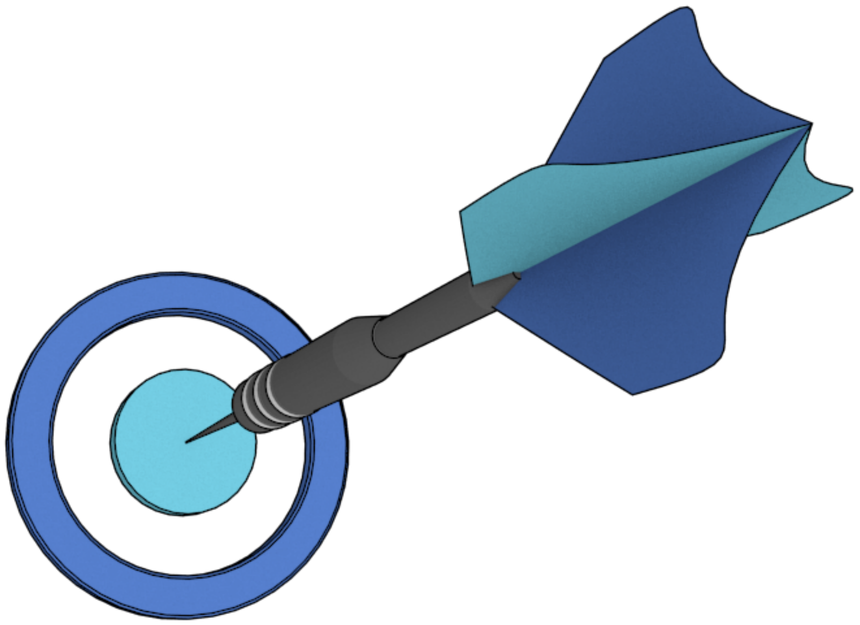
Ranger Getting started

William R. DeVore



Ranger Getting started
Ranger version 0.1.x

Contents

Goal

Our goal is to put together a starter game shell. The shell consists of nothing but a Splash Scene and Game Scene each with corresponding Layers.

Dart SDK

If you haven't done so yet, you will need to pull down the Dart SDK and install it. If you are on a Mac then it is as easy as installing the .dmg. Once installed you will now have the Dart Editor and Dartium available.

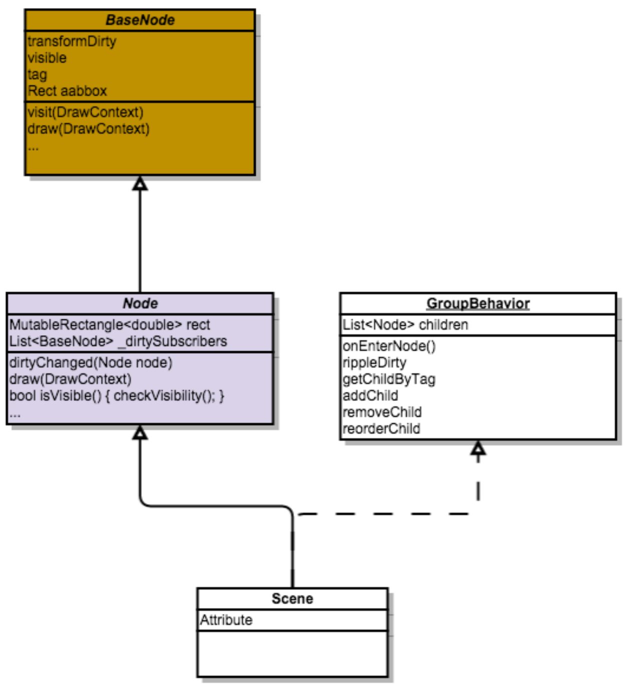
So how about we start with a very brief overview of **Ranger**'s Nodes.

Scenes and Layers

A *Scene* in **Ranger** is simply a container for other *Node*s. It is a Node with *GroupBehavior* mixed in. Scenes are also used as Nodes during transitions.

A Node without GroupingBehavior can't collect Nodes and hence it it is a Leaf. A *TextNode* is an example of a leaf Node.

