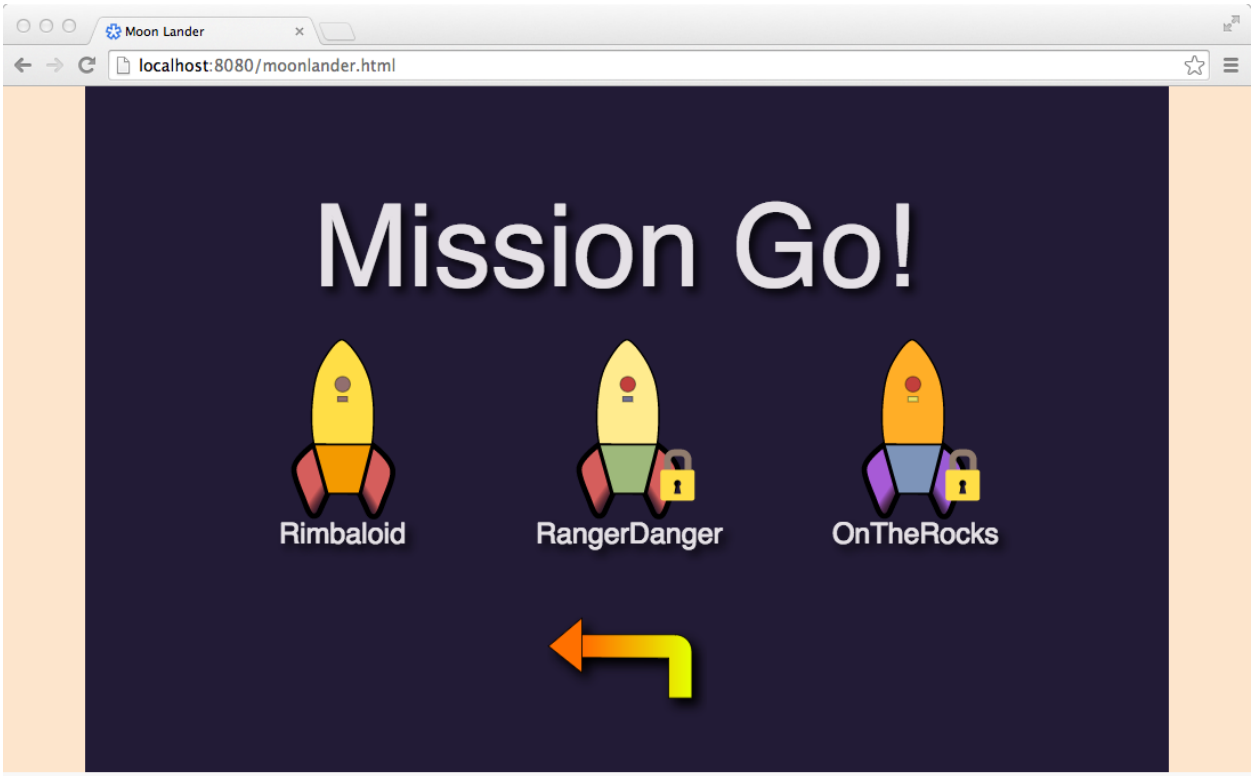


Figure 6.20 LevelSelectionScene

Returning home

Things are looking good! But wait! How do we get back home (aka MainScene)? For that we need to hook up the **Back** icon.

Because we copied MainLayer we should have code in *onMouseDown*. Go ahead and remove the gear mapping and collision code then retrofit the lander code to check the back icon instead. That takes care of detecting the Back icon, but how do we get back to the MainScene? Recall back in MainLayer._*transitionToSelection* that we used SceneManager's *pushScene* method to push LevelSelectionScene (actually TransitionSlideIn pushed it) onto the stack.

```
ranger.sceneManager.pushScene(transition);
```

What this means is that MainScene is still on the stack just below LevelSelectionScene. Therefore, all we have to do is "pop" what is on the top of the stack and we return back to MainScene (see Code 6.21)

Code 6.21

```
  @override
  bool onMouseDown(MouseEvent event) {
    Ranger.Vector2P nodeP = ranger.drawContext.mapViewToNode(_back,
event.offset.x, event.offset.y);

    if (_back.pointInside(nodeP.v)) {
      nodeP.moveToPool();
      // Transition back to MainScene.
      ranger.sceneManager.popScene();
      return true;
    }
    nodeP.moveToPool();

    return false;
  }
```

If we want to "slide in" back to the MainScene instead then we need to use another transition scene.

Beware

By keeping Scenes on the stack we are consuming resources that may be needed by the currently running scene. Depending on you game consider which Scenes need to be transient and which are not.

Recap

In this chapter we created our first Scene flow originating from the SplashScene all the way to LevelSelectionScene (see Figure 6.22.)

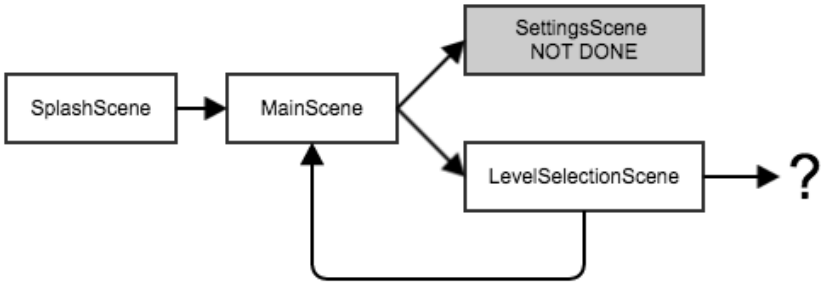


Figure 6.22 Scene flow

We learned how to map from View-space to Local-space in relation to each Node such that we can detect click/touch events, and in part we also attached an animation that animates complete with a callback. We also learned how to “push” and “pop” Scenes in order to create a simple scene flow. In the process we had to consider that some scenes are reentrant (aka MainScene) and as such we needed to reset various properties; in MainScene’s case we needed to reset its position because it underwent a scene transition.

We learned that in order to see something animate after a scene transition we needed to place code in *onEnterTransitionDidFinish*. Had we put the animation in the *onEnter* we may not actually see the whole animation.

Chapter 7 Nodes

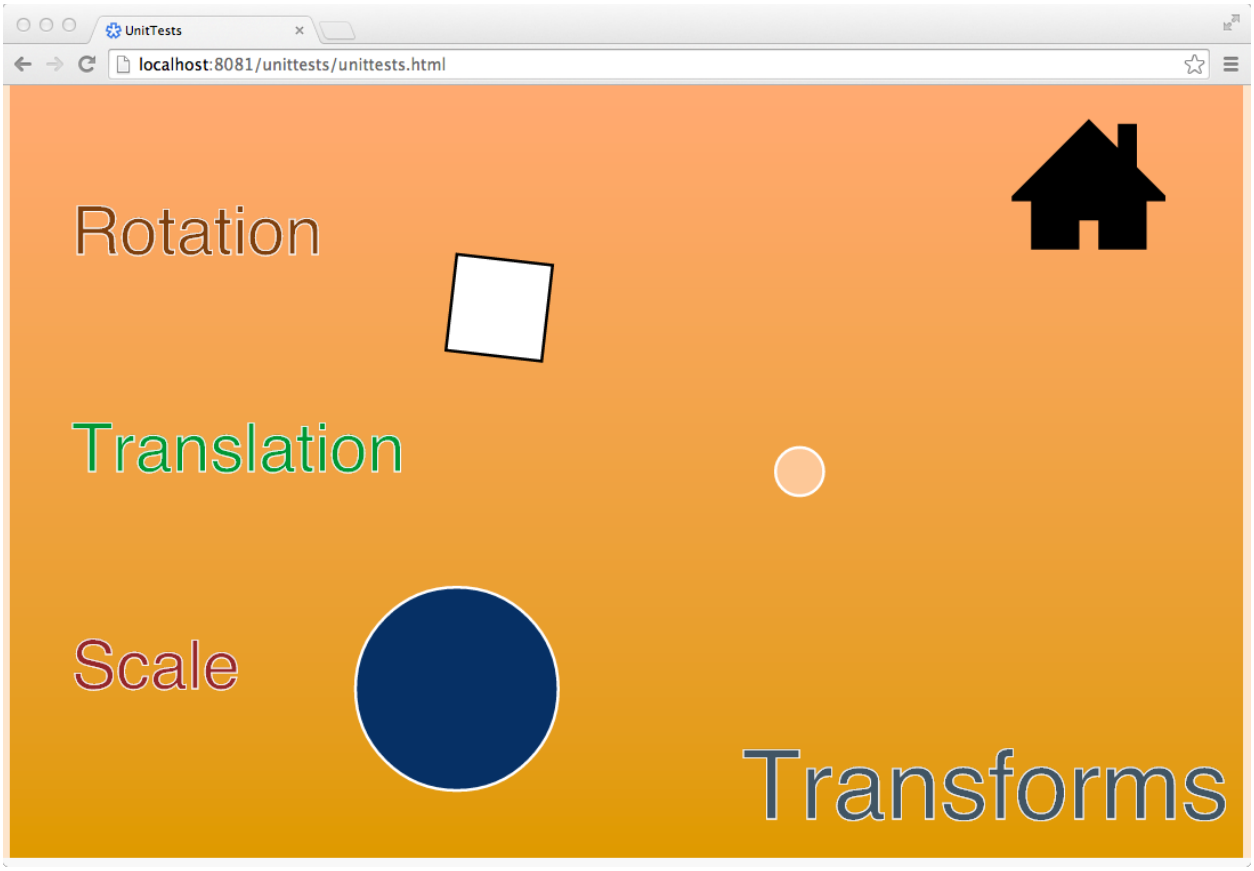


Figure 7.0 Ranger transforms

Above in Figure 7.0 a few Nodes undergo simple transformations.

In this chapter we design and construct a basic first level shell and populate it with a lander (aka Actor). We get our first experience with the scene graph using relative Nodes including animating the landing gear of the lander.

Goal

At this point we have a very basic framework, enough to get us just one click away from an actual game level. Our goal in this chapter is to setup a precursor to an actual level. We need a “test” level (aka Scene and Layer) where we can tryout and learn a few things.

Our level should have the follow things:

- A Lander EngineRocket3.svg
- Input to control the Lander (Thrust, Pitch)
- Landing gear that extends and retracts.
- Simple physics (Gravity, Velocity)
- HUD for displaying: Fuel, Velocity and a Pause button .

Level Up!

Lets call the level **LevelRimbaloidScene** and **LevelRimbaloidLayer** respectively and place them in a new folder under the *game* folder called *levels*. Again, we base the level on a previous Scene and Layer, in this case we copy LevelSelectionScene and Layer respectively.

It also appears that we have two new assets (**EngineRocket3.svg** and **pause.svg**) so lets return to the Resources class and add the respective incrementing and loading code, and then copy the assets from the Book’s assets repository. Once you have the folder and files setup we can begin refactoring.

Begin by stripping down the Layer to include just code for the Lander and Pause button, this will give us a clean starting point. Your *onMouseDown* (Code 7.1) should have code for detecting the Pause button with a comment-placeholder on what to do. Your *_configure* method should have code for creating two Nodes: **_rocket** and **_pause**.

Code 7.1

```
  @override
  bool onMouseDown(MouseEvent event) {
    Ranger.Vector2P nodeP =
ranger.drawContext.mapViewToNode(_pause, event.offset.x,
event.offset.y);

    if (_pause.pointInside(nodeP.v)) {
      nodeP.moveToPool();
      // Pop up a dialog
      return true;
    }
    nodeP.moveToPool();

    return false;
  }
```

Code 7.2

```
  void _configure() {
    double h = contentSize.height / 2.0;
    double w = contentSize.width / 2.0;

    _rocket = new
Ranger.SpriteImage.withElement(gm.resources.engineRocket3);
    _rocket..uniformScale = 0.5
      ..setPosition(0.0, 300.0);
    addChild(_rocket);

    _pause = new
Ranger.SpriteImage.withElement(gm.resources.pause);
    _pause..uniformScale = 0.25
      ..setPosition(w - (w * 0.1), h - (h * 0.15));
    addChild(_pause);
  }
```

Now jump back to LevelSelectionLayer's *onMouseDown* and add code to handle clicking on the first "unlocked" rocket. We will use the **TransitionMoveInFrom** Transition, and this time we will "*replace*" the LevelSelectionScene with a Transition instead (Code 7.3.)

Code 7.3

```
    nodeP = ranger.drawContext.mapViewToNode(_rocket1,
event.offset.x, event.offset.y);

    if (_rocket1.pointInside(nodeP.v)) {
      nodeP.moveToPool();
      LevelRimbaloidScene inComingScene = new
LevelRimbaloidScene();
      Ranger.TransitionMoveInFrom transition = new
Ranger.TransitionMoveInFrom.initWithDurationAndScene(0.5,
inComingScene, Ranger.TransitionSlideIn.FROM_BOTTOM);

      ranger.sceneManager.replaceScene(transition);

      return true;
    }
    nodeP.moveToPool();
```

This allows us to "jump" back to MainScene if they Die.

> **Note**
>
> By default all Transitions use the SceneManager's *replaceScene* on the incoming Scene. This means you either "push" the TransitionScene onto the top of the stack (as was done in MainLayer) or "replace" the top of the stack with the transition Scene.

For now the Pause icon will also return us back to MainScene. Remember we "replaced" the Scene stack top with a TransitionScene which effectively threw away LevelSelectionScene leaving MainScene on the top. So if we want to return to MainScene we simply "pop" the stack in the LevelRimbaloidLayer.*onMouseDown* method:

```
ranger.sceneManager.popScene();
```

If you run Moon Lander now and click your way through the rockets you should have something like Figure 7.4.

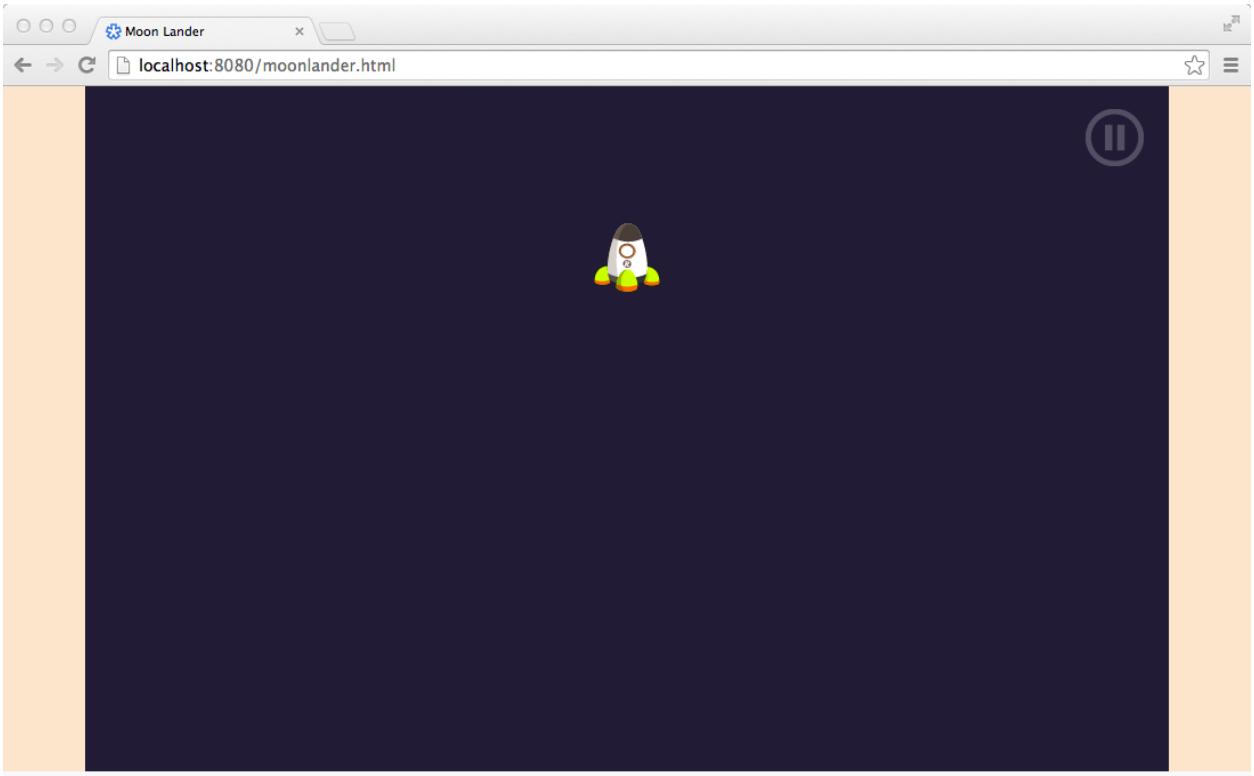


Figure 7.4 LevelRimbaloidScene Pre

Not bad, but…it doesn't quite meet the requirements which is to have a Heads-Up-Display (HUD). Other than aesthetic purposes HUDs allows us to isolate Nodes, for example, the gauges—like Fuel, Velocity, and Pause button. By placing them onto the HUD layer we gain isolation during zooming—which will be added later.

GroupNode

In order to add a HUD we need to introduce the **GroupNode** as the primary layer of our Scene. GroupNodes are stripped down versions of the more common Layer Node. Their main purpose is simply to collect other Nodes, and their only visible aspect is a unit size cross hair that is invisible by default.

They are useful in a large amount of situations, for example, composite type Nodes and as Layer collections or even anchors.

GroupNodes are also just as valid as a primary Node just as Layers are. Layers just happen to have more features mixed in.

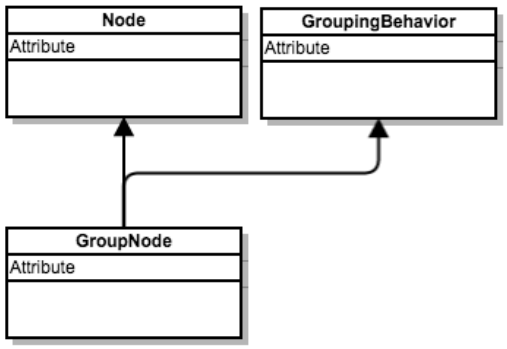


Figure 7.5 GroupNode

HUD

We want the HUD to be an “overlay” on top of the level layer (Figure 7.6.) Recall that the Scene Graph renders from top-to-bottom left-to-right. This means we need to add the level-layer to the GroupNode first and then the HUD layer.

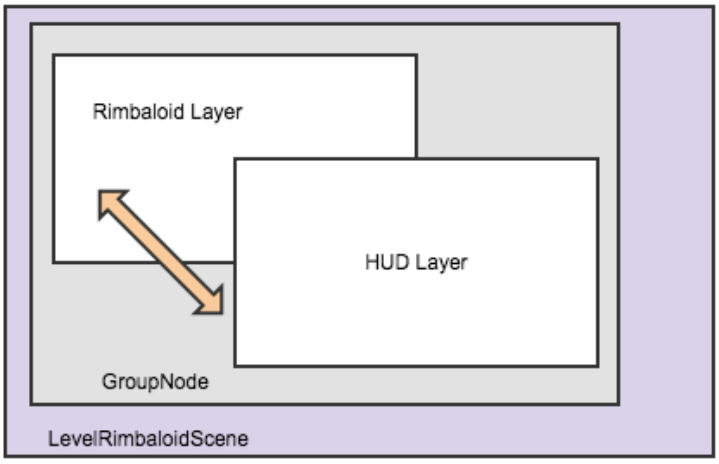


Figure 7.6 Scene layout

In order to do this we need to refactor LevelRimbaloidScene a bit. First we **override** Scene.*init* method and add code that setups the primary Node—instead of it being done in *onEnter* (Code 7.7.)

If you run Moon Lander again you will see that nothing really changed. This is as expected we simply added a GroupNode and “appended” on the original LevelRimbaloidLayer. We need to create a new layer that acts a HUD—lets call it HudRimbaloidLayer. We can then migrate the Pause icon to it.

Code 7.7

```
  @override
  bool init([int width, int height]) {
    if (super.init()) {
      _group = new Ranger.GroupNode();
      initWithPrimary(_group);

      LevelRimbaloidLayer gameLayer = new
LevelRimbaloidLayer.withColor(Ranger.color4IFromHex("#221c35"));
      addLayer(gameLayer);
    }
    return true;
  }
```

HUD layers are typically transparent and rarely translucent. It is pretty simple to create one, just set the ***transparentBackground*** to **true**—that is it! So go ahead and copy just the Layer and rename it to HudRimbaloidLayer. Don't forget to include it using the **part**s statement in *main.dart* as usual.

Remove the following code from the HUD Layer that got copied over:

- Rocket code from layer: That will be on the Rimbaloid layer.
- GroupNode from scene: The HUD is a Layer in and of itself so we don't need to collect other layers on it. The primary will be the layer itself.
- Remove Input from the Rimbaloid layer: this includes the *onMouseDown* method and input enablement.
- Remove the Pause code from the LevelRimbaloidLayer.

Finally we update LevelRimbaloidScene.*init* in order to "append" on the HUD Layer (Code 7.8.)

Code 7.8

```
  @override
  bool init([int width, int height]) {
    if (super.init()) {
      _group = new Ranger.GroupNode();
      initWithPrimary(_group);

      LevelRimbaloidLayer gameLayer = new
LevelRimbaloidLayer.withColor(Ranger.color4IFromHex("#221c35"));
      addLayer(gameLayer);

      HudRimbaloidLayer hud = new HudRimbaloidLayer.withColor();
      addLayer(hud);
    }
    return true;
  }
```

Again, if you run Moon Lander things look exactly the same—visually yes—Scene Graph wise nope.

From a scene graph perspective Figure 7.9 is what we have. Rendering will happen on the Rimbaloid layer first followed by the HUD layer. This causes Nodes on the HUD layer to appear “above” those on the Rimbaloid layer.

All input goes to the HUD layer by design. This could be keyboard and-or mouse/touch for Button icons.

The primary Node is now a GroupNode that contains the two child layers: LevelRimbaloidLayer and HudRimbaloidLayer.

We are now ready to add some input control for the lander itself. We already have the first part done and that was enabling the input on the HUD layer, now we need to construct a “useful” lander such that we can feed control information to it. A simple SpriteImage just doesn’t cut it. I think we need an **Actor** framework.

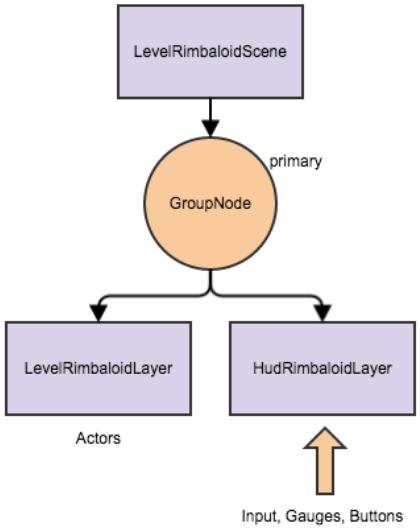


Figure 7.9 Rimbaloid Scene

Mobile Actor Class

Our Actor class (named **MobileActor**) will establish a base for all objects in the game that are mobile; asteroids and ships. Its main components will be Velocity and Momentum, and its secondary components are properties for controlling Thrust and Pitch.

With MobileActor we can construct a child class called **TriEngineRocket**. This child class defines the physics properties for a specific lander. Some landers are bigger and heavy while others are small and nimble. The properties effect things like turning rate and fuel consumption per unit of time. The class also contains any Nodes that represent the visual aspect of the lander.

These classes aren’t really relevant to Ranger so they have already been coded we just need to copy and bind controls to them, and as usual you can copy the classes from the book’s repository, but first create a new folder under *game* called *actors*, then copy *mobile_actor.dart* and *tri_engine_rocket.dart*—don’t forget to include them in *main.dart*.

> Note
>
> This book isn’t trying to promote any type of purist coding style. Its goal is to teach Ranger and as such the design and coding is focused on conveying Ranger and not so much as perfect coding nor any particular algorithms. As you design your own game I am sure you will follow standards you already follow today.

Keyboard Input

For this version of Moon Lander control input will come from the keyboard. Three keys are needed:

- A key: Pitch counter clockwise, tilt to the left
- S key: Pitch clockwise, tilt to the right
- / key: Thrust
- E key: eventually used for landing gear control.

Enabling keyboard is as simple as was enabling the mouse. Add the line to HudRimbaloidLayer.*onEnter* method:

```
enableKeyboard = true;
```

All that is left is to **override** the appropriate keyboard mix-in methods. Code 7.10 shows just the pre-setup for key-down.

Code 7.10

```
@override
bool onKeyDown(KeyboardEvent event) {
  switch (event.keyCode) {
    case 65:
      // "A" key. Pitch CCW
      return true;
    case 83:
      // "S" key. Pitch CW
      return true;
    case 191:
      // "/" key. Thrust
      return true;
  }

  return false;
}
```

For controlling the lander we will also need the key-up event as well. This allows the player to hold a key, Thrust being an example of this requirement.

Hud continued

Before we can bind the input to our Actor we need access to the Actor. Lets add a factory to the **GameManager** (GM) class that constructs and initializes (*_init*) the GM. In there we will construct the TriEngineRocket (Code 7.11.)

Code 7.11

```
factory GameManager() {
  GameManager gm = new GameManager();
  if (gm._init()) {
    return gm;
  }

  return null;
}

bool _init() {
  triEngineRocket = new TriEngineRocket();

  return true;
}
```

Now we can route control to the lander. Return to the keyboard event overrides in HudRimbaloidLayer and add calls to the lander's control methods; there are three: pitchLeft, pitchRight and Thrust. Here is the "**A**" key-down pitch left—of course there is a matching key-up that passes **false**:

```
      case 65:
        // "A" key. Pitch CCW
        gm.triEngineRocket.pitchLeft(true);
        return true;
```

Next lets migrate the SpriteImage into Actor. Keep in mind we are only moving the creation and storage of the sprite, the sprite Node itself will still be a child of the LevelRimbaloidLayer, but the Actor will manipulate the properties of the Node. How about we go ahead and code the Actor to adjust the lander's Pitch (i.e. rotation.)

Node Rotation

Nodes have three common transformations supplied by behaviors/mixins:

- Scale: **ScaleBehavior**
- Rotation: **RotationBehavior**
- Translation: **PositionalBehavior**

If you want to transform any Node just call one of the behavior's methods. Any of these methods will result in the Node's internal affine transform matrix being modified. For example, we want to change the lander's Pitch. This requires a property change namely the sprite's Rotation property.

In order to detect the input-control values we need to track and pole them. To poll these values we need to **override** the HudRimbaloidLayer.*update* method and schedule the layer for updates. We can then propagate the updates to the Actor.

Because all Ranger Nodes are **TimingTarget**s we can schedule the Node with the **ScheduleManager** (SM). By doing so our Node will receive updates along with a time delta

relative to the previous frame. But we must also remember the flip side signal—*onExit*—as well. If a Scene exits the “stage” we also want the Layer to stop receiving updates. The SM does not look at a Node to see if a Scene it is associated with has exited the stage so our Nodes must be proactive.

The easiest way to do this is to use the Node’s *scheduleUpdate* method during the Layer’s *onEnter* method, (Code 7.12), *unscheduleUpdate* method during the layer’s *onExit*.

Code 7.12

```
@override
void onEnter() {
  enableMouse = true;
  enableKeyboard = true;
  super.onEnter();

  scheduleUpdate();
}
```

Now that our Node is scheduled for timing events we need to override BaseNode’s *update* method, (Code 7.13), and propagate the event to the Actor.

Code 7.13

```
@override
void update(double dt) {
  gm.triEngineRocket.update();
}
```

And we can complete the key-down and key-up events respectively. Code 7.14 shows the key-down event bindings only.

Beware

By default Ranger is using the window’s *animationFrame* method which is non-mutable at ~60 frames per second. Any type of polling will not happen any faster than that.

Code 7.14

```
@override
bool onKeyDown(KeyboardEvent event) {
  switch (event.keyCode) {
    case 65:
      // "A" key. Pitch CCW
      gm.triEngineRocket.pitchLeft(true);
      return true;
    case 83:
      // "S" key. Pitch CW
      gm.triEngineRocket.pitchRight(true);
      return true;
    case 191:
      // "/" key. Thrust
      gm.triEngineRocket.thrust(true);
      return true;
  }

  return false;
}
```

Run Moon Lander now and navigate through the rockets. Once you are at the Rimbaloid scene you can fiddle around with the “A” and “S” keys and see that the sprite rotates. This happens because TriEngineRocket’s *update* method is being called which in turn updates the sprite’s Rotational property (*rotationByDegrees*) based on a rotation rate (Code 7.15.) Nice!

Code 7.15

```
void update() {
  if (_pitchingLeft && !_pitchingRight) {
    _rocket.rotationByDegrees = _rocket.rotationInDegrees + pitchRate;
  }
  else if (!_pitchingLeft && _pitchingRight) {
    _rocket.rotationByDegrees = _rocket.rotationInDegrees - pitchRate;
  }
}
```

The only problem is that the lander isn’t descending, it is stuck in the “sky”. Introducing some simply physics should fix that.

Velocity vectors

As mentioned earlier Moon Lander won’t use a complex physic engine like Box2D. We will use a plan old approach, Ranger Velocity Vectors. Gravity in our game will be a constant force applied to the lander. The force is chosen such that it points in the -Y direction with a certain magnitude.

Go ahead and add a Velocity object called **gravity** to the GameManager class and update the *postLoad* method to create it with a downward force (Code 7.16.)

Code 7.16

```
  static const double ACCELERATION = 0.02;
  Ranger.Velocity gravity;

  bool postLoad() {
    triEngineRocket = new TriEngineRocket();
    triEngineRocket.init();

    gravity = new Ranger.Velocity.withComponents(0.0, -1.0, ACCELERATION);

    return true;
  }
```

Return to TriEngineRocket's *update* method and add some code that modifies the lander's position. Remember this is physics so we don't apply a force directly to the lander but instead apply the force to a momentum Vector. This momentum is then applied to the rocket (Code 7.17.)

Code 7.16

```
    _momentum.add(_thrust);
    _momentum.add(gm.gravity);

    v.setFrom(_rocket.position);
    _momentum.applyTo(v);
    _rocket.position = v;
```

Note that I copy the lander's position into a variable, modify it and then copy it back. I didn't do something like: _momentum.applyTo(_rocket.position);. The reason for this is because modifying a Node's **position** directly bypasses the Node's "**dirty**" flag. The dirty flag is an optimization property whereby the affine transformation matrix is only computed if a transform property is changed. This flag is set correctly if you use the Node's getter/setters. If you decide to manually update them yourself then you must remember to set the flag, otherwise the matrix becomes stale and nothing happens.

That takes care of gravity. Now we work on the opposing force; Thrust.

Thrust

Thrust should match the visual direction of the lander. Our lander was designed in InkScape with the nose of the lander pointing in the +Y direction. Ranger's Velocity vectors default to "pointing" in the +X direction meaning there is a 90 degree rotational offset between systems—of course the lander could have been pre-rotated in InkScape to align their direction vectors. What this means is that we need to add 90 degrees to the lander's direction when configuring the thrust vector.

We also need to adjust the Thrust magnitude based on the lander's mass to simulate pseudo inertia. Keep in mind this is game physics and as such nothing is to scale, whatever appears to work is the values we use. Modify the *update* method, (Code 7.18), and *init* method, (Code 7.17), to include Thrust code.

Code 7.17

```
    _thrust.maxMagnitude = THRUST_MAX;
    // Default direction with the vertical
    _thrust.directionByDegrees = VISUAL_COORD_SYSTEM;
```

Code 7.18

```
void update() {
  if (_pitchingLeft && !_pitchingRight) {
    …
    // The sprite SVG is visually designed as if 90 degrees rotated
    // from the +X axis.
    _thrust.directionByDegrees = _rocket.rotationInDegrees + VISUAL_COORD_SYSTEM;
  }
  else if (!_pitchingLeft && _pitchingRight) {
    …
    _thrust.directionByDegrees = _rocket.rotationInDegrees + VISUAL_COORD_SYSTEM;
  }

  if (_thrustOn) {
    _thrust.speed = THRUST_POWER / mass;
  }

  // Apply Thrust and Gravity
  _momentum.add(_thrust);
  _momentum.add(gm.gravity);

  v.setFrom(_rocket.position);
  _momentum.applyTo(v);
  _rocket.position = v;

  // If there is no thrust being applied then the thrust slowly dies off.
  if (_thrust.speed > 0.0)
    _thrust.decreaseSpeed(THRUST_DAMPING / mass);
}
```

Now run Moon Lander. Slowly the lander should begin to fall at an every increasing descent. If you hold the "/" key down the lander's descent will slow until gravity is overcome by the thrust and the lander begins to ascend. Yeah! Moon Lander is starting to feel like a game with fun written all over it! Lets bolt on a set of landing gear next.

Landing Gear

Adding landing gear is going to require a little bit of refactoring in terms of what actually defines the lander. At the moment we think of the lander as a sprite contained with in a class. Luckily we thought ahead and didn't expose the sprite directly but instead exposed a property called "*node*". This means we can reorganize TriEngineRocket internally to place a GroupNode as the main Node for defining the lander.

Add a GroupNode called ***_centroid*** to TriEngineRocket and create it in the *init* method. With the GroupNode in hand add the ***_rocket*** sprite as a child, then change the *node* property to return ***_centroid*** instead of ***_rocket***, and change the return type to Node instead of SpriteImage.

Now replace all the code that was manipulating ***_rocket*** with ***node***. If you run Moon Lander now the lander acts exactly the same—good. Now we can add landing gear; admittedly the

landing gear will be pretty simple, just a stick for a **Leg** and a thin square for a **Toe**. To pull this off we are going to create our first custom geometry Nodes.

Custom Nodes

Up to this point we have been using SVG sprites. However, for the landing gear we are going to use simple geometry, a thin rectangle for the leg and toe with a bit of animation thrown in for good measure.

Creating a custom Node is pretty simple, just inherit from **Node** and **override** Node's *draw* method—of course there are many other methods you can override but the most important, at least if you want to "see" your Node, is the *draw* method.

Of course to draw something we need to know something about the coordinate system in which we are drawing. When you override the *draw* method you are effectively in the local-space of the Node, and everything you draw is drawn relative to the local origin (Figure 7.19.)

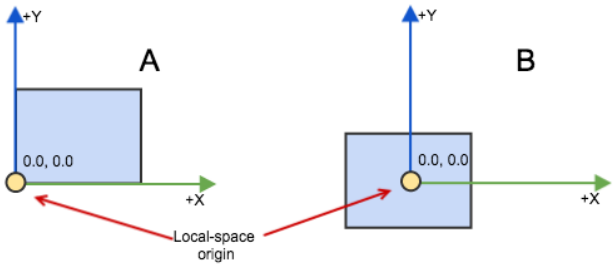


Figure 7.19 Local-space

In addition, all transforms act on the local-space of a Node which can result in unexpected outcomes if you don't understand how transforms work. It is beyond the scope of this book to cover transforms but there are an almost uncountable number of resources (i.e. books, website etc.) that cover transformations relative to a coordinate system. Having said that it is important how you define your geometry relative to the local origin. The more thought you put into the geometry layout the less transforms you need to apply.

In Figure 7.19 we see two local-spaces. On the left (A) a rectangle is defined with the lower-left corner at the origin (aka centroid), and on the right (B) is a rectangle with the centroid at the area mid point. Applying a rotation transformation to each one will result in something very different to each one. The left rectangle will "swing" around the origin while the right rectangle will rotate in place about the origin. Thus, being aware of how transformations effect your Node can guide you in properly laying out your geometry. Note, both the leg and toe geometry will be laid out according to (B).

Because these are custom Nodes we are responsible for drawing the rectangles using the **DrawContext** passed to our *draw* method. So lets create our very first custom Node called **RectangleNode**. Create a new folder called *nodes* under the *game* folder. In there create a new file called *rectangle_node.dart*, and remember to include the file using the **part** statement.

First we add properties to contain the fill and outline colors (Code 7.20.)

Code 7.20

```
  String outlineColor;
  String fillColor;

  static const double outlineThickness = 3.0;
```

Next we add a factory to properly construct the Node and set the properties (Code 7.21.)

Code 7.21

```
  factory RectangleNode.basic(Ranger.Color4<int> fillColor, Ranger.Color4<int>
outlineColor) {
    RectangleNode o = new RectangleNode();
    if (o.init()) {
      o.fillColor = fillColor.toString();
      o.outlineColor = outlineColor.toString();
      return o;
    }

    return null;
  }
```

Finally we override the *draw* method and add our very first drawing code. Our rectangle is a basic unit sized square centered around the origin (Code 7.22.)

Code 7.22

```
  @override
  void draw(Ranger.DrawContext context) {
    context.save();

    CanvasRenderingContext2D context2D = context.renderContext as
CanvasRenderingContext2D;

    if (fillColor != null) {
      context2D..fillStyle = fillColor
        ..fillRect(-0.5, -0.5, 1.0, 1.0);
    }

    // Note: strokes can scale to the point that the fill can't be seen.
    // An inverse scale is needed if the Node is scaled.
    if (outlineColor != null) {
      context2D..strokeStyle = outlineColor
        ..lineWidth = outlineThickness / calcUniformScaleComponent()
        ..strokeRect(-0.5, -0.5, 1.0, 1.0);
    }

    context.restore();
  }
```

Note that we also inverse scale the lineWidth. Because Ranger is a scene graph rendering properties propagate downward. If a Node's parent has a scale value then any lines drawn at

and below the parent are also scaled. For our Node we don't want that so we negate it by computing the scale "upwards" and dividing that into whatever line width we want. You can optimize this by overriding a "dirty" method that indicates that the current Node's state has changed where upon you would then compute the lineWidth rather than compute it on every draw.

We just created our very first custom Node! Lets prepare to use it.

Having a better understanding of local-spaces lets us think about how our landing gear is going to operate. When the gear is retracted it is hidden behind the lander's body and invisible. You may wonder why we want to hide the gear even though it is behind the lander. The reason is optimization, when a node isn't visually visible it is still visited and drawn—even if it is behind another Node. The less work the scene graph has to do the more processing time your code gets. Thus setting a Node invisible literally tells the scene graph to ignore the Node.

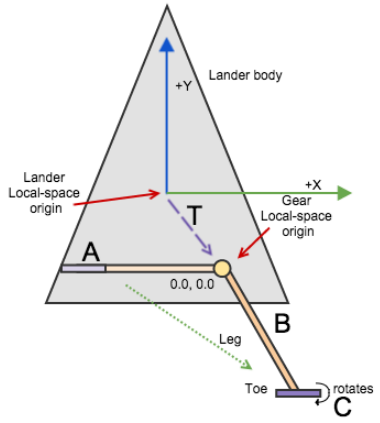


Figure 7.23 Right Landing gear

Lets look at the extend and retract sequences to get a better sense of direction (reference Figure 7.23.)

The extend sequence for the gear is:

- Make gear visible
- Rotate leg N degrees from Position A to B
- Rotate toe N degrees (C)

The retract sequence is:

- Rotate toe -N degrees (C)
- Rotate leg -N degrees from Position B to A
- Make gear invisible

Go ahead and create a class called **TriEngineGear** in the *actors* folder. Inside the class create a GroupNode called **Joint** to act as a joint connection to the lander's body and as a parent to the leg and toe.

We are literally going to create geometry that matches almost exactly the rectangles in Figure 7.23. In order to do that we need to apply transforms to them. Again, understanding how transforms effect Nodes in a hierarchy is key to knowing which Node arrangement requires the least amount of effort.

In figure 7.24 we see two Node arrangements. On the left, (A), the Toe is a child of Leg and on the right, (B), the Toe is a sibling of Leg. Because transforms propagate downward in the graph the Toe will be affected differently depending on which arrangement is used. For example, when a Scale transformation is applied to the Leg in (A) the Toe will also be scaled as well. If we want the Toe to remain the same size then we need to apply an inverse scale based on its parent (aka Leg)—much like we did with the RectangleNode's lineWidth property.

However, if we use the arrangement in (B) then any transformation applied to the Leg will not effect the Toe. For our landing gear we are going to use arrangement (B) with the Toe added last so that it appears "above" the Leg.

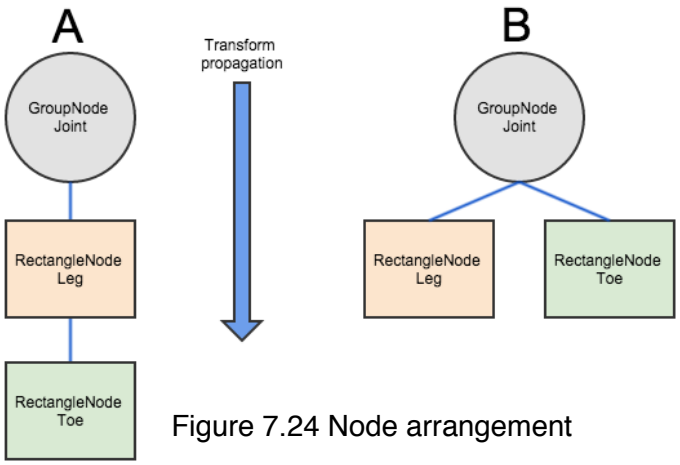


Figure 7.24 Node arrangement

Landing Gear continued

Lets work on the right lander gear first. Using arrangement (B) we code and configure the TriEngineGear's *init* method as shown in (Code 7.25.)

Code 7.25

```
bool init() {
  _joint = new Ranger.GroupNode();

  double sx = 110.0;
  double xOff = sx / 2.0;
  _leg = new RectangleNode.basic(Ranger.Color4IOrange);
  _leg.scaleTo(sx, 10.0);
  _leg.positionX = xOff;
  _joint.addChild(_leg);

  _toe = new RectangleNode.basic(Ranger.Color4IGreen);
  double tsx = 30.0;
  xOff = sx;
  _toe.scaleTo(tsx, 10.0);
  _toe.positionX = xOff;
  _toe.rotationByDegrees = 90.0;
  _joint.addChild(_toe);

  return true;
}
```

This coding results in a gear that is “pointing” to the right. What we are striving for is a gear pointing inward such that when we animate the gear the leg drops downward from inside and swings outward to lock into place. For that to happen we need to add a 180 degree rotation to the joint just after its creation: [_joint.rotationByDegrees = 180.0;].

Return to TriEngineRocket class and add TriEngineGear property called ***_rightGear***. Then create an instance of it inside *init* and add it as a child to ***_centroid*** “*before*” the _rocket is added. This insures that the gear is rendered first thus placing them underneath the lander—of course we could have changed the Z-order but in this case we didn’t.

According to Figure 7.23 we need to translate the joint down and to the right. Code 7.26 shows this using the *setPosition* method.

Code 7.26

```
  _centroid.addChild(_rocket);

  _rightGear = new TriEngineGear.basic();
  _rightGear.node..setPosition(50.0, -50.0)
    ..uniformScale = 1.25;
  _centroid.addChild(_rightGear.node);
```

However, this design isn’t flexible enough. The class can’t be used for both the left *and* right gears. Lets make a slight adjustment to TriEngineGear to allow that. Add two methods for setting the position and rotation (Code 7.27.)

Code 7.27

```
  void setRotation(double r) {
   _joint.rotationByDegrees = _initialOrientation;
  }

  void setPosition(double x, double y) {
    _joint.setPosition(x, y);
  }
```

Now we can update TriEngineRocket.*init* to create both left and right gears (Code 7.28.)

Code 7.28

```
    _rightGear = new TriEngineGear.basic();
    _rightGear.setRotation(180.0);
    _rightGear.setPosition(40.0, -50.0);
    _rightGear.node.uniformScale = 1.25;
    _centroid.addChild(_rightGear.node);

    _leftGear = new TriEngineGear.basic();
    _leftGear.setRotation(0.0);
    _leftGear.setPosition(-40.0, -50.0);
    _leftGear.node.uniformScale = 1.25;
    _centroid.addChild(_leftGear.node);
```

Extend and Retract

Now that we have two sets of landing gear we can animate them. By default the code has the landing gear in the retracted position, (see Figure 7.23 (A)), thus the first animation we need to create is the Extend animation.

As part of animating the gears we are going to create our first custom Tween animation. How? By extending TriEngineGear and turning it into a **Tweenable** (see Chapter 5 “Asset Loading”, section Animations.)

First extend TriEngineRocket to inherit from Tweenable:

```
class TriEngineGear extends UTE.Tweenable {
```

Because we extended the Tweenable interface we are now obligated to implement Tweenable’s interface which consists of two methods:

```
int getTweenableValues(Tween tween, int tweenType, List<num> returnValues);
void setTweenableValues(Tween tween, int tweenType, List<num> newValues);
```

It is the tweenType that controls what is being animated.

tweenType

These getters/setters methods are called during the animation. It's the responsibility of the developer to provide the appropriate data in both directions for the respective ***tweenType***. There will be two tween types one for each Node we are going to animate.

Note

tweenType can mean just about anything, it is a constant you define that TweenEngine will pass to you for tweening information. For example, you could have one tweenType associated with two Nodes doing two different things like Rotating and Scaling.

For our landing gear we add one tweenType for each Node: ***_joint*** and ***_toe***, both designed to respond to rotational information from the TweenEngine:

```
class TriEngineGear extends UTE.Tweenable {
  static const int JOINT_ROTATE = 1;
  static const int TOE_ROTATE = 2;
```

While we are at it, we also need two angles for each Node in order to remember the Extended and Retracted orientations:

```
  double legExtendOrientation = 0.0;
  double legRetractOrientation = 0.0;
  double toeExtendOrientation = 0.0;
  double toeRetractOrientation = 0.0;
```

The tweenType is the key used by the Tweenable implementations. Whenever TweenEngine needs a value about a Node (that it is animating) it calls back to the Node asking for a value using one of the "key"s you defined. Our implementations determine which value to give or receive. (Code 7.29) shows how the tweenTypes are used to return the correct value.

