第4章 Pong Game—2D动画和手势

动画是一项重要的功能，可以添加至应用程序，以使应用程序更具吸引力。动画能够以用户满意的方式添加重要功能。例如，可采用动画通知用户操作已完成，或者通过动画获取用户输入。在任何一种情况下，动画需要赋予应用程序一个较好的外观，以帮助用户实现成功操作，好消息是Flutter对动画有很好的支持。

动画通常是游戏的组成部分，因此本章将创建一个简化的、单人版本的、古老的Pong游戏，其间将制作一个会在屏幕上弹跳的球，并使用球拍使其避免碰撞到屏幕的底部。

构建这个游戏将使我们有机会详细了解动画如何在Flutter中工作。此外还将看到如何在微件中添加手势检测，这是另一项重要功能。最后，将在游戏中添加一些随机性，以使其更加有趣。

像往常一样，我们将从头开始操作。最终结果可能并不会发布至应用商店，但这是一种以非传统方式看待动画的有趣方式，也是思考游戏逻辑的良好起点。

特别地，动画主题包含以下内容。

 使用Stack和Positioned构建用户界面。

 使用Animation和AnimationController构建Tween动画。

 使用GestureDetector。

 使用Dart Math库中的Random()方法。

4.1 技 术 需 求

读者可访问本书的GitHub存储库查看完整的应用程序代码，对应网址为https:// github.com/PacktPublishing/Google-Flutter-Projects。

为了理解书中的示例代码，应在Windows、Mac、Linux或Chrome OS设备上安装下列软件。

 Flutter软件开发工具包（SDK）。

 当进行Android开发时，需要安装Android SDK。Android SDK可通过Android Studio方便地进行安装。

 当进行iOS开发时，需要安装MacOS和Xcode。

 模拟器（Android）、仿真器（iOS）、连接的iOS或Android设备，以供调试使用。

 编辑器：建议安装Visual Studio Code (VS Code)、Android Studio或IntelliJ IDEA。

4.2 构建应用程序的UI

创建游戏的第一步是构建基本的UI组件。在生成新的应用程序后，将构建一个球体、一个球拍和表示分数的文本。

（1）创建一个名为simple\_pong的新的应用程序。

（2）在main.dart文件中，在MyApp无状态微件的build()方法中，将返回一个MaterialApp，其标题为Pong Demo。对于主题，则使用经典的蓝色作为primarySwatch。

（3）在MaterialApp的主目录中，将放置一个Scaffold，它的AppBar将接收一个包含Simple Pong的文本。

（4）在body中，此时将放置一个空的容器，并于随后对其进行更改。具体步骤如下。

import 'package:flutter/material.dart';

void main() => runApp(MyApp());

|  |
| --- |
| 图4.1 |

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

title: 'Pong Demo',

theme: ThemeData(

primarySwatch: Colors.blue,

),

home: Scaffold(

appBar: AppBar(

title: Text('Simple Pong'),

),

body: Container()

));

}}

可以看到，大多数应用程序倾向于使用相同的模板代码。你可以改变颜色和样式，但大多数应用程序都可能有一个包含Scaffold的MaterialApp作为起点。

如图4.1所示，当前应用程序包含用户可以看到的3个UI元素：球体、球拍和分数文本。这3个元素需要包含在游戏自身的网格布局中。

接下来将构建UI组件。

4.2.1 创建球体

下面首先处理球体。这里，在项目中需有一个名为ball.dart的新文件。

（1）创建名为ball.dart的新文件，并在该文件中创建一个称作"Ball"的无状态微件，因为该微件在应用程序运行期间无须了解其位置或状态。动画将从调用类中修改球体微件的位置。

 提示：

记住，为了创建无状态微件，如果使用的是Visual Studio Code、Android Studio或IntelliJ Idea，只需输入stless，编辑器本身就会创建模板代码。

该文件的完整版本如下。

import 'package:flutter/material.dart';

class Ball extends StatelessWidget {

@override

Widget build(BuildContext context) {

final double diam = 50;

return Container(

width: diam,

height: diam,

decoration: new BoxDecoration(

color: Colors.amber[400],

shape: BoxShape.circle,

),);

}}

（2）在Ball类中，首先可以将形状的直径设置为50个逻辑像素。当然，也可以根据自己的喜好随意缩小或扩大直径。

（3）然后返回一个容器，其高度和宽度将是刚刚设置的直径，装饰的形状将是BoxShape.circle。

创建容器时，默认形状是矩形。通过指定BoxShape.circle，可以非常简单地避免处理角度从而实现圆形形状。此处将颜色设置为Colors.amber[400]。

 注意：

上述示例采用了Colors.amber[400]。大多数颜色的值为100～900，增量为100，再加上颜色50。数字越大，颜色越深，如ColorsblueAccent这样的强调色，具有较小的取值范围：100、200、400和700。

