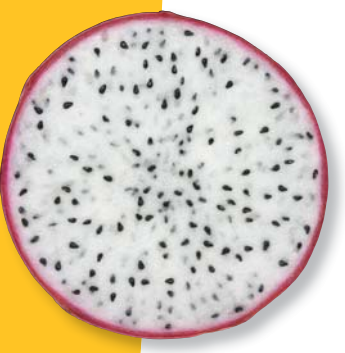
iPhone Cool Projects

DAVE MARK, SERIES EDITOR

GARY BENNETT
WOLFGANG ANTE
MIKE ASH
BENJAMIN JACKSON
NEIL MIX
STEVEN PETERSON
MATTHEW "CANIS" ROSENFELD



Apress®

iPhone Cool Projects

ISBN-13 (pbk): 978-1-4302-2357-3

ISBN-13 (electronic): 978-1-4302-2358-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Clay Andres
Development Editor: Douglas Pundick
Technical Reviewers: Glenn Cole, Gary Bennett
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Copy Editor: Heather Lang
Associate Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor: Dina Quan
Proofreader: April Eddy
Indexer: BIM Indexing & Proofreading Services
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`. You will need to answer questions pertaining to this book in order to successfully download the code.

Benjamin Jackson

Company: *Brainjuice*

Location: *Rio de Janeiro, Brazil*

Former life as a developer: *Languages, roughly in order: BASIC, Pascal, C/C++, Objective Caml, Java, Flash/ActionScript, and Ruby/Rails. Development environments: Eclipse, Visual Studio, and Flash. Programs used: TextMate, Creative Suite, and Terminal.app.*

Life as an iPhone developer: *Created the Arcade Hockey game and Town Hall reference application*

iPhone development toolset: *TextMate for editing, Xcode for building, and instruments and clang static analyzer for detecting leaks*

What's in this chapter: *This chapter covers setup with cocos2d, responding to touch events, and collision detection*

Key technologies

- *cocos2d*
- *Chipmunk*
- *OpenGL ES*

Chapter 5

Physics, Sprites, and Animation with the cocos2d-iPhone Framework

So you want to write an iPhone game with realistic physics. This chapter will get you started.

Here's our story: Brainjuice was founded in 2007 by myself and Ivan Neto as the product division of our agency, INCOMUM Design & Concept. That December, we launched Blogo, a weblog editor and our first Mac desktop application.

After the first version of the iPhone SDK was released, developing native applications for the phone was a natural next step. We were already well versed in Objective-C, and the freedom to mix C with Objective C when needed (as well as take advantage of all of the existing C libraries out there) is a big help. The touch screen interaction is unmatched on any other device, and when you add in the three-way accelerometer, the possibilities for different interfaces and controls are endless. Gestural controls mixed with realistic physics can create a scarily immersive game experience.

We made the choice to hold off an iPhone version of Blogo, which we knew would be a complex problem both in terms of the interface design and the number of different services we would have to integrate with. Instead, we wanted to cut our teeth on a game, which would help us get a feel for developing in a multitouch environment. Our experience at the time included 2D

scrollers with Flash, 3D augmented reality simulation, and virtual world development on multiple platforms, but we still hadn't done anything with two-player competition.

Air hockey is an arcade classic with mass appeal. We all played it as kids, and none of the existing hockey games in the App Store was up to our standards in terms of playability and realism. Factors like the goal size, the friction on the table, and the size of the mallet have a huge impact on the game play experience, and none of the applications we tested had the right combination. Most were free.

In this chapter, I'm going to explain some of the concepts behind 2D game programming. To illustrate, I'll discuss the process we went through developing Arcade Hockey, including the challenges we faced and how we got around them. I'll explain lighting, collision detection, physics simulation, and how to get a game from prototype to playable.

Getting Started with Game Programming

Programming a game is fundamentally different from other types of applications, on or off the iPhone. Rather than designing a series of screens for the user to navigate, the developer has to design a virtual world for the user to interact with. This space can be either 2D or 3D, and the programmer's task is twofold:

- Design the space, and define its constraints. These can be the outer boundaries of the space as well as the internal ones (for example, in a maze game), but constraints are not limited to position. Friction, elasticity, gravity, and even lighting can all play a role in defining the user's experience in the virtual space.
- Define rules for how objects interact with the space. This generally comes down to defining what happens when objects collide with each other. For example, a user might score when the ball hits the back wall of a goal or die when the player comes into contact with a lethal object.

In this chapter, we're going to look at the unique challenges presented by developing a 2D game on the iPhone. To do this, we're going to use OpenGL ES with help from the cocos2d and Chipmunk libraries to create a simple miniature golf game (as shown in Figure 5-1).

