# So Far

We've learned quite a bit of things that can be used in a game
- Moving components
- Random number generator
- Movement and time for a smooth game

One thing is still needed – we need some sort of user input.

We'll make a new ball releaser that demonstrates how the user can interact with the screen
- User touches where they want the ball
- User drags finger across screen to show direction of the ball
- We will create a line showing the direction the ball will go
- The longer the drag, the faster the ball will go

This type of interaction could be used in a real game
Would allow for careful aiming and speed control

# New Component

As an exercise, see if you can make a new game component in file interactive_ball_releaser.dart and call the class InteractiveBallReleaser. Put in all of the needed methods but leave them blank. Have the game create and add this component.

- Make a constructor where you can pass it a game object
- It will need a width and height variable which are set by the resize() method
- We will not need a destroy() method, since this component will stay around
- It will need a render() method so that it can draw the aiming line
- It will need an update method
- You can put some print statements in resize() and the constructor so you can see if it works. Try running the game and make sure the component runs properly.

We will come back to the component later, but now we will go over Gestures

# Gestures

A gesture is something you do with your finger on a phone screen.  Flutter has support for all the most common gestures like double-tap, pinch and zoom, drag, and single tap.

We will be adding the drag gesture support to our game.

Since many different game components might want to use the drag gesture, we will add its support in our game itself.  That way, any component can see if it is being dragged.

We add gesture support in our main() method where we make the game.

Here is our updated main.dart file.  We will discuss the various things that have been added.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_ball/flutterball_game.dart';
import 'package:flutter/gestures.dart';
import 'package:flutter/services.dart';
import 'package:flame/util.dart';

void main() async {
  Util flameUtil = Util();
  await flameUtil.fullScreen();
  await flameUtil.setOrientation(DeviceOrientation.portraitUp);
  FlutterballGame game = FlutterballGame();

  // define gestures the game can handle
  PanGestureRecognizer dragger = PanGestureRecognizer();
  dragger.onStart = game.onDragStart;
  dragger.onEnd = game.onDragEnd;
  dragger.onUpdate = game.onDragUpdate;

  runApp(game.widget);
  flameUtil.addGestureRecognizer(dragger);
}
```

We have a number of new imports, because we are using some new classes in various libraries.  The first of these is Util.

We make a flameUtil object from the Util class and use it to control some various game functions.  From the code, you can tell that we are telling the game to always be in full screen and to be in portrait mode.  Notice the await keyword in front of those method calls and the async keyword in our main() declaration.  These are needed so that we can run certain functions that may take a long time without causing the entire app to stop and wait.  Main can continue doing other statements while those await ones are waiting to run.

We create our game object as before.

```
PanGestureRecognizer dragger = PanGestureRecognizer();
```

We create an object we call dragger from the class PanGestureRecognizer.  A pan gesture recognizes dragging in all directions.  There are other gesture recognizers that can be used.

Google flutter gesture recognizer if you want to see some other ones.

We have a number of new imports, because we are using some new classes in various libraries.  The first of these is Util.

We make a flameUtil object from the Util class and use it to control some various game functions.  From the code, you can tell that we are telling the game to always be in full screen and to be in portrait mode.  Notice the await keyword in front of those method calls and the async keyword in our main() declaration.  These are needed so that we can run certain functions that may take a long time without causing the entire app to stop and wait.  Main can continue doing other statements while those await ones are waiting to run.

We create our game object as before.

```
PanGestureRecognizer dragger = PanGestureRecognizer();
```

We create an object we call dragger from the class PanGestureRecognizer.  A pan gesture recognizes dragging in all directions.  There are other gesture recognizers that can be used.

Google flutter gesture recognizer if you want to see some other ones.

```
dragger.onStart = game.onDragStart;
dragger.onEnd = game.onDragEnd;
dragger.onUpdate = game.onDragUpdate;
```

These method are how you hook the dragging gesture to the game.  When the drag starts, it will call an onDragStart method in our game.  We have to make this method.

We make an onDragEnd method that gets called when dragging ends (user lifts finger)

A method onDragUpdate is called as the finger is moved around.

You will see errors in your file until we make these methods in our game.  That's next.

One last thing is to tell the Flame engine about the new gesture recognizer we created. That is always done after the runApp which starts the game.

Can you guess what the new game methods will be passed by the gesture recognizer during a drag?

# Drag Methods in Game

When our game runs, any time a user drags their finger on the screen, the three methods will be called.  It doesn't matter what components have been added or what the game is doing.

OnDragUpdate can be called very often as it tells the game exactly where the user is moving their finger.

Time to update flutterball_game.dart

If you have already made the interactive ball releaser class, add it to the constructor.  If not, we will go over that class later on, so leave the constructor empty for now.

```
// make a new game
FlutterballGame() {
  // make a new ball releaser game component
  var interactiveBallReleaser = InteractiveBallReleaser(this);

  // tell the game about this component
  add(interactiveBallReleaser);
}
```

Add in the 3 methods that our game needs to handle the drag gesture.

```
// gesture handlers
void onDragStart(DragStartDetails d) {
}

void onDragEnd(DragEndDetails d) {
}

void onDragUpdate(DragUpdateDetails d) {
}
```

Notice that the gesture recognizer passes a details object to each of our methods. This object lets the method determine things that are happening in the drag. The only thing we care about is the x and y coordinates of the finger, and the details objects have a property that gives us this information. It's sort of like resize() which is passed a screen size object.

We need some instance variables that these methods will set to tell us what's going on

```
// instance variables
bool isDragging = false;  // true when user is dragging across the screen
double dragX;   // x coordinate uf oser's finger
double dragY;   // x coordinate uf oser's finger
```

Our onDragStart method tells us that dragging has started and saves the finger location. Add this code to that method.

```
isDragging = true;
dragX = d.globalPosition.dx;
dragY = d.globalPosition.dy;
```

onDragEnd just tells us that dragging is over.  Add this.

```
isDragging = false;
```

The onDragUpdate method just updates the finger location variables.  Add this.

```
dragX = d.globalPosition.dx;
dragY = d.globalPosition.dy;
```

Now, if any component wants to check if dragging is going on, it just needs to read the 3 instance variables of the game.

Try adding print statements in the 3 methods and run the game to watch what happens when you drag.

```
print("drag start: ${d.globalPosition.dx}, ${d.globalPosition.dy}");
print("drag end");
print("drag update: ${d.globalPosition.dx}, ${d.globalPosition.dy}");
```

# Wrapping Up

We have the ability to detect drags, and we will use this next time when we finish our interactive ball releaser component.

If you run it with the print statements, you will see hundreds of updates being generated as you move your finger around.  It may even cause errors if you are running in the emulator since so many prints are happening together.

You should notice that the finger coordinates are the same as the screen coordinates we have been using all along.