Most of the time a Scene will contain only *Layer*s. Layers are grouping Nodes that provide both Input capability and a good place to put visual Nodes and code logic.



Creating the project

With that *very* brief overview lets get started building something. First we create a "New project" of type "Web application [mobile friendly]" using the Dart Editor that came with the Dart SDK.

Lets call the new project "MyRanger".

You should now have a project folder like the following:

```
MyRanger/
      packages/
      pubspec.lock
      pubspec.yaml
      web/
          myranger.css
          myranger.dart
          myranger.html
```

The default content of each of the files will be overwritten with our own code.

Open "**myranger.html**" and add a DIV with an id="gameSurface" and a width and height of 720x700; as shown below.

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

This DIV is what **Ranger** will attach an HTML Canvas to, and its dimensions--which act as the device dimensions--will be used to calculate a scaling ratio based on the "Design dimensions" that you specify in the "**myranger.dart**" file coming up next. The "id" is used to locate the DIV for insertion. But lets talk about resolutions first.

Design Dimension
The Design Dimension is a resolution you specify that is independent of any device. For example, if you specify a resolution of 1920x800 and the DIV (aka device) dimensions are 720x700 then the Canvas will have a scaling transform applied equal to (720/1920, 700/800) = (0.375, 0.875). Thus squeezing/scaling-down your Design “requirements” into the DIV which fits your design into device-space. Knowing this, it is a good rule-of-thumb to pick a design resolution that is somewhere in the middleground. For example, if you feel that your target devices are somewhere between 640x400 and 1920x800 then choose a design resolution of say 1280x600. This way the scaling will not be as harsh when running on either device. Nothing is perfect and sometimes you just have to use different design resolutions for each device.

CSS

The default project has some CSS entries that we need to replace. We want 0 paddings and margins, plus we want the gameSurface to be positioned relative.

```
body {
  background-color: #F8F8F8;
  font-family: 'Open Sans', sans-serif;
  font-size: 14px;
  font-weight: normal;
  line-height: 1.2em;
  padding: 0px;
  margin: 0px;
}

#gameSurface {
  margin: 0 auto;
  position: relative;
  overflow: hidden;
}
```

myranger.dart

Now lets add code to the main entry point in “myranger.dart”. The code is broken up into three mains sections: Imports, main() and preconfigure. For the imports we need the **Ranger** library of course, dart HTML Pub and two Scenes that we have yet to create. We will import the HTML and **Ranger** Pubs first.

```
library myranger;

import 'dart:html';

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

Notice several things: I gave the Pub an alias called "Ranger" and there is an error on the **Ranger** import itself. The first is because **Ranger** has a class called Node that will clash with Dart's Node class; adding the alias moves **Ranger** into its own namespace. The second is because we are missing a Pub reference in the **pubspec.yaml** file. Finally, because we are going to "modularize" the code I add a library name called "myranger"; we will use it later when creating the other "parts" of the application.

Pubspec.yaml

In order to use Pubs in Dart you need to reference them first. We reference them in the *pubspec.yaml* file. Double click the yaml file to open the configuration view.

There are several ways you can reference a Pub:

1. Local path (a folder on your desktop) or
2. on Dart's Pub environment (hosted) or
3. on Github.

Github clone

If you cloned **Ranger** from Github to your desktop with the intention of making your own modifications then you could use option #1 (Local path).

In this case you would add a "**path:**" entry to your pubspec that points to the *GitReposes* folder where **Ranger** was cloned to. For example:

```
ranger:
  path: /Users/some/where/GitReposes/Ranger-Dart
```

We are going to use option #3 because we want to reference **Ranger** directly from Github.

Click the "Add…" button and type in the name for the package. Lets call it "ranger", and click Okay. In the "Path:" field--in the "Dependency Details" area--enter:

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

This is the Github location of **Ranger**. Now save.

You will notice that once you "save" the Dart Editor begins pulling **Ranger** from Github:

