Making Things See

Greg Borenstein

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Making Things See
by Greg Borenstein

Revision History for the :
See *http://oreilly.com/catalog/errata.csp?isbn=9781449307073* for release details.

ISBN: 978-1-449-30707-3
1317923969

Table of Contents

Preface

When Microsoft first released the Kinect, Matt Webb, CEO of design and invention firm Berg London, captured the sense of possibility that had so many programmers, hardware hackers, and tinkerers so excited:

> WW2 and ballistics gave us digital computers. Cold War decentralisation gave us the Internet. Terrorism and mass surveillance: Kinect.

Why the Kinect Matters

The Kinect announces a revolution in technology akin to those that shaped the most fundamental breakthroughs of the 20th Century. Just like the premiere of the personal computer or the Internet, the release of the Kinect was another moment when the fruit of billions of dollars and decades of research that had previously only been available to the military and the intelligence community fell into the hands of regular people.

Face recognition, gait analysis, skeletonization, depth imaging — this cohort of technologies that had been developed to detect terrorists in public spaces could now suddenly be used for creative civilian purposes: building gestural interfaces for software, building cheap 3D scanners for personalized fabrication, using motion capture for easy 3D character animation, using biometrics to create customized assistive technologies for people with disabilities, etc.

While this development may seem wide-ranging and diverse, it can be summarized simply: for the first time, computers can see. While we've been able to use computers to process still images and video for decades, simply iterating over red, green, and blue pixels misses most of the amazing capabilities that we take for granted in the human vision system: seeing in stereo, differentiating objects in space, tracking people over time and space, recognizing body language, etc. For the first time, with this revolution in camera and image-processing technology, we're starting to build computing applications that take these same capabilities as a starting point. And, with the arrival of the Kinect, the ability to create these applications is now within the reach of even weekend tinkerers and casual hackers.

Just like the personal computer and internet revolutions before it, this Vision Revolution will surely also lead to an astounding flowering of creative and productive projects. Comparing the arrival of the Kinect to the personal computer and the internet may sound absurd. But keep in mind that when the personal computer was first invented it was a geeky toy for tinkerers and enthusiasts. The internet began life as a way for government researchers to access each others' mainframe computers. Each of these technologies only came to assume their critical roles in contemporary life slowly as individuals used them to make creative and innovative applications that eventually became fixtures in our daily lives. Right now it may seem absurd to compare the Kinect with the PC and the internet, but a few decades from now we may look back on it and compare it with the Altair or the ARPAnet as the first baby step towards a new technological world.

The purpose of this book is to provide the context and skills needed to build exactly these projects that reveal this newly possible world. Those skills include:

- working with depth information from 3D cameras
- analyzing and manipulating point clouds
- tracking the movement of people's joints
- background removal and scene analysis
- pose and gesture detection

The first three chapters of this book will introduce you to all of these skills. You'll learn how to implement each of these techniques in the Processing programming environment. We'll start with the absolute basics of accessing the data from the Kinect and build up your ability to write ever more sophisticated programs throughout the book. But learning these skills means not just mastering a particular software library or API, but understanding the principles behind them so that you can apply them even as the practical details of the technology rapidly evolve.

And yet even mastering these basic skills will not be enough to build the projects that really make the most of this Vision Revolution. To do that you also need to understand some of the wider context of the fields that will be revolutionized by the cheap, easy availability of depth data and skeleton information. To that end, this book will provide introductions and conceptual overviews of the fields of 3D scanning, digital fabrication, robotic vision, and assistive technology. You can think of these sections as teaching you what you can do with the depth and skeleton information once you've gotten it. They will include topics like:

- building meshes
- preparing 3D models for fabrication
- defining and detecting gestures
- displaying and manipulating 3D models
- designing custom input devices for people with limited ranges of motion

- forward and inverse kinematics

In covering these topics, our focus will expand outward from simply working with the Kinect to using a whole toolbox of software and techniques. The last three chapters of this book will explore these topics through a series of in-depth projects. We'll write a program that uses the Kinect as a scanner to produce physical objects on a 3D printer, we'll create a game that will help a stroke patient with their physical therapy, and we'll construct a robot arm that copies the motions of your actual arm. In these projects we'll start by introducing the basic principles behind each general field and then seeing how our newfound knowledge of programming with the Kinect can put those principles into action. But we won't stop with Processing and the Kinect. We'll work with whatever tools are necessary to build each application, from 3D modeling programs to microcontrollers.

This book will not be a definitive reference to any of these topics; each of them is vast, comprehensive, and filled with its own fascinating intricacies. This book aims to serve as a provocative introduction to each of these areas: giving you enough context and techniques to start using the Kinect to make interesting projects and hoping that your progress will inspire you to follow the leads provided to investigate further.

Who This Book Is For

At its core, this book is for anyone who wants to learn more about building creative interactive applications with the Kinect from interaction and game designers who want to build gestural interfaces to makers who want to work with a 3D scanner to artists who want to get started with computer vision.

That said, you will get the most out of it if you are one of the following: a beginning programmer looking to learn more sophisticated graphics and interactions techniques, specifically how to work in three dimensions, or, an advanced programmer who wants a shortcut to learning the ins and outs of working with the Kinect and a guide to some of the specialized areas that enables.

You don't have to be an expert graphics programmer or experienced user of Processing to get started with this book, but if you've never programmed before there are probably other much better places to start.

As a starting point, I'll assume that you have some exposure to the Processing creative coding language (or can figure teach yourself that as you go). You should know the basics from Getting Started with Processing by Casey Reas and Ben Fry, Learning Processing by Dan Shiffman, or the equivalent. This book is designed to proceed slowly from introductory topics into more sophisticated code and concepts, giving you a smooth introduction to the fundamentals of making interactive graphical applications while teaching you about the Kinect. At the beginning I'll explain nearly everything about each example and as we go, I'll leave more and more of the details to you to figure out. The goal is for you to level up from a beginner to a confident intermediate.

The Structure of This Book

The goal of this book is to unlock your ability to build interactive applications with the Kinect. It's meant to make you into a card-carrying member of the Vision Revolution I described at the beginning of this introduction. Membership in this Revolution has a number of benefits. Once you've achieved it you'll be able to play an invisible drum set that makes real sounds, make 3D scans of objects and print copies of them, and teach robots to copy the motions of your arm.

However, membership in this Revolution does not come for free. To gain entry into its ranks you'll need to learn a series of fundamental programming concepts and techniques. These skills are the basis of all the more advanced benefits of membership and all of those cool abilities will be impossible without them. This book is designed to build up those skills one at a time, starting from the simplest and most fundamental and building towards the more complex and sophisticated. We'll start out with humble pixels and work our way up to intricate three dimensional gestures.

Towards this end, the first half of this book will act as a kind of primer in these programming skills. Before we dive into controlling robots or 3D printing our faces, we need to start with the basics. The first four chapters of this book cover the fundamentals of writing Processing programs that use the data from the Kinect.

Processing is a creative coding environment that uses the Java programming language to make it easy for beginners to write simple interactive applications that include graphics and other rich forms of media. As mentioned in the introduction, this book assumes basic knowledge of Processing (or equivalent programming chops), but as we go through these first four chapters, I'll build up your knowledge of some of the more advanced Processing concepts that are most relevant to working with the Kinect. These concepts include looping through arrays of pixels, basic 3D drawing and orientation, and some simple geometric calculations. If you've never used Processing before I highly recommend Getting Started with Processing by Casey Reas and Ben Fry or Learning Processing by Dan Shiffman two excellent introductory texts.

I will attempt to explain each of these concepts clearly and in depth. The idea is for you not to just to have a few project recipes that you can make by rote, but to actually understand enough of the flavor of the basic ingredients to be able to invent your own "dishes" and modify the ones I present here. At times you may feel that I'm beating some particular subject to death, but stick with it—you'll frequently find that these details become critically important later on when trying to get your own application ideas to work.

One nice side benefit to this approach is that these fundamental skills are relevant to a lot more than just working with the Kinect. If you master them here in the course of your work with the Kinect, they will serve you well throughout all your other work with Processing, unlocking many new possibilities in your work, and really pushing you decisively beyond beginner status.

There are three fundamental techniques that we need to build all of the fancy applications that make the Kinect so exciting: processing the depth image, working in 3D, and accessing the skeleton data. From 3D scanning to robotic vision, all of these applications measure the distance of objects using the depth image, reconstruct the image as a three dimensional scene, and track the movement of individual parts of a user's body. The first half of this book will serve as an introduction to each of these techniques. I'll explain how the data provided by the Kinect makes each of these techniques possible, demonstrate how to implement them in code, and walk you through a few simple examples to show what they might be good for.

Working with the Depth Camera

First off, you'll learn how to work with the depth data provided by the Kinect. As I explained in the introduction, the Kinect uses an IR projector and camera to produce a "depth image" of the scene in front of it. Unlike conventional images where each pixel records the color of light that reached the camera from that part of the scene, each pixel of this depth image records the distance of the object in that part of the scene from the Kinect. When we look at depth images, they will look like strangely distorted black and white pictures. They look strange because the color of each part of the image indicates not how bright that object is, but how far away it is. The brightest parts of the image are the closest and the darkest parts are the furthest away. If we write a Processing program that examines the brightness of each pixel in this depth image, we can figure out the distance of every object in front of the Kinect. Using this same technique and a little bit of clever coding, we can also follow the closest point as it moves, which can be a convenient way of tracking a user for simple interactivity.

Working with Point Clouds

This first approach treats the depth data as if it was only two dimensional. It looks at the depth information captured by the Kinect as a flat image when really it describes a three dimensional scene. In the third chapter, we'll start looking at ways to translate from these two dimensional pixels into points in three dimensional space. For each pixel in the depth image we can think of its position within the image as its x-y coordinates. That is, if we're looking at a pixel that's 50 pixels in from top left corner and 100 pixels down, it has an x-coordinate of 50 and a y-coordinate of 100. But the pixel also has a grayscale value. And we know from our initial discussion of the depth image that each pixel's grayscale value corresponds to the depth of the image in front of it. Hence, that value will represent the pixel's z-coordinate.

Once we've converted all our two-dimensional grayscale pixels into three dimensional points in space, we have what is called a "point cloud", i.e. a bunch of disconnected points floating near each other in three-dimensional space in a way that corresponds to the arrangement of the objects and people in front of the Kinect. You can think of this point cloud as the 3D equivalent of a pixelated image. While it might look solid

from far away, if we look closely the image will break down into a bunch of distinct points with space visible between them. If we wanted to convert these points into a smooth continuous surface we'd need to figure out a way to connect them with a large number of polygons to fill in the gaps. This is a process called "constructing a mesh" and it's something we'll cover extensively later in the book in the chapters on physical fabrication and animation.

For now though, there's a lot we can do with the point cloud itself. First of all, the point cloud is just cool. Having a live 3D representation of yourself and your surroundings on your screen that you can manipulate and view from different angles feels a little bit like being in the future. It's the first time in using the Kinect that you'll get a view of the world that feels fundamentally different that those that you're used to seeing through conventional cameras.

In order to make the most of this new view, you're going to learn some of the fundamentals of writing code that navigates and draws in 3D. When you start working in 3D there are a number of common pitfalls that I'll try to help you avoid. For example, it's easy to get so disoriented as you navigate in 3D space that the shapes you draw end up not being visible. I'll explain how the 3D axes work in Processing and show you some tools for navigating and drawing within them without getting confused. Another frequent area of confusion in 3D drawing is the concept of the camera. In order to translate our 3D points from the Kinect into a 2D image that we can actually draw on our flat computer screens, Processing uses the metaphor of a camera. After we've arranged our points in 3D space, we place a virtual camera at a particular spot in that space, aim it at the points we've drawn, and, basically, take a picture. Just as a real camera flattens the objects in front of it into a 2D image, this virtual camera does the same with our 3D geometry. Everything that the camera sees gets rendered onto the screen from the angle and in the way that it sees it. Anything that's out of the camera's view doesn't get rendered. I'll show you how to control the position of the camera so that all of the 3D points from the Kinect that you want to see end up rendered on the screen. I'll also demonstrate how to move the camera around so we can look at our point cloud from different angles without having to ever physically move the Kinect.

Working with the Skeleton Data

The third technique is in some ways both the simplest to work with and the most powerful. In addition to the raw depth information we've been working with so far, the Kinect can, with the help of some additional software, recognize people and tell us where they are in space. Specifically, our Processing code can access the location of each part of the user's body in 3D: we can get the exact position of their hands, head, elbows, feet, etc.

One of the big advantages of depth images is that computer vision algorithms work better on them than on conventional color images. The reason Microsoft developed and shipped a depth camera as a controller for the XBox was not to show players cool

looking point clouds, but because they could run software on the XBox that processes the depth image in order to locate people and find the positions of their body parts. This process is known as "skeletonization" because the software infers the position of the user's skeleton (specifically, their joints and the bones that connect them) from the data in the depth image.

By using the right Processing library, we can get access to this user position data without having to implement this incredibly sophisticated skeletonization algorithm ourself. We can simply ask for the 3D position of any joint we're interested in and then use that data to make our applications interactive. In Chapter 4, I'll demonstrate how to access the skeleton data from the Kinect Processing library and how to use it to make our applications interactive. To create truly rich interactions we'll need to learn some more sophisticated 3D programming. In Chapter 3, when working with point clouds, we'll cover the basics of 3D drawing and navigation. In this chapter we'll add to those skills by learning more advanced tools for comparing 3D points with each other, tracking their movement, and even recording it for later playback. These new techniques will serve as the basic vocabulary for some exciting new interfaces we can our sketches, letting users communicate with us by striking poses, doing dance moves, and performing exercises (amongst many other natural human movements).

Once we've covered all three of these fundamental techniques for working with the Kinect, we'll be ready to move on to the cool applications that probably drew you to this book in the first place. This book's premise is that what's truly exciting about the Kinect is that it unlocks areas of computer interaction that were previously only accessible to researchers with labs full of expensive experimental equipment. With the Kinect things like 3D scanning and advanced robotic vision are suddenly available to anyone with a Kinect and an understanding of the fundamentals described here. But in order to make the most of these new possibilities, you need a bit of background in the actual application areas. To build robots that mimic human movements, it's not enough just to know how to access the Kinect's skeleton data, you also need some familiarity with inverse kinematics, the study of how to position a robot's joints in order to achieve a particular pose. To create 3D scans that can be used for fabrication or computer graphics, it's not enough to understand how to work with the point cloud from the Kinect, you need to know how to build up a mesh from those points and how to prepare and process it for fabrication on a Makerbot, a CNC, or 3D printer. To build gestural interfaces that are useful for actual people, it's not enough to just know how to find their hands in 3D space, you need to know something about the limitations and abilities of the particular group of people you're designing for and what you can build that will really help them.

The following three chapters will provide you with introductions to exactly these three topics: gestural interfaces for assistive technology, 3D scanning for fabrication, and 3D vision for robotics.

Gestural Interfaces for Assistive Technology

In Chapter 6 we'll conduct a close case study of an assistive technology project. Assistive technology is a branch of gestural interface design that uses this alternate form of human computer interaction to make digital technology accessible to people who have limited motion due to a health condition. People helped by assistive technology have a wide range of needs. There are older patients trying to use a computer after the loss of vision and fine motor control due to stroke. There are kids undergoing physical and occupational therapy to recover from surgery who need their exercises to be more interactive and more fun.

Assistive technology projects make a great case study for building gestural interfaces because its users tend to be even more limited than "average" computer users. Any interface ideas that work for an elderly stroke victim or a kid in physical therapy will probably work for average users as well. The rigors and limitations of assistive technology act as a gauntlet for interactive designers, bringing out their best ideas.

3D Scanning for Digital Fabrication

In Chapter 5 we'll move from people to objects. We'll use the Kinect as a 3D scanner to capture the geometry of a physical object in digital form and then we'll prepare that data for fabrication on a 3D printer. We'll learn how to process the depth points from the Kinect to turn them into a continuous surface or mesh. Then we'll learn how to export this mesh in a standard file format so we can work with it outside of Processing. I'll introduce you to a few free programs that help you clean up the mesh and prepare it for fabrication. Once our mesh is ready to go we'll examine what it takes to print it out on a series of different rapid prototyping systems. We'll use a Makerbot to print it out in plastic and we'll submit it to Shapeways, a website that will print out our object in a variety of materials from sandstone to steel.

Computer Vision for Robotics

In Chapter 7, we'll see what the Kinect can do for robotics. Robotic vision is a huge topic that's been around for more than 50 years. Its achievements include robots that have driven on the moon and ones that assemble automobiles. For this chapter we'll we'll build a simple robot arm that reproduces the position of your real arm as detected by the Kinect. We'll send the joint data from Processing to the robot over a serial connection. Our robot's brain will be an Arduino microcontroller. Arduino is Processing's electronic cousin; it makes it just as easy to create interactive electronics as Processing does interactive graphical applications. The Arduino will listen to the commands from Processing and control the robot's motors to execute them.

We'll approach this project in two different ways. First we'll reproduce the angles of your joints as detected by the Kinect. This approach falls into "forward kinematics", an approach to robotics in which the robot's final position is the result of setting its

joints to a series of known angles. Then we'll reprogram our robot so that it can follow the movement of any of your joints. This will be an experiment in "inverse kinematics". Rather than knowing exactly how we want our robot to move, we'll only know what we want its final position to be. We'll have to teach it how to calculate all the individual angle changes necessary to get there. This is a much harder problem than the forward kinematic problem. A serious solution to it can involve complex math and confusing code. Ours will be quite simple and not very sophisticated, but will provide an interesting introduction to the problems you'd encounter in more advanced robotics applications.

None of these chapters are meant to be definitive guides to their respective areas, but instead to give you just enough background to get started applying these Kinect fundamentals in order to build your own ideas.

Unlike the first four chapters which attempt to instill fundamental techniques deeply, these last three are meant to inspire a sense of the breadth and diversity of what's possible with the Kinect. Instead of proceeding slowly and thoroughly through comprehensive explanations of principles, these later chapters are structured as individual projects. They'll take a single project idea from one of these topic areas and execute it completely from beginning to end. In the course of these projects we'll frequently find ourselves moving beyond just writing Processing code. We'll have to interview occupational therapists, work with assistive technology patients, clean up 3D meshes, use a 3D animation program, solder a circuit, and program an Arduino. Along the way, you'll gain brief exposure to a lot of new ideas and tools, but nothing like the in depth understanding of the first four chapters. We'll move fast. It will be exciting. You won't believe the things you'll make.

At every step of the way in these projects, we'll rely on your knowledge from the first half of the book. So pay close attention as we proceed through these fundamentals, they're the building blocks of everything else throughout this book and getting a good grasp on them will make it all the easier for you to build whatever it is you're dreaming of.

Then, at the end of the book, our scope will widen. Having come so far in your 3D programming chops and your understanding of the Kinect I'll point you towards next steps that you can take to take your applications even further. We'll discuss other environments and programming languages besides Processing where you can work with the Kinect. These range from creative coding libraries in other languages like C++ to interactive graphical environments like Max/MSP, Pure Data, and Quartz Composer. And there's also Microsoft's own set of development tools, which let you deeply integrate the Kinect with Windows. I'll explain some of the advantages and opportunities of each of these environments to give you a sense of why you'd want to try them out. Also, I'll point you towards other resources that you can use to get started in each of them.

In addition to exploring other programming environments, you can also take your Kinect work further by learning about 3D graphics in general. Under the hood Processing's 3D drawing code is based on OpenGL, a widely used standard for computer graphics. OpenGL is a huge, complex, and powerful system and Processing only exposes you to the tiniest bit of it. Learning more about OpenGL itself will unlock all kinds of more advanced possibilities for your Kinect applications. I'll point you towards resources both within Processing and outside of it that will let you continue your graphics education and make ever more beautiful and compelling 3D graphics.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example

code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2011 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/<catalog page>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1

What is the Kinect?

We've talked a little bit about all the amazing applications that depth cameras like the Kinect make possible. But how does the Kinect actually work? What kind of image does it produce and why is it useful? How does the Kinect gather depth data about the scene in front of it? What's inside that sleek little black box?

How does it work? Where did it come from?

In the next few sections of this introduction, I'll answer these questions as well as giving you some background about where the Kinect came from. This issue of the Kinect's provenance may seem like it's only of academic interest. However, as we'll see, it is actually central when deciding which of the many available libraries we should use to write our programs with the Kinect. It's also a fascinating and inspiring story of what the open source community can do.

What Does the Kinect Do?

The Kinect is a *depth camera*. Normal cameras collect the light that bounces off of the objects in front of them. They turn this light into an image that resembles what we see with our own eyes. The Kinect, on the other hand, records the distance of the objects that are placed in front of it. It uses infrared light to create an image (a *depth image*) that captures not what the objects look like, but where they are in space. In the next section of this introduction, I'll explain how the Kinect actually works. I'll describe what hardware it uses to capture this depth image and explain some of its limitations. But first I'd like to explain why you'd actually want a depth image. What can you do with a depth image that you can't with a conventional color image?

First of all, a depth image is much easier for a computer to "understand" than a conventional color image. Any program that's trying to understand an image starts with its pixels and tries to find and recognize the people and objects represented by them. If you're a computer program and you're looking at color pixels, it's very difficult to differentiate objects and people. So much of the color of the pixels is determined by

the light in the room at the time the image was captured, the aperture and color shift of the camera, etc. How would you even know where one object begins and another ends, let alone which object was which and if there were any people present? In a depth image, on the other hand, the color of each pixel tells you how far that part of the image is from the camera. Since these values directly correspond to where the objects are in space, they're much more useful in determining where one object begins, where another ends, and if there are any people around. Also, because of how the Kinect creates its depth image (about which more in a second) it is not sensitive to the light conditions in the room at the time it was captured. The Kinect will capture the same depth image in a bright room as in a pitch black one. This makes depth images more reliable and even easier for a computer program to understand.

We'll explore this aspect of depth images much more thoroughly in Chapter 2.

A depth image also contains accurate three dimensional information about whatever's in front of it. Unlike a conventional camera, which captures how things *look*, a depth camera captures where things *are*. The result is that we can use the data from a depth camera like the Kinect to reconstruct a 3D model of whatever the camera sees. We can then manipulate this model, viewing it from additional angles interactively, combining it with other pre-existing 3D models, and even using it as part of a digital fabrication process to produce new physical objects. None of this can be done with conventional color cameras.

We'll begin exploring these possibilities in Chapter 3 and then continue with them in Chapter 5 when we investigate scanning for fabrication.

And finally, since depth images are so much easier to process than conventional color images, we can run some truly cutting edge processing on them. Specifically, we can use them to detect and track individual people, even locating their individual joints and body parts. In many ways this is the Kinect's most exciting capability. In fact, Microsoft developed the Kinect specifically for the opportunities this body-detection ability offered to video games (more about this later in "Who Made the Kinect?" on page 6. Tracking user's individual body parts creates amazing possibilities for our own interactive applications. Thankfully, we have access to software that can perform this processing and simply give us the location of the users. We don't have to analyze the depth image ourselves in order to obtain this information, but it's only accessible because of the depth image's suitability for processing.

We'll work extensively with the user-tracking data in Chapter 4.

What's Inside? How Does it Work?

If you remove the black plastic casing from the Kinect what will you find? What are the hardware components that make the Kinect work and how do they work together to give the Kinect its abilities? Let's take a look. Figure 1-1 shows a picture of a Kinect that's been freed from its case.

Figure 1-1. A Kinect with its plastic casing removed, revealing (from left to right) its IR projector, RGB camera, and IR camera. Photo courtesy of iFixit.

The first thing I always notice when looking at the Kinect *au natural* is its uncanny resemblance to various cute movie robots. From *Short Circuit*'s Johnny 5 to Pixar's WALL-E, for decades movie designers have been creating human-looking robots with cameras for eyes. It seems somehow appropriate (or maybe just inevitable) that the Kinect, the first computer peripheral to bring cutting-edge computer vision capabilities into our homes would end up looking so much like one of these robots.

Unlike these movie robots though, the Kinect seems to actually have three eyes: the two in its center and one off all the way to one side. That "third eye" is the secret to how the Kinect works. Like most robot "eyes", the two protuberances at the center of the Kinect are cameras, but the Kinect's third eye is actually an infrared projector. Infrared light has a wavelength that's longer than that of visible light so we cannot see it with the naked eye. Infrared is perfectly harmless, we're constantly exposed to every day in the form of sunlight.

The Kinect's infrared projector shines a grid of infrared dots over everything in front of it. These dots are normally invisible to us, but it is possible to capture a picture of them using a IR camera. Figure 1-2 shows an example of what the dots from the Kinect's projector look like:

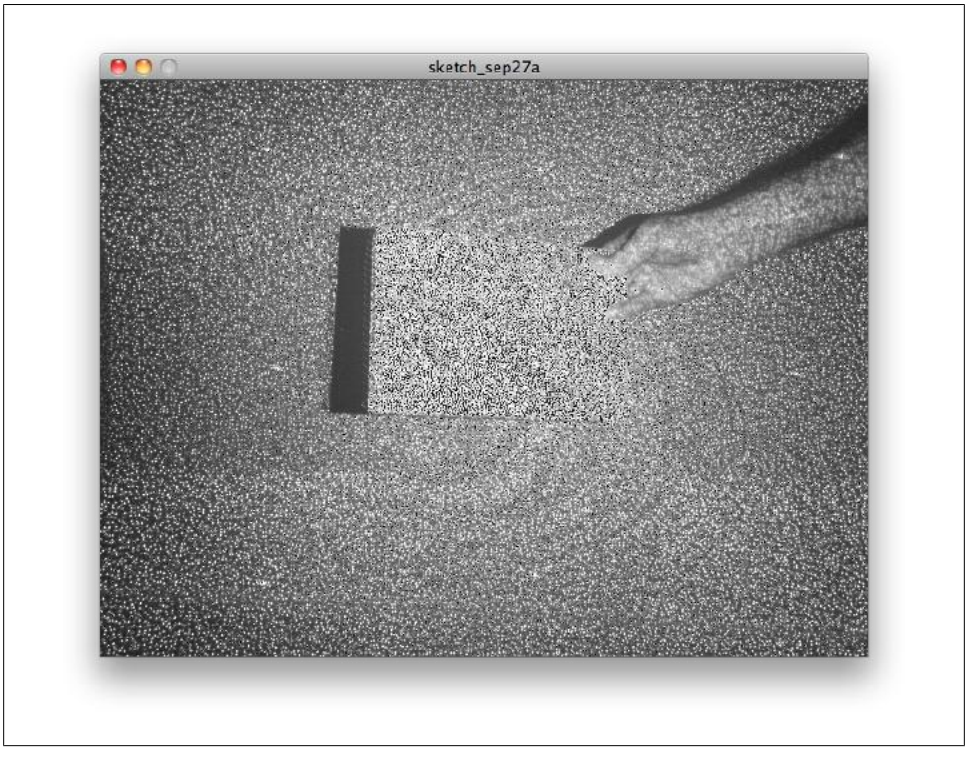


Figure 1-2. An image of the normally invisible grid of dots from the Kinect's infrared projector. Taken with the Kinect's IR camera.

I actually captured this image using the Kinect itself. One of those two cameras I pointed out earlier (one of the Kinect's two "eyes") is an IR camera. It's a sensor specifically designed to capture infrared light. In Figure 1-1, an image of the Kinect naked without its outer case, the IR camera is the one on the right. If you look closely, you can see that this camera's lens has a greenish iridescent sheen as compared with the standard visible light camera next to it.

So, the Kinect can see the grid of infrared dots that is projecting onto the objects in front of it. But how does it translate this image into information about the distance of those objects? In the factory where it was made, each Kinect is calibrated to know exactly where each of the dots from its projector appears when projected against a flat wall at a known distance. Look at the image of the IR projection again. Notice how the dots on the notebook I'm holding up in front of the wall seem pushed forward and shifted out of position? Any object that is closer than the Kinect's calibration distance will push these dots out of position in one direction and any object that's further away will push them out of position in the other direction. Since the Kinect is calibrated to know the original position of all of these dots, it can use their displacement to figure out the distance of the objects in that part of the scene. In every part of the image that

the Kinect captures from the IR camera, each dot will be a little out of position from where the Kinect was expecting to see it. The result is that the Kinect can turn this IR image of a grid of dots into depth data that captures the distance of everything it can see.

There are certain limitations that are inherent in how this system works. For example, notice the black shadow at the edge of the objects in the IR image. None of the dots from the Kinect's infrared projection are reaching that part of the scene. All of them are being stopped and reflected back by a closer object. That means the Kinect won't be able to figure out any depth information about that part of the scene. We'll discuss these limitations in much more detail in the first chapter when we start working with depth images in code. And we'll revisit them again throughout the book every time we introduce a new technique from drawing 3D point clouds to tracking users. As you learn these techniques, keep in mind how the Kinect is actually gathering its depth data. A lot of the data the Kinect provides seems so magical that it's easy to fall into thinking of it as having a perfect three dimensional picture of the scene in front of it. If you're ever tempted to think this way, remember back to this grid of dots. The Kinect can only see what these dots from its projector can hit.