在球体制作完毕后，接下来将处理球拍。

4.2.2 创建球拍

球拍是用来防止球掉到屏幕底部的长方形图块，需要一个单独的文件。这里，创建一个名为bat.dart的文件。

（1）这也将包含一个无状态微件，因为球拍无须了解其位置或处理用户。所有动作将由调用者执行。

import 'package:flutter/material.dart';

class Bat extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Container();

}

}

（2）球拍的宽度和高度取决于屏幕的大小，这些值将通过调用者传递。所以在Bat类中，创建两个final型双精度值，即width和height。

final double width;

final double height;

（3）创建一个构造函数，接收两个参数并填充两个变量。

Bat(this.width, this.height);

（4）同样，build()方法将返回一个容器，其宽度和高度是在构造函数中传递的参数值，将有一个装饰，用于将背景颜色设置为Colors.blue[900]。该类的最终代码如下。

import 'package:flutter/material.dart';

class Bat extends StatelessWidget {

final double width;

final double height;

Bat(this.width, this.height);

@override

Widget build(BuildContext context) {

return Container(

width: width,

height: height,

decoration: new BoxDecoration(

color: Colors.blue[900],

),);

}}

此时，我们有了UI的两个主要元素。稍后将在结合游戏逻辑时处理分数文本。现在，需要一个网格来容纳球体和球拍。

4.2.3 创建网格布局

下面针对游戏创建一个新文件，即pong.dart。此处需要创建一个有状态微件，因为在类的生命周期内多个值发生了变化。

（1）使用stful快捷方式，创建名为Pong的新的有状态微件。

import 'package:flutter/material.dart';

import './ball.dart';

import './bat.dart';

class Pong extends StatefulWidget {

@override

\_PongState createState() => \_PongState();

}

class \_PongState extends State<Pong> {

@override

Widget build(BuildContext context) {

return Container();

}}

（2）此处不返回Container，而是返回LayoutBuilder。当想测量上下文（包括父约束）中的可用空间时，这是一个有用的微件，并以此确保球体不会超出应用程序中的可见空间。

LayoutBuilder微件在其构造函数中需要一个构建器，并接收一个带有上下文和约束的函数。在这个函数内部，将返回一个Stack。

return LayoutBuilder(

builder: (BuildContext context, BoxConstraints constraints)

{

return Stack();

});

 注意：

有几种方法可以实现这一点，但是在Flutter可用的容器中，有一种方法非常适合我们的目的，这就是Stack。Stack是一个微件，它根据框的边缘定位其子元素。

因为球体和球拍在整个游戏过程中都需要移动，所以可以通过修改它们与Stack边界的距离来改变它们的位置。Stack微件有一个children属性，可以在其中放置Stack自身包含的所有元素。定位其中元素的一种方法是使用Positioned微件。其中，可以指定顶部、左侧、底部或右侧属性。

 注意：

Positioned是一个微件，它控制Stack的子元素的位置。

下面将Ball和Bat添加到Stack中。当前，把球体放在位置top:0处，这意味着在可用空间的顶部，而球拍则放在位置bottom:0处，这意味着在可用空间的底部。球拍的宽度为200，高度为50，但稍后将对此进行修改。

return Stack(

children: <Widget>[

Positioned(

child: Ball(),

top: 0

),

Positioned(

bottom: 0,

child: Bat(200,25),)

], );

到目前为止，为了能够尝试使用这个布局，只需要从main.dart文件中的MyApp类调用Pong()微件。首先，需要在MyApp类的顶部导入。

import './pong.dart';

随后导入Scaffold的body。

body: SafeArea(

child: Pong()

)

 注意：

SafeArea是一个微件，它会自动在子元素上添加一些间距，以避免操作系统的扰乱，如屏幕顶部的状态栏或新款iPhone上的凹槽。

|  |
| --- |
| 图4.2 |

如果尝试运行应用程序，则可在屏幕的左上角和左下角分别看到球体和球拍，如图4.2所示。

在处理动画之前，首先准备该布局，以处理尺寸和位置方面的变化。在\_PongState类的顶部，创建几个变量以处理可用的空间、球拍的尺寸以及球拍和球体的位置。

double width;

double height;

double posX = 0;

double posY = 0;

double batWidth = 0;

double batHeight = 0;

double batPosition = 0;

其中，width和height表示屏幕上的可用空间；posX和posY表示球体的水平和垂直位置；batWidth和batHeight表示球拍的尺寸；batPosition表示球拍的水平位置。由于球拍将保持在屏幕的底部，因而它无须实现垂直移动。

在LayoutBuilder内部，首先设置包含布局高度和宽度以及球拍尺寸的变量，这些值包含在BoxConstraints实例中，作为参数传递至LayoutBuilder的builder()方法。

 提示：

BoxContraints包含4个有用的属性，即minWidth、minHeight、maxWidth和maxHeight，它们在运行时设置，在需要了解微件父元素的约束时非常有用。

球拍的大小相对于屏幕的尺寸而设定。因此，宽度为屏幕的20%（width/5），高度为可用空间的5%（height/20）。

在\_PongState类的build()方法中，以及LayoutBuilder的builder()方法中，添加下列代码。

builder: (BuildContext context, BoxConstraints constraints) {

**height = constraints.maxHeight;**

**width = constraints.maxWidth;**

**batWidth = width / 5;**

**batHeight = height / 20;**

return Stack(

...

