



Quick answers to common problems

Cinder Creative Coding Cookbook

Create compelling animations and graphics with Kinect and camera input, using one of the most powerful C++ frameworks available

Dawid Gorny

Rui Madeira

[PACKT] open source*
PUBLISHING community experience distilled

www.allitebooks.com

Cinder Creative Coding Cookbook

Create compelling animations and graphics with Kinect and camera input, using one of the most powerful C++ frameworks available

Dawid Gorny

Rui Madeira

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Cinder Creative Coding Cookbook

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2013

Production Reference: 1160513

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-870-3

www.packtpub.com

Cover Image by Dawid Górný (hello@dawidgorny.com)

Credits

Authors

Dawid Gorny

Rui Madeira

Reviewers

Vladimir Gusev

Dofl Y. H. Yun

Acquisition Editors

Joanne Fitzpatrick

James Jones

Lead Technical Editor

Dayan Hyames

Technical Editors

Soumya Kanti

Devdutt Kulkarni

Veena Pagare

Project Coordinator

Arshad Sopariwala

Proofreaders

Maria Gould

Paul Hindle

Indexer

Rekha Nair

Production Coordinators

Aditi Gajjar

Prachali Bhiwandkar

Cover Work

Aditi Gajjar

About the Authors

Dawid Gorny is a creative coder and a creative technologist who is into computational design, art, and interaction design.

He has worked as a professional web and Flash developer for several years, then took the lead of the research and development department at a digital production house. He has worked on concepts and technical solutions for a wide variety of interdisciplinary projects involving mobile development, cameras, sensors, custom electronic circuits, motors, augmented reality, and projection mapping. His installations engage people in malls, airports, exhibition spaces, and other public venues.

He is the founder, organizer, and program director of the art+bits festival in Katowice—the encounter of art and technology.

You can find a more about his projects and experiments at <http://www.dawidgorny.com>

Rui Madeira is a computational designer, educator, and founder of the interaction design studio Estudio Ruim. He has been exploring and creating unique and engaging interactive experiences for cultural, artistic, and commercial purposes. His works are born from the intersection of several disciplines including illustration, animation, and interaction design. By using programming languages as the main building blocks for his works, he builds specific and adaptive systems that break apart from the limitations of traditional tools. He has participated in several projects, both collaborative and solo, including interactive performances and concerts, generative visuals for print and motion graphics, mobile applications, interactive installations, and video mapping.

He has collaborated for several institutions including the London College of Fashion, Belém Cultural Center, Pavillion of Knowledge, Portuguese Foundation of Communications, Moda Lisboa, National Ballet of Portugal, and the Monstra Animation Festival

About the Reviewers

Vladimir Gusev is a scientist turned generative graphics stage designer and producer. Vladimir Gusev received his Ph.D. from the Russian Academy of Sciences, and continued scientific research in the Ukraine (Kiev Polytechnic Institute) and the USA (Yale University). His main interest was computer molecular simulations, which led him into industrial bioinformatics and software development (molecular visualization and visual languages and platforms). His latest interest lies in theatre multimedia environments, which resulted in the production of works at the Budapest Summer Festival (Aida by G. Verdi), Anton Chekhov Moscow Art Theatre, Petr Fomenko Theatre Workshop (Moscow), and Satyricon (Moscow). He also co-founded the One Way Theater Company in New York City. Two theatrical productions with Vladimir's engagements as a videographer were nominated for the National Golden Mask Awards.

Some of the respectable institutions with which he collaborated are: The Institute of Physical Chemistry, Kiev Polytechnic Institute, Yale University, TRI/Princeton, Curagen Corporation, GraphLogic (Co-founder), Streambase, Ab Initio, Conde Nast, and One Way Theater Company.

He has also been a reviewer of several international journals on physical chemistry.

I would like to thank the creators of the wonderful Cinder framework.

Dofl Y.H. Yun is an interactive technologist with over 12 years of development experience, and he has established himself as a visionary leader in interactive design in South Korea, Hong Kong, the United Kingdom, and more recently in the USA. Much of his focus is on technologies such as computer vision, 3D depth camera sensors, and multitouch applications.

Dofl received his MA degree in Interactive Media from the London College of Communication, University of the Arts London with a thesis entitled: "Ensemble-Interactive Musical Instruments." His MA thesis won the Experimental/Art category at the Flashforward Film Festival 2008 in San Francisco.

Since August 2009 he has been working for Firstborn, a digital agency in New York City. His recent work focuses on exploring the intersection between physical space and interaction design.

I want to especially mention the efforts of Cinder's original author and current lead architect, Andrew Bell, and would like to thank my family for their support and my friends from the CinderDome community.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print and bookmark content
- ▶ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Getting Started	5
Introduction	5
Creating a project for a basic application	6
Creating a project for a screensaver application	8
Creating a project for an iOS touch application	9
Understanding the basic structure of an application	10
Responding to mouse input	13
Responding to key input	15
Responding to touch input	16
Accessing files dropped onto the application window	20
Adjusting a scene after resizing the window	22
Using resources on Windows	24
Using resources on iOS and OS X	26
Using assets	28
Chapter 2: Preparing for Development	31
Introduction	31
Setting up a GUI for tweaking parameters	31
Saving and loading configurations	36
Making a snapshot of the current parameter state	39
Using MayaCamUI	41
Using 3D space guides	43
Communicating with other software	47
Preparing your application for iOS	53
Chapter 3: Using Image Processing Techniques	55
Introduction	56
Transforming image contrast and brightness	56
Integrating with OpenCV	59

Detecting edges	62
Detecting faces	65
Detecting features in an image	67
Converting images to vector graphics	70
Chapter 4: Using Multimedia Content	77
Introduction	77
Loading and displaying video	77
Creating a simple video controller	80
Saving window content as an image	84
Saving window animations as video	86
Saving window content as a vector graphics image	90
Saving high resolution images with the tile renderer	94
Sharing graphics between applications	97
Chapter 5: Building Particle Systems	101
Introduction	101
Creating a particle system in 2D	101
Applying repulsion and attraction forces	109
Simulating particles flying in the wind	111
Simulating flocking behavior	112
Making our particles sound reactive	117
Aligning particles to a processed image	121
Aligning particles to the mesh surface	124
Creating springs	128
Chapter 6: Rendering and Texturing Particle Systems	137
Introduction	137
Texturing particles	137
Adding a tail to our particles	139
Creating a cloth simulation	142
Texturing a cloth simulation	147
Texturing a particle system using point sprites and shaders	149
Connecting the dots	154
Connecting particles with spline	157
Chapter 7: Using 2D Graphics	163
Drawing 2D geometric primitives	163
Drawing arbitrary shapes with the mouse	166
Implementing a scribbler algorithm	169
Implementing 2D metaballs	171
Animating text around curves	174
Adding a blur effect	180
Implementing a force-directed graph	184

Chapter 8: Using 3D Graphics	189
Introduction	189
Drawing 3D geometric primitives	189
Rotating, scaling, and translating	193
Drawing to an offscreen canvas	195
Drawing in 3D with the mouse	198
Adding lights	201
Picking in 3D	205
Creating a height map from an image	210
Creating a terrain with Perlin noise	213
Saving mesh data	217
Chapter 9: Adding Animation	219
Animating with the timeline	219
Creating animation sequences with the timeline	221
Animating along a path	224
Aligning camera motion to a path	226
Animating text – text as a mask for a movie	230
Animating text – scrolling text lines	233
Creating a flow field with Perlin noise	236
Creating an image gallery in 3D	240
Creating a spherical flow field with Perlin noise	245
Chapter 10: Interacting with the User	249
Introduction	249
Creating an interactive object that responds to the mouse	250
Adding mouse events to our interactive object	255
Creating a slider	260
Creating a responsive text box	264
Dragging, scaling, and rotating objects using multi-touch	268
Chapter 11: Sensing and Tracking Input from the Camera	277
Capturing from the camera	277
Tracking an object based on color	279
Tracking motion using optical flow	284
Object tracking	287
Reading QR code	292
Building UI navigation and gesture recognition with Kinect	296
Building an augmented reality with Kinect	304
Chapter 12: Using Audio Input and Output	311
Generating a sine oscillator	311
Generating sound with frequency modulation	314

Table of Contents

Adding a delay effect	317
Generating sound upon the collision of objects	319
Visualizing FFT	323
Making sound-reactive particles	325
Appendix: Integrating with Bullet Physics	
This chapter is available as a downloadable file at: http://www.packtpub.com/sites/default/files/downloads/Integrating_with_Bullet_Physics.pdf	
Index	331

Preface

Cinder is one of the most exciting frameworks available for creative coding. It is developed in C++ for increased performance and allows for the fast creation of visually complex and interactive applications. The big advantage of Cinder is that it can target many platforms such as Mac, Windows, and iOS with the exact same code.

Cinder Creative Coding Cookbook will show you how to develop interactive and visually dynamic applications using simple-to-follow recipes.

You will learn how to use multimedia content, draw generative graphics in 2D and 3D, and animate them in compelling ways.

Beginning with creating simple projects with Cinder, you will use multimedia, create animations, and interact with the user.

From animation with particles to using video, audio, and images, the reader will gain a broad knowledge of creating creative applications using Cinder.

With recipes that include drawing in 3D, image processing, and sensing and tracking in real-time from camera input, this book will teach you how to develop interactive applications that can be run on a desktop computer, mobile device, or be a part of an interactive installation.

This book will give you the necessary knowledge to start creating projects with Cinder that use animations and advanced visuals.

What this book covers

Chapter 1, Getting Started, teaches you the fundamentals of creating applications using Cinder.

Chapter 2, Preparing for Development, introduces several simple recipes that can be very useful during the development process.

Chapter 3, Using Image Processing Techniques, consists of examples of using image processing techniques implemented in Cinder and using third-party libraries.

Chapter 4, Using Multimedia Content, teaches us how to load, manipulate, display, save, and share videos, graphics, and mesh data.

Chapter 5, Building Particle Systems, explains how to create and animate particles using popular and versatile physics algorithms.

Chapter 6, Rendering and Texturing Particle Systems, teaches us how to render and apply textures to our particles in order to make them more appealing.

Chapter 7, Using 2D Graphics, is about how to work and draw with 2D graphics using the OpenGL and built-in Cinder tools.

Chapter 8, Using 3D Graphics, goes through the basics of creating graphics in 3D using OpenGL and some useful wrappers that Cinder includes in some advanced OpenGL features.

Chapter 9, Adding Animation, presents the techniques of animating 2D and 3D objects. We will also introduce Cinder's features in this field such as Timeline and math functions.

Chapter 10, Interacting with the User, creates the graphical objects that react to the user using both mouse and touch interaction. It also teaches us how to create simple graphical interfaces that have their own events for greater flexibility, and integrate with the popular physics library Bullet Physics.

Chapter 11, Sensing and Tracking Input from the Camera, explains how to receive and process data from input devices such as a camera or a Microsoft Kinect sensor.

Chapter 12, Using Audio Input and Output, is about generating sound with the examples, where sound is generated on object's collision in physics simulation. We will present examples of visualizing sound with audio reactive animations.

Appendix, Integrating with Bullet Physics, will help us learn how to integrate Bullet Physics library with Cinder.

This chapter is available as a downloadable file at: http://www.packtpub.com/sites/default/files/downloads/Integrating_with_Bullet_Physics.pdf

What you need for this book

Mac OS X or Windows operating system. Mac users will need XCode, which is available free from Apple and iOS SDK, if they wish to use iOS recipes. Windows users will need Visual C++ 2010. Express Edition is available for free. Windows users will also need Windows Platform SDK installed. While writing this book the latest release of Cinder was 0.8.4.

Who this book is for

This book is for C++ developers who want to start or already began using Cinder for building creative applications. This book is easy to follow for developers who use other creative coding frameworks and want to try Cinder.

The reader is expected to have basic knowledge of C++ programming language.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
gl::setMatricesWindow(getWindowWidth(), getWindowHeight());
gl::color( ColorA(0.f,0.f,0.f, 0.05f) );
gl::drawSolidRect(getWindowBounds());
gl::color( ColorA(1.f,1.f,1.f, 1.f) );
mParticleSystem.draw();
```

Any command-line input or output is written as follows:

```
$ ./fullbuild.sh
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Getting Started

In this chapter we will cover:

- ▶ Creating a project for a basic application
- ▶ Creating a project for a screensaver application
- ▶ Creating a project for an iOS touch application
- ▶ Understanding the basic structure of an application
- ▶ Responding to mouse input
- ▶ Responding to key input
- ▶ Responding to touch input
- ▶ Accessing the files dropped onto the application window
- ▶ Adjusting a scene after resizing the window
- ▶ Using resources on Windows
- ▶ Using resources on OSX and iOS
- ▶ Using assets

Introduction

In this chapter we'll learn the fundamentals of creating applications using Cinder.

We'll start by creating different types of applications on the different platforms that Cinder supports using a powerful tool called TinderoBox.

We'll cover the basic structure of an application and see how to respond to user input events.

Finally, we will learn how to use resources on Windows and Mac.

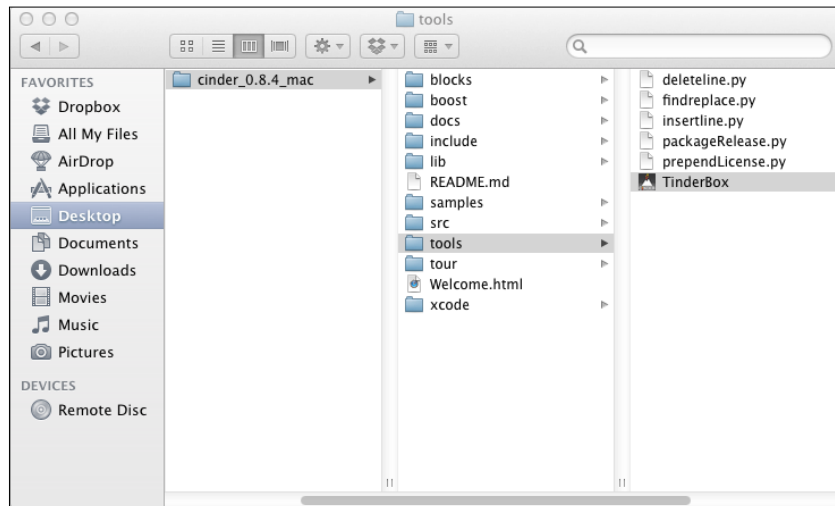
Creating a project for a basic application

In this recipe, we'll learn how to create a project for a basic desktop application for Windows and Mac OSX.

Getting ready

Projects can be created using a powerful tool called TinderBox. TinderBox comes bundled in your Cinder download and contains templates for creating projects for different applications for both Microsoft Visual C++ 2010 and OSX Xcode.

To find Tinderbox, go to your Cinder folder, inside which you will find a folder named `tools` with, TinderBox application in it.

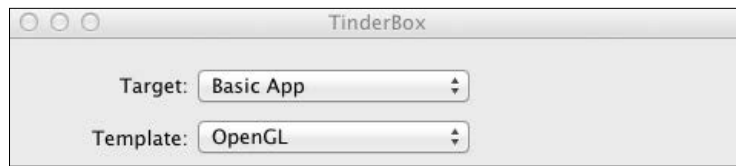


The first time you open TinderBox, you'll be asked to specify the folder where you installed Cinder. You'll need to do this only the first time you open TinderBox. If you need to redefine the location of Cinder installation, you can do so by selecting the **File** menu and then **Preferences** on Windows or selecting the **TinderBox** menu and then **Preferences** on OS X.

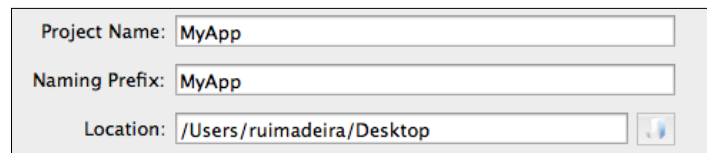
How to do it...

We'll use TinderBox, a utility tool that comes bundled with Cinder that allows for the easy creation of projects. Perform the following steps to create a project for a basic application:

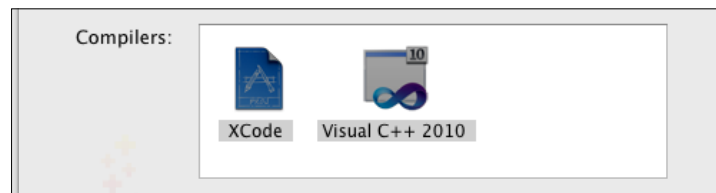
1. Open TinderBox and choose your project's location. In the main **TinderBox** window select **BasicApp** as **Target** and **OpenGL** as **Template**, as shown in the following screenshot:



- Choose your project's location. The **Naming Prefix** and **Project Name** fields will default to the project's name, as shown in the following screenshot:



- Select the compilers you want to use for your project, either Microsoft Visual C++ 2010 and/or OS X Xcode.



- Click on the **Create** button and TinderBox will show you the folder where your new project is located. TinderBox will remain open; you can close it now.

How it works...

TinderBox will create the selected projects for the chosen platforms (Visual C++ 2010 and OS X Xcode) and create references to the compiled Cinder library. It will also create the application's class as a subclass of `ci::app::AppBasic`. It will also create some sample code with a basic example to help you get started.

There's more...

Your project name and naming prefix will be, by default, the name of the folder in which the project is being created. You can edit this if you want, but always make sure both **Project Name** and **Naming Prefix** fields do not have spaces as you might get errors.

The naming prefix will be used to name your application's class by adding the `App` suffix. For example, if you set your **Naming Prefix** field as `MyCinderTest`, your application's class will be `MyCinderTestApp`.

Creating a project for a screensaver application

In this recipe, we will learn how to create a project for a desktop screensaver for both Windows and Mac OS X.

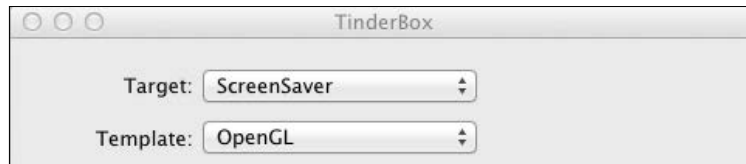
Getting ready

To get ready with TinderBox, please refer to the *Getting ready* section of the previous *Creating a project for a basic application* recipe.

How to do it...

We'll use TinderBox, a utility tool that comes bundled with Cinder that allows easy creation of projects. Perform the following steps to create a project for a screensaver application:

1. Open TinderBox and choose your project's location. In the main **TinderBox** window select **ScreenSaver** as **Target** and **OpenGL** as **Template**, as shown in the following screenshot:



2. Select the compilers you want to create a project to, either Microsoft Visual C++ 2010 and/or OS X Xcode.
3. Click on **Create** and TinderBox will direct you to the folder where your project was created.

How it works...

TinderBox will create both a project for you and link it against the compiled Cinder library. It will also create the application's class and make it a subclass of `ci::app::AppScreenSaver`, which is the class with all the basic functionality for a screensaver application. It will also create some sample code with a basic example to help you get started.

Creating a project for an iOS touch application

In this recipe, we'll learn how to create a project for an application that runs on iOS devices such as iPhone and iPad.

Getting ready

To get ready with TinderBox, please refer to the *Getting ready* section of the *Creating a project for a basic application* recipe.

Please note that the iOS touch application will only work on iOS devices such as iPhones and iPads, and that the projects created with TinderBox will be for OSX Xcode only.

How to do it...

We'll use TinderBox, a utility tool that comes bundled with Cinder that allows easy creation of projects. Perform the following steps to create a project for an iOS touch application:

1. Open TinderBox and choose your project's location. In the main **TinderBox** window select **Cocoa Touch** as **Target** and **Simple** as **Template**, as shown in the following screenshot:



2. Select the compilers you want to create a project to, either Microsoft Visual C++ 2010 and/or OS X Xcode.
3. Click on **Create** and TinderBox will direct you to the folder where your project was created.

How it works...

TinderBox will create an OS X Xcode project and create the references to link against the compiled Cinder library. It will also create the application's class as a subclass of `ci::app::AppCocoaTouch`, which is the class with all the basic functionality for a screensaver application. It will also create some sample code with a basic example to help you get started.

This application is built on top of Apple's Cocoa Touch framework to create iOS applications.

Understanding the basic structure of an application

Your application's class can have several methods that will be called at different points during the execution of the program. The following table lists these methods:

Method	Usage
<code>prepareSettings</code>	This method is called once at the very beginning of the application, before creating the renderer. Here, we may define several parameters of the application before the application gets initialized, such as the frame rate or the size of the window. If none are specified, the application will initialize with default values.
<code>setup</code>	This method is called once at the beginning of the application lifecycle. Here, you initialize all members and prepare the application for running.
<code>update</code>	This method is called in a loop during the application's runtime before the <code>draw</code> method. It is used to animate and update the states of the application's components. Even though you may update them during the <code>draw</code> method, it is recommended you keep the update and drawing routines separate as a matter of organization.
<code>draw</code>	This method is called in a loop during the application's runtime after the update. All drawing code should be placed here.
<code>shutdown</code>	This method is called just before the application exits. Use it to do any necessary cleanup such as freeing memory and allocated resources or shutting down hardware devices.

To execute our code, we must overwrite these methods with our own code.

Getting ready

It is not mandatory to override all of the preceding methods; you can use the ones that your application requires specifically. For example, if you do not want to do any drawing, you may omit the `draw` method.

In this recipe and for the sake of learning, we will implement all of them.

Declare the following methods in your class declaration:

```
Void prepareSettings( Settings *settings );  
Void setup();  
Void update();  
Void draw();  
Void shutdown();
```

How to do it...

We will implement several methods that make up the basic structure of an application. Perform the following steps to do so:

1. Implement the `prepareSettings` method. Here we can define, for example, the size of the window, its title, and the frame rate:

```
void MyApp::prepareSettings( Settings *settings ){  
    settings->setSize( 1024, 768 );  
    settings->setTitle( "My Application Window" );  
    settings->setFrameRate( 60 );  
}
```

2. Implement the `setup` method. Here we should initialize all members of the application's class. For example, to initialize capturing from a webcam we would declare the following members:

```
int mCamWidth;  
int mCamHeight;  
Capture mCapture;  
And initialize them in the setup  
void MyApp::setup(){  
    mCamWidth = 640;  
    mCamHeight = 480;  
    mCapture = Capture( mCamWidth, mCamHeight );  
}
```

3. Implement the `update` method. As an example, we will print the current frame count to the console:

```
void MyApp::update(){  
    console() << geElapsedFrames() << std::endl;  
}
```


4. Implement the `draw` method with all the drawing commands. Here we clear the background with black and draw a red circle:

```
void MyApp::draw() {
    gl::clear( Color::black() );
    gl::color( Color( 1.0f, 0.0f, 0.0f ) );
    gl::drawSolidCircle( Vec2f( 300.0f, 300.0f ), 100.0f );
}
```

5. Implement the `shutdown` method. This method should take code for doing cleanup, for example, to shut down threads or save the state of your application.
6. Here's a sample code for saving some parameters in an XML format:

```
void MyApp::shutdown() {
    XmlTree doc = XmlTree::createDoc();
    XmlTree settings = xmlTree( "Settings", "" );
    //add some attributes to the settings node
    doc.push_back( settings );
    doc.write( writeFile( "Settings.xml" ) );
}
```

How it works...

Our application's superclass implements the preceding methods as virtual empty methods.

When the application runs, these methods are called, calling our own code we implemented or the parent class' empty method if we didn't.

In step 1 we defined several application parameters in the `prepareSettings` method. It is not recommended to use the `setup` method to initialize these parameters, as it means that the renderer has to be initialized with the default values and then readjusted during the setup. The result is extra initialization time.

There's more...

There are other callbacks that respond to user input such as mouse and keyboard events, resizing of the window, and dragging files onto the application window. These are described in more detail in the *Responding to mouse input*, *Responding to key input*, *Responding to touch input*, *Accessing files dragged on the application window*, and *Adjusting a scene after resizing the window* recipes.

See also

To learn how to create a basic app with `TinderBox`, read the *Creating a project for a basic application* recipe.

Responding to mouse input

An application can respond to mouse interaction through several event handlers that are called depending on the action being performed.

The existing handlers that respond to mouse interaction are listed in the following table:

Method	Usage
<code>mouseDown</code>	This is called when the user presses a mouse button
<code>mouseUp</code>	This is called when the user releases a mouse button
<code>mouseWheel</code>	This is called when the user rotates the mouse wheel
<code>mouseMove</code>	This is called when the mouse is moved without any button pressed
<code>mouseDrag</code>	This is called when the mouse is moved with any button pressed

It is not mandatory to implement all of the preceding methods; you can implement only the ones required by your application.

Getting ready

Implement the necessary event handlers according to the mouse events you need to respond to. For example, to create an application that responds to all available mouse events, you must implement the following code inside your main class declaration:

```
void mouseDown( MouseEvent event );
void mouseUp( MouseEvent event );
void mouseWheel( MouseEvent event );
void mouseMove( MouseEvent event );
void mouseDrag( MouseEvent event );
```

The `MouseEvent` object passed as a parameter contains information about the mouse event.

How to do it...

We will learn how to work with the `ci::app::MouseEvent` class to respond to mouse events. Perform the following steps to do so:

1. To get the position where the event has happened, in terms of screen coordinates, we can type in the following line of code:

```
Vec2i mousePos = event.getPos();
```

Or we can get the separate x and y coordinates by calling the `getX` and `getY` methods:

```
int mouseX = event.getX();
int mouseY = event.getY();
```

2. The `MouseEvent` object also lets us know which mouse button triggered the event by calling the `isLeft`, `isMiddle`, or `isRight` methods. They return a `bool` value indicating if it was the left, middle, or right button, respectively.

```
bool leftButton = event.isLeft();
bool rightButton = event.isRight();
bool middleButton = event.isMiddle();
```

3. To know if the event was triggered by pressing a mouse button, we can call the `isLeftDown`, `isRightDown`, and `isMiddleDown` methods that return `true` depending on whether the left, right, or middle buttons of the mouse were pressed.

```
bool leftDown = event.isLeftDown();
bool rightDown = event.isRightDown();
bool middleDown = event.isMiddleDown();
```

4. The `getWheelIncrement` method returns a `float` value with the movement increment of the mouse wheel.

```
float wheelIncrement = event.getWheelIncrement();
```

5. It is also possible to know if a special key was being pressed during the event. The `isShiftDown` method returns `true` if the *Shift* key was pressed, the `isAltDown` method returns `true` if the *Alt* key was pressed, `isControlDown` returns `true` if the *control* key was pressed, and `isMetaDown` returns `true` if the Windows key was pressed on Windows or the *option* key was pressed on OS X, `isAccelDown` returns `true` if the *Ctrl* key was pressed on Windows or the *command* key was pressed on OS X.

How it works

A Cinder application responds internally to the system's native mouse events. It then creates a `ci::app::MouseEvent` object using the native information and calls the necessary mouse event handlers of our application's class.

There's more...

It is also possible to access the native modifier mask by calling the `getNativeModifiers` method. These are platform-specific values that Cinder uses internally and may be of use for advanced uses.

Responding to key input

A Cinder application can respond to key events through several callbacks.

The available callbacks that get called by keyboard interaction are listed in the following table:

Method	Usage
<code>keyDown</code>	This is called when the user first presses a key and called repeatedly if a key is kept pressed.
<code>keyUp</code>	This is called when a key is released.

Both these methods receive a `ci::app::KeyEvent` object as a parameter with information about the event such as the key code being pressed or if any special key (such as *Shift* or *control*) is being pressed.

It is not mandatory to implement all of the preceding key event handlers; you can implement only the ones that your application requires.

Getting ready

Implement the necessary event handlers according to what key events you need to respond to. For example, to create an application that responds to both key down and key up events, you must declare the following methods:

```
void keyDown( KeyEvent event );
void keyUp( KeyEvent event );
```

The `ci::app::KeyEvent` parameter contains information about the key event.

How to do it...

We will learn how to work with the `ci::app::KeyEvent` class to learn how to understand key events. Perform the following steps to do so:

1. To get the ASCII code of the character that triggered the key event, you can type in the following line of code:

```
char character = event.getChar();
```

2. To respond to special keys that do not map to the ASCII character table, we must call the `getCode` method that retrieves an `int` value that can be mapped to a character table in the `ci::app::KeyEvent` class. To test, for example, if the key event was triggered by the *Esc* key you can type in the following line of code:

```
bool escPressed = event.getCode() == KeyEvent::KEY_ESCAPE;
escPressed will be true if the escape key triggered the event, or false otherwise.
```

- The `ci::app::KeyEvent` parameter also has information about modifier keys that were pressed during the event. The `isShiftDown` method returns `true` if the *Shift* key was pressed, `isAltDown` returns `true` if the *Alt* key was pressed, `isControlDown` returns `true` if the *control* key was pressed, `isMetaDown` returns `true` if the Windows key was pressed on Windows or the *command* key was pressed on OS X, and `isAccelDown` returns `true` if the *Ctrl* key was pressed on Windows or the *command* key was pressed on OS X.

How it works...

A Cinder application responds internally to the system's native key events. When receiving a native key event, it creates a `ci::app::KeyEvent` object based on the native information and calls the correspondent callback on our application's class.

There's more...

It is also possible to access the native key code by calling the `getNativeKeyCode` method. This method returns an `int` value with the native, platform-specific code of the key. It can be important for advanced uses.

Responding to touch input

A Cinder application can receive several touch events.

The available touch event handlers that get called by touch interaction are listed in the following table:

Method	Usage
<code>touchesBegan</code>	This is called when new touches are detected
<code>touchesMoved</code>	This is called when existing touches move
<code>touchesEnded</code>	This is called when existing touches are removed

All of the preceding methods receive a `ci::app::TouchEvent` object as a parameter with a `std::vector` of `ci::app::TouchEvent::Touch` objects with information about each touch detected. Since many devices can detect and respond to several touches simultaneously, it is possible and common for a touch event to contain several touches.

It is not mandatory to implement all of the preceding event handlers; you can use the ones your application requires specifically.

Cinder applications can respond to touch events on any touch-enabled device running Windows 7, OS X, or iOS.

Getting ready

Implement the necessary touch event handlers according to the touch events you want to respond to. For example, to respond to all available touch events (touches added, touches moved, and touches removed), you would need to declare and implement the following methods:

```
void touchesBegan( TouchEvent event );
void touchesMoved( TouchEvent event );
void touchesEnded( TouchEvent event );
```

How to do it...

We will learn how to work with the `ci::app::TouchEvent` class to understand touch events. Perform the following steps to do so:

1. To access the list of touches, you can type in the following line of code:

```
const std::vector<TouchEvent::Touch>& touches = event.
getTouches();
```

Iterate through the container to access each individual element.

```
for( std::vector<TouchEvent::Touch>::const_iterator it = touches.
begin(); it != touches.end(); ++it ){
    const TouchEvent::Touch& touch = *it;
    //do something with the touch object
}
```

2. You can get the position of a touch by calling the `getPos` method that returns a `Vec2f` value with its position or using the `getX` and `getY` methods to receive the x and y coordinates separately, for example:

```
for( std::vector<TouchEvent::Touch>::const_iterator it = touches.
begin(); it != touches.end(); ++it ){
    const TouchEvent::Touch& touch = *it;
    vec2f pos = touch.getPos();
    float x = touch.getX();
    float y = touch.getY();
}
```

3. The `getId` method returns a `uint32_t` value with a unique ID for the touch object. This ID is persistent throughout the lifecycle of the touch, which means you can use it to keep track of a specific touch as you access it on the different touch events.

For example, to make an application where we draw lines using our fingers, we can create `std::map` that associates each line, in the form of a `ci::PolyLine<Vec2f>` object, with a `uint32_t` key with the unique ID of a touch.

We need to include the file with `std::map` and `PolyLine` to our project by adding the following code snippet to the beginning of the source file:

```
#include "cinder/polyline.h"
#include <map>
```

4. We can now declare the container:

```
std::map< uint32_t, PolyLine<Vec2f> > mLines;
```

5. In the `touchesBegan` method we create a new line for each detected touch and map it to the unique ID of each touch:

```
const std::vector<TouchEvent::Touch>& touches = event.
getTouches();
for( std::vector<TouchEvent::Touch>::const_iterator it = touches.
begin(); it != touches.end(); ++it ){
    const TouchEvent::Touch& touch = *it;
    mLines[ touch.getId() ] = PolyLine<Vec2f>();
}
```

6. In the `touchesMoved` method, we add the position of each touch to its corresponding line:

```
const std::vector<TouchEvent::Touch>& touches = event.
getTouches();
for( std::vector<TouchEvent::Touch>::const_iterator it = touches.
begin(); it != touches.end(); ++it ){
    const TouchEvent::Touch& touch = *it;
    mLines[ touch.getId() ].push_back( touch.getPos() );
}
```

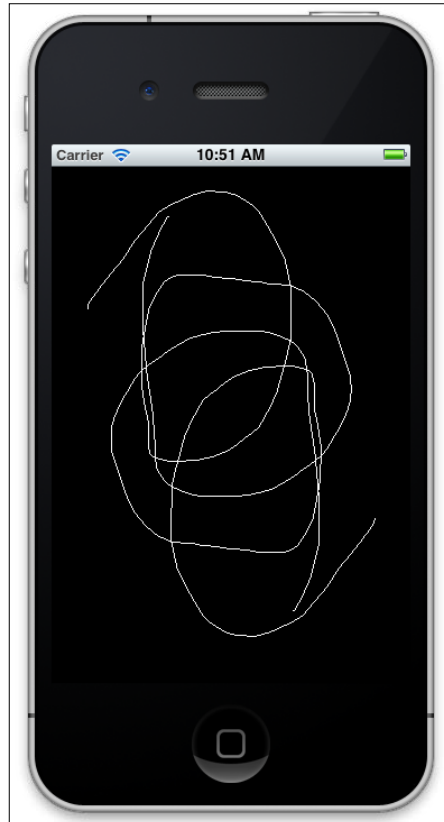
7. In the `touchesEnded` method, we remove the line that corresponds to a touch being removed:

```
const std::vector<TouchEvent::Touch>& touches = event.
getTouches();
for( std::vector<TouchEvent::Touch>::const_iterator it = touches.
begin(); it != touches.end(); ++it ){
    const TouchEvent::Touch& touch = *it;
    mLines.erase( touch.getId() );
}
```

8. Finally, the lines can be drawn. Here we clear the background with black and draw the lines with in white. The following is the implementation of the `draw` method:

```
gl::clear( Color::black() );
gl::color( Color::white() );
for( std::map<uint32_t, PolyLine<Vec2f> >::iterator it = mLines.
begin(); it != mLines.end(); ++it ){
    gl::draw( it->second );
}
```

The following is a screenshot of our app running after drawing some lines:



How it works...

A Cinder application responds internally to the system calls for any touch event. It will then create a `ci::app::TouchEvent` object with information about the event and call the corresponding event handler in our application's class. The way to respond to touch events becomes uniform across the Windows and Mac platforms.

The `ci::app::TouchEvent` class contains only one accessor method that returns a `const` reference to a `std::vector<TouchEvent::Touch>` container. This container has one `ci::app::TouchEvent::Touch` object for each detected touch and contains information about the touch.

The `ci::app::TouchEvent::Touch` object contains information about the touch including position and previous position, unique ID, the time stamp, and a pointer to the native event object which maps to `UITouch` on Cocoa Touch and `TOUCHPOINT` on Windows 7.

There's more...

At any time, it is also possible to get a container with all active touches by calling the `getActiveTouches` method. It returns a `const` reference to a `std::vector<TouchEvent::Touch>` container. It offers flexibility when working with touch applications as it can be accessed outside the touch event methods.

For example, if you want to draw a solid red circle around each active touch, you can add the following code snippet to your `draw` method:

```
const std::vector<TouchEvent::Touch>&activeTouches =
getActiveTouches();
gl::color( Color( 1.0f, 0.0f, 0.0f ) );
for( std::vector<TouchEvent::Touch>::const_iterator it =
activeTouches.begin(); it != activeTouches.end(); ++it ){
    const TouchEvent::Touch& touch = *it;
    gl::drawSolidCircle( touch.getPos(), 10.0f );
}
```

Accessing files dropped onto the application window

Cinder applications can respond to files dropped onto the application window through the callback, `fileDrop`. This method takes a `ci::app::FileDropEvent` object as a parameter with information about the event.

Getting ready

Your application must implement a `fileDrop` method which takes a `ci::app::FileDropEvent` object as a parameter.

Add the following method to the application's class declaration:

```
void fileDrop( FileDropEvent event );
```

How to do it...

We will learn how to work with the `ci::app::FileDropEvent` object to work with file drop events. Perform the following steps to do so:

1. In the method implementation you can use the `ci::app::FileDropEvent` parameter to access the list of files dropped onto the application by calling the `getFiles` method. This method returns a `const std::vector` container with `fs::path` objects:

```
const vector<fs::path >& files = event.getFiles();
```

2. The position where the files were dropped onto the window can be accessed through the following callback methods:

- To get a `ci::Vec2i` object with the position of the files dropped, type in the following line of code:

```
Vec2i dropPosition = event.getPos();
```

- To get the x and y coordinates separately, you can use the `getX` and `getY` methods, for example:

```
int posX = event.getX();
```

```
int posY = event.getY();
```

3. You can find the number of dropped files by using the `getNumFiles` method:

```
int numFiles = event.getNumFiles();
```

4. To access a specific file, if you already know its index, you can use the `getFile` method and pass the index as a parameter.

For example, to access the file with an index of 2, you can use the following line of code:

```
const fs::path& file = event.getFile( 2 );
```

How it works...

A Cinder application will respond to the system's native event for file drops. It will then create a `ci::app::FileDropEvent` object with information about the event and call the `fileDrop` callback in our application. This way Cinder creates a uniform way of responding to file drop events across the Windows and OS X platforms.

There's more...

Cinder uses `ci::fs::path` objects to define paths. These are `typedef` instances of `boost::filesystem::path` objects and allow for much greater flexibility when working with paths. To learn more about the `fs::path` objects, please refer to the `boost::filesystem` library reference, available at http://www.boost.org/doc/libs/1_50_0/libs/filesystem/doc/index.htm.

Adjusting a scene after resizing the window

Cinder applications can respond to resizing the window by implementing the `resize` event. This method takes a `ci::app::ResizeEvent` parameter with information about the event.

Getting ready

If your application doesn't have a `resize` method, implement one. In the application's class declaration, add the following line of code:

```
void resize( ResizeEvent event );
```

In the method's implementation, you can use the `ResizeEvent` parameter to find information about the window's new size and format.

How to do it...

We will learn how to work with the `ci::app::ResizeEvent` parameter to respond to window resize events. Perform the following steps to do so:

1. To find the new size of the window, you can use the `getSize` method which returns a `ci::Vec2i` object, the window's width as the x component, and the height as the y component.

```
Vec2i windowSize = event.getSize();
```

The `getWidth` and `getHeight` methods both return `int` values with the window's width and height respectively, for example:

```
int width = event.getWidth();  
int height = event.getHeight();
```

2. The `getAspectRatio` method returns a `float` value with the aspect ratio of the window, which is the ratio between its width and height:

```
float ratio = event.getAspectRatio();
```

- Any element on screen that needs adjusting must use the new window size to recalculate its properties. For example, to have a rectangle that is drawn at the center of the window with a 20 pixel margin on all sides, we must first declare a `ci::Rectf` object in the class declaration:

```
Rect frect;
```

In the setup we set its properties so that it has a 20 pixel margin on all sides from the window:

```
rect.x1 = 20.0f;  
rect.y1 = 20.0f;  
rect.x2 = getWindowWidth() - 20.0f;  
rect.y2 = getWindowHeight() - 20.0f;
```

- To draw the rectangle with a red color, add the following code snippet to the draw method:

```
gl::color( Color( 1.0f, 0.0f, 0.0f ) );  
gl::drawSolidRect( rect );
```

- In the `resize` method, we must recalculate the rectangle properties so that it resizes itself to maintain the 20 pixel margin on all sides of the window:

```
rect.x1 = 20.0f;  
rect.y1 = 20.0f;  
rect.x2 = event.getWidth() - 20.0f;  
rect.y2 = event.getHeight() - 20.0f;
```

- Run the application and resize the window. The rectangle will maintain its relative size and position according to the window size.



How it works...

A Cinder application responds internally to the system's window resize events. It will then create the `ci::app::ResizeEvent` object and call the `resize` method on our application's class. This way Cinder creates a uniform way of dealing with resize events across the Windows and Mac platforms.

Using resources on Windows

It is common for Windows applications to use external files either to load images, play audio or video, or to load or save settings on XML files.

Resources are external files to your application that are embedded in the application's executable file. Resource files are hidden from the user to avoid alterations.

Getting ready

Resources should be stored in a folder named `resources` in your project folder. If this folder does not exist, create it.

Resources on Windows must be referenced in a file called `Resources.rc`. This file should be placed next to the Visual C++ solution in the `vc10` folder. If this file does not exist, you must create it as an empty file. If the `resources.rs` file is not included already in your project solution, you must add it by right-clicking on the **Resources** filter and choosing **Add** and then **ExistingItem**. Navigate to the file and select it. As a convention, this file should be kept in the same folder as the project solution.

How to do it...

We will use Visual C++ 2010 to add resources to our applications on Windows. Perform the following steps to do so:

1. Open the Visual C++ solution and open the `resources.h` file inside the **Header Files** filter.
2. Add the `#pragma once` macro to your file to prevent it from being included more than once in your project and include the `CinderResources.h` file.

```
#pragma once
#include "cinder/CinderResources.h"
```

3. On Windows, each resource must have a unique ID number. As a convention, the IDs are defined as sequential numbers starting from 128, but you can use other IDs if it suits you better. Make sure to never use the same ID twice. You must also define a type string. The type string is used to identify resources of the same type, for example, the string `IMAGE` may be used when declaring image resources, `VIDEO` for declaring video resources, and so on.
4. To simplify writing multiplatform code, Cinder has a macro for declaring resources that can be used on both Windows and Mac.

For example, to declare the resource of an image file named `image.png`, we would type in the following line of code:

```
#define RES_IMAGE CINDER_RESOURCE(../resources/, image.png, 128, IMAGE)
```

The first parameter of the `CINDER_RESOURCE` macro is the relative path to the folder where the resource file is, in this case the default `resources` folder.

The second parameter is the name of the file, and after that comes the unique ID of this resource, and finally its type string.

5. Now we need to add our `resources` macro to the `resources.rs` file, as follows:

```
#include "../include/Resources.h"  
RES_IMAGE
```

6. This resource is now ready to be used in our application. To load this image into `ci::gl::Texture` we simply include the `Texture.h` file in our application's source code:

```
#include "cinder/gl/Texture.h"
```

7. We can now declare the texture:

```
gl::Texture mImage;
```

8. In the setup, we create the texture by loading the resource:

```
mImage = gl::Texture( loadImage( loadResource( RES_IMAGE ) ) );
```

9. The texture is now ready to be drawn on screen. To draw the image at position (20, 20), we will type in the following line of code inside the `draw` method:

```
gl::draw( mImage, Vec2f( 20.0f, 20.0f ) );
```

How it works...

The `resources.rc` file is used by a resource compiler to embed resources into the executable file as binary data.

There's more...

Cinder allows writing code to use resources that is coherent across all supported platforms, but the way resources are handled on Windows and OS X/iOS is slightly different. To learn how to use resources on a Mac, please read the *Using resources on iOS and OS X* recipe.

Using resources on iOS and OS X

It is common for Windows applications to use external files either to load images, play audio or video, or to load or save settings on XML files.

Resources are external files to your application that are included in the applications bundle. Resource files are hidden from the user to avoid alterations.

Cinder allows writing code to use resources that is equal when writing Windows or Mac applications, but the way resources are handled is slightly different. To learn how to use resources on Windows, please read the *Using resources on Windows* recipe.

Getting ready

Resources should be stored in a folder named `resources` in your `project` folder. If this folder does not exist, create it.

How to do it...

We will use Xcode to add resources to our application on iOS and OS X. Perform the following steps to do so:

1. Place any resource file you wish to use in the `resources` folder.
2. Add these files to your project by right-clicking on the **Resources** filter in your Xcode project and selecting **Add** and then **ExistingFiles**, navigate to the `resources` folder, and select the resource files you wish to add.
3. To load a resource in your code, you use the `loadResource` method and pass the name of the resource file. For example, to load an image named `image.png`, you should first create the `gl::Texture` member in the class declaration:

```
gl::Texture mImage;
```

4. In the `setup` method, we initialize the texture with the following resource:

```
mImage = loadImage( loadImage( "image.png" ) );
```

5. The texture is now ready to be drawn in the window. To draw it at position (20, 20), type in the following line of code inside the `draw` method:

```
gl::draw( mImage, Vec2f( 20.0f, 20.0f ) );
```

How it works...

On iOS and OS X, applications are actually folders that contain all the necessary files to run the application, such as the Unix executable file, the frameworks used, and the resources. You can access the content of these folders by clicking on any Mac application and selecting **Show Package Contents**.

When you add resources to the `resources` folder in your Xcode project, these files are copied during the build stage to the `resources` folder of your application bundle.

There's more...

You can also load resources using the same `loadResource` method that is used in Windows applications. This is very useful when writing cross-platform applications so that no changes are necessary in your code.

You should create the `resource` macro in the `Resources.h` file, and add the unique resource ID and its type string. For example, to load the image `image.png`, you can type in the following code snippet:

```
#pragma once
#include "cinder/CinderResources.h"
#define RES_IMAGE CINDER_RESOURCE(../resources/, image.png, 128,
IMAGE)
```

And this is what the `Resources.rc` file should look like:

```
#include "..\include\Resources.h"

RES_IMAGE
```

Using the preceding example to load an image, the only difference is that we would load the texture with the following line of code:

```
mImage = loadImage( loadImage( RES_IMAGE ) );
```

The resource unique ID and type string will be ignored in Mac applications, but adding them allows creating code that is cross-platform.

Using assets

In this recipe, we will learn how we can load and use assets.

Getting ready

As an example for this recipe, we will load and display an asset image.

Place an image file inside the `assets` folder in your project directory and name it `image.png`.

Include the following files at the top of your source code:

```
#include "cinder/gl/Texture.h"
#include "cinder/ImageIO.h"
```

Also add the following useful `using` statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

As an example, we will learn how we can load and display an image asset. Perform the following steps to do so:

1. Declare a `ci::gl::Texture` object:
`gl::Texture image;`
2. In the `setup` method let's load the image asset. We will use a `try/catch` block in if it is not possible to load the asset.

```
try{
    image = loadImage( loadAsset( "image.png" ) );
} catch( ... ){
    console() << "asset not found" << endl;
}
```

3. In the `draw` method we will draw the texture. We will use an `if` statement to check if the texture has been successfully initialized:

```
if( image ){
    gl::draw( image, getWindowBounds() );
}
```

How it works...

The first application uses an asset Cinder, which will try to find its default `assets` folder. It will begin by searching the executable or application bundle folder, depending on the platform, and continue searching its parent's folder up to five levels. This is done to accommodate for different project setups.

There's more...

You can add an additional `assets` folder using the `addAssetDirectory` method, which takes a `ci::fs::path` object as a parameter. Every time Cinder searches for an asset, it will first look in its default `asset` folder and then in every folder the user may have added.

You can also create subfolders inside the `assets` folder, for example, if our image was inside a subfolder named `My Images`, we would type in the following code snippet in the `setup` method:

```
try{
    image = loadImage( loadAsset( "My Images/image.png" ) );
}catch( ... ){
    console() << "asset not found" << endl;
}
```

It is also possible to know the path where a specific folder lies. To do this, use the `getAssetPath` method, which takes a `ci::fs::path` object as a parameter with the name of the file.

2

Preparing for Development

In this chapter, we will cover:

- ▶ Setting up a GUI for tweaking parameters
- ▶ Saving and loading configurations
- ▶ Making a snapshot of the current parameter state
- ▶ Using MayaCamUI
- ▶ Using 3D space guides
- ▶ Communicating with other software
- ▶ Preparing your application for iOS

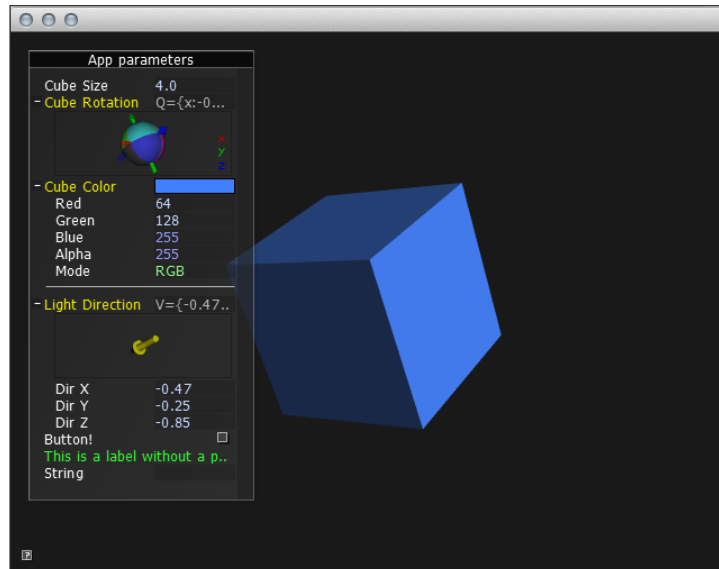
Introduction

In this chapter, we will introduce several simple recipes that can be very useful during the development process.

Setting up a GUI for tweaking parameters

Graphical User Interface (GUI) is often required for controlling and tuning your Cinder application. In many cases, you spend more time tweaking the application parameters to achieve the desired result than writing the code. It is true especially when you are working on some generative graphics.

Cinder provides a convenient and easy-to-use GUI via the `InterfaceGl` class.



Getting ready

To make the `InterfaceGl` class available in your Cinder application, all you have to do is include one header file.

```
#include "cinder/params/Params.h"
```

How to do it...

Follow the steps given here to add a GUI to your Cinder application.

1. Let's start with preparing different types of variables within our main class, which we will be manipulating using the GUI.

```
float mObjSize;  
Quatf mObjOrientation;  
Vec3f mLightDirection;  
ColorA mColor;
```

2. Next, declare the `InterfaceGl` class member like this:

```
params::InterfaceGl mParams;
```

3. Now we move to the `setup` method and initialize our GUI window passing "Parameters" as the window caption and size to the `InterfaceGl` constructor:



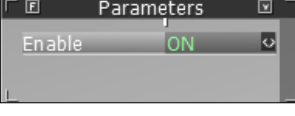

```
mParams = params::InterfaceGl("Parameters", Vec2i(200,400));
```

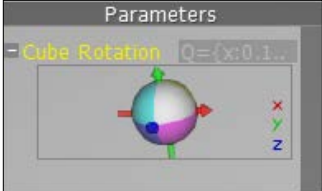
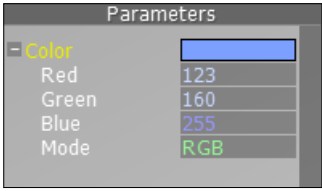

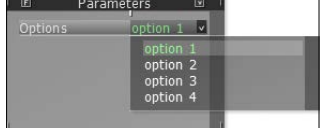
4. And now we can add and configure controls for our variables:

```
mParams.addParam( "Cube Size", &mObjSize, "min=0.1 max=20.5
step=0.5 keyIncr=z keyDecr=Z" );
mParams.addParam( "Cube Rotation", &mObjOrientation ); // Quatf
type
mParams.addParam( "Cube Color", &mColor, "" ); // ColorA
mParams.addSeparator(); // add horizontal line separating controls
mParams.addParam( "Light Direction", &mLightDirection, "" ); //
Vec3f
mParams.addParam( "String ", &mString, "" ); // string
```

Take a look at the `addParam` method and its parameters. The first parameter is just the field caption. The second parameter is a pointer to the variable where the value is stored. There are a bunch of supported variable types, such as `bool`, `float`, `double`, `int`, `Vec3f`, `Quatf`, `Color`, `ColorA`, and `std::string`.

The possible variables types and their interface representations are tabulated in the following table:

Type	Representation
<code>std::string</code>	
Numerical: <code>int</code> , <code>float</code> , <code>double</code>	
<code>bool</code>	
<code>ci::Vec3f</code>	

Type	Representation
<code>ci::Quatf</code>	
<code>ci::Color</code>	
<code>ci::ColorA</code>	
Enumerated parameter	

The third parameter defines the control options. In the following table, you can find some commonly used options and their short explanations:

Name	Explanation
<code>min</code>	The minimum possible value of a numeric variable
<code>max</code>	The maximum possible value of a numeric variable
<code>step</code>	Defines the number of significant digits printed after the period for floating point variables
<code>key</code>	Keyboard shortcut for calling button callback
<code>keyIncr</code>	Keyboard shortcut for incrementing the value

Name	Explanation
<code>keyDecr</code>	Keyboard shortcut for decrementing the value
<code>readonly</code>	Setting the value to <code>true</code> makes a variable read-only in GUI
<code>precision</code>	Defines the number of significant digits printed after the period for floating point variables



You can find the complete documentation of the available options on the *AntTweakBar* page at the following address: <http://anttweakbar.sourceforge.net/doc/tools:anttweakbar:varparamsyntax>.

- The last thing to do is invoke the `InterfaceGl::draw()` method. We will do this at the end of the `draw` method in our main class by typing the following code line:

```
params::InterfaceGl::draw();
```

How it works...

In the `setup` method we will set up the GUI window and then add controls, setting up a name in the first parameter of the `addParam` method. In a second parameter, we are pointing to the variable we want to link the GUI element to. Whenever we change values through the GUI, the linked variable will be updated.

There's more...

There are a few more options for `InterfaceGl`, if you need more control over built-in GUI mechanism, please refer to the *AntTweakBar* documentation which you can find on the project page mentioned in the *See also* section of this recipe.

Buttons

You can also add buttons to the `InterfaceGl` (CIT) panel with callbacks to some functions. For example:

```
mParams.addButton("Start", std::bind(&MainApp::start, this));
```

Clicking on the **Start** button in the GUI fires the `start` method of the `MainApp` class.

Panel position

A convenient way to control the position of the GUI panel is through the usage of the *AntTweakBar* facility. You have to include an additional header file:

```
#include "AntTweakBar.h"
```

And now you can change the position of the GUI panel with this code line:

```
TwDefine("Parameters position='100 200' ");
```

In this case, *Parameters* is the GUI panel name and the *position* option takes *x* and *y* as values.

See also

There are some good looking GUI libraries available as CinderBlocks. Cinder has an extensions system called blocks. The idea behind CinderBlocks is to provide easy-to-use integration with many third-party libraries. You can find how to add examples of CinderBlocks to your project in the *Communicating with other software* recipe.

SimpleGUI

An alternative GUI developed by *Marcin Ignac* as a CinderBlock can be found at <https://github.com/vorg/MowaLibs/tree/master/SimpleGUI>.

ciUI

You can check out an alternative user interface developed by *Reza Ali* as a CinderBlock at <http://www.syedrezaali.com/blog/?p=2366>.

AntTweakBar

InterfaceGl in Cinder is built on top of *AntTweakBar*; you can find its documentation at <http://www.antisphere.com/Wiki/tools:anttweakbar>.

Saving and loading configurations

Many applications that you will develop operate on input parameters set by the user. For example, it could be the color or position of some graphical elements or parameters used to set up communication with other applications. Reading configurations from external files is necessary for your applications. We will use a built-in Cinder support for reading and writing XML files to implement the configuration persistence mechanism.

Getting ready

Create two configurable variables in the main class: the IP address and the port of the host we are communicating with.

```
string mHostIP;
int mHostPort;
```

How to do it...

Now we will implement the `loadConfig` and `saveConfig` methods and use them to load the configuration on application startup and save the changes while closing.

1. Include the two following additional headers:

```
#include "cinder/Utilities.h"
#include "cinder/Xml.h"
```

2. We will prepare two methods for loading and saving the XML configuration file.

```
void MainApp::loadConfig()
{
    try {
        XmlTree doc( loadFile( getAppPath() / fs::path("config.xml") )
);
        XmlTree &generalNode = doc.getChild( "general" );

        mHostIP = generalNode.getChild("hostIP").getValue();
        mHostPort = generalNode.getChild("hostPort").getValue<int>();

    } catch(Exception e) {
        console() << "ERROR: loading/reading configuration file." <<
endl;
    }
}

void MainApp::saveConfig()
{
    std::string beginXmlStr( "<?xml version=\"1.0\"
encoding=\"UTF-8\" ?>" );
    XmlTree doc( beginXmlStr );

    XmlTree generalNode;
    generalNode.setTag("general");
    generalNode.push_back( XmlTree("hostIP", mHostIP) );
    generalNode.push_back( XmlTree("hostPort", toString(mHostPort))
);
}
```

```
doc.push_back (generalNode) ;

doc.write(writeFile( getAppPath() / fs::path("config.xml")) );
}
```

3. Now in the `setup` method, inside our main class, we will put:

```
// setup default values
mHostIP = "127.0.0.1";
mHostPort = 1234;

loadConfig();
```

4. After this we will implement the `shutdown` method as follows:

```
void MainApp::shutdown()
{
    saveConfig();
}
```

5. And don't forget to declare the `shutdown` method in the main class:

```
void shutdown();
```

How it works...

The first two methods, `loadConfig` and `saveConfig`, are essential. The `loadConfig` method tries to open the `config.xml` file and find the `general` node. Inside the `general` node should be the `hostIP` and `hostPort` nodes. The values of these nodes will be assigned to corresponding variables in our application: `mHostIP` and `mHostPort`.