This depth information is the basis of the most fun stuff the Kinect can do from acting as a 3D scanner to detecting the motion of people. We're going to spend the rest of the book working with it in one way or another. However, capturing depth images isn't the only thing the Kinect can do. There's a lot of other hardware inside this case and as long as we're in here it's worth pointing it out.

The first additional piece of hardware we've already mentioned. It's the Kinect's other "eye". Next to the IR camera, the Kinect also has a color camera. This camera has a digital sensor that's similar to the one in many web cams and small digital cameras. It has a relatively low resolution (640 by 480 pixels). By itself, this color camera is not particularly interesting. It's just a run-of-the mill low quality web cam. But since it's attached to the Kinect at a known distance from the IR camera, the Kinect can line up the color image from this camera with the depth information captured by its IR camera. That means it's possible to alter the color image based on its depth (for example, hiding anything more than a certain distance away). And, conversely, it's possible to "color in" the 3D images created from the depth information, creating 3D scans or virtual environments with realistic color.

In addition to cameras, the Kinect also has four other sensors you might find slightly surprising in a depth camera: microphones. These microphones are distributed around the Kinect much as your ears are distributed around your head. Their purpose is not just to let the Kinect capture sound. For that, one microphone would have been enough. By using many microphones together, the Kinect can not just capture sound, but also locate that sound within the room. For example, if multiple players are speaking voice commands to control a game, the Kinect can tell which commands are coming from which player. This is a powerful and intriguing feature, but we're not going to explore it in this book. Just covering the Kinect's imaging features is rich enough topic to keep us busy. And also, at the time of this writing the Kinect's audio features are not available

in the library we'll be using to access the Kinect. At the end of the book, we'll discuss ways you can move beyond Processing to work with the Kinect in other environments and languages. One of the options discussed there is Microsoft's own Kinect Software Developer Kit, which provides access to the Kinect's spatial audio features. If you are especially intrigued by this possibility, I recommend exploring that route (and sharing what you learn online! The Kinect's audio capabilities are amongst its less well-explored).

The last feature of the Kinect may seem even more surprising: it can move. Inside the Kinect's plastic base is a small motor and a series of gears. By turning this motor, the Kinect can tilt its cameras and speakers up and down. The motor's range of motion is limited to about 30 degrees. Microsoft added the motor to the Kinect in order to allow the Kinect to work in a greater variety of rooms. Depending on the size of the room and the position of the furniture, people playing with the XBox may stand closer to the Kinect or further away from it and they may be more or less spread out. The motor gives the Kinect the ability to aim itself at the best point for capturing the people who are trying to play with it. Like the Kinect's audio capabilities, control of the motor is not accessible in the library we'll be using throughout this book. However it is accessible in one of the other open source libraries that lets you work with the Kinect in Processing Dan Shiffman's Kinect Library.

I'll explain shortly why I chose a library that does not have access to the motor over this one that does for the examples in this book. In fact I'll give you a whole picture of the ins-and-outs of Kinect development: who created the Kinect in the first place, how it became possible for us to use it in our own applications, and all the different options we have for doing so. But before we complete our discussion of the Kinect's hardware I want to point out a resource you can use to find out more. iFixit is a website that takes apart new electronic gadgets in order to document what's inside of them. Whenever a new smartphone or tablet comes out, the staff of iFixit goes out and buys one and carefully disassembles it, photographing and describing everything the find inside, and then posting the results online. On the day of its launch, they performed one of these "teardowns" on the Kinect. If you want to learn more about how the Kinect's hardware works, from the kinds of screws it has to all of its electronic components, their report is the place to look: iFixit's Kinect Teardown

Who Made the Kinect?

The Kinect is a Microsoft product. It's a peripheral for Microsoft's XBox 360 video game system. However, Microsoft did not create the Kinect entirely on its own. In the broad sense, the Kinect is the product of many years of academic research conducted both by Microsoft (at their Microsoft Research division) and elsewhere throughout the computer vision community. If you're mathematically inclined, Richard Szeliski of Microsoft Research created a textbook based on computer vision courses he taught at the University of Washington that covers many of the recent advances that lead up to the Kinect: Computer Vision: Algorithms and Applications.

In a much narrower sense, the Kinect's hardware was developed by PrimeSense, an Israeli company that had previously produced other depth cameras using the same basic IR projection technique. PrimeSense worked closely with Microsoft to produce a depth camera that would work with the software and algorithms Microsoft had developed in their research. PrimeSense licensed the hardware design to Microsoft to create the Kinect but still owns the basic technology themselves. In fact, PrimeSense has already announced that they're working with ASUS to create a product called the Wavi Xtion, a depth camera similar to the Kinect that is meant to integrate with your TV and personal computer for apps and games.

Until November 2010, the combination of Microsoft's software with PrimeSense's hardware was known by its codename, "Project Natal". On November 4th, the device was launched as the "Microsoft Kinect" and went on public sale for the first time. At this point a new chapter began in the life of the project. The Kinect was a major commercial success. It sold upwards of 10 million units in the first month after its release, making it the fastest selling computer peripheral in history.

But for our purposes maybe an even more important landmark was the launch of Adafruit's Kinect bounty on the day of the Kinect's release. Adafruit is a New York-based company that sells kits for open source hardware projects, many of them based on the Arduino microcontroller. On the day the Kinect was released, Limor Fried, Adafruit's founder, announced a bounty of $2,000 to the first person who produced open source drivers that would let anyone access the Kinect's data.

The Kinect plugs into your computer via USB, just like many other devices such as mice, keyboards, and conventional web cams. All USB devices require software "drivers" to operate. Device drivers are special pieces of software that run on your computer and communicate with external pieces of hardware on behalf of other programs. Once you have the driver for a particular piece of hardware, no other program needs to understand how to talk to that device. Your chat program doesn't need to know about your particular brand of web cam because your computer has a driver that makes it accessible to every program. Microsoft only intended the Kinect to work with the XBox 360 so it did not release drivers that let programs access the Kinect on normal personal computers. By funding the creation of an open source driver for the Kinect, Adafruit was attempting to make the Kinect accessible to all programs on every operating system.

After a Microsoft spokesperson reacted negatively to the idea of the bounty, Adafruit responded by increasing the total sum to $3,000. Within a week, Hector Martin claimed the bounty. Martin created the first version of the driver and initiated the Open Kinect project, the collaborative open source effort which rapidly formed to improve the driver and build other free resources on top of it.

The creation of an open source driver lead to an explosion of libraries that made the Kinect accessible in a variety of environments. Quickly thereafter the first demonstration projects that used these libraries began to emerge. Amongst the early exciting projects that demonstrated the possibilities of the Kinect was the work of Oliver Krey-

los, a computer researcher at UC Davis. Kreylos had previously worked extensively with various virtual reality and remote presence systems. The Kinect's 3D scanning capabilities fit neatly into his existing work and he very quickly demonstrated sophisticated applications that used the Kinect to reconstruct a full-color 3D scene including integrating animated 3D models and point-of-view controls. Kreylos' demonstrations caught the imagination of many people online and were even featured prominently in a New York Times article reporting on early Kinect "hacks".

A Word About the Word "Hack"

A word about the word "hack". When used to describe a technical project, "hack" has two distinct meanings. It is most commonly used amongst geeks as a form of endearment to indicate a clever idea that solves a difficult problem. For example, you might say "Wow, you managed to scan your cat using just a Kinect and a mirror? Nice hack!". This usage is not fully positive. It implies that the solution, though clever, might be a temporary stopgap measure inferior to permanently solving the problem. For example, you might say "I got the app to compile by copying the library into is dependencies folder. It's a hack but it works." In the popular imagination the word is connected with intentional violation of security systems for nefarious purposes. In that usage, "hack" would be used to refer to politically-motivated distributed denial of service attacks or social engineering attempts to steal credit card numbers rather than clever or creative technical solutions.

Since the release of the Open Kinect driver the word "hack" has become the default term to refer to most work based on the Kinect, especially in popular media coverage. The trouble with this usage is that it conflates the two definitions of "hack" that I described above. In addition to appreciating clever or creative uses of the Kinect, referring to them as "hacks" implies that they involve nefarious or illicit use of Microsoft's technology. The Open Kinect drivers do not allow programmers to interfere with the XBox in anyway, to cheat at games, or otherwise violate the security of Microsoft's system. In fact after the initial release of the Open Kinect drivers Microsoft themselves made public announcements explaining that no "hacking" had taken place and that, while they encouraged players to use the Kinect with their XBox for "the best possible experience", they would not interfere with the open source effort. So, while many Kinect applications are "hacks" in the sense that they are creative and clever, none are "hacks" in the more popular sense of nefarious or illicit. Therefore, I think it is better not to refer to applications that use the Kinect as "hacks" at all. In addition to avoiding confusion, since this is a book designed to teach you some of the fundamental programming concepts you'll need to work with the Kinect, we don't want our applications to be "hacks" in the sense of badly designed or temporary. They're lessons and starting points, not "hacks".

Not long thereafter, this work started to trickle down from computer researchers to students and others in the creative coding community. Dan Shiffman, a professor at NYU's Interactive Telecommunications Program, built on the work of the Open Kinect project to create a library for working with the Kinect in Processing, a toolkit for soft-

ware sketching that's used to teach the fundamentals of computer programming to artists and designers.

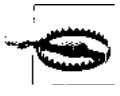
In response to all of this interest, PrimeSense released their software for working with the Kinect. In addition to drivers that allowed programmers to access the Kinect's depth information, PrimeSense included more sophisticated software that would process the raw depth image to detect users and locate the position of their joints in three dimensions. They called their system, OpenNI, for "Natural Interaction". OpenNI represented a major advance in the capabilities available to the enthusiast community working with the Kinect. For the first time, the user data that made the Kinect such a great tool for building interactive projects became available to creative coding projects. While the Open Kinect project spurred interest in the Kinect and created many of the applications that demonstrated the creative possibilities of the device, OpenNI's user data opened a whole new set of opportunities. The user data provided by OpenNI gives an application accurate information on the position of each user's joints (head, shoulders, elbows, wrists, chest, hips, knees, ankles, and feet) at all times while they're using the application. This information is the holy grail for interactive applications. If you want your users to be able to control something in your application using hand gestures or dance moves, or their position within the room, by far the best data to have is the precise position of their hands, hips, and feet. While the Open Kinect project may eventually be able to provide this data and while Microsoft's SDK provides it for developers working on Windows, at this time, OpenNI is the best option for programmers who want to work with this user data (in addition to the depth data as well) in their choice of platform and programming language.

So now, at the end of the history lesson, we come to the key issue for this book. The reason we care about the many options for working with the Kinect and the diverse history that lead to them is that each of them offers a different set of affordances for the programmer trying to learn to work with the Kinect.

For example, the Open Kinect drivers provide access to the Kinect's servos while OpenNI's do not. Another advantage of Open Kinect is its software license. The contributors to the Open Kinect project released their drivers under a fully open source license, specifically a dual Apache 2.0/GPL 2.0 license. Without getting into the particulars of this license choice, this means that you can use Open Kinect code in your own commercial and open source projects without having to pay a license to anyone or worrying about the intellectual property in it belonging to anyone else. For the full details about the particulars of this license choice see the policy page on Open Kinect's wiki.

The situation with OpenNI, on the other hand, is more complex. PrimeSense has provided two separate pieces of software that are useful to us. First is the OpenNI framework. This includes the drivers for accessing the basic depth data from the Kinect. OpenNI is licensed under the LGPL, a license similar in spirit to Open Kinect's license. (For more about this license see its page on the GNU site.)However one of OpenNI's most exciting features is its user tracking. This is the feature I discussed above where

an algorithm processes the depth image to determine the position of all of the joints of any users within the camera's range. This feature is not covered by OpenNI LGPL license. Instead it (along with many other of OpenNI's more advanced capabilities) is provided by an external module, called NITE. NITE is not available under an open source license. It is a commercial product belonging to PrimeSense. Its source code is not available online. However, PrimeSense does provide a royalty-free license that you can use to make projects that use NITE with OpenNI, but it is not currently clear if you can use this license to produce commercial projects, even though using it for education purposes like working through the examples of this book is clearly allowed.



To learn more about the subtleties and complexities involved in open source licensing, consult Understanding Open Source and Free Software Licensing by Andrew St. Laurent.

OpenNI is governed by a consortium of companies led by PrimeSense that includes robotics research lab Willow Garage as well as the computer manufacturer Asus. OpenNI is designed to work not just with the Kinect but with other depth cameras. It is already compatible with PrimeSense's reference camera as well as the upcoming Asus Xtion camera which I mentioned earlier. This is a major advantage because it means code we write using OpenNI to work with the Kinect will continue to work with newer depth cameras as they are released preventing us from needing to rewrite our applications depending on what camera we want to use.

All of these factors add up to a difficult decision about what platform to use. On the one hand, Open Kinect's clear license and the open and friendly community that surrounds the project is very attractive. The speed and enthusiasm with which Open Kinect evolved from Adafruit's bounty to a well-organized and creative community is one of the high points of the recent history of open source software. On the other hand, at this point in time OpenNI offers some compelling technical advantages, most importantly the ability to work with the user tracking data. Since this feature is maybe the Kinect's single biggest selling point for interactive applications, it seems critical to cover it in this book. Further, for a book author, the possibility that the code examples you use will stay functional for a longer time period is an attractive proposition. By their very nature, technical books like this one begin to go stale the moment they are written. Hence anything that keeps the information fresh and useful to the reader is a major benefit. In this context the possibility that code written with OpenNI will still work on next year's model of the Kinect, or even a competing depth camera, is hard to pass up.

Taking all of these factors into account, I chose to use OpenNI as the basis of the code in this book. Thankfully, one thing that both OpenNI and Open Kinect have in common is a good Processing library that works with them. I mentioned above that Dan Shiffman created a Processing library on top of Open Kinect soon after its drivers were first released. Max Rheiner, an artist and lecturer at Zurich University, has created a similar Processing library that works with OpenNI. Rheiner's library is called Sim-

pleOpenNI and it supports many of OpenNI's more advanced features. SimpleOpenNI comes with an installer that makes it straightforward to install all of the OpenNI code and modules that you need to get started. I'll walk you through installing it and using it starting at the beginning of Chapter 2. We'll spend much of that chapter and the following two learning all of the functions that SimpleOpenNI provides for working with the Kinect, from simple access to the depth image to building a three dimensional scene to tracking users.

Kinect Artists

This book will introduce you to the Kinect in a number of ways. We've already looked at how the Kinect works and how it came into existence. In the next few chapters, we'll cover the technical and programming basics and we'll build a raft of example applications. In the last three chapters we'll get into some of the application areas opened up the Kinect, covering the basics of what it takes to work in those fascinating fields.

However, before we dive into all of that, I wanted to give you a sense of some of the work that has gone before you. Ever since its release, the Kinect has been used by a diverse set of artists and technologists to produce a wide range of interesting projects. Many of these practitioners had been working in related fields for years before the release of the Kinect and were able to rapidly integrate it into their work. A few others have come to it from other fields and explored how the possibilities it introduces could transform their own work. All of them have demonstrated their own portion of the range of creative and technical possibilities opened up by this new technology. Together, they've created a community of people working with the Kinect that can inspire and inform you as you begin working with the Kinect yourself.

In this section, I will introduce the work of seven of these practitioners: Kyle McDonald, Robert Hodgin, Nicholas Burrus, Lady Ada, Oliver Kreylos, Elliot Woods, and blablabLAB. I'll provide some brief background on their work and then the text of an interview that I performed with each of them. In these interviews, I asked each of them to discuss how the came to integrate the Kinect into their own work, how their background transformed how they saw and what they wanted to do with the Kinect, how they work with it and think about it, and what they're excited about doing with the Kinect and related technologies in the future.

I hope that reading these interviews will give you ideas for your own work and also make you feel like you yourself could become a part of the thriving collaborative community that's formed up around this amazing piece of technology.

Kyle McDonald

Kyle McDonald is an artist, technologist, and teacher living in New York city. He is a core contributor to OpenFrameworks, a creative coding framework in the C++ programming language. Since 2009 he has conducted extensive work towards democratizing realtime

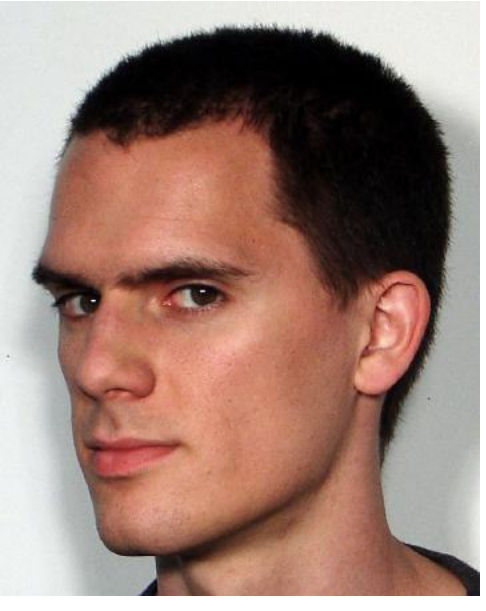


Figure 1-3. Artist and technologist Kyle McDonald has been building DIY 3D scanners for years.

3D scanning beginning by producing his own DIY structured-light scanner using a projector and a web cam. He has worked widely with the Kinect since its release including as artist-in-residence at Makerbot where he put together a complete toolkit for creating 3D scans with the Kinect and printing them on the Makerbot. Artistically, his work frequently explores ideas of public performance and extremely long duration. In his 2009 "keytweeter" performance, he broadcast every keystroke he entered into his personal computer for a full year via (Twitter. In 2010 he created "The Janus Machine" with fellow OpenFrameworks contributors Zach Lieberman and Theo Watson. "The Janus Machine" is a 3D photo booth that turns the user into a two-faced Janus by pairing them with a structured light scan of their own face. Watch video of The Janus Machine.

How did you first get interested in 3D scanning? What drew you to it as a technique and how did you first set out to learn about it?

At the beginning of 2009 I saw some work called "Body/Traces" from artist Sophie Kahn and dancer/choreographer Lisa Parra. They were using the DAVID 3D scanning software, with a Lego-driven line laser and a webcam, to scan a dancer. It took about one minute to take a single scan, which meant that one second of stop-motion 3D-scanned video could take ten to twenty minutes to shoot. I was enamored with the quality of the scans, and immediately started dreaming about the possibilities of real-time capture for interaction. So I began working on a practical problem: making a faster 3D scanner. My first scanner was based on a simple intuition about using a projector to display gray codes instead of a laser with its single line. That brought the time down

from a few minutes to a few seconds. Then I discovered structured light research while digging around Wikipedia, started reading academic papers, and emailing every researcher who would answer my naive questions. This was months after Radiohead's "House of Cards" video was released, so I should have known about structured light already. But the tagline for that video was "made without cameras", so I assumed it was built on high end technology that I didn't have access to.

What interactive possibilities do 3D scanning techniques open up? How are the affordances they offer fundamentally different from other forms of digital imaging?

Realtime 3D scanning is fundamentally different from other kinds of imaging in that it focuses on geometry rather than light; or shape rather than texture. This can make a huge difference when trying to accomplish something as simple as presence detection. Having access to information like position, surface normals, or depth edges opens up possibilities for understanding a scene in ways that are otherwise practically impossible. For example, knowing which direction someone's palm is facing can easily be determined by a lenient plane fitting algorithm, or a simple blur across the surface normals. This kind of information has made a huge difference for skeleton tracking research, as our shape is much less variable than the way we look.

Much of the development that led to the Kinect came out of surveillance and security technology. Do you think this provenance makes privacy and surveillance natural themes for creative work that uses the Kinect? How can art and interactive design inform our understanding of these kinds of issues?

Whenever you have a computer making judgements about a scene based on camera input, there is the potential to engage with surveillance as a theme. But if you look at the first demos people made with the Kinect, you'll find an overwhelming majority explore the creative potential, and don't address surveillance. I think artists have just as much a responsibility to comment directly on the social context of a technology as they have a responsibility to create their own context. By addressing the creative potential of the Kinect, artists are simultaneously critiquing the social context of the device: they're rejecting the original application and appropriating the technology, working towards the future they want to see.

Much of your technical work with the Kinect has centered on using it as a 3D scanner for digital fabrication. What are the fundamental challenges involved in using the Kinect in this way? What role do you see for 3D scanners in the future of desktop fabrication?

The Kinect was built for skeleton tracking, not for 3D scanning. As a 3D scanner, it has problems with noise, holes, accuracy, and scale. And in the best case, a depth image is still just a single surface. If you want something that's printable, you need to scan it from all sides and reconstruct a single closed volume. Combining, cleaning, and simplifying the data from a Kinect for 3D printing can be a lot of work. There are other scanners that mostly solve these problems, and I can imagine combining them with

desktop printing into something like a 3D "photocopier" or point-and-shoot "Polaroid" replicator.

As part of your artist residency at MakerBot you disassembled a Kinect in order to look at its internals. What did you learn in this process that surprised you? Do you think we'll see open source versions of this hardware soon?

I was surprised with how big it is! The whole space inside the Kinect is really filled up with electronics. There's even a fan on one side, which I've only ever heard turn on once: when I removed the Peltier cooler from the IR projector. The tolerance is also really tight: for all the times I've dropped my Kinect, I'm surprised it hasn't gone out of alignment. If you unscrew the infrared projector or the infrared camera and just wiggle them a little bit, you can see that the depth image slowly disappears from the sides of the frame because the chip can't decode it anymore.

I don't expect to see open source versions of the Kinect hardware any time soon, mainly because of the patents surrounding the technique. That said, I'd love to see a software implementation of the decoding algorithm that normally runs on the Kinect. This would allow multiple cameras to capture the same pattern from multiple angles, increasing the resolution and accuracy of the scan.

You've done a lot of interesting work making advanced computer vision research accessible to creative coding environments like OpenFrameworks. How do you keep up with current research? Where do you look to find new work and how do you overcome the sometimes steep learning curve required to read papers in this area? More generally, how do you think research can and should inform creative coding as a whole?

The easiest way to keep up with the current research is to work on impossible projects. When I get to the edge of what I think is possible, and then go a little further, I discover research from people much smarter than myself who have been thinking about the same problem. The papers can be tough at first, but the more you read, the more you realize they're just full of idiosyncrasies. For example, a lot of image processing papers like to talk about images as continuous when in practice you're always dealing with discrete pixels. Or they'll write huge double summations with lots of subscripts just to be super clear about what kind of blur they're doing. Or they'll use unfamiliar notation for talking about something simple like the distance between two points. As you relate more of these idiosyncrasies to the operations you're already familiar with, the papers become less opaque.

Artists regularly take advantage of current research in order to solve technical problems that come up in the creation of their work. But I feel that it's also important to engage the research on its own terms: try implementing their ideas, tweaking their work, understanding their perspective. It's a kind of political involvement, and it's not for everyone. But if you don't address the algorithms and ideas directly, your work will be governed by their creators' intentions.

If you could do one thing with the Kinect (or a future depth camera) that seems impossible now what would that be?

I want to scan and visualize breaking waves. I grew up in San Diego, near the beach, and I've always felt that there's something incredibly powerful about ocean waves. They're strong, but ephemeral. Built from particles, they're more of a connotation of a form than a solid object. Each one is unique and unpredictable. I think it also represents a sort of technical impossibility in my mind: there's no way to project onto it, and the massive discontinuities preclude most stereo approaches. But if studying 3D scanning has taught me anything, it's that there's always some trick to getting the data you want that comes from the place you least expect.

Robert Hodgin

Robert Hodgin is an artist and programmer working in San Francisco. He was one of the founders of the Barbarian Group, a digital marketing and design agency in New York. Hodgin has been a prominent member of the creative coding community since before that community had a name, creating groundbreaking work in Flash, Processing, and the C++ framework, Cinder. He is known for creating beautiful visual experiences using simulations of natural forces and environments. His work tends to have a high degree of visual polish that makes sophisticated use of advanced features of graphical programming techniques such as OpenGL and GLSL shaders. He has produced visuals to accompany the live performances of such well-known musicians as Aphex Twin, Peter Gabriel, and Zoe Keating. Soon after the release of the Kinect, Hodgin released Body Dysmorphia, an interactive application that used the depth data from the Kinect to distort the user's body interactively in realtime to make it appear fat and bloated or thin and drawn. Body Dysmorphia was one of the first applications of the Kinect to connect depth imagery with a vivid artist subject and to produce results that had a high degree of visual polish. Hodgin is currently Creative Director at Bloom, a San Francisco startup working to combine data visualization with game design to create tools for visual discovery.

How did you first hear about the Kinect/Project Natal? Why did it capture your interest?

I first heard about the Kinect when it was making the rounds during E3 in 2009. I am a bit of a video game fan so I try to keep up to date with the latest and greatest. When I saw the Kinect demos, I must admit the whole thing seemed ridiculous to me. The demos were rather odd and didn't make me want to play with the Kinect at all. I had the same reaction I had to the PlayStation Move.

Because of my pessimistic attitude, it is no surprise the Kinect did not capture my interest until right around the time people started posting open source drivers to allow my Mac laptop to get ahold of the depth information. That is when I decided to go buy one. Shortly after the Kinect CinderBlock was released, I hooked up the Kinect and started to play around with the depth data.

Prior to the Kinect's release, I had done some experimentation with augmenting live webcam feeds using hand-made depth maps. I set a camera on the roof of my old office and pointed it towards the Marina district of San Francisco. Since the camera was stationary, I was able to take a still image from the cam and trace out a rudimentary depth map. Using this 5 layered depth information, I could add smoke and particle effects to the view so that it looked like a couple buildings were on fire. The effect was fairly basic, but it helped me appreciate how depth information could be very useful for such effects. With it, I could make sure foreground buildings occluded the smoke and particle effects.

Once I bought the Kinect, I knew I wanted to explore these older concepts using proper depth data. That is when I realized how fantastic it was to have access to an extremely affordable, good quality depth camera. After a couple weeks, I had made the Body Dysmorphia project and it got a lot of good reactions. It is one of my favorite projects to date.

I still have not hooked the Kinect up to my XBox 360.

A lot of your work has involved creating screen-based visuals that are inspired by nature. Flocking and particle systems. Light effects and planet simulations. However, in Body Dysmorphia you used your actual body as an input. How does having input from the physical world alter the way you think about the project? Is part of what you're designing for here your interaction with the app or are you exclusively focused on the final visual result? Have you thought about installing Body Dysmorphia (or any of your other Kinect-based work) in a public exhibition venue or otherwise distributing it in order to share the interaction as well as the visual results?

For all of the Kinect projects I have created, I rarely got to work with a set idea of what I wanted to make. The Body Dysmorphia piece actually came about by accident. I was finding ways to turn the depth map into a normal map because I thought it might be interesting to use virtual lighting to augment the actual lighting in the room. I was getting annoyed by the depth shadow which appears on the left side of all the objects in the Kinect's view. I think this is just a result of the depth camera not being able to be situated in the exact same place as the infrared projection. I wanted to find ways to hide this depth shadow so I tried pushing the geometry out along the normals I had calculated to try and cover up the gap in the depth data. It was one of those surprising a-ha moments. It took me by surprise and instantly became the focus of the next month of experimentation. The effect was so lush and unexpected.

I am currently working on a Cinder and Kinect tutorial which I will be releasing soon. It will cover everything from just creating a point cloud from the depth data all the way up to recreating the Body Dysmorphia project. I am doing this for a couple reasons. I wanted to clean up the code and make it more presentable, but I was also approached by a couple different people who want to use Body Dysmorphia as installations for festivals. The code was definitely sloppy and hacked together so I broke it all down and

rebuilt it from scratch with extra emphasis on extensibility and ease of implementation. I look forward to releasing the code so that I can see what others can do with it.

You've worked with musicians such as Aphex Twin, Peter Gabriel, and Zoe Keating to produce visuals for live performance. These projects have ranged from pre-rendered to synced to audio to fully interactive. How do you approach music as a starting point for visual work? Do you think of this work as a visualization of the music or as part of a theatrical performance? Have you considered including music or sound as an accompaniment to your other visual work?

Music has played a really large role in my development as a creative coder. Early on, when I was just getting started with particle engines, I got bored with having to provide all the input. I was just making elaborate cursor trails. I wanted something more automated. So I entered my Perlin noise phase where I let Perlin noise control the behavior of the particles. But eventually, this became a bit frustrating because I wanted something more organic. That is when I tried using live audio data to affect the visuals. I usually listen to music while I code so if I could use microphone input for the parameterization of the project, I could have a much larger variety of behaviors. I didn't set out to make audio visualizers. I just wanted some robust realtime organic data to control my simulations.

When I need to make an audio visualization, as opposed to just using audio as an easy input data, I try to consider the components of the audio itself. If it is electronica and not too beat heavy and is completely lacking in vocals, the visualizations are very easy. It just works. But once you start adding string instruments or vocals or multi-layered drum tracks, the FFT analysis can very quickly start to look like random noise. It becomes very difficult to isolate beats or match vocals.

