XcodeによるiPhone/Mac アプリの制作 (3)

Core Graphics Programming & Cocoa Touch

平面上の球の壁への衝突をシミュレーション
create Custom View-based Application

株式会社あいはら 山田 泰司 <taiji@aihara.co.jp>

平面上の球の壁への衝突をシミュレーション: WallBall の設計

- 平面上の球を転がすシミュレータ
  — 球は壁に衝突して、跳ね返る

こういったものを作っていく ☞
ジェスチャその他を試みながら行う

1. 壁や球は Core Graphics で描画
   → UIView を継承したカスタムViewの定義

2. タイマーで繰り返し、球の状態を更新して描画

3. 横向きのデバイスへの対応を試みる

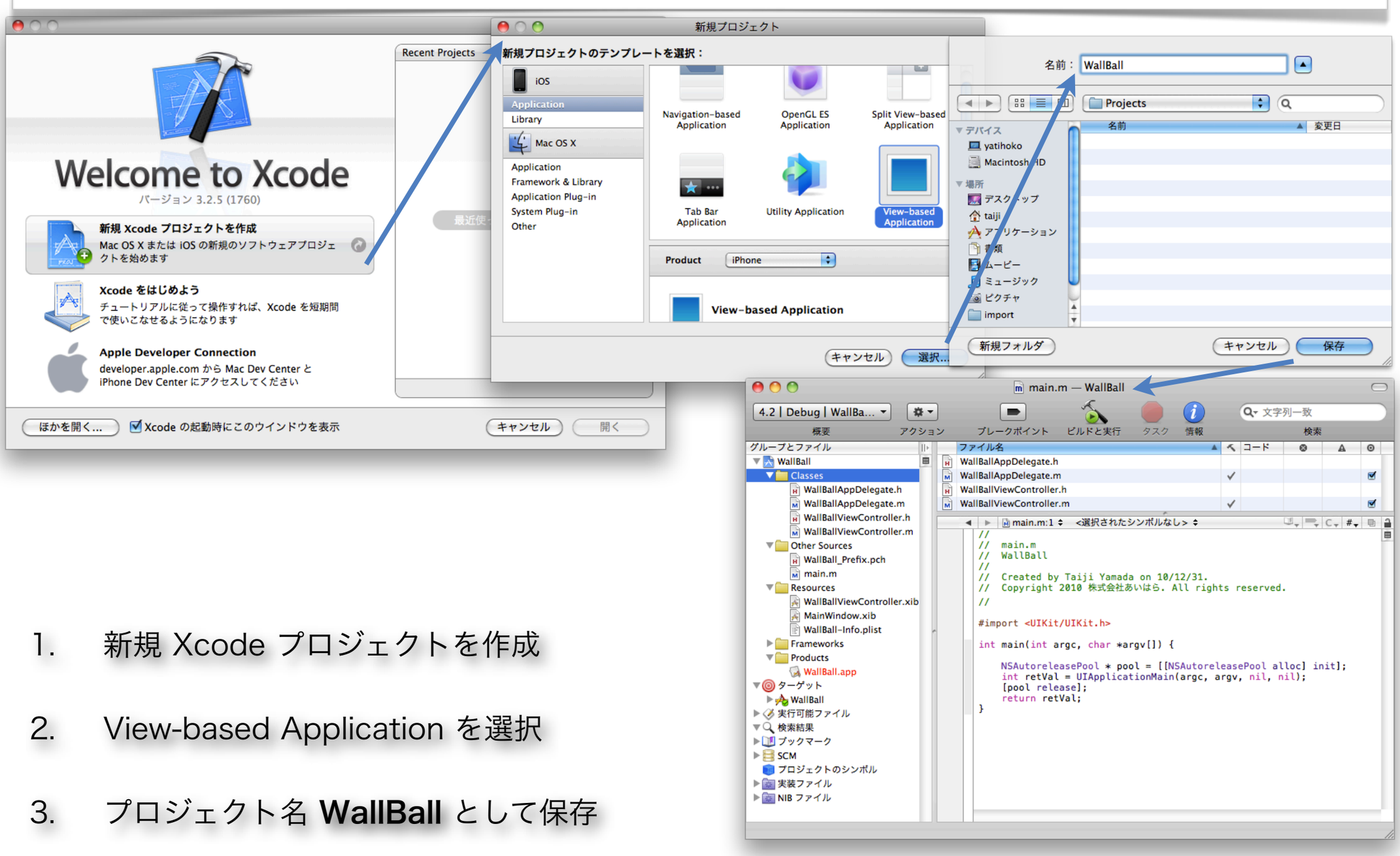
4. ピンチジェスチャで拡大縮小して描画

5. ドラッグジェスチャで、球の速度を変更

6. 加速度センサで、球の速度を変更

WallBall の作成(1)

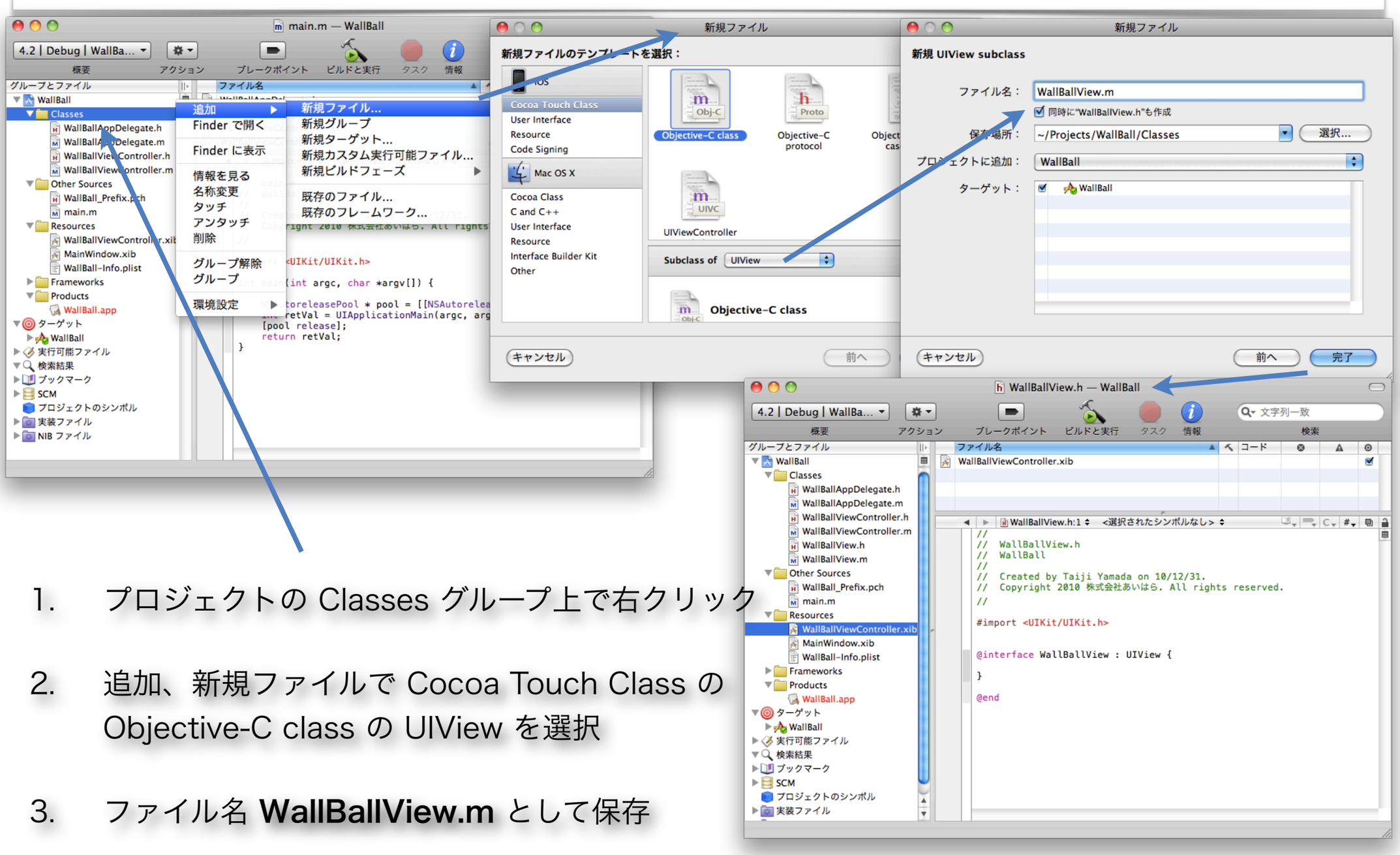
create View-based Application



1. 新規 Xcode プロジェクトを作成
2. View-based Application を選択
3. プロジェクト名 **WallBall** として保存

WallBall の作成(2)：カスタムView

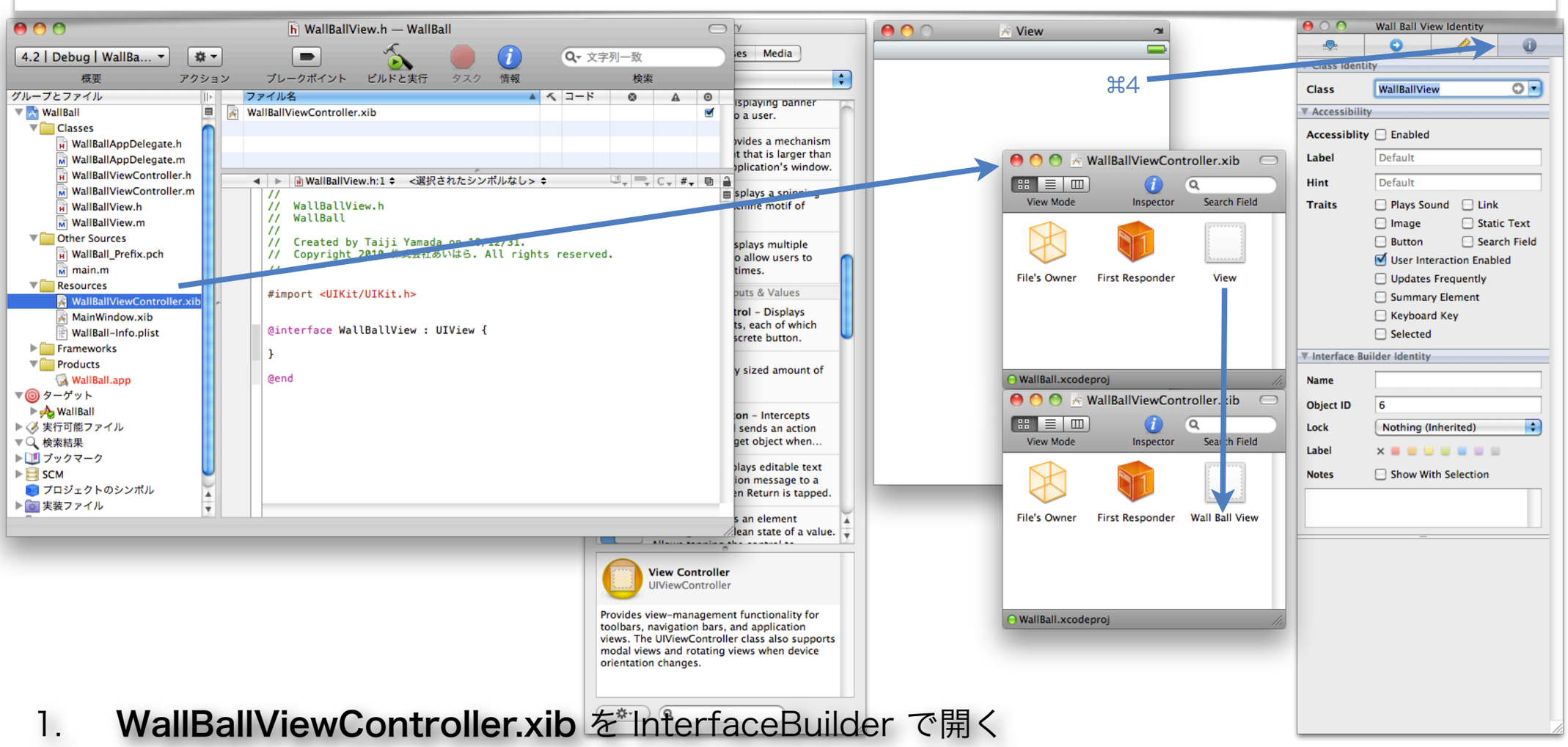
UIView のサブクラス **WallBallView** を追加



1. プロジェクトの Classes グループ上で右クリック
2. 追加、新規ファイルで Cocoa Touch Class の Objective-C class の UIView を選択
3. ファイル名 **WallBallView.m** として保存

WallBall の作成(3)：カスタムView

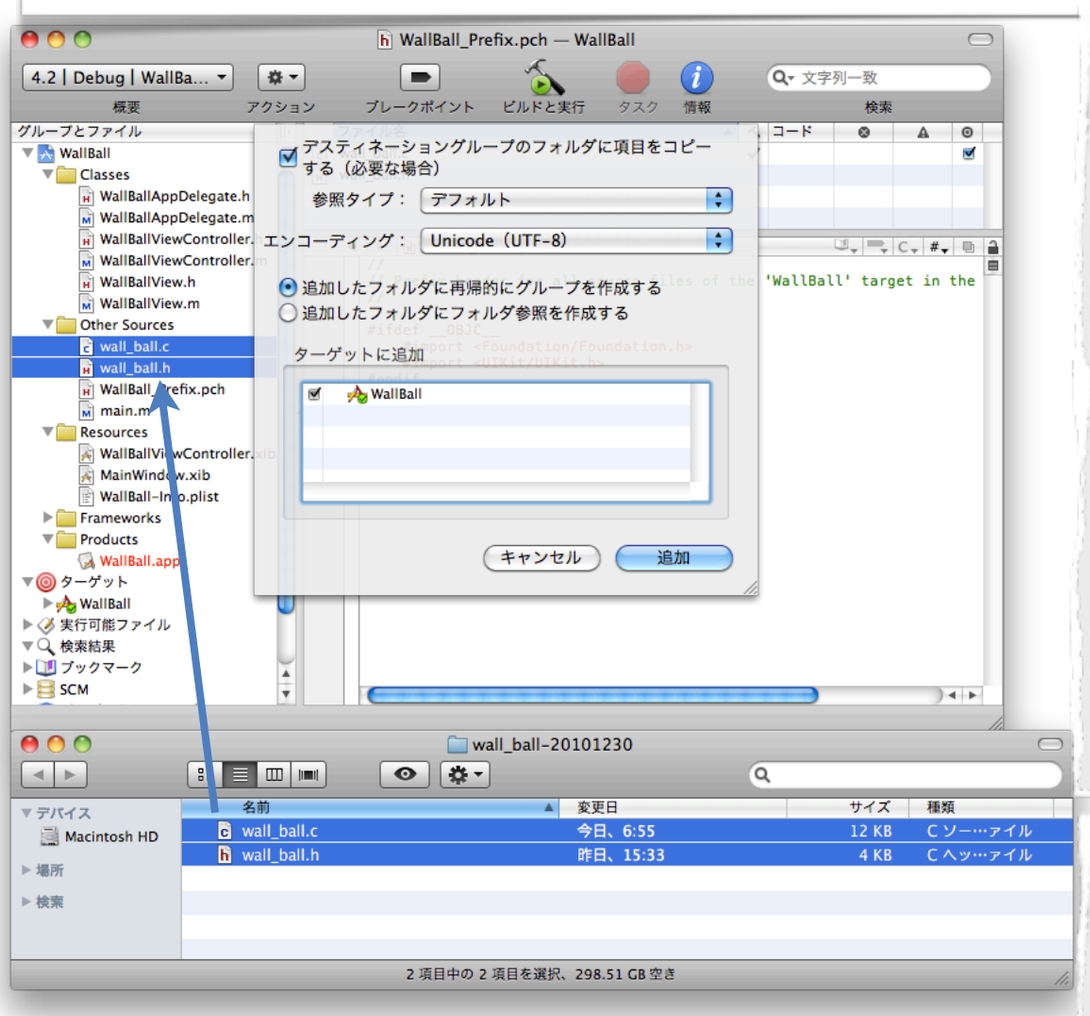
View のクラスを **WallBallView** に変更



1. **WallBallViewController.xib** を InterfaceBuilder で開く
2. View の背景を Attributes Inspector(⌘1) で白にしておく
3. View の Class Identity Inspector(⌘4) を UIView から **WallBallView** に変更し、⌘Sで保存

WallBall の作成(4)

Other Sources に **wall_ball.[hc]** を追加



1. **wall_ball.h** と **wall_ball.c** を Other Source グループに追加 — 球と壁の形状が記されている

```c
// wall_ball.h の主要箇所
enum shape_t { // 形状の種類
  shape_line = 1,
  shape_ellipse,
  shape_arc,
  shape_rarc,
};
      :
typedef struct { // 壁の定義：形状の種類とパラメータ
  int type, dir;
  union {
    struct { double xs, ys, xe, ye; } line;
    struct { double x, y, xr, yr; } ellipse;
    struct { double x, y, xr, yr, as, ae; } arc;
  } shape;
} wall_t;
      :
typedef struct { // 球と壁の定義
  double r; // 球の半径
  int m; // 壁の数
  wall_t *W; // 壁の配列へのポインタ
  double v[2*2], v0[2*2], v1[2*2];
} ball_walls_t;
      :
#define rad(d) (M_PI*(d)/180.0)

extern ball_walls_t ball_walls_set[], *ball_walls;
      :
```

```c
// wall_ball.c の主要箇所
wall_t round_rect_walls[] = { // 丸みを帯びた矩形の壁
  { .type = 1, .dir =  1, .shape.line = {  1.0,-0.7,  1.0, 0.7 } },
  { .type = 3, .dir =  1, .shape.arc = {  0.7, 0.7, 0.3,0.3,   0.0, 90.0 } },
  { .type = 1, .dir =  1, .shape.line = {  0.7, 1.0, -0.7, 1.0 } },
  { .type = 3, .dir =  1, .shape.arc = { -0.7, 0.7, 0.3,0.3,  90.0,180.0 } },
  { .type = 1, .dir =  1, .shape.line = { -1.0, 0.7, -1.0,-0.7 } },
  { .type = 3, .dir =  1, .shape.arc = { -0.7,-0.7, 0.3,0.3, 180.0,270.0 } },
  { .type = 1, .dir =  1, .shape.line = { -0.7,-1.0,  0.7,-1.0 } },
  { .type = 3, .dir =  1, .shape.arc = {  0.7,-0.7, 0.3,0.3, 270.0,360.0 } },
};
ball_walls_t ball_walls_set[] = { // さまざまな球と壁
      :
  {
    .r = 0.05,
    .m = sizeof(round_rect_walls)/sizeof(round_rect_walls[0]), .W = round_rect_walls,
    .v = { 0.5, 0.5, -0.1/2, 0.05/2 },
  },
      :
}, *ball_walls = &ball_walls_set[2]; // 球と丸みを帯びた矩形の壁へのポインタ
```

WallBall の作成(5)：Core Graphics

WallBallView.m に描画コードを記述

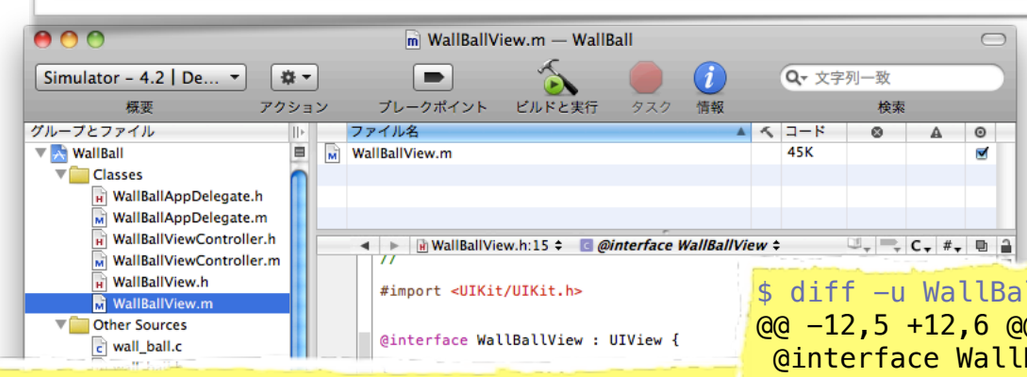
1. **WallBallView.m** に
   `#include “wall_ball.h”` を追加

2. **WallBallView.m** で
   コメントアウトされている
   `drawRect:`を右のように編集

```
- (void)drawRect:(CGRect)rect {
  CGContextRef gc = UIGraphicsGetCurrentContext(); // グラフィックスコンテキスト

  CGContextTranslateCTM(gc, rect.size.width/2, rect.size.height/2); // 中央を原点に

  CGContextScaleCTM(gc, 1, -1); // 上下反転

  CGContextSetRGBStrokeColor(gc, 0, 0, 0, 1); // RGBA
  CGContextSetLineWidth(gc, 1);
  {
    int i;
    CGPoint p, q;
    CGRect r;
    CGSize s;
    CGFloat as, ae;

    double scale = 100;
    for (i=0; i<ball_walls->m; i++) { // 壁の描画

      switch (ball_walls->W[i].type) {
        case shape_line: // 線分を描く

          p = CGPointMake(ball_walls->W[i].shape.line.xs*scale, ball_walls->W[i].shape.line.ys*scale);
          q = CGPointMake(ball_walls->W[i].shape.line.xe*scale, ball_walls->W[i].shape.line.ye*scale);
          CGContextMoveToPoint(gc, p.x, p.y);
          CGContextAddLineToPoint(gc, q.x, q.y);
          CGContextDrawPath(gc, kCGPathStroke);
          break;
        case shape_ellipse: // 楕円を描く

          p = CGPointMake(ball_walls->W[i].shape.ellipse.x*scale, ball_walls->W[i].shape.ellipse.y*scale);
          s = CGSizeMake(ball_walls->W[i].shape.ellipse.xr*scale*2, ball_walls->W[i].shape.ellipse.yr*scale*2);
          r = CGRectMake(p.x - s.width/2, p.y - s.height/2, s.width, s.height);
          CGContextStrokeEllipseInRect(gc, r);
          CGContextDrawPath(gc, kCGPathStroke);
          break;
        case shape_arc:
        case shape_rarc: // 弧を描く

          p = CGPointMake(ball_walls->W[i].shape.arc.x*scale, ball_walls->W[i].shape.arc.y*scale);
          s = CGSizeMake(ball_walls->W[i].shape.arc.xr*scale*2, ball_walls->W[i].shape.arc.yr*scale*2);
          as = rad(ball_walls->W[i].shape.arc.as);
          ae = rad(ball_walls->W[i].shape.arc.ae);
          CGContextAddArc(gc, p.x, p.y, s.width/2, as, ae, (ball_walls->W[i].type == shape_arc) ? 0 : 1);
          CGContextDrawPath(gc, kCGPathStroke);
          break;
      }
    }
    // 球の描画

    p = CGPointMake(ball_walls->v0[0]*scale, ball_walls->v0[1]*scale);
    q = CGPointMake(p.x-ball_walls->v0[2]*scale, p.y-ball_walls->v0[3]*scale);
    s = CGSizeMake(ball_walls->r*scale*2, ball_walls->r*scale*2);
    CGContextMoveToPoint(gc, p.x, p.y);
    CGContextAddLineToPoint(gc, q.x, q.y);
    CGContextDrawPath(gc, kCGPathStroke);
    CGContextAddArc(gc, p.x, p.y, s.width/2, rad(0), rad(360), 0);
    CGContextDrawPath(gc, kCGPathStroke);
  }
  CGContextFlush(gc);
}
```

WallBall の作成(6)：タイマー

WallBallView.[hm] を編集：状態の初期化と更新を定義




```
$ diff -u WallBallView.m~ WallBallView.m
@@ -11,6 +11,21 @@

 @implementation WallBallView

+- (void)resetState
+{
+  memcpy(ball_walls->v0, ball_walls->v, sizeof(ball_walls->v0));
+  ball_walls_map(ball_walls);
+  [self setNeedsDisplay];
+}
+- (void)updateState:(NSTimer *)timer
+{
+  ball_walls_map(ball_walls);
+  ball_walls->v0[0] += (ball_walls->v1[0] - ball_walls->v0[0])/(long)(10*(ball_walls->v1[0] - ball_walls->v0[0])/ball_walls->v0[2]);
+  ball_walls->v0[1] += (ball_walls->v1[1] - ball_walls->v0[1])/(long)(10*(ball_walls->v1[1] - ball_walls->v0[1])/ball_walls->v0[3]);
+  [self setNeedsDisplay];
+  if (ball_walls->v0[0] == ball_walls->v1[0] && ball_walls->v0[1] == ball_walls->v1[1])
+    memcpy(ball_walls->v0, ball_walls->v1, sizeof(ball_walls->v0));
+}
```

```
$ diff -u WallBallView.h~ WallBallView.h
@@ -12,5 +12,6 @@
 @interface WallBallView : UIView {

 }
+- (void)resetState;

 @end
```

```
// wall_ball.h の主要箇所
      :
typedef struct { // 球と壁の定義
  double r; // 球の半径
  int m; // 壁の数
  wall_t *W; // 壁の配列へのポインタ
  double v[2*2], v0[2*2], v1[2*2]; // 球の状態
  // 初期値，衝突前の状態，衝突後の状態
} ball_walls_t;

      :
#define sqr(a) ((a)*(a))
#define deg(r) (180.0*(r)/M_PI)
      :
void ball_walls_map(ball_walls_t *p); // 球の状態を算出
      :
```

```
// wall_ball.c の主要箇所
      :
void ball_walls_map(ball_walls_t *p) // 球の状態を算出
{
  // p->v0[4]: 球の衝突前の状態(x, y座標, x, y速度)から
  // p->v1[4]: 球の衝突後の状態(x, y座標, x, y速度)を算出
      :
}
```

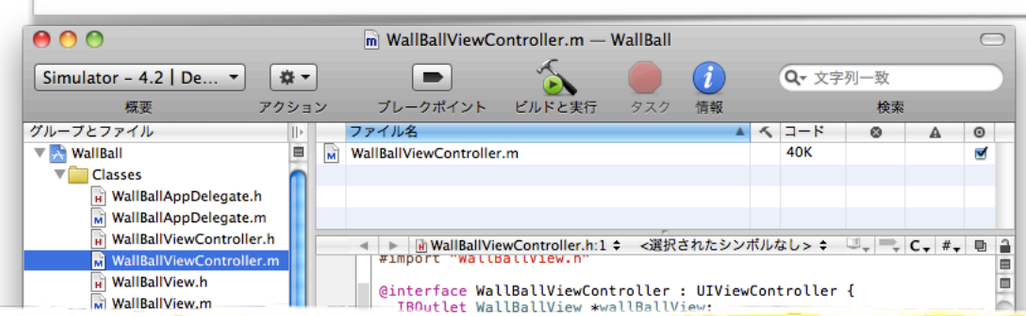
＊編集内容は diff -u (Unified diff.形式)「行頭"-"が削除行、"+"が追加行」で表記

1. **WallBallView.h** を編集し、**resetState:**メソッドを宣言、
   WallBallView.m を編集し、**resetState:**メソッドを定義

2. **WallBallView.m** を編集し、タイマーから呼ばれる **updateState:**メソッドを定義

WallBall の作成(7)：タイマー

WallBallViewController.[hm] を編集：状態の初期化と更新




```
$ diff -u WallBallViewController.h~ WallBallViewController.h
@@ -7,10 +7,14 @@
 //

 #import <UIKit/UIKit.h>
+#import "WallBallView.h"

 @interface WallBallViewController : UIViewController {
-
+  IBOutlet WallBallView *wallBallView;
+  NSTimer *timer;
 }
+- (IBAction)buttonResetPressed:(UIButton *)sender;
+- (IBAction)switchActionValueChanged:(UISwitch *)sender;

 @end
```

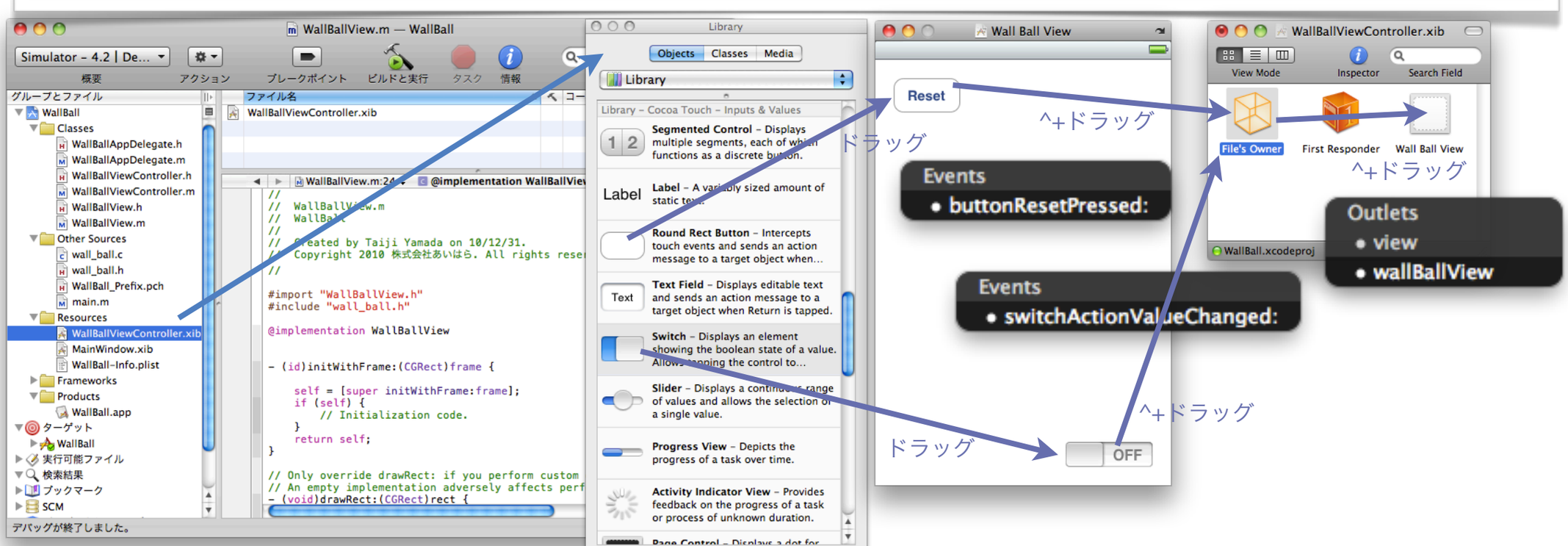
```
$ diff -u WallBallViewController.m~ WallBallViewController.m
@@ -10,7 +10,19 @@

 @implementation WallBallViewController

-
+- (IBAction)buttonResetPressed:(UIButton *)sender
+{
+  [wallBallView resetState];
+}
+- (IBAction)switchActionValueChanged:(UISwitch *)sender
+{
+  if (sender.on)
+    timer = [NSTimer scheduledTimerWithTimeInterval:.01 target:wallBallView selector:@selector(updateState:) userInfo:nil repeats:YES];
+  else {
+    [timer invalidate];
+    timer = nil;
+  }
+}

@@ -30,12 +42,11 @@

-/*
 // Implement viewDidLoad to do additional setup after loading the view, typically from a nib.
 - (void)viewDidLoad {
     [super viewDidLoad];
+  [wallBallView resetState];
 }
-*/
```

1. **WallBallViewController.h** を編集し、上記のように IBOutlet、IBAction と **timer** を宣言

2. **WallBallViewController.m** を編集し、上記のように Action を定義。**timer** を発動。

WallBall の作成(8)：タイマー

WallBallView を Outlet、ボタンとスイッチを Action



1. **WallBallViewController.xib** を InterfaceBuilder で開く

2. UIButtonからFile's Owner(**WallBallViewController**)へ^+ドラッグで**buttonResetPressed:**メソッドにAction

3. UISwitchからFile's Ownerへ^+ドラッグで**switchActionValueChanged:**メソッドにAction

4. File's OwnerからWall Ball Viewへ^+ドラッグで**wallBallView**にOutlet

WallBall の作成(9)：デバイスの向き

WallBallViewController.m を編集し、デバイスの向きに追従してみる

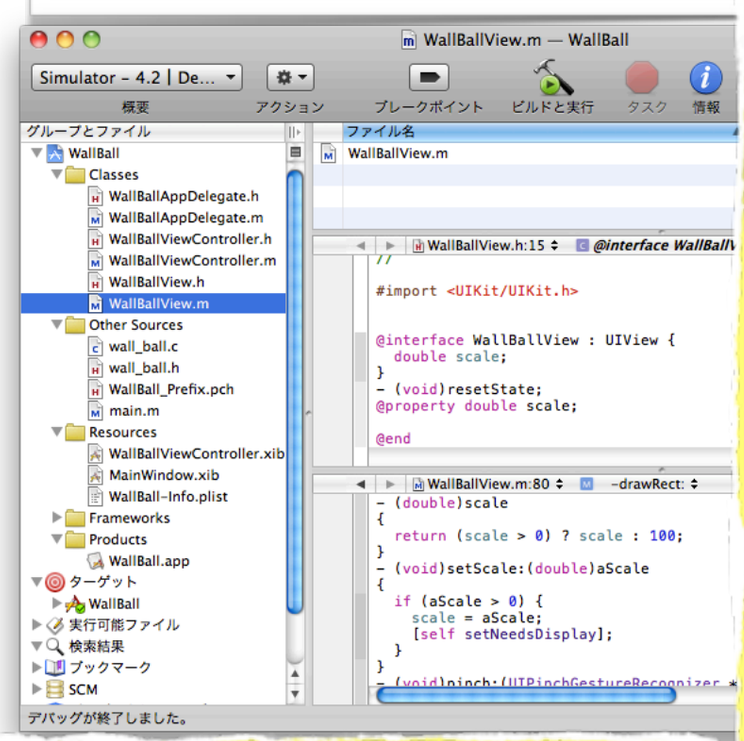
1. **WallBallViewController.m** を編集して、上記メソッドを追加するとデバイスの向きに応じてUIが自動的に再配置される

2. ⌘Sで保存、⌘Rでシミュレータで実行、デバイスの向きは⌘←で変更

デバイスの向きを変えるとUIが自動的に追従するようになった 👌

WallBall の作成(10)：ピンチジェスチャ

WallBallView.[hm] を編集：pinch:メソッドを追加




```
$ diff -u WallBallView.h~ WallBallView.h
@@ -10,8 +10,9 @@

 @interface WallBallView : UIView {
-
+  double scale;
 }
 - (void)resetState;
+@property double scale;

 @end
```

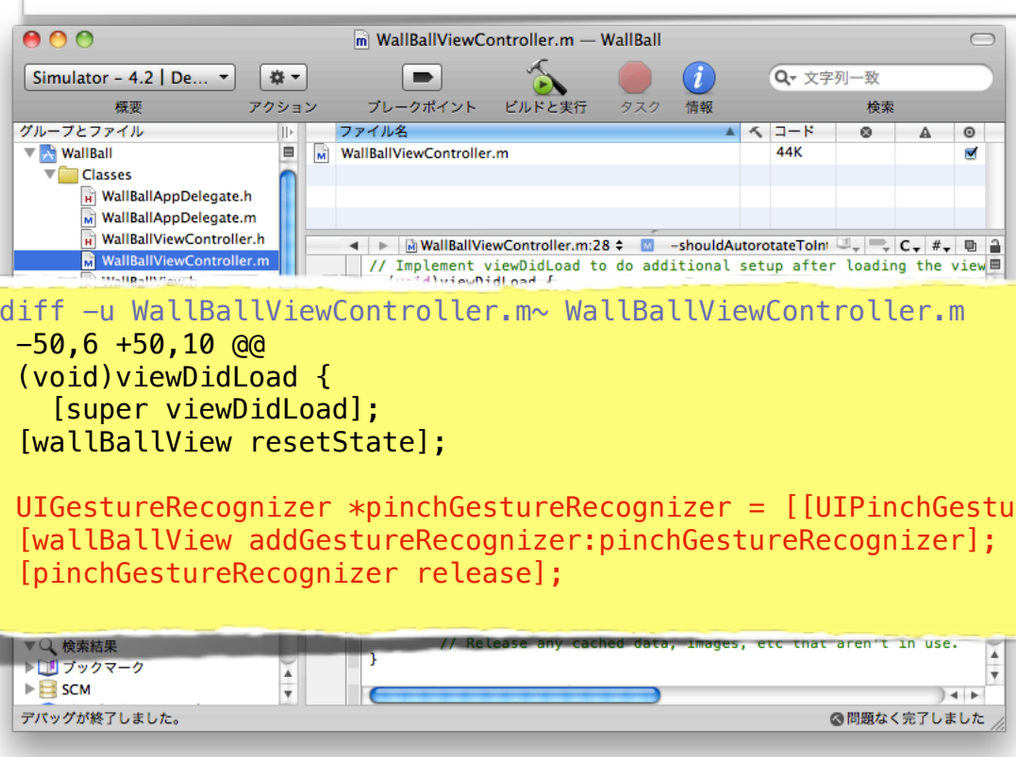
```
$ diff -u WallBallView.m~ WallBallView.m
@@ -26,6 +26,25 @@
    if (ball_walls->v0[0] == ball_walls->v1[0] && ball_walls->v0[1] == ball_walls->v1[1])
      memcpy(ball_walls->v0, ball_walls->v1, sizeof(ball_walls->v0));
 }
+- (double)scale
+{
+  return (scale > 0) ? scale : 100;
+}
+- (void)setScale:(double)aScale
+{
+  if (aScale > 0) {
+    scale = aScale;
+    [self setNeedsDisplay];
+  }
+}
+- (void)pinch:(UIPinchGestureRecognizer *)gesture
+{
+  if ((gesture.state == UIGestureRecognizerStateChanged) ||
+      (gesture.state == UIGestureRecognizerStateEnded)) {
+    self.scale *= gesture.scale;
+    gesture.scale = 1;
+  }
+}

@@ -54,7 +73,7 @@
     CGSize s;
     CGFloat as, ae;

-    double scale = 100;
+    scale = self.scale;
     for (i=0; i<ball_walls->m; i++) {
       switch (ball_walls->W[i].type) {
         case shape_line:
```

1. **WallBallView.h** を編集し、double **scale** をインスタンス変数にし、プロパティとしてアクセス可能にする(アクセサ)。
   WallBallView.m を編集し、そのセッタメソッド(**setScale:**)とゲッタメソッド(**scale**)を実装する。

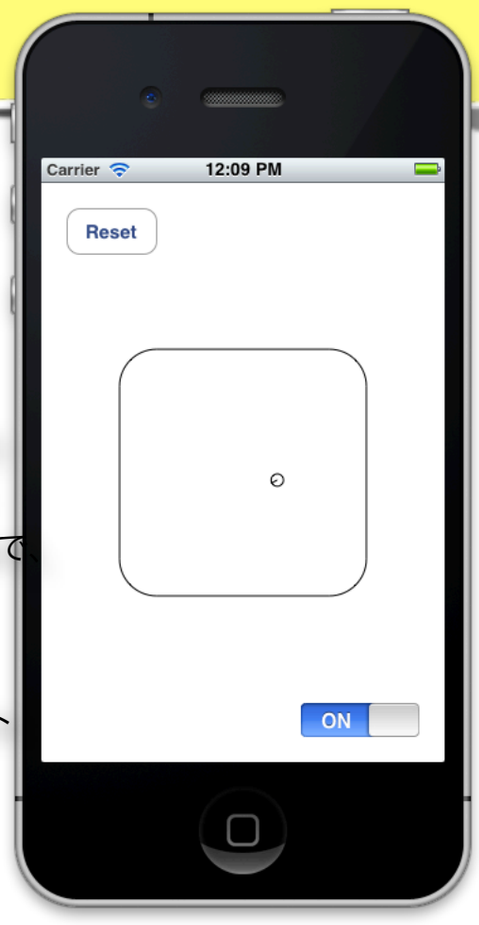
2. **pinch:**メソッドを実装し、ピンチジェスチャリコグナイザの scale で self.**scale**(ドット記法：セッタ)を乗ずる。

WallBall の作成(8)：ピンチジェスチャ

WallBallViewController.m を編集：pinch:メソッドを指定する




```
$ diff -u WallBallViewController.m~ WallBallViewController.m
@@ -50,6 +50,10 @@
 - (void)viewDidLoad {
     [super viewDidLoad];
    [wallBallView resetState];
+
+   UIGestureRecognizer *pinchGestureRecognizer = [[UIPinchGestureRecognizer alloc] initWithTarget:wallBallView action:@selector(pinch:)];
+   [wallBallView addGestureRecognizer:pinchGestureRecognizer];
+   [pinchGestureRecognizer release];
 }
```