The `shutdown` method is automatically triggered by Cinder just before the application closes, so our configuration values will be stored in the XML file when we quit the application. Finally, our configuration XML file looks like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<general>
<hostIP>127.0.0.1</hostIP>
<hostPort>1234</hostPort>
</general>
```

You can see clearly that the nodes are referring to application variables.

See also

You can write your own configuration loader and saver or use the existing CinderBlock.

Cinder-Config

Cinder-Config is a small CinderBlock for creating configuration files along with `InterfaceGl`.

<https://github.com/dawidgorny/Cinder-Config>

Making a snapshot of the current parameter state

We will implement a simple but useful mechanism for saving and loading the parameters' states. The code used in the examples will be based on the previous recipes.

Getting ready

Let's say we have a variable that we are changing frequently. In this case, it will be the color of some element we are drawing and the main class will have the following member variable:

```
ColorA mColor;
```

How to do it...

We will use a built-in XML parser and the `fileDrop` event handler.

1. We have to include the following additional headers:

```
#include "cinder/params/Params.h"
#include "cinder/ImageIo.h"
#include "cinder/Utilities.h"
#include "cinder/Xml.h"
```

2. First, we implement two methods for loading and saving parameters:

```
void MainApp::loadParameters(std::string filename)
{
    try {
        XmlTree doc( loadFile( fs::path(filename) ) );
        XmlTree &generalNode = doc.getChild( "general" );

        mColor.r = generalNode.getChild("ColorR").
            getValue<float>();
    }
```

```
        mColor.g = generalNode.getChild("ColorG").
getValue<float>();
        mColor.b = generalNode.getChild("ColorB").
getValue<float>();

    } catch(XmlTree::Exception e) {
        console() << "ERROR: loading/reading configuration file." <<
e.what() << std::endl;
    }
}

void MainApp::saveParameters(std::string filename)
{
    std::string beginXmlStr( "<?xml version=\"1.0\"
encoding=\"UTF-8\" ?>" );
    XmlTree doc( beginXmlStr );

    XmlTree generalNode;
    generalNode.setTag("general");
    generalNode.push_back(XmlTree("ColorR", toString(mColor.r)));
    generalNode.push_back(XmlTree("ColorG", toString(mColor.g)));
    generalNode.push_back(XmlTree("ColorB", toString(mColor.b)));

    doc.push_back(generalNode);

    doc.write( writeFile( getAppPath() / fs::path("../") /
fs::path(filename) ) );
}
```

3. Now we declare a class member. It will be the flag to trigger snapshot creation:

```
bool mMakeSnapshot;
```

4. Assign a value to it value inside the setup method:

```
mMakeSnapshot = false;
```

5. At the end of the draw method we put the following code, just before the `params::InterfaceGl::draw();` line:

```
if(mMakeSnapshot) {
    mMakeSnapshot = false;

    double timestamp = getElapsedSeconds();
    std::string timestampStr = toString(timestamp);

    writeImage(getAppPath() / fs::path("../") / fs::path("snapshot_"
+ timestampStr + ".png"), copyWindowSurface());
    saveParameters("snapshot_" + timestampStr + ".xml");
}
```

6. We want to make a button in our `InterfaceGl` window:

```
mParams.addButton( "Make snapshot", std::bind(
&MainApp::makeSnapshotClick, this ) );
```

As you can see we don't have the `makeSnapshotClick` method yet. It is simple to implement:

```
void MainApp::makeSnapshotClick()
{
    mMakeSnapshot = true;
}
```

7. The last step will be adding the following method for *drag-and-drop* support:

```
void MainApp::fileDrop( FileDropEvent event )
{
    std::string filepath = event.getFile( event.getNumFiles() - 1
).generic_string();
    loadParameters(filepath);
}
```

How it works...

We have two methods for loading and storing the `mColor` values in an XML file. These methods are `loadParameters` and `saveParameters`.

The code we put inside the `draw` method needs some explanation. We are waiting for the `mMakeSnapshot` method to be set to `true` and then we are creating a timestamp to avoid overwriting previous snapshots. The next two lines store the chosen values by invoking the `saveParameters` method and save a current window view as a PNG file using the `writeImage` function. Please notice that we have put that code before invoking `InterfaceGl::draw`, so we save the window view without the GUI.

A nice thing we have here is the *drag-and-drop* feature for loading snapshot files. It's implemented in the `fileDrop` method; Cinder invokes this method every time files are dropped to your application window. First, we get a path to the dropped file; in the case of multiple files, we are taking only one. Then we invoke the `loadParameters` method with the dropped file path as an argument.

Using MayaCamUI

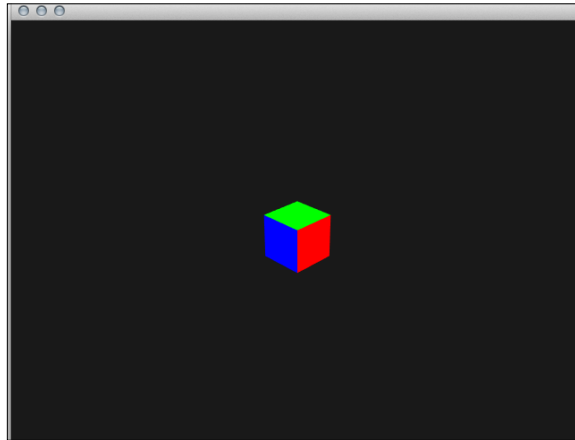
We are going to add to your 3D scene a navigation facility known to us since we modelled a 3D software. Using `MayaCamUI`, you can do this with just a few lines of code.

Getting ready

We need to have some 3D objects in our scene. You can use some primitives provided by Cinder, for example:

```
gl::drawColorCube(Vec3f::zero(), Vec3f(4.f, 4.f, 4.f));
```

A color cube is a cube with a different color on each face, so it is easy to determine the orientation.



How to do it...

Perform the following steps to create camera navigation:

1. We need the `MayaCam.h` header file:

```
#include "cinder/MayaCamUI.h"
```

2. We also need some member declarations in the main class:

```
CameraPersp mCam;  
MayaCamUI mMayaCam;
```

3. Inside the `setup` method, we are going to set up the camera's initial state:

```
mCam.setPerspective(45.0f, getWindowAspectRatio(), 0.1, 10000);  
mMayaCam.setCurrentCam(mCam);
```

4. Now we have to implement three methods:

```
void MainApp::resize( ResizeEvent event )
{
    mCam = mMayaCam.getCamera();
    mCam.setAspectRatio(getWindowAspectRatio());
    mMayaCam.setCurrentCam(mCam);
}

void MainApp::mouseDown( MouseEvent event )
{
    mMayaCam.mouseDown( event.getPos() );
}

void MainApp::mouseDrag( MouseEvent event )
{
    mMayaCam.mouseDrag( event.getPos(), event.isLeftDown(), event.
isMiddleDown(), event.isRightDown() );
}
```

5. Apply camera matrices before your 3D drawing stuff inside the `draw` method:

```
gl::setMatrices(mMayaCam.getCamera());
```

How it works...

Inside the `setup` method, we set the initial camera settings. While the window is resizing, we have to update the aspect ratio of our camera, so we put the code for this in the `resize` method. This method is automatically invoked by Cinder each time the window of our application is resized. We catch mouse events inside the `mouseDown` and `mouseDrag` methods. You can click and drag your mouse for tumbling, right-click for zooming, and use the middle button for panning. Now you have interaction similar to a common 3D modeling software in your own application.

Using 3D space guides

We will try to use built-in Cinder methods to visualize some basic information about the scene we are working on. It should make working with 3D space more comfortable.

Getting ready

We will need the `MayaCamUI` navigation that we have implemented in the previous recipe.

How to do it...

We will draw some objects that will help to visualize and find the orientation of a 3D scene.

1. We will add another camera besides `MayaCamUI`. Let's start by adding member declarations to the main class:

```
CameraPersp    mSceneCam;
int            mCurrentCamera;
```

2. Then we will set the initial values inside the `setup` method:

```
mCurrentCamera = 0;

mSceneCam.setEyePoint(Vec3f(0.f, 5.f, 10.f));
mSceneCam.setViewDirection(Vec3f(0.f, 0.f, -1.f) );
mSceneCam.setPerspective(45.0f, getWindowAspectRatio(), 0.1, 20);
```

3. We have to update the aspect ratio of `mSceneCamera` inside the `resize` method:

```
mSceneCam.setAspectRatio(getWindowAspectRatio());
```

4. Now we will implement the `keyDown` method that will switch between two cameras by pressing the `1` or `2` keys on the keyboard:

```
void MainApp::keyDown( KeyEvent event )
{
    if(event.getChar() == '1') {
        mCurrentCamera = 0;
    } else if(event.getChar() == '2') {
        mCurrentCamera = 1;
    }
}
```

5. Another method we are going to use is `drawGrid`, which looks like this:

```
void MainApp::drawGrid(float size, float step)
{
    gl::color( Color(0.7f, 0.7f, 0.7f) );

    //draw grid
    for(float i=-size;i<=size;i+=step) {
        gl::drawLine(Vec3f(i, 0.f, -size), Vec3f(i, 0.f, size));
        gl::drawLine(Vec3f(-size, 0.f, i), Vec3f(size, 0.f, i));
    }

    // draw bold center lines
    glLineWidth(2.f);
    gl::color(Color::white());
}
```

```

        gl::drawLine(Vec3f(0.f, 0.f, -size), Vec3f(0.f, 0.f, size));
        gl::drawLine(Vec3f(-size, 0.f, 0.f), Vec3f(size, 0.f, 0.f));

        glLineWidth(1.f);
    }

```

6. After that, we can implement our main drawing routine, so here is the whole draw method:

```

void MainApp::draw()
{
    gl::enable(GL_CULL_FACE);
    gl::enableDepthRead();
    gl::enableDepthWrite();
    gl::clear( Color( 0.1f, 0.1f, 0.1f ) );

    if(mCurrentCamera == 0) {
        gl::setMatrices(mMayaCam.getCamera());

        // draw grid
        drawGrid(100.0f, 10.0f);

        // draw coordinate guide
        gl::pushMatrices();
        gl::translate(0.f, 0.4f, 0.f);
        gl::drawCoordinateFrame(5.0f, 1.5f, 0.3f);
        gl::popMatrices();

        // draw scene camera frustum
        gl::color(Color::white());
        gl::drawFrustum(mSceneCam);

        // draw vector guide
        gl::color(Color(1.f,0.f,0.f));
        gl::drawVector(Vec3f(-3.f, 7.f, -6.f),
            Vec3f(3.f, 10.f, -9.f), 1.5f, 0.3);

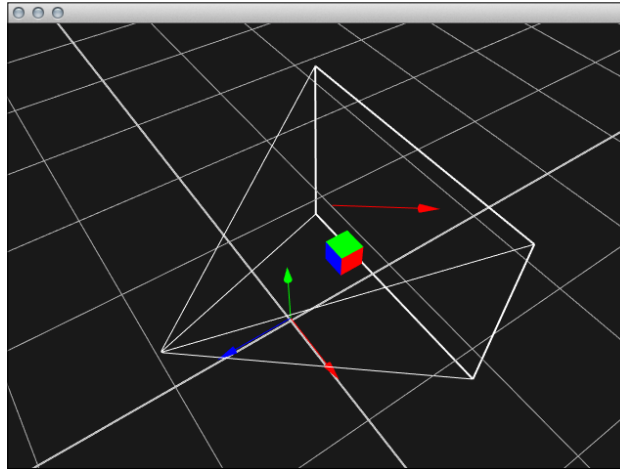
    } else {
        gl::setMatrices(mSceneCam);
    }

    // draw some 3D object
    gl::rotate(30);
    gl::drawColorCube(Vec3f(0.f, 5.f, -5.f),
        Vec3f(2.f, 2.f, 2.f));
}

```

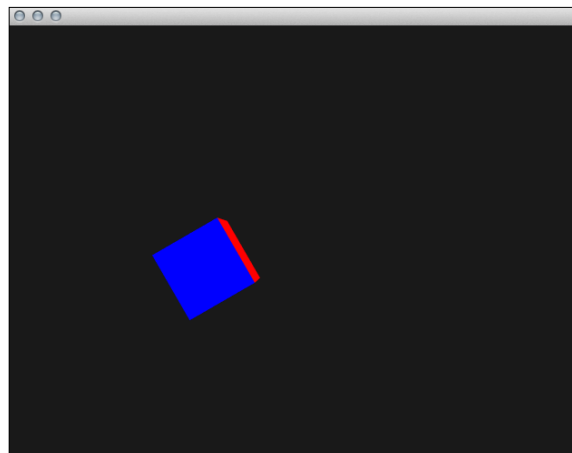
How it works...

We have two cameras; `mSceneCam` is for final rendering and `mMayaCam` is for the preview of objects in our scene. You can switch between them by pressing the `1` or `2` keys. The default camera is `MayaCam`.



In the previous screenshot, you can see the whole scene set up with the elements, such as the origin of the coordinate system, the construction grid that lets you keep orientation in 3D space easily, and the `mSceneCam` frustum and vector visualization between two points in 3D space. You can navigate through this space using `MayaCamUI`.

If you press the `2` key, you will switch to the view of `mSceneCam`, so you will see only your 3D objects without guides as shown in the following screenshot:



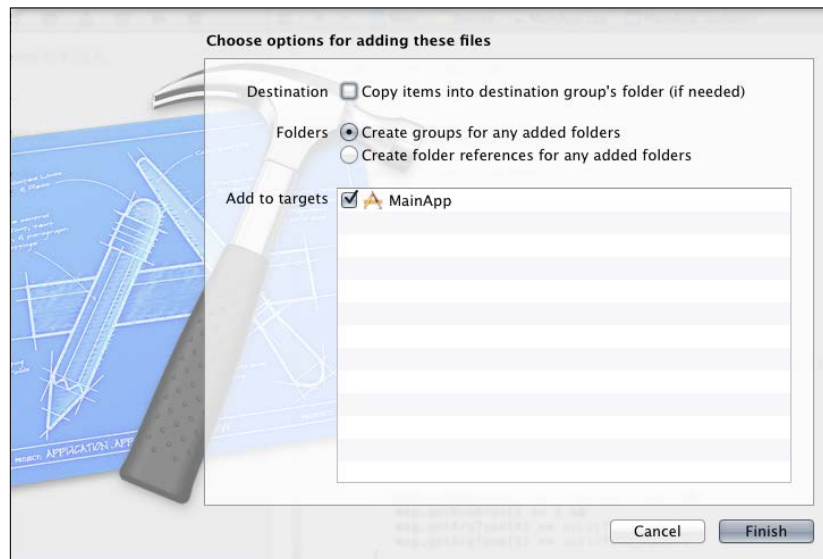
Communicating with other software

We will implement an example communication between two Cinder applications written in Cinder to illustrate how we can send and receive signals. Each of these two applications can be replaced by a non-Cinder application very easily.

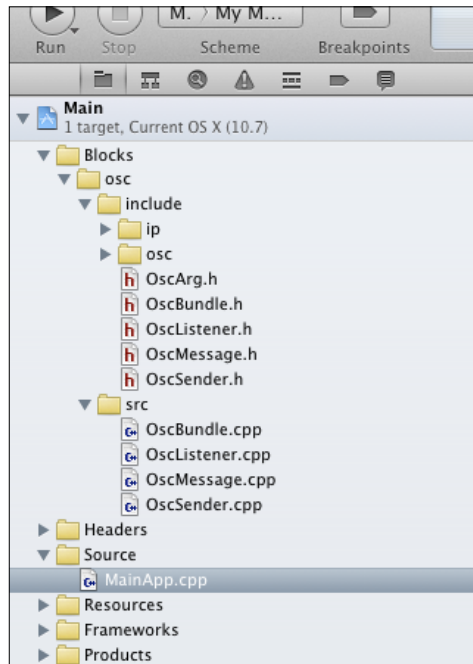
We are going to use the **Open Sound Control (OSC)** messaging format, which is dedicated for communication between wide ranges of multimedia devices over the network. OSC uses UDP protocol, providing flexibility and performance. Each message consists of URL-like addresses and arguments of integer, float, or string type. The popularity of OSC makes it a great tool for connecting different environments or applications developed with different technologies over the network or even on the local machine.

Getting ready

While downloading the Cinder package we are also downloading four primary blocks. One of them is the `osc` block located in the `blocks` directory. First, we will add a new group to our XCode project root and name it `Blocks`, and after that we will drag the `osc` folder inside the `Blocks` group. Be sure the **Create groups for any added folders** options and **MainApp** in the **Add to targets** section are checked.




We only need to include an `src` from the `osc` folders, so we will delete references to the `lib` and `samples` folders from our project tree. The final project structure should look like the following screenshot:



Now we have to add a path to the OSC library file as another linker flag's position in your project's build settings:

```
$(CINDER_PATH)/blocks/osc/lib/macosx/osc.a
```

 **CINDER_PATH** should be set as a user-defined setting in the build settings of your project and it should be the path to Cinder root directory.

How to do it...

First we will cover instructions for the *sender*, and then for the *listener*.

Sender

We will implement an application that sends OSC messages.

1. We have to include an additional header file:

```
#include "OSCSender.h"
```

2. After that we can use the `osc::Sender` class, so let's declare the needed properties in the main class:

```
osc::Sender mOSCSender;
std::string mDestinationHost;
int         mDestinationPort;

Vec2f      mObjPosition;
```

3. Now we have to set up our sender inside the `setup` method:

```
mDestinationHost = "localhost";
mDestinationPort = 3000;
mOSCSender.setup(mDestinationHost, mDestinationPort);
```

4. Set the default value for `mObjPosition` to be the center of the window:

```
mObjPosition = Vec2f(getWindowWidth()*0.5f,
                    getWindowHeight()*0.5f);
```

5. We can now implement the `mouseDrag` method, which includes two major operations—updating the object position according to the mouse position and sending the position information via OSC.

```
void MainApp::mouseDrag(MouseEvent event)
{
    mObjPosition.x = event.getX();
    mObjPosition.y = event.getY();

    osc::Message msg;
    msg.setAddress("/obj/position");
    msg.addFloatArg(mObjPosition.x);
    msg.addFloatArg(mObjPosition.y);
    msg.setRemoteEndpoint(mDestinationHost, mDestinationPort);
    mOSCSender.sendMessage(msg);
}
```

6. The last thing we need to do is to draw a method just to visualize the position of the object:

```
void MainApp::draw()
{
    gl::clear(Color(0.1f, 0.1f, 0.1f));
    gl::color(Color::white());
    gl::drawStrokedCircle(mObjPosition, 50.f);
}
```

Listener

We will implement an application that receives OSC messages.

1. We have to include an additional header file:

```
#include "OSCListener.h"
```

2. After that we can use the `osc::Listener` class, so let's declare the required properties in the main class:

```
osc::Listener  mOSCListener;  
Vec2f          mObjPosition;
```

3. Now we have to set up our listener object inside the `setup` method, passing the port number for listening as a parameter:

```
mOSCListener.setup(3000);
```

4. And the default value for `mObjPosition` to be the center of the window:

```
mObjPosition = Vec2f(getWindowWidth()*0.5f,  
                    getWindowHeight()*0.5f);
```

5. Inside the `update` method, we will be listening for the incoming OSC messages:

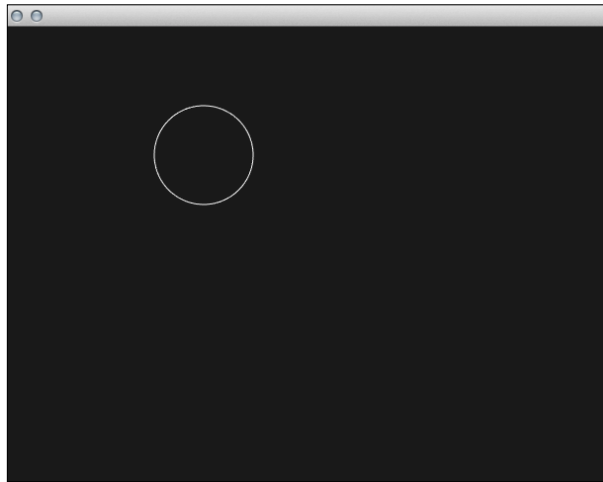
```
void MainApp::update()  
{  
    while (mOSCListener.hasWaitingMessages()) {  
        osc::Message msg;  
        mOSCListener.getNextMessage(&msg);  
  
        if(msg.getAddress() == "/obj/position" &&  
            msg.getNumArgs() == 2 &&  
            msg.getArgType(0) == osc::TYPE_FLOAT &&  
            msg.getArgType(1) == osc::TYPE_FLOAT)  
        {  
            mObjPosition.x = msg.getArgAsFloat(0);  
            mObjPosition.y = msg.getArgAsFloat(1);  
        }  
    }  
}
```

6. Our `draw` method will be almost the same as the sender version, but instead of stroked circle we will draw a filled circle:

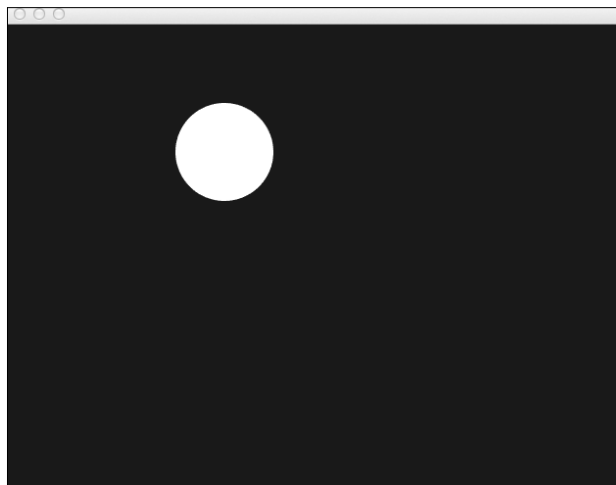
```
void MainApp::draw()  
{  
    gl::clear( Color( 0.1f, 0.1f, 0.1f ) );  
    gl::color(Color::white());  
    gl::drawSolidCircle(mObjPosition, 50.f);  
}
```

How it works...

We have implemented the sender application that sends the position of the mouse via OSC protocol. Those messages, with the address `/obj/position`, can be received by any non-Cinder OSC application implemented in many other frameworks and programming languages. The first argument in the message is the x axis position of the mouse and the second argument is the y axis position. Both are of the `float` type.



In our case, the application that receives messages is another Cinder application that draws a filled circle at exactly the same position where you point it in the sender application window.



There's more...

That was just a short example of the possibilities that OSC offers. This simple communication method can be applied even in very complex projects. OSC works great when several devices are working as independent units. But at some point, data coming from them is processed; for example, frames coming from the camera can be processed by the computer vision software and results sent over the network to another machine projecting the visualization. Implementation on top of the UDP protocol gives not only performance, because of the fact that transmitting data is faster than using TCP, but also implementation is much simpler without a connection handshake.

Broadcast

You can send OSC messages to all the hosts on your network by setting a broadcast address as a destination host: 255.255.255.255. For example, in case of subnets, you can use 192.168.1.255.



If you have problems with compilation under Mac OS X 10.7 because of a linker error, try to set **Inline Methods Hidden** to **No** in your project's build settings.

See also

You can find more information about OSC implementations by checking out the following links.

OSC in Flash

To support receiving and sending OSC messages in your ActionScript 3.0 code you can use the following library: <http://bubblebird.at/tuioflash/>

OSC in Processing

To support **OSC** protocol in your **Processing** sketch you can use following library: <http://www.sojamo.de/libraries/oscP5/>

OSC in openFrameworks

To support receiving and sending OSC messages in your `openFrameworks` project, you can use the `ofxOsc` add-on: <http://ofxaddons.com/repos/112>

OpenSoundControl Protocol

You can find more information about OSC protocol and related tools at its official site: <http://opensoundcontrol.org/>.

Preparing your application for iOS

The big benefit of using Cinder is the resulting multiplatform code. In most cases, your application can be compiled on Windows, Mac OS X, and iOS without significant modifications.

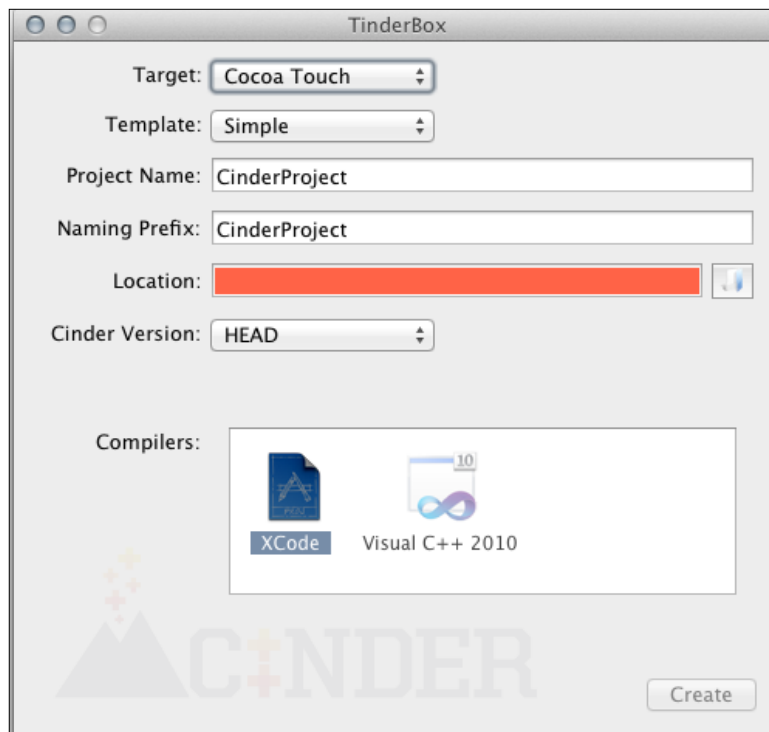
Getting ready

If you want to run your applications on iOS devices, you will need to register as an Apple Developer and purchase the iOS Developer Program.

How to do it...

After registering yourself as an Apple Developer or purchasing the iOS Developer Program, you can create an initial XCode project for iOS using Tinderbox.

1. After running Tinderbox you have to set **Target** to **Cocoa Touch**.



2. It will generate a project structure for you, supporting iOS events that are specific for multitouch screens.

We can use events for multiple touches and for easy access to accelerometer data. The main difference between touch and mouse events is that there can be more than one active touch points while there is only one mouse cursor. Because of that, each touch session has an ID that can be read from `TouchEvent` object.

Method	Describe
<code>touchesBegan(TouchEvent event)</code>	Beginning of a multitouch sequence
<code>touchesMoved(TouchEvent event)</code>	Drags during a multitouch sequence
<code>touchesEnded(TouchEvent event)</code>	The end of a multitouch sequence
<code>getActiveTouches()</code>	Returns all active touches
<code>accelerated(AccelEvent event)</code>	Vector 3D of the acceleration direction

See also

I recommend you take a look at the sample projects included in the Cinder package: `MultiTouchBasic` and `iPhoneAccelerometer`.

Apple Developer Center

You can find more information about the iOS Developer Program here:
<https://developer.apple.com/>

3

Using Image Processing Techniques

In this chapter we will cover:

- ▶ Transforming image contrast and brightness
- ▶ Integrating with OpenCV
- ▶ Detecting edges
- ▶ Detecting faces
- ▶ Detecting features in image
- ▶ Converting images to vector graphics

Introduction

In this chapter, we will show examples of using image processing techniques implemented in Cinder and using third-party libraries. In most of the examples, we will use the following famous test image widely used to illustrate computer vision algorithms and techniques:



You can download Lenna's image from Wikipedia (<http://en.wikipedia.org/wiki/File:Lenna.png>).

Transforming image contrast and brightness

In this recipe we will cover basic image color transformations using the `Surface` class for pixel manipulation.

Getting ready

To change the values of contrast and brightness we will use `InterfaceGl` covered in *Chapter 2, Preparing for Development in the Setting up GUI for parameters tweaking* recipe. We will need a sample image to proceed with; save it in your `assets` folder as `image.png`.

How to do it...

We will create an application with simple GUI for contrast and brightness manipulation on the sample image. Perform the following steps to do so:

1. Include necessary headers:

```
#include "cinder/gl/gl.h"
#include "cinder/gl/Texture.h"
#include "cinder/Surface.h"
#include "cinder/ImageIo.h"
```

2. Add properties to the main class:

```
float mContrast, mContrastOld;
float mBrightness, mBrightnessOld;
Surface32f mImage, mImageOutput;
```

3. In the setup method an image is loaded for processing and the Surface object is prepared to store processed image:

```
mImage = loadImage( loadAsset("image.png") );
mImageOutput = Surface32f(mImage.getWidth(),
    mImage.getHeight(), false);
```

4. Set window size to default values:

```
setWindowSize(1025, 512);
mContrast = 0.f;
mContrastOld = -1.f;
mBrightness = 0.f;
mBrightnessOld = -1.f;
```

5. Add parameter controls to the InterfaceGl window:

```
mParams.addParam("Contrast", &mContrast,
    "min=-0.5 max=1.0 step=0.01");
mParams.addParam("Brightness", &mBrightness,
    "min=-0.5 max=0.5 step=0.01");
```

6. Implement the update method as follows:

```
if(mContrastOld != mContrast || mBrightnessOld != mBrightness) {
float c = 1.f + mContrast;
    Surface32f::IterpixelIter = mImage.getIter();
    Surface32f::IterpixelOutIter = mImageOutput.getIter();

    while( pixelIter.line() ) {
        pixelOutIter.line();
        while( pixelIter.pixel() ) {
```

```
pixelOutIter.pixel();

// contrast transformation
pixelOutIter.r() = (pixelIter.r() - 0.5f) * c + 0.5f;
pixelOutIter.g() = (pixelIter.g() - 0.5f) * c + 0.5f;
pixelOutIter.b() = (pixelIter.b() - 0.5f) * c + 0.5f;

// brightness transformation
pixelOutIter.r() += mBrightness;
pixelOutIter.g() += mBrightness;
pixelOutIter.b() += mBrightness;

    }
}

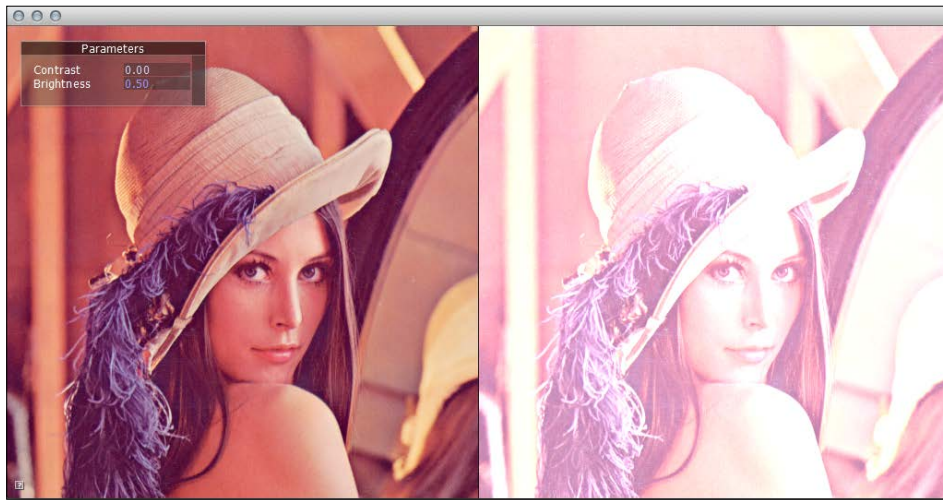
mContrastOld = mContrast;
mBrightnessOld = mBrightness;
}
```

7. Lastly, we will draw the original and processed images by adding the following lines of code inside the `draw` method:

```
gl::draw(mImage);
gl::draw(mImageOutput, Vec2f(512.f+1.f, 0.f));
```

How it works...

The most important part is inside the `update` method. In step 6 we checked if the parameters for contrast and brightness had been changed. If they have, we iterate through all the pixels of the original image and store recalculated color values in `mImageOutput`. While modifying the brightness is just increasing or decreasing each color component, calculating contrast is a little more complicated. For each color component we are using the multiplying formula, $color = (color - 0.5) * contrast + 0.5$, where contrast is a number between 0.5 and 2. In the GUI we are setting a value between -0.5 and 1.0, which is more natural range; it is then recalculated at the beginning of step 6. While processing the image we have to change color value of all pixels, so later in step 6, you can see that we iterate through later columns of each row of the pixels using two `while` loops. To move to the next row we invoked the `line` method on the `Surface` iterator and then the `pixel` method to move to the next pixel of the current row. This method is much faster than using, for example, the `getPixel` and `setPixel` methods.



Our application is rendering the original image on the left-hand side and the processed image on the right-hand side, so you can compare the results of color adjustment.

Integrating with OpenCV

OpenCV is a very powerful open-source library for computer vision. The library is written in C++ so it can be easily integrated in your Cinder application. There is a very useful OpenCV Cinder block provided within Cinder package available at the GitHub repository (<https://github.com/cinder/Cinder-OpenCV>).

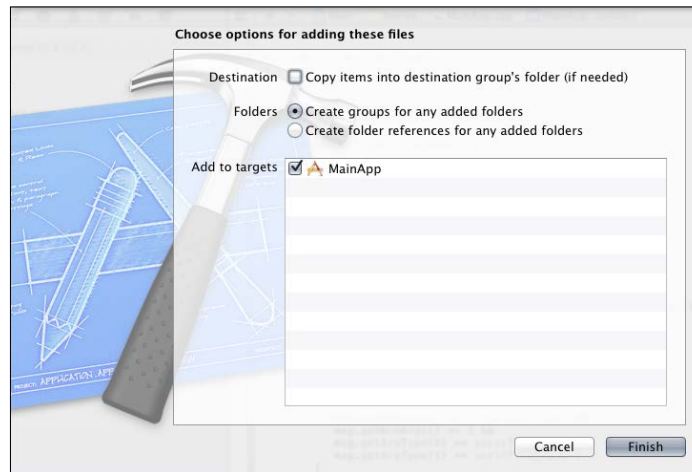
Getting ready

Make sure you have Xcode up and running with a Cinder project opened.

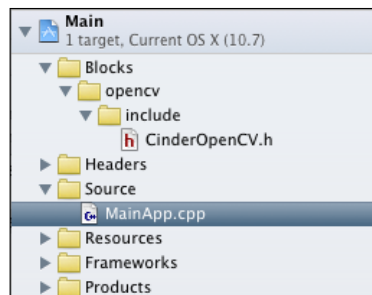
How to do it...

We will add OpenCV Cinder block to your project, which also illustrates the usual way of adding any other Cinder block to your project. Perform the following steps to do so:

1. Add a new group to our Xcode project root and name it `Blocks`. Next, drag the `opencv` folder inside the `Blocks` group. Be sure to select the **Create groups for any added folders** radio button, as shown in the following screenshot:



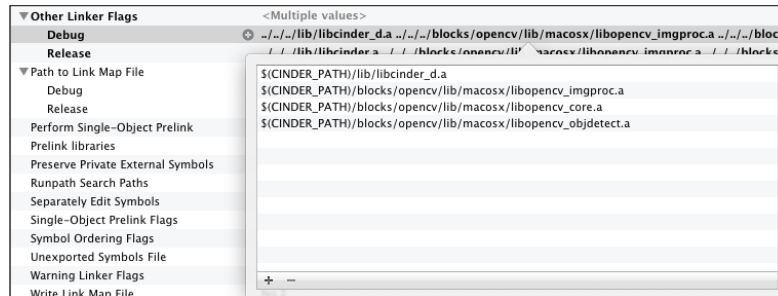
2. You will need only the `include` folder inside the `opencv` folder in your project structure, so delete any reference to others. The final project structure should look like the following screenshot:



3. Add the paths to the OpenCV library files in the **Other Linker Flags** section of your project's build settings, for example:

```
$(CINDER_PATH)/blocks/opencv/lib/macosx/libopencv_imgproc.a  
$(CINDER_PATH)/blocks/opencv/lib/macosx/libopencv_core.a  
$(CINDER_PATH)/blocks/opencv/lib/macosx/libopencv_objdetect.a
```

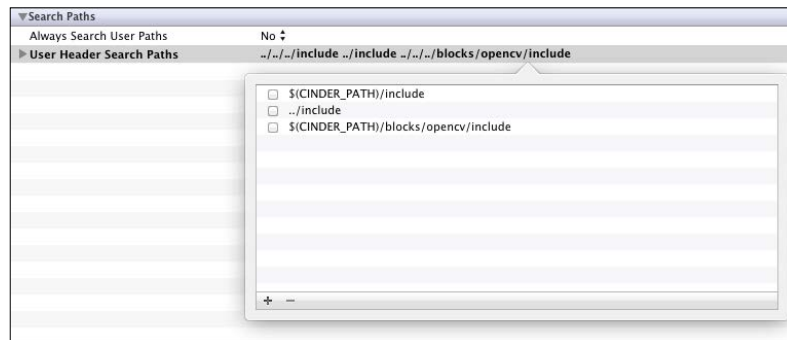
These paths are shown in the following screenshot:



4. Add the paths to the OpenCV Cinder block headers you are going to use in the **User Header Search Paths** section of your project's build settings:

```
$(CINDER_PATH)/blocks/opencv/include
```

This path is shown in the following screenshot:



5. Include OpenCV Cinder block header file:

```
#include "CinderOpenCV.h"
```

How it works...

OpenCV Cinder block provides the `toOcv` and `fromOcv` functions for data exchange between Cinder and OpenCV. After setting up your project you can use them, as shown in the following short example:

```
Surface mImage, mImageOutput;
mImage = loadImage( loadAsset("image.png") );
cv::Mat ocvImage(toOcv(mImage));
cv::cvtColor(ocvImage, ocvImage, CV_BGR2GRAY );
mImageOutput = Surface(fromOcv(ocvImage));
```

You can use the `toOcv` and `fromOcv` functions to convert between Cinder and OpenCV types, storing image data such as `Surface` or `Channel` handled through the `ImageSourceRef` type; there are also other types, as shown in the following table:

Cinder types	OpenCV types
<code>ImageSourceRef</code>	<code>Mat</code>
<code>Color</code>	<code>Scalar</code>
<code>Vec2f</code>	<code>Point2f</code>
<code>Vec2i</code>	<code>Point</code>
<code>Area</code>	<code>Rect</code>

In this example we are linking against the following three files from the OpenCV package:

- ▶ `libopencv_imgproc.a`: This image processing module includes image manipulation functions, filters, feature detection, and more
- ▶ `libopencv_core.a`: This module provides core functionality and data structures
- ▶ `libopencv_objdetect.a`: This module has object detection tools such as cascade classifiers

You can find the documentation on all OpenCV modules at <http://docs.opencv.org/index.html>.

There's more...

There are some features that are not available in precompiled OpenCV libraries packaged in OpenCV Cinder block, but you can always compile your own OpenCV libraries and still use exchange functions from OpenCV Cinder block in your project.

Detecting edges

In this recipe, we will demonstrate how to use edge detection function, which is one of the image processing functions implemented directly in Cinder.

Getting ready

Make sure you have Xcode up and running with an empty Cinder project opened. We will need a sample image to proceed, so save it in your assets folder as `image.png`.

How to do it...

We will process the sample image with the edge detection function. Perform the following steps to do so:

1. Include necessary headers:

```
#include "cinder/gl/Texture.h"
#include "cinder/Surface.h"
#include "cinder/ImageIo.h"

#include "cinder/ip/EdgeDetect.h"
#include "cinder/ip/Grayscale.h"
```

2. Add two properties to your main class:

```
Surface8u mImageOutput;
```

3. Load the source image and set up `Surface` for processed images inside the `setup` method:

```
mImage = loadImage( loadAsset("image.png") );
mImageOutput = Surface8u(mImage.getWidth(), mImage.getHeight(),
false);
```

4. Use image processing functions:

```
ip::grayscale(mImage, &mImage);
ip::edgeDetectSobel(mImage, &mImageOutput);
```

5. Inside the `draw` method add the following two lines of code for drawing images:

```
gl::draw(mImage);
gl::draw(mImageOutput, Vec2f(512.f+1.f, 0.f));
```

How it works...

As you can see, detecting edges in Cinder is pretty easy because of implementation of basic image processing functions directly in Cinder, so you don't have to include any third-party libraries. In this case we are using the `grayscale` function to convert the original image color space to grayscale. It is a commonly used feature in image processing because many algorithms work more efficiently on grayscale images or are even designed to work only with grayscale source images. The edge detection is implemented with the `edgeDetectSobel` function and uses the Sobel algorithm. In this case, the first parameter is the source original grayscale image and the second parameter, is the output `Surface` object in which the result will be stored.

Inside the `draw` method we are drawing both images, as shown in the following screenshot:



There's more...

You may find the image processing functions implemented in Cinder insufficient, so you can also include to your project, third-party library such as OpenCV. We explained how we can use Cinder and OpenCV together in the preceding recipe, *Integrating with OpenCV*.

Other useful functions in the context of edge detection are `Canny` and `findContours`. The following is the example of how we can use them:

```
vector<vector<cv::Point> > contours;
cv::Mat inputMat( toOcv( frame ) );
// blur
cv::cvtColor( inputMat, inputMat, CV_BGR2GRAY );
cv::Mat blurMat;
cv::medianBlur(inputMat, blurMat, 11);

// threshold
cv::Mat thresholdMat;
cv::threshold(blurMat, thresholdMat, 50, 255, CV_8U );

// erode
cv::Mat erodeMat;
cv::erode(thresholdMat, erodeMat, 11);

// Detect edges
cv::Mat cannyMat;
int thresh = 100;
cv::Canny(erodeMat, cannyMat, thresh, thresh*2, 3 );

// Find contours
cv::findContours(cannyMat, contours, CV_RETR_TREE, CV_CHAIN_APPROX_
SIMPLE);
```

After executing the preceding code, the points, which form the contours are stored in the `contours` variable.

Detecting faces

In this recipe, we will examine how our application can be used to recognize human faces. Thanks to the OpenCV library, it is really easy.

Getting ready

We will be using the OpenCV library, so please refer to the *Integrating with OpenCV* recipe for information on how to set up your project. We will need a sample image to proceed, so save it in your `assets` folder as `image.png`. Put the Haar cascade classifier file for frontal face recognition inside the `assets` directory. The cascade file can be found inside the downloaded OpenCV package or in the online public repository, located at https://github.com/Itseez/opencv/blob/master/data/haarcascades/haarcascade_frontalface_alt.xml.

How to do it...

We will create an application that demonstrates the usage of cascade classifier from OpenCV with Cinder. Perform the following steps to do so:

1. Include necessary headers:

```
#include "cinder/gl/Texture.h"
#include "cinder/Surface.h"
#include "cinder/ImageIo.h"
```

2. Add the following members to your main class:

```
Surface8u mImage;
cv::CascadeClassifier mFaceCC;
std::vector<Rectf> mFaces;
```

3. Add the following code snippet to the `setup` method:

```
mImage = loadImage( loadAsset("image.png") );
mFaceCC.load( getAssetPath( "haarcascade_frontalface_alt.xml" )
).string() );
```

4. Also add the following code snippet at the end of the `setup` method:

```
cv::Mat cvImage( toOcv( mImage, CV_8UC1 ) );
std::vector<cv::Rect> faces;
mFaceCC.detectMultiScale( cvImage, faces );
std::vector<cv::Rect>::const_iterator faceIter;
for( faceIter = faces.begin(); faceIter != faces.end(); ++faceIter
) {
    Rectf faceRect( fromOcv( *faceIter ) );
    mFaces.push_back( faceRect );
}
```

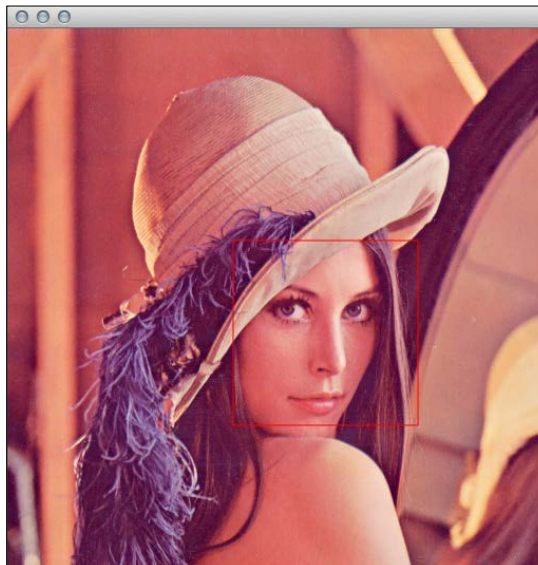
5. At the end of the draw method add the following code snippet:

```
gl::color( Color::white() );
gl::draw(mImage);
gl::color( ColorA( 1.f, 0.f, 0.f, 0.45f ) );
std::vector<Rectf>::const_iterator faceIter;
for(faceIter = mFaces.begin(); faceIter != mFaces.end();
++faceIter ) {
    gl::drawStrokedRect( *faceIter );
}
```

How it works...

In step 3 we loaded an image file for processing and an XML classifier file, which has description of the object features to be recognized. In step 4 we performed an image detection by invoking the `detectMultiScale` function on the `mFaceCC` object, where we pointed to `cvImage` as an input and stored the result in a vector structure, `cvImage` is converted from `mImage` as an 8-bit, single channel image (`CV_8UC1`). What we did next was iterating through all the detected faces and storing `Rectf` variable, which describes a bounding box around the detected face. Finally, in step 5 we drew our original image and all the recognized faces as stroked rectangles.

We are using cascade classifier implemented in OpenCV, which can be trained to detect a specific object in the image. More on training and using cascade classifier for object detection can be found in the OpenCV documentation, located at http://docs.opencv.org/modules/objdetect/doc/cascade_classification.html.



There's more...

You can use a video stream from your camera and process each frame to track faces of people in real time. Please refer to the *Capturing from the camera* recipe in *Chapter 11, Sensing and Tracking Input from the Camera*.

Detecting features in an image

In this recipe we will use one of the methods of finding characteristic features in the image. We will use the SURF algorithm implemented by the OpenCV library.

Getting ready

We will be using the OpenCV library, so please refer to the *Integrating with OpenCV* recipe for information on how to set up your project. We will need a sample image to proceed, so save it in your `assets` folder as `image.png`, then save a copy of the sample image as `image2.png` and perform some transformation on it, for example rotation.

How to do it...

We will create an application that visualizes matched features between two images. Perform the following steps to do so:

1. Add the paths to the OpenCV library files in the **Other Linker Flags** section of your project's build settings, for example:

```
$(CINDER_PATH)/blocks/opencv/lib/macosx/libopencv_imgproc.a
$(CINDER_PATH)/blocks/opencv/lib/macosx/libopencv_core.a
$(CINDER_PATH)/blocks/opencv/lib/macosx/libopencv_objdetect.a
$(CINDER_PATH)/blocks/opencv/lib/macosx/libopencv_features2d.a
$(CINDER_PATH)/blocks/opencv/lib/macosx/libopencv_flann.a
```

2. Include necessary headers:

```
#include "cinder/gl/Texture.h"
#include "cinder/Surface.h"
#include "cinder/ImageIo.h"
```

3. In your main class declaration add the method and properties:

```
int matchImages(Surface8u img1, Surface8u img2);

Surface8u  mImage, mImage2;
gl::Texture mMatchesImage;
```


4. Inside the `setup` method load the images and invoke the matching method:

```
mImage = loadImage( loadAsset("image.png") );
mImage2 = loadImage( loadAsset("image2.png") );

int numberOfMatches = matchImages(mImage, mImage2);
```

5. Now you have to implement previously declared `matchImages` method:

```
int MainApp::matchImages(Surface8u img1, Surface8u img2)
{
    cv::Mat image1(toOcv(img1));
    cv::cvtColor( image1, image1, CV_BGR2GRAY );

    cv::Mat image2(toOcv(img2));
    cv::cvtColor( image2, image2, CV_BGR2GRAY );

    // Detect the keypoints using SURF Detector
    std::vector<cv::KeyPoint> keypoints1, keypoints2;

    cv::SurfFeatureDetector detector;
    detector.detect( image1, keypoints1 );
    detector.detect( image2, keypoints2 );

    // Calculate descriptors (feature vectors)
    cv::SurfDescriptorExtractor extractor;
    cv::Mat descriptors1, descriptors2;

    extractor.compute( image1, keypoints1, descriptors1 );
    extractor.compute( image2, keypoints2, descriptors2 );

    // Matching
    cv::FlannBasedMatcher matcher;
    std::vector<cv::DMatch> matches;
    matcher.match( descriptors1, descriptors2, matches );

    double max_dist = 0;
    double min_dist = 100;

    for( int i = 0; i< descriptors1.rows; i++ )
    {
        double dist = matches[i].distance;
        if( dist<min_dist ) min_dist = dist;
        if( dist>max_dist ) max_dist = dist;
    }

    std::vector<cv::DMatch> good_matches;

    for( int i = 0; i< descriptors1.rows; i++ )
    {
```

```

    if( matches[i].distance<2*min_dist )
    good_matches.push_back( matches[i]);
    }

    // Draw matches
    cv::Mat img_matches;
    cv::drawMatches(image1, keypoints1, image2, keypoints2,
    good_matches, img_matches, cv::Scalar::all(-1),
    cv::Scalar::all(-1),
    std::vector<char>(), cv::DrawMatchesFlags::NOT_DRAW_SINGLE_
    POINTS );

    mMatchesImage = gl::Texture(fromOcv(img_matches));

    return good_matches.size();
    }

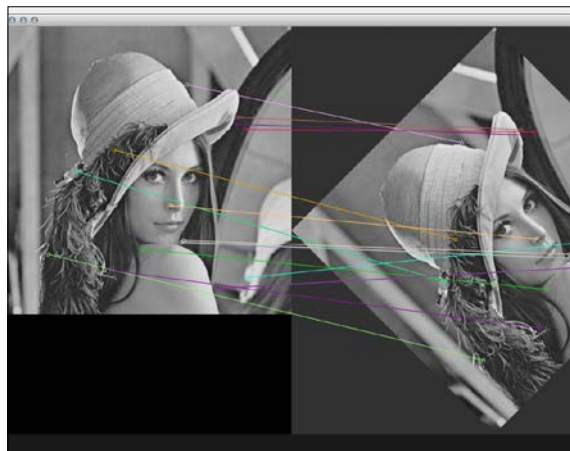
```

6. The last thing is to visualize the matches, so put the following line of code inside the draw method:

```
gl::draw(mMatchesImage);
```

How it works...

Let's discuss the code under step 5. First we are converting `image1` and `image2` to an OpenCV Mat structure. Then we are converting both images to grayscale. Now we can start processing images with SURF, so we are detecting keypoints – the characteristic points of the image calculated by this algorithm. We can use calculated keypoints from these two images and match them using FLANN, or more precisely the `FlannBasedMatcher` class. After filtering out the proper matches and storing them in the `good_matches` vector we can visualize them, as follows:



Please notice that second image is rotated, however the algorithm can still find and link the corresponding keypoints.

There's more...

Detecting characteristic features in the images is crucial for matching pictures and is part of more advanced algorithms used in augmented reality applications.

If images match

It is possible to determine if one of the images is a copy of another or is it rotated. You can use a number of matches returned by the `matchImages` method.

Other possibilities

SURF is rather a slow algorithm for real-time matching so you can try the FAST algorithm for your project if you need to process frames from the camera at real time. The FAST algorithm is also included in the OpenCV library.

See also

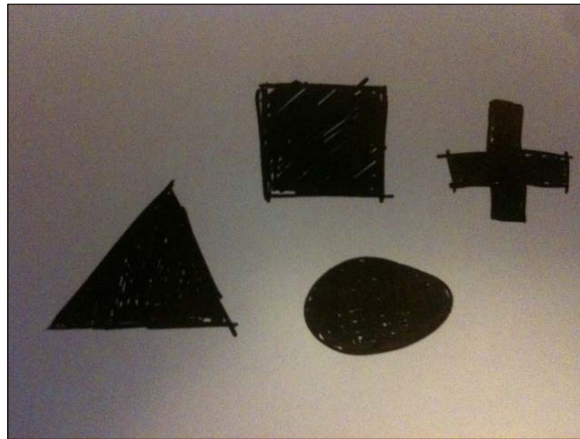
- ▶ The comparison of the OpenCV's feature detection algorithms can be found at <http://computer-vision-talks.com/2011/01/comparison-of-the-opencvs-feature-detection-algorithms-2/>

Converting images to vector graphics

In this recipe, we will try to convert simple, hand-drawn sketches to vector graphics using image processing functions from the OpenCV library and Cairo library for vector drawing and exporting.

Getting started

We will be using the OpenCV library, so please refer to the *Integrating with OpenCV* recipe earlier in this chapter for information on how to set up your project. You may want to prepare your own drawing to be processed. In this example we are using a photo of some simple geometric shapes sketched on paper.



How to do it...

We will create an application to illustrate the conversion to vector shapes. Perform the following steps to do so:

1. Include necessary headers:

```
#include "cinder/gl/Texture.h"
#include "cinder/Surface.h"
#include "cinder/ImageIo.h"
#include "cinder/cairo/Cairo.h"
```

2. Add the following declarations to your main class:

```
void renderDrawing( cairo::Context&ctx );

Surface mImage, mIPImage;
std::vector<std::vector<cv::Point> >mContours, mContoursApprox;
double mApproxEps;
int mCannyThresh;
```

3. Load your drawing and set default values inside the setup method:

```
mImage = loadImage( loadAsset("drawing.jpg") );

mApproxEps = 1.0;
mCannyThresh = 200;
```

4. At the end of the `setup` method add the following code snippet:

```
cv::Mat inputMat( toOcv( mImage ) );

cv::Mat bgr, gray, outputFrame;
cv::cvtColor(inputMat, bgr, CV_BGRA2BGR);
double sp = 50.0;
double sr = 55.0;
cv::pyrMeanShiftFiltering(bgr.clone(), bgr, sp, sr);

cv::cvtColor(bgr, gray, CV_BGR2GRAY);
cv::cvtColor(bgr, outputFrame, CV_BGR2BGRA);
mIPImage = Surface(fromOcv(outputFrame));
cv::medianBlur(gray, gray, 7);

// Detect edges using
cv::MatcannyMat;
cv::Canny(gray, cannyMat, mCannyThresh, mCannyThresh*2.f, 3 );
mIPImage = Surface(fromOcv(cannyMat));

// Find contours
cv::findContours(cannyMat, mContours, CV_RETR_LIST, CV_CHAIN_
APPROX_SIMPLE);

// prepare outline
for( int i = 0; i<mContours.size(); i++ )
{
std::vector<cv::Point> approxCurve;
cv::approxPolyDP(mContours[i], approxCurve, mApproxEps, true);
mContoursApprox.push_back( approxCurve );
}
```

5. Add implementation for the `renderDrawing` method:

```
void MainApp::renderDrawing( cairo::Context&ctx )
{
    ctx.setSource( ColorA( 0, 0, 0, 1 ) );
    ctx.paint();

    ctx.setSource( ColorA( 1, 1, 1, 1 ) );
    for( int i = 0; i<mContoursApprox.size(); i++ )
    {
        ctx.newSubPath();
        ctx.moveTo(mContoursApprox[i][0].x, mContoursApprox[i][0].y);
        for( int j = 1; j <mContoursApprox[i].size(); j++ )
        {
```

```

ctx.lineTo(mContoursApprox[i][j].x, mContoursApprox[i][j].y);
    }
ctx.closePath();
ctx.fill();

ctx.setSource(Color( 1, 0, 0 ));
for( int j = 1; j <mContoursApprox[i].size(); j++ )
    {
ctx.circle(mContoursApprox[i][j].x, mContoursApprox[i][j].y, 2.f);
    }
ctx.fill();
    }
}

```

6. Implement your draw method as follows:

```

gl::clear( Color( 0.1f, 0.1f, 0.1f ) );

gl::color(Color::white());

gl::pushMatrices();
gl::scale(Vec3f(0.5f,0.5f,0.5f));
gl::draw(mImage);
gl::draw(mIPImage, Vec2i(0, mImage.getHeight()+1));
gl::popMatrices();

gl::pushMatrices();
gl::translate(Vec2f(mImage.getWidth()*0.5f+1.f, 0.f));
gl::color( Color::white() );

cairo::SurfaceImage vecSurface( mImage.getWidth(), mImage.
getHeight() );
cairo::Context ctx( vecSurface );
renderDrawing( ctx );
gl::draw( vecSurface.getSurface() );

gl::popMatrices();

```

7. Inside the keyDown method insert the following code snippet:

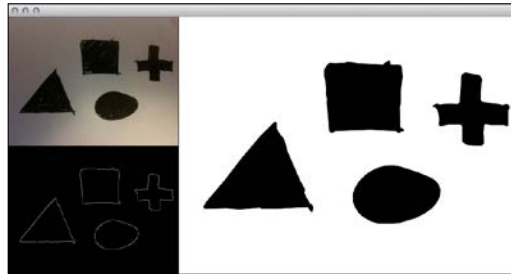
```

if( event.getChar() == 's' ) {
cairo::Context ctx( cairo::SurfaceSvg( getAppPath() /
fs::path("../") / "output.svg",mImage.getWidth(), mImage.
getHeight() ) );
renderDrawing( ctx );
}

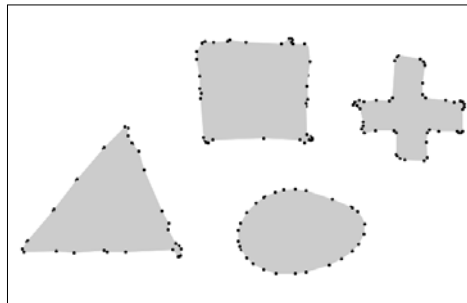
```

How it works...

The key part is implemented in step 4 where we are detecting edges in the image and then finding contours. We are drawing vector representation of processed shapes in step 5, inside the `renderDrawing` method. For drawing vector graphics we are using the Cairo library, which is also able to save results into a file in several vector formats. As you can see in the following screenshot, there is an original image in the upper-left corner and just under it is the preview of the detected contours. The vector version of our simple hand-drawn image is on the right-hand side:



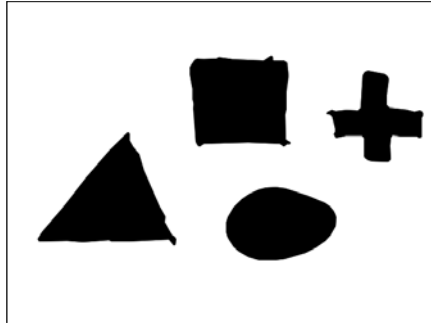
Each shape is a filled path with black color. Paths consist of points calculated in step 4. The following is the visualization with highlighted points:



You can save a vector graphic as a file by pressing the S key. The file will be saved in the same folder as application executable under the name `output.svg`. SVG is only one of the following available exporting options:

Method	Usage
<code>SurfaceSvg</code>	Preparing context for SVG file rendering
<code>SurfacePdf</code>	Preparing context for PDF file rendering
<code>SurfacePs</code>	Preparing context for PostScript file rendering
<code>SurfaceEps</code>	Preparing context for Illustrator EPS file rendering

The exported graphics look as follows:



See also

- ▶ **Cairo:** <http://cairographics.org/>

4

Using Multimedia Content

In this chapter we will learn about:

- ▶ Loading and displaying video
- ▶ Creating a simple video controller
- ▶ Saving window content as an image
- ▶ Saving window animation as video
- ▶ Saving window content as a vector graphics image
- ▶ Saving high resolution images with tile renderer
- ▶ Sharing graphics between applications

Introduction

Most interesting applications use multimedia content in some form or another. In this chapter we will start by learning how to load, manipulate, and display video. We will then move on to saving our graphics into images, image sequences, or video, and then we will move to recording sound visualization.

Lastly, we will learn how to share graphics between applications and how to save mesh data.

Loading and displaying video

In this recipe, we will learn how to load a video from a file and display it on screen using Quicktime and OpenGL. We'll learn how to load a file as a resource or from a file selected by the user using a file open dialog.

Getting ready

You need to have QuickTime installed and also a video file in a format compatible with QuickTime.

To load the video as a resource it is necessary to copy it to the `resources` folder in your project. To learn more on resources, please read the recipes *Using resources on Windows* and *Using resources on OSX and iOS* from *Chapter 1, Getting Started*.

How to do it...

We will use Cinder's QuickTime wrappers to load and display video.

1. Include the headers containing the Quicktime and OpenGL functionality by adding the following at the beginning of the source file:

```
#include "cinder/qtime/QuickTime.h"
#include "cinder/gl/gl.h"
#include "cinder/gl/Texture.h"
```

2. Declare a `ci::qtime::MovieGl` member in you application's class declaration. This example will only need the `setup`, `update`, and `draw` methods, so make sure at least these are declared:

```
using namespace ci;
using namespace ci::app;

class MyApp : public AppBasic {
public:
    void setup();
    void update();
    void draw();

    qtime::MovieGl mMovie;
    gl::Texture mMovieTexture;
};
```

3. To load the video as a resource use the `ci::app::loadResource` method with the file name as parameter and pass the resulting `ci::app::DataSourceRef` when constructing the movie object. It is also good practice to place the loading resource inside a `trycatch` segment in order to catch any resource loading errors. Place the following code inside your `setup` method:

```
try{
mMovie = qtime::MovieGl( loadResource( "movie.mov" ) );
} catch( Exception e){
console() <<e.what()<<std::endl;
}
```

4. You can also load the video by using a file open dialog and passing the file path as an argument when constructing the `mMovie` object. Your `setup` would instead have the following code:

```
try{
fs::path path = getOpenFilePath();
mMovie = qtime::MovieGl( path );
} catch( Exception e){
console() <<e.what()<<std::endl;
}
```

5. To play the video, call the `play` method on the movie object. You can test the successful instantiation of `mMovie` by placing it inside an `if` statement just like an ordinary pointer:

```
If( mMovie ){
mMovie.play();
}
```

6. In the `update` method we copy the texture of the current movie frame into our `mMovieTexture` to draw it later:

```
void MyApp::update(){
if( mMovie ){
mMovieTexture = mMovie.getTexture();
}
```

7. To draw the movie we simply need to draw our texture on screen using the method `gl::draw`. We need to check if the texture is valid because `mMovie` may take a while to load. We'll also create `ci::Rectf` with the texture size and center it on screen to keep the drawn video centered without stretching:

```
gl::clear( Color( 0, 0, 0 ) );
if( mMovieTexture ){
Rect frect = Rectf( mMovieTexture.getBounds() ).getCenteredFit(
getWindowBounds(), true );
gl::draw( mMovieTexture, rect );
}
```

How it works...

The `ci::qtime::MovieGl` class allows playback and control of movies by wrapping around the QuickTime framework. Movie frames are copied into OpenGL textures for easy drawing. To access the texture of the current frame of the movie use the method `ci::qtime::MovieGl::getTexture()` which returns a `ci::gl::Texture` object. Textures used by `ci::qtime::MovieGl` are always bound to the `GL_TEXTURE_RECTANGLE_ARB` target.

There's more

If you wish to do iterations over the pixels of a movie consider using the class `ci::qtime::MovieSurface`. This class allows playback of movies by wrapping around the QuickTime framework, but converts movie frames into `ci::Surface` objects. To access the current frame's surface, use the method `ci::qtime::MovieSurface::getSurface()` which returns a `ci::Surface` object.

Creating a simple video controller

In this recipe we'll learn how to create a simple video controller using the built-in GUI functionalities of Cinder.

We'll control movie playback, if the movie loops or not, the speed rate, volume, and the position.

Getting ready

You must have Apple's QuickTime installed and a movie file in a format compatible with QuickTime.

To learn how to load and display a movie please refer to the previous recipe *Loading and displaying Video*.

How to do it...

We will create a simple interface using Cinder `params` classes to control a video.

1. Include the necessary files to work with Cinder `params` (QuickTime and OpenGL) by adding the following at the top of the source file:

```
#include "cinder/gl/gl.h"
#include "cinder/gl/Texture.h"
#include "cinder/qtime/QuickTime.h"
#include "cinder/params/Params.h"
#include "cinder/Utilities.h"
```

2. Add the `using` statements before the application's class declaration to simplify calling Cinder commands as shown in the following code lines:

```
using namespace ci;
using namespace ci::app;
using namespace ci::gl;
```

3. Declare a `ci::qttime::MovieGl`, `ci::gl::Texture`, and a `ci::params::InterfaceGl` object to play, render, and control the video respectively. Add the following to your class declaration:

```
Texture mMovieTexture;
qttime::MovieGl mMovie;
params::InterfaceGl mParams;
```

4. Select a video file by opening an open file dialog and use that path to initialize our `mMovie`. The following code should go in the `setup` method:

```
try{
fs::path path = getOpenFilePath();
mMovie = qttime::MovieGl( path );
}catch( ... ){
    console() << "could not open video file" <<std::endl;
}
```

5. We'll also need some variables to store the values which we'll manipulate. Each controllable parameter of the video will have two variables to represent the current and the previous value of that parameter. Now declare the following variables:

```
float mMoviePosition, mPrevMoviePosition;
float mMovieRate, mPrevMovieRate;
float mMovieVolume, mPrevMovieVolume;
bool mMoviePlay, mPrevMoviePlay;
bool mMovieLoop, mPrevMovieLoop;
```

6. Set the default values in the `setup` method:

```
mMoviePosition = 0.0f;
mPrevMoviePosition = mMoviePosition;
mMovieRate = 1.0f;
mPrevMovieRate = mMovieRate;
mMoviePlay = false;
mPrevMoviePlay = mMoviePlay;
mMovieLoop = false;
mPrevMovieLoop = mMovieLoop;
mMovieVolume = 1.0f;
mPrevMovieVolume = mMovieVolume;
```

7. Now let's initialize `mParams` and add a control for each of the previously defined variables and set the `max`, `min`, and `step` values when necessary. The following code must go in the `setup` method:

```
mParams = params::InterfaceGl( "Movie Controller", Vec2i( 200, 300
) );
if( mMovie ){
```

```
string max = ci::toString( mMovie.getDuration() );
mParams.addParam( "Position", &mMoviePosition, "min=0.0 max=" +
max + " step=0.5" );

mParams.addParam( "Rate", &mMovieRate, "step=0.01" );

mParams.addParam( "Play/Pause", &mMoviePlay );

mParams.addParam( "Loop", &mMovieLoop );

mParams.addParam( "Volume", &mMovieVolume, "min=0.0 max=1.0
step=0.01" );
}
```

8. In the update method we'll check if the movie was valid and compare each of the parameters to their previous state to see if they changed. If it did, we'll update mMovie and set the parameter to the new value. The following code lines go in the update method:

```
if( mMovie ){

if( mMoviePosition != mPrevMoviePosition ){
mPrevMoviePosition = mMoviePosition;
mMovie.seekToTime( mMoviePosition );
} else {
mMoviePosition = mMovie.getCurrentTime();
mPrevMoviePosition = mMoviePosition;
}

if( mMovieRate != mPrevMovieRate ){
mPrevMovieRate = mMovieRate;
mMovie.setRate( mMovieRate );
}

if( mMoviePlay != mPrevMoviePlay ){
mPrevMoviePlay = mMoviePlay;
if( mMoviePlay ){
mMovie.play();
} else {
mMovie.stop();
}
}

if( mMovieLoop != mPrevMovieLoop ){
mPrevMovieLoop = mMovieLoop;
mMovie.setLoop( mMovieLoop );
}

if( mMovieVolume != mPrevMovieVolume ){
mPrevMovieVolume = mMovieVolume;
```

```

mMovie.setVolume( mMovieVolume );
    }
}

```

9. In the update method it is also necessary to get a handle to the movie texture and copy it to our previously declared `mMovieTexture`. In the update method we write:

```

if( mMovie ){
mMovieTexture = mMovie.getTexture();
}

```

10. All that is left is to draw our content. In the draw method we'll clear the background with black. We'll check the validity of `mMovieTexture` and draw it in a rectangle that fits on the window. We also call the draw command of `mParams` to draw the controls on top of the video:

```

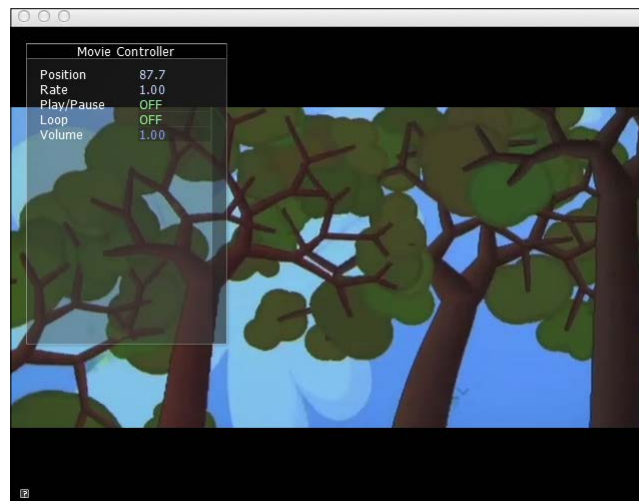
gl::clear( Color( 0, 0, 0 ) );

if( mMovieTexture ){
Rectf rect = Rectf( mMovieTexture.getBounds() ).getCenteredFit(
getWindowBounds(), true );
gl::draw( mMovieTexture, rect );
}

mParams.draw();

```

11. Draw it and you'll see the application's window with a black background along with the controls. Change the various parameters in the parameters menu and you'll see it affecting the video:



How it works...

We created a `ci::params::InterfaceGl` object and added a control for each of the parameters we wanted to manipulate.

We created a variable for each of the parameters we want to manipulate and a variable to store their previous value. In the update we checked to see if these values differ, which will only happen when the user has changed their value using the `mParams` menu.

When the parameter changes we change the `mMovie` parameter with the value the user has set.

Some parameters must be kept in a specific range. The movie position is set in seconds from 0 to the maximum duration of the video in seconds. The volume must be a value between 0 and 1, 0 meaning no audio and 1 being the maximum volume.

Saving window content as an image

In this example we will show you how to save window content to the graphic file and how to implement this functionality in your Cinder application. This could be useful to save output of a graphics algorithm.

How to do it...

We will add a window content saving function to your application:

1. Add necessary headers:

```
#include "cinder/ImageIo.h"
#include "cinder/Utilities.h"
```
2. Add property to your application's main class:

```
bool mMakeScreenshot;
```
3. Set a default value inside the `setup` method:

```
mMakeScreenshot = false;
```
4. Implement the `keyDown` method as follows:

```
void MainApp::keyDown(KeyEvent event)
{
    if(event.getChar() == 's') {
        mMakeScreenshot = true;
    }
}
```

5. Add the following code at the end of the draw method:

```
if(mMakeScreenshot) {
mMakeScreenshot = false;
writeImage( getDocumentsDirectory() / fs::path("MainApp_
screenshot.png"), copyWindowSurface() );
}
```

How it works...

Every time you set `mMakeScreenshot` to `true` the screenshot of your application will be selected and saved. In this case the application waits for the S key to be pressed and then sets the flag `mMakeScreenshot` to `true`. The current application window screenshot will be saved inside your documents directory under the name `MainApp_screenshot.png`.

There's more...

This is just the basic example of common usage of the `writeImage` function. There are many other practical applications.

Saving window animation as image sequences

Let's say you want to record a sequence of images. Perform the following steps to do so:

1. Modify the previous code snippet shown in step 5 for saving the window content as follows:

```
if(mMakeScreenshot || mRecordFrames) {
mMakeScreenshot = false;
writeImage( getDocumentsDirectory() / fs::path("MainApp_
screenshot_" + toString(mFramesCounter) + ".png"),
copyWindowSurface() );
mFramesCounter++;
}
```

2. You have to define `mRecordFrames` and `mFrameCounter` as properties of your main application class:

```
bool mRecordFrames;
int mFramesCounter;
```

3. Set initial values inside the `setup` method:

```
mRecordFrames = false;
mFramesCounter = 1;
```

Recording sound visualization

We assume that you are using `TrackRef` from the `audio` namespace to play your sound. Perform the following steps:

1. Implement the previous steps for saving window animations as image sequences.
2. Type the following lines of code at the beginning of the `update` method:

```
if (mRecordFrames) {  
    mTrack->setTime (mFramesCounter / 30.f);  
}
```

We are calculating the desired audio track position based on the number of frames that passed. We are doing that to synchronize animation with the music track. In this case we want to produce 30 fps animation so we are dividing `mFramesCounter` by 30.

Saving window animations as video

In this recipe, we'll start by drawing a simple animation and learning how to export it to video. We will create a video where pressing any key will start or stop the recording.

Getting ready

You must have Apple's QuickTime installed. Make sure you know where you want your video to be saved, as you'll have to specify its location at the beginning.

It could be anything that is drawn using OpenGL but for this example, we'll create a yellow circle at the center of the window with a changing radius. The radius is calculated by the absolute value of the sine of the elapsed seconds since the application launched. We multiply this value by 200 to scale it up. Now add the following to the `draw` method:

```
gl::clear( Color( 0, 0, 0 ) );  
float radius = fabsf( sinf( getElapsedSeconds() ) ) * 200.0f;  
Vec2f center = getWindowCenter();  
gl::color( Color( 1.0f, 1.0f, 0.0f ) );  
gl::drawSolidCircle( center, radius );
```

How to do it...

We will use the `ci::qttime::MovieWriter` class to create a video of our rendering.

1. Include the OpenGL and QuickTime files at the beginning of the source file by adding the following:

```
#include "cinder/gl/gl.h"  
#include "cinder/qttime/MovieWriter.h"
```

- Now let's declare a `ci::qtime::MovieWriter` object and a method to initialize it. Add the following to your class declaration:

```
qtime::MovieWriter mMovieWriter;
void initMovieWriter();
```

- In the implementation of `initMovieWriter` we start by asking the user to specify a path using a save file dialog and use it to initialize the movie writer. The movie writer also needs to know the window's width and height. Here's the implementation of `initMovieWriter`.