Zoe Keating and I have worked together a couple times. I love collaborating with her simply because I love her music. The couple times we have collaborated were for live performance. I did very little live audio analysis. String instruments can be challenging to analyze effectively. I ended up doing manually triggered effects so essentially, I played the visuals while she played the audio.

Peter Gabriel was different in that he was performing with a full orchestra. They practiced to a click track and I knew they would not be deviating from this. I was able to make a render that was influenced by the beat on the click track which made sure the final piece stayed in sync with the vocals.

Aphex Twin visuals were the easiest to make. I made an application using Cinder that took in Kinect data and created a handful of preset modes that could be triggered and modified by Aphex Twin's concert VJ. There was no audio input for that project simply because the deadline was very short.

I am very excited to start playing around with the OpenNI library. Getting to use the skeleton data is going to be a fantastic addition to the Kinect projects. I will easily be able to determine where the head and hands are in 3D space so I could have the audio

emanate out from these points in the form of a environment altering shockwave. The early tests have been very promising. Soon, I will post some test applications that will take advantage of this effect.

You've written eloquently about balancing rigorous technical learning with unstructured exploration and play in creative coding work. How do your technical ideas interact with your creative ones? Which comes first the chicken or the egg, the technical tools or the creative ideas?

Many people would be surprised to hear I have no idea what I am doing half the time. I just like to experiment. I like to play and I constantly wonder, "what if?" When I first started to learn to code in ActionScript, much of my coding process could be described as experimental trigonometry. I would make something then start slapping sine and cosine on all the variables to see what they did. I didn't know much about trig so my way of learning was to just start sticking random trig in my code haphazardly until I found something interesting.

The programming tools you use have evolved from Flash to Processing to OpenGL with C++ and shaders. What tools are you interested in starting with now? What do you have your eye on that's intriguing but you haven't played with yet? Also, do you ever think about jumping back to Flash or Processing for a project to see if those constraints might bring out something creative?

I still have an intense love for Processing. Without Processing, I am not sure I would have ever learned enough coding to feel comfortable attempting to learn C++. Casey and Ben and the rest of the Processing community have made something wonderful. I am forever in their debt for making something so approachable and easy to learn, but still so very powerful.

I am pretty set on sticking with Cinder. Thanks to the Cinder creator, Andrew Bell, I have developed a love for C++. I still hate it at times and curses can fly, but I love working with a coding language that has such a long history of refinement. It also helps that I am friends with Andrew so he holds my hand and guides me through the prickly bits. The more I use Cinder, the more in awe I am at the amount of work that went into it. I also plan on continuing to learn GLSL. I am a big fan of shaders.

If you could do one thing with the Kinect that seems impossible now what would that be?

I would love to be able to revisit the early project where I try and augment a webcam view of a cityscape. If the range of the Kinect could be extended for a couple miles, that would be fantastic. Alternately, I would also love to have a Kinect that is capable of doing high resolution scans of objects or faces. If the Kinect had an effective range of 1" to 2 miles, that would be perfect.

Elliot Woods

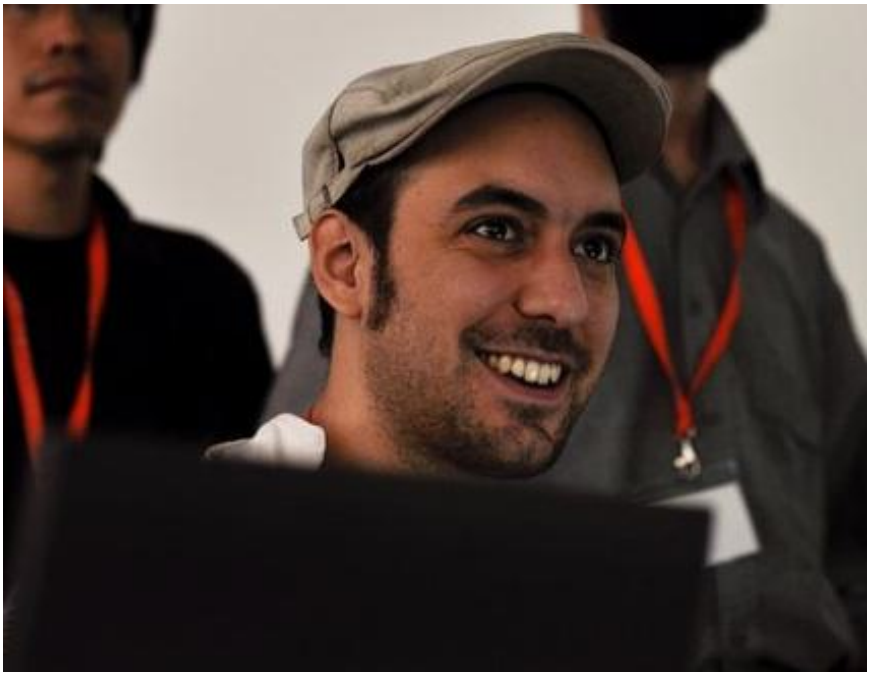


Figure 1-4. Programmer and artist Elliot Woods specializes in using the Kinect for projection mapping. Photo courtesy of Elliot Woods.

*Elliot Woods is a programmer, designer, artist, and physicist. He is co-director of Kimchi and Chips, an interdisciplinary art and design studio based in London and Seoul. Kimchi and Chips is known for innovative use of the Kinect as a tool for projection mapping, a technique that matches projected computer graphics to the spatial features of architecture and objects. Woods has pioneered the use of the Kinect to extend the art of projection mapping to moving dynamic objects. His Kinect Haidouken project used the Kinect to give users control of a projected light source. His *Lit Tree installation used the Kinect to do projection mapping onto the moving leaves of a tree. His work has been exhibited in Yokohama, Manchester, London, Berlin, Milan, and Arhus, Denmark.*

A lot of your work at Kimchi and Chips uses projection to bring objects and spaces to life. What is about projection that draws you to it as a medium?

We see projection as a great prototyping tool.

I think of a projector as a dense array of little colored spotlights laid out in a grid.

Its invention is an artifact of consumer/business/education technology, but it is in itself quite a peculiar machine capable of visually controlling its surroundings. Its general use also means that there's plenty of fantastic affordable stuff that you can plug a projector into (powerful graphics card, video hardware, games consoles and all that).

Our eyes (our cameras) are obviously very important to us. Most creatures have them, and for decades computer scientists have been dedicated to giving useful eyes to our artificial creatures. They give us a sense of the world around us and are arguably the most visceral. Cameras are our proxy eyes onto the real and virtual worlds, through streaming video, films, television, photos, video conferencing, but also now computer vision systems.

The projector is the antithesis to the camera. It's pretty much exactly the same thing, but acting in an inverse way to the camera. It sends light down millions of little beams, where a camera would collect light along millions of little beams. Through this it can (given enough brightness) affect a scene, as much as a camera can sense a scene. If the camera is the sensor, the projector is the actuator.

Since projectors are so closely matched with cameras, I'm surprised that there aren't known biological instances of the projector. Why wouldn't a creature want to be able to project and see the same way it can speak and listen? An octopus can create highly convincing visual color images across its skin, demonstrating an evolutionary ability to generate images. But if you had a built-in light engine, what could you do if you had as much control over our local visual environment as a human eye could see?

Since we're cursed with the disability of having no built-in light engine, we have to rationally define what we'd like to achieve so that a computer and an electronic projector can do it for us, this is why I work with projectors.

How did you first start working with the Kinect? What made you excited about using it in your projects?

I heard about the Kinect (at the time Project Natal) around 2009. I wasn't very much believing in it at the time, it sounded too unreal.

I'd been playing with the Wiimote and PS3eyes as HCI devices, and knew that if Project Natal ever came out, then the first thing to do would be to get that depth data out and into open development environments, and was suggesting to friends that we start a bounty for the hack. Luckily the Adafruit/JCL bounty came out, and the closed doors got busted ajar.

Kinect to me is about giving computers a 3D understanding of the scene in a cheap and easy to process way. The OpenNI/Kinect SDK options obviously allow for automatic understanding of people, but I'm more interested in getting a computer to know as much about its surroundings as it knows about the virtual worlds inside its memory.

A lot of the most prominent uses of projection mapping have taken place outside in public spaces. You tend to use it in more intimate interior spaces where it becomes interactive. How are the challenges different when making a projection interactive? What are the advantages of working in this more intimate setting?

There's a few reasons we keep a lot of our works *indoors*.

To make something big, you quickly lose control of your environment, and need a lot of cash to keep on top of things. The cash element also comes in because big public works attract sponsors/commissions from brands, which makes working in that space quite competitive. We don't want to become another *projection mapping* company who'll push your logo onto a 5 story building. The whole field's becoming a little irresponsible and messy, we missed out on the innovation period for projecting onto buildings, and hope that advertising agencies/advertising rhetoric might back out of the field a bit so people can start working on something new.

We want to carry our identity into large scale outdoor works, which includes our research into projecting onto trees. We think this medium carries a lot of challenge and has surprisingly beautiful results, but most importantly is an untouched frontier to explore.

What programming languages and frameworks do you use? Has that changed or evolved over time?

If you fancy the long story...

I started out in Basic when I was about 7, copying code out the back of books. Sometimes I'd get my dad to do the typing for me, I was totally addicted to the idea of being part of the making of a piece of software. It took a while to grasp the basics past PRINT A$ and INPUT A$, and I cant remember much apart from the line numbers and getting quickly bogged down when creating repeatable logic.

After that, QuickBasic, then Visual Basic where I made my RCCI (Resistor Colour Code Interpreter) at the age of about 11. I was so proud of myself, I spent days trying to show off to my family. I'd made something that was useful, visible and worked. This was the first real feeling of closure, or *deployment*. It's an emotion I still feel today when a project is complete.

At the end of college I was getting more into Visual Basic and started developing games, which felt like a decent career move at the time. Then into University I worked on mathematically driven interfaces and graphical elements for websites (now called generative graphics). Mid-university I started developing visual installations with projectors and Max/MSP with a group of friends, then I discovered VVVV which blew my world.

Onto the short story...

I love to use VVVV. VVVV is a massively underused/overpowered toolkit for media development. Whilst the VVVV plugin system was still in its infancy (and I'd just gotten a Mac), I started developing with openFrameworks in parallel. This development was much slower than with VVVV, but more flexible, and importantly for a new Mac owner, platform independent.

Now that the VVVV plugin system has matured, it's capable of many of the things openFrameworks/Cinder can achieve, whilst retaining the runtime development paradigm which makes it so strong/quick to use/learn.

What was your background before starting Kimchi and Chips? With one of you based in London and one in Seoul how did you meet? How do you collaborate remotely on projects that frequently have a physical or site-specific component to them?

I studied Physics at Manchester, Mimi was running a successful design firm in Seoul. Then I started working in the digital media design field (multitouch, projection mapping, etc.) and Mimi started studying Interaction Design in Copenhagen Institute of Interaction Design.

We met at a conference in Aarhus, Denmark. She needed some help with a friend's project and saw that I might be a good person to ask. After that we took on more projects and founded the company. Mimi moved back to Seoul after studying to be with her family, so the company moved with her.

Working apart, we keep in regular contact but also avoid getting too involved with what the other is doing. Often I'm doing a lot of coding and hardware design, whilst Mimi works on interaction scenarios, motion design and communicating with the Korean clients.

How does the Kinect fit into your work? What new possibilities does it open up? Do you think it will be important to your work in the future?

Kinect gives us geometry information about a scene. We've begun to show what we'd love to use that for. Hopefully we'll get enough time to show all of our ideas.

What was the seed of the Kinct Haidouken project? What inspired it? How did it relate to what you'd been working on before?

I knew I had to try getting a projector working with the geometry from the Kinect, to see what the results were like. A light source became a good way to demonstrate how the projection could be sensitive to the scene, and *create* something meaningful to the viewer. After waving the virtual light around with my mouse, I wanted to hold it in my hand. then once I had it in my hand I wanted to throw it. Then I realised what I'd made.

What language and framework did you use for it?

It was C#/OpenNI.Net for the tracking bits and getting the data into a useful form to shift onto the GPU. Then we go into VVVV, where there's a chain of shaders that perform the normal calculations/filtering/lighting and also some patching for the throw dynamics (and a little C# plugin for the thrown particles).

Will you talk about some of the challenges of matching the 3D data from the Kinect with the point of view of the projector? How accurately did you have to measure the position of the projector relative to the Kinect? Did you have to take into account lens distortion or any other details like that?

This was all really easy. We use a system me and a friend made called Padé projection mapping. The calibration takes about 1 minute, and just involves clicking on points on the screen. No measuring tape needed!

Figure 1-5. Elliot Woods projecting onto himself by using the Kinect to build a 3D model of the space. Photo courtesy of Elliot Woods.

Where will you take this technique next?

We're looking into trying this technique out with a stage/audience performance. We want people to experience it, and we want to show what it really means to have all these controllable surfaces in your surroundings.

If you could do one thing with the Kinect that seems impossible now what would that be?

I'd be really interested if we could get a really low latency/high framerate (>100fps) 3D scanner with zero kernel size (so an object of 1 pixel size can be scanned) that connects over USB. That'd be incredible, and something i'm finding myself fighting to achieve with projectors and structured light.

blablabLAB

blablabLAB is an art and design collective based in Barcelona. The describe themselves as "a structure for transdisciplinary collaboration. It imagines strategies and creates tools to make society face its complex reality (urban, technological, alienated, hyper-consumerist). It works without preset formats nor media and following an extropianist philosophy, approaching the knowledge generation, property and diffusion of it, very close to the

Figure 1-6. Two members of blablabLAB, an art and design collective that produced souvenir 3D prints of people from Kinect scans on the streets of Barcelona. Photo courtesy of blablabLAB.

DIY principles." Their work explores the impact of technology on public life, from urban space to food. In January of 2011 they produced an installation in Barcelona called "Be Your Own Souvenir". The installation offered passersby the opportunity to have their bodies scanned and then receive small plastic figurines 3D printed from the scans on the spot. "Be Your Own Souvenir" won a 2011 Prix Arts at the Ars Electronica festival. blablabLAB expresses a highly pragmatic and hybrid approach to using technology for cultural work, exploring a wide variety of platforms and programming languages.

Was "Be Your Own Souvenir" your first project using the Kinect? What made you excited to work with the Kinect?

We presented the project to a contest in summer 2010. By then the Kinect was called Project Natal, nobody knew much and rumors were the camera was based on the Time of Flight principle. Our project was selected and the exhibition was scheduled for January 2011.

When we first thought about scanning people we knew it could be somehow done, since we had seen Kyle McDonald's app for Processing. We researched a bit and found he had ported the code to openFrameworks.

We were working with some previous implementation of a structured light scanner from Brown University when our friend Takahiro pointed out the new trend. Everybody out there was developing and doing experiments with the Kinect! We went to the shop and got Theo's first ofxKinect to work. Suddenly we gained a lot of time, and the deadline was achievable.

So, it was the first and till now, the only one project we have done with a Kinect. Maybe the last one. Our focus is not the technology.

How did people react to the prospect of being scanned? Was everyone excited about it? Was anyone nervous? How did you explain to them what was going to happen?

We showed a video in-place to try to explain what we were doing. It was all about daring to stand still for 4 minutes in front of a crowd, so most of the people were nervous but some felt like movie-stars. As expected, the reward, the figurine, was in many cases strong enough to let the shyness aside.

Before the scan we made them sign a contract authorizing us to capture and store their volumetric data. We wanted to raise their awareness about their private data being collected. Something that already happens ubiquitously in our everyday's life with other kinds of data, and that may happen in the near future with these new kind of technologies.

Did you show people what their scan was going to look like while they were posing? Did that make a difference in how they posed?

No. The point was to replicate the real street artist, so no visualization was planned in the beginning, neither for the statue nor the public.

We noticed though that many people really enjoyed viewing the 3D data on our screens, so we started displaying our working monitor, but because of the physical configuration we had, still, the statue couldn't see him/herself. This lead to a funny situation where the statue would act as a traditional one, while the crowd was enjoying a digital spectacle.

Why did you do this project in a public space?

The project is actually about the public space: about the users of La Rambla (the main pedestrian street in Barcelona), how they interact with each other and the city.

Probably there haven't been many Kinect projects related to the public sphere yet (privacy issues, pervasiveness, etc.), but the technology seems to be in the focus of the marketing and commercial sectors. This is something we wanted to deal with, since we believe unfortunately a pervasion of the public space with this kind of hidden technologies, difficult to notice, may happen soon.

How many people worked on "Be Your Own Souvenir"? What is your background and experience?

We were three people coming from different backgrounds like architecture, electronic engineering, computer science, urban planning, design, cinema, or molecular cuisine.

Tell me a little bit about your workflow. How did you create a 3D file from the Kinect data? What processing did you have to do it to get a file that could print in good quality in a reasonable time? What software did you use to control the printer? What was the biggest obstacle you faced?

All the software used in this project is free and open. Custom software has been developed using openFrameworks and openKinect in order to produce a tunable full 360 degree point cloud. We avoid the occlusions using mechanic shutters controlled by Arduino. Using a MIDI controller, the three different input point-clouds (3 Kinects) can be adjusted in space and resolution. The resulting combined point cloud is processed by Meshlab to produce a mesh reconstruction. Skeinforge takes the mesh, previously cleaned up and tuned through Blender, and outputs a gcode file, which can feed a CNC machine.

The main problem we faced, and we are still facing, is that the time and quality is different for the scanning part and the printing part. You can scan in less than a second but you need much more to print, while the scan resolution is pretty low, with the printer you can get awesome details. Tweaking the printer to get the equilibrium between fast and reasonable print quality brought us to a 10 minutes average per figurine.

Besides this tradeoff, the mesh reconstruction process is the chain-step most likely to be improved. Instead of a one-step reconstruction operation using the three point-clouds at the same time, a two-step process using high-level information to produce X meshes (X being the number of Kinects), to further zipper them would be a nice way to start.

What gear did you bring with you to do this project? How did you approach setting everything up on a public street? Was it hard to get permission to use public space?

We built ourselves three translucent plastic boxes to keep the cameras protected and shelter the screens, and be able to display vinyl labeling with rear lighting. One of the boxes also hosted the computer we use for the scans (i7 desktop, screen, etc). The printer was placed inside the museum a few meters away, since one of the goals of the project was also to approach people to this art center.

Thankfully, we were supervised by a production team that dealt with all the permissions. Barcelona is a really hard place when it comes to public space bureaucracy, unless you are Woody Allen...

How did people react to the quality of the prints? Could they recognize themselves?

They could recognize themselves only for the pose, by the complexity or a characteristic piece of clothing. A few reproached us about it but most of them were amazed with their own souvenir.

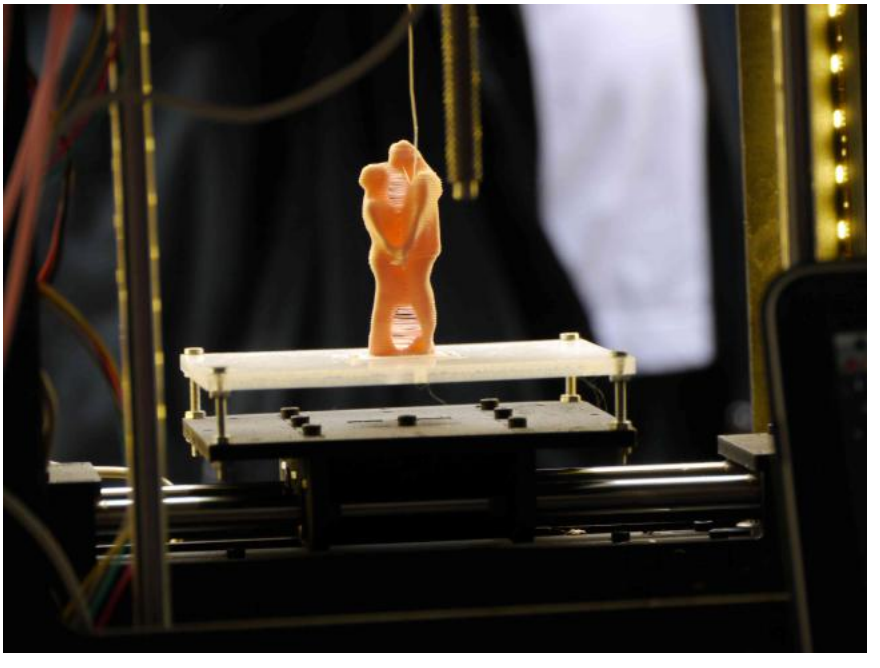


Figure 1-7. A print produced by blablabLAB's "Be Your Own Souvenir" project based on a scan of two lovers holding hands. Photo courtesy of blablabLAB.

It's pretty funny to see yourself without color, even with high resolution scanners, so lot of people were actually kind of impressed. Kids were actually the most shocked.

What possibilities do you see for using the Kinect in future projects?

We've been thinking about the possibilities of combining Kinect and realtime mapping with some sort of feature recognition. The Kinect has been designed for human gesture tracking, so pushing this boundary is always exciting. Maybe in the near future higher specification cameras will be available and the horizon will expand.

If you could do one thing with the Kinect that seems impossible now what would that be?

For us would be synchronize them with software to be able to connect more than one to avoid IR occlusions, but we don't even know whether than can be done with the Windows SDK.

Nicolas Burrus

Nicolas Burris is a postdoctoral scholar in computer vision at Carlos III University in Madrid. He works on building computer vision systems for robotics, from coordinating the movements of dextrous robotic hands to studying the use of 3D cameras for airplanes

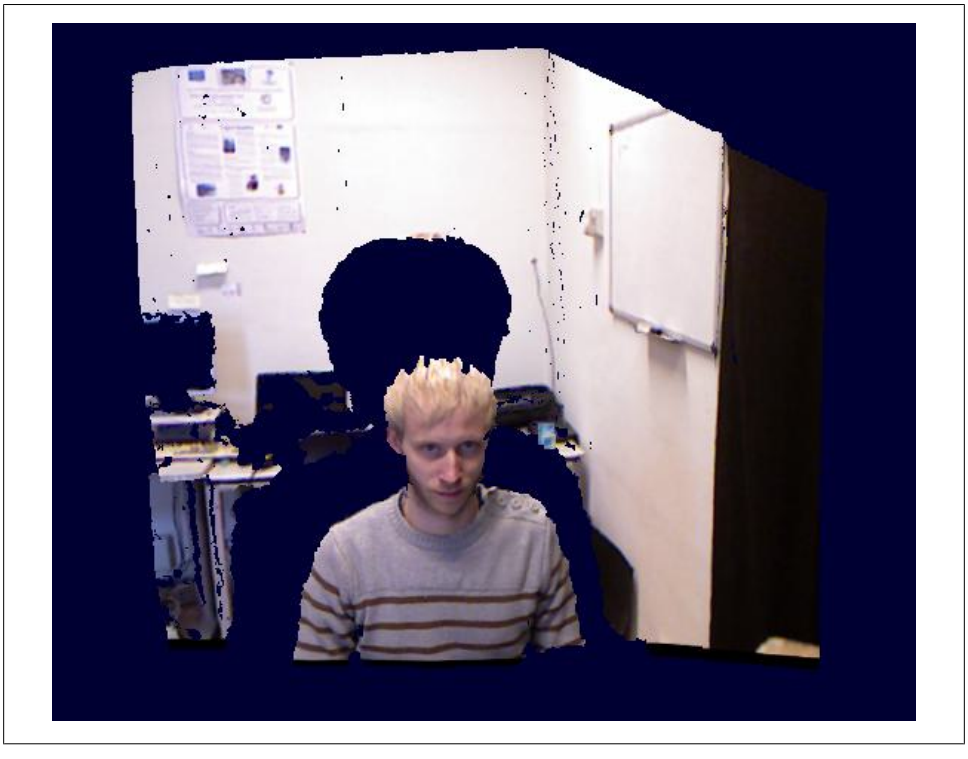


Figure 1-8. Researcher Nicolas Burrus captured in a 3D point cloud using his open source RGBDemo software. Burrus' work makes advanced computer vision algorithms accessible to beginner Kinect hackers. Photo courtesy of Nicolas Burrus.

for Airbus. Burrus created Kinect RGBDemo, a suite of applications that uses the Kinect to demonstrate advanced computer vision algorithms. These include the ability to construct a complete 3D model of a room by moving a Kinect around it, an application that calibrates the Kinect's color and depth cameras, and an application that can detect objects sitting on a flat surface such as a table. RGBDemo uses the OpenKinect driver to access the Kinect so its applications are completely open source and accessible to everyone.

How would you describe the overall mission of your research? What drew you to computer vision work in the first place?

I have been working on statistical image processing during my PhD thesis, and then decided to apply these tools to computer vision to give robots the ability to see, interpret, and interact with their environment using visual information, pretty much as humans do. The visual data captured by current sensors is very rich, but their automatic analysis, i.e. getting useful information out of the raw pixels is still a major and exciting challenge.

One portion of your work before the release of the Kinect was focused on vision systems for robotics, particularly robots that would interact with their environment with artificial hands. How did the release of the Kinect change that work? Are you using the Kinect in your current robotics research or are other 3D capture technologies still more effective in the lab? What do you think the future holds for this kind of research as 3D capture technologies like the Kinect become more widespread?

We have been working with stereo cameras, lasers and even some depth cameras in our lab for a long time before the release of the Kinect. All these devices were already able to output 3D data of the robot environment. The revolution brought by the Kinect is the combination of the best features of these devices. Stereo cameras can be cheap but have a worse depth estimation and require complex processing to compute depth estimations, especially in homogeneous areas. Lasers are precise but are slow to compute a full scan, and are very expensive. Earlier depth cameras were already attractive but had a significantly worse depth precision, and ours was about 50 times more expensive!

The Kinect camera is the first device capable of delivering real-time, precise, and dense 3D scans for a very cheap price. Its wide availability is also a chance for academics to get more impact, since their results can directly be used and repeated by others. So we progressively switched to Kinect for the perception of our robots, when possible.

Other technologies still have a future though. The Kinect cannot work outdoors, and is limited to short range (about 5-6 meters). Its precision also rapidly decreases with the distance, which is not the case for the existing professional devices. But its impressive success will definitely open the door to new generations of cheap 3D cameras, hopefully more compact and even more precise.

Your RGBDemo project is designed to introduce people to all the things that can be done with the data from a depth camera like the Kinect. What advice would you give to a programmer who wanted to explore those possibilities further? What skills should they be studying? What fundamentals are most important to master?

I think algorithmic skills are very important, since 3D means a lot of data, and thus naive implementations usually result in unusable programs. A good mathematical background in geometry and linear algebra also helps. I would recommend to acquire and get a practical knowledge of these fundamentals by trying to program small 3D video games.

The area of GPU programming is also becoming more popular and can offer huge performance increase, and is perfectly adapted to the processing of such big quantities of data.

RGBDemo brings together algorithms and techniques from a number of different research efforts such as the Point Cloud Library. What advice would you give

to a programmer who was excited about the Kinect and wanted to learn more about the principles of computer vision and to learn more about the current state of the research?

Research works with communities, so the hardest step is to find an entry point. Then, following the mailing list of major software in the field and having a look at the demos and papers of the major conferences (e.g. CVPR, ICCV) is a good way to keep up to date. For example, Willow Garage is definitely one of the main actors of the robotics community, and following the development of its vision software libraries such as PCL or OpenCV is very useful.

What have been some of your favorite projects to come out of the "Kinect hacking" community? Are you inspired at all by the interface and artistic applications of the technology in your research?

I have seen a lot of very inspiring and impressive hacks or projects. This is definitely an additional motivation to bring the latest research results to a wider audience, that will find new ways to apply it, be it for fun, artistic or medical applications. If I had to mention a few projects, I would probably include the Intel RGBD Project, that did a lot on automatic mapping and object model acquisition, people from Konstanz university that developed a system to guide blind people, the Kinemmings game that proposed an innovative way to interact with games, the many hacks that used the Kinect to create virtual instruments, such as the Kinect Anywhere virtual piano, and all these nice visual effects we have seen on top of the depth images, such as the Dirty Pearls clip.

What are your plans for continuing your work with the Kinect going forward? Do you plan to continue its use in research or in commercial applications?

Both! I will keep working in the academic world and adding new demos to RGBDemo as part of that work. With the big interest that raised the Kinect for computer vision in the consumer market, we also have plans to help transferring latest developments from academics to end users through a new consultancy company.

Are there other computer vision technologies that are currently in the research phase that you can imagine having a similar impact to the Kinect if made equally accessible? What's the "next Kinect"? What will be the state of the art in this field in 5 years?

I think the next phase for the consumer market is in the convergence of devices that output 3D data, devices that display 3D data, and software able to process and analyze all this data. I expect mobile devices to embed 3D screens and cameras in the near future. Interactions with computers will also change significantly with the widespread use of multitouch and touch-free interfaces. With cameras capable of detecting objects and people in a scene, our homes might also change considerably. Virtual worlds and crowds could also take a new dimension with animated and realistic avatars. Exciting times are waiting for us!