在构造器返回的Stack中，当构造球拍时使用这些值，代码如下。

return Stack(

children: <Widget>[

Positioned(child: Ball(), top: 0),

Positioned(

bottom: 0,

**child: Bat(batWidth, batHeight),**

)

],

);

在布局的主要元素完成后，下面开始构建动画。

4.3 使 用 动 画

为了创建动画，并使球体在屏幕中运动，需要使用3个类，这也是Flutter中大多数动画的基础。

 第一个类被称为Animation。Animation类接收一些值并将它们转换为动画。Animation的实例并没有绑定到屏幕上的任何微件，因此它不知道屏幕上发生了什么，但该类设置了监听器，可以在每次帧变化期间检查动画的状态。

 第二个类是AnimationController。顾名思义，AnimationController控制动画对象。例如，可以使用它来启动动画、设置一个持续时间，并在需要时重复动画。AnimationController可以控制多个动画。对于本章的项目，将只使用一个动画。

 最后一个类是Tween。Tween是in between的缩写，包含在动画过程中需要改变的属性值，例如，如果一个微件在左侧从0～200实现动画，那么Tween将表示为1、2、3……直至200。

接下来将在代码中实际查看这些类，并在屏幕间移动球体。

（1）在\_PongState类顶部，创建包含Animation和AnimationController实例的变量。

Animation<double> animation;

AnimationController controller;

（2）重载initState()方法。

@override

void initState() {

super.initState();