```
Resolving dependencies...
+ color_slider_control 1.0.0 from git git://github.com/wdevore/color_slider_control.git
+ event_bus 0.3.0+2
+ gradient_colorstops_control 1.0.0 from git git://github.com/wdevore/gradient_colorstops_control.git
+ lawndart 0.6.5
+ ranger 0.1.0 from git git://github.com/wdevore/ranger-dart.git
+ tweenengine 0.11.1
+ vector_math 1.4.3
Changed 7 dependencies!
```

Also notice that a bunch of other Pubs have been pulled/fetched as well. Those additional Pubs are Pubs that web applications depend on. **Ranger** only relies on 3 of those directly: EventBus, TweenEngine and VectorMath. The others are actually dependencies of the Particle System application that comes with **Ranger** located in the "web/applications" folder.

Ranger-Sack
In the future the web applications included in **Ranger** will be migrated over to Ranger-Sack. This will reduce the extraneous dependencies.

Now the import is error free.

Next we add Dart's entry point "main()" and code it to instantiate the **Ranger** application.

```
Ranger.Application _app;

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

}
```

Ranger has several factory constructors, but the two most likely to be used are: **fitDesignToContainer** and **fitDesignToWindow**. The ParticleSystem app uses fitDesignToContainer because it needs an exact fit with no scaling; wyswyg. Our game--running on a desktop--needs more flexibility so it uses fitDesignToWindow. fitDesignToWindow will configure the Canvas such that it meets the design resolution you specified. Our code is specifying 1280x800 (aka Nexus 7 2012 model).

Mobile verse Desktop
So which constructor do you use? It depends on your requirements. If your game doesn't care about design resolution because say, it is a vector rendered game, then you could use **fitDesignToContainer** and your vector graphics scale accordingly. If your game depends on always having a known resolution--even if it's scaled--then **fitDesignToWindow** is your option.

One of the parameters to the constructors is a callback method where **Ranger** is indicating that it is a good time to build your Splash scene and main Game scene. Lets call our callback method "preConfigure" and code it.

To code it we are looking at a chicken-and-egg scenario. Do we code our two Scenes first or code preConfigure first with the intention of creating the Scenes second. I'm going to go with coding preConfigure first.

```
void preConfigure() {
  //---------------------------------------------------------------
  // The main game scene that contains the game layers.
  // It is also the Incoming scene after the splash screen.
  //---------------------------------------------------------------
  GameScene gameScene = new GameScene(2001);

  //---------------------------------------------------------------
  // Create a splash scene with a layer that will be shown prior
  // to transitioning to the main game scene.
  //---------------------------------------------------------------
  SplashScene splashScene = new SplashScene.withReplacementScene(gameScene);
  splashScene.pauseFor = 3.0;

  // Create BootScene and push it onto the currently empty scene stack.
  Ranger.BootScene bootScene = new Ranger.BootScene(splashScene);

  // Once the boot scene's onEnter is called it will immediately replace
  // itself with the replacement Splash screen.
  _app.sceneManager.pushScene(bootScene);

  // Now complete the pre configure by signaling Ranger.
  _app.gameConfigured();
}
```

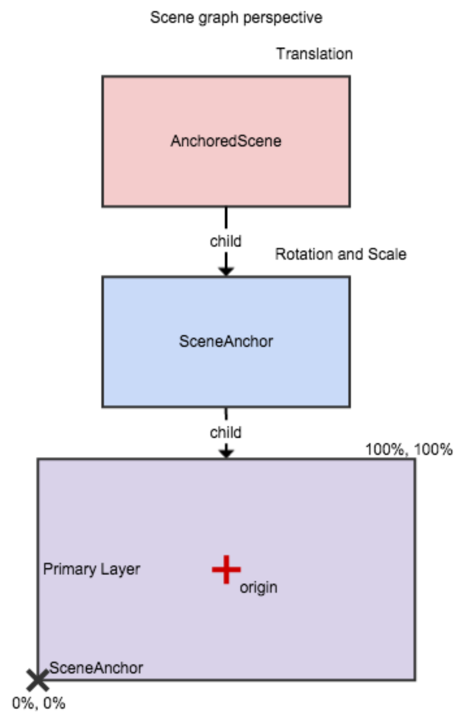
Notice I am using a strange Scene called BootScene. This scene does nothing more than wait for **Ranger** and Dartium to complete their launching cycle. For example, when Dartium launches it very briefly navigates to a default page and then navigates to localhost where your application is temporarily running.

This whole sequence generates a few events that **Ranger** listens to, one being the *window-resize* event and the other being *page-show* event.

It is the page-show event that dimensions are finally accurate and thus **Ranger** can call your *preConfigure* callback with good data.

Once we have built our lead-in Scenes we signal **Ranger** to continue booting by calling *gameConfigured*().

Splash Scene



Now we code the Splash Scene. I like to structure code in folders so lets create a folder for all the scenes called "scenes", and in that folder we create another folder called "splash". In there we will create a Scene class called SplashScene and a Layer called SplashLayer.

Create a dart file by right clicking the splash folder and selecting "New File". Name the file *splash_scene.dart*. Now create a class called SplashScene that extends AnchoredScene.

AnchoredScene is a special scene where an internal SceneAnchor Node is transparently added "hierarchically above" your Layer (aka primary layer) that allows transitions to work correctly. The primary layer in our code will end up being the SplashLayer that we create later.

Transitions typically do three types of transformations: Translations, rotations and scales. The Transition classes know to apply translations to the AnchoredScene and scales and rotations to the SceneAnchor.
Here is our SplashScene up to now:

```
class SplashScene extends Ranger.AnchoredScene {
  /**
   * How long to pause (in seconds) before transitioning to the [_replacementScene]
   * [Scene]. Default is immediately (aka 0.0)
   */
  double pauseFor = 0.0;
  Ranger.Scene _replacementScene;

  SplashScene.withReplacementScene(Ranger.Scene replacementScene, [Function
completeVisit = null]) {
    tag = 405;  // An optional arbitrary number usual for debugging.
    completeVisitCallback = completeVisit;
    _replacementScene = replacementScene;
  }

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

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

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

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

Named constructor

The *withReplacementScene* constructor takes a replacement Scene which just happens to be our GameScene that we haven't coded yet.

The *completeVisit* is a callback called when **Ranger** has completed a single visit of your scene. Visiting a Scene is the act of traversing all the children's visit() methods. This is useful if you need to perform some work after a single sweep through your scene.

onEnter

Our *onEnter* method does three main things: create the matching Layer, create a transition for transitioning to the replacement scene (aka GameScene) and finally ask the SceneManager to replace the current scene (aka SplashScene) with the transition scene.
onEnter is called by the SceneManager when your Scene has become active, and if your Scene is visible it will be immediately visible after the call. This knowledge is important when you start to code your own Transitions.

Transitions

Transition classes are Nodes too and they also have an onEnter method. The very first thing they do is "push" the Scene out of view or make it invisible, if they didn't the incoming scene would be immediatel visible thus negating the idea of "transitioning" the incoming scene into view. Case in point, look at the onEnter of TransitionInstant:

```
    Application app = Application.instance;

    // We have two choices:
    // 1) make the Scene invisible (probably the better choice for
    //    this transition) or
    // 2) Move the Scene out of view.

    // Option #1
    inScene.visible = false;

    // Option #2
    // Capture position before moving out of view.
    //prevPos.setFrom(position);
    // At this point the incoming scene is visible. So we move it out
    // of view. In this case way off to the left.
    //inScene.setPosition(-app.designSize.width, 0.0);
```

Splash Layer

Now we code the matching Layer called SplashLayer. This is where visual Nodes go. So create a new file called *splash_layer.dart*.

Our layer will be: centered, non transparent background with a background color, and on it we will have 2 Nodes of type TextNode: a title and version. The named constructor will do the centering and color while the onEnter will create the TextNodes.

```
part of myranger;

class SplashLayer extends Ranger.BackgroundLayer {
  SplashLayer();

  factory SplashLayer.withColor(Ranger.Color4<int> backgroundColor, [bool centered =
true, int width, int height]) {
    SplashLayer layer = new SplashLayer()
    ..centered = centered
    ..init(width, height)
    ..transparentBackground = false
    ..color = backgroundColor;
    return layer;
  }

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

    _configure();
  }

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

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

Game Scene

Create a new folder under the *scenes* folder called "game" and then create a new file called "game_scene.dart" to contain our GameScene class. This scene will be very simple as the only thing it does is create a matching Layer and reset the scene's position to (0.0, 0.0).