Oliver Kreylos

Oliver Kreylos is a computer scientist at UC Davis. His work focuses on using virtual reality to improve interfaces for data visualization in scientific research. Immediately after the launch of the Kinect in November 2010, Kreylos demonstrated an application that used the Kinect to create a complete live full-color 3D reconstruction of his room including integrating animated digital characters. His results stunned the nascent Kinect-hacking community and gained wide attention including in an article in the New York Times. Kreylos has gone on to integrate the Kinect thoroughly into his virtual reality research and to explore its use in remote communication systems, allowing multiple scientists to inhabit a shared virtual environment where they can manipulate data for collaboration.

You began your work with the Kinect immediately after the open source driver was first released. Why were you excited about the Kinect? What work were you doing at the time and how did the Kinect relate to that work?

I had been using 3D video for several years before the Kinect's release, using a 3D camera system based on clusters of regular (2D) cameras developed at UC Berkeley. We were already using 3D video as an integral component of 3D tele-collaborative applications back then, but the Berkeley camera system was tricky to set up, very expensive, and fairly low fidelity (320x240 pixels at 15 fps per cluster). I had been looking for a low-cost replacement for a long time, and heard about PrimeSense's goal to develop a consumer 3D camera several years back, at which time I contacted them directly. Turns out they had entered their exclusivity agreement with Microsoft just days prior to me contacting them, so they couldn't provide any test hardware at the time. When Kinect was announced later, I expected that it would employ strong encryption to only work with Xbox (I assumed that Microsoft would sell it under cost, like the Xbox itself). When Hector Marcan's work showed that Kinect was not encrypted after all, I jumped right in. Since I already had the larger 3D video environment in place at the time, all I had to do after picking up my first Kinect was to develop a low-level driver (based on Hector's reverse-engineered USB startup command sequence) and some calibration utilities. That's why I got a full working 3D video application only three days or so later.

Since then, Kinect cameras have replaced our previous 3D video hardware, and I'm currently looking into improving the video quality.

From your videos and your open source code releases, it seems like a lot of your work with the Kinect was directly translated from your existing work with other 3D video capture systems. Will you describe the system you were using before the release of the Kinect? What role does the Kinect play today in your ongoing research? How easy was it to "port" the work you were doing to use the Kinect?

The Berkeley 3D video system uses independent clusters of 2D cameras. Each cluster of three cameras is functionally equivalent to a single Kinect. Instead of structured light, the old system uses multi-view stereo reconstruction, or "depth from stereo." Two cameras in each cluster are black and white and reconstruct depth; the third camera is color and captures a texture to map onto the reconstructed geometry. Geometry from

multiple clusters is joined just as I do for the Kinect, to increase coverage and reduce occlusion shadows. Compared to structured light, depth from stereo is computationally very complex, and you need one high-end PC to run each cluster, for about 15 fps at a reduced resolution of 320x240. Due to the special FireWire cameras, each cluster (including the PC) costs about $5000. That's why I was waiting for Kinect as an affordable replacement, especially for our line of low-cost VR environments.

When I replaced the old system with Kinect, I used the chance to also remove some idiosyncrasies from the old code, so I ended up replacing more than just the bare minimum. In our larger software, 3D video is/was handled by a plug-in to our tele-collaboration infrastructure. Instead of adapting that plug-in for Kinect, I wrote a different plug-in from scratch, using an improved network protocol. This has the added benefit that I can use both systems at the same time, to stay compatible with our Berkeley colleagues. But in the larger context of the software, that was still a tiny component of the overall software. So, essentially, the Kinect dropped right in.

The Kinect's, or, more precisely, 3D video's, role in my work is to provide a mechanism for two or more people from distributed locations to work together in a virtual 3D environment as if they were in the same place. Our main targeted display environments are not desktop PCs, but fully immersive VR environments such as CAVEs. In those, the 3D video is not rendered on a screen, but appears as a free-standing life-size virtual hologram, which unfortunately never comes across in any videos or pictures. In those environments, 3D video enables a completely natural way to talk to and work with people that are long distances away.

How would you describe the overall mission of your research? What drew you to computer vision work in the first place?

My overall research is to develop software for interactive visualization and data analysis, particularly in VR environments using virtual reality interaction techniques, i.e., using your hands to directly manipulate 3D data instead of using a mouse or other 2D devices as intermediaries.

From observing our scientist users, we learned that data analysis is often a group task, where two to four people work together. It turns out that they spend the majority of the time not actually looking at the data, but talking to each other about the data. So when we started developing a tele-collaboration system to allow spatially distributed teams of scientists to work together effectively, without having to travel, we quickly realized that we needed a way to create convincing "avatars" of remote participants to be displayed at each site to support natural interaction. It turned out that 3D video is a great way of doing that — while we haven't done any formal studies, I believe that 3D video, even at low fidelity, is much more natural than animated or even motion-captured computer-generated avatars. So we were forced to implement a 3D video system, and since we're not computer vision experts, we teamed up with UC Berkeley to integrate their system into our larger software. When Kinect came around, the com-

puter-vision aspects of 3D video were already integrated into the hardware, so I could develop the rest of the necessary driver software myself.

A lot of your work focuses on using depth cameras to integrate 3D footage of people into virtual environments for telepresence and other communications applications. What are the areas of technology that you think will be most altered by the arrival of depth cameras like the Kinect? What application areas do you find to be the most promising?

I think the most immediate transformational aspect of cheap 3D cameras is tele-presence. Even if you don't have an immersive virtual reality environment, 3D video is still better at teleconferencing than 2D cameras, due to 3D video's elegant solution to the gaze direction problem. I think that we'll quickly see new videoconferencing software that will use 3D video even in desktop or laptop settings, slowly replacing Skype et al.; alternatively, it would make a whole lot of sense for Skype to have a plug-in for 3D video in the near future. Especially now that it's owned by Microsoft...

Apart from that, there are obvious implications for novel user interfaces. People might not want to have to wave their hands in front of a computer to do most work, as I don't see how typical tasks like web browsing or text editing could really benefit from hands-free interfaces beyond gimmickry, but there are other aspects. One is simple and accurate head tracking for 3D graphics such as in computer games. Combined with stereoscopic displays, that could really change things a lot. As an aside, stereoscopy without head tracking doesn't really work, which is why 3D movies and 3D video games aren't taking off as quickly as one might have thought. but throw in head tracking, and you suddenly have 3D displays that appear just like real holograms, only at much better visual quality. Combined with games or other 3D applications exploiting that fact, this could be a big thing. The other line is for more professional applications, where we're already doing direct 3D interaction. Using Kinect, or a higher-resolution future version of the same, we could do away with some of the gadgets we're using, for an even less intrusive user interface. This kind of stuff is not going to go mainstream any time soon, but it will have huge benefits inside its application areas.

What skills do you think beginning programmers should work on if they want to create interesting projects with 3D capture technologies? What are the current problems in developing the technology further? Is it a matter of user interface? Real world applications with users? More basic research?

This sounds bad, but from my experience of doing 3D computer graphics research at universities, and having taught 3D CG classes, I have learned that most people, even those working in 3D, don't really "get" 3D. Most people tend to think of 3D as 2D plus depth, but that leads to cumbersome approaches, applications, and, particularly, bad user interfaces. You can see the effects of this mindset in a lot of places, especially in 3D movies and many 3D computer games.

As one concrete example, there are de-facto standard ways of using 2D input devices (mouse, keyboard, joystick) to control 3D worlds. Think WASD+mouse look or virtual

trackballs. When faced with a true 3D input device, like the extracted skeletons provided by OpenNI or Microsoft's Kinect SDK, many people's first approach is to develop a wrapper that translates body motion into inputs for the existing 2D control metaphors, which really doesn't make any sense whatsoever. 3D input devices, instead, should come with completely new 3D interaction metaphors. These kinds of things have been looked into in the scientific field of virtual reality, and that's where budding Kinect developers should be looking. For example, what I am doing with 3D video and Kinect has already been described, down to some algorithms, in a 1997 VR/multimedia paper by Kanade et al. But who'd have thought to look there?

In a nutshell, what I'm trying to say is that the most important requirement for doing exciting stuff with 3D capture is not anything technical, but a mind set that fully understands and embraces 3D. The rest are just technicalities.

So I think the most important avenues of further development are two-fold: developing better 3D capture hardware (the Kinect, after all, is a first-generation piece of technology and still quite low-fidelity), and fostering a paradigm that embraces true 3D, and consequently leads to better applications, and better interfaces to use them.

What advice would you give to a programmer who was excited about the Kinect and wanted to learn more about the principles of computer vision and to learn more about the current state of the research?

Not being a computer vision expert myself, I don't have good advice. I'd say the basic requirement is a good understanding of 3D, and linear algebra. A good way of following state-of-the-art computer vision research, besides following the literature, is keeping in touch with software projects such as OpenCV, where researchers often implement and release their latest algorithms.

What do you think of all of the "Kinect hacks" that have been published online since the launch of the Kinect? What are some of your favorites? What capabilities do you think have failed to be explored?

This might sound strange, but I haven't really kept up with that scene. Most of the initial hacks were just that, one-off projects showcasing some technical feature or novel idea. It only gets interesting once they move into doing something useful or interesting. In that sense, the "real-time motion capture for 3D CG avatars" ones were showing possibilities for exciting real applications, and the 3D body scanning and rapid prototyping booth that someone built and exhibited to the public was an immediately useful new thing [editorial note: Kreylos is referring to Spanish collective blablabLAB who is interviewed elsewhere in this chapter]. I have not, so far, seen any real progress on novel 3D user interfaces. Some preliminary work like the gestural web browsing interface didn't excite me personally, because I honestly don't believe that anybody would actually do that once the novelty wears off. What I haven't seen, and maybe I haven't been looking in the right places, are true 3D interfaces for 3D modeling or 3D data analysis.

Figure 1-9. Alejandro Crawford used four Kinect to build a VJ system to accompany the band MGMT on tour.

Are there other computer vision technologies that are currently in the research phase that you can imagine having a similar impact to the Kinect if made equally accessible? What's the "next Kinect"? What will be the state of the art in this field in 5 years?

My crystal ball is in the shop right now, so... I really don't know, not working in the computer vision field. But I hope that the next Kinect will have higher resolution, and the ability to use multiple ones in the same capture space without mutual interference. Those are not very lofty aspirations, but they would really improve my particular use of the technology, and would also improve 3D user interface work, for example by the ability to reliably track individual fingers.

Whose work in this field do you find most exciting? Who else should I be looking at and talking to?

Since they're doing more or less exactly what I'm doing (certain aspects at least), and, in those aspects, arguably do a better job, you could contact the group around Henry Fuchs at UNC Chapel Hill, particularly Andrew Maimone regarding his impressive implementation of 3D video conferencing.

Alejandro Crawford

Alejandro Crawford is a video artist and a student at NYU's Interactive Telcommunication Program. During 2011 he toured with the indie rock band MGMT creating live visuals for their performances using the Kinect. Crawford used four Kinects to create a psychedelic video projection to accompany the band's shows. The tour visited more than 65 venues in 30 countries and was seen by thousands of fans.

What kind of work had you done before being tapped by MGMT to do visuals? How did you get involved with working for the band?

I honestly hadn't done that much. Before I started at ITP I didn't know what a MAX/MSP Jitter was. I come from a poetry/writing background and as a kid I wanted to be a director and lots of my later "writing" endeavors started to stray away from linguistic language for video — or audio video. commercial poem is probably pretty indicative of that time, and in many ways a mock-up for ideas I'd be able to explore thru programming a year or so later.

As to the band, starts in undergrad really with my friend (and roommate at the time) Hank Sullivant. Hank and Andrew van Wyngarden grew up in Memphis together and there was a time either during or right after their time at Weslyan that Andrew and Ben Goldwasser came or would come down to Athens and party and there'd be a keg party and at that time, MGMT was like hilarious avant-karaoke, where they'd play their tracks off the iPod and there'd be some mics and lots of everybody dancing and it being really hot.

Hank was the original lead guitarist of MGMT when James Richardson was the drummer. But my first year at ITP Hank and James all lived in a house together in Bushwick off Morgan stop. Hank and I actually shared bunk beds. I'm talking, like Walmart.com build them yourself boi-bunks. Gnarly times.

But I guess it was really ITP and specifically Luke Dubois' Live Image Processing and Performance class. Josh Goldberg subbed for Luke and showed us the LFOs that control his work "Dervish", which basically involves video feedback and rotation and oscillators. That blew me away and a lot of how I went about learning Jitter (and i'm very much still learning Jitter), messing with video, "what happens when I do this to these numbers and this thing down here?" etc.

James, being my roommate and a close friend, was around for a lot of that. Later I got this really random email from James (who was on tour at the time) saying MGMT had had a meeting and decided they wanted to know if I wanted to do visuals for them. It was then that I asked ITP for a leave of absence. I went down to Athens, GA shortly after that and started writing VJ software, collaborating with Alfredo Lapuz Jr., better known as IMMUZIKATION.

How did you decide to use the Kinect in your work for the band? Was it your idea or theirs? Did you know you wanted to work with the Kinect as a piece of technology or did you choose it to achieve a particular aesthetic result? What did the band want in the visuals? Had you worked with the Kinect before?

I asked Santa for a Kinect Christmas 2010 and after New Years I brought it to athens; Alfredo's the golden goose DJ of Athens so it was always nice to be able to test out new prototypes by doing visuals beside him; this is how the Kinect stuff started.

I was already very into and very obsessed with Andrew Benson; his HSflow and flowRepos GL shaders (for optical flow distortion) were the most gorgeous, flowy Takeshe Murata-type "data-moshy" impressionistic thing i'd come across. So rad. The idea of working with the Kinect really started with me me and Ben Goldwasser (MGMT bandmember) nerding out. Ben's a rad programmer himself and we'd talk shop lots so the "wouldn't it be cool?" seed was already planted. Plus i had already seen Flight404's Cinder Kinect radness and was hooked on the idea.

The wonderful thing about working for MGMT is the faith they have in me. I basically said to them, wouldn't it be cool if we had a Kinect for every member in the band and I could play psychedelic TV producer: "camera 2, camera5, camera1, go cam 4...". I also wanted to include the benson shaders and LFO feedback effects, so I started programming, sewing it all together (with one Kinect). Then I got back to New York and one of MGMT's managers and i went to Best Buy and got 5 Kinects then B&H video got a USB hub and this Pelican road case.

Talk about the development process behind this project. What environment or programming language did you use to work with the Kinect? How did you develop the look and interaction of the piece? What was the band's input in the process? How did you test it and prepare it for tour?

I spent about a month getting familiar with the Kinect, a lot of "what happens when i do this?" and Jitter coding. I was particularly interested in the ability to slice the depth mapping into color zones. It was very raw (in the sense that it was arguably employing a very primitive aspect of the Kinect's abilities) but on the other hand it morphed into this kind of stuff once the Benson shaders took hold.

But that was still one kinect. Once i had all 5 I remember very well the first attempt: placing them all out around me in my home, plugging them all into the USB hub, turning them all on and...like, nothing. I went white then went to Google and soon learned all about how much juice these puppies need per port. So i had my first big obstacle. Five Kinects into on Macbook Pro: how? I emailed Luke duBois and Jean Marc Pelletier, who made the Jitter Kinect module (among other things including Jitter's computer vision lib). I seem to remember Pelletier saying: not gonna happen. And luke was like: you need to make sure you can even address that many Kinects.

Express card was not an option. Buying a tower was not an option. USB 3.0 cards were not an option.

Resigning myself to four USB ports and therefore four Kinects, I decided to try running two Kinects into my personal MacBook Pro and then sending their feeds over UDP to the MGMT video MacBook Pro, with two more Kinects jacked into it. There was terrible delay and I felt more and more screwed. Finally I tried out connecting the com-

puters via Cat6 ethernet cable. The cabled connection was much faster and I was down to milliseconds of latency between the Kinects coming from the second computer.

Basically the patch allows modular control of many of the parameters pertaining to the optical flow distortion of the color-sliced depth mapping and color tweaking and cross fading between a feedback-driven (shape-rendering side). I also had buttons mapped to switch from one Kinect to another and faders to control tilt, buttons to switch mode, etc.

The band saw it for the first time in Japan. All the testing/building was done in not-site-specific environments (aka inside apartments, mostly, my own). I didn't really realize how much of a factor that was, but we didn't have any production rehearsal days, we hadn't rented a warehouse/stage where I could mock it up, set it up. I was a bit anxious about the first show in this regard. I knew I needed to mount the Kinects on the tops of mic stands but it'd be the first time I really set it up right.

What was the process like for setting up the Kinects and the rest of your system at each show? Did anything ever go wrong?

I always carry a Folsom Image Pro HD with me (usually it's rack mounted) but for the Asia tour, because there were so many different countries and flights we were sourcing equipment locally all the time. I didn't want to get in a situation where they told me they didn't have a Folsom, because they're basically like scan converter swiss army knives. Good luck not being able to pass your video signal along in whatever format or size. So, in total: scan converter, two MacBook Pros, an apc40, four Kinects, lots of active-USB cabling, daisy-chained, maybe one or two other minor things like projectors.

Every show day I'd build VideoLand, oversee the setting up of projectors, show some stage-hands how to do the first Kinect-mounting-to-mic-stand and then we'd lay all cabling. Then I'd wait until soundcheck to finesse the Kinect positions and distances from the band members. Then I'd "spike" the positions with gaffer tape then remove the Kinects till showtime, unplugging them at the connectors.

Things went wrong all the time in a kind of weird Murphy's law way, from electricity crossovers between video sound and lights to the Jitter patch crashing once (at the outro of a song, luckily). I just restarted the patch for the next song.

The first time I tried using the Kinects during a daylight festival show I thought they were broken until I realized the sun was just blinding them. This blinding effect happened once or twice also with particular lights on stage. And maybe once or twice Matt, the bass player, picked up a Kinect and placed it in front of the crowd. That wasn't so much an error as unaccounted for (distances weren't calibrated for that) but it still looked cool, hands swaying, moving paint on the screen.

Do you have plans to work with the Kinect more in the future in other contexts? What do you wish you could do with the Kinect that it can't do right now or that you haven't figured out yet?

I want to get into skeleton tracking stuff and augmented reality with the next album/tour. I want to build an island video game in Unity3D with lots of cool psychedelic spots. I want the band skeleton tracked and 3D modeled on that island. I want to be able to teleport from one place to another (like cutting back and forth between video sources). I wanna VJ with a video game!

Adafruit

Limor Fried and Phil Torrone

When did you first hear about the Kinect/Project Natal? Why did it capture your interest?

Limor and I had heard about when Microsoft was demo'ing it at the 2009 E3. At the time we just filed it away in our heads in the "future hack" folder. At the time our thought was "hey, this is going to be a low cost RGB + distance camera that someone will probably reverse engineer". We thought it would be really useful for art and robotics projects, we guessed right!

Why did you think it was important that there be open source drivers for the Kinect?

When the Kinect was launched we received a mysterious email from Johnny Lee (wii mote hack fame, now at Google, at the time at Microsoft) and he wanted to chat, I had asked if he could drop an email but he wanted to chat on the phone. After a series of conversations he really inspired us to create the open-source driver bounty (and he helped fund it). It was pretty clear to us Microsoft didn't have any plans, at all, to open any parts of the Kinect and while we could have just reverse engineered the Kinect on our own and released the drivers we thought we could really get a lot of community support with an open bounty effort. We didn't want this to be about us (Adafruit) we wanted to celebrate and demonstrate the power of open source. We knew once there were free and open drivers projects, companies and ideas would flourish.

A lot of Adafruit's efforts are put into supporting hardware hackers through kits and information. How do you see the Kinect's relationship to the world of hardware hacking?

Our goal want to show that commodity hardware or subsided hardware from large companies can have a life beyond what they were intended for. I think it became really clear within weeks of the open source drivers that the projects created were more interesting and had more potential for humankind then the shipping games. Companies like Microsoft create businesses, we create causes.

What possibilities does it open up?

Open-source is usually associated with education, so right away any open-source drivers for Kinect would be embraced and used in education (and that happened right away). Open-source also attracts artists, the most interesting things about Kinect when

it launched wasn't the games it was the amazing projects the hackers and makers released.

What problems does it solve?

It was incredibly expensive to develop a RGB + distance camera, every group that wanted to do this would need to reinvent everything. Now you can just grab one at the local big box store for $150.

What do you think about PrimeSense's efforts to open up their tools? What are they getting right? What are they getting wrong? What's the relationship between OpenNI and OpenKinect?

We haven't followed this too much, our general opinion is that "this is good!" - they saw all the interest in the Kinect and they want people to do the same thing and I think they want to make sure they public credit for their work. Most people assume Microsoft invented the Kinect.

What are the prospects for additional depth cameras coming in the future?

It all depends on how the patents around these things works, technically it's possible and there's no reason not see more of them - it might just depend on who owns what.

Will there be an open source depth camera at some point?

Probably not, but every depth-sensing camera that's useful will have open-source drivers.

What is the single coolest Kinect hack you've seen? What was the first thing built with OpenKinect that made you say: this was worth it!

Every project has been worth it, that's for sure. But we like this early one the best: Interactive Puppet Prototype by Emily Gobeille and Theo Watson. It captures imagination, art and what's possible - you can imagine using this to control a surgical robot or just for a kid's video game.

CHAPTER 2

Working With the Depth Image

In this chapter we're going to learn how to work with the depth image from the Kinect. As I explained in Chapter 1, the Kinect is a depth camera. This means that it captures the distance of the objects and people in front of it in the form of an image. This depth image is similar to the normal color images we're used to seeing all the time from digital cameras except instead of recording the color of light that reached the camera, it records the distance of the objects in front of it. I'll show you how we can write Processing programs that access the depth image from the Kinect, display it on the screen, analyze its pixels in order to make measurements of the space in front of the Kinect, and use the results of those measurements as a simple user interface.

We'll start with the very basics. I'll show you how to install the Processing library that lets us access the Kinect and then we'll use that library to display a depth image as well as a regular color image. Once we've got that up and running, I'll show you how to access individual pixels in order to determine their color and therefore the depth measurements that they represent. Once you're comfortable with accessing individual pixels, I'll show you how to write code that analyzes all of the pixels in the depth image to do more advanced calculations like finding the closest point in front of the Kinect. We'll then use this ability to build some basic interactive applications: we'll draw by waving our hands around, take some measurements of our room, and browse photos with some basic gestures.

Let's get started!

Images and Pixels

What is an image? Whether it's a depth image from the Kinect, a frame from your computer's built-in webcam, or a high resolution photo from the latest digital SLR, digital images are all simply collections of pixels. A pixel is the smallest unit of an image. Each pixel is a single solid color and by arranging many pixels next to one another, digital images can create illusions like smooth gradients and subtle shading. With

enough pixels to create smooth enough images, you get the realistic digital images that we see everyday.

Depending on the type of image, the variety of possible colors a pixel can be may vary. In a black and white image, each pixel will be either black, white, or some shade of gray. In a color image, each pixel has a red, green, and blue component that are combined together in different quantities to create any color. Each component of the color is like a single black and white pixel, but instead of ranging from from black to white through hundreds of shades of gray, each color component ranges from black to the full intensity of its color through a spectrum of darker versions. So a red pixel component, for example, can be any color from black through deep maroon, to a bright red. When this red component is combined with similar green and blue pixel components at particular shades in their own spectrums, the pixel can represent any in a large spectrum of colors.

As we start writing programs that process images, we'll see very quickly that each pixel is just a number. Each pixel's number value represents its intensity. In a grayscale image, the number value represents how dark the pixel is, with zero being black and 255 being white. In a color image, each pixel component (red, green, and blue) can range from zero to 255 with zero being black and 255 being the full color.

Where did this number 255 come from? The range of colors a pixel can represent is determined by how big of a number we're willing to use to store its largest value. The larger a number we're willing to accept for each pixel, the more memory each image will take up. And since images can have millions of pixels (and always have at least thousands of them), larger pixel sizes will add up quickly. Images that use big numbers to store each pixel will quickly get unwieldy, taking up huge amounts of memory space and becoming very slow to process. Hence, Processing (and many other programming environments) have chosen a maximum value for pixels that's modest, but still large enough to create nice looking images: 255. This value is known as the image's "bit depth".

Depending on its resolution, every digital image has thousands or millions of pixels arranged into a grid. This grid consists of columns and rows. Like with any grid, the number of columns is equal to the width of the image and the numbers of rows is equal to its height. As you learned in Chapter 1, the depth image captured by the Kinect is 640 pixels wide by 480 pixels high. This means that each frame that comes in from the Kinect's depth camera will have a total of 307,200 pixels. And, unlike in the grayscale and color images we've mostly discussed so far, each of these pixels will be doing double duty. Each pixel's number, between zero and 255, will represent both a color of gray and a distance in space. We can use that same number to either view the data as an image or calculate the distance to the object that pixel is a part of. We can treat zero as black and 255 as white or we can treat zero as really far away and 255 as close-up. This is the magic of the depth image: it's both a picture of the scene in front of the Kinect and a series of measurements of it.

But the Kinect is not just a still camera. In fact, it's more like the webcams we commonly use for online video chat. Like those cameras, the Kinect captures a live movie of the scene in front of it and sends it to our computer so fast that we don't notice the time it took to get there: it's image appears to match the movements of the objects and people in front of it in "real time". Of course, unlike those cameras, the movie that the Kinect captures contains information about depth as well as color.

What is a movie? Up to this point we've only been talking about images and pixels. Well, a movie is just a series of images over time. Just like an old fashioned film reel consists of thousands of still photos, a digital movie (or live stream) consists of a sequence of these individual frames, one after another. The Kinect captures images one at a time and sends them to our Processing sketch which then displays them on the screen so quickly that we see the result as a smooth movie.

And since a digital movie is just a series of images over time, our approach to it will be to process each frame as it comes in and then use the results from each frame to update some information that we're tracking over the duration. For example, if we were trying to track the closest point over time, we'd do it by always finding the closest pixel in the current image while we had it around and then we'd use that result to update some object or output that persisted over time like the position of a circle. Pixels. Image. Movie.

But now we're starting to get ahead of ourselves. Before we can process the thousands of pixels in each of the endless images streaming in from the Kinect, we need to get our hands on a single pixel. And that means starting at the beginning: plugging in our Kinect and writing our first Kinect program.

Now that we know a little bit about how images and pixels work in theory, we're ready to write our first Kinect program so we can start working with some in practice. Before we can dive into the code, though, we have some administrivia to get through, namely installing the Processing library that will give us access to the Kinect's data. This should only take a couple of minutes and we'll only have to do it this once.

Project 1: Installing the SimpleOpenNI Processing Library

If you're buying a Kinect just for hacking, make sure that you don't select the version that comes bundled with an XBox. That version will lack the USB connection on its cable that you need to attach the Kinect to your computer. If you plan on using your Kinect with your XBox as well (or if you already have a Kinect that came bundled with an XBox) you can buy the separate power supply and connection cable you need to use the Kinect with your personal computer from Microsoft's online store under the product name: Kinect Sensor Power Supply.

Before we get started writing code and looking at Kinect data, we need to install the Processing library we'll be using throughout this book. As I mentioned in Chapter 1, we'll be using a library called `SimpleOpenNI`. This library provides access to all of the data from the Kinect that we'll need as well as a series of tools and helpers that will prove invaluable along the way.

This book is not intended to teach you Processing from scratch. However if you've had even a little exposure to Processing you should do fine. I'll start at a beginner level and gradually teach you all of the more advanced concepts and techniques you'll need to work with the Kinect. If you have some experience with another programming language you should be able to pick up Processing as we go.

If you've never used Processing before, you'll need to install it before you get started. The Processing website has downloads available for Mac, Windows, and Linux. At the time of this writing, 1.5.1 is the most recent version of Processing, but version 2.0 is expected soon. All of the examples in this book should work in either of these versions.

The steps for installing a library in Processing differ slightly based on your operating system. I'll provide instructions here for installing `SimpleOpenNI` on Mac OS X, Windows, and Linux. These instructions may change over time, so if you encounter any problems please check the SimpleOpenNI Installation Page for up-to-date details.

Installing `SimpleOpenNI` happens in two phases. First you install OpenNI itself. This is the software system provided by PrimeSense that communicates with the Kinect to access and process its data. The steps involved in this phase differ based on your operating system. Below I've included step-by-step guides for getting OpenNI installed on each major operating system. Find the guide for your operating system and work through all the steps.

After you've got OpenNI successfully installed, the final step is to install the `SimpleOpenNI` Processing library itself. This is much easier than installing OpenNI and much more standard across operating systems. I've provided instructions for this as well at the end of this section. Once you've completed the install steps for your specific operating jump down to there to finish the process.