```
void MyApp::initMovieWriter() {
    fs::path path = getSaveFilePath();
    if( path.empty() == false ) {
        mMovieWriter = qtime::MovieWriter( path, getWindowWidth(),
        getWindowHeight() );
    }
}
```

- Let's declare a key event handler by declaring the `keyUp` method.

```
void keyUp( KeyEvent event );
```

- In its implementation we will see if there is already a movie being recorded by checking the validity of `mMovieWriter`. If it is a valid object then we must save the current movie by destroying the object. We can do so by calling the `ci::qtime::MovieWriter` default constructor; this will create a null instance. If `mMovieWriter` is not a valid object then we initialize a new movie writer by calling the method `initMovieWriter()`.

```
void MovieWriterApp::keyUp( KeyEvent event ) {
    if( mMovieWriter ) {
        mMovieWriter = qtime::MovieWriter();
    } else {
        initMovieWriter();
    }
}
```

- The last two steps are to check if `mMovieWriter` is valid and to add a frame by calling the method `addFrame` with the window's surface. This method has to be called in the draw method, after our drawing routines have been made. Here's the final draw method, including the circle drawing code.

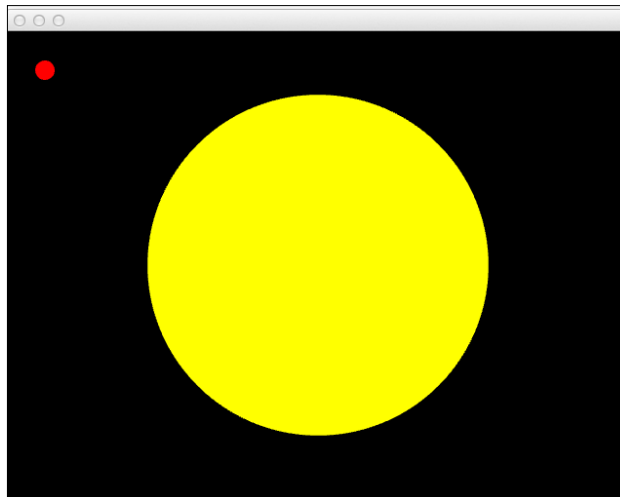
```
void MyApp::draw()
{
    gl::clear( Color( 0, 0, 0 ) );

    float radius = fabsf( sinf( getElapsedSeconds() ) ) * 200.0f;
```

```
Vec2f center = getWindowCenter();
gl::color( Color( 1.0f, 1.0f, 0.0f ) );
gl::drawSolidCircle( center, radius );

if( mMovieWriter ){
mMovieWriter.addFrame( copyWindowSurface() );
}
}
```

7. Build and run the application. Pressing any key will start or end a video recording. Each time a new recording begins, the user will be presented with a save file dialog to set where the movie will be saved.



How it works...

The `ci::qttime::MovieWriter` object allows for easy movie writing using Apple's QuickTime. Recordings begin by initializing a `ci::qttime::MovieWriter` object and are saved when the object is destroyed. By calling the `addFrame` method, new frames are added.

There's more...

You can also define the format of the video by creating a `ci::qttime::MovieWriter::Format` object and passing it as an optional parameter in the movie writer's constructor. If no format is specified, the movie writer will use the default PNG codec and 30 frames per second.

For example, to create a movie writer with the H264 codec with 50 percent quality and 24 frames per second, you could write the following code:

```
qtime::MovieWriter::Format format;
format.setCodec( qtime::MovieWriter::CODEC_H264 );
format.setQuality( 0.5f );
format.setDefaultDuration( 1.0f / 24.0f );
qtime::MovieWriter mMovieWriter = ci::Qtime::MovieWriter( "mymovie.
mov", getWindowWidth(), getWindowHeight(), format );
```

You can optionally open a **Settings** window and allow the user to define the video settings by calling the static method `qtime::MovieWriter::getUserCompressionSettings`. This method will populate a `qtime::MovieWriter::Format` object and return `true` if successful or `false` if the user canceled the change in the setting.

To use this method for defining the settings and creating a movie writer, you can write the following code:

```
qtime::MovieWriter::Format format;
qtime::MovieWriter mMovieWriter;
bool formatDefined = qtime::MovieWriter::getUserCompressionSettings(
&format );
if( formatDefined ){
mMovieWriter = qtime::MovieWriter( "mymovie.mov", getWindowWidth(),
getWindowHeight(), format );
}
```

It is also possible to enable **multipass** encoding. For the current version of Cinder it is only available using the H264 codec. Multipass encoding will increase the movie's quality but at the cost of a greater performance decrease. For this reason it is disabled by default.

To write a movie with multipass encoding enabled we can write the following:

```
qtime::MovieWriter::Format format;
format.setCodec( qtime::MovieWriter::CODEC_H264 );
format.enableMultiPass( true );
qtime::MovieWriter mMovieWriter = ci::Qtime::MovieWriter( "mymovie.
mov", getWindowWidth(), getWindowHeight(), format );
```

There are plenty of settings and formats that can be set using the `ci::qtime::MovieWriter::Format` class and the best way to know the full list of options is to check the documentation for the class at http://libcinder.org/docs/v0.8.4/guide__qtime__movie_writer.html.

Saving window content as a vector graphics image

In this recipe we'll learn how to draw 2D graphics on screen and save it to an image in a vector graphics format using the cairo renderer.

Vector graphics can be extremely useful when creating visuals for printing as they can be scaled without losing quality.

Cinder has an integration for the cairo graphics library; a powerful and full-featured 2D renderer, capable of outputting to a variety of formats including popular vector graphics formats.

To learn more about the cairo library, please go to its official web page:
<http://www.cairographics.org>

In this example we'll create an application that draws a new circle whenever the user presses the mouse. When any key is pressed, the application will open a save file dialog and save the content in a format defined by the file's extension.

Getting ready

To draw graphics created with the cairo renderer we must define our renderer to be `Renderer2d`.

At the end of the source file of our application class there's a *macro* to initialize the application where the second parameter defines the renderer. If your application is called `MyApp`, you must change the macro to be the following:

```
CINDER_APP_BASIC( MyApp, Renderer2d )
```

The cairo renderer allows exporting of PDF, SVG, EPS, and PostScript formats. When specifying the file to save, make sure you write one of the supported extensions: `pdf`, `svg`, `eps`, or `ps`.

Include the following files at the top of your source file:

```
#include "cinder/Rand.h"  
#include "cinder/cairo/Cairo.h"
```

How to do it...

We will use Cinder's cairo wrappers to create images in vector formats from our rendering.

1. To create a new circle every time the user presses the mouse we must first create a `Circle` class. This class will contain position, radius, and color parameters. Its constructor will take `ci::Vec2f` to define its position and will generate a random radius and color.

Write the following code before the application's class declaration:

```
class Circle{
public:
    Circle( const Vec2f&pos ){
this->pos = pos;
radius = randFloat( 20.0f, 50.0f );
color = ColorA( randFloat( 1.0f ), randFloat( 1.0f ), randFloat(
1.0f ), 0.5f );
    }

    Vec2f pos;
    float radius;
    ColorA color;
};
```

2. We should now declare `std::vector` of circles where we'll store the created circles. Add the following code to your class declaration:

```
std::vector< Circle >mCircles;
```

3. Let's create a method which will draw the circles that will take `cairo::Context` as their parameter:

```
void renderScene( cairo::Context &context );
```

4. In the method definition, iterate over `mCircles` and draw each one in the context:

```
void MyApp::renderScene( cairo::Context &context ){
for( std::vector< Circle >::iterator it = mCircles.begin(); it !=
mCircles.end(); ++it ){
context.circle( it->pos, it->radius );
context.setSource( it->color );
context.fill();
    }
}
```

5. At this point we only need to add a circle whenever the user presses the mouse. To do this, we must implement the `mouseDown` event handler by declaring it in the class declaration.

```
void mouseDown( MouseEvent event );
```

6. In its implementation we add a `Circle` class to `mCircles` using the mouse position.

```
void MyApp::mouseDown( MouseEvent event ){
    Circle circle( event.getPos() );
mCircles.push_back( circle );
}
```


7. We can now draw the circles on the window by creating `cairo::Context` bound to the window's surface. This will let us visualize what we're drawing. Here's the draw method implementation:

```
void CairoSaveApp::draw()
{
    cairo::Context context( cairo::createWindowSurface() );
    renderScene( context );
}
```

8. To save the scene to an image file we must create a context bound to a surface that represents a file in a vector graphics format. Let's do this whenever the user releases a key by declaring the `keyUp` event handler.

```
void keyUp( KeyEvent event );
```

9. In the `keyUp` implementation we create `ci::fs::path` and populate it by calling a save file dialog. We'll also create an empty `ci::cairo::SurfaceBase` which is the base for all the surfaces that the cairo renderer can draw to.

```
fs::path filePath = getSaveFilePath();
cairo::SurfaceBase surface;
```

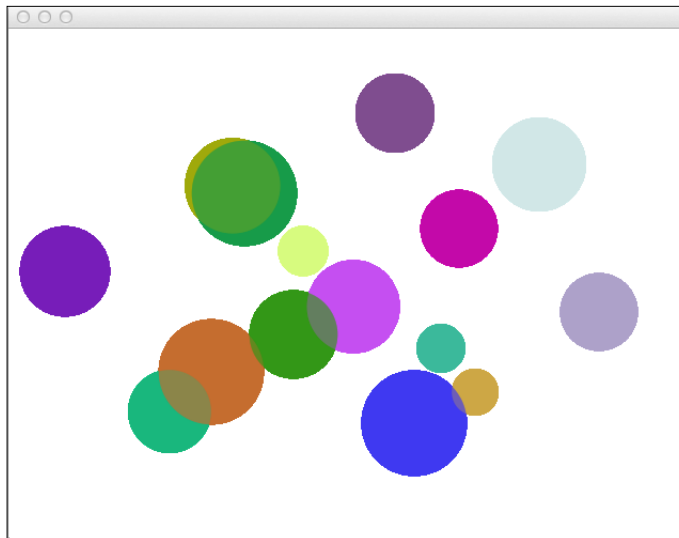
10. We'll now compare the extension of the path with the supported formats and initialize the surface accordingly. It can be initialized as `ci::cairo::SurfacePdf`, `ci::cairo::SurfaceSvg`, `ci::cairo::SurfaceEps`, or as `ci::cairo::SurfacePs`.

```
if( filePath.extension() == ".pdf" ){
    surface = cairo::SurfacePdf( filePath, getWindowWidth(),
    getWindowHeight() );
} else if( filePath.extension() == ".svg" ){
    surface = cairo::SurfaceSvg( filePath, getWindowWidth(),
    getWindowHeight() );
} else if( filePath.extension() == ".eps" ){
    surface = cairo::SurfaceEps( filePath, getWindowWidth(),
    getWindowHeight() );
} else if( filePath.extension() == ".ps" ){
    surface = cairo::SurfacePs( filePath, getWindowWidth(),
    getWindowHeight() );
}
```

11. Now we can create `ci::cairo::Context` and render our scene to it by calling the `renderScene` method and passing the context as a parameter. The circles will be rendered to the context and a file will be created in the specified format. Here's the final `keyUp` method implementation:

```
void CairoSaveApp::keyUp( KeyEvent event ){
    fs::path filePath = getSaveFilePath();
    cairo::SurfaceBase surface;
```

```
if( filePath.extension() == ".pdf" ){
    surface = cairo::SurfacePdf( filePath, getWindowWidth(),
        getWindowHeight() );
    } else if( filePath.extension() == ".svg" ){
    surface = cairo::SurfaceSvg( filePath, getWindowWidth(),
        getWindowHeight() );
    } else if( filePath.extension() == ".eps" ){
    surface = cairo::SurfaceEps( filePath, getWindowWidth(),
        getWindowHeight() );
    } else if( filePath.extension() == ".ps" ){
    surface = cairo::SurfacePs( filePath, getWindowWidth(),
        getWindowHeight() );
    }
    cairo::Context context( surface );
    renderScene( context );
}
```



How it works...

Cinder wraps and integrates the cairo 2D vector renderer. It allows use of Cinder's types to draw and interact with cairo.

The complete drawing is made by calling the drawing methods of a `ci::cairo::Context` object. The context in turn, must be created by passing a surface object extending `ci::cairo::SurfaceBase`. All drawings will be made in the surface and rasterized according to the type of the surface.

The following surfaces allow saving images in a vector graphics format:

Surface type	Format
<code>ci::cairo::SurfacePdf</code>	PDF
<code>ci::cairo::SurfaceSvg</code>	SVG
<code>ci::cairo::SurfaceEps</code>	EPS
<code>ci::cairo::SurfacePs</code>	PostsScript

There's more...

It is also possible to draw using other renderers. Though the renderers aren't able to create vector images, they can be useful in other situations.

Here are the other available surfaces:

Surface Type	Format
<code>ci::cairo::SurfaceImage</code>	Anti-aliased pixel-based rasterizer
<code>ci::cairo::SurfaceQuartz</code>	Apple's Quartz
<code>ci::cairo::SurfaceCgBitmapContext</code>	Apple's CoreGraphics
<code>ci::cairo::SurfaceGdi</code>	Windows GDI

Saving high resolution images with the tile renderer

In this recipe we'll learn how to export a high-resolution image of the content being drawn on screen using the `ci::gl::TileRender` class. This can be very useful when creating graphics for print.

We'll start by creating a simple scene and drawing it on screen. Next, we'll code our example so that whenever the user presses any key, a save file dialog will appear and a high-resolution image will be saved to the specified path.

Getting ready

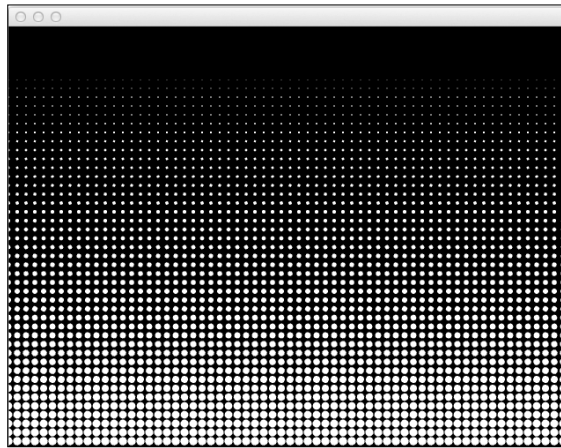
The `TileRender` class can create high resolution images from anything being drawn on screen using OpenGL calls.

To save an image with `TileRender` we must first draw some content on screen. It can be anything but for this example let's create a nice simple pattern with circles to fill the screen.

In the implementation of your draw method write the following code:

```
void MyApp::draw()
{
    gl::clear( Color( 0, 0, 0 ) );
    gl::color( Color::white() );
    for( float i=0; i<getWidth(); i+=10.0f ){
        for( float j=0; j<getHeight(); j += 10.0f ){
            float radius = j * 0.01f;
            gl::drawSolidCircle( Vec2f( i, j ), radius );
        }
    }
}
```

Remember that this could be anything that is drawn on screen using OpenGL.



How to do it...

We will use the `ci::gl::TileRender` class to generate high-resolution images of our OpenGL rendering.

1. Include the necessary headers by adding the following at the top of the source file:

```
#include "cinder/gl/TileRender.h"
#include "cinder/ImageIo.h"
```

2. Since we'll save a high-resolution image whenever the user presses any key, let's implement the `keyUp` event handler by declaring it in the class declaration.

```
void keyUp( KeyEvent event );
```

3. In the `keyUp` implementation we start by creating a `ci::gl::TileRender` object and then set the width and height of the image we are going to create. We are going to set it to be four times the size of the application window. It can be of any size you want, just take in to account that if you don't respect the window's aspect ratio, the image will become stretched.

```
gl::TileRender tileRender( getWindowWidth() * 4, getWindowHeight() * 4 );
```

4. We must define our scene's `Modelview` and `Projection` matrices to match our window. If we are using only 2D graphics we can call the method `setMatricesWindow`, as follows:

```
tileRender.setMatricesWindow( getWindowWidth(), getWindowHeight() );
```

To define the scene's `Modelview` and `Projection` matrices to match the window while drawing 3D content, it is necessary to call the method `setMatricesWindowPersp`:

```
tileRender.setMatricesWindowPersp( getWindowWidth(), getWindowHeight() );
```

5. Next we'll draw our scene each time a new tile is created by using the method `nextTile`. When all the tiles have been created the method will return `false`. We can create all the tiles by redrawing our scene in a `while` loop while asking if there is a next tile, as follows:

```
while( tileRender.nextTile() ){  
    draw();  
}
```

6. Now that the scene is fully rendered in `TileRender`, we must save it. Let's ask the user to indicate where to save by opening a save file dialog. It is mandatory to specify an extension for the image file as it will be used internally to define the image format.

```
fs::path filePath = getSaveFilePath();
```

7. We check if `filePath` is not empty and write the tile render surface as an image using the `writeImage` method.

```
if( filePath.empty() == false ){  
    writeImage( filePath, tileRender.getSurface() );  
}
```

8. After saving the image it is necessary to redefine the window's `Modelview` and `Projection` matrices. If drawing in 2D you can set the matrices to their default values by using the method `setMatricesWindow` with the window's dimensions, as follows:

```
gl::setMatricesWindow( getWindowWidth(), getWindowHeight() );
```

How it works...

The `ci::gl::TileRender` class makes it possible to generate high-resolution versions of our rendering by scaling individual portions of our drawing to the entire size of the window and storing them as `ci::Surface`. After the entire scene has been stored in individual portions it is stitched together as tiles to form a single high-resolution `ci::Surface`, which can then be saved as an image.

Sharing graphics between applications

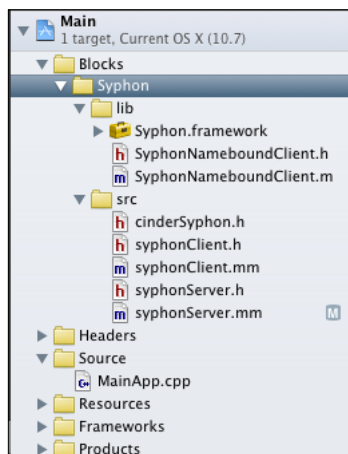
In this recipe we will show you the way of sharing graphic in real time between applications under Mac OS X. To do that, we will use **Syphon** and its implementation for Cinder. Syphon is an open source tool that allows an application to share graphics as still frames or real-time updated frame sequence. You can read more about Syphon here: <http://syphon.v002.info/>

Getting ready

To test if the graphic shared by our application is available, we are going to use **Syphon Recorder**, which you can find here: <http://syphon.v002.info/recorder/>

How to do it...

1. Checkout Syphon CinderBlock from the *syphon-implementations* repository <http://code.google.com/p/syphon-implementations/>.
2. Create a new group inside your project tree and name it `Blocks`.
3. Drag-and-drop Syphon CinderBlock into your newly created `Blocks` group.



4. Make sure **Syphon.framework** is added to the **Copy Files** section of **Build Phases** in the **target** settings.

5. Add necessary header files:

```
#include "cinderSyphon.h"
```

6. Add property to your main application class:

```
syphonServer mScreenSyphon;
```

7. At the end of setup method, add the following code:

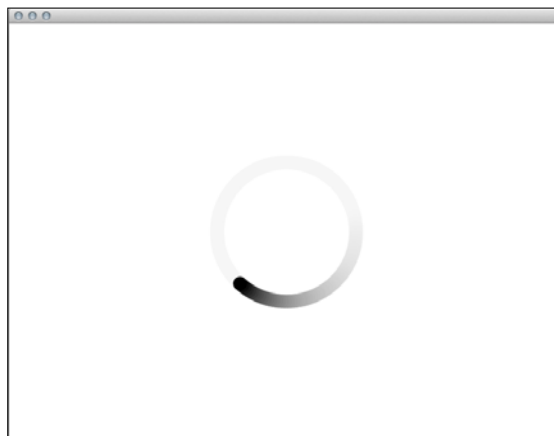
```
mScreenSyphon.setName("Cinder Screen");  
gl::clear(Color::white());
```

8. Inside the draw method add the following code:

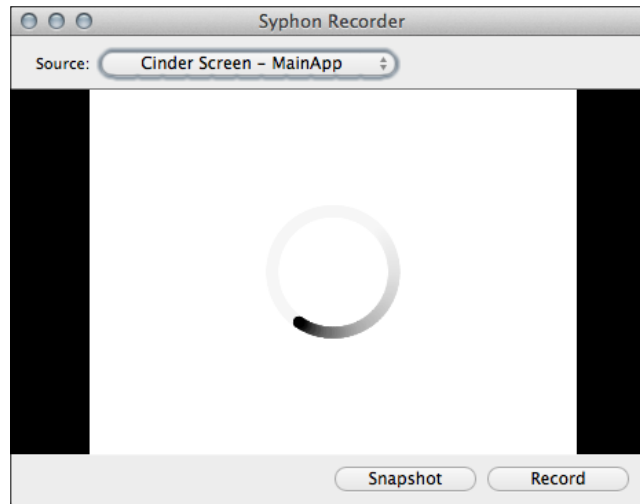
```
gl::enableAlphaBlending();  
  
gl::color( ColorA(1.f, 1.f, 1.f, 0.05f) );  
gl::drawSolidRect( getWindowBounds() );  
  
gl::color( ColorA::black() );  
Vec2f pos = Vec2f( cos(getElapsedSeconds()),  
sin(getElapsedSeconds()) * 100.f );  
gl::drawSolidCircle(getWindowCenter() + pos, 10.f);  
  
mScreenSyphon.publishScreen();
```

How it works...

Application draws a simple rotating animation and shares the whole window area via Syphon library. Our application window looks like the following screenshot:



To test if the graphic can be received by other applications, we will use Syphon Recorder. Run Syphon Recorder and find our Cinder application in the drop-down menu under the name: **Cinder Screen – MainApp**. We set up the first part of this name at the step 6 of this recipe in the *How to do it...* section while the second part is an executable file name. Now, the preview from our Cinder application should be available and it would look like the following screenshot:



There's more...

The Syphon library is very useful, simple to use, and is available for other applications and libraries.

Receiving graphics from other applications

You can receive textures from other applications as well. To do this, you have to use the `syphonClient` class as shown in the following steps:

1. Add a property to your application main class:


```
syphonClient mClientSyphon;
```
2. Initialize `mClientSyphon` inside the CIT method:


```
mClientSyphon.setApplicationName("MainApp Server");
mClientSyphon.setServerName("");
mClientSyphon.bind();
```
3. At the end of the `draw` method add the following line which draws graphics that the other application is sharing:


```
mClientSyphon.draw(Vec2f::zero());
```


5

Building Particle Systems

In this chapter we will cover:

- ▶ Creating a particle system in 2D
- ▶ Applying repulsion and attraction forces
- ▶ Simulating particles flying in the wind
- ▶ Simulating flocking behavior
- ▶ Making our particles sound reactive
- ▶ Aligning particles to processed images
- ▶ Aligning particles to mesh surfaces
- ▶ Creating springs

Introduction

Particle systems are a computational technique of using a large number of small graphic objects to perform different types of simulations such as explosions, wind, fire, water, and flocking.

In this chapter, we are going to learn how to create and animate particles using popular and versatile physics algorithms.

Creating a particle system in 2D

In this recipe, we are going to learn how we can build a basic particle system in two dimensions using the Verlet algorithm.

Getting ready

We will need to create two classes, a `Particle` class representing a single particle, and a `ParticleSystem` class to manage our particles.

Using your IDE of choice, create the following files:

- ▶ `Particle.h`
- ▶ `Particle.cpp`
- ▶ `ParticleSystem.h`
- ▶ `ParticleSystem.cpp`

How to do it...

We will learn how we can create a basic particle system. Perform the following steps to do so:

1. First, let's declare our `Particle` class in the `Particle.h` file and include the necessary Cinder files:

```
#pragma once

#include "cinder/gl/gl.h"
#include "cinder/Vector.h"

class Particle{
};
```

2. Let's add, to the class declaration, the necessary member variables – `ci::Vec2f` to store the position, previous position, and applied forces; and `float` to store particle radius, mass, and drag.

```
ci::Vec2f position, prevPosition;
ci::Vec2f forces;
float radius;
float mass;
float drag;
```

3. The last thing needed to finalize the `Particle` declaration is to add a constructor that takes the particle's initial position, radius, mass, and drag, and methods to update and draw the particle.

The following is the final `Particle` class declaration:

```
class Particle{
public:
```

```

Particle( const ci::Vec2f& position, float radius,
float mass, float drag );

void update();
void draw();

ci::Vec2f position, prevPosition;
ci::Vec2f forces;
float radius;
float mass;
float drag;
};

```

- Let's move on to the `Particle.cpp` file and implement the `Particle` class.

The first necessary step is to include the `Particle.h` file, as follows:

```
#include "Particle.h"
```

- We initialize the member variables to the values passed in the constructor. We also initialize `forces` to zero and `prevPosition` to the initial position.

```

Particle::Particle( const ci::Vec2f& position, float radius, float
mass, float drag ){
    this->position = position;
    this->radius = radius;
    this->mass = mass;
    this->drag = drag;
    prevPosition = position;
    forces = ci::Vec2f::zero();
}

```

- In the `update` method, we need to create a temporary `ci::Vec2f` variable to store the particle's position before it is updated.

```
ci::Vec2f temp = position;
```

- We calculate the velocity of the particle by finding the difference between current and previous positions and multiplying it by `drag`. We store this value in `ci::Vec2f` temporarily for clarity.

```
ci::Vec2f vel = ( position - prevPosition ) * drag;
```

- To update the particle's position, we add the previously calculated velocity and add `forces` divided by `mass`.

```
position += vel + forces / mass;
```

9. The final steps in the update method are to copy the previously stored position to `prevPosition` and reset forces to a zero vector.

The following is the complete update method implementation:

```
void Particle::update() {
    ci::Vec2f temp = position;
    ci::Vec2f vel = ( position - prevPosition ) * drag;
    position += vel + forces / mass;
    prevPosition = temp;
    forces = ci::Vec2f::zero();
}
```

10. In the draw implementation, we simply draw a circle at the particle's position using its radius.

```
void Particle::draw() {
    ci::gl::drawSolidCircle( position, radius );
}
```

11. Now with the `Particle` class complete, we need to begin working on the `ParticleSystem` class. Move to the `ParticleSystem.h` file, include the necessary files, and create the `ParticleSystem` class declaration.

```
#pragma once

#include "Particle.h"
#include <vector>

class ParticleSystem{
public:

};
```

12. Let's add a destructor and methods to update and draw our particles. We'll also need to create methods to add and destroy particles and finally a `std::vector` variable to store the particles in this system. The following is the final class declaration:

```
Class ParticleSystem{
public:
    ~ParticleSystem();

    void update();
    void draw();

    void addParticle( Particle *particle );
    void destroyParticle( Particle *particle );
};
```

```

        std::vector<Particle*> particles;

};

```

13. Moving to the `ParticleSystem.cpp` file, let's begin working on the implementation. The first thing we need to do is include the file with the class declaration.

```
#include "ParticleSystem.h"
```

14. Now let's implement the methods one by one. In the destructor, we iterate through all the particles and delete them.

```

ParticleSystem::~ParticleSystem() {
    for( std::vector<Particle*>::iterator it = particles.begin(); it
        != particles.end(); ++it ) {
        delete *it;
    }
    particles.clear();
}

```

15. The update method will be used to iterate all the particles and call `update` on each of them.

```

void ParticleSystem::update() {
    for( std::vector<Particle*>::iterator it = particles.begin(); it
        != particles.end(); ++it ) {
        (*it)->update();
    }
}

```

16. The draw method will iterate all the particles and call `draw` on each of them.

```

void ParticleSystem::draw() {
    for( std::vector<Particle*>::iterator it = particles.begin(); it
        != particles.end(); ++it ) {
        (*it)->draw();
    }
}

```

17. The `addParticle` method will insert the particle on the `particles` container.

```

void ParticleSystem::addParticle( Particle *particle ) {
    particles.push_back( particle );
}

```

18. Finally, `destroyParticle` will delete the particle and remove it from the particles' list.

We'll find the particles' iterator and use it to delete and later remove the object from the container.

```
void ParticleSystem::destroyParticle( Particle *particle ){
    std::vector<Particle*>::iterator it = std::find( particles.
begin(), particles.end(), particle );
    delete *it;
    particles.erase( it );
}
```

19. With our classes ready, let's go to our application's class and create some particles.

In our application's class, we need to include the `ParticleSystem` header file and the necessary header to use random numbers at the top of the source file:

```
#include "ParticleSystem.h"
#include "cinder/Rand.h"
```

20. Declare a `ParticleSystem` object on our class declaration.

```
ParticleSystem mParticleSystem;
```

21. In the `setup` method we can create 100 particles with random positions on our window and random radius. We'll define the mass to be the same as the radius as a way to have a relation between size and mass. `drag` will be set to 9.5.

Add the following code snippet inside the `setup` method:

```
int numParticle = 100;
for( int i=0; i<numParticle; i++ ){
    float x = ci::randFloat( 0.0f, getWindowWidth() );
    float y = ci::randFloat( 0.0f, getWindowHeight() );
    float radius = ci::randFloat( 5.0f, 15.0f );
    float mass = radius;radius;
    float drag = 0.95f;
    Particle *particle = new Particle
        ( Vec2f( x, y ), radius, mass, drag );
    mParticleSystem.addParticle( particle );
}
```

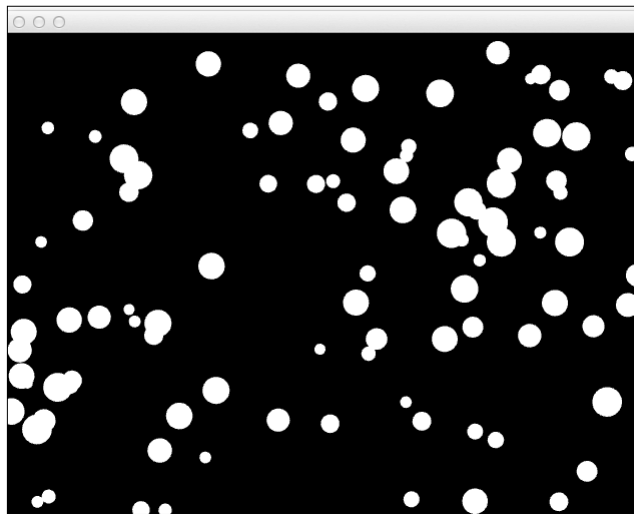
22. In the `update` method, we need to update the particles by calling the `update` method on `mParticleSystem`.

```
void MyApp::update() {
    mParticleSystem.update();
}
```

23. In the `draw` method we need to clear the screen, set up the window's matrices, and call the `draw` method on `mParticleSystem`.

```
void ParticlesApp::draw()
{
    gl::clear( Color( 0, 0, 0 ) );
    gl::setMatricesWindow( getWindowWidth(), getWindowHeight() );
    mParticleSystem.draw();
}
```

24. Build and run the application and you will see 100 random circles on screen, as shown in the following screenshot:



In the next recipes we will learn how to animate the particles in organic and appealing ways.

How it works...

The method described previously uses a popular and versatile Verlet integrator. One of its main characteristics is an implicit approximation of velocity. This is accomplished by calculating, on each update of the simulation, the distance traveled since the last update of the simulation. This allows for greater stability as velocity is implicit to position and there is less chance these will ever get out of sync.

The `drag` member variable represents resistance to movement and should be a number between 0.0 and 1.0. A value of 0.0 represents such a great resistance that the particle will not be able to move. A value of 1.0 represents absence of resistance and will make the particle move indefinitely. We applied `drag` in step 7, where we multiplied `drag` by the velocity:

```
ci::Vec2f vel = ( position - prevPosition ) * drag;
```


There's more...

To create a particle system in 3D it is necessary to use a 3D vector instead of a 2D one.

Since Cinder's vector 2D and 3D vector classes have a very similar class structure, we simply need to change `position`, `prevPosition`, and `forces` to be `ci::Vec3f` objects.

The constructor will also need to take a `ci::Vec3f` object as an argument instead.

The following is the class declaration with these changes:

```
class Particle{
public:

    Particle( const ci::Vec3f& position,
             float radius, float mass, float drag );

    void update();
    void draw();

    ci::Vec3f position, prevPosition;
    ci::Vec3f forces;
    float radius;
    float mass;
    float drag;
};
```

The draw method should also be changed to allow for 3D drawing; we could, for example, draw a sphere instead of a circle:

```
void Particle::draw(){
    ci::gl::drawSphere( position, radius );
}
```

See also

- ▶ For more information on the implementation of the Verlet algorithm, please refer to the paper by Thomas Jakobsen, located at <http://www.pagines.ma1.upc.edu/~susin/contingut/AdvancedCharacterPhysics.pdf>
- ▶ For more information on the Verlet integration, please read the wiki at http://en.wikipedia.org/wiki/Verlet_integration

Applying repulsion and attraction forces

In this recipe, we will show how you can apply repulsion and attraction forces to the particle system that we have implemented in the previous recipe.

Getting ready

In this recipe, we are going to use the code from the *Creating particle system in 2D* recipe.

How to do it...

We will illustrate how you can apply forces to the particle system. Perform the following steps:

1. Add properties to your application's main class.

```
Vec2f attrPosition;
float attrFactor, repulsionFactor, repulsionRadius;
```

2. Set the default value inside the `setup` method.

```
attrPosition = getWindowCenter();
attrFactor = 0.05f;
repulsionRadius = 100.f;
repulsionFactor = -5.f;
```

3. Implement the `mouseMove` and `mouseDown` methods, as follows:

```
void MainApp::mouseMove(MouseEvent event)
{
    attrPosition.x = event.getPos().x;
    attrPosition.y = event.getPos().y;
}

void MainApp::mouseDown(MouseEvent event)
{
    for( std::vector<Particle*>::iterator it = mParticleSystem.
particles.begin(); it != mParticleSystem.particles.end(); ++it ) {
        Vec2f repulsionForce = (*it)->position - event.getPos();
        repulsionForce = repulsionForce.normalized() *
math<float>::max(0.f, repulsionRadius - repulsionForce.
length());
        (*it)->forces += repulsionForce;
    }
}
```

4. At the beginning of the update method, add the following code snippet:

```
for( std::vector<Particle*>::iterator it = mParticleSystem.  
particles.begin(); it != mParticleSystem.particles.end(); ++it ) {  
    Vec2f attrForce = attrPosition - (*it)->position;  
    attrForce *= attrFactor;  
    (*it)->forces += attrForce;  
}
```

How it works...

In this example we added interaction to the particles engine introduced in the first recipe. The attraction force is pointing to your mouse cursor position but the repulsion vector points in the opposite direction. These forces were calculated and applied to each particle in steps 3 and 4, and then we made the particles follow your mouse cursor, but when you click on the left mouse button, they are suddenly moves away from the mouse cursor. This effect can be achieved with basic vector operations. Cinder lets you perform vector calculations pretty much the same way you usually do on scalars.

The repulsion force is calculated in step 3. We are using the normalized vector beginning at the mouse cursor position and the end of the particle position, multiplied by the repulsion factor, calculated on the basis of the distance between the particle and the mouse cursor position. Using the `repulsionRadius` value, we can limit the range of the repulsion.

We are calculating the attraction force in step 4 taking the vector beginning at the particle position and the end at the mouse cursor position. We are multiplying this vector by the `attrFactor` value, which controls the strength of the attraction.



Simulating particles flying in the wind

In this recipe, we will explain how you can apply Brownian motion to your particles. Particles are going to behave like snowflakes or leaves flying in the wind.

Getting ready

In this recipe we are going to use the code base from the *Creating a particle system in 2D* recipe.

How to do it...

We will add movement to particles calculated from the Perlin noise and sine function. Perform the following steps to do so:

1. Add the necessary headers.

```
#include "cinder/Perlin.h"
```

2. Add properties to your application's main class.

```
float    mFrequency;
Perlin   mPerlin;
```

3. Set the default value inside the `setup` method.

```
mFrequency = 0.01f;
mPerlin = Perlin();
```

4. Change the number of the particles, their radius, and mass.

```
int numParticle = 300;
float radius = 1.f;
float mass = Rand::randFloat(1.f, 5.f);
```

5. At the beginning of the `update` method, add the following code snippet:

```
Vec2f oscilationVec;
oscilationVec.x = sin(getElapsedSeconds()*0.6f)*0.2f;
oscilationVec.y = sin(getElapsedSeconds()*0.2f)*0.1f;
std::vector<Particle*>::iterator it;
for(it = mParticleSystem.particles.begin(); it != mParticleSystem.
particles.end(); ++it ) {
    Vec2f windForce = mPerlin.dfBm( (*it)->position * mFrequency );
    (*it)->forces += windForce * 0.1f;
    (*it)->forces += oscilationVec;
}
```

How it works...

The main movement calculations and forces are applied in step 5. As you can see we are using the Perlin noise algorithm implemented as a part of Cinder. It provides a method to retrieve Brownian motion vectors for each particle. We also add `oscillationVec` that makes particles swing from left-to-right and backwards, adding more realistic behavior.



See also

- ▶ **Perlin noise original source:** <http://mrl.nyu.edu/~perlin/doc/oscar.html#noise>
- ▶ **Brownian motion:** http://en.wikipedia.org/wiki/Brownian_motion

Simulating flocking behavior

Flocking is a term applied to the behavior of birds and other flying animals that are organized into a swarm or flock.

From our point of view, it is especially interesting that flocking behavior can be simulated by applying only three rules to each particle (Boid). These rules are as follows:

- ▶ **Separation:** Avoid neighbors that are too near
- ▶ **Alignment:** Steer towards the average velocity of neighbors
- ▶ **Cohesion:** Steer towards the average position of neighbors

Getting ready

In this recipe, we are going to use the code from the *Creating a particle system in 2D* recipe.

How to do it...

We will implement the rules for flocking behavior. Perform the following steps to do so:

1. Change the number of the particles, their radius, and mass.

```
int numParticle = 50;
float radius = 5.f;
float mass = 1.f;
```

2. Add a definition for new methods and properties to the `Particle` class inside the `Particle.h` header file.

```
void flock(std::vector<Particle*>& particles);
ci::Vec2f steer(ci::Vec2f target, bool slowdown);
void borders(float width, float height);
ci::Vec2f separate(std::vector<Particle*>& particles);
ci::Vec2f align(std::vector<Particle*>& particles);
ci::Vec2f cohesion(std::vector<Particle*>& particles);
```

```
float maxspeed;
float maxforce;
ci::Vec2f vel;
```

3. Set the default values for `maxspeed` and `maxforce` at the end of the `Particle` constructor inside the `Particle.cpp` source file.

```
this->maxspeed = 3.f;
this->maxforce = 0.05f;
```

4. Implement the new methods of the `Particle` class inside the `Particle.cpp` source file.

```
void Particle::flock(std::vector<Particle*>& particles) {
    ci::Vec2f acc;
    acc += separate(particles) * 1.5f;
    acc += align(particles) * 1.0f;
    acc += cohesion(particles) * 1.0f;
    vel += acc;
    vel.limit(maxspeed);
}

ci::Vec2f Particle::steer(ci::Vec2f target, bool slowdown) {
    ci::Vec2f steer;
    ci::Vec2f desired = target - position;
    float d = desired.length();
    if (d > 0) {
        desired.normalize();
```

```
        if ((slowdown) && (d <100.0)) desired *= (maxspeed*(d/100.0));
        else desired *= maxspeed;
        steer = desired - vel;
        steer.limit(maxforce);
    }
else {
    steer = ci::Vec2f::zero();
}
return steer;
}

void Particle::borders(float width, float height) {
    if (position.x< -radius) position.x = width+radius;
    if (position.y< -radius) position.y = height+radius;
    if (position.x>width+radius) position.x = -radius;
    if (position.y>height+radius) position.y = -radius;
}
```

5. Add a method for the separation rule.

```
ci::Vec2f Particle::separate(std::vector<Particle*>& particles) {
    ci::Vec2f resultVec = ci::Vec2f::zero();
    float targetSeparation = 30.f;
    int count = 0;
    for( std::vector<Particle*>::iterator it = particles.begin(); it
    != particles.end(); ++it ) {
        ci::Vec2f diffVec = position - (*it)->position;
        if( diffVec.length() >0&&diffVec.length() <targetSeparation ) {
            resultVec += diffVec.normalized() / diffVec.length();
            count++;
        }
    }

    if (count >0) {
        resultVec /= (float)count;
    }

    if (resultVec.length() >0) {
        resultVec.normalize();
        resultVec *= maxspeed;
        resultVec -= vel;
        resultVec.limit(maxforce);
    }

    return resultVec;
}
```

6. Add a method for the alignment rule.

```

ci::Vec2f Particle::align(std::vector<Particle*>& particles) {
    ci::Vec2f resultVec = ci::Vec2f::zero();
    float neighborDist = 50.f;
    int count = 0;
    for( std::vector<Particle*>::iterator it = particles.begin(); it
        != particles.end(); ++it ) {
        ci::Vec2f diffVec = position - (*it)->position;
        if( diffVec.length() >0 && diffVec.length() <neighborDist ) {
            resultVec += (*it)->vel;
            count++;
        }
    }

    if (count >0) {
        resultVec /= (float)count;
    }

    if (resultVec.length() >0) {
        resultVec.normalize();
        resultVec *= maxspeed;
        resultVec -= vel;
        resultVec.limit(maxforce);
    }

    return resultVec;
}

```

7. Add a method for the cohesion rule.

```

ci::Vec2f Particle::cohesion(std::vector<Particle*>& particles) {
    ci::Vec2f resultVec = ci::Vec2f::zero();
    float neighborDist = 50.f;
    int count = 0;
    for( std::vector<Particle*>::iterator it = particles.begin(); it
        != particles.end(); ++it ) {
        float d = position.distance( (*it)->position );
        if( d >0 && d <neighborDist ) {
            resultVec += (*it)->position;
            count++;
        }
    }

    if (count >0) {
        resultVec /= (float)count;
    }
}

```



```
        return steer(resultVec, false);
    }

    return resultVec;
}
```

8. Change the update method to read as follows

```
void Particle::update() {
    ci::Vec2f temp = position;
    position += vel + forces / mass;
    prevPosition = temp;
    forces = ci::Vec2f::zero();
}
```

9. Change the drawing method of Particle, as follows:

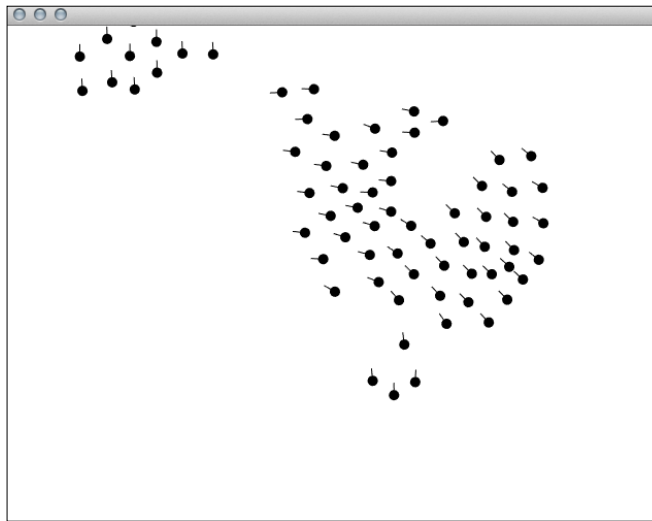
```
void Particle::draw() {
    ci::gl::color(1.f, 1.f, 1.f);
    ci::gl::drawSolidCircle( position, radius );
    ci::gl::color(1.f, 0.f, 0.f);
    ci::gl::drawLine(position,
        position+( position - prevPosition).normalized()*(radius+5.f) );
}
```

10. Change the update method of ParticleSystem inside the ParticleSystem.cpp source file, as follows:

```
void ParticleSystem::update() {
    for( std::vector<Particle*>::iterator it = particles.begin(); it
        != particles.end(); ++it ) {
        (*it)->flock(particles);
        (*it)->update();
        (*it)->borders(640.f, 480.f);
    }
}
```

How it works...

Three rules for flocking—separation, alignment, and cohesion—were implemented starting from step 4 and they were applied to each particle in step 10. In this step, we also prevented Boids from going out of the window boundaries by resetting their positions.



See also

- **Flocking:** [http://en.wikipedia.org/wiki/Flocking_\(behavior\)](http://en.wikipedia.org/wiki/Flocking_(behavior))

Making our particles sound reactive

In this recipe we will pick on the previous particle system and add animations based on **fast Fourier transform (FFT)** analysis from an audio file.

The FFT analysis will return a list of values representing the amplitudes of several frequency windows. We will match each particle to a frequency window and use its value to animate the repulsion that each particle applies to all other particles.

This example uses Cinder's FFT processor, which is only available on Mac OS X.

Getting ready

We will be using the same particle system developed in the previous recipe, *Creating a particle system in 2D*. Create the `Particle` and `ParticleSystem` classes described in that recipe, and include the `ParticleSystem.h` file at the top of the application's source file.

How to do it...

Using values from the FFT analysis we will animate our particles. Perform the following steps to do so:

1. Declare a `ParticleSystem` object on your application's class and a variable to store the number of particles we will create.

```
ParticleSystem mParticleSystem;  
int mNumParticles;
```

2. In the `setup` method we'll create 256 random particles. The number of particles will match the number of values we will receive from the audio analysis.

The particles will begin at a random position on the window and have a random size and mass. `drag` will be 0.9.

```
mNumParticles = 256;  
for( int i=0; i<mNumParticles; i++ ){  
    float x = ci::randFloat( 0.0f, getWindowWidth() );  
    float y = ci::randFloat( 0.0f, getWindowHeight() );  
    float radius = ci::randFloat( 5.0f, 15.0f );  
    float mass = radius;  
    float drag = 0.9f;  
    Particle *particle = new Particle  
        ( Vec2f( x, y ), radius, mass, drag );  
    mParticleSystem.addParticle( particle );  
}
```

3. In the `update` method, we have to call the `update` method on the particle system.

```
void MyApp::update() {  
    mParticleSystem.update();  
}
```

4. In the `draw` method, we have to clear the background, calculate the window's matrices, and call the `draw` method on the particle system.

```
void MyApp::draw()  
{  
    gl::clear( Color( 0, 0, 0 ) );  
    gl::setMatricesWindow( getWindowWidth(), getWindowHeight() );  
    mParticleSystem.draw();  
}
```

5. Now let's load and play an audio file. We start by including the necessary files to load, play, and perform the FFT analysis. Add the following code snippet at the top of the source file:

```
#include "cinder/audio/IO.h"
#include "cinder/audio/FftProcessor.h"
#include "cinder/audio/PcmBuffer.h"
#include "cinder/audio/Output.h"
```

6. Now declare `ci::audio::TrackRef`, which is a reference to an audio track.

```
Audio::TrackRef mAudio;
```

7. In the `setup` method we will open a file dialog to allow the user to select which audio file to play.

If the retrieved path is not empty, we will use it to load and add a new audio track.

```
fs::path audioPath = getOpenFilePath();
if( audioPath.empty() == false ){
    mAudio = audio::Output::addTrack( audio::load( audioPath.
        string() ) );
}
```

8. We'll check if `mAudio` was successfully loaded and played. We will also enable the PCM buffer and looping.

```
if( mAudio ){
    mAudio->enablePcmBuffering( true );
    mAudio->setLooping( true );
    mAudio->play();
}
```

9. Now that we have an audio file playing, we need to start animating the particles. First we need to apply an elastic force towards the center of the window. We do so by iterating over all particles and adding a force, which is one-tenth of the difference between the particle's position and the window's center position.

Add the following code snippet to the `update` method:

```
Vec2f center = getWindowCenter();
for( vector<Particle*>::iterator it = mParticleSystem.particles.
begin(); it != mParticleSystem.particles.end(); ++it ){
    Particle *particle = *it;
    Vec2f force =
        ( center - particle->position ) * 0.1f;
    particle->forces += force;
}
```

10. Now we have to calculate the FFT analysis. This will be done once after every frame in the update.

Declare a local variable `std::shared_ptr<float>`, where the result of the FFT will be stored.

We will get a reference to the PCM buffer of `mAudio` and perform an FFT analysis on its left channel. It is a good practice to perform a test to check the validity of `mAudio` and its buffer.

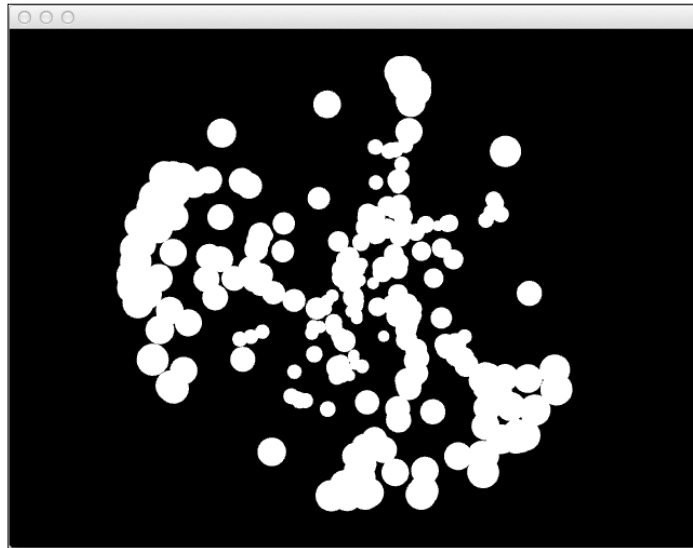
```
std::shared_ptr<float>fft;
if( mAudio ){
    audio::PcmBuffer32fRef pcmBuffer = mAudio->getPcmBuffer();
    if( pcmBuffer ){
        fft = audio::calculateFft( pcmBuffer->getChannelData(
            audio::CHANNEL_FRONT_LEFT ), mNumParticles );
    }
}
```

11. We will use the values from the FFT analysis to scale the repulsion each particle is applying.

Add the following code snippet to the update method:

```
if( fft ){
    float *values = fft.get();
    for( int i=0; i<mParticleSystem.particles.size()-1; i++ ){
        for( int j=i+1; j<mParticleSystem.particles.size(); j++ ){
            Particle *particleA =
                mParticleSystem.particles[i];
            Particle *particleB =
                mParticleSystem.particles[j];
            Vec2f delta = particleA->position -
                particleB->position;
            float distanceSquared = delta.lengthSquared();
            particleA->forces += ( delta / distanceSquared ) * particleB->
                mass * values[j] * 0.5f;
            particleB->forces -= ( delta / distanceSquared ) * particleA->
                mass * values[i] * 0.5f;
        }
    }
}
```

12. Build and run the application; you will be prompted to select an audio file. Select it and it will begin playing. The particles will move and push each other around according to the audio's frequencies.



How it works...

We created a particle for each one of the values the FFT analysis returns and made each particle repulse every other particle according to its correspondent frequency window amplitude. As the music evolves, the animation will react accordingly.

See also

- ▶ To learn more about fast Fourier transform please visit http://en.wikipedia.org/wiki/Fast_Fourier_transform

Aligning particles to a processed image

In this recipe, we will show how you can use techniques you were introduced to in the previous recipes to make particles align to the edge detected in the image.

Getting ready

In this recipe, we are going to use the particles' implementation from the *Creating a particle system in 2D* recipe; the image processing example from the *Detecting edges* recipe in *Chapter 3, Using Image Processing Techniques*; as well as simulating repulsion covered in the *Applying repulsion and attraction forces* recipe.

How to do it...

We will create particles aligning to the detected edges in the image. Perform the following steps to do so:

1. Add an anchor property to the Particle class in the Particle.h file.
`ci::Vec2f anchor;`
2. Set the anchor value at the end of the Particle class constructor in the Particle.cpp source file.
`anchor = position;`
3. Add a new property to your application's main class.
`float maxAlignSpeed;`
4. At the end of the setup method, after image processing, add new particles, as follows:

```
mMouseDown = false;
repulsionFactor = -1.f;
maxAlignSpeed = 10.f;

mImage = loadImage( loadAsset("image.png") );
mImageOutput = Surface8u(mImage.getWidth(), mImage.getHeight(),
false);

ip::grayscale(mImage, &mImage);
ip::edgeDetectSobel(mImage, &mImageOutput);

Surface8u::Iter pixelIter = mImageOutput.
getIter(Area(1,1,mImageOutput.getWidth()-1,mImageOutput.
getHeight()-1));

while( pixelIter.line() ) {
    while( pixelIter.pixel() ) {
        if(pixelIter.getPos().x < mImageOutput.getWidth()
&& pixelIter.getPos().y <
mImageOutput.getHeight()
&& pixelIter.r() > 99) {
            float radius = 1.5f;
            float mass = Rand::randFloat(10.f, 20.f);
            float drag = 0.9f;
            Particle *particle = new Particle(
pixelIter.getPos(), radius, mass, drag );
mParticleSystem.addParticle( particle );
        }
    }
}
```

5. Implement the `update` method for your main class, as follows:

```
void MainApp::update() {
    for( std::vector<Particle*>::iterator it = mParticleSystem.
        particles.begin(); it != mParticleSystem.particles.end(); ++it )
    {

        if(mMouseDown) {
            Vec2f repulsionForce = (*it)->position - getMousePos();
            repulsionForce = repulsionForce.normalized() *
                math<float>::max(0.f, 100.f - repulsionForce.length());
            (*it)->forces += repulsionForce;
        }

        Vec2f alignForce = (*it)->anchor - (*it)->position;
        alignForce.limit(maxAlignSpeed);
        (*it)->forces += alignForce;
    }

    mParticleSystem.update();
}
```

6. Change the `draw` method for `Particle` inside the `Particle.cpp` source file to read as follows

```
void Particle::draw() {
    glBegin(GL_POINTS);
    glVertex2f(position);
    glEnd();
}
```

How it works...

The first major step was to allocate particles at some characteristic points of the image. To do so, we detected the edges, which was covered in the *Detecting edges* recipe in *Chapter 3, Using Image Processing Techniques*. In step 4 you can see that we iterated through each pixel of each processed image and placed particles only at detected features.

You can find another important calculation in step 5, where we tried to move back the particles to their original positions stored in the `anchor` property. To disorder particles, we used the same repulsion code that we used in the *Applying repulsion and attraction forces* recipe.



See also

- ▶ To learn more about fast Fourier transform, please visit http://en.wikipedia.org/wiki/Fast_Fourier_transform

Aligning particles to the mesh surface

In this recipe, we are going to use a 3D version of the particles' code base from the *Creating a particle system in 2D* recipe. To navigate in 3D space, we will use `MayaCamUI` covered in the *Using MayaCamUI* recipe in *Chapter 2, Preparing for Development*.

Getting ready

To simulate repulsion, we are using the code from the *Applying repulsion and attraction forces* recipe with slight modifications for three-dimensional space. For this example, we are using the `ducky.mesh` mesh file that you can find in the `resources` directory of the `Picking3D` sample inside the `Cinder` package. Please copy this file to the `assets` folder in your project.

How to do it...

We will create particles aligned to the mesh. Perform the following steps to do so:

1. Add an anchor property to the Particle class in the Particle.h file.
`ci::Vec3f anchor;`
2. Set the anchor value at the end of the Particle class constructor in the Particle.cpp source file.
`anchor = position;`

3. Add the necessary headers in your main class.

```
#include "cinder/TriMesh.h"
```

4. Add the new properties to your application's main class.

```
ParticleSystem mParticleSystem;
```

```
float repulsionFactor;
```

```
float maxAlignSpeed;
```

```
CameraPersp mCam;
```

```
MayaCamUI mMayaCam;
```

```
TriMesh mMesh;
```

```
Vec3f mRepPosition;
```

5. Set the default values inside the setup method.

```
repulsionFactor = -1.f;
```

```
maxAlignSpeed = 10.f;
```

```
mRepPosition = Vec3f::zero();
```

```
mMesh.read( loadAsset("ducky.msh") );
```

```
mCam.setPerspective(45.0f, getWindowAspectRatio(), 0.1, 10000);
```

```
mCam.setEyePoint(Vec3f(7.f,7.f,7.f));
```

```
mCam.setCenterOfInterestPoint(Vec3f::zero());
```

```
mMayaCam.setCurrentCam(mCam);
```

6. At the end of the setup method, add the following code snippet:

```
for(vector<Vec3f>::iterator it = mMesh.getVertices().begin(); it
!= mMesh.getVertices().end(); ++it) {
    float mass = Rand::randFloat(2.f, 15.f);
    float drag = 0.95f;
    Particle *particle = new Particle
    ( (*it), 0.f, mass, drag );
    mParticleSystem.addParticle( particle );
}
```

7. Add methods for camera navigation.

```
void MainApp::resize( ResizeEvent event ){
    mCam = mMayaCam.getCamera();
    mCam.setAspectRatio(getWindowAspectRatio());
    mMayaCam.setCurrentCam(mCam);
}

void MainApp::mouseDown(MouseEvent event){
    mMayaCam.mouseDown( event.getPos() );
}

void MainApp::mouseDrag( MouseEvent event ){
    mMayaCam.mouseDrag( event.getPos(), event.isLeftDown(),
        event.isMiddleDown(), event.isRightDown() );
}
```

8. Implement the update and draw methods for your main application class.

```
void MainApp::update() {

    mRepPosition.x = cos(getElapsedSeconds()) * 3.f;
    mRepPosition.y = sin(getElapsedSeconds()*2.f) * 3.f;
    mRepPosition.z = cos(getElapsedSeconds()*1.5f) * 3.f;

    for( std::vector<Particle*>::iterator it = mParticleSystem.
particles.begin(); it != mParticleSystem.particles.end(); ++it ) {

        Vec3f repulsionForce = (*it)->position - mRepPosition;
        repulsionForce = repulsionForce.normalized() *
        math<float>::max(0.f, 3.f - repulsionForce.length());
        (*it)->forces += repulsionForce;

        Vec3f alignForce = (*it)->anchor - (*it)->position;
        alignForce.limit(maxAlignSpeed);
        (*it)->forces += alignForce;
    }

    mParticleSystem.update();
}

void MainApp::draw()
{
    gl::enableDepthRead();
    gl::enableDepthWrite();
    gl::clear( Color::black() );
}
```

```
gl::setViewport(getWindowBounds());
gl::setMatrices(mMayaCam.getCamera());

gl::color(Color(1.f,0.f,0.f));
gl::drawSphere(mRepPosition, 0.25f);

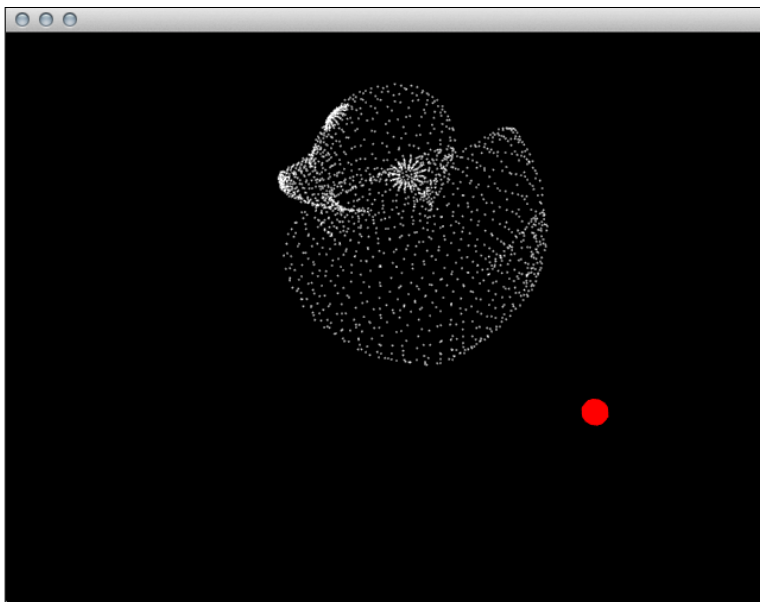
gl::color(Color::white());
mParticleSystem.draw();
}
```

9. Replace the draw method for Particle inside the Particle.cpp source file to read as follows

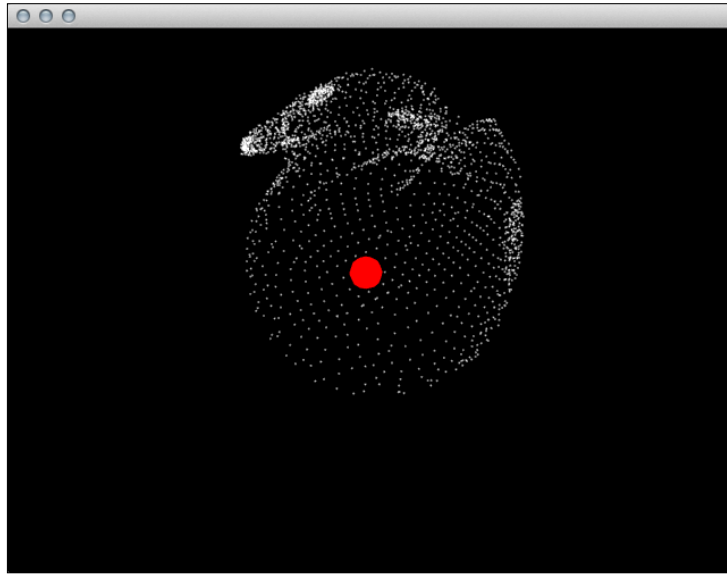
```
void Particle::draw(){
    glBegin(GL_POINTS);
    glVertex2f(position);
    glEnd();
}
```

How it works...

Firstly, we created particles in place of vertices of the mesh that you can see in step 6.



You can find another important calculation in step 8 where we tried to move particles back to their original positions stored in the `anchor` property. To displace the particles, we used the same repulsion code that we used in the *Applying repulsion and attraction forces* recipe but with slight modifications for three-dimensional space. Basically, it is about using `Vec3f` types instead of `Vec2f`.



Creating springs

In this recipe, we will learn how we can create springs.

Springs are objects that connect two particles and force them to be at a defined rest distance.

In this example, we will create random particles, and whenever the user presses a mouse button, two random particles will be connected by a new spring with a random rest distance.

Getting ready

We will be using the same particle system developed in the previous recipe, *Creating a particle system in 2D*. Create the `Particle` and `ParticleSystem` classes described in that recipe and include the `ParticleSystem.h` file at the top of the application source file.

We will be creating a `Spring` class, so it is necessary to create the following files:

- ▶ `Spring.h`
- ▶ `Spring.cpp`

How to do it...

We will create springs that constrain the movement of particles. Perform the following steps to do so:

1. In the `Spring.h` file, we will declare a `Spring` class. The first thing we need to do is to add the `#pragma once` macro and include the necessary files.