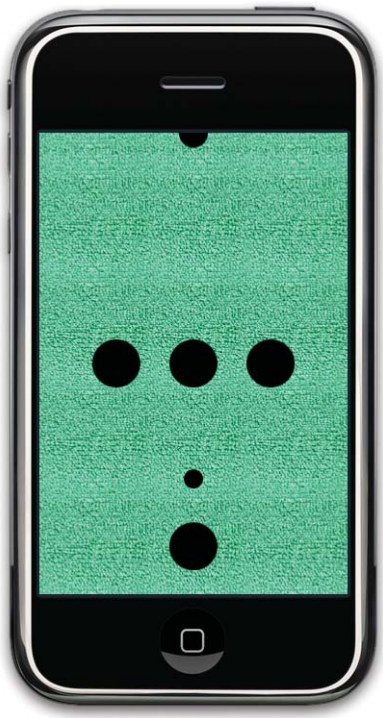


Figure 5-1. *The miniature golf game layer*

Introducing OpenGL ES

OpenGL ES is the most widely used graphics programming API today. Implemented as a standard C library, it forms the basis for countless 2D and 3D games and development frameworks. On systems with processors optimized for OpenGL, the heavy computation is passed off to the graphics card, resulting in much better performance and less load on the CPU.

OpenGL ES is a subset of the OpenGL API optimized for embedded devices like mobile phones, PDAs, and handheld video game consoles like the Nintendo DS and PlayStation Portable (PSP).

Introducing cocos2d and Chipmunk

OpenGL is powerful, but hand-coding the same calls to its cryptic method names repeatedly will have any sane person leaving head-shaped holes in the wall before the first prototype is out the door. Enter cocos2d and Chipmunk—two libraries that take the raw power of OpenGL and add a layer of abstraction to package it up into the language of 2D game programming.

In 3D game programming, rendering is based on sets of 3D coordinates mapped to **textures**. In 2D game programming, we base the display on 2D coordinates mapped to **sprites** (flat graphics moved in the two-dimensional space). Sprites are organized in **layers** to allow for simple stacking on the Z axis. cocos2d works with 2D **vectors** (sets of coordinates) to define both points on the stage and transformations on those points (for example, "move vector X by vector Y"). We use vectors to define our sprites' positions.

To simulate real-world physics, we piggyback Chipmunk on top of cocos2d. Chipmunk adds functions that let us define the physical properties of our sprites (friction and elasticity), as well as how they interact with each other when they collide.

Developing Arcade Hockey

In this section, we're going to discuss the process behind creating Arcade Hockey (shown in Figure 5-2). During the development, we learned a lot about what it takes to make a game not only playable but fun, and we ran into more than a few stops that required some creative workarounds.

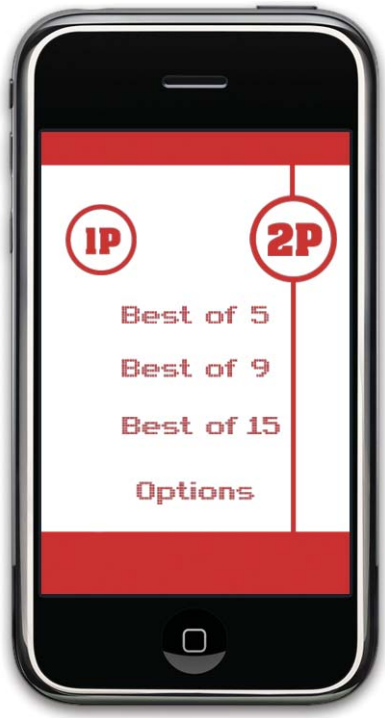


Figure 5-2. *The main menu of Arcade Hockey*

We began by studying the other hockey games in the App Store, and a couple of paid competitors gave us an early scare. We downloaded and tested all of the other applications to see what the competition had come up with already.

We noticed that the goals were all so small that scoring was nearly impossible if the opponent kept the mallet front and center. We also noticed that the artificial intelligence's strategies invariably revolved around keeping the mallet front and center—no coincidence there. After we had an idea of what needed improvement, we moved on to sketching out the screens and basic options.

For the brand and interface design, we chose a creative direction based around the old-school 80s video-game graphics, which were the norm in the arcades where air hockey became popular. All graphics were done as vector illustrations in Adobe Illustrator, which allowed us the liberty of scaling them for multiple uses without losing quality. For example, Arcade Hockey offers multiple mallet and puck sizes (see Figure 5-3). We mocked up a prototype with a bare table and placeholder graphics to begin working on the physics while the sprites were being designed.

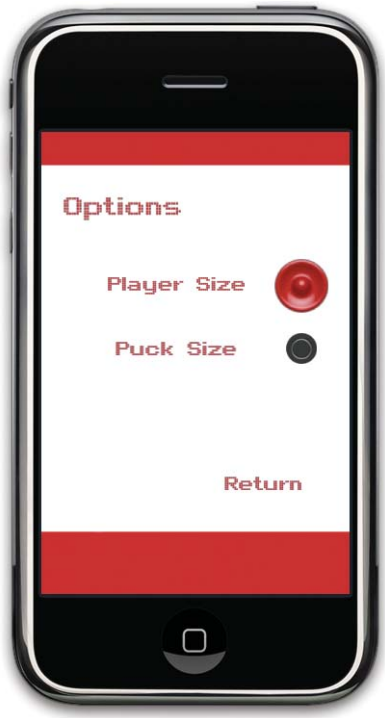


Figure 5-3. *The Options screen*

Once the mallets felt right, we began sketching out the basic game play with the puck. We quickly learned that collision detection was far from plug and play, and we began adjusting the code to avoid bugs like the multiple collisions that would pile up when the puck was moving slowly and barely touched a mallet or when pressing the puck up against a wall. To keep these kinds of bugs from hitting your own application, make sure to set a flag when the first collision is detected and set a timer to clear it after a fraction of a second.