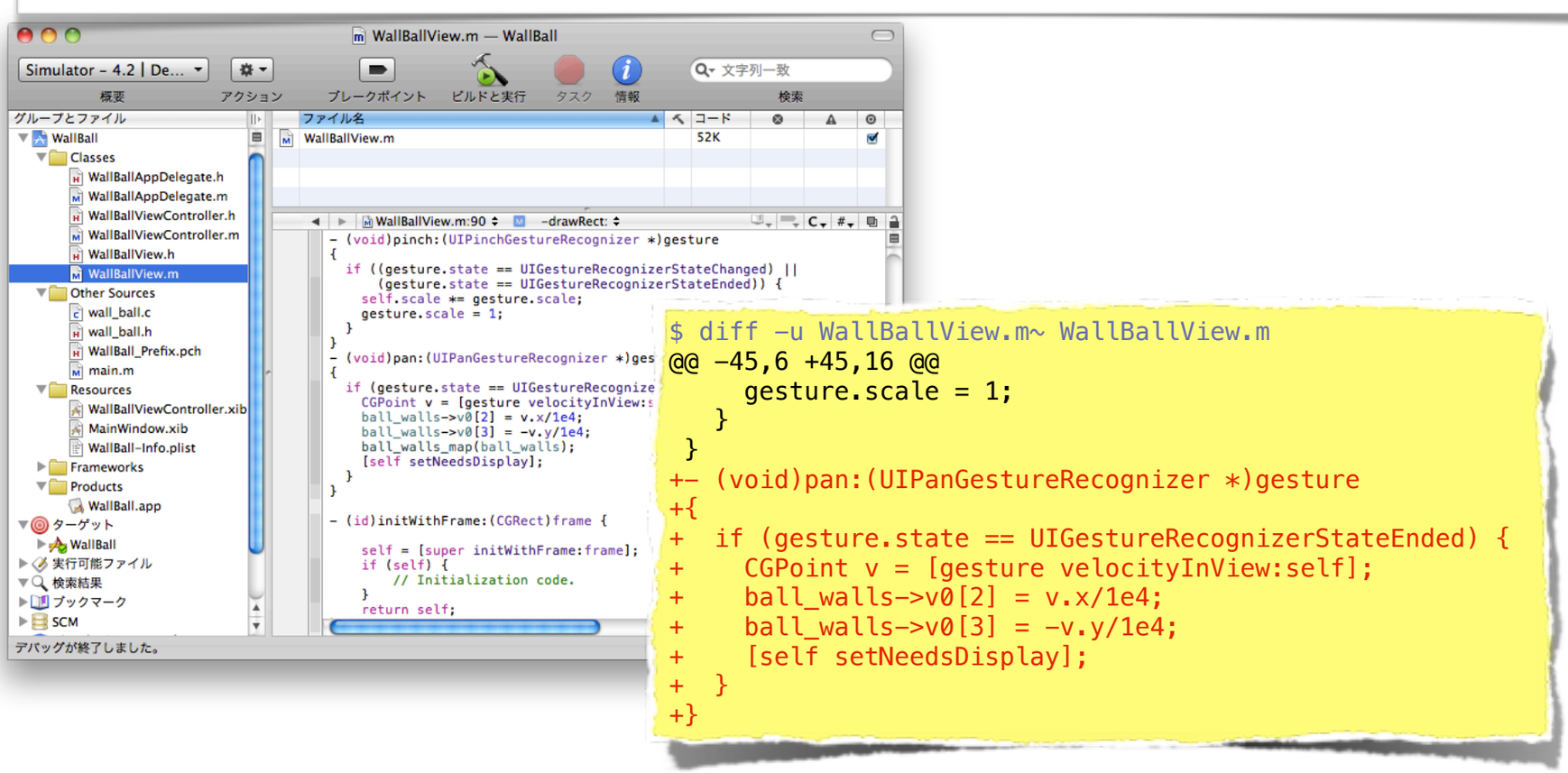
1. **WallBallViewController.m** を編集し、
   UIPinchGestureRecognizer の initWithTarget:action: メソッドで
   ターゲットを **wallBallView** インスタンス、
   アクションをセレクタディレクティブ@selector(**pinch:**)（**WallBallView** の **pinch:** メソッド）
   を指定。さらに、**wallBallView** インスタンスに **pinchGestureRecognizer** インスタンスを
   addGestureRecognizer:メソッドで追加。**pinchGestureRecognizer** は alloc されているので、
   追加が終わったら参照カウンタを release で下げておくことを忘れないこと!

2. ⌘Sで保存、⌘Rでシミュレータで実行。ピンチジェスチャは⌥(option)+ドラッグでエミュレート

ピンジェスチャでスケールを変えることが可能になった ☞

WallBall の作成(10)：ドラッグジェスチャ

WallBallView.m を編集：pan:メソッドを追加

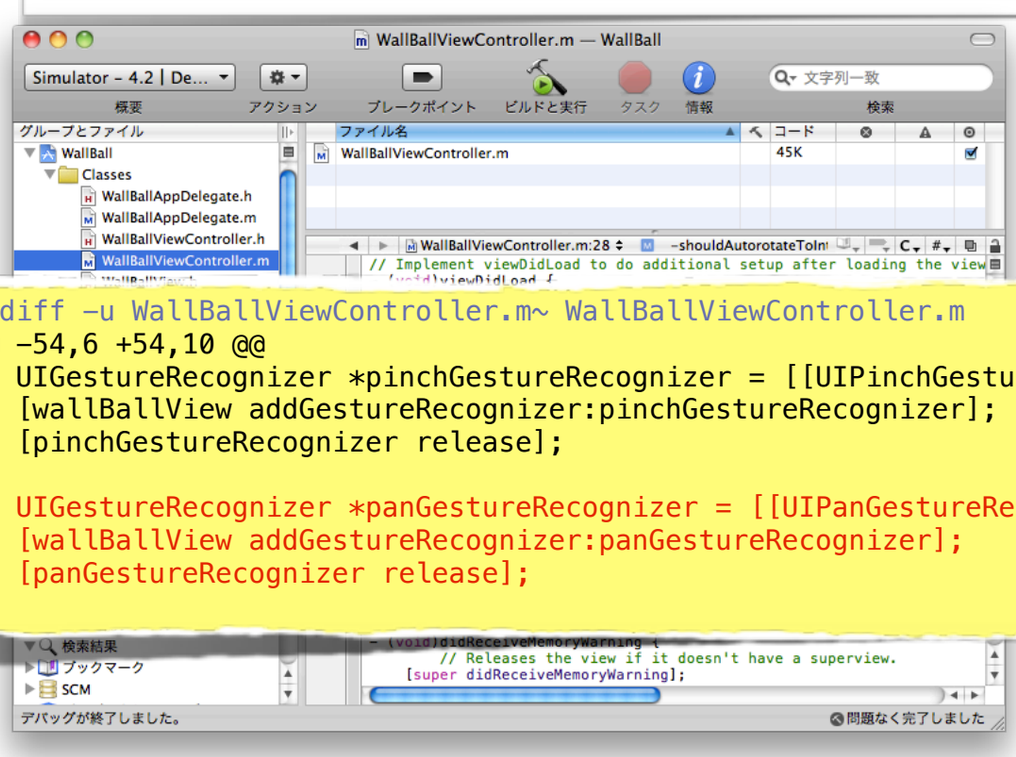



```
$ diff -u WallBallView.m~ WallBallView.m
@@ -45,6 +45,16 @@
     gesture.scale = 1;
   }
 }
+- (void)pan:(UIPanGestureRecognizer *)gesture
+{
+  if (gesture.state == UIGestureRecognizerStateEnded) {
+    CGPoint v = [gesture velocityInView:self];
+    ball_walls->v0[2] = v.x/1e4;
+    ball_walls->v0[3] = -v.y/1e4;
+    [self setNeedsDisplay];
+  }
+}
```

1. **WallBallView.m** を編集し、**pan:**メソッドを実装。ここでは、ドラッグの速度をvelocityInView:メソッドで取得して、それを球の衝突前の状態の速度に代入している。

WallBall の作成(8)：ドラッグジェスチャ

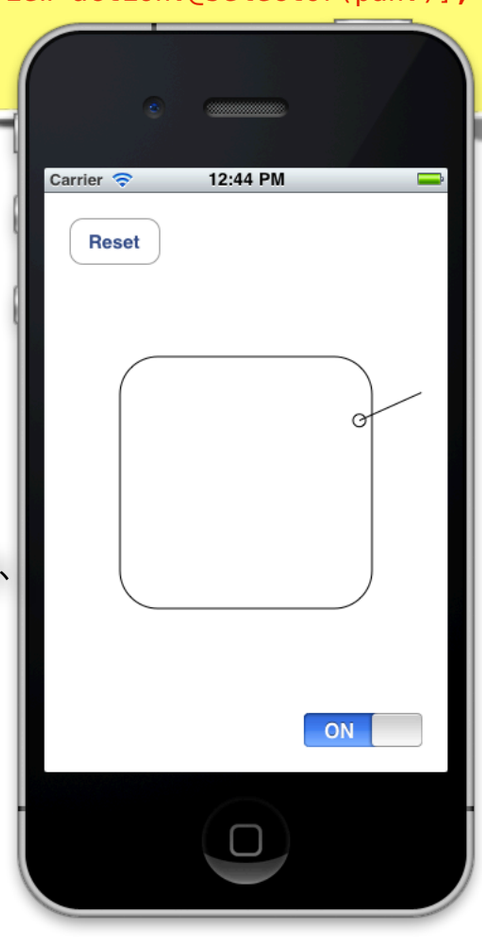
WallBallViewController.m を編集：pan:メソッドを指定




```
$ diff -u WallBallViewController.m~ WallBallViewController.m
@@ -54,6 +54,10 @@
    UIGestureRecognizer *pinchGestureRecognizer = [[UIPinchGestureRecognizer alloc] initWithTarget:wallBallView action:@selector(pinch:)];
    [wallBallView addGestureRecognizer:pinchGestureRecognizer];
    [pinchGestureRecognizer release];
+
+   UIGestureRecognizer *panGestureRecognizer = [[UIPanGestureRecognizer alloc] initWithTarget:wallBallView action:@selector(pan:)];
+   [wallBallView addGestureRecognizer:panGestureRecognizer];
+   [panGestureRecognizer release];
 }
```

1. **WallBallViewController.m** を編集し、
   UIPanGestureRecognizer の initWithTarget:action: メソッドで
   ターゲットを **wallBallView** インスタンス、
   アクションをセレクタディレクティブ@selector(**pan:**)（**WallBallView** の **pan:** メソッド）
   を指定。さらに、**wallBallView** インスタンスに **panGestureRecognizer** インスタンスを
   addGestureRecognizer:メソッドで追加。**panGestureRecognizer** も alloc されているので、
   追加が終わったら参照カウンタを release で下げておくことを忘れないこと!

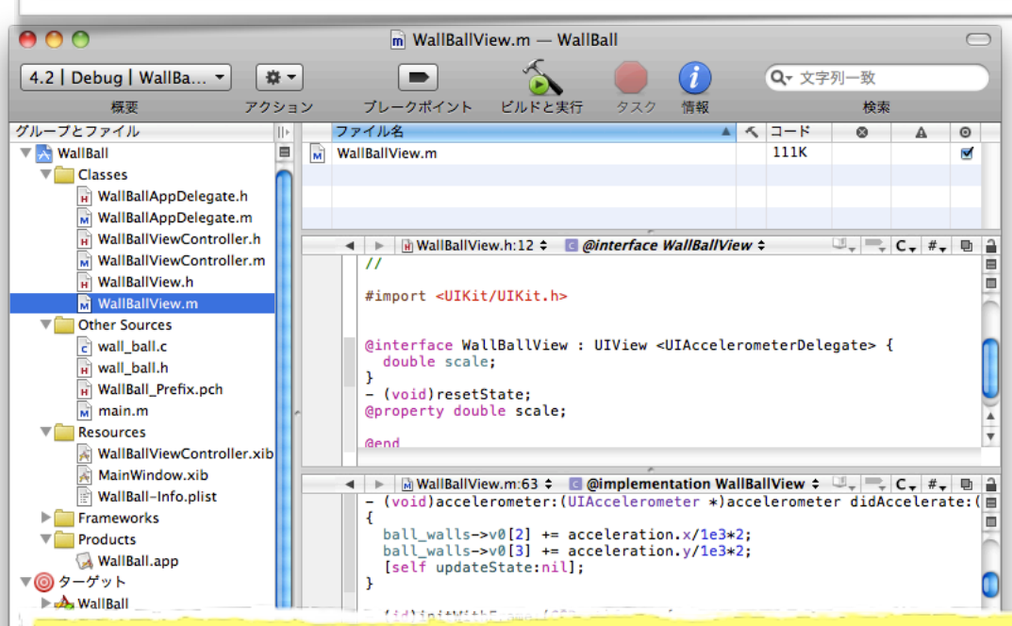
2. ⌘Sで保存、⌘Rでシミュレータで実行。パンジェスチャはマウスのドラックでエミュレート



ドラッグで球の速度を変えることが可能になった ☞

WallBall の作成(10)：加速度センサ

WallBallView.m を編集：accelerometer:didAccelerate:メソッドを追加




```
$ diff -u WallBallView.h~ WallBallView.h
@@ -9,7 +9,7 @@
 #import <UIKit/UIKit.h>


-@interface WallBallView : UIView {
+@interface WallBallView : UIView <UIAccelerometerDelegate> {
    double scale;
 }
 - (void)resetState;
```

```
$ diff -u WallBallView.m~ WallBallView.m
@@ -55,6 +55,12 @@
      [self setNeedsDisplay];
    }
 }
+- (void)accelerometer:(UIAccelerometer *)accelerometer didAccelerate:(UIAcceleration *)acceleration
+{
+  ball_walls->v0[2] += acceleration.x/1e3*2;
+  ball_walls->v0[3] += acceleration.y/1e3*2;
+  [self updateState:nil];
+}
```

1. **WallBallView.h** を編集し、クラスを UIAccelerometerDelegate プロトコルへの準拠を宣言

2. **WallBallView.m** を編集し、acceletometer:didAccelerate:メソッドを実装。ここでは、加速度を引数 acceleration から取得し、それを球の衝突前の状態の速度に加算している。

WallBall の作成(11)：加速度センサ

WallBallViewController.[hm] を編集：accelerometer.delegateを指定

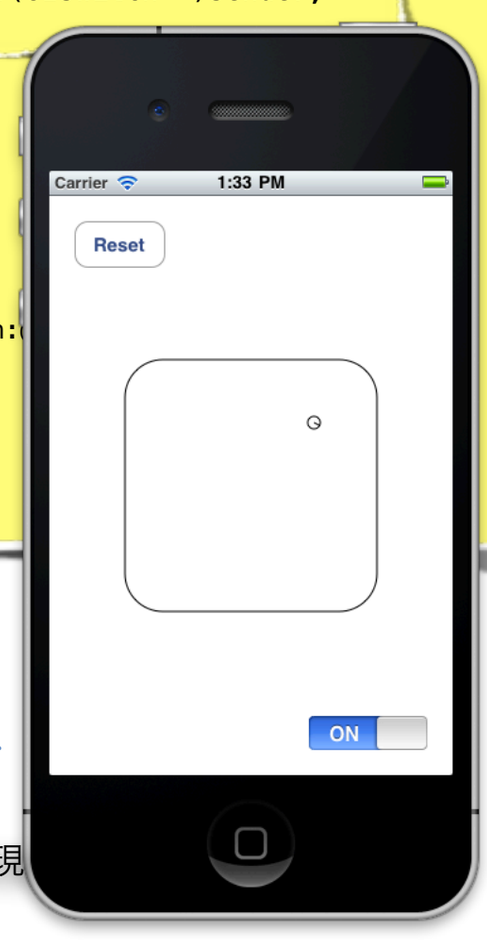
```
$ diff -u WallBallViewController.m~ WallBallViewController.m
@@ -16,16 +16,27 @@
 }
 - (IBAction)switchActionValueChanged:(UISwitch *)sender
 {
+#if 0
    if (sender.on)
      timer = [NSTimer scheduledTimerWithTimeInterval:.01 target:wallBallView selector:@selector(updateState:) userInfo:nil repeats:YES];
    else {
      [timer invalidate];
      timer = nil;
    }
+#else
+   if (!accelerometer.delegate)
+     accelerometer.delegate = wallBallView;
+   else
+     accelerometer.delegate = nil;
+#endif
 }
 - (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientatio
 {
+#if 0
    return YES;
+#else
+   return NO;
+#endif
 }

 /*
@@ -58,6 +69,10 @@
    UIGestureRecognizer *panGestureRecognizer = [[UIPanGestureRecognizer alloc] initWithTarget:wallBallView action:(
    [wallBallView addGestureRecognizer:panGestureRecognizer];
    [panGestureRecognizer release];
+
+   accelerometer = [UIAccelerometer sharedAccelerometer];
+   accelerometer.updateInterval = .01;
+   accelerometer.delegate = nil;
 }
```

```
$ diff -u WallBallViewController.h~ WallBallViewController.h
@@ -12,6 +12,7 @@
 @interface WallBallViewController : UIViewController {
    IBOutlet WallBallView *wallBallView;
    NSTimer *timer;
+   UIAccelerometer *accelerometer;
 }
 - (IBAction)buttonResetPressed:(UIButton *)sender;
 - (IBAction)switchActionValueChanged:(UISwitch *)sender;
```

1. WallBallViewController.h を編集し、UIAccelerometer ***accelerometer*** を宣言。
2. WallBallViewController.m を編集し、viewDidLoadメソッド内で加速度センサを取得、時間間隔を 0.01 秒に設定。デリゲートはまずは nil にしておく。

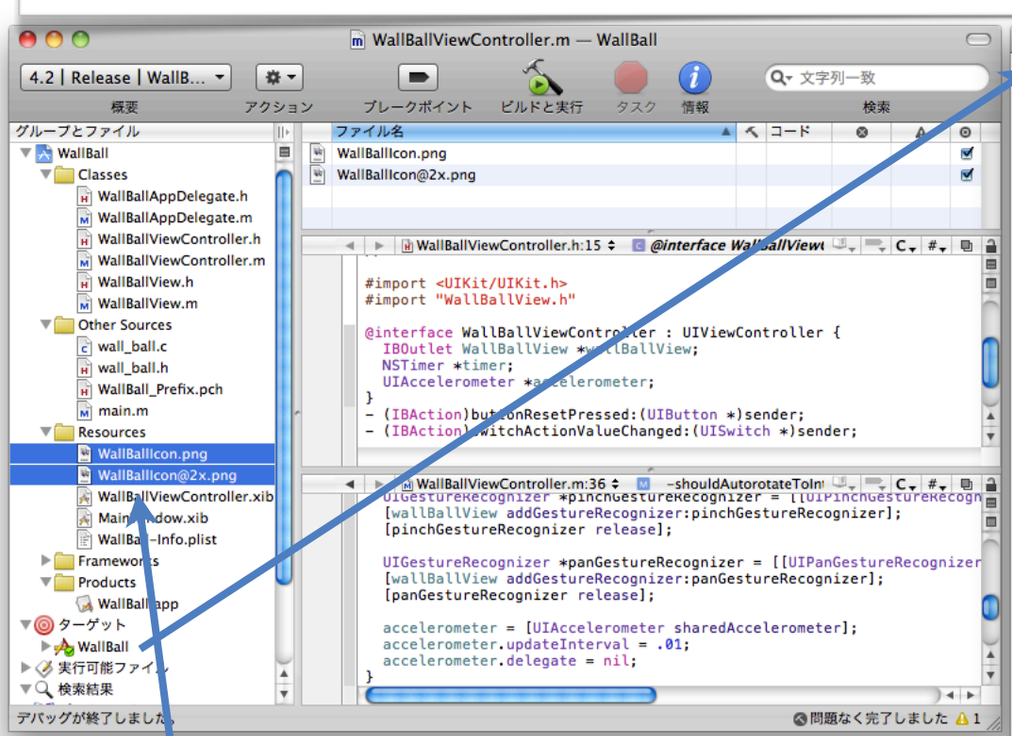
動かない。シミュレータでは加速度センサはエミュレートできないので… ☞

3. スイッチがオンになったときに、タイマーを発動する代わりに、**accelerometer** のデリゲートに wallBallView を指定することで、繰り返し加速度の取得を実現



WallBall の作成(12)：ジェスチャ～加速度センサ

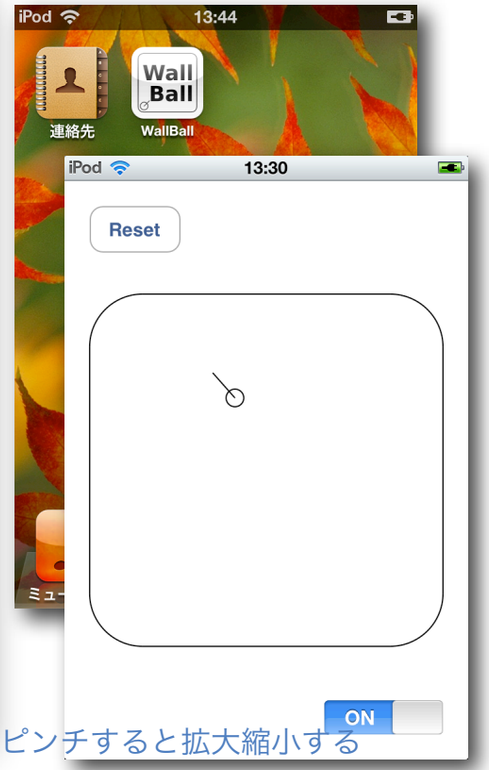
実機で実行してみよう！



サイズ57x57と114x114のPNG形式のアイコンを
WallBallIcon.png, **WallBallIcon@2x.png** として作成し
プロジェクトに追加

アイコンファイルは **WallBallIcon** に

ターゲット WallBall の情報(⌘I)で、
iOS Provisioning Portal で作成かつ
オーガナイザに設定したAppID の識別子に



- ピンチすると拡大縮小する
- 素早くドラッグすると球も素早く転がる
- 傾けると落ちるかのように引き寄せられる

1. デバイスの向きに追従する機能と、加速度センサから球の状態を変更する機能とが、
やってみるとうまく噛み合ないので、前ページで、デバイスの向きに追従する機能をオフにした

2. スイッチのドラッグと **WallBallView** のドラッグの境界がなく、些かシビアである。なので、
WallBallView は元々の UIView 上に貼り付けた独立した View にした方がよかったかも…

プログラミング言語 Objective-C

主な各種ディレクティブ、型、記法(1)

`@interface` *ClassName* : *SuperClass* <*Protocol*> {
  double *value*;
}
+ *classWithValue*:(double)*aValue*;
- *initWithValue*:(double)*aValue*;
- (double)*value*;
- (void)*setValue*:(double)*aValue*;
`@property` double *value*;
`@end`
	クラスのインターフェースの宣言開始のディレクティブ（*Protocol*を採用し*SuperClass*を継承）
	インスタンス変数の宣言
	
	クラスメソッドの宣言（簡易コンストラクタ）
	インスタンスメソッドの宣言（指定イニシャライザ）
	インスタンスメソッドの宣言（ゲッターメソッド）
	インスタンスメソッドの宣言（セッターメソッド）
	アクセサ（インスタンス）メソッドの宣言、上記２行と等価
	宣言または定義終了のディレクティブ

`@implementation` *ClassName*
+ *classWithValue*:(double)*aValue*
{
  return [[[self alloc] *initWithValue*:*aValue*]
  autorelease];
}
- *initWithValue*:(double)*aValue* {
  if ((self = [super init]))
  value = *aValue*;
  return self;
}
- (double)*value*
{
  return *value*;
}
- (void)*setValue*:(double)*aValue*
{
  value = *aValue*;
}
`@synthesize` *value*;
`@end`
	クラスのインターフェースの定義開始のディレクティブ
	クラスメソッドの定義（簡易コンストラクタ）
	
	インスタンスメソッドの定義（指定イニシャライザ）
	
	インスタンスメソッドの定義（ゲッターメソッド）
	
	インスタンスメソッドの定義（セッターメソッド）
	
	アクセサ（インスタンス）メソッドの定義、上記２ブロックと同等
	宣言または定義終了のディレクティブ

プログラミング言語 Objective-C

主な各種ディレクティブ、型、記法(2)

id	オブジェクトへのポインタ型
nil	NULLオブジェクトポインタ、(id)NULL
BOOL	真偽型、char型
NO	偽値、(BOOL)0
YES	真値、(BOOL)1
#import	二度読み防止済み#includeディレクティブ
self	受信側オブジェクトへのポインタ変数
super	継承されている受信側オブジェクトへのポインタ変数
@"文字列"	NSString型オブジェクト定数ディレクティブ
@"文字列*1*" @"文字列*2*" … @"文字列*n*"	NSString型オブジェクト定数ディレクティブ、*n*個のディレクティブで指定された文字列を結合
[*receiver message*]	メッセージ式、セレクタ＝メソッド名は *message*
[*string* isEqual:*@"文字列"*]	単一引数をもつメッセージ式、セレクタは isEqual:
[NSString stringWithCstring:*buffer* encoding:*stringEncoding*]	複数引数をもつメッセージ式、セレクタは stringWithCstring:encoding:
[string withFormat:*@"%@=%g\n", @"文字列", 100.0*]	可変引数をもつメッセージ式、セレクタは withFormat:
receiver.value	ドット記法、アクセサのゲッターメソッドの呼び出し、[*receiver value*]と等価
receiver.value = 100.0	ドット記法、アクセサのセッターメソッドの呼び出し、[*receiver* set*Value*:*100.0*]と等価
@selector(*method_name*)	コンパイル済みセレクタを返すディレクティブ

まとめ

Core Graphics programming & Cocoa Touch

1. 平面上の球の壁への衝突のシミュレータを制作した
2. UIView を継承したカスタム View「**WallBallView**」で
   Core Graphics プログラミングを行った
3. NSTimer による View の更新：セレクタディレクティブ
4. デバイスの向きへの追従を試み：
   UIView の shouldAutorotateToInterfaceOrientation: メソッド
5. ピンチジェスチャの取得：UIPinchGestureRecognizer クラス
6. ドラッグジェスチャの取得：UIPanGestureRecognizer クラス
7. 加速度センサの利用：UIAccelerometer クラスとプロトコル

参考文献

Core Graphics Programming & Cocoa Touch

1. Apple Inc., “iPhoneヒューマンインターフェイスガイドライン,” 2010. ジェスチャを適切にサポートするための注意事項の紹介

2. Apple Inc., “iPhone開発ガイド,” 2010. 簡単なiPhoneアプリ作成からシミュレータでのジェスチャの実行方法などの紹介

3. Apple Inc., “iPhoneアプリケーションプログラミングガイド,” 2010. グラフィックスおよび描画システムの概要や横長モードでの起動の方法の紹介

4. Apple Inc., “Quartz 2D Programming Guide,” 2010. Core Graphics の Quartz 2D の詳細

5. Apple Inc., “Core Animationプログラミングガイド,” 2008. 本稿では扱わなかった Core Animation の詳細