```
#pragma once
#include "Particle.h"
#include "cinder/gl/gl.h"
```

2. Next, declare the `Spring` class.

```
class Spring{

};
```

3. We will add member variables, two `Particle` pointers to reference the particles that will be connected by this spring, and the `rest` and `strength` float variables.

```
class Spring{
public:
    Particle *particleA;
    Particle *particleB;
    float strength, rest;
};
```

4. Now we will declare the constructor that will take pointers to two `Particle` objects, and the `rest` and `strength` values.

We will also declare the `update` and `draw` methods.

The following is the final `Spring` class declaration:

```
class Spring{
public:

    Spring( Particle *particleA, Particle *particleB,
           float rest, float strength );

    void update();
    void draw();

    Particle *particleA;
    Particle *particleB;
    float strength, rest;

};
```

5. Let's implement the `Spring` class in the `Spring.cpp` file.

In the constructor, we will set the values of the member values to the ones passed in the arguments.

```
Spring::Spring( Particle *particleA, Particle *particleB, float
rest, float strength ){
    this->particleA = particleA;
    this->particleB = particleB;
    this->rest = rest;
    this->strength = strength;
}
```

6. In the update method of the `Spring` class, we will calculate the difference between the particles' distance and the spring's rest distance, and adjust them accordingly.

```
void Spring::update(){
    ci::Vec2f delta = particleA->position - particleB->position;
    float length = delta.length();
    float invMassA = 1.0f / particleA->mass;
    float invMassB = 1.0f / particleB->mass;
    float normDist = ( length - rest ) / ( length * ( invMassA +
invMassB ) ) * strength;
    particleA->position -= delta * normDist * invMassA;
    particleB->position += delta * normDist * invMassB;
}
```

7. In the draw method of the `Spring` class, we will simply draw a line connecting both particles.

```
void Spring::draw(){
    ci::gl::drawLine
    ( particleA->position, particleB->position );
}
```

8. Now we will have to make some changes in the `ParticleSystem` class to allow the addition of springs.

In the `ParticleSystem` file, include the `Spring.h` file.

```
#include "Spring.h"
```

9. Declare the `std::vector<Spring*>` member in the class declaration.

```
std::vector<Spring*> springs;
```

10. Declare the `addSpring` and `destroySpring` methods to add and destroy springs to the system.

The following is the final `ParticleSystem` class declaration:

```
class ParticleSystem{
public:

    ~ParticleSystem();

    void update();
    void draw();

    void addParticle( Particle *particle );
    void destroyParticle( Particle *particle );
    void addSpring( Spring *spring );
    void destroySpring( Spring *spring );

    std::vector<Particle*> particles;
    std::vector<Spring*> springs;

};
```

11. Let's implement the `addSpring` method. In the `ParticleSystem.cpp` file, add the following code snippet:

```
void ParticleSystem::addSpring( Spring *spring ){
    springs.push_back( spring );
}
```

12. In the implementation of `destroySpring`, we will find the correspondent iterator for the argument `Spring` and remove it from `springs`. We will also delete the object.

Add the following code snippet in the `ParticleSystem.cpp` file:

```
void ParticleSystem::destroySpring( Spring *spring ){
    std::vector<Spring*>::iterator it = std::find( springs.begin(),
    springs.end(), spring );
    delete *it;
    springs.erase( it );
}
```

13. It is necessary to alter the `update` method to update all springs.

The following code snippet shows what the final `update` should look like:

```
void ParticleSystem::update(){
    for( std::vector<Particle*>::iterator it = particles.begin(); it
    != particles.end(); ++it ){
        (*it)->update();
    }
    for( std::vector<Spring*>::iterator it =
```



```
        springs.begin(); it != springs.end(); ++it ){
            (*it)->update();
        }
    }
```

14. In the draw method, we will also need to iterate over all springs and call the draw method on them.

The final implementation of the `ParticleSystem::draw` method should be as follows:

```
void ParticleSystem::draw(){
    for( std::vector<Particle*>::iterator it = particles.begin();
        it != particles.end(); ++it ){
        (*it)->draw();
    }
    for( std::vector<Spring*>::iterator it =
        springs.begin(); it != springs.end(); ++it ){
        (*it)->draw();
    }
}
```

15. We have finished creating the `Spring` class and making all necessary changes to the `ParticleSystem` class.

Let's go to our application's class and include the `ParticleSystem.h` file:

```
#include "ParticleSystem.h"
```

16. Declare a `ParticleSystem` object.

```
ParticleSystem mParticleSystem;
```

17. Create some random particles by adding the following code snippet to the `setup` method:

```
for( int i=0; i<100; i++ ){
    float x = randFloat( getWindowWidth() );
    float y = randFloat( getWindowHeight() );
    float radius = randFloat( 5.0f, 15.0f );
    float mass = radius;
    float drag = 0.9f;
    Particle *particle =
    new Particle( Vec2f( x, y ), radius, mass, drag );
    mParticleSystem.addParticle( particle );
}
```

18. In the `update` method, we will need to call the `update` method on `ParticleSystem`.

```
void MyApp::update() {
    mParticleSystem.update();
}
```

19. In the `draw` method, clear the background, define the window's matrices, and call the `draw` method on `mParticleSystem`.

```
void MyApp::draw() {
    gl::clear( Color( 0, 0, 0 ) );
    gl::setMatricesWindow( getWindowWidth(), getWindowHeight() );
    mParticleSystem.draw();
}
```

20. Since we want to create springs whenever the user presses the mouse, we will need to declare the `mouseDown` method.

Add the following code snippet to your application's class declaration:

```
void mouseDown( MouseEvent event );
```

21. In the `mouseDown` implementation we will connect two random particles.

Start by declaring a `Particle` pointer and defining it as a random particle in the particle system.

```
Particle *particleA = mParticleSystem.particles[ randInt(
mParticleSystem.particles.size() ) ];
```

22. Now declare a second `Particle` pointer and make it equal to the first one. In the `while` loop, we will set its value to a random particle in `mParticleSystem` until both particles are different. This will avoid the case where both pointers point to the same particle.

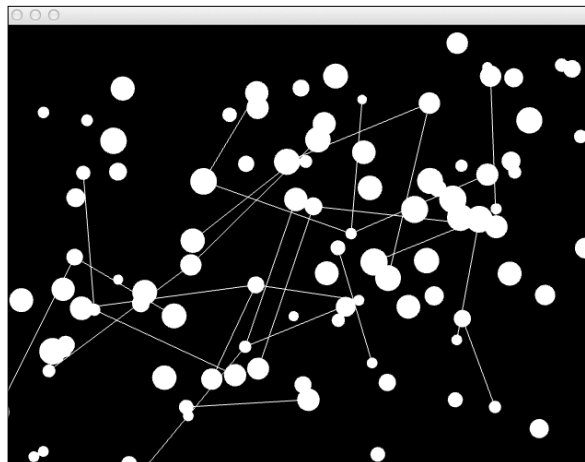
```
Particle *particleB = particleA;
while( particleB == particleA ){
    particleB = mParticleSystem.particles[ randInt( mParticleSystem.
particles.size() ) ];
}
```

23. Now we'll create a `Spring` object that will connect both particles, define a random rest distance, and set `strength` to `1.0`. Add the created spring to `mParticleSystem`.

The following is the final mouseDown implementation:

```
void SpringsApp::mouseDown( MouseEvent event )
{
    Particle *particleA = mParticleSystem.particles[
        randInt( mParticleSystem.particles.size() ) ];
    Particle *particleB = particleA;
    while( particleB == particleA ){
        particleB = mParticleSystem.particles[ randInt( mParticleSystem.
            particles.size() ) ];
    }
    float rest = randFloat( 100.0f, 200.0f );
    float strength = 1.0f;
    Spring *spring = new Spring
        ( particleA, particleB, rest, strength );
    mParticleSystem.addSpring( spring );
}
```

24. Build and run the application. Every time a mouse button is pressed, two particles will become connected with a white line and their distance will remain unchangeable.



How it works...

A `Spring` object will calculate the difference between two particles and correct their positions, so that the distance between the two particles will be equal to the springs' rest value.

By using their masses, we will also take into account each particle's mass, so that the correction will take into account the particles' weight.

There's more...

The same principle can also be applied to particle systems in 3D.

If you are using a 3D particle, as explained in the *There's more...* section of the *Creating a particle system in 2D* recipe, the `Spring` class simply needs to change its calculations to use `ci::Vec3f` instead of `ci::Vec2f`.

The update method of the `Spring` class would need to look like the following code snippet:

```
void Spring::update(){
    ci::Vec3f delta = particleA->position - particleB->position;
    float length = delta.length();
    float invMassA = 1.0f / particleA->mass;
    float invMassB = 1.0f / particleB->mass;
    float normDist = ( length - rest ) / ( length * ( invMassA +
    invMassB ) ) * strength;
    particleA->position -= delta * normDist * invMassA;
    particleB->position += delta * normDist * invMassB;
}
```


6

Rendering and Texturing Particle Systems

In this chapter we will learn about:

- ▶ Texturing particles
- ▶ Adding a tail to our particles
- ▶ Creating a cloth simulation
- ▶ Texturing a cloth simulation
- ▶ Texturing the particle system using point sprites and shaders
- ▶ Connecting particles

Introduction

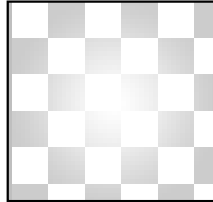
Continuing from *Chapter 5, Building Particle Systems*, we will learn how to render and apply textures to our particles in order to make them more appealing.

Texturing particles

In this recipe we will render particles introduced in the previous chapter using texture loaded from the PNG file.

Getting started

This recipe code base is an example of the recipe *Simulating particles flying on the wind* from *Chapter 5, Building Particle Systems*. We also need a texture for a single particle. You can prepare one easily with probably any graphical program. For this example, we are going to use a PNG file with transparency stored inside the `assets` folder with a name, `particle.png`. In this case it is just a radial gradient with transparency.



How to do it...

We will render particles using the previously created texture.

1. Include the necessary header files:

```
#include "cinder/gl/Texture.h"
#include "cinder/ImageIo.h"
```

2. Add a member to the application main class:

```
gl::Texture particleTexture;
```

3. Inside the `setup` method load `particleTexture`:

```
particleTexture=gl::Texture(loadImage(loadAsset("particle.png")));
```

4. We also have to change the particle size for this example:

```
float radius = Rand::randFloat(2.f, 10.f);
```

5. At the end of the `draw` method we will draw our particles as follows:

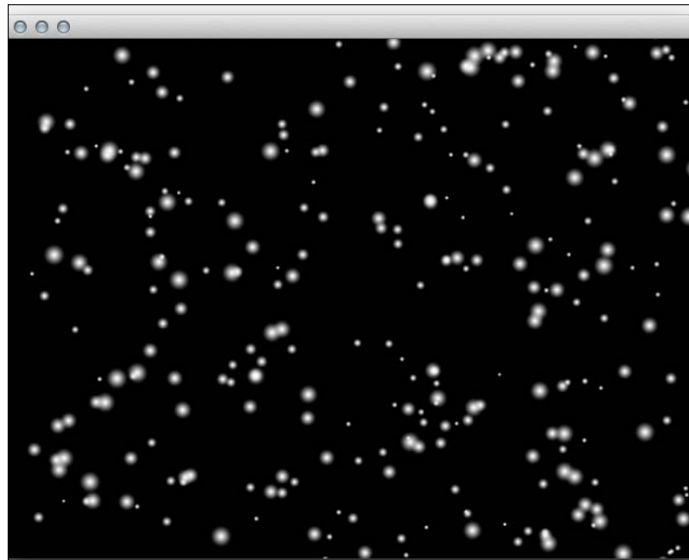
```
gl::enableAlphaBlending();
particleTexture.enableAndBind();
gl::color(ColorA::white());
mParticleSystem.draw();
```

6. Replace the `draw` method inside the `Particle.cpp` source file with the following code:

```
void Particle::draw() {
    ci::gl::drawSolidRect(ci::Rectf(position.x-radius, position.y-
    radius,
    position.x+radius, position.y+radius));
}
```

How it works...

In step 5, we saw two important lines. One enables alpha blending and the other binds our texture stored in the `particleTexture` property. If you look at step 6, you can see we drew each particle as a rectangle and each rectangle had texture applied. It is a simple way of texturing particles and not very performance effective, but in this case, it works quite well. It is possible to change the color of drawing particles by changing the color just before invoking the `draw` method on `ParticleSystem`.



See also

Look into the recipe *Texturing the particle system using Point sprites and shaders*

Adding a tail to our particles

In this recipe, we will show you how to add a tail to the particle animation.

Getting started

In this recipe we are going to use the code base from the recipe *Applying repulsion and attraction forces* from *Chapter 5, Building Particle Systems*.

How to do it...

We will add a tail to the particles using different techniques.

Drawing history

Simply replace the draw method with the following code:

```
void MainApp::draw()
{
    gl::enableAlphaBlending();
    gl::setViewport(getWindowBounds());
    gl::setMatricesWindow(getWindowWidth(), getWindowHeight());

    gl::color( ColorA(0.f,0.f,0.f, 0.05f) );
    gl::drawSolidRect(getWindowBounds());
    gl::color( ColorA(1.f,1.f,1.f, 1.f) );
    mParticleSystem.draw();
}
```

Tail as a line

We will add a tail constructed from several lines.

1. Add new properties to the Particle class inside the Particle.h header file:

```
std::vector<ci::Vec2f> positionHistory;
int tailLength;
```
2. At the end of the Particle constructor, inside the Particle.cpp source file, set the default value to the tailLength property:

```
tailLength = 10;
```
3. At the end of the update method of the Particle class add the following code:

```
positionHistory.push_back(position);
if(positionHistory.size() >tailLength) {
    positionHistory.erase( positionHistory.begin() );
}
```
4. Replace your Particle::draw method with the following code:

```
void Particle::draw(){
    glBegin( GL_LINE_STRIP );
    for( int i=0; i<positionHistory.size(); i++ ){
        float alpha = (float)i/(float)positionHistory.size();
```

```
ci::gl::color( ci::ColorA(1.f,1.f,1.f, alpha));
ci::gl::vertex( positionHistory[i] );
}
glEnd();

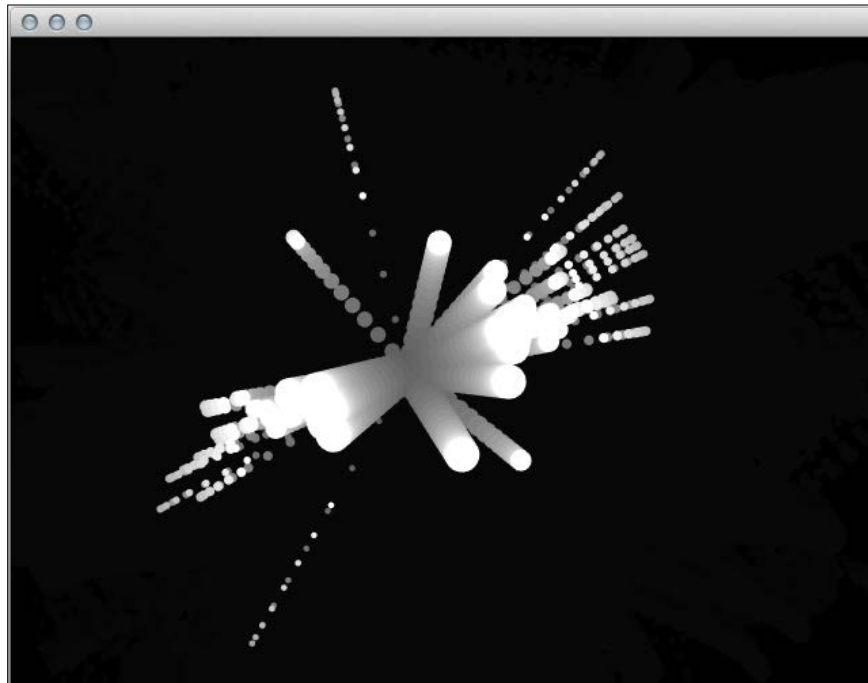
ci::gl::color( ci::ColorA(1.f,1.f,1.f, 1.f) );
ci::gl::drawSolidCircle( position, radius );
}
```

How it works...

Now, we will explain how each technique works.

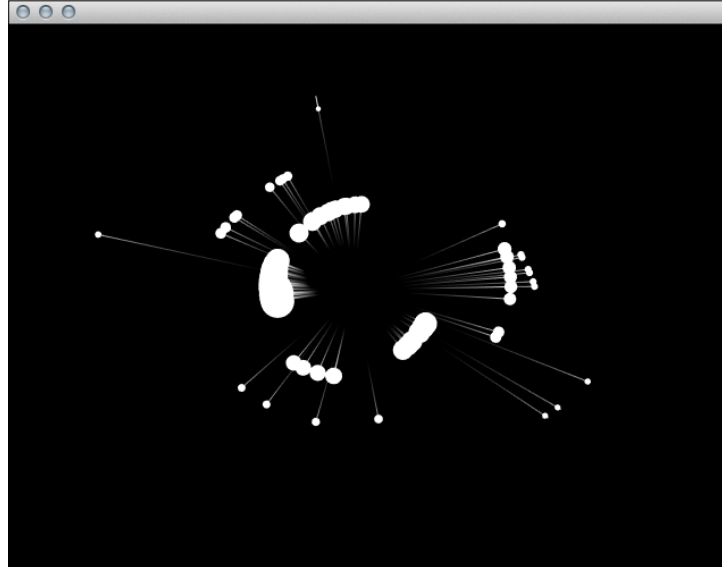
Drawing history

The idea behind this method is very simple, instead of clearing the drawing area, we are continuously drawing semi-transparent rectangles that cover old drawing states more and more. This very simple method can give you interesting effects with particles. You can also manipulate the opacity of each rectangle by changing the alpha channel of the rectangle color, which becomes a color of the background.



Tail as a line

To draw a tail with lines, we have to store several particle positions and draw a line through these locations with variable opacity. The rule for opacity is just to draw older locations with less opacity. You can see the drawing code and alpha channel calculation in step 4



Creating a cloth simulation

In this recipe we will learn how to simulate cloth by creating a grid of particles connected by springs.

Getting Ready

In this recipe, we will be using the particle system described in the recipe *Creating a particle system in 2D* from *Chapter 5, Building Particle Systems*.

We will also be using the `Springs` class created in the recipe *Creating springs* from *Chapter 5, Building Particle Systems*.

So, you will need to add the following files to your project:

- ▶ `Particle.h`
- ▶ `ParticleSystem.h`
- ▶ `Spring.h`
- ▶ `Spring.cpp`

How to do it...

We will create a grid of particles connected with springs to create a cloth simulation.

1. Include the particle system file in your project by adding the following code on top of your source file:

```
#include "ParticleSystem.h"
```

2. Add the `using` statements before the application class declaration as shown in the following code:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

3. Create an instance of a `ParticleSystem` object and member variables to store the top corners of the grid. We will also create variables to store the number of rows and lines that make up our grid. Add the following code in your application class:

```
ParticleSystem mParticleSystem;
Vec2f mLeftCorner,
mRightCorner;
int mNumRows, mNumLines;
```

4. Before we start creating our particle grid, let's update and draw our particle system in our application's update and draw methods.

```
void MyApp::update() {
    mParticleSystem.update();
}

void MyApp::draw() {
    gl::clear( Color( 0, 0, 0 ) );
    mParticleSystem.draw();
}
```

5. In the `setup` method, let's initialize the grid corner positions and number of rows and lines. Add the following code at the top of the `setup` method:

```
mLeftCorner = Vec2f( 50.0f, 50.0f );
mRightCorner = Vec2f( getWindowWidth() - 50.0f, 50.0f );
mNumRows = 20;
mNumLines = 15;
```

6. Calculate the distance between each particle on the grid.

```
float gap = ( mRightCorner.x - mLeftCorner.x ) / ( mNumRows-1 );
```

7. Let's create a grid of evenly spaced particles and add them to `ParticleSystem`. We'll do this by creating a nested loop where each loop index will be used to calculate the particle's position. Add the following code in the `setup` method:

```
for( int i=0; i<mNumRows; i++ ){
for( int j=0; j<mNumLines; j++ ){
float x = mLeftCorner.x + ( gap * i );
float y = mLeftCorner.y + ( gap * j );
Particle *particle = new Particle( Vec2f( x, y ), 5.0f, 5.0f,
0.95f );
mParticleSystem.addParticle( particle );
}
}
```

8. Now that the particles are created, we need to connect them with springs. Let's start by connecting each particle to the one directly below it. In a nested loop, we will calculate the index of the particle in `ParticleSystem` and the one below it. We then create a `Spring` class connecting both particles using their current distance as `rest` and a strength value of `1.0`. Add the following to the bottom of the `setup` method:

```
for( int i=0; i<mNumRows; i++ ){
for( int j=0; j<mNumLines-1; j++ ){
int indexA = i * mNumLines + j;
int indexB = i * mNumLines + j + 1;
Particle *partA = mParticleSystem.particles[ indexA ];
Particle *partB = mParticleSystem.particles[ indexB ];
float rest = partA->position.distance( partB->position );
Spring *spring = new Spring( partA, partB, rest, 1.0f
);
mParticleSystem.addSpring( spring );
}
}
```

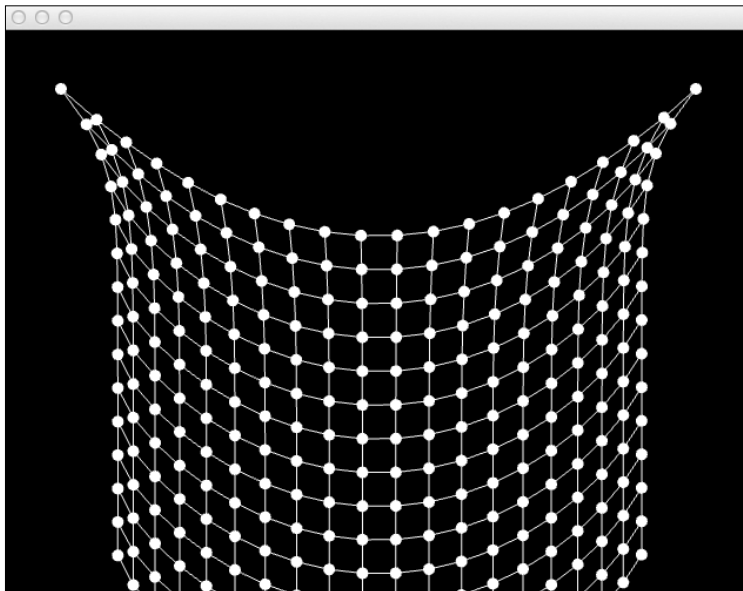
9. We now have a static grid made out of particles and springs. Let's add some gravity by applying a constant vertical force to each particle. Add the following code at the bottom of the `update` method:

```
Vec2f gravity( 0.0f, 1.0f );
for( vector<Particle*>::iterator it = mParticleSystem.particles.
begin(); it != mParticleSystem.particles.end(); ++it ){
(*it)->forces += gravity;
}
}
```

10. To prevent the grid from falling down, we need to make the particles at the top edges static in their initial positions, defined by `mLeftCorner` and `mRightCorner`. Add the following code to the `update` method:

```
int topLeftIndex = 0;
int topRightIndex = ( mNumRows-1 ) * mNumLines;
mParticleSystem.particles[ topLeftIndex ]->position = mLeftCorner;
mParticleSystem.particles[ topRightIndex ]->position =
mRightCorner;
```

11. Build and run the application; you'll see a grid of particles falling down with gravity, locked by its top corners.



12. Let's add some interactivity by allowing the user to drag particles with the mouse. Declare a `Particle` pointer to store the particle being dragged.

```
Particle *mDragParticle;
```

13. In the `setup` method initialize the particle to `NULL`.

```
mDragParticle = NULL;
```

14. Declare the `mouseUp` and `mouseDown` methods in the application's class declaration.

```
void mouseDown( MouseEvent event );  
void mouseUp( MouseEvent event );
```

15. In the implementation of the `mouseDown` event, we iterate the overall particles and, if a particle is under the cursor, we set `mDragParticle` to point to it.

```
void MyApp::mouseDown( MouseEvent event ){  
    for( vector<Particle*>::iterator it = mParticleSystem.particles.  
        begin(); it != mParticleSystem.particles.end(); ++it ){  
        Particle *part = *it;  
        float dist = part->position.distance( event.getPos() );  
        if( dist < part->radius ){  
            mDragParticle = part;  
            return;  
        }  
    }  
}
```

16. In the `mouseUp` event we simply set `mDragParticle` to `NULL`.

```
void MyApp::mouseUp( MouseEvent event ){  
    mDragParticle = NULL;  
}
```

17. We need to check if `mDragParticle` is a valid pointer and set the particle's position to the mouse cursor. Add the following code to the `update` method:

```
if( mDragParticle != NULL ){  
    mDragParticle->position = getMousePos();  
}
```

18. Build and run the application. Press and drag the mouse over any particle and drag it around to see how the cloth simulation reacts.

How it works...

The cloth simulation is achieved by creating a two dimensional grid of particles and connecting them with springs. Each particle will be connected with a spring to the ones next to it and to the ones above and below it.

There's more...

The density of the grid can be changed to accommodate the user's needs. Using a grid with more particles will generate a more precise simulation but will be slower.

Change `mNumLines` and `mNumRows` to change the number of particles that make up the grid.

Texturing a cloth simulation

In this recipe, we will learn how to apply a texture to the cloth simulation we created in the *Creating a cloth simulation* recipe of the current chapter.

Getting ready

We will be using the cloth simulation developed in the recipe *Creating a cloth Simulation* as the base for this recipe.

You will also need an image to use as texture; place it inside your `assets` folder. In this recipe we will name our image `texture.jpg`.

How to do it...

We will calculate the correspondent texture coordinate to each particle in the cloth simulation and apply a texture.

1. Include the necessary files to work with the texture and read images.


```
#include "cinder/gl/Texture.h"
#include "cinder/ImageIo.h"
```
2. Declare a `ci::gl::Texture` object in your application's class declaration.


```
gl::Texture mTexture;
```
3. In the `setup` method load the image.


```
mTexture = loadImage( loadAsset( "image.jpg" ) );
```
4. We will remake the `draw` method. So we'll erase everything in it which was changed in the *Creating a cloth simulation* recipe and apply the `clear` method. Your `draw` method should be like the following:

```
void MyApp::draw() {
    gl::clear( Color( 0, 0, 0 ) );
}
```


5. After the `clear` method call, enable the `VERTEX` and `TEXTURE COORD` arrays and bind the texture. Add the following to the `draw` method:

```
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
mTexture.enableAndBind();
```

6. We will now iterate over all particles and springs that make up the cloth simulation grid and draw a textured triangle strip between each row and the row next to it. Start by creating a `for` loop with `mNumRows-1` iterations and create two `std::vector<Vec2f>` containers to store vertex and texture coordinates.

```
for( int i=0; i<mNumRows-1; i++ ){
    vector<Vec2f>vertexCoords, textureCoords;
}
```

7. Inside the loop we will create a nested loop that will iterate over all lines in the cloth grid. In this loop we will calculate the index of the particles whose vertices will be drawn, calculate their correspondent texture coordinates, and add them with the positions of `textureCoords` and `vertexCoords`. Type the following code into the loop that we created in the previous step:

```
or( int j=0; j<mNumLines; j++ ){
    int indexTopLeft = i * mNumLines + j;
    int indexTopRight = ( i+1 ) * mNumLines + j;
    Particle *left = mParticleSystem.particles[ indexTopLeft ];
    Particle *right = mParticleSystem.particles[indexTopRight ];
    float texX = ( (float)i / (float)(mNumRows-1) ) * mTexture.
getRight();
    float texY = ( (float)j / (float)(mNumLines-1) ) * mTexture.
getBottom();
    textureCoords.push_back( Vec2f( texX, texY ) );
    vertexCoords.push_back( left->position );
    texX = ( (float)(i+1) / (float)(mNumRows-1) ) * mTexture.
getRight();
    textureCoords.push_back( Vec2f( texX, texY ) );
    vertexCoords.push_back( right->position );
}
```

Now that the vertex and texture coordinates are calculated and placed inside `vertexCoords` and `textureCoords` we will draw them. Here is the complete nested loop:

```
for( int i=0; i<mNumRows-1; i++ ){
    vector<Vec2f> vertexCoords, textureCoords;
    for( int j=0; j<mNumLines; j++ ){
```

```

int indexTopLeft = i * mNumLines + j;
int indexTopRight = ( i+1 ) * mNumLines + j;
Particle *left = mParticleSystem.particles[ indexTopLeft ];
Particle *right = mParticleSystem.particles[ indexTopRight ];
float texX = ( (float)i / (float)(mNumRows-1) ) * mTexture.
getRight();
float texY = ( (float)j / (float)(mNumLines-1) ) * mTexture.
getBottom();
textureCoords.push_back( Vec2f( texX, texY ) );
vertexCoords.push_back( left->position );
texX = ( (float)(i+1) / (float)(mNumRows-1) ) * mTexture.
getRight();
textureCoords.push_back( Vec2f( texX, texY ) );
vertexCoords.push_back( right->position );
}
glVertexPointer 2, GL_FLOAT, 0, &vertexCoords[0] );
glTexCoordPointer( 2, GL_FLOAT, 0, &textureCoords[0] );
glDrawArrays( GL_TRIANGLE_STRIP, 0, vertexCoords.size() );
}

```

8. Finally we need to unbind `mTexture` by adding the following:

```
mTexture.unbind();
```

How it works...

We calculated the correspondent texture coordinate according to the particle's position on the grid. We then drew our texture as triangular strips formed by the particles on a row with the particles on the row next to it.

Texturing a particle system using point sprites and shaders

In this recipe, we will learn how to apply a texture to all our particles using OpenGL point sprites and a GLSL Shader.

This method is optimized and allows for a large number of particles to be drawn at fast frame rates.

Getting ready

We will be using the particle system developed in the recipe *Creating a particle system in 2D* from *Chapter 5, Building Particle Systems*. So we will need to add the following files to your project:

- ▶ Particle.h
- ▶ ParticleSystem.h

We will also be loading an image to use as texture. The image's size must be a power of two, such as 256 x 256 or 512 x 512. Place the image inside the `assets` folder and name it `particle.png`.

How to do it...

We will create a GLSL shader and then enable OpenGL point sprites to draw textured particles.

1. Let's begin by creating the GLSL Shader. Create the following files:

- `shader.frag`
- `shader.vert`

Add them to the `assets` folder.

2. Open the file `shader.frag` in your IDE of choice and declare a uniform `sampler2D`:

```
uniform sampler2D tex;
```

3. In the `main` function we use the texture to define the fragment color. Add the following code:

```
void main (void) {  
    gl_FragColor = texture2D(tex, gl_TexCoord[0].st) * gl_Color;  
}
```

4. Open the `shader.vert` file and create `float` attribute to store the particle's radiuses. In the `main` method we define the position, color, and point size attributes. Add the following code:

```
attribute float particleRadius;  
void main(void)  
{  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
    gl_PointSize = particleRadius;  
    gl_FrontColor = gl_Color;  
}
```

5. Our shader is done! Let's go to our application source file and include the necessary files. Add the following code to your application source file:

```
#include "cinder/gl/Texture.h"
#include "cinder/ImageIo.h"
#include "cinder/Rand.h"
#include "cinder/gl/GlslProg.h"
#include "ParticleSystem.h"
```

6. Declare the member variables to create a particle system and arrays to store the particle's positions and radiuses. Also declare a variable to store the number of particles.

```
ParticleSystem mParticleSystem;
int mNumParticles;
Vec2f *mPositions;
float *mRadiuses;
```

7. In the setup method, let's initialize mNumParticles to 1000 and allocate the arrays. We will also create the random particles.

```
mNumParticles = 1000;
mPositions = new Vec2f[ mNumParticles ];
mRadiuses = new float[ mNumParticles ];

for( int i=0; i<mNumParticles; i++ ){
    float x = randFloat( 0.0f, getWindowWidth() );
    float y = randFloat( 0.0f, getWindowHeight() );
    float radius = randFloat( 5.0f, 50.0f );
    Particle *particle = new Particle( Vec2f( x, y ), radius, 1.0f,
    0.9f );
    mParticleSystem.addParticle( particle );
}
mParticleSystem.addParticle( particle );
```

8. In the update method, we will update mParticleSystem and the mPositions and mRadiuses arrays. Add the following code to the update method:

```
mParticleSystem.update();
for( int i=0; i<mNumParticles; i++ ){
    mPositions[i] = mParticleSystem.particles[i]->position;
    mRadiuses[i] = mParticleSystem.particles[i]->radius;
}
```

9. Declare the shaders and the particle's texture.

```
gl::Texture mTexture;
gl::GlslProg mShader;
```

10. Load the shaders and texture by adding the following code in the `setup` method:

```
mTexture = loadImage( loadAsset( "particle.png" ) );  
mShader = gl::GslProg( loadAsset( "shader.vert" ), loadAsset( "shader.frag" ) );
```

11. In the `draw` method, we will start by clearing the background with black, set the window's matrices, enable the additive blend, and bind the shader.

```
gl::clear( Color( 0, 0, 0 ) );  
gl::setMatricesWindow( getWindowWidth(), getWindowHeight() );  
gl::enableAdditiveBlending();  
mShader.bind();
```

12. Get the location for the `particleRadius` attribute in the Vertex shader. Enable vertex attribute arrays and set the pointer to `mRadiuses`.

```
GLint particleRadiusLocation = mShader.getAttribLocation( "particleRadius" );  
glEnableVertexAttribArray( particleRadiusLocation );  
glVertexAttribPointer( particleRadiusLocation, 1, GL_FLOAT, false, 0, mRadiuses );
```

13. Enable point sprites and enable our shader to write to point sizes.

```
glEnable( GL_POINT_SPRITE );  
glTexEnvi( GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE );  
glEnable( GL_VERTEX_PROGRAM_POINT_SIZE );
```

14. Enable vertex arrays and set the vertex pointer to `mPositions`.

```
glEnableClientState( GL_VERTEX_ARRAY );  
glVertexPointer( 2, GL_FLOAT, 0, mPositions );
```

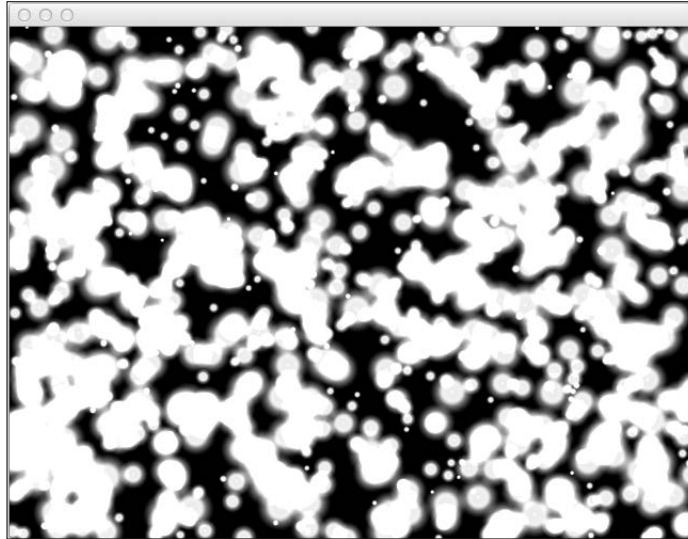
15. Now enable and bind the texture, draw the vertex array as points, and unbind the texture.

```
mTexture.enableAndBind();  
glDrawArrays( GL_POINTS, 0, mNumParticles );  
mTexture.unbind();
```

16. All we need to do now is disable the vertex arrays, disable the vertex attribute arrays, and unbind the shader.

```
glDisableClientState( GL_VERTEX_ARRAY );  
glDisableVertexAttribArrayARB( particleRadiusLocation );  
mShader.unbind();
```

17. Build and run the application and you will see 1000 random particles with the applied texture.



How it works...

Point sprites is a nice feature of OpenGL that allows for the application of an entire texture to a single point. It is extremely useful when drawing particle systems and is quite optimized, since it reduces the amount of information sent to the graphics card and performs most of the calculations on the GPU.

In the recipe we also created a GLSL shader, a high-level programming language, that allows more control over the programming pipeline, to define individual point sizes for each particle.

In the `update` method we updated the `Positions` and `Radiuses` arrays, so that if the particles are animated the arrays will represent the correct values.

There's more...

Point sprites allow us to texturize points in 3D space. To draw the particle system in 3D do the following:

1. Use the `Particle` class described in the *There's more...* section of the recipe *Creating a Particle system in 2D* from Chapter 5, *Building Particle Systems*.
2. Declare and initialize `mPositions` as a `ci::Vec3f` array.

3. In the `draw` method, indicate that the vertex pointer contains 3D information by applying the following change:

```
glVertexPointer(2, GL_FLOAT, 0, mPositions);
```

Change the previous code line to:

```
glVertexPointer(3, GL_FLOAT, 0, mPositions);
```

4. The vertex shader needs to adjust the point size according to the depth of the particle. The `shader.vert` file would need to read the following code:

```
attribute float particleRadius;

void main(void)
{
    vec4eyeCoord = gl_ModelViewMatrix * gl_Vertex;
    gl_Position = gl_ProjectionMatrix * eyeCoord;
    float distance = sqrt(eyeCoord.x*eyeCoord.x +
eyeCoord.y*eyeCoord.y + eyeCoord.z*eyeCoord.z);
    float attenuation = 3000.0 / distance;
    gl_PointSize = particleRadius * attenuation;
    gl_FrontColor = gl_Color;
}
```

Connecting the dots

In this recipe we will show how to connect particles with lines and introduce another way of drawing particles.

Getting started

This recipe's code base is an example from the recipe *Simulating particles flying on the wind* (from *Chapter 5, Building Particle Systems*), so please refer to this recipe.

How to do it...

We will connect particles rendered as circles with lines.

1. Change the number of particles to create inside the `setup` method:

```
int numParticle = 100;
```

2. We will calculate radius and mass of each particle as follows:

```
float radius = Rand::randFloat(2.f, 5.f);
float mass = radius*2.f;
```

3. Replace the draw method inside the Particle.cpp source file with the following:

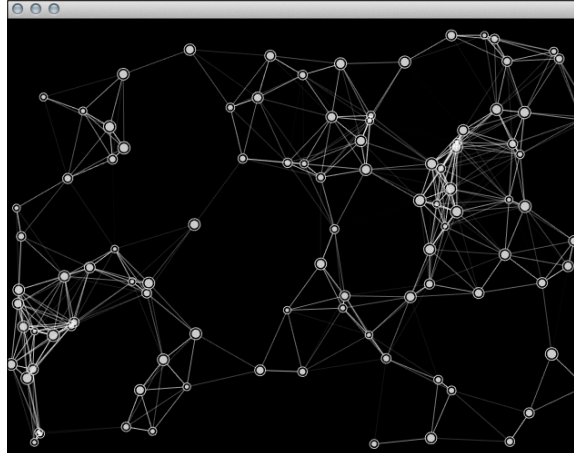
```
void Particle::draw(){
    ci::gl::drawSolidCircle(position, radius);
    ci::gl::drawStrokedCircle(position, radius+2.f);
}
```

4. Replace the draw method inside the ParticleSystem.cpp source file as follows:

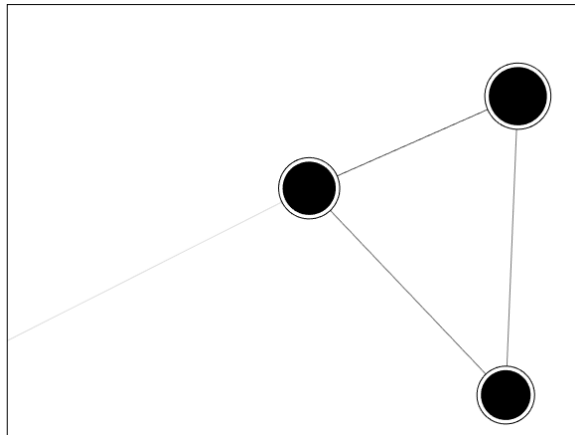
```
void ParticleSystem::draw(){
    gl::enableAlphaBlending();
    std::vector<Particle*>::iterator it;
    for(it = particles.begin(); it != particles.end(); ++it){
        std::vector<Particle*>::iterator it2;
        for(it2=particles.begin(); it2!= particles.end(); ++it2){
            float distance = (*it)->position.distance(
                (*it2)->position );
            float per = 1.f - (distance / 100.f);
            ci::gl::color( ci::ColorA(1.f,1.f,1.f, per*0.8f) );
            ci::Vec2f conVec = (*it2)->position-(*it)->position;
            conVec.normalize();
            ci::gl::drawLine(
                (*it)->position+conVec * ((*it)->radius+2.f),
                (*it2)->position-conVec * ((*it2)->radius+2.f ));
        }
    }
    ci::gl::color( ci::ColorA(1.f,1.f,1.f, 0.8f) );
    std::vector<Particle*>::iterator it3;
    for(it3 = particles.begin(); it3!= particles.end(); ++it3){
        (*it3)->draw();
    }
}
```


How it works...

The most interesting part of this example is mentioned in step 4. We are iterating through all the points, actually through all possible pairs of the points, to connect it with a line and apply the right opacity. The opacity of the line connecting two particles is calculated from the distance between these two particles; the longer distance makes the connection line more transparent.



Have a look at how the particles are been drawn in step 3. They are solid circles with a slightly bigger outer circle. The nice detail is the connection line that we are drawing between particles that stick to the edge of the outer circle, but don't cross it. We have done it in step 4, where we calculated the normalized vector of the vectors connecting two particles, then used them to move the attachment point towards that vector, multiplied by the outer circle radius.



Connecting particles with spline

In this recipe we are going to learn how to connect particles with splines in 3D.

Getting started

In this recipe we are going to use the particle's code base from the recipe *Creating a particle system*, from *Chapter 5, Building Particle Systems*. We are going to use the 3D version.

How to do it...

We will create splines connecting particles.

1. Include the necessary header file inside `ParticleSystem.h`:

```
#include "cinder/BSpline.h"
```
2. Add a new property to the `ParticleSystem` class:

```
ci::BSpline3f spline;
```
3. Implement the `computeBSpline` method for the `ParticleSystem` class:

```
void ParticleSystem::computeBSpline() {
    std::vector<ci::Vec3f> splinePoints;
    std::vector<Particle*>::iterator it;
    for(it = particles.begin(); it != particles.end(); ++it) {
        ++it;
        splinePoints.push_back( ci::Vec3f( (*it)->position ) );
    }
    spline = ci::BSpline3f( splinePoints, 3, false, false );
}
```
4. At the end of the `ParticleSystem` update method, invoke the following code:

```
computeBSpline();
```
5. Replace the draw method of `ParticleSystem` with the following:

```
void ParticleSystem::draw() {
    ci::gl::color(ci::Color::black());
    if(spline.isUniform()) {
        glBegin(GL_LINES);
        float step = 0.001f;
        for( float t = step; t <1.0f; t += step ) {
            ci::gl::vertex( spline.getPosition( t-step ) );
            ci::gl::vertex( spline.getPosition( t ) );
        }
    }
}
```

```
        glEnd();
    }
    ci::gl::color(ci::Color(0.0f,0.0f,1.0f));
    std::vector<Particle*>::iterator it;
    for(it = particles.begin(); it != particles.end(); ++it ){
        (*it)->draw();
    }
}
```

6. Add headers to your main Cinder application class files:

```
#include "cinder/app/AppBasic.h"
#include "cinder/gl/Texture.h"
#include "cinder/Rand.h"
#include "cinder/Surface.h"
#include "cinder/MayaCamUI.h"
#include "cinder/BSpline.h"

#include "ParticleSystem.h"
```

7. Add members for your main class:

```
ParticleSystem mParticleSystem;

float repulsionFactor;
float maxAlignSpeed;

CameraPersp      mCam;
MayaCamUI mMayaCam;

Vec3f mRepPosition;

BSpline3f spline;
```

8. Implement the setup method as follows:

```
void MainApp::setup()
{
    repulsionFactor = -1.0f;
    maxAlignSpeed = 10.f;
    mRepPosition = Vec3f::zero();

    mCam.setPerspective(45.0f, getWindowAspectRatio(), 0.1, 10000);
    mCam.setEyePoint(Vec3f(7.f,7.f,7.f));
    mCam.setCenterOfInterestPoint(Vec3f::zero());
}
```

```

mMayaCam.setCurrentCam(mCam);
vector<Vec3f> splinePoints;
float step = 0.5f;
float width = 20.f;
for (float t = 0.f; t < width; t += step) {
    float mass = Rand::randFloat(20.f, 25.f);
    float drag = 0.95f;
    splinePoints.push_back( Vec3f(math<float>::cos(t),
    math<float>::sin(t),
    t - width*0.5f) );
    Particle *particle;
    particle = new Particle(
        Vec3f( math<float>::cos(t)+Rand::randFloat(-0.8f,0.8f),
        math<float>::sin(t)+Rand::randFloat(-0.8f,0.8f),
        t - width*0.5f),
        1.f, mass, drag );
    mParticleSystem.addParticle( particle );
}
spline = BSpline3f( splinePoints, 3, false, false );
}

```

9. Add members for camera navigation:

```

void MainApp::resize( ResizeEvent event ){
    mCam = mMayaCam.getCamera();
    mCam.setAspectRatio(getWindowAspectRatio());
    mMayaCam.setCurrentCam(mCam);
}

void MainApp::mouseDown(MouseEvent event){
    mMayaCam.mouseDown( event.getPos() );
}

void MainApp::mouseDrag( MouseEvent event ){
    mMayaCam.mouseDrag( event.getPos(), event.isLeftDown(), event.
isMiddleDown(), event.isRightDown() );
}

```

10. Implement the update method as follows:

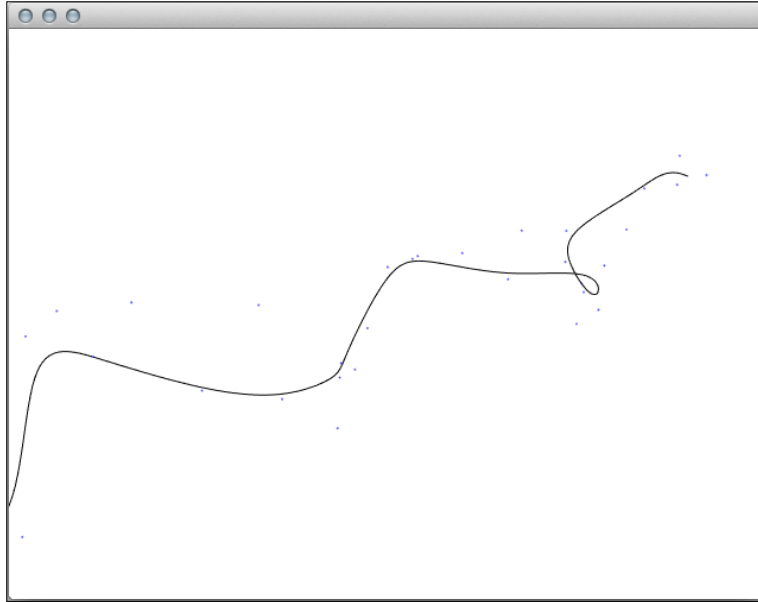
```
void MainApp::update() {
    float pos=math<float>::abs(sin(getElapsedSeconds()*0.5f));
    mRepPosition = spline.getPosition( pos );
    std::vector<Particle*>::iterator it;
    it = mParticleSystem.particles.begin();
    for(; it != mParticleSystem.particles.end(); ++it ) {
        Vec3f repulsionForce = (*it)->position - mRepPosition;
        repulsionForce = repulsionForce.normalized() *
            math<float>::max(0.f, 3.f-repulsionForce.length());
        (*it)->forces += repulsionForce;
        Vec3f alignForce = (*it)->anchor - (*it)->position;
        alignForce.limit(maxAlignSpeed);
        (*it)->forces += alignForce;
    }
    mParticleSystem.update();
}
```

11. Implement the draw method as follows:

```
void MainApp::draw() {
    gl::enableDepthRead();
    gl::enableDepthWrite();
    gl::clear( Color::white() );
    gl::setViewport( getWindowBounds() );
    gl::setMatrices( mMayaCam.getCamera() );
    gl::color( Color(1.f,0.f,0.f) );
    gl::drawSphere( mRepPosition, 0.25f );
    mParticleSystem.draw();
}
```

How it works...

B-spline lets us draw a very smooth curved line through some given points, in our case, particle positions. We can still apply some attraction and repulsion forces so that the line behaves quite like a spring. In Cinder, you can use B-splines in 2D and 3D space and calculate them with the `BSpline` class.



See also

More details about B-spline are available at <http://en.wikipedia.org/wiki/B-spline>.

7

Using 2D Graphics

In this chapter, we will learn how to work and draw with 2D graphics and built-in Cinder tools.

The recipes in this chapter will cover the following:

- ▶ Drawing 2D geometric primitives
- ▶ Drawing arbitrary shapes with the mouse
- ▶ Implementing a scribbler algorithm
- ▶ Implementing 2D metaballs
- ▶ Animating text around curves
- ▶ Adding a blur effect
- ▶ Implementing a force-directed graph

Drawing 2D geometric primitives

In this recipe, we will learn how to draw the following 2D geometric shapes, as filled and stroked shapes:

- ▶ Circle
- ▶ Ellipse
- ▶ Line
- ▶ Rectangle

Getting ready

Include the necessary header to draw in OpenGL using Cinder commands.

Add the following line of code at the top of your source file:

```
#include "cinder/gl/gl.h"
```

How to do it...

We will create several geometric primitives using Cinder's methods for drawing in 2D. Perform the following steps to do so:

1. Let's begin by declaring member variables to keep information about the shapes we will be drawing.

Create two `ci::Vec2f` objects to store the beginning and end of a line, a `ci::Rectf` object to draw a rectangle, a `ci::Vec2f` object to define the center of the circle, and a `float` object to define its radius. Finally, we will create `aci::Vec2f` to define the ellipse's radius and two `float` objects to define its width and height.

Let's also declare two `ci::Color` objects to define the stroke and fill colors.

```
Vec2f mLineBegin, mLineEnd;  
Rect  fmRect;  
Vec2f mCircleCenter;  
float mCircleRadius;  
Vec2f mEllipseCenter;  
float mEllipseWidth, mEllipseHeight;  
Color mFillColor, mStrokeColor;
```

2. In the `setup` method, let's initialize the preceding members:

```
mLineBegin = Vec2f( 10, 10 );  
mLineEnd = Vec2f( 400, 400 );  
  
mCircleCenter = Vec2f( 500, 200 );  
mCircleRadius = 100.0f;  
  
mEllipseCenter = Vec2f( 200, 300 );  
mEllipseWidth = 200.0f;  
ellipseHeight = 100.0f;  
  
mRect = Rectf( Vec2f( 40, 20 ), Vec2f( 300, 100 ) );  
  
mFillColor = Color( 1.0f, 1.0f, 1.0f );  
mStrokeColor = Color( 1.0f, 0.0f, 0.0f );
```

3. In the `draw` method, let's start by drawing filled shapes.

Let's clear the background and set `mFillColor` to be the drawing color.

```
gl::clear( Color( 0, 0, 0 ) );  
gl::color( mFillColor );
```

4. Draw the filled shapes by calling the `ci::gl::drawSolidRect`, `ci::gl::drawSolidCircle`, and `ci::gl::drawSolidEllipse` methods.

Add the following code snippet inside the `draw` method:

```
gl::drawSolidRect( mRect );  
gl::drawSolidCircle( mCircleCenter, mCircleRadius );  
gl::drawSolidEllipse( mEllipseCenter, mEllipseWidth, ellipseHeight  
);
```

5. To draw our shapes as stroked graphics, let's first set `mStrokeColor` as the drawing color.

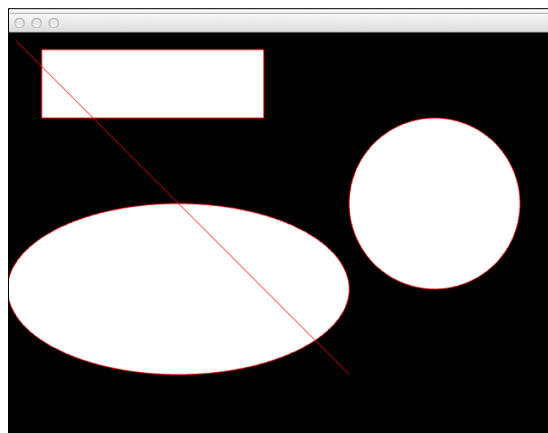
```
gl::color( mStrokeColor );
```

6. Let's draw our shapes again, this time using only strokes by calling the `ci::gl::drawLine`, `ci::gl::drawStrokeRect`, `ci::gl::drawStrokeCircle`, and `ci::gl::drawStrokedEllipse` methods.

Add the following code snippet inside the `draw` method:

```
gl::drawLine( mLineBegin, mLineEnd );  
gl::drawStrokedRect( mRect );  
gl::drawStrokedCircle( mCircleCenter, mCircleRadius );  
gl::drawStrokedEllipse( mEllipseCenter, mEllipseWidth,  
ellipseHeight );
```

This results in the following:



How it works...

Cinder's drawing methods use OpenGL calls internally to provide fast and easy drawing routines.

The `ci::gl::color` method sets the drawing color so that all shapes will be drawn with that color until another is set by calling `ci::gl::color` again.

There's more...

You can also set the stroke width by calling the `glLineWidth` method and passing a `float` value as a parameter.

For example, to set the stroke to be 5 pixels wide you should write the following:

```
glLineWidth( 5.0f );
```

Drawing arbitrary shapes with the mouse

In this recipe, we will learn how to draw arbitrary shapes using the mouse.

We will begin a new contour every time the user presses the mouse button, and draw when the user drags the mouse.

The shape will be drawn using fill and stroke.

Getting ready

Include the necessary files to draw and create a `ci::Shape2d` object.

Add the following code snippet at the top of your source file:

```
#include "cinder/gl/gl.h"  
#include "cinder/shape2d.h"
```

How to do it...

We will create a `ci::Shape2d` object and create vertices using mouse coordinates. Perform the following steps to do so:

1. Declare a `ci::Shape2d` object to define our shape and two `ci::Color` objects to define the fill and stroke colors.

```
Shape2d mShape;  
Color fillColor, strokeColor;
```

2. Initialize the colors in the `setup` method.

We'll be using black for stroke and yellow for fill.

```
mFillColor = Color( 1.0f, 1.0f, 0.0f );  
mStrokeColor = Color( 0.0f, 0.0f, 0.0f );
```

3. Since the drawing will be made with the mouse, it is necessary to use the `mouseDown` and `mouseDrag` events.

Declare the necessary callback methods.

```
void mouseDown( MouseEvent event );  
void mouseDrag( MouseEvent event );
```

4. In the implementation of `mouseDown` we will create a new contour by calling the `moveTo` method.

The following code snippet shows what the method should look like:

```
void MyApp::mouseDown( MouseEvent event ){  
    mShape.moveTo( event.getpos() );  
}
```

5. In the `mouseDrag` method we will add a line to our shape by calling the `lineTo` method.

Its implementation should look like the following code snippet:

```
void MyApp::mouseDrag( MouseEvent event ){  
    mShape.lineTo( event.getPos() );  
}
```

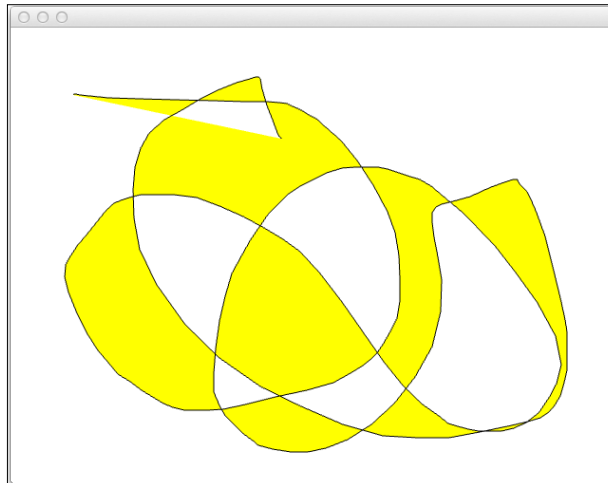
6. In the `draw` method, we will first need to clear the background, then set `mFillColor` as the drawing color, and draw `mShape`.

```
gl::clear( Color::white() );  
gl::color( mFillColor );  
gl::drawSolid( mShape );
```

7. All there is left to do is to set `mStrokeColor` as the drawing color and draw `mShape` as a stroked shape.

```
gl::color( mStrokeColor );  
gl::draw( mShape );
```

- Build and run the application. Press the mouse button to begin drawing a new contour, and drag to draw.



How it works...

`ci::Shape2d` is a class that defines an arbitrary shape in two dimensions allowing multiple contours.

The `ci::Shape2d::moveTo` method creates a new contour starting at the coordinate passed as a parameter. Then, the `ci::Shape2d::lineTo` method creates a straight line from the last position to the coordinate which is passed as a parameter.

The shape is internally tessellated into triangles when drawing a solid graphic.

There's more...

It is also possible to add curves when constructing a shape using `ci::Shape2d`.

Method	Explanation
<code>quadTo (constVec2f& p1, constVec2f& p2)</code>	Adds a quadratic curve from the last position to <code>p2</code> , using <code>p1</code> as a control point
<code>curveTo (constVec2f& p1, constVec2f& p2, constVec2f& p3)</code>	Adds a curve from the last position to <code>p3</code> , using <code>p1</code> and <code>p2</code> as control points
<code>arcTo (constVec2f& p, constVec2f& t, float radius)</code>	Adds an arc from the last position to <code>p1</code> using <code>t</code> as the tangent point and <code>radius</code> as the arc's radius

Implementing a scribbler algorithm

In this recipe, we are going to implement a scribbler algorithm, which is very simple to implement using Cinder but gives an interesting effect while drawing. You can read more about the concept of connecting neighbor points at <http://www.zefrank.com/scribbler/about.html>. You can find an example of scribbler at <http://www.zefrank.com/scribbler/> or <http://mrdoob.com/projects/harmony/>.

How to do it...

We will implement an application illustrating scribbler. Perform the following steps to do so:

1. Include the necessary headers:

```
#include<vector>
```

2. Add properties to your main application class:

```
vector <Vec2f> mPath;
float mMaxDist;
ColorA mColor;
bool mDrawPath;
```

3. Implement the `setup` method, as follows:

```
void MainApp::setup()
{
    mDrawPath = false;
    mMaxDist = 50.f;
    mColor = ColorA(0.3f,0.3f,0.3f, 0.05f);
    setWindowSize(800, 600);

    gl::enableAlphaBlending();
    gl::clear( Color::white() );
}
```

4. Since the drawing will be made with the mouse, it is necessary to use the `mouseDown` and `mouseUp` events. Implement these methods, as follows:

```
void MainApp::mouseDown( MouseEvent event )
{
    mDrawPath = true;
}

void MainApp::mouseUp( MouseEvent event )
{
    mDrawPath = false;
}
```

5. Finally, the implementation of drawing methods looks like the following code snippet:

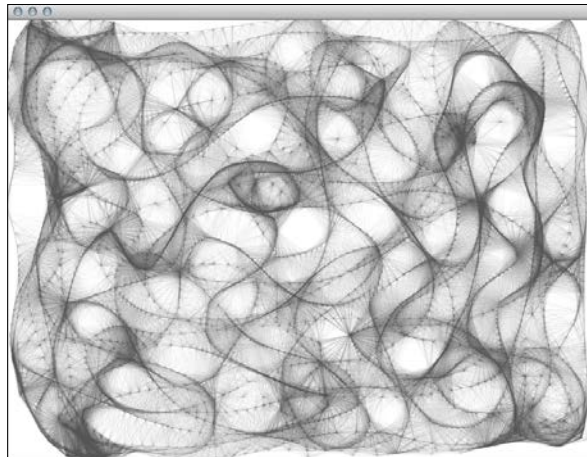
```
void MainApp::draw() {
    if( mDrawPath ) {
        drawPoint( getMousePos() );
    }
}

void MainApp::drawPoint( Vec2f point ) {
    mPath.push_back( point );

    gl::color(mColor);
    vector<Vec2f>::iterator it;
    for(it = mPath.begin(); it != mPath.end(); ++it) {
        if( (*it).distance(point) < mMaxDist ) {
            gl::drawLine(point, (*it));
        }
    }
}
```

How it works...

While the left mouse button is down, we are adding a new point to our container and drawing lines connecting it with other points near it. The distance between the newly-added point and the points in its neighborhood we are looking for to draw a connection line has to be less than the value of the `mMaxDist` property. Please notice that we are clearing the drawing area only once, at the program startup at the end of the `setup` method, so we don't have to redraw all the connections to each frame, which would be very slow.



Implementing 2D metaballs

In this recipe, we will learn how we can implement organic looking objects called metaballs.

Getting ready

In this recipe, we are going to use the code base from the *Applying repulsion and attraction forces* recipe in *Chapter 5, Building Particle Systems*.

How to do it...

We will implement the metaballs' rendering using a shader program. Perform the following steps to do so:

1. Create a file inside the `assets` folder with a name, `passThru_vert.glsl`, and put the following code snippet inside it:

```
void main()
{
    gl_Position = ftransform();
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_FrontColor = gl_Color;
}
```

2. Create a file inside the `assets` folder with a name, `mb_frag.glsl`, and put the following code snippet inside it:

```
#version 120

uniform vec2 size;
uniform int num;
uniform vec2 positions[100];
uniform float radius[100];

void main(void)
{

    // Get coordinates
    vec 2 texCoord = gl_TexCoord[0].st;

    vec4 color = vec4(1.0,1.0,1.0, 0.0);
    float a = 0.0;

    int i;
    for(i = 0; i<num; i++) {
```



```
        color.a += (radius[i] / sqrt( ((texCoord.x*size.x) -
        positions[i].x)*((texCoord.x*size.x)-positions[i].x) +
        ((texCoord.y*size.y) -
        positions[i].y)*((texCoord.y*size.y)-positions[i].y) )
        );
    }

    // Set color
    gl_FragColor = color;
}
```

3. Add the necessary header files.