From there, we moved on to the artificial intelligence (AI). This turned out to be a huge part of the work, and we went through several different iterations with varying strategies until we got the AI smart enough to present a real challenge.

For example, how do you know whether or not the enemy is on the attack or on the defensive? One way is to look at which half of the table the puck is in. But this doesn't take into account cases where the puck is in the enemy's area but moving toward the goal. In the end, the best strategies combine knowledge of the puck's position relative to the mallet and its velocity.

Finally, we fleshed out the Options screen, as shown in Figure 5-3, by adding different puck and mallet sizes and added the finished art to the program, as well as some final touches like the simulated game noises in the first screen, and the flaming puck that appears in a

sudden-death match point. We also did some fine-tuning on the game play, adjusting the maximum speed of the puck to keep the game from getting out of hand.

Next, I'll go into some more detail on a couple of the specific challenges we faced and show you how we solved them.

Tracking the User's Finger

Once we were ready to start coding, we started by getting the mallets working and tracking the user's finger. This proved to be a bit of a difficult problem because of an important usability issue: the user needed to be able to see the mallet while controlling it at the same time. We found, after playing around for a bit, that the best position for the mallet is slightly in front of the finger.

However, this meant that there was no way for the user to protect the goal when pulling back to defend. In the end, we had to adjust the puck's position relative to the user's finger so that, when at the back of the screen, it would be directly underneath (as shown in Figure 5-4), and gradually move forward as the mallet approached the center line.

Figure 5-4. *Finger tracking while guarding the goal (shaded circles show finger positions)*

Listing 5-1 shows a code snippet for compensating for the user's finger.

Listing 5-1. *Compensating for the User's Finger*

```
int finger_padding = 30;
int fat_fingers_offset = 40; // we add a little bit more to the offset

// ... snip
```

First, we set some values, which we'll use later to offset the puck in relation to the user's finger:.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  for (UITouch *myTouch in touches) {

    CGPoint location = [myTouch locationInView: [myTouch view]];

    // translate location coordinates into game coordinates
    location = [[Director sharedDirector] convertCoordinate: location];
```

Next, we get the touch location relative to the view. We need to convert this coordinate relative to the current layer using the `convertCoordinate:` method in the `Director` object:

```
    if (CGRectContainsPoint(player1AreaRect, location)) {

      // don't let the mallet pass into the other player's area
      if (CGRectContainsPoint(notPlayer1AreaRect, location)) {
        break;
      }
```

We block the mallet from hitting any point outside its area and break out if we find it inside `notPlayer1AreaRect` (defined earlier in the code):

```
      // calculate the padding based on the location
      cpFloat padding = finger_padding * ((120 - location.y) / 100);
      location.y -= padding;
      location.y += fat_fingers_offset;
```

This is the trick: we cut the mallet's `y` value down by a factor that approaches zero as the finger reaches the bottom edge of the table and increases linearly to a maximum of 80 pixels ahead of the finger at midfield:

```
      // lock the location to the center line
      if (location.y > 240) location.y = 240;
      cpVect p1position = cpv(location.x, location.y);
      cpMouseMove(player1Mouse, p1position);
    }
```

Since we're pushing the puck ahead, we want to lock it to the middle of the table as well:

```
    else if (CGRectContainsPoint(player2AreaRect, location) &&
             !singlePlayerMode) {

      // don't let the mallet pass into the other player's area
      if (CGRectContainsPoint(notPlayer2AreaRect, location)) {
        break;
      }

      // calculate the padding based on the location
      cpFloat padding = finger_padding * (( location.y - 360 ) / 100);
      location.y += padding;
      location.y -= fat_fingers_offset;

      // lock the location to the center line
      if (location.y < 240) location.y = 240;
      cpVect p2position = cpv(location.x, location.y);
      cpMouseMove(player2Mouse, p2position);
    }
  }
}
```

We then do the same for the second player, unless we're in single-player mode.

Detecting Collisions

Collision detection is at the heart of any game that attempts to simulate reality. In the real world, objects bounce off each other with a specific direction and velocity depending on the inertia of the two bodies and their speed and direction.

In most games, we're not only interested in the objects colliding realistically but also in triggering events based on those collisions (for example, a rocket colliding with its target). To handle these special cases, we attach callbacks (stored as C function pointers) to event types (stored as C constants).

We were surprised by the difficulty of getting collision detection working and playable. In particular, getting the mallets to continue on their trajectories after the user released them was impossible until we noticed on the mailing list that Chipmunk, the physics library we were using, had been updated with the new `cpMouse` functions. These functions simulate a mouse pointer in 2D space, allowing you to programmatically move an object while retaining its momentum after being released and were essential for the mallet to move realistically under the user's finger.

Sometimes, the puck would go so fast that it passed the wall without generating any collision event at all. After banging our heads against the wall trying to figure out how to

prevent this, we ended up treating it as the virtual equivalent of the puck flying off the table with a careless hit.

We also noticed that the corners of the table would end up trapping the puck and made it impossible to continue play without pushing it off the table (as shown in Figure 5-5).

Figure 5-5. *Avoid getting the puck trapped in the corner.*

Giving the puck a small impulse toward the center when it hit the corners was enough to fix the problem, as shown in Listing 5-2.

Listing 5-2. *Keeping the Puck Out of the Corner*

```
enum {
  kColl_Puck,
  kColl_Goal,
  kColl_Horizontal,
  kColl_Player1,
  kColl_Player2,
  kColl_Pusher
};

// snip... initialize other variables
```

The first step to detecting collisions is initializing the types of collisions in an `enum`. We'll use these later to associate collision areas with C functions.

```
@implementation GameLayer

- (id) init {

  // snip... sets up the board, players etc.

  // add the puck
  puck = [self addSpriteNamed:@"puck.png"
    x:160 y:240 type:kColl_Puck];
```

First, we add the puck using `addSpriteNamed:`, the convenience function that follows. It takes an image, its dimensions, and a collision type (in this case `kColl_Puck`).

```
  // and a shape to the corner which will push it out
  cpShape *pusher = cpCircleShapeNew(staticBody, radius,
    cpv(distance, distance));
  pusher -> collision_type = kColl_Pusher;
  cpSpaceAddStaticShape(space, pusher);
```

Next, we add a collision shape to the corner and give it the `kColl_pusher` type we defined earlier.

```
  // finally add a collision pair function which
  // will be called when they collide
  cpSpaceAddCollisionPairFunc(space, kColl_Puck, kColl_Pusher,
    &puckHitPusher, puck);
}
```

Finally, we associate the collision area with the C function. To do this, we call `cpSpaceAddCollisionPairFunc` and send it our space (defined in the `GameLayer`'s `init` function with `cpSpaceNew()`), the collision types that we want to associate, a reference to the C function, and a reference to the puck (this is not used, and could be `NULL`, but it doesn't hurt in case we need it later).

```
- (void)pushPuckFromCorner{
  cpBodyApplyImpulse(puck, cpvsub(cpv(160,240), puck->p), cpvzero);
}
```

Here's the key to the fix: we define a function to push the puck from the corner that will be called later from our C function callback. This just applies an impulse on the puck toward the center of the table.

```
- (cpBody *) addSpriteNamed: (NSString *)name x: (float)x
    y:(float)y type:(unsigned int)type {

  // add a new sprite and center it
  Sprite *sprite = [Sprite spriteFromFile:name];
  [self add: sprite z:2];
  sprite.position = cpv(x,y);
```

Now, we get to the convenience function we used earlier. We start by creating and positioning a new sprite.

```
  // set up the vertexes based on the image dimensions
  UIImage *image = [UIImage imageNamed:name];
  int num_vertexes = 4;

  cpVect verts[] = {
    cpv([image size].width/2 * -1, [image size].height/2 * -1),
    cpv([image size].width/2 * -1, [image size].height/2),
    cpv([image size].width/2, [image size].height/2),
    cpv([image size].width/2, [image size].height/2 * -1)
  };
```

We then set up its collision vertices based on the image dimensions, centering the box on the center of the sprite.

```
  // every object needs a body
  cpBody *body = cpBodyNew(1.0, cpMomentForPoly(1.0, num_vertexes,
    verts, cpvzero));
  body->p = cpv(x, y);
  cpSpaceAddBody(space, body);

  // as well as a shape to represent its collision box
  cpShape* shape = cpCircleShapeNew(body, [image size].width / 2,
    cpvzero);
  shape->e = 0.5;  // elasticity
  shape->u = 0.5;  // friction
  shape->data = sprite;
  shape -> collision_type = type;
```

We give the object a body and a shape with the vertices we defined and set its elasticity, friction, sprite data, and collision type.

```
  cpSpaceAddShape(space, shape);
  return body;
}
```

Finally, we add the shape to the space so it will show up on our main layer.

```
@end

int puckHitPusher(cpShape *a, cpShape *b, cpContact *contacts,
  int numContacts, cpFloat normal_coef, void *data) {
  [(GameLayer *) mainLayer pushPuckFromCorner];
  return 0;
}
```

Our C function callback simply forwards the message on to the main layer and pushes the puck out from the corner.

Simulating 3D Lighting in 2D Space

To simulate realistic lighting on the mallets, we resorted to a classic trick. If we center the light on the table, we can fake the direction of the light by rotating the mallets based on their position relative to the center (as shown in Figure 5-6). This will make the mallets' highlights and shadows appear to respond realistically to the light.

Figure 5-6. *The minigolf game layer with player sprites rotated based on their angles to the center*

In Listing 5-3, you'll learn how to rotate sprites relative to the center of the playing area to simulate a central light source.

Listing 5-3. *3D Lighting Simulation*

```
- (void)step: (ccTime) delta {

  // snip...

  // handle the rotation of the pucks to emulate
  // the effect of a light source
  cpVect newPosition;

  // find the center of the stage
  CGRect wins = [[Director sharedDirector] winSize];
  cpVect centerPoint = cpv(wins.size.width/2, wins.size.height/2);

  // rotate player 1
  newPosition = cpvsub(centerPoint, player1->p);
  [(Sprite*) p1_shape->data setRotation:90 -
    RADIANS_TO_DEGREES(cpvtoangle(newPosition))];

  // rotate player 2
  newPosition = cpvsub(centerPoint, player2->p);
  [(Sprite*)p2_shape->data setRotation: 90 -
    RADIANS_TO_DEGREES(cpvtoangle(newPosition))];
}
```

We first find the center of the stage by cutting the main window's width and height in half. For each player, we subtract the mallet's position from the center point and use that to calculate the angle with `cpvtoangle`, using the result to set the rotation of the sprite. We call the standard OpenGL `RADIANS_TO_DEGREES` macro to convert from radians returned by `cpvtoangle` into the degrees passed to `setRotation`.

Creating a Simple Application

In this section, we're going to build a simple prototype of a miniature golf game (shown in Figure 5-1). The user will drag a mallet with a finger and try to get the ball through the barriers to score a hole in one. Touching the back wall will cause the ball to return to the starting point.

Setting Up the Xcode Project

We start with a view-based application in Xcode (as shown in Figure 5-7), though we'll be trashing the view code and using the cocos2d `Layer` class for all of our graphics.

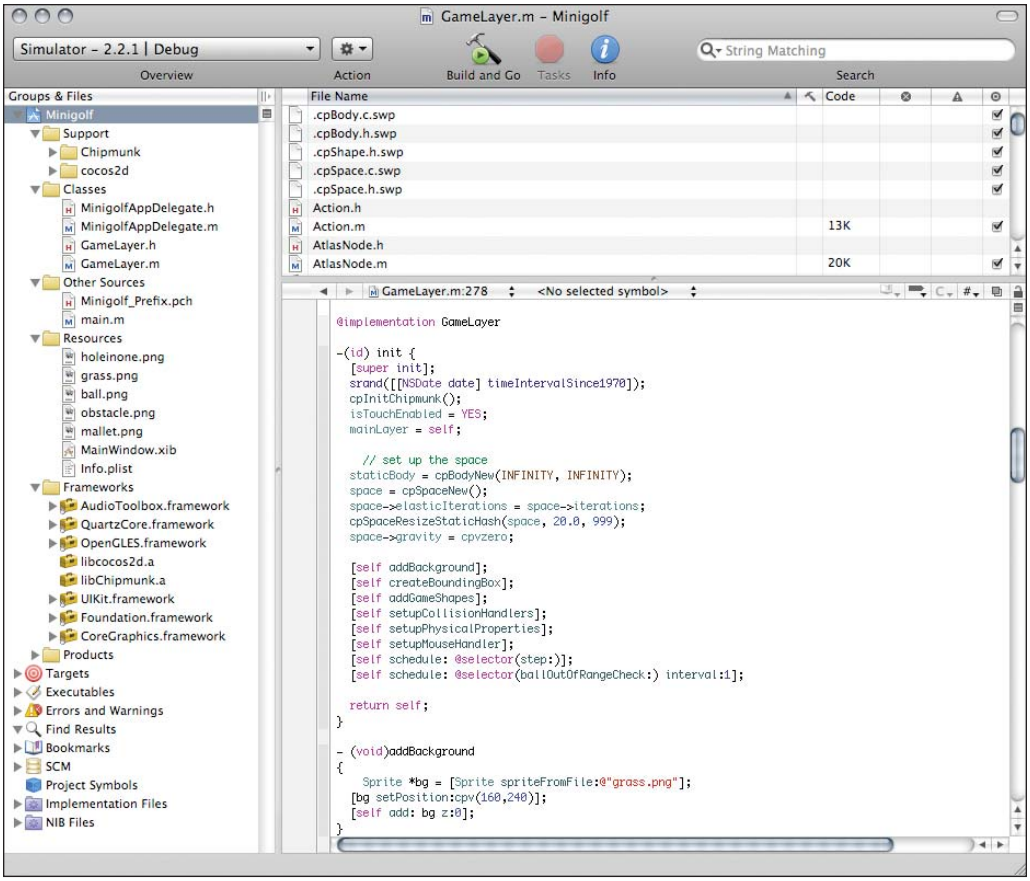


Figure 5-7. *The miniature golf Xcode project*

The application uses the following frameworks:

- *libcocos2d*: The 2D gaming library available at `http://cocos2d.org`. The version used in this project is 0.5.2; you may need to adjust your code if you use later versions, as the API is not yet stable.
- *libChipmunk*: The rigid body physics library at `http://code.google.com/p/chipmunk-physics/`.
- *AudioToolbox*: Makes the phone vibrate. (Yes, vibration is treated like any other sound.)
- *OpenGL ES*: A dependency of cocos2d. (See "Getting Started with Game Programming" for more details.)
- *CoreGraphics*: For drawing graphics on-screen and used by OpenGL ES.
- *QuartzCore*: Also for drawing graphics on-screen and used by OpenGL ES.

You'll need to drag the AudioToolbox, CoreGraphics, QuartzCore, and OpenGL ES frameworks to the Frameworks group in Xcode to start.

We also need to add the header and source files for cocos2d and Chipmunk to the Xcode project by dragging them to the Support group and to add some images to the Resources group for our different sprites.

Setting the Scene

We set the scene in the application delegate, a class that receives notifications from the application on major events like startup and shutdown. In this section, we'll define the scene and set things in motion for our main layer to start running.

Listing 5-4 shows how to set up the scene in the application delegate and hand off control to the `Director`.

Listing 5-4. *Setting Up the Scene in the Application Delegate*