Installing OpenNI on OS X

This is the easiest platform on which to install OpenNI. Max Rheiner, the author of `SimpleOpenNI` has provided an installer that makes the process straightforward and simple. The installer works best on Mac OS X 10.6 and above. Running it will involve a little bit of work at the command line, but I'll show you just what to do. Here are the steps:

- Go to the the SimpleOpenNI Google Code site

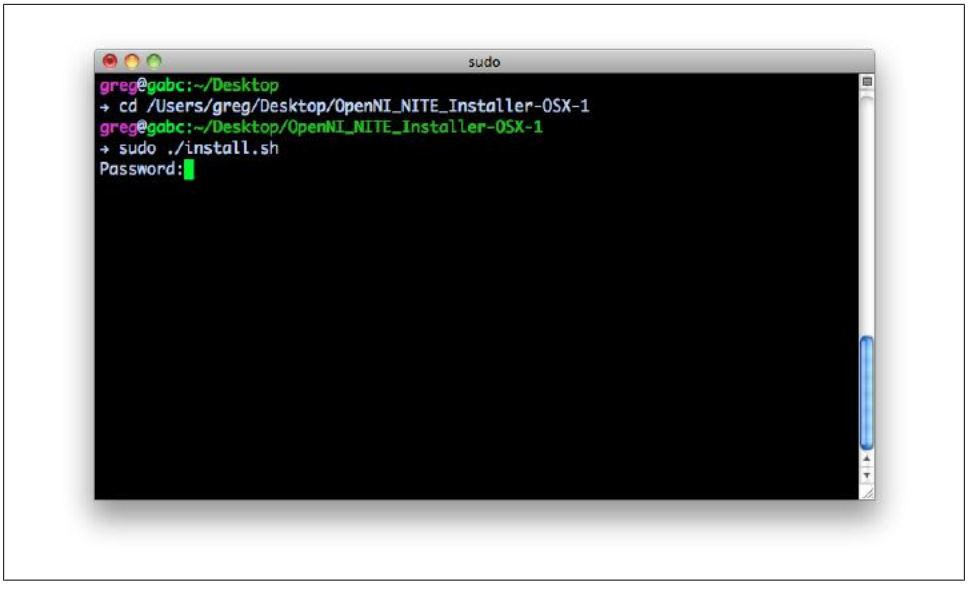


Figure 2-1. To install OpenNI on OS X, download the installer, unzip it, navigate to that directory, and run the install script as shown here.

- Find the Mac OS X installation section. Click to download the installer.
- Double click on the .zip file you just downloaded to unarchive it. A directory named "OpenNI_NITE_Installer-OSX" will appear.
- Launch the Terminal. You can find it in the Finder by going to Applications → Utilities → Terminal.
- Change directory to the downloaded folder. To do this type "cd" (for change directory) followed by a space, and then drag the folder you just unzipped from the Finder into Terminal. This will paste the correct path to that folder into your prompt. Hit return. (Terminal commands for this and the next few steps can be seen in Figure 2-1)
- Run the installer with the command: `sudo ./install.sh`. Type your password when asked. (You must be logged in as a user with administrative privileges to do this.)

This should successfully complete the installation of OpenNI on Mac OS X. Proceed to the section below on installing the Processing library. If you encounter any errors visit the SimpleOpenNI Google Code site for debugging tips. Otherwise proceed with the instructions for installing the Processing library below.

Installing OpenNI on Windows

Installing OpenNI on Windows takes a number of steps. There are a number of pieces of supporting software that have to be installed one at a time. I'll provide the basic steps

here along with links to each of the pieces. Make sure you read the pages for each component carefully to ensure that you're following the correct steps to install them for your system.

- Download the latest version of OpenNI from PrimeSense. Make sure you find the correct version for your system on that page (for example, if you're using a 64-bit version of Windows, you'd choose the one labeled "OpenNI Unstable Build for Windows x64 (64-bit) Development Edition" or similar).
- Double click the .msi installer file you just downloaded and follow the instructions.
- Download the latest version of NITE from PrimeSense. NITE is the PrimeSense middleware that provides skeleton tracking capabilities. It is necessary for installing SimpleOpenNI and using it in this book. Again, find the appropriate version for your system (something like "PrimeSense NITE Unstable Build for Windows x64 (64-bit) Development Edition").
- Double click the .msi installer file and follow the instructions.
- Some versions of the NITE installer may ask you for a key. If yours does use this value: `0KOIk2JeIBYClPWVnMoRKn5cdY4=`, including the final equals sign.
- Visit the SensorKinect Github page.
- Click the "Downloads" link on the right side of the page and select "Download .zip".
- Open the zip file (named something like *avin2-SensorKinect-2d13967.zip*) and navigate to the "Bin" folder inside of it.
- Double click the .msi installer for your version of Windows (32-bit or 64-bit), named something like *SensorKinect-Win-OpenSource32-X.msi* where *X* is a dot-separated version number.

This should successfully complete the install of OpenNI on Windows. Plug your Kinect into your computer's USB port. If everything went smoothly the device drivers will install automatically, and Kinect should up in your Device Manager under PrimeSense as Kinect Audio, Kinect Camera, and Kinect Motor. If you encounter any errors visit the SimpleOpenNI Google Code site for debugging tips. Otherwise proceed with the instructions for installing the Processing library below.

Installing OpenNI on Linux

If you're a Linux user then you're probably quite comfortable with installing software for your platform. I'll simply point you towards the SimpleOpenNI install instructions for Linux and wish you good luck. The process is relatively similar to the Windows installation process described above, but instead of binaries you'll need to download the source code for OpenNI, NITE, and SensorKinect and then build it yourself including providing the product key for NITE which is: `0KOIk2JeIBYClPWVnMoRKn5cdY4=`.

Installing the Processing Library

Once you've gotten OpenNI installed (including the NITE middleware and the SensorKinect driver), it's time to install the Processing library. Thankfully this is dramatically easier than installing OpenNI and pretty much the same for all platforms. Here are the steps:

- Visit the downloads page on the SimpleOpenNI Google Code site.
- Select the version of SimpleOpenNI that is appropriate for your operating system and download it.
- Unzip the downloaded file (this may happen automatically depending on your browser configuration).
- Locate your Processing *libraries* folder. If you don't already know where that is, you can find out by looking at the Processing Preferences window. As you can see in Figure 2-2, the Processing Preferences window allows you to select your "Sketchbook location". In my case that location was */Users/greg/Documents/Processing*. Yours may differ based on your operating system and where you installed Processing. Your *libraries* folder will be located inside of this sketchbook folder. For me it is: */Users/greg/Documents/Processing/libraries*. If the *libraries* folder does not exist, create it.
- Drag the unzipped SimpleOpenNI directory to your Processing libraries folder.
- Quit and restart Processing.
- SimpleOpenNI should now show up in Processing's list of installed libraries (Figure 2-3). You can confirm this by looking for it under the Sketch → Import Library menu.

Now that you've got the library installed, grab your Kinect and plug it in. Plug the electrical plug into a power outlet as shwn in Figure 2-4, then plug the Kinect into the female end of the Y-shaped cable coming off of the power cable as demonstrated in Figure 2-5. This should have a glowing green LED on it. The other end of this Y-cable has a standard male USB plug on it. Connect that to your computer as shown in Figure 2-6. Stick the Kinect somewhere convenient, preferably so it's facing you.

You can test out your install by running one of the built-in examples that ships with SimpleOpenNI. Within Processing, navigate to the File menu and select Examples. This will cause a window to popup with all of Processing's built-in sketches as well as examples from all of the external libraries such as SimpleOpenNI. Figure 2-7 shows you where to look. Scroll down the list until you hit the entry for Contributed Libraries; open that entry and look for an entry named SimpleOpenNI. Inside, there will be folders for NITE and OpenNI. Inside of OpenNI, find DepthImage and double-click it.

This will cause Processing to open the DepthImage example sketch that is provided with SimpleOpenNI. This is a basic sketch, quite similar to what we'll construct in the next section of this chapter. You'll learn more about this code soon. For now, just hit

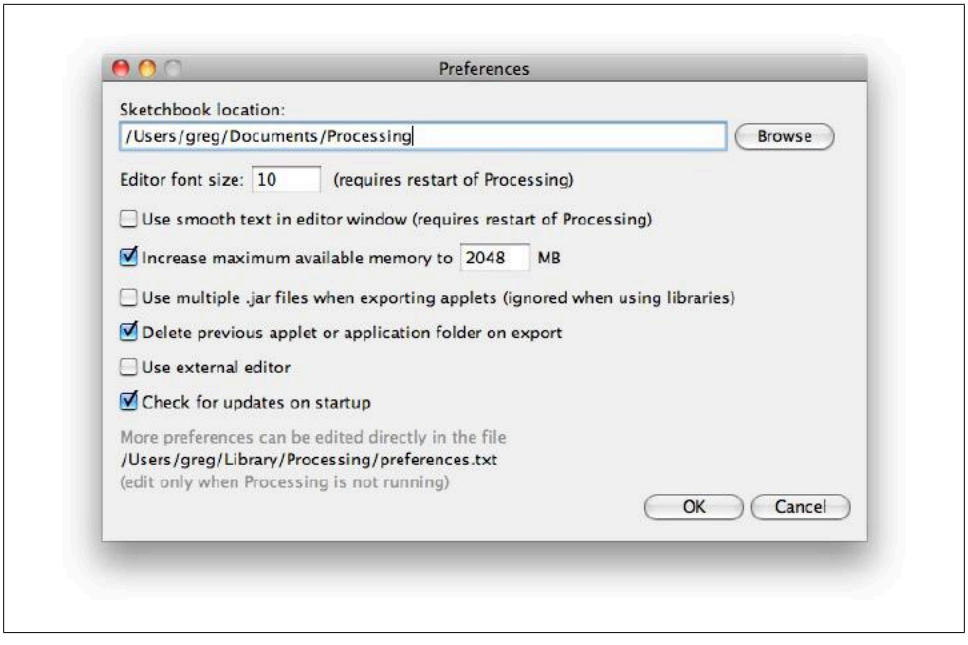


Figure 2-2. The Processing Preferences window. The "Sketchbook location" entry determines where Processing stores its libraries. You should install the SimpleOpenNI library in a folder called "libraries" inside this directory.

the play button in the top left corner of the Processing window to run the sketch (see Figure 2-8).

If everything installed correctly, a new window will pop-up and, eventually, you'll see a color image and a gray scale image side by side, representing the Kinect's view of the room. If it takes your sketch a little while to start up, don't worry about it. Slow start up is normal when using SimpleOpenNI.

However, your sketch may raise an error and fail to start. If it does it will print out a message in the Processing window. The most common of these messages is this:

```
Invalid memory access of location 0x8 eip=0xb48d8fda
```

If you see that it most likely means that your Kinect is not plugged in or not fully powered. Make sure your Kinect is connected to a power outlet and plugged into your computer's USB port. If you're working on a laptop also make sure that your laptop is plugged into power. Some laptops (especially Macs of recent vintage) will provide inadequate power to their USB ports when running off a battery.

If you encounter other errors consult the SimpleOpenNI Google Code site or review the install instructions provided here to make sure that you didn't skip any steps. If all else fails, try sending an email to the SimpleOpenNI mailing list (accessibly through the Google Code site). The project's maintainers are friendly and quite capable of

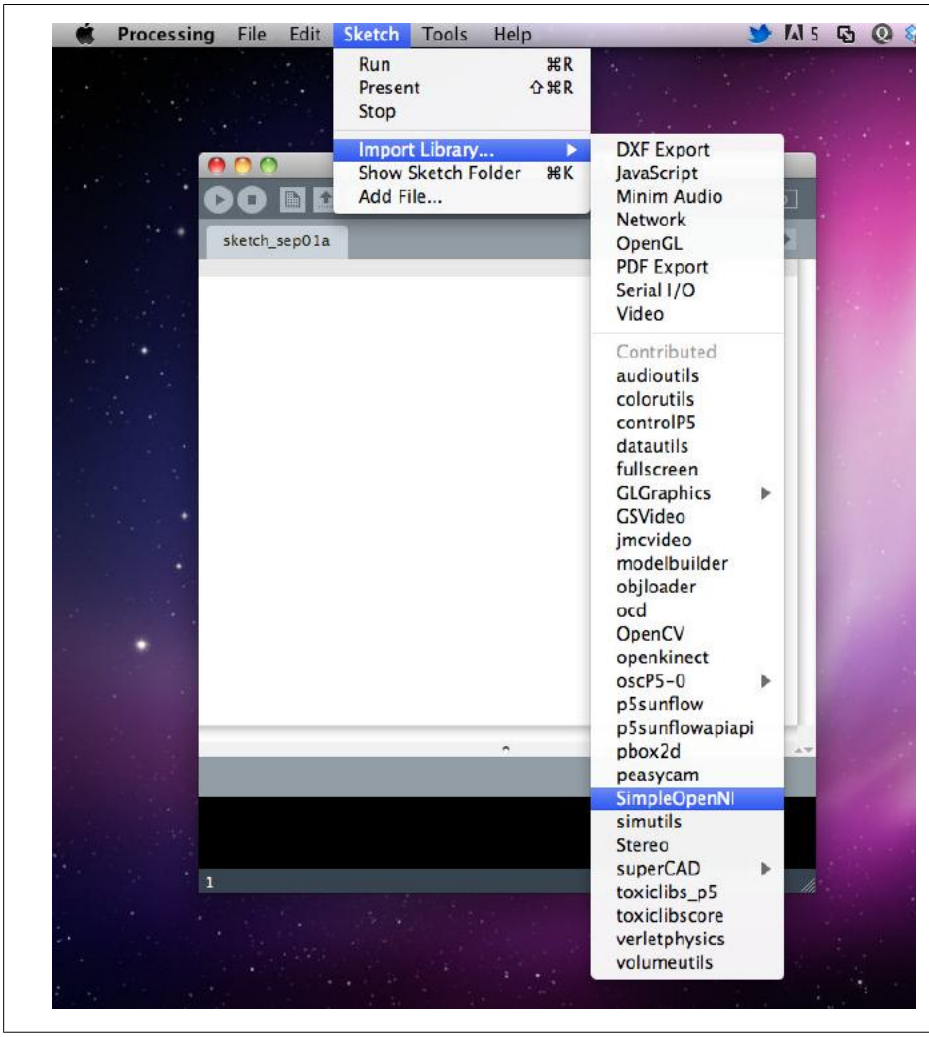


Figure 2-3. After a successful install, SimpleOpenNI shows up in Processing's list of libraries.

helping you debug your particular situation. You should search the mailing list archives before you post, just in case someone else has had the same problem as you.

Project 2: Your First Kinect Program

That's it for setup. Now we're ready to start writing our own code. Our first program is going to be pretty simple. It's just going to access the Kinect, read the images from both its depth camera and its color camera, and then display them both on the screen

Figure 2-4. The Kinect's power plug has a y-connector on the end of it. One leg of that connector has a female plug for an XBox connector the other has a male USB plug.

side-by-side. Once that's working, we'll gradually add to this program in order to explore the pixels of both images.

You've got the Kinect library installed and your Kinect plugged into your computer, so launch Processing and run the program below. Read through it, run it, and take a look at what it displays. Spend some time waving your hands around in front of your Kinect (this, you'll find, is one of the core activities that make up the process of Kinect development) and then, when you're ready, meet me after the code listing. I'll walk through each line of this first program and make sure you understand everything about how it works.

Figure 2-5. The Kinect has one wire coming off of it with a male XBox connector at the end. This plugs into the female XBox connector attached to the power plug.

```
import SimpleOpenNI.*;
SimpleOpenNI kinect;

void setup()
{
  size(640*2, 480);
  kinect = new SimpleOpenNI(this);

  kinect.enableDepth();
  kinect.enableRGB();
}

void draw()
{
  kinect.update();

  image(kinect.depthImage(), 0, 0);
  image(kinect.rgbImage(), 640, 0);
}
```

When you run this sketch you'll have a cool moment that's worth noting: your first time looking at a live depth image. Not to get too cheesy, but this is a bit of landmark like the first time your parents or grandparents saw color television. This is your first experience with a new way of seeing and it's a cool sign that you're living in the future! Shortly, we'll go through this code line-by-line. I'll explain each part of how it works and start introducing you to the `SimpleOpenNI` library we'll be using to access the Kinect throughout this book.

Figure 2-6. The male USB connector from the power supply plugs into your computer's USB port. With the Kinect plugged into the XBox plug and the power plug in a socket this completes the physical setup for the Kinect.

Observations about the Depth Image

What do you notice when you look at the output from the Kinect? I'd like to point out a few observations that are worth paying attention to because they illustrate some key properties and limitations of the Kinect that you'll have to understand to build effective applications with it. For reference, Figure 2-9 shows a screen capture of what I see when I run this app:

What do you notice about this image besides my goofy haircut and awkward grin?

First of all look at the right side of the depth image, where my arm disappears off camera towards the Kinect. Things tend to get brighter as they come towards the camera: my shoulder and upper arm are brighter than my neck, which is brighter than the chair, which is much brighter than the distant kitchen wall. This makes sense. We know by now that the color of the pixels in the depth image represent how far away things are, with brighter things being closer and darker things farther away. If that's the case, then why is my forearm, the thing in the image closest to the camera, black?

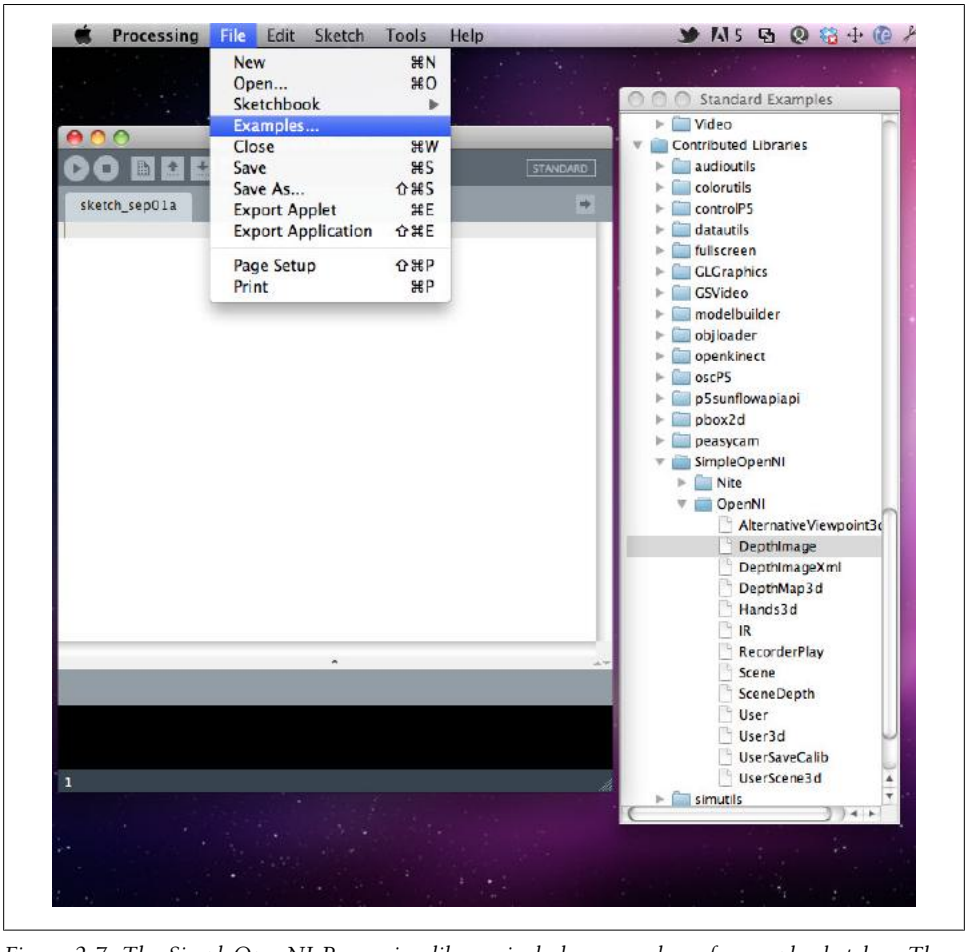


Figure 2-7. The SimpleOpenNI Processing library includes a number of example sketches. These sketches are a great way to test to see if your installation of the library is working successfully and to see

There are some other parts of the image that also look black when we might not expect them to. While it makes sense that the back wall of the kitchen would be black as it's quite far away from the Kinect, what's with all the black splotches on the edges of my shoulders and on my shirt? And while we're at it, why is the mirror in the top left corner of the image so dark? It's certainly not any further away than the wall that it's mounted on. And finally, what's with the heavy dark shadow behind my head?

I'll answer these questions one at a time as they each demonstrate an interesting aspect of depth images that we'll see coming up constantly as we work with them throughout this book.

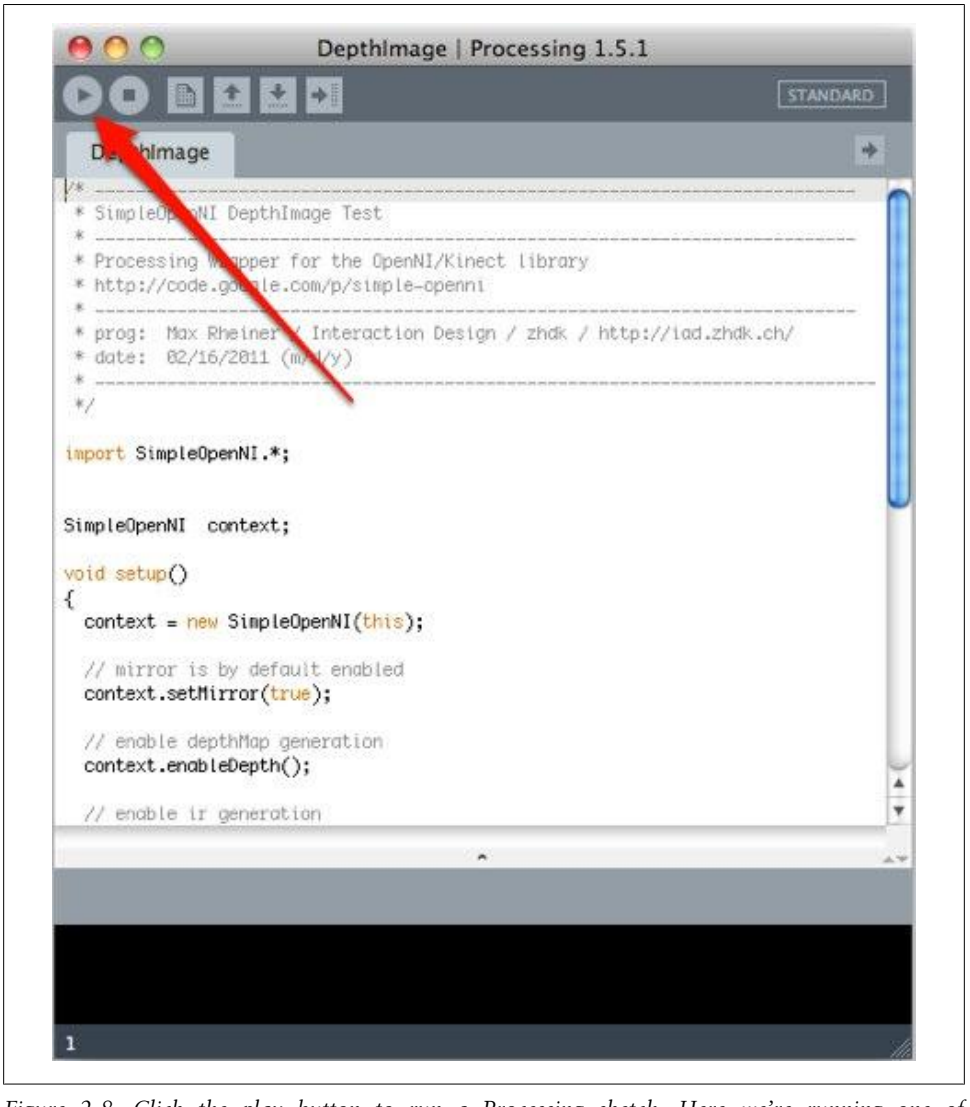


Figure 2-8. Click the play button to run a Processing sketch. Here we're running one of SimpleOpenNI's built-in example sketches.

Minimum range

As I explained in Chapter 1, the Kinect's depth camera has some limitations due to how it works. We're seeing evidence of one of these here. The Kinect's depth camera has a minimum range of about 20 inches. Closer than that and the Kinect can't accurately calculate distances based on the displacement of the infrared dots. Since it can't figure out an accurate depth, the Kinect just treats anything closer

Figure 2-9. A screen capture of our first Processing sketch showing the depth image side-by-side with a color image from the Kinect.

than this minimum range as if it had a depth value of zero, in other words as if it was infinitely far away. That's why my forearm shows up as black in the depth image, it's closer than the Kinect's minimum range.

Noise at Edges

First, what's with splotches around the edges of my shoulders? Whenever you look at a moving depth image from the Kinect you'll tend to see splotches of black appearing and disappearing at the edges of objects that should really be some solid shade of gray. This happens because the Kinect can only calculate depth where the dots from its infrared projector are reflected back to it. The edges of objects like my shoulders or the side of my face tend to deflect some of the dots away at odd angles so that they don't actually make it back to the Kinect's infrared camera at all. Where no IR dots reach the infrared camera, the Kinect can't calculate the depth of the object and so, just like in the case of objects close than 20 inches, there's a hole in the Kinect's data and the depth image turns black. We'll see later on in the book that if we want to work around this problem, we can use the data from many depth images over time to smooth out the gaps in these edges. However, this method only works if we've got an object that's sitting still.

Reflection Causes Distortion

Next, why does the mirror look so weird? If you look at the color image, you can see that the mirror in the top left corner of the frame is just a thin slab of glass sitting on the wall. Why then does it appear so much darker than the wall it's on? Instead of the wall's even middle gray, the mirror shows up in the depth image as a thick band of full black and then, inside of that, a gradient that shifts from dark gray down to black again. What is happening here?

Well, being reflective, the mirror bounces away the infrared dots that are coming from the Kinect's projector. These then travel across the room until they hit some wall or other non-reflective surface. At that point they bounce off, travel back to the mirror, reflect off of it, and eventually make their way to the Kinect's infrared camera. This is exactly how mirrors normally work with visible light to allow you to see reflections. If you look at the RGB image closely you'll realize that the mirror is reflecting a piece of the white wall on the opposite side of the room in front of me.

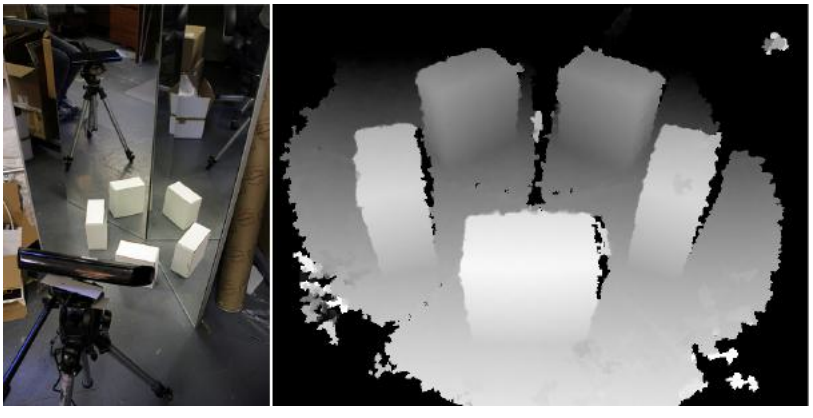


Figure 2-10. Artist Kyle McDonald's setup using mirrors to turn the Kinect into a 360 degree 3D scanner. Photos courtesy of Kyle McDonald.

In the case of a depth image, however, there's a twist. Since the IR dots were displaced further, the Kinect calculates the depth of the mirror to be the distance between the Kinect and the mirror plus the distance between the mirror and the part of the room reflected in it. It's like the portion of the wall reflected in the mirror had been picked up and moved so that it was actually behind the mirror instead of in front of it.

This effect can be inconvenient at times when reflective surfaces show up accidentally in spaces you're trying to map with the Kinect, for example windows and glass doors. If you don't plan around them, these can cause strange distortions that can screw up the data from the Kinect and frustrate your plans. However, if you account for this reflective effect by getting the angle just right between the Kinect and any partially reflective surface you can usually work around them without too much difficulty.

Further, some people have actually taken advantage of this reflective effect to do clever things. For example, artist and researcher Kyle McDonald set up a series of mirrors similar to what you might see in a tailor's shop around a single object, reflecting it so that all of its sides are visible simultaneously from the Kinect, letting him make a full 360 degree scan of the object all at once without having to rotate it or move it. Figure 2-10 shows Kyle's setup and the depth image that results.

Occlusion and Depth Shadows

Finally, what's up with that shadow behind my head? If you look at the depth image I captured you can see a solid black area to the left of my head, neck, and shoulder that looks like a shadow. But if we look at the color image, we see no shadow at all there. What's going on? The Kinect's projector shoots out a pattern of IR dots. Each dot travels until it reaches an object and then it bounces back to the Kinect to be read by the infrared camera and used in the depth calculation. But what about other objects in the scene that were behind that first object? No IR dots will ever reach those objects. They're stuck in the closer object's IR shadow. And

since no IR dots ever reach them, the Kinect won't get any depth information about them and they'll be another black whole in the depth image.