```
#include "cinder/Utilities.h"
#include "cinder/gl/GlslProg.h"
```

4. Add a property to your application's main class, which is the GlslProg object for our GLSL shader program.

```
gl::GlslProg mMetaballsShader;
```

5. In the setup method, change the values of repulsionFactor and numParticle.

```
repulsionFactor = -40.f;
int numParticle = 10;
```

6. At the end of the setup method, load our GLSL shader program, as follows:

```
mMetaballsShader = gl::GlslProg( loadAsset("passThru_vert.glsl"),
loadAsset("mb_frag.glsl" ) );
```

7. The last major change is in the draw method, which looks like the following code snippet:

```
void MainApp::draw()
{
    gl::enableAlphaBlending();
    gl::clear( Color::black() );

    int particleNum = mParticleSystem.particles.size();

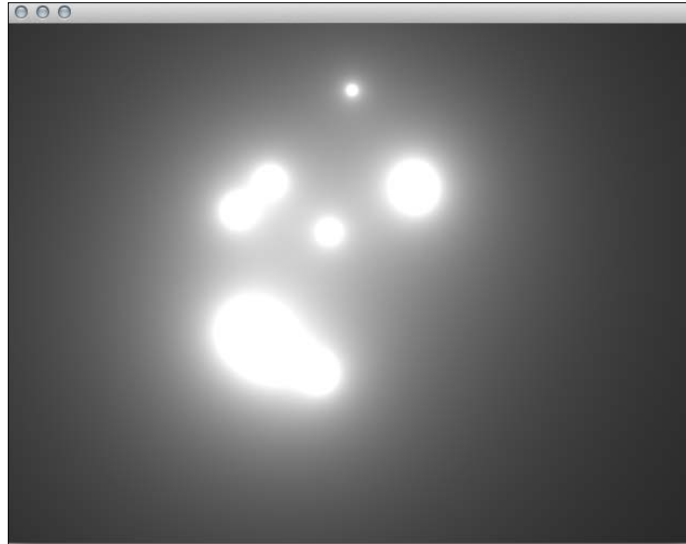
    mMetaballsShader.bind();
    mMetaballsShader.uniform( "size", Vec2f(getWindowSize()) );
    mMetaballsShader.uniform( "num", particleNum );

    for (int i = 0; i<particleNum; i++) {
        mMetaballsShader.uniform( "positions[" + toString(i) +
        "]", mParticleSystem.particles[i]->position );
        mMetaballsShader.uniform( "radius[" + toString(i) +
```

```
        "]", mParticleSystem.particles[i]->radius );  
    }  
  
    gl::color(Color::white());  
    gl::drawSolidRect( getWindowBounds() );  
    mMetaballsShader.unbind();  
}
```

How it works...

The most important part of this recipe is the fragment shader program mentioned in step 2. The shader generates texture with rendered metaballs based on the positions and radius passed to the shader from our particle system. In step 7, you can find out how to pass information to the shader program. We are using `setMatricesWindow` and `setViewport` to set OpenGL for drawing.



See also

- ▶ **A Wikipedia article on metaballs:** <http://en.wikipedia.org/wiki/Metaballs>

Animating text around curves

In this recipe, we will learn how we can animate text around a user-defined curve.

We will create the `Letter` and `Word` classes to manage the animation, a `ci::Path2d` object to define the curve, and a `ci::Timer` object to define the duration of the animation.

Getting ready

Create and add the following files to your project:

- ▶ `Word.h`
- ▶ `Word.cpp`
- ▶ `Letter.h`
- ▶ `Letter.cpp`

How to do it...

We will create a word and animate its letters along a `ci::Path2d` object. Perform the following steps to do so:

1. In the `Letter.h` file, include the necessary to use the `text`, `ci::Vec2f`, and `ci::gl::Texture` files.

Also add the `#pragma once` macro

```
#pragma once

#include "cinder/vector.h"
#include "cinder/text.h"
#include "cinder/gl/Texture.h"
```

2. Declare the `Letter` class with the following members and methods:

```
class Letter{
public:
    Letter( ci::Font font, conststd::string& letter );

    void draw();
    void setPos( const ci::Vec2f& newPos );

    ci::Vec2f pos;
    float rotation;
```

```

        ci::gl::Texture texture;
        float width;
    };

```

3. Move to the `Letter.cpp` file to implement the class.

In the constructor, create a `ci::TextBox` object, set its parameters, and render it to texture. Also, set the width as the texture's width plus a padding value of 10:

```

Letter::Letter( ci::Font font, const std::string& letter ) {
    ci::TextBox textBox;
    textBox = ci::TextBox().font( font ).size( ci::Vec2i(
        ci::TextBox::GROW, ci::TextBox::GROW ) ).text( letter
    ).premultiplied();
    texture = textBox.render();
    width = texture.getWidth() + 10.0f;
}

```

4. In the `draw` method, we will draw the texture and use OpenGL transformations to translate the texture to its position, and rotate according to the rotation:

```

void Letter::draw() {
    glPushMatrix();
    glTranslatef( pos.x, pos.y, 0.0f );
    glRotatef( ci::toDegrees( rotation ), 0.0f, 0.0f, 1.0f );
    glTranslatef( 0.0f, -texture.getHeight(), 0.0f );
    ci::gl::draw( texture );
    glPopMatrix();
}

```

5. In the `setPos` method implementation, we will update the position and calculate its rotation so that the letter is perpendicular to its movement. We do this by calculating the arc tangent of its velocity:

```

void Letter::setPos( const ci::Vec2f&newPos ) {
    ci::Vec2f vel = newPos - pos;
    rotation = atan2( vel.y, vel.x );
    pos = newPos;
}

```

6. The `Letter` class is ready! Now move to the `Word.h` file, add the `#pragma once` macro, and include the `Letter.h` file:

```

#pragma once
#include "Letter.h"

```

7. Declare the `Word` class with the following members and methods:

```
class Word{
public:
    Word( ci::Font font, conststd::string& text );

    ~Word();

    void update( const ci::Path2d& curve, float curveLength, float
progress );
    void draw();

    std::vector< Letter* > letters;
    float length;
};
```

8. Move to the `Word.cpp` file and include the `Word.h` file:

```
#include "Word.h"
```

9. In the constructor, we will iterate over each character of `text` and add a new `Letter` object. We will also calculate the total length of the text by calculating the sum of widths of all the letters:

```
Word::Word( ci::Font font, conststd::string& text ){
    length = 0.0f;
    for( int i=0; i<text.size(); i++ ){
        std::string letterText( 1, text[i] );
        Letter *letter = new Letter( font, letterText );
        letters.push_back( letter );
        length += letter->width;
    }
}
```

In the destructor, we will delete all the `Letter` objects to clean up memory used by the class:

```
Word::~~Word(){
    for( std::vector<Letter*>::iterator it = letters.begin(); it !=
letters.end(); ++it ){
        delete *it;
    }
}
```

10. In the `update` method, we will pass a reference to the `ci::Path2d` object, the total length of the path, and the progress of the animation as a normalized value from 0.0 to 1.0.

We will calculate the position of each individual letter along the curve taking into account the length of `word` and the current progress:

```
void Word::update( const ci::Path2d& curve, float curveLength,
float progress ) {
    float maxProgress = 1.0f - ( length / curveLength );
    float currentProgress = progress * maxProgress;
    float progressOffset = 0.0f;
    for( int i=0; i<letters.size(); i++ ){
        ci::Vec2f pos = curve.getPosition
            ( currentProgress + progressOffset );
        letters[i]->setPos( pos );
        progressOffset += ( letters[i]->width / curveLength );
    }
}
```

11. In the `draw` method, we will iterate over all letters and call the `draw` method of each letter:

```
void Word::draw() {
    for( std::vector< Letter* >::iterator it = letters.begin(); it
        != letters.end(); ++it ) {
        (*it)->draw();
    }
}
```

12. With the `Word` and `Letter` classes ready, it's time to move to our application's class source file. Start by including the necessary source files and adding the helpful `using` statements:

```
#include "cinder/Timer.h"
#include "Word.h"

using namespace ci;
using namespace ci::app;
using namespace std;
```

13. Declare the following members:

```
Word * mWord;
Path2d mCurve;
float mPathLength;
Timer mTimer;
double mSeconds;
```

14. In the `setup` method, we will start by creating `std::string` and `ci::Font` and use them to initialize `mWord`. We will also initialize `mSeconds` with the seconds we want our animation to last for:

```
string text = "Some Text";
Font font = Font( "Arial", 46 );
mWord = new Word( font, text );
mSeconds = 5.0;
```

15. We now need to create the curve by creating the keypoints and connecting them by calling `curveTo`:

```
Vec2f curveBegin( 0.0f, getWindowCenter().y );
Vec2f curveCenter = getWindowCenter();
Vec2f curveEnd( getWindowWidth(), getWindowCenter().y );

mCurve.moveTo( curveBegin );
mCurve.curveTo( Vec2f( curveBegin.x, curveBegin.y + 200.0f ),
Vec2f( curveCenter.x, curveCenter.y + 200.0f ), curveCenter );
mCurve.curveTo( Vec2f( curveCenter.x, curveCenter.y - 200.0f ),
Vec2f( curveEnd.x, curveEnd.y - 200.0f ), curveEnd );
```

16. Let's calculate the length of the path by summing the distance between each point and the one next to it. Add the following code snippet inside the `setup` method:

```
mPathLength = 0.0f;
for( int i=0; i<mCurve.getNumPoints()-1; i++ ){
    mPathLength += mCurve.getPoint( i ).distance( mCurve.getPoint(
    i+1 ) );
}
```

17. We need to check if `mTimer` is running and calculate the progress by calculating the ratio between the elapsed seconds and `mSeconds`. Add the following code snippet inside the `update` method:

```
if( mTimer.isStopped() == false ){
    float progress;
    if( mTimer.getSeconds() >mSeconds ){
        mTimer.stop();
    }
}
```

```

        progress = 1.0f;
    } else {
        progress = (float)( mTimer.getSeconds() / mSeconds );
    }
    mWord->update( mCurve, mPathLength, progress );
}

```

18. In the draw method, we will need to clear the background, enable alpha blending, draw mWord, and draw the path:

```

gl::clear( Color( 0, 0, 0 ) );
gl::enableAlphaBlending();
mWord->draw();
gl::draw( mCurve );

```

19. Finally, we need to start the timer whenever the user presses any key.

Declare the keyUp event handler:

```
void keyUp( KeyEvent event );
```

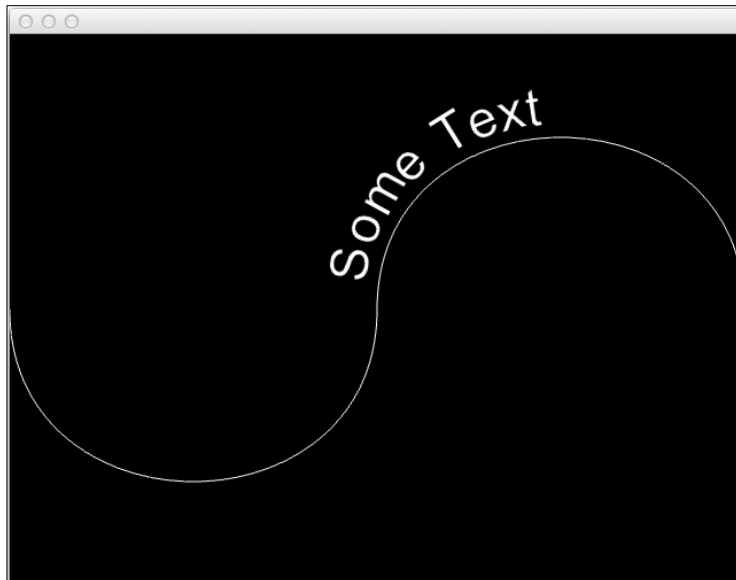
20. And the following is the implementation of the the keyUp event handler:

```

void CurveTextApp::keyUp( KeyEvent event ){
    mTimer.start();
}

```

21. Build and run the application. Press any key to begin the animation.



Adding a blur effect

In this recipe, we will learn how we can apply a blur effect while drawing a texture.

Getting ready

In this recipe, we are going to use a Gaussian blur shader provided by Geeks3D at <http://www.geeks3d.com/20100909/shader-library-gaussian-blur-post-processing-filter-in-glsl/>.

How to do it...

We will implement a sample Cinder application to illustrate the mechanism. Perform the following steps:

1. Create a file inside the `assets` folder with the name `passThru_vert.glsl` and put the following code snippet inside it:

```
void main()
{
    gl_Position = ftransform();
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_FrontColor = gl_Color;
}
```

2. Create a file inside the `assets` folder with the name `gaussian_v_frag.glsl` and put the following code snippet inside it:

```
#version 120

uniform sampler2D sceneTex; // 0

uniform float rt_w; // render target width
uniform float rt_h; // render target height
uniform float vx_offset;

float offset[3] = float[]( 0.0, 1.3846153846, 3.2307692308 );
float weight[3] = float[]( 0.2270270270, 0.3162162162,
0.0702702703 );

void main()
{
    vec3 tc = vec3(1.0, 0.0, 0.0);
    if (gl_TexCoord[0].x < (vx_offset - 0.01)) {
```

```

vec2 uv = gl_TexCoord[0].xy;
tc = texture2D(sceneTex, uv).rgb * weight[0];
for (int i=1; i<3; i++) {
tc += texture2D(sceneTex, uv + vec2(0.0, offset[i])/rt_h).rgb *
weight[i];
tc += texture2D(sceneTex, uv - vec2(0.0, offset[i])/rt_h).rgb *
weight[i];
}
}
else if (gl_TexCoord[0].x>=(vx_offset+0.01)){
tc = texture2D(sceneTex, gl_TexCoord[0].xy).rgb;
}
}
gl_FragColor = vec4(tc, 1.0);
}

```

Create a file inside the assets folder with the name `gaussian_h_frag.glsl` and put the following code snippet inside it:

```

#version 120

uniform sampler2D sceneTex; // 0

uniform float rt_w; // render target width
uniform float rt_h; // render target height
uniform float vx_offset;

float offset[3] = float[]( 0.0, 1.3846153846, 3.2307692308 );
float weight[3] = float[]( 0.2270270270, 0.3162162162,
0.0702702703 );

void main()
{
vec3 tc = vec3(1.0, 0.0, 0.0);
if (gl_TexCoord[0].x<(vx_offset-0.01)){
vec2 uv = gl_TexCoord[0].xy;
tc = texture2D(sceneTex, uv).rgb * weight[0];
for (int i=1; i<3; i++)
{
tc += texture2D(sceneTex, uv + vec2(offset[i])/rt_w, 0.0).rgb
* weight[i];
tc += texture2D(sceneTex, uv - vec2(offset[i])/rt_w, 0.0).rgb
* weight[i];
}
}
}

```

```
    }
    else if (gl_TexCoord[0].x >= (vx_offset + 0.01))
    {
        tc = texture2D(sceneTex, gl_TexCoord[0].xy).rgb;
    }
    gl_FragColor = vec4(tc, 1.0);
}
```

3. Add the necessary headers:

```
#include "cinder/Utilities.h"
#include "cinder/gl/GlslProg.h"
#include "cinder/gl/Texture.h"
#include "cinder/ImageIo.h"
#include "cinder/gl/Fbo.h"
```

4. Add the properties to your application's main class:

```
gl::GlslProg  mGaussianVShader, mGaussianHShader;
gl::Texture  mImage, mImageBlur;
gl::Fbo      mFboBlur1, mFboBlur2;
float        offset, level;
params::InterfaceGl mParams;
```

5. Implement the `setup` method, as follows:

```
void MainApp::setup() {
    setWindowSize(512, 512);

    level = 0.5f;
    offset = 0.6f;

    mGaussianVShader = gl::GlslProg( loadAsset("passThru_vert.
    glsl"), loadAsset("gaussian_v_frag.glsl") );
    mGaussianHShader = gl::GlslProg( loadAsset("passThru_vert.
    glsl"), loadAsset("gaussian_h_frag.glsl") );
    mImage = gl::Texture(loadImage(loadAsset("image.png")));

    mFboBlur1 = gl::Fbo
        (mImage.getWidth(), mImage.getHeight());
    mFboBlur2 = gl::Fbo
        (mImage.getWidth(), mImage.getHeight());

    // Setup the parameters
    mParams = params::InterfaceGl
        ( "Parameters", Vec2i( 200, 100 ) );
    mParams.addParam
```

```

    ( "level", &level, "min=0 max=1 step=0.01" );
    mParams.addParam
    ( "offset", &offset, "min=0 max=1 step=0.01");
}

```

6. At the beginning of the draw method calculate the blur intensity:

```

float rt_w = mImage.getWidth()*3.f-mImage.getWidth()*2.f*level;
float rt_h = mImage.getHeight()*3.f-mImage.getHeight()*2.f*level;

```

7. In the draw function render an image to mFboBlur1 with a first step shader applied:

```

mFboBlur1.bindFramebuffer();
gl::setViewport( mFboBlur1.getBounds() );
mImage.bind(0);
mGaussianVShader.bind();
mGaussianVShader.uniform("sceneTex", 0);
mGaussianVShader.uniform("rt_w", rt_w);
mGaussianVShader.uniform("rt_h", rt_h);
mGaussianVShader.uniform("vx_offset", offset);
gl::drawSolidRect(mFboBlur1.getBounds());
mGaussianVShader.unbind();
mFboBlur1.unbindFramebuffer();

```

8. In the draw function render a texture from mFboBlur1 with a second step shader applied:

```

mFboBlur2.bindFramebuffer();
mFboBlur1.bindTexture(0);
mGaussianHShader.bind();
mGaussianHShader.uniform("sceneTex", 0);
mGaussianHShader.uniform("rt_w", rt_w);
mGaussianHShader.uniform("rt_h", rt_h);
mGaussianHShader.uniform("vx_offset", offset);
gl::drawSolidRect(mFboBlur2.getBounds());
mGaussianHShader.unbind();
mFboBlur2.unbindFramebuffer();

```

9. Set mImageBlur to the result texture from mFboBlur2:

```

mImageBlur = mFboBlur2.getTexture();

```

10. At the end of the draw method draw a texture with the result and GUI:

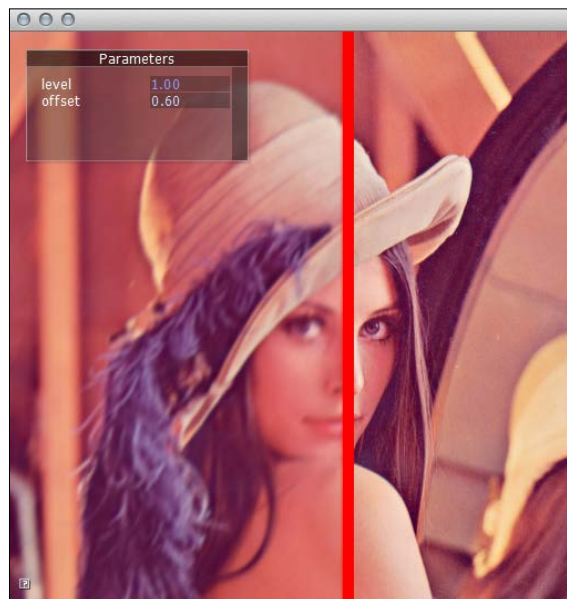
```

gl::clear( Color::black() );
gl::setMatricesWindow(getWindowSize());
gl::setViewport(getWindowBounds());
gl::draw(mImageBlur);
params::InterfaceGl::draw();

```

How it works...

Since a Gaussian blur shader needs to be applied twice—for the vertical and horizontal processing—we have to use **frame buffer object (FBO)**, a mechanism of drawing to the texture in the memory of graphic card. In step 8, we are drawing the original image from the `mImage` object and applying shader program stored in the `gaussian_v_frag.glsl` file loaded into `mGaussianVShaderobject`. At this point, everything is drawn into `mFboBlur1`. The next step is to use a texture from `mFboBlur2` and apply a shader to the second pass which you can find in step 9. The final processed texture is stored in `mImageBlur` in step 10. In step 7 we are calculating blur intensity.



Implementing a force-directed graph

A force-directed graph is a way of drawing an aesthetic graph using simple physics such as repelling and springs. We are going to make our graph interactive so that users can drag nodes around and see how graph reorganizes itself.

Getting ready

In this recipe we are going to use the code base from the *Creating a particle system in 2D* recipe in *Chapter 5, Building Particle Systems*. To get some details of how to draw nodes and connections between them, please refer to the *Connecting particles* recipe in *Chapter 6, Rendering and Texturing Particle Systems*.

How to do it...

We will create an interactive force-directed graph. Perform the following steps to do so:

1. Add properties to your main application class.

```
vector< pair<Particle*, Particle*> > mLinks;
float mLinkLength;
Particle* mHandle;
bool mIsHandle;
```

2. In the setup method set default values and create a graph.

```
void MainApp::setup() {
    mLinkLength = 40.f;
    mIsHandle = false;

    float drag = 0.95f;

    Particle *particle = newParticle(getWindowCenter(), 10.f, 10.f,
    drag );
    mParticleSystem.addParticle( particle );

    Vec2f r = Vec2f::one()*mLinkLength;
    for (int i = 1; i<= 3; i++) {
        r.rotate( M_PI * (i/3.f) );
        Particle *particle1 = newParticle( particle->position+r, 7.f,
        7.f, drag );
        mParticleSystem.addParticle( particle1 );
        mLinks.push_back(make_pair(mParticleSystem.particles[0],
        particle1));

        Vec2f r2 = (particle1->position-particle->position);
        r2.normalize();
        r2 *= mLinkLength;
        for (int ii = 1; ii <= 3; ii++) {
            r2.rotate( M_PI * (ii/3.f) );
            Particle *particle2 = newParticle( particle1->position+r2,
            5.f, 5.f, drag );
            mParticleSystem.addParticle( particle2 );
            mLinks.push_back(make_pair(particle1, particle2));

            Vec2f r3 = (particle2->position-particle1->position);
            r3.normalize();
            r3 *= mLinkLength;
            for (int iii = 1; iii <= 3; iii++) {
```

```

r3.rotate( M_PI * (iii/3.f) );
Particle *particle3 = newParticle( particle2->position+r3, 3.f,
3.f, drag );
mParticleSystem.addParticle( particle3 );
mLinks.push_back(make_pair(particle2, particle3));
    }
    }
}

```

3. Implement interaction with the mouse.

```

void MainApp::mouseDown(MouseEvent event){
    mIsHandle = false;

    float maxDist = 20.f;
    float minDist = maxDist;
    for( std::vector<Particle*>::iterator it = mParticleSystem.
particles.begin(); it != mParticleSystem.particles.end(); ++it )
    {
        float dist = (*it)->position.distance( event.getPos() );
        if(dist<maxDist&&dist<minDist) {
            mHandle = (*it);
            mIsHandle = true;
            minDist = dist;
        }
    }
}

void MainApp::mouseUp(MouseEvent event){
    mIsHandle = false;
}

```

4. Inside the update method, calculate all forces affecting particles.

```

void MainApp::update() {
    for( std::vector<Particle*>::iterator it1 = mParticleSystem.
particles.begin(); it1 != mParticleSystem.particles.end(); ++it1
)
    {
        for( std::vector<Particle*>::iterator it2 = mParticleSystem.
particles.begin(); it2 != mParticleSystem.particles.end();
++it2 ){
            Vec2f conVec = (*it2)->position - (*it1)->position;
            if(conVec.length() <0.1f)continue;

```

```

        float distance = conVec.length();
        conVec.normalize();
        float force = (mLinkLength*2.0f - distance)* -0.1f;
        force = math<float>::min(0.f, force);

        (*it1)->forces += conVec * force*0.5f;
        (*it2)->forces += -conVec * force*0.5f;
    }
}

for( vector<pair<Particle*, Particle*> > ::iterator it = mLinks.
begin(); it != mLinks.end(); ++it ){
    Vec2f conVec = it->second->position - it->first->position;
    float distance = conVec.length();
    float diff = (distance-mLinkLength)/distance;
    it->first->forces += conVec * 0.5f*diff;
    it->second->forces -= conVec * 0.5f*diff;
}

if(mIsHandle) {
    mHandle->position = getMousePos();
    mHandle->forces = Vec2f::zero();
}

mParticleSystem.update();
}

```

5. In the draw method implement drawing particles and links between them.

```

void MainApp::draw()
{
    gl::enableAlphaBlending();
    gl::clear( Color::white() );
    gl::setViewport( getWindowBounds() );
    gl::setMatricesWindow( getWindowWidth(), getWindowHeight() );

    gl::color( ColorA(0.f,0.f,0.f, 0.8f) );
    for( vector<pair<Particle*, Particle*> > ::iterator it = mLinks.
begin(); it != mLinks.end(); ++it )
    {
        Vec2f conVec = it->second->position - it->first->position;
        conVec.normalize();
        gl::drawLine(it->first->position + conVec * ( it->first-
>radius+2.f ),

```



```
        it->second->position - conVec * ( it->second->radius+2.f ) );
    }

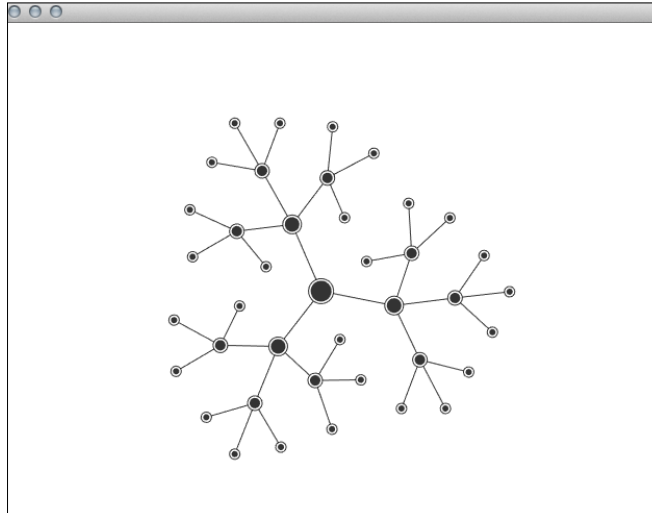
    gl::color( ci::ColorA(0.f,0.f,0.f, 0.8f) );
    mParticleSystem.draw();
}
```

6. Inside the `Particle.cpp` source file, drawing of each particle should be implemented, as follows:

```
void Particle::draw(){
    ci::gl::drawSolidCircle( position, radius);
    ci::gl::drawStrokedCircle( position, radius+2.f);
}
```

How it works...

In step 2, in the `setup` method, we are creating our particles for each level of the graph and adding links between them. In the `update` method in step 4, we are calculating forces affecting all particles, which is repelling each particle from each other, and forces coming from the springs connecting the nodes. While repelling spreading particles, springs try to keep them at a fixed distance defined in `mLinkLength`.



See also

- ▶ **The Wikipedia article on Force-directed graph drawing:** [http://en.wikipedia.org/wiki/Force-based_algorithms_\(graph_drawing\)](http://en.wikipedia.org/wiki/Force-based_algorithms_(graph_drawing))

8

Using 3D Graphics

In this chapter, we will learn how to work and draw with 3D graphics. The recipes in this chapter will cover the following:

- ▶ Drawing 3D geometric primitives
- ▶ Rotating, scaling, and translating
- ▶ Drawing to an offscreen canvas
- ▶ Drawing in 3D with the mouse
- ▶ Adding lights
- ▶ Picking in 3D
- ▶ Creating a height map from an image
- ▶ Creating a terrain with Perlin noise
- ▶ Saving mesh data

Introduction

In this chapter, we will learn the basics of creating graphics in 3D. We will use OpenGL and some useful wrappers that Cinder includes on some advanced OpenGL features.

Drawing 3D geometric primitives

In this recipe, we will learn how to draw the following 3D geometric shapes:

- ▶ Cube
- ▶ Sphere
- ▶ Line

- ▶ Torus
- ▶ Cylinder

Getting ready

Include the necessary header to draw in OpenGL using Cinder commands and statements. Add the following code to the top of your source file:

```
#include "cinder/gl/gl.h"
#include "cinder/Camera.h"

using namespace ci;
```

How to do it...

We will create several geometric primitives using Cinder's methods for drawing in 3D.

1. Declare the member variables with information of our primitives:

```
Vec3f mCubePos, mCubeSize;
Vec3f mSphereCenter;
float mSphereRadius;
Vec3f mLineBegin, mLineEnd;
Vec3f mTorusPos;
float mTorusOuterRadius, mTorusInnerRadius;
Vec3f mCylinderPos;
float mCylinderBaseRadius, mCylinderTopRadius, mCylinderHeight;
```

2. Initialize the member variables with the position and sizes of the geometry. Add the following code in the setup method:

```
mCubePos = Vec3f( 100.0f, 300.0f, 100.0f );
mCubeSize = Vec3f( 100.0f, 100.0f, 100.0f );

mSphereCenter = Vec3f( 500, 250, 0.0f );
mSphereRadius = 100.0f;

mLineBegin = Vec3f( 200, 0, 200 );
mLineEnd = Vec3f( 500, 500, -200 );

mTorusPos = Vec3f( 300.0f, 100.0f, 0.0f );
mTorusOuterRadius = 100.0f;
mTorusInnerRadius = 20.0f;

mCylinderPos = Vec3f( 500.0f, 0.0f, -200.0f );
mCylinderBaseRadius = 50.0f;
```

```
mCylinderTopRadius = 80.0f;
mCylinderHeight = 100.0f;
```

3. Before we draw the shapes, let's also create a camera to rotate around our shapes to give us a better sense of perspective. Declare a `ci::CameraPersp` object:

```
CameraPersp mCamera;
```

4. Initialize it in the `setup` method:

```
mCamera = CameraPersp( getWindowWidth(), getWindowHeight(), 60.0f
);
```

5. In the `update` method, we will make the camera rotate around our scene. Add the following code in the `update` method:

```
Vec2f windowCenter = getWindowCenter();
float cameraAngle = getElapsedSeconds();
float cameraDist = 450.0f;
float x = sinf( cameraAngle ) * cameraDist + windowCenter.x;
float z = cosf( cameraAngle ) * cameraDist;
mCamera.setEyePoint( Vec3f( x, windowCenter.y, z ) );
mCamera.lookAt( Vec3f( windowCenter.x, windowCenter.y, 0.0f ) );
```

6. In the `draw` method, we will clear the background with black and use `mCamera` to define the window's matrices. We will also enable OpenGL to read and write to the depth buffers. Add the following code in the `draw` method:

```
gl::clear( Color::black() );
gl::setMatrices( mCamera );
gl::enableDepthRead();
gl::enableDepthWrite();
```

7. Cinder allows you to draw filled and stroked cubes, so let's draw a cube with a white fill and black stroke:

```
gl::color( Color::white() );
gl::drawCube( mCubePos, mCubeSize );
gl::color( Color::black() );
gl::drawStrokedCube( mCubePos, mCubeSize );
```

8. Let's define the drawing color again as white, and draw a sphere with `mSphereCenter` and `mSphereRadius` as the sphere's position and radius, and the number of segments as 30.

```
gl::color( Color::white() );
gl::drawSphere( mSphereCenter, mSphereRadius, 30 );
```

9. Draw a line that begins at `mLineBegin` and ends at `mLineEnd`:

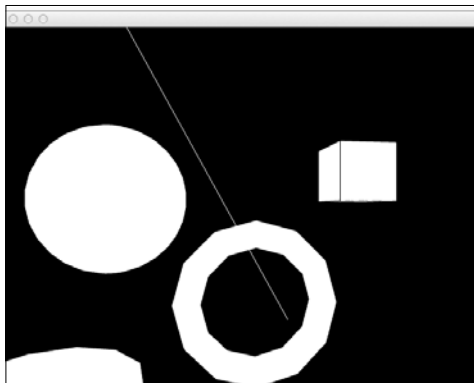
```
gl::drawLine( mLineBegin, mLineEnd );
```

10. Cinder draws a `Torus` at the coordinates of the origin `[0, 0]`. So, we will have to translate it to the desired position at `mTorusPos`. We will be using `mTorusOuterRadius` and `mTorusInnerRadius` to define the shape's inner and outer sizes:

```
gl::pushMatrices();  
gl::translate( mTorusPos );  
gl::drawTorus( mTorusOuterRadius, mTorusInnerRadius );  
gl::popMatrices();
```

11. Finally, Cinder will draw a cylinder at the origin `[0, 0]`, so we will have to translate it to the position defined in `mCylinderPosition`. We will also be using `mCylinderBaseRadius` and `mCylinderTopRadius`, to set the cylinder's bottom and top sizes and `mCylinderHeight`, to set its height:

```
gl::pushMatrices();  
gl::translate( mCylinderPos );  
gl::drawCylinder( mCylinderBaseRadius, mCylinderTopRadius,  
mCylinderHeight );  
gl::popMatrices();
```



How it works...

Cinder's drawing methods use OpenGL calls internally to provide fast and easy drawing routines.

The method `ci::gl::color` sets the drawing color so that all shapes will be drawn with that color until another color is set by calling `ci::gl::color` again.

See also

To learn more about OpenGL transformations such as translation, scale, and rotation, please read the recipe *Rotating, scaling, and translating*.

Rotating, scaling, and translating

In this recipe, we will learn how to transform our graphics using OpenGL transformations.

We will draw a unit cube at $[0, 0, 0]$ coordinates and then we will translate it to the center of the window, apply rotation, and scale it to a more visible size.

Getting ready

Include the necessary files to draw with OpenGL and add the helpful `using` statements. Add the following code to the top of the source file:

```
#include "cinder/gl/gl.h"
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will apply rotation, translation, and scaling to alter the way our cube is rendered. We will use Cinder's wrappers for OpenGL.

1. Let's declare variables to store our values for the translation, rotation, and scale transformations:

```
Vec3f mTranslation;
Vec3f mScale;
Vec3f mRotation;
```

2. To define the translation amount, let's translate half the window's width on the x axis and half the window's height on the y axis. This will bring anything we draw at $[0, 0, 0]$ to the center of the window. Add the following code in the `setup` method:

```
mTranslation.x = getWindowWidth() / 2;
mTranslation.y = getWindowHeight() / 2;
mTranslation.z = 0.0f;
```

3. Let's set the scale factor to be 100 on the x axis, 200 on the y axis, and 100 on the z axis. Anything we draw will be 100 times bigger on the x and z axes and 200 times bigger on the y axis. Add the following code in the `setup` method:

```
mScale.x = 100.0f;
mScale.y = 200.0f;
mScale.z = 100.0f;
```

4. In the `update` method, we will animate the rotation values by incrementing the rotation on the x and y axes.

```
mRotation.x += 1.0f;  
mRotation.y += 1.0f;
```

5. In the `draw` method, let's begin by clearing the background with black, setting the windows matrices to allow for drawing in 3D, and enabling OpenGL to read and write the depth buffer:

```
gl::clear( Color( 0, 0, 0 ) );  
gl::setMatricesWindowPersp( getWindowWidth(), getWindowHeight() );  
gl::enableDepthRead();  
gl::enableDepthWrite();
```

6. Let's add a new matrix to the stack and translate, scale, and rotate using the previously defined variables:

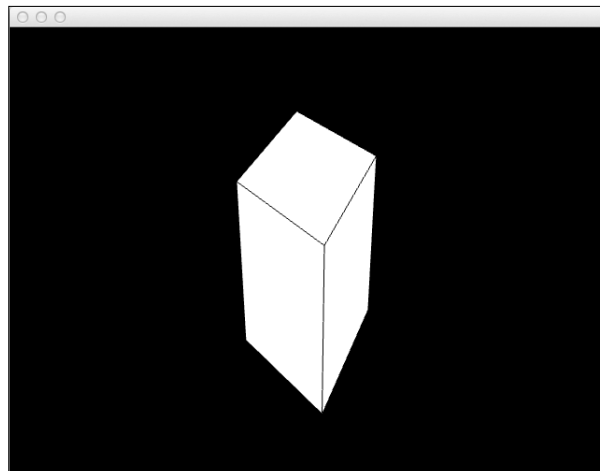
```
gl::pushMatrices();  
gl::translate( mTranslation );  
gl::scale( mScale );  
gl::rotate( mRotation );
```

7. Draw a unit quad at the origin `[0, 0, 0]` with a white fill and black stroke:

```
gl::color( Color::white() );  
gl::drawCube( Vec3f(), Vec3f( 1.0f, 1.0f, 1.0f ) );  
gl::color( Color::black() );  
gl::drawStrokedCube( Vec3f(), Vec3f( 1.0f, 1.0f, 1.0f ) );
```

8. Finally, remove the previously added matrix:

```
gl::popMatrices();
```



How it works...

The calls to `ci::gl::enableDepthRead` and `ci::gl::enableDepthWrite` respectively, enable reading and writing to the depth buffer. The depth buffer is where the depth information is stored.

When reading and writing to the depth buffer is enabled, OpenGL will sort objects so that closer objects are drawn in front of farther objects. When reading and writing to the depth buffer, the disabled objects will be drawn in the order they were created.

The methods `ci::gl::translate`, `ci::gl::rotate`, and `ci::gl::scale` are wrappers of OpenGL commands for translating, rotating, and scaling, which allow you to pass Cinder types as parameters.

Transformations in OpenGL are applied by multiplying vertex coordinates with transformation matrices. When we call the method `ci::gl::pushMatrices`, we add a copy of the current transformation matrix to the matrix stack. Calls to `ci::gl::translate`, `ci::gl::rotate`, or `ci::gl::scale` will apply the correspondent transformations to the last matrix in the stack, which will be applied to whatever geometry is created after calling the transformation methods. A call to `ci::gl::popMatrix` will remove the last transformation matrix in the stack so that transformations added to the last matrix will no longer affect our geometry.

Drawing to an offscreen canvas

In this recipe, we will learn how to draw in an offscreen canvas using the OpenGL **Frame Buffer Object (FBO)**.

We will draw in an FBO and draw it onscreen as well as texture a rotating cube.

Getting ready

Include the necessary files to work with OpenGL and the FBOs as well as the useful `include` directives.

Add the following code to the top of the source file:

```
#include "cinder/gl/gl.h"
#include "cinder/gl/Fbo.h"

using namespace ci;
```


How to do it...

We will use a `ci::gl::Fbo` object, a wrapper to an OpenGL FBO, to draw in an offscreen destination.

1. Declare a `ci::gl::Fbo` object as well as a `ci::Vec3f` object to define the cube's rotation:

```
gl::Fbo mFbo;  
Vec3f mCubeRotation;
```

2. Initialize `mFbo` with a size of 256 x 256 pixels by adding the following code in the `setup` method:

```
mFbo = gl::Fbo( 256, 256 );
```

3. Animate `mCubeRotation` in the `update` method:

```
mCubeRotation.x += 1.0f;  
mCubeRotation.y += 1.0f;
```

4. Declare a method where we will draw to the FBO:

```
void drawToFbo();
```

5. In the implementation of `drawToFbo`, we will begin by creating a `ci::gl::SaveFramebufferBinding` object and then bind `mFbo`.

```
gl::SaveFramebufferBinding fboBindingSave;  
mFbo.bindFramebuffer();
```

6. Now we will clear the background with a dark gray color and set the matrices using the FBO's width and height.

```
gl::clear( Color( 0.3f, 0.3f, 0.3f ) );  
gl::setMatricesWindowPersp( mFbo.getWidth(), mFbo.getHeight() );
```

7. Now we will draw a rotating color cube at the center of the FBO with size 100 and using `mCubeRotation` to rotate the cube.

```
gl::pushMatrices();  
Vec3f cubeTranslate( mFbo.getWidth() / 2, mFbo.getHeight() / 2,  
0.0f );  
gl::translate( cubeTranslate );  
gl::rotate( mCubeRotation );  
gl::drawColorCube( Vec3f(), Vec3f( 100, 100, 100 ) );  
gl::popMatrices();
```

8. Let's move to the implementation of the `draw` method. Start by calling the method `drawToFbo`, clearing the background with black, setting the window's matrices, and enable reading and writing to the depth buffer. Add the following code in the `draw` method:

```
drawToFbo();
gl::clear( Color( 0, 0, 0 ) );
gl::setMatricesWindowPersp( getWindowWidth(), getWindowHeight() );
gl::enableDepthRead();
gl::enableDepthWrite();
```

Lets draw our Fbo at the top left corner of the window using mFbo texture:

```
gl::draw( mFbo.getTexture(), Rectf( 0.0f, 0.0f, 100.0f, 100.0f )
);
```

9. Enable and bind the texture of mFbo:

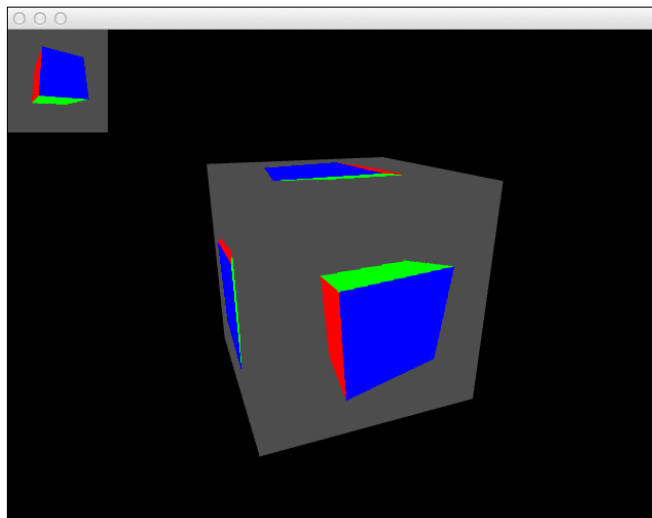
```
mFbo.getTexture().enableAndBind();
```

10. Draw a rotating cube at the center of the window using mCubeRotation to define its rotation:

```
gl::pushMatrices();
Vec3f center( getWindowWidth() / 2, getWindowHeight() / 2, 0.0f );
gl::translate( center );
gl::rotate( mCubeRotation );
gl::drawCube( Vec3f(), Vec3f( 200.0f, 200.0f, 200.0f ) );
gl::popMatrices();
```

11. To finalize, unbind the texture of mFbo:

```
mFbo.unbindTexture();
```



How it works...

The class `ci::gl::Fbo` wraps an OpenGL FBO.

Frame Buffer Objects are OpenGL objects that contain a collection of buffers that can be used as rendering destinations. The OpenGL context provides a default frame buffer where rendering occurs. Frame Buffer Objects allow rendering to alternative, offscreen locations.

The FBO has a color texture where the graphics are stored, and it can be bound and drawn like a regular OpenGL texture.

On step 5, we created a `ci::gl::SaveFramebufferBinding` object, which is a helper class that restores the previous FBO state. When using OpenGL ES, this object will restore and bind the previously bound FBO (usually the *screen* FBO) when it is destroyed.

See also

See the recipe *Rotating, scaling, and translating* to learn more about OpenGL transformations.

Drawing in 3D with the mouse

In this recipe, we will draw with the mouse on a 3D space. We will draw lines when dragging the mouse or rotate the scene in 3D when dragging and pressing the *Shift* key simultaneously.

Getting ready

Include the necessary files to draw using OpenGL, as well as the files needed to use Cinder's perspective, Maya camera, and poly lines.

```
#include "cinder/gl/gl.h"
#include "cinder/Camera.h"
#include "cinder/MayaCamUI.h"
#include "cinder/PolyLine.h"
```

Also, add the following `using` statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will use the `ci::CameraPersp` and `ci::Ray` classes to convert the mouse coordinates to our rotated 3D scene.

1. Declare a `ci::MayaCamUI` object and a `std::vector` object of `ci::PolyLine<ci::Vec3f>` to store the drawn lines:

```
MayaCamUI mCamera;
vector<PolyLine<Vec3f> > mLines;
```

2. In the `setup` method, we will create `ci::CameraPersp` and set it up so that the point of interest is the center of the window. We will also set the camera as the current camera of `mCamera`:

```
CameraPersp cameraPersp( getWindowWidth(),
    getWindowHeight(), 60.0f );
Vec3f center( getWindowWidth() / 2, getWindowHeight() / 2,
    0.0f );
cameraPersp.setCenterOfInterestPoint( center );
mCamera.setCurrentCam( cameraPersp );
```

3. In the `draw` method, let's clear the background with black and use our camera to set the window's matrices.

```
gl::clear( Color( 0, 0, 0 ) );
gl::setMatrices( mCamera.getCamera() );
```

4. Now let's iterate `mLines` and draw each `ci::PolyLine`. Add the following code to the `draw` method:

```
for( vector<PolyLine<Vec3f> > ::iterator it = mLines.begin(); it
    != mLines.end(); ++it ){
    gl::draw( *it );
}
```

5. With our scene set up and the lines being drawn, we need to create the 3D perspective! Let's start by declaring a method to convert coordinates from the screen position to world position. Add the following method declaration:

```
Vec3f screenToWorld( const Vec2f&point ) const;
```

6. In the `screenToWorld` implementation, we need to generate a ray from `point` using the camera's perspective. Add the following code in `screenToWorld`:

```
float u = point.x / (float)getWindowWidth();
float v = point.y / (float)getWindowHeight();

const CameraPersp& cameraPersp = mCamera.getCamera();

Ray ray = cameraPersp.generateRay( u, 1.0f - v, cameraPersp.
    getAspectRatio() );
```

7. Now we need to calculate where the ray will intersect with a perpendicular plane at the camera's center of interest and then return the intersection point. Add the following code in the `screenToWorld` implementation:

```
float result = 0.0f;
Vec3f planePos = cameraPersp.getCenterOfInterestPoint();
Vec3f normal = cameraPersp.getViewDirection();

ray.calcPlaneIntersection( planePos, normal, &result );

Vec3f intersection= ray.calcPosition( result );
return intersection;
```

8. Let's use the previously defined method to draw with the mouse. Declare the `mouseDown` and `mouseDrag` event handlers:

```
void mouseDown( MouseEvent event );
void mouseDrag( MouseEvent event );
```

9. In the implementation of `mouseDown`, we will check if the *Shift* key is being pressed. If it is, we will call the `mouseDown` method of `mCamera`, otherwise, we will add `ci::PolyLine<ci::Vec3f>` to `mLines`, calculate the world position of the mouse cursor using `screenToWorld`, and add it:

```
void MyApp::mouseDown( MouseEvent event ){
    if( event.isShiftDown() ){
        mCamera.mouseDown( event.getPos() );
    }
    else {
        mLines.push_back( PolyLine<Vec3f>() );
        Vec3f point = screenToWorld( event.getPos() );
        mLines.back().push_back( point );
    }
}
```

10. In the implementation of `mouseDrag`, we will check if the *Shift* key is being pressed. If it is, we will call the `mouseDrag` method to `mCamera`, otherwise, we will calculate the world position of the mouse cursor and add it to last line in `mLines`.

```
void Pick3dApp::mouseDrag( MouseEvent event ){
    if( event.isShiftDown() ){
        mCamera.mouseDrag( event.getPos(), event.isLeftDown(), event.isMiddleDown(), event.isRightDown() );
    } else {
        Vec3f point = screenToWorld( event.getPos() );
        mLines.back().push_back( point );
    }
}
```

11. Build and run the application. Press and drag the mouse to draw a line. Press the *Shift* key and press and drag the mouse to rotate the scene.

How it works...

We use `ci::MayaCamUI` to easily rotate our scene.

The `ci::Ray` class is a representation of a ray, containing an origin, direction, and an infinite length. It provides useful methods to calculate intersections between rays and triangles or planes.

To calculate the world position of the mouse cursor we calculated a ray going from the camera's eye position in the camera's view direction.

We then calculated the intersection of the ray with the plane at the center of the scene, perpendicular to the camera.

The calculated position is then added to a `ci::PolyLine<ci::Vec3f>` object to draw the lines.

See also

- ▶ To learn more on how to use `ci::MayaCamUI`, please refer to the recipe *Using MayaCamUI* from *Chapter 2, Preparing for Development*.
- ▶ To learn how to draw in 2D, please read the recipe *Drawing arbitrary shapes with the mouse* from *Chapter 7, Using 2D Graphics*.

Adding lights

In this chapter, we will learn how to illuminate a 3D scene using OpenGL lights.

Getting ready

Include the necessary files to use OpenGL lights, materials, and draw. Add the following code to the top of the source file:

```
#include "cinder/gl/gl.h"
#include "cinder/gl/Light.h"
#include "cinder/gl/Material.h"
```

Also add the following `using` statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will use the default OpenGL light rendering methods to illuminate our scene. We will use the `ci::gl::Material` and `ci::gl::Light` classes, which are wrappers around the OpenGL functionality.

1. Declare `ci::gl::Material` to define the material properties of the objects being drawn and `ci::Vec3f` to define the lights position.

```
gl::Material mMaterial;  
Vec3f mLightPos;
```

2. Let's set the materials Ambient, Diffuse, Specular, Emission, and Shininess properties by adding the following code in the `setup` method:

```
mMaterial.setAmbient( Color::black() );  
mMaterial.setDiffuse( Color( 1.0f, 0.0f, 0.0f ) );  
mMaterial.setSpecular( Color::white() );  
mMaterial.setEmission( Color::black() );  
mMaterial.setShininess( 128.0f );
```

3. In the `update` method, we will use the mouse to define the light position. Add the following code in the `update` method:

```
mLightPos.x = getMousePos().x;  
mLightPos.y = getMousePos().y;  
mLightPos.z = 200.0f;
```

4. In the `draw` method, we will begin by clearing the background, setting the window's matrices, and enabling reading and writing to the depth buffer.

```
gl::clear( Color::black() );  
gl::setMatricesWindowPersp( getWindowWidth(), getWindowHeight() );  
gl::enableDepthWrite();  
gl::enableDepthRead();
```

5. Let's create an OpenGL light using a `ci::gl::Light` object. We will define it as a POINT light and set its ID to 0. We will also set its position to `mLightPos` and define its attenuation.

```
gl::Light light( gl::Light::POINT, 0 );  
light.setPosition( mLightPos );  
light.setAttenuation( 1.0f, 0.0f, 0.0f );
```

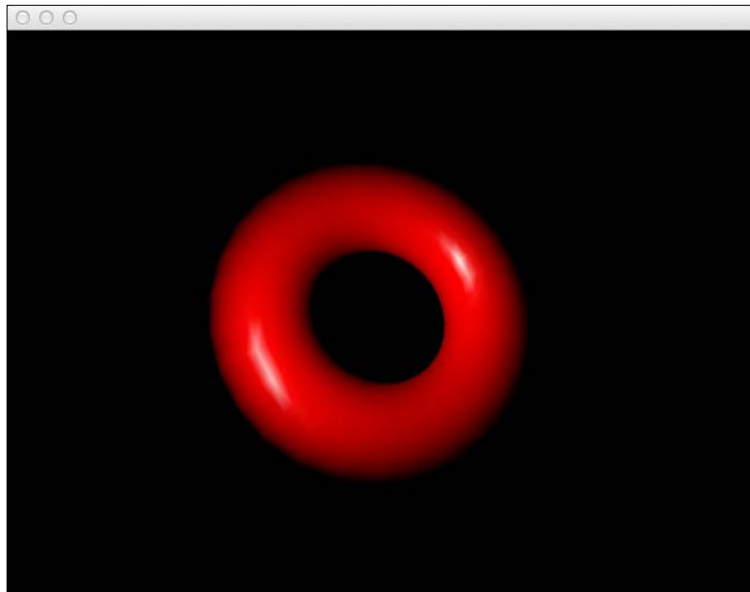
6. Let's enable OpenGL lighting, the previously created light, and apply the material.

```
glEnable( GL_LIGHTING );  
light.enable();  
mMaterial.apply();
```

- Let's draw a rotating `Torus` at the center of the window and use the elapsed seconds to rotate it. Add the following code to the `draw` method:

```
gl::pushMatrices();  
gl::translate( getWindowCenter() );  
float seconds = (float)getElapsedSeconds() * 100.0f;  
glRotatef( seconds, 1.0f, 0.0f, 0.0f );  
glRotatef( seconds, 0.0f, 1.0f, 0.0f );  
gl::drawTorus( 100.0f, 40.0f, 30, 30 );  
gl::popMatrices();
```

- Finally, disable the light:
`light.disable();`
- Build and run the application; you will see a red rotating torus. Move the mouse to change the lights position.



How it works...

We are using the `ci::gl::Material` and `ci::gl::Light` objects, which are helper classes to define the properties of lights and materials.

The material properties defined in the `setup` method, work in the following ways:

Material Property	Function
Ambient	How an object can reflect light that comes in all directions.
Diffuse	How an object reflects light that comes from a specific direction or position.
Specular	The light that an object will reflect as a result of diffuse lighting.
Emission	Light emitted by the object.
Shininess	The angle that the object will reflect specular light. Has to be a value between 1 and 128.

The material ambient, diffuse, and specular colors will multiply with the ambient, diffuse, and specular colors coming from the light source, which are all white by default.

It is possible to define three different types of lights. In the previous example, we defined our light source to be of type `ci::gl::Light::POINT`.

Here are the available types of light and their properties:

Light Type	Properties
<code>ci::gl::Light::POINT</code>	Point light is the light coming from a specific position in space and illuminating in all directions.
<code>ci::gl::Light::DIRECTION</code>	Directional light simulates light coming from a position so far away that all light rays are parallel and arrive in the same direction.
<code>ci::gl::Light::SPOTLIGHT</code>	Spotlight is the light coming from a specific position in space and a specific direction.

We also defined the attenuation values. Lights in OpenGL allow for defining the values for the constant attenuation, linear attenuation, and quadratic attenuation. These define how the light becomes dimmer as the distance from the light source increases.

To illuminate geometry, it is necessary to calculate the normal for each vertex. All shapes created using Cinder's commands have their normal calculated for us, so we don't have to worry about that.

There's more...

It is also possible to define the ambient, diffuse, and specular colors coming from the light source. The values defined in these colors will multiply with the correspondent colors of the material.

Here are the `ci::gl::Light` methods that allow you to define the light colors:

Method	Light
<code>setAmbient(const Color& color)</code>	Color of the ambient light.
<code>setDiffuse(const Color& color)</code>	Color of the diffuse light.
<code>setSpecular(const Color& color)</code>	Color of the specular light.

It is possible to create more than one light source. The amount of lights is dependent on the implementation of the graphics card, but it is always at least 8.

To create more light sources, simply create more `ci::gl::Light` objects and make sure each gets a unique ID.

See also

Please read the recipe *Calculating vertex normals* to learn how to calculate the vertex normals for user created geometry.

Picking in 3D

In this recipe, we will calculate the intersection of the mouse cursor with a 3D model.

Getting ready

Include the necessary files to draw using OpenGL, use textures and load images, load 3D models, define OpenGL lights and materials, and use Cinder's Maya camera.

```
#include "cinder/gl/gl.h"
#include "cinder/gl/Texture.h"
#include "cinder/gl/Light.h"
#include "cinder/gl/Material.h"
#include "cinder/TriMesh.h"
#include "cinder/ImageIo.h"
#include "cinder/MayaCamUI.h"
```

Also, add the following using statements:

```
using namespace ci;  
using namespace ci::app;  
using namespace std;
```

We will use a 3D model, so place a file and its texture in the `assets` folder. For this example, we will be using a mesh file named `ducky.msh` and a texture named `ducky.png`.

How to do it...

1. We will use the `ci::CameraPersp` and `ci::Ray` classes to convert the mouse coordinates to our rotated 3D scene and calculate the intersection with a 3D model.
2. Declare the members to define the 3D model and its intersection with the mouse, as well as a `ci::MayaCamUI` object for easy navigation, and a `ci::gl::Material` for lighting:

```
TriMesh mMesh;  
gl::Texture mTexture;  
MayaCamUI mCam;  
bool mIntersects;  
Vec3f mNormal, mHitPos;  
AxisAlignedBox3f mMeshBounds;  
gl::Material mMaterial;
```

3. Declare a method where we will calculate the intersection between a `ci::Ray` class and the triangles that make up `mMesh`.

```
void calcIntersectionWithMeshTriangles( const ci::Ray& ray );
```
4. In the `setup` method, let's load the model and texture and calculate its bounding box:

```
mMesh.read( loadAsset( "ducky.msh" ) );  
mTexture = loadImage( loadAsset( "ducky.png" ) );  
mMeshBounds = mMesh.calcBoundingBox();
```
5. Let's define the camera and make it look as if it's at the center of the model. Add the following code in the `setup` method:

```
CameraPersp cam;  
Vec3f modelCenter = mMeshBounds.getCenter();  
cam.setEyePoint( modelCenter + Vec3f( 0.0f, 0.0f, 20.0f ) );  
cam.setCenterOfInterestPoint( modelCenter );  
mCam.setCurrentCam( cam );
```

6. Finally, set up the material for the model's lighting.

```
mMaterial.setAmbient( Color::black() );
mMaterial.setDiffuse( Color::white() );
mMaterial.setEmission( Color::black() );
```

7. Declare the handlers for the `mouseDown` and `mouseDrag` events.

```
void mouseDown( MouseEvent event );
void mouseDrag( MouseEvent event );
```

8. Implement these methods by calling the necessary methods of `mCam`:

```
void MyApp::mouseDown( MouseEvent event ){
    mCam.mouseDown( event.getPos() );
}

void MyApp::mouseDrag( MouseEvent event ){
    mCam.mouseDrag( event.getPos(), event.isLeftDown(), event.
        isMiddleDown(), event.isRightDown() );
}
```

9. Let's implement the `update` method and calculate the intersection between the mouse cursor and our model. Let's begin by getting the mouse position and then calculate `ci::Ray` emitting from our camera:

```
Vec2f mousePos = getMousePos();
float u = mousePos.x / (float)getWindowWidth();
float v = mousePos.y / (float)getWindowHeight();
CameraPersp cameraPersp = mCam.getCamera();
Ray ray = cameraPersp.generateRay( u, 1.0f - v, cameraPersp.
    getAspectRatio() );
```

10. Let's perform a fast test and check if the ray intersects with the model's bounding box. If the result is `true`, we will call the `calcIntersectionWithMeshTriangles` method.

```
if( mMeshBounds.intersects( ray ) == false ){
    mIntersects = false;
} else {
    calcIntersectionWithMeshTriangles( ray );
}
```

11. Let's implement the `calcIntersectionWithMeshTriangles` method. We will iterate over all the triangles of our model and calculate the nearest intersection and store its index.

```
float distance = 0.0f;
float resultDistance = 999999999.9f;
int resultIndex = -1;
```

```
int numTriangles = mMesh.getNumTriangles();
for( int i=0; i<numTriangles; i++ ){
    Vec3f v1, v2, v3;
    mMesh.getTriangleVertices( i, &v1, &v2, &v3 );
    if( ray.calcTriangleIntersection( v1, v2, v3, &distance )
    ){
        if( distance <resultDistance ){
            resultDistance = distance;
            resultIndex = i;
        }
    }
}
```

12. Let's check if there was any intersection and calculate its position and normal. If no intersection was found, we will simply set `mIntersects` to false.

```
if( resultIndex > -1 ){
    mHitPos = ray.calcPosition( resultDistance );
    mIntersects = true;
    Vec3f v1, v2, v3;
    mMesh.getTriangleVertices( resultIndex, &v1, &v2, &v3 );
    mNormal = ( v2 - v1 ).cross( v3 - v1 );
    mNormal.normalize();
} else {
    mIntersects = false;
}
```

13. With the intersection calculated, let's draw the model, intersection point, and normal. Start by clearing the background with black, setting the window's matrices using our camera, and enabling reading and writing to the depth buffer. Add the following code in the `draw` method:

```
gl::clear( Color( 0, 0, 0 ) );
gl::setMatrices( mCam.getCamera() );
gl::enableDepthRead();
gl::enableDepthWrite();
```

14. Now let's create a light and set its position as the camera's eye position. We'll also enable the light and apply the material.

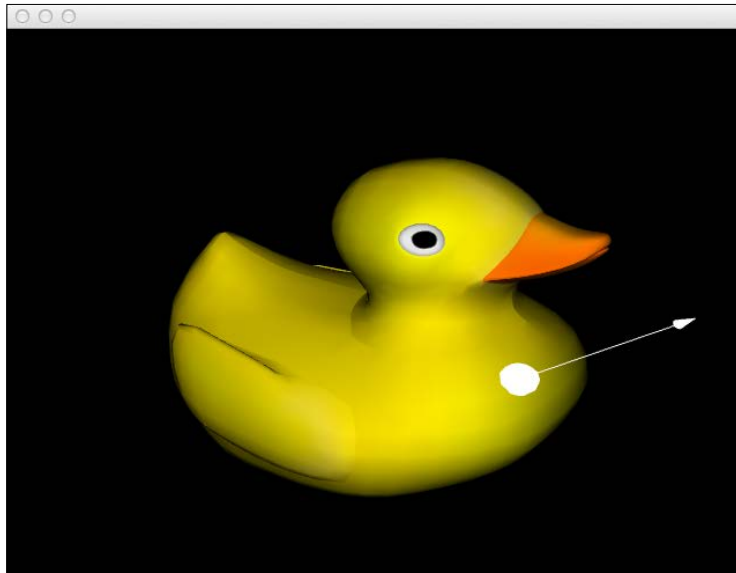
```
gl::Light light( gl::Light::POINT, 0 );
light.setPosition( mCam.getCamera().getEyePoint() );
light.setAttenuation( 1.0f, 0.0f, 0.0f );
glEnable( GL_LIGHTING );
light.enable();
mMaterial.apply();
```

15. Now enable and bind the models texture, draw the model, and disable both texture and lighting.

```
mTexture.enableAndBind();  
gl::draw( mMesh );  
mTexture.unbind();  
glDisable( GL_LIGHTING );
```

16. Finally, we will check if `mIntersects` is true and draw a sphere at the intersection point and the normal vector.

```
if( mIntersects ){  
    gl::color( Color::white() );  
    gl::drawSphere( mHitPos, 0.2f );  
    gl::drawVector( mHitPos, mHitPos + ( mNormal * 2.0f ) );  
}
```



How it works...

To calculate the intersection of the mouse with the model in 3D, we generated a ray from the mouse position towards the view direction of the camera.

For performance reasons, we first calculate if the ray intersects with the model's bounding box. In case there is an intersection with the model, we further calculate the intersection between the ray and each triangle that makes up the model. For every intersection found, we check its distance and calculate the intersection point and the normal of only the nearest intersection.

Creating a height map from an image

In this recipe, we will learn how to create a point cloud based on an image selected by the user. We will create a grid of points where each point will correspond to a pixel. The x and y coordinates of each point will be equal to the pixel's position on the image, and the z coordinate will be calculated based on its color.

Getting ready

Include the necessary files to work with OpenGL, image surfaces, VBO meshes, and loading images.

Add the following code to the top of the source file:

```
#include "cinder/gl/gl.h"
#include "cinder/Surface.h"
#include "cinder/gl/Vbo.h"
#include "cinder/MayaCamUI.h"
#include "cinder/ImageIo.h"
```

Also, add the following `using` statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will learn how to read pixel values from an image and create a point cloud.