}

读者可能想知道initState()方法是什么。对此，查看图4.3。

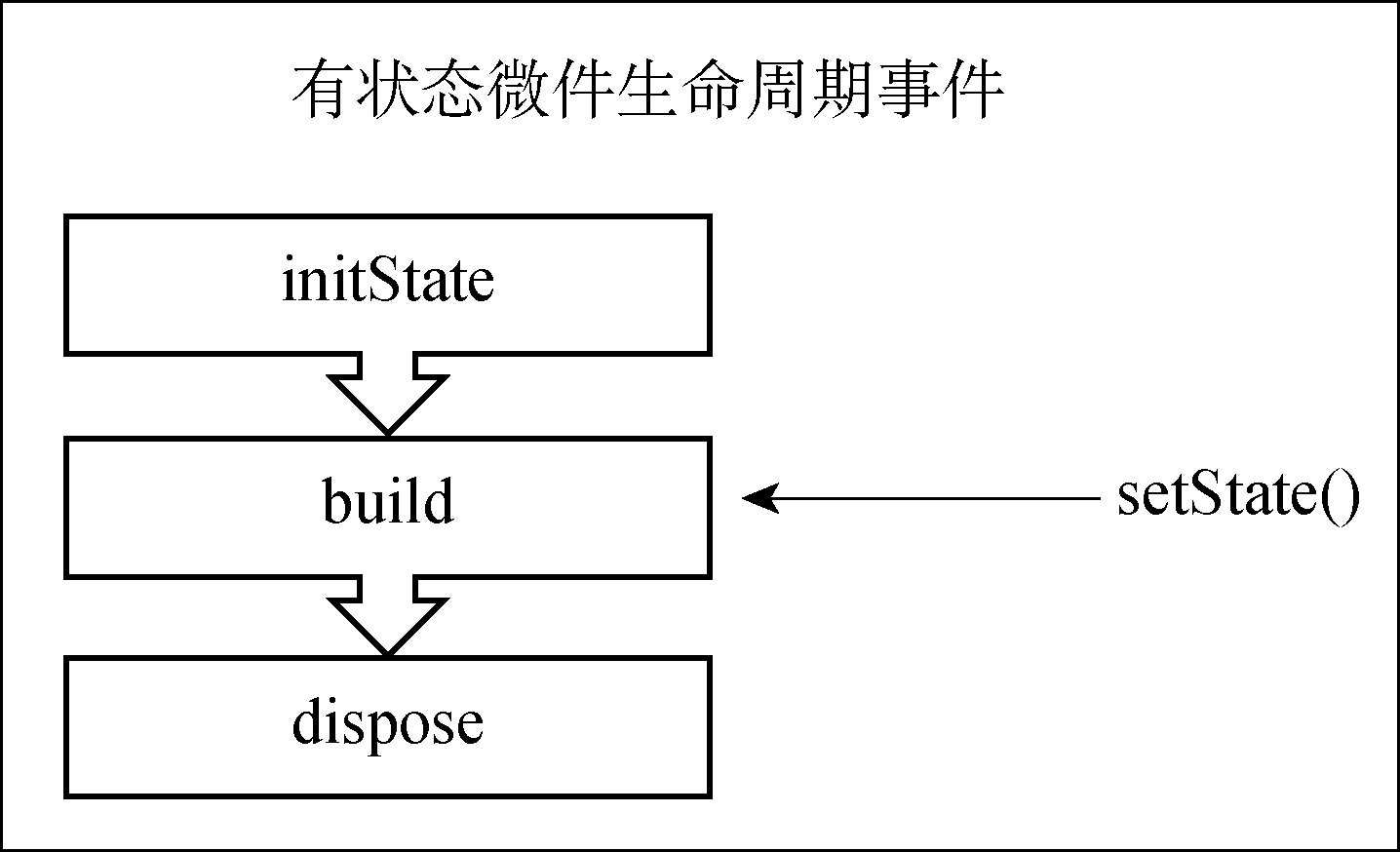


图4.3

有几个事件方法在有状态微件的生命周期中会被调用。

当一个State被创建时，initState()方法被调用。可以将此方法用于任何初始化，因为它只被调用一次。

在本项目和以前的项目中，我们已经多次使用build()方法。值得注意的是，每次调用setState()方法时，build()方法都会自动触发。这里可以放置那些变化的值，但对于任何初始化值都是无用的，因为每次调用setState()方法时都会覆盖这些值。在任何有状态微件的生命周期结束时，都可以重载dispose()方法。这有助于从系统中释放资源。

因此，根据上述描述，读者可能猜到需要在initState()方法中设置球体动画，并在build()方法中设置球体的位置。这里将构建的第一个动画是将球体从位置(top:0,left:0)移动到(top:100, left:100)。

（1）讨论initState()方法。此处使用posX表示球体的水平位置，posY表示球体的垂直位置。在动画开始时，两个值皆为0。

（2）初始化AnimationController，其持续时间为3s，因为我们希望球体花费3s从位置(0,0)移动至位置(100,100)。

（3）AnimationController需要一个TickerProvider，它通过构造函数中的vsync参数来配置。将vsync属性设置为this。

@override

void initState() {

posX = 0;

posY = 0;

controller = AnimationController(

duration: const Duration(seconds: 3),

vsync: this, );

super.initState();

}

（4）可以看到，此处将得到一个错误，这是因为vsync需要一个TickerProvider。为了解决这个问题，需要在状态中添加with SingleTickerProviderStateMixin子句。

class \_PongState extends State<Pong> with

SingleTickerProviderStateMixin

在面向对象编程语言中，Mixin是一个包含方法的类，这些方法可以被其他类使用，而无须成为其他类的父类。这就是在Flutter中使用with子句的原因，因为通过这种方式，我们是在包含类，而不是从类继承。换句话说，Mixin是一种在多个类层次结构中重用类中代码的方法。

 提示：

如果打算深入了解Dart中的Mixin，Medium上提供了一篇优秀的文章，对应网址为https://medium.com/flutter-community/dart-what-are-mixins-3a72344011f3。

SingleTickerProviderStateMixin提供一个Ticker。简单来说，Ticker是一种以几乎有规律的间隔发送信号的类，在Flutter中，大约是每秒60次，或者如果你的设备支持的话，每16毫秒一次，这种频率通常被称为帧速率。

（5）在\_PongState类的initState()方法中，在AnimationController下创建动画自身。

animation = Tween<double>(begin: 0, end: 100).animate(controller);

animation.addListener(() {

setState(() {

posX++;

posY++;

});

这里使用了Tween。如前所述，Tween是开始值和结束值之间的线性插值。起始值为0，结束值为100。随后调用animate()方法，传递刚刚创建的控制器。最后结果将返回动画本身。

（6）在动画中调用addListener()方法设置一个监听器，这将在对象发生变化时被调用。

（7）在setState()方法中，只需在动画的每次迭代中递增水平和垂直位置，这样球体就会在两个方向上移动100个像素。

|  |
| --- |
| 图4.4 |

（8）最后一步是更改球体Positioned微件的顶部和左侧参数。它们都将获取动画值，这是在创建Tween时定义的，介于0～100。基本上，球体从位置(0,0)移动到位置(100,100)需要3s。

return Stack(

children: <Widget>[

Positioned(

child: Ball(),

**top: posY,**

**left: posX,**

),

（9）为了启动动画，在initState()方法中，在super. initState()指令之前，调用控制器上的forward()方法。

controller.forward();