This problem is called *occlusion*. Since the Kinect can't see through or around objects, there will always be parts of the scene that are occluded or blocked from view and that we don't have any depth data about. What parts of the scene will be occluded is determined by the position and angle of the Kinect relative to the objects in the scene.

One useful way to think about occlusion is that the Kinect's way of seeing is like lowering a very thin and delicate blanket over a complicated pile of objects. The blanket only comes down from one direction and if it settles on a taller object in one area then the objects underneath that won't ever make contact with the blanket unless they extend out from underneath the section of the blanket that's touching the taller object. The blanket is like the grid of IR dots only instead of being lowered onto an object, the dots are spreading out away from the Kinect to cover the scene.

Misalignment Between the Color and Depth Images

Finally, before we move on to looking more closely at the code, there's one other subtle thing I wanted to point out about this example. Look closely at the depth image and the color image. Are they framed the same? In other words, do they capture the scene from exactly the same point of view? Look at my arm, for example. In the color image it seems to come off camera to the right at the very bottom of the frame, not extending more than about a third of the way up. In the depth image, however, it's quite a bit higher. My arm looks like it's bent at a more dramatic angle and it leaves the frame clearly about halfway up. Now, look at the mirror in both images. A lot more of the mirror is visible in the RGB image than the depth image. It extends further down into the frame and further into the right. The visible portion of it is taller than it is wide. In the depth image on the other hand, the visible part of the mirror is nothing more than a small square in the upper left corner.

What is going on here? As we know from the introduction, the Kinect captures the depth image and the color image from two different cameras. These two cameras are separated from each other on the front of the Kinect by a couple of inches. Because of this difference in position, the two cameras will necessarily see slightly different parts of the scene and they will see them from slightly different angles. This difference is a little bit like the difference between your two eyes. If you close each of your eyes one at a time and make some careful observations, you'll notice similar types of differences of angle and framing that we're seeing between the depth image and the color image.

These differences between these two images are more than just a subtle technical footnote. As we'll see later in the book, aligning the color and depth images, in other words overcoming the differences we're observing here with code that takes them into account, allows us to do all kinds of cool things like automatically removing the background from the color image or producing a full-color three dimensional scan of the scene. But that alignment is an advanced topic we won't get into until later.

Understanding the Code

Now that we've gotten a feel for the depth image, let's take a closer look at the code that displayed it.

I'm going to walk through each line of this example rather thoroughly. Since it's our first time working with the Kinect library, it's important for you to understand this example in as much detail as possible. As the book goes on and you get more comfortable with using this library, I'll progress through examples more quickly, only discussing whatever is newest or trickiest. But the concepts in this example are going to be the foundation of everything we do throughout this book and we're right at the beginning so, for now, I'll go slowly and thoroughly through everything.

On line 1 of this sketch, we start by importing the library:

```
import SimpleOpenNI.*;
```

This works just like importing any other Processing library and should be familiar to anyone who's worked with Processing (if you're new to Processing, check out Getting Started with Processing from O'Reilly). The library is called "SimpleOpenNI" because it's a Processing wrapper for the OpenNI toolkit provided by PrimeSense that I discussed earlier. As a wrapper, SimpleOpenNI just makes the capabilities of OpenNI available in Processing, letting us write code that takes advantage of all of the powerful stuff PrimeSense has built into their framework. That's why we had to install OpenNI and NITE as part of the setup process for working with this library: when we call our Processing code the real heavy lifting is going to be done by OpenNI itself. We won't have to worry about the details of that too frequently as we write our code, but it's worth noting here at the beginning.

The next line declares our SimpleOpenNI object and names it "kinect":

```
SimpleOpenNI kinect;
```

This is the object we'll use to access all of the Kinect's data. We'll call functions on it to get the depth and color images and, eventually, the user skeleton data as well. Here we've just declared it but not instantiated it so that's something we'll have to be sure to look out for in the setup function below.

Now we're into the setup function. The first thing we do here is declare the size of our app:

```
void setup()
{
  size(640*2, 480);
```

I mentioned earlier that the images that come from the Kinect are 640 pixels wide by 480 tall. In this example, we're going to display two images from the Kinect side-by-side: the depth image and the RGB image. Hence, we need an app that's 480 pixels tall to match the Kinect's images in height, but is twice as wide so it can contain two of them next to each other, that's why we set the width to `640*2`.

Once that's done, as promised earlier we need to actually instantiate the SimpleOpenNI instance that we declared at the top of the sketch, which we do on line 8:

```
  kinect = new SimpleOpenNI(this);
```

Having that in hand, we then proceed to call two methods on our instance: `enableDepth()` and `enableRGB()`, and that's the end of the setup function, so we close that out with a `}`:

```
  kinect.enableDepth();
  kinect.enableRGB();
}
```

These two methods are our way of telling the library that we're going to want to access both the depth image and the RGB image from the Kinect. Depending on our application, we might only want one, or even neither of these. By telling the library in advance what kind of data we're going to want to access, we give it a chance to do just enough work to provide us what we need. The library only has to ask the Kinect for the data we actually plan to use in our application and so it's able to update faster, letting our app run faster and smoother in turn.

At this point, we're done setting up. We've created an object for accessing the Kinect and we've told it that we're going to want both the RGB data and the depth data. Now, let's look at the draw loop to see how we actually access that data and do something with it.

We kick off the draw loop by calling the `update()` function on our Kinect object:

```
void draw()
{
  kinect.update();
```

This tells the library to get fresh data from the Kinect so that we can work with it. It'll pull in different data depending on which enable functions we called it setup; in our case here that means we'll now have fresh depth and RGB images to work with.

Frame Rates

The Kinect camera captures data at a rate of 30 frames per second. In other words, every 1/30th of a second, the Kinect makes a new depth and RGB image available for us to read. If our app runs faster than 30 frames a second, the draw function will get called multiple times before a new set of depth and RGB images is available from the Kinect. If our app runs slower than 30 frames a second, we'll miss some images.

But how fast does our app actually run? What is our frame rate? The answer is that we don't know. By default, Processing simply runs our draw function as fast as possible before starting over and doing it again. How long each run of the draw function takes depends on a lot of factors including what we're asking it to do and how much of our computer's resources are available for Processing to use. For example, if we had an ancient really slow computer and we were asking Processing to print out every word of Dickens' A Tale of Two Cities on every run of the draw function, we'd likely have a

very low frame rate. On the other hand, when running Processing on a typical modern computer with a draw loop that only does some basic operations, we might have a frame rate significantly above 30 frames per second. And further, in either of these situations our frame rate might vary over time both as our app's level of exertion varied with user input and the resources available to it varied with what else was running on our computer.

For now in these beginning examples you won't have to worry too much about the frame rate, but as we start to build more sophisticated applications this will be a constant concern. If we try to do too much work on each run of our draw function, our interactions may get slow and jerky, but if we're clever we'll be able to keep all our apps just as smooth as this initial example.

We're down to the last two lines, which are the heart of this example. Let's take the first one, line 17, first:

```
image(kinect.depthImage(), 0, 0);
```

Starting from the inside out, we first call `kinect.depthImage()` which asks the library for the most recently available depth image. This image is then handed to Processing's built-in `image()` function along with two other arguments both set to zero. This tells processing to draw the depth image at `0,0` in our sketch, or at the very top left of our app's window.

Line 18 does nearly the same exact thing except with two important differences:

```
  image(kinect.rgbImage(), 640, 0);
}
```

It calls `kinect.rgbImage()` to get the color image from the Kinect and it passes `640,0` to `image()` instead of `0,0`, which means that it will place the color image at the top of the app's window, but 640 pixels from the left side. In other words, the depth image will occupy the leftmost 640 pixels in our app and the color image the rightmost ones.

One more note about how these lines work. By calling `kinect.depthImage()` and `kinect.rgbImage()` inline within the arguments to `image()` we're hiding one important part of how these functions work together: we're never seeing the return value from `kinect.depthImage()` or `kinect.rgbImage()`. This is an elegant and concise way to write a simple example like this, but right now we're trying for understanding rather than elegance, so we might learn something by rewriting our examples like this:

```
import SimpleOpenNI.*;
SimpleOpenNI  kinect;

void setup()
{
  // double the width to display two images side by side
  size(640*2, 480);
  kinect = new SimpleOpenNI(this);

  kinect.enableDepth();
```

```
  kinect.enableRGB();
}

void draw()
{
  kinect.update();

  PImage depthImage = kinect.depthImage();
  PImage rgbImage = kinect.rgbImage();

  image(depthImage, 0, 0);
  image(rgbImage, 640, 0);
}
```

In this altered example, we've introduced two new lines to our sketch's draw function. Instead of implicitly passing the return values from `kinect.depthImage()` and `kinect.rgbImage()` to Processing's image function, we're now storing them in local variables and then passing those variables to image. This has changed the functionality of our sketch not at all and if you run it you'll see no difference in the behavior. What it has done is make the return type of our two image-accessing functions explicit: both `kinect.depthImage()` and `kinect.rgbImage()` return a PImage, Processing's class for storing image data. This class provides all kinds of useful functions for working with images such as the ability to access the image's individual pixels and to alter them, something we're going to be doing later on in this chapter. Having the Kinect data in the form of a PImage is also a big advantage because it means that we can automatically use the Kinect data with other libraries that don't know anything at all about the Kinect but do know how to process PImages.

Project 3: Looking at a Pixel

At this point, we've worked our way completely through this first basic example. You've learned how to include the Kinect SimpleOpenNI library in a Processing sketch, how to turn on access to both the RGB and depth images, how to update these images from the Kinect and how to draw them to the screen.

Now that we've got all that under our belt, it's time to start learning more about the individual pixels that make up these images. I told you in the intro to this section that the color of each pixel in the depth image represents the distance of that part of the scene. And, on the other hand, each pixel of the color image will have three components, one each for red, green, and blue. Since we've now got an app running that displays both of these types of images, we can make a simple addition to it that will let us explore both sets of pixels so we can start to get some hands-on experience with them. Once you're starting to feel comfortable with individual pixels, we'll move on to operations that process every pixel in an image so that we can do things like finding the closest part of the depth image. And once we're at that point, we'll be ready to build our first truly interactive projects with the Kinect.

Let's grab some pixels! We'll start by modifying our existing sketch to give us some more information about the pixels of the depth and color images. Specifically, we're going to make it so that whenever you click on one of these images, Processing will print out information about the particular pixel you clicked on.

Here's the code:

```
import SimpleOpenNI.*;
SimpleOpenNI  kinect;

void setup()
{
  size(640*2, 480);
  kinect = new SimpleOpenNI(this);

  kinect.enableDepth();
  kinect.enableRGB();
}

void draw()
{
  kinect.update();

  PImage depthImage = kinect.depthImage();
  PImage rgbImage = kinect.rgbImage();

  image(depthImage, 0, 0);
  image(rgbImage, 640, 0);
}

void mousePressed(){
  color c = get(mouseX, mouseY);
  println("r: " + red(c) + " g: " + green(c) + " b: " + blue(c));
}
```

The new lines are the last four at the bottom, the `mousePressed()` function. Add that to your sketch and run it. Arrange the windows on your computer so that you can see your running sketch as well as the Processing development environment at the same time. Click around in a few different places in the depth image and the color image. You should see a line of text appear in the output area at the bottom of the Processing editor every time you click. This is the output from our new mousePressed function. Experiment a bit until you've gotten a feel for these numbers and then come back here and I'll explain this new function, the data it's showing us, and what we might learn from it.

As you've probably figured out by now, this new function gets called whenever you click your mouse within the running Processing app. You've probably seen something like this before, but I'll go through this example just in case you haven't and also so you understand exactly what data we're seeing here about the depth and color images.

Our implementation of the mousePressed function does two things: it figures out the color of the pixel you click on and it prints out some information about that pixel.

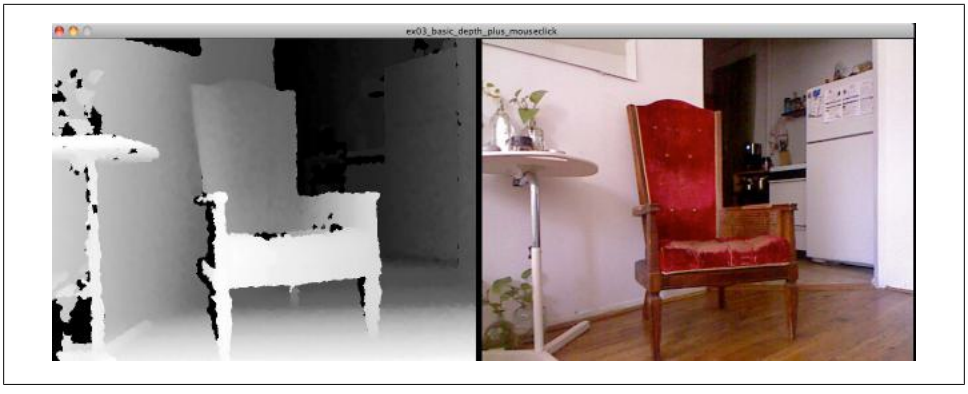


Figure 2-11. Depth and RGB images of the old chair where I wrote this book. Clicking around on this image in our sketch will reveal its depth values.

Unsurprisingly, then, mousePressed only has two lines in it, one to accomplish each of these things. Let's look at how they work.

The first line of `mousePressed()` calls `get()`, which is a function that Processing provides to let us access the value of any pixel that we've already drawn on the screen. In this case, all the pixels that we've drawn to the screen are from the Kinect's depth and color images, so that's what we'll be accessing. This function takes two arguments, the x- and y-coordinates of the pixel whose value we'd like to view. In this case, we want to inspect a different pixel with every click (whichever pixel happens to be under the mouse) so instead of using constants for x and y, we use `mouseX` and `mouseY`, which are special variables that Processing automatically pre-populates with the current coordinates of the mouse. Calling `get()` within the `mousePressed()` function with these arguments will tell us exactly the color of the pixel that the user just clicked on. And, since `get()` returns a color, we store its return value in a local variable, `c`, that is declared as a `color`, the Processing type for storing, you guessed it, colors.

Once we've got that color, all we have left to do is print out some information about it. And we know from our discussion about pixels at the start of this chapter that we want the red, green, and blue components if those are available. So the second line of `mousePressed()` calls more built-in Processing functions to extract each of these component values from our color. We call `red()`, `green()`, and `blue()` one at a time, passing `c` as an argument, the color we extracted from the clicked pixel. Once we've got those values, we just combine them together into one long string (using `"+"`) that we can print to the Processing output window with `println()`.

Now that we've got the ability to print out information about any pixel in either the depth or color image at will, let's use it to do some concrete experiments to see how the pixels work in each of these types of images. We'll start with the color image.

Figure 2-11 shows a screen capture I took of this new version of the app.

It shows my favorite old beaten-up red chair where I wrote much of this book. I'll use this image to illustrate the pixel values shown by our new `mousePressed` function. You can follow along with whatever your own Kinect is seeing.

Color Pixels

I started by clicking on the color image right in the middle of the chair's back, the part that's been worn shiny by my squirming against it. When I clicked there, I saw the following print out in the Processing output area: `r: 124.0 g: 3.0 b: 9.0`. Since the pixel under my mouse was part of a color image, the `red()`, `blue()`, and `green()` functions each extracted a different component of the pixel, telling me out of a range of 0-255 how much red, blue, and green there was in the image.

The resulting data has a couple of surprises. First of all, to my eye the back of the chair where I clicked looks almost perfectly red, but the red component of the pixel was only 124 or less than half the possible total of 255. This shows a key difference between how we see as people with eyes and the pixel values that a computer calculates by processing the data from a digital camera. As people our impressions of what we see are relative. We focus on the differences between the parts of whatever we're seeing rather than their absolute levels. We think the back of the chair is fully red not because it has a red value close to some theoretical absolute limit, but because its red value is so much higher than its green and blue values. With just 3 for green and 9 for blue, that 124 for red looks like pure red to us.

Now, let's look at a couple of other parts of the color image to explore these ideas further before turning our attention to the depth image. To my eye, the brightest whitest part of the image is the strip of wall on the right side of the image. When I click there I get these pixel components printed out by the sketch: "r: 245.0 g: 238.0 b: 245.0". So, what does white look like in a color image? A perfectly white pixel would be one that has all three of the red, green, and blue components at values of 255. This pixel on the white wall is nearly there, it has very high values for all three components and all three components are nearly equal.

Let's compare this perfectly white wall with the slightly dimmer refrigerator door just behind it. When I click there I get `r: 182.0 g: 165.0 b: 182.0`. These values are still very close to equal with each other, but are now each slightly further away from 255.

So, what have we learned about color pixels? First of all, we've learned that color itself comes from there being a difference between the individual color components. A pixel that has all three of its components nearly equal will look white or some shade of gray. Further, while it's the relative levels of the color components that determines which color we perceive, it's the sum total between the levels of all three of the pixels that determines its brightness. The white that had all three values close to 255 was brighter than the one that had all three values clustered around 180.

Let's look at an even darker part of the image to get one more interesting piece of data. When I click on the green curtains that are deep in the shade of my kitchen, I get pixel values that look like this: `r: 15.0 g: 6.0 b: 3.0`. From what we just learned about brightness, we might've expected to see very low numbers like these for each of the pixel components. But, it's surprising that the red pixel has the highest value of the three when the curtains are obviously green or, in their darkest parts, black. What's going on here? Well, if you look at the image for awhile, or if you click around on other parts that appear to be white or neutral (for example the white wall behind the chair), you'll find that the whole image actually has a reddish or purplish hue. That is, the red and blue components of each pixel will always have slightly higher values than we'd expect.

Again, a color shift like this is one of those qualities that it's very easy for our eye to ignore, but makes a really big impact on the pixel values that our Processing code will see when looking at a color image. You can look at this picture of those curtains and in an instant know that they were green. Your brain automatically translates away things like the darker lighting and the red-shifted image. It's not confused by them. But imagine if I gave you the assignment of writing a program to figure out the color of those curtains. Imagine if I gave you all the pixels that corresponded to the curtain and you had to use only their component color values to identify the curtain's real color. Since most of the pixels have very low red, green, and blue values and many of them have red as their leading component, nearly any approach that you came up with would end up concluding that the curtains were black with shades of red.

In order to get a better answer you'd need access to all of the pixels in the image in order to compare the colors in different areas (so you could do things like detect the red shift) and you might even need a lot of data from other similar images so you'd know what other kinds of distortions and aberrations to expect. In the end you'd be faced with the extremely difficult problem of inventing a whole system of heuristics, assumptions, and fiendishly complex algorithms in an attempt to extract the kind of information our brain can do effortlessly on seeing nearly any image. And even if you went ahead and got a degree in Computer Vision, the academic field dedicated to doing exactly this kind of thing, you could still only achieve limited success while working with badly-lit, low resolution, color images like these.

Conventional color images are just fundamentally ill-suited to being processed by computer programs. Sure applications like Photoshop can manipulate and alter these types of images in a million ways to change their appearance, but trying to write programs that understand these images, that extract from them something like our own understanding of our physical environment (how far away things are from us, where one object ends and another begins, etc.), is a fool's errand. In fact, color images are so bad for this task that we're basically never going to look at them again in this book. (The only exceptions to this will be when we're talking about ways of aligning color images with depth images so that we can apply realistic color to our visualizations after we've done the heavy lifting by processing the depth data.)

Depth Pixels

If you were a computer vision researcher, frustrated by these limitations you'd quickly find yourself wanting a whole different kind of image that didn't suffer from these kinds of problems. You'd want something like a depth image.

Thankfully even though we're not computer vision researchers (or, at least we're only amateurs), we do happen to have a depth image sitting right here in our Processing sketch next to our color image. Let's click around on that image some to get a feel for its pixel values and how they might be easier to work with than those we've seen so far.

I'll start at the same place I started in the color image: by clicking on the back of my comfy red chair. Only this time I'll obviously make my click on the left side of the sketch, within the depth image, rather than in the color image. When I click on the back of my chair in the depth image our mousePressed code prints out the following output: `r: 96.0 g: 96.0 b: 96.0`.

A couple of things should jump out at you immediately about this. In contrast to any of the color pixels we looked at, all three of the components of this pixel are exactly the same. Not nearly the same as was the case in the parts of the white wall we examined, but exactly the same. After our discussion of color pixels above this should make total sense. We know that the color of a pixel is determined by the difference between its red, blue, and green components. Since the depth image is grayscale, its pixels don't have color and hence its components are all exactly identical. A grayscale pixel does, however, have brightness. In fact, brightness is just about all a grayscale pixel has since in this case `brightness` really means "distance between white and black".

As you saw earlier, brightness is determined by the combination of all three of the pixel components. And since all three components of each grayscale pixel are always going to be exactly equal, the brightness of a grayscale pixel will be equal to this same value we're seeing for each of the components, in the case of this pixel from the back of my chair, 96. In fact, Processing actually has a `brightness()` function that we can use instead of `red()`, `blue()`, or `green()` that will give us the exact same result as any of these when used on grayscale images. As we go forward and start to work more exclusively with depth images, we'll use `brightness()` instead of `red()`, `green()`, or `blue()` to access the value of pixels because it's silly to talk about the red or green component of an image which is obviously black and white.

Now, let's click around on a few other parts of the depth image to see how the brightness of these depth pixels vary. If I click on those pesky dark green window curtains, I get a brightness reading of 8.0 (technically, Processing prints out `r: 8.0 g: 8.0 b: 8.0`, but we now know that we can summarize that as brightness). In the depth image the curtains look nearly completely black. They are, in fact, so dark that they're actually hard to even distinguish from their surroundings with the naked eye. However if I click around that area of the image, I get a different brightness reading each time. There's detailed depth information even in parts of the image where the eye can't see different shades of gray.

Let's look at a part of the depth image that has some dramatic difference from the color images: my comfy old red chair. By comparing some depth and color readings from different parts of the chair we can really start to see the advantages of the depth image in overcoming the limitations of the color image we so vividly encountered above.

If I click on different parts of the chair in the color image, the pixel values I measure don't change that much. For example, a pixel from the front of the chair (the part of the seat that's facing forwards towards the Kinect) has components: `r: 109.0 g: 1.0 b: 0.0` while a pixel from low on the chair's back, just above the seat has components: `r: 112.0 g: 4.0 b: 2.0`. These values are barely different from each other and if I clicked around more in each of these parts of the image, I'd certainly find a pixel reading `r: 109.0 g: 1.0 b: 0.0` on the chair's back and vice versa. There's nothing in these color pixels that tells me that the chair's seat and its back are physically or spatially different.

But what about depth pixels from the same parts of the chair? Clicking on the front of the chair in the depth image, I get a pixel value of 224.0 and on the lower back I get one of 115.0. Now these are dramatically different. Unlike with the color pixels, there's a big jump in values that corresponds between these depth pixels. This jump corresponds clearly to the two or three feet of depth between the front and back of this chair. Imagine I gave you a similar assignment as the one above with the green curtain, but this time I handed you all the pixels representing the chair and asked you to separate it in space, to distinguish the seat from the back, only this time I gave you depth pixels. This time, you'd be able to do it. The pixels on the front of the chair all cluster between 220 and 230 while the pixels on the back of the chair range from about 90 to 120. If you sorted the pixels by brightness, there'd be a big jump at some point in the line-up. All the pixels above that jump would be part of the front of the chair and all the pixels below it would be part of the back. Easy.

For basic problems like this the information in the depth image is so good that we don't need any super sophisticated computer vision algorithms in order to do useful work. Basic logic like looking for gaps in the pixels' brightness or (like we'll see extensively later in this chapter) looking for the brightest pixel can achieve surprisingly complex and effective applications. And just imagine what we could do once we start using some of those super sophisticated algorithms, which we'll dive into in the next chapter.

Converting to Real World Distances

But what about these pixels' actual depth? We know that the color of each pixel in the depth image is supposed to relate to the distance of that part of the scene, but how far away is 96 or 115 or 224? How do we translate from these brightness values to a distance measurement in some kind of useful units like inches or centimeters?

Well, let's start with some basic proportions. We know from our experiments earlier in this chapter that the Kinect can detect objects within a range of about 20 inches to

25 feet. And we know that the pixels in our depth image have brightness values that range between 255 and 0. So, logically, it must be the case that pixels in the depth image that have a brightness of 255 must correspond to objects that are 20 inches away, pixels with a value of 0 must be at least 25 feet away, and values in between must represent the range in between, getting further away as their values descend towards 0.

So, how far away is the back of our chair with the depth pixel reading of 96? Given the logic just described, we can make an estimate. Doing a bit of back-of-the-envelope calculation tells me that 96 is approximately 37% of the way between 0 and 255. Hence the simplest possible distance calculation would hold that the chair back is 37% of the way between 25 feet and 20 inches or about 14 and a half feet. This is clearly not right. I'm sitting in front of the chair now and it's no more than eight or 10 feet away from the Kinect.

Did we go wrong somewhere in our calculation? Is our Kinect broken? No. The answer is much simpler. The relationship between the brightness value of the depth pixel and the real world distance it represents is more complicated than a simple linear ratio. There are really good reasons for this to be the case.

First of all consider the two ranges of values being covered. As we've thoroughly discussed, the depth pixels represent grayscale values between 0 and 255. The real world distance covered by the Kinect, on the other hand, ranges between 20 inches and 25 feet. Further, the Kinect's depth readings have millimeter precision. That means they need to report not just a number ranging from zero to a few hundred inches, but a number ranging from 0 to around 8000 millimeters in order to cover the entirety of the distance the Kinect can see. That's obviously a much larger range than can fit in the bit depth of the pixels in our depth image.

Now, given these differences in range, we could still think of some simple ways of converting between the two. For example, it might have occurred to you to use Processing's `map()` function which scales a variable from an expected range of input values to a different set of output values. However, using `map()` is a little bit like pulling on a piece of stretchy cloth to get it to cover a larger area: you're not actually creating more cloth, you're just pulling the individual strands of the cloth apart, putting space between them to increase the area covered by the whole. If there was a pattern on the surface of the cloth it would get bigger, but you'd also start to see spaces within it where the strands separated. Processing's `map()` function does something similar. When you use it to convert from a small range of input values to a larger output, it can't create new values out of thin air, it just stretches the existing values out so that they cover the new range. Your highest input values will get stretched up to your highest output value and your lowest input to your lowest output, but just like with the piece of cloth there won't be enough material to cover all the intermediate values. There will be holes. And in the case of mapping our depth pixels from their brightness range of 0 to 255 to the physical range of 0 to 8000 millimeters, there will be a lot of holes. Those 256 brightness values will only cover a small minority of the 8000 possible distance values.

In order to cover all of those distance values without holes, we'd need to access the depth data from the Kinect in some higher resolution form. As I mentioned in the introduction, the Kinect actually captures the depth information at a resolution of 11 bits per pixel. This means that these raw depth readings have a range of 0 to 2047, much better than the 0 to 255 available in the depth pixels we've looked at so far. And, the SimpleOpenNI library gives us a way to access these raw depth values.

But, wait! Have we been cheated? Why don't the depth image pixels have this full range of values? Why haven't we been working with them all along?

Remember the discussion of images and pixels at the start of this chapter? Back then, I explained that the bigger of a number we use to store each pixel in an image the more memory it takes to store that image and the slower any code runs that has to work with it. Also, it's very hard for people to visually distinguish between more shades of gray than that anyway. Think back to our investigations of the depth image. There were areas back of the scene that looked like flat expanses of black pixels, but when we clicked around on them to investigate it turned out that even they had different brightness values, simply with differences that were too small for us to see. For all of these reasons, all images in Processing have pixels that are 8-bits in depth, i.e. whose values only range from 0 to 255. The PImage class we've used in our examples that allows us to load images from the Kinect library and display them on the screen enforces the use of these smaller pixels.

What it really comes down to is this: when we're displaying depth information on the screen as images we make a set of trade-offs. We use a rougher representation of the data because it's easier to work with and we can't really see the differences anyway. But now that we want to use the Kinect to make precise distance measurements, we want to make a different set of trade-offs. We need the full resolution data in order to make more accurate measurements, but since that's more unwieldy to work with, we're going to use it more sparingly.

Let's make a new version of our Processing sketch that uses the higher resolution depth data to turn our Kinect into a wireless tape measure. With this higher resolution data we actually have enough information to calculate the precise distance between our Kinect and any object in its field of view and to display it in real world units like inches and millimeters.

Project 4: A Wireless Tape Measure

The code in this section is going to introduce a new programming concept that I haven't mentioned before and that might not be completely familiar to you from your previous work in Processing: accessing arrays of pixels. Specifically, we'll be learning how to translate between a one dimensional array of pixels and the two dimensional image that it represents. I'll explain how to do all of the calculations necessary to access the entry in the array that corresponds to any location in the image. And, more importantly,

I'll explain how to think about the relationship between the image and the array so that these calculations are intuitive and easy to remember.

At first this discussion of pixels and arrays may feel like a diversion from our core focus on working with the Kinect. But the Kinect is, first and foremost, a camera, so much of our work with it will be based on processing its pixels.

