Acceleration & Dampening ON UR iPAD

LAB 10X: iPad - Acceleration

Start where you left off...

We'll start off now, using the Bounce project that you completed at the end of LAB 07.

If you had problems with LAB 07 and it's not working properly, please let me know. Also, take a look at my completed version of LAB 07 on GitHub, if you'd like to see how to get it working.

https://github.com/AgencyAgency/iPadDemos/tree/master/Bouncing

Remember all that bouncing and dampening that we accomplished in Processing? We're going to do the same thing on the iPad. Except, we're going to figure out which way the the force of gravity will pull the ball based on the orientation of the iPad in space.

This way, no matter how you hold the iPad, the ball will always fall toward the center of the earth.

I know. Crazy.

You must use your iPad.

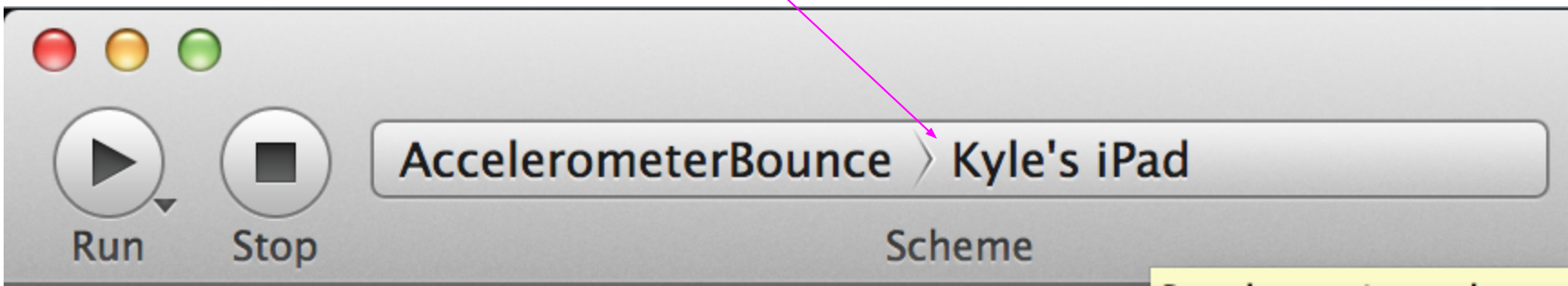
We're going to be using the accelerometers in your iPad.

The Simulator, unfortunately, doesn't support accelerometers.

So, you must deploy to your iPad in order to finish this lab.

In general, when I say to **Run** this app, be sure you're running it on your iPad.

Hint: Make sure this say's **iPad**.



First, add gravity.

If you remember all the way back to LAB 07, you'll recall that this simulation doesn't include the effects of gravity.

Let's **fix** that now.

Add a new property to hold your **gravity**.

We also have to initialize its value. 5.0 sounds good for now. Don't worry about units, and we only need to approximate reality. So, guessing a good value is a good way to go for now.

Finally, we have to add this **gravity** value to the **velocity**'s **y** component.

Do you see that **+=** there? That line means the same things as:

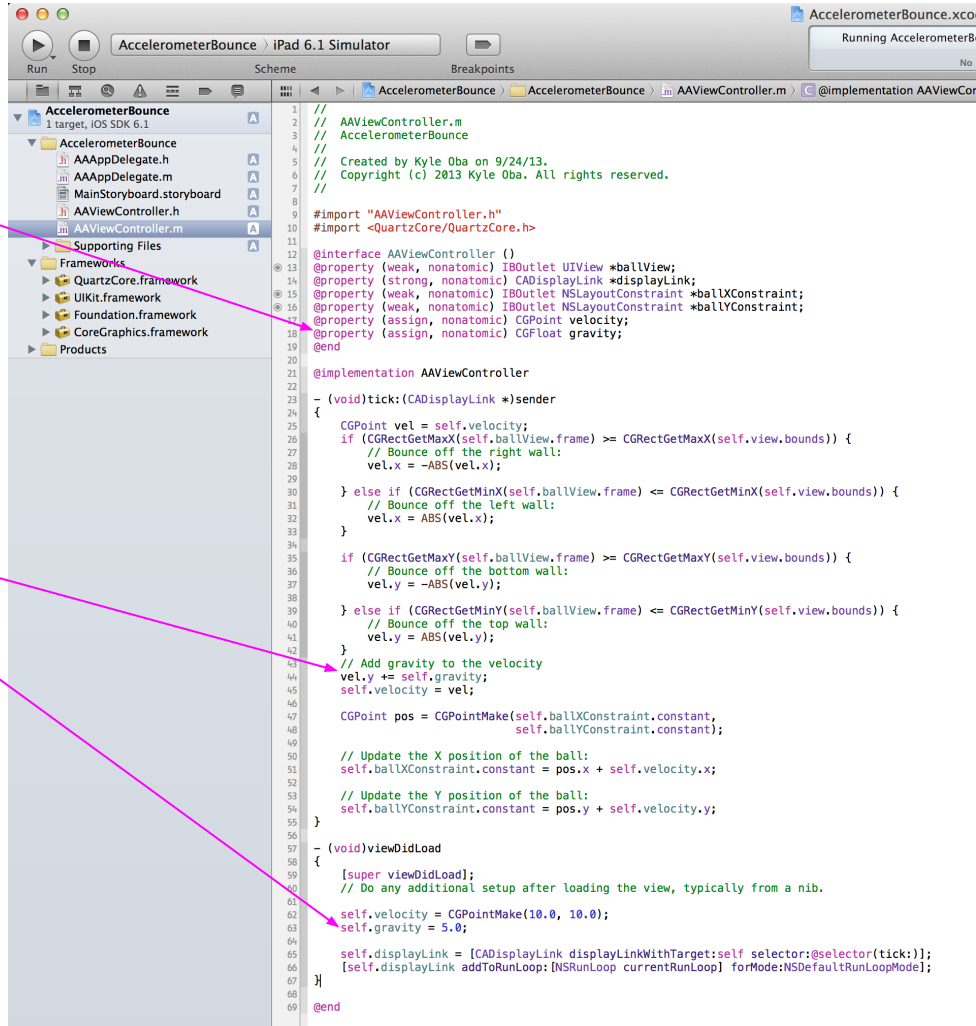
vel.y = vel.y + self.gravity;

So, using **+=** is a shortcut for saying, add this thing to my current value.

x = 1;

x += 4 → 5

Run your app. Watch what happens when the ball loses all of its bounce. It eventually gets *smushed* off the screen by the force of gravity. (Yes, "smushed" is a technical term.)



In Processing, you keep the ball on the screen like this...

How do we prevent the smushing (the ball leaving the screen)? That's a bit of a trick question.

In the last lab (with Processing), we used a hack to keep the ball from getting smushed off the screen.

I kinda mentioned it at the time, but that's why you added **g** to **dy** in a conditional (**if** statement).

```
if (y + R < height) {
  dy = dy + g;
}
```

(This is the Processing code you used for that. Don't try to use it in Xcode. It won't end well.)

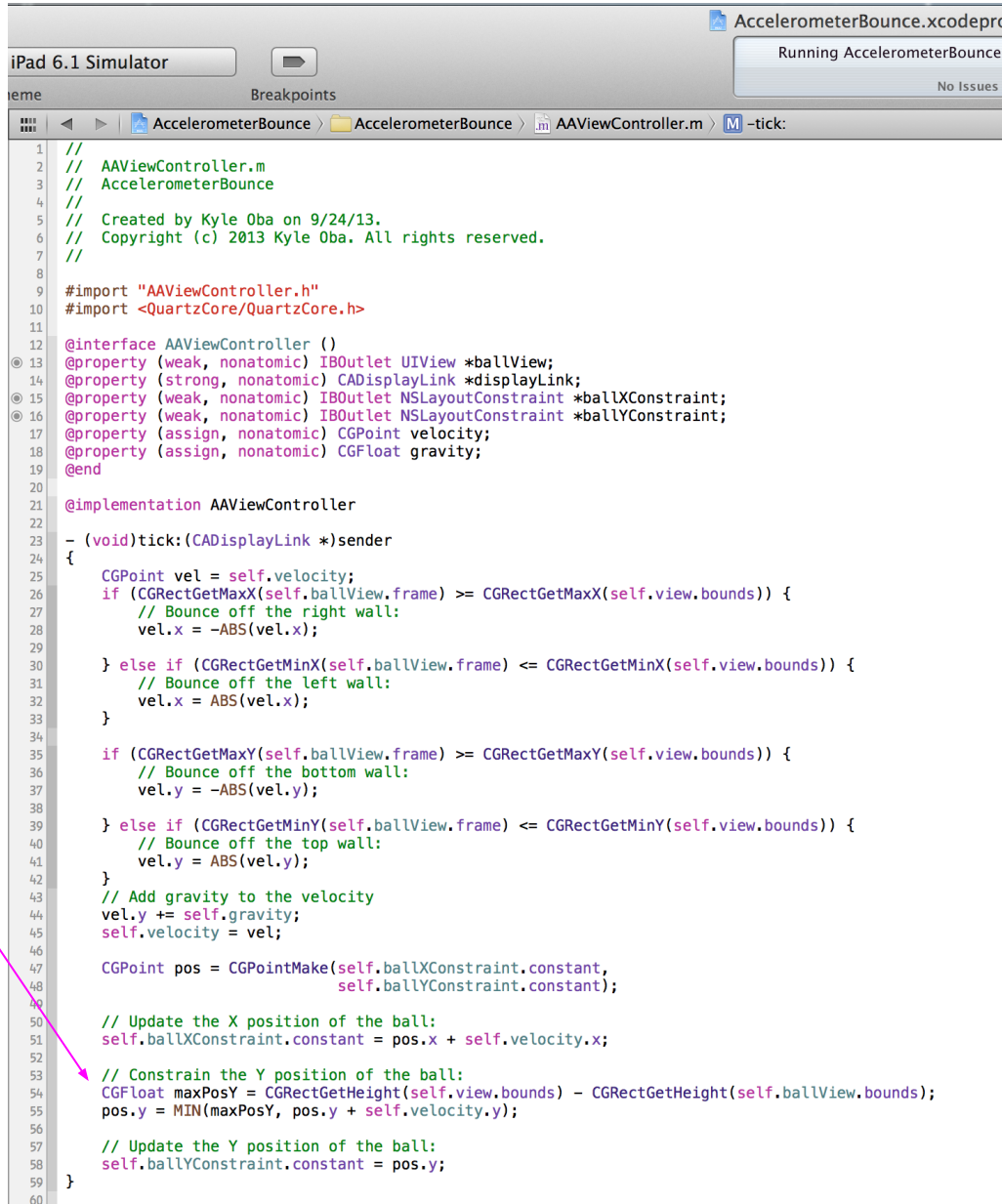
On the iPad, you keep the ball on the screen like this...