（10）如果尝试运行应用程序，可看到球体缓慢移至右下角，最终位置如图4.4所示。

综上所述，使用动画控制器定义动画应该运行多长时间，使用Tween设置线性值增量，并使用forward()方法启动动画。如果让这款应用程序保持现状，肯定会是一款无聊的游戏。这里需要做的第一件事是保持球体的移动，当它到达可用空间的边缘时，需要它改变方向。接下来对应用进行一些调整。

4.4 添加游戏逻辑

球体不应该处于停止状态，它应该在可用空间的边缘反弹。此处看到的弹跳实际上是运动方向的改变。所以当球碰到右侧边时，它应该向左移动，反之亦然。针对垂直方向也是如此。当球碰到上边界时，它应该改变方向并向下移动。此外，动画值也不会有用，所以需要将球体的位置与动画值分开，并将动画用作在正确位置重新绘制球体的方法。

（1）在pong.dart文件中，在import下方创建一个用于表示方向的enum，代码如下。

enum Direction { up, down, left, right }

enum关键字将创建枚举类型。这是一种特殊的类，可以使用它来表示固定数量的常数值。在当前示例中，创建一个名为Direction的枚举器，它可以有4个值：up、down、left和right。这将使代码更具可读性，它是使用数字或常量来指示方向的另一种选择。

（2）在\_PongState类的顶部，添加两个Direction类型的变量，它们将包含垂直（vDir）和水平（hDir）方向。开始时，球体需要向下和向左移动。

Direction vDir = Direction.down;

Direction hDir = Direction.right;

（3）检查球体是否到达了边界。此处已经知道了应用程序的边界，因为已经在LayoutBuilder构建器方法中设置了宽度和高度。所以，只需要检查球体的位置，看看它是否到达了这些边界。在PongState类中，创建一个名为checkBorders()的方法，它将检查球体是否到达其边界，并在到达时改变方向。

void checkBorders() {

if (posX <= 0 && hDir == Direction.left) {

hDir = Direction.right;

}

if (posX >= width - 50 && hDir == Direction.right) {

hDir = Direction.left;

}

if (posY >= height - 50 && vDir == Direction.down) {

vDir = Direction.up;

}

if (posY <= 0 && vDir == Direction.up) {

vDir = Direction.down;

}

}

（4）动画应一直处于持续状态。在\_PongState类的initState()方法中，现在将其设置为10000min而不是3s（一个非常长的游戏）。

controller = AnimationController(

duration: const Duration(**minutes: 10000**),

vsync: this,

);

（5）根据方向移动球体。仍然在initState()方法中，当创建动画时，在addListener()中，按照这里所示的内容更改setState()的代码。

animation.addListener(() {

setState(() {

**(hDir == Direction.right)? posX += 1 : posX -= 1;**

**(vDir == Direction.down)? posY += 1 : posY -= 1;**

});

**checkBorders();**

});

上述代码使用了三元运算符，并根据方向移动球体。如果水平方向为Direction.right，则需要递增水平位置，否则递减水平位置。同样的逻辑也可应用于垂直位置上：当方向为down时，递增posY；当方向为up时，则递减posY。每次移动后，调用checkBorders()查看是否需要调整方向。

当尝试运行代码时，球体应可在屏幕间弹跳。读者可能发现，球体的移动速度较慢。对此，可通过改变位置的增量来调整球体的速度。在当前示例中，我们总是加1或减1。如果想让动画更快，可将位置增量设置为3或5（而不是1）；如果想让动画稍慢，则可将位置增量设置为小于1，如0.5。

（6）创建一个变量以包含增量值。对于笔者的模拟器，增量设置为5效果已经很好。在\_PongState类的顶部，添加如下声明。

double increment = 5;

（7）随后在代码中使用increment。

animation.addListener(() {

setState(() {

(hDir == Direction.right)? **posX += increment :**

**posX -= increment;**

(vDir == Direction.down)? **posY += increment :**

**posY -= increment;**

});

checkBorders();

});

当尝试运行应用程序时，可以看到，球体的速度变得更快。我们可根据游戏速度的需要尝试调整increment。

接下来将移动球拍，以防止球体落入屏幕之外。

4.5 使用GestureDetector

顾名思义，GestureDetector是一个检测手势的微件。在布局的主体中，插入一个GestureDetector。这个微件具有响应用户手势的属性，并可以响应多个用户手势。最常见的包括onTap、onDoubleTap和onLongPress。在这些手势属性中，可以添加响应用户手势所需的代码。通常，我们要做的是更改微件的状态，但不限于此。

在当前实例中，需要移动球拍，因此将改变的状态值是包含球拍的定位微件的left属性。此处只需要对水平拖动做出反应，因为球拍不需要垂直移动。

（1）在pong.dart文件的build()方法中，作为batPositioned微件的子元素，我们添加一个带有onHorizontalDragUpdate参数的GestureDetector。这将接受一个DragUpdateDetails对象，它包含在屏幕上发生的拖动信息。