1. Declare `ci::Surface32f` to store the image pixels, `ci::gl::VboMesh` that we will use as the point cloud, and `ci::MayaCamUI` for easy rotation of our scene.

```
Surface32f mImage;
gl::VboMesh mPointCloud; gl::VboMesh mPointCloud;
MayaCamUI mCam;
```

2. In the `setup` method, we will first open a file load dialog and then let the user select the image to use and check if it returns a valid path.

```
fs::path imagePath = getOpenFilePath( "",
ImageIo::getLoadExtensions() );
if( imagePath.empty() == false ){
```

3. Next, let's load the image and initialize `mPointCloud`. We will set the `ci::gl::VboMesh::Layout` to have dynamic positions and colors so that we will be able to change them later.

```
mImage = loadImage( imagePath );
int numPixels = mImage.getWidth() * mImage.getHeight();
gl::VboMesh::Layout layout;
layout.setDynamicColorsRGB();
layout.setDynamicPositions();
mPointCloud = gl::VboMesh( numPixels, 0, layout, GL_POINTS );
```

4. Next, we'll iterate over the image's pixels and update the vertices in `mPointCloud`.

```
Surface32f::IterpixelIt = mImage.getIter();
gl::VboMesh::VertexItervertexIt( mPointCloud );
while( pixelIt.line() ){
    while( pixelIt.pixel() ){
        Color color( pixelIt.r(), pixelIt.g(),
pixelIt.b() );
        float height = color.get( CM_RGB ).length();
        float x = pixelIt.x();
        float y = mImage.getHeight() - pixelIt.y();
        float z = height * 100.0f;
        vertexIt.setPosition( x,y, z );
        vertexIt.setColorRGB( color );
        ++vertexIt;
    }
}
```

5. Now we will set up the camera so that it will rotate around the center of the point cloud and close the `if` statement we began on the second step.

```
Vec3f center( (float)mImage.getWidth()/2.0f, (float)
mImage.getHeight()/2.0f, 50.0f );
CameraPersp camera( getWindowWidth(), getWindowHeight(), 60.0f
);
camera.setEyePoint( Vec3f( center.x, center.y, (float)mImage.
getHeight() ) );
camera.setCenterOfInterestPoint( center );
mCam.setCurrentCam( camera );
}
```

6. Let's declare and implement the necessary mouse event handlers to use `mCam`.

```
void mouseDown( MouseEvent event );
void mouseDrag( MouseEvent event );
```


7. And implement them:

```
void MyApp::mouseDown( MouseEvent event ){
    mCam.mouseDown( event.getPos() );
}

void MyApp::mouseDrag( MouseEvent event ){
    mCam.mouseDrag( event.getPos(), event.isLeft(), event.
        isMiddle(), event.isRight() );
}
```

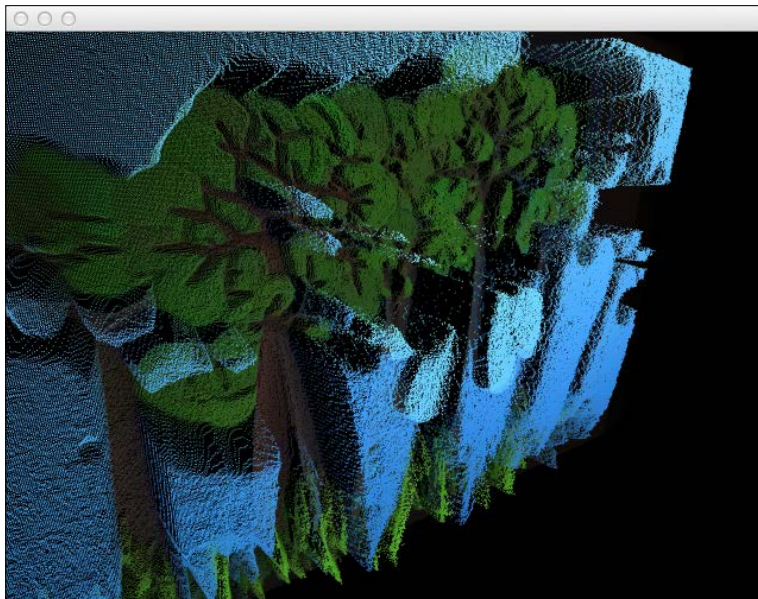
8. In the draw method, we will begin by clearing the background, setting the window's matrices defined by mCam, and enable reading and writing the depth buffer.

```
gl::clear( Color( 0, 0, 0 ) );
gl::setMatrices( mCam.getCamera() );
gl::enableDepthRead();
gl::enableDepthWrite();
```

9. Finally, we will check if mPointCloud is a valid object and draw it.

```
if( mPointCloud ){
    gl::draw( mPointCloud );
}
```

10. Build and run the application. You will be prompted with a dialog box to select an image file. Select it and you will see a point cloud representation of the image. Drag the mouse cursor to rotate the scene.



How it works...

We started by loading an image into `ci::Surface32f`. This surface stores pixels as float numbers in the range from 0 to 1.

We created a grid of points where the `x` and `y` coordinates represented the pixel's position on the image and the `z` coordinate was the length of the color's vector.

The point cloud is represented by a `ci::gl::VboMesh`, which is a mesh of vertices, normal, colors, and indexes with an underlying Vertex Buffer Object. It allows for optimized drawing of geometry.

Creating a terrain with Perlin noise

In this recipe, we will learn how to construct a surface in 3D using **Perlin noise** to create organic deformations that resemble a piece of terrain.

Getting ready

Include the necessary files to draw using OpenGL, Perlin noise, a Maya camera for navigation, and Cinder's math utilities. Add the following code to the top of the source file:

```
#include "cinder/gl/gl.h"
#include "cinder/Perlin.h"
#include "cinder/MayaCamUI.h"
#include "cinder/CinderMath.h"
```

Also, add the following `using` statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will create a grid of 3D points and use Perlin noise to calculate a smooth surface.

1. Declare `struct` to store the vertices of the terrain by adding the following code before the applications class declaration:

```
struct Vertice{
    Vec3f position;
    Color color;
};
```

2. Add the following members to the applications class declaration:

```
vector< vector<Vertice> > mTerrain;  
int mNumRows, mNumLines;  
MayaCamUI mCam;  
Perlin mPerlin;
```

3. In the `setup` method, define the number of rows and lines that will make up the terrain's grid. Also, define the gap distance between each point.

```
mNumRows = 50;  
mNumLines = 50;  
float gap = 5.0f;
```

4. Add the vertices to `mTerrain` by creating a grid of points laid on the `x` and `z` axis. We will use the values generated by `ci::Perlin` to calculate each points height. We will also use the height of the points to define their color:

```
mTerrain.resize( mNumRows );  
for( int i=0; i<mNumRows; i++ ){  
    mTerrain[i].resize( mNumLines );  
    for( int j=0; j<mNumLines; j++ ){  
        float x = (float)i * gap;  
        float z = (float)j * gap;  
        float y = mPerlin.noise( x*0.01f, z*0.01 ) * 100.0f;  
        mTerrain[i][j].position = Vec3f( x, y, z );  
        float colorVal = lmap( y, -100.0f, 100.0f, 0.0f, 1.0f  
    );  
        mTerrain[i][j].color = Color( colorVal, colorVal,  
colorVal );  
    }  
}
```

5. Now let's define our camera so that it points to the center of the terrain.

```
float width = mNumRows * gap;  
float height = mNumLines * gap;  
Vec3f center( width/2.0f, height/2.0f, 0.0f );  
Vec3f eye( center.x, center.y, 300.0f );  
CameraPersp camera( getWindowWidth(), getWindowHeight(), 60.0f );  
camera.setEyePoint( eye );  
camera.setCenterOfInterestPoint( center );  
mCam.setCurrentCam( camera );
```

6. Declare the mouse event handlers to use `mCam`.

```
void mouseDown( MouseEvent event );  
void mouseDrag( MouseEvent event );  
}
```

7. Now let's implement the mouse handlers.

```
void MyApp::mouseDown( MouseEvent event ){  
    mCam.mouseDown( event.getPos() );  
}  
void MyApp::mouseDrag( MouseEvent event ){  
    mCam.mouseDrag( event.getPos(), event.isLeft(), event.  
        isMiddle(), event.isRight() );  
}
```

8. In the draw method, let's start by clearing the background, setting the matrices using `mCam`, and enabling reading and writing of the depth buffer.

```
gl::clear( Color( 0, 0, 0 ) );  
gl::setMatrices( mCam.getCamera() );  
gl::enableDepthRead();  
gl::enableDepthWrite();
```

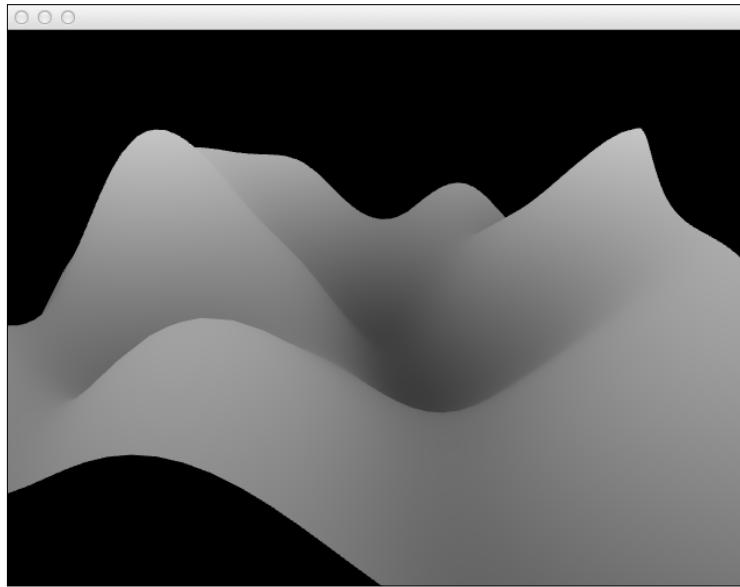
9. Now enable OpenGL to use the `VERTEX` and `COLOR` arrays:

```
glEnableClientState( GL_VERTEX_ARRAY );  
glEnableClientState( GL_COLOR_ARRAY );
```

10. We will use a nested `for` loop to iterate over the terrain and draw each strip of terrain as `GL_TRIANGLE_STRIP`.

```
for( int i=0; i<mNumRows-1; i++ ){  
    vector<Vec3f> vertices;  
    vector<ColorA> colors;  
    for( int j=0; j<mNumLines; j++ ){  
  
        vertices.push_back( mTerrain[i][j].position );  
        vertices.push_back( mTerrain[i+1][j].position );  
        colors.push_back( mTerrain[i][j].color );  
        colors.push_back( mTerrain[i+1][j].color );  
  
    }  
}
```

```
glColor3f( 1.0f, 1.0f, 1.0f );
glVertexPointer( 3, GL_FLOAT, 0, &vertices[0] );
glColorPointer( 4, GL_FLOAT, 0, &colors[0] );
glDrawArrays( GL_TRIANGLE_STRIP, 0, vertices.size() );
}
```



How it works...

Perlin noise is a coherent random number generator capable of creating organic textures and transitions.

We used the values created by the `ci::Perlin` object to calculate the height of the vertices that make up the terrain and create smooth transitions between vertices.

There's more...

We can also animate our terrain by adding an increasing offset to the coordinates used to calculate the Perlin noise. Declare the following member variables in your class declaration:

```
float offsetX, offsetZ;
```

In the `setup` method, initialize them.

```
offsetX = 0.0f;
offsetZ = 0.0f;
```

In the `update` method animate each offset value by adding `0.01`.

```
offsetX += 0.01f;
offsetZ += 0.01f;
```

Also in the `update` method, we will iterate over all the vertices of `mTerrain`. For each vertex we will use its `x` and `z` coordinates to calculate the `Y` coordinate with `mPerlin` noise, but we will offset the coordinates.

```
for( int i=0; i<mNumRows; i++ ){
  for( int j=0; j<mNumLines; j++ ){
    Vertice& vertice = mTerrain[i][j];
    float x = vertice.position.x;
    float z = vertice.position.z;
    float y = mPerlin.noise( x*0.01f + offsetX, z*0.01f + offsetZ ) *
    100.0f;
        vertice.position.y = y;
    }
  }
```

Saving mesh data

Provided that you are using a `TriMesh` class to store 3D geometry, we will show you how to save it in a file.

Getting ready

We are assuming that you are using a 3D model stored in `TriMesh` object. Sample application loading 3D geometry can be found in `Cinder` `samples` directory in the folder: `OBJLoaderDemo`.

How to do it...

We will implement saving a 3D mesh data.

1. Include necessary headers:
2. Implement your `keyDown` method as follows:

```
#include "cinder/ObjLoader.h"
#include "cinder/Utilities.h"

if( event.getChar() == 's' ) {
  fs::path path = getSaveFilePath(getDocumentsDirectory() /
  fs::path("mesh.trimesh") );
```

```
if( ! path.empty() ) {
    mMesh.write( writeFile( path ) );
}
else if( event.getChar() == 'o' ) {
    fs::path path = getSaveFilePath(getDocumentsDirectory() /
    fs::path("mesh.obj") );
    if( ! path.empty() ) {
        ObjLoader::write( writeFile( path ), mMesh );
    }
}
```

How it works...

In Cinder we are using a `TriMesh` class to store 3D geometry. Using `TriMesh` we can store and manipulate geometry loaded from 3D model files or add each vertices with code.

Every time you hit the S key on the keyboard, a saving dialog pops up to ask you where to save binary data of the `TriMesh` object. When you press the O key, the OBJ format file will be saved into your `documents` folder. If you don't have to exchange data with other software, binary data saving and loading is usually faster.

9

Adding Animation

In this chapter, we will learn the techniques of animating 2D and 3D objects. We will introduce Cinder's features in this field, such as `timeline` and `math` functions.

The recipes in this chapter will cover the following:

- ▶ Animating with the timeline
- ▶ Creating animation sequences with the timeline
- ▶ Animating along a path
- ▶ Aligning camera motion to a path
- ▶ Animating text – text as a mask for a movie
- ▶ Animating text – scrolling text lines
- ▶ Creating a flow field with Perlin noise
- ▶ Creating an image gallery in 3D
- ▶ Creating a spherical flow field with Perlin noise

Animating with the timeline

In this recipe, we will learn how we can animate values using Cinder's new feature; the timeline.

We animate the background color and a circle's position and radius whenever the user presses the mouse button.

Getting ready

Include the necessary files to use the timeline, generate random numbers, and draw using OpenGL. Add the following code snippet at the top of the source file:

```
#include "cinder/gl/gl.h"
#include "cinder/Timeline.h"
#include "cinder/Rand.h"
```

Also, add the following useful using statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will create several parameters that will be animated with the timeline. Perform the following steps to do so:

1. Declare the following members to be animated:

```
Anim<Color> mBackgroundColor;
Anim<Vec2f> mCenter;
Anim<float> mRadius;
```

2. Initialize the parameters in the setup method.

```
mBackgroundColor = Color( CM_HSV, randFloat(), 1.0f, 1.0f );
mCenter = getWindowCenter();
mRadius = randFloat( 20.0f, 100.0f );
```

3. In the draw method, we need to clear the background using the color defined in mBackgroundColor and draw a circle at mCenter with mRadius as the radius.

```
gl::clear( mBackgroundColor.value() );
gl::drawSolidCircle( mCenter.value(), mRadius.value() );
```

4. To animate the values whenever the user presses the mouse button, we need to declare the mouseDown event handler.

```
void mouseDown( MouseEvent event );
```

5. Let's implement the mouseDown event handler and add the animations to the main timeline. We will animate mBackgroundColor to a new random color, set mCenter to the mouse cursor's position, and set mRadius to a new random value.

```
Color backgroundColor( CM_HSV, randFloat(), 1.0f, 1.0f );
timeline().apply( &mBackgroundColor, backgroundColor, 2.0f,
EaseInCubic() );
```

```

timeline().apply( &mCenter, (Vec2f)event.getPos(), 1.0f,
EaseInCirc() );
timeline().apply( &mRadius, randFloat( 20.0f, 100.0f ), 1.0f,
EaseInQuad() );

```

How it works...

The timeline is a new feature of Cinder introduced in version 0.8.4. It permits the user to animate parameters by adding them to the timeline once, and everything gets updated behind the scenes.

Animations must be objects of the template class `ci::Anim`. This class can be created using any template type that supports the `+` operator.

The main `ci::Timeline` object can be accessed by calling the `ci::app::App::timeline()` method. There is always a main timeline and the user can also create other `ci::Timeline` objects.

The fourth parameter in the `ci::Timeline::apply` method is a functor object that represents a Tween method. Cinder has several Tweens available that can be passed as a parameter to define the type of animation.

There's more...

The `ci::Timeline::apply` method used in the preceding example uses the initial value of the `ci::Anim` object, but it is also possible to create an animation where both the beginning and end values are passed.

For example, if we wanted to animate `mRadius` from a starting value of 10.0 to the end value of 100.0 seconds, we would call the following method:

```

timeline().apply( &mRadius, 10.0f, 100.0f 1.0f, EaseInQuad() );

```

See also

- ▶ To see all the available easing functions, please refer to the Cinder documentation, located at http://libcinder.org/docs/v0.8.4/_easing_8h.html.

Creating animation sequences with the timeline

In this recipe, we will learn how to use the powerful timeline features of Cinder to create sequences of animations. We will draw a circle and animate the radius and color in a sequenced manner.

Getting ready

Include the necessary files to use the timeline, draw in OpenGL, and generate random numbers.

```
#include "cinder/gl/gl.h"
#include "cinder/Timeline.h"
#include "cinder/Rand.h"
```

Also, add the following useful `using` statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will animate several parameters sequentially using the timeline. Perform the following steps to do so:

1. Declare the following members to define the circle's position, radius, and color:

```
Anim<float> mRadius;
Anim<Color> mColor;
Vec2f mPos;
```

2. In the `setup` method, initialize the members. Set the position to be at the center of the window, the radius as 30, and a random color using the HSV color mode.

```
mPos = (Vec2f)getWindowCenter();
mRadius = 30.0f;
mColor = Color( CM_HSV, randFloat(), 1.0f, 1.0f );
```

3. In the `draw` method, we will clear the background with black and draw the circle using the previously defined members.

```
gl::clear( Color::black() );
gl::color( mColor.value() );
gl::drawSolidCircle( mPos, mRadius.value() );
```

4. Declare the `mouseDown` event handler.

```
void mouseDown( MouseEvent event );
```

5. In the implementation of `mouseDown`, we will apply the animations to the main timeline.

We will first animate `mRadius` from 30 to 200 and append another animation to `mRadius` from 200 to 30.

Add the following code snippet to the `mouseDown` method:

```
timeline().apply( &mRadius, 30.0f, 200.0f, 2.0f, EaseInOutCubic()
);
timeline().appendTo( &mRadius, 200.0f, 30.0f, 1.0f,
EaseInOutCubic() );
```

- Let's create a random color using the HSV color mode and use it as the target color to animate `mColor` and then append this animation to `mRadius`.

Add the following code snippet inside the `mouseDown` method:

```
Color targetColor = Color( CM_HSV, randFloat(), 1.0f, 1.0f );
timeline().apply( &mColor, targetColor, 1.0f, EaseInQuad()
).appendTo( &mRadius );
```

How it works...

Appending animations is a powerful and easy way to create complex animation sequences.

In step 5 we append an animation to `mRadius` using the following line of code:

```
timeline().appendTo( &mRadius, 200.0f, 30.0f, 1.0f, EaseInOutCubic()
);
```

This means this animation will only occur after the previous `mRadius` animation has finished.

In step 6 we append the `mColor` animation to `mRadius` using the following line of code:

```
timeline().apply( &mColor, targetColor, 1.0f, EaseInQuad() ).appendTo(
&mRadius );
```

This means the `mColor` animation will only occur when the previous `mRadius` animation has finished.

There's more...

When appending two different animations, it is possible to offset the start time by defining the offset seconds as a second parameter.

So, for example, change the line in step 6 to read the following:

```
timeline().apply( &mColor, targetColor, 1.0f, EaseInQuad() ).appendTo(
&mRadius, -0.5f );
```

This would mean that the `mColor` animation would begin 0.5 seconds before `mRadius` has finished.

Animating along a path

In this recipe, we will learn how to draw a smooth B-spline in the 3D space and animate the position of an object along the calculated B-spline.

Getting ready

To navigate in the 3D space, we will use `MayaCamUI` covered in the *Using MayaCamUI* recipe in *Chapter 2, Preparing for Development*.

How to do it...

We will create an example animation of an object moving along the spline. Perform the following steps to do so:

1. Include necessary header files.

```
#include "cinder/Rand.h"
#include "cinder/MayaCamUI.h"
#include "cinder/BSpline.h"
```

2. Begin with the declaration of member variables to keep the B-spline and current object's position.

```
Vec3f      mObjPosition;
BSpline3f  spline;
```

3. Inside the `setup` method prepare a random spline:

```
mObjPosition = Vec3f::zero();

vector<Vec3f> splinePoints;
float step = 0.5f;
float width = 20.f;
for (float t = 0.f; t < width; t += step) {
    Vec3f pos = Vec3f(
        cos(t)*randFloat(0.f,2.f),
        sin(t)*0.3f,
        t - width*0.5f);
    splinePoints.push_back( pos );
}
spline = BSpline3f( splinePoints, 3, false, false );
```

4. Inside the `update` method, retrieve the position of the object moving along the spline.

```
float dist = math<float>::abs(sin( getElapsedSeconds()*0.2f ));
mObjPosition = spline.getPosition( dist );
```

5. The code snippet drawing our scene will look like the following:

```
gl::enableDepthRead();
gl::enableDepthWrite();
gl::enableAlphaBlending();
gl::clear( Color::white() );
gl::setViewport( getWindowBounds() );
gl::setMatrices( mMayaCam.getCamera() );

// draw dashed line
gl::color( ColorA(0.f, 0.f, 0.f, 0.8f) );
float step = 0.005f;
glBegin( GL_LINES );
for (float t = 0.f; t <= 1.f; t += step) {
    gl::vertex( spline.getPosition(t) );
}
glEnd();

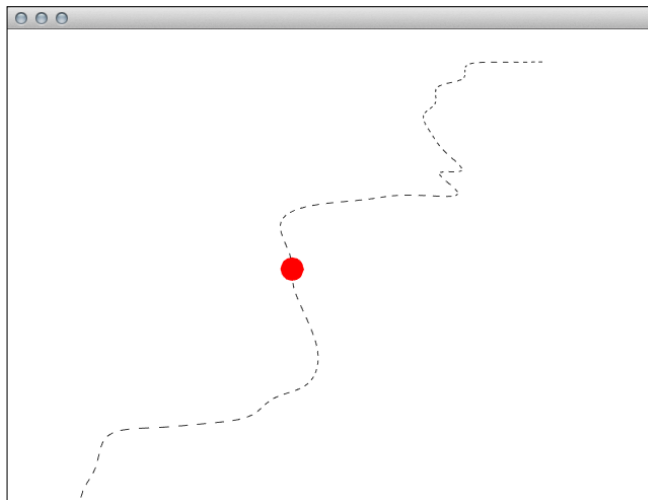
// draw object
gl::color( Color(1.f, 0.f, 0.f) );
gl::drawSphere( mObjPosition, 0.25f );
```

How it works...

First, have a look at step 3 where we are calculating a B-spline through points with coordinates based on the sine and cosine functions and some random points on the x axis. The path is stored in the `spline` class member.

Then we can easily retrieve the position in 3D space at any distance of our path. We are doing this in step 4; using the `getPosition` method on the `spline` member. The distance on the path is been passed as a float value in the range of 0.0 to 1.0 where 0.0 means the beginning of the path and 1.0 means the end.

Finally, in step 5 we are drawing an animation as a red sphere traveling along our path, represented as a black dashed line, as shown in the following screenshot:



See also

- ▶ The *Aligning camera motion to path* recipe
- ▶ The *Animating text around curves* recipe in *Chapter 7, Using 2D Graphics*

Aligning camera motion to a path

In this recipe we will learn how we can animate the camera position on our path, calculated as a B-spline.

Getting ready

In this example, we will use `MayaCamUI`, so please refer to the *Using MayaCamUI* recipe in *Chapter 2, Preparing for Development*.

How to do it...

We will create an application illustrating the mechanism. Perform the following steps to do so:

1. Include necessary header files.

```
#include "cinder/Rand.h"
#include "cinder/MayaCamUI.h"
#include "cinder/BSpline.h"
```

2. Begin with the declaration of member variables.

```

MayaCamUI mMayaCam;
BSpline3f spline;
CameraPersp mMovingCam;
Vec3f mCamPosition;
vector<Rectf> mBoxes;

```

3. Set up the initial values of members.

```

setWindowSize(640*2, 480);
mCamPosition = Vec3f::zero();

CameraPersp mSceneCam;
mSceneCam.setPerspective(45.0f, 640.f/480.f, 0.1, 10000);
mSceneCam.setEyePoint(Vec3f(7.f,7.f,7.f));
mSceneCam.setCenterOfInterestPoint(Vec3f::zero());
mMayaCam.setCurrentCam(mSceneCam);

mMovingCam.setPerspective(45.0f, 640.f/480.f, 0.1, 100.f);
mMovingCam.setCenterOfInterestPoint(Vec3f::zero());

vector<Vec3f> splinePoints;
float step = 0.5f;
float width = 20.f;
for (float t = 0.f; t < width; t += step) {
    Vec3f pos = Vec3f( cos(t)*randFloat(0.8f,1.2f),
        0.5f+sin(t*0.5f)*0.5f,
        t - width*0.5f);
    splinePoints.push_back( pos );
}
spline = BSpline3f( splinePoints, 3, false, false );

for(int i = 0; i<100; i++) {
    Vec2f pos = Vec2f(randFloat(-10.f,10.f),
        randFloat(-10.f,10.f));
    float size = randFloat(0.1f,0.5f);
    mBoxes.push_back(Rectf(pos, pos+Vec2f(size,size*3.f)));
}

```

4. Inside the update method update the camera properties.

```

float step = 0.001f;
float pos = abs(sin( getElapsedSeconds()*0.05f ));
pos = min(0.99f, pos);
mCamPosition = spline.getPosition( pos );

mMovingCam.setEyePoint(mCamPosition);
mMovingCam.lookAt(spline.getPosition( pos+step ));

```


5. The whole draw method now looks like the following code snippet:

```
gl::enableDepthRead();
gl::enableDepthWrite();
gl::enableAlphaBlending();
gl::clear( Color::white() );
gl::setViewport( getWindowBounds() );
gl::setMatricesWindow( getWindowSize() );

gl::color( ColorA( 0.f, 0.f, 0.f, 1.f ) );
gl::drawLine( Vec2f( 640.f, 0.f ), Vec2f( 640.f, 480.f ) );

gl::pushMatrices();
gl::setViewport( Area( 0, 0, 640, 480 ) );
gl::setMatrices( mMayaCam.getCamera() );

drawScene();

// draw dashed line
gl::color( ColorA( 0.f, 0.f, 0.f, 0.8f ) );
float step = 0.005f;
glBegin( GL_LINES );
for ( float t = 0.f; t <= 1.f; t += step ) {
    gl::vertex( spline.getPosition( t ) );
}
glEnd();

// draw object
gl::color( Color( 0.f, 0.f, 1.f ) );
gl::drawFrustum( mMovingCam );

gl::popMatrices();

// -----

gl::pushMatrices();
gl::setViewport( Area( 640, 0, 640*2, 480 ) );
gl::setMatrices( mMovingCam );
drawScene();
gl::popMatrices();
```

6. Now we have to implement the drawScene method, which actually draws our 3D scene.

```
GLfloat light0_position[] = { 1000.f, 500.f, -500.f, 0.1f };
GLfloat light1_position[] = { -1000.f, 100.f, 500.f, 0.1f };
```

```

GLfloat light1_color[] = { 1.f, 1.f, 1.f };

glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
glLightfv( GL_LIGHT1, GL_POSITION, light1_position );
glLightfv( GL_LIGHT1, GL_DIFFUSE, light1_color );

glEnable( GL_LIGHTING );
glEnable( GL_LIGHT0 );
glEnable( GL_LIGHT1 );

ci::ColorA diffuseColor(0.9f, 0.2f, 0.f );
gl::color(diffuseColor);
glMaterialfv( GL_FRONT, GL_DIFFUSE,  diffuseColor );

vector<Rectf>::iterator it;
for(it = mBoxes.begin(); it != mBoxes.end(); ++it) {
    gl::pushMatrices();
    gl::translate(0.f, it->getHeight()*0.5f, 0.f);
    Vec2f center = it->getCenter();
    gl::drawCube(Vec3f(center.x, 0.f, center.y),
        Vec3f(it->getWidth(),
            it->getHeight(), it->getWidth()));
    gl::popMatrices();
}

glDisable( GL_LIGHTING );
glDisable( GL_LIGHT0 );
glDisable( GL_LIGHT1 );

// draw grid
drawGrid(50.0f, 2.0f);

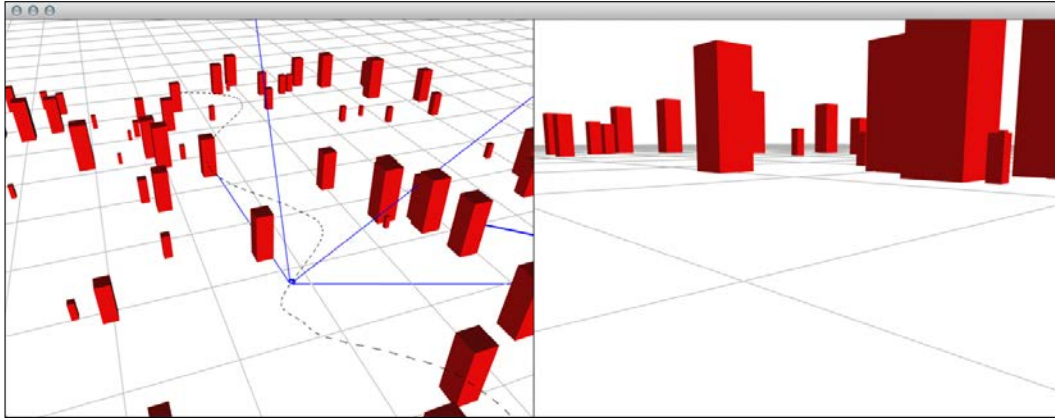
```

7. The last thing we need is the `drawGrid` method, the implementation of which can be found in the *Using 3D space guides* recipe in *Chapter 2, Preparing for Development*.

How it works...

In this example we are using a B-spline as a path that our camera is moving along. Please refer to the *Animating along a path* recipe to see the basic implementation of an object animating on a path. As you can see in step 4 we are setting the camera position by invoking the `setEyePosition` method on the `mMovingCam` member, and we have to set the camera view direction. To do that we are taking the position of the next point on the path and passing it to the `lookAt` method.

We are drawing a split screen, where on the left-hand side is a preview of our scene, and on the right-hand side we can see what is in a frustum of the camera moving along the path.



See also

- ▶ The *Animating along a path* recipe
- ▶ The *Using 3D space guides* recipe in *Chapter 2, Preparing for Development*
- ▶ The *Using MayaCamUI* recipe in *Chapter 2, Preparing for Development*

Animating text – text as a mask for a movie

In this recipe, we will learn how we can use text as a mask for a movie using a simple shader program.

Getting ready

In this example, we are using one of the amazing videos provided by NASA taken by an ISS crew that you can find at <http://eol.jsc.nasa.gov/>. Please download one and save it as `video.mov` inside the `assets` folder.

How to do it...

We will create a sample Cinder application to illustrate the mechanism. Perform the following steps to do so:

1. Include the necessary header files.

```
#include "cinder/gl/Texture.h"  
#include "cinder/Text.h"
```

```
#include "cinder/Font.h"
#include "cinder/qtime/QuickTime.h"
#include "cinder/gl/GlslProg.h"
```

2. Declare the member variables.

```
qtime::MovieGl mMovie;
gl::Texture      mFrameTexture, mTextTexture;
gl::GlslProg     mMaskingShader;
```

3. Implement the setup method, as follows:

```
setWindowSize(854, 480);

TextLayout layout;
layout.clear( ColorA(0.f,0.f,0.f, 0.f) );
layout.setFont( Font("Arial Black", 96) );
layout.setColor( Color( 1, 1, 1 ) );
layout.addLine( "SPACE" );
Surface8u rendered = layout.render( true );

gl::Texture::Format format;
format.setTargetRect();
mTextTexture = gl::Texture( rendered, format );

try {
    mMovie = qtime::MovieGl( getAssetPath("video.mov") );
    mMovie.setLoop();
    mMovie.play();
} catch( ... ) {
    console() <<"Unable to load the movie."<<endl;
    mMovie.reset();
}

mMaskingShader = gl::GlslProg( loadAsset("passThru_vert.glsl"),
loadAsset("masking_frag.glsl") );
```

4. Inside the update method we have to update our mFrameTexture where we are keeping the current movie frame.

```
if( mMovie ) mFrameTexture = mMovie.getTexture();
```

5. The draw method will look like the following code snippet:

```
gl::enableAlphaBlending();
gl::clear( Color::gray(0.05f) );
gl::setViewport( getWindowBounds() );
gl::setMatricesWindow( getWindowSize() );
```

```

gl::color(ColorA::white());
if(mFrameTexture) {
    Vec2f maskOffset = (mFrameTexture.getSize()
        - mTextTexture.getSize() ) * 0.5f;
    mFrameTexture.bind(0);
    mTextTexture.bind(1);
    mMaskingShader.bind();
    mMaskingShader.uniform("sourceTexture", 0);
    mMaskingShader.uniform("maskTexture", 1);
    mMaskingShader.uniform("maskOffset", maskOffset);
    gl::pushMatrices();
    gl::translate(getWindowCenter()-mTextTexture.getSize()*0.5f);
    gl::drawSolidRect( mTextTexture.getBounds(), true );
    gl::popMatrices();
    mMaskingShader.unbind();
}

```

6. As you can see in the `setup` method we are loading a shader to do the masking. We have to pass through vertex shader inside the `assets` folder in a file named `passThru_vert.glsl`. You can find this in the *Implementing 2D metaballs* recipe in *Chapter 7, Using 2D Graphics*.
7. Finally, the fragment shader program code will look like the following code snippet, and should also be inside the `assets` folder under the name `masking_frag.glsl`.

```

#extension GL_ARB_texture_rectangle : require

uniform sampler2DRect sourceTexture;
uniform sampler2DRect maskTexture;
uniform vec2 maskOffset;

void main()
{
    vec2 texCoord = gl_TexCoord[0].st;

    vec4 sourceColor = texture2DRect( sourceTexture,
    texCoord+maskOffset );
    vec4 maskColor = texture2DRect( maskTexture, texCoord );

    vec4 color = sourceColor * maskColor;

    gl_FragColor = color;
}

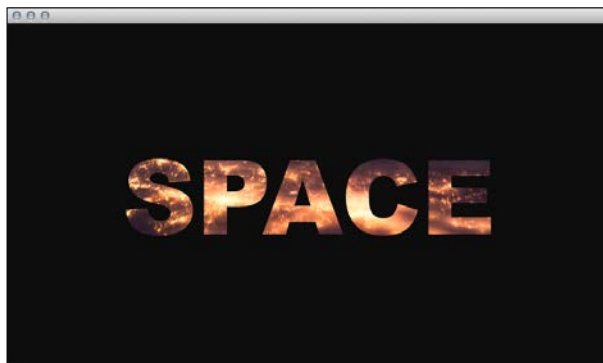
```

How it works...

Inside the `setup` method in step 3 we are rendering our text as `Surface` and then converting it to `gl::Texture` that we will use later as a masking texture. It is important here to set a rectangle format for masking texture while we are using it as a mask for a movie, because `qtime::MovieGl` is creating a texture with a frame that is rectangular. To do that we are defining the `gl::Texture::Format` object named `format` and invoking the `setTargetRect` method on it. While creating `gl::Texture` we have to pass `format` to the constructor as a second parameter.

To draw a movie frame we are using our masking shader program applied on the rectangle in step 5. We have to pass three parameters, which are the movie frame as `sourceTexture`, mask texture with text as `maskTexture`, and the position of the mask as `maskOffset`.

In step 7 you can see the fragment shader code, which simply multiplies the colors of the corresponding pixels from `sourceTexture` and `maskTexture`. Please notice that we are using `sampler2DRect` and `texture2DRect` to handle rectangular textures.



Animating text – scrolling text lines

In this recipe we will learn how we can create text scrolling line-by-line.

How to do it...

We will now create an animation with scrolling text. Perform the following steps to do so:

1. Include the necessary header files.

```
#include "cinder/gl/Texture.h"
#include "cinder/Text.h"
#include "cinder/Font.h"
#include "cinder/Utilities.h"
```

2. Add the member values.

```
vector<gl::Texture> mTextTextures;  
Vec2f mTextSize;
```

3. Inside the setup method we need to generate textures for each line of text.

```
setWindowSize(854, 480);  
string font( "Times New Roman" );  
  
mTextSize = Vec2f::zero();  
}  
for(int i = 0; i<5; i++) {  
    TextLayout layout;  
    layout.clear( ColorA(0.f,0.f,0.f, 0.f) );  
    layout.setFont( Font( font, 48 ) );  
    layout.setColor( Color( 1, 1, 1 ) );  
    layout.addLine( "Animating text " + toString(i) );  
    Surface8u rendered = layout.render( true );  
    gl::Texture textTexture = gl::Texture( rendered );  
    textTexture.setMagFilter(GL_NICEST);  
    textTexture.setMinFilter(GL_NICEST);  
    mTextTextures.push_back(textTexture);  
    mTextSize.x = math<float>::max(mTextSize.x,  
        textTexture.getWidth());  
    mTextSize.y = math<float>::max(mTextSize.y,  
        textTexture.getHeight());  
}
```

4. The draw method for this example looks as follows:

```
gl::enableAlphaBlending();  
gl::clear( Color::black() );  
gl::setViewport( getWindowBounds() );  
gl::setMatricesWindowPersp( getWindowSize() );  
  
gl::color( ColorA::white() );  
  
float time = getElapsedSeconds()*0.5f;  
float timeFloor = math<float>::floor( time );  
int textIdx = 1 + ( (int)timeFloor % (mTextTextures.size()-1) );  
float step = time - timeFloor;  
  
gl::pushMatrices();  
gl::translate( getWindowCenter() - mTextSize*0.5f );  
  
float radius = 30.f;
```

```

gl::color(ColorA(1.f,1.f,1.f, 1.f-step));
gl::pushMatrices();
gl::rotate( Vec3f(90.f*step,0.f,0.f) );
gl::translate(0.f,0.f,radius);
gl::draw(mTextTextures[texIdx-1], Vec2f(0.f,
-mTextTextures[texIdx-1].getHeight()*0.5f) );
gl::popMatrices();

gl::color(ColorA(1.f,1.f,1.f, step));
gl::pushMatrices();
gl::rotate( Vec3f(-90.f + 90.f*step,0.f,0.f) );
gl::translate(0.f,0.f,radius);
gl::draw(mTextTextures[texIdx], Vec2f(0.f, -mTextTextures[texIdx] .
getHeight()*0.5f) );
gl::popMatrices();

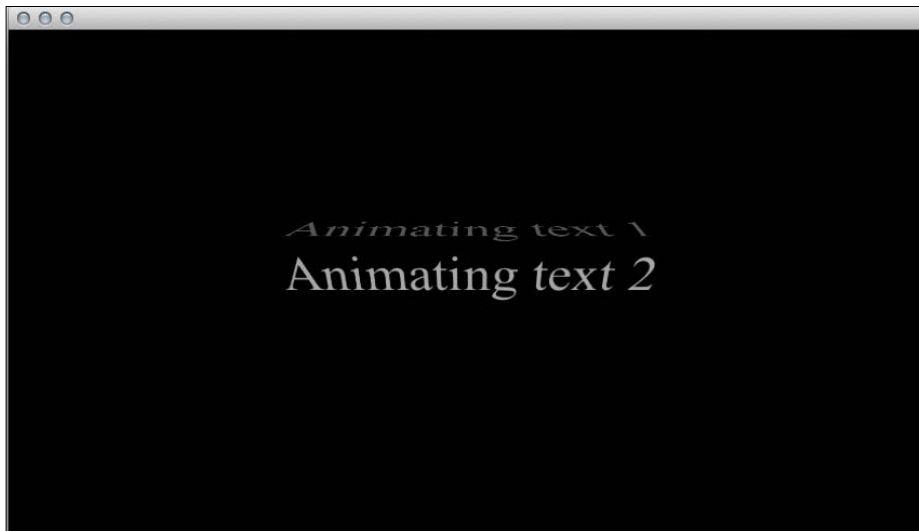
gl::popMatrices();

```

How it works...

What we are doing first inside the `setup` method in step 3 is generating a texture with rendered text for each line and pushing it to the vector structure `mTextTextures`.

In step 4 you can find the code for drawing current and previous text to build a continuous looped animation.



Creating a flow field with Perlin noise

In this recipe we will learn how we can animate objects using a flow field. Our flow field will be a two-dimensional grid of velocity vectors that will influence how objects move.

We will also animate the flow field using vectors calculated with Perlin noise.

Getting ready

Include the necessary files to work with OpenGL graphics, Perlin noise, random numbers, and Cinder's math utilities.

```
#include "cinder/gl/gl.h"
#include "cinder/Perlin.h"
#include "cinder/Rand.h"
#include "cinder/CinderMath.h"
```

Also, add the following useful using statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will create an animation using the flow field. Perform the following steps to do so:

1. We will begin by creating a `Follower` class to define the objects that will be influenced by the flow field.

Declare the following class before the main application class:

```
class Follower{
public:
    Follower( const Vec2f& pos ){
        this->pos = pos;
    }
    void update( const Vec2f& newVel ){
        vel += ( newVel - vel ) * 0.2f;
        pos += vel;
        if( pos.x < 0.0f ){
            pos.x = (float)getWindowWidth();
            vel = Vec2f();
        }
        if( pos.x > (float)getWindowWidth() ){
```

```

    pos.x = 0.0f;
    vel = Vec2f();
}
if( pos.y < 0.0f ){
    pos.y = (float)getWindowHeight();
    vel = Vec2f();
}
if( pos.y > (float)getWindowHeight() ){
    pos.y = 0.0f;
    vel = Vec2f();
}
}
void draw(){
    gl::drawSolidCircle( pos, 5.0f );
    gl::drawLine( pos, pos + ( vel * 20.0f ) );
}
Vec2f pos, vel;
};

```

- Let's create the flow field. Declare a two-dimensional `std::vector` to define the flow field, and variables to define the gap between vectors and the number of rows and columns.

```

vector< vector< Vec2f> > mFlowField;
Vec2f mGap;
float mCounter;
int mRows, mColumns;

```

- In the `setup` method we will define the number of rows and columns, and calculate the gap between each vector.

```

mRows = 40;
mColumns = 40;
mGap.x = (float)getWindowWidth() / (mRows-1);
mGap.y = (float)getWindowHeight() / (mColumns-1);

```

- Based on the number of rows and columns we can initialize `mFlowField`.

```

mFlowField.resize( mRows );
for( int i=0; i<mRows; i++ ){
    mFlowField[i].resize( mColumns );
}

```

- Let's animate the flow field using Perlin noise. To do so declare the following members:

```

Perlin mPerlin;
float mCounter;

```

6. In the `setup` method initialize `mCounter` to zero.

```
mCounter = 0.0f;
}
```

7. In the `update` method we will increment `mCounter` and iterate `mFlowField` using a nested `for` loop, and use `mPerlin` to animate the vectors.

```
for( int i=0; i<mRows; i++ ){
  for( int j=0; j<mColumns; j++ ){
    float angle= mPerlin.noise( ((float)i)*0.01f + mCounter,
      ((float)j)*0.01f ) * M_PI * 2.0f;
    mFlowField[i][j].x = cosf( angle );
    mFlowField[i][j].y = sinf( angle );
  }
}
```

8. Now iterate over `mFlowField` and draw a line indicating the direction of the vectors.

Add the following code snippet inside the `draw` method:

```
for( int i=0 i<mRows; i++ ){
  for( int j=0; j<mColumns; j++ ){
    float x = (float)i*mGap.x;
    float y = (float)j*mGap.y;
    Vec2f begin( x, y );
    Vec2f end = begin + ( mFlowField[i][j] * 10.0f );
    gl::drawLine( begin, end );
  }
}
```

9. Let's add some `Followers`. Declare the following member:

```
vector<shared_ptr<Follower>> mFollowers;
```

10. In the `setup` method we will initialize some followers and add them at random positions in the window.

```
int numFollowers = 50;
for( int i=0; i<numFollowers; i++ ){
  Vec2f pos( randFloat( getWindowWidth() ),
    randFloat( getWindowHeight() ) );
  mFollowers.push_back(
    shared_ptr<Follower>( new Follower( pos ) ) );
}
```

11. In the `update` we will iterate `mFollowers` and calculate the corresponding vector in `mFlowField` based on its position.

We will then update the `Follower` class using that vector.

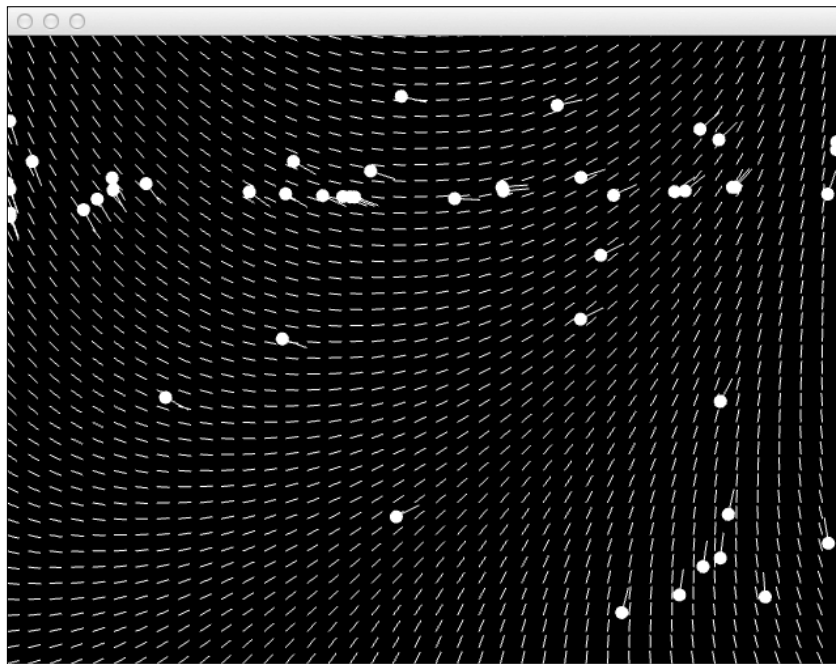
```
for( vector<shared_ptr<Follower> >::iterator it =
mFollowers.begin(); it != mFollowers.end(); ++it ){
    shared_ptr<Follower> follower = *it;
    int indexX= ci::math<int>::clamp(follower->pos.x / mGap.x,0,
        mRows-1 );
    int indexY= ci::math<int>::clamp(follower->pos.y / mGap.y,0,
        mColumns-1 );
    Vec2f flow = mFlowField[ indexX ][ indexY ];
    follower->update( flow );
}
```

12. Finally, we just need to draw each `Follower` class.

Add the following code snippet inside the draw method:

```
for( vector< shared_ptr<Follower> >::iterator it =
mFollowers.begin(); it != mFollowers.end(); ++it ){
    (*it)->draw();
}
```

The following is the result:



How it works...

The `Follower` class represents an agent that will follow the flow field. In the `Follower::update` method a new velocity vector is passed as a parameter. The `follower` object will interpolate its velocity into the passed value and use it to animate. The `Follower::update` method is also responsible for keeping each agent inside the window by warping its position whenever it is outside the window.

In step 11 we calculated the vector in the flow field that will influence the `Follower` object using its position.

Creating an image gallery in 3D

In this recipe we will learn how we can create an image gallery in 3D. The images will be loaded from a folder selected by the user and displayed in a three-dimensional circular fashion. Using the keyboard, the user will be able to change the selected image.

Getting ready

When starting the application you will be asked to select a folder with images, so make sure you have one.

Also, in your code include the necessary files to use OpenGL drawing calls, textures, the timeline, and loading images.

```
#include "cinder/gl/gl.h"
#include "cinder/gl/Texture.h"
#include "cinder/Timeline.h"
#include "cinder/ImageIo.h"
```

Also, add the following useful `using` statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will display and animate images in 3D space. Perform the following steps to do so:

1. We will start by creating an `Image` class. Add the following code snippet before the main application class:

```
class Image{
public:
```

```

Image( gl::Texture texture, const Rectf& maxRect ){
    this->texture = texture;
    distance = 0.0f;
    angle = 0.0f;
    Vec2f size = Vec2f(texture.getWidth(), texture.getHeight());
    rect = Rectf(-size * 0.5f, size*0.5f).getCenteredFit(
        maxRect, true );
}

void draw(){
    gl::pushMatrices();
    glRotatef( angle, 0.0f, 1.0f, 0.0f );
    gl::translate( 0.0f, 0.0f, distance );
    gl::draw( texture, rect );
    gl::popMatrices();
}

gl::Texture texture;
float distance;
float angle;
Rectf rect;
}

```

2. In the main application's class we will declare the following members:

```

vector<shared_ptr<Image>> mImages;
int mSelectedImageIndex;
Anim<float> mRotationOffset;

```

3. In the setup method we will ask the user to select a folder and then try to create a texture from each file in the folder. If a texture is successfully created, we will use it to create an Image object and add it to mImages.

```

fs::path imageFolder = getFolderPath( "" );
if( imageFolder.empty() == false ){
    for( fs::directory_iterator it( imageFolder ); it !=
        fs::directory_iterator(); ++it ){
        const fs::path& file = it->path();
        gl::Texture texture;
        try {
            texture = loadImage( file );
        } catch ( ... ) { }
        if( texture ){
            Rectf maxRect( RectfmaxRect( Vec2f( -50.0f, -50.0f ),
                Vec2f( 50.0f, 50.0f ) );
            mImages.push_back( shared_ptr<Image>(
                new Image( texture, maxRect ) );

```

```

    }
  }
}

```

4. We need to iterate over `mImages` and define the angle and distance that each image will have from the center.

```

float angle = 0.0f;
float angleAdd = 360.0f / mImages.size();
float radius = 300.0f;
for( int i=0; i<mImages.size(); i++ ){
    mImages[i]->angle = angle;
    mImages[i]->distance = radius;
    angle += angleAdd;
}

```

5. Now we can initialize the remaining members.

```

mSelectedImageIndex = 0;
mRotationOffset = 0.0f;

```

6. In the draw method, we will start by clearing the window, setting the window's matrices to support 3D, and enabling reading and writing in the depth buffer:

```

gl::clear( Color( 0, 0, 0 ) );
gl::setMatricesWindowPersp( getWindowWidth(), getWindowHeight() );
gl::enableDepthRead();
gl::enableDepthWrite();

```

7. Next we will draw the images. Since all our images have been displayed around the origin, we must translate them to the center of the window. We will also rotate them around the y axis using the value in `mRotationOffset`. Everything will go in an `if` statement that will check if `mImages` contains any image, in case no image was generated during the setup.

8. Add the following code snippet inside the draw method:

```

if( mImages.size() ){
    Vec2f center = (Vec2f)getWindowCenter();
    gl::pushMatrices();
    gl::translate( center.x, center.y, 0.0f );
    glRotatef( mRotationOffset, 0.0f, 1.0f, 0.0f );
    for(vector<shared_ptr<Image> >::iterator it=mImages.begin();
        it != mImages.end(); ++it ){
        (*it)->draw();
    }
    gl::popMatrices();
}

```

9. Since the user will be able to switch images using the keyboard, we must declare the `keyUp` event handler.

```
void keyUp( KeyEvent event );
```

10. In the implementation of `keyUp` we will move the images on to the left or right-hand side depending on whether the left or right key was released.

If the selected image was changed, we animate `mRotationOffset` to the correspondent value, so that the correct image is now facing the user.

Add the following code snippet inside the `keyUp` method:

```
bool imageChanged = false;
if( event.getCode() == KeyEvent::KEY_LEFT ){
    mSelectedImageIndex--;
    if( mSelectedImageIndex < 0 ){
        mSelectedImageIndex = mImages.size()-1;
        mRotationOffset.value() += 360.0f;
    }
    imageChanged = true;
} else if( event.getCode() == KeyEvent::KEY_RIGHT ){
    mSelectedImageIndex++;
    if( mSelectedImageIndex > mImages.size()-1 ){
        mSelectedImageIndex = 0;
        mRotationOffset.value() -= 360.0f;
    }
    imageChanged = true;
}
if( imageChanged ){
    timeline().apply( &mRotationOffset,
        mImages[ mSelectedImageIndex ]->angle, 1.0f,
        EaseOutElastic() );
}
```


11. Build and run the application. You will be prompted to select a folder containing images that will then be displayed in a circular fashion. Press the left or right key on the keyboard to change the selected image.



How it works...

The `draw` method of the `Image` class rotates the coordinate system around the `y` axis and then translates the image drawing on the `z` axis. This will extrude the image from the center facing outwards on the given angle. It is an easy and convenient way of achieving the desired effect without dealing with coordinate transformations.

The `Image::rect` member is used to draw the texture and is calculated to fit inside the rectangle passed in the constructor.

When selecting the image to be displayed in front, the value of `mRotationOffset` will be the opposite of the image's angle, making it the image being drawn in front of the view.

In the `keyUp` event we check whether the left or right key was pressed and animate `mRotationOffset` to the desired value. We also take into account if the angle wraps around, as to avoid glitches in the animation.

Creating a spherical flow field with Perlin noise

In this recipe we will learn how to use Perlin noise with a spherical flow field and animate objects in an organic way around a sphere.

We will animate our objects using spherical coordinates and then transform them into Cartesian coordinates in order to draw them.

Getting ready

Add the necessary files to use Perlin noise and draw with OpenGL:

```
#include "cinder/gl/gl.h"
#include "cinder/Perlin.h"
#include "cinder/Rand.h"
```

Add the following useful using statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will create the `Follower` objects that move organically in a spherical flow field. Perform the following steps to do so:

1. We will start by creating a `Follower` class representing an object that will follow the spherical flow field.

Add the following code snippet before the application's class declaration:

```
class Follower{
public:
Follower(){
    theta = 0.0f;
    phi = 0.0f;
}
void moveTo( const Vec3f& target ){
    prevPos = pos;
    pos += ( target - pos ) * 0.1f;
}
void draw(){
    gl::drawSphere( pos, 10.0f, 20 );
}
```

```
    Vec3f vel = pos - prevPos;
    gl::drawLine( pos, pos + ( vel * 5.0f ) );
}
Vec3f pos, prevPos;
float phi, theta;
};
```

2. We will be using spherical to Cartesian coordinates, so declare the following method in the application's class:

```
Vec3f sphericalToCartesians(sphericalToCartesians( float radius,
float theta, float phi );
```

3. The implementation of this method is as follows:

```
float x = radius * sinf( theta ) * cosf( phi );
float y = radius * sinf( theta ) * sinf( phi );
float z = radius * cosf( theta );
return Vec3f( x, y, z );
```

4. Declare the following members in the application's class:

```
vector<shared_ptr< Follower > > mFollowers;
float mRadius;
float mCounter;
Perlin mPerlin;
```

5. In the `setup` method we will begin by initializing `mRadius` and `mCounter`:

```
mRadius = 200.0f;
mCounter = 0.0f;
```

6. Now let's create 100 followers and add them to `mFollowers`. We will also attribute random values to the `phi` and `theta` variables of the `Follower` objects and set their initial positions:

```
int numFollowers = 100;
for( int i=0; i<numFollowers; i++ ){
    shared_ptr<Follower> follower( new Follower() );
    follower->theta = randFloat( M_PI * 2.0f );
    follower->phi = randFloat( M_PI * 2.0f );
    follower->pos = sphericalToCartesian( mRadius,
        follower->theta, follower->phi );
    mFollowers.push_back( follower );
}
```

7. In the `update` method we will animate our objects. Let's start by incrementing `mCounter`.

```
mCounter += 0.01f;
```

8. Now we will iterate over all the objects in `mFollowers` and use Perlin noise, based on the follower's position, to calculate how much it should move on spherical coordinates. We will then calculate the correspondent Cartesian coordinates and move the object.

Add the following code snippet inside the `update` method:

```
float resolution = 0.01f;
for( int i=0; i<mFollowers.size(); i++ ){
    shared_ptr<Follower> follower = mFollowers[i];
    Vec3f pos = follower->pos;
    float thetaAdd = mPerlin.noise( pos.x * resolution,
        pos.y * resolution, mCounter ) * 0.1f;
    float phiAdd = mPerlin.noise( pos.y * resolution,
        pos.z * resolution, mCounter ) * 0.1f;
    follower->theta += thetaAdd;
    follower->phi += phiAdd;
    Vec3f targetPos = sphericalToCartesian( mRadius,
        follower->theta, follower->phi );
    follower->moveTo( targetPos );
}
```

9. Let's move to the `draw` method and begin by clearing the background, setting the windows matrices, and enabling reading and writing in the depth buffer.

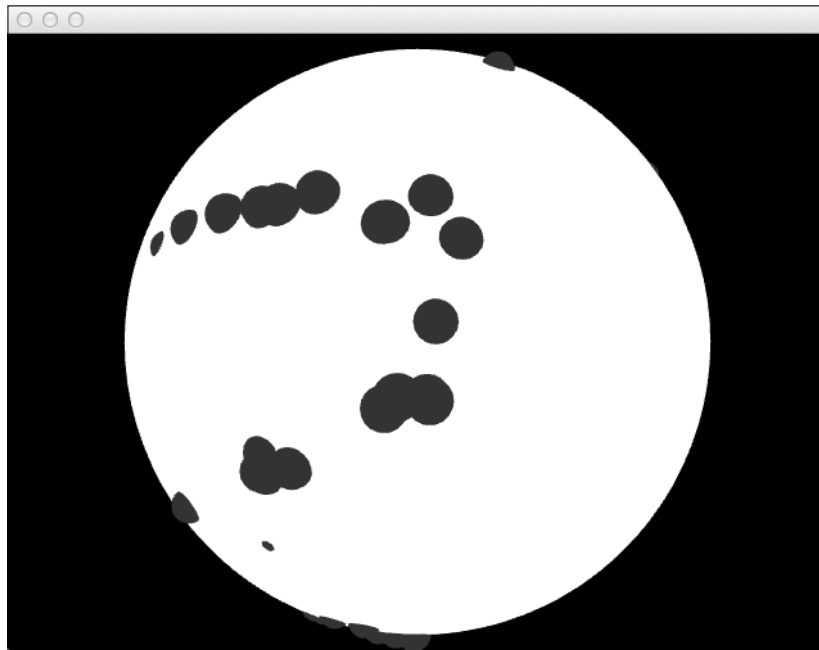
```
gl::clear( Color( 0, 0, 0 ) );
gl::setMatricesWindowPersp( getWindowWidth(), getWindowHeight() );
gl::enableDepthRead();
gl::enableDepthWrite();
```

10. Since the followers are moving around the origin, we will draw them translated to the origin using a dark gray color. We will also draw a white sphere to get a better understanding of the movement.

```
gl::pushMatrices();
Vec2f center = getWindowCenter();
gl::translate( center );
gl::color( Color( 0.2f, 0.2f, 0.2f ) );
for(vector<shared_ptr<Follower> >::iterator it =
    mFollowers.begin(); it != mFollowers.end(); ++it ){
    (*it)->draw();
}
gl::color( Color::white() );
gl::drawSphere( Vec3f(), mRadius, 100 );
gl::popMatrices();
```

How it works...

We use Perlin noise to calculate the change in the `theta` and `phi` members of the `Follower` objects. We use these, together with `mRadius`, to calculate the position of the objects using the standard spherical to Cartesian coordinate transformation. Since Perlin noise gives coherent values based on coordinates by using the current position of the `Follower` objects, we get the equivalent of a flow field. The `mCounter` variable is used to animate the flow field in the third dimension.



See also

- ▶ To learn more about the Cartesian coordinate system, please refer to http://en.wikipedia.org/wiki/Cartesian_coordinate_system
- ▶ To learn more about the spherical coordinate system, please refer to http://en.wikipedia.org/wiki/Spherical_coordinate_system
- ▶ To learn more about spherical to Cartesian coordinate transformations, please refer to http://en.wikipedia.org/wiki/List_of_common_coordinate_transformations#From_spherical_coordinate

10

Interacting with the User

In this chapter we will learn how to receive and respond to input from the user. The following recipes will be covered in the chapter:

- ▶ Creating an interactive object that responds to the mouse
- ▶ Adding mouse events to our interactive object
- ▶ Creating a slider
- ▶ Creating a responsive text box
- ▶ Dragging, scaling, and rotating objects using multi-touch

Introduction

In this chapter we will create graphical objects that react to the user using both mouse and touch interaction. We will learn how to create simple graphical interfaces that have their own events for greater flexibility.

Creating an interactive object that responds to the mouse

In this recipe, we will create an `InteractiveObject` class for making graphical objects that interact with the mouse cursor and executes the following actions:

Action	Description
Pressed	The user pressed the mouse button while over the object.
Pressed outside	The user pressed the mouse button while outside the object.
Released	The mouse button is released after being pressed over the object and is still over the object.
Released outside	The mouse button is released outside the object.
Rolled over	The cursor moves over the object.
Rolled out	The cursor moves out of the object.
Dragged	The cursor is dragged while being over the object and after having pressed the object.

For each of the previous actions, a virtual method will be called, and it would change the color of the object been drawn.

This object can be used as a base class to create interactive objects with more interesting graphics, such as textures.

Getting ready

Create and add the following files to your project:

- ▶ `InteractiveObject.h`
- ▶ `InteractiveObject.cpp`

In the source file with your application class, include the `InteractiveObject.h` file and add the following `using` statements:

```
#include "InteractiveObject.h"
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will create an `InteractiveObject` class and make it responsive to mouse events.

1. Move to the file `InteractiveObject.h` and add the `#pragma once` directive and include the following files:

```
#pragma once

#include "cinder/Rect.h"
#include "cinder/Color.h"
#include "cinder/app/MouseEvent.h"
#include "cinder/gl/gl.h"
#include "cinder/app/App.h"
```

2. Declare the class `InteractiveObject`:

```
class InteractiveObject{
public:
    InteractiveObject( const ci::Rectf& rect );
    virtual ~InteractiveObject();
    virtual void draw();
    virtual void pressed();
    virtual void pressedOutside();
    virtual void released();
    virtual void releasedOutside();
    virtual void rolledOver();
    virtual void rolledOut();
    virtual void dragged();
    void mouseDown( ci::app::MouseEvent& event );
    void mouseUp( ci::app::MouseEvent& event );
    void mouseDrag( ci::app::MouseEvent& event );
    void mouseMove( ci::app::MouseEvent& event );

    ci::Rectf rect;
    ci::Color pressedColor, idleColor, overColor, strokeColor;

protected:
    bool mPressed, mOver;
};
```

3. Move on to the `InteractiveObject.cpp` file, and let's begin by including the `InteractiveObject.h` file and adding the following `using` statements:

```
#include "InteractiveObject.h"

using namespace ci;
using namespace ci::app;
using namespace std;
```


4. Let's begin by implementing constructor and destructor.

```
InteractiveObject::InteractiveObject( const Rectf& rect ){
    this->rect = rect;
    pressedColor = Color( 1.0f, 0.0f, 0.0f );
    idleColor = Color( 0.7f, 0.7f, 0.7f );
    overColor = Color( 1.0f, 1.0f, 1.0f );
    strokeColor = Color( 0.0f, 0.0f, 0.0f );
    mPressed = false;
    mOver = false;
}

InteractiveObject::~InteractiveObject(){
}
```

5. In the `InteractiveObject::draw` method we will draw the rectangle using the appropriate colors:

```
void InteractiveObject::draw(){
    if( mPressed ){
        gl::color( pressedColor );
    } else if( mOver ){
        gl::color( overColor );
    } else {
        gl::color( idleColor );
    }
    gl::drawSolidRect( rect );
    gl::color( strokeColor );
    gl::drawStrokedRect( rect );
}
```

6. In the `pressed`, `released`, `rolledOver`, `rolledOut`, and `dragged` methods we will simply output to the console on which the action just happened:

```
void InteractiveObject::pressed(){
    console() << "pressed" << endl;
}

void InteractiveObject::pressedOutside(){
    console() << "pressed outside" << endl;
}

void InteractiveObject::released(){
    console() << "released" << endl;
}

void InteractiveObject::releasedOutside(){
```

```

        console() << "released outside" << endl;
    }

    void InteractiveObject::rolledOver(){
        console() << "rolled over" << endl;
    }

    void InteractiveObject::rolledOut(){
        console() << "rolled out" << endl;
    }

    void InteractiveObject::dragged(){
        console() << "dragged" << endl;
    }

```

7. In the mouse event handlers we will check if the cursor is inside the object and update the `mPressed` and `mOver` variables accordingly. Every time the action is detected, we will also call the correspondent method.