Up to this point, whenever we've wanted to access the Kinect's pixels, we've first displayed them on the screen in the form of images. However, as we just discussed, this becomes impractical when we want to access the Kinect's data at a higher resolution or when we want to access more of it than a single pixel's worth. Therefore, we need to access to the data in a manner that doesn't first require us to display it on the screen. In Processing (and most other graphics programming environments) this image data is stored as arrays of pixels behind the scenes. Accessing these arrays of pixels directly (while they are still off stage) will let our programs run fast enough to do some really interesting things like working with the higher resolution data and processing more than one pixel at a time.

Even though Processing stores images as flat arrays of pixels, we still want to be able to think of them two-dimensionally. We want to be able to figure out which pixel a user clicked on or draw something on the screen where we found a particular depth value. In this section, I'll teach you how to make these kinds of translations. We'll learn how to convert between the array the pixels are stored in and their position on the screen.

In this section, I'll introduce this conversion technique by showing you how to access an array of higher resolution values from the Kinect. Then we'll use that to turn the Kinect into a wireless tape measure, converting these depth values into accurate real world units. Once we've got this down, we'll be ready to take things one step further and start working with all of the pixels coming from the Kinect.

We'll start our tape measure with a new version of our Processing sketch. This version is going to be very much along the same lines as the sketch we've been working with but with a few important differences. The basic structure is going to be the same. We'll still display images from the Kinect and then output information about them when we click, but we're going to change what we display, both on the screen and in the output when we click.

First of all we're going to forget about the color image from the Kinect. We've learned all that it has to teach us for now and so we're banishing it to focus more on the depth image. Secondly, we're going to rewrite our `mousePressed()` function to access and display the higher resolution depth data from the Kinect. I'll explain how this works in some detail, but first take a look at the full code, noticing the changes to `setup()` and `draw()` that come from eliminating the color image:

```
import SimpleOpenNI.*;
SimpleOpenNI  kinect;
```

```
void setup()
{
  size(640, 480);
  kinect = new SimpleOpenNI(this);
  kinect.enableDepth();
}

void draw()
{
  kinect.update();

  PImage depthImage = kinect.depthImage();
  image(depthImage, 0, 0);
}

void mousePressed(){
  int[] depthValues = kinect.depthMap();
  int clickPosition = mouseX + (mouseY * 640);
  int clickedDepth = depthValues[clickPosition];

  float inches = clickedDepth / 25.4;

  println("inches: " + inches);
}
```

The changes to `setup()` and `draw()` are minimal: I'm no longer accessing the RGB image from the Kinect and no longer displaying it. And since we're now only displaying one image, I went ahead and made the whole sketch smaller because we don't need all that horizontal real estate just to show the depth image.

Now, let's talk about the real substantial difference here: the changes I've made to mousePressed. First of all, mousePressed calls a new function on our `kinect` object that we haven't seen before: `depthMap()`. This is one of a few functions that SimpleOpenNI provides that give us access to the higher resolution depth data. This is the simplest one. It returns all of the higher resolution depth values unmodified — neither converted or processed.

In what form does `kinect.depthMap()` return these depth values? Up until now all the depth data we've seen has reached us in the form of images. We know that the higher resolution values that `kinect.depthMap()` returns can't be stored as the pixels of an image. So, then, in what form are they stored? The answer is: an array of integers. We have one integer for each depth value that the Kinect recorded and they're all stored in one array. That's why the variable we use to save the results of `kinect.depthMap()` is declared thusly: `int[] depthValues`. That `int[]` means that our `depthValues` variable will store an array of integers. If you have a hard time remembering how array declarations like this one work in Processing (as I often do), you can think of the square brackets as being a box that will contain all the values of the array and the "int" that comes before it as a label telling us that everything that goes in this box must be an integer.

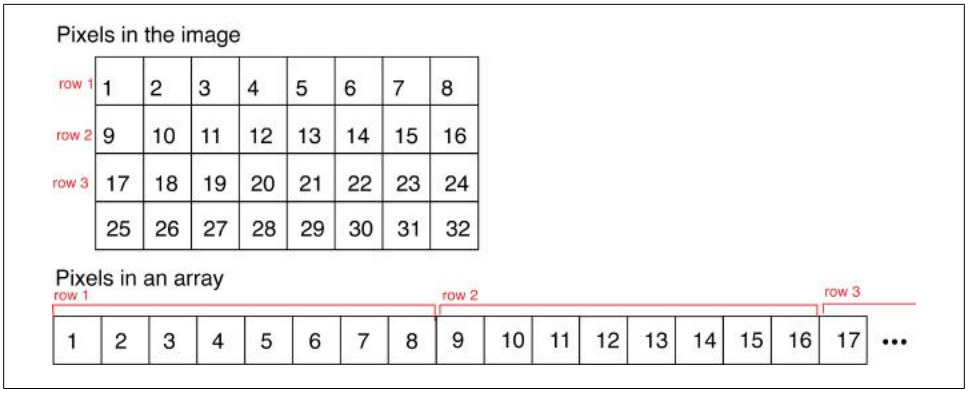


Figure 2-12. Pixels in a two-dimensional image get stored as a flat array. Understanding how to split this array back up into rows is key to processing images.

So, we have an array of integers. How can this box full of numbers store the same kind of information we've so far seen in the pixels of an image? The Kinect is, after all, a camera. The data that comes from it is two dimensional, representing all the depth values in its rectangular field of view, whereas an array is one dimensional, it can only store a single stack of numbers. How do you represent an image as a box full of numbers?

Here's how. Start with the pixel in the top-leftmost corner of the image. Put it in the box. Then, moving to the right along the top row of pixels, put each pixel into the box on top of the previous ones. When you get to the end of the row jump back to left side of the image, move down one row, and repeat the procedure, continuing to stick the pixels from the second row on top of the ever-growing stack you began in the first row. Continue this procedure for each row of pixels in the image until you reach the very last pixel in the bottom right. Now, instead of a rectangular image, you'll have a single stack of pixels: a one-dimensional array. All the pixels from each row will be stacked together and the last pixel from each row will be right in front of the first pixel from the next row, as Figure 2-12 shows.

This is exactly how the array returned by `kinect.depthMap()` is structured. It has one high resolution depth value for each pixel in the depth image. Remember that the depth image's resolution is 640 by 480 pixels. That means that it has 480 rows of pixels each of which is 640 pixels across. So, from the logic above, we know that the array returned by `kinect.depthMap()` contains 307,200 (or 640 times 480) integers arranged in a single linear stack. The first integer in this stack corresponds to the top left pixel in the image. Each following value corresponds to the next pixel across each row until the last value finally corresponds to the last pixel in the bottom right.

But how do we access the values of this array? More specifically, how do we pull out the integer value that corresponds to the part of the image that the user actually clicked on? This is the mousePressed event, after all, and so all we have available to us is the position of the mouse at the time that the user clicked. As we've seen, that position is

expressed as an x-y coordinate in the variables `mouseX` and `mouseY`. In the past versions of the sketch, we used these coordinates to access the color value of a given pixel in our sketch using `get()`, which specifically accepted x-y coordinates as its arguments. However, now we have a stack of integers in an array instead of a set of pixels arranged into a rectangle. Put another way, instead of having a set of x-y coordinates in two axes, we only have a single axis: the integers in our single array. In order to access data from the array we need not a pair of x-y coordinates, but an "index", a number that tells us the position in the array of the value we're looking for. How do we translate from the two axes in which the depth image is displayed and the user interacts with it to the single axis of our integer array? In other words, how do we convert mouseX and mouseY into the single position in the array that corresponds to the the user's click?

To accomplish this we'll have to do something that takes into account how the values were put into the array in the first place. In filling the array, we started at the top left corner of the image, went down each pixel in each row to the end adding values and then jumped back to the beginning of the next row when we reached the edge of the image. Imagine that you were counting values as we did this, adding one to your count with each pixel that got converted into a value and added to the array. What would your count look like as we progressed through the image?

For the first row, it's pretty obvious. You'd start your count at zero (programmers always start counting at zero) and work your way up as we went across the first row. When we reached the last pixel in the first row, your count would be 639 (there are 640 pixels in the row and you started counting at 0 for the first pixel). Then, when we jumped back to the left side of the image to continue on the second row, you'd keep counting. So pixel one on row two would be 640, pixel two would be 641 and so on until you reached the end of row two. At the last pixel of row two, you'd be up to 1279, which means that the first pixel in row three would be 1280. If you continued on for another row, you'd finish row three at 1919 and the first pixel of row four would be 1920.

Notice how the first pixel of every row is always a multiple of 640? If I asked what the number would be for the first pixel in the 20th row in the image, instead of counting, you could just multiply: 640 times 20 is 12,800. In other words, the number for the first pixel in each row is the width of the image (i.e. 640) multiplied by which row we're on (i.e. how far down we are from the top of the image).

Let's come back to our mousePressed function for a second. In that function we happen to have a variable that's always set to exactly how far down the mouse is from the top of the image: mouseY. Our goal is to translate from mouseX and mouseY to the number in our count corresponding to the pixel the mouse is over. With mouseY and the observation we just made, we're now halfway there. We can translate our calculation of the first pixel of each row to use mouseY: mouseY times 640 (the width of the row) will always get us the value of the array corresponding to the first pixel in the row.

But what about all the other pixels? Now that we've figured out what row a pixel is in how can we figure out how far to the left or right that pixel is in the row? We need to take mouseX into account.

Pick out a pixel in the middle of a row, say row 12. Imagine that you clicked the mouse on a pixel somewhere in this row. We know that the pixel's position in the array must be greater than the first pixel in that row. Since we count up as we move across rows, this pixel's position must be the position of the first pixel in its row plus the number of pixels between the start of the row and this pixel. Well, we happen to know the position of the first pixel on the previous row. It's just 12 times 640, the number of the row times the number of pixels on each row. But what about the number of pixels to the left of the pixel we're looking at? Well, in mousePressed, we have a variable that tells us exactly how far the mouse is from the left side of the sketch: mouseX. All we have to do is add mouseX to the value at the start of the row: mouseY times 640.

And, lo and behold, we'e now got our answer. The position in the array of a given pixel will be: `mouseX + (mouseY * 640)`. If at any point in this circuitous discussion you happened to peek at the next line in `mousePressed()`, you would have ruined the surprise because look what that line does: performs this exact calculation:

```
int clickPosition = mouseX + (mouseY * 640);
```

And then the line after that uses its result to access the array of depthValues to pull out the value at the point where the user clicked. That line uses `clickPosition`, the result of our calculation, as an index to access the array. Just like `int[] depthValues` declared `depthValues` as an array—a box into which we could put a lot of integers—`depthValues[clickPosition]` reaches into that box and pulls out a particular integer. The value of clickPosition tells us how far to reach into the box and which integer to pull out.

Higher Resolution Depth Data

That integer we found in the box is one of our new higher resolution depth values. As we've been working towards all this time, it's exactly the value that corresponds to the position in the image where the user clicked. Once we've accessed it, we store it in another variable `clickedDepth` and use that to print it to Processing's output window.

If you haven't already, run this sketch and click around on various parts of the depth image. You'll see values printing out to the Processing output area much like they did in all of our previous examples, only this time they'll cover a different range. When I run the sketch, I see values around 450 for the brightest parts of the image (i.e. the closest parts of the scene) and around 8000 for the darkest (i.e. furthest) parts. The parts of the image that are within the Kinect's minimum range or hidden in the shadows of closer images give back readings of zero. That's the Kinect's way of saying that there is no data available for those points.

This is obviously a higher range than the pixel values of zero to 255 we'd previously seen. In fact, it's actually spookily close to the 0 to 8000 range we were hoping to see

to cover the Kinect's full 25 foot physical range at millimeter precision. This is extremely promising for our overall project of trying to convert the Kinect's depth readings to accurate real world measurements. In fact, it sounds an awful lot like the values we're pulling out of `kinect.depthMap()` are the accurate distance measurements in millimeters. In other words, each integer in our new depth readings might actually correspond to a single millimeter of physical distance.

With a few alterations to our mousePressed function (and the use of a handy tape measure) we can test out this hypothesis. Here's the new version of the code:

```
import SimpleOpenNI.*;
SimpleOpenNI  kinect;

void setup()
{
  size(640, 480);
  kinect = new SimpleOpenNI(this);

  kinect.enableDepth();
}

void draw()
{
  kinect.update();

  PImage depthImage = kinect.depthImage();

  image(depthImage, 0, 0);
}

void mousePressed(){
  int[] depthValues = kinect.depthMap();
  int clickPosition = mouseX + (mouseY * 640);

  int millimeters = depthValues[clickPosition];
  float inches = millimeters / 25.4;

  println("mm: " + millimeters + " in: " + inches);
}
```

First of all, I renamed our `clickDepth` variable to `millimeters` since our theory is that it actually represents the distance from the Kinect to the object clicked as measured in millimeters. Second, I went ahead and wrote another line of code to convert our millimeter reading to inches. Being American, I think in inches so it helps me to have these units on hand as well. A few seconds Googling taught me that to convert from millimeters to inches, all you have to do is divide your value by 25.4. Finally, I updated the `println()` statement to output both the millimeter and inch versions of our measurement.

Once I had this new code in place, I grabbed my tape measure. I put one end of it under the Kinect and extended it towards myself as you can see in Figure 2-13.

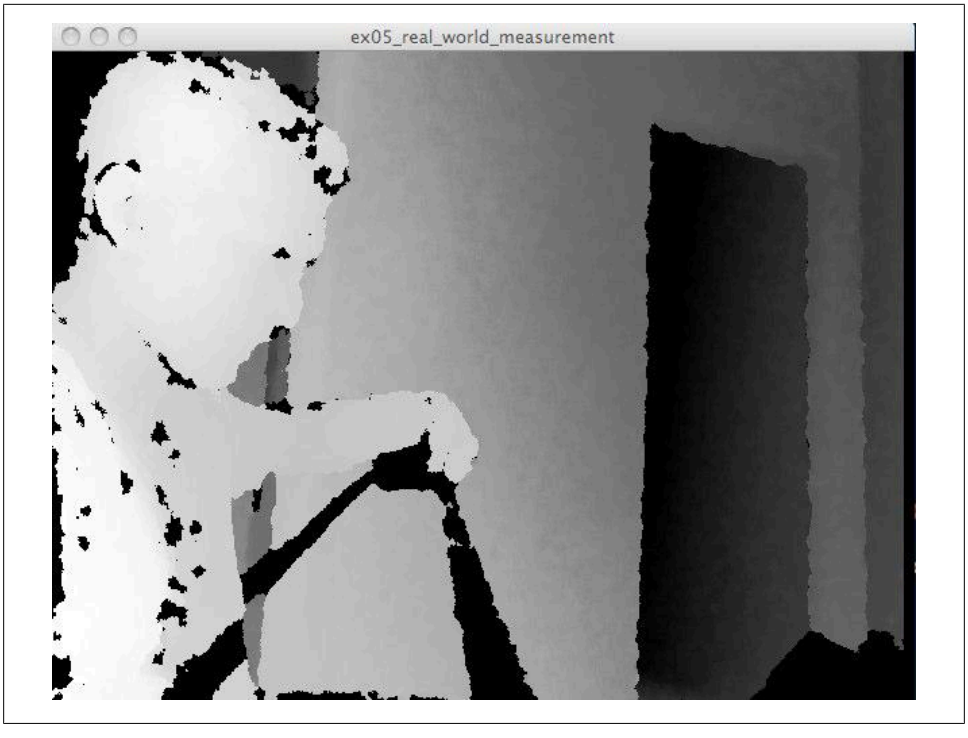


Figure 2-13. I held up a tape measure in front of my Kinect to check our depth measurements against the real world.

The tape shows up as a black line because most of it is inside of the Kinect's minimum range and because all of it is reflective. Once I had the tape measure extended, I locked it down at 32 inches (or about 810 millimeters). Then I could use my free hand to click on the depth image to print out measurements to the Processing output area. It was a little bit hard to distinguish between my hand and the tape measure itself so I just clicked in that general vicinity. When I did that, Processing printed out: `mms: 806 in: 31.732285`. Dead on! Taking into account the sag in the measuring tape as well as my poorly aimed clicking, this is an extremely accurate result. And more clicking around at different distances confirmed it: our distance calculations lined up with the tape measure every time. We've now turned out Kinect into an accurate digital "tapeless" measuring tape!

Try it out yourself. Get out a tape measure, run this sketch, and double check my results. Then, once you've convinced yourself that it's accurate, use the Kinect to take some measurements of your room, your furniture, your pets, whatever you have handy.

In this section, you learned two fundamental skills: how to access the Kinect's data as an array of values and how to translate between that array and the position of a particular value in the image.

We're now going to extend those skills to let us work with all of the depth data coming from the Kinect. Instead of translating from a single x-y coordinate to a single array index, we're going to loop through all of the values in the array in order to make general conclusions about the depth data by comparing all of the values. This technique will let us do things like finding and tracking the closest part of the image. At that point we'll be ready to use the Kinect to build our first real user interfaces. We'll be able to start doing something more interesting than printing numbers in Processing's output area.

This chapter will conclude with a couple of projects that explore some of the possibilities that are opened up by our ability to track the closest point. We'll write a sketch that lets us draw a line by waving our hands and other body parts around. Then we'll go even further and make another sketch that lets us lay out photos by dragging them around in midair Minority Report-style.

Project 5: Tracking the Nearest Object

In order to build useful interactive applications with the Kinect, we need to write sketches that respond to the scene in a way that people find intuitive and clear. People have an inherent instinct for the objects and spaces around them. When you walk through a doorway, you don't have to think about how to position yourself so you don't bump into the doorframe. If I asked you to extend your arm towards me or towards some other object, you could do it without thinking. When you walk up to an object, you know when it's within reach before you even extend your arm to pick it up.

Because people have such a powerful understanding of their surrounding spaces, physical interfaces are radically easier for them to understand than abstract screen-based ones. And this is especially true for physical interfaces that provide simple and direct feedback where the user's movements immediately translate into action.

Having a depth camera gives us the opportunity to provide interfaces like these without needing to add anything physical to the computer. With just some simple processing of the depth data coming in from the Kinect, we can give the user the feeling of directly controlling something on the screen.

The simplest way to achieve this effect is by tracking the point that is closest to the Kinect. Imagine that you're standing in front of the Kinect with nothing between you and it. If you extend your arm towards the Kinect then your hand will be the closest point that the Kinect sees. If our code is tracking the closest point then suddenly your hand will be controlling something on screen. If our sketch draws a line based on how you move your hand, then the interface should feel as intuitive as painting with a brush. If your sketch moves photos around to follow your hand it should feel as intuitive as a table covered in prints.

Regardless of the application, though, all of these intuitive interfaces begin by tracking the point closest to the Kinect. How would we start going about that? What are the

steps between accessing the depth value of a single point and looking through all of the points to find the closest one? Further all of these interfaces translate the closest point into some kind of visible output. So, once we've found the closest point how do we translate that position into something that the user can see?

Later in this chapter, you're going to write a sketch that accomplishes all of these things. But before we dive into writing real code, I want to give you a sense of the overall procedure that we're going to follow for finding the closest pixel. This is something that you'll need over-and-over in the future when you make your own Kinect apps and so rather than simply memorizing or copy-and-pasting code, it's best to understand the ideas behind it so that you can reinvent it yourself when you need it. To that end, I'm going to start by explaining the individual steps involved in plain English so that you can get an idea of what we're trying to do before we dive into the forest of variables, for-loops, and type declarations. Hopefully this *pseudo code*, as its known, will act as a map so you don't get lost in these nitty gritty details when they do arise.

So now: the plan. First a high-level overview.

Finding the Closest Pixel

Look at every pixel that comes from the Kinect one at a time. When looking at a single pixel, if that pixel is the closest one we've seen so far, save its depth value to compare against later pixels and save its position. Once we've finished looking through all the pixels, what we'll be left with is the depth value of the closest pixel and its position. Then we can use those to display a simple circle on the screen that will follow the user's hand (or whatever else they wave at their Kinect).

Sounds pretty straightforward. Let's break it down into a slightly more concrete form to make sure we understand it:

```
get the depth array from the kinect

for each row in the depth image
  look at each pixel in the row
  for each pixel, pull out the corresponding value from the depth array
    if that pixel is the closest one we've seen so far
      save its value
      and save its position (both X and Y coordinates)

then, once we've looked at every pixel in the image, whatever value we saved last will be the closest de

draw the depth image on the screen

draw a red circle over it, positioned at the X and Y coordinates we saved of the closest pixel.
```

Now this version is starting to look a little bit more like code. I've even indented it like it's code. In fact, we could start to write our code by replacing each line in this pseudo-code with a real line of code and we'd be pretty much on the right track.

But before we do that, let's make sure that we understand some of the subtleties of this plan and how it will actually find us the closest point. The main idea here is that we're going to loop over every point in the depth array comparing depth values as we go. If the depth value of any point is closer than the closest one we've seen before, then we save that new value and compare all future points against it instead.

It's a bit like keeping track of the leader during an Olympic competition. The event starts without a leader. By definition whoever goes first becomes the leader. Their distance or speed becomes the number to beat. During the event if any athlete runs a faster time or jumps a farther distance then they become the new leader and their score becomes the one to beat. At the end of the event, after all the athletes have had their turn, whoever has the best score is the winner and whatever their score is becomes the winning time or distance.

Our code is going to work exactly like the judges in that Olympic competition. But instead of looking for the fastest time or furthest distance, we're looking for the closest point. As we go through the loop, we'll check each point's distance and if it's closer than the closest one we've seen so far, that point will become our leader and its distance will be the one to beat. And then, when we get to the end of the loop, after we've seen all the points, whichever one is left as the leader will be the winner, we'll have found our closest point.

Ok. At this point you should understand the plan for our code and be ready to look at the actual sketch. I'm presenting it here with the pseudo-code included as comments directly above the lines that implement the corresponding idea. Read through it, run it, see what it does and then I'll explain a few of the nitty gritty details that we haven't covered yet.

```
import SimpleOpenNI.*;
SimpleOpenNI kinect;

int closestValue;
int closestX;
int closestY;

void setup()
{
  size(640, 480);
  kinect = new SimpleOpenNI(this);
  kinect.enableDepth();
}

void draw()
{
  closestValue = 8000;

  kinect.update();

  // get the depth array from the kinect
  int[] depthValues = kinect.depthMap();
```

```
    // for each row in the depth image
    for(int y = 0; y < 480; y++){
      // look at each pixel in the row
      for(int x = 0; x < 640; x++){
        // pull out the corresponding value from the depth array
        int i = x + y * 640;
        int currentDepthValue = depthValues[i];
        // if that pixel is the closest one we've seen so far
        if(currentDepthValue > 0 && currentDepthValue < closestValue){
          // save its value
          closestValue = currentDepthValue;
          // and save its position (both X and Y coordinates)
          closestX = x;
          closestY = y;
        }
      }
    }

    //draw the depth image on the screen
    image(kinect.depthImage(),0,0);

    // draw a red circle over it,
    // positioned at the X and Y coordinates
    // we saved of the closest pixel.
    fill(255,0,0);
    ellipse(closestX, closestY, 25, 25);
  }
```

Hey, a sketch that's actually interactive! When you run this sketch you should see a red dot floating over the depth image following whatever is closest to the Kinect. If you face the Kinect so that there's nothing between you and the camera and extend your hand, then the red dot should follow your hand around when you move it.

For example, Figure 2-14 shows the red dot following my extended fist when I run the sketch.

The tracking is good enough that if you point a single finger at the Kinect and wag it back and forth disapprovingly, the dot should even stick to the tip of your outstretched finger.

Now you should understand most of what's going in this code based on our discussion of the pseudo-code, but there's a few details that are worth pointing out and clarifying.

First, let's look at "closestValue". This is going to be the variable that holds the current record holder for closest pixel as we work our way though the image. Ironically, the winner of this competition will be the pixel with the lowest value, not the highest. As we saw in Example 5 the depth values range from about 450 to just under 8000 and lower depth values correspond to closer points.

At the beginning of `draw()`, we set closestValue to 8000. That number is so high that it's actually outside of the range of possible of values that we'll see from the depth map. Hence, all of our actual points within the depth image will have lower values. This

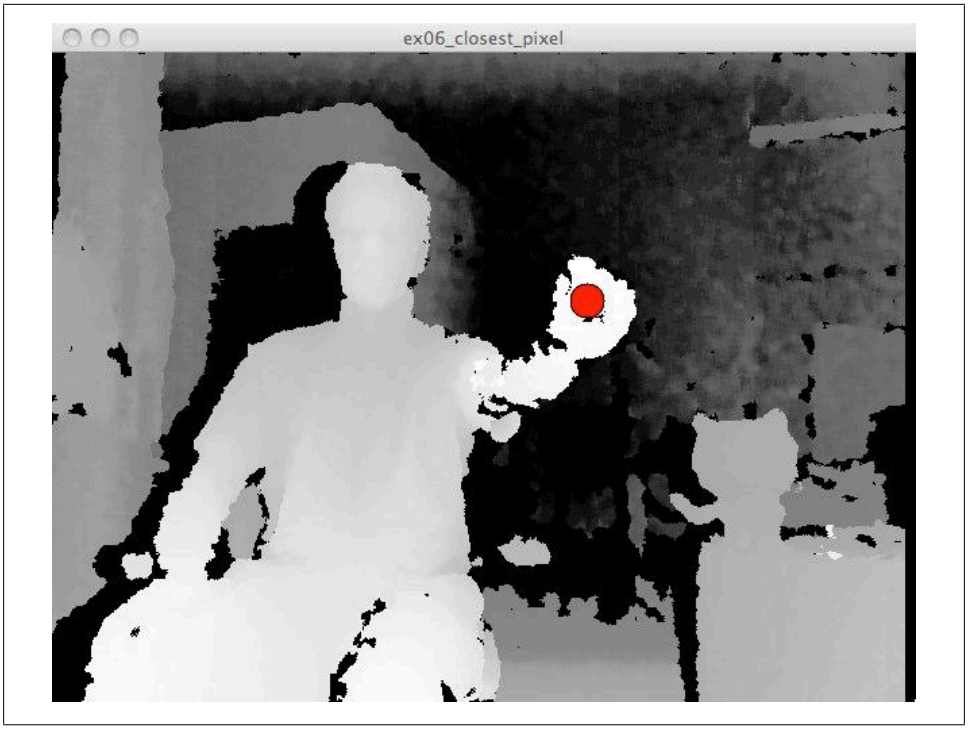


Figure 2-14. Our red circle following my outstretched fist.

guarantees that all our pixels will be considered in the competition for closest point and that one of them will actually win.

Another interesting twist comes up with this line near the middle of `draw()`:

```
if(currentDepthValue > 0 && currentDepthValue < closestValue){
```

Here we're comparing the depth reading of the current point with closestValue, the current record holder, to see if the current point should be crowned as the new closest point. However, we don't just compare currentDepthValue with closest value, we also check to see if it is greater than zero. Why?

In general we know that the lower a point's depth value, the closer that point is to the Kinect. Back in "Higher Resolution Depth Data" on page 74, though, when we were first exploring these depth map readings, we discovered an exception to this rule. The closest points in the image have depth readings of around 450, but there are some other points that have readings of zero. These are the points that the Kinect can't see and hence doesn't have data for. They might be so close that their within the Kinect's minimum range or obscured by the shadow of some closer object. Either way, we know that none of these points are the closest one and so we need to discard them. That's why we added the check for `currentDepthValue > 0` to our if statement.

Next, let's look at our two for-loops. We know from our pseudo-code that we want to go through every row in the image and within every row we want to look at every point in that row. How did we translate that into code?

```
// for each row in the depth image
for(int y = 0; y < 480; y++){
  // look at each pixel in the row
  for(int x = 0; x < 640; x++){
```

What we've got here is two for-loops, one inside the other. The outer one increments a variable `y` from zero up to 479. We know that the depth image from the Kinect is 480 pixels tall. In other words, it consists of 480 rows of pixels. This outer loop will run once for each one of those rows, setting `y` to the number of the current row (starting at zero).

The next line kicks off a for-loop that does almost the same thing, but with a different variable, `x`, and a different constraint, 640. This inner loop is going to run once per row. We want it to cover every pixel in the row. Since the depth image from the Kinect is 640 pixels wide, we know that it'll have to run 640 times in order to do so.

The code inside of this inner loop, then, is going to run once per pixel in the image. It will proceed across each row in turn, left to right, before jumping down to the next row until it reaches the bottom right corner of the image and stops.

But as we well know from our previous experience with `kinect.depthMap()`, our `depthValues` array doesn't store rows of pixels; it's just a single flat stack. Hence we need to invoke the same logic we just learned for converting between the x-y coordinate of the image and the position of a value in the array. And that's exactly what we do inside the inner `for` look:

```
// pull out the corresponding value from the depth array
int i = x + y * 640;
```

That line should look familiar to you from the example in "Higher Resolution Depth Data" on page 74. It converts the x-y coordinates of a pixel in the image to the index of the corresponding value in the array. And once we've got that index, we can use it to access the depthValues array and pull out the value for the current point, which is exactly what we do on the next line. This again, should look familiar from our previous work.