（2）调用moveBat()方法，该方法接收更新后的值。

Positioned(

bottom: 0,

**left: batPosition,**

**child: GestureDetector(**

**onHorizontalDragUpdate: (DragUpdateDetails update)**

**=> moveBat(update),**

**child: Bat(batWidth, batHeight))**

),

（3）在\_PongState类底部，编写moveBat()方法。

void moveBat(DragUpdateDetails update) {

setState(() {

batPosition += update.delta.dx;

});

}

DragUpdateDetails有一个delta属性，它包含在拖动操作期间移动的距离。dx是水平的delta。我们通过添加delta更新batPosition。这里，delta可以是正数也可以是负数。

如果尝试运行应用程序，应可在屏幕上以水平方向移动球拍。

在赋予用户与游戏交互能力之前，重写\_PongState类中的一个重要方法dispose()：使用该方法释放动画所使用的资源。当\_PongState对象被丢弃时，dispose()方法将自动被调用。在该方法中，添加了对动画控制器的dispose()方法的调用，以防止内存泄漏。

@override

void dispose() {

controller.dispose();

super.dispose();

}

当前，球体和球拍没有以任何方式链接，接下来将处理这一问题。

4.6 检查球拍位置

既然球体已经移动，球拍也对手势做出了反应，那么就需要判断球体何时到达屏幕底部，如果没有碰到球拍，这说明我们输掉了游戏。

此处需要修改checkBorders()方法。这里，我们处理的是屏幕的4个边界：顶部、左侧、右侧和底部。唯一需要修改的是底部，即检查球拍是否在正确的位置，使球体反弹起来，或者游戏是否需要停止。

在pong.dart文件中，在检查Direction.down位置处编辑checkBorders()方法，代码如下。

if (posY >= height - 50 - batHeight && vDir == Direction.down) {

// check if the bat is here, otherwise loose

if (posX >= (batPosition - 50) && posX <= (batPosition +

batWidth + 50)) {

vDir = Direction.up;

} else {

controller.stop();

dispose();

}

}

其中，50表示球体的直径。球体需要在球拍上而不是在屏幕的最底部反弹。所以，需要检查球体何时到达底部并减去球体的直径。

在嵌套的if语句中，检查水平位置。球拍的“理想”位置是球拍的水平起始位置（batPosition）和球拍的水平终点位置（batPosition + batWidth）之间，这样球体才能反弹。在此基础上，再加上球的直径。如果球体的位置包含在这两个值中，球体就会弹回来。否则将停止动画并释放系统资源。

由于多次使用数字50，对此，可添加一个变量并使用该变量来代替。

（1）在checkBorders()方法顶部，添加如下代码。

double diameter = 50;

（2）使用diameter变量进行检查。最终的checkBorders()方法如下。

void checkBorders() {

**double diameter = 50;**

if (posX <= 0 && hDir == Direction.left) {

hDir = Direction.right;

}

if (posX >= width - **diameter** && hDir == Direction.right) {

hDir = Direction.left;

}

if (posY >= height - **diameter** - batHeight && vDir ==

Direction.down) {

// check if the bat is here, otherwise loose

if (posX >= (batPosition - **diameter**) && posX <= (batPosition

+ batWidth + **diameter**)) {

vDir = Direction.up;

} else {

controller.stop();

dispose();

}

}

if (posY <= 0 && vDir == Direction.up) {

vDir = Direction.down;

}

}

当在对象上调用dispose()方法时，该对象就不可用了。任何后续调用都将引发错误。为了防止在应用程序中出现错误，可以在调用setState()方法之前创建一个方法，用于检查控制器是否仍然挂载以及控制器是否处于活动状态。

（1）在\_PongState结尾处添加下列代码。

void safeSetState(Function function) {

if (mounted && controller.isAnimating) {

setState(() {

function();

});

}

}

 注意：

mounted属性用于检查状态对象当前是否已挂载。状态对象在调用initState()之前被“挂载”，直到调用dispose()方法。当mounted不为true时调用setState()方法将引发错误。

（2）在initState()方法中，在animation.addListener中调用safeSetState()方法，而非setState()。

animation.addListener(() {

**safeSetState(() {**

(hDir == Direction.right) ? posX += increment :

posX -= increment;

(vDir == Direction.down) ? posY += increment :

posY -= increment;

});

checkBorders();

（3）在moveBat()方法中，同样调用safeSetState()方法。

void moveBat(DragUpdateDetails update) {

safeSetState(() {

batPosition += update.delta.dx;

});

}

如果尝试运行应用程序，可以看到当前游戏可以运行。当前游戏仍存在改进空间，稍后将对此加以讨论，但基础知识已介绍完毕。

4.7 向游戏中添加随机性

随机性是让游戏变得有趣的基本元素之一。在当前游戏中，存在两个环节可添加一些随机性。其中之一是弹跳角度—不需要每次弹跳都是45°。弹跳变得不规律会让游戏变得难以预测，还可以利用球体的速度来添加随机性。