```
- (void)applicationDidFinishLaunching:(UIApplication*)application
{
  [[Director sharedDirector] setAnimationInterval:1.0/60];
  Scene *scene = [Scene node];
  [scene add: [GameLayer node] z:0];
  [[Director sharedDirector] runScene: scene];
}
```

Now that we've set up the application delegate, we can get started on the main layer. The application starts in the `MinigolfAppDelegate`'s `applicationDidFinishLaunching:` method. Here, we set the frame rate (60 frames per second in our example), create the main scene, and add a layer to it. Finally, we kick-start the cocos2d director into run mode by passing the scene we created.

```
- (void)applicationWillResignActive:(UIApplication *)application
{
  [[Director sharedDirector] pause];
}

- (void)applicationDidBecomeActive:(UIApplication *)application
{
  [[Director sharedDirector] resume];
}
```

If the user gets a call, we prepare our application to pause and resume sensibly by calling the relevant methods `pause` and `resume` on the `Director` singleton.

Creating the Game Layer

The game layer is where the action happens. This layer contains the walls of our space with a recessed area in the back, which will serve as the hole. We'll also add the ball, club, and some obstacles to make things interesting.

After the call to `runScene` (shown in Listing 5-4), our `Layer` subclass's `init` method is called (see Listing 5-5). This is where we create all our sprites, set them on the stage, and assign masses and frames to them so they'll act like real objects. We also set up all of the collision callbacks (C function pointers that will fire when objects collide), so we can respond to collision events between different types of objects. The collision types are defined in the `enum` at the top of *GameLayer.m*.

Listing 5-5. *Defining the Space in the Main Game Layer*

```
@implementation GameLayer

- (id) init {
  [super init];
  srand([[NSDate date] timeIntervalSince1970]);
  isTouchEnabled = YES;
  mainLayer = self;

  // set up the space for Chipmunk
  staticBody = cpBodyNew(INFINITY, INFINITY);
  space = cpSpaceNew();
  space->elasticIterations = space->iterations;
  cpSpaceResizeStaticHash(space, 20.0, 999);
  space->gravity = cpvzero;

  [self addBackground];
  [self createBoundingBox];
  [self addGameShapes];
  [self setupCollisionHandlers];
  [self setupPhysicalProperties];
  [self setupMouseHandler];
  [self schedule: @selector(step:)];
  [self schedule: @selector(ballOutOfRangeCheck:) interval:1];

  return self;
}
```

We start by seeding the random number generator and initializing Chipmunk (shown in Listing 5-5) and then create a space and enable bounce. We set the gravity to zero, since the field is flat and objects shouldn't be drawn in any direction by default.

```
- (void)addBackground
{
  Sprite *bg = [Sprite spriteFromFile:@"grass.png"];
  [bg setPosition:cpv(160,240)];
  [self add: bg z:0];
}
```

The background is just a sprite created from a resource in the Xcode project. We add it at the lowest z-index.

```
#define GOAL_MARGIN 145

// snip...

- (void)createBoundingBox
{
  cpShape *shape;

  CGRect wins = [[Director sharedDirector] winSize];
  startPoint = cpv(160,120);

  // make bounding box
  cpFloat top = wins.size.height;
  cpFloat WIDTH_MINUS_MARGIN = wins.size.width - GOAL_MARGIN;

  // bottom
  shape = cpSegmentShapeNew(staticBody, cpv(0,0),
    cpv(wins.size.width,0), 0.0f);
  shape->e = 1.0; shape->u = 1.0;
  cpSpaceAddStaticShape(space, shape);

  // top
  shape = cpSegmentShapeNew(staticBody, cpv(0,top),
  cpv(GOAL_MARGIN ,top), 0.0f);
  shape->e = 1.0; shape->u = 1.0;
  cpSpaceAddStaticShape(space, shape);
  shape -> collision_type = kColl_Horizontal;

  // and so on...
```

We then set up the bounding box, creating a small box behind the far wall, which we'll be using as our hole. The shapes are given collision types of `kColl_Horizontal` and `kColl_Goal`, which will later be hooked into the hole in one callback.

```
- (void)addGameShapes
{
  ball = [self addSpriteNamed:@"ball.png" x:160 y:120 type:kColl_Ball];
  obstacle1 = [self addSpriteNamed:@"obstacle.png" x:80
```

```
    y:240 type:kColl_Ball];
  obstacle2 = [self addSpriteNamed:@"obstacle.png" x:160
    y:240 type:kColl_Ball];
  obstacle3 = [self addSpriteNamed:@"obstacle.png" x:240
    y:240 type:kColl_Ball];
  player = [self addSpriteNamed:@"mallet.png" x:160
    y:50 type: kColl_Player];
}
```

To add the shapes, we extract out a convenience function, which sets up the sprite and its physical properties.

```
- (cpBody *) addSpriteNamed: (NSString *)name x: (float)x
    y:(float)y type:(unsigned int) type {

  UIImage *image = [UIImage imageNamed:name];
  Sprite *sprite = [Sprite spriteFromFile:name];
  [self add: sprite z:2];
  sprite.position = cpv(x,y);
```

We start by grabbing the image, creating and positioning its sprite to the x and y values passed in as arguments.

```
  int num_vertices = 4;
  cpVect verts[] = {
    cpv([image size].width/2 * -1, [image size].height/2 * -1),
    cpv([image size].width/2 * -1, [image size].height/2),
    cpv([image size].width/2, [image size].height/2),
    cpv([image size].width/2, [image size].height/2 * -1)
  };

  // all objects need a body
  cpBody *body = cpBodyNew(1.0, cpMomentForPoly(1.0, num_vertices,
    verts, cpvzero));
  body->p = cpv(x, y);
  cpSpaceAddBody(space, body);

  // as well as a shape to represent their collision box
  cpShape* shape = cpCircleShapeNew(body, [image size].width / 2,
    cpvzero);
  shape->data = sprite;
  shape -> collision_type = type;

  if (type == kColl_Ball) {
    shape->e = 0.5f; // elasticity
    shape->u = 1.0f; // friction
  } else {
```

```
    shape->e = 0.5; // elasticity
    shape->u = 0.5; // friction
  }
```

We then give it a body and a shape, assign elasticity and friction, and set the shape's data and collision type. If you want different values for different shapes, you'll want to set that here as it's difficult to get at the shape later.

```
  cpSpaceAddShape(space, shape);
  return body;
}
```

Finally, we add the shape to the space before returning the body.

```
// collision types
enum {
  kColl_Ball,
  kColl_Goal,
  kColl_Horizontal,
  kColl_Player
};

// snip...


- (void)setupCollisionHandlers
{
  cpSpaceAddCollisionPairFunc(space, kColl_Ball, kColl_Goal,
    &holeInOne, ball);
  cpSpaceAddCollisionPairFunc(space, kColl_Ball, kColl_Horizontal,
    &restart, ball);
}
```

Setting up the collision handlers is just a matter of calling `cpSpaceAddCollisionPairFunc` with our space, the two types colliding, a pointer to the function we want to call when they collide, and some data to be passed to the callback (we won't be using this, but you have to send something).

```
void resetPosition(cpBody *ball) {
  cpBodyResetForces(ball);
  ball -> v = cpvzero;
  ball -> f = cpvzero;
  ball -> t = 0;
  ball -> p = startPoint;
  AudioServicesPlaySystemSound(kSystemSoundID_Vibrate);
}
```

```
static int holeInOne(cpShape *a, cpShape *b, cpContact *contacts,
  int numContacts, cpFloat normal_coef, void *data) {
  GameLayer *gameLayer = (GameLayer *) mainLayer;
  [gameLayer holeInOne];
  return 0;
}

static int restart(cpShape *a, cpShape *b, cpContact *contacts,
  int numContacts, cpFloat normal_coef, void *data) {
  cpBody *ball = (cpBody*) data;
  resetPosition(ball);
  return 0;
}
```

The callback functions themselves are straightforward. A hole in one forwards the message to the `GameLayer`, where victory is signaled with a sprite overlaid on top of the main layer and a quick vibration of the phone.

```
- (void)setupPhysicalProperties
{
  cpBodySetMass(ball, 25);
  cpBodySetMass(obstacle1, INFINITY);
  cpBodySetMass(obstacle2, INFINITY);
  cpBodySetMass(obstacle3, INFINITY);
  cpBodySetMass(player, 2000);
}
```

We set the mass of the player to be substantially larger than the mass of the ball to get a good speed when hitting and set the obstacles to `INFINITY` so that they don't move when hit (they are anchored to the ground after all).

```
- (void)setupMouseHandler
{
  playerMouse = cpMouseNew(space);
  playerMouse->body->p = player->p;
  playerMouse->grabbedBody = player;

  // create two joints so the body isn't rotated
  // around the finger point
  playerMouse->joint1 = cpPivotJointNew(playerMouse->body,
    playerMouse->grabbedBody,
    cpv(playerMouse->body->p.x - 1.0f,
    playerMouse->body->p.y));
  cpSpaceAddJoint(playerMouse->space, playerMouse->joint1);

  playerMouse->joint2 = cpPivotJointNew(playerMouse->body,
    playerMouse->grabbedBody,
```

```
    cpv(playerMouse->body->p.x + 1.0f,
    playerMouse->body->p.y));
  cpSpaceAddJoint(playerMouse->space, playerMouse->joint2);
}
```

Next, we set up the mouse handler. This lets us treat the mallet as a draggable cursor while retaining its physical properties when it interacts with the other objects. We give it two **joints** (points which serve as axes for rotation in the mouse), because we'll be positioning it in front of the finger, and we don't want it to rotate around the touch point when moved.

```
- (void)touchesMoved:(NSSet*)touches withEvent:(UIEvent*)event{
  CGPoint playerTouchLocation = CGPointMake(-300, 240);

  for (UITouch *myTouch in touches) {
    CGPoint location = [myTouch locationInView: [myTouch view]];
    location = [[Director sharedDirector] convertCoordinate: location];
    // set the finger location to be the lowest touch
    playerTouchLocation.x = location.x;
    playerTouchLocation.y = location.y;
  }

  // into game coords...
  CGPoint location = playerTouchLocation;
  cpFloat padding = finger_padding * ((120 - location.y) / 100);
  location.y -= padding;
  location.y += fat_fingers_offset;
```

In the touch event handler, we grab the last touch in the set (in this case, the only touch) and compensate for the user's finger based on how far up it is on the field. This way, the user can see the mallet and still bring the mallet all the way back to the beginning of the field.

```
  // trap the location to half-field
  if (location.y > 230) location.y = 230;
  if (location.y < 0) location.y = 0;
```

We trap the position to half-field so that the user can't cheat.

```
  cpVect playerposition = cpv(location.x, location.y);
  cpMouseMove(playerMouse, playerposition);
}
```

And finally, we create a vector and move the mouse to it.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  [self touchesMoved:touches withEvent:event];
}
```

The `touchesBegan: withEvent:` handler is identical to `touchesBegan: withEvent:` but could be different depending on what you want to trigger when dragging starts.

Finally, we schedule the following two callbacks:

```
static void eachShape(void *ptr, void* unused) {
  cpShape *shape = (cpShape*) ptr;
  Sprite *sprite = shape->data;
  if (sprite) {
    cpBody *body = shape->body;
    [sprite setPosition: cpv(body->p.x, body->p.y)];
  }
}

// snip...

- (void)step: (ccTime) delta {
  int steps = 1;
  cpFloat dt = delta/(cpFloat)steps;

  for (int i=0; i<steps; i++) {
    cpSpaceStep(space, dt);
  }

  cpSpaceHashEach(space->activeShapes, &eachShape, nil);
  cpSpaceHashEach(space->staticShapes, &eachShape, nil);

}
```

The first is a C function, `eachShape`, which makes the animation engine run. It starts a timer that calls `step:` on our layer at regular intervals. In the `step:` function, we increment the physical properties of the layer by calling `cpSpaceStep` with the time difference. We step the display by sending `cpSpaceHashEach` for the active and static shapes, passing it a function that positions the sprites.

```
- (void) ballOutOfRangeCheck: (ccTime) delta {
  if (ball -> p.x > 320 || ball -> p.x < -80 ||
     kball -> p.y > 550 || ball -> p.y < -80) {
        resetPosition(ball);
    }
}
```

The function `ballOutOfRangeCheck:` checks to see if the ball has been pushed so fast that it passed through the wall. If so, we treat it as having gone off the course in the same way we treated the puck going off the hockey table, and we reset the position for the user to try again.

```
- (void)holeInOne
{
  AudioServicesPlaySystemSound(kSystemSoundID_Vibrate);
  holeInOneBG = [Sprite spriteFromFile:@"holeinone.png"];
  [holeInOneBG setPosition:cpv(160,240)];
  [self add:holeInOneBG z:10];
  [self performSelector:@selector(resetGame:) withObject:nil
    afterDelay:2.0];
}

- (void)resetGame:(id)object
{
  [self remove:holeInOneBG];
  resetPosition(ball);
}
```

Finally, we include the function `holeInOne`, which is called when the user scores. Vibration is done with `AudioServicesPlaySystemSound` using the `kSystemSoundID_Vibrate` constant. Finally, the function sets a timer to remove the message and reset the game after 2 seconds.

Summary

You now have enough knowledge to start messing around with the laws of nature.

We've really only begun to scratch the surface of what can be done with Chipmunk and cocos2d. You can create skeletons and joint systems, which can be snapped together like LEGO bricks to simulate anything from a simple bicycle to a human being. You can then throw your objects into a space with whatever goals and constraints you choose to define to make the game a challenge. If you want to make things even more difficult, you can play with gravity, or even use the accelerometer to alter the gravity of the scene. Of course, as with any decision you make in your code, you should always be questioning whether or not you're adding to the playability of the game.