```
part of myranger;

class GameScene extends Ranger.AnchoredScene {
  Ranger.Scene _replacementScene;
  GameLayer _gameLayer;
  Ranger.GroupNode _group;

  GameScene([int tag = 0]) {
    this.tag = tag;
  }

  @override
  bool init([int width, int height]) {
    if (super.init()) {
      _group = new Ranger.GroupNode();
      _group.tag = 2011; // An optional arbitrary number usual for debugging.
      initWithPrimary(_group);

      _gameLayer = new GameLayer.withColor(Ranger.color4IFromHex("#666666"), true);
      addLayer(_gameLayer, 0, 2010);
    }
    return true;
  }

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

    // We set the position because a transition may have changed it during
    // an animation.
    setPosition(0.0, 0.0);
  }
}
```

The Scene itself will call initWithPrimary() instead of requiring the caller to create the Layer first.

Game Layer

Now create the last component the game. Nothing much new, we create a layer that is centered with a background color and a TextNode.

Create a new file called "game_layer.dart", inside the game folder, and create a GameLayer class:

```
part of myranger;

class GameLayer extends Ranger.BackgroundLayer {
  GameLayer();

  factory GameLayer.withColor(Ranger.Color4<int> backgroundColor, [bool centered =
true, int width, int height]) {
    GameLayer layer = new GameLayer()
      ..centered = centered
      ..init(width, height)
      ..transparentBackground = false
      ..color = backgroundColor
      ..showOriginAxis = false;
    return layer;
  }

  @override
  bool init([int width, int height]) {
    super.init(width, height);

    _configure();

    return true;
  }

  void _configure() {
    Ranger.TextNode desc = new Ranger.TextNode.initWith(Ranger.Color4IDartBlue)
       ..text = "Ranger GameLayer"
```

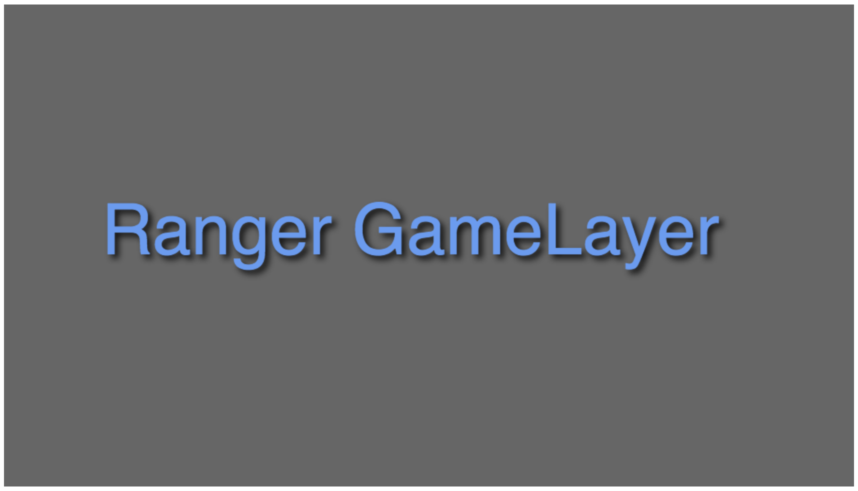
```
        ..shadows = true
        ..setPosition(-450.0, 0.0)
        ..uniformScale = 10.0;
      addChild(desc, 10, 445);
    }
  }
```

That is all there is to it. The whole game consists of just the following:

```
MyRanger/
     packages/
     pubspec.yaml
     web/
          scenes/
               game/
                   game_layer.dart
                   game_scene.dart
               splash/
                   splash_scene.dart
                   splash_layer.dart
     myranger.css
     myranger.dart
     myranger.html
```

Launching game

Now right click on "myranger.html" and select "Run in Dartium". You should see a Splash scene for 3 seconds then the Game scene with the text "Ranger GameLayer":


Ranger GameLayer

Tada! You just coded your first Ranger game!