Now, at this point you've made the big transition. You've switched from working with a single depth pixel at a time to processing the entire depth image. You understand how to write the nested loops that let your code run over every point in the depth image.

Once you've got that down, the only other challenge in this sketch is understanding how we use that ability to answer questions about the entire depth image as a whole. In this case, the question we're answering is: which point in the depth image is closest to the Kinect? In order to answer that question we need to translate from code that runs on a series of individual points to information that holds up for all points in the image. In this sketch, our answer to that question is contained in our three main variables:

closestValue, closestX, and closestY. They're where we store the information that we build up from processing each individual depth point.

Using Variable Scope

In order to understand how this works, how these variables can aggregate data from individual pixels into a more widely useful form, we need to talk about "scope". When it comes to code, "scope" describes how long a variable sticks around. Does it exist only inside of a particular for-loop? Does it exist only in a single function? Or does it persist for the entire sketch? In this example, we have variables that have all three of these different scopes and these variables work together to aggregate data from each pixel to create useful information about the entire depth image. The data tunnels its way out from the innermost scope where it relates only to single pixels to the outer scope where it contains information about the entire depth image: the location and distance of its closest point.

Our Processing sketch is like an onion. It has many layers and each scope covers a different set of these layers. Once a variable is assigned it stays set for all of the layers inside of the one on which it was originally defined. So, variables defined outside of any function are available everywhere in the sketch. For example `kinect` is defined at the top of this sketch and we use it in both our `setup()` and `draw()` functions. We don't have to reset our `kinect` variable at the start of `draw()` each time, we can just use it.

Variables defined on inner layers, on the other hand, disappear whenever we leave that layer. Our variable `i`, for example, which gets declared just inside the inner `for` loop—at the innermost core of the onion—represents the array index for each individual point in the depthMap. It disappears and gets reset for each pixel *every time* our inner loop runs. We wouldn't want its value to persist because each pixel's array index is independent of all the ones that came before. We want to start that calculation from scratch each time, not build it up over time.

Another piece of information that we want to change with every pixel is `currentDepthValue`. That's the high resolution depth reading that corresponds to each pixel. Every time the inner loop runs, we want to pull a new depth reading out of the `depthValues` array for each new pixel, we don't care about the old ones. That's why both `i` and `currentDepthValue` are declared in this most inner scope. We want them to constantly change with each pixel.

There's also some data that we want to change every time `draw()` runs, but stay the same through both of our for-loops. This data lives on an intermediate layer of the onion. The key variable here is `depthValues` which stores the array of depth readings from the Kinect. We want to pull in a new frame from the Kinect every time `draw()` runs. Hence this variable should get reset every time the `draw()` function restarts. But we also want the `depthValues` array to stick around long enough so that we can process all of its points. It needs to be available inside of our inner for-loop so we can access each point to read out its value and do our calculations. That's why `depthValues` is in

this intermediate scope, available throughout each run of draw, but not across the entire sketch.

And finally, moving out to the outermost layer of the onion, we find our three key variables: `closestValue`, `closestX`, and `closestY`. Just like `kinect`, we declared these at the very top of our sketch, outside of either the `setup()` or draw function and so they are available everywhere. And, more than that, they persist over time. No matter how many times the inner pixel-processing loop runs, no matter how many times the `draw()` function itself runs, these variables will retain their values until we intentionally change them. That's why we can use them to build up an answer to find the closest pixel. Even though the inner loop only knows the distance of the current pixel it can constantly compare this distance with the closestValue, changing the `closestValue` if necessary. And we've seen in our discussion of the pseudo-code (and of Olympic records) how that ability leads to eventually finding the closest point in the whole depth image. This all works because of the difference in scope. If closestValue didn't stick around as we processed all of the pixels, it wouldn't end up with the right value when we were done processing them.

Do we actually need `closestValue`, `closestX`, and `closestY` to be global variables, available everywhere, just like `kinect`? Unlike our `kinect` object, we don't access any of these three in our setup function, we only use them within draw(). Having them be global also allows their values to persist across multiple runs of `draw()`. Are we taking advantage of this?

Well, certainly not for closestValue. The very first line of `draw()` sets closestValue to 8000, discarding whatever value it ended up with after the last run through all of the depth image's pixels. If we didn't reset closestValue, we would end up comparing every pixel in each new frame from the Kinect to the closest pixel that we'd ever seen since our sketch started running. Instead of constantly tracking the closest point in our scene, causing the red circle to track your extended hand, the sketch would lock onto the closest point and only move if some closer point emerged in the future. If you walked up to the Kinect and you were the closest thing in the the scene, the red circle might track you, but then, when you walked away, it would get stuck at your closest point to the Kinect.

By resetting `closestValue` for each run of `draw()` we ensure that we find the closest point in each frame from the Kinect no matter what happened in the past. This tells us that we could move the scope of `closestValue` to be contained within `draw()` without changing how our sketch works.

But what about `closestX` and `closestY`? We don't set these to a default value at the top of `draw()`. They enter our nested for-loops still containing their values from the previous frame. But what happens then? Since closestValue is set above the range of possible depth values, any point in the depth map that has an actual depth value will cause closestX and closestY to change, getting set to that point's x and y coordinates. Some point in the depth image has to be the closest one.

The only way that `closestX` and `closestY` could make it all the way through our for-loops without changing their values would be if every single point in the depth map had a value of zero. This can only happen if you've covered the Kinect with a cloth or pointed it face-first up against a wall or done something else to make sure that it's entire field of view is covered by something within the Kinect's minimum range. This is a rare, and clearly not very useful, situation. So, in any practical scenario, `closestX` and `closestY` will always get changed somewhere within each run of `draw()`. That means that `closestX` and `closestY` don't need to be global variables either. In practice, their values will never actually persist between multiple runs of `draw()`. We could declare them within `draw()` without changing how our sketch works.

So, then, why did we declare all three of these variables at the top of the sketch? There are two reasons. First of all, it's nice to have them at the top of the sketch because they're important. Having them at the top of the sketch means that you see them on first glance when looking at the code. You can look at just those first six lines of this Processing sketch and have a pretty good idea of what it's going to be doing. It imports the Kinect library and it defines some variables that are clearly meant to store the value and location of the closest point. Obviously this code is going to be using the Kinect to track the closest point in front of it. Having these variables at the top of the sketch doesn't just declare their scope, it also declares the sketch's intention.

Also, if we wanted to make this sketch more sophisticated we'd immediately start taking advantage of the global scope of these variables. For example, imagine we wanted to add some smoothing to the movement of our red circle: instead of just jumping from point-to-point as the closest object in the scene shifted, we wanted it to make a more gradual transition. To accomplish this we'd want `closestX` and `closestY` to not simply get reset with every frame. We'd want them to be more like a running average so if the closest point suddenly jumped they would fill in the intermediate values keeping things smooth and continuous. To make our sketch work like this, we'd need to use the global scope of `closestX` and `closestY` explicitly, updating them between each run of `draw()` rather than resetting them.

There are similar things we might want to do with `closestValue`. Imagine if you wanted the circle to track the user's hand only if it moved by more than a certain amount. This might allow the user to draw straight lines in a drawing app or allow you to ignore a user who wasn't moving at all. To do something like that you'd need to start taking advantage of the global scope of `closestValue` so that you could compare data across multiple runs of `draw()`. You'd still calculate the closest value of the depth map in each new run of `draw()` but then you'd compare what you found to the persistent global `closestValue` and only update `closestValue` if that difference was more than some minimum threshold.

For an example of a sketch that uses global scope in this way, take a look at `closest_pixel_running_average.pde` in Chapter 9. That sketch uses closestX and closestY's global scope to implement the smoothing example I discussed above.

Projects

To finish up this chapter, we're going to make a couple of small projects that use what we've learned about processing the depth data to create actual fun applications. Each of them builds on the techniques you've already learned: looping through the depth map to find the closest pixel, persisting data across multiple runs of the draw function, and using that data to provide an actual user interface.

For each of the applications, we'll start with a basic version that just takes a small next step beyond the code we've already seen. Once we have that working, we'll add a few refinements and advanced features and then conclude with a discussion of future possibilities that you can explore yourself.

First, we're going to build an "invisible pencil" (Figure 2-15): a sketch that lets you draw by waving your hand around in front of the Kinect. This sketch will use the tracking we've already done to generate a line that follows your hand. In the advanced version, we'll learn how to smooth out the movement of the line to give you more control of the drawing and how to save the final drawing as an image.

Second, we'll create a photo layout program that lets you drag a series of images around to arrange them on the screen by moving your hands in space a la Tom Cruise in Minority Report. We'll build on the tracking and smoothing of the invisible pencil example, but this time we'll learn how to display and manipulate images. In the advanced version, we'll add the ability to cycle through control of multiple images in order to lay them out like snapshots on a table top.

Project 6: Invisible Pencil

This application is going to take the tracking interface we created in the last section and turn it into a drawing program. We'll still track the closest point that the Kinect can see, but now instead of simply displaying it as a circle on top of the depth image, we'll use that point to draw a line.

In order to draw a line in Processing you need two points. Processing's line() function takes four values: the x and y coordinates of the first point followed the x and y coordinates of the second point. It draws a line on the screen that connects these two points that you give it. To transform our closest point tracking code into drawing code, we'll need to draw a line on each frame instead of a circle. And, we'll want our line to connect the current closest point to the previous one. That way we'll get a series of short lines that connect the positions of our closest point over time. Since the end of each of these individual lines will also be the start of the next one, together they'll join up to create a single line that flows around our sketch following the closest point over time.

To get the two points we need to draw our line, we'll need more than just the closest point we've tracked in any individual frame of the depth image. We'll also need the

Figure 2-15. The final output of the Invisible Pencil project: smooth handwriting created by waving your hand around in space.

previous closest point, the one we just tracked in the last frame. Our line will connect this older point to the new one.

At the end of the last section we discussed strategies for building up information about the depth image across multiple frames. We explored using global variables to save the closest point in a particular depth image so that it would be available when we were processing the depth image. We then showed how you could use that older point to smooth out the movement of the red circle around the screen by averaging the current closest point with the previous closest one (check out Chapter 9 if you don't remember that discussion).

This time we're going to use that same logic to keep track of the current closest point and the previous one, but instead of using them for smoothing, we'll use them as the start and end of each our line segments.

Let's take a look at the code. This code picks up exactly where the last example we discussed ("Finding the Closest Pixel" on page 78) left off. I've included comments about all the new lines that differ from that older example. Read through this code and see if you can understand how it implements the line drawing strategy I just described.

```
import SimpleOpenNI.*;
SimpleOpenNI kinect;
```

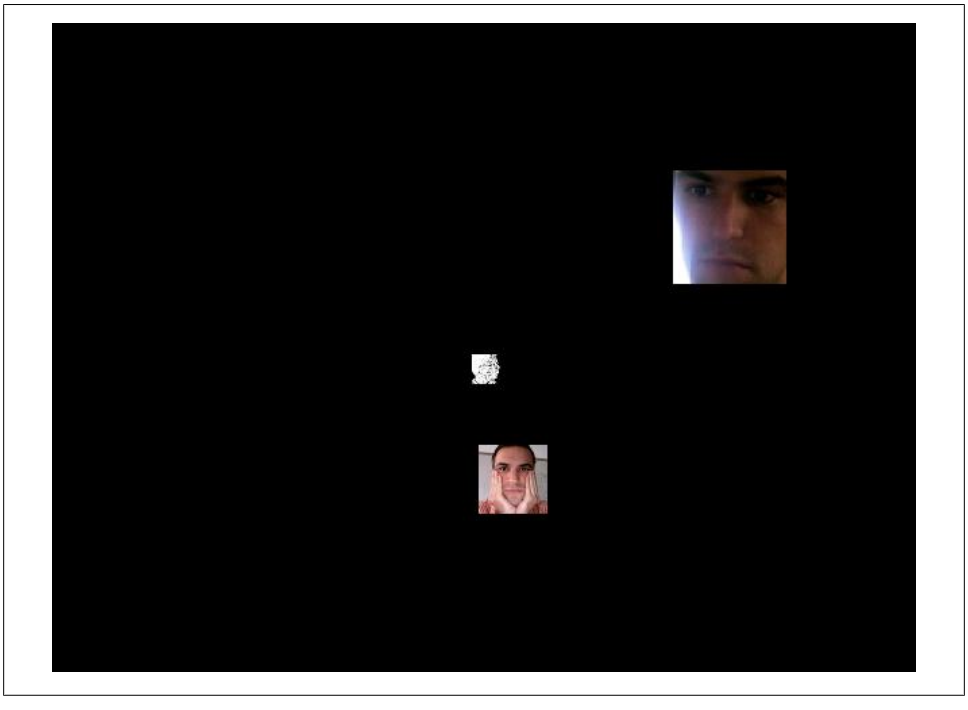


Figure 2-16. The final output of the Minority Report project: controlling the position and size of three images, one at a time with our closest point.

```
int closestValue;
int closestX;
int closestY;

// declare global variables for the
// previous x and y coordinates
int previousX;
int previousY;

void setup()
{
  size(640, 480);
  kinect = new SimpleOpenNI(this);
  kinect.enableDepth();
}

void draw()
{
  closestValue = 8000;

  kinect.update();

  int[] depthValues = kinect.depthMap();
```

```
    for(int y = 0; y < 480; y++){
      for(int x = 0; x < 640; x++){
        int i = x + y * 640;
        int currentDepthValue = depthValues[i];

        if(currentDepthValue > 0 && currentDepthValue < closestValue){

          closestValue = currentDepthValue;
          closestX = x;
          closestY = y;
        }
      }
    }

  image(kinect.depthImage(),0,0);

  // set the line drawing color to red
  stroke(255,0,0);

  // draw a line from the previous point
  // to the new closest one
  line(previousX, previousY, closestX, closestY);

  // save the closest point
  // as the new previous one
  previousX = closestX;
  previousY = closestY;


}
```

So, how did we do it? Well, first we added two new global variables, `previousX` and `previousY`. These are going to hold the coordinates of the closest point before the current frame of the depth image, the one our line will connect with the new closest point.

After that, our setup function didn't change and neither did our code that loops through all the points to find the closest one. The only other addition is four new lines at the bottom of the sketch. The first of these, `stroke(255,0,0)` tells Processing to set the color for line drawing to red. This is like dipping our pen in red ink, it means that any lines we draw after this will take that color. In our previous sketch we'd used `fill()` in a similar manner to set the color of our circle; `stroke()` is like `fill()` for lines. The next line actually draws the line between the previous closest point and the new closest point. This works just how we expected from the description above of Processing's `line()` function.

Now, the final two lines here are a bit subtle, but are important for understanding how this sketch works. Right before the end of the `draw()` function, we set `previousX` equal to `closestX` and `previousY` equal to `closestY`. This is our way of holding onto the current x- and y-coordinates of the closest point so that they'll be available on the next run of the draw function to compare with the next closest point that we find. Remember that `previousX` and `previousY` are global variables. That means that they stick around across

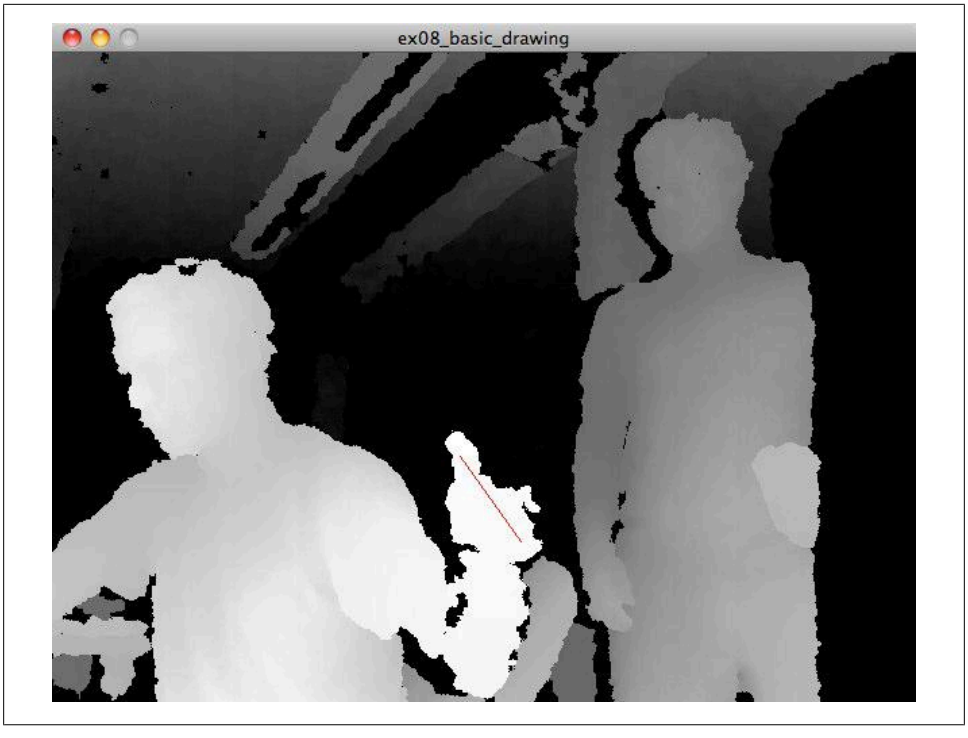


Figure 2-17. A tiny line of red flickers over the depth image. The depth image is covering over our line so we can't see it build up.

runs of the `draw()` function. That's what makes it possible to connect each previous point with each new one. If we didn't set `previousX +and +previousY` like this every time, instead of drawing a series of line segments that were connected end-to-end and followed the track of the closest point, `previousX` and `previousY` would never change. You'd end up with a bunch of lines that all radiated out from a shared starting point. Not a very good drawing interface.

Now let's run this code. We should see a line on top of the depth image tracking the closest point, right?

But wait, what's going on? Instead of a continuous line being drawn over the image, when I run this code, I get a flickering tiny scrap of red that disappears and reappears in each frame. You can see the effect in Figure 2-17.

Why don't we see the whole line? Why are we only seeing the last line segment at any given frame? Let's look at our sketch again. In addition to drawing the line segment connecting the current closest point to the previous one, there's something else we're drawing as well: the depth image itself. On this line (just after the `for` loops), we display the depth image from the Kinect:

```
image(kinect.depthImage(),0,0);
```

In Processing, every new thing that you display goes down right on top of what was there previously. Therefore an image that fills the whole sketch will cover over all the drawing that had been done up to that point, essentially clearing the sketch. Think of Processing like a piece of paper on which you're making an ongoing collage. If you draw some lines and then paste down a photograph, the photograph will cover over the lines, making them invisible. You'll only see the lines you scribble over the top of that photograph. That's exactly what's happening here with each frame of the depth image. We're laying down one frame of the depth image then drawing the appropriate line segment for that frame, but then the next frame immediately comes in and covers it over.

To fix this, we need to stop displaying our depth image. Let's comment out that line (put `//` in front of `image(kinect.depthImage(),0,0);`) and run the sketch again. Now you should see a red line moving around the sketch, following the closest point in front of the Kinect. Try waving your hand around a bit to see if you can draw something. Note how the line moves around and rapidly builds up filling the sketch.

I don't know about you, but my sketch is getting messy quickly, filling up with skittering red lines. Let's add a function to clear the screen when we click the mouse. We'll use Processing's mousePressed function to do this. The code is very simple. Add this to the bottom of your sketch:

```
void mousePressed(){
 background(0);
}
```

Stop your sketch and run it again. Now, when you click anywhere within your sketch's window, your drawing should disappear and be replaced with a clean black screen.

So at this point, we've got a working drawing app. You wave your hand around and you can control a line showing up on your screen. However, it has some limitations. First of all, our line is quite skittery. It's very sensitive to even very small movements of the closest point in the image. And it has a tendency to make big jumps even when you're not consciously moving around the point you think it's tracking.

When I run it, my output looks something like Figure 2-18.

All I did to create this drawing was extend my hand towards the Kinect so that it would be tracked as the closest point and then raise and lower it in an arc to my right. The jittery lines in the center of the sketch are where it first started tracking my hand while it was remaining still. The rest of the line is where it followed my hand along the arc.

One of the hallmarks of a good user interface is responsiveness. The user should feel a tight connection between their actions and the results in the application. There shouldn't be any delay between the user's input and the resulting action and everything that happens within the app should be the result of an intentional action on the part of the user. So far, our app is fairly responsive in the sense that the there's nearly no delay between our movement and the movement of the line. However the line also seems to move quite a bit on its own. In addition to the user's intentional action, there's

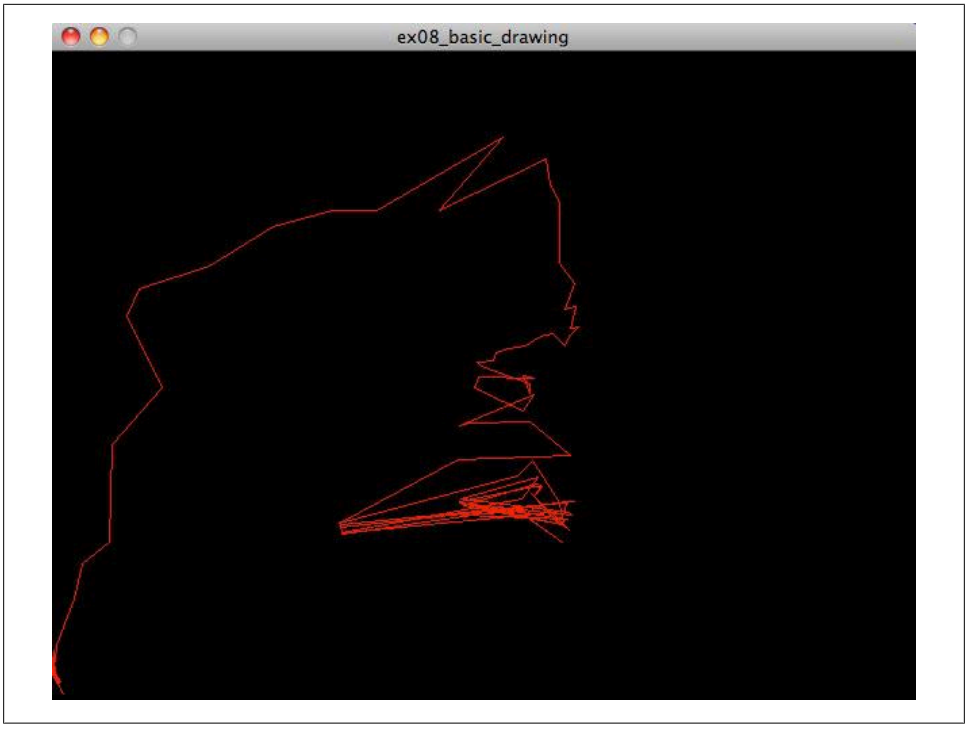


Figure 2-18. Our hand-tracking drawing app's first output.

some randomness in the controls as the closest point hops to unexpected spots or even simply makes tiny shifts within individual points within the user's out-stretched hand. As a user of this sketch I find this lack of control quite frustrating.

To eliminate these large random jumps we need to tighten down the focus of our tracking. Instead of considering any part of the scene, we want to focus on the area that is most likely to have the user in it: the distance from about two feet to about five feet.

To eliminate the tiny jitters we need to smooth out the movement of the line. We can do this by not simply jumping directly to the closest point on every frame, but instead "interpolating" between each previous point and each new one that comes in. Interpolation is the process of filling in the missing space between two known points. In other words, instead of jumping to the next point, we'll orient our line towards that next point but only take the first step in that direction. If we do this for every new closest point, we'll always be following the user's input, but we'll do so along a much smoother path.

The second big problem with this basic version is that it's backwards. That is, when you move your hand to your left, the line moves to the right on the screen and vice versa. This happens because the Kinect's depth camera is facing you. It sees you the way other people do: your left hand is on its right. What we'd really like is for the sketch

to act like a piece of paper: if you move your hand to the left the line should also move to the left. That would make for a much more intuitive interface.

To achieve this, we need to mirror the image coming in from the Kinect. In a mirror, the image of your left hand is on your left and that of your right hand is on the right. It's just the reverse of what a camera looking at you normally sees. So to convert the Kinect's image into a mirror image we need to flip the order of the Kinect depth points on the x-axis.

Let's take a look at an advanced version of the sketch that corrects both of these problems and adds one more nice feature: saving an image of your drawing when you clear it.

```
import SimpleOpenNI.*;
SimpleOpenNI kinect;

int closestValue;
int closestX;
int closestY;

float lastX;
float lastY;

void setup()
{
  size(640, 480);
  kinect = new SimpleOpenNI(this);
  kinect.enableDepth();

  // start out with a black background
  background(0);
}

void draw()
{
  closestValue = 8000;

  kinect.update();

  int[] depthValues = kinect.depthMap();

    for(int y = 0; y < 480; y++){
      for(int x = 0; x < 640; x++){

        // reverse x by moving in from
        // the right side of the image
        int reversedX = 640-x-1;

        // use reversedX to calculate
        // the array index
        int i = reversedX + y * 640;
        int currentDepthValue = depthValues[i];

        // only look for the closestValue within a range
```

```
          // 610 (or 2 feet) is the minimum
          // 1525 (or 5 feet) is the maximum
          if(currentDepthValue > 610 && currentDepthValue < 1525
             && currentDepthValue < closestValue){

            closestValue = currentDepthValue;
            closestX = x;
            closestY = y;
          }
        }
      }

    // "linear interpolation", i.e.
    // smooth transition between last point
    // and new closest point
    float interpolatedX = lerp(lastX, closestX, 0.3f);
    float interpolatedY = lerp(lastY, closestY, 0.3f);

    stroke(255,0,0);

    // make a thicker line, which looks nicer
    strokeWeight(3);

    line(lastX, lastY, interpolatedX, interpolatedY);
    lastX = interpolatedX;
    lastY = interpolatedY;

}

void mousePressed(){
    // save image to a file
    // then clear it on the screen
    save("drawing.png");
    background(0);
}
```

This code makes a number of improvements over the basic version. Some of them are simple and cosmetic and a couple of them are more sophisticated. Again, I've added comments in the code to every line that I've changed from the previous version. Run the app and see how it behaves and then I'll explain how these changes address the problems we identified above.

First off, I made a couple of simple cosmetic changes that improve the appearance of the sketch. In `setup()`, I start the sketch off with a black background. In the basic version, we were already clearing the app to black so I added this call to `background(0)` to to keep the background for our drawing consistent.

Secondly, I increased the thickness of the line that we draw. In the 4th line from the bottom of `draw()`, I called Processing's `strokeWeight()` function which sets the thickness. This defaults to one, so giving it a value of three triples the line's thickness. This makes the line more visible from further away from the screen (which you tend to be when waving your hand around in front of the Kinect) and also makes the image look better when you save it. Which brings me to the third cosmetic improvement. I modified

our `mousePressed()` event so that it saves a copy of the drawing to a file before clearing the screen. The code there simply calls `save()` to save the image into the sketch folder (choose Sketch→Show Sketch Folder to see it). It's worth noting that if you save more than once, this sketch will overwrite the earlier images with the newer ones and you'll only end up with the most recent. To prevent this when you save an image move or rename it before saving a new one.

So those are the cosmetic improvements, but what about the substantial ones? What have I done to address the erratic movement of our line and to mirror it to improve our app's usability?

Let's discuss the mirroring technique first as it's actually quite simple. To create a mirror image of the depth data, all we need to do is reverse the x-coordinate from the one we've currently been calculating from our point array. Think back to our technique for converting from x-y coordinates to the correct position within our linear array of depth points. To make that conversion, we multiply the number of row we're on (the y-coordinate) by the width of the row and then add the x-coordinate. The first number gets us the number of points already accounted for on previous rows and the second one gets us the distance between this point and the row's left edge.

This time, though, we want to flip the image over from left to right. Instead of calculating the point's distance from the left edge, we want to translate it into a distance from the right edge, so we can grab the equivalent pixel on the opposite side of the image. If we do this for every pixel across every row, we'll have a mirror image.

You can see from our loop that our x-coordinate counts up from zero to 639 (`x < 640` in our loop declaration means that `x` stops just before reaching 640). When we used it directly, this progression of - values meant that our position in each row marched directly from left to right. In order to reverse this march, we need to calculate a new number that starts at 639 and then counts down to zero. And given our current `x`, this is very easy to do: our new reversedX will simply be `640-x-1` (as you can see on line 34). When `x` is at zero (i.e. the very left of the row), reversedX will be at 639 (the very right). As `x` increases `reversedX` will decrease until it reaches zero and jumps to the next row. Just what we want.

Once we've calculated that `reversedX` all we have to do is swap it into the original formula for calculating the array index in place of x like so:

```
// reverse x by moving in from
// the right side of the image
int reversedX = 640-x-1;

// use reversedX to calculate
// the array index
int i = reversedX + y * 640;
```

Boom. In one line of code, we've reversed our