Code 7.29

```
  int getTweenableValues(UTE.Tween tween, int tweenType, List<num> returnValues) {
    switch(tweenType) {
      case JOINT_ROTATE:
        returnValues[0] = node.rotationInDegrees;
        return 1;
      case TOE_ROTATE:
        returnValues[0] = _toe.rotationInDegrees;
        return 1;
    }

    return 0;
  }
```

Code 7.30 shows how the tweenTypes are used to feed the tween values into the Nodes.

Code 7.30

```
void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) {
  switch(tweenType) {
    case JOINT_ROTATE:
      node.rotationByDegrees = newValues[0];
      break;
    case TOE_ROTATE:
      _toe.rotationByDegrees = newValues[0];
      break;
  }
}
```

Now we can fill out the *retract* method. First we “kill/stop” any animations in progress for “this” Tweenable:

```
    ranger.animations.tweenMan.killTarget(this, JOINT_ROTATE);
    ranger.animations.tweenMan.killTarget(this, TOE_ROTATE);
```

For the Retraction animation we want both the Leg and Toe to animate at the same time (i.e. in parallel.) Therefore we need to create a parallel **TimeLine** and add two **Tween**s to it (Code 7.31.)

Code 7.31

```
    UTE.Timeline par = new UTE.Timeline.parallel();

    UTE.Tween rotateTo = new UTE.Tween.to(this, JOINT_ROTATE, 2.0)
      ..targetValues = [legRetractOrientation]
      ..callback = _retractComplete
      ..callbackTriggers = UTE.TweenCallback.COMPLETE
      ..userData = node
      ..easing = UTE.Linear.INOUT;
    par.push(rotateTo);

    UTE.Tween rotateToeTo = new UTE.Tween.to(this, TOE_ROTATE, 1.0)
      ..targetValues = [toeRetractOrientation]
      ..easing = UTE.Bounce.OUT;
    par.push(rotateToeTo);
```

Notice three things being set on the **rotateTo** Tween:

1. A callback is being set. The method called during any of the filters.
2. A trigger filter. We want the event that indicates when the animation is COMPLETE.
3. The **userData** property is set. Setting user data allows you to “capture” and associated data with a callback event.

This allows us to “watch” for a certain events associated with an animation. Once we detect that event we do something, and in this case we simply make the joint invisible (Code 7.32.)

Code 7.32

```
void _retractComplete(int type, UTE.BaseTween source) {
  Ranger.GroupNode joint = source.userData as Ranger.GroupNode;
  joint.visible = false;
}
```

Notice in the callback, (Code 7.32), that I am not actually detecting which event occurred. This is because I know that COMPLETE is only called once at the very end of a simple animation sequence, otherwise I would add a **switch** statement to detect the type.

The Extension animation is just a little bit more complex in that we have both a Parallel and Sequential animation in play. Like the retraction we first kill any animations:

```
    ranger.animations.tweenMan.killTarget(this, JOINT_ROTATE);
    ranger.animations.tweenMan.killTarget(this, TOE_ROTATE);
```

We then make the joint visible:

```
node.visible = true;
```

We want the Toe animation to delay for 0.5 seconds before its rotation begins, this gives the first animation enough time to swing the Legs into view. This requirement needs a sequential animation embedded inside a parallel animation (see Figure 7.33.)

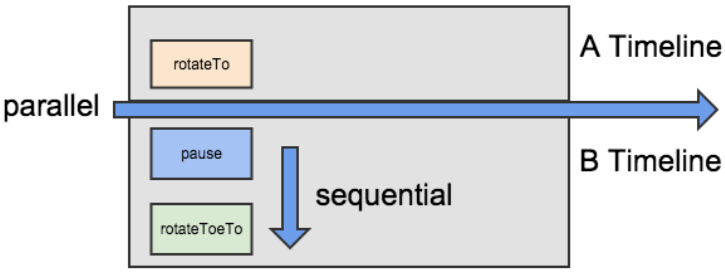


Figure 7.33 Embedded animations

Timeline A and B will both animate at the same time, but B is actually a sequential animation part of which is a pause type Tween. The code for this is show in (Code 7.34.)

Code 7.34

```
UTE.Timeline par = new UTE.Timeline.parallel();

UTE.Tween rotateTo = new UTE.Tween.to(this, JOINT_ROTATE, 2.0)
  ..targetValues = [legExtendOrientation]
  ..easing = UTE.Expo.OUT;
par.push(rotateTo);

UTE.Timeline seq = new UTE.Timeline.sequence();

UTE.Tween rotateToeTo = new UTE.Tween.to(this, TOE_ROTATE, 1.0)
  ..targetValues = [toeExtendOrientation]
  ..easing = UTE.Bounce.OUT;
seq.pushPause(0.5);
seq.push(rotateToeTo);
par.push(seq);
```

I wanted the Legs to “pop” out so I chose an Exponential-Out easing. I also added a bit-of-flare to the Toes by using the Bounce-Out easing.

Now we go back and retrofit (Code 7.28) to include the gear configurations (Code 7.35.) The values used are ones that I guessed at by repeatedly running the game and experimenting with values until I got what I wanted.

Code 7.35

```
_rightGear = new TriEngineGear.basic();
_rightGear..setRotation(180.0)
  ..legRetractOrientation = 180.0
  ..legExtendOrientation = _rightGear.legRetractOrientation + 115.0
  ..toeRetractOrientation = 0.0
  ..toeExtendOrientation = 65.0
  ..setPosition(40.0, -50.0)
  ..node.uniformScale = 1.25;
_centroid.addChild(_rightGear.node);

_leftGear = new TriEngineGear.basic();
_leftGear..legRetractOrientation = 0.0
  ..legExtendOrientation = -115.0
  ..toeRetractOrientation = 0.0
  ..toeExtendOrientation = -65.0
  ..setRotation(0.0)
  ..setPosition(-40.0, -50.0)
  ..node.uniformScale = 1.25;
_centroid.addChild(_leftGear.node);
```

Nice. Now we need to add some visual gauges to give us feedback on piloting.

Gauges

It's time to add the Fuel and Velocity gauges to the HUD layer. The Velocity data is already present in TriEngineRocket as ***_momentum***. For the Fuel gauge we need to add a new property that tracks fuel when the thrust is turned on.

Go ahead and add a ***fuel*** property to TriEngineRocket with an arbitrary default value of 200.0. Whenever the thrust is turned on a burn-rate is subtracted from the fuel. All the gauges, including fuel, will show their values based on percentages.

Each gauge's visual will be a horizontal outlined rectangle located near the top of the view. The Fuel gauge is a one-sided gauge that moves from 100% to 0%. Whereas Velocity is a vertical two-sided gauge that moves from -Max to +Max centered around zero.

How about we add the Fuel gauge first by creating a new class called **FuelGauge** in the *actors* folder. The gauge functions by scaling a rectangle along the X axis while another rectangle marks the boundaries. This means we need to create a GroupNode to keep them locked together:

```
class FuelGauge {
  Ranger.GroupNode _base;
  NonUniformRectangleNode _solid;
  NonUniformRectangleNode _outline;
  Ranger.TextNode _label;
```

The above code is using a new NonUniform Node. This Node is almost identical to RectangleNode except that the dimensions are specified relative to design resolution rather than scaling a unit size rectangle. Scaling a unit size rectangle can start to show problems during stroke drawing, the lines become heavily aliased. At times it is better to fill and stroke at the actual size rather than scaling unit dimensions.

Figure 7.36 shows the scene graph layout for the FuelGauge. The solid rectangle is scaled according to a simple dimensional analysis.

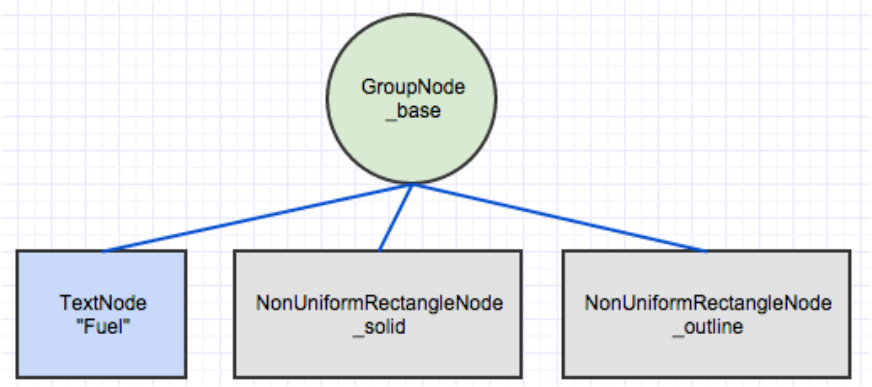


Figure 7.36 FuelGauge

The two main methods of FuelGauge do the math accordingly as well as colorize the bar based on percentages (Code 7.37.)

Code 7.37

```
  set value(double d) {
    // Scale solid bar
    _solid.width = _ratio * d;

    double p = 1.0 - (_max - d) / _max;

    if (p < 0.25)
      _solid.fillColor = Ranger.Color4IRed.toString();
    else if (p < 0.5)
      _solid.fillColor = Ranger.Color4IGoldYellow.toString();
  }

  set max(double d) {
    _max = d;
    _ratio = _width / _max;
  }
```

We can now go back to the HudRimbaloidLayer and add a new FuelGauge property then create and configure it such that it is located in the upper-left corner:

```
    _fuel = new FuelGauge.basic()
      ..node.setPosition(-w + (w * 0.15), h - (h * 0.1))
      ..max = TriEngineRocket.MAX_FUEL
      ..value = TriEngineRocket.MAX_FUEL;
    addChild(_fuel.node);
```

Once configured we can “update” the fuel gauge by setting the FuelGauge.***value*** property in the *update* method:

```
_fuel.value = gm.triEngineRocket.fuel;
```

Finally we need to actually update the fuel property based on a burn rate when thrust is being applied, we do this inside the TriEngineRocket.*update* method:

```
    if (_thrustOn) {
      _thrust.speed = THRUST_POWER / MASS;
      fuel -= BURN_RATE;
      fuel = fuel.clamp(0.0, MAX_FUEL);
    }
```

That takes care of the Fuel gauge. Now lets work on the Velocity gauge.

The Velocity gauge is actually two speed gauges; one vertical and the other horizontal. They each have a center point where the bar can rise above or drop below. The gauge is very similar to the Fuel gauge, enough so that we can simply copy the FuelGauge class and tweak it.

This has already been done and for now just copy VelocityGauge from the book asset repository to save some typing.

All we need to do now is add two VelocityGauge properties to the HudRimbaloidLayer and add create and configure them in the *_configure* method (Code 7.38.)

Code 7.38

```
    Ranger.GroupNode velocityGrp = new Ranger.GroupNode();
    velocityGrp.positionX = 10.0;
    _verticalVelocity = new VelocityGauge.basic()
      ..label = ""
      ..node.setPosition(-w + (w * 0.20), h - (h * 0.4))
      ..node.rotationByDegrees = 90.0
      ..max = 10.0
      ..value = 0.0;
    velocityGrp.addChild(_verticalVelocity.node);

    _horizontalVelocity = new VelocityGauge.basic()
      ..TextLabel.setPosition(-170.0, 35.0)
      ..node.setPosition(-w + (w * 0.20), h - (h * 0.4))
      ..max = 10.0
      ..value = 0.0;
    velocityGrp.addChild(_horizontalVelocity.node);
    addChild(velocityGrp);
```

Then add two lines in the *update* method for updating the gauge's values:

```
    _verticalVelocity.value = gm.triEngineRocket.verticalVelocity;
    _horizontalVelocity.value = gm.triEngineRocket.horizontalVelocity;
```

Recap

In this chapter we learned a little bit more about scene graph node arrangements. Some arrangements are more appropriate than others. We also got our first look at local-space during the rendering of a Node and how arrangements can effect the rendered outcome.

We learned how to transform Nodes using rotation and scaling in such a way as to predictably know how the Node would be effected.

We created our first “transparent” overlay layer in order to create a HUD like display.

We also learned how to work with the keyboard and how to control a Node through polling.

And finally we added a bit on non related Ranger code to handle physics both for Thrust and Gravity.

Launching Moon Lander now we can see our gauges in action as we pitch and thrust (Figure 7.39.)

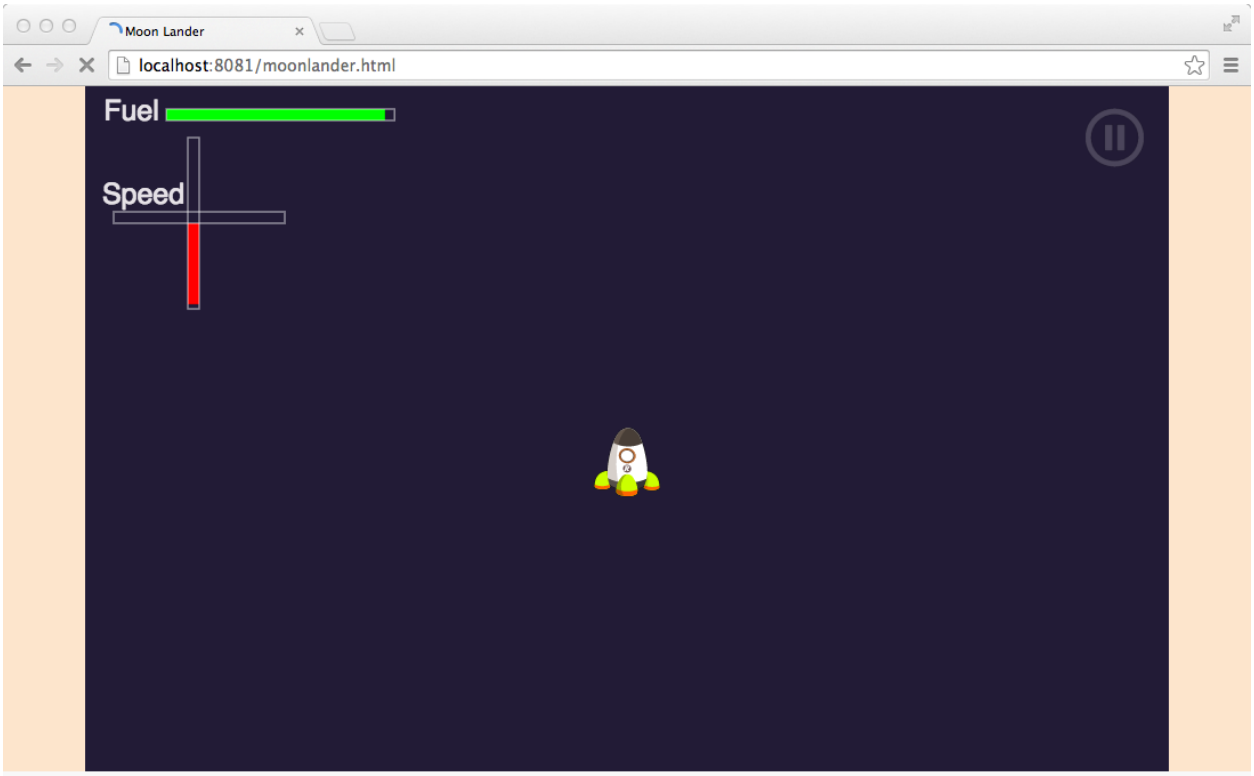


Figure 7.39 Gauges in action

Sweet! However, at the moment we can’t really tell that the thrusters are really engaged other than how the momentum of the lander changes. What we need is a visual clue, something like an exhaust of sorts. How about a particle system!

Chapter 8 Particle Systems

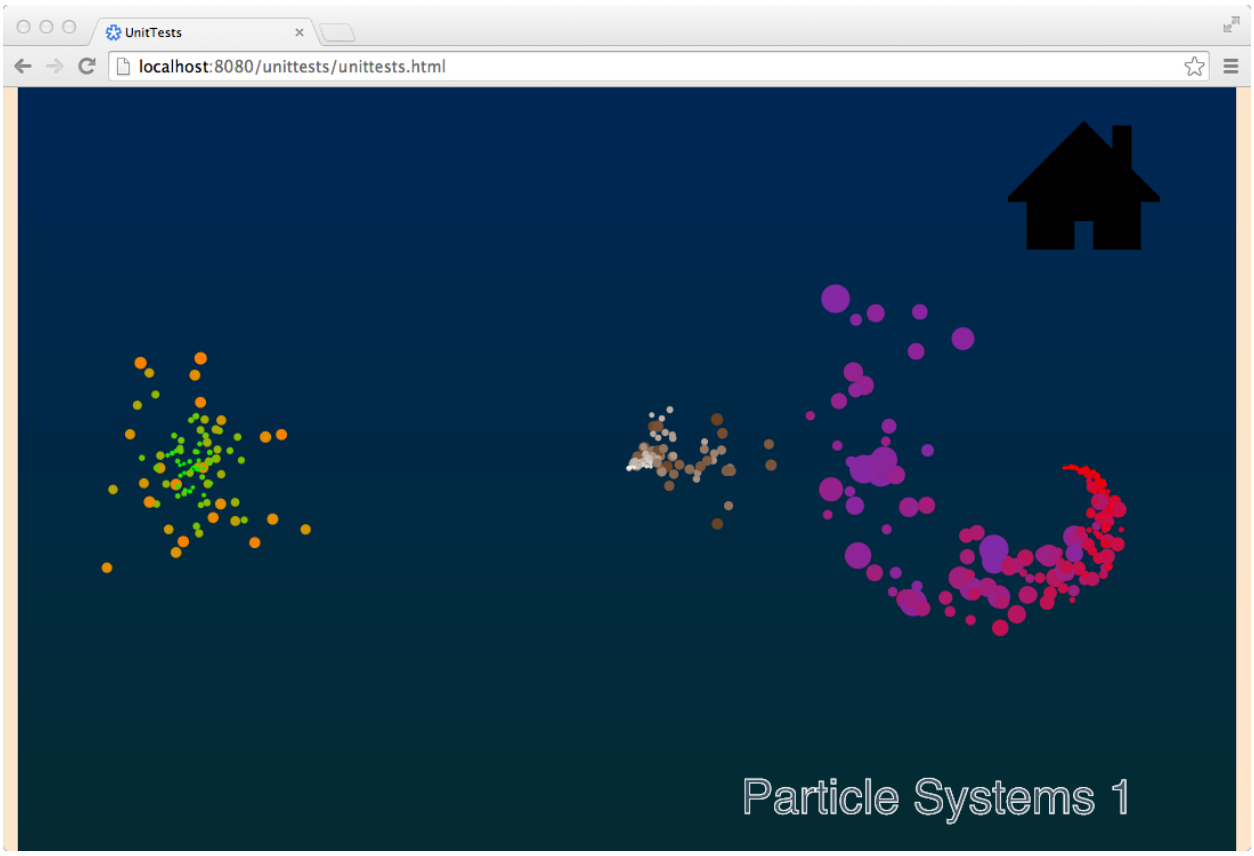


Figure 8.0 Swirling particle systems

Above in Figure 8.0 we one of two particle system unit tests.

Our lander is a bit old fashioned and still uses thrust in high gravity fields. In this chapter we outfit our lander with not one but three thrusters! We also learn about two Particle Systems supplied with Ranger.

Particle Systems are infinite in nature. There is no end-all to be-all Particle System and Ranger doesn't attempt one either. However, Ranger does supply two "starter" systems, where depending on your requirements, can be used in an actual game. They both provide the basic effects such as rotations, translations and color cycling.

It is highly likely that once you see Ranger's systems you will become inspired to create your own, besides it is fun creating particle systems.

Particle

To understand Particle Systems (PS) we need to look at Particles first. The most basic property of a particle is its lifetime. From there properties start to pile on; position, color, scale, rotation etc. Except for lifetime all these properties can have variations applied to them over the lifetime of the particle.

Ranger splits a particle into two aspects: Data and Visual. Data are non visual aspects of a particle, for example lifetime, velocity, transforms. Visual is the rendered representation of a particle, for example, circles, squares, stars etc.

Particles can have behaviors applied to them. These behaviors modify a particle's property over time, for example transitioning from a beginning color to an end color.

Ranger comes with three particles that range from simplistic (SimpleParticle) to complex (TweenParticle) see Figure 8.1. All of them are poolable. Why? Because particles live and die quickly and it would be disastrous on the GC if they were allocated/deallocated continuously. Having said that, it is the PSs that chooses to either manage the particles or pool them. All of Ranger's PSs manage the particles until the system is released. Alternatively, you could define your own PS that pools particles in order to conserve memory.

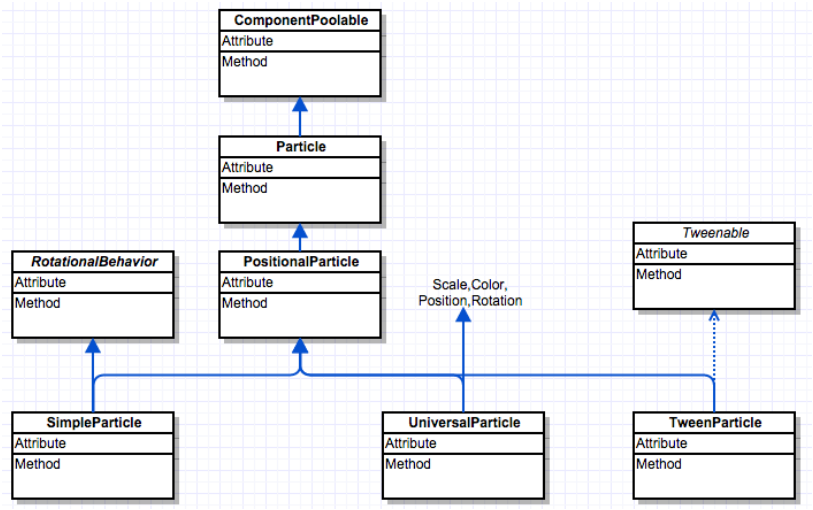


Figure 8.1 Particles

The simplest particle is **SimpleParticle**, it's pretty basic. It can translate and rotate particles and that is it. It exists solely as a clean example of constructing a particle behavior. The particle you will most like use (if you haven't coded your own) is either **UniversalParticle** or **TweenParticle**. Tweenable is the more capable particle that can manipulate all four common properties: Scale, Rotation, Translation and Color, and it does this by implementing TweenEngine's Tweenable. It works similar to the landing gear in that animation is handled by Tweens.

UniversalParticle is a bit faster in that animations are handled by dedicated LERP functions instead of Tweens being created on a per particle basic. Again, these three particles are examples that your are free to use, however, it's generally recommend that you create your own in order for you to optimize performance. PSs are very computationally intense and knowing exacting what your particle needs to do is critical. Hint, we will create our own custom Particle.

Particle Systems

Particle Systems (PS) manage particles by generating, emitting and handling each particle's active state. It is the PS that activates/deactivates a particle's state, Particles never manage their own. Ranger comes with two PSs:

- **BasicParticleSystem**: provides standard trigger and explode activation features.
- **ModerateParticleSystem**: provides the basic plus prototype cloning, variable emission rate and periodic pausing.

Activators

However, PSs do not actually activate particles, that is the responsibility of Activators. Figure 8.2 shows the Activator framework and how it relates to PSs. Activators are associated with a PS and when the PS needs to activate a particle it calls on the Activator to do so.

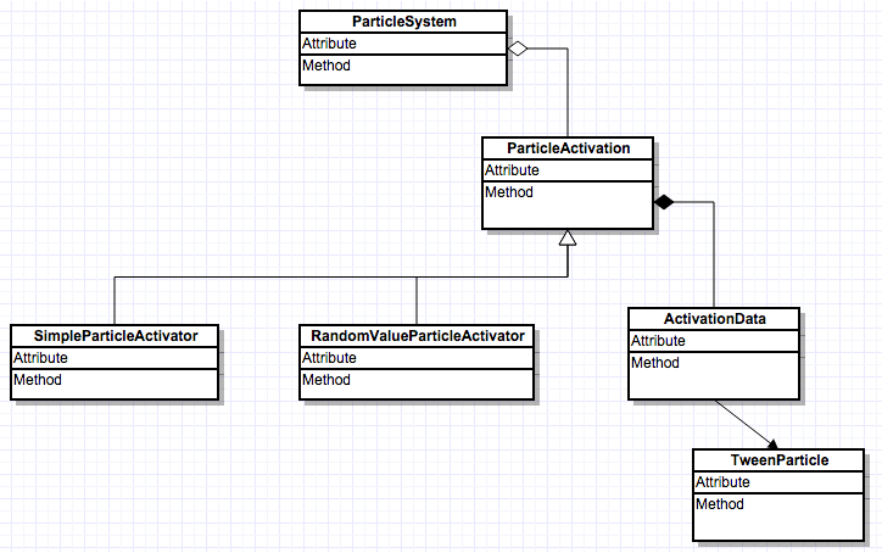


Figure 8.2 Activator framework

Once the Activator has generated pre-emission values (i.e. **ActivationData**) it attaches the data to the particle and calls the particle's *activateAt* method. The particle knows to access the data through the ***data*** property. Code 8.3 shows a snippet from UniversalParticle's *activateAt* **override**.

Code 8.3

```
@override
void activateAt(double x, double y) {
  // Take the activation values and configure the behaviors.

  if (data != null) {
    ActivationData pd = data as ActivationData;

    fromColor.setWith(pd.startColor);
```

Emitters

Emitters describe *how* particles are emitted spatially, and they are described by their shape. More advanced PSs can emit particles based on a point, areas, shape boundaries or curves. Ranger has only one emitter type: point. This type is implicit by design, meaning particles are simply emitted at a location/position.

Direction

Emitters define *where* a particle is emitted but they don't define the *direction*. Direction is specified at the very moment a particle is to be emitted. Currently there are six different directions defined in the **ParticleActivation** class (Figure 8.0 shows Omni, Drift and RadialSweep respectively):

1. **Uni**: used for bullets, for example, a gun.
2. **Omni**: used for explosions and fire effects.
3. **Drift**: could be used for physics effects.
4. **Variance**: typically used for rocket engine exhausts.
5. **PingPong**: could be used for physics like effects.
6. **RadialSweep**: could be used to augment other effects.

Each time you activate a particle you specify the emission style (aka direction):

```
if (_firing) {
  _firing = false;
  gunPS.activateByStyle(Ranger.ParticleActivation.UNI_DIRECTIONAL);
}
```

Behaviors

Once a particle is emitted it needs to behave in a certain way, for example, transitioning from a start color, at the beginning of the particle's life, to an end color just as the particle dies. This transition is a behavior. Some particles have this behavior directly embedded—the TweenParticle for example. While others (UniversalParticle) *mix-in* their behavior piecemeal.

Lets not kid ourselves there is an infinite number of behavioral patterns possible, and Ranger doesn't even try to go down that path. Nonetheless, Ranger does provide a few "templates" that you can use or copy. Because particles can be pretty intense computationally I would seriously suggest that you create particles that match your exact requirements. Each additional "thing" added to a particle quickly adds up when you are dealing with hundreds (if not thousands) of active particles.

Knowing about Particles, Particle Systems, Activators, Emitters, Direction and Behaviors lets create our own Particle and Activator, and use one of the supplied Particle Systems. Our particle will be a simple circle with two behaviors: Scale and Color. When the particle is first emitted our Activator will generate a start/end scale and color.

Particle Construction

To construct our particle we will mix-in three behaviors:

1. **PositionalParticle**: Applies a velocity and acceleration. It also contains the visual aspect of the particle.
2. **ParticleScaleBehavior**: Linear interpolates (LERP) from a "beginning" scale to an "end" scale over the lifetime of the particle. Lerping is fairly simple. We will use a slightly faster but imprecise version.
3. **ParticleColorBehavior**: Linear interpolates from a "beginning" color—RGBA—to an "end" color over the lifetime of the particle.

Create a new class called **CircleParticle** in the *actors* folder. Extend the class using PositionalParticle and mix-in Scale and Color behavior:

```
class CircleParticle extends Ranger.PositionalParticle with
Ranger.ParticleScaleBehavior, Ranger.ParticleColorBehavior {

}
```

PositionalParticle carries pooling functionality which necessitates that we add factories to support it. Create a factory method that takes a **Node**. This Node is the visual representation of the particle:

```
  CircleParticle();

  factory CircleParticle.withLifeAndNode(double lifespan, Ranger.Node node) {
    CircleParticle p = new CircleParticle._poolable();
    p.initWithNode(node);
    return p;
  }

  factory CircleParticle._poolable() {
    CircleParticle poolable = new Ranger.Poolable.of(CircleParticle,
_createPoolable);
    poolable.pooled = true;
    return poolable;
  }

  static CircleParticle _createPoolable() => new CircleParticle();
```

Particles could number in the hundreds if not thousands so it would convenient if our particle could clone itself. Add a clone method:

```
  CircleParticle clone() {
   Ranger.Node nodeClone = node.clone();

   CircleParticle p = new CircleParticle.withNode(nodeClone);

   p.initWithColor(fromColor, toColor);
   p.initWithScale(fromScale, toScale);

    return p;
  }
```

Now we need to handle the activation of our custom particle. This happens in the *activateAt* **override** method that expects its activation data to arrive in the ***data*** property. *activateAt* is called by an Activator that will we be creating very soon.

```
@override
void activateAt(double x, double y) {
 if (data != null) {
    Ranger.ActivationData pd = data as Ranger.ActivationData;
    // TODO transfer data
 }
  else {
    throw new Exception("Particle's Data not present.");
  }

  super.activateAt(x, y);
}
```

Finally we **override** the PositionalParticle's *step* method to allow "stepping" the behaviors we mixed-in:

```
void step(double time) {
  velocity.applyTo(node.position);

  stepColorBehavior(time);
  _colorMix.color.setWith(color);

  stepScaleBehavior(time);
  node.uniformScale = scale;
}
```

Notice that I don't call super.*step*. This is because I don't want the super's behavior which is to apply acceleration.

That takes care of a first draft of our particle which represents the Data aspect. Now we need to construct the Visual aspect.

Particle Node

Our particle's visual requirements are that it has a circle shape with the ability to change color over time. This shape Node is pretty simple and has already been coded. Go ahead and copy the *circle_particle_node.dart* file from the book's asset repository into the *nodes* folder. The class includes the standard factory creation code, cloning code used during population of Particle Systems and color the **Color4Mixin** behavior used in the **override** *draw* method:

```
class CircleParticleNode extends Ranger.Node with Ranger.Color4Mixin {
…
  @override
  void draw(Ranger.DrawContext context) {
    context.save();

    CanvasRenderingContext2D context2D = context.renderContext as
CanvasRenderingContext2D;

    context2D..fillStyle = color.toString()
       ..beginPath()
       ..arc(0.0, 0.0, 1.0, 0, Ranger.DrawContext.TPi)
       ..closePath()
       ..fill();
    context.restore();
  }
```

This Node will be used to render each particle emitted by our Activator. So how about we setup a particle activator to feed data into our particle so that we can finish the *activateAt* method.

Particle Activation

Prior to activating a particle we need to generate its starting and ending conditions. This happens via an Activator. Activators extend the **ParticleActivation** class and implement two important methods: *activate* and *deactivate*.

Create a new class called ParticleActivator inside the *actors* folder and add stubs for the required methods:

```
class ParticleActivator extends Ranger.ParticleActivation {

  void activate(Ranger.Particle particle, int emissionStyle, double posX, double
posY) {
  }

  void deactivate(Ranger.Particle particle) {

  }
}
```

For our particle we need values for the following properties (We could add more but this should be plenty for a rocket exhaust):

- **Lifetime**
- **Start scale**
- **End scale**
- **Speed**
- **Start color**
- **End color**

To generate the values Ranger has a handy **Variance** class that can generate values over a range that deviates by a variance value. Lets add the Lifetime variance property to ParticleActivator:

```
Variance lifetime = new Variance.initWith(1.0, 2.0, 1.0);
```

This gives us a lifetime that varies by 1 second between 1 and 2 seconds. Each time we need a new value we call the Variance's ***value*** property. Lets transfer that value into the ActivationData during the *activate* method, and add the other properties too (Code 8.4.) The *initWith* factory parameters are basically arbitrary as they will overridden by the Activator during activation.

Code 8.4

```
class ParticleActivator extends Ranger.ParticleActivation {
  Ranger.Variance lifetime = new Ranger.Variance.initWith(0.5, 1.0, 0.5);
  Ranger.Variance speed = new Ranger.Variance.initWith(1.0, 3.0, 2.0);
  Ranger.Variance startScale = new Ranger.Variance.initWith(5.0, 25.0, 10.0);
  Ranger.Variance endScale = new Ranger.Variance.initWith(5.0, 25.0, 10.0);
  Ranger.Color4<int> startColor = Ranger.Color4IRed;
  Ranger.Color4<int> endColor = Ranger.Color4IYellow;

  void activate(Ranger.Particle particle, int emissionStyle, double posX, double posY) {

    activationData.lifespan = lifetime.value;
    activationData.speed = speed.value;
    activationData.startScale = startScale.value;
    activationData.endScale = endScale.value;
    activationData.startColor.setWith(startColor);
    activationData.endColor.setWith(endColor);
  }
...
```

Now lets handle the emission style. For our lander we need a particle emission that emits in a certain direction but deviates by a certain variance (aka spread) or ParticleActivation.**VARIANCE_DIRECTIONAL**:

```
    switch(emissionStyle) {
      case Ranger.ParticleActivation.VARIANCE_DIRECTIONAL:
        angle.min = -angleVariance;
        angle.max = angleVariance;
        angle.variance = angleVariance / 2.0;

        activationData.velocity.directionByDegrees = angleDirection + angle.value;
        break;
      default:
        throw new Exception("Only VARIANCE_DIRECTIONAL is supported by this
Activator.");
        break;
    }
```

This is what forms a simple “spreading” like particle motion. As each particle is emitted a slight angle variation is added to the emitter direction. A slightly better approach would be to have an emitter (shaped like a curve) such that the particle’s emission follows the tangent of the shape thus mirroring the engine’s exhaust more closely. In our case we will “move” the point-emitter slightly inward to try and create the same effect.

Now that we have variance values configured we can pass this data to the particle and finally activate it:

```
    // Pass the activation data along with the particle about to be
    // activated.
    particle.data = activationData;

    // Now that the particle is configured we can finally activate it.
    particle.activateAt(posX, posY);
```

Together with Ranger’s PS, our **ParticleActivator** and **CircleParticle** we can now begin emitting particles—but where?

Emitter Target Location

Particles are emitted to a location and this location will always be in some Space, (Lander or LevelRimbaloidLayer space.) These Spaces where the particles are emitted can have a large effect on how they move once emitted. For example, if you were to emit the particles in the same Space as the Lander, whenever the Lander turned your particles would swing around with the Lander—not really a good behavior for rocket exhausts. What we want is for the particle to have motion of their “own” independent of the Lander. This means particles must be emitted into a different Space, which would imply two Spaces: the emitter Space and particle motion Space.

If we were to emit particles from the Lander’s Space (aka Lander local origin) then all the particles would appear to come from roughly the middle of the Lander’s hull, not quite what we want. What we need is a way to specify an arbitrary location relative to the Lander’s origin, and we can, it is called an **EmptyNode**.

EmptyNodes

An EmptyNode is basically a Node with no visual aspect—actually it does have a visual shape shaped like a tiny cross, but you have to enable it to see it. EmptyNodes are used for many things but the two most often uses are for target references (our Lander’s exhaust ports for example) or for anchors.

We will use three EmptyNodes one for each engine location (Figure 8.5.)

Enabling the EmptyNode’s visual and adding them as the “last” Nodes helps visually position them during coding  . Note: Once you hide the visual it doesn’t matter if the EmptyNode is “above” or “below” your Lander visually because the EmptyNode draws nothing. In your TriEngineRocket.*init* method create and configure each EmptyNode:

```
    _exhaust1 = new Ranger.EmptyNode();
    _exhaust1.uniformScale = 250.0;
    _exhaust1.setPosition(-90.0, -80.0);
    _exhaust1.drawColor = Ranger.Color3IWhite.toString();
    _exhaust1.iconVisible = true;
    _centroid.addChild(_exhaust1);
```

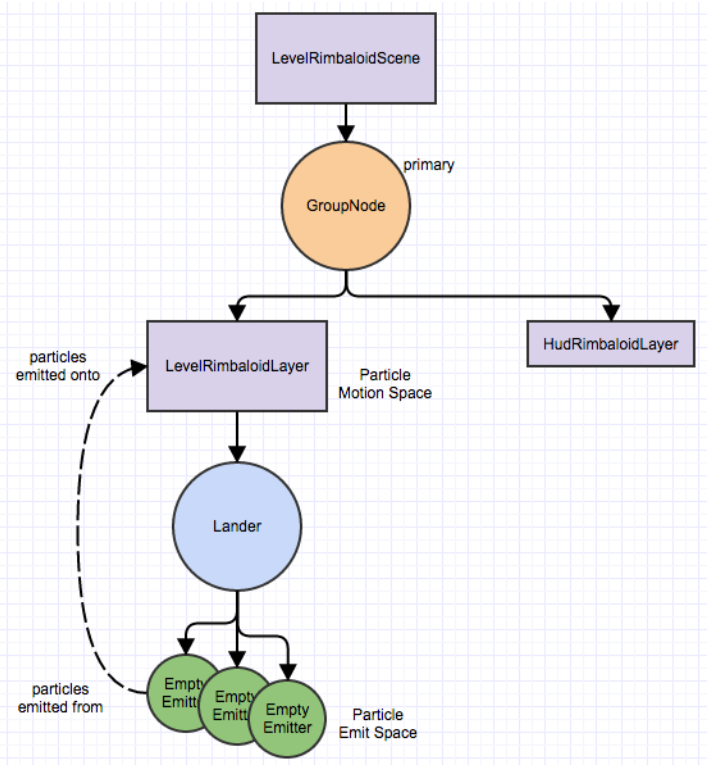


Figure 8.5 EmptyNodes

Once you have them visually where you want them simply remove the ***uniformScale***, ***drawColor*** and ***iconVisible*** calls and the icon disappears. These EmptyNodes are where the particles will be emitted from, but their placement in the Scene Graph is on the LevelRimbaloidLayer layer.

Integrating our Particles

We have everything thing we need to begin integrating our Nodes and classes into the TriEngineRocket Actor. Because we have three engines on our Lander we have two approaches we can take when emitting particles; in the first approach we use three separate PSs one for each engine, in the second approach, we use a single PSs for generating particles at each engine exhaust EmptyNode. Lets chose option two, a single PS. On each “update” we generate three particles that are emitted at each exhaust port. We have several things to do so lets get started:

- Create BasicParticleSystem
- Create ParticleActivator

- Bind Activator to ParticleSystem
- Create Particle Node prototype visual
- Create Particle prototype
- Populate Particle System with prototypes

First add a Ranger.**ParticleSystem** to TriEngineRocket:

```
Ranger.ParticleSystem _exhaustPS;
```

Add three methods called *_constructExhaust*, *_constructActivator* and *_populateParticleSystem*. In *_constructExhaust* we create and configure a BasicParticleSystem and then call the other methods for creating and populating our system:

```
void _constructExhaust() {
  _exhaustPS = new Ranger.BasicParticleSystem.initWith(30);

  ParticleActivator pa = _constructActivator();
  pa.angleDirection = _thrust.asAngleInDegrees;

  _exhaustPS.particleActivation = pa;

  // Construct Exhaust Particles
  _populateParticleSystemWithCircles(_exhaustPS);
  _exhaustPS.active = true;
}
```

I chose a particle count of 30 in the *initWith* factory method because the PS will be emitting for three engines.

In *_constructActivator* create our ParticleActivator class and set range values on all the relevant properties, for example, lifetime, speed, scale, angle and color. There are quite a few so only lifetime is shown:

```
ParticleActivator _constructActivator() {
  ParticleActivator pa = new ParticleActivator();
  pa.lifetime.min = 0.2;
  pa.lifetime.max = 0.5;
  pa.lifetime.variance = 0.2;

  ...
```

Populating a Particle System

PSs are typically made up of tens, hundreds or even thousands of particles. Specifying every one of them would be unwieldily, instead we create prototype for both Data and Visual to clone from. As mentioned earlier, a particle can't be seen by itself it requires a Node visual. The Visual is added to the Scene Graph but is manipulated by the particle. Therefore, in order to populate a PS we need to create both a Node and Particle and assign the Node to the particle.

Remember when we created our **CircleParticleNode** and **CircleParticle** class and we created *clone* methods as part of the class, it is through these methods that a PS will reference in order to populate itself.

Go ahead and create a *_populateParticleSystem* method and start it off by creating a CircleParticleNode, this will be our prototype Visual:

```
  void _populateParticleSystem(Ranger.ParticleSystem ps) {
   CircleParticleNode protoVisual = new CircleParticleNode.initWith(Ranger.Color4IBlack);
```

It typically doesn't matter what color you provide to the *initWith* factory method because most of the time you will "override" it with a ParticleActivator. The only time it really matters is if your particles lack a color behavior.

Next create the particle prototype using the Particle's *withNode* factory method supplied with the prototype Visual:

```
CircleParticle prototype = new CircleParticle.withNode(protoVisual);
```

And finally populate the PS using the prototype:

```
ps.addByPrototype(_particleEmissionSpace, prototype);
```

Notice that I also supply a Space (_particleEmissionSpace) where the particles will be emitted "into". Recall that we typically don't emit particles in the same Space as the emitter otherwise our particles would move in sync with the emitter which isn't what we want. Consider that the emitter EmptyNode is a child of the Lander's centroid GroupNode and when it rotates so does the EmptyNode which means any particles emitted into the Lander's Space would also rotate—not good.

Okay, now it is time to finish the job by filling in the finer details so that everything works correctly. First we go back CircleParticle's *activateAt* method add code to transfer the activation data into the behaviors:

```
      fromColor.setWith(pd.startColor);
      toColor.setWith(pd.endColor);

      fromScale = pd.startScale;
      toScale = pd.endScale;

      velocity.speed = pd.speed;
```

And we must not forget to "activate" the particle using the velocity and lifetime values. Not doing this will cause the particle to simply sit in one location and live "forever"!:

```
activateWithVelocityAndLife(pd.velocity, pd.lifespan);
```

Now we return to **TriEngineRocket** and add code to change the particle's direction and switch the PS to point to each port. First add the pitching code by creating a *_updateThrustOrientation* method that will modify both the Thrust vector and PS direction:

```
  void _updateThrustOrientation() {
    _thrust.directionByDegrees = node.rotationInDegrees + VISUAL_COORD_SYSTEM;
    if (_exhaustPS != null)
      _exhaustPS.particleActivation.angleDirection = _thrust.asAngleInDegrees +
180.0;
  }
```

Note that the PS emission direction is 180 degrees that of the Thrust Vector. This because the Thrust vector is "added" to the Lander's current position so the vector "faces" the direction of movement. However, the PS needs to emit particles in the "opposite" direction.

Now modify the *update* pitching code to call *_updateThrustOrientation*:

```
    if (_pitchingLeft && !_pitchingRight) {
      node.rotationByDegrees = node.rotationInDegrees + PITCH_RATE;
      _updateThrustOrientation();
    }
    else if (!_pitchingLeft && _pitchingRight) {
      node.rotationByDegrees = node.rotationInDegrees - PITCH_RATE;
      _updateThrustOrientation();
    }
```

Now we work on "cycling" the PS to emit a particle from each EmptyNode (aka Port) between each update event. First add a class scoped integer to track the port:

```
int _engineIndex = 0;
```

Now add cycling code using the modulus operator:

```
    _engineIndex = (_engineIndex + 1) % 3;
    Vector2 port = _exhaust1.position;
    if (_engineIndex == 1)
      port = _exhaust2.position;
    else if (_engineIndex == 2)
      port = _exhaust3.position;
```

We are almost done. Now we need to emit particles to the proper space which means we need to map from emitter-space to particle-space.

Space mapping

In Chapter 6 "Scene Graph" we got our first taste of Scene Graphs (SG) and how Spaces work in relation to a Node hierarchy. We have that very same situation now with emitting particles. Looking at Figure 8.5 we see that the EmptyNodes (Ports) are children of the Lander (***_centroid*** GroupNode.) If we were to simply set the PS's location to one of the Ports then emitted particles

would visually appears somewhere near the center of the view. Why? because the center of the emitter in local-space is 0,0 which maps to 0.0 in some other space. But this is clearly not what we want. We want to "use" the Port's location "as it appears" relative to the Lander's position but actually use the equivalent position in LevelRimbaloidLayer Space.

To do this requires that we utilize Space-mapping. Ranger's Nodes all come with methods for mapping from a Node's local-space to a root Space (aka World-space) and back into another Node's local-space. In our case we want to map from _***centroid*** Space to particle emission Space (aka LevelRimbaloidLayer).

Lets create a method called *_convertToEmissionSpace* to do just that (see Code 8.6.)

Code 8.6

```
Vector2 _convertToEmissionSpace(Vector2 location) {
  Ranger.Vector2P ws = _centroid.convertToWorldSpace(location);
  Ranger.Vector2P ns = _particleEmissionSpace.convertWorldToNodeSpace(ws.v);

  // Clean up.
  ns.moveToPool();
  ws.moveToPool();

  return ns.v;
}
```

In Code 8.6 we first map from _centroid Space to World-Space then turn around and map from World-Space to particle emission-Space. Note that I have moved the **ns** variable back to the pool *before* returning from the method. This is because I know that **ns** was "only" moved back to the pool—not deleted. I also know that my code will use the returned vector immediately and NOT hold a reference to it.