Now, instead of checking to if the ball is in the screen before adding gravity, we'll try something else. We'll try making sure that the y position of the ball is always on the screen.

How do you do that?

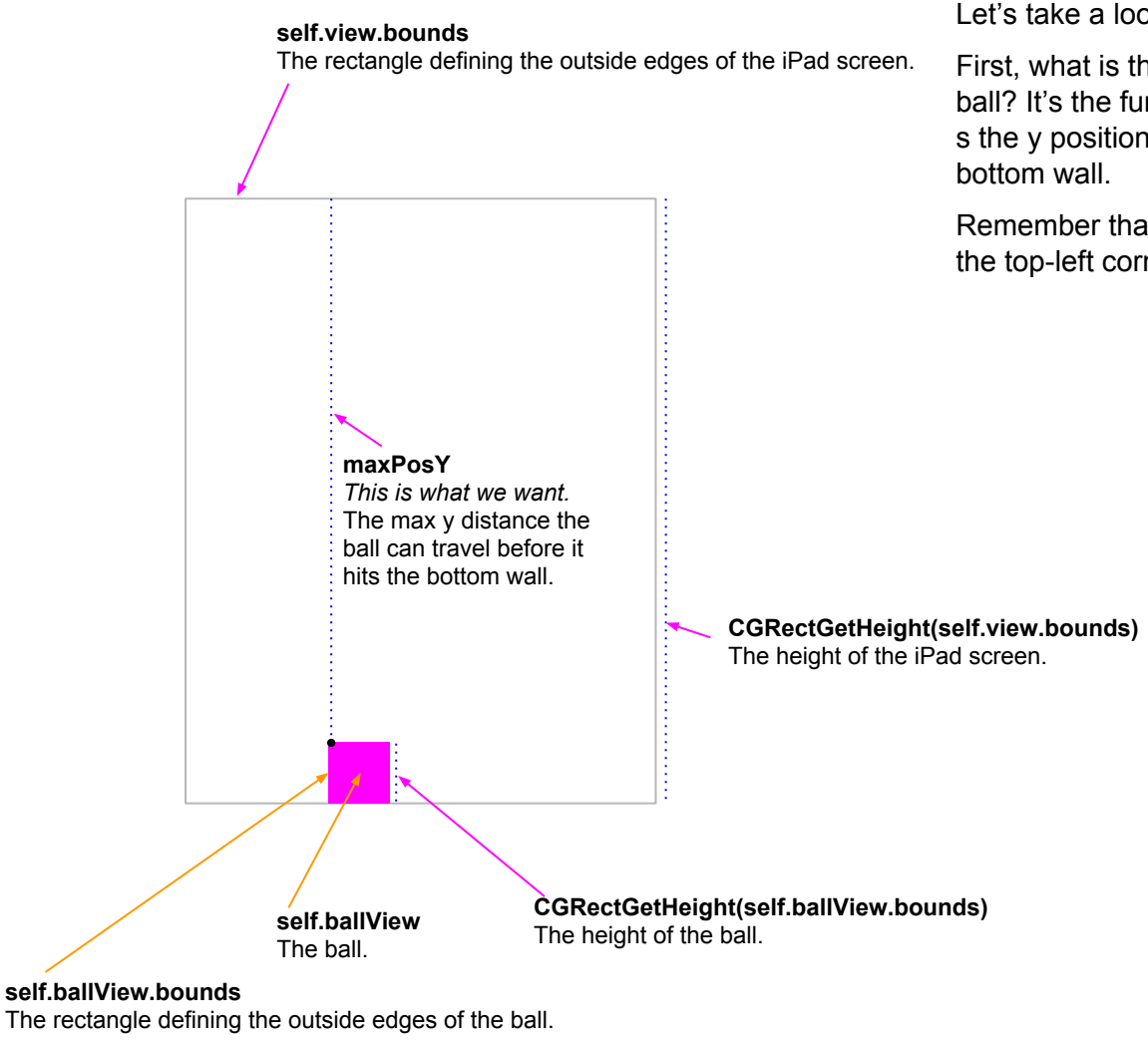
Well, one way is to constrain the **y** position of the ball, so that it is never larger than the screen height, minus the ball height.

Perhaps a diagram will help.




```
//
//  AAViewController.m
//  AccelerometerBounce
//
//  Created by Kyle Oba on 9/24/13.
//  Copyright (c) 2013 Kyle Oba. All rights reserved.
//

#import "AAViewController.h"
#import <QuartzCore/QuartzCore.h>

@interface AAViewController ()
@property (weak, nonatomic) IBOutlet UIView *ballView;
@property (strong, nonatomic) CADisplayLink *displayLink;
@property (weak, nonatomic) IBOutlet NSLayoutConstraint *ballXConstraint;
@property (weak, nonatomic) IBOutlet NSLayoutConstraint *ballYConstraint;
@property (assign, nonatomic) CGPoint velocity;
@property (assign, nonatomic) CGFloat gravity;
@end

@implementation AAViewController

- (void)tick:(CADisplayLink *)sender
{
    CGPoint vel = self.velocity;
    if (CGRectGetMaxX(self.ballView.frame) >= CGRectGetMaxX(self.view.bounds)) {
        // Bounce off the right wall:
        vel.x = -ABS(vel.x);

    } else if (CGRectGetMinX(self.ballView.frame) <= CGRectGetMinX(self.view.bounds)) {
        // Bounce off the left wall:
        vel.x = ABS(vel.x);
    }

    if (CGRectGetMaxY(self.ballView.frame) >= CGRectGetMaxY(self.view.bounds)) {
        // Bounce off the bottom wall:
        vel.y = -ABS(vel.y);

    } else if (CGRectGetMinY(self.ballView.frame) <= CGRectGetMinY(self.view.bounds)) {
        // Bounce off the top wall:
        vel.y = ABS(vel.y);
    }
    // Add gravity to the velocity
    vel.y += self.gravity;
    self.velocity = vel;

    CGPoint pos = CGPointMake(self.ballXConstraint.constant,
                              self.ballYConstraint.constant);

    // Update the X position of the ball:
    self.ballXConstraint.constant = pos.x + self.velocity.x;

    // Constrain the Y position of the ball:
    CGFloat maxPosY = CGRectGetHeight(self.view.bounds) - CGRectGetHeight(self.ballView.bounds);
    pos.y = MIN(maxPosY, pos.y + self.velocity.y);

    // Update the Y position of the ball:
    self.ballYConstraint.constant = pos.y;
}
```

First let's take a look at the "view."



Let's take a look at that bit of code a little closer.

First, what is the **maxPosY** (maximum y position) of the ball? It's the furthest down the screen the ball can go. It's the y position of the ball, exactly when it touches the bottom wall.

Remember that the position of the ball is measured from the top-left corner of the square.

Calculate the y constraint.

```
52
53      // Constrain the Y position of the ball:
54      CGFloat maxPosY = CGRectGetHeight(self.view.bounds) - CGRectGetHeight(self.ballView.bounds);
55      pos.y = MIN(maxPosY, pos.y + self.velocity.y);
56
57      // Update the Y position of the ball:
58      self.ballYConstraint.constant = pos.y;
59  }
60
```

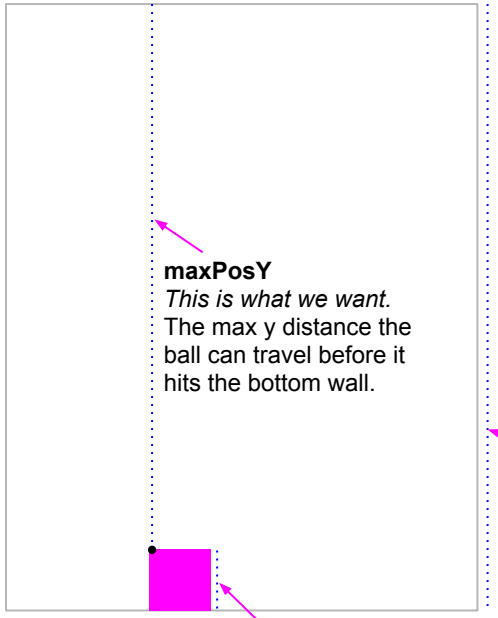
What is the furthest that the ball can go in the y direction before it hits the bottom wall?

Let's calculate that value and call it **maxPosY**.

To do this, we take the *height of the screen* and subtract the *height of the ball*.

This calculation gives us the exact position of the ball when it appears to hit the bottom of the screen.

Remember the ball's position is measured from the top-left corner.

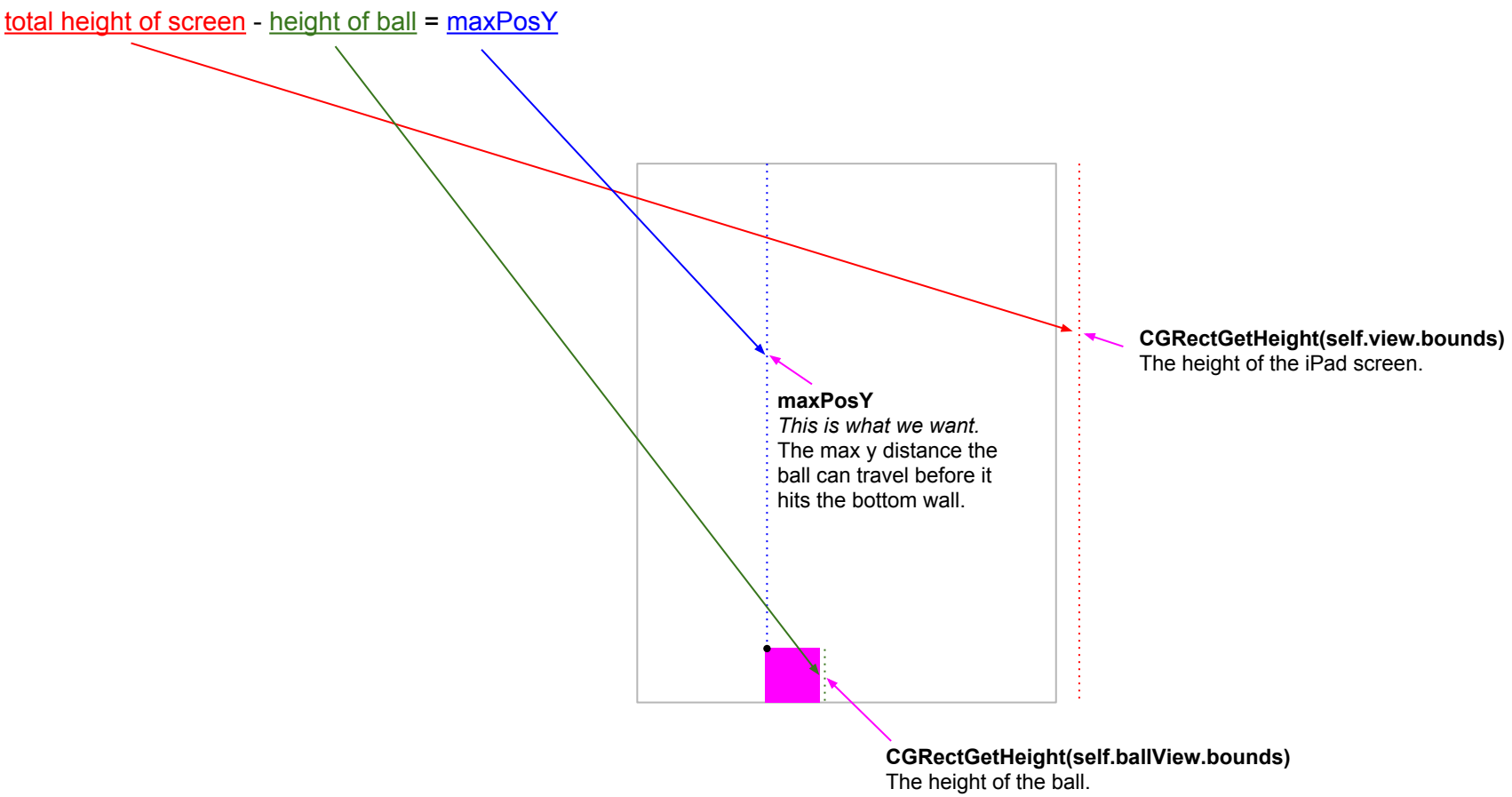


maxPosY
This is what we want.
The max y distance the ball can travel before it hits the bottom wall.

CGRectGetHeight(self.view.bounds)
The height of the iPad screen.

CGRectGetHeight(self.ballView.bounds)
The height of the ball.

In other words...



Constrain the ball's y position.

```
52
53      // Constrain the Y position of the ball:
54      CGFloat maxPosY = CGRectGetHeight(self.view.bounds) - CGRectGetHeight(self.ballView.bounds);
55      pos.y = MIN(maxPosY, pos.y + self.velocity.y);
56
57      // Update the Y position of the ball:
58      self.ballYConstraint.constant = pos.y;
59  }
60
```

Now we have to make sure the **y** position of the ball never gets a higher value than this **maxPosY** value. This will make sure the ball always appears to be on the screen.

We do this with the **MIN()** function.

The **MIN()** function take two values and returns the lower one. We will pass it the **maxPosY** and the new calculated position (**y** position + **y** velocity).

We want to use the one that's lower.

That way, we'll never end up with a **y** position that is greater than the **maxPosY** value.

If you **Run** your app now, the ball should bounce and appear to suffer from downward pull of gravity.

We'll skip the dampening for now, since things look *real* enough. We'll add it later.

Let's *really* use the iPad now. Enter the CMMotionManager class.

What the heck is the **CMMotionManager** class?

And, for that matter, what the heck is a **class**?

We'll learn more about **classes** later. For now, suffice it to say that it's a *thing*. A thing that you'll use to make your program do *stuff*.

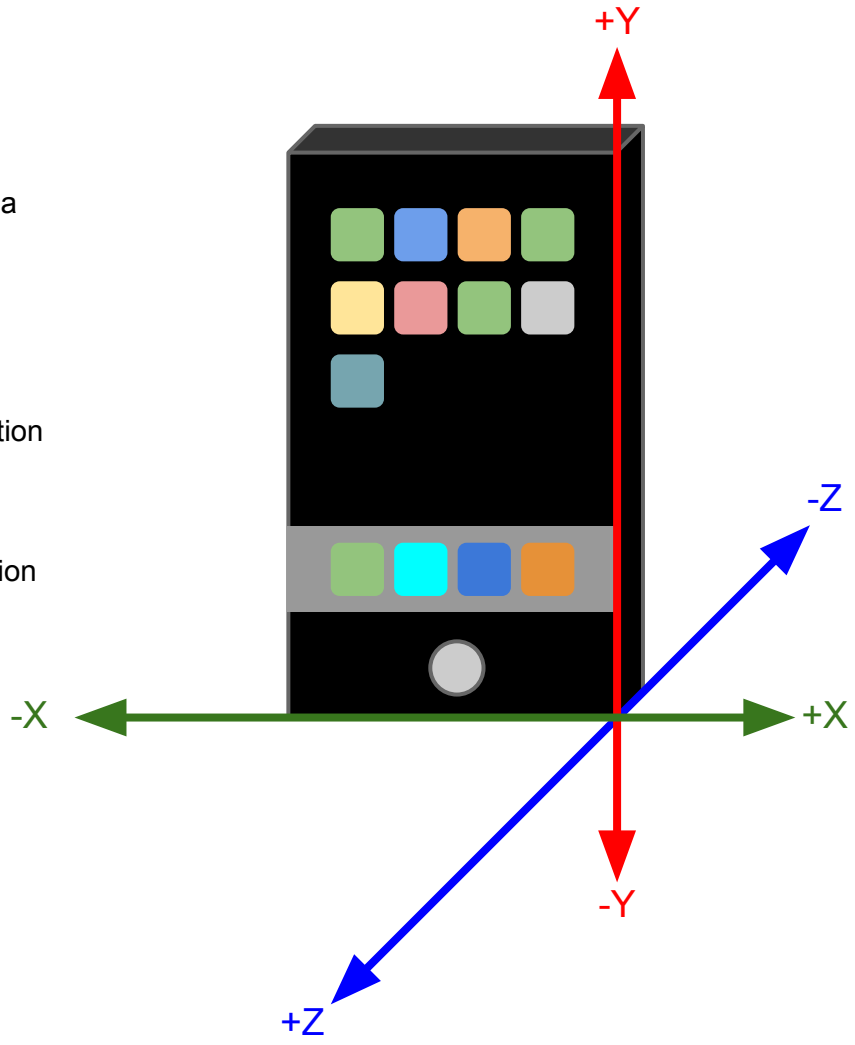
Science!

Okay, so what's a **CMMotionManager**?

All iPads (and iPhones and iPod Touches) come with a small detector embedded inside them. This detector allows us to know what acceleration the device feels in the x, y, and z directions.

It's a piece of electronic circuitry that allows them to detect the orientation of the device in space. It uses

I'll allow you to find out for yourself...



Use the Xcode documentation.

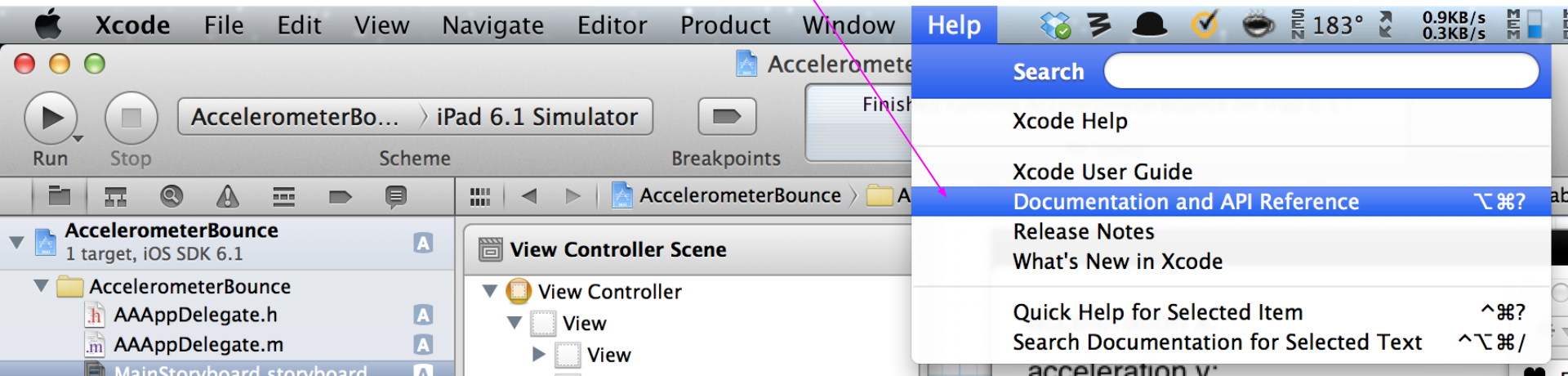
Find it in the documentation and read about it.

Hit **option-⌘-SHIFT-?** all at the same time. That will open up Xcode's documentation.

If you don't want to remember (or forget) that keyboard shortcut, you can use the help menu.

Hit **Help > Documentation and API Reference**

to open up the same thing.



Search for help...

In the search bar, enter **Handling Accelerometer Events**.

Then, hit enter.

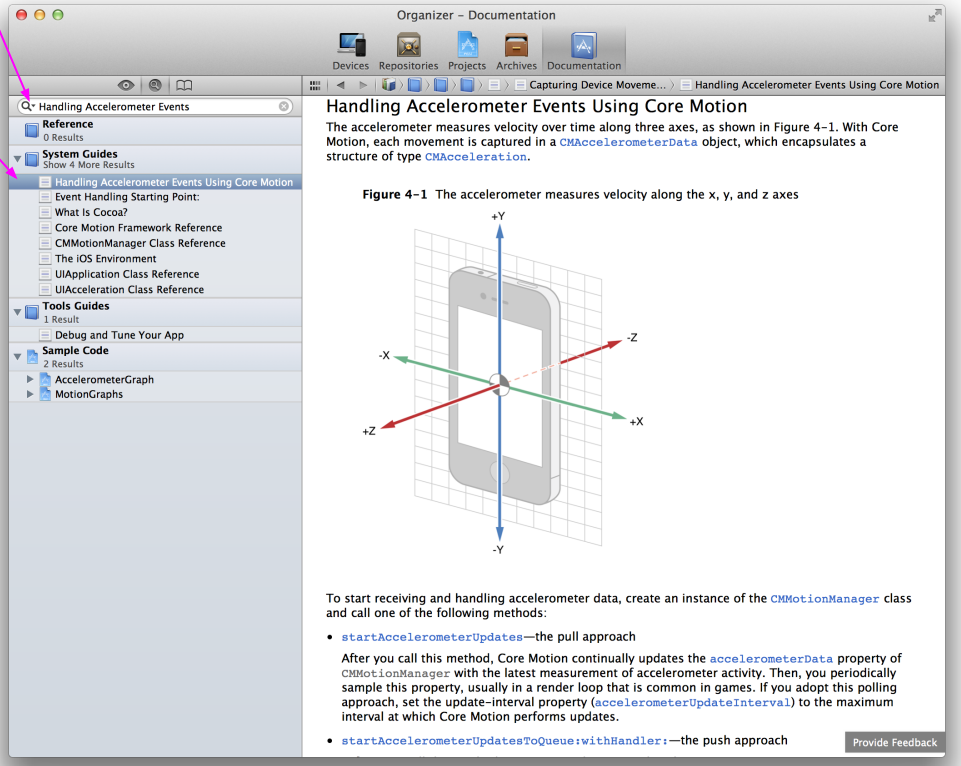
You'll be shown a list of search results here.

Click on the one that says **Handling Accelerometer Events Using Core Motion**.

Pro-tip: That **CM** in **CMMotionManager** stands for Core Motion.

Once you select the item you want, you will be taken to that exact section of the documentation.

This is but a small section in a vast pile of documentation.



Find where you are in the docs.

To find out exactly where this section lives in the documentation library, two-finger tap (right-click) the detail side of the documentation (right-side).

This context menu will pop up.

Select **Reveal in Library**.

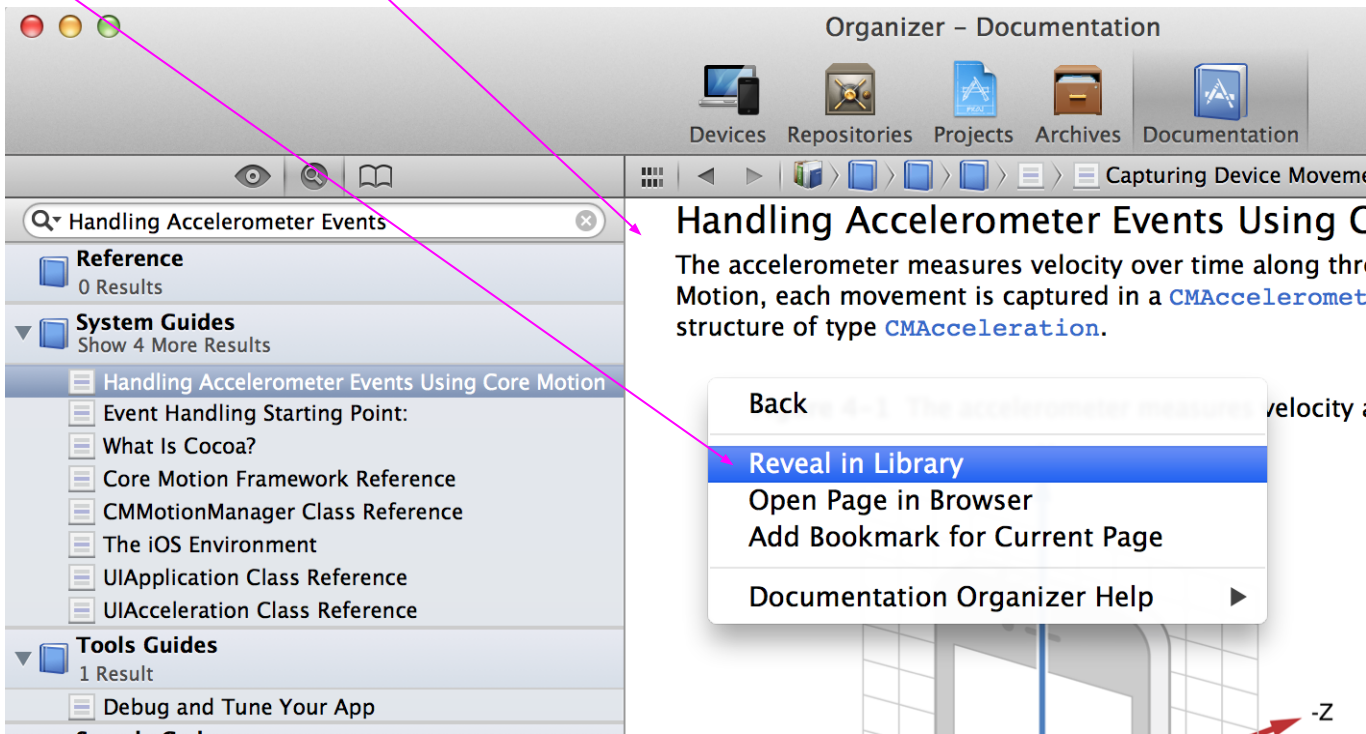


Table of Contents, of sorts

If you look at the left pane, you'll see that you're now looking at all the documentation topics in the iOS doc set (for your current version, 6.1).

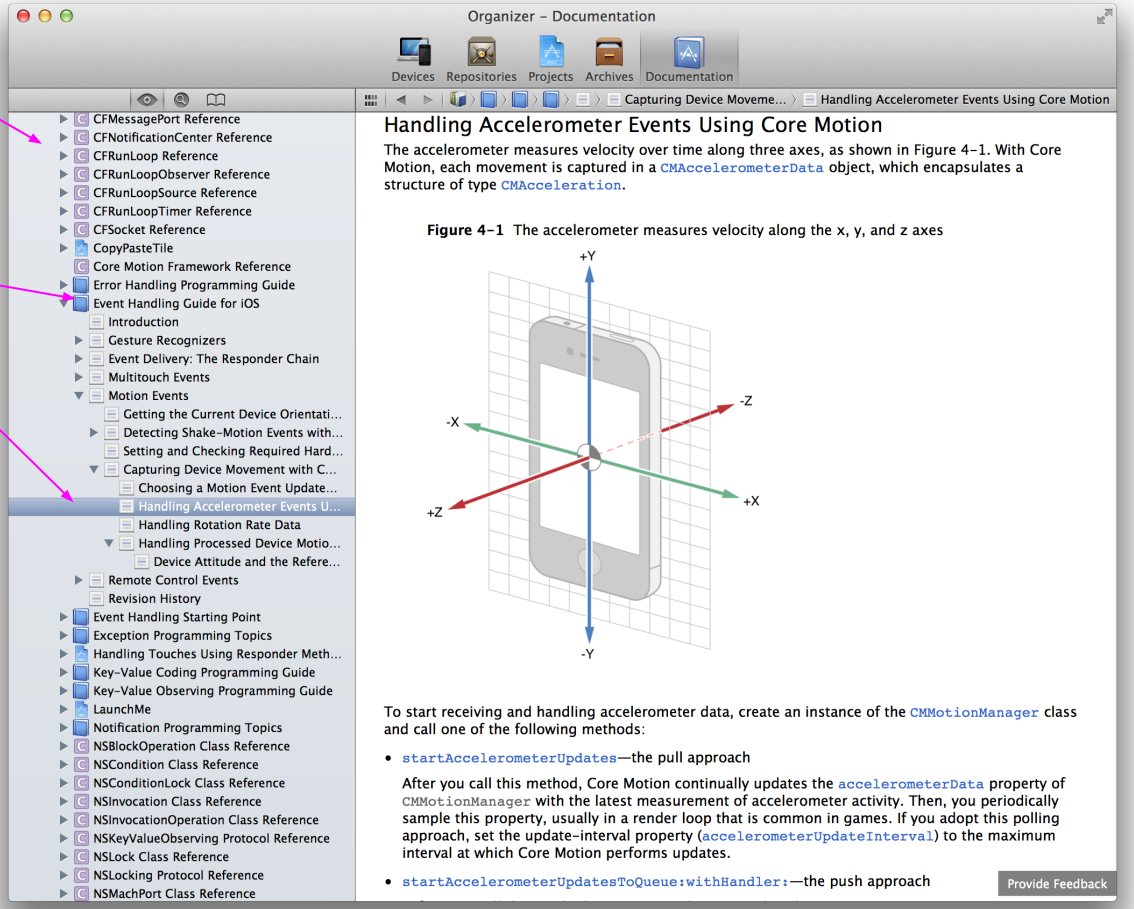
The selection you made is placed in context here.

What I'd like to point out here is that the section you're looking at is part of a larger *guide* named **Event Handling Guide for iOS**.

Guides are denoted by this blue book icon.

There are several guides available, on a bunch of different topics. This is how you'll learn a lot about how to build things on iOS (if you ever want to build something cool, or get paid).

If you want to build apps that suck. Totally ignore these guides.



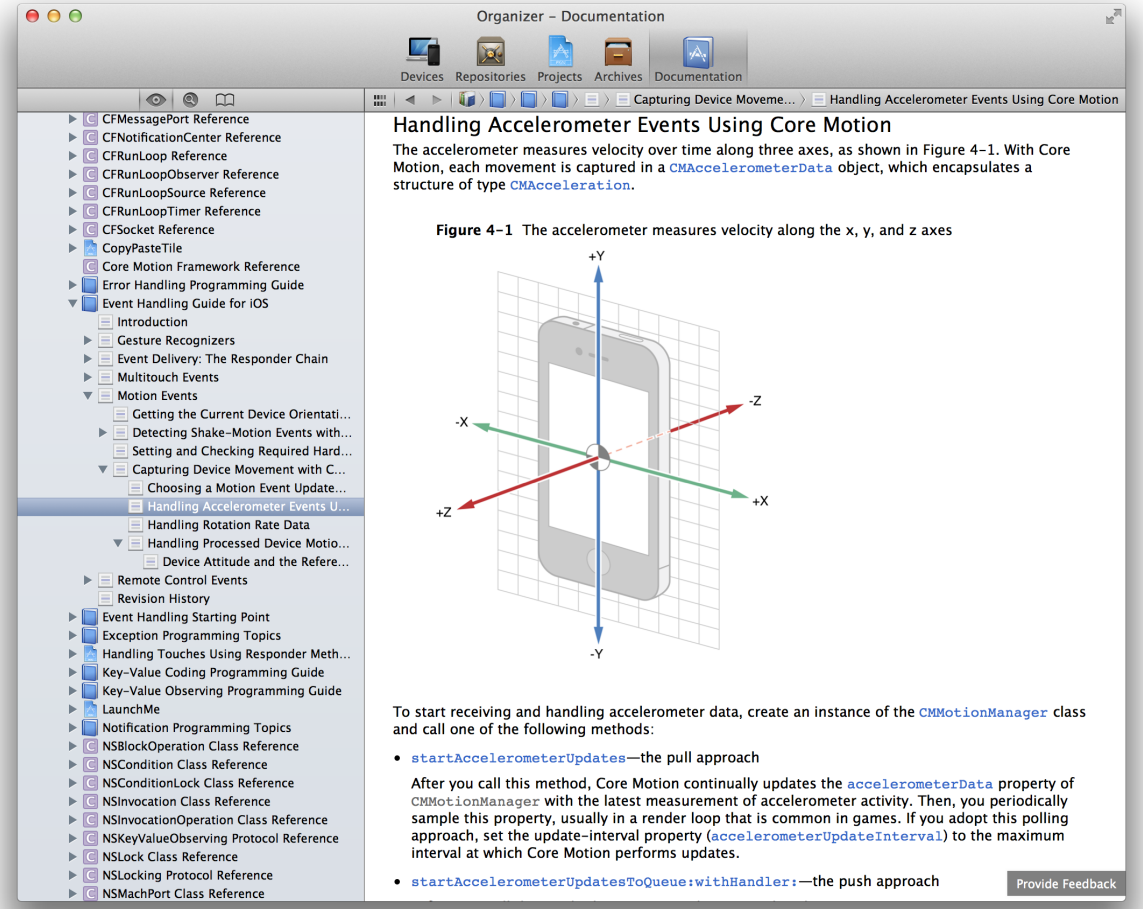
Skim (TL;DR)

I was going to tell you to read the section titled **Handling Accelerometer Events Using Core Motion**, but then I realized you would probably think it was ridiculously insane.

So, instead, just skim it. I want you to get a feel for this now. Don't try to understand every bit of it. But, you should try to see what it kind of is talking about.

We'll dive into this type of stuff later in the semester, after you've learned a few more advanced topics.

Stop skimming when you get to the **Handling Rotation Rate Data** section.

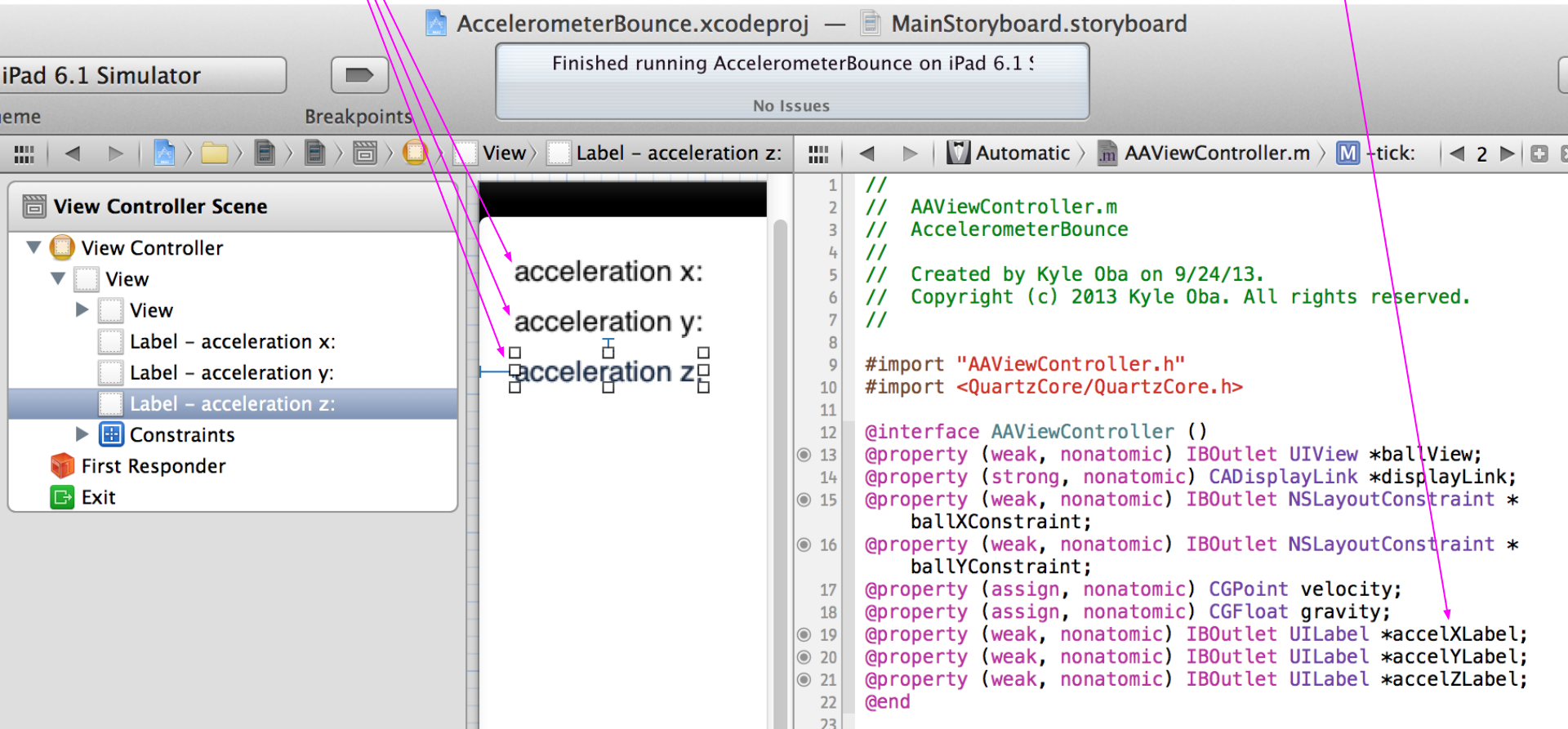


We're going to read the device's x, y, & z acceleration.

Now, add three labels (UILabels) to your storyboard. We'll use these labels to view the acceleration felt by your iPad.

Then connect your labels to the implementation (.m) file. You remember how to do that, right?

Refresher: To connect, hold down control, then click and drag into your implementation file. See the names I gave them?



Add the Core Motion framework.

Remember when we added the Quartz Core framework? We needed it to work with the **displayLink**.

Now we're going to add the Core Motion framework. We need it to work with the accelerometer.

Click on the project to select it.

Then, select the targets item.

Scroll down to the **Linked Frameworks** section.

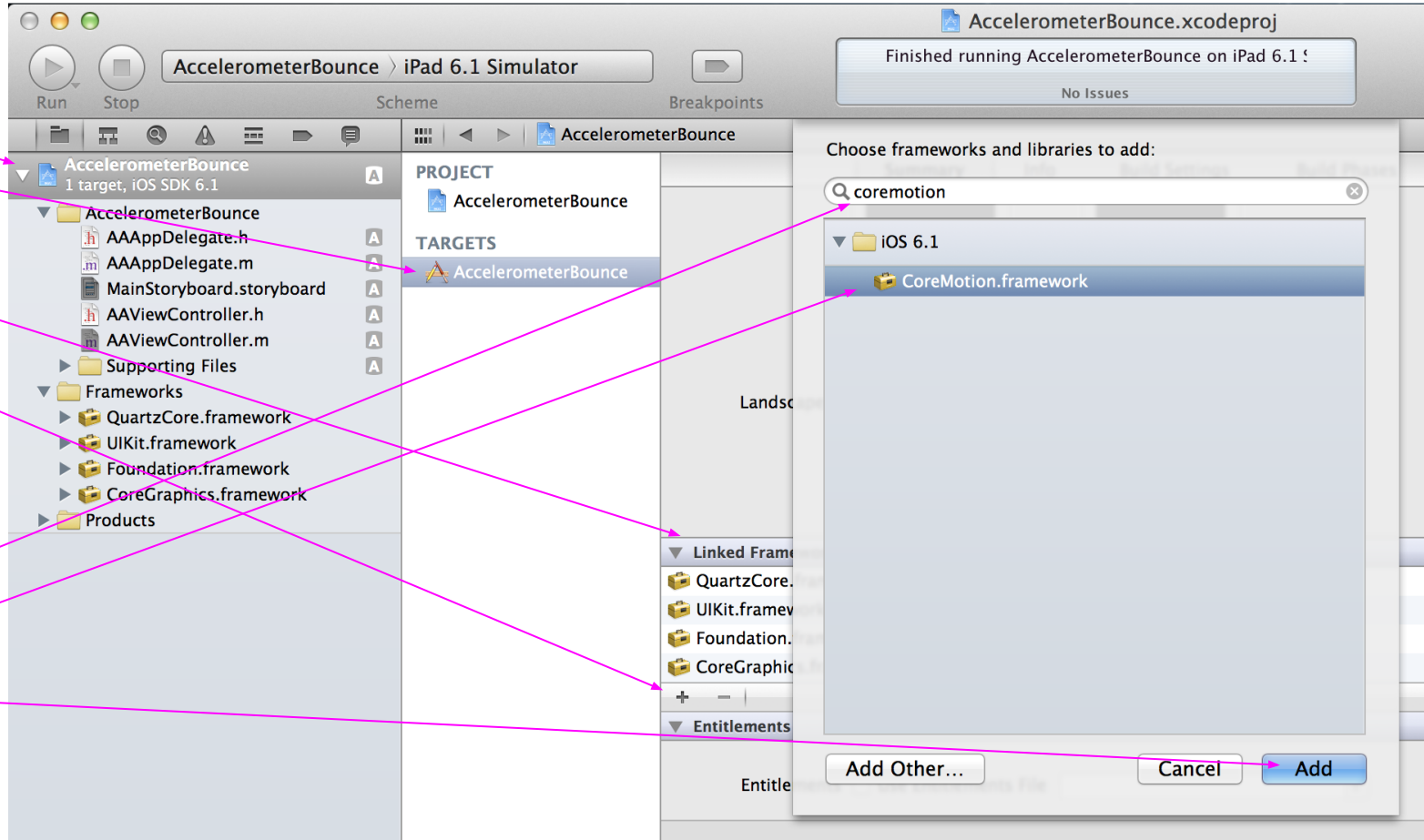
Press the **+** button to add a new framework.

Search for **coremotion**.

Select it from the list.

Press the **Add** button.

Linking complete.



Take a deep breath.

This next section is going to get a little crazy.

We're going to create the Core Motion Motion Manager, and use it to read data from the accelerometer.

The accelerometer data will tell us what acceleration the iPad is feeling (once per frame).

Accelerometer data is measured in units of gravity (g). So, at rest, sitting flat on your table, it should register about 1g toward the center of the Earth.

So, imaging this iPad here is sitting on its back on a horizontal table. In this position, its screen would be facing the sky.

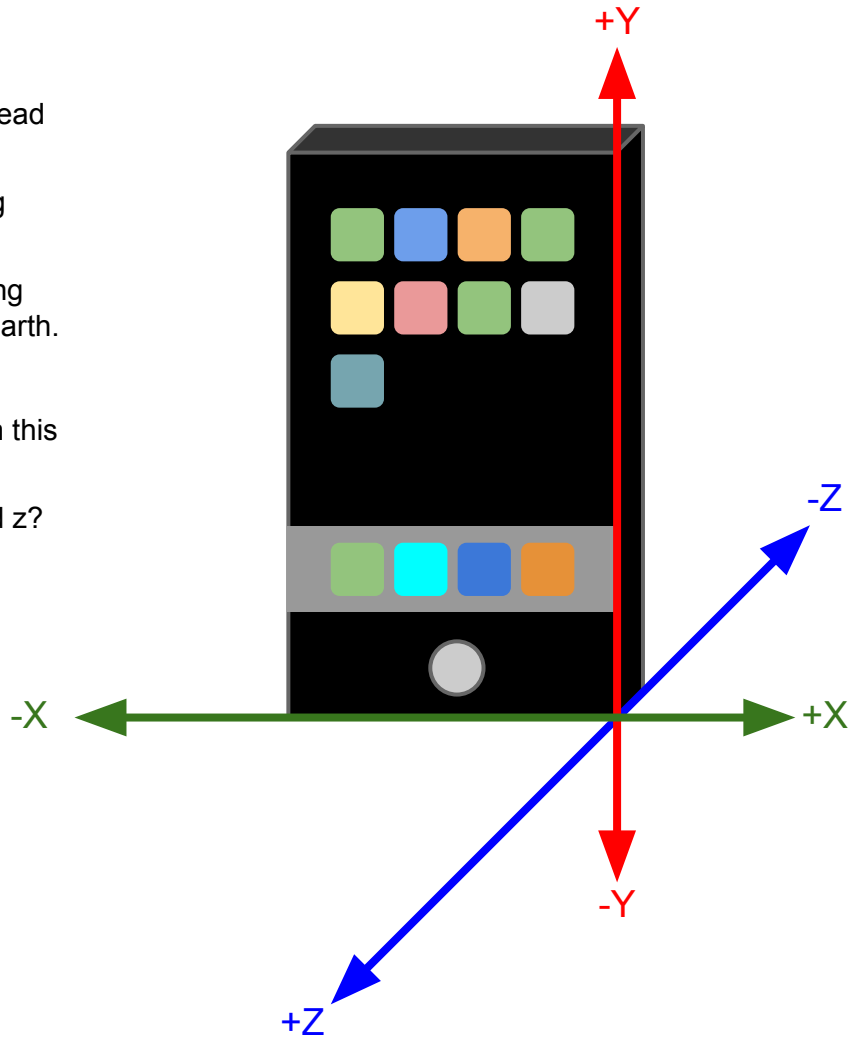
What would be the accelerometer readings for each direction, x, y, and z?

Answer:

x: 0 g

y: 0 g

z: -1 g



Ready?

Add a Core Motion, Motion Manager.

Okay, we've got the Core Motion framework in place.

Now we can add a Core Motion Motion Manager.

We'll call ours **motionManager**.

Add a property for it here.

You also want to import the **CoreMotion** header here.

```
1  //
2  //  AAViewController.m
3  //  AccelerometerBounce
4  //
5  //  Created by Kyle Oba on 9/24/13.
6  //  Copyright (c) 2013 Kyle Oba. All rights reserved.
7  //
8
9  #import "AAViewController.h"
10 #import <QuartzCore/QuartzCore.h>
11 #import <CoreMotion/CoreMotion.h>
12
13 @interface AAViewController ()
14 @property (weak, nonatomic) IBOutlet UIView *ballView;
15 @property (strong, nonatomic) CADisplayLink *displayLink;
16 @property (weak, nonatomic) IBOutlet NSLayoutConstraint *ballXConstraint;
17 @property (weak, nonatomic) IBOutlet NSLayoutConstraint *ballYConstraint;
18 @property (assign, nonatomic) CGPoint velocity;
19 @property (assign, nonatomic) CGFloat gravity;
20
21 @property (weak, nonatomic) IBOutlet UILabel *accelXLabel;
22 @property (weak, nonatomic) IBOutlet UILabel *accelYLabel;
23 @property (weak, nonatomic) IBOutlet UILabel *accelZLabel;
24 @property (strong, nonatomic) CMMotionManager *motionManager;
25 @end
```

Create the Motion Manager.

Remember how we added a **displayLink** to manage getting our app to call our **tick** method on every frame?

Well, now we need a **motionManager** to manage getting accelerometer readings on every frame.

Here's where we're going to start mixing it up. Instead of dumping a whole bunch of code right there under the displayLink stuff, we're going to create a new method to handle setting up the **motionManger**.

We call this method **setupMotionManager**.

```
73
74  - (void)viewDidLoad
75  {
76      [super viewDidLoad];
77      // Do any additional setup after loading the view, typically from a nib.
78
79      self.velocity = CGPointMake(10.0, 10.0);
80      self.gravity = 5.0;
81
82      self.displayLink = [CADisplayLink displayLinkWithTarget:self selector:@selector(tick:)];
83      [self.displayLink addToRunLoop:[NSRunLoop currentRunLoop] forMode:NSDefaultRunLoopMode];
84
85      [self setupMotionManager];
86  }
87
88  - (void)setupMotionManager
89  {
90      self.motionManager = [[CMMotionManager alloc] init];
91
92      if ([self.motionManager isAccelerometerAvailable]) {
93          self.motionManager.accelerometerUpdateInterval = 1.0/60.0;
94          [self.motionManager startAccelerometerUpdates];
95      } else {
96          NSLog(@"Accelerometer is not active.");
97      }
98  }
```

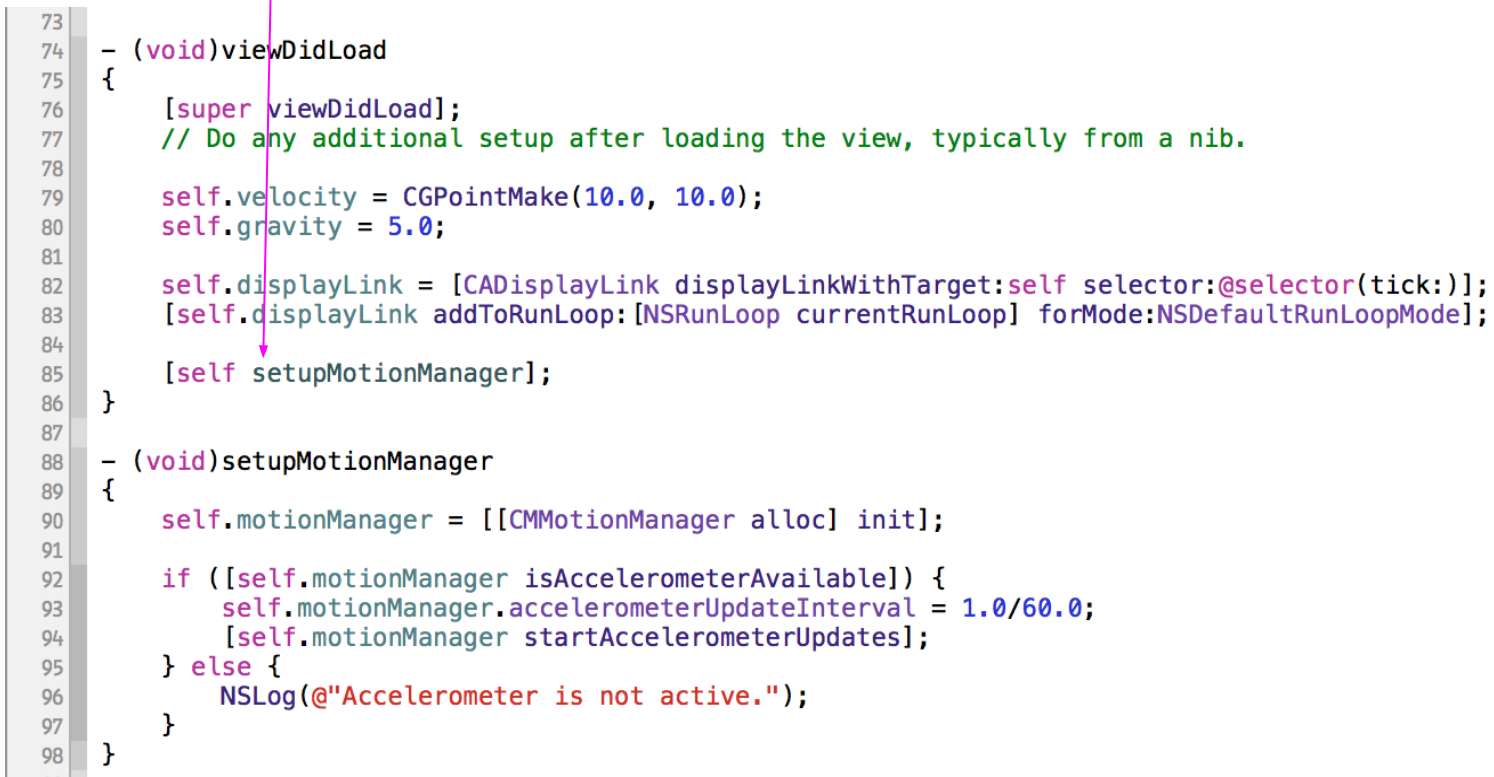
Execute the method.

It's not enough to just have a well-written **setupMotionManager** method.

We also have to *execute* this method.

Here's how you execute the **setupMotionManager** method.

Executing the method will cause our app to run all the code, just as if it was cut-&-pasted into the spot where we executed it.



```
73
74  - (void)viewDidLoad
75  {
76      [super viewDidLoad];
77      // Do any additional setup after loading the view, typically from a nib.
78
79      self.velocity = CGPointMake(10.0, 10.0);
80      self.gravity = 5.0;
81
82      self.displayLink = [CADisplayLink displayLinkWithTarget:self selector:@selector(tick:)];
83      [self.displayLink addToRunLoop:[NSRunLoop currentRunLoop] forMode:NSDefaultRunLoopMode];
84
85      [self setupMotionManager];
86  }
87
88  - (void)setupMotionManager
89  {
90      self.motionManager = [[CMMotionManager alloc] init];
91
92      if ([self.motionManager isAccelerometerAvailable]) {
93          self.motionManager.accelerometerUpdateInterval = 1.0/60.0;
94          [self.motionManager startAccelerometerUpdates];
95      } else {
96          NSLog(@"Accelerometer is not active.");
97      }
98  }
```

Let’s take a closer look.

Okay. All fine and good. But, what’s all that code *in* the **setupMotionManager** method?

Pretend you understand why the wording is so weird. I just want you to understand, in general what we’re doing here.

First we create the **motionManager**.

Then we check to see if there is an accelerometer available. Some devices may restrict your use of it, or not have one at all (in the future). Also, there’s no accelerometer on the Simulator.

If it’s available, we set our **accelerometerUpdateInterval** to 60 times per second. This is measured in seconds. So, 1/60 is 1/60th of a second. This is about how often we see a new frame.

We then tell the motionManager to start generating data. This pulls data from the accelerometer. We have to explicitly turn this on, because by default it’s off. Using the accelerometer drains the battery. So, you wouldn’t want this on unless someone (you) wants the data.

Finally, we just write a log message if the accelerometer isn’t available.

```
87
88  - (void)setupMotionManager
89  {
90      self.motionManager = [[CMMotionManager alloc] init];
91
92      if ([self.motionManager isAccelerometerAvailable]) {
93          self.motionManager.accelerometerUpdateInterval = 1.0/60.0;
94          [self.motionManager startAccelerometerUpdates];
95      } else {
96          NSLog(@"Accelerometer is not active.");
97      }
98  }
99
```

Display the accelerometer data.

Okay, now you've created and set up your motionManager.

You've told it to pull acceleration values off your accelerometer ever 1/60th of a second (approximately once per frame).

Now, before we plug that data into the velocity of our ball, let's just print the values on the screen.

We'll use those labels you created way back a few slides ago. Remember those slides? Those were the days.

See if this makes sense to you.

We're taking the **x**, **y** & **z** values of the **accelerometerData**'s **acceleration** readings.

We're then formatting a string of text. It uses **%.2f** to limit the digits displayed to two after the decimal point. If you didn't do this, you'd see 1.0010101010101010 or something crazy like that.

and then setting it to the **text** value of the labels.

Go ahead and Run it (on your iPad). You should see fluctuating acceleration numbers.

Rotate your iPad around to see how the numbers change. Remember it's in units of gravity.

```
66
67      // Update accelerometer labels:
68      CMAccelerometerData *accelerometerData = self.motionManager.accelerometerData;
69      self.accelXLabel.text = [NSString stringWithFormat:@"acceleration x: %.2f",accelerometerData.acceleration.x];
70      self.accelYLabel.text = [NSString stringWithFormat:@"acceleration y: %.2f",accelerometerData.acceleration.y];
71      self.accelZLabel.text = [NSString stringWithFormat:@"acceleration z: %.2f",accelerometerData.acceleration.z];
72  }
73
74  - (void)viewDidLoad
75  {
```

Run it and relax.

Go ahead and Run it (on your iPad). You should see fluctuating acceleration numbers.

Rotate your iPad around to see how the numbers change.

Seriously, hold it at all angles and upside down and stuff.

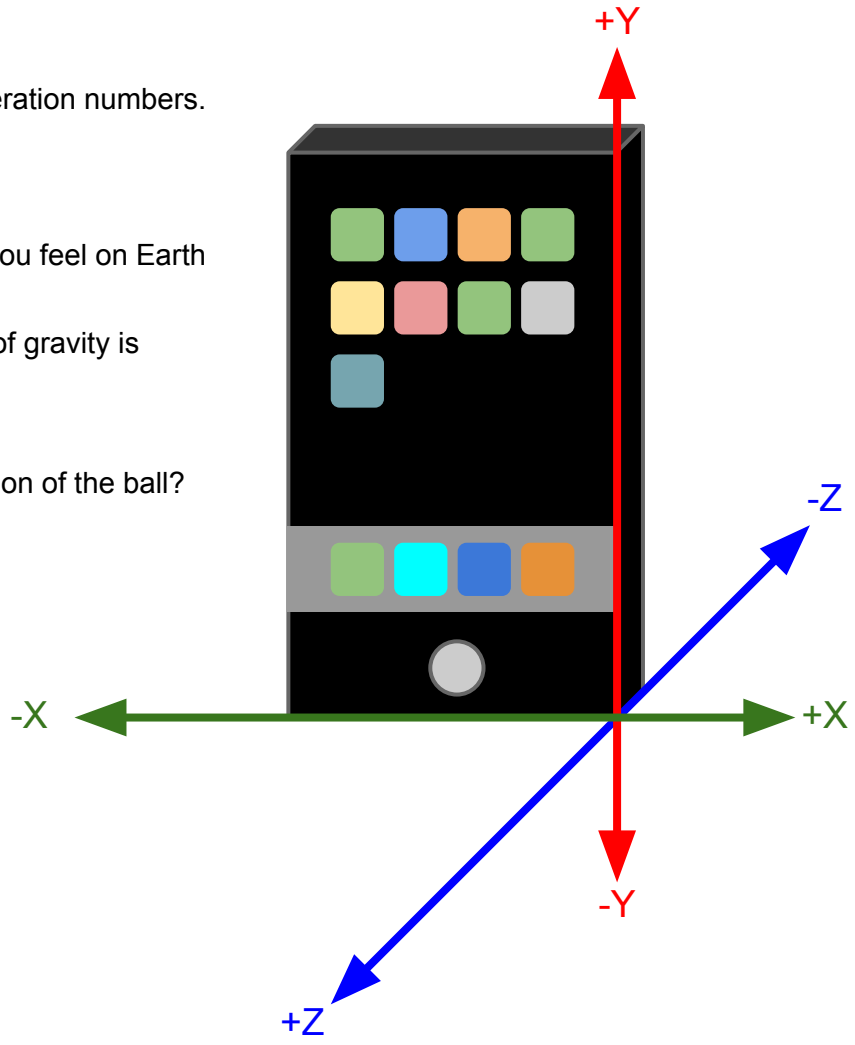
Remember it's in units of gravity. So, 1 g is approximately the gravity you feel on Earth (at see level presumably).

The negative/positive sign is just there to tell you which way the force of gravity is pulling.

How are we going to use these accelerometer values to affect the motion of the ball?

Any ideas?

I'll try to keep it simple. Proceed when ready.



Simplify. Remove rotation.

Did you notice how, when you rotated the screen, the whole display rotated?

That's the iPad trying to make sure the screen is always oriented so that you can read things.

Well, with our bouncing ball example, we don't really want that. So, lets turn rotation off.

Remember how we added the Core Motion framework? We need to go to the same place to turn rotation off.

Click on your project to select it.

Select the targets item.

Find the **Supported Interface Orientations** section.

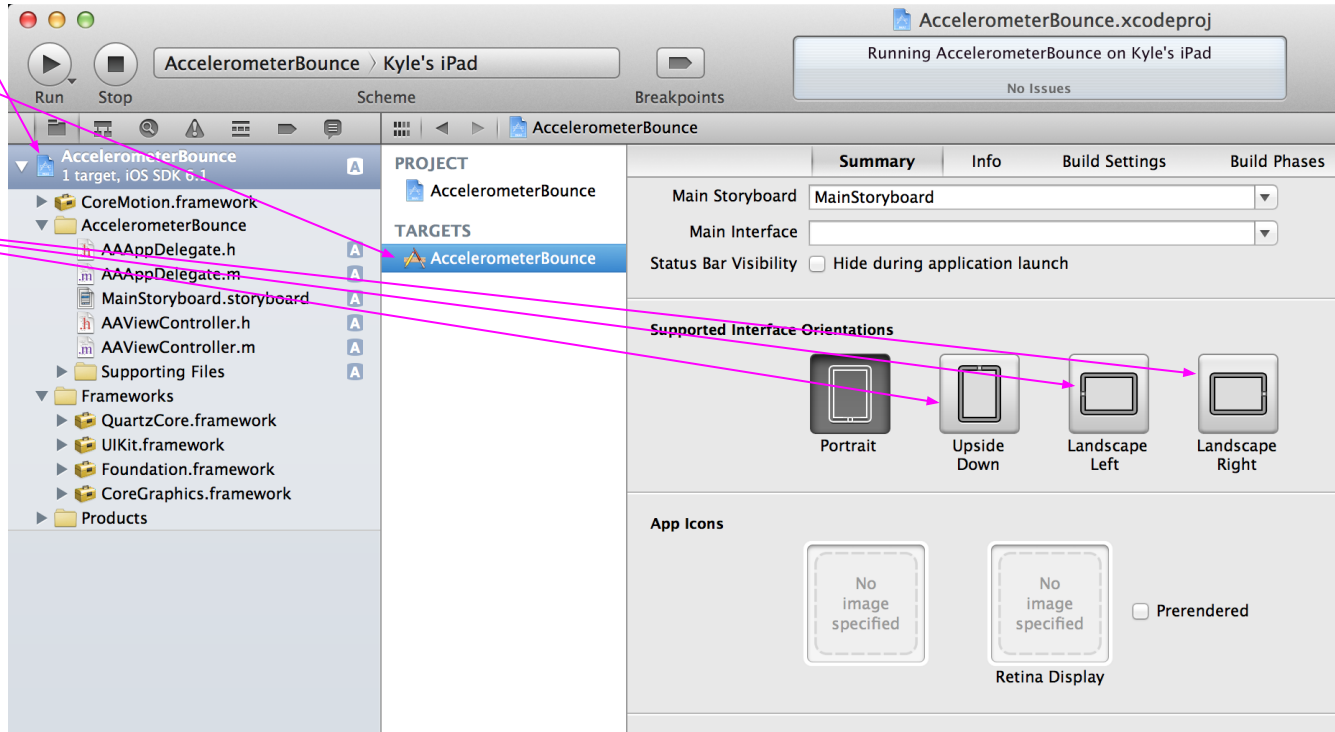
You need to click on all the orientation buttons, except for the **Portrait** one.

Leave the **Portrait** one selected.

What this does is tell your app that you don't want to allow rotation to any of those other orientations.

So, now, your app will always stay in the **Portrait** orientation.

Try running it now to see.



The pull of gravity.

If you've been paying rapt attention, you'll notice that we need to use the **x** & **y** components of the acceleration.

We can ignore the **z** value, because this is a two-dimensional (2D) simulation.

When the **x** acceleration value is positive (between 0 and 1) we want to push the ball right.

When the **x** acceleration value is negative (between -1 and 0) we want to push it left.

Here's the catch, the y value is flipped.

When the **y** acceleration value is positive (between 0 and 1) we want to push the ball LEFT.

When the **y** acceleration value is negative (between -1 and 0) we want to push it RIGHT.

Add gravitational pull to the velocity.

We're going to implement a new method.

I'm calling it **updateVelocityWithAcceleration**.

We'll execute it here.

Note, we're executing it *after* the bouncing-off-the-walls code updates the velocity's direction.

We're replacing our old method of adding gravity to velocity. I've commented it out.

You can just delete this line.

Also, look how I've replaced all that business about updating the labels which display the acceleration data in your labels.

We've moved it. It's not here any more.

See next slide.

```
28
29  - (void)tick:(CADisplayLink *)sender
30  {
31      CGPoint vel = self.velocity;
32      if (CGRectGetMaxX(self.ballView.frame) >= CGRectGetMaxX(self.view.bounds)) {
33          // Bounce off the right wall:
34          vel.x = -ABS(vel.x);
35
36      } else if (CGRectGetMinX(self.ballView.frame) <= CGRectGetMinX(self.view.bounds)) {
37          // Bounce off the left wall:
38          vel.x = ABS(vel.x);
39      }
40
41      if (CGRectGetMaxY(self.ballView.frame) >= CGRectGetMaxY(self.view.bounds)) {
42          // Bounce off the bottom wall:
43          vel.y = -ABS(vel.y);
44
45      } else if (CGRectGetMinY(self.ballView.frame) <= CGRectGetMinY(self.view.bounds)) {
46          // Bounce off the top wall:
47          vel.y = ABS(vel.y);
48      }
49      // Add gravity to the velocity
50  //    vel.y += self.gravity;
51      self.velocity = vel;
52      [self updateVelocityWithAcceleration];
53
54      CGPoint pos = CGPointMake(self.ballXConstraint.constant,
55                                self.ballYConstraint.constant);
56
57      // Update the X position of the ball:
58      self.ballXConstraint.constant = pos.x + self.velocity.x;
59
60      // Constrain the Y position of the ball:
61      CGFloat maxPosY = CGRectGetHeight(self.view.bounds) - CGRectGetHeight(self.ballView.bounds);
62      pos.y = MIN(maxPosY, pos.y + self.velocity.y);
63
64      // Update the Y position of the ball:
65      self.ballYConstraint.constant = pos.y;
66  }
67
```

Create the new method.

Here's the new **updateVelocityWithAcceleration** method.

See how we moved the label update code here? It's good to keep all related things in one place.

Also, we're going to use that **accelerometerData** thing again.

Here's where we multiply scale our acceleration.x value by the gravity value.

We then add it to the x velocity.

We do the same for the y direction, but keeping in mind that we must flip the y direction of the acceleration.

```
81
82  - (void)updateVelocityWithAcceleration
83  {
84      // Update accelerometer labels:
85      CMAccelerometerData *accelerometerData = self.motionManager.accelerometerData;
86      self.accelXLabel.text = [NSString stringWithFormat:@"acceleration x: %.2f",accelerometerData.acceleration.x];
87      self.accelYLabel.text = [NSString stringWithFormat:@"acceleration y: %.2f",accelerometerData.acceleration.y];
88      self.accelZLabel.text = [NSString stringWithFormat:@"acceleration z: %.2f",accelerometerData.acceleration.z];
89
90      // Update velocity:
91      CGPoint vel = self.velocity;
92      vel.x += accelerometerData.acceleration.x * self.gravity;
93      vel.y += -accelerometerData.acceleration.y * self.gravity;
94      self.velocity = vel;
95  }
```

Run it.

Now your ball bounces and appears to feel the pull of gravity.

Yay.

Some things

The ball can disappear off the other sides (top, left, and right sides), because we haven't constrained the position of the ball like we did with the bottom wall.

How would you implement dampening (when the ball hits a wall)?

There's no dampening. If you shake your iPad (making the acceleration get larger than 1 g), the ball gets pretty fast. Dampening would help this look more *real*.

One more thing… fix the dampening.

Let's fix the dampening. The trick is to reduce the velocity by some small amount each time the ball hits a wall.

Here's I'm defining a constant that says we'll reduce the velocity to 90% of it's former value.

How do I tell when the ball hits a wall?

Well, let's do it a little different from how we did it in Processing.

I'll just check to see if we've reversed the velocity. If we have, then we'll dampen the velocity a bit.

How do I check if we've reversed the velocity?

If the **vel.x** value is not the same as the original **self.velocity.x** value, then it has changed.

Same goes for **vel.y**.

If dampening occurs, then we multiply either **vel.x** or **vel.y** by the **DAMPENING_FACTOR** of 90%.

That's it. **Run** it again.

```
26
27  @implementation AAViewController
28
29  #define DAMPENING_FACTOR 0.9;
30
31  - (void)tick:(CADisplayLink *)sender
32  {
33      CGPoint vel = self.velocity;
34      if (CGRectGetMaxX(self.ballView.frame) >= CGRectGetMaxX(self.view.bounds)) {
35          // Bounce off the right wall:
36          vel.x = -ABS(vel.x);
37
38      } else if (CGRectGetMinX(self.ballView.frame) <= CGRectGetMinX(self.view.bounds)) {
39          // Bounce off the left wall:
40          vel.x = ABS(vel.x);
41      }
42
43      if (CGRectGetMaxY(self.ballView.frame) >= CGRectGetMaxY(self.view.bounds)) {
44          // Bounce off the bottom wall:
45          vel.y = -ABS(vel.y);
46
47      } else if (CGRectGetMinY(self.ballView.frame) <= CGRectGetMinY(self.view.bounds)) {
48          // Bounce off the top wall:
49          vel.y = ABS(vel.y);
50      }
51      // Add dampening:
52      if (self.velocity.x != vel.x) {
53          vel.x *= DAMPENING_FACTOR;
54      }
55      if (self.velocity.y != vel.y) {
56          vel.y *= DAMPENING_FACTOR;
57      }
58
59      // Add gravity to the velocity
60      self.velocity = vel;
61      [self updateVelocityWithAcceleration];
62
63      CGPoint pos = CGPointMake(self.ballXConstraint.constant,
64                                self.ballYConstraint.constant);
65
66      // Update the X position of the ball:
67      self.ballXConstraint.constant = pos.x + self.velocity.x;
68
69      // Constrain the Y position of the ball:
70      CGFloat maxPosY = CGRectGetHeight(self.view.bounds) - CGRectGetHeight(self.ballView.bounds);
71      pos.y = MIN(maxPosY, pos.y + self.velocity.y);
72
73      // Update the Y position of the ball:
74      self.ballYConstraint.constant = pos.y;
75  }
76
```

Run it again.

Now your ball bounces and seems to slow down due to collisions with the walls.

Another thing.

Hold your iPad upside down, or on a side.

The ball disappears again. Remember how we fixed that problem for the bottom wall?

Now we need to fix it for all the walls, since our ball can bounce on whatever wall, depending on how we're holding the iPad.

One more thing (again)… fix the walls.

This is going to look a bit strange. I'm warning you now.

Just copy what I did here and try to think about why it works.

We used the **MIN()** function to keep the ball from going over the **maxPosY**.

We're not going to use the **MAX()** function to keep the ball from going *under* the minimum **y** position.

Note that the minimum y value is 0.

Same goes for the **x** values. We'll do the exact same thing here.

One more small thing. Make sure to change this line here, since we don't don't need to add the velocity any more.

We're already adding it in the **MIN-MAX** line.

```
58
59      // Add gravity to the velocity
60      self.velocity = vel;
61      [self updateVelocityWithAcceleration];
62
63      CGPoint pos = CGPointMake(self.ballXConstraint.constant,
64                                self.ballYConstraint.constant);
65
66      // Constrain the X position of the ball:
67      CGFloat maxPosX = CGRectGetWidth(self.view.bounds) - CGRectGetWidth(self.ballView.bounds);
68      pos.x = MIN(maxPosX, MAX(0, pos.x + self.velocity.x));
69
70      // Update the X position of the ball:
71      self.ballXConstraint.constant = pos.x;
72
73      // Constrain the Y position of the ball:
74      CGFloat maxPosY = CGRectGetHeight(self.view.bounds) - CGRectGetHeight(self.ballView.bounds);
75      pos.y = MIN(maxPosY, MAX(0, pos.y + self.velocity.y));
76
77      // Update the Y position of the ball:
78      self.ballYConstraint.constant = pos.y;
79  }
80
```

If you get lost...

Also, if at any time you get confused about what to type where, you can check out the implementation file here:

https://github.com/AgencyAgency/iPadDemos/blob/master/AccelerometerBounce/AccelerometerBounce/AAViewController.m

The entire project can be downloaded via the **Download Zip** button.

Too much work.

I know, it’s crazy bunch of work to do.

But, hopefully you can see that we now have an interesting platform on which to build things that don’t suck.

Why all this work?

Because, we’re building our own simulated 2D physics framework. We’re building our own game engine. It’s a big deal.

Also, now that you can do this. You can use someone elses framework and know what’s going on under the hood.

And, you are all now officially awesome.