假设完美弹跳表示为1，如果弹跳的值介于0.5和1.5之间，那么弹跳将不那么规律，但仍然保持一定程度的真实感。

（1）为了在Flutter和Dart中使用随机值，需要导入math库。在pong.dart文件中，添加下列导入语句。

import 'dart:math';

（2）在\_PongState类中，编写一个名为randomNumber()的方法，该方法返回0.5～1.5的随机双精度数值。

double randomNumber() {

// this is a number between 0.5 and 1.5;

var ran = new Random();

int myNum = ran.nextInt(101);

return (50 + myNum) / 100;

}

Random类生成随机bool、int或双精度值。它的nextInt()方法返回一个随机整数，范围为从0（包含）到传递的参数（不包含）。在本例中，它将产生一个介于0和100之间的数字。

随后加上50，再加上生成的整数，将得到一个介于50和150之间的数值，然后除以100。因此，该函数将返回一个介于0.5和1.5之间的数值。

（3）在\_PongState类的顶部，创建两个变量，一个用于垂直方向，另一个用于水平方向，它们将包含随机数。在执行的开始，randX和randY的值都是1。

double randX = 1;

double randY = 1;

（4）每次球反弹时，都希望根据到达的边界改变随机数的值。所以，当球向左或向右弹跳时，需要改变randX的值；当球在顶部或底部反弹时，需要改变randY的值。修改checkBorders()函数，并添加对randomNumber()方法的调用。

void checkBorders() {

double diameter = 50;

if (posX <= 0 && hDir == Direction.left) {

hDir = Direction.right;

**randX = randomNumber();**

}

if (posX >= width - diameter && hDir == Direction.right) {

hDir = Direction.left;

**randX = randomNumber();**

}

// check the bat position as well

if (posY >= height - diameter - batHeight && vDir ==

Direction.down) {

// check if the bat is here, otherwise loose

if (posX >= (batPosition - diameter) && posX <= (batPosition

+ batWidth + diameter)) {

vDir = Direction.up;

**randY = randomNumber();**

} else {

controller.stop();

dispose();

}}

if (posY <= 0 && vDir == Direction.up) {

vDir = Direction.down;

**randY = randomNumber();**

} }

（5）返回initState()方法中定义的Tween，并替换动画定义，将使用随机数来改变速度，而不是以固定值增加位置。

animation = Tween<double>(begin: 0, end: 100).animate(controller);

animation.addListener(() {

safeSetState(() {

(hDir == Direction.right)

**? posX += ((increment \* randX).round())**

**: posX -= ((increment \* randX).round());**

(vDir == Direction.down)

**? posY += ((increment \* randY).round())**

**: posY -= ((increment \* randY).round());**

});

checkBorders();

});

如果尝试运行游戏，可以看到球体的速度和弹跳将不那么规律，这让游戏变得更加不可预测。当然，也可以通过在randomNumber()函数中返回不同的范围的值来增加或减少随机元素。

每款游戏都应该拥有最后一个元素，那就是分数。下面将其添加到应用程序中。

4.8 添加分数并完成游戏

如果没有衡量游戏结果的方法，游戏则缺乏应有的完整性。在这种情况下，要执行的操作非常明显，即每次球体触碰到球拍则加一分。

（1）在\_PongState类顶部，创建一个包含分数的变量。

int score = 0;

（2）在build()方法中，向栈中添加一个新的Positioned微件，这将包含一个基于分数的Text。

return Stack(

children: <Widget>[

**Positioned(**

**top: 0,**

**right: 24,**

**child: Text('Score: ' + score.toString()),**

**),**

（3）在checkBorders()方法中，每次球体接触球拍后更新分数。

if (posX >= (batPosition - diameter) && posX <= (batPosition +

batWidth + diameter)) {

vDir = Direction.up;

randY = randomNumber();

safeSetState(() {

score++;

});

}

当尝试运行游戏时，可以看到分数出现在屏幕的右上角，如图4.5所示。

现在，给应用添加最后的润色：当玩家输了，则显示一条消息询问是否想再玩一次，如果是，则重新开始动画。

（1）在\_PongState类中，创建一个名为showMessage的方法，该方法在屏幕上显示一个对话框，如图4.6所示。

（2）调用showDialog()方法，该方法在屏幕上方显示一个带有材质设计动画的对话框窗口。这个方法采用一个builder()方法，其中可以构建一个Dialog微件，并使用当前的BuildContext。

|  |  |
| --- | --- |
|  |  |
| 图4.5 | 图4.6 |

void showMessage(BuildContext context) {

showDialog(

context: context,

builder: (BuildContext context) {

return AlertDialog(

);});

}

（3）在当前应用程序中，将使用AlertDialog询问用户是否再玩一次。警告对话框接收一个title、content和actions。

（4）showMessage()方法包含以下内容。

 title显示在对话框窗口上方。

 content表示对话框的主要内容。

 actions表示一个微件数组，指定用户能够执行的动作。在当前应用程序中，将使用两个按钮作为动作，即Yes和No。

（5）如果用户单击Yes按钮，则调用setState()方法将球体放置在(0,0)处，并将分数重置为0。

（6）Navigator的pop()方法将从屏幕上移除对话框，controller.repeat()方法将再次播放动画。AlertDialog的最终代码如下。

return AlertDialog(

title: Text('Game Over'),

content: Text('Would you like to play again?'),

actions: <Widget>[

FlatButton(

child: Text('Yes'),

onPressed: () {

setState(() {

posX = 0;

posY = 0;

score = 0;

});

Navigator.of(context).pop();

controller.repeat();

},

),

FlatButton(

child: Text('No'),

onPressed: () {

Navigator.of(context).pop();

dispose();

},

)

],

);

（7）在checkBorders()方法中，调用showMessage()方法，而非controller的dispose()方法。

controller.stop();

showMessage(context);

如果你尝试这个功能，会发现每当游戏失败时，对话框就会出现。

至此，该项目完成。此外，还可以添加一些功能来改进这款游戏，包括创建名人墙、将最佳成绩保存到设备中、在屏幕顶部添加砖块、添加声音、根据球拍位置改变弹跳角度，以及为第二名玩家添加第二支球拍。如果想挑战一下自己，可以尝试在这款应用程序中添加其中一些功能。这也许是提高Flutter技能的一种有趣方式。

4.9 本 章 小 结

本章根据动画和用户手势检测构建了一款简单游戏。

Flutter中的动画相对容易实现。动画中涉及的活动部件包括：Animation（该类接收一些值并将其转换为动画）、AnimationController（该类控制动画对象）、Tween（该类包含动画需要更改的属性值）以及每帧动画调用一次回调的Ticker。

将任何部件封装在GestureDetector中，就能监听用户在用户界面上做出的手势。这样，就可以利用GestureDetector的onHorizontalDragUpdate属性在屏幕上构建一只移动的球拍。

添加一些随机性通常会让游戏变得更有趣。对此，本章介绍了如何使用Random类通过nextInt()方法生成一个随机整数值。

在构建游戏时，本章还讨论了如何使用LayoutBuilder来获得屏幕上的可用空间，以及如何使用Stack来精确控制微件在应用程序屏幕上的位置。

为了向用户提供一些反馈，并执行一项选择，我们还使用了AlertDialog，并设置其title、content和actions。

在第5章中将创建一个电影应用程序，并利用HTTP库服务连接Web服务。

4.10 本 章 练 习

在项目的最后，提供了一些问题以帮助读者记忆和复习本章所涵盖的内容。请尝试回答以下问题，如果有疑问，可查看本章中的内容，你会在那里找到所有的答案。

（1）可以在堆栈中使用哪个微件来确定其相对于堆栈边界的确切位置？

（2）initState()和build()方法之间的区别是什么？

（3）如何设置动画的持续时间？

（4）如何在自己的类中使用Mixin类？

（5）什么是Ticker？

（6）Animation和AnimationController的区别是什么？

（7）如何停止正在运行的动画？如何释放它的资源？

（8）如何生成0～10的随机数？

（9）如果希望响应用户在一个微件（如容器）上的单击操作，那么可以使用哪个微件？

（10）如何在应用程序中显示AlertDialog？

4.11 进一步阅读

如果打算在应用程序中使用动画，首先要参考的是Flutter官方指南，对应网址为https://flutter.dev/docs/development/ui/animations。另一个包含视频、示例和指南的资源是https://buildflutter.com/functional-programming-with-flutter/。

本章从头开始创建了一款游戏。对于现实世界的游戏来说，这可能不是最常见的情况，因为你可以利用一些库和工具包来创建复杂的应用和游戏。如果真的打算制作引人注目的动画，可参考一个第三方工具，叫作Rive，它可以让你制作令人难以置信的动画，并将其添加至Flutter应用程序。读者可访问https://rive.app/以查看更多与Rive相关的内容。

关于如何利用Flutter构建多平台游戏，读者可访问https://medium.com/flutter- community/from-zero-to-a-multiplatform-flutter-game-in-aweek-8245da931c7e并查看相关示例。另外，读者还可访问https://flutterawesome.com/high-performance-animations-and-2d-games- with-flutter/以寻找创意并查看与Flutter相关的各种可能。

尽管Flutter还很新，但已经有许多库可以让您更轻松地在Flutter中创建游戏和动画，而无须从头开始，而且项目在不断添加。例如，可访问https://pub.dev/packages/flame查看Flame，还可以找到有关使用Flutter创建游戏的精彩教程和文档。