```

void InteractiveObject::mouseDown( MouseEvent& event ){
    if( rect.contains( event.getPos() ) ){
        mPressed = true;
        mOver = false;
        pressed();
    }else{
        pressedOutside();
    }
}

void InteractiveObject::mouseUp( MouseEvent& event ){
    if( rect.contains( event.getPos() ) ){
        if( mPressed ){
            mPressed = false;
            mOver = true;
            released();
        }
    } else {
        mPressed = false;
        mOver = false;
        releasedOutside();
    }
}

void InteractiveObject::mouseDrag( MouseEvent& event ){
    if( mPressed && rect.contains( event.getPos() ) ){

```

```
        mPressed = true;
        mOver = false;
        dragged();
    }
}

void InteractiveObject::mouseMove( MouseEvent& event ) {
    if( rect.contains( event.getPos() ) ) {
        if( mOver == false ) {
            mPressed = false;
            mOver = true;
            rolledOver();
        }
    } else {
        if( mOver ) {
            mPressed = false;
            mOver = false;
            rolledOut();
        }
    }
}
```

8. With our InteractiveObject class ready, let's move to our application's class source file. Let's begin by declaring an InteractiveObject object.

```
shared_ptr<InteractiveObject> mObject;
```

9. In the setup method we will initialize mObject.

```
Rectf rect( 100.0f, 100.0f, 300.0f, 300.0f );
mObject = shared_ptr<InteractiveObject>( new InteractiveObject(
rect ) );
```

10. We will need to declare the mouse event handlers.

```
void mouseDown( MouseEvent event );
void mouseUp( MouseEvent event );
void mouseDrag( MouseEvent event );
void mouseMove( MouseEvent event );
```

11. In the implementation of the previous methods we will simply call the corresponding method of mObject.

```
void MyApp::mouseDown( MouseEvent event ) {
    mObject->mouseDown( event );
}

void MyApp::mouseUp( MouseEvent event ) {
```

```

        mObject->mouseUp( event );
    }

    void MyApp::mouseDrag( MouseEvent event ){
        mObject->mouseDrag( event );
    }

    void MyApp::mouseMove( MouseEvent event ){
        mObject->mouseMove( event );
    }

```

12. In the implementation of the `draw` method, we will clear the background with black and call the `draw` method of `mObject`.

```

gl::clear( Color( 0, 0, 0 ) );
mObject->draw();

```

13. Now build and run the application. Use the mouse to interact with the object. Whenever you press, release, or roll over or out of the object, a message will be sent to the console indicating the behavior.

How it works...

The `InteractiveObject` class is to be used as a base class for interactive objects. The methods `pressed`, `released`, `rolledOver`, `rolledOut`, and `dragged` are specifically designed to be overridden.

The mouse handlers of `InteractiveObject` call the previous methods whenever an action is detected. By overriding the methods, it is possible to implement specific behavior.

The virtual destructor is declared so that extending classes can have their own destructor.

Adding mouse events to our interactive object

In this recipe, we will continue with the previous recipe, *Creating an interactive object that responds to the mouse* and add the mouse events to our `InteractiveObject` class so that other objects can register and receive notifications whenever a mouse event occurs.

Getting ready

Grab the code from the recipe *Creating an interactive object that responds to the mouse* and add it to your project, as we will continue on from what was made earlier.

How to do it...

We will make our `InteractiveObject` class and send its own events whenever it interacts with the cursor.

1. Let's create a class to use as an argument when sending events. Add the following code in the file `InteractiveObject.h` right before the `InteractiveObject` class declaration:

```
class InteractiveObject;
class InteractiveObjectEvent: public ci::app::Event{
public:
enum EventType{ Pressed, PressedOutside, Released,
ReleasedOutside, RolledOut, RolledOver, Dragged };
InteractiveObjectEvent( InteractiveObject *sender,
EventType type ){
this->sender = sender;
this->type = type;
}

InteractiveObject *sender;
EventType type;
};
```

2. In the `InteractiveObject` class, we will need to declare a member to manage the registered objects using the `ci::CallbackMgr` class. Declare the following code as a protected member:

```
ci::CallbackMgr< void(InteractiveObjectEvent) > mEvents;
```

3. Now we will need to add a method so that other objects can register to receive events. Since the method will use a template, we will declare and implement it in the `InteractiveObject.h` file. Add the following member method:

```
template< class T >
ci::CallbackId addListener( T* listener,
void (T::*callback)(InteractiveObjectEvent) ){
return mEvents.registerCb( std::bind1st(
std::mem_fun( callback ), listener ) );
}
```

4. Let's also create a method so that objects can unregister from receiving further events. Declare the following method:

```
void removeListener( ci::CallbackId callId );
```

5. Let's implement the `removeListener` method. Add the following code in the `InteractiveObject.cpp` file:

```
void InteractiveObject::removeListener( CallbackId callbackId ){
    mEvents.unregisterCb( callbackId );
}
```

6. Modify the methods `mouseDown`, `mouseUp`, `mouseDrag`, and `mouseMove` so that `mEvents` gets called whenever an event occurs. The implementation of these methods should be as follows:

```
void InteractiveObject::mouseDown( MouseEvent& event ){
    if( rect.contains( event.getPos() ) ){
        mPressed = true;
        mOver = false;
        pressed();
        mEvents.call( InteractiveObjectEvent( this,
            InteractiveObjectEvent::Pressed ) );
    } else {
        pressedOutside();
        mEvents.call( InteractiveObjectEvent( this,
            InteractiveObjectEvent::PressedOutside ) );
    }
}
```

```
void InteractiveObject::mouseUp( MouseEvent& event ){
    if( rect.contains( event.getPos() ) ){
        if( mPressed ){
            mPressed = false;
            mOver = true;
            released();
            mEvents.call( InteractiveObjectEvent( this,
                InteractiveObjectEvent::Released ) );
        }
    } else {
        mPressed = false;
        mOver = false;
        releasedOutside();
        mEvents.call( InteractiveObjectEvent( this,
            InteractiveObjectEvent::ReleasedOutside ) );
    }
}
```

```
void InteractiveObject::mouseDrag( MouseEvent& event ){
    if( mPressed && rect.contains( event.getPos() ) ){
        mPressed = true;
```

```
        mOver = false;

        dragged();
        mEvents.call( InteractiveObjectEvent( this,
        InteractiveObjectEvent::Dragged ) );
    }
}

void InteractiveObject::mouseMove( MouseEvent& event ){
    if( rect.contains( event.getPos() ) ){
        if( mOver == false ){
            mPressed = false;
            mOver = true;
            rolledOver();
            mEvents.call( InteractiveObjectEvent( this,
            InteractiveObjectEvent::RolledOver ) );
        }
    } else {
        if( mOver ){
            mPressed = false;
            mOver = false;
            rolledOut();
            mEvents.call( InteractiveObjectEvent( this,
            InteractiveObjectEvent::RolledOut ) );
        }
    }
}
```

7. With our `InteractiveObject` class ready, we need to register our application class to receive its events. In your application class declaration add the following method:
`void receivedEvent(InteractiveObjectEvent event);`
8. Let's implement the `receivedEvent` method. We will check what type of event has been received and print a message to the console.

```
void MyApp::receivedEvent( InteractiveObjectEvent event ){
    string text;
    switch( event.type ){
        case InteractiveObjectEvent::Pressed:
            text = "Pressed event";
            break;
        case InteractiveObjectEvent::PressedOutside:
            text = "Pressed outside event";
            break;
        case InteractiveObjectEvent::Released:
            text = "Released event";
```

```

break;
case InteractiveObjectEvent::ReleasedOutside:
text = "Released outside event";
break;
case InteractiveObjectEvent::RolledOver:
text = "RolledOver event";
break;
case InteractiveObjectEvent::RolledOut:
text = "RolledOut event";
break;
case InteractiveObjectEvent::Dragged:
text = "Dragged event";
break;
default:
text = "Unknown event";
}
console() << "Received " + text << endl;
}

```

9. All that is left is to register for the events. In the `setup` method add the following code after `mObject` has been initialized:

```

mObject->addListener( this, &InteractiveObjectApp::receivedEvent
);

```

10. Now build and run the application and use the mouse to interact with the rectangle on the window. Whenever a mouse event occurs on `mObject`, our method, `receivedEvent`, will be called.

How it works...

We are using the template class `ci::CallbackMgr` to manage our event listeners. This class takes a template with the signature of the methods that can be registered. In our previous code, we declared `mEvents` to be of type `ci::CallbackMgr<void(InteractiveObjectEvent)>`; it means that only methods that return `void` and receive `InteractiveObjectEvent` as a parameter can be registered.

The template method `registerEvent` will take an object pointer and method pointer. These are bound to `std::function` using `std::bind1st` and added to `mEvents`. The method will return `ci::CallbackId` with the identification of the listener. The `ci::CallbackId` can be used to unregister listeners.

There's more...

The `InteractiveObject` class is very useful for creating user interfaces. If we want to create a `Button` class using three textures (for displaying when pressed, over, and idle), we can do so as follows:

1. Include the `InteractiveObject.h` and `cinder/gl/texture.h` files:

```
#include "InteractiveObject.h"
#include "cinder/gl/Texture.h"
```

2. Declare the following class:

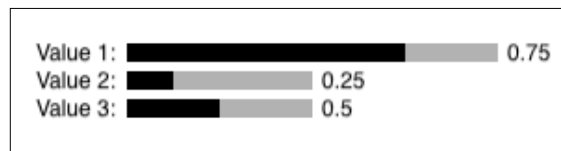
```
class Button: public InteractiveObject{
public:
    Button( const ci::Rectf& rect, ci::gl::Texture idleTex,
           ci::gl::Texture overTex, ci::gl::Texture pressTex)
        :InteractiveObject( rect )
    {
        mIdleTex = idleTex;
        mOverTex = overTex;
        mPressTex = pressTex;
    }

    virtual void draw(){
        if( mPressed ){
            ci::gl::draw( mPressTex, rect );
        } else if( mOver ){
            ci::gl::draw( mOverTex, rect );
        } else {
            ci::gl::draw( mPressTex, rect );
        }
    }

protected:
    ci::gl::Texture mIdleTex, mOverTex, mPressTex;
};
```

Creating a slider

In this recipe we will learn how to create a slider UI element by extending the `InteractiveObject` class mentioned in the *Creating an interactive object that responds to the mouse* recipe of this chapter.



Getting ready

Please refer to the *Creating an interactive object that responds to the mouse* recipe to find the `InteractiveObject` class headers and source code.

How to do it...

We will create a `Slider` class and show you how to use it.

1. Add a new header file named `Slider.h` to your project:

```
#pragma once

#include "cinder/gl/gl.h"
#include "cinder/Color.h"

#include "InteractiveObject.h"

using namespace std;
using namespace ci;
using namespace ci::app;

class Slider : public InteractiveObject {
public:
Slider( ) : InteractiveObject( Rectf(0,0, 100,10) ) {
    mValue = 0.f;
}
Vec2f    getPosition() { return rect.getUpperLeft(); }
void    setPosition(Vec2f position) { rect.offset(position); }
void    setPosition(float x, float y) { setPosition(Vec2f(x,y)); }
float    getWidth() { return getSize().x; }
float    getHeight() { return getSize().y; }
Vec2f    getSize() { return rect.getSize(); }
void    setSize(Vec2f size) {
    rect.x2 = rect.x1+size.x; rect.y2 = rect.y1+size.y;
}
void    setSize(float width, float height) {
    setSize(Vec2f(width,height));
}
}
```

```
virtual float getValue() { return mValue; }
virtual void setValue(float value) {
    mValue = ci::math<float>::clamp(value);
}

virtual void pressed() {
    InteractiveObject::pressed();
    dragged();
}

virtual void dragged() {
    InteractiveObject::dragged();
    Vec2i mousePos = AppNative::get()->getMousePos();
    setValue( (mousePos.x - rect.x1) / rect.getWidth() );
}

virtual void draw() {
    gl::color(Color::gray(0.7f));
    gl::drawSolidRect(rect);
    gl::color(Color::black());
    Rectf fillRect = Rectf(rect);
    fillRect.x2 = fillRect.x1 + fillRect.getWidth() * mValue;
    gl::drawSolidRect( fillRect );
}

protected:
float mValue;
};
```

2. Inside the source file of your main application class, include the previously created header file:

```
#include "Slider.h"
```

3. Add the new properties to your main class:

```
shared_ptr<Slider> mSlider1;
shared_ptr<Slider> mSlider2;
shared_ptr<Slider> mSlider3;
```

4. Inside the `setup` method do the initialization of the `slider` objects:

```
mSlider1 = shared_ptr<Slider>( new Slider() );
mSlider1->setPosition(70.f, 20.f);
mSlider1->setSize(200.f, 10.f);
mSlider1->setValue(0.75f);

mSlider2 = shared_ptr<Slider>( new Slider() );
mSlider2->setPosition(70.f, 35.f);
mSlider2->setValue(0.25f);

mSlider3 = shared_ptr<Slider>( new Slider() );
mSlider3->setPosition(70.f, 50.f);
mSlider3->setValue(0.5f);
```

5. Add the following code for drawing sliders inside your `draw` method:

```
gl::enableAlphaBlending();
gl::clear( Color::white() );
gl::setViewport( getWindowBounds() );
gl::setMatricesWindow( getWindowSize() );

mSlider1->draw();
gl::drawStringRight( "Value 1:", mSlider1->getPosition()+Vec2f(-
5.f, 3.f), Color::black() );
gl::drawString( toString( mSlider1->getValue() ), mSlider1-
>getPosition()+Vec2f( mSlider1->getWidth()+5.f, 3.f ),
Color::black() );

mSlider2->draw();
gl::drawStringRight( "Value 2:", mSlider2->getPosition()+Vec2f(-
5.f, 3.f), Color::black() );
gl::drawString( toString( mSlider2->getValue() ), mSlider2-
>getPosition()+Vec2f( mSlider2->getWidth()+5.f, 3.f ),
Color::black() );

mSlider3->draw();
gl::drawStringRight( "Value 3:", mSlider3->getPosition()+Vec2f(-
5.f, 3.f), Color::black() );
gl::drawString( toString( mSlider3->getValue() ), mSlider3-
>getPosition()+Vec2f( mSlider3->getWidth()+5.f, 3.f ),
Color::black() );
```

How it works...

We created the `Slider` class by inheriting and overriding the `InteractiveObject` methods and properties. In step 1, we extended it with methods for controlling the position and dimensions of the `slider` object. The methods `getValue` and `setValue` can be used to retrieve or set the actual state of `slider`, which can vary from 0 to 1.

In step 4, you can find the initialization of example sliders by setting the initial position, size, and value just after creating the `Slider` object. We are drawing example sliders along with captions and information about current state.

See also

- ▶ The recipe *Creating interactive object that responds to the mouse*.
- ▶ The recipe *Dragging scaling, and rotating objects using multi-touch*.

Creating a responsive text box

In this recipe we will learn how to create a text box that responds to the user's keystrokes. It will be active when pressed over by the mouse and inactive when the mouse is released outside the box.

Getting ready

Grab the following files from the recipe *Creating an interactive object that responds to the mouse* and add them to your project:

- ▶ `InteractiveObject.h`
- ▶ `InteractiveObject.cpp`

Create and add the following files to your project:

- ▶ `InteractiveTextBox.h`
- ▶ `InteractiveTextBox.cpp`

How to do it...

We will create an `InteractiveTextBox` class that inherits from `InteractiveObject` and adds text functionality.

1. Go to the file `InteractiveTextBox.h` and add the `#pragma once` macro and include the necessary files.

```
#pragma once

#include "InteractiveObject.h"
#include "cinder/Text.h"
#include "cinder/gl/Texture.h"
#include "cinder/app/KeyEvent.h"
#include "cinder/app/AppBasic.h"
```

2. Now declare the `InteractiveTextBox` class, making it a subclass of `InteractiveObject` with the following members and methods:

```
class InteractiveTextBox: public InteractiveObject{
public:
    InteractiveTextBox( const ci::Rectf& rect );

    virtual void draw();
    virtual void pressed();
    virtual void releasedOutside();

    void keyDown( ci::app::KeyEvent& event );
protected:
    ci::TextBox mTextBox;
    std::string mText;
    bool mActive;
    bool mFirstText;
};
```

3. Now go to `InteractiveTextBox.cpp` and include the `InteractiveTextBox.h` file and add the following using statements:

```
#include "InteractiveTextBox.h"

using namespace std;
using namespace ci;
using namespace ci::app;
```

4. Now let's implement the constructor by initializing the parent class and setting up the internal `ci::TextBox`.

```
InteractiveTextBox::InteractiveTextBox( const Rectf& rect ):
InteractiveObject( rect )
{
    mActive = false;
```

```
mText = "Write some text";
mTextBox.setText( mText );
mTextBox.setFont( Font( "Arial", 24 ) );
mTextBox.setPremultiplied( true );
mTextBox.setSize( Vec2i( rect.getWidth(), rect.getHeight() ) );
mTextBox.setBackgroundColor( Color::white() );
mTextBox.setColor( Color::black() );
mFirstText = true;
}
```

5. In the `InteractiveTextBox::draw` method we will set the background color of `mTextBox` depending if it is active or not. We will also render `mTextBox` into `ci::gl::Texture` and draw it.

```
void InteractiveTextBox::draw(){
    if( mActive ){
        mTextBox.setBackgroundColor( Color( 0.7f, 0.7f, 1.0f ) );
    } else {
        mTextBox.setBackgroundColor( Color::white() );
    }
    gl::color( Color::white() );
    gl::Texture texture = mTextBox.render();
    gl::draw( texture, rect );
}
```

6. Now let's implement the overridden methods `pressed` and `releasedOutside` to define the value of `mActive`.

```
void InteractiveTextBox::pressed(){
    mActive = true;
}

void InteractiveTextBox::releasedOutside(){
    mActive = false;
}
```

7. Finally, we need to implement the `keyPressed` method.

If `mActive` is false this method will simply return. Otherwise, we will remove the last letter of `mText` if the key released was the *Backspace* key, or, add the corresponding letter if any other key was pressed.

```
void InteractiveTextBox::keyDown( KeyEvent& event ){
    if( mActive == false ) return;
    if( mFirstText ){
        mText = "";
        mFirstText = false;
    }
}
```

```

    }
    if( event.getCode() == KeyEvent::KEY_BACKSPACE ){
        if( mText.size() > 0 ){
            mText = mText.substr( 0, mText.size()-1 );
        }
    } else {
        const char character = event.getChar();
        mText += string( &character, 1 );
    }
    mTextBox.setText( mText );
}

```

8. Now move to your application class source file and include the following file and the using statements:

```

#include "InteractiveTextBox.h"

using namespace ci;
using namespace ci::app;
using namespace std;

```

9. In your application class declare the following member:

```

shared_ptr<InteractiveTextBox> mTextBox;

```

10. Let's initialize mTextBox in the setup method:

```

Rectf rect( 100.0f, 100.0f, 300.0f, 200.0f );
mTextBox = shared_ptr<InteractiveTextBox>( new InteractiveTextBox(
rect ) );

```

11. In the draw method we will clear the background with black, enable AlphaBlending, and draw our mTextBox:

```

gl::enableAlphaBlending();
gl::clear( Color( 0, 0, 0 ) );
mTextBox->draw();

```

12. We now need to declare the following mouse event handlers:

```

void mouseDown( MouseEvent event );
void mouseUp( MouseEvent event );
void mouseDrag( MouseEvent event );
void mouseMove( MouseEvent event );

```

13. And implement them by calling the respective mouse event handler of mTextBox:

```

void MyApp::mouseDown( MouseEvent event ){
    mTextBox->mouseDown( event );
}

void MyApp::mouseUp( MouseEvent event ){

```



```
mTextBox->mouseUp( event );
}

void MyApp::mouseDrag( MouseEvent event ){
    mTextBox->mouseDrag( event );
}

void MyApp::mouseMove( MouseEvent event ){
    mTextBox->mouseMove( event );
}
```

14. Now we just need to do the same with the key released event handler. Start by declaring it:

```
void keyDown( KeyEvent event );
```

15. And in its implementation we will call the `keyUp` method of `mTextBox`.

```
void InteractiveObjectApp::keyDown( KeyEvent event ){
    mTextBox->keyDown( event );
}
```

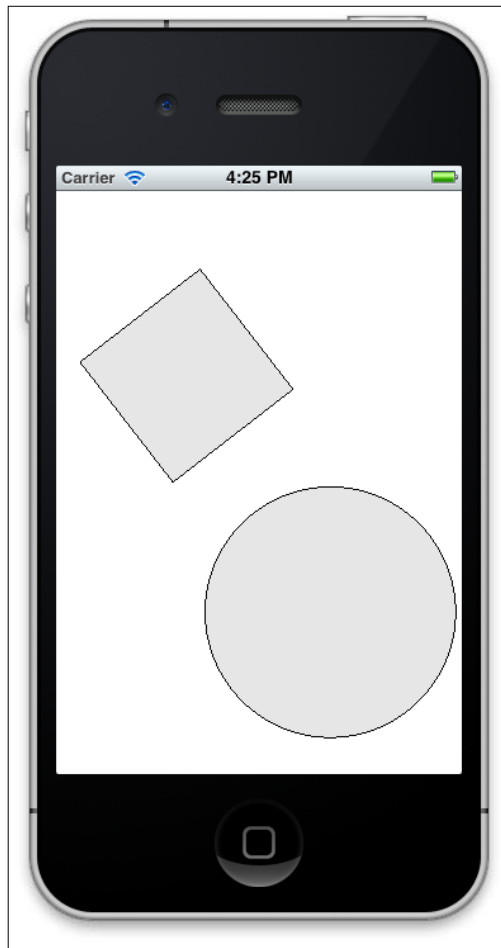
16. Now build and run the application. You will see a white textbox with the phrase **Write some text**. Press the text box and write some text. Click outside the text box to set the textbox as inactive.

How it works...

Internally, our `InteractiveTextBox` uses a `ci::TextBox` object. This class manages the text inside a box with a specified width and height. We take advantage of that and update the text according to the keys that the user presses.

Dragging, scaling, and rotating objects using multi-touch

In this recipe, we will learn how to create objects responsible to multi-touch gestures, such as dragging, scaling, or rotating by extending the `InteractiveObject` class mentioned in the *Creating an interactive object that responds to the mouse* recipe of this chapter. We are going to build an iOS application that uses iOS device multi-touch capabilities.



Getting ready

Please refer to the *Creating an interactive object that responds to the mouse* recipe to find the `InteractiveObject` class headers and source code and *Creating a project for an iOS touch application* recipe from Chapter 1.

How to do it...

We will create an iPhone application with sample objects that can be dragged, scaled, or rotated.

1. Add a new header file named `TouchInteractiveObject.h` to your project:

```
#pragma once

#include "cinder/app/AppNative.h"
#include "cinder/gl/gl.h"
#include "cinder/Color.h"

#include "InteractiveObject.h"

using namespace std;
using namespace ci;
using namespace ci::app;

class TouchInteractiveObject : public InteractiveObject {
public:
    TouchInteractiveObject( const Vec2f& position,
        const Vec2f& size );
    bool    touchesBegan(TouchEvent event);
    bool    touchesMoved(TouchEvent event);
    bool    touchesEnded(TouchEvent event);
    Vec2f   getPosition() { return position; }
    void    setPosition(Vec2f position) { this->position = position; }
    void    setPosition(float x, float y) { setPosition(Vec2f(x,y)); }
    float   getWidth() { return getSize().x; }
    float   getHeight() { return getSize().y; }
    Vec2f   getSize() { return rect.getSize(); }
    void    setSize(Vec2f size) {
        size.x = max(30.f,size.x);
        size.y = max(30.f,size.y);
        rect = Rectf(getPosition()-size*0.5f,getPosition()+size*0.5f);
    }
    void    setSize(float width, float height) {
        setSize(Vec2f(width,height));
    }
    float   getRotation() { return rotation; }
    void    setRotation( float rotation ) {
        this->rotation = rotation;
    }
}
```

```

virtual void draw();

protected:
Vec2f  position;
float  rotation;
bool   scaling;

unsigned int    dragTouchId;
unsigned int    scaleTouchId;
};

```

2. Add a new source file named `TouchInteractiveObject.cpp` to your project and include the previously created header file by adding the following code line:

```
#include "TouchInteractiveObject.h"
```

3. Implement the constructor of `TouchInteractiveObject`:

```

TouchInteractiveObject::TouchInteractiveObject(
    const Vec2f& position, const Vec2f& size )
    : InteractiveObject( Rectf() )
{
    scaling = false;
    rotation = 0.f;
    setPosition(position);
    setSize(size);
    AppNative::get()->registerTouchesBegan(this,
        &TouchInteractiveObject::touchesBegan);
    AppNative::get()->registerTouchesMoved(this,
        &TouchInteractiveObject::touchesMoved);
    AppNative::get()->registerTouchesEnded(this,
        &TouchInteractiveObject::touchesEnded);
}

```

4. Implement the handlers for touch events:

```

bool TouchInteractiveObject::touchesBegan(TouchEvent event)
{
    Vec2f bVec1 = getSize()*0.5f;
    Vec2f bVec2 = Vec2f(getWidth()*0.5f, -getHeight()*0.5f);
    bVec1.rotate((rotation) * (M_PI/180.f));
    bVec2.rotate((rotation) * (M_PI/180.f));
    Vec2f bVec;
    bVec.x = math<float>::max( abs(bVec1.x), abs(bVec2.x) );
    bVec.y = math<float>::max( abs(bVec1.y), abs(bVec2.y) );
    Area activeArea = Area(position-bVec, position+bVec);
    for (vector<TouchEvent::Touch>::const_iterator it

```

```
        = event.getTouches().begin();
        it != event.getTouches().end(); ++it) {
    if(activeArea.contains( it->getPos() )) {
        if(mPressed) {
            scaling = true;
            scaleTouchId = it->getId();
        } else {
            mPressed = true;
            dragTouchId = it->getId();
        }
    }
}
return false;
}

bool TouchInteractiveObject::touchesMoved(TouchEvent event)
{
    if(!mPressed) return false;
    const TouchEvent::Touch* dragTouch;
    const TouchEvent::Touch* scaleTouch;
    for (vector<TouchEvent::Touch>::const_iterator it
        = event.getTouches().begin();
        it != event.getTouches().end(); ++it) {
        if (scaling && scaleTouchId == it->getId()) {
            scaleTouch = &>(*it);
        }
        if(dragTouchId == it->getId()) {
            dragTouch = &>(*it);
        }
    }
    if(scaling) {
        Vec2f prevPos = (dragTouch->getPrevPos()
            + scaleTouch->getPrevPos()) * 0.5f;
        Vec2f curPos = (dragTouch->getPos()
            + scaleTouch->getPos()) * 0.5f;
        setPosition(getPosition() + curPos - prevPos);
        Vec2f prevVec = dragTouch->getPrevPos()
            - scaleTouch->getPrevPos();
        Vec2f curVec = dragTouch->getPos() - scaleTouch->getPos();

        float scaleFactor = (curVec.length() - prevVec.length())
            / prevVec.length();
        float sizeFactor = prevVec.length() / getSize().length();
        setSize(getSize() + getSize() * sizeFactor * scaleFactor);
    }
}
```

```

float angleDif = atan2(curVec.x, curVec.y)
  - atan2(prevVec.x, prevVec.y);
rotation += -angleDif * (180.f/M_PI);
} else {
  setPosition(getPosition() + dragTouch->getPos()
    - dragTouch->getPrevPos() );
}
return false;
}

bool TouchInteractiveObject::touchesEnded(TouchEvent event)
{
  if(!mPressed) return false;
  for (vector<TouchEvent::Touch>::const_iterator it
    = event.getTouches().begin();
    it != event.getTouches().end(); ++it) {
    if(dragTouchId == it->getId()) {
      mPressed = false;
      scaling = false;
    }
    if(scaleTouchId == it->getId()) {
      scaling = false;
    }
  }
  return false;
}

```

5. Now, implement the basic draw method for TouchInteractiveObjects:

```

void TouchInteractiveObject::draw() {
  Rectf locRect = Rectf(Vec2f::zero(), getSize());
  gl::pushMatrices();
  gl::translate(getPosition());
  gl::rotate(getRotation());
  gl::pushMatrices();
  gl::translate(-getSize()*0.5f);
  gl::color(Color::gray( mPressed ? 0.6f : 0.9f ));
  gl::drawSolidRect(locRect);
  gl::color(Color::black());
  gl::drawStrokedRect(locRect);
  gl::popMatrices();
  gl::popMatrices();
}

```

6. Here is the class, which inherits all the features of `TouchInteractiveObject`, but overrides the `draw` method and, in this case, we want our interactive object to be a circle. Add the following class definition to your main source file:

```
class Circle : public TouchInteractiveObject {
public:
    Circle(const Vec2f& position, const Vec2f& size)
        : TouchInteractiveObject(position, size) {}

    virtual void draw() {
        gl::color(Color::gray( mPressed ? 0.6f : 0.9f ));
        gl::drawSolidEllipse(getPosition(),
            getSize().x*0.5f, getSize().y*0.5f);
        gl::color(Color::black());
        gl::drawStrokedEllipse(getPosition(),
            getSize().x*0.5f, getSize().y*0.5f);
    }
};
```

7. Now take a look at your main application class file. Include the necessary header files:

```
#include "cinder/app/AppNative.h"
#include "cinder/Camera.h"
#include "cinder/Rand.h"

#include "TouchInteractiveObject.h"
```

8. Add the typedef declaration:

```
typedef shared_ptr<TouchInteractiveObject> tio_ptr;
```

9. Add members to your application class to handle the objects:

```
tio_ptr mObj1;
tio_ptr mCircle;
```

10. Inside the `setup` method initialize the objects:

```
mObj1 = tio_ptr( new TouchInteractiveObject( getRandPos(),
    Vec2f(100.f,100.f) ) );
mCircle = tio_ptr( new Circle( getRandPos(), Vec2f(100.f,100.f) ) );
```

11. The `draw` method is simple and looks as follows:

```
gl::setMatricesWindow( getWindowSize() );
gl::clear( Color::white() );
mObj1->draw();
mCircle->draw();
```

12. As you can see in the `setup` method we are using the function `getRandPos`, which returns a random position in screen boundaries with some margin:

```
Vec2f MainApp::getRandPos()
{
    return Vec2f( randFloat(30.f, getWindowWidth()-30.f),
                 randFloat(30.f, getWindowHeight()-30.f));
}
```

How it works...

We created the `TouchInteractiveObject` class by inheriting and overriding the `InteractiveObject` methods and properties. We also extended it with methods for controlling position and dimensions.

In step 3, we are initializing properties and registering callbacks for touch events. The next step is to implement these callbacks. On the `touchesBegan` event, we are checking if the object is touched by any of the new touches, but all the calculations of movements and gestures happen during `touchesMoved` event.

In step 6, you can see how simple it is to change the appearance and keep all the interactive capabilities of `TouchInteractiveObject` by overriding the `draw` method.

There is more...

You can notice an issue that you are dragging multiple objects while they are overlapping. To solve that problem, we will add a simple object activation manager.

1. Add a new class definition to your Cinder application:

```
class ObjectsManager {
public:
    ObjectsManager() { }

    void addObject( tio_ptr obj) {
        objects.push_front(obj);
    }

    void update() {
        bool rel = false;
        deque<tio_ptr>::const_iterator it;
        for(it = objects.begin(); it != objects.end(); ++it) {
            if( rel )
                (*it)->release();
            else if( (*it)->isActive() )
                rel = true;
        }
    }
};
```



```
    }  
}
```

```
protected:  
    deque<tio_ptr> objects;  
};
```

2. Add a new member to your application's main class:

```
shared_ptr<ObjectsManager> mObjMgr;
```

3. At the end of the `setup` method initialize `mObjMgr`, which is the object's manager, and add the previously initialized interactive objects:

```
mObjMgr = shared_ptr<ObjectsManager>( new ObjectsManager() );  
mObjMgr->addObject( mObj1 );  
mObjMgr->addObject( mCircle );
```

4. Add the `update` method to your main class as follows:

```
void MainApp::update()  
{  
    mObjMgr->update();  
}
```

5. Add two new methods to the `TouchInteractiveObject` class:

```
bool    isActive() { return mPressed; }  
void    release() { mPressed = false; }
```

11

Sensing and Tracking Input from the Camera

In this chapter, we will learn how to receive and process data from input devices such as a camera or a Microsoft Kinect sensor.

The following recipes will be covered:

- ▶ Capturing from the camera
- ▶ Tracking an object based on color
- ▶ Tracking motion using optical flow
- ▶ Object tracking
- ▶ Reading QR code
- ▶ Building UI navigation and gesture recognition with Kinect
- ▶ Building an augmented reality with Kinect

Capturing from the camera

In this recipe we will learn how to capture and display frames from a camera.

Getting ready

Include the necessary files to capture images from a camera and draw them to OpenGL textures:

```
#include "cinder/gl/gl.h"
#include "cinder/gl/Texture.h"
#include "cinder/Capture.h"
```

Also add the following using statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will now capture and draw frames from the camera.

1. Declare the following members in your application class:

```
Capture mCamera;
gl::Texture mTexture;
```

2. In the `setup` method we will initialize `mCamera`:

```
try{
    mCamera = Capture( 640, 480 );
    mCamera.start();
} catch( ... ){
    console() << "Could not initialize the capture" << endl;
```

3. In the `update` method, we will check if `mCamera` was successfully initialized. Also if there is any new frame available, copy the camera's image into `mTexture`:

```
if( mCamera ){
    if( mCamera.checkNewFrame() ){
        mTexture = gl::Texture( mCamera.getSurface() );
    }
}
```

4. In the `draw` method, we will simply clear the background, check if `mTexture` has been initialized, and draw it's image on the screen:

```
gl::clear( Color( 0, 0, 0 ) );
if( mTexture ){
    gl::draw( mTexture, getWindowBounds() );
}
```

How it works...

The `ci::Capture` is a class that wraps around Quicktime on Apple computers, AVFoundation on iOS platforms, and Directshow on Windows. Under the hood it uses these lower level frameworks to access and capture frames from a webcam.

Whenever a new frame is found, its pixels are copied into the `ci::Surface` method. In the previous code we check on every `update` method if there is a new frame by calling the `ci::Capture::checkNewFrame` method, and update our texture with its surface.

There's more...

It is also possible to get a list of available capture devices and choose which one you wish to start with.

To ask for a list of devices and print their information, we could write the following code:

```
vector<Capture::DeviceRef> devices = Capture::getDevices();
for( vector<Capture::DeviceRef>::iterator it = devices.begin();
    it != devices.end(); ++it ){
    Capture::DeviceRef device = *it;
    console() << "Found device:"
    << device->getName()
    << " with ID:" << device->getUniqueId() << endl;
}
```

To initialize `mCapture` using a specific device, you simply pass `ci::Capture::DeviceRef` as a third parameter in the constructor.

For example, if you wanted to initialize `mCapture` with the first device, you should write the following code:

```
vector<Capture::DeviceRef> devices = Capture::getDevices();
mCapture = Capture( 640, 480 devices[0] );
```

Tracking an object based on color

In this recipe we will show how to track objects with a specified color using the OpenCV library.

Getting ready

In this recipe we will use OpenCV, so please refer to the *Integrating with OpenCV* recipe from *Chapter 3, Using Image Processing Techniques*. We will also need *InterfaceGl* which is covered in the *Setting up a GUI for parameter tweaking* recipe from *Chapter 2, Preparing for Development*.

How to do it...

We will create an application that tracks an object with a selected color.

1. Include the necessary header files:

```
#include "cinder/gl/gl.h"
#include "cinder/gl/Texture.h"
#include "cinder/Surface.h"
#include "cinder/ImageIo.h"
#include "cinder/Capture.h"
#include "cinder/params/Params.h"
#include "CinderOpenCV.h"
```

2. Add members to store the original and processed frame:

```
Surface8u mImage;
```

3. Add members to store the tracked object's coordinates:

```
vector<cv::Point2f> mCenters;
vector<float> mRadius;
```

4. Add members to store the parameters that will be passed to the tracking algorithms:

```
double mApproxEps;
int mCannyThresh;
```

```
ColorA mPickedColor;
cv::Scalar mColorMin;
cv::Scalar mColorMax;
```

5. Add members to handle the capturing device and frame texture:

```
Capture mCapture;
gl::Texture mCaptureTex;
```

6. In the `setup` method we will set the window dimensions and initialize capturing device:

```
setWindowSize(640, 480);
```

```

try {
    mCapture = Capture( 640, 480 );
    mCapture.start();
}
catch( ... ) {
    console() <<"Failed to initialize capture"<<std::endl;
}

```

7. In the setup method we have to initialize variables and setup the GUI for a preview of the tracked color value:

```

mApproxEps = 1.0;
mCannyThresh = 200;

mPickedColor = Color8u(255, 0, 0);
setTrackingHSV();

// Setup the parameters
mParams = params::InterfaceGl( "Parameters", Vec2i( 200, 150 ) );
mParams.addParam( "Picked Color", &mPickedColor, "readonly=1" );

```

8. In the update method, check if there is any new frame to process and convert it to `cv::Mat`, which is necessary for further OpenCV operations:

```

if( mCapture&& mCapture.checkNewFrame() ) {
    mImage = mCapture.getSurface();
    mCaptureTex = gl::Texture( mImage );

    cv::Mat inputMat( toOcv( mImage ) );
    cv::resize(inputMat, inputMat, cv::Size(320, 240) );

    cv::Mat inputHSVMat, frameTresh;
    cv::cvtColor(inputMat, inputHSVMat, CV_BGR2HSV);

```

9. Process the captured frame:

```

cv::inRange(inputHSVMat, mColorMin, mColorMax, frameTresh);

cv::medianBlur(frameTresh, frameTresh, 7);

cv::Mat cannyMat;
cv::Canny(frameTresh, cannyMat, mCannyThresh, mCannyThresh*2.f,
3 );

```

```

vector< std::vector<cv::Point> > contours;
cv::findContours(cannyMat, contours, CV_RETR_LIST,
    CV_CHAIN_APPROX_SIMPLE);
mCenters = vector<cv::Point2f>(contours.size());
mRadius = vector<float>(contours.size());
for( int i = 0; i < contours.size(); i++ ) {
    std::vector<cv::Point> approxCurve;
    cv::approxPolyDP(contours[i], approxCurve,
        mApproxEps, true);
    cv::minEnclosingCircle(approxCurve, mCenters[i],
        mRadius[i]);
}

```

10. Close the if statement's body.

```

}

```

11. Implement the method `setTrackingHSV`, which sets color's values for tracking:

```

void MainApp::setTrackingHSV()
{
void MainApp::setTrackingHSV() {
    Color8u col = Color( mPickedColor );
    Vec3f colorHSV = col.get(CM_HSV);
    colorHSV.x *= 179;
    colorHSV.y *= 255;
    colorHSV.z *= 255;
    mColorMin = cv::Scalar(colorHSV.x-5, colorHSV.y -50,
        colorHSV.z-50);
    mColorMax = cv::Scalar(colorHSV.x+5, 255, 255);
}
}

```

12. Implement the `mouseDown` event handler:

```

void MainApp::mouseDown(MouseEvent event) {
    if( mImage&&Image.getBounds().contains( event.getPos() ) ) {
        mPickedColor = mImage.getPixel( event.getPos() );
        setTrackingHSV();
    }
}
}

```

13. Implement the `draw` method as follows:

```

void MainApp::draw()
{
    gl::clear( Color( 0.1f, 0.1f, 0.1f ) );
    gl::color(Color::white());
    if(mCaptureTex) {
        gl::draw(mCaptureTex);
        gl::color(Color::white());
    }
}

```

```
for( int i = 0; i < mCenters.size(); i++ )
{
    Vec2f center = fromOcv(mCenters[i])*2.f;
    gl::begin(GL_POINTS);
    gl::vertex( center );
    gl::end();
    gl::drawStrokedCircle(center, mRadius[i]*2.f );
}
}
params::InterfaceGl::draw();
}
```

How it works...

By preparing the captured frame for processing we are converting it into a **hue, saturation, and value (HSV)** color space description method, which will be very useful in this case. Those are the properties describing the color in the HSV color space in a more intuitive way for color tracking. We can set a fixed hue value for detection, while saturation and value can vary with in a specified range. This can eliminate a noise caused by constantly changing light in the camera view. Take a look at the first step of the frame image processing; we are using the `cv::inRange` function to get a mask of pixels that fits our tracking color range. The range of the tracking colors is calculated from the color value picked by clicking inside the window, which is implemented inside the `mouseDown` handler and the `setTrackingHSV` method.

As you can see inside `setTrackingHSV`, we are calculating `mColorMin` and `mColorMax` by simply widening the range. You may have to adjust these calculations depending on your camera noise and lighting conditions.

See also

- ▶ HSV on Wikipedia: http://en.wikipedia.org/wiki/HSL_and_HSV
- ▶ The OpenCV documentation: <http://opencv.willowgarage.com/documentation/cpp/>

Tracking motion using optical flow

In this recipe we will learn how to track motion in the images produced from a webcam using OpenCV using the popular Lucas Kanade optical flow algorithm.

Getting ready

We will need to use OpenCV in this recipe, so please refer to the *Integrating with OpenCV* recipe from *Chapter 3, Using Image Processing Techniques* and add OpenCV and its CinderBlock to your project. Include the following files to your source file:

```
#include "cinder/Capture.h"
#include "cinder/gl/Texture.h"
#include "CinderOpenCV.h"
```

Add the following using statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will read frames from the camera and track motion.

1. Declare the `ci::gl::Texture` and `ci::Capture` objects to display and capture from a camera. Also, declare a `cv::Mat` object as the previous frame, two `std::vector<cv::Point2f>` objects to store the current and previous features, and a `std::vector<uint8_t>` object to store the status of each feature:

```
gl::Texture mTexture;
Capture mCamera;
cv::Mat mPreviousFrame;
vector< cv::Point2f > mPreviousFeatures, mFeatures;
vector< uint8_t > mFeatureStatuses;
```

2. In the `setup` method we will initialize `mCamera`:

```
try{
    mCamera = Capture( 640, 480 );
    mCamera.start();
} catch( ... ){
    console() << "unable to initialize device" << endl;
}
```

3. In the `update` method we need to check if `mCamera` has been correctly initialized and whether it has a new frame available:

```
if( mCamera ){
    if( mCamera.checkNewFrame() ){
```

4. After those `if` statements we will get a reference to `ci::Surface` of `mCamera` and then copy it to our `mTexture` for drawing:

```
Surface surface = mCamera.getSurface();
mTexture = gl::Texture( surface );
```

5. Now let's create a `cv::Mat` with the current camera frame. We will also check if `mPreviousFrame` contains any initialized data, calculate the good features to track, and calculate their motion from the previous camera frame to the current frame:

```
cv::Mat frame( toOcv( Channel( surface ) ) );
if( mPreviousFrame.data != NULL ){
    cv::goodFeaturesToTrack( frame, mFeatures, 300,
0.005f, 3.0f );
    vector<float> errors;
    mPreviousFeatures = mFeatures;
    cv::calcOpticalFlowPyrLK( mPreviousFrame, frame,
mPreviousFeatures, mFeatures, mFeatureStatuses, errors );
}
```

6. Now we just need to copy the frame to `mPreviousFrame` and close the initial `if` statements:

```
mPreviousFrame = frame;
}
}
```

7. In the `draw` method we will begin by clearing the background with black and drawing `mTexture`:

```
gl::clear( Color( 0, 0, 0 ) );
if( mTexture ){
    gl::color( Color::white() );
    gl::draw( mTexture, getWindowBounds() );
}
```

8. Next, we will draw red lines on the features we have tracked, using `mFeatureStatus` to draw the features that have been matched:

```
glColor4f( 1.0f, 0.0f, 0.0f, 1.0f );
for( int i=0; i<mFeatures.size(); i++ ){
    if( (bool)mFeatureStatuses[i] == false ) continue;
    gl::drawSolidCircle( fromOcv( mFeatures[i] ), 5.0f );
}
```

9. Finally, we will draw a line between the previous features and the current ones, also using `mFeatureStatus` to draw one of the features that has been matched:

```
for( int i=0; i<mFeatures.size(); i++ ){
    if( (bool)mFeatureStatuses[i] == false ) continue;
    Vec2f pt1 = fromOcv( mFeatures[i] );
    Vec2f pt2 = fromOcv( mPreviousFeatures[i] );
    gl::drawLine( pt1, pt2 );
}
```

In the following image, the red dots represent good features to track:



How it works...

The optical flow algorithm will make an estimation of how much the tracked point has moved from one frame to the other.

There's more...

In this recipe we are using the `cv::goodFeaturesToTrack` object to calculate which features are optimal for tracking, but it is also possible to manually choose which points we wish to track. All we have to do is populate `mFeatures` manually with whatever points we wish to track and pass it to the `cv::calcOpticalFlowPyrLK` object

Object tracking

In this recipe, we will learn how to track specific planar objects in our webcam using OpenCV and its corresponding CinderBlock.

Getting ready

You will need an image depiction of the physical object you wish to track in the camera. For this recipe place that image in the `assets` folder and name it `object.jpg`.

We will use the OpenCV CinderBlock in this recipe, so please refer to the *Integrating with OpenCV* recipe from *Chapter 3, Using Image Processing Techniques* and add OpenCV and its CinderBlock to your project.

If you are using a Mac, you will need to compile the OpenCV static libraries yourself, because the OpenCV CinderBlock is missing some needed libraries on OSX (it will work fine on Windows). You can download the correct version from the following link: <http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.3/>.

You will need to compile the static libraries yourself using the provided CMake files. Once your libraries are correctly added to your project, include the following files:

```
#include "cinder/Capture.h"
#include "cinder/gl/Texture.h"
#include "cinder/ImageIo.h"
```

Add the following `using` statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

How to do it...

We will track an object in the camera frames based on an image depicting the object

1. Let's begin by creating a `struct` method to store the necessary objects for feature tracking and matching. Add the following code before your application class declaration:

```
struct DetectionInfo{
    vector<cv::Point2f> goodPoints;
    vector<cv::KeyPoint> keyPoints;
    cv::Mat image, descriptor;
    gl::Texture texture;
};
```

2. In your class declaration add the following member objects:

```
DetectionInfo mObjectInfo, mCameraInfo;
cv::Mat mHomography;
cv::SurfFeatureDetector mFeatureDetector;
cv::SurfDescriptorExtractor mDescriptorExtractor;
cv::FlannBasedMatcher mMatcher;
vector<cv::Point2f> mCorners;
```

3. In the `setup` method let's start by initializing the camera:

```
try{
    mCamera = Capture( 640, 480 );
    mCamera.start();
} catch( ... ){
    console() << "could not initialize capture" << endl;
}
```

4. Let's resize `mCorners`, load our object image, and calculate its image, `keyPoints`, texture, and descriptor:

```
mCorners.resize( 4 );
Surface objectSurface = loadImage( loadAsset( "object.jpg" )
);
mObjectInfo.texture = gl::Texture( objectSurface );
mObjectInfo.image = toOcv( Channel( objectSurface ) );
mFeatureDetector.detect( mObjectInfo.image, mObjectInfo.
keyPoints );
mDescriptorExtractor.compute( mObjectInfo.image, mObjectInfo.
keyPoints, mObjectInfo.descriptor );
```

5. In the update method, we will check if `mCamera` has been initialized and whether we have a new frame to process:

```
if( mCamera ){
    if( mCamera.checkNewFrame() ){
```

6. Now let's get the surface of `mCamera` and initialize texture and image objects of `mCameraInfo`. We will create a `ci::Channel` object from `cameraSurface` that converts color surfaces to gray channel surfaces:

```
Surface cameraSurface = mCamera.getSurface();
mCameraInfo.texture = gl::Texture( cameraSurface );
mCameraInfo.image = toOcv( Channel( cameraSurface ) );
```

7. Let's calculate features and descriptor values of `mCameraInfo`:

```
mFeatureDetector.detect( mCameraInfo.image, mCameraInfo.
keyPoints);
mDescriptorExtractor.compute( mCameraInfo.image, mCameraInfo.
keyPoints, mCameraInfo.descriptor );
```

8. Now let's use `mMatcher` to calculate the matches between `mObjectInfo` and `mCameraInfo`:

```
vector<cv::DMatch> matches;
mMatcher.match( mObjectInfo.descriptor, mCameraInfo.
descriptor, matches );
```

9. To perform a test to check for false matches, we will calculate the minimum distance between matches:

```
double minDist = 640.0;
for( int i=0; i<mObjectInfo.descriptor.rows; i++ ){
    double dist = matches[i].distance;
    if( dist < minDist ){
        minDist = dist;
    }
}
```

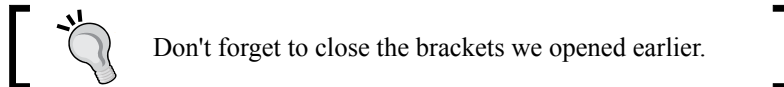
10. Now we will add all the points whose distance is less than `minDist*3.0` to `mObjectInfo.goodPoints.clear()`:

```
mCameraInfo.goodPoints.clear();
for( vector<cv::DMatch>::iterator it = matches.begin();
it != matches.end(); ++it ){
    if( it->distance < minDist*3.0 ){
        mObjectInfo.goodPoints.push_back(
mObjectInfo.keyPoints[ it->queryIdx ].pt );
        mCameraInfo.goodPoints.push_back(
mCameraInfo.keyPoints[ it->trainIdx ].pt );
    }
}
```

11. }With all our points calculated and matched, we need to calculate the homography between the points of `mObjectInfo` and `mCameraInfo`:

```
mHomography = cv::findHomography( mObjectInfo.goodPoints,
mCameraInfo.goodPoints, CV_RANSAC );
```

12. Let's create `vector<cv::Point2f>` with the corners of our object and perform a perspective transform to calculate the corners of our object in the camera image:



```
vector<cv::Point2f> objCorners( 4 );
objCorners[0] = cvPoint( 0.0f, 0.0f );
objCorners[1] = cvPoint( mObjectInfo.image.cols, 0.0f);
objCorners[2] = cvPoint( mObjectInfo.image.cols,
mObjectInfo.image.rows );
objCorners[3] = cvPoint( 0.0f, mObjectInfo.image.rows);
mCorners = vector< cv::Point2f >( 4 );
cv::perspectiveTransform( objCorners, mCorners,
mHomography );
}
}
```

13. Let's move to the `draw` method and begin by clearing the background and drawing the camera and object textures:

```
gl::clear( Color( 0, 0, 0 ) );

gl::color( Color::white() );
if( mCameraInfo.texture ){
    gl::draw( mCameraInfo.texture, getWindowBounds() );
}

if( mObjectInfo.texture ){
    gl::draw( mObjectInfo.texture );
}
}
```

14. Now let's iterate over `goodPoints` values in both `mObjectInfo` and `mCameraInfo` and draw them:

```
for( int i=0; i<mObjectInfo.goodPoints.size(); i++ ){
    gl::drawStrokedCircle( fromOcv( mObjectInfo.goodPoints[ i ] ),
5.0f );
    gl::drawStrokedCircle( fromOcv( mCameraInfo.goodPoints[ i ] ),
```

```

    5.0f );
    gl::drawLine( fromOcv( mObjectInfo.goodPoints[ i ] ),
                  fromOcv( mCameraInfo.goodPoints[ i ] ) );
}

```

15. Now let's iterate over `mCorners` and draw the corners of the found object:

```

gl::color( Color( 1.0f, 0.0f, 0.0f ) );
gl::begin( GL_LINE_LOOP );
    for( vector<cv::Point2f>::iterator it = mCorners.begin(); it
!= mCorners.end(); ++it ) {
        gl::vertex( it->x, it->y );
    }
gl::end();

```

16. Build and run the application. Grab the physical object you depicted in the `object.jpg` image and put it in front of the image. The program will try to track that object in the camera image and draw its corners in the image.

How it works...

We are using a **Speeded Up Robust Features (SURF)** feature detector and descriptor to identify features. In the step 4, we are calculating the features and descriptor. We use a `cv::SurfFeatureDetect` object or that calculates good features to track on our object. The `cv::SurfDescriptorExtractor` object then uses these features to create a description of our object. In the step 7, we do the same for the camera image.

In the step 8, we then use a **Fast Library for Approximate Nearest Neighbor (FLANN)** called `cv::FlannBasedMatcher`. This matcher takes the description from both the camera frame and our object, and calculates matches between them.

In steps 9 and 10, we use the minimum distance between matches to eliminate the possible false matches. The result is passed into `mObjectInfo.goodPoints` and `mCameraInfo.goodPoints`.

In the step 11, we calculate the homography between image and camera. A homography is a projection transformation from one space to another using projective geometry. We use it in the step 12 to apply a perspective transformation to `mCorners` to identify the object corners in the camera image.

There's more...

To learn more about what SURF is and how it works, please refer to the following web page: <http://en.wikipedia.org/wiki/SURF>.

To learn more about FLANN, please refer to the web page http://en.wikipedia.org/wiki/Nearest_neighbor_search.

To learn more about homography please refer to the following web page:

<http://en.wikipedia.org/wiki/Homography>.

Reading QR code

In this example we will use the ZXing library for QR code reading.

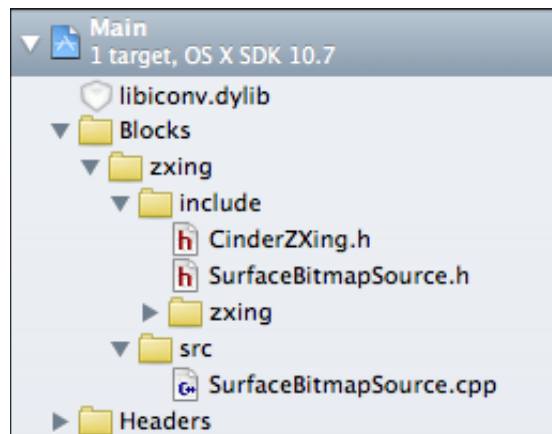
Getting ready

Please download the Cinder ZXing block from GitHub and unpack it to the `blocks` folder: <https://github.com/dawidgorny/Cinder-ZXing>

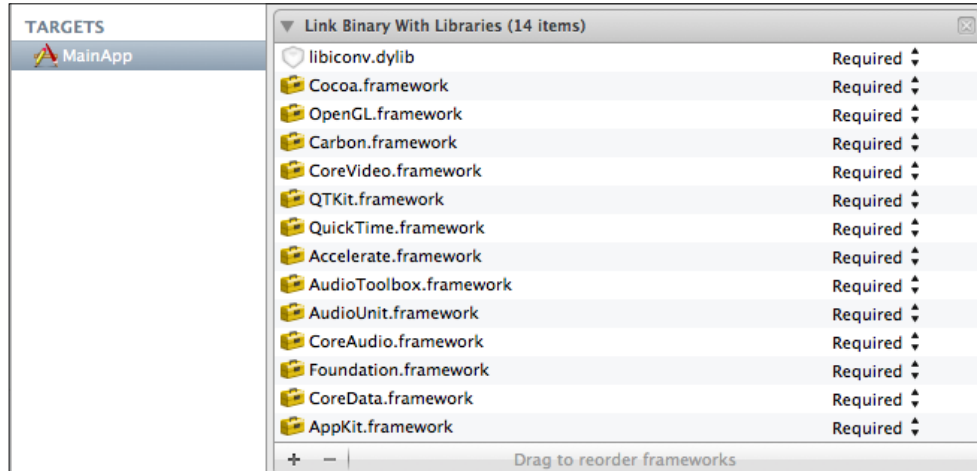
How to do it...

We will now create a QR code reader:

1. Add a header search path to the build settings of your project:
`$(CINDER_PATH)/blocks/zxing/include`
2. Add a path from the precompiled ZXing library to the build settings of your project: `$(CINDER_PATH)/blocks/zxing/lib/macosx/libzxing.a`. For a debug configuration, use `$(CINDER_PATH)/blocks/zxing/lib/macosx/libzxing_d.a`.
3. Add Cinder ZXing block files to your project structure as follows:



4. Add the `libiconv.dylib` library to the Link Binary With Libraries list:



5. Add the necessary header files:

```
#include "cinder/gl/Texture.h"
#include "cinder/Surface.h"
#include "cinder/Capture.h"

#include <zxing/qrcode/QRCodeReader.h>
#include <zxing/common/GlobalHistogramBinarizer.h>
#include <zxing/Exception.h>
#include <zxing/DecodeHints.h>

#include "CinderZXing.h"
```

6. Add the following members to your main application class:

```
Capture      mCapture;
Surface8u    mCaptureImg;
gl::Texture  mCaptureTex;
bool         mDetected;
string       mData;
```

7. Inside the `setup` method, set window dimensions and initialize capturing from camera:

```
setWindowSize(640, 480);

mDetected = false;

try {
```

```

        mCapture = Capture( 640, 480 );
        mCapture.start();
    }
    catch( ... ) {
        console() <<"Failed to initialize capture"<< std::endl;
    }

```

8. Implement the update function as follows:

```

if( mCapture && mCapture.checkNewFrame() ) {
    mCaptureImg = mCapture.getSurface();
    mCaptureTex = gl::Texture( mCaptureImg );

    mDetected = false;

    try {
        zxing::Ref<zxing::SurfaceBitmapSource> source(new zxing::S
urfaceBitmapSource(mCaptureImg));

        zxing::Ref<zxing::Binarizer> binarizer(NULL);
        binarizer = new zxing::GlobalHistogramBinarizer(source);

        zxing::Ref<zxing::BinaryBitmap> image(new zxing::BinaryBit
map(binarizer));
        zxing::qrcode::QRCodeReader reader;
        zxing::DecodeHints hints(zxing::DecodeHints::BARCODEFORM
AT_QR_CODE_HINT);

        zxing::Ref<zxing::Result> result( reader.decode(image,
hints) );

        console() <<"READ(" << result->count() <<" ) : " << result-
>getText()->getText() << endl;

        if( result->count() ) {
            mDetected = true;
            mData = result->getText()->getText();
        }

    } catch (zxing::Exception& e) {
        cerr <<"Error: " << e.what() << endl;
    }
}

```

9. Implement the `draw` function as follows:

```
gl::clear( Color( 0.1f, 0.1f, 0.1f ) );

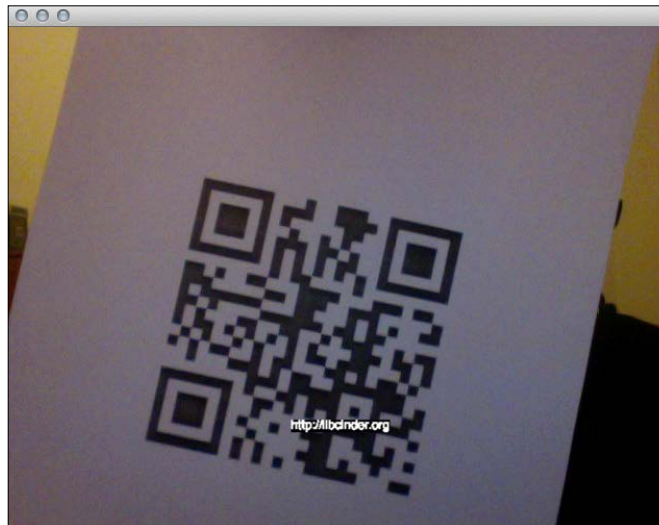
gl::color(Color::white());

if(mCaptureTex) {
    gl::draw(mCaptureTex);
}

if(mDetected) {
    Vec2f pos = Vec2f( getWindowWidth()*0.5f, getWindowHeight()-
100.f );
    gl::drawStringCentered(mData, pos);
}
```

How it works...

We are using regular ZXing library methods. The `SurfaceBitmapSource` class delivered by the Cinder ZXing block provides integration with Cinder `Surface` type objects. While the QR code is detected and read, the `mDetected` flag is set to `true` and the read data is stored in the `mData` member.

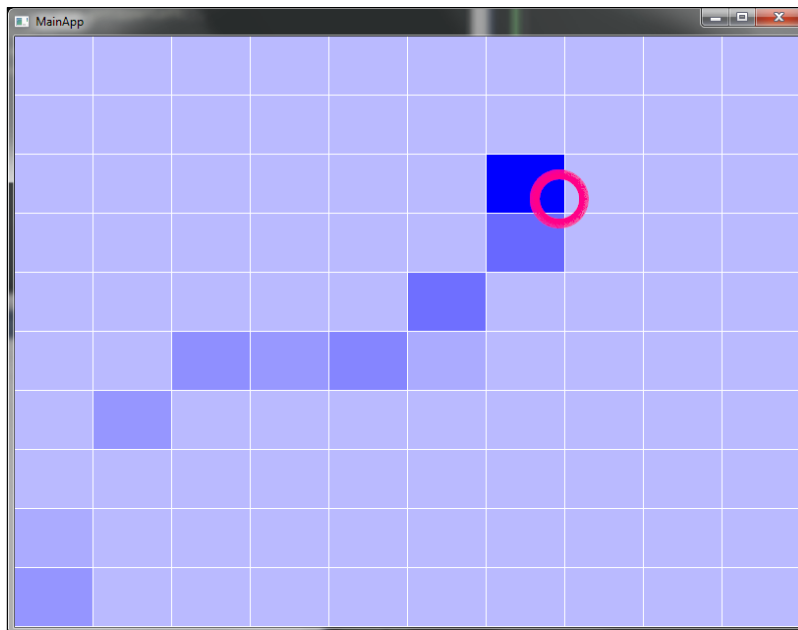


Building UI navigation and gesture recognition with Kinect

In this recipe we will create interactive GUI controlled with a Kinect sensor.



Since the **Kinect for Windows SDK** is available only for Windows, this recipe is written for Windows users only.



Getting ready

In this example we are using the `InteractiveObject` class that we covered in the *Creating an interactive object that responds to the mouse* recipe from *Chapter 10, Interacting with the User*.

Download and install the Kinect for Windows SDK from <http://www.microsoft.com/en-us/kinectforwindows/>.

Download the KinectSDK CinderBlock from GitHub at <https://github.com/BanTheRewind/Cinder-KinectSdk>, and unpack it to the `blocks` directory.

How to do it...

We will now create a Cinder application controlled with hand gestures.

1. Include the necessary header files:

```
#include "cinder/Rand.h"
#include "cinder/gl/Texture.h"
#include "cinder/Utilities.h"

#include "Kinect.h"
#include "InteractiveObject.h";
```

2. Add the Kinect SDK using the following statement:

```
using namespace KinectSdk;
```

3. Implement the class for a waving hand gesture recognition as follows:

```
class WaveHandGesture {
public:
    enum GestureCheckResult { Fail, Pausing, Succeed };

private:
    GestureCheckResult checkStateLeft( const Skeleton & skeleton ) {
        // hand above elbow
        if (skeleton.at(JointName::NUI_SKELETON_POSITION_HAND_RIGHT).y
> skeleton.at(JointName::NUI_SKELETON_POSITION_ELBOW_RIGHT).y)
        {
            // hand right of elbow
            if (skeleton.at(JointName::NUI_SKELETON_POSITION_HAND_
RIGHT).x > skeleton.at(JointName::NUI_SKELETON_POSITION_ELBOW_
RIGHT).x)
            {
                return Succeed;
            }
            return Pausing;
        }
        return Fail;
    }
    GestureCheckResult checkStateRight( const Skeleton & skeleton )
    {
        // hand above elbow
        if (skeleton.at(JointName::NUI_SKELETON_POSITION_HAND_RIGHT).y
> skeleton.at(JointName::NUI_SKELETON_POSITION_ELBOW_RIGHT).y)
        {
            // hand left of elbow
```