Pseudo Root

Considering that each mapping call involves an upward matrix traversal of the SG it would be in our best interest if we could "cut-some-corners." And there is one, called Pseudo Root. A pseudo root is a Node in the SG that represents a Node where any of the Nodes above it all have an Identity matrix. A Node with an Identity matrix contributes nothing to the upward traversal (i.e. it is wasted effort.)

In almost all cases a Scene's parent, and above, are "holding" an Identity matrix. Why? Because the Scene isn't under going any kind of transform when a transition is complete. The Scene (and it corresponding Layer) just sit there while the Actors do all the work. This means that our LevelRimbaloidLayer is pretty much a Pseudo Root which also means we really don't need to traverse upward past it when calculating a mapping matrix.

It just so happens that most of the mapping methods take an optional second parameter that identifies a pseudo root. Lets revisit *_convertToEmissionSpace* and add a pseudo root to both mapping calls:

```
  Ranger.Vector2P ws = _centroid.convertToWorldSpace(location, _particleEmissionSpace);
  Ranger.Vector2P ns = _particleEmissionSpace.convertWorldToNodeSpace(ws.v, _particleEmissionSpace);
```

_particleEmissionSpace *is* LevelRimbaloidLayer. This property is set using a setter that is called during LevelRimbaloidLayer._*configure*:

```
gm.triEngineRocket.particleEmissionSpace = this;
```

Now we can return to the *update* method and map the Port location and set the PS location appropriately:

```
    Vector2 gs = _convertToEmissionSpace(port);
    _exhaustPS.setPosition(gs.x, gs.y);
```

Although that was a fair amount of code the bulk of which was the construction of the particle system and its configuration. If you run Moon Lander and navigate through the rockets then hold the "/" key down you will see three streams of particles flowing from the bottom of the Lander. Yeah! Lets review.

Recap

We learned that there are four components when dealing with particle systems:

1. ParticleSystems,
2. Activators,
3. Particle
4. and Particle Node.

Each plays a vital role; the ParticleSystem manages particle updates and provides methods for triggering particles, Activators generate and activate particles, Particles live and die and whose's visual aspect manifests through particle Nodes.

Once we had our components created and configured we could integrate them into actual code inside TriEngineRocket, and instead of using three PSs we decided to use one PS and round-robin its location at each EmptyNode port. Then we modified the pitching code to update the PS's direction to coincide with the Thrust vector, and finally we added mapping code that emits particles in the equivalent emission space relative to the port.

The key for the whole thing to work was the mapping from one space to another via World-space. Looks like it is time to step back and look at the bigger pitcher—the realm of the GUI!

Part III

Leveling Up

In Part III we advance a more formal structure of our Ranger game. This includes things such as menus, popups, storage, audio.

- Chapter 9: Dialogs
- Chapter 10: Popups
- Chapter 11: Audio Effects
- Chapter 12: Storage
- Chapter 13: Physics collision

Chapter 9 Dialogs

Ranger doesn't have a formal GUI api. You build them using either Nodes or HTML/CSS with a little bit of imagination sprinkled in.

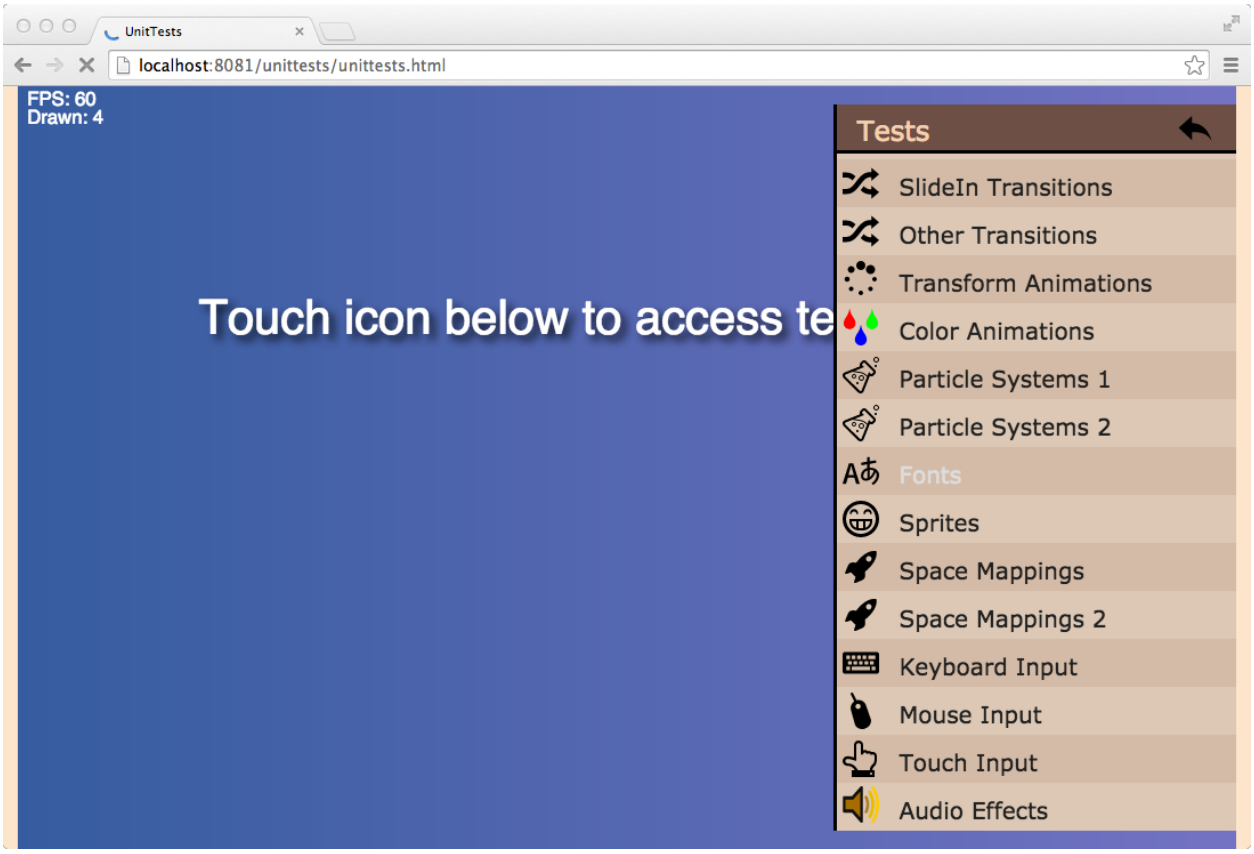


Figure 9.0 Unit test HTML menu

In Figure 9.0 we see Ranger's Unit test GUI built using HTML/CSS. The menu animation is driven my Tween animations custom designed to manipulate the properties of HTML elements. This is just one approach to GUIs in Ranger games.

Flow

GUIs are infinite in nature and Ranger makes no attempt to wrangle them. However, that doesn't mean you don't have a UI-kit, it's called HTML/CSS. But that's not all, you can also build your GUI with Ranger's Nodes! All it takes is a little bit of imagination and you're not restricted from using either approach—as a matter of fact you can use them both at the same time!

Pretty much all of Moon Lander's GUI will be built with Ranger Nodes—it is possible that a few pieces may be HTML, for example player initials input.

How about we start by laying out a general flow of Moon Lander. Take a look at Figure 9.1.

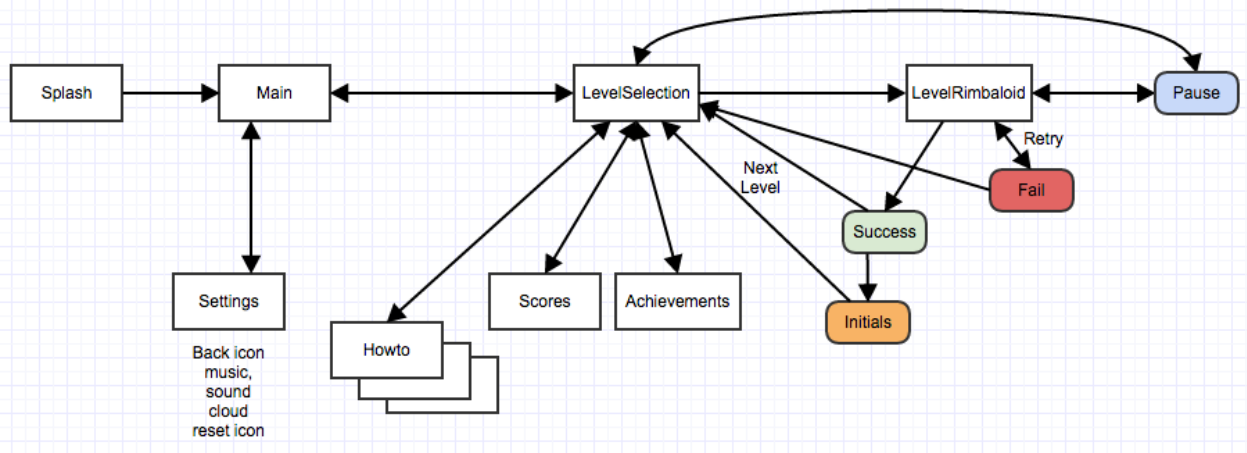


Figure 9.1 Flow Layout.

In previous chapters we created some of the elements in Figure 9.1, for example, the Splash Scene, Main Scene, Level Selection and LevelRimbaLoid Scene. Flow wise they aren't complete. We coded them with just enough logic to get us to the first game level. Now it's time to make another pass and in the process learn more about Ranger and its Nodes.

We will start with MainScene's Settings scene. This will give us our first chance at creating GUI nodes in combination with animations.

Settings Dialog Draft

Out first GUI component we will create is a settings dialog. This dialog requires the following features:

- Music
- Sound
- Cloud
- Credits and copyrights.
- Game reset. Issues a Popup (Chapter 10)
- Back button
- Title

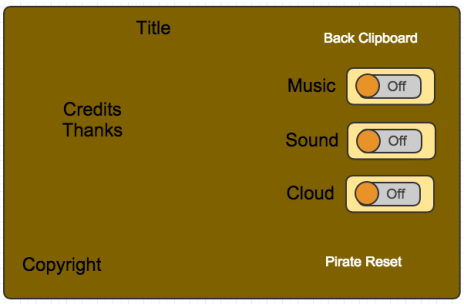


Figure 9.2 Settings draft

It is always handy to have a rough draft of what we are trying to achieve. Again I turned to my trusty Chrome Gliffy diagramming application to drum up a rough draft (Figure 9.2.) This draft gives us something to aim for, how do we show it? We already have a button on the MainLayer for launching our dialog we just need to "wire" it up.

There are many approaches we can take when dealing with dialog flow, for example, *who* actually launches the dialog? For this book we are going to take a third person perspective approach. What this means is that a piece of code makes a request to show a dialog rather than instantiating one directly and showing it. Some "other" piece of code will listen for the request and do the work of instantiating the dialog and showing it. How do we make this happen? Easy, using an Event/Message Bus.

Note

Ranger uses Marco Jakob's EventBus Pub package for an Event bus framework. It's simple to use, and it follows the ubiquitous publish/subscribe pattern using Dart's Streams. We don't need to import EventBus because Ranger has already imported it for its own use.

In our case the MainLayer will make a request (transmit) to show the Settings dialog box and the MainScene will honor (listen-for) the request by showing it. On the left half of Figure 9.3 you can see the MainScene and MainLayer communicating through an Event Bus.

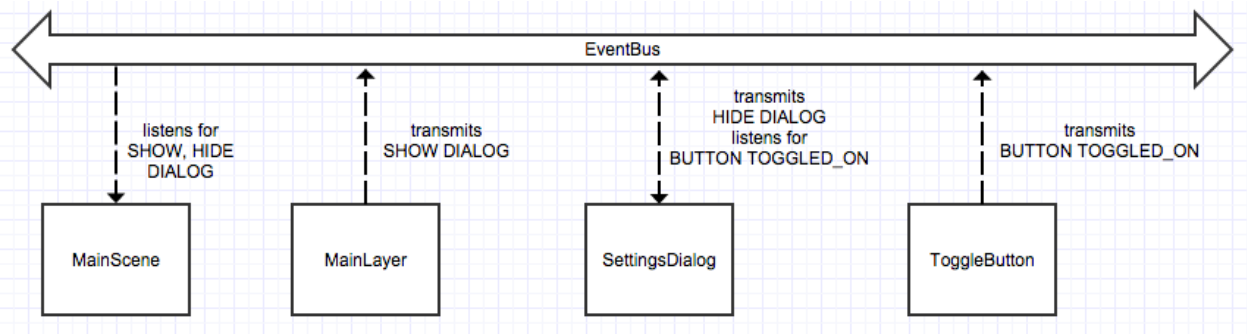


Figure 9.3 Event Bus

This type of loose coupling gives us the luxury of using GUI components from just about anywhere. On a side note a fair amount of UI-kits use messaging as a foundation in their SDKs.

So what is the general flow of events when we want to show a dialog? How about we list them out:

1. Caller disables its inputs (the MainLayer)
2. Receiver dim itself (the MainScene)
3. Receiver shows Dialog
4. Dialog enables its inputs
5. Dialog starts any animations
6. User interacts with dialog
7. Dialog either holds data or transmits on the fly
8. User signals interaction is complete
9. Dialog transmits Hide-Dialog

10. Receiver hides Dialog
11. Dialog disables its inputs
12. Caller enables it inputs (the MainLayer)
13. Caller un-dims itself
14. End

This whole sequence is facilitated using the EventBus.

EventBus

The EventBus couldn't be any simpler to use. Just "listen" to or "fire/transmit" to the Bus. The question is just what are we listening for and firing out—anything. You can listen/fire any object you like as long as both parties are in agreement. To listen for a message you specify an object that you want to listen for by using the EventBus's *on* method:

```
ranger.eventBus.on(object).listen((object o){...})
```

If a message shows up on the Bus your anonymous method will be called with the object. To "put/transmit/fire" something onto the Bus you use the *fire* method:

```
ranger.eventBus.fire(object);
```

For our game we will transmit a custom object called MessageData.

MessageData

MessageData is a simple custom class that identifies what is being sent on the Bus. It's tailored specifically for our game's GUI system. The MessageData class has properties that help us identify various GUI related Bus events. For example, we need to indicate what types of actions are supported:

- Show,
- Hide,
- Toggled,
- Clicked

And *what* the action is associated with:

- Dialog,
- Popup,
- Button

And finally we need a property for specifying something unknown. There isn't much to the code (see Code 9.4.)

Code 9.4

```
class MessageData {
  static const int UNKNOWN = -1;

  static const int SHOW = 100;
  static const int HIDE = 101;
  static const int TOGGLED_ON = 102;
  static const int TOGGLED_OFF = 103;
  static const int CLICKED = 104;

  static const int DIALOG = 200;
  static const int POPUP = 201;
  static const int BUTTON = 202;

  bool handled = false;

  int actionData = UNKNOWN;
  int whatData = UNKNOWN;

  String data = "";
}
```

Now that we have a message object defined we can write receiver code that listens for our custom data. We return to MainScene were we add code that listens for messages requesting the Settings dialog to show.

First add a new method called *_listenToBus* that registers with the EventBus and call it from the *onEnter* event:

```
  @override
  void onEnter() {
    ...
    _listenToBus();
  }
```

Now code the *_listenToBus* method (see Code 9.5 minus comments.) Pretty simple, we just call EventBus's *on* method giving it an object-type to listen for. The *listen* method takes a handler that is called when the object-type shows up on the bus. Our handler is actually an anonymous function that filters on both the ***whatData***, ***data*** and ***actionData*** properties.

Notice also that the *listen* method returns a **StreamSubscription<T>**. We need to capture this object so we can cancel the stream when we are done listening to the EventBus. This allow us to create a property of Type **MessageData** that we can cancel on the **onExit** method:

```
StreamSubscription<MessageData> _busStream;
...
void onExit() {
  super.onExit();
  if (_busStream != null) {
    _busStream.cancel();
  }
}
```

If you don't cancel the stream then the next time we show the dialog we will have subscribed to the stream again which means will we get repeated events—not good.

You can see that we are listening for a very specific combination namely: DIALOG-SHOW-Settings and DIALOG-HIDE-Settings. We haven't created the dialog yet so we have nothing inside the **if** statements other than a **print** statement, the dialog will have the corresponding "firing" code to request that the dialog be hidden.

Code 9.5

```
  void _listenToBus() {
    _busStream = ranger.eventBus.on(MessageData).listen(
    (MessageData md) {
      switch(md.whatData) {
        case MessageData.DIALOG:
          if (md.actionData == MessageData.SHOW) {
            if (md.data == "Settings") {
              // Show dialog here
              print("Show dialog here");
            }
          }
          else if (md.actionData == MessageData.HIDE) {
            if (md.data == "Settings") {
              // Hide dialog here
            }
          }
          break;
      }
    });
  }
```

But there *is* something we can add immediately, a translucent blackout overlay. The overlay helps signify that the MainScene isn't in focus while the dialog box is visible. To add the blackout overlay add a RectangleNode property to MainScene:

```
RectangleNode _blackout;
```

In the *onEnter* method/event create and configure the Node. The default visibility is set to false such that the dialog is ignored by the Scene Graph until requested:

```
      _blackout = new RectangleNode.basic(new Ranger.Color4<int>.withRGBA(0, 0, 0,
128));
      _blackout.centered = false;
      _blackout.visible = false;
      _blackout.scaleX = dw;
      _blackout.scaleY = dh;
      _blackout.setPosition(0.0, dh / 2.0);
      addNode(_blackout);
```

Now set the blackout node's visibility in the SHOW and HIDE **if** statements in the *_listenToBus* method:

```
        if (md.actionData == MessageData.SHOW) {
          if (md.data == "Settings") {
            _blackout.visible = true;
          }
        }
        else if (md.actionData == MessageData.HIDE) {
          if (md.data == "Settings") {
            _blackout.visible = false;
          }
        }
```

We can also do one other thing while we are here. Whenever the dialog is shown or hidden we need to remove or add focus to any Layers, which in our case is the MainLayer. Go ahead and override the Scene's *focus* method making sure to call the Layer's *enable* method:

```
@override
void focus(bool gain) {
  super.focus(gain);
  _mainLayer.enable(gain);
}
```

By calling Layer's *enable* method you give the Layer a chance to do any prep work prior to any dialogs appearing.

Now we are ready for the other end of the "pipe" so to speak—transmission onto the EventBus from the MainLayer (reference Figure 9.3.) We are now ready to finish "wiring up" the hit detection code for the rotating gear  . Remember back in Chapter 6 Code 6.14? We are going to finally complete that code by firing off a message onto the EventBus. Return to the MainLayer.*onMouseDown* method and add the following code inside the *pointInside* **if** statement (see Code 9.6.)

Code 9.6

```
    if (_gear.pointInside(nodeP.v)) {
      nodeP.moveToPool();

      // Send message requesting "Settings Dialog". The MainScene will be
      // listening to this event.
      MessageData md = new MessageData();
      md.actionData = MessageData.SHOW;
      md.whatData = MessageData.DIALOG;
      md.data = "Settings";
      ranger.eventBus.fire(md);

      return true;
    }
```

Notice how I create a MessageData object and transmit it using the *fire* method. That's all there is to it. Now run Moon Lander and click the gear icon. You should see in the Console (aka moonlander.html output tab) the text "Show dialog here" and the MainLayer should have dimmed indicating loss of focus.

We have now laid the ground work for launching our Settings dialog that we drafted earlier (see Figure 9.2.)

Settings Dialog Implementation Part 1

But just what is a Dialog? Is it a special Node? Nope, it turns out it is any kind of Node Ranger understands. However, for our dialogs we are going to extend BackgroundLayer in order to gain input functionality automatically. Recall that Scenes can have multiple Layers—actually Layers can have Layers within Layers too. I prefer to have only Layers as children of Scenes, which means Moon Lander's approach will follow this style.

Before we implement our dialog lets think ahead just a little bit. We know we are going to have several dialogs within in our game and all of them will have pretty much that same basic lifecycle features, for example, show, hide and focus. Perhaps we should create an abstract base class with a few basic methods we expect each of our dialogs to have. Create a new file called *dialog.dart* in the *dialog* folder and code it as shown in Code 9.7.

Code 9.7

```
abstract class Dialog extends Ranger.BackgroundLayer {
  void show();
  void hide();
  void focus(bool b);
}
```

Now extend the **Dialog** class and call it **SettingsDialog**. But wait. One of our requirements was to "slide" the dialog in from the bottom, this means we need to mix-in the **Tweenable** class too. Code 9.8 is our basic starter, it's going to grow to several hundred lines of code when all is said and done.

AutoInputs

Notice something slightly different in the *withSize* factory method? The property *autoInputs* is being set to **false** (Code 9.8.) By default Layers automatically enable-disable their Inputs whenever *onEnter* or *onExit* messages are sent. This happens when a Scene enters and exits the Stage (aka pushed or popped.) We will see why this is important after we code the dialog.

We don't want this default behavior, so our dialog class will control enablement of the Inputs manually when the dialog is shown or hidden hence the property is set to **false**.

Code 9.8

```
class SettingsDialog extends Dialog with UTE.Tweenable {
 SettingsDialog();

  factory SettingsDialog.withSize([int width, int height]) {
    SettingsDialog layer = new SettingsDialog()
      ..init(width, height)
      ..autoInputs = false
      ..tag = 800
    return layer;
  }

  @override
  bool init([int width, int height]) {
    if (super.init(width, height)) {
    }
   return true;
  }

  void show() {}
  void hide() {}
  void focus(bool b) {}

  int getTweenableValues(UTE.Tween tween, int tweenType, List<num> returnValues) {
    return 0;
  }

  void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) { }
}
```

The next thing to do is “inflate” our dialog so we can see it and position accordingly. Our dialog won’t **override** the BackgroundLayer’s *drawBackground* method but instead use a RoundRectangleNode to act as the background. In addition, we set the Layer’s ***transparentBackground*** property to true in the factory method:

```
RoundRectangleNode _background;
...
..transparentBackground = true;
```

And because our dialog can go out of focus during Popups we also add a blackout rectangle too:

```
RoundRectangleNode _blackoutOverlay;
```

These Nodes need to be created and configured so lets add a *_configure* method and call it from the overridden *init* method:

```
bool init([int width, int height]) {
   if (super.init(width, height)) {
     _configure();...
```

The *_configure* method will definitely grow as we add more Nodes to complete the requirements. But for now we are just going to add the background and blackout Nodes so we can see something when we click the gear icon (see Code 9.9.) Actually we only need the background Node but lets add the blackout while we are here.

Code 9.9

```
  void _configure() {
    _background = new RoundRectangleNode.basic(Ranger.Color4IRed)
      ..width = contentSize.width
      ..height = contentSize.height;
    addChild(_background);

    _blackoutOverlay = new RoundRectangleNode.basic(new
Ranger.Color4<int>.withRGBA(0, 0, 0, 128))
      ..visible = false
      ..width = contentSize.width
      ..height = contentSize.height;
    addChild(_blackoutOverlay);
  }
```

Next we add a few properties to control the general appearance. Add three properties for the background fill, outline color and corner radius:

```
  set backgroundColor(Ranger.Color4<int> c) => _background.fillColor = c.toString();
  set outlineColor(Ranger.Color4<int> c) => _background.outlineColor = c.toString();
  set cornerRadius(double r) => _background.cornerRadius = r;
```

The are a few other areas of code we need complete in order to see our dialog slide up from the bottom, and those are:

- *focus* method (this could be done later, but we get it out of the way.) Code 9.10.
- positioning out of view in the *onEnter* event
- *show* method. Code 9.11.
- complete the Tweenable methods that actually modify the position property.

Code 9.10

```
void focus(bool b) {
  if (b) {
    enableMouse = true;
    _blackoutOverlay.visible = false;
    enableInputs();
  }
  else {
    _blackoutOverlay.visible = true;

    disableInputs();
  }
}
```

Inside the *onEnter* event we center and position the dialog **out-of-view** in preparation of animating it into view:

```
  double strokeGap = 3.0;
  double s = 1.5;

  double hh = ranger.designSize.height / s;
  double hw = ranger.designSize.width / s;
  double dw = ranger.designSize.width;
  double dh = ranger.designSize.height;

  visible = false;

  // Both center and position the Dialog out of view.
  _verticalSlideDistance = hh + strokeGap;
  setPosition(dw - (dw / 2.0) - hw / 2.0, -_verticalSlideDistance);
```

Next we code the *show* method (Code 9.11.) Because our dialog mixes in Tweenable we can pass our dialog (“this”) directly to the TweenManager for animating.

Code 9.11

```
void show() {
  focus(true);
  visible = true;

  ranger.animations.track(_reset, Ranger.TweenAnimation.ROTATE);

  UTE.Tween moveBy = new UTE.Tween.to(this, TWEEN_TRANSLATE_Y, 0.25)
    ..targetRelative = [_verticalSlideDistance]
    ..easing = UTE.Sine.OUT;
  ranger.animations.add(moveBy);
}
```

The dialog will *not* animate into view until we complete the Tweenable methods:

```
  int getTweenableValues(UTE.Tween tween, int tweenType, List<num> returnValues) {
    switch(tweenType) {
      case TWEEN_TRANSLATE_Y:
        returnValues[0] = position.y;
        return 1;
    }
    return 0;
  }
 void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) {
    switch(tweenType) {
      case TWEEN_TRANSLATE_Y:
        setPosition(position.x, newValues[0]);
        break;
    }
  }
```

Above we are filtering on our custom defined tweenType defined as a static class property:

```
  static const int TWEEN_TRANSLATE_Y = 1;
```

On every call by the animation system the *setTweenableValues* takes the incoming value (**newValues[0]**) and feeds it directly into the dialog's ***position*** property. This is exactly how our dialog animates into view. The *show* method setups up the animation and *setTweenableValues* "animates" the dialog by changing its position.

The basics of the Settings dialog is complete. But someone needs to show it, and for that we return to MainScene where it is listening on the EventBus for the request.

First we add a **SettingsDialog** property:

```
SettingsDialog _settingsDialog;
```

Next create and configure the dialog Node in the *onEnter* event:

```
      double s = 1.5;

      double hh = ranger.designSize.height / s;
      double hw = ranger.designSize.width / s;

      _settingsDialog = new SettingsDialog.withSize(hw.toInt(), hh.toInt());
      _settingsDialog.backgroundColor = Ranger.color4IFromHex("#866761");
      _settingsDialog.outlineColor = Ranger.color4IFromHex("#e4d5d3");
      _settingsDialog.hide();
      addLayer(_settingsDialog);
```

Finally we can code the actual call to show the dialog in the *_listenToBus* method where we perform three things:

1. Disable focus of MainLayer
2. Show the Settings dialog

3. Make MainScene's blackout Node visible

```
        case MessageData.DIALOG:
          if (md.actionData == MessageData.SHOW) {
            if (md.data == "Settings") {
              focus(false);
              _settingsDialog.show();
              _blackout.visible = true;
            }
```

That's it! We now have enough code to actually show the dialog, sure there won't be anything in it but at least we can see it slide up from the bottom. Go ahead and launch Moon Lander now. Once you see the  icon click on it and you should see our dialog slide up (see Figure 9.12.)

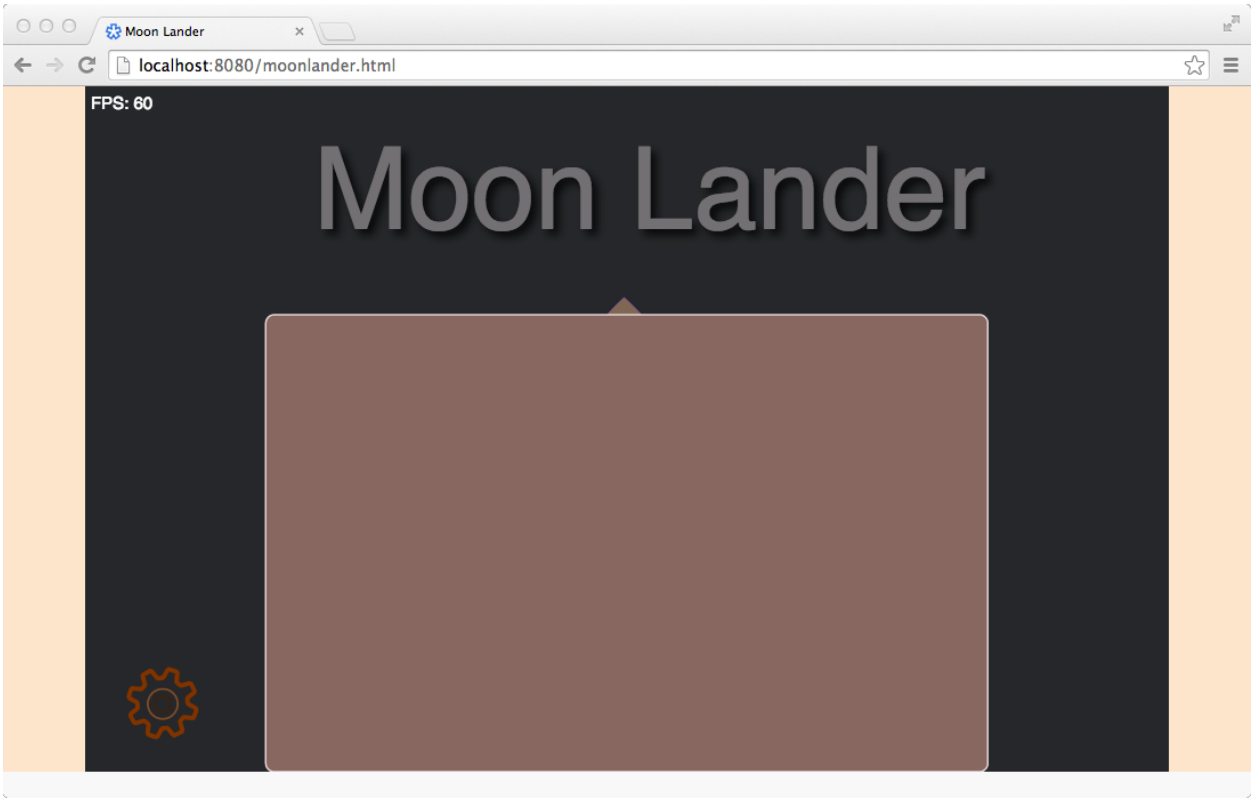


Figure 9.12 Basic Settings shell

Sweet!

Midway Recap

We just created and animated a dialog into view when an icon is clicked. How did we do it? Here are the steps:

- Created a base **Dialog** class that specifies a common set of behaviors.

- Created a **SettingsDialog** class by extending **Dialog** and implementing the **Tweenable** interface (Figure 9.13.)
- Modified MainScene to listen for dialog requests
- Modified MainLayer to transmit a dialog request when gear icon clicked.
- Overrode MainLayer's *enable* method control gear icon animation when focus is gained or lost.
- Added a Blackout Node for both MainScene and SettingsDialog when focus is gained or lost.
- Coded SettingsDialog's *show* and *hide* methods to animate the dialog into and out of view.

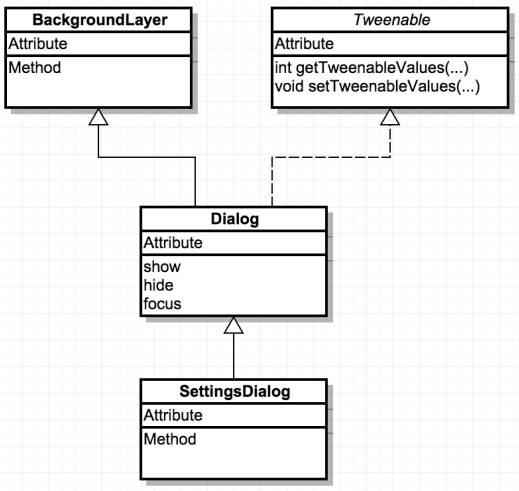


Figure 9.13 Settings Dialog

- Code Tweenable's get/set animation callback to apply tween values to the dialog's position property.

In a nutshell, the MainLayer transmits a request to show the SettingsDialog where upon MainScene responds by showing the dialog.

Toggle Buttons

Looking back at our design draft (Figure 9.2) we can see we need two icon buttons and a bunch of TextNodes, but we also need some sort of widget that can be toggled on and off. We have already handled icon buttons and TextNodes, but what about toggle buttons? For that we create a new base class called Button that layout a common behavior (see Figure 9.14.)

Notice that we don't extend **BackgroundLayer** this time but instead extend **GroupNode**. This is because our button will "share" Input with the current BackgroundLayer (aka SettingsDialog) that is in focus. Whenever a click occurs on the **SettingsDialog** we also call the button's *check* method. This allows our buttons to be used on any Layer without having to juggle focus. The reason for extending GroupNode is because our buttons may have several Nodes that make up the actual button.

Our **Button** class still mixes in the **Tweenable** interface because our toggle buttons will want to animate a toggle knob back and forth, plus we want to fade the text in on each toggle.

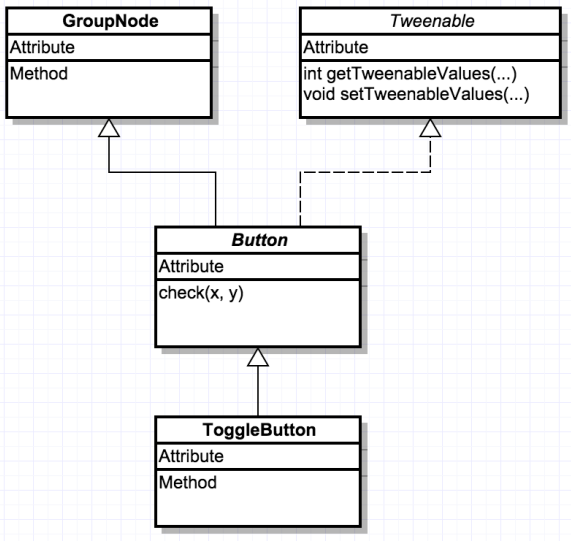


Figure 9.14 Buttons

As hinted above our Button class will specify a *check* method that should be implemented by a concrete **ToggleButton**. Lets go ahead and create a Button and ToggleButton class that can be used on the Settings dialog. First is the Button class:

```
abstract class Button extends Ranger.GroupNode with UTE.Tweenable {

  bool check(int x, int y);
}
```

Next the ToggleButton class. The Node arrangement will follow the Scene Graph structure depicted in (Figure 7.24 B) in that all the children Nodes are siblings. You could use arrangement A as long as you control child transform correctly. Again, each has it advantages, but for this button I have decided on arrangement B. Note: the code for it is about 200 lines so for brevity I will list just the highlights that bring attention to key points; you can download the class from the book's repository.

Our goal is to create something similar to this 

. Each time they click the yellow area we toggle the button. This means we need at least four Nodes. The picture seems to indicate that some Nodes should be children of each other but that isn't required. Looking at the class from the repository you can see the four main Node properties:

```
class ToggleButton extends Button {
  ...
  RoundRectangleNode _background;
  RoundRectangleNode _track;
  CircleNode _knob;
  Ranger.TextNode _caption;
```

The *_construct* method does the usual create and configure by positioning all the Nodes relative to each other. Looking at just the *setTweenableValues* method we can see the ***_knob***'s position is being animated in the X direction. We also see the ***_stateNode***'s text opacity value being animated for a fade-in/fade-out effect:

```
  void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) {
    switch(tweenType) {
      case TWEEN_TRANSLATE_X:
        _knob.setPosition(newValues[0], _knob.position.y);
        break;
      case TWEEN_FADE:
        _stateNode.opacity = newValues[0].toInt();
        break;
    }
  }
```

The button's *check* method performs a *pointInside* test on the ***_background*** Node. If it was touched we animate both the knob and text, and at the same time transmit a message on the EventBus:

```
    MessageData md = new MessageData();
    md.whatData = MessageData.BUTTON;
    if (_state)
      md.actionData = MessageData.TOGGLED_ON;
    else
      md.actionData = MessageData.TOGGLED_OFF;
    md.data = caption;
    ranger.eventBus.fire(md);

    _animateText();
    _animateKnob();
```

I chose the background Node because it is the largest Node and easier to touch with a finger. To make a slightly larger target to hit I could have sized the background such that it encompassed the text/caption.

To be honest figuring out all the positional offset was a test and run approach. I created the Nodes with default settings except for some rough guesses on width and height. The knob animation was configured the same way using a testing and run approach until the knob animated the correct horizontal distance in both directions.

Now we can return to the SettingsDialog class and finish the job by adding all the text, buttons, toggles and their associated click code.

Settings Dialog Implementation Part 2

Return to SettingsDialog class and add three of our shiny new toggle buttons as properties, and two SpriteImage properties for the configured/back icon  and pirate  icon:

```
  Ranger.SpriteImage _configured;
  Ranger.SpriteImage _reset;
  ToggleButton _music;
  ToggleButton _sounds;
  ToggleButton _cloud;
```

Remember we need to add entries to the **Resources** class to fetch the two new SVG icons as well:

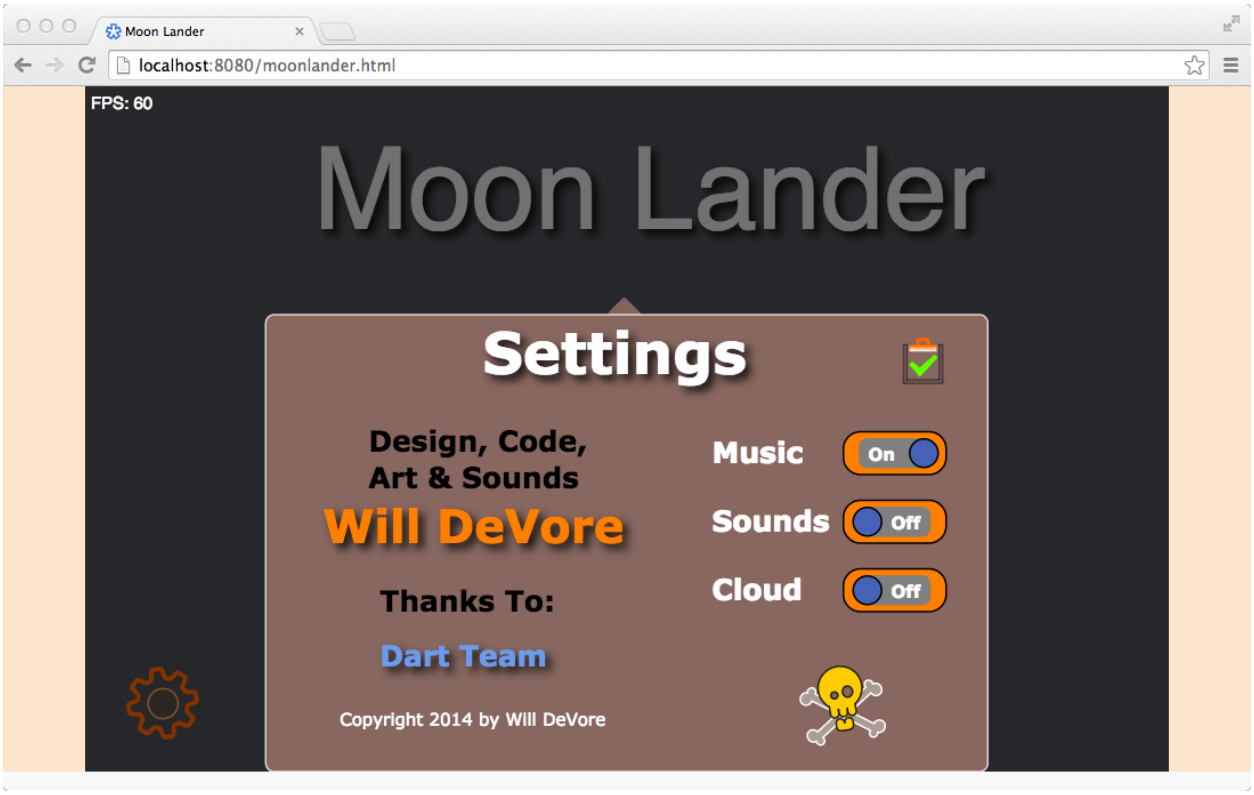
```
    _resourceTotal += 2;    // chicken and nuke
    loadImage("resources/Chicken_icon.svg", 128, 128).then((ImageElement ime) {
      chicken = ime;
      _updateLoadStatus();
    });
    loadImage("resources/nuc_it.svg", 60, 60).then((ImageElement ime) {
      nuke = ime;
      _updateLoadStatus();
    });
```

Next, update the *_configure* method by piling in judicious amounts of create and configuration code. There is quite a bit because of all the text Nodes—again, it is easier if you copy the book's repository code, nonetheless, lets review some of the code to see how it works.

First we look at the *onMouseDown* **override**. Here code was added to handle hit testing for both the ***_configured*** and ***_reset*** icons. Notice that they transmit onto the EventBus. When the user clicks on the ***_configured*** icon we make a request to hide the SettingsDialog, and when the user clicks on the _reset icon we make a request to show the **ConfirmReset** popup.

Secondly, we have overridden the *onMouseMove* event in order to "shake" the pirate icon whenever the mouse enters the icon and stop it when it exits.

Now if you run Moon Lander and click on the gear icon you should see a fully decked out dialog box complete with ToggleButtons:



Sweet again! However, we need to remember the settings the user made. That brings us to storage.

Lawndart

Lawndart is a Pub package that simplifies working with the browser's local storage. It's really easy to use so lets get started. Lawndart has a main object called **Store** that you create in order to access storage. It is basically a set of key/value pairs. We are going to harbor it in the **Resources** class and provide properties to control and store to it. But first things first, we need to reference the Pub and **import** it. If you don't remember how to reference a pub don't worry we will cover it here again:

1. Open the packages/pubspec.yaml file by double clicking it and switching to *Overview* tab.
2. Click the "Add…" button in the dependencies section.

3. In the dialog that pops up enter “lawn”. You should see Lawndart appear in the list. See Figure 9.15.
4. Click OK.
5. Save pubspec.yaml. As soon as you do Lawndart is pulled down into your project.
6. Finally, open *main.dart* and import the package:

```
import 'package:lawndart/lawndart.dart';
```

That wasn't to difficult. See figure 9.15 for what your pubspec.yaml should now look like.

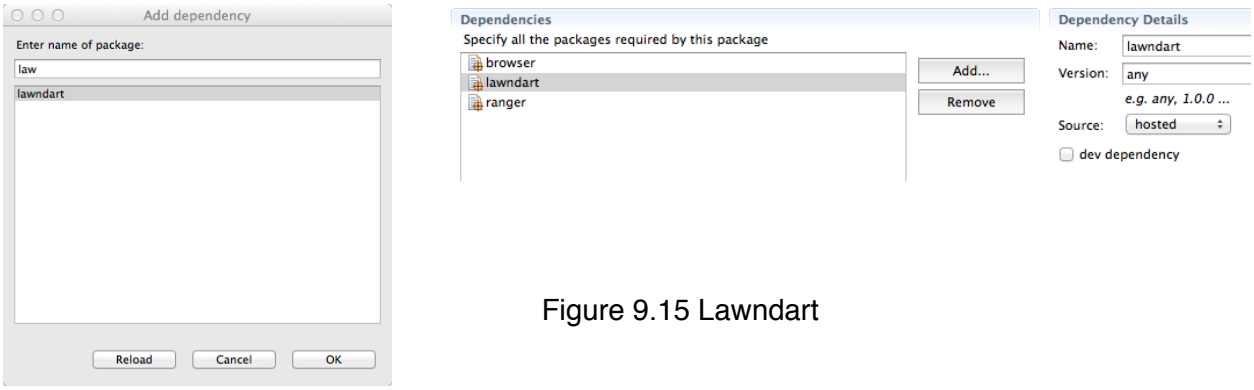


Figure 9.15 Lawndart

Now return to the **Resources** class and add a **Store** property:

```
Store gameStore;
```

With the store property we can create an actual store. Go to the **Resources**.*load* method and create the store:

```
_resourceTotal += 1;    // storage
gameStore = new Store("MoonLanderDB", "MoonLanderConfig");
```

If the Store doesn’t exist prior to access then it is created on the fly and a new store is returned. To create the store we need to give it a minimum of two things:

1. A top level database name and
2. a store name.

You can see above I named the database “**MoonLanderDB**” and the store name “**MoonLanderConfig**”. With the Store object in hand we can now open it and begin retrieving and storing key/value pairs. The Store object has an *open* method that needs to be called first. Once open we can get the stored keys. For Moon Lander we are going to store a single pair. The key is named “**Config**” and the value is a JSON oriented **Map** object. The Map object contains everything about the game. Go ahead and create a Map property for our configuration:

```
Map gameConfig;
```

Now lets open the store and get our single key/value pair. Note, Lawndart is **Future** based so you will see lots of "**then**"s, for example, *open()*.**then** do something.

```
    gameStore.open().then((_) {
      gameStore.getByKey("Config").then((Map value) {
        if (value == null) {
          print("No app configuration present. Defaulting to a preset.");
          gameConfig = _buildDefaultConfig();
        }
        else {
          print("App config present.");
          gameConfig = value;
        }
        _updateLoadStatus();
      });
    });
```

Notice above that the Map may not yet have been stored and if that is the case then we need to create a new Map for storage later. So how is the Map structured? Like this:

Code 9.15B

```
Map _buildDefaultConfig() {
  Map m = {
    "Music": false,
    "Sound": true,
    "Cloud": false,
    "Scores": {
      4: "Moo",
      3: "n L",
      2: "and",
      1: "er"
    }
  };

  return m;
}
```

We now have a Map either retrieved from Storage or a default Map created on the fly. Lets create some helper properties for updating these configuration values. For now we create just the three values exposed by the Settings dialog:

```
  bool get isMusicOn => gameConfig["Music"] as bool;
  set musicOn(bool b) {
    gameConfig["Music"] = b;
}

  bool get isSoundOn => gameConfig["Sound"] as bool;
  set soundOn(bool b) {
    gameConfig["Sound"] = b;
}

  bool get isCloudOn => gameConfig["Cloud"] as bool;
  set cloudOn(bool b) {
    gameConfig["Cloud"] = b;
}
```

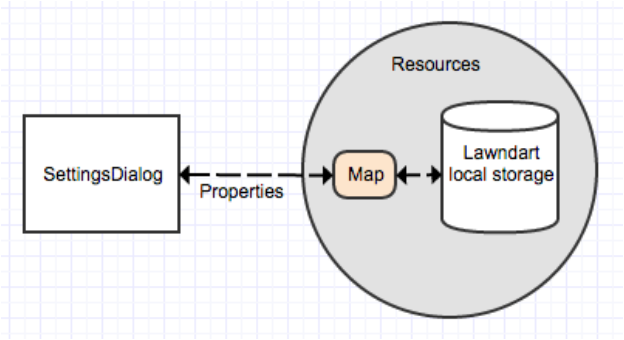


Figure 9.16 Storage flow

In Figure 9.16 you can see how the Map object sits between local-storage and the dialog. Any setting changes are stored in the Map and when the user clicks the close icon  we command the Resources to save the Map to local-storage. Is its the *save* method that will actually store the Map back into storage:

```
  void save() {
    _updateStore();
  }

  void _updateStore() {
    gameStore.save(gameConfig, "Config");
  }
```

And how about a *reset* method when the user wants to “nuke” it all and start over:

```
Future reset() {
  return _nukeStore();
}

Future _nukeStore() {
  _nukeWorker = new Completer();

  if (!gameStore.isOpen) {
    gameStore.open().then((_) {
      print("Store opened. Nuking game.");
      gameStore.nuke().then((_) {
        print("Game nuked.");
        gameConfig = _buildDefaultConfig();
        _nukeWorker.complete();
      }).catchError((Error e) => print(e));
    }).catchError((DomException e) => print(e));
  }
  else {
    gameStore.nuke().then((_) {
      print("Game nuked.");
      gameConfig = _buildDefaultConfig();
      _nukeWorker.complete();
    }).catchError((Error e) => print(e));
  }

  return _nukeWorker.future;
}
```

The reset works by using a **Completer**. Completers are part of Dart's asynchronous framework. A Completer allows us to create a **Future** and return it immediately. When the actual work is done (i.e. complete) we call the Completer's *complete* method to signal the code holding onto the Future that the Future has finished. Lets jump ahead a bit and look at the "other end" of this concept—the caller calling *reset*:

```
              if (md.choice == MessageData.YES) {
                gm.resources.reset().then((_) {
                  _settingsDialog.reset();
                });
              }
```

Notice how the *reset* method is called and takes the Future returned and uses the **then** method on the Future. This is typical Dart async code. When the Future completes the **then** is called where upon we call the dialog's *reset* method. We do this because we don't want to call the dialog until the resources have actually finished "resetting", otherwise we could end up updating the dialog with stale values.

Note

As of **Dart** 1.9 and greater Dart now has a more "natural" approach to async programing using **async**/**await**. See this Dart article on dartlang.org for greater detail: https://www.dartlang.org/articles/await-async/

Now we can integrate this with our Settings dialog while making a minor enhancement to our **ToggleButton** class to handle setting its value. Lets tweak the ToggleButton class first by adding a property called ***on***:

```
set on(bool b) {
  _state = b;
  if (_state) {
    _stateNode.positionX = _stateOnPosition;
    _stateNode.text = "On";
    _knob.positionX = _knobOnPosition;
  }
  else {
    _stateNode.positionX = _stateOffPosition;
    _stateNode.text = "Off";
    _knob.positionX = _knobOffPosition;
  }
}
```

The ***on*** property will position the knob and set the button's text value and position. If **True** then the knob is on the right otherwise Left. Now we can return to the **SettingsDialog** class's *show* method and set each button based on the value from storage:

```
void _loadValues() {
  _music.on = gm.resources.isMusicOn;
  _sounds.on = gm.resources.isSoundOn;
  _cloud.on = gm.resources.isCloudOn;
}

void show() {
 ...
 // Update widgets based on configuration
  _loadValues();
 ...
```

And while we are here lets add a *reset* method that can be called if the user decides to "nuke it" all:

```
void reset() {
  _loadValues();
}
```

With the *reset* method we can now go back to the MainScene class's *_listenToBus* method and update it to call the Resources *reset* method. Remember the Resource's *reset* method is

asynchronous which means we need to use the returned **Future**'s **then** method. Once the **then** is called we ripple this towards the SettingsDialog's *reset* method (repeated from above):

```
            if (md.choice == MessageData.YES) {
              gm.resources.reset().then((_) {
                _settingsDialog.reset();
              });
            }
```

You may have noticed a new property on the MessageData object above called ***choice***. That is set by the popup—for which we will be covering next—and allows us to tell which button was clicked.

Recap

That was quite a bit of code. Lets look at a round trip for a reset starting from the SettingsDialog.

1. First the pirate icon is clicked causing a SHOW-DIALOG-ConfirmReset message to be sent.
2. MainScene receives the message and shows the ResetPopupDialog.
3. The Accept button (aka "Nuke it") from the YesNoPopupDialog subclass is clicked.
4. The button sends a HIDE-DIALOG-ConfirmReset-YES message.
5. MainScene receives the message and hides the dialog then calls the Resource's *reset* method upon which it receives a Future.
6. The Future completes after having nuked local-storage.
7. This causes the **then** method to execute where upon the SettingsDialog's *reset* method is finally called.

Something to remember when showing and hiding dialogs is that you need to properly control focus of both the incoming and outgoing dialogs. Scenes and Dialogs have a *focus* method to control enablement and disablement of input. Only one object should have focus at anyone time otherwise you will certainly get unexpected results, for example multiple events being triggered.

Also, remember to set the autoInputs property to false on Dialogs that you can control focus yourself as only you know when dialogs will appears and disappear.

Chapter 10 Popups

As you have just seen in the previous chapter Ranger is quite flexible when designing and constructing dialogs. In this chapter we continue with simpler dialogs called popups.

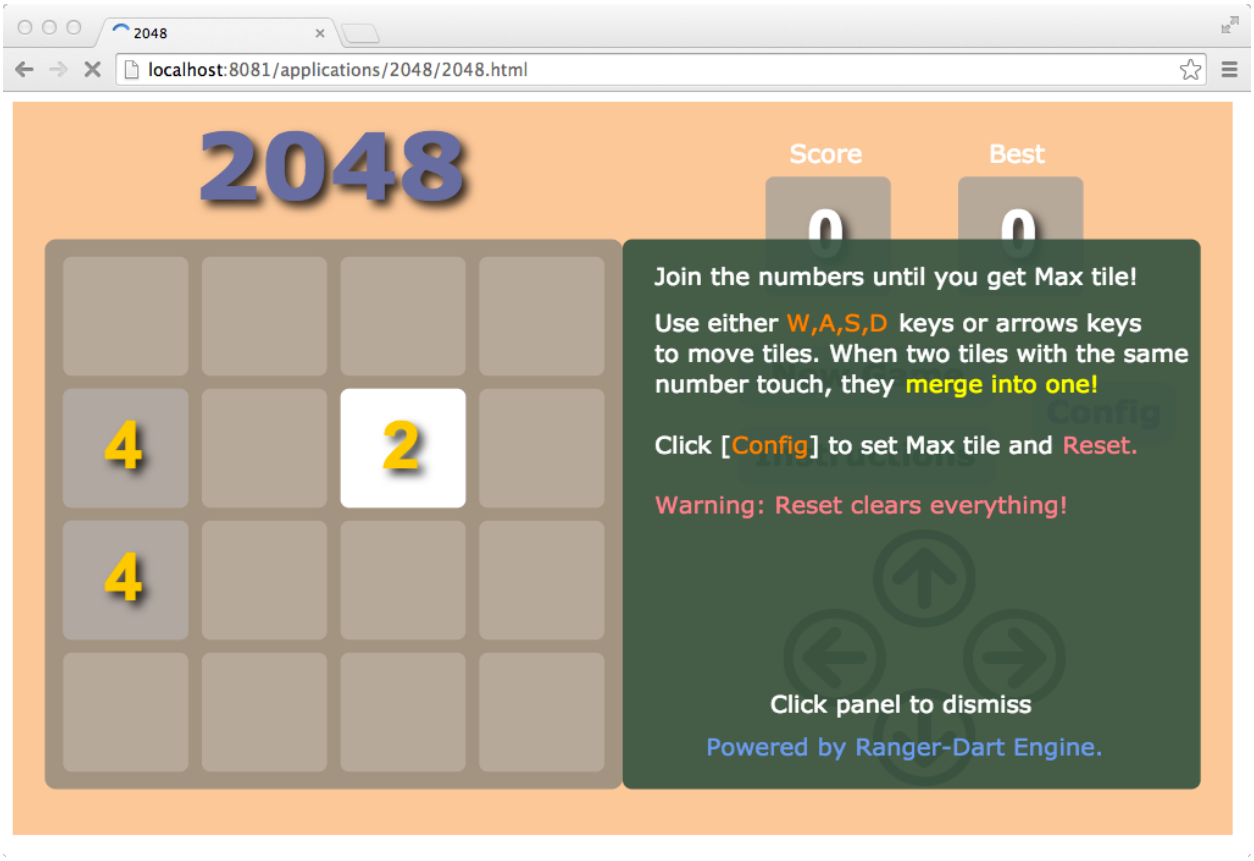


Figure 10.0 2048 Game Slide-out

In Figure 10.0 we see one of Ranger Sack's example games 2048. The green box with instructions is an example of a popup that actually slides out from underneath the main game grid as a mode of "popping".

Similarities

Popups and Dialogs have many similarities, for one they are both dialogs that interact with the user. I think of dialogs as more complex than popups. Popups tend to be simple dialogs with a very minimal set of GUI elements.

Figure 10.1 shows our final goal. The classes in green are the new classes we are going to build for our first popup dialog. But it is interesting to look at what we have so far.

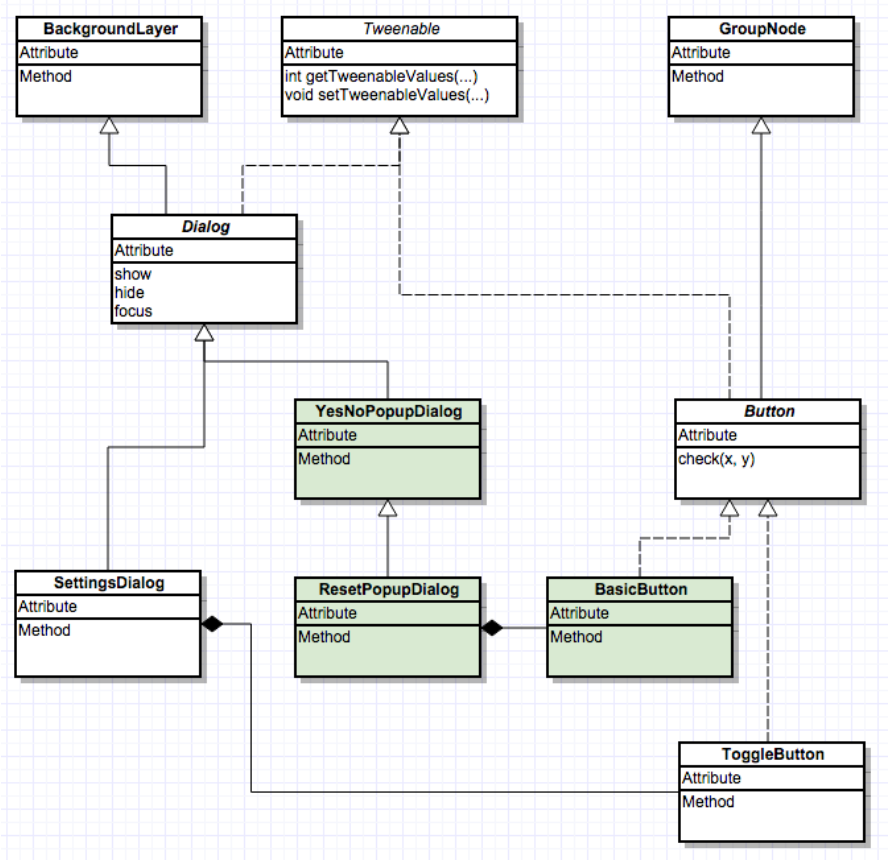


Figure 10.1 Dialog Model

We can see that the **SettingsDialog** is a **Dialog** and **BackgroundLayer** that also implements the ***Tweenable*** interface. The dialog also contains **ToggleButton**s of type **Button**.

Our popup dialog will also inherit from Dialog because our popup needs input while being animatable. So lets begin putting together our base popup dialog.

YesNoPopupDialog

As usual create a new file called *yesno_popup_dialog.dart* in the *game/dialog* folder that is structured as shown in Figure 10.1:

```
abstract class YesNoPopupDialog extends Dialog with UTE.Tweenable {...}
```

All YesNo dialogs need a background and there are two ways you can accomplish this, either override the BackgroundLayer's *drawBackground* method or supply some sort of Node. For our dialog we are going to reuse the **RoundRectangleNode** as it nicely meets our needs. So go ahead and add a RoundRectangleNode property.

Next we **override** the base *init* method so we capture a few values and set some flags. Because we are providing our own background we don't need the default background rendered. In order to save on wasted rendering we set the **transparentBackground** flag to **true** as is done below:

```
@override
bool init([int width, int height]) {
  if (super.init(width, height)) {
    _width = width.toDouble();
    _height = height.toDouble();

    transparentBackground = true;
    tag = 1001;

    _configure();
    _listenToBus();
  }

  return true;
}
```

Next up we override the *onEnter* method/event and set this dialog/node's visibility to false. This way when we first create the dialog and add it to the scene graph it doesn't appear immediately.

```
@override
void onEnter() {
  // For dialogs we don't enable inputs here for two reasons; one we
  // don't need inputs until the dialog is shown and two it would
  // cause duplicate enablement causing multiple events to be generated.
  super.onEnter();

  visible = false;
}
```

As the comments are indicating we don't want input enabled by default because we will control input from the *show*/*hide* methods. Now what about appearance? Where will the dialog appear, in a corner or perhaps the center? For our popup we want the dialog to appear in the center and to do that we need to add an extra Node to act like an anchor.

Anchors/Pivots

Remember when we talked about Coordinate Systems back in Chapter 5. By default the base coordinate system is in the lower-left with +Y upward. This means that a non-positioned Node is

at the lower left. Our dialog is a Node that it will appear in the lower-left corner which isn't what we want.

But we also have one more requirement and that is we want to "pop" the dialog into view by scaling from a small value to 1.0. We can do part of what we need by simply positioning the parent Dialog based on half the size of the design dimensions. But the dialog will still not be center aligned because, again, the default for a Node is the lower-left and what you end up with is the dialog's lower-left corner sitting in the center of the Scene, in other words the dialog will appear in the upper right quadrant. We want the dialog itself to be center aligned as well. And for that we need to introduce another GroupNode that acts as an *Anchor* where both position and scale can occur independently of the Dialog (see Figure 10.2.)

Using the Anchor the Dialog's position remains at the lower-left but we can translate and scale the Anchor anywhere we want relative to the Dialog.

By the way, the Anchor also doubles as a pivot point about which rotation and scaling occurs. This works well for us because we want the dialog to "pop" out from the center of the dialog not the lower-left.

With the Anchor concept in mind lets go ahead and add a GroupNode in our YesNoPopupDialog base class to act as the Anchor:

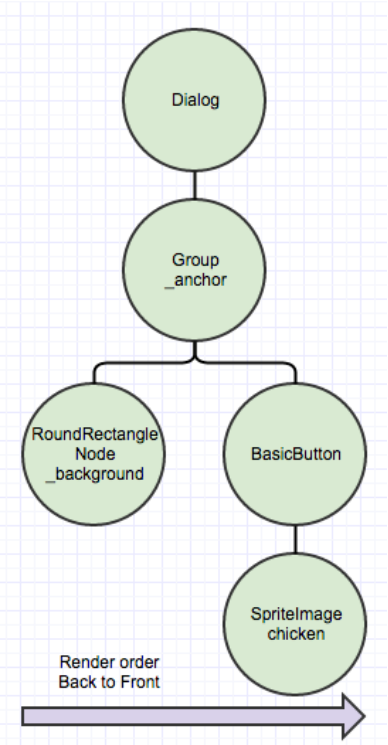


Figure 10.2 Dialog SG

```
Ranger.GroupNode _anchor;
```

With this anchor we can add our *_configure* method and center it—remember this will only center the dialog's lower-left at the view center:

```
  void _configure() {
    double dw = ranger.designSize.width;
    double dh = ranger.designSize.height;

    _anchor = new Ranger.GroupNode();

    addChild(_anchor);
    _anchor.setPosition(dw / 2.0, dh / 2.0);
...
```

It is the background Node that will be centered relative to the anchor, and this second positioning will finally center the dialog. This same center "offset" will be used for all subsequent child Nodes like buttons and text. So continuing with the *_configure* method we add the RoundRectangleNode and position it using a center offset and add it as a child to the anchor:

```
    double centerOffsetX = -_width / 2.0;
    double centerOffsetY = -_height / 2.0;

    _background = new RoundRectangleNode.basic(Ranger.color4IFromHex("#866761"))
      ..outlineColor = Ranger.color4IFromHex("B2FF59").toString()
      ..width = _width
      ..height = _height
      ..setPosition(centerOffsetX, centerOffsetY);
    _anchor.addChild(_background);
```

Okay, we have the underpinnings of a popup but we still need buttons: an accept/yes button and cancel/no button.

BasicButton

Most buttons differ by only an Icon and text which means our **BasicButton** class needs properties for a **SpriteImage** and **TextNode**. But all buttons are something you click on, so we need a base class **Button** that describes a basic feature set—in this case a *check* method:

```
abstract class Button extends Ranger.GroupNode with UTE.Tweenable {
  bool check(int x, int y);
}
```

Our BasicButton class will inherit from Button plus augment it with a RoundRectangleNode, SpriteImage and TextNode. We will also include plenty of properties for styling the appearance. The code is fairly large so I will just cover the most important aspects that make it a button. Of course remember to copy the book's source asset. We start off by adding the three main properties:

```
class BasicButton extends Button {
  RoundRectangleNode _background;
  Ranger.SpriteImage _icon;
  Ranger.TextNode _captionNode;
```

We also implement the base class *check* method by implementing a check for the background being clicked:

```
bool check(int x, int y) {
  Ranger.Vector2P nodeP = ranger.drawContext.mapViewToNode(_background, x, y);

  if (_background.pointInside(nodeP.v)) {
     nodeP.moveToPool();
     MessageData md = new MessageData();
     md.whatData = MessageData.BUTTON;
     md.actionData = MessageData.CLICKED;
     md.data = _caption;
     ranger.eventBus.fire(md);

    return true;
   }

   nodeP.moveToPool();

  return false;
 }
```

YesNoPopupDialog continued

Now that we have our shiny new buttons lets add them to our base dialog class.

```
  BasicButton _cancel;  // chicken
  BasicButton _accept;  // Nuke
```

After we create the new buttons we can **override** the *onMouseDown* event/method and begin checking for clicks. Here is the code for checking for the accept button including sending a message onto the EventBus:

```
  @override
  bool onMouseDown(MouseEvent event) {

    bool clicked = _cancel.check(event.offset.x, event.offset.y);
    if (clicked) {
      if (clicked) {
        MessageData md = new MessageData();
        md.actionData = MessageData.HIDE;
        md.whatData = MessageData.DIALOG;
        md.data = "ConfirmReset";
        md.choice = MessageData.NO;
        ranger.eventBus.fire(md);
        return true;
      }
    }
...
```

Again, the code is fairly large so we cover just the main concepts beginning with the *show* method. Of course don't forget bind the Nuke  and Chicken  icons to your buttons.

To begin with we want our popup to “pop and bounce in” from out-of-nowhere, remember when we added the Anchor well now that anchor really comes into play. First off we set focus to **true** because our dialog needs input:

```
void show() {
  // We enable mouse here instead of the onEnter method as we don't
  // want inputs enabled until the dialog is shown not when it is
  // created.
  focus(true);
```

Of course we can’t forget to set our visibility:

```
visible = true;
```

So what about that “popping out of nowhere” effect? Simple set the Anchor scale to something small and combine it with an animation that animates the Anchor’s scale back to 1.0. I find an Elastic-Out easing motion a pretty nice effect:

```
  _anchor.uniformScale = 0.25;

  ranger.animations.flush(this);

  UTE.Tween scaleUp = new UTE.Tween.to(this, TWEEN_SCALE, 1.0)
    ..targetValues = [1.0]
    ..easing = UTE.Elastic.OUT;
  ranger.animations.add(scaleUp);
  ranger.animations.track(this, TWEEN_SCALE);
```

It is one thing to configure and start an animation but who services the animation? Our dialog of course which is why the “this” object is passed above. Do you remember what the other half of the animation code would be? If you said Tweenable you would be correct, and guess what object they will interpret, you guessed it—the anchor:

```
int getTweenableValues(UTE.Tween tween, int tweenType, List<num> returnValues) {
  switch(tweenType) {
    case TWEEN_SCALE:
      returnValues[0] = _anchor.uniformScale;
      return 1;
  }
  return 0;
}

void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) {
  switch(tweenType) {
    case TWEEN_SCALE:
      _anchor.uniformScale = newValues[0];
      break;
  }
}
```

That takes care of the base class for popups. So lets augment this class to present itself as a reset dialog in the off chance that a user loses their mind and and wants to forget about their dismal landing record.

ResetPopupDialog

As Figure 10.1 shows our reset dialog will extend the **YesNoPopupDialog** class. It is pretty simple, we add a few lines of text and make sure our factory correctly set the autoInputs flag:

```
class ResetPopupDialog extends YesNoPopupDialog {
  Ranger.TextNode _line1;
  Ranger.TextNode _line2;
  Ranger.TextNode _line3;
  Ranger.TextNode _line4;
  Ranger.TextNode _line5;

  ResetPopupDialog();

  factory ResetPopupDialog.withSize([int width, int height]) {
    ResetPopupDialog layer = new ResetPopupDialog()
      ..autoInputs = false
      ..init(width, height);
    return layer;
  }
```

Other than the obligatory configure method we also provide a method for setting each line:

```
  void setMessage(String msg, int line, double scale, double px, double py,
Ranger.Color4<int> color) {
    Ranger.TextNode textNode;
    switch (line) {
      case 1: textNode = _line1; break;
      case 2: textNode = _line2; break;
      case 3: textNode = _line3; break;
      case 4: textNode = _line4; break;
      case 5: textNode = _line5; break;
    }

    textNode..text = msg
      ..uniformScale = scale
      ..color = color
      ..setPosition(px, py);
  }
```

MainScene revisited

With our new Popup dialog we can now integrate it into our MainScene class in the hopes a user is crazy enough to click the pirate.

Return to MainScene's *_listenToBus* method and add a block of code that recognizes the DIALOG-SHOW-Settings-ConfirmReset message transmitted by the SettingsDialog class. The

SettingsDialog should do two main things: set focus to **false** for itself and construct and show the reset popup:

```
            else if (md.data == "ConfirmReset") {
              _settingsDialog.focus(false);
              _resetConfirmDialog = new ResetPopupDialog.withSize((dw *
0.5).toInt(), (dh * 0.5).toInt());
              addChild(_resetConfirmDialog);
              _resetConfirmDialog..setMessage("Well now...", 1, 10.0, -400.0,
200.0, Ranger.Color4IOrange)
...
                ..acceptCaption = "Nuke it"
                ..cancelCaption = "Rethink it"
                ..show();
            }
```

We have finally reached a point where we can launch the popup by clicking on the pirate icon. Lets try it shall we (see Figure 10.3.)

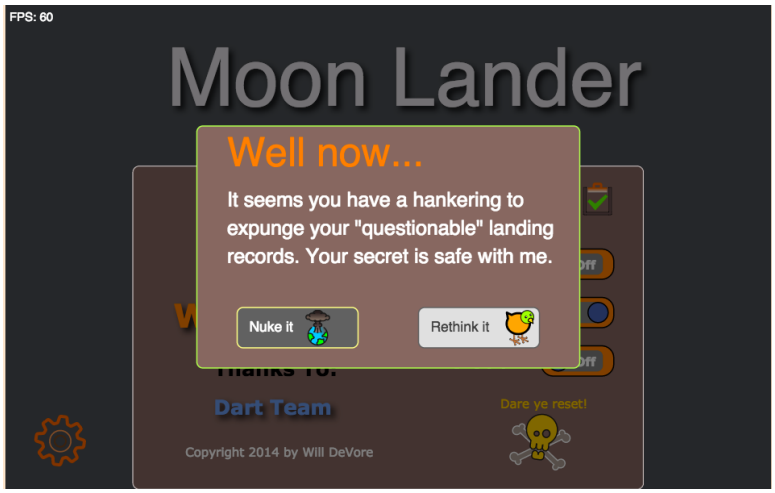


Figure 10.3 Reset popup

When you run it you should see our popup “spring” into action with a “boing”!

This is great but what about the buttons, what should happen when they click on one? We will start with the easy button first—“Rethink it”. For this button we simply send out a message, for which, again, MainScene listens for. So we add another **if**-block of code to handle hiding/dismissing the popup:

```
            else if (md.data == "ConfirmReset") {
              _resetConfirmDialog.hide();
              _settingsDialog.focus(true);
              removeChild(_resetConfirmDialog);
            }
```

Above we hide the popup, set focus back to the SettingsDialog and finally remove the popup from the scene graph. To handle the “Nuke it” button we simply add an additional check on the message to see if they clicked the Accept/Yes button:

```
            else if (md.data == "ConfirmReset") {
              _resetConfirmDialog.hide();
              _settingsDialog.focus(true);
              removeChild(_resetConfirmDialog);
              if (md.choice == MessageData.YES) {
                gm.resources.reset().then((_) {
                  _settingsDialog.reset();
                });
              }
            }
```

Once we detect the YES message we call our *reset* method on the **Resources** class. Again, notice the “**then**” method being used. Remember that Lawndart is Future based and the *reset* method was coded to return that **Future**. As mentioned earlier, when the Future completes our “**then**” statement is executed which in turns call the anonymous “(_)” method that tells the SettingsDialog to reset itself. Failure to use the “**then**” statement can leads to some weird updates on the SettingsDialog.

Recap

That was fairly straight forward. We built a base YesNo abstract class then augmented it with another class loaded with TextNodes with everything loosely coupled via the EventBus.

So now we have a functioning dialog and popup, but wouldn’t it be nice if there were sound effects to go along with it. I don’t know about you but I think it is time we have some fun with sounds.

Chapter 11 Audio Effects

Every decent game has at least some sort of audio associated with the GUI. Sound acts as an excellent feedback to users.

Figure 11.0 Ranger Sfxr

In Figure 11.0 we see one of Ranger Sack's auxiliary application called Ranger Sfxr (aka RSfxr). RSfxr is an adaptation of a port of jsfxr (http://github.grumdrig.com/jsfxr/).

Sfxr and RSfxr sounds

Ranger has two sound effect generators: Sfxr and RSfxr. Each generates sounds in completely different ways, and they also sound very different too.

Sfxr is a direct port of a javascript generator that it itself being a port of Dr. Petter's flash version (http://www.drpetter.se/project_sfxr.html).

> **Note**
>
> Because of javascript's poor design a number of errors exist in jsfxr. For example, divide by zero and out-of-bounds array errors. These were disappointingly discovered while porting to a more robust language such as Dart.

The sounds that Sfxr produces (and the javascript version) are more "crushed" as if generated with a maximum 8bits of resolution, and in fact they are when you look at the code's algorithm.

RSfxr took the spirit of Sfxr and converted it into a pure Web Audio implementation. The sounds tend to be more "pure" because they are produced at the default resolution of Web Audio. A later version of RSfxr may add an actual bit crusher node in order to reintroduce the classic sounds of 8bit audio. Nonetheless, the choice is your which style of sound you want to hear.

Having fun with RSfxr generator

To get an idea of sound effects how about we crank up Ranger's classic-arcade sound effects generator and smash a few buttons! The generator is part of Ranger Sack's application suite.

You can run it by navigating to *web/applications/ranger_rsfxr* and launching ranger_rsfxr.html in Dartium. When it first comes up you won't hear anything because by default the micro mixer has all the channels disabled. Enable the first channel 

and enable the Frequency section 

. Immediately you will hear the default tone of 340 Hertz. Yeah! Your first sound effect. Lets step it up a bit and tryout the various preset generators. The most familiar sound is probably Mario's coin pickup. By clicking on the 

button you should hear the familiar but classic platform coin pickup sound. Sweet! Each time you click the button a slight variation is applied to the parameters; try it and pay attention to the sliders on the right. If you were paying extra special attention you may have noticed that only the Frequency and Arpeggiation were changed. This is because the base characteristics of the coin pickup is nothing more than a sine wave frequency and a few stepping notes.

Try out 

and note which parameters change. Did you noticed that Lasers are pretty much a sawtooth frequency slide with a bit of slide-acceleration combined with variations in the duty cycle of the sawtooth.

As you go through each generator style you may also notice that certain parameters are either enabled or disabled. This is because some generators either lack such a concept or don't really have a noticeable effect. For example, 

doesn't have a Duty Cycle because it's just noise. However, some parameters default to disabled but can be reenabled in case you really want to try it anyway. The Explosion generator is a good example of adding a Tremolo effect which can really have a dramatic impact on the sound.

If you jump all the way down to 

generator you will notice that the Retrigger section kicks in. High frequency alarms are typically an arpeggiation pingpong style combined with multiple triggers firing rapidly several times in a row.

The generators are good starting points. They help configure the sliders to match the given generator style. But that is all they do. It is up to you to “tweak” the sliders to form your own effects, and considering you have a mixer available the possibilities are almost endless.

Go ahead and have some fun playing around with the various generators. In the meantime you may be wondering how it actually works. In the case of RSfxr it is something called Web Audio

Web Audio

Web Audio is a W3C standard:

> Quote
>
> …The primary paradigm is of an audio routing graph, where a number of AudioNode objects are connected together to define the overall audio rendering.

RSfxr is literally a Web Audio application all the way to the core, and in the case of RSfxr what this means is that sound is generated on the fly and not played back from a sampled source like an ogg file.

The app came about as an exercise in learning Web Audio. After I had ported the javascript sounds effects generator (https://github.com/grumdrig/jsfxr) to Dart I realized that it should be possible to create an equivalent Web Audio variation, and that is what RSfxr is. However, RSfxr is more than that, as I was converting various components over to their Web Audio counter part I discovered I could extend and even enhance the original generator.

If your’e curious about how the components are connected look at Figure 10.1 (the image is also located with the book’s assets (WebAudioSoundEffects.png).

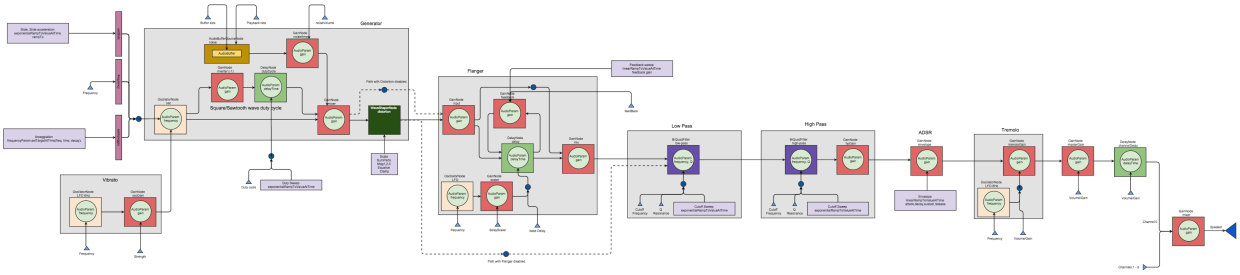


Figure 11.1 Web Audio components

Pretty crazy looking right? Not really once you’ve played around with Web Audio for about a week. After you stare at it for a few hours you may begin to notice that there are seven main areas all of which are exposed as buttons and sliders in RSfxr’s GUI including all of the input controls. In addition, RSfxr includes a ten channel micro mixer for an added bonus.

Well all this “sounds” cool but how do you save your awesome sound effect?

Saving a sound

Figure 11.2 Output

At the moment RSfxr isn't a chrome app nor does it have GDrive integration and as such it can't directly save to a file. So for the time being it's a simple, but manual, process of copying JSON and pasting into a text file.

Before you dump the settings give your effect a name in the Output field (see Figure 11.2.) This name will be embedded into the JSON such that you can later reference it in your game.

Once you have your awesome sound effect click the "Dump Settings" button (see Figure 11.2.) This pastes, and highlights, JSON data into the output area. Then just Ctrl-C or Cmd-C to copy the highlighted JSON text into the clipboard.

Now open your favorite text editor, paste the clipboard contents, then save it to the assets folder of your game. It doesn't matter what the name is but I standardized on a file extension of

".rsfxr" to indicate that the file is a RSfxr JSON file.

The JSON data represents RSfxr's components and control relationships. Remember, RSfxr generates sounds on the fly which means there isn't any recording to ogg vorbis buffers or files.

Opening a sound

The good thing is that browsers do allow opening files. Just click the "Open from desktop" button and navigate to the folder where your .rsfxr file is. Because RSfxr doesn't have GDrive support integrated directly you will need to install Google's GDrive first in order to open files off of the internet.

This is all well and good, but how about we start creating sounds for Moon Lander.

Creating Slide-in/Slide-out sounds

We start with the SettingsDialog Slide-in sound. Here are the steps I used to create the sound:

- Start Sfxr fresh
- Enable the first channel
- Enable Frequency and set it to around ~434Hz. Leave the gains untouched.
- Enable High-Pass Filter
- Set the high-pass cutoff frequency to ~1400Hz
- Set the cutoff sweep to ~425Hz
- Set the resonance to ~12%
- Set the core signal to "Pink Noise" with a playback rate of ~2.0
- Set the noise override to about 0.4
- Set the Envelope ADSR to:
  - Attack to ~0.02
  - Decay to ~0.5

- Sustain ~0.35
- Release ~0.003

What this produces is a soft noise that last for about 1 second. The high-pass filter combined with the sweep restricts the noise above a certain threshold and then abruptly clamps it. The clamp is emphasized even more by the ADSR's release.

To create the Slide-out sound I simply bumped the frequency up about 10Hz or so.

Remember add a name to the Output field and then save each sound in a separate file.

Copy and paste the JSON configuration into a text file and save it as *SlideIn.rsfxr* and *SlideOut.rsfxr*.

Creating the Button sound

I chose a button click effect that sounds like a very short bell. Here are the steps I used to create the sound:

- Start RSfxr fresh
- Enable the first channel
- Enable Frequency and set it to around ~897Hz. Leave the gains untouched.
- Enable Low-Pass Filter
- Set the cutoff frequency to ~1400Hz
- Set the cutoff sweep to ~34Hz
- Set the resonance to 0%
- Set the core signal to "Square"
- Set the Envelope ADSR to:
  - Attack to 0.0
  - Decay to ~0.03
  - Sustain ~0.3
  - Release 0.0

This produces is a short but tight bell sound with a slight positive feel. Copy and paste the JSON configuration into a text file and save it as *Click.rsfxr*.

Creating the Toggle sound

For this sound I mixed it up a bit by adding a re-trigger with a longer trigger rate. Here are the steps I used to create the sound:

- Start RSfxr fresh
- Enable the first channel
- Enable Frequency and set it to around ~405Hz. Leave the gains untouched.
- Set the frequency Slide to -0.130
- Set the frequency Slide acceleration to 0.109
- Enable Low-Pass Filter
- Set the lowpass cutoff frequency to ~5800Hz
- Set the lowpass cutoff sweep to ~170Hz
- Set the resonance to 7%
- Set the core signal to "Triangle"

- Set Retrigger to 2 with a Rate of ~0.147
- Set the Envelope ADSR to:
  - Attack to 0.137
  - Decay to ~0.217
  - Sustain 0.0
  - Release ~0.083

Copy and paste the JSON configuration into a text file and save it as *Toggle.rsfxr*.

Creating a Popup sound

For this effect I want a sound that “jiggles” with the popup’s elastic visual. Here are the steps I used to create the sound:

- Start RSfxr fresh
- Enable the first channel
- Enable Frequency and set it to around ~405Hz. Leave the gains untouched.
- Set the frequency Slide to -0.130
- Set the frequency Slide acceleration to 0.109
- Enable Low-Pass Filter
- Set the lowpass cutoff frequency to ~5800Hz
- Set the lowpass cutoff sweep to ~170Hz
- Set the resonance to 7%
- Set the core signal to “Triangle”
- Set Retrigger to 2 with a Rate of ~0.147
- Enable Vibrato and select Square
- Set Depth to ~55
- Set Speed to ~10
- Set the Envelope ADSR to:
  - Attack to 0.137
  - Decay to ~0.217
  - Sustain 0.0
  - Release ~0.083

It is the Vibrato component that really gives this effect its true characteristics. Copy and paste the JSON configuration into a text file and save it as *Popup.rsfxr*.

Now that we have some sound effects how do we go about adding them to Moon Lander?

Integrating our sound effects

Our first step in integrating our sounds is to return to the **Resources** class and load them into **Map**s. We begin by defining properties for each sound:

```
Map slideInSound;
Map slideOutSound;
Map toggleSound;
Map popupSound;
Map clickSound;
```

Before we can load our Maps we need to add an **import** for Dart's convert package. Jump over to *main.dart* and add the required import:

```
import 'dart:convert';
```

Now that we have the convert package referenced we can return to the Resources *load* method and retrieve the JSON files. In this case we are going to use the **HttpRequest** object to simplify our work, but we still need to increment the resource count to track loading status. Here is the ***slideInSound*** being fetched:

```
    _resourceTotal += 1;    // sound effect
    HttpRequest.getString("resources/SlideIn.rsfxr").then((sfxr){
      slideInSound = JSON.decode(sfxr);
      _updateLoadStatus();
    });
```

There is that "**then**" statement again. You guessed it, the *getString* method returns a **Future**. Once it has finished pulling the resource from the server we immediately decode it into a Map and update the load status. We do this for each *.rsfxr* resource we have created. But how do we play them?

AudioEffects class

The AudioEffects class provides functionality for both RSfxr files or Sfxr files. As Figure 11.3 shows it can contain and play either type.

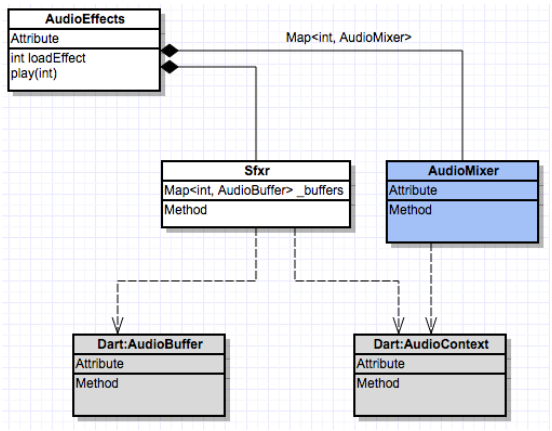


Figure 11.3 AudioEffect

For our Moon Lander game the GameManager will contain the AudioEffects object. Open the GameManager class and add an AudioEffects property:

```
Ranger.AudioEffects audioEffects;
```

However, before we can create it using one of AudioEffect's factories we need to return to *main.dart* and once again import another package, this one called Web Audio:

```
import 'dart:web_audio';
```

Now we can create the AudioEffects object by returning to the GameManager's *postLoad*:

```
audioEffects = new Ranger.AudioEffects.basic(new AudioContext());
```

This *postLoad* method is called once all the resources have been loaded which means we can also preload all the sounds effects into the AudioEffects object too:

```
    slideInSoundId = audioEffects.loadEffectByMap(resources.slideInSound);
    slideOutSoundId = audioEffects.loadEffectByMap(resources.slideOutSound);
    popupSoundId = audioEffects.loadEffectByMap(resources.popupSound);
    toggleSoundId = audioEffects.loadEffectByMap(resources.toggleSound);
    clickSoundId = audioEffects.loadEffectByMap(resources.clickSound);
```

Each *loadEffectByMap* call returns an Id. You squirrel this Id away for later use when you want to play the sound effect, however, we need to add one more method to the GameManager to provide a central location for playing sounds effect—*only*—if sound is enabled:

```
  void playSound(int id) {
    if (resources.isSoundOn)
      audioEffects.play(id);
  }
```

Return to MainScene's *_listenToBus* and add the following line of code just after the cascading **if** statements that check for the SHOW message targeting the SettingsDialog:

```
      switch(md.whatData) {
        case MessageData.DIALOG:
          if (md.actionData == MessageData.SHOW) {
            if (md.data == "Settings") {
              gm.playSound(gm.slideInSoundId)
...
```

Lets give it a try! Run Moon Lander and click on the gear icon. Sweet! Our slide in sound plays as the dialog slides in. Well that wasn't to hard. How about we go through and sprinkle *play* calls in all the places necessary. For starters add the corresponding slide out sound to the matching HIDE message a few lines down:

```
          else if (md.actionData == MessageData.HIDE) {
            if (md.data == "Settings") {
              gm.playSound(gm.slideOutSoundId);
...
```

A few lines further back up add the popup sound for the reset dialog:

```
            else if (md.data == "ConfirmReset") {
              gm.playSound(gm.popupSoundId);
...
```

Head over to SettingsDialog's *onMouseDown* method and refactor the code a bit to check for a click on the toggle buttons:

```
    bool clicked = _music.check(event.offset.x, event.offset.y);
    if (clicked)
        gm.playSound(gm.toggleSoundId);
...
```

Finally visit the YesNoPopupDialog's *onMouseDown* method and add *play* calls for both buttons:

```
    bool clicked = _cancel.check(event.offset.x, event.offset.y);
    if (clicked) {
      if (clicked) {
        gm.playSound(gm.clickSoundId);
...
```

Now our game is wired up for sound effects! Lets Recap.

Recap

First we created our sound effects with the Ranger RSfxr application.

- As a starter we used some of the preset generator styles then tweaked the controls to get precisely what we wanted.
- We copied the .rsfxr files into our resources folder.
- Updated the Resources class to load and decode the RSfxr JSON data into Maps.
- We then loaded each Map into the AudioEffects object while collecting the Ids for later play.
- Finally we traversed the code sprinkling code to call the *playSound* method.

Now that we have sound effects cover how about we return to physics, after all our Lander needs somewhere to land right?

Chapter 12 Physics

Our Lander needs a place to land!

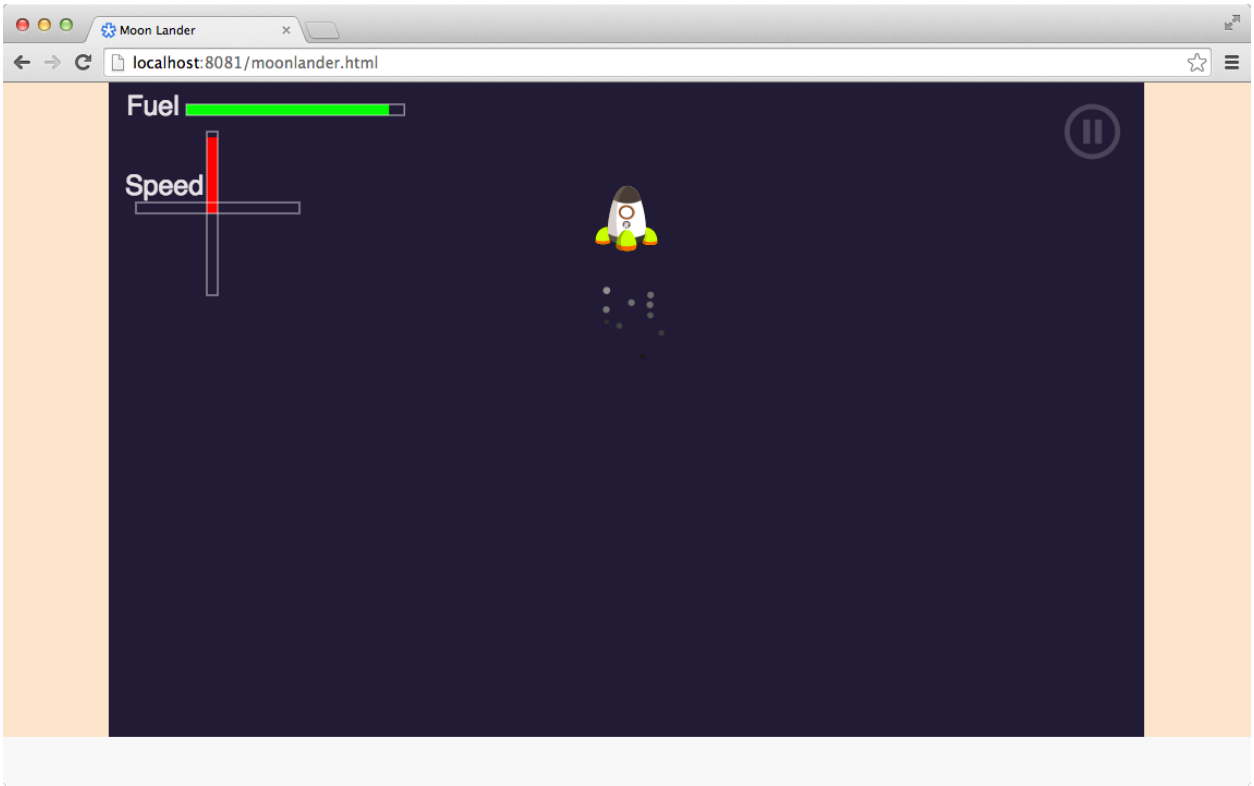


Figure 12.0 Moon Lander doomed

In this chapter we put together a simple horizontal plane for the lander to land on. Later chapters will cover coding a terrain builder. The chapter won't be introducing any new Ranger concepts but we will have fun!

Goal

Our mission (should we chose to accept it ;-) is to land a Lander on an orange colored rectangle:

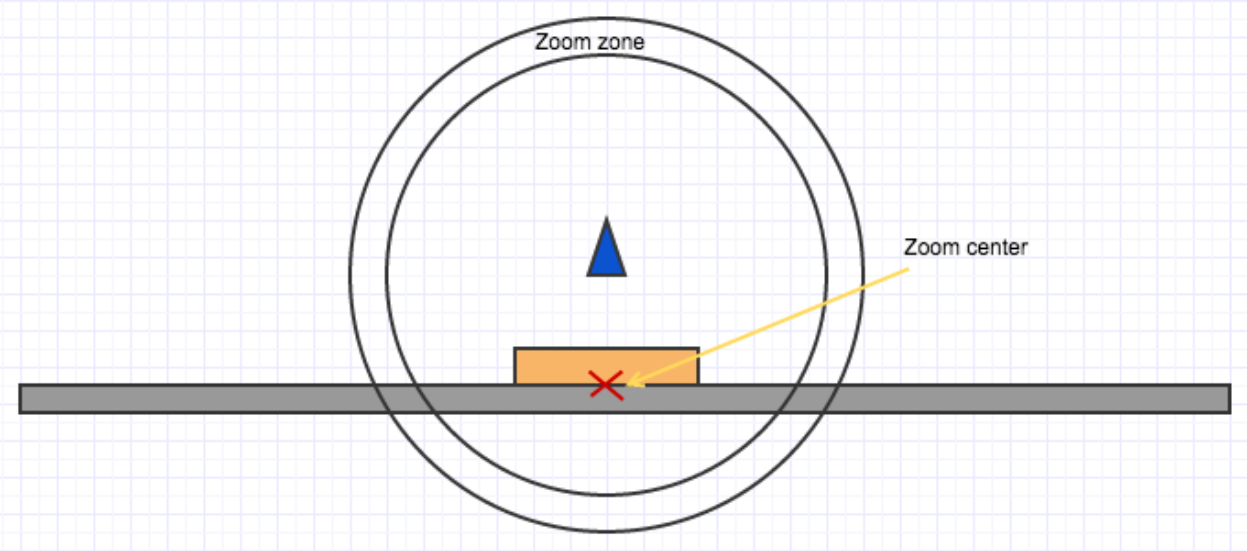


Figure 12.1 Zooming

When the lander enters an invisible zone we zoom in for better visibility of the orange landing pad. To accomplish all this we are going to need simple physics, zooming, zones and Nodes. Some of these items (zooming and zones) can be borrowed from Ranger Sack as that is what Sack is for. Others will be new code.

Setup

We begin by introducing the landing pad. We need two rectangles, one is a visual plane for reference and the other is a landing pad. Head over to LevelRimbaloidLayer class and add properties for the plane and pad:

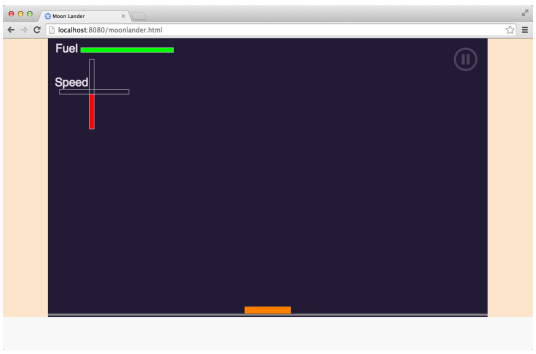
```
RectangleNode _plane;
RectangleNode _pad;
```

Now update the *_configure* method to create and configure their locations such that they appear at the bottom (Code 12.2.)

Code 12.2

```
double h = contentSize.height / 2.0;
double w = contentSize.width / 2.0;
double yOff = -h + 10.0;

_plane = new RectangleNode.basic(Ranger.Color4IGrey)
  ..positionY = yOff
  ..scaleX = contentSize.width
  ..scaleY = 10.0;
addChild(_plane);

_pad = new RectangleNode.basic(Ranger.Color4IOrange)
  ..positionY = yOff + 15.0 + _plane.scaleY / 2.0
  ..scaleX = 200.0
  ..scaleY = 30.0;
addChild(_pad);
```

If you run Moon Lander now you should see the plane and landing pad at the bottom:



Recall that one of our goals is to have the lander start very high up and then begin descending. However, with the lander high up, the landing pad starts to look very tiny making it difficult to locate. It would be nice if as the lander approached the pad the view zoomed in on the pad. Well to do that we need to introduce a new feature—zooming.

Zoom zoom

Zooming isn't really part of Ranger it is actually a Node, called **ZoomGroup**, available as part of Sack. To use it we just integrate it with our game, and a good example of how to do that can be found in one of Sack's demo applications called RangerRocket. But for brevity I will cover it here because it is a very handy Node.

ZoomGroup works by managing its own Affine *transform* matrix through accumulation, and because it inherits from GroupNode and Tweenable (see Figure 12.3 left side) it can also collect other Nodes and be animated. The animation is handy if you want to animate the centering of a Node or zoom in and out in animation style.

In the current code base for **LevelRimbaloidLayer** the Layer itself contains the Lander and Pad. In order to use ZoomGroup we need to refactor the layer such that ZoomGroup becomes the parent to all Nodes of the Layer (see Figure 12.3 right side.)

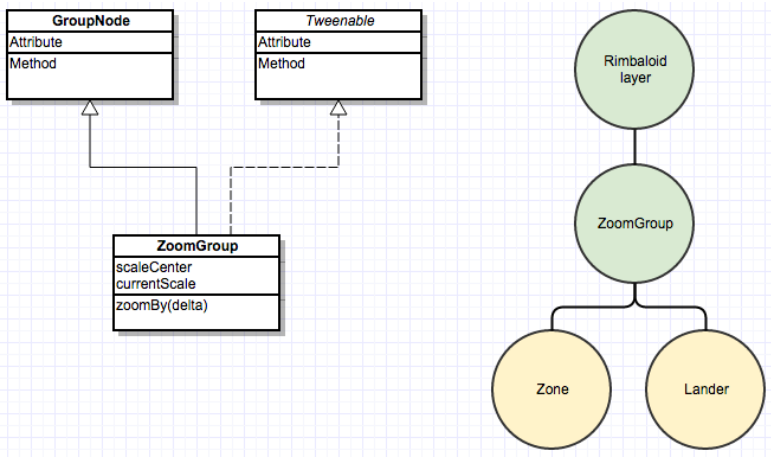


Figure 12.3 Model and Graph

Go ahead and copy the ZoomGroup class from RocketRanger into our *nodes* folder and update *main.dart* to include the class using the **part** statement:

```
part 'scenes/game/nodes/zoom_group.dart';
```

Now add add a ZoomGroup property to the LevelRimbaloidLayer:

```
ZoomGroup _zoomGroup;
```

With the ***_zoomGroup*** property we can refactor the *_configure* method to comply with Figure 12.3. First we create and add the ZoomGroup as a direct child of the layer:

```
    _zoomGroup = new ZoomGroup.basic();
    addChild(_zoomGroup);
```

Now we go through and re-parent the Lander, plane and pad to be children of ZoomGroup. Below is how the Lander is re-parented:

```
_zoomGroup.addChild(gm.triEngineRocket.node);
```

When you run Moon Lander now you will see that nothing has changed visually. Lets test out the zooming by temporarily setting the zoom scale to say 1.6 and enabling the zoom icon:

```
    _zoomGroup.zoomIconVisible = true;
    _zoomGroup.iconScale = 50.0;
    _zoomGroup.zoomBy(1.6);   // <-- This
```

When you run Moon Lander now you should see a white crosshair in the middle of the view. In a second or two the lander will appear as it descends into view from the top. Nice! We now have zooming hooked up!

Now that we have zooming integrated lets adjust the zoom center and place it where it needs to be—on the landing pad. Set the zoom center (aka scale center) just after the yOff variable is defined (reference Code 12.2):

```
    _zoomGroup.scaleCenter.setValues(0.0, yOff);
    _zoomGroup.zoomBy(1.6);    // <- move to here
```

This will keep the view horizontally centered as before but shift the center vertically downward right at the bottom of the pad. Also, note that I "moved" the *zoomBy* call to below the *scaleCenter* call. This is because the matrix calculations are order dependent.

However, if you run Moon Lander now you will notice that the lander's exhaust particles are scaled to match the zoom value. This is because the particles are still being emitted on the BackgroundLayer (Rimbaloid layer). We need to move them to the ZoomGroup Node, but in order to do that we need to refactor the **TriEngineRocket** class a bit.

TriEngineRocket is currently coded to accept only a BackgroundLayer object:

```
Ranger.BackgroundLayer _particleEmissionSpace;
```

We have two choices: either we change the property type to **ZoomGroup** or we change it to the **GroupingBehavior** type. Each has its pluses and minuses. Even though passing in a GroupingBehavior is convenient it would force us to type cast on every particle emission call and the *_convertToEmissionSpace* call would end up looking like this:

```
    if (_particleEmissionSpace is Ranger.BackgroundLayer) {
      Ranger.BackgroundLayer b = _particleEmissionSpace as Ranger.BackgroundLayer;
      Ranger.Vector2P ws = _centroid.convertToWorldSpace(location, b);
      ns = b.convertWorldToNodeSpace(ws.v, b);
      ws.moveToPool();
    }
    else if (_particleEmissionSpace is Ranger.GroupNode) {
      Ranger.GroupNode b = _particleEmissionSpace as Ranger.GroupNode;
      // We can't use a psuedo node because the GroupNode most likely
      // is a ZoomGroup type.
      Ranger.Vector2P ws = _centroid.convertToWorldSpace(location);
      ns = b.convertWorldToNodeSpace(ws.v);
      ws.moveToPool();
    }
```

This isn't a good approach for two reasons: type casting is a last resort and is expensive. Instead we are going to change the property to ZoomGroup:

```
ZoomGroup _particleEmissionSpace;
```

Now refactor _convertToEmissionSpace, but notice we no longer use a pseudo root as before:

```
  Vector2 _convertToEmissionSpace(Vector2 location) {
    Ranger.Vector2P ws = _centroid.convertToWorldSpace(location);
    Ranger.Vector2P ns = _particleEmissionSpace.convertWorldToNodeSpace(ws.v);

    // Clean up.
    ns.moveToPool();
    ws.moveToPool();

    return ns.v;
  }
```

This is because the ZoomGroup maintains its own matrix, however, if we wanted we could pass in the layer and use that as a pseudo root but for brevity it was skipped. Remember, pseudo roots save matrix multiplications. Without specifying a pseudo root the conversion methods have to walk all the way up to the scene root which could be wasted effort if all of them are the Identity matrix.

Okay, things are looking good. For now return to the Rimbaloid layer and replace the *zoomBy* method with a direct scaling so we are be "zoom out" be default, and move the Lander a little farther up so it has more room to descend:

```
    gm.triEngineRocket.node..uniformScale = 0.5
      ..setPosition(0.0, 500.0);
...
_zoomGroup.currentScale = 0.5;
```

Now how do we know when to zoom? There are at least two ways: track the distance from the lander to the pad and as soon as the distance is less than some predetermined value you start zooming. The other way is to use something I call zones.

Zones

Zones are another Node available from Sack—if you haven't noticed by now Sack is an excellent repository of useful code for a wide range of applications and games. For Moon Lander we are going to use an object called **DualRangeZone** (see Figure 12.4 B.) It works by maintaining enter and exit states on two rings. If a given (x,y) coordinate is between the rings

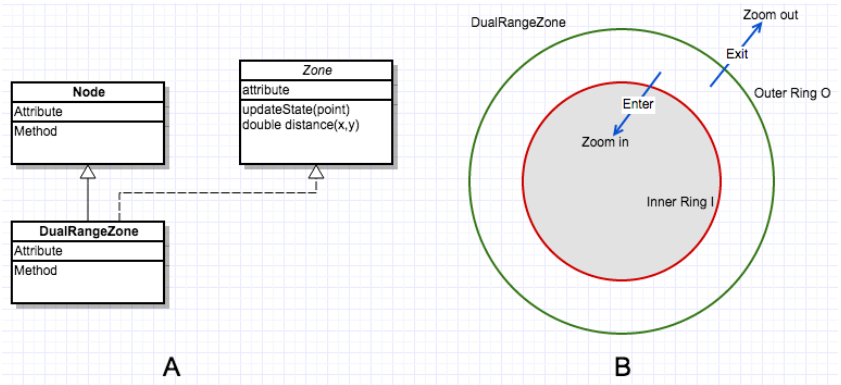


Figure 12.4 DualRangeZone

then the zone sees no change. If the coordinate “leaves” the outer ring then the zone changes state to “exited”. If the coordinate had instead moved inside the inner ring then the zone would change to an “entered” state. What is the purpose of this you ask—stability.

If for example we had a zone that comprised of only one ring then when an object is “hovering” around the ring’s edge you would get a stream of enter/exit events which can be very irritating in most situations, especially zooming. A dual ring arrangement reduces this erratic behavior considerably.

How do we know when an object has entered or exited the zone? Easy, just listen on the EventBus. The zone is coded to transmit either a DualRangeZone.*ZONE_INWARD_ACTION* or *ZONE_OUTWARD_ACTION*.

In Moon Lander our zone will control when zooming occurs, and if we place the zone correctly the user will get a nice zoomed in view of the pad prior to entering the pad’s range. Go ahead and copy both the **Zone** and **DualRangeZone** class from Sack’s *application/rocket/game/nodes* folder into our *nodes* folder, and make sure you update *main.dart* to include them using the **part** statement.

Because DualRangeZone is a Node, in and of itself, we can add it directly to our layer (aka ***_zoomGroup***). As usual, first we add a property to the **LevelRimbaloidLayer** class:

```
DualRangeZone _zone;
```

Next, add code to create and configure the zone:

```
    _zone = new DualRangeZone.initWith(Ranger.color4IFromHex("#ffaa00"),
Ranger.color4IFromHex("#ffaa77"), 400.0, 600.0)
      ..positionX = 0.0
      ..positionY = yOff + 450.0
      ..iconsVisible = true
      ..zoneId = 1;
    _zoomGroup.addChild(_zone);
```

The positioning values place the zone in such a location that when the lander enters the inner-ring zooming begins without “pushing” the lander out of view as a result of the zoom. What about the events? For that we add code to listen for an EventBus message transmitted by the zone:

```
    ranger.eventBus.on(DualRangeZone).listen(
    (DualRangeZone zone) {
      _dualRangeZoneAction(zone);
    });
```

The *_dualRangeZoneAction* method will handle the message by inspecting the ***action*** property on the zone:

```
  void _dualRangeZoneAction(DualRangeZone zone) {
    double zoom = 1.0;

    switch (zone.zoneId) {
      case 1:
        zoom = 1.6;
      break;
    }

    ranger.animations.stopAndUntrack(_zoomGroup, ZoomGroup.TWEEN_SCALE);

    switch (zone.action) {
      case DualRangeZone.ZONE_INWARD_ACTION:
          UTE.Tween tw = new UTE.Tween.to(_zoomGroup, ZoomGroup.TWEEN_SCALE, 2.0)
            ..targetValues = [zoom]
            ..easing = UTE.Sine.INOUT;
          ranger.animations.add(tw);
          ranger.animations.track(_zoomGroup, ZoomGroup.TWEEN_SCALE);
        break;
      case DualRangeZone.ZONE_OUTWARD_ACTION:
          UTE.Tween tw = new UTE.Tween.to(_zoomGroup, ZoomGroup.TWEEN_SCALE, 2.0)
            ..targetValues = [widePerspective]
            ..easing = UTE.Sine.INOUT;
          ranger.animations.add(tw);
          ranger.animations.track(_zoomGroup, ZoomGroup.TWEEN_SCALE);
        break;
    }
  }
```

Notice that I pass the ***_zoomGroup*** Node as a target for the animation, this is because ZoomGroup is animatable because it implements the **Tweenable** interface. If you peruse ZoomGroup's code you will see that you can animate both the scale and position:

```
class ZoomGroup extends Ranger.GroupNode with UTE.Tweenable {
  static const int TWEEN_SCALE = 1;
  static const int TRANSLATE_XY = 3;
...
```

However, we only need to animate the scale, hence we call **Tween**.*to* passing TWEEN_SCALE.

If you run Moon Lander at this point nothing will happen when the lander descends past the inner-ring. Why? Because the zone needs to be updated in order to recognize a state change. Do you remember how to get "updates" routed to you layer? Correct, by using the scheduler. Because this is a common requirement, Layers have a convenience method for doing just that called *scheduleUpdate*. A good place to call it is in the *onEnter* event:

```
scheduleUpdate();
```

Even though we scheduled our layer for updates our the Rimbaloid layer the Zone will still won't receive them. To complete the code we need to **override** the *update* method and place our zone update code inside. Notice that I pass the Lander's position to the zone's *updateState*:

```
@override
void update(double dt) {
  _zone.updateState(gm.triEngineRocket.node.position);
}
```

Now if you run Moon Lander you will see that as the lander passes the inner-ring the view zooms in around the landing pad, Sweet!

However, there is one little problem—what if the user aborts and begins to ascend back upwards? Well, for a moment the lander disappears out of view at the top of the screen until it passes out of the outer-ring where upon the view zooms back out to reveal the lander ascending. We need to correct this, and there are several ways to do this, but for this chapter we will settle on simply adjusting the zone rings until they are small enough and their displacement from each other is such that the ship stays within view.

These are the values I used to shrink the zone (marked in red):

```
  _zone = new DualRangeZone.initWith(
      Ranger.color4IFromHex("#ffaa00"),
      Ranger.color4IFromHex("#ffaa77"),
      200.0, 300.0)
    ..positionX = 0.0
    ..positionY = yOff + 250.0
```

With these values the lander stays in view, even during ascending. Nice, but we still can't land. Oops. Guess what the next task is? If you said collision you would be correct.

Bounding boxes

Moon Lander's collision will be based off of bounding boxes (bbox) or more specifically axis aligned bounding boxes (aabbox) (see Figure 12.5.) In Figure 12.5 A the bboxes are the same because the triangle has no rotation. In B the bboxes are different due to a rotation.

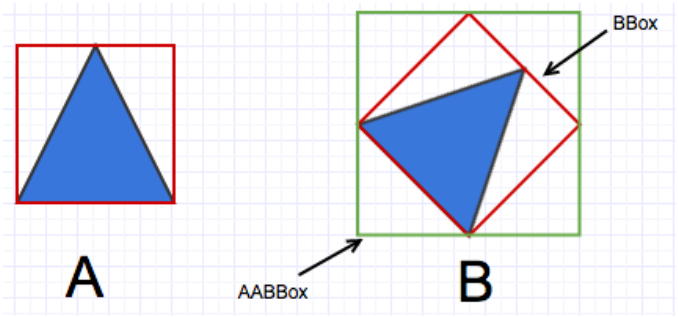


Figure 12.5 Bounding boxes.

Our lander is made up of both a SpriteImage and two landing gear assemblies. We could check each landing gear's legs and toes but for brevity we are just going to check the lander's

legs and body instead. When the lander's aabbox touches the pad's aabbox then we look at the current status of the lander to determine if the landing was successful.

So how do we form the lander's aabbox?

Axis aligned bounding box

Our goal is to detect when the lander has "touched" the orange landing pad. For that we are going to use axis aligned bounding boxes this means we need to form them based on the orientation of the lander. Each Node is unique and has different ways of representing its local Bounding Box (bbox). The SpriteImage can either return a VectorMath Aabb2 object or a mutable rectangle. The rectangle is a better match for us as you will see in a moment.

Before we get started lets disable gravity so we don't have to hold the thrust button while we inspect our work, and zoom in a bit for better visibility. Return to TriEngineRocket's *update* method and comment out the line:

```
//_momentum.add(gm.gravity);
```

Now adjust the lander position to -400.0:

```
gm.triEngineRocket.node..uniformScale = 0.5
  ..setPosition(0.0, -400.0);
```

and zoom to 2.0:

```
_zoomGroup.currentScale = 2.0;//widePerspective;
```

You should be able to see the lander quite clearly now. The first aabbox we are going to create is the leg portion of the landing gear. We are only doing the leg because the leg covers a majority of the area and the toe only adds more work for little benefit.

In order to form an aabbox we take the local bbox and transform it just as if it was going to be passed through the scene graph. To do that we need two things: the transformation matrix that would be applied to the Node and the local aabbox of the Node—note, the local aabbox of a Node is the bbox before any transformations have been applied (see Figure 12.5 A red box). We can then apply the transformation matrix to the local aabbox to get the bbox (see Figure 12.5 B red box). This red bbox is used to form an new aabbox based on the rotated bbox (see Figure 12.5 B green box.) The idea is to take each of these aabbox(s) and accumulate them into a single aabbox. This aabbox is intersected with the aabbox of the pad.

We start with the leg of the landing gear. Head over to the TriEngineGear class and add a new method called *calcAABBox*. The first thing we calculate is the Affine transformation matrix of the leg:

```
Ranger.MutableRectangle<double> calcAABBox() {
  Ranger.AffineTransform at = _leg.calcTransform();
...
```

Now we calculate the local aabbox of the leg:

```
Ranger.MutableRectangle<double> localB = _leg.localBounds;
```

Finally we transform the local aabbox to parent-space relative bbox and return it:

```
Ranger.RectApplyAffineTransformTo(localB, rect, at);

return rect;
```

We now combine this bbox with the lander's body/hull bbox. In order to accumulate all the aabboxes we need to add a MutableRectangle property at the class scope. Return TriEngineRocket class add the property:

```
Ranger.MutableRectangle<double> accum = new Ranger.MutableRectangle<double>(0.0,
0.0, 0.0, 0.0);
```

Now **override** the *calcAABBox* and get the lander's Affine transformation matrix, this is the matrix applied to all the lander's child components:

```
@override
Ranger.MutableRectangle<double> calcAABBox() {
 Ranger.AffineTransform at = new
Ranger.AffineTransform.withAffineTransformP(node.calcTransform());
```

The first component is the lander's body. For this we do roughly the same thing we did for the gear in that we get the bounds and transform it, and then set the accumulation aabbox:

```
_rocket.getLocalBounds(); // Set node's rect data

Ranger.RectApplyAffineTransformTo(_rocket.rect, rect, at);

accum.setWith(rect);
```

Next are the legs. When we call the gear's *calcAABBox* method what we get back is a aabbox relative to the gear's leg:

```
Ranger.MutableRectangle<double> lgRect = _leftGear.calcAABBox();
```

This aabbox still needs to be transformed to both the gear and lander's space. So we retrieve the gear's transformation matrix and concatenate it with the lander's matrix.

Note

The *multiply* method is non-commutative so order *does* matter.

The last step for a leg is to apply the concatenated matrix to the leg and accumulate it with the running boolean union:

```
Ranger.AffineTransform atlg = _leftGear.node.calcTransform();
cat.multiply(atlg, at);
Ranger.RectApplyAffineTransformTo(lgRect, rect, cat);

accum.union(rect);
```

Then rinse and repeat for the other lander gear. But we need to verify all this work, so how about we add a NonUniformRectangleNode property to the GameManager:

```
// DEBUG
NonUniformRectangleNode aabbox = new NonUniformRectangleNode.basic(null,
Ranger.Color4IRed);
```

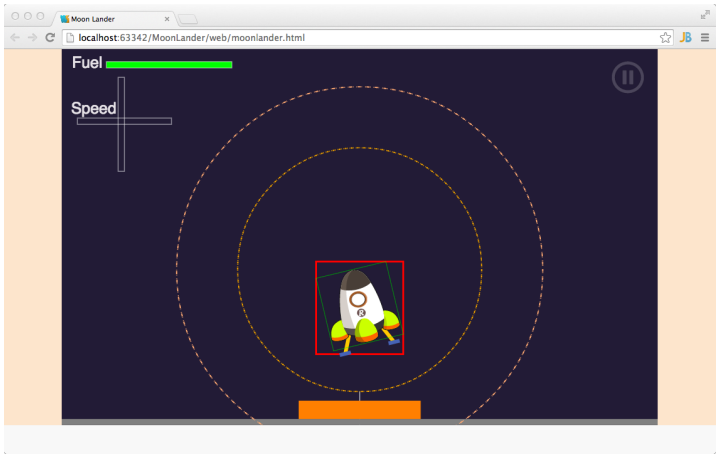
Then hope over to LevelRimbaloidLayer and add it as a child to the ***_zoomGroup***:

```
// DEBUG
_zoomGroup.addChild(gm.aabbox);
```

Return to TriEngineRocket's *calcAABBox* and set the rectangle property to the accumulation aabbox:

```
gm.aabbox.bottom = accum.bottom;
gm.aabbox.left = accum.left;
gm.aabbox.width = accum.width;
gm.aabbox.height = accum.height;
```

Now run Moon Lander and you should see the lander adorned with a red rectangle. When you rotate and extend/retract the lander gear you will see the red rectangle expand/contract according the accumulated aabboxes—you can see the lander's main-hull local-bbox in green:



Now we do the same thing for the landing pad. Return to LevelRimbaloidLayer and create a method called *calcAABBox* and calculate the aabbox for the pad:

```
Ranger.MutableRectangle<double> calcAABBox() {
  Ranger.AffineTransform at = new
Ranger.AffineTransform.withAffineTransformP(_pad.calcTransform());

  Ranger.MutableRectangle<double> padRect = _pad.localBounds;

  Ranger.RectApplyAffineTransformTo(_pad.rect, rect, at);

 return rect;
}
```

Then call *calcAABBox* in the layer's *update* method:

```
@override
void update(double dt) {
  _zone.updateState(gm.triEngineRocket.node.position);

  Ranger.MutableRectangle<double> padBox = calcAABBox();
}
```

Okay, now that we have our aabbox(s) we are ready to test for an intersection between them in the hope of detecting a collision.

Collision

Testing for collision is simply a matter of taking the lander's accumulated aabbox and the pad's aabbox and checking for an intersection. Extend the *update* method to check for an intersection, and for now just change the color of the aabbox:

```
@override
void update(double dt) {
  _zone.updateState(gm.triEngineRocket.node.position);

  Ranger.MutableRectangle<double> padBox = calcAABBox();

  Ranger.MutableRectangle<double> landerBox = gm.triEngineRocket.calcAABBox();

  if (padBox.intersects(landerBox)) {
    gm.aabbox.outlineColor = Ranger.Color4IGoldYellow.toString();
  }
  else {
    gm.aabbox.outlineColor = Ranger.Color4IRed.toString();
  }
}
```

Run Moon Lander now and maneuver the lander until the red aabbox touches the pad; it turns golden yellow! Sweet!

For now restore the gravity and original lander position—and don't forget to practice touching the pad for a round of fun. Later we will remove the aabbox(s) visuals.

Recap

We covered everything from zooming to collision:

- We added a ZoomGroup node to allow animated zooming.
- We added a Zone to trigger zooming.
- We covered the usage of Affine matrices for generating bounding boxes.
- We accumulated aabbox(s) to form a total aabbox.
- Finally we used the aabbox(s) to perform collision checks.

Now that we have the ability to detect collisions what do we when we actually detect a collision? How about we add some rules so we can update scores and finish the basic draft of Moon Lander.

Chapter 13 Scores and More

In the previous chapter we worked our way towards detecting when the lander had touched the

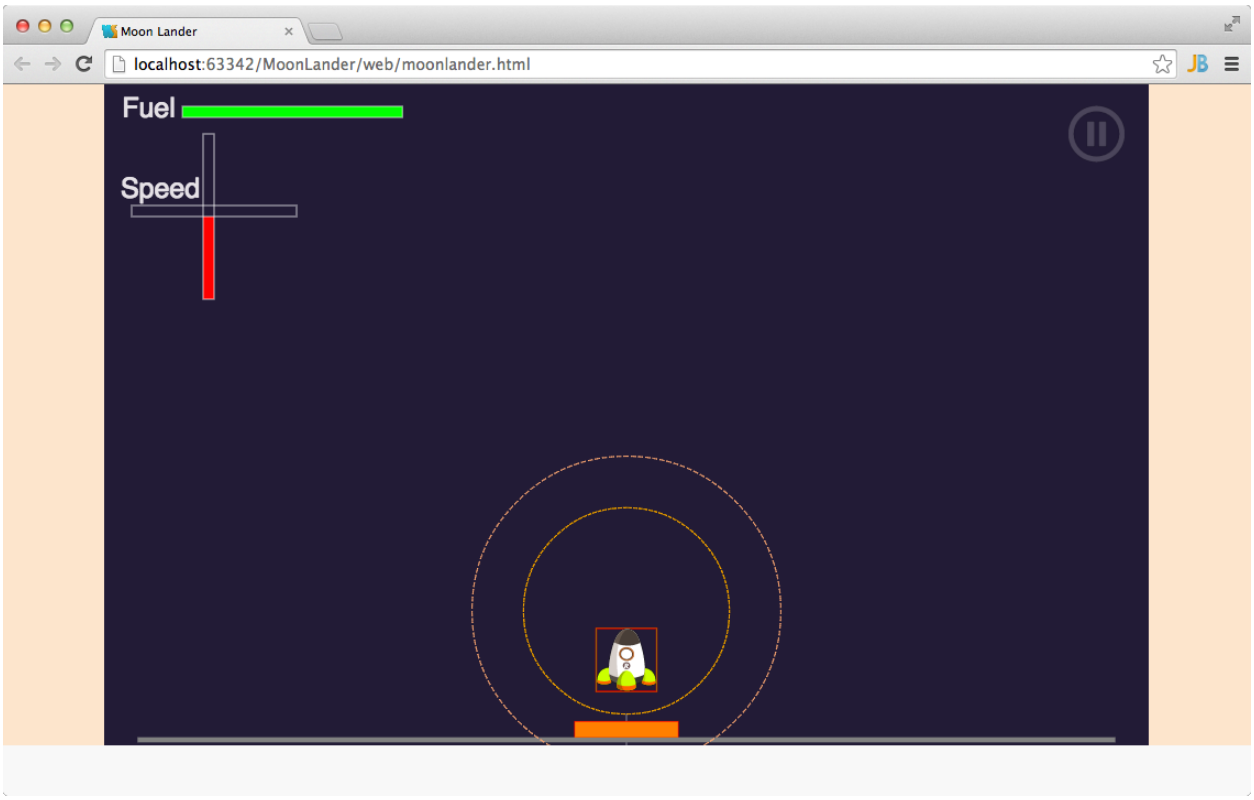


Figure 13.0 Moon Lander landing

pad. But there is more to do, and in this chapter we complete the collision code and connect it to the scoring system.

Goal

In this chapter we will complete the final draft of the game for this book. This includes finalizing landing, updating scores, updating hud to show high score pop out, issuing dialog boxes and landing animations. This code is also available with the book assets.

We start by finalizing the landing, and to do that we need a rule set that indicates what a "safe" landing is.

Rules

A successful landing consists of:

1. The lander must land on the pad where both sides of the lander are still over the pad.
2. The lander must be vertically aligned with no more than a maximum rotation tilt from the vertical.
3. The lander's horizontal and vertical velocity must be less than a certain maximum.
4. The landing gear must be extended. Doing so will add a finer degree of rotation control.

Failure to meet any one of the above rules results in a failed landing.

Finalizing the landing

Lets start by working on rule #1. In this case the lander's aabbox left-right edges must be "inside" the edges of the pad's aabbox. Out coding stopped inside LevelRimbaloidLayer's *update* method and now we resume there. Add a rule check for the alignment on the pad:

```
bool ruleCheckPadAligned(Ranger.MutableRectangle<double> landerBox,
Ranger.MutableRectangle<double> padBox) {
  bool rulePassed = true;

  // Check left/right edges.
  if (landerBox.left > padBox.left && landerBox.right < padBox.right) {
    gm.aabbox.outlineColor = Ranger.Color4IBlue.toString();
  }
  else {
    rulePassed = false; // Failed
  }

  return rulePassed;
}
```

Next add rule #2 where we check for vertical alignment:

```
bool ruleCheckVerticalAligned() {
  bool rulePassed = true;

  if (gm.triEngineRocket.node.rotationInDegrees.abs() < tiltTolerance) {
    gm.aabbox.outlineColor = Ranger.Color4IGreen.toString();
  }
  else {
    rulePassed = false; // Failed
  }

  return rulePassed;
}
```

Next we add rule #3 that checks for velocities. In this case we use the gauges to convert velocities into range values, which are the same values used to color the gauges:

```
int ruleCheckVelocities() {
  int rulePassed = 0;

  final double vValue =
gm.verticalVelocity.getValue(gm.triEngineRocket.verticalVelocity);
  final double hValue =
gm.horizontalVelocity.getValue(gm.triEngineRocket.horizontalVelocity);

  if (vValue > 0.1) {
    rulePassed = -1; // Failed
  }
  else if (vValue > 0.05) {
    gm.aabbox.outlineColor = Ranger.Color4IYellow.toString();
    rulePassed = 1; // Warning. Less points
  }
  else {
    gm.aabbox.outlineColor = Ranger.Color4IPurple.toString();
  }

  return rulePassed;
}
```

Finally we add rule #4 that checks for the landing gear being in the retracted state. Recall that the gear isn't extended until the gear reaches its full extension which happens at the end of the animation sequence:

```
if (gm.triEngineRocket.gearsRetracted) {
  _landingFailed();
  return;
}
```

With all the rules in place we can refactor the *update* method to call each rule (hint, this is going to be refactored again when we introduce a State machine.) When you run Moon Lander now you will see various colors for each rule. For now the colors are used to indicate state, White indicates all is good, red means failure, any other colors simply indicate a specific rule has passed:

```
if (padBox.intersects(landerBox)) {
  bool successful = ruleCheckPadAligned(landerBox, padBox);

  if (!successful) {
    _landingFailed();
    return;
  }

  successful = ruleCheckVerticalAligned();
  if (!successful) {
    _landingFailed();
    return;
  }

  int ruleRes = ruleCheckVelocities();
  if (ruleRes < 0) {
    _landingFailed();
    return;
  }

  if (gm.triEngineRocket.gearsRetracted) {
    _landingFailed();
    return;
  }

  return;
}

gm.aabbox.outlineColor = Ranger.Color4IWhite.toString();
```

Now we can recognize if a potential landing will be successful—or not. Lets handle a successful landing first.

> Note
>
> You can disable the aabbox visuals once you are confident they are working.

Successful landing

Once we detect a successful landing we immediately remove control from the player and begin the auto-landing sequence. This sequence comprises of—in no particular order:

- Disable controls
- Thrust and Momentum to zero
- Gravity ignored
- Rotating the lander completely vertical
- Adjust the altitude such that the landing gear toes touch properly.
- Animating the lander's hull to bounce a bit.

To disable the controls we add a property to **TriEngineRocket** that is checked during the *update* method. In each location where a control is checked we add a boolean &&. Below is the check for pitch:

```
void update(double dt) {
  if (_pitchingLeft && !_pitchingRight && controlsEnabled) {
```

To disable gravity we add another property that is used to bypass momentum which will simulate the force the pad applies to the lander:

```
if (!ignoreGravity)
  _momentum.add(gm.gravity);
```

Handling thrust and momentum is simply setting them to zero:

```
void cutThrust() {
  _thrust.speed = 0.0;
  _momentum.speed = 0.0;
}
```

Next is aligning the lander. For this we use an animation to minimize sudden snaps which would come across as unnatural:

```
void alignUpright() {
  UTE.Tween rotateTo = new UTE.Tween.to(this, HULL_ROTATE, 1.0)
    ..targetValues = [0.0]
    ..easing = UTE.Expo.OUT;

  ranger.animations.add(rotateTo);
}
```

Of course you can guess that in order for this tween animation to work on the TriEngineRocket we need to implement the Tweenable interface such that it controls both the lander's body and rotation. First we mix-in the Tweenable:

```
class TriEngineRocket extends MobileActor with UTE.Tweenable {
  static const int HULL_ROTATE = 10;
  static const int TWEEN_TRANSLATE_Y = 20;
```

Then implement the interface (only the set is shown):

```
void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) {
  switch(tweenType) {
    case HULL_ROTATE:
      _centroid.rotationByDegrees = newValues[0];
      break;
    case TWEEN_TRANSLATE_Y:
      _rocket.positionY = newValues[0];
      break;
  }
}
```

Now we add the animation to bounce the lander's body just a bit. The values chosen seem to produce a good bouncing effect:

```
void restHull() {
  UTE.Tween dropBy = new UTE.Tween.to(this, TWEEN_TRANSLATE_Y, 2.0)
    ..targetRelative = [-10.0]
    ..easing = UTE.Bounce.OUT;

  ranger.animations.add(dropBy);
}
```

All these can now be called in a cascade manor from the layer. Return LevelRimbaloidLayer and add a new method called *_landingSuccessful* that makes all the calls:

```
void _landingSuccessful() {
  gm.aabbox.outlineColor = Ranger.Color4IBlue.toString();

  // Auto land
  gm.triEngineRocket..controlsEnabled = false
    ..cutThrust()
    ..ignoreGravity = true
    ..restHull()
    ..alignUpright();
}
```

Be sure to call it from the *update* method after all the rules have been run:

```
  if (padBox.intersects(landerBox)) {
...
    _landingSuccessful();

    return;
  }
```

Lets have some fun. Run Moon Lander and do your best in trying to land the lander on the orange pad. If you succeed your lander will auto-land. If you fail you will magically fall right through. Lets fix that.

Failed landing

The graphics for a failed landing are more complex because we have to “destroy” the lander. To do this is going to require a fair amount code so again I will only show high level code snippets, and as usual you can see the final code on the book’s website.

For a failed landing we have a laundry list of visuals effects we want to pull off:

- Break lander into discrete parts
- Explosion ring
- Explosion flash disc
- Particle system explosion
- Impulse forces with gravity

We begin with the exploding ring first.

Ring Explosion

Begin by copying **CircleNode** into a new class called **AnimatableCircleNode**—don't forget to update *main.dart* to include the new class. Refactor the class to mix-in **Tweenable** and change the ***fillColor*** and ***outlineColor*** properties to **Color4**<**int**> objects instead. We change the color properties because it is easier to animate an integer, and in this case we are animating the alpha component of the color:

```
class AnimatableCircleNode extends Ranger.Node with UTE.Tweenable {
  static const int SCALE = 10;
  static const int ALPHA = 20;

  Ranger.Color4<int> fillColor;
  Ranger.Color4<int> outlineColor;
```

Now implement the Tweenable interface to support both **SCALE** and **ALPHA**:

```
void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) {
  switch(tweenType) {
    case SCALE:
      uniformScale = newValues[0];
      break;
    case ALPHA:
      outlineColor.a = newValues[0];
      break;
  }
}
```

Because the Node will be used on different layers we add a Node property to the GameManager and create in the *postLoad* method:

```
AnimatableCircleNode explodingRing;
...
explodingRing = new AnimatableCircleNode.basic();
```

Like all Nodes, in order to be visible we need to add the Node somewhere in the scene graph. Return the Rimbaloid layer's *_configure* method and add the Node while configuring it:

```
gm.explodingRing..visible = false
  ..outlineThickness = 5.0
  ..outlineColor = Ranger.Color4IWhite;

_zoomGroup.addChild(gm.explodingRing);
```

Lets trigger the ring to explode! Create a new method called *_explodeRing*. We do three things: position the ring, start a scale animation and start a fade animation. The code prepares the ring first by setting its position to reflect the lander's, then making it visible and setting its initial scale to something small.

With the ring's initial values set we can construct two animations each acting on a separate ring property. I chose an exponential scaling rate but a linear fade rate just because it looks fairly stylistic:

```
void _explodeRing() {
  gm.explodingRing..setPosition(gm.triEngineRocket.node.position.x,
gm.triEngineRocket.node.position.y)
    ..visible = true
    ..uniformScale = 0.1;

 UTE.Tween scaleUp = new UTE.Tween.to(gm.explodingRing,
AnimatableCircleNode.SCALE, 1.0)
    ..targetRelative = [300.0]
    ..easing = UTE.Expo.OUT;
  ranger.animations.add(scaleUp);

 gm.explodingRing.outlineColor = Ranger.Color4IWhite;
  UTE.Tween fadeOut = new UTE.Tween.to(gm.explodingRing,
AnimatableCircleNode.ALPHA,0.8)
    ..targetValues = [0]
    ..easing = UTE.Linear.INOUT;
  ranger.animations.add(fadeOut);
}
```

When you run Moon Lander now simply let the lander fall. When it hits the pad the *_landingFailed* method is called causing the ring to explode! How about we animate the lander explosion as well.

Glorious Lander Destruction

On a failed landing we want the lander to “go up” in glorious fashion. For this we need to break the lander SVG is to smaller pieces. This has already been done and you can get the assets from the book’s asset site. There are four of them called:

- EngineRocket3_hull.svg
- EngineRocket3_centercell.svg
- EngineRocket3_leftcell.svg
- EngineRocket3_rightcell.svg

Each one becomes a Node in it own right, the landing gear Nodes are simply rectangles.

Once the new assets have been copied add them as resources to the **Resources** class. In order to create the effect that the lander has exploded we perform an instantaneous switch between a complete lander and a discrete lander. The individual parts don’t have to be exactly aligned to the complete lander because we are going to immediately apply an impulse force to them. This force will cause the parts to “fly” upwards in a arc while still being subject to gravity. In addition, a spin will be imparted on each lander part.

Thus each part is launched in a random upward direction with a random spin rate and impulse force. Each part has a mass that directly interacts with gravity providing the allusion of different inertias.

We start by creating a new **MobileActor** class that is modeled after the **TriEngineRocket**, call the class **TriEngineRocketPart**. This class will only have the base features of TriEngineRocket namely it won’t have any controls like pitch, thrust or animations. To separate them out we create an intermediate class called **PhysicsActor** and refactor the physics code into it (see Figure 13.1.)

Next we allocate all the parts in the GameManager class. These include: Hull, engine cells, landing gear legs and toes. Then we add them to the Rimbaloid layer in the class’s _configure

method defaulting them to invisible. Because each part is a physics part we need to update them on each system update but only when the pad has been “hit” to save processing cycles.

Once the lander crashes we call the _landingFailed method which does a host of things:

- Disable controls
- Cut thrust
- Ignore gravity
- Hide lander and show parts
- Explode Ring, disc and particles
- Explode Lander

Each has a number of steps involved. For example, exploding the ring requires we:

- Set its position to the lander’s position.
- Make it visible.
- Scale it down in preparation for an animated scale up.
- Create a fade in and fade out and add a callback on the fade out.
- Add the animations to get them started.
- When fade out is complete create ring scaling animations and start them too.

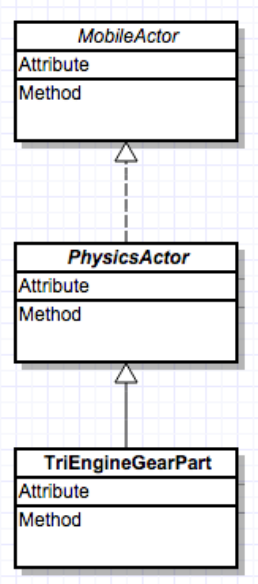


Figure 13.1 PhysicsActor

Exploding the lander is a series of random number generations one for each part. Each part is subjected to a force applied in a random direction for a random amount of time with a random amount of power.

For the particles we simply copy the rocket engine particles and make a small adjustment in order to turn it into an explosion instead of a stream of particles. To do that we create a particle system in the layer class with a larger particle count, then tweak the activation parameters for longer life times and bigger particle sizes. In addition, we need to go back to our ParticleActivation class and update it to understand the **OMNI_DIRECTIONAL** style:

```
case Ranger.ParticleActivation.OMNI_DIRECTIONAL:
  activationData.velocity.directionByDegrees = _randGen.nextDouble() * 359.0;
  break;
```

And finally, in order to see most of the parts fly upward we need to zoom out a bit during the explosion. I wasn’t kidding that it is a large amount of code. No worries though, just inspect the book’s asset code. If you run Moon Lander now you will see a white flash followed by a ring expanding and the lander flying upward in pieces. Nice!

Now it is time to complete the overall structure of the game. For that we need to add a few more popups and the ability to reset. So lets get cranking shall we.

Handling Success

We continue to follow the general flow referred to back in Figure 9.1. At this point we can detect both success and failure. Lets handle success first. When a lander lands successfully we float a

score number upwards that fades out, and show pulsing text with an icon. Each time they land successfully we increment their landings. They keep landing until they fail. Once they fail we check their total successful landings, and if they ranked/placed, we ask them for their initials.

We start by adding a landing counter to **LevelRimbaloidLayer**:

```
int successfulLandings = 0;
```

When we detect a successful landing we increment the counter and switch to a new State. What are these States you ask?

State machines

Most games are really just state machines and Moon Lander isn't any different. In software design there are several approaches for handling program state. For simpler games perhaps a **switch** statement is used, and for larger games an actual State machine Design Pattern is used. Because the book's version of Moon Lander is fairly simple we are going to use a **switch** statement. However, this is going to require a bit of a refactor and as such a fare amount of code is going to change, so rather than put you through a time consuming refactor I will review from a high level the refactor that was performed. In order to do that we first take a look at the state machine that makes up the Rimbaloid level (see Figure 13.2.)

Each state has a corresponding constant:

```
static const int GAME_STATE_RESET
```

The **switch** statement checks the gameState property on every update and calls the appropriate method depending on the state.

The level starts by performing a GAME_STATE_RESET then immediately switches to the GAME_STATE_NEW_LANDING state which configures the lander for landing and again immediately switches to the GAME_STATE_LANDING state where the lander begins to descend. The player either successfully lands or crashes. If they crashed we check if they ranked on the scoreboard GAME_STATE_FAIL. If so we ask for their initials and then ask if they want to play again. If yes then we reset else we pop the current scene (aka Rimbaloid scene) which returns us to the level selection scene.

Each state is responsible for either performing some sort of animation, for example, a popup asking the player if they want to play again or animating an explosion.

If they click the pause button we verify that they want to quit and lose their current landing record.

We define these states as a set of constants:

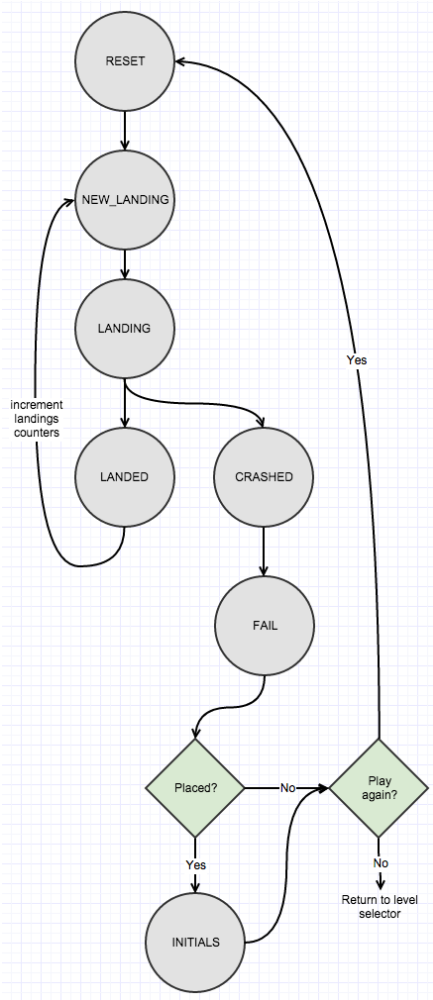


Figure 13.2 State machine

```
static const int GAME_STATE_RESET = 0;
static const int GAME_STATE_INTRO = 3;
static const int GAME_STATE_NEW_LANDING = 5;
static const int GAME_STATE_LANDING = 10;
// Show floating text and number
static const int GAME_STATE_LANDED = 20;
static const int GAME_STATE_CRASHED = 30;
static const int GAME_STATE_WIN = 40;
static const int GAME_STATE_FAIL = 50;
static const int GAME_STATE_INITIALS = 60;
static const int GAME_STATE_PLAYAGAIN = 70;
static const int GAME_STATE_END = 80;
static const int GAME_STATE_PAUSE = 90;

int gameState = GAME_STATE_PLAY;
```

Now our layer's *update* method becomes a mini state machine driven by a **switch** statement that branches depending on the current state:

```
@override
void update(double dt) {
  switch (gameState) {
    case GAME_STATE_RESET:
      _resetState();
      break;
    case GAME_STATE_INTRO:
      _updateIntroState(dt);
      break;
    case GAME_STATE_NEW_LANDING:
      _updateNewLandingState(dt);
      break;
...
```

As you can probably see by now the code is becoming progressively more complex most of which is just dealing with State flow. Make sure to review the various methods in the book's final rendition to see what was refactored in order to support States. For the most part the code was split into separate update methods matching each State. Lets take a look at coding an input dialog using HTML/CSS.

HTML Initials dialog

We could create a custom Node to handle input but this would require a considerable amount of work putting together a Node to handle character input and control. Instead there is a much simpler way that already exists—HTML. The trick to using HTML is make it appear seamlessly integrated with the other Node based dialogs. We can do this by styling our HTML dialog using CSS.

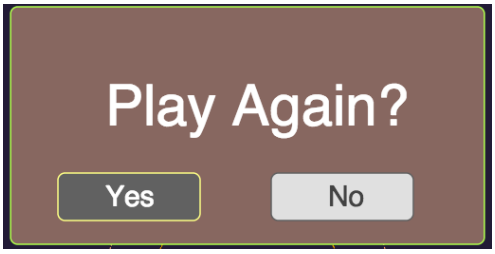


Figure 13.3 Color Scheme

So far our dialogs have been styled using the color scheme shown in Figure 13.3. You can see I have been using a chocolate brown background with a yellow-green border. We can replicate this precisely using CSS.

Lets start by creating a abstract base html dialog that represents a simple set of features:

- hide and show
- width and height
- background

See Code 13.4.

Code 13.4

```
abstract class BaseHtmlDialog {
  final DivElement _content;

  bool _built = false;

  BaseHtmlDialog() :
    _content = new DivElement()
  {
    _content.id = "defaultDialogStyle";
  }

  DivElement get content => _content;

  void hide();

  void show();
}
```

Notice that there is a **DivElement** property. This is a **Dart** class that wraps an actual HTML element. Also notice that the constructor sets the Div's ***id*** property to some string value ("defaultDialogStyle"). This id is a CSS reference defined in main.css. However, it isn't the style we want so we create our own style:

```
#initials_dialog_modalContent {
  display: block;
  position: absolute;
  background-color: #866761;    // Chocolate brown background
  border: #B2FF59 solid 1px;    // yellow-green border
  border-radius: 15px;
  visibility: hidden;
  overflow-x: hidden;
}
```

We use this style instead when we build out our Initials dialog. Lets do that now. As usual we create a new class called **InitialsDialog** that extends **BaseHtmlDialog** while mixing in **Tweenable**:

```
class InitialsDialog extends BaseHtmlDialog with UTE.Tweenable {
```

The *_configure* method augments the style information by computing the centering information. The code checks the surface's dimensions against those of the canvas. The dimensions can differ if the app is running on the desktop where the browser can be much

larger than the Design specifications. On mobile devices the dimensions will generally be the same which means the offsets will equate to zero:

```
int canvasXInset = (surfaceWidth - canvas.clientWidth) ~/ 2.0;
centerXOffset = ((canvas.clientWidth) - width) ~/ 2.0 + canvasXInset;

int canvasYInset = (surfaceHeight - canvas.clientHeight) ~/ 2.0;
int centerYOffset = ((canvas.clientHeight) - height) ~/ 2.0 + canvasYInset;
outOfViewX = (-width - outlineWidth).toInt();

content.style
  ..left = "${outOfViewX}px"
  ..top = "${centerYOffset}px"
  ..width = "${width.toInt()}px"
  ..height = "${height.toInt()}px";
```

From here we can pile on the elements that make up the initials dialog. These include:

- A Label element for the text body.
- Input element for capturing the initials.
- Another Div element to contain a button Span element.
- Register events on the Span for click, enter and exit.

In all these elements we assign ***id*** values that are defined *main.css*. Here is a snippet of the "Okay" button:

```
DivElement buttonContainer = new DivElement()
  ..id = "initial_dialog_buttonContainer";

SpanElement okayButton = new SpanElement()
  ..id = "initial_dialog_okayButton"
  ..text = "Okay";

okayButton.onClick.listen(
    (Event e) => _validate(initialInput.value)
  );
okayButton.onMouseEnter.listen(
    (Event e) =>
    okayButton.style.color = Ranger.color4IFromHex("#dddddd").toString()
  );
okayButton.onMouseLeave.listen(
    (Event e) =>
    okayButton.style.color = Ranger.color4IFromHex("#222222").toString()
  );
buttonContainer.nodes.add(okayButton);
```

When they click on the "Okay" button we need to validate that they entered either 2 or 3 letters only. Values like; "ab1" or "c2" are not valid, they must enter only alpha characters. We can control this using a regular expression:

```
RegExp exp = new RegExp(r"(^[A-Za-z]{2,3})+$");
```

I built and tested the expression using http://regexr.com/. It is a very handy tool for working with regular expressions. To use the expression we call the *firstMatch* method on the expression

object. If a match is found a non-null **Match** object is returned. If we get a null returned then we animate the Input element's background color:

```
Match m = exp.firstMatch(value);
if (m == null) {
  gm.playSound(gm.inCorrectSoundId);
  // Tint text box
  _tintAnimation();
} else {...}
```

This leads to the reason why we mixed in **Tweenable**, we need to animate both the dialog's position and the input's background color. You can see this by looking at the dialog's *setTweenableValues*:

```
void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) {
  switch(tweenType) {
    case TWEEN_X:
      content.style.left = "${newValues[0].toInt()}px";
      break;
    case TINT:
      initialInput.style.backgroundColor = "rgba(${newValues[0].toInt()}, 0, 0,
1.0)";
      break;
  }
}
```

Above we animate the Div's left style property in a way that it animates the Div horizontally. If we view the InitialsDialog now we would get what you see in Figure 13.5.

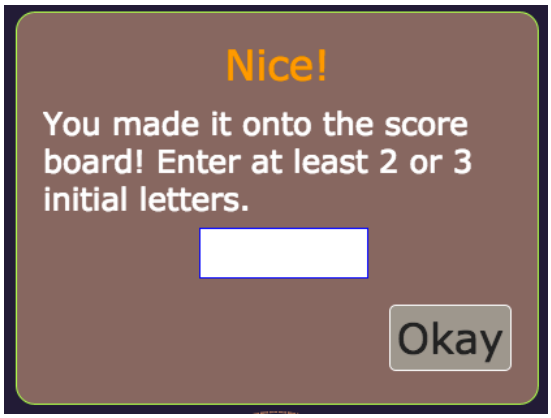


Figure 13.5 Initials dialog

Style wise it looks very much like the PlayAgain dialog but instead of it being a Ranger Node it's actually an HTML/CSS component. Nice. We now have a way to capture initials but what about updating the score board?

Updating the scores

In order to manage the scoreboard we need two basic functions: a function to see if we placed/ranked and another to update the scores. The rules for updating are simple, if there are already

10 scores present then you must have bested one of them to rank. If you did rank then the smallest score is thrown away otherwise your new score is added.

A good place to put these two functions is in the **GameManager** class:

```
bool placed(int landings) {
  // Did they beat one of the scores
  int score = scores.keys.firstWhere((int i) => landings >= i, orElse: () =>
null);

  return score != null;
}

void updateScores(int landings, String initials) {
 if (scores.length == 10) {
    if (!scores.keys.contains(landings)) {
      List<int> sortedScores = scores.keys.toList();
      sortedScores.sort();

      // Remove the lowest score
      scores.remove(sortedScores.elementAt(0));
    }
  }

  scores[landings] = initials;
}
```

These two functions work off the Map structure defined back in Chapter 9, see Code 9.15B, but it's shown here again for reference:

```
Map _buildDefaultConfig() {
  Map m = {
    "Music": false,
    "Sound": true,
    "Cloud": false,
    "Scores": {
      4: "Moo",
      3: "n L",
      2: "and",
      1: "er"
    }
  };

  return m;
}
```

The most appropriate location to call the *placed* method is in the Rimbaloid layer when the player has "finally" crashed and we **switch**ed to the GAME_STATE_FAIL State:

```
void _updateFailState(double dt) {
 _updateExplosion(dt);
 if (gm.placed(successfulLandings)) {
    _changeGameState(GAME_STATE_INITIALS);

    gm.initialsDialog.show();
  }
...
```

But where do we “view” the scoreboard. For that we need to create a new Scene and Layer.

Scores Scene/Layer

The last area of Moon Lander that is missing is the the score board. It is relatively simple, we just list the top ten scores. The score board can be accessed from the same Scene that the SettingsDialog is accessed from namely the MainScene. Access is via a new icon that appears like a awards medal.

After pulling in the medal.svg asset and updating the Resources class to load it we can update the **MainLayer** class to configure, show and bind to it as we have done previously with the Gears icon. First we need a new property for the *medal.svg* icon:

```
Ranger.SpriteImage _scores;
```

Next we add a SpriteImage based off that resource in the *_configure* method. It is positioned on the opposite side of the gear:

```
// Scores icon
_scores = new Ranger.SpriteImage.withElement(gm.resources.medal)
  ..uniformScale = 0.35
  ..rotationByDegrees = 10.0
  ..setPosition(w - (w / 7.0), -h + (h / 5.0));
addChild(_scores);
```

And for extra measure we slowly rock the medal for added effect:

```
void _rockNode(Ranger.BaseNode node) {
  UTE.BaseTween wobbleCW = ranger.animations.rotateBy(
      node,
      2.0,
      -node.rotationInDegrees * 2.0,
      UTE.Quad.INOUT,
      null,
      false);

  wobbleCW.repeatYoyo(10000, 0.0);
  ranger.animations.track(node, Ranger.TweenAnimation.ROTATE);

  ranger.animations.add(wobbleCW, true);
}
```

Now we “bind” click functionality in the *onMouseDown* method, and upon a click detected we transition to a new scene called **ScoresScene**:

```
if (_scores.pointInside(nodeP.v)) {
  nodeP.moveToPool();

  // Transition to Scores scene
  ScoresScene inComingScene = new ScoresScene();
  Ranger.TransitionMoveInFrom transition = new
Ranger.TransitionMoveInFrom.initWithDurationAndScene(0.5, inComingScene,
Ranger.TransitionSlideIn.FROM_BOTTOM)
    ..name = "TransitionMoveInFrom";

  ranger.sceneManager.pushScene(transition);

  return true;
}
```

Next we create the ScoresScene. It is as simple as the LevelSelectionScene, it just creates and sets the primary layer. The real work is done in the ScoresLayer. The ScoresLayer has three key elements:

1. Title
2. Scores
3. Return icon

For the Title we create and configure a TextNode and then animate it into view. This functionality is identical to the MainScene's title. The Return's icon functionality is also identical to the LevelSelectonLayer's icon. The new element is the scores.

Each time the ScoresLayer is viewed we sort the scores and then "load" them as a collection of TextNodes:

```
List<int> sortedScores = scores.keys.toList();
sortedScores.sort((x, y) => y.compareTo(x));
double x = -750.0;
double y = 200.0;
for(int i = 0; i < rows; i++) {
  int score = sortedScores[i];
  String initials = scores[score];

  Ranger.TextNode initialsTN = new
Ranger.TextNode.initWith(Ranger.color4IFromHex("#284734"))
    ..uniformScale = 15.0
    ..text = "${initials} - "
    ..setPosition(x, y)
    ..shadows = false;
  addChild(initialsTN);

  Ranger.TextNode scoreTN = new Ranger.TextNode.initWith(Ranger.Color4IBlack)
    ..uniformScale = 15.0
    ..text = "$score"
    ..setPosition(x + 450.0, y)
    ..shadows = true;
  addChild(scoreTN);

  y -= 150.0;
  if (i == 4) {
    x = 250.0;
    y = 200.0;
  }
}
```

If you were to run Moon Lander now and click on the medal icon (Figure 13.6 A) you would see the default scoreboard (Figure 13.6 B):

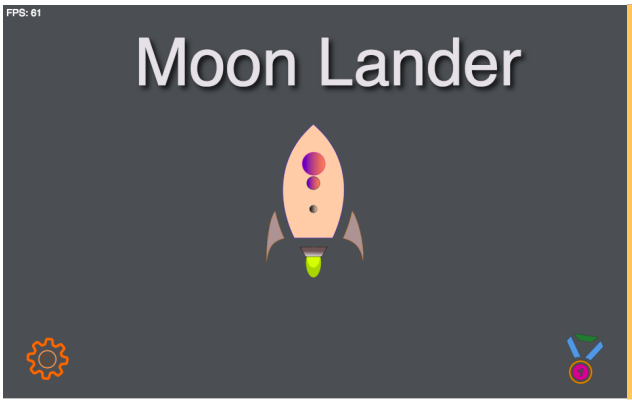


Figure 13.6 A MainScene

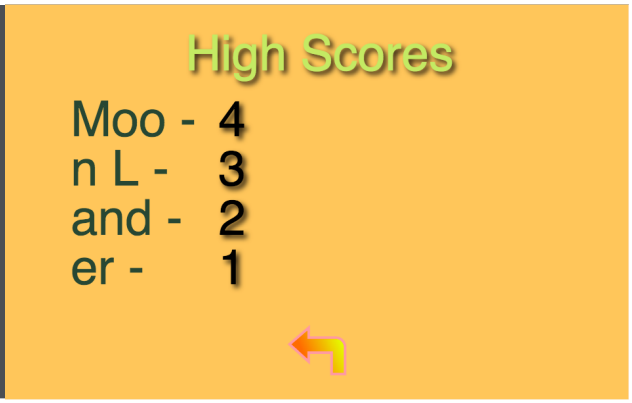


Figure 13.6 B ScoresScene

Yeah! We can now practice landing and reap a reward for our hard work.

Closing of Moon Lander

It has been quite a journey both in this chapter and all the preceding ones. In this chapter we blazed through completing the Rimbaloid level, from an exploding Lander sequence to a State machine to a Scoreboard scene, and this only just scratches the surface of what we could do.

This book's version of Moon Lander is certainly not feature complete. The original intent was to limit the feature set in order to focus on Ranger and its framework. But the journey doesn't stop here. There is much more that can be done to make Moon Lander into a full fledged game, or any game for that matter. You are limited only by your imagination. Remember the all the code for Moon Lander is available with the book's assets.

Coming up

In the previous chapters we covered quite a bit about Ranger; Nodes, Scenes, Layers, Animation, Sound, Multilayers, Particle systems, GUIs to name just a few, and combined together you can create an infinite number of games. But each was covered relative to an ever growing code base which can make it difficult to understand the finer details of each piece of Ranger. In the coming chapter(s) we will isolate most of these high-level pieces in such a way as to make them easier to identify and evaluate as something relevant to what you may be working on.

Chapter 14 Reference

In the previous chapters we learned Ranger via a game called Moon Lander.

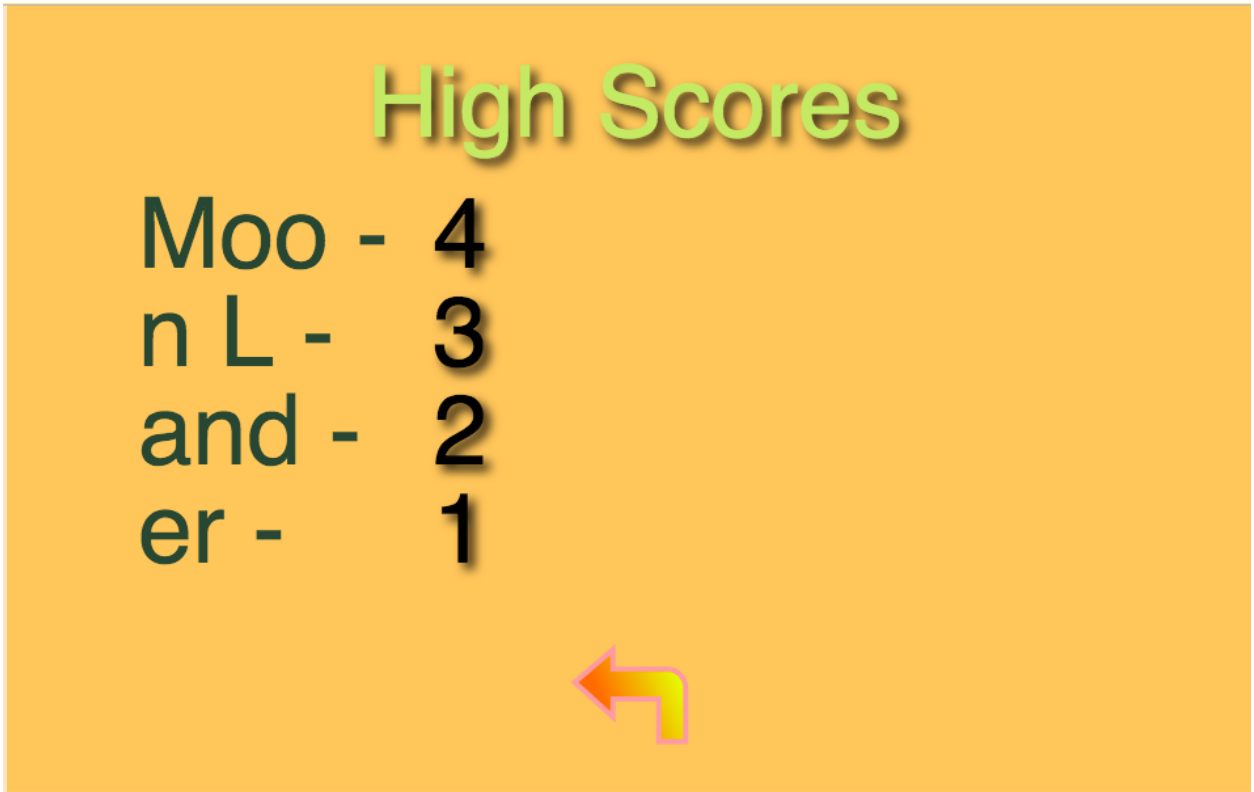


Figure 14.0 Score board

It gave us a chance to learn and familiarize ourselves with how Ranger works and how to use it to make games. But as you saw when you moved through the chapters the code was becoming more complex which eventually drifted away from Ranger specifics.

In this chapter we focus on the the basics, starting with setting up a simple Ranger app framework. With this framework app we can cover each entity within Ranger.

Basic App

There are several ways to create a basic Ranger app:

- Use one of the templates in Ranger-Sack
- Use one of unit test cases
- Use an IDE with web app generators

Whichever approach you take the minimum framework will consists of the following things:

Pubspec.yaml

You need an entry for Ranger:

```
ranger:
  git: git://github.com/wdevore/ranger-dart.git
```

Depending on which IDE you are using; Dart Editor or WebStorm 10+, you will need to "update" the dependency you just added to your yaml file.

main.dart

You need an import for Ranger:

```
import 'package:ranger/ranger.dart' as Ranger;
```

Next update the *main* entry point to create an instance of ranger. If you are developing on the desktop *fitDesignToWindow* is the more appropriate factory to use:

```
Ranger.Application ranger;

void main() {
  ranger = new Ranger.Application.fitDesignToWindow(
   window,
   Ranger.CONFIG.surfaceTag,
   preConfigure,
   1900, 1200
   );
}
```

Choose a Design resolution that is appropriate for your target. Above I have chosen 1900x1200 for a Nexus 7 tablet.

The *preConfigure* callback is your chance to configure your boot sequence. The typical arrangement is a main Scene and splash Scene all preceded by a BootScene:

```
void preConfigure() {
  MainScene mainScene = new MainScene()
    ..name = "MainScene";

 SplashScene splashScene = new SplashScene.withReplacementScene(mainScene)
    ..pauseFor = 2.0
    ..name = "SplashScene";

 Ranger.BootScene bootScene = new Ranger.BootScene(splashScene)
    ..name = "BootScene";

 ranger.sceneManager.pushScene(bootScene);

 ranger.gameConfigured();
}
```

index.html

Your main html file needs to have one key **DIV** element with both its ***width***, ***height*** and ***id*** attributes defined. The ***width*** and ***height*** attributes are a simulated device dimensions. The ***id*** attribute is what Ranger looks for to "inject" the Canvas element into. It should match the second parameter to the *fitToDesignToWindow* factory method.

The other elements are standard Dart or HTML requirements:

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Moon Lander</title>
    <link rel="stylesheet" href="main.css">
  </head>
  <body>
    <Div id="gameSurface" width="1000" height="600"></Div>
    <script type="application/dart" src="main.dart"></script>
    <script src="packages/browser/dart.js"></script>
  </body>
</html>
```

main.css

The id "gameSurface" value from index.html is defined in main.css. The minimum styles requirements are:

```
body {
  background-color: #F8F8F8;
  padding: 0;
  margin: 0;
}

#gameSurface {
  margin: 0 auto;
  position: relative;
  overflow: hidden;
}
```

gameSurface selector's ***position*** value is set to "relative" to position the Canvas *within* the DIV container.

That is the minimum Ranger framework. Of course you would have designed and coded both main Scene and splash Scene.

Nodes

Almost everything thing in Ranger is a **Node**. Ranger is functions entirely on Nodes placed into a Scene Graph. Ranger's Scene Graph is a blend of two types Nodes: *Monolithic* and *Polylithic*. The *Monolithic* type is where each Node encompasses a large amount of functionality, whereas *Polylithic* Nodes are lightweight with very minimal functionality. Ranger uses Dart's mix-ins to form a blend of these two types. However, the default Node that comes with Ranger leans closer to a *Monolithic* Node because almost all the mix-ins have been "mixed in" for you. If you decide that you need leaner Nodes you can always clone the **BaseNode** and **Node** classes and strip off any mix-ins you don't need. With these mix-ins come a range of characteristics.

Node Characteristics

Nodes can be categorized into several main areas:

1. Visiting
2. Transformations
3. Drawing
4. Mapping
5. Timing
6. Object pooling

Visiting

On each frame (aka clock tick or Core.step—which is about 1/60 of a second) Ranger traverses its internal Scene Graph by "visiting" each Node and checking for visibility. If the Node isn't visible it is skipped. By default all Nodes are visible even if they are not within the viewport. To change this behavior you would **override** the *visit* method of your Node and cull against the viewport (Canvas context.)

The *visit* algorithm works from the Root to Leaf and left to right hierarchically (see Figure 3.2.) Nodes that are added as children first are drawn first meaning they will be "underneath" a Node added last. You can artificially change this approach by using the Node's ***zOrder*** value which can only be specified during construction of the Node.

Transformations

As the traversal progresses through each Node an Affine Transform Matrix is calculated just prior to drawing the Node. A dirty flag is used to optimize unnecessary matrix calculations. If a transform property is changed, for example rotation, then the dirty flag becomes true and on the next pass through the graph the matrix transform is updated.

This matrix is concatenated to the current Canvas context. If there are more children then those children are traversed before the current Node's draw method is called.

Drawing

Prior to each draw call the Canvas *context* is saved. Once the draw is complete the *context* is restored.

> **Note**
>
> Currently Ranger only supports HTML Canvas but future versions will support WebGL most likely through an intermediate package like Three.dart.

Part of the state that is stored/restored is the context's transform matrix. This state process, combined with a Scene Graph, creates a transformation stack which in turn allows Ranger to form hierarchal transforms.

Mapping

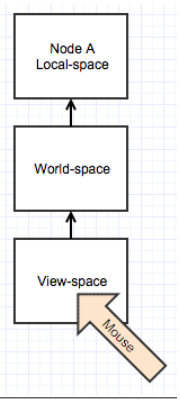
Every Scene Graph worth its weight comes with a corresponding set of mapping methods. These mapping methods map data (typically Points or Rectangles) from one Space to another. There are four important Spaces:

- **Local**: Each Node has its own local-space (also called node-space). If, for example, you rotated a Node then its local-space remains unchanged. However, it does show a rotation from an outside perspective, for example Parent-space.
- **Parent**: Each Node belongs to a parent-space including the Root Node whose parent-space is world-space.
- **World**: World-space is an intermediate space used for mapping between between two disparate Nodes.
- **View**: View-space is typically thought of as Mouse-space. You may be inclined to think of it as Design-space but remember that Design dimensions are scaled by the DIV container's dimensions (or simulated physical dimensions.) It may happen that Design and Mouse dimensions are the same but don't always count on it. Note: The mapping methods for this space are supplied by the rendering context. View-space is mentioned here only for completeness.

The two most common forms of mapping are from the local-space of a Node to the local-space of another Node via world-space, or parent-space, or from view-space to local-space.

The first is used most commonly to track a Node's local-space relative to another Node, for example, emitting a particle into the local-space of a Node—perhaps a layer—based on the position of another Node—perhaps an EmptyNode representing the exhaust port of a space ship.

The second is most commonly used to determine if a Node has been selected by a mouse click (shown to the right.)



Timing

Every Node is a TimingTarget and as such can schedule itself with the Scheduler to receive periodic updates. You schedule updates by typically calling *scheduleUpdates* from within the *onEnter* method that you would normally override. To actually receive these updates you would override the Node's *update* method:

```
@override
void update(double dt) {...}
```

Object pooling

Every Node can be optionally pooled. To make a Node pooling capable you create a factory method that creates and configures the Node for pooling—this means pooling is an "opt in" choice. Particles that stem from Particle Systems are an excellent candidate for pooling.

Here is the typical approach for adding pooling to a Node, it consists of five things:

A default constructor for instances where a user doesn't want a pooled version:

```
CircleParticleNode();
```

A private constructor for use with a private static property:

```
CircleParticleNode._();
```

A *pooled* factory method for complex clients that want to manage configuration themselves:

```
factory CircleParticleNode.pooled() {
  CircleParticleNode poolable = new Ranger.Poolable.of(CircleParticleNode,
_createPoolable);
  poolable.pooled = true;
  return poolable;
}
```

A factory method with known effects:

```
factory CircleParticleNode.initWith(Ranger.Color4<int> from, [double fromScale =
1.0]) {
  CircleParticleNode poolable = new CircleParticleNode.pooled();
  if (poolable.init()) {
    poolable.initWithColor(from);
    poolable.initWithUniformScale(poolable, fromScale);
    return poolable;
  }
  return null;
}
```

A private static property that can be directed at potentially different constructors:

```
static CircleParticleNode _createPoolable() => new CircleParticleNode._();
```

Node API

The api is a combination of overrides, methods and properties.

Overrides

The overrides allow your Node to receive events from Ranger's core system. Below is a list of the most commonly overridden methods. There are others but they are rarely overridden unless you need finer control over timing, cleaning or visiting.

onEnter
: Invoked every time the Node enters the stage. If the Node enters the stage with a transition, this event is called when the transition starts. During *onEnter* you can't access a sibling. If you override *onEnter*, you should call its super.
During this event you should create Nodes, listen to the EventBus and, if it is a Layer, enable inputs.

onExitTransitionDidStart
: Invoked every time the Node begins to enter the stage.

onEnterTransitionDidFinish
: Invoked every time the Node finishes entering the stage. During *onEnterTransitionDidFinish* you can't access a sibling. If you override *onEnterTransitionDidFinish*, you should call its super. If you create any Nodes in this event they won't be visible until the transition finishes. This event is more typically used for starting animations so that the animation can be seen in its entirety.

onExit
: Invoked every time the Node leaves the stage. During this event you should cancel any streams obtained from listening to the EventBus and stop any animations that may be in progress.

update (TimingTarget interface)
: Invoked on each pass through the Scene Graph. A pass is a tick of the clock (aka Core.step). This event gives you an opportunity to process logic or scan inputs such as performing physics, collision detection or running rules.

completeVisit
: Invoked after a complete pass through the Scene Graph. This event gives you opportunity to evaluate the state of a pass incase you need post state information, for example, counting how many Nodes where visible for debugging.

isVisible
: Invoked at the beginning of each visit of a Node which is prior to any transforms or rendering. This event gives you opportunity to run visibility logic to determine if the visit should follow through to make a complete visit or stop and move to the next Node. Typically you would apply some form of axis aligned bounding box test against the view-space. An example is the VisibilityBehavior mix-in.

draw
: Invoked when it is time to render the Node; all transforms have been concatenated at this point. Your draw override is passed a valid rendering context.

addedAsChild
: Invoked when a Node has been added as a child. This only applies to Nodes that have the GroupingBehavior mixed-in. It is useful for monitoring Grouping related events.

pointInside
: Override if your Node has an algorithm for determining if a point is inside. The default is no point is inside.

collide
: If your Node has an algorithm for determining if another Node intersects. The default is nothing intersects, both Nodes would seemly pass through each other.

clone
: Typically overridden if your Node is going to be cloned in high frequency. Particles are an example of clone requirements.

isRunning
: A Node is “running” if it is on Stage otherwise it is not. Mostly used internally.

pooled
: During cleanup if pooled is true then the Node is returned to the pool. The default is false meaning you must “opt in” for the Node to be returned to the pool. Your factory methods will generally set this property to true.

parent
: Every Node has a parent except the root. It is typically set when a Node is added as a child, which implies Scenes can’t have a parent because they aren’t added as children to other Nodes.

visible
: Is the Node visible. The *isVisible* method will set this value based on the *checkVisibility* method. If *isVisible* hasn’t been overridden then this value remains at it default value of true meaning all Nodes are visible by default even if they are out of view.

Methods and Properties

Nodes have various methods and properties that help either work with Nodes or determine the state of the Node.

dirty
: This property changes whenever a transform property is changed, for example, position, scale and/or rotation. Most Nodes will only read the property or ignore it all together, However, some Nodes take control of this flag themselves, for example, ZoomGroup or Particles.

scheduleUpdate
: This method will register **this** Node for periodic updates from the Scheduler. Calling this method depends on your Nodes functionality. Nodes that have either custom animations or physics will usually call this method during the *onEnter* event. Note: if your Node called *scheduleUpdate* make sure to call *unScheduleUpdate* in the *onExit* event otherwise the Scheduler will still call your Node's *update* method even if the Node has exited the stage—this would waste cycles clearly spent on a Node that was actually on stage.

unScheduleUpdate
: This method will unregister this Node from the Scheduler. Call this during the *onExit* event.

calcScaleComponent
: Attempts to determine the current scale within the Node's transform matrix. The algorithm is extremely simple and may not return accurate results.

calcUniformScaleComponent
: Attempts to determine the current scale within the Node's transform matrix. This one is a bit more accurate because it only inspects one of the scale components—namely the X component.

calcScaleRotationComponents
: Attempts to determine the current scale and rotation within the Node's transform matrix. This method is computation intensive because it traverses the matrix stack. Depending on the stack this value may not be accurate. Accurate results require too much computation to make it worth further research. This method isn't really used much if at all.

convertWorldToNodeSpace
: A mapping method that maps a Point in world-space to Node's local-space. You can optionally supply a pseudo root (see Chapter 8) to add a potential performance boost.

convertWorldRectToNode
: A mapping method that maps a Rectangle in world-space to Node's local-space.

convertToWorldSpace
: A mapping method that maps a Point from this Node's space to world-space.

dirtyChanged
: Notify **this** Node that the given Node has become dirty from a transformation. Use full if you want to setup dependencies. Warning: it doesn't check for cyclic dependencies at the moment.

addDirtyListener
: Subscribe as a listener on this Node for dirty changes.

removeDirtyListener
: UnSubscribe as a listener on this Node for dirty changes.

scale (ScaleBehavior)

The Node's scale in the form of a Vector. Automatically sets the Node's dirty flag.

scaleX and scaleY (ScaleBehavior)
The Node's scale in component form. Allows non-uniform scaling. Automatically sets the Node's dirty flag.

uniformScale (ScaleBehavior)
The Node's scale as a uniform value (internally it sets both the X and Y values). Automatically sets the Node's dirty flag.

rotation (RotationBehavior)
The Node's rotation in radians. Automatically sets the Node's dirty flag.

rotationInDegrees(getter) and rotationByDegrees(setter) (RotationBehavior)
The Node's rotation in degrees. Automatically sets the Node's dirty flag.

rotationRate (RotationBehavior)
The Node's rotation rate is animations are in play. WARNING: Does NOT sets the Node's dirty flag.

position (PositionalBehavior)
The Node's position. Automatically sets the Node's dirty flag.

positionX and positionY (ScaleBehavior)
The Node's position in component form. Automatically sets the Node's dirty flag.

moveByComp and moveBy (PositionalBehavior)
Update Node's position by a delta in component or vector form. Automatically sets the Node's dirty flag.

moveToPool (ComponentPoolable)
Returns Node back to pool if the poolable has been marked as poolable otherwise the object is picked up by the garbage collector.

transform
The Node's transform as a result of concatenating position, scale and rotation. It is updated when the dirty flag is true.

tag and name
Useful for both find/locating a Node and to identify Nodes within the scene graph during debugging.

drawOrder
A factory/constructor time value. Setting this value after creation has no affect. It comes in handy if you want to explicitly control render order regardless of the creation order. Negative values render "underneath" positive values.

managedTransform

Some Nodes will manage their own transform matrix. A perfect example is the ZoomGroup Node which sets this property to **true**.

Main Node Types

Ranger comes with several “prebuilt” Nodes. Of those only **Scene** or its variation **AnchoredScene** are required. The other Nodes are optional but it’s almost certain you will use, clone or derive from them, they are: **SceneAnchor**, **BackgroundLayer**, **EmptyNode**, **GroupNode** and **Node**.

Scenes

Scenes are the main containers of Nodes (see Figure 14.1.) Scenes have a primary Node, called *Primary*, that transformations are applied to. *Primary* is either a **GroupNode** or **Layer** Node depending on your requirements.

When a Scene is animated using a **Transition** it is both the *Primary* Node and the Scene Node itself that are acted upon by the animation system.

A Scene will rarely have a visual Node as a direct child, for example a Rectangle, Circle or Sprite. Normally you would add a Layer Node either directly as the primary or as a child of a GroupNode. GroupNodes (covered later) gives you the ability to “stack” multiple layers within the same Scene. The most common case is a background layer with a heads up display (HUD) on top. The GroupNode thus becomes the primary.

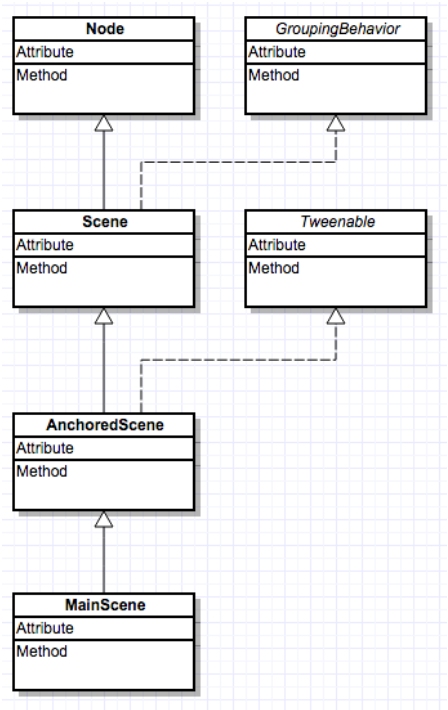


Figure 14.1 Scenes

Hands on

Lets create a super simple Scene that contains a little blue rectangle as a child—no Layers or GroupNodes of any kind. Seen from the perspective of a Scene Graph we would have the arrangement shown in Figure 14.2 right side; the circle in light blue is the first scene to appear after the

Figure 14.2 Simple Scene

BootScene completes, and it has one child called RectangleNode. That is as simple as it gets.

To create our simple scene we first build a basic web app as described above in the section called “Basic App”, except we leave out the SplashScene. The MainScene class is literally about 15 lines of code:

```
part of scene;

class MainScene extends Ranger.Scene {

  MainScene();

  @override
  void onEnter() {
    super.onEnter();

    RectangleNode rectangle = new RectangleNode()
      ..init()
      ..setPosition(500.0, 500.0)
      ..uniformScale = 150.0;

    addChild(rectangle);

  }
}
```

The RectangleNode class is just as simple:

```
part of scene;

class RectangleNode extends Ranger.Node {
  String fillColor = Ranger.Color4IBlue.toString();

  @override
  void draw(Ranger.DrawContext context) {
    context.save();

    CanvasRenderingContext2D context2D = context.renderContext as
CanvasRenderingContext2D;

    context2D..fillStyle = fillColor
      ..fillRect(-0.5, -0.5, 1.0, 1.0);

    context.restore();
  }
}
```

Your *preConfigure* method reduces to just a MainScene and BootScene:

```
void preConfigure() {
  MainScene mainScene = new MainScene()
    ..name = "MainScene";

 Ranger.BootScene bootScene = new Ranger.BootScene(mainScene)
    ..name = "BootScene";

 ranger.sceneManager.pushScene(bootScene);

 ranger.gameConfigured();
}
```

If you run it you would get what you see in Figure 14.2 left side; an orange background with a blue rectangle positioned at 500,500 relative to a Design of 1900x1200. The orange background is the Canvas clear color. The “problems” with this simple scene are:

- The Canvas clear color (aka orange) is visible
- No input capture ability
- No Transitioning ability

We can fix all of these problems in due time, but a first option would be to add Transitioning by simply inheriting from AnchorScene.

AnchoredScene

Typically you would not extend from a Scene class directly but instead extend from **AnchoredScene**. AnchoredScenes provide the backbone for scene transitions both through the **Tweenable** interface and an internal anchor Node. The anchor Node acts as a pivot point around which rotations can be applied and as a center for scaling.

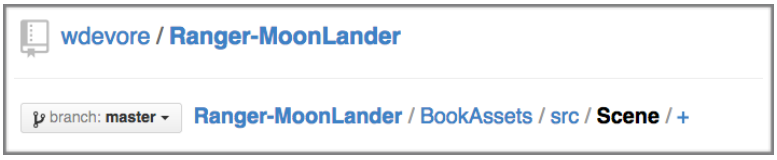
By inheriting from AnchoredScene you can transition to and from any other scene that also inherits from AnchoredScene. Here is an excerpt using the “MoveInFrom” Transition effect:

```
ScoresScene inComingScene = new ScoresScene();

Ranger.TransitionMoveInFrom transition = new
Ranger.TransitionMoveInFrom.initWithDurationAndScene(0.5, inComingScene,
Ranger.TransitionSlideIn.FROM_BOTTOM)
  ..name = "TransitionMoveInFrom";

ranger.sceneManager.pushScene(transition);
```

The ScoresScene is the destination scene we want to transition towards. So we use a “temporary” transition scene called **TransitionMoveInFrom** to transition our current scene to the new ScoresScene—transitions are covered later. This works because ScoresScene also inherits from AnchoredScene.

Code

You can pull the basic Scene code from the book’s assets at Github:



https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Scene

Note: this code *doesn't* inherit from AnchoredScene as it is meant to represent the absolute minimum.

Scene API

anchor
: The anchor is always the first Node added to a Scene. It is primarily used by Transitions, although you are free to transform it yourself.

primaryLayer
: PrimaryLayer can either be an actual Layer or GroupNode. The primary layer is alway translated inversely to the anchor such that the background always cover the view. This keeps the Canvas clear-color from showing. This Node is *always* a child to the anchor Node. This arrangement provides for various Scene transition effects.

focus
: Override this method if your Scene will lose focus without leaving the stage. This can happen if you have a popup dialog appear as an overlay over a Scene. Your overridden focus method should control enablement of your scene's input.

Layers

The **BackgroundLayer** is the main class you will use almost exclusively. This class is "generally" a child of a **Scene** or **GroupNode**. However, there are instances where your Scene may not use one, for example, a splash scene doesn't really need a BackgroundLayer because they typically don't have a need for input or color cascading, and you can create a background fill color simply by overriding the Scene's draw method.

The BackgroundLayer provides three main things:

1. An optionally filled background to cover up the Canvas clearing color.
2. Input mixins for mouse, keyboard and touch.
3. Color cascading.

To add a Layer to a Scene you create a custom layer by extending the BackgroundLayer class:

```
class LevelSelectionLayer extends Ranger.BackgroundLayer {
```

Then create a factory method to property configure a set of basic properties for the layer:

```
factory LevelSelectionLayer.withColor(Ranger.Color4<int> backgroundColor, [bool
centered = true, int width, int height]) {
  LevelSelectionLayer layer = new LevelSelectionLayer()
    ..centered = centered
    ..init(width, height)
    ..transparentBackground = false
    ..color = backgroundColor;
 return layer;
}
```

The more common settings are:

- **Centered**: Basically this marks where the layer's coordinate origin is relative to the Scene.
- **Width** and **Height**: These default to the Design dimensions.
- **TransparentBackground**: If transparent is **true** then you either have another layer that is covering the Canvas or you really like the color orange.
- **Background** color: This is the color that will always cover the layer before anything is rendered onto the layer.

Layer API

The Layer api is a combination overrides, methods and properties.

autoInputs (BackgroundLayer)

The property defaults to t**rue** which means any inputs that have been enabled during the *onEnter* event will be enabled automatically. If it is **false** then Nodes are responsible for manually controlling input. Some Layers control their own inputs for example, popups and dialogs.

constrainBackground (BackgroundLayer)

The property defaults to t**rue** which means the background fill color is constrained to always fill the view regardless of where the layer has been translated.

centered (LayerCascade)
By default Layers are not centered. The origin will either be in the upper-left or lower-right (default).

transparentBackground (BackgroundLayer)
The property defaults to t**rue**. Background layer types will not fill their backgrounds automatically.

setContentSize (BackgroundLayer)
Layers default to the Design dimensions if a width and height are not specified. This is the typical case as you don't want the Canvas clear color to show through.

color (LayerCascade)
Changes the background color as well as iterating through any children changing their background color.

setOpacity (BackgroundLayer)
Changes the background transparency component as well as iterating through any children changing their alpha component.

setColor (BackgroundLayer)
Changes the background color components as well as iterating through any children changing their color components.

cascadeOpacityEnabled (RGBACascadeMixin)
By default color and transparency cascading is disable by default. Only the Node's properties are changed.

displayedOpacity (RGBACascadeMixin)
This is the displayed value of the background value versus an intermediate animated transparency. Values are 0 -> 255.

displayedColor (RGBACascadeMixin)
This is the displayed color of the background color versus an intermediate animated color. Values are 0 -> 255.

enableInputs (BackgroundLayer)
Enable any inputs that are currently eligible.

disableInputs (BackgroundLayer)
Disable any inputs that are currently eligible.

Overrides

onMouseDown, onMouseMove, onMouseUp, onMouseWheel (MouseInputMixin)

Override to receive events for Mouse.

onKeyDown, onKeyUp, onKeyPress (KeyboardInputMixin)
Override to receive events for Keyboard.

onTouchStart, onTouchMove, onTouchEnd, onTouchCancel (TouchInputMixin)
Override to receive events for Touch.

GroupNode

Nodes, by design, are Leaf nodes, they can't contain children, TextNodes are an example of a Leaf type. The GroupNode solves this by mix-ing in GroupingBehavior. The GroupNode has no visual aspect, it merely collects other Nodes. Scenes and Layers also mix-in GroupingBehavior but they provide a whole host of other functionality. The GroupNode's sole job is to collect Nodes and nothing else.

GroupNode API

The API is based on the GroupingBehavior mix-in.

hasNegZOrders (GroupingBehavior)
Controls whether to check for Nodes with negative zOrder values. If nothing more that a performance flag.

enableParentDirting (GroupingBehavior)
Controls whether the dirty flag propagates upward towards the parent, it is *false* by default.

updateTransforms (GroupingBehavior)
Manually update transforms of all children.

rippleDirty (GroupingBehavior)
Performs a full traversal of the children marking them dirty.

getChildByTag (GroupingBehavior)
Find a child Node by Tag value. Tags can also be helpful for debugging.

addChild (GroupingBehavior)
Add a Node as a child to the Node. An optional zOrder value can be given. To change the zOrder after addition you need to call *reorderChild*. The second option of changing rendering order is to preform a remove/add sequence.

addChildAt (GroupingBehavior)
Add a Node at a specific ordinal position. Helpful if you wish to temporarily remove a Node and re-add it to the same ordinal position.

removeChild (GroupingBehavior)
Removes a child from the collection and returns the ordinal position that it had prior to removal. There are two ways to remove a child: 1) Simply remove it without cleanup.

However, scheduling still occurs and as such it will still receive timing but nothing else. 2) Remove and cleanup (default). This completely removes the Node.
There maybe times when you want a Node removed from the Scene Graph yet still receive active timing. When a Node is removed from the Scene Graph it allows the graph to spend more cycles on Nodes that are within its control. Simply making a Node invisible still causes the graph to spend cycles on the Node in order to check for visibility.

removeAllChildren (GroupingBehavior)
Removes all Nodes from graph even if some of them are Scenes or Layers.

reorderChild (GroupingBehavior)
Reorder all the children of this collection after the given Node's zOrder was changed.

EmptyNode

A simple Node that has no visual. It is meant as a place holder and is typically used for tracking. For example, to track the position in one Space from the perspective of another Space. We saw an example of this in the code for lander's exhaust particles.

Chapter 15 Simplicity

In this chapter we describe a slew of examples. Each example shows a specific aspect of Ranger. Ranging from a simple Layer to something as complex as zooming. In each case the focus is on a specific concept or task.

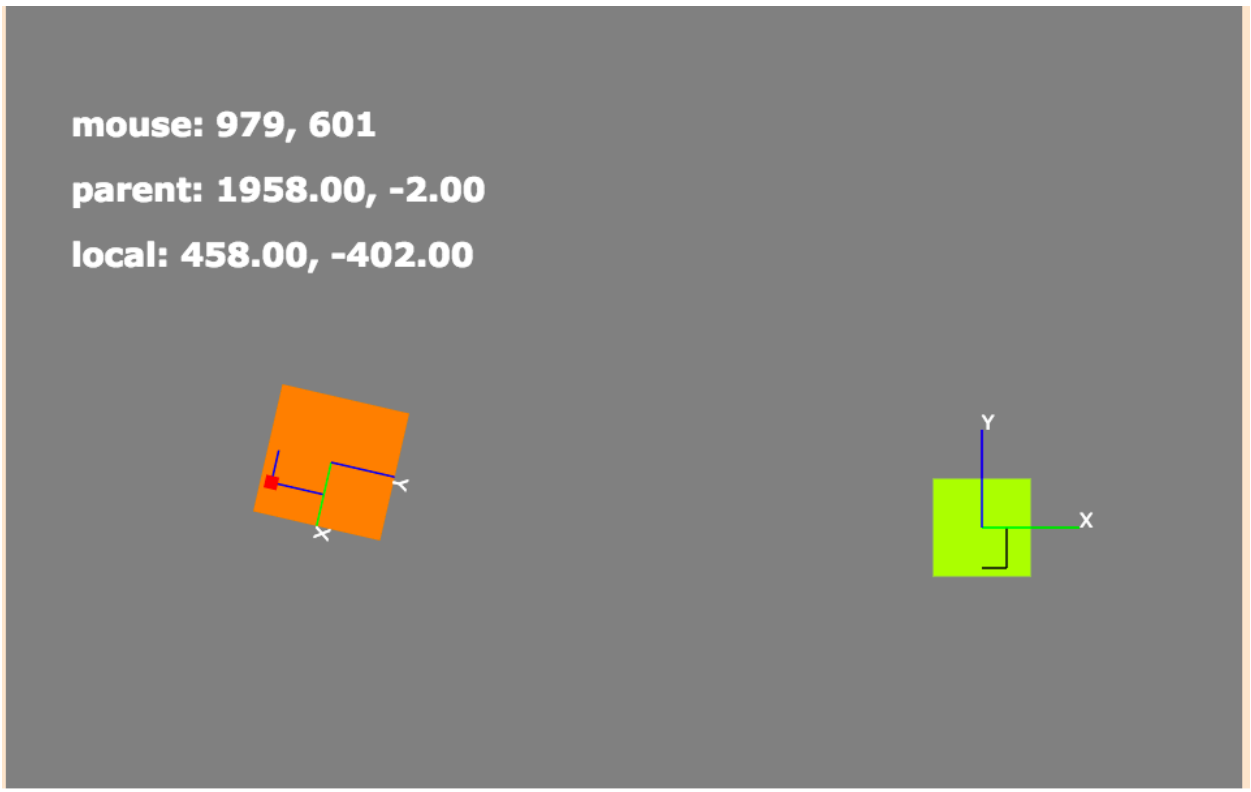


Figure 15.0 Simplicity

Example (1): Layer

Our first example is to put together a really simple scene with a BackgroundLayer that recognizes mouse down clicks. Note: going forward, when I say scene I am implying a Scene and Layer arrangement.

Here are the steps that produced the Layer example:

- Create a simple web app (called "**Uber simple web application**" in WebStorm)
- Include a Git dependency for Ranger in pubspec.yaml
  - ranger:
  - git: git://github.com/wdevore/ranger-dart.git
- Add imports to main.dart for both Ranger and TweenEngine
  - import 'package:ranger/ranger.dart' as **Ranger**
- Replace ** ** with **<Div id="gameSurface" width="1000" height="600"></Div>**. Note: replace the **width** and **height** attribute values with values that match your requirements.
- Add a Ranger.Application var to main.dart
- Refactor *main*() create a Ranger Application using one of the factories. The example uses *fitDesignToWindow*.
- Add a library name called "layer".
- Create a MainScene Scene and MainLayer BackgroundLayer.
- Update MainScene to create MainLayer and add layer as child.
- Update MainLayer to override *onEnter* and enable Mouse input.
- Update MainLayer to override *onMouseDown* to print "clicked".
- Update main.dart to include the scene and layer using Dart's **part** statement.
- Add a preConfigure callback method to configure the boot sequence using MainScene and Ranger's **BootScene**.

When you run it you will see a grey background, and touching anywhere will print ("clicked") in the Debug console.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Layer

Example (2): Spinning Rectangle

This example shows how to create an animation using Ranger' TweenAnimation helper, and also how to start and stop an animation. The animation is a spinning rectangle. Every time you click somewhere the spinning stops or starts.

- Follow the steps from Example 1 or clone it.
- Create a new custom Node called **RectangleNode** and color it **Orange**.
- In the Layer class:
  - Create RectangleNode in the *onEnter* event of the Layer and add it as a child.
  - Call TweenEngine's *registerAccessor* method to register the RectangleNode. This notifies TweenEngine that RectangleNode is going to be animated.
  - Create a method to animate the Rectangle using Ranger's TweenAnimation for simplicity.
  - This method stops and un-tracks animation in progress first.

    - If a _rotate property is true then create a new animation and start it animating immediately.
    - Track the new animation such that it can be stopped and untracked later.
- Import rectangle_node.dart in main.dart

When you run Example 2 you will see a grey background with a rotating orange rectangle. Clicking anywhere to start or stop the rectangle.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex2_SpinRect

Example (3): Clicking on a Node

This example shows how to detect clicking on a Node. In this example the RectangleNode was modified to add hit detection by overriding Node's *pointInsideByComp* method.

- Follow the steps from Example 2 or clone it.
- Modify RectangleNode to include a Ranger.MutableRectangle (called _bbox) to represent the local-space bounding box, note that this rectangle isn't an axis aligned bounding box (aabbox) but a local-space bounding box. This is an important concept to understand. As the rectangle rotates you will notice that clicking in a area where there isn't orange will yield no click and that is because the local-space bbox has been rotated.
- Override Node's *pointInsideByComp* and call _bbox's *containsPointByComp* method returning the boolean from the results.
- In MainLayer add a call to drawContext's *mapViewToNode* using the event's ***offset*** property as input. This will map the mouse location to the local-space of the Node provided and in this case it is the rotating rectangle.
- With the local-space location we can finally call the rectangle's new *pointInsideByComp* method to check for a hit. If we got a hit we call the *_rotateNode* method as before.

When you run Example 3 you will see a grey background with a rotating orange rectangle. Clicking on the orange area toggles the rotation. Notice that as the rectangle rotates areas around the rectangle are grey and by clicking there nothing happens. This is because the mouse was mapped to the local-space of the rectangle which itself has been rotated.

If you need to click on a aabbox then you take the rotated local-space bbox and apply its min and max points to the a second parent aabbox.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex3_ClickOnNode

Example (4): Transforming a Node

This example shows how to transform a Node using scale, rotation and translation. It uses the keyboard to control each transform.

Keys 1 and 2 changes Scale.

Keys 3 and 4 changes orientation (aka Rotation).
Keys 5 and 6 Translate along X axis.

- Clone Example 2 and then remove the animation code.
- Update *onEnter* to enable keyboard instead.
- Override the *onKeyDown* method from the **KeyboardInputMixin** class and check for the event.keyCode value.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex4_Transforming

Example (5): Node hierarchy

This example shows how Nodes can be formed into hierarchies and how animations function relative to the hierarchy. In this example there are three rectangles, a parent orange rectangle which has a blue child rectangle of which the blue rectangle has a child green-yellow rectangle.

Keys 1 and 2 rotate the orange rectangle.
Keys 3 and 4 rotate the blue rectangle.
Keys 5 and 6 rotate the green-yellow rectangle.
Keys 7 and 8 scale the orange rectangle.
Keys A and S translate the blue rectangle on the blue rectangle's local X axis by 1.0 unit.
Keys A and X translate the blue rectangle on the blue rectangle's local X axis by 10.0 units.

- Clone Example 4.
- Renamed rectangle to orangeRect
- Added a blueRect and greenYellowRect
- Refactored RectangleNode to mix-in GroupingBehavior and implement a constructor that initializes the behavior.
- Construct and configure all three rectangles in the *onEnter* method. Make sure that the orange rectangle adds the blue rectangle as a child, for example, orangeRect.addChild(blueRect), and do the same relationship with the blue and green-yellow rectangles.

Notice when you rotate a parent rectangle the children rotate in sync.

Things to understand in this example are:

- Transforms propagate downward. For example, scaling the orange rectangle also scales the children.
- Transforms are relative to a parent. For example, translating the blue rectangle's X position is "seen" as translating on the orange rectangle's local X axis. In other words the blue rectangle's transforms are relative to it parent (the orange rectangle.)
- Because the rectangles local dimensions are unit size (aka 1.0 x 1.0) translating a child by 1.0 is really translating by the (parent's scale * 1.0). This means the parent "scales" children transforms. In this example the orange rectangle is scaled by 150.0 meanings a translation by a child (aka blue rectangle) by 1.0 will yield a translation of 150.0. If you want to translate

the blue rectangle by 1.0 you would divide by the parent's scale. This is what the Z and X keys do.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex5_NodeHierarchy

Example (6): Alpha Fade

This example shows how a Node's transparency is animated. To do this we refactor RectangleNode to extend ***Tweenable*** and then code the get/set properties to accomplish our goal.

- Clone Example 4.
- Refactor RectangleNode to extend TweenEngine's ***Tweenable*** interface.
- Change the fillColor property from a String to Ranger's **Color4**<int> type.
- Implement *getTweenableValues* and *setTweenableValues* to modify the fillColor's alpha property.
- Create a tween animation that targets the orange rectangle and fades in and out under a 1 second interval using an exponential easing rate.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex6_AlphaFade

Example (7): Color Tint

This example shows how a Node's color can be animated between two colors. To do this we refactor RectangleNode to change the r,g,b components.

- Clone Example 6.
- Refactor *getTweenableValues* and *setTweenableValues* to modify the fillColor's r,g,b component properties.
- Create a tween animation that targets the orange rectangle and tint in and out under a 1 second interval using an exponential easing rate.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex7_ColorTint

Example (8): Local-Parent mapping

This example shows both the local and parent position of the mouse relative to the background layer and rectangle. Three TextNodes are used to display the local-space, parent-space and mouse-space coordinates.

Keys 1 and 2 changes Scale.
Keys 3 and 4 changes orientation (aka Rotation).
Keys 5 and 6 Translate along X axis.

- Clone Example 4.
- Create three TextNodes and position above rectangle.
- Enable mouse along side currently enabled keyboard.
- Override *onMouseMove* update text nodes by mapping mouse to rectangle and layer.

Notice as you move the mouse the local-space coordinates show relative to the rectangle even if you scale or rotate it.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex8_LocalParentMapping

Example (9): Local-Parent mapping enhanced

This example enhances Example 8 by adding tracking elements to help emphasize the local-space “nature” of the orange node.

Keys 1 and 2 changes Scale.
Keys 3 and 4 changes orientation (aka Rotation).
Keys 5 and 6 Translate along X axis.

- Clone Example 8.
- Create two TextNodes for axis notation.
- Create two LineNodes and RectangleNode for visual tracking.

Notice as you rotate the rectangle how the local-spaces rotates.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex8_LocalParentMapping

Example (10): Inter Node tracking

This example shows how a local-space position is mapped into the local-space of another Node. One rectangle is fixed while the other is rotating. An affine transformation matrix is created that maps the yellowGreen rectangle’s local-space coord to the local-space of the orangeRect.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex10_InterNodeTracking

Example (11): Sprite

This example shows how to load a Sprite. The resource for the sprite is an embedded base64 SVG icon.

- Clone Example 2.
- Refactor to replace RectangleNode with a Ranger SpriteImage.
- In *onEnter* create an ImageElement using Ranger's BaseResources spinner.
- Create a SpriteImage from the ImageElement and apply an animation.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex11_Sprite

Example (12): Async Sprite loading

This example shows how to async load a Sprite. It uses a simulated network delay as a method to delay loading. A placebo is shown while loading. This placebo is an embedded base64 resource that is always immediately available.

- Clone Example 11.
- import 'dart:async';
- Use Ranger's ImageLoader to async load a resource.

Click anywhere to start the loading.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex11_Sprite

Example (13): MultiFrame sprite animations

This example shows one possible way to animate frames of a sprite atlas. The animation is controlled by Ranger's **SpriteSheetImage** class combined with a simple JSON config file.

- Use a **SpriteSheeImage** class to manage the atlas and read the configuration.
- Use a **CanvasSprite** handle the animation.

This example is just one simple way to work with Sprite atlases, but there are many many more.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex13_MultiframeSpriteAnimation

Example (14): HUD

This example shows how to put together a simple Heads-up-display. A HUD is basically an overlay above another Layer meant to isolate transforms that are occurring in a background layer. The example places a TextNode on the HUD and a rectangle of the background. The background is then scaled up and down. This demonstrates that the TextNode is unaffected by the background scaling.

HUDs are typically used for visual controls overlaid above a background. Examples, are frames-per-seconds information or flight controls of a space ship.

- Use a GroupNode as the primary.
- Make MainLayer a child first.
- Create a new Layer, called HUD, and make it a child second.
- Add a TextNode to the HUD layer.
- Create a Scale animation on the MainLayer.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex14_HUD

Example (15): Scene transition

This example shows how to transition back and forth between two scenes: MainScene and SecondScene. Main Scene has the orange rectangle on a grey background and the SecondScene has the green-yellow rectangle on a light blue background. This example also shows how to properly “dispose” of animations as Scenes enter and exit the stage.

Notice in this example I use the *onEnterTransitionDidFinish* method to start the rotating animations. This is because I choose to use the *flushAll* in the *onExit* which also happens to cancel the incoming Scene’s animations because incoming scenes *onEnter* is called “before” the Layer enters the stage and “before” the out going scene has exited the stage. What this means is that you need to decide when and where you want to do your work.

The most important thing to learn from this, aside of learning about Transitions, is that you should cancel any animations when a scene exits the stage. If you don’t cancel animations those Nodes will still receive timing events thus wasting cycles on Nodes that aren’t even on stage.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex15_SceneTransition

Example (16): Zooming

This example shows how to use the ZoomGroup Node. For visibility the ZoomGroup's icon is made visible and scaled up, it shows as a white cross. Each time you click you either zoom-in or zoom-out at the mouse position.

- Copy the ZoomGroup class from one of Ranger Sack's examples.
- Move all Nodes as children of ZoomGroup.
- Set scale center where you want to zoom.

Notice when you click that zooming occurs at the cursor point. You can modify this behavior to instead zoom about some point dictated by some other Node, for example, a Zone type object. To zoom in (things appear bigger) you use a zoom value > 1.0. To zoom out (things appear smaller) you use a zoom value < 1.0. You aren't restricted to any particular animation style, use whatever is best for you.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex16_Zooming

Example (17): Dragging

This example shows how to drag a Node using the mouse. Dragging a Node is the same as moving it by a delta. You can drag a Node by modifying its local-space position or its position relative to a parent (parent-space.) This example shows the more typical approach of translating relative to the parent.

- Override both *onMouseMove* and *onMouseUp*.
- Mouse-move calculates the delta from the previous move and the current one.
- Mouse-up reset the dragging flag.

Click and drag on the orange rectangle.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex17_Dragging

Example (18): Zones and Dragging

This example shows how zones work by dragging a rectangle in to and out of two different Zones.

- Copy DualRangeZone from Ranger-Sack.
- Create two zones and add them as children to MainLayer.
- Listen on EventBus for DualRangeZone events.

- Upon an Enter event start an animation.
- Upon an Exit event stop an animation.

Click and drag on the orange rectangle.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex18_ZoneAndDragging

Example (19): Visibility

This example demonstrates Node visibility relative to a simulated viewport Node. Typically the viewport Node would be on a separate Layer away from any type of zooming. Whenever the zoom changs the viewport's bbox would be remapped. In this example there is no zooming and thus the viewport is on the same Layer as the orange rectangle. The example is simply to show how intersection can be combined with visibility.

The example has two rectangles that move synchronously. Dragging the outlined rectangle will drag both rectangles. This allows you to see something as you drag. The other rectangle (orange) has its visibility adjusted depending on the intersection with the viewport Node. Notice that the orange rectangle "disappears" when NOT intersecting the viewport. This is because invisible Nodes are not visited which means they are not rendered either, this includes any children of the invisible Node.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex19_Visibility

Example (20,a-g): Transitions

See Transitions chapter.

Example (21): Hierarchal arrangements

This example demonstrates hierarchal arrangements of Nodes through the use of GroupNodes. By using GroupNodes we can create "anchors" for Nodes to transform around. Look at the **Planet** class, there you will see a GroupNode that holds the planet (aka RectangleNode). This allows you to apply a rotation on both the GroupNode and RectangleNode creating the illusion of both an orbit and rotation.

Example (22): Drag and Zoom

This example demonstrates dragging Nodes with zoom applied. Dragging is done on the **GroupDragNode** and Zoom is done with the **ZoomGroup** Node. The main concept to learn in this example is what happens in the *onMouseMove* method. Notice that the mouse location is being mapped to the ***_dragGroup*** Node but the delta calculated is being applied to the ***_zoom*** Node. The reason stems from the usage of the ZoomGroup Node. Once you start using the

ZoomGroup to manage zooming you also need to use it for translations as well meaning you wouldn't call the ZoomGroup's base *moveBy* method but instead call the ZoomGroup's *translateByComp* method. This is the most important concept to garner from this example. You will see this usage again in the DragZoomScroll example.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex22_DragZoom

Example (23): Drag, Zoom and Scroll (Intermediate)

This example is an intermediate level demonstration of scrolling using a rectangle as an edge detection device. There are many ways to implement scrolling, however, this example uses a dual group approach making scrolling much easier.

So how does it work? Take a look at Figure 15.1, notice how ***_dragGroup*** has two GroupNodes as children. These two groups provide for a "lift and drop" type of drag. From a top level here is the sequence of events for a complete drag:

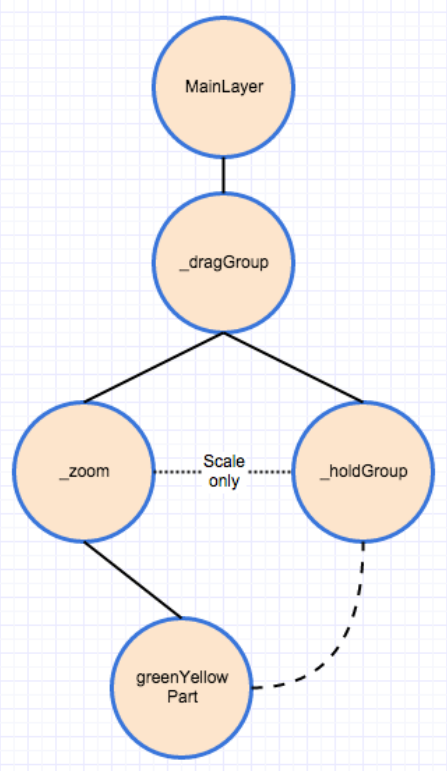


Figure 15.1 Dual Groups

1. On mouse down we take the current scale of the zoom group and copy it to the hold group—to keep the example simple only Scale was copied.
2. Map the rectangle's position (prior to removal) into Parent-space (aka _dragGroup space).
3. Map from Parent-space to the local-space of the hold group.
4. Now remove the rectangle from the zoom group record its original ordinal position such that we can add it back under its original position. Note, make sure you set the **cleanUp** flag to "**false**" otherwise your Node will not only be removed from the scene graph but will also be deleted.
5. Add the orphaned rectangle to the hold group.
6. Finally position the rectangle to the mapped position we got before we removed it from the zoom group.
7. Drag the rectangle around checking for scrolling. If scrolling needed then calculate distance from scroll edge and translate zoom position in the opposite direction.
8. On mouse up we perform nearly the opposite of the mouse down. We map the rectangle from hold group space back into Parent-space.
9. Map from Parent-space into zoom group space.
10. Remove rectangle from hold group remembering to set the **cleanUp** flag to false otherwise we lose the rectangle too.
11. Add the rectangle back to the zoom group making sure to use the ordinal position we obtained when we first removed it from zoom group during the mouse down event.

12. Finally position the rectangle using the mapped position.

The idea is that when we drag a rectangle we temporarily move it to a GroupNode. This effectively “isolates” the rectangle from any transforms being applied to the zoom group namely scrolling translations. When we are done dragging we map and move the rectangle back to the zoom group. But why all this work just to scroll? Simple, our goal was to drag a rectangle past a boundary that triggers scrolling. If the rectangle was still on the zoom group Node during scrolling then the rectangle would be “yanked” away from the cursor. You may think that applying an inverse translation to the rectangle would solve the problem but that would only cause the rectangle to “freeze” right at the scroll edge—not really what we want, we want to be able to freely grab the rectangle and have it “stick” to the cursor during dragging *and* scrolling. The easiest way to do that is move the rectangle out of the line-of-fire (aka scrolling group) and then plop it back in when done dragging.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex23_DragZoomScroll

Example (24, 25, 26): Complex Node (Intermediate)

These examples are an intermediate level demonstrations of a complex Node. Example 24 accomplishes two things: alpha clearing using a fractional alpha and a Node with a complex rendering algorithm. To do this the MainLayer does two things to prepare the alpha effect: first it clears the background once and only once and then tells the SceneManager to bypass clearing the background.

Why don’t we clear the background? The alpha trail effect works by repeatedly clearing the background with a translucent color, in this case translucent black with a small alpha value. On each render pass the background is ever so slightly cleared with an accumulating translucent color.

The Node is complex because it contains a complex algorithm for rendering itself— the Lissajous curve. Lets face it you could literally create a game inside a Node if you really wanted to!

Example 25 is an example of rendering without any background clearing at all. Example 26 is a form of art controlled in a single node.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex24_ComplexCustomNode
https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex25_ComplexHexNode%20
https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex26_ComplexArtNode

Chapter 16 Transitions

In this chapter we build a new Transition called Chomp.

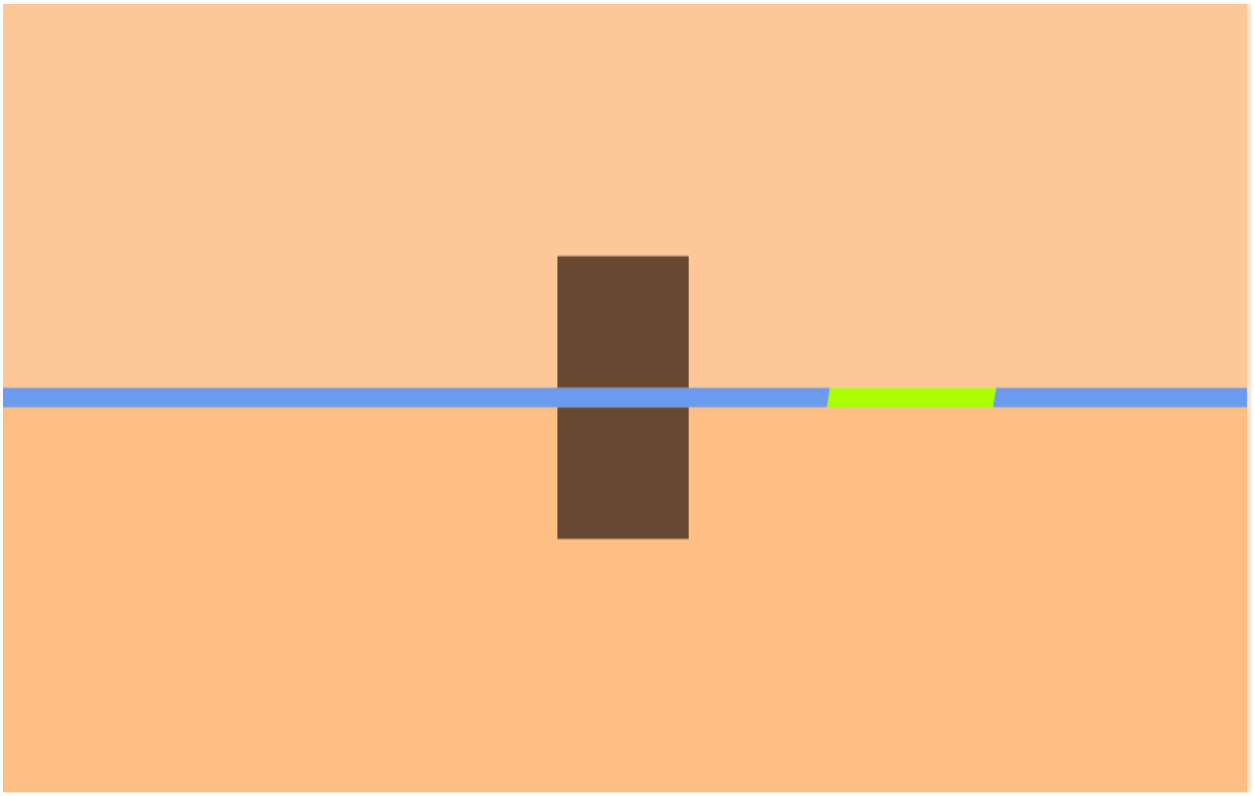


Figure 16.0 Chomp

Chomp isn't the only transition of its type, there is also: Slats, Blade, Swipe and Iris all available alongside Chomp.

Transition review

Transition Scenes are a transient type of Scene. They only appear on stage during the duration of the transition. Some Transitions are transparent by operating only in the “background” as a coordinator, for example all the Transitions that are part of Ranger. But that doesn’t mean others are transparent, for example the ChompTransition we will are building in this chapter is *very* visible (see Figure 16.0.)

Take a look at Figure 16.1. During any transition there are three Scenes active at the same time: outgoing, incoming and the Transition scene itself.

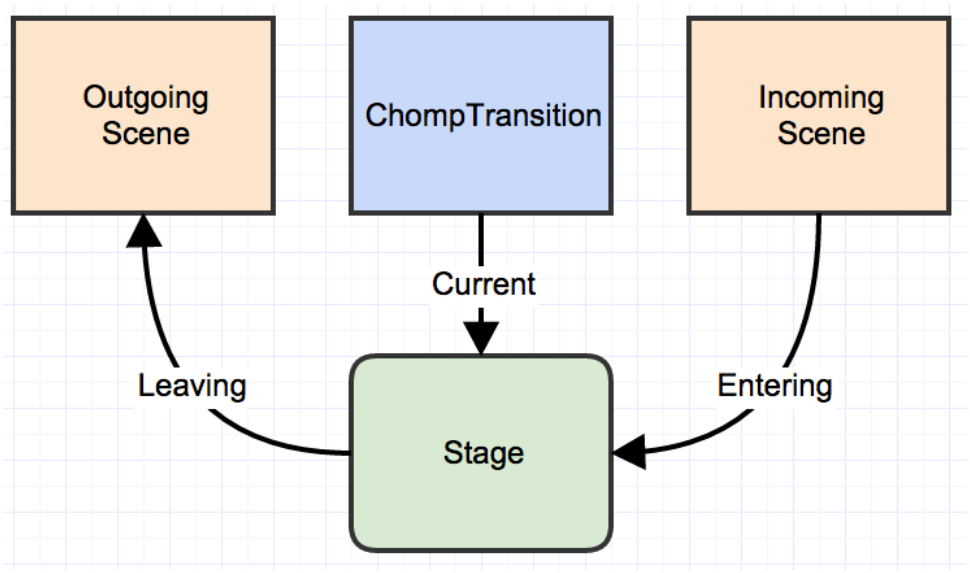


Figure 16.1 Transition flow

They all must be active so each can do their part. The outgoing scene needs to be active in order to exit the stage properly, the incoming scene needs to be active in order to enter the stage properly. The Transition Scene is responsible for managing how each enters and exits.

When the Transition has completed its final task it calls *finish*. Once this happens the Transition disappears into the void which is why they are considered transient.

As mentioned above some Transitions do not have a visual aspect, meaning they have no Nodes and/or Layers. Take for example Ranger’s **TransitionMoveInFrom** transition. This transition simply “pushes” the incoming scene out of view by setting the incoming scene’s position clearly off in the distance along the X axis:

```
inScene.setPosition(-Application.instance.designSize.width, 0.0);
```

It then animates it back into view using a Tween:

```
seq.push(app.animations.moveTo(inScene, duration, 0.0, 0.0, UTE.Sine.INOUT,
AnchoredScene.TRANSLATE_X, null, false));
```

When the animation completes the *_finishCallFunc* is called which intern calls *finish* thus ending the transition:

```
void _finishCallFunc(int type, UTE.BaseTween source) {
  finish(null);
}
```

TransitionMoveInFrom has no visual Nodes whatsoever, however, our new custom Transition called ChompTransition will.

ChompTransition

Recall that Transitions are thinly veiled Scenes, and just like Scenes Transitions can optionally contain visual Nodes. Chomp is going to be one of those types that have visual Nodes namely two large rectangles acting as doors that animate from the top and bottom meeting in the middle.

To make things easier I cloned Example 15, it's already setup for transitioning between two scenes. Both **MainLayer** and **SecondLayer** will have changed very little, the only code that actually changes is the code that was creating the **TransitionMoveInFrom**:

```
Ranger.TransitionMoveInFrom transition = new
Ranger.TransitionMoveInFrom.initWithDurationAndScene(0.5, inComingScene,
Ranger.TransitionSlideIn.FROM_LEFT);
```

Creating Chomp

With the cloned project I created a new class called **ChompTransition** that extends Ranger's **TransitionScene**:

```
part of layer;

class ChompTransition extends Ranger.TransitionScene {
...
```

For the chomp effect I needed a custom Node that is shaped like a rectangular door and can be animated. I decided that the doors will animate from top and bottom much like your mouth works. This means the doors needs to be half the size of the Design size, and because I want the doors to have a buckle I need **GroupingBehavior** on top of **Tweenable**:

```
class DoorNode extends Ranger.Node with Ranger.GroupingBehavior, UTE.Tweenable {
...
```

The door has two rectangle Nodes: one is the actual door and the other is the buckle:

```
RectangleNode rectangle = new RectangleNode();
RectangleNode buckle = new RectangleNode();
```

As usual I override the *init* method to initialize both rectangles:

```
@override
bool init() {
  if (super.init()) {
    rectangle.init();
    buckle.init();
...
```

The *onEnter* adds the Nodes as children:

```
@override
void onEnter() {

  addChild(rectangle);
  addChild(buckle);
...
```

And finally the Tweenable api is implemented. The setter updates the Node's position accordingly:

```
void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) {
  switch (tweenType) {
    case TRANSLATE_X:
      positionX = newValues[0];
      break;
    case TRANSLATE_Y:
      positionY = newValues[0];
      break;
  }
}
```

Not much to it as you can see. Returning to **ChompTransition** class I added the doors and include some state properties for controlling how long the doors pause while closed:

```
DoorNode _upperDoor;
DoorNode _lowerDoor;
double pause = 0.0;
double _pauseCount = 0.0;
bool _opening = false;
bool _closing = true;
```

I added the ubiquitous factory for constructing the transition:

```
factory ChompTransition.initWithDurationAndScene(double duration, Ranger.BaseNode
scene) {
  ChompTransition tScene = new ChompTransition()
    ..initWithDuration(duration, scene);
  return tScene;
}
```

In the *onEnter* method I created and configured both doors such that they are placed out of view for later animation *into* view:

```
_upperDoor = new DoorNode()
  ..rectangle.fillColor = Ranger.color4IFromHex("#fcc89b").toString()
  ..setPosition(0.0, ranger.designSize.height)
  ..width = ranger.designSize.width
  ..height = ranger.designSize.height / 2.0
  ..buckle.setPosition(ranger.designSize.width / 2.0 - buckleSize /2.0, 0.0)
  ..buckle.fillColor = Ranger.color4IFromHex("#674736").toString()
  ..buckle.uniformScale = buckleSize;
addChild(_upperDoor);
...
```

I also made sure the incoming scene was invisible until the doors closed:

```
inScene.visible = false;
```

Next I started the animations on the doors:

```
UTE.Tween doorDown = new UTE.Tween.to(_upperDoor, DoorNode.TRANSLATE_Y, duration)
  ..targetRelative = -_upperDoor.height
  ..easing = UTE.Sine.OUT
  ..callback = _closeComplete
  ..callbackTriggers = UTE.TweenCallback.COMPLETE;
ranger.animations.add(doorDown);
...
```

Finally I scheduled timing so I can time the doors pausing:

```
scheduleUpdate();
```

Because I scheduled updates I need to override the *update* method so I can track the pause duration

```
@override
void update(double dt) {
  if (_closing)
    return;

  _pauseCount += dt;
 if (_pauseCount > pause && !_opening) {
    _opening = true;
    UTE.Tween doorDown = new UTE.Tween.to(_upperDoor, DoorNode.TRANSLATE_Y,
duration)
      ..targetRelative = _upperDoor.height
      ..easing = UTE.Sine.IN
      ..callback = _openComplete
      ..callbackTriggers = UTE.TweenCallback.COMPLETE;
    ranger.animations.add(doorDown);

    UTE.Tween doorUp = new UTE.Tween.to(_lowerDoor, DoorNode.TRANSLATE_Y,
duration)
      ..targetRelative = -_lowerDoor.height
      ..easing = UTE.Sine.IN;
    ranger.animations.add(doorUp);
  }
}
```

Each animation has a completion and we need to act upon them to control sequencing. There is completion for both close-complete and open-complete:

```
void _closeComplete(int type, UTE.BaseTween source) {
  if (type == UTE.TweenCallback.COMPLETE) {
    _closing = false;
    _pauseCount = 0.0;
    inScene.visible = true;
  }
}

void _openComplete(int type, UTE.BaseTween source) {
  if (type == UTE.TweenCallback.COMPLETE) {
    finish(null);
  }
}
```

The last thing to do is return to each layer and update the code to create our new custom transition instead of the current one. Here is the MainLayer's *onMouseDown* which is identical to SecondLayer's:

```
@override
bool onMouseDown(MouseEvent event) {

  // Transition to the second scene
  SecondScene inComingScene = new SecondScene();

  ChompTransition transition = new ChompTransition.initWithDurationAndScene(0.35,
inComingScene)
    ..pause = 0.5;

  ranger.sceneManager.replaceScene(transition);

  return true;
}
```

If you run the example and click anywhere the transition will start, and each time you click again it transitions back the other way. Sweet!

ChompTransition is an excellent example of a Scene having visual Nodes. We could have used a heavy **BackgroundLayer** but there was no need because Transitions are lightweight requiring no input all while being short lived.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex20_ChompTransition

Bonus:

Code

There are whole bunch of extra example transitions in the same folder: Slats, Blade, Swipe, Iris, Pixels, Bars and FadeBlocks. They all function very similar to Chomp.

Chapter 17 Tween Animations

In this chapter we cover animations using both Ranger's TweenAnimation, a wrapper around

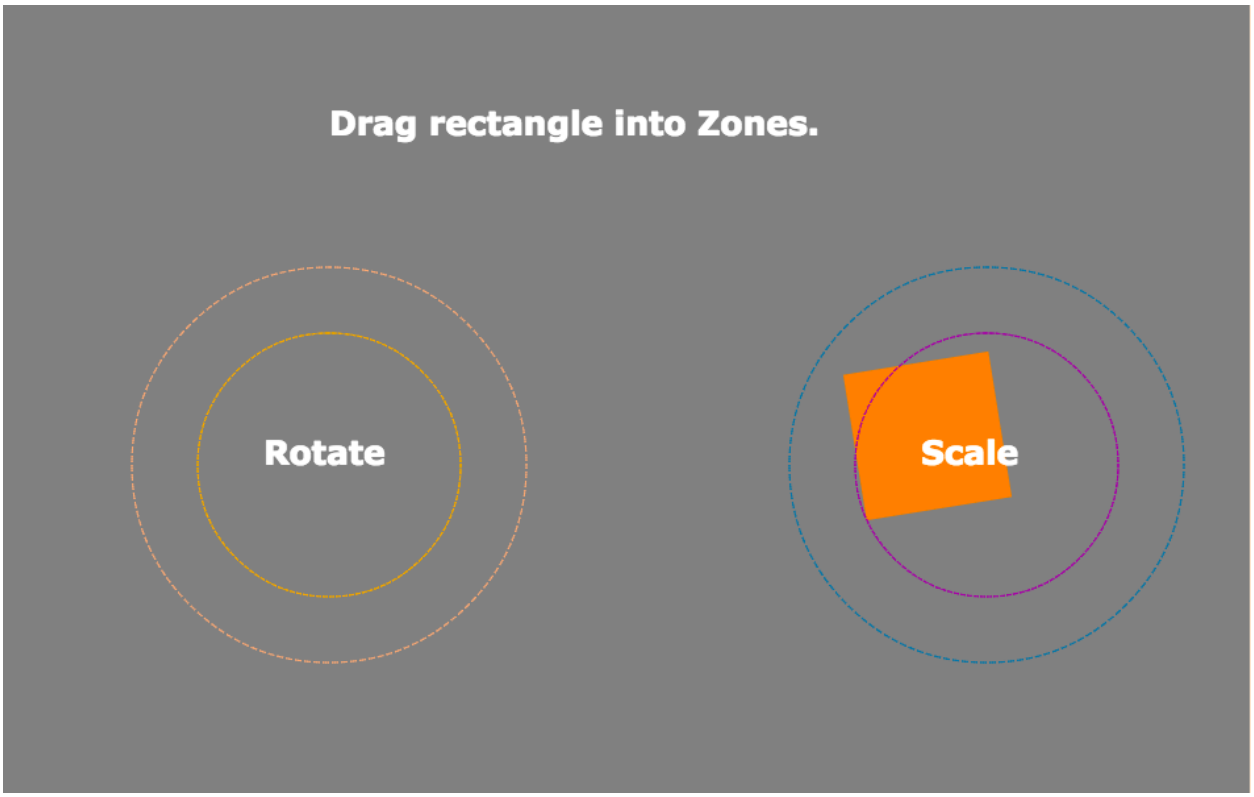


Figure 17.0 Zones and dragging

TweenEngine and TweenEngine directly.

TweenEngine (TE)

TweenEngine is a port of Universal Tween Engine. With it you can animate just about any object you have access to. It has three main components of which only two do you need to be directly aware of relative to Ranger.

TweenManager

The first component is **TweenManager**. This component is created and managed by Ranger's TweenAnimation class. It is the component that handles the timing for your animations. Technically you can have several of them going at the same time but you really only need one and "that" one is managed by the TweenAnimation object that you access via Ranger's **Application** class:

```
Ranger.Application ranger;

void main() {
  ranger = new Ranger.Application.fitDesignToWindow(
...
```

With the Ranger application object you can add tween animations:

```
ranger.animations.add(tw);
```

Tweenable

The second component is the **Tweenable** interface. You extend this interface when you have access to the code of the thing you want to animate. It is the preferred way to animate objects as you can closely tie animation code with Nodes. You can see an example of this approach in Ex7_ColorTint example. In that example the RectangleNode extended the Tweenable interface and directly modified the Node's fillColor:

```
class RectangleNode extends Ranger.Node with UTE.Tweenable {
  ...
  void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) {
    switch (tweenType) {
      case TINT:
        fillColor.r = newValues[0].ceil();
        fillColor.g = newValues[1].ceil();
        fillColor.b = newValues[2].ceil();
        break;
    }
  }
...
```

Also notice in the example that we didn't need to register anything we simply created an animation that directly specified RectangleNode (aka orangeRect) as the target, also notice that we still used TweenAnimation's TweenManager by calling the *add* method:

```
UTE.Tween tw = new UTE.Tween.to(orangeRect, RectangleNode.TINT, 1.0)
  ..targetValues = [toColor.r, toColor.g, toColor.b]
  ..easing = UTE.Expo.INOUT
  ..repeatYoyo(UTE.Tween.INFINITY, 0.0);

ranger.animations.add(tw);
```

TweenAccessor

The third component is **TweenAccessor**. This component is designed so that a class can act as a liaison on behalf on some object that you either do not have source access or simply don't want to modify code. This way an object can be animated without the object actually knowing that it is being animated.

As a matter of fact **TweenAnimation** is one of these:

```
class TweenAnimation extends TimingTarget implements UTE.TweenAccessor<Node> {
```

As you can see above TweenAnimation is a specific type of TweenAccessor namely **Node**.

TweenAnimation (TA)

TweenAnimation (see Figure 17.1) is Ranger's simplistic wrapper around (TE). It provides a typical, yet simple, set of canned animations, however, it can only animate Range Nodes which means it can't animate something like HTML/CSS dialogs. But it does allow you to animate any Node without making any modifications to that Node.

TA is also meant as an easy route to learning animation. Its sole purpose is to ease the learning curve when adding animations to your project. In the long run you are strongly encouraged to "cook-up" your own animations using TE directly via Tweenables, and in some respects TA can help you—not to mention the plethora of Examples and Unit-tests.

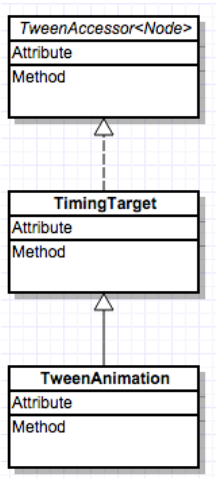


Figure 17.1 TweenAnimation

The biggest thing you will almost always use TA for is "running" your animations, even if you don't use one of TA's methods, this is because TA is automatically scheduled as a **TimingTarget**, and because it is already scheduled it also updates an instance of a TE **TweenManager** which releases you from having to deal with TweenManagers. This allows you to create custom animations using TE directly and instead use TA's TweenManager. TA's TweenManager isn't exclusive to TA it's also freely available—and encouraged—for you to use it for Tweenables.

Using TA's TweenManager is simple just call TA's *add* method to add an animation. Ultimately, you will always use TA, but how you use it is up to you. We will cover both so that you understand the two differences in using TA.

TA also has tracking methods that help you stop/cancel animations that are in progress. They hide any interaction you would need to have with the TweenManager contained within TA. Any time you want to track an animation you simply call the *track* method:

```
ranger.animations.track(rectangle, Ranger.TweenAnimation.ROTATE);
```

Now you can stop the animation by using one of several methods: *flush*, *flushAll*, or *stopAndUntrack*:

```
ranger.animations.stopAndUntrack(rectangle, Ranger.TweenAnimation.ROTATE);
```

or perhaps:

```
ranger.animations.flushAll();
```

TA Total Control

The first way to use TA is by putting total control in TA's hands, however, this means you will only be able to animate Ranger Nodes—which is highly probable seeing as you are using Ranger. Why is it that TA can only animate Nodes, because it implements TE's TweenAccessor of type Node. In other words TA is a TweenAccessor<Node> with the "knowledge" of working with Ranger Nodes. It forms a bridge between TE and Ranger. TE can animate anything as long as you do one of two things to help it out:

1. Either implement **Tweenable** or
2. Implement **TweenAccessor**<*type*> and register the it.

The second approach is what TA does. TA knows all about Ranger Nodes and as such it can "act" as a liaison between your Nodes and TE. Each time you want to animate one of your Nodes you need to tell TE the class type and who the liaison is—you only need to do this once during your app lifetime. Take a look at Ex2_SpinRect in which I purposefully use TA. In the *onEnter* method a call is made to TE's *registerAccessor* method:

```
UTE.Tween.registerAccessor(RectangleNode, ranger.animations);
```

The above code registers the example's custom RectangleNode *and* provides the liaison (aka TA) that will do the tween work (recall Chapter 5 "Animations".) Now we can apply an animation to RectangleNode using TA:

```
ranger.animations.rotateBy(
    rectangle,
    4.0,
    360.0,
    UTE.Linear.INOUT, null, false)
  ..repeat(10000, 0.0)
  ..start();
```

TA (aka ranger.animations) will handle the tween events that TE emits during an animation, and because TA is a TweenAcccessor it will have implemented the *getValues* and *setValues* methods. These two methods are large **switch** statements that handle a wide range of

animations. Here is a tiny snippet of just the **case** statement that handles the rotation specified above:

```
void setValues(Node target, UTE.Tween tween, int tweenType, List<num> newValues) {
  switch (tweenType) {
    ...
    case ROTATE:
      target.rotationByDegrees = newValues[0];
     break;
    ...
  }
}
```

If you were to attempt to call the *rotateBy* (or any TA animation method) without registering with the TE first it would immediately throw an exception indicating it can't animate your object because there isn't an TweenAccessor available to do the tween work. This is because you asked TE to animate an object but you didn't tell it what TweenAccessor was going to handle the tweens.

Infinite Animations and Stragglers

One of the things to be aware of is having animations that are still active on a Node even if the Node has left the stage or was removed from the scene graph. This is of concern with both pooled and non-pooled Nodes because it is still referenced by TE, and it gets more complicated when the Node is pulled from the pool and used elsewhere. The solution, remember to stop animations when a Scene or Layer exits during the *onExit* event.

You may have noticed that some of the examples don't stop their animations and that is because they are simple examples in that the scene *is* the application and never leaves the stage. However, any of the examples where there are transitions you will notice that the *onExit* "flushes" any animations during the exit. Take a look at Ex15_SceneTransition's *onExit*:

```
@override
void onExit() {
  disableInputs();

  ranger.animations.flushAll();

  super.onExit();
}
```

Even though RectangleNode isn't pooled TE will still have a reference to it which means it won't be garbage collected making it a valid reference and thus continue to send tween events to TA's *setValues* method which is certainly as waste of cycles. The key point here is that any animations (infinite or not) should be properly controlled and managed—don't leave any stragglers behind especially pooled Nodes.

As mentioned before TA is meant for simple things, eventually you would wean yourself away and begin using Tweenable.

Tweenable with TA

If TweenAccessor allows you to animate an object by augmenting through containment then Tweenable is it opposite. The second way to use TA is via implementing Tweenable, however, this requires a direct modification to an object which means you need to have access to object's

code. By extending your Nodes with Tweenable your object gets tween events directly. Keep in mind that you should still use TA's TweenManager and tracking methods but nothing else.

This time lets look at example Ex6_AlphaFade. In this example RectangleNode extended Tweenable and implemented its interface just like Ex7_ColorTint. This allows us to directly pass the Node to Tween. Again, we can use TA's TweenManager by calling *add*:

Code 17.2

```
UTE.Tween tw = new UTE.Tween.to(orangeRect, RectangleNode.FADE, 1.0)
  ..targetValues = [0.0]
  ..easing = UTE.Expo.INOUT
  ..repeatYoyo(UTE.Tween.INFINITY, 0.0);
ranger.animations.add(tw);
```

Nice. How about we look at a few animations to get a better understanding of creating and using animations.

Animations

As you have seen so far you can either use TA's animation creation methods or you can roll your own and use TA's internal TweenManager. Moving forward we are going to look at some simple animations using the preferred Tweenable approach. We have already seen two examples above in Ex6_AlphaFade and Ex7_ColorTint. Lets look at Ex6_AlphaFade first.

AlphaFade animation

Open up Ex6_AlphaFade project and navigate to the *onEnter* method. There you will see the an animation configured to animate the alpha property of the RectangeNode (see Code 17.2 above).

Lets break it down. First we are calling TE's *to* method to create a tween animation. The first parameter is the object that has implemented the Tweenable interface, the second parameter is the tween type which you defined yourself. This parameter helps guide your code as to what to do with the tween event routed to your object. The third parameter is simply the duration of the animation, the meaning of this changes if you are using the Tween's *repeat* or *yoyorepeat* methods.

As with any animation you need to configure the tween for how it will act during the animation. You do this by calling methods on the tween, examples are: the target values (aka end values), any type of easing, callbacks and trigger filters and repetitions.

In Code 17.2 we have set a target value of 0.0 which means we want our animation to interpolate from what ever default value our Node has to 0.0. How the interpolation occurs is defined by the **easing** property—the default is Linear. In Code 17.2 we chose an Exponential easing simply because it looks nicer.

If you don't configure a repetition the default is to run the animation for the duration specified and stop. In Code 17.2 we decided to set a Yoyo repetition that lasts FOREVER…or at least until you exit the app.

The ***tweenType*** is a value that you define in your object (aka Nodes). This value is reflected back to your Node during the animation:

```
static const int FADE = 0;
```

Your Node then decides how to respond to it by using something like a **select** statement:

```
switch (tweenType) {
  case FADE:
    fillColor.a = newValues[0].ceil();
    break;
}
```

This same usage applies to any kind animation you can dream up, for example, if you wanted to animate the color of the same Node you would create a new tweenType called, say TINT:

```
static const int TINT = 1;
```

Then update the switch statement to respond to the tweenType:

```
case TINT:
  fillColor.r = newValues[0].ceil();
  fillColor.g = newValues[1].ceil();
  fillColor.b = newValues[2].ceil();
  break;
```

Now you can create a color changing tween based on the tweenType:

```
UTE.Tween tw = new UTE.Tween.to(orangeRect, RectangleNode.TINT, 1.0)
  ..targetValues = [toColor.r, toColor.g, toColor.b]
  ..easing = UTE.Expo.INOUT
  ..repeatYoyo(UTE.Tween.INFINITY, 0.0);
```

Nice! Hopefully you are starting to see that you can pretty much do anything you want and you are only limited by your imagination. However, there is still more we can do aside of creating tweenTypes. We can organize each animation into groups that either operate in parallel or sequential.

Parallel

Lets say we wanted our ubiquitous rectangle to change color *and* move back and forth on the X axis all at the same time. No problem just use TE's **Timeline** Tweens. To show this I cloned and refactored the Ex7_ColorTint project and named it Ex7a_ColorTintAndMove.

The first thing to refactor is the RectangleNode. A new tweenType called MOVE_X is added:

```
static const int MOVE_X = 1;
```

With this new tweenType the *getTweenableValues* and *setTweenableValues* methods can be updated to respond to it:

```
int getTweenableValues(UTE.Tween tween, int tweenType, List<num> returnValues) {
  switch (tweenType) {
    case TINT:
      returnValues[0] = fillColor.r;
      returnValues[1] = fillColor.g;
      returnValues[2] = fillColor.b;
      return 3;
    case MOVE_X:
      returnValues[0] = position.x;
      return 1;
  }

  return 0;
}

void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) {
  switch (tweenType) {
    case TINT:
      fillColor.r = newValues[0].ceil();
      fillColor.g = newValues[1].ceil();
      fillColor.b = newValues[2].ceil();
      break;
    case MOVE_X:
      positionX = newValues[0];
      break;
  }
}
```

With the RectangleNode refactored we can move on to the MainLayer and update *onEnter* to create a Parallel Timeline tween:

```
UTE.Timeline timeline = new UTE.Timeline.parallel();
```

At this point we are going to hit our first restriction when using Timelines, none of the tweens “pushed” onto the time line can have their repetition set to INFINITY. The reason is fairly straight foreword, a Timeline needs to “complete”. A Timeline is completed once all Tweens have been executed and finished. Given that a tween with infinite repetition will not finish (only if you stop it), the Timeline will never finish and that cannot happen. This isn’t a problem because you can always manually control your animations using TE’s callback feature. For now though we simply set the repeat to something really large, for example 100000.

Now we create the two animations, one for color and the other for translation:

```
UTE.Tween tint = new UTE.Tween.to(orangeRect, RectangleNode.TINT, 1.0)
  ..targetValues = [toColor.r, toColor.g, toColor.b]
  ..easing = UTE.Expo.INOUT
  ..repeatYoyo(100000, 0.0);
timeline.push(tint);

UTE.Tween move = new UTE.Tween.to(orangeRect, RectangleNode.MOVE_X, 4.0)
  ..targetRelative = [200.0]
  ..easing = UTE.Sine.INOUT
  ..repeatYoyo(100000, 0.0);
timeline.push(move);
```

For each animation created we push it onto the timeline. This takes care of the timeline, now we need to add the Timeline to the TweenManager. We do this exactly like any other type of tween by calling *add*:

```
ranger.animations.add(timeline);
```

If you run the example now you will see the rectangle changing color *and* moving left and right, back and forth all at the same time. But what if we wanted them to occur one after the other. For that we use the sequential Timeline.

Sequence

For Sequence Timelines the repetition value has an actual impact. Simply setting the repetition to 100000 will cause all other tweens down the time line to *never* animate because the first animation will never complete/finish. Knowing this if we simply switch the timeline to a sequence like this:

```
UTE.Timeline timeline = new UTE.Timeline.sequence();
```

It will be a few “days” before we see the rectangle begin to move!—not good. So we change the repetitions to something more meaningful, and while we are at it decrease the duration of the move:

```
UTE.Tween tint = new UTE.Tween.to(orangeRect, RectangleNode.TINT, 1.0)
  ..targetValues = [toColor.r, toColor.g, toColor.b]
  ..easing = UTE.Expo.INOUT
  ..repeatYoyo(1, 0.0);
timeline.push(tint);

UTE.Tween move = new UTE.Tween.to(orangeRect, RectangleNode.MOVE_X, 1.0)
  ..targetRelative = [200.0]
  ..easing = UTE.Sine.INOUT
  ..repeatYoyo(1, 0.0);
timeline.push(move);
```

If you run the example now you will see the rectangle change color first then move back and forth once. If you want the whole sequence to repeat then set the timeline’s repetition:

```
timeline.repeatYoyo(100000, 0.0);
```

Now the animation will repeat the color and move sequence for several days.

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex7a_ColorTintAndMove

Callbacks and trigger filters

There are times when you need to know when an animation has reached a certain state, this can be done using callbacks and filters. Tween has two additional properties: ***callback*** and ***callbackTriggers***. Callback as you have probably guessed allow you to specify a function, to callback on, who's signature matches a particular **typedef**:

```
typedef void TweenCallbackHandler(int type, BaseTween source);
```

The callbackTriggers are a group of flags for filtering, and there is a set for both forward and backward, however, we will focus on forward only. The lifecycle of a Tween animation is shown in Figure 17.3.

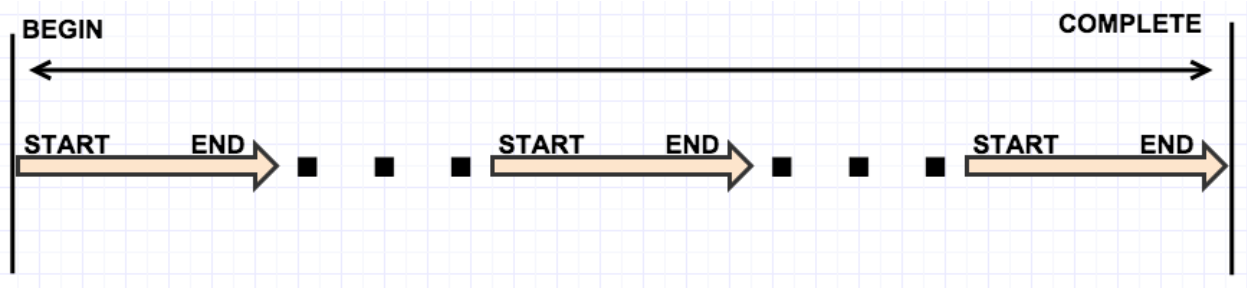


Figure 17.3 Tween lifecycle

You can see above that TE emits four unique events one for each state of an animation: BEGIN, START, END and COMPLETE.

What if we wanted our rectangle to move left and right for 2 reps and then switch to an up and down motion for 2 reps. To do this we need to use the Tween's callback features. To show this I cloned and refactored the Ex7a_ColorTintAndMove project and named it Ex7b_ColorTintAndMove.

First I upgraded RectangleNode with a new tweenType:

```
static const int MOVE_Y = 2;
```

Using the new tweenType I updated the Tweenable getters and setters to respond to the tweenType (only setter shown):

```
void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) {
  switch (tweenType) {
    case TINT:
      fillColor.r = newValues[0].ceil();
      fillColor.g = newValues[1].ceil();
      fillColor.b = newValues[2].ceil();
      break;
    case MOVE_X:
      positionX = newValues[0];
      break;
    case MOVE_Y:
      positionY = newValues[0];
      break;
  }
}
```

I then updated MainLayer by removing the Timeline Tween, by doing this I effectively separated the animations leaving the tint animation to run by itself while alternating the motion animation:

```
UTE.Tween tint = new UTE.Tween.to(orangeRect, RectangleNode.TINT, 1.0)
  ..targetValues = [toColor.r, toColor.g, toColor.b]
  ..easing = UTE.Expo.INOUT
  ..repeatYoyo(100000, 0.0);
ranger.animations.add(tint);

UTE.Tween move = new UTE.Tween.to(orangeRect, RectangleNode.MOVE_X, 1.0)
  ..targetRelative = [200.0]
  ..easing = UTE.Sine.INOUT
  ..callback = _moveComplete
  ..callbackTriggers = UTE.TweenCallback.COMPLETE
  ..repeatYoyo(1, 0.0);
ranger.animations.add(move);
```

Notice that I have assigned the *_moveComplete* method to the ***callback*** property and supplied a trigger filter of type COMPLETE. This says that once the animation is complete call *_moveComplete*. The callback is defined as:

```
void _moveComplete(int type, UTE.BaseTween source) {
  if (type == UTE.TweenCallback.COMPLETE) {
    if (_flip) {

      UTE.Tween move = new UTE.Tween.to(orangeRect, RectangleNode.MOVE_Y, 1.0)
        ..targetRelative = [200.0]
        ..easing = UTE.Sine.INOUT
        ..callback = _moveComplete
        ..callbackTriggers = UTE.TweenCallback.COMPLETE
        ..repeatYoyo(1, 0.0);

      ranger.animations.add(move);
    }
    else {
      UTE.Tween move = new UTE.Tween.to(orangeRect, RectangleNode.MOVE_X, 1.0)
        ..targetRelative = [200.0]
        ..easing = UTE.Sine.INOUT
        ..callback = _moveComplete
        ..callbackTriggers = UTE.TweenCallback.COMPLETE
        ..repeatYoyo(1, 0.0);

      ranger.animations.add(move);
    }
  }

  _flip = !_flip;
}
```

You can see that when the current move animation completes we toggle a flag and start a new move animation using the exact same callback and trigger flag. Nice! Now animate away!

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex7b_ColorTintAndMove

Chapter 18 Scene Graph

In this chapter we cover Ranger's Scene Graph from the perspective of Node usage and

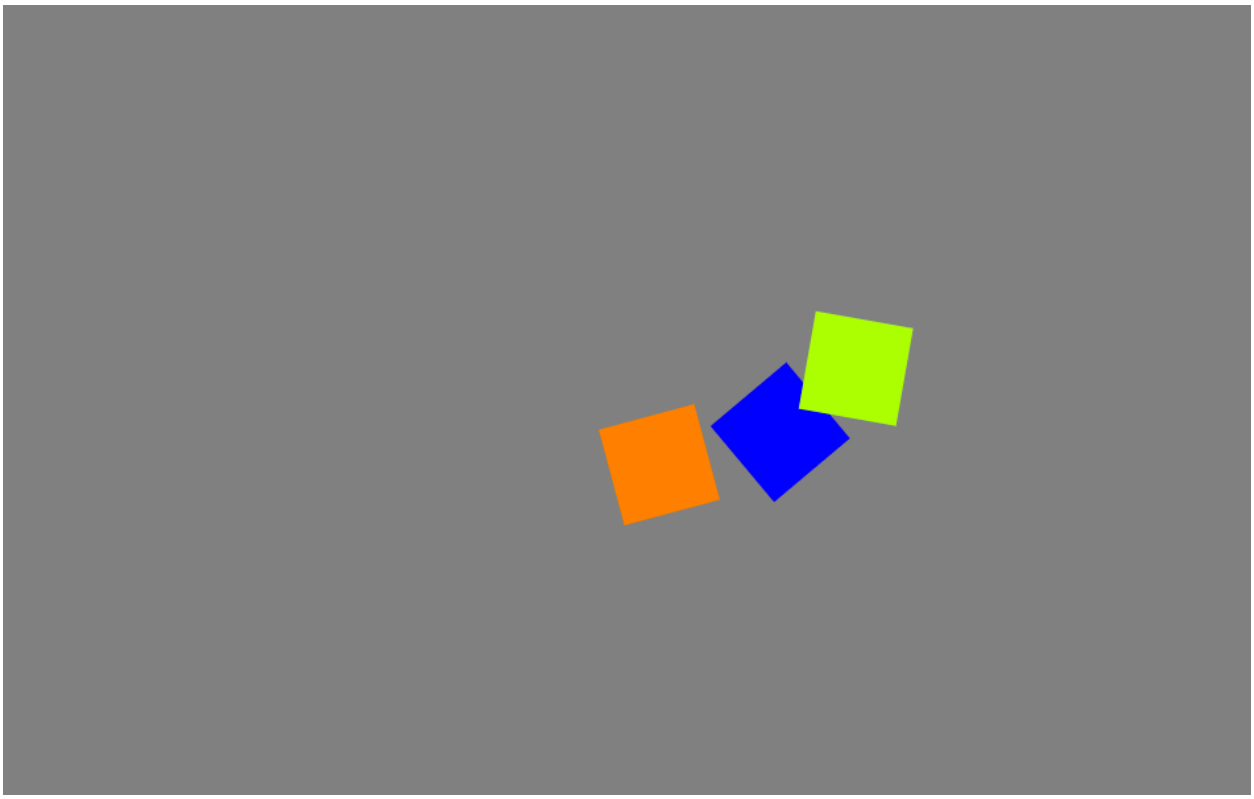


Figure 18.0 Node Hierarchies

Mapping. We go a bit more in-depth on Scene Graphs than we did in chapter 6 "Menus and Main Course".

Directed Acyclic Graph

Most game engines that implement a Scene Graph (SG) purposefully restrict the graph to a certain type called a Directed Acycle Graph (DAG) and Ranger is no exception. You can't have cycles within Ranger's SG and if you try an exception is thrown. This isn't really a problem because for games cyclic arrangements aren't really needed, however, if you were building a 3D modeler then it's possible you may introduce cyclic behavior but almost certainly keep it under control with a dependency graph.

As mentioned in chapter 6 Ranger's SG provides three main functions: render order, Node hierarchy, and a matrix stack. The rendering order manifests from a top-to-bottom left-to-right traversal order. This means Nodes that are added last are rendered on top of those that were added first.

Each Node contains an Affine matrix making it possible to traverse the graph "backward/upward" to form a cumulative matrix which can be used for Space mapping. When ever you map from one space to another you are effectively causing a traversal upward to a parent Node. Internally each Node has a dirty flag to help optimize excessive matrix concatenations.

Node hierarchies are effectively the opposite of space mapping. The Node matrices are applied in a "forward" notion as the renderer works its way down the graph concatenating onto the current rendering context matrix. When you are space mapping you are working your way up the graph and when you rendering you are working your way down the graph.

Example 5

Lets take a look at Ex5_NodeHierarchy's Node hierarchy arrangement. In this example RectangleNode has extended the **GroupingBehavior** mix-in to allow a hierarchy to form (see Figure 18.0.) The hierarchy was easily formed by making each succeeding Node a child of the preceding Node. Here is a snippet:

```
blueRect = new RectangleNode()
  ..init()
  ..fillColor = Ranger.Color4IBlue.toString()
  ..setPosition(1.0, 0.0)
  ..uniformScale = 1.0;

orangeRect.addChild(blueRect);

greenYellowRect = new RectangleNode()
  ..init()
  ..fillColor = Ranger.Color4IGreenYellow.toString()
  ..setPosition(1.0, 0.0)
  ..uniformScale = 1.0;

blueRect.addChild(greenYellowRect);
```

Recall that as Ranger traverses downward in the scene graph it concatenates each Node's Affine matrix onto the current rendering context. This means that if a parent Node has a transform applied to it then the child Node will inherit that transform on top of its own. In this example the orange rectangle has a scale transform of 150.0 which means all the children will have this scale because it was concatenated onto the rendering context. Your first instinct is to scale the blue rectangle so it will match the parent. If for example you scaled the blue rectangle to 150.0 then this is the same as scaling by 22500.0! And if you did the same to the green rectangle you are actually scaling by 3375000.0!

If you want to scale a child Node independent of it parent(s) then you need to divide by the accumulative parent scale. An example you will often see are outlines (or strokes) on a Node, if you simply attempt to stroke a shape without out the parent's inverse scale then your stroke will be scaled. Take a quick look at Ex19_Visibility project. The rectangle can be drawn with an outline, but notice the lineWidth being divided by a uniformScale value:

```
if (stroke) {
  context2D
    ..strokeStyle = strokeColor
    ..lineWidth = 3.0 / calcUniformScaleComponent()
    ..strokeRect(-0.5, -0.5, 1.0, 1.0);
}
```

calcUniformScaleComponent computes the accumulative scale starting at the current Node and working it way upward through the parents. Note: this calculation is an "approximation" because finding the accumulative scale requires a more complex approach called Matrix Decomposition. One has to make certain assumptions about the order of transformations; but even then, rotations are inherently ambiguous, as there is always more than one sequence of rotations to achieve a desired basis. Knowing the order of your transforms is important for decomposition.

Rotations and translations are compounded just the same, Scale was simply being used as an example. Notice when you rotate the orange rectangle that all the children rotate in sync. Again, rotation is being propagated down and as such if you were to rotate the blue rectangle too you would get double the rotation!

Ubiquitous Solar System

The classic solar system is a great way to show how the scene graph works in relation to each Node (reference Ex21_SolarSystem example for this discussion). The SG layout of the example is shown in Figure 18.2, page 254. It starts with the MainScene and works all the way down to a moon/planet Node.

Recall that each Node is a Space in and of itself (local-space), and it is also within a Parent-space too. The moon is a child of a planet system and planet is a child of a sun system. If we wanted to check for collisions between two planets we would map each planet their parent-space (i.e. sun system) and perform a check. You could alternatively map all the way to world-space but that wouldn't be necessary and it would be wasteful because both Nodes are siblings. Siblings Nodes only require parent-space for mapping. If however you wanted to check for the collision between a space ship that is in the universe (aka _system GroupNode) you would have to map both the ship and planet to world-space in order to make the collision check.

World-space is generally the "catch-all" for mapping. You typically use it when you don't have access to some common Space between your Nodes.

World-space boundary

As briefly mentioned earlier world-space is the boundary between your physical device and the actual scene graph that makes up the visual aspect of your game (see Figure 18.1.) In Figure 18.1 you can see there is a very distinct line (marked by **World-space**), that delineates the boundary. The only time you would do any kind of mapping "above" the boundary is when you are mapping mouse/touch inputs, and theses mapping methods are associated with the rendering context where each context knows how to map from input-space to world-space. World-space is literally the "bridge" between the real world and the virtual world.

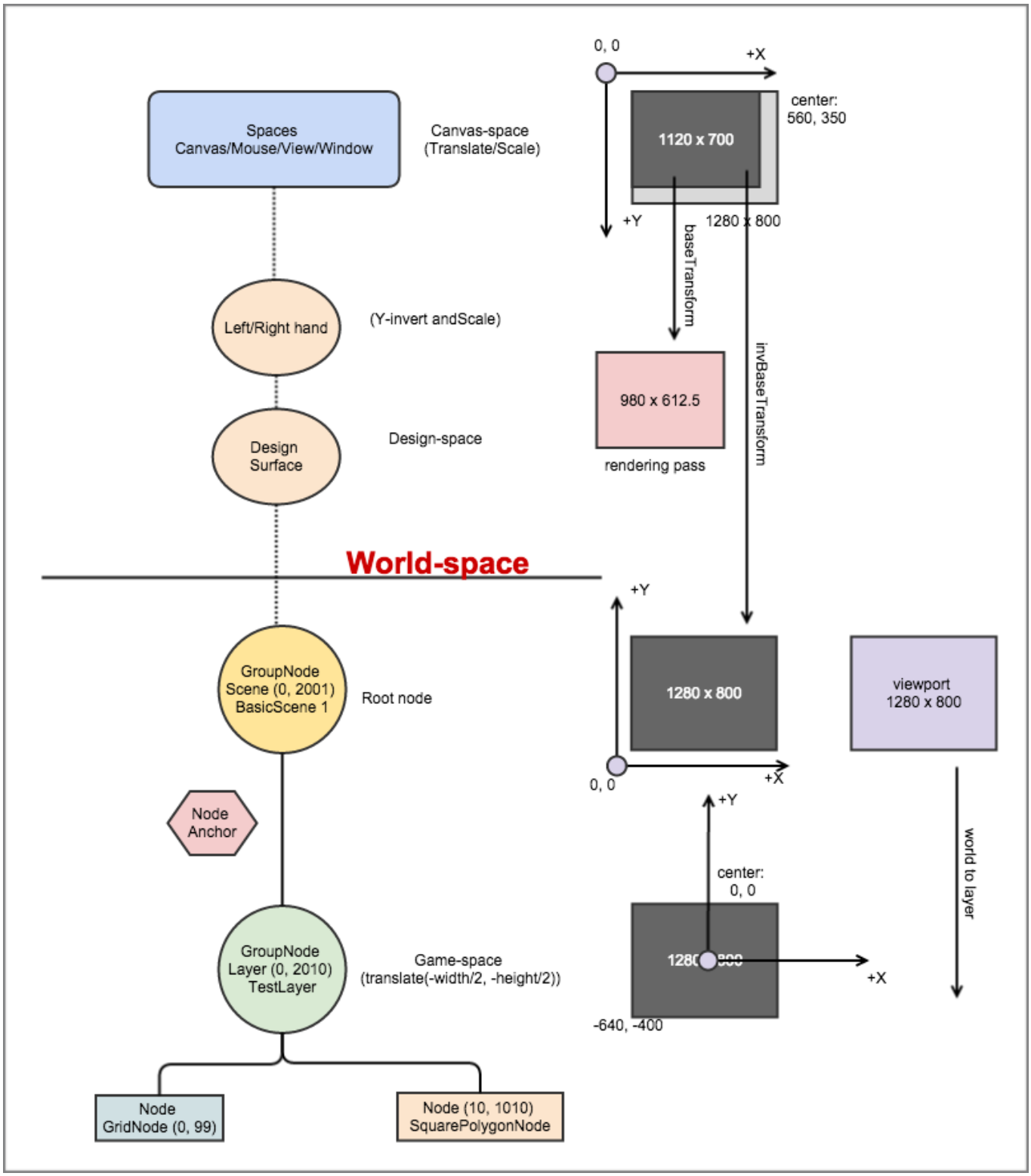


Figure 18.1 World-space

Once you have world-space mapping you can then map to any Node using the Node's

mapping methods to map from world-space to the local-space of any other Node.

Figure 18.2 is only the “lower” half of Figure 18.1. If, for example, you have a coordinate from say, a planet, you can find the equivalent coordinate in moon-space by mapping to world-space first and then mapping from world-space to the local-space of the moon. Because they are not siblings you can’t use either’s parent as a short cut.

Recall the topic of pseudo roots, you can, in this case, use a pseudo root as parent to reduce matrix calculations. The pseudo root in this example would be the _system GroupNode because it doesn’t have any transforms applied to it, in other words, it has an Identity matrix and Nodes with an Identity matrix can act like pseudo roots.

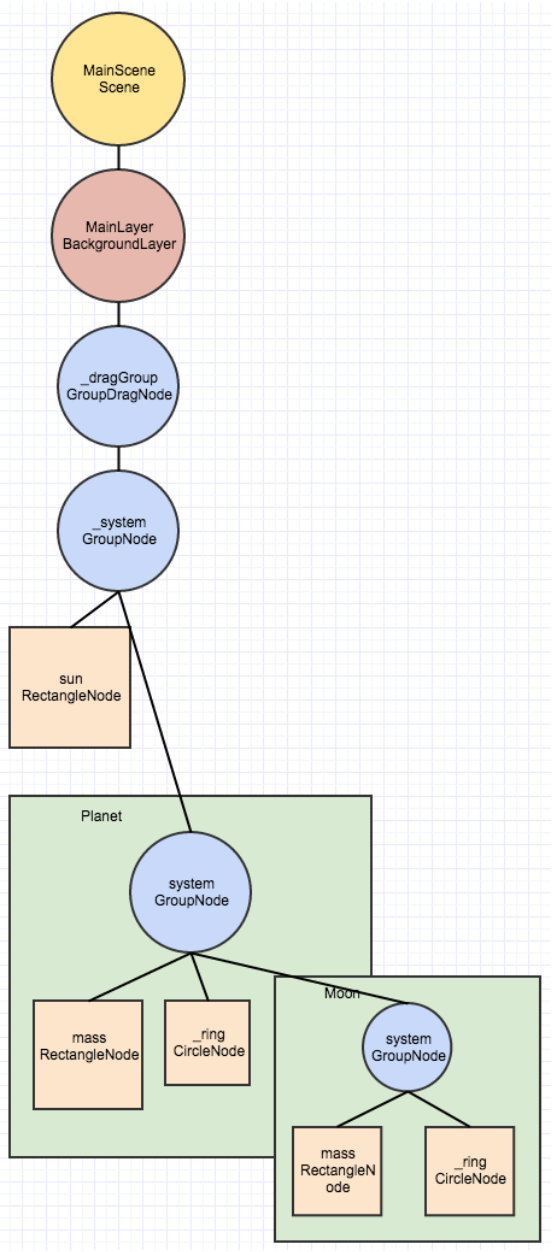


Figure 18.2 Scene Graph

Dragging

In the solar system example notice that dragging is begin done on the **GroupDragNode**. The GroupDragNode is a GroupNode with a MutableRectangle attached. This rectangle artificially “inflates” the GroupNode, which normally doesn’t have a size, this allows gives us the ability to map input coordinates to it.

Also notice that the drag isn’t actually applied to the GroupDragNode but to the _system Node. We do this so that the GroupDragNode isn’t actually dragged out of view thus allow us to repeatedly drag, and this works because _system is a child of GroupDragNode. This arrangement was intentionally done in order to create a situation where we can infinitely drag the solar system around. Again, because GroupDragNode isn’t being transformed its transform matrix remains the Identity matrix which means it could be used as a pseudo root.

Orbiting

To make a Node orbit another Node you take your Node and make it a child of an anchor like Node. For example, to make mercury orbit the sun, the example creates a **Planet** class that has a GroupNode called ***system***. It is the system GroupNode that is added as a child to _system. An animation is then applied to the mercury’s anchor Node to create the orbit effect. To have the planet rotate you apply an animation directly to the planet (aka mass Node.) The same

approach is used for earth's moon. The moon's anchor Node is made a child of earth's anchor Node and the same type of animations are applied to both the anchor and mass Nodes.

Once the anchor/child relationship is created you then translate them away from each other and apply a rotation to the anchor of which the child will "appear" to orbit the anchor.

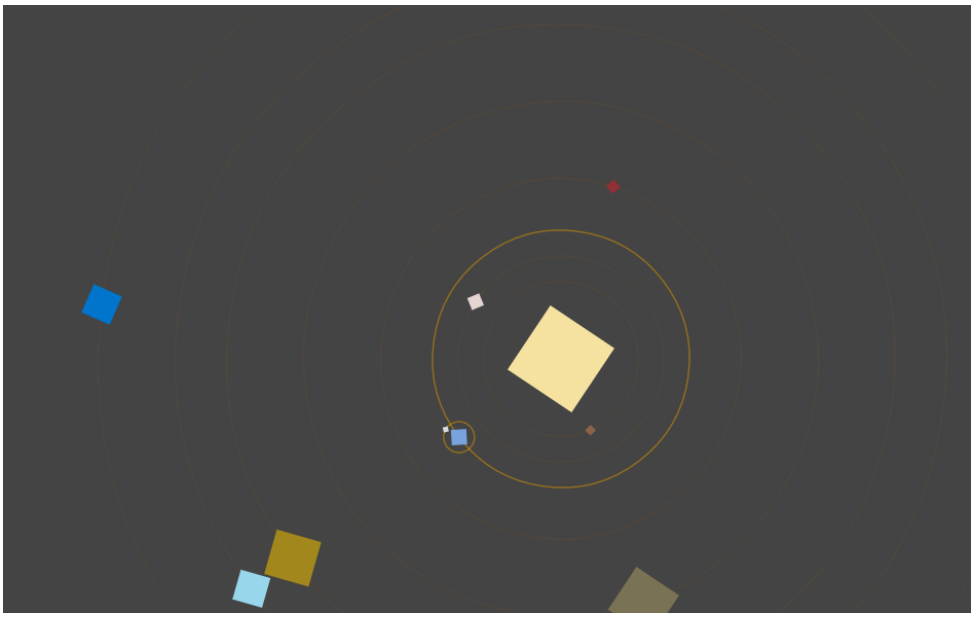


Figure 18.3 Solar System example

Above in Figure 18.3 you can see the solar system example. Each planet (rectangle) orbits the center yellowish rectangle. Rolling over any planet or moon triggers an animation that reveals the planet's orbital path.

Recap

We learned that Scene Graphs (SG) have a certain constraint applied to them namely they can't have cycles. Without cycles we can be certain that traversing the graph will be simple and predictable. We also learned that SGs provide a hierarchal rendering and mapping solution that helps with dividing up "space" both ordinal-wise and space-wise. By structuring our Nodes accordingly we can create many different effects such as orbiting and dragging.

We also learned that the SG is only "half" the picture with the other half embedded in the render context.

Challenge

One interesting challenge you can do with the example is to add Zooming, recall the ZoomGroup Node?

Code

https://github.com/wdevore/Ranger-MoonLander/tree/master/BookAssets/src/Ex21_SolarSystem

Chapter 19 Mobile

This chapter focuses more on mobile game development than on Ranger.

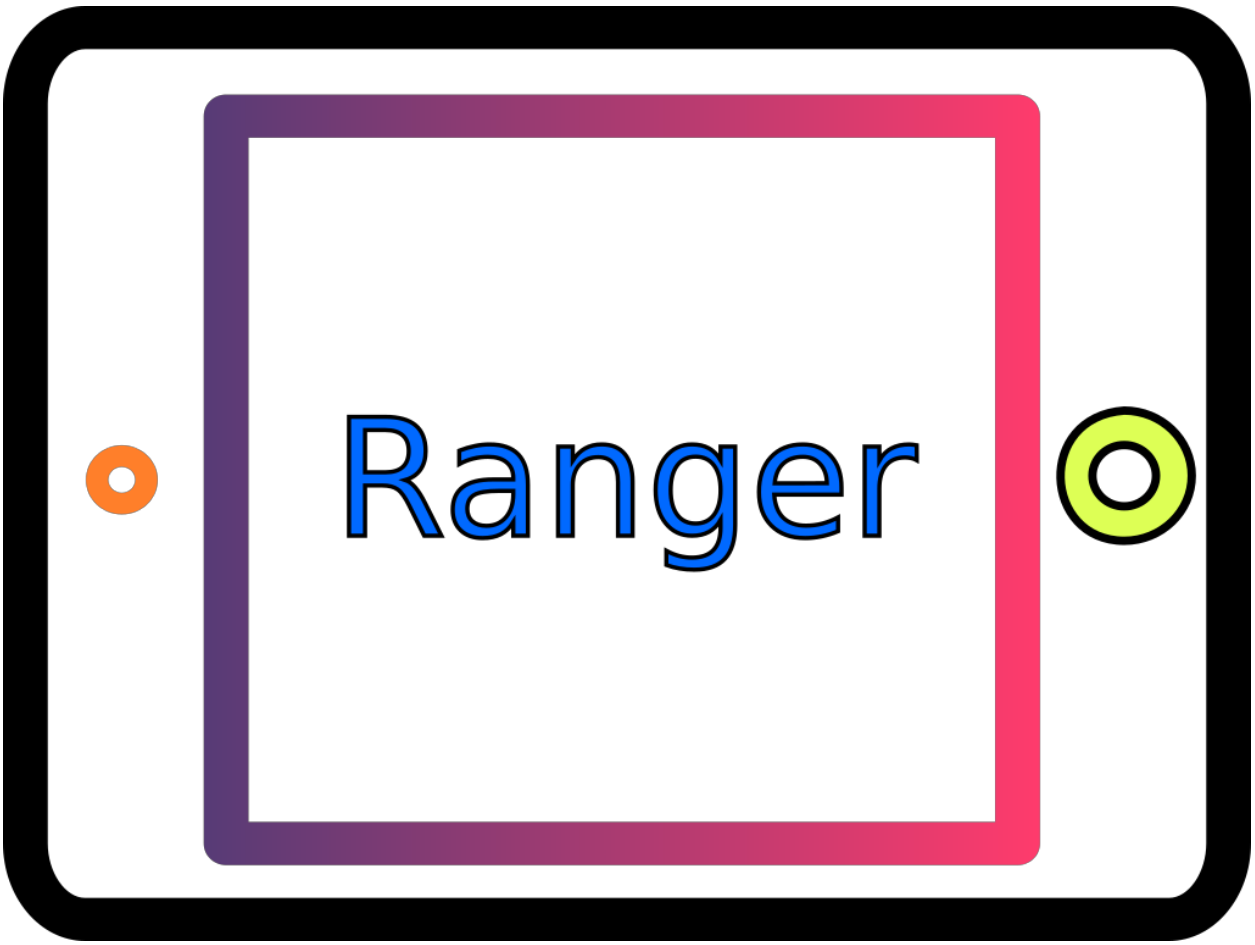
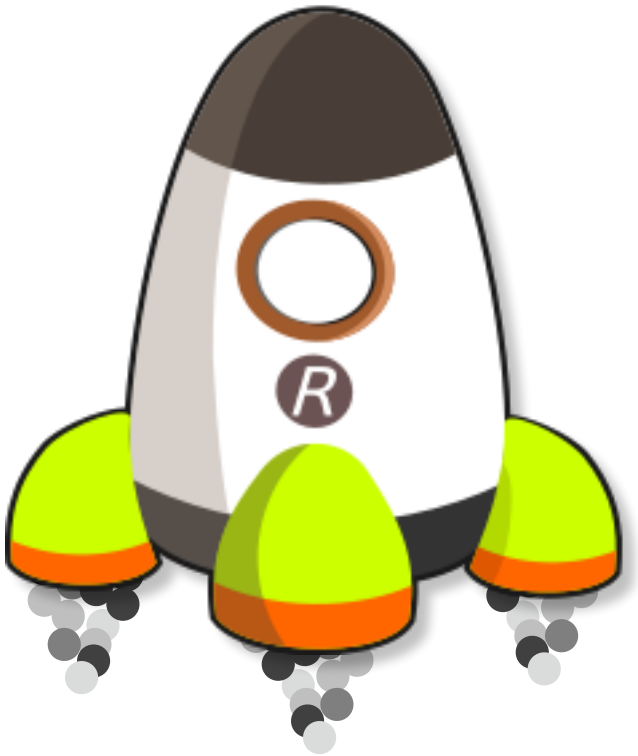


Figure 19.0 Mobile

In the works. This chapter will be available as a free upgrade to the book. It will cover either PhoneGap or Sky.

Chapter 20 Conclusion



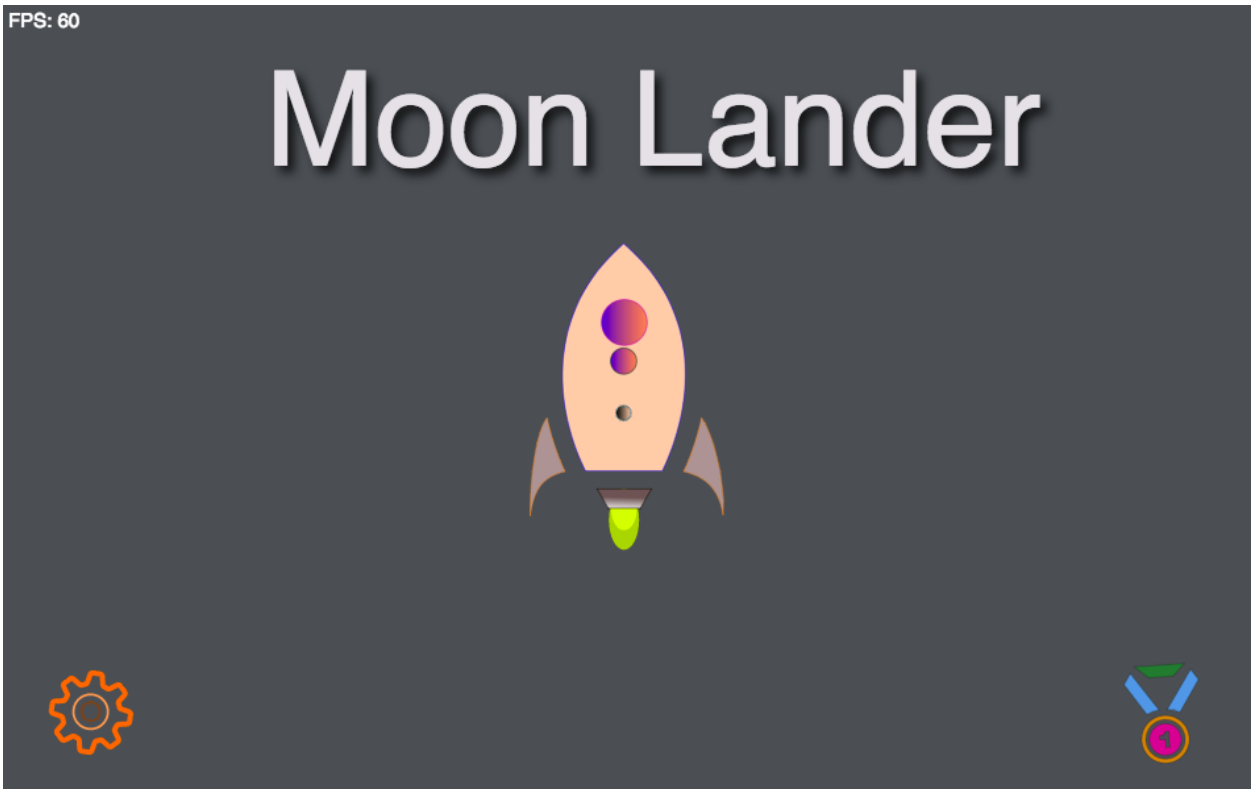
I hope you learned a lot about Ranger and hopefully your imagination has been invigorated with fresh new ideas for games.

Review

We started out with a look at the ubiquitous HelloWorld app then worked our way through a high level overview focusing on configuration and setup. We then set out to design a preliminary idea called Moon Lander to help guide us.

Having an idea of what we wanted to do we still needed to know how to do it which lead us to learning about Ranger's environment built around Dart using libraries (i.e. packages) and importing through parts. But that was only half the piece of the puzzle, the other half was knowing how Ranger displays itself through a concept we learned as a "fitting" policy.

We then began learning about the basics of Ranger starting with Scenes and Layers and then moving on to designing a simple edition of Moon Lander (see below.)



We learned that to move from Scene to Scene we use the SceneManager in conjunction with an AnchoredScene. AnchoredScenes required that we understand how coordinates systems work relative to Ranger and the viewport. From there we were able to create our first Layer and deal with a DrawContext and Pooling.

Next we got our first taste of animations by learning a little about TweenEngine and how it applies to Nodes. With our new knowledge of Layers we covered adding a HUD Layer and configuring it for Input. Input control allowed use to add simple thruster physics to Moon Lander's main actor, the Lander itself.

Through the process of building the lander we learned about custom Nodes arranged in hierarchal patterns to create landing gear and GUI controls like gauges. Because our lander is a rocket we covered the realm of particle systems which included Activators, Emitters, Direction and Behaviors. We learned about Space mapping for the first time in order to handle emission of particles from the lander's engines, part of the process brought to light the need for pseudo roots to help minimize matrix calculations.

Next we needed dialogs to handle game configurations for controlling sounds. In the process we learned how to create custom dialogs and popups using Ranger's Nodes and Tween animations. In order to loosely couple our dialogs and popups we learned about the EventBus package and how to transmit and receive messages along the bus.

With the ability to enable and disable sound effects it was time to learn about Ranger's audio abilities by using either a Sfxr or RSfxr JSON file descriptors, both cause Ranger to generate sounds on the fly rather than dealing with large audio files.

To take Moon Lander to the next level we needed to refactor the game by introducing bounding boxes and Zones, combined they allow the lander to sense for a touch down on the landing pad. Each landing is tracked until a crash occurs. This tracked information required we introduce a new Dart library called Lawndart for easy access to local storage. However, part of the data required that we ask the player for their initials and as such we learned how to create an HTML dialog that looked just like a Ranger Node dialog but allowed us to leverage HTML's input element to collect the player's initials.

By now we learned a great deal about how Ranger worked and how to interface with the real world, but that was just a focused view of Ranger, and that is where the Reference section came in. With the Reference's wider view of Ranger we were able to cover a variety of examples each one isolated from the other to help clarify each feature.

In the final few chapters we spread out even further covering more advanced ideas such as Transitions and Tween animations and finally ending with a deeper coverage of Ranger's scene graph.

Final thoughts

I hope you had fun learning Ranger-Dart, I know I had fun writing about it! There is no time to waste just start cranking out some super fun games.

Cheers. かんぱい!