```
        if (skeleton.at(JointName::NUI_SKELETON_POSITION_HAND_
RIGHT).x < skeleton.at(JointName::NUI_SKELETON_POSITION_ELBOW_
RIGHT).x)
        {
            return Succeed;
        }
        return Pausing;
    }
    return Fail;
}

int currentPhase;

public:
    WaveHandGesture() {
        currentPhase = 0;
    }

    GestureCheckResult check( const Skeleton & skeleton )
    {
        GestureCheckResult res;
        switch(currentPhase) {
            case0: // start on left
            case2:
                res = checkStateLeft(skeleton);
                if( res == Succeed ) { currentPhase++; }
                elseif( res == Fail ) { currentPhase = 0; return Fail; }
                return Pausing;
                break;
            case1: // to the right
            case3:
                res = checkStateRight(skeleton);
                if( res == Succeed ) { currentPhase++; }
                elseif( res == Fail ) { currentPhase = 0; return Fail; }
                return Pausing;
                break;
            case4: // to the left
                res = checkStateLeft(skeleton);
                if( res == Succeed ) { currentPhase = 0; return Succeed; }
                elseif( res == Fail ) { currentPhase = 0; return Fail; }
                return Pausing;
                break;
        }
    }
}
```

```

        return Fail;
    }
};

```

4. Implement `NuiInteractiveObject` extending the `InteractiveObject` class:

```

class NuiInteractiveObject: public InteractiveObject {
public:
    NuiInteractiveObject(const Rectf & rect) :
    InteractiveObject(rect) {
        mHighlight = 0.0f;
    }

    void update(bool activated, const Vec2f & cursorPos) {
        if(activated && rect.contains(cursorPos)) {
            mHighlight += 0.08f;
        } else {
            mHighlight -= 0.005f;
        }
        mHighlight = math<float>::clamp(mHighlight);
    }

    virtualvoid draw() {
        gl::color(0.f, 0.f, 1.f, 0.3f+0.7f*mHighlight);
        gl::drawSolidRect(rect);
    }

    float mHighlight;
};

```

5. Implement the `NuiController` class that manages the active objects:

```

class NuiController {
public:
    NuiController() {}

    void registerObject(NuiInteractiveObject *object) {
        objects.push_back( object );
    }

    void unregisterObject(NuiInteractiveObject *object) {
        vector<NuiInteractiveObject*>::iterator it = find(objects.
begin(), objects.end(), object);
        objects.erase( it );
    }
}

```



```
void clear() { objects.clear(); }

void update(bool activated, const Vec2f & cursorPos) {
    vector<NuiInteractiveObject*>::iterator it;
    for(it = objects.begin(); it != objects.end(); ++it) {
        (*it)->update(activated, cursorPos);
    }
}

void draw() {
    vector<NuiInteractiveObject*>::iterator it;
    for(it = objects.begin(); it != objects.end(); ++it) {
        (*it)->draw();
    }
}

vector<NuiInteractiveObject*> objects;
};
```

6. Add the members to you main application class for handling Kinect devices and data:

```
KinectSdk::KinectRef      mKinect;
vector<KinectSdk::Skeleton> mSkeletons;
gl::Texture              mVideoTexture;
```

7. Add members to store the calculated cursor position:

```
Rectf  mPIZ;
Vec2f  mCursorPos;
```

8. Add the members that we will use for gesture recognition and user activation:

```
vector<WaveHandGesture*>  mGestureControllers;
bool  mUserActivated;
int   mActiveUser;
```

9. Add a member to handle NuiController:

```
NuiController* mNuiController;
```

10. Set window settings by implementing prepareSettings:

```
void MainApp::prepareSettings(Settings* settings)
{
    settings->setWindowSize(800, 600);
}
```

11. In the `setup` method, set the default values for members:

```
mPIZ = Rectf(0.f,0.f, 0.85f,0.5f);
mCursorPos = Vec2f::zero();

mUserActivated = false;
mActiveUser = 0;
```

12. In the `setup` method initialize Kinect and gesture recognition for 10 users:

```
mKinect = Kinect::create();
mKinect->enableDepth( false );
mKinect->enableVideo( false );
mKinect->start();

for(int i = 0; i <10; i++) {
    mGestureControllers.push_back( new WaveHandGesture() );
}
```

13. In the `setup` method, initialize the user interface consisting of objects of type `NuiInteractiveObject`:

```
mNuiController = new NuiController();

float cols = 10.f;
float rows = 10.f;

Rectf rect = Rectf(0.f,0.f, getWindowWidth()/cols - 1.f,
getWindowHeight()/rows - 1.f);

or(int ir = 0; ir < rows; ir++) {
    for(int ic = 0; ic < cols; ic++) {
        Vec2f offset = (rect.getSize()+Vec2f::one())
            * Vec2f(ic,ir);
        Rectf r = Rectf( offset, offset+rect.getSize() );
        mNuiController->registerObject(
            new NuiInteractiveObject® );
    }
}
```

14. In the `update` method, we are checking if the Kinect device is capturing, getting tracked skeletons, and iterating:

```
if ( mKinect->isCapturing() ) {
    if ( mKinect->checkNewSkeletons() ) {
        mSkeletons = mKinect->getSkeletons();
    }
    uint32_t i = 0;
    vector<Skeleton>::const_iterator skeletonIt;
    for (skeletonIt = mSkeletons.cbegin();
        skeletonIt != mSkeletons.cend(); ++skeletonIt, i++ ) {
```

15. Inside the loop, we are checking if the skeleton is complete and deactivating the cursor controls if it is not complete:

```
if(mUserActivated && i == mActiveUser
    && skeletonIt->size() !=
    JointName::NUI_SKELETON_POSITION_COUNT ) {
    mUserActivated = false;
}
```

16. Inside the loop check if the skeleton is valid. Notice we are only processing 10 skeletons. You can modify this number, but remember to provide sufficient number of gesture controllers in mGestureControllers:

```
if ( skeletonIt->size() == JointName::NUI_SKELETON_POSITION_COUNT
    && i <10 ) {
```

17. Inside the loop and the if statement, check for the completed activation gesture. While the skeleton is activated, we are calculating person interaction zone:

```
if( !mUserActivated || ( mUserActivated && i != mActiveUser ) ) {
    WaveHandGesture::GestureCheckResult res;
    res = mGestureControllers[i]->check( *skeletonIt );

    if( res == WaveHandGesture::Succeed && ( !mUserActivated || i
        != mActiveUser ) ) {
        mActiveUser = i;

        float armLen = 0;
        Vec3f handRight = skeletonIt->at(JointName::NUI_SKELETON_
            POSITION_HAND_RIGHT);
        Vec3f elbowRight = skeletonIt->at(JointName::NUI_SKELETON_
            POSITION_ELBOW_RIGHT);
        Vec3f shoulderRight = skeletonIt->at(JointName::NUI_
            SKELETON_POSITION_SHOULDER_RIGHT);

        armLen += handRight.distance( elbowRight );
        armLen += elbowRight.distance( shoulderRight );

        mPIZ.x2 = armLen;
        mPIZ.y2 = mPIZ.getWidth() / getWindowAspectRatio();

        mUserActivated = true;
    }
}
```

18. Inside the loop and the `if` statement, we are calculating cursor positions for active users:

```

if(mUserActivated && i == mActiveUser) {
    Vec3f handPos = skeletonIt->at(JointName::NUI_SKELETON_
    POSITION_HAND_RIGHT);

    Rectf piz = Rectf(mPIZ);
    piz.offset( skeletonIt->at(JointName::NUI_SKELETON_POSITION_
    SPINE).xy() );

    mCursorPos = handPos.xy() - piz.getUpperLeft();
    mCursorPos /= piz.getSize();
    mCursorPos.y = (1.f - mCursorPos.y);
    mCursorPos *= getWindowSize();
}

```

19. Close the opened `if` statements and the `for` loop:

```

    }
}

```

20. At the end of the update method, update the `NuiController` object:

```

mNuiController->update(mUserActivated, mCursorPos);

```

21. Implement the draw method as follows:

```

void MainApp::draw()
{
    // Clear window
    gl::setViewport( getWindowBounds() );
    gl::clear( Color::white() );
    gl::setMatricesWindow( getWindowSize() );
    gl::enableAlphaBlending();

    mNuiController->draw();

    if(mUserActivated) {
        gl::color(1.f,0.f,0.5f, 1.f);
        glLineWidth(10.f);
        gl::drawStrokedCircle(mCursorPos, 25.f);
    }
}

```

How it works...

The application is tracking users using Kinect SDK. Skeleton data of the active user are used to calculate the cursor position by following the guidelines provided by Microsoft with Kinect SDK documentation. Activation is invoked by a hand waving gesture.

This is an example of UI responsive to cursor controlled by a user's hand. Elements of the grid light up under the cursor and fade out on roll-out.

Building an augmented reality with Kinect

In this recipe we will learn how to combine both Kinect's depth and image frames to create augmented reality application.



Since Kinect for Windows SDK is available only for Windows, this recipe is written for Windows users only.

Getting ready

Download and install Kinect for Windows SDK from <http://www.microsoft.com/en-us/kinectforwindows/>.

Download KinectSDK CinderBlock from GitHub at <https://github.com/BanTheRewind/Cinder-KinectSdk>, and unpack it to the `blocks` directory.

In this example, we are using assets from one of the sample programs delivered with the Cinder package. Please copy the `ducky.mshducky.png`, `phong_vert.glsl`, and `phong_frag.glsl` files from `cinder_0.8.4_mac/samples/Picking3D/resources/` into your assets folder.

How to do it...

We will now create an augmented reality application using a sample 3D model.

1. Include the necessary header files:

```
#include "cinder/app/AppNative.h"
#include "cinder/gl/Texture.h"
#include "cinder/gl/GlslProg.h"
#include "cinder/TriMesh.h"
#include "cinder/ImageIo.h"
#include "cinder/MayaCamUI.h"
#include "cinder/params/Params.h"
```

- ```
#include "cinder/Utilities.h"

#include "Kinect.h"
```
2. Add the using statement of the Kinect SDK:

```
using namespace KinectSdk;
```
  3. Add the members to you main application class for handling Kinect device and data:

```
KinectSdk::KinectRef mKinect;
vector<KinectSdk::Skeleton> mSkeletons;
gl::Texture mVideoTexture;
```
  4. Add members to store 3D camera scene properties:

```
CameraPersp mCam;
Vec3f mCamEyePoint;
float mCamFov;
```
  5. Add members to store calibration settings:

```
Vec3f mPositionScale;
float mActivationDist;
```
  6. Add members that will store geometry, texture, and shader program for 3D object:

```
gl::Gls1Prog mShader;
gl::Texture mTexture;
TriMesh mMesh;
```
  7. Inside the setup method, set the window dimensions and initial values:

```
setWindowSize(800, 600);

mCamEyePoint = Vec3f(0.f, 0.f, 1.f);
mCamFov = 33.f;

mPositionScale = Vec3f(1.f, 1.f, -1.f);
mActivationDist = 0.6f;
```
  8. Inside the setup method load geometry, texture, and shader program for 3D object:

```
mMesh.read(loadFile(getAssetPath("ducky.msh")));

gl::Texture::Format format;
format.enableMipmapping(true);
ImageSourceRef img = loadImage(getAssetPath("ducky.png"));
if(img) mTexture = gl::Texture(img, format);

mShader = gl::Gls1Prog(loadFile(getAssetPath("phong_vert.gls1")),
loadFile(getAssetPath("phong_frag.gls1")));
```

9. Inside the `setup` method, initialize the Kinect device and start capturing:

```
mKinect = Kinect::create();
mKinect->enableDepth(false);
mKinect->start();
```

10. At the end of the `setup` method, create GUI for parameter tweaking:

```
mParams = params::InterfaceGl("parameters", Vec2i(200, 500));
mParams.addParam("Eye Point", &mCamEyePoint);
mParams.addParam("Camera FOV", &mCamFov);
mParams.addParam("Position Scale", &mPositionScale);
mParams.addParam("Activation Distance", &mActivationDist);
```

11. Implement the `update` method as follows:

```
void MainApp::update()
{
 mCam.setPerspective(mCamFov, getWindowAspectRatio(), 0.1, 10000
);
 mCam.setEyePoint(mCamEyePoint);
 mCam.setViewDirection(Vec3f(0.f,0.f, -1.f*mCamEyePoint.z));

 if (mKinect->isCapturing()) {
 if (mKinect->checkNewVideoFrame()) {
 mVideoTexture = gl::Texture(mKinect->getVideo());
 }
 if (mKinect->checkNewSkeletons()) {
 mSkeletons = mKinect->getSkeletons();
 }
 }
}
```

12. Implement the `drawObject` method that will draw our 3D model with the texture and shading applied:

```
void MainApp::drawObject()
{
 mTexture.bind();
 mShader.bind();
 mShader.uniform("tex0", 0);

 gl::color(Color::white());
 gl::pushModelView();
 gl::scale(0.05f,0.05f,0.05f);
 gl::rotate(Vec3f(0.f,-30.f,0.f));
 gl::draw(mMesh);
}
```

```

 gl::popModelView();

 mShader.unbind();
 mTexture.unbind();
}

```

13. Implement the draw method as follows:

```

void MainApp::draw()
{
 gl::setViewport(getWindowBounds());
 gl::clear(Colorf(0.1f, 0.1f, 0.1f));
 gl::setMatricesWindow(getWindowSize());

 if (mKinect->isCapturing() && mVideoTexture) {
 gl::color(ColorAf::white());
 gl::draw(mVideoTexture, getWindowBounds());
 draw3DScene();
 }

 params::InterfaceGl::draw();
}

```

14. The last thing that is missing is the draw3DScene method invoked inside the draw method. Implement the draw3DScene method as follows:

```

gl::enableDepthRead();
gl::enableDepthWrite();

Vec3f mLightDirection = Vec3f(0, 0, -1);
ColorA mColor = ColorA(0.25f, 0.5f, 1.0f, 1.0f);

gl::pushMatrices();
gl::setMatrices(mCam);

vector<KinectSdk::Skeleton>::const_iterator skelIt;
for (skelIt = mSkeletons.cbegin(); skelIt != mSkeletons.cend();
 ++skelIt) {

 if (skelIt->size() == JointName::NUI_SKELETON_POSITION_COUNT) {
 KinectSdk::Skeleton skel = *skelIt;

 Vec3f pos, dV;
 float armLen = 0;
 Vec3f handRight = skeletonIt->at(JointName::NUI_SKELETON_
 POSITION_HAND_RIGHT);
 }
}

```



```
 Vec3f elbowRight = skeletonIt->at(JointName::NUI_SKELETON_
POSITION_ELLOW_RIGHT);
 Vec3f shoulderRight = skeletonIt->at(JointName::NUI_
SKELETON_POSITION_SHOULDER_RIGHT);

 armLen += handRight.distance(elbowRight);
 armLen += elbowRight.distance(shoulderRight);

 pos = skel[JointName::NUI_SKELETON_POSITION_HAND_RIGHT];
 dV = pos - skel[JointName::NUI_SKELETON_POSITION_SHOULDER_
RIGHT];
if(dV.z < -armLen*mActivationDist) {
 gl::pushMatrices();
 gl::translate(pos*mPositionScale);
 drawObject();
 gl::popMatrices();
}
}

gl::popMatrices();

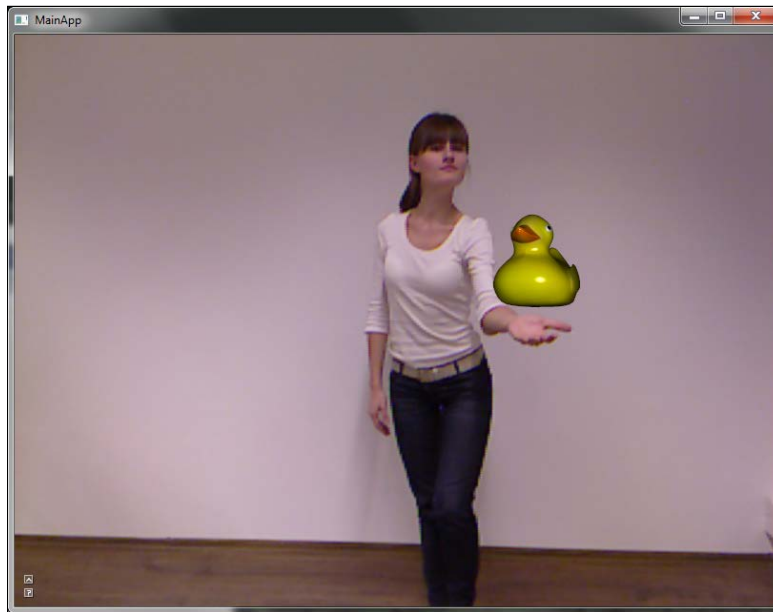
gl::enableDepthRead(false);
gl::enableDepthWrite(false);
```

15. Implement the shutdown method to stop capturing from Kinect on program termination:

```
void MainApp::shutdown()
{
 mKinect->stop();
}
```

## How it works...

The application is tracking users using the Kinect SDK. Skeleton data of the users are used to calculate the coordinates of the 3D duck model taken from one of the Cinder sample programs. The 3D model is rendered right above the right hand of the user when the user's hand is in front of the user. The activation distance is calculated using the `mActivationDist` member value.



To properly overlay 3D scene onto a video frame, you have to set the camera FOV according to the Kinect video camera. To do this, we are using the `Camera.FOV` property.



# 12

## Using Audio Input and Output

In this chapter, we will learn how to generate sounds using examples of ways to generate sounds driven by physics simulation. We will also present examples of visualizing sound with audio reactive animations.

The following recipes will cover:

- ▶ Generating a sine oscillator
- ▶ Generating sound with frequency modulation
- ▶ Adding a delay effect
- ▶ Generating sound upon the collision of objects
- ▶ Visualizing FFT
- ▶ Making sound-reactive particles

### Generating a sine oscillator

In this recipe, we will learn how to generatively create a sine wave oscillator by manipulating the sound card's **PCM (Pulse-code Modulation)** audio buffer. The frequency of the sine wave will be defined by the mouse's y coordinate.

We will also draw the sine wave for a visual representation.

## Getting ready

Include the following files:

```
#include "cinder/audio/Output.h"
#include "cinder/audio/Callback.h"
#include "cinder/Rand.h"
#include "cinder/CinderMath.h"
```

And add the following useful using statements:

```
using namespace ci;
using namespace ci::app;
using namespace std;
```

## How to do it...

We will create a sine wave oscillator using the following steps:

1. Declare the following member variables and the callback method:

```
void audioCallback(uint64_t inSampleOffset, uint32_t
ioSampleCount, audio::Buffer32f *buffer);
float mFrequency;
float mPhase, mPhaseAdd;
vector<float> mOutput;
```

2. In the `setup` module we will initialize the variables and create the audio callback using the following code:

```
mFrequency = 0.0f;
mPhase = 0.0f;
mPhaseAdd = 0.0f;
audio::Output::play(audio::createCallback(this,
&SineApp::audioCallback));
```

3. In the `update` module we will update `mFrequency` based on the mouse's `y` position. The mouse's position will be mapped and clamped to a frequency value between 0 and 5000:

```
float maxFrequency = 5000.0f;
float targetFrequency = (getMousePos().y / (float)
getWindowHeight()) * maxFrequency;
mFrequency = math<float>::clamp(targetFrequency, 0.0f,
maxFrequency);
```

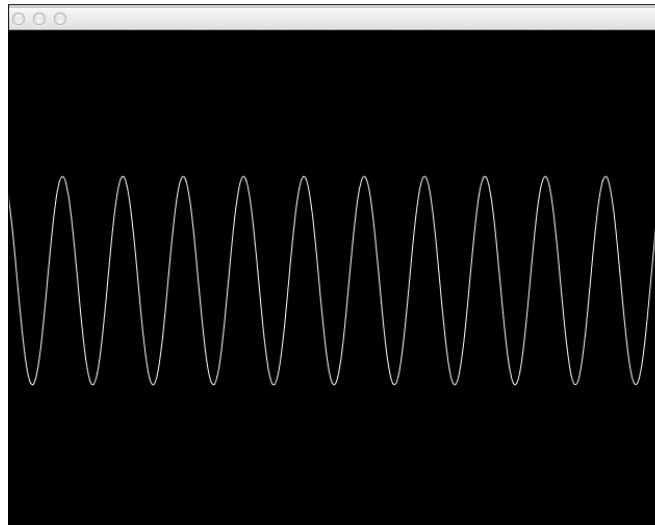
Let's implement the audio callback. We'll begin by resizing `mOutput` if necessary. Then we will calculate and interpolate `mPhaseAdd`, and then loop through all the values in the audio buffer and calculate their values based on the sine of `mPhase` and add `mPhaseAdd` to `mPhase`:

```
if(mOutput.size() != ioSampleCount){
 mOutput.resize(ioSampleCount);
}
int numChannels = buffer->mNumberChannels;
mPhaseAdd += ((mFrequency / 44100.0f) - mPhaseAdd) * 0.1f;
for(int i=0; i<ioSampleCount; i++){
 mPhase += mPhaseAdd;
 float output = math<float>::sin(mPhase * 2.0f * M_PI);
 for(int j=0; j<numChannels; j++){
 buffer->mData[i*numChannels + j] = output;
 }
 mOutput[i] = output;
}
```

4. Finally, we need to draw the sine wave. In the `draw` method, we will clear the background with black and draw a scaled up sine wave with a line strip using the values stored in `mOutput`:

```
gl::clear(Color(0, 0, 0));
if(mOutput.size() > 0){
 Vec2f scale;
 scale.x = (float)getWindowWidth() / (float)mOutput.size();
 scale.y = 100.0f;
 float centerY= getWindowHeight() / 2.0f;
 gl::begin(GL_LINE_STRIP);
 for(int i=0; i<mOutput.size(); i++){
 float x = (float)i * scale.x;
 float y = mOutput[i] * scale.y + centerY;
 gl::vertex(x, y);
 }
 gl::end();
}
```

5. Build and run the application. Move the mouse vertically to change the frequency. A line representing the generated sine wave is shown in the following screenshot:



### How it works...

We are manipulating the PCM buffer. PCM is a method to represent audio through values' samples at regular intervals. By accessing the PCM buffer, we can directly manipulate the audio signal that will be output by the sound card.

Every time the `audioCallback` method is called, we receive a sample of the PCM buffer, where we calculate the values to generate a continuous sine wave.

In the `update` module, we calculate the frequency by mapping the mouse's `y` position.

In the following line in the `audioCallback` implementation, we calculate how much `mPhase` has to increase based on a sample rate of 44100 to generate a wave with a frequency of `mFrequency`:

```
mPhaseAdd += ((mFrequency / 44100.0f) - mPhaseAdd) * 0.1f;
```

## Generating sound with frequency modulation

In this recipe, we will learn how to modulate a sine wave oscillator using another low frequency sine wave.

We will be basing this recipe on the previous recipe, where the  $y$  position of the mouse controlled the frequency of the sine wave; in this recipe, we will use the  $x$  position of the mouse to control the modulation frequency.

## Getting ready

We will be using the code from the previous recipe, *Generating a sine oscillator*.

## How to do it...

We will multiply the sine wave created in the previous recipe with another low frequency sine wave.

1. Add the following member variables:
 

```
float mModFrequency;
float mModPhase, mModPhaseAdd;
```
2. Add the following in the `setup` module to initialize the variables created previously:
 

```
mModFrequency = 0.0f;
mModPhase = 0.0f;
mModPhaseAdd = 0.0f;
```
3. In the `update` module, add the following code to calculate the modulation frequency based on the  $x$  position of the mouse cursor:
 

```
float maxModFrequency= 30.0f;
float targetModFrequency= (getMousePos().x / (float)
getWindowWidth()) * maxModFrequency;
mModFrequency = math<float>::clamp(targetModFrequency, 0.0f,
maxModFrequency);
```
4. We will need to calculate another sine wave using `mModFrequency`, `mModPhase`, and `mModPhaseAdd`, and use it to modulate our first sine wave.

The following is the implementation of `audioCallback`:

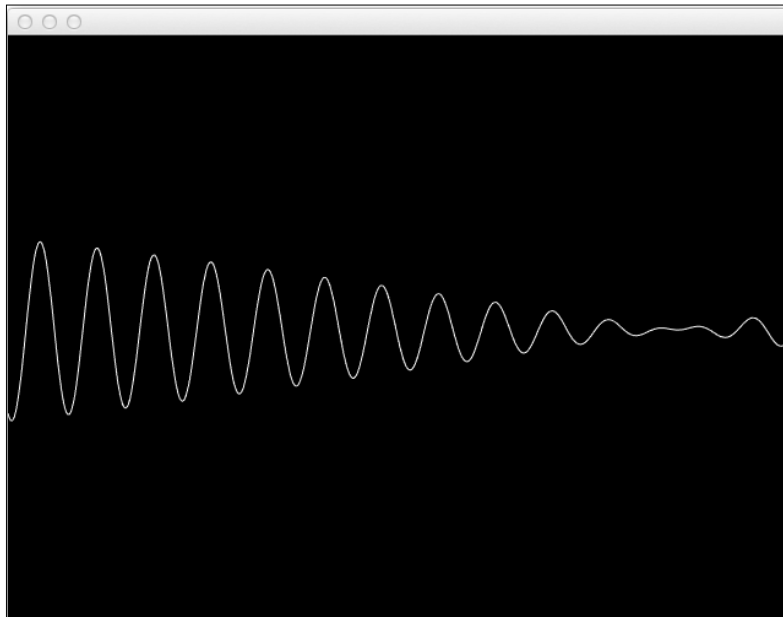
```
if(mOutput.size() != ioSampleCount){
 mOutput.resize(ioSampleCount);
}
mPhaseAdd += ((mFrequency / 44100.0f) - mPhaseAdd) * 0.1f;
mModPhaseAdd += ((mModFrequency / 44100.0f) - mModPhaseAdd)
 * 0.1f;
int numChannels= buffer->mNumberChannels;
for(int i=0; i<ioSampleCount; i++){
 mPhase += mPhaseAdd;
 mModPhase += mModPhaseAdd;
```



```
float output = math<float>::sin(mPhase * 2.0f * M_PI)
 * math<float>::sin(mModPhase * 2.0f * M_PI);
for(int j=0; j<numChannels; j++){
 buffer->mData[i*numChannels + j] = output;
}
mOutput[i] = output;
}
```

5. Build and run the application. Move the mouse cursor over the y axis to determine the frequency, and over the x axis to determine the modulation frequency.

We can see how the sine wave created changes in the previous recipe, in the amplitude as it is multiplied by another low frequency sine wave.



### How it works...

We calculate a second sine wave with a **low frequency oscillation (LFO)** and use it to modulate the first sine wave. To modulate the waves, we multiply them by each other.

## Adding a delay effect

In this recipe, we will learn how to add a delay effect to the frequency modulation audio generated in the previous recipe.

### Getting ready

We will use the source code from the previous recipe, *Generating sound with frequency modulation*.

### How to do it...

We will store our audio values and play them after an interval to achieve a delay effect using the following steps:

1. Add the following member variables:

```
int mDelay;
float mMix, mFeedback;
vector<float> mDelayLine;
int mDelayIndex;
int mDelaySize;
```

Let's initialize the variables created above and initialize our delay line with zeros.

Then add the following in the `setup` method:

```
mDelay = 200;
mMix = 0.2f;
mFeedback = 0.3f;
mDelaySize = mDelay * 44.1f;
for(int i=0; i<mDelaySize; i++){
 mDelayLine.push_back(0.0f);
}
```

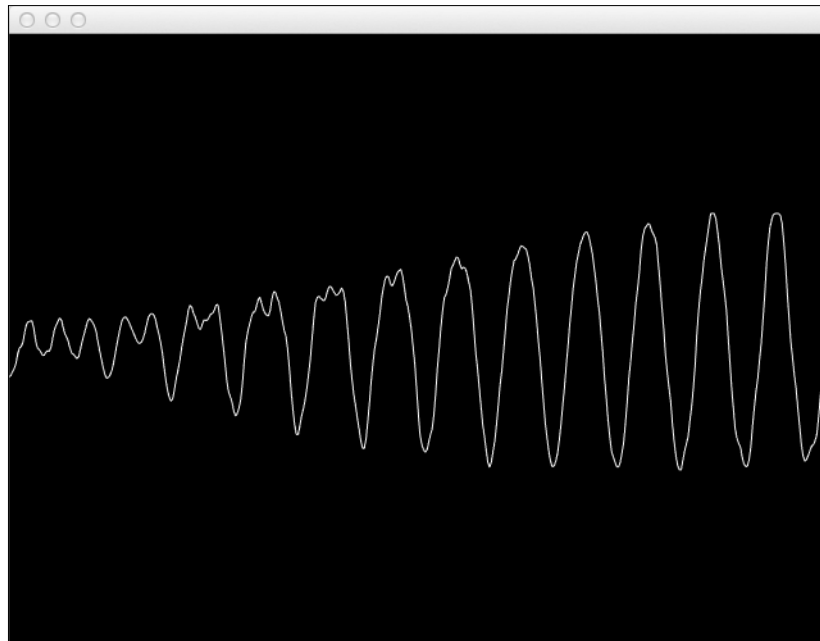
2. In the implementation of our `audioCallback` method, we will read back from the buffer the values that were generated in the frequency modulation and calculate the delay.

The final value is again passed into the buffer for output.

Add the following code in the `audioCallback` method:

```
for(int i=0; i<ioSampleCount; i++){
 float output = buffer->mData[i*numChannels];
 int readIndex= mDelayIndex - mDelaySize + 1;
 if(readIndex< 0) readIndex += mDelaySize;
 float delay = mDelayLine[readIndex * numChannels];
 mDelayLine[mDelayIndex] = output + delay * mFeedback;
 if(++mDelayIndex == mDelaySize){
 mDelayIndex = 0;
 }
 output = math<float>::clamp(output+mMix*delay, -1.0f, 1.0f);
 mOutput[i] = output;
 for(int j=0; j<numChannels; j++){
 buffer->mData[i*numChannels + j] = output;
 }
}
```

3. Build and run the application. By moving the mouse in the x axis, you control the oscillator frequency, and by moving the mouse in the y axis, you control the modulation frequency. The output will contain a delay effect as shown in the following screenshot:



## How it works...

A delay is an audio effect where an input is stored and then played back after a determined amount of time. We achieve this by creating a buffer the size of `mDelay` multiplied by the frequency rate. Each time `audioCallback` gets called, we read from the delay line and update the delay line with the current output value. We then add the delay value to the output and advance `mDelayIndex`.

## Generating sound upon the collision of objects

In this recipe, we will learn how to apply simple physics to object particles and generate sound upon the collision of two objects.

## Getting ready

In this example, we are using code described in the recipe *Generating a sine oscillator* in this chapter, so please refer to that recipe.

## How to do it...

We will create a Cinder application to illustrate the mechanism:

1. Include the following necessary header files:

```
#include "cinder/audio/Output.h"
#include "cinder/audio/Callback.h"
#include "cinder/Rand.h"
#include "cinder/CinderMath.h"
#include "ParticleSystem.h"
```

2. Add members to the application's main class for particle simulation:

```
ParticleSystem mParticleSystem;
Vec2f attrPosition;
float attrFactor;
float attrRadius;
```

3. Add members to the application's main class to make the particles interactive:

```
bool dragging;
Particle *dragParticle;
```

4. Add members for the generation of sound:

```
void audioCallback(uint64_t inSampleOffset, uint32_t
ioSampleCount, audio::Buffer32f *buffer);
float mSndFrequency;
```

```
float mPhase, mPhaseAdd;
vector<float> mOutput;
```

5. Initialize the particle system inside the setup method:

```
mRunning= true;
dragging = false;
attrPosition = getWindowCenter();
attrRadius = 75.f;
attrFactor = 0.02f;
int numParticle= 10;
for(int i=0; i<numParticle; i++){
 float x = Rand::randFloat(0.0f, getWindowWidth());
 float y = Rand::randFloat(0.0f, getWindowHeight());
 float radius = Rand::randFloat(2.f, 40.f);
 Rand::randomize();
 float mass = radius;
 float drag = 0.95f;
 Particle *particle = new Particle(Vec2f(x, y), radius,
 mass, drag);
 mParticleSystem.addParticle(particle);
}
```

6. Initialize the members to generate sound and register an audio callback inside the setup method:

```
mSndFrequency = 0.0f;
mPhase = 0.0f;
mPhaseAdd = 0.0f;
audio::Output::play(audio::createCallback(this,
 &MainApp::audioCallback));
```

7. Implement the `resize` method to update the attractor position whenever an application window will be resized:

```
void MainApp::resize(ResizeEvent event)
{
 attrPosition = getWindowCenter();
}
```

8. Implement the mouse events handlers for mouse interaction with particles:

```
void MainApp::mouseDown(MouseEvent event)
{
 dragging = false;
 std::vector<Particle*>::iterator it;
 for(it = mParticleSystem.particles.begin();
 it != mParticleSystem.particles.end(); ++it) {
```

```

 if((*it)->position.distance(event.getPos())
 < (*it)->radius) {
 dragging = true;
 dragParticle = (*it);
 }
}
}

```

```

void MainApp::mouseUp(MouseEvent event) {
 dragging = false;
}

```

9. Inside the update method, add the following code for sound frequency calculation:

```

float maxFrequency = 15000.0f;
float targetFrequency = (getMousePos().y / (float)
getWindowHeight()) * maxFrequency;
targetFrequency = mSndFrequency - 10000.f;
mSndFrequency = math<float>::clamp(targetFrequency, 0.0f,
maxFrequency);

```

10. Inside the update method, add the following code for particle movement calculation. At this point, we are detecting collisions and calculating the sound frequency:

```

std::vector<Particle*>::iterator it;
for(it = mParticleSystem.particles.begin();
 it != mParticleSystem.particles.end(); ++it) {
 std::vector<Particle*>::iterator it2;
 for(it2 = mParticleSystem.particles.begin();
 it2 != mParticleSystem.particles.end(); ++it2) {
 float d = (*it)->position.distance((*it2)->position);
 float d2 = (*it)->radius + (*it2)->radius;
 if(d > 0.f && d <= d2) {
 (*it)->forces += -1.1f * ((*it2)->position
 - (*it)->position);
 (*it2)->forces += -1.1f * ((*it)->position
 - (*it2)->position);
 mSndFrequency = 2000.f;
 mSndFrequency+= 10000.f
 * (1.f - ((*it)->radius / 40.f));
 mSndFrequency+= 10000.f
 * (1.f - ((*it2)->radius / 40.f));
 }
 }
}
Vec2f attrForce = attrPosition - (*it)->position;
attrForce *= attrFactor;
(*it)->forces += attrForce;

```

```
}
mSndFrequency = math<float>::clamp(mSndFrequency,
 0.0f, maxFrequency);maxFrequency);
```

11. Update position of dragging particle, if any, and update particle system:

```
if(dragging) {
 dragParticle->forces = Vec2f::zero();
 dragParticle->position = getMousePos();
}
```

```
mParticleSystem.update();
```

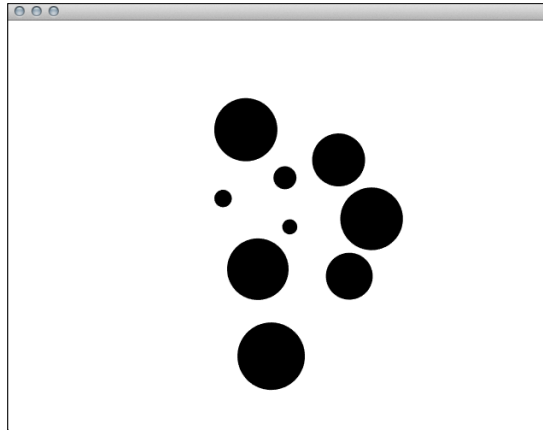
12. Draw particles by implementing the draw method as follows:

```
gl::clear(Color::white());
gl::setViewport(getWindowBounds());
gl::setMatricesWindow(getWindowWidth(), getWindowHeight());
gl::color(Color::black());
mParticleSystem.draw();
```

13. Implement audio callback handler as covered in the recipe *Generating a sine oscillator*.

## How it works...

We are generating random particles with applied physics and collision detection. While collision is detected, a frequency of a sine wave is calculated based on the particles' radii.



Inside the `update` method, we are iterating through the particles and checking the distance between each of them to detect collision, if it occurs. A generated frequency is calculated from the radii of the colliding particles—the bigger the radius, the lower the frequency of the sound.

## Visualizing FFT

In this recipe, we will show an example of **FFT (Fast Fourier Transform)** data visualization on a circular layout with some smooth animation.

### Getting ready

Save your favorite music piece in the assets folder with the name `music.mp3`.

### How to do it...

We will create visualization based on an example FFT analysis using the following steps:

1. Include the following necessary header files:

```
#include "cinder/gl/gl.h"
#include "cinder/audio/IO.h"
#include "cinder/audio/Output.h"
#include "cinder/audio/FftProcessor.h"
#include "cinder/audio/PcmBuffer.h"
```

2. Add the following members to your main application class:

```
void drawFft();
audio::TrackRef mTrack;
audio::PcmBuffer32fRef mPcmBuffer;
uint16_t bandCount;
float levels[32];
float levelsPts[32];
```

3. Inside the `setup` method, initialize the members and load the sound file from the assets folder. We are decomposing the signal into 32 frequencies using FFT:

```
bandCount = 32;
std::fill(boost::begin(levels), boost::end(levels), 0.f);
std::fill(boost::begin(levelsPts), boost::end(levelsPts), 0.f);
mTrack = audio::Output::addTrack(audio::load(
 getAssetPath("music.mp3").c_str()));
mTrack->enablePcmBuffering(true);
```

4. Implement the `update` method as follows:

```
mPcmBuffer = mTrack->getPcmBuffer();
for(int i = 0; i < (bandCount); i++) {
 levels[i] = max(0.f, levels[i] - 1.f);
 levelsPts[i] = max(0.f, levelsPts[i] - 0.95f);
}
```



5. Implement the draw method as follows:

```
gl::enableAlphaBlending();
gl::clear(Color(1.0f, 1.0f, 1.0f));
gl::color(Color::black());
gl::pushMatrices();
gl::translate(getWindowCenter());
gl::rotate(getElapsedSeconds() * 10.f);
drawFft();
gl::popMatrices();
```

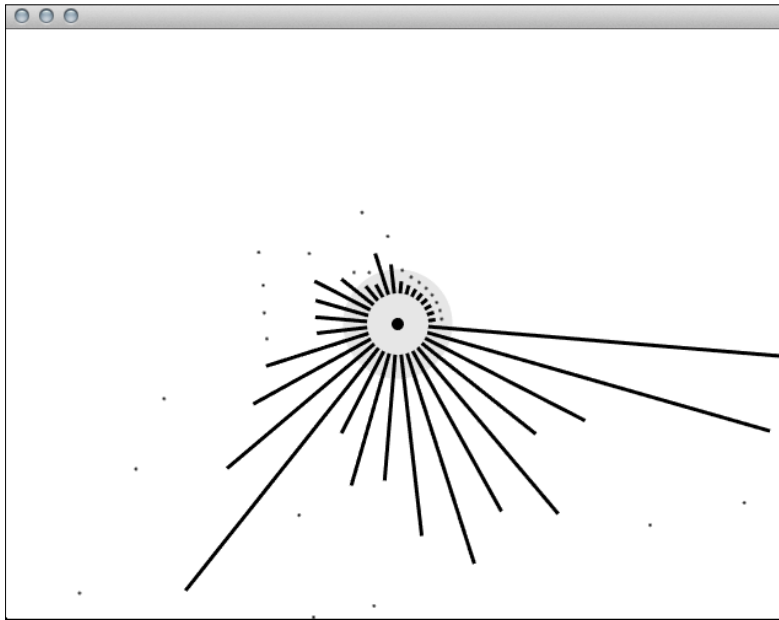
6. Implement the drawFft method as follows:

```
float centerMargin= 25.0f;
if(!mPcmBuffer) return;
std::shared_ptr<float> fftRef = audio::calculateFft(
 mPcmBuffer->getChannelData(audio::CHANNEL_FRONT_LEFT),
 bandCount);
if(!fftRef) {
 return;
}
float *fftBuffer = fftRef.get();
gl::color(Color::black());
gl::drawSolidCircle(Vec2f::zero(), 5.f);
glLineWidth(3.f);
float avgLvl= 0.f;
for(int i= 0; i<bandCount; i++) {
 Vec2f p = Vec2f(0.f, 500.f);
 p.rotate(2.f * M_PI * (i/(float)bandCount));
 float lvl = fftBuffer[i] / bandCount * p.length();
 lvl = min(lvl, p.length());
 levels[i] = max(levels[i], lvl);
 levelsPts[i] = max(levelsPts[i], levels[i]);
 p.limit(1.f + centerMargin + levels[i]);
 gl::drawLine(p.limited(centerMargin), p);
 glPointSize(2.f);
 glBegin(GL_POINTS);
 gl::vertex(p+p.normalized()*levelsPts[i]);
 glEnd();
 glPointSize(1.f);
 avgLvl += lvl;
}
avgLvl /= (float)bandCount; glLineWidth(1.f);
gl::color(ColorA(0.f,0.f,0.f, 0.1f));
gl::drawSolidCircle(Vec2f::zero(), 5.f+avgLvl);
```

## How it works...

We can divide visualization into bands, and the grey circle with alpha in the center. Bands are straight representations of data calculated by the `audio::calculateFft` function, and animated with some smoothing by going back towards the center. The grey circle shown in the following screenshot represents the average level of all the bands.

FFT is an algorithm to compute **DFT (Discrete Fourier Transform)**, which decomposes the signal into list of different frequencies.



## Making sound-reactive particles

In this recipe, we will show an example of audio visualization based on audio-reactive particles.

### Getting ready

Save your favorite music piece in assets folder with the name `music.mp3`.

Please refer to *Chapter 6, Adding a Tail to Our Particles*, for instructions on how to draw particles with a tile.

## How to do it...

We will create a sample audio-reactive visualization using the following steps:

1. Add the following necessary header files:

```
#include "cinder/Rand.h"
#include "cinder/MayaCamUI.h"
#include "cinder/audio/IO.h"
#include "cinder/audio/Output.h"
#include "cinder/audio/FftProcessor.h"
#include "cinder/audio/PcmBuffer.h"
#include "ParticleSystem.h"
```

2. Add the following members for audio playback and analysis:

```
audio::TrackRef mTrack;
audio::PcmBuffer32fRef mPcmBuffer;
float beatForce;
float beatSensitivity;
float avgLvlOld;
float randAngle;
```

3. Add the following members for particle simulation:

```
ParticleSystem mParticleSystem;
Vec3f attrPosition;
float attrFactor;
CameraPersp mCam;
```

4. Inside the `setup` method, initialize the simulation of the members and particles:

```
beatForce = 150.f;
beatSensitivity = 0.03f;
avgLvlOld = 0.f;
randAngle = 15.f;
attrPosition = Vec3f::zero();
attrFactor = 0.05f;
int numParticle = 450;
for(int i=0; i<numParticle; i++){
 float x = Rand::randFloat(0.0f, getWindowWidth());
 float y = Rand::randFloat(0.0f, getWindowHeight());
 float z = Rand::randFloat(0.0f, getWindowHeight());
 float radius = Rand::randFloat(2.f, 5.f);
 float mass = radius;
 if(i>300) {
 radius = 1.f;
 mass = 1.0f;
 }
}
```

```

 }
 float drag = 0.95f;
 Particle *particle = new Particle(Vec3f(x, y, z), radius,
 mass, drag);
 mParticleSystem.addParticle(particle);
}

```

5. Inside the `setup` method, initialize camera and audio playback:

```

mCam.setPerspective(45.0f, 640.f/480.f, 0.1, 10000);
mCam.setEyePoint(Vec3f(0.f,0.f,500.f));
mCam.setCenterOfInterestPoint(Vec3f::zero());
mTrack = audio::Output::addTrack(audio::load(
 getAssetPath("music.mp3").c_str()));
mTrack->enablePcmBuffering(true);

```

6. Implement the `resize` method for updating camera properties in regards to resizing windows:

```

void MainApp::resize(ResizeEvent event)
{
 mCam.setPerspective(45.0f, getWindowAspectRatio(), 0.1, 10000);
}

```

7. Inside the `update` method, implement a simple beat detection. We are decomposing the signal into 32 frequencies using FFT:

```

float beatValue = 0.f;
mPcmBuffer = mTrack->getPcmBuffer();
if(mPcmBuffer) {
 int bandCount= 32;
 std::shared_ptr<float> fftRef = audio::calculateFft(
 mPcmBuffer->getChannelData(audio::CHANNEL_FRONT_LEFT),
 bandCount);
 if(fftRef) {
 float * fftBuffer = fftRef.get();
 float avgLvl= 0.f;
 for(int i= 0; i<bandCount; i++) {
 avgLvl += fftBuffer[i] / (float)bandCount;
 }
 avgLvl /= (float)bandCount;
 if(avgLvl>avgLvlOld+beatSensitivity) {
 beatValue = avgLvl - beatSensitivity;
 }
 avgLvlOld = avgLvl;
 }
}
}

```

8. Also, inside the update method, calculate the particle simulation:

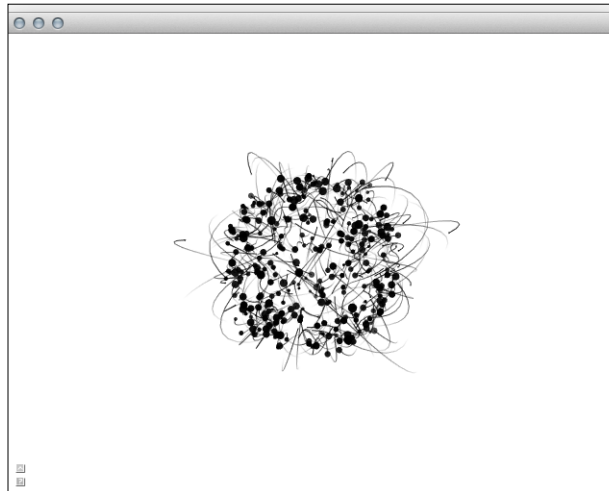
```
std::vector<Particle*>::iterator it;
for(it = mParticleSystem.particles.begin(); it !=
mParticleSystem.particles.end(); ++it) {
 Vec3f attrForce = attrPosition - (*it)->position;
 attrForce *= attrFactor;
 if(attrPosition.distance((*it)->position) <100.f) {
 attrForce = (*it)->position - attrPosition;
 attrForce *= 0.02f;
 }
 (*it)->forces += attrForce;
 Vec3f bearForceVec = (*it)->position - attrPosition;
 bearForceVec.normalize();
 bearForceVec.rotate(randVec3f(), randAngle);
 bearForceVec *= beatValue*randFloat(beatForce*0.5f,
beatForce);
 (*it)->forces += bearForceVec;
 std::vector<Particle*>::iterator it2;
 for(it2 = mParticleSystem.particles.begin(); it2 !=
mParticleSystem.particles.end(); ++it2) {
 (*it)->forces += ((*it)->position - (*it2)->position) *
0.5f * 0.0001f;
 }
}
mParticleSystem.update();
```

9. Implement the draw method as follows:

```
gl::enableAlphaBlending();
gl::clear(ColorA::white());
gl::setViewport(getWindowBounds());
gl::setModelView(mCam);
float r = getElapsedSeconds()*10.f;
gl::rotate(Vec3f::one()*r);
mParticleSystem.draw();
```

## How it works...

A particle is drawn as a black dot, or more precisely a sphere and a line as a tail. Due to specific frequency difference, forces repelling particles from the center of the attractor are applied, with a random vector added to these forces.



### There's more...

You might want to customize the visualization for a specific music piece.

### Adding GUI to tweak parameters

We will add GUI that affects particles' behavior using the following steps:

1. Add the following necessary header file:  

```
#include "cinder/params/Params.h"
```
2. Add the following member to your application's main class:  

```
params::InterfaceGl mParams;
```
3. At the end of the `setup` method, initialize GUI using the following code:  

```
mParams = params::InterfaceGl("Parameters", Vec2i(200, 100));
mParams.addParam("beatForce", &beatForce, "step=0.01");
mParams.addParam("beatSensitivity", &beatSensitivity, "step=0.01"
);
mParams.addParam("randAngle", &randAngle, "step=0.01");
```
4. At the end of the `draw` method, add the following code:  

```
params::InterfaceGl::draw();
```



# Index

## Symbols

### 2D

particle system, creating 101-108

### 2D geometric primitives

drawing 163-165

working 166

### 2D graphics

2D metaballs, implementing 171

about 163

arbitrary shapes, drawing 166

blur effect, adding 180

force-directed graph, implementing 184

geometric primitives, drawing 163

scribbler algorithm, implementing 169

text around curves, animating 174

### 2D metaballs

about 171

implementing 171, 172

wikipedia 173

working 173

### 3D

image gallery, creating 240-244

### 3D drawing

mouse, using 198-201

### 3D geometric primitives

about 189

drawing 190-192

working 192

### 3D graphics

about 189

drawing, with mouse 198

geometric primitives, drawing 189

height map, creating from image 210

lights, adding 201

mesh data, saving 217

mouse intersection 205

offscreen canvas, drawing 195

rotating 193

scaling 193

terrain creating, Perlin noise used 213

transforming, OpenGL transformations

used 193-195

translating 193

### 3D space guides

about 43

using 44-46

## A

**accelerated( AccelEvent event ) method 54**

**addAssetDirectory method 29**

**addParticle method 105**

**addSpring method 131**

**anchor property 125**

**animation sequences**

creating, with timeline 221-223

**Apple Developer Center 54**

**application basic structure**

about 10-12

implementing 11

working 12

**application window**

dropped files, accessing 20, 21

**arbitrary shapes**

drawing, with mouse 166-168

**assets**

about 28

using 28, 29

working 29

**attraction forces**

applying 109, 110



**audioCallback method** 314, 317  
**augmented reality**  
building, Kinect used 304-309

## B

**basic application project**  
creating 6, 7  
**blur effect**  
adding 180-183  
working 184  
**bool type** 33  
**brightness**  
transforming 56, 58  
**Brownian motion**  
applying, particle system 111, 112  
**B-spline** 161  
**buttons** 35

## C

**Cairo** 75  
**calcIntersectionWithMeshTriangles**  
method 207  
**camera**  
frames, capturing 277-279  
frames, displaying 277-279  
**camera motion**  
aligning, to path 226, 228-230  
**ci::app::FileDropEvent object** 20, 21  
**ci::app::KeyEvent class** 15  
**ci::app::MouseEvent class** 13  
**ci::app::ResizeEvent object** 24  
**ci::app::ResizeEvent parameter** 22  
**ci::app::TouchEvent class** 17  
**ci::ColorA type** 34  
**ci::Color type** 34  
**ci::fs::path object** 29  
**ci::gl::color method** 166  
**ci::gl::Light::DIRECTION light** 204  
**ci::gl::Light::POINT light** 204  
**ci::gl::Light::SPOTLIGHT** 204  
**Cinder** 5  
**Cinder-Config** 39  
**CINDER\_PATH** 48  
**ci::Quatf type** 34  
**ci::Shape2d::moveTo method** 168  
**ci::Timeline::apply method** 221

**ci::Vec3f type** 33  
**cloth simulation**  
creating 142-146  
texturing 147-151  
working 146  
**collision sounds**  
generating 319-322  
**color based object, tracking**  
frames, displaying 279-283  
**command key** 16  
**configuration**  
Cinder-Config 39  
loading 36-38  
saving 36-38  
**current parameter state**  
snapshot, making 39-41  
working 41

## D

**delay effect**  
about 317  
adding 317-319  
**destroySpring methods** 130  
**detectMultiScale function** 66  
**DFT** 325  
**Discrete Fourier Transform.** *See* DFT  
**dots**  
connecting 154, 156  
**draw3DScene method** 307  
**drawing method** 116  
**draw method** 10, 23, 50, 73, 108, 215  
**drawObject method** 306

## E

**edgeDetectSobel function** 63  
**edges**  
detecting 62, 63  
working 63, 64  
**example communication**  
about 47, 48  
broadcast 52  
Isiener 50  
OpenSoundControl Protocol 52  
OSC in Flash 52  
OSC in openFrameworks 52  
OSC in Processing 52

sender 48  
working 51, 52

## F

### faces

detecting 65-67

**Fast Fourier Transform.** *See* FFT

**Fast Library for Approximate Nearest Neighbor.** *See* FLANN

### FFT

about 117, 323  
visualizing 323, 325

**fileDrop method** 20

**FLANN** 291

**FlannBasedMatcher class** 69

**flocking** 117

### flocking behavior

alignment rule 112  
cohesion rule 112  
separation rule 112  
simulating 112-116

### flow field

creating, with Perlin noise 236

### force-directed graph

about 184  
implementing 185-187  
working 188

**Frame Buffer Object (FBO)** 184, 195

**fromOcv functions** 62

## G

**getActiveTouches() method** 54

**getAspectRatio method** 22

**getAssetPath method** 29

**getFile method** 21

**getId method** 17

**getNativeKeyCode method** 16

**getNativeModifiers method** 14

**getPosition method** 225

**getPos method** 17

**getSize method** 22

**getWheelIncrement method** 14

**getY methods** 17

**GitHub repository** 59

**Graphical User Interface (GUI)** 31

## H

### height map

creating, from image 210-213

### high resolution images

saving, with tile renderer 94-96  
working 97

### HSV 283

**hue, saturation, and value.** *See* HSV

## I

### image

height map, creating from 210-213

### image contrast

transforming 56, 58

### image gallery

creating, in 3D 240-244

**image processing techniques** 56

### images

converting, to vector graphics 70-74  
features, detecting 67-70  
matching case 70  
possibilities 70

### interactive object

mouse events, adding 255-260

**InteractiveObject class, for graphical object creation**

creating 250-255

**InterfaceGl::draw() method** 35

### iOS

resources, using in 26, 27

### iOS application

Apple Developer Center 54  
preparing 53, 54

### iOS touch application project

creating 9

**isShiftDown method** 14

## K

**key** 34

**keyDecr** 35

**keyDown method** 15, 73

**keyIncr** 34

### key input

responding to 15

responding to, steps 15

**keyPressed method 266**

**keyUp method 15**

**Kinect**

used, for augmented reality building 304-309

used, for gesture recognition 296-304

used, for UI navigation building 296-304

## L

**LFO 316**

**libopencv\_core.a module 62**

**libopencv\_imgproc.a module 62**

**libopencv\_objdetect.a module 62**

**lights**

adding 201-204

ambient property 204

diffuse property 204

emission property 204

shininess property 204

specular property 204

types 204, 205

**listener 50**

**loadResource method 27**

**low frequency oscillation.** *See* LFO

## M

**makeSnapshotClick method 41**

**matchImages method 68**

**max 34**

**MayaCamUI**

about 41, 42

setup method, working 43

using 43

**mesh data**

saving 217, 218

**mesh surface**

particles, aligning to 124-128

**min 34**

**motion tracking**

optical flow, using 284-286

**mouse**

used, for 3D drawing 198-201

used, for arbitrary shape creating 166-168

**mouse cursor interaction**

calculating 205-209

**mouseDown event 146**

**mouseDown implementation 133**

**mouseDown method 13, 109, 133**

**mouseDrag method 13, 49, 167**

**MouseEvent object 14**

**mouse events**

adding, to interactive object 255-260

**mouse input**

responding to 13, 14

working 14

**mouseMove method 13, 109**

**mouseUp event 146**

**mouseUp method 13**

**mouseWheel method 13**

**multi-touch**

used, for object dragging 268-276

used, for object rotation 268-276

used, for object scaling 268-276

## N

**Numerical: int, float, double type 33**

## O

**object tracking**

about 287

steps 288-291

**offscreen canvas**

drawing 195-198

**O key 218**

**OpenCV**

interacting with 59-62

**Open Sound Control.** *See* OSC

**optical flow**

used, for motion tracking 284-286

**OSC 47**

**OSC protocol 52**

**OS X**

resources, using in 26, 27

## P

**panel position 36**

**parameters tweaking**

AntTweakBar 36

buttons 35

ciUI 36

GUI, setting up 31-35  
panel position 36  
SimpleGUI 36

### **particles**

aligning, to mesh surface 124-128  
aligning, to processed image 121-124  
connecting, with spline 157-161  
FFT analysis, adding 117-121  
tail, adding to 139, 141  
texturing 137-139

### **particle system**

about 101  
Brownian motion, applying 111, 112  
creating, in 2D 101-108  
texturing, point sprites used 149-153  
texturing, shaders used 149-153

### **ParticleSystem::draw method 132**

### **path**

animating along 224-226  
camera motion, aligning 226-230

### **PCM**

about 311

### **Perlin noise**

about 213  
used, for flow field creation 236-240  
used, for spherical flow field  
creating 245-248  
used, for terrain creating 213-217

### **Perlin noise original source 112**

### **precision 35**

### **prepareSettings method 10-12**

### **project**

for basic application, creating 6, 7  
for iOS touch application, creating 9  
for screensaver application, creating 8

### **Pulse-code Modulation. See PCM**

## **Q**

### **QR code**

reading 292-295

## **R**

### **readonly 35**

### **receivedEvent method 258**

### **renderDrawing method 74**

### **repulsion forces**

applying 109, 110

### **repulsionRadius value 110**

### **resize method 24, 320**

### **resources**

using, on iOS 26, 27  
using, on OS X 26, 27  
using, on Windows 24, 25

### **responsive text box**

about 264  
creating 264-268  
working 268

## **S**

### **saveParameters method 41**

### **scene**

adjusting, after window resize 22, 24

### **screensaver application project**

creating 8

### **scribbler algorithm**

about 169  
implementing 169, 170

### **sender 48**

### **setAmbient( const Color& color ) method 205**

### **setDiffuse( const Color& color ) method 205**

### **setPos method 175**

### **setSpecular( const Color& color ) method 205**

### **setup method 10, 43, 68, 72, 109, 188, 216, 276**

### **Shift key 200**

### **shutdown method 10, 12, 38, 308**

### **simple video controller**

about 80  
creating 80-84

### **sine oscillator**

about 311  
generating 312, 313  
working 314

### **sine wave oscillator**

modulating, with low frequency 314, 316

### **S key 74, 218**

### **slider**

about 260  
creating 261-264

**sound-reactive particles**  
about 325  
creating 325-328  
GUI, adding for parameter tweaking 329  
working 328

**Speeded Up Robust Features.** *See* **SURF**

**spherical flow field**  
creating, with Perlin noise 245-248

**spline**  
used, for particles connecting 157-161

**springs**  
about 128  
creating 128-133  
working 134, 135

**step 34**

**st::string type 33**

**SURF 291**

**SurfaceEps method 74**

**SurfacePdf method 74**

**SurfacePs method 74**

**SurfaceSvg method 74**

## T

**tail**

adding, from several lines 140  
adding, to particle animation 139-141  
drawing history 141  
drawing, with lines 142

**terrain**

creating, with Perlin noise 213-217

**text**

animating around a user-defined  
curve 174-179  
using, as mask for movie 230-233

**text scrolling line by line**

creating 233-235

**tile renderer**

using, for high resolution images 94

**timeline**

animating 219-221  
animation sequences, creating 221-223

**TinderBox 5**

**touchesBegan event 275**

**touchesBegan method 16, 18**

**touchesBegan( TouchEvent event )  
method 54**

**touchesEnded method 16, 18**

**touchesEnded( TouchEvent event )  
method 54**

**touchesMoved method 16, 18**

**touchesMoved( TouchEvent event )  
method 54**

**touch input**

responding to 16

responding to, steps 17-20

**TouchInteractiveObject class 275**

## U

**update method 10, 11, 106, 160**

**user interaction 249**

## V

**video**

about 77

displaying 77-80

loading 78-80

## W

**window animation**

saving, as vector graphics image 90-94

saving, as video 86-89

**window content**

saving, as image 84, 85

saving, as image sequences 85

sound visualization, recording 86

**Windows**

resources, using 24, 25



## **Thank you for buying Cinder Creative Coding Cookbook**

### **About Packt Publishing**

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.packtpub.com](http://www.packtpub.com).

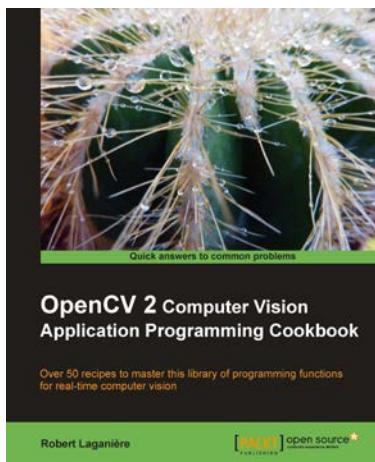
### **About Packt Open Source**

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

### **Writing for Packt**

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

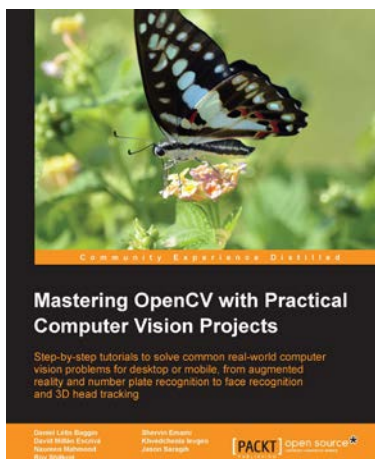


## OpenCV 2 Computer Vision Application Programming Cookbook

ISBN: 978-1-84951-324-1 Paperback: 304 pages

Over 50 recipes to master this library of programming functions for real-time computer vision

1. Teaches you how to program computer vision applications in C++ using the different features of the OpenCV library
2. Demonstrates the important structures and functions of OpenCV in detail with complete working examples
3. Describes fundamental concepts in computer vision and image processing



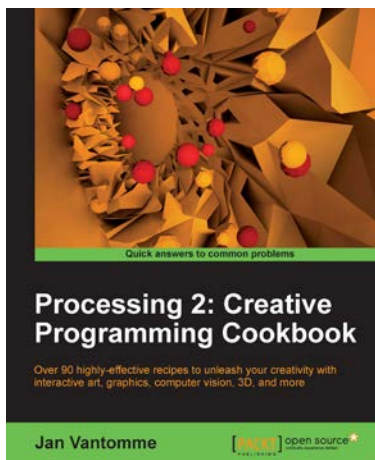
## Mastering OpenCV with Practical Computer Vision Projects

ISBN: 978-1-84951-782-9 Paperback: 340 pages

Step-by-step tutorials to solve common real-world computer vision problems for desktop or mobile, from augmented reality and number plate recognition to face recognition and 3D head tracking

1. Allows anyone with basic OpenCV experience to rapidly obtain skills in many computer vision topics, for research or commercial use
2. Each chapter is a separate project covering a computer vision problem, written by a professional with proven experience on that topic
3. All projects include a step-by-step tutorial and full source-code, using the C++ interface of OpenCV

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles



## Processing 2: Creative Programming Cookbook

ISBN: 978-1-84951-794-2      Paperback: 306 pages

Over 90 highly-effective recipes to unleash your creativity with interactive art, graphics, computer vision, 3D and more

1. Explore the Processing language with a broad range of practical recipes for computational art and graphics
2. Wide coverage of topics including interactive art, computer vision, visualization, drawing in 3D, and much more with Processing
3. Create interactive art installations and learn to export your artwork for print, screen, Internet, and mobile devices



## Mastering openFrameworks: Creative Coding Demystified

ISBN: 978-1-84951-804-8      Paperback: 340 pages

Boost your creativity and develop highly-interactive projects for art, 3D, graphics, computer vision and more, with this comprehensive tutorial.

1. A step-by-step practical tutorial that explains openFrameworks through easy to understand examples
2. Makes use of next generation technologies and techniques in your projects involving OpenCV, Microsoft Kinect, and so on
3. Sample codes and detailed insights into the projects, all using object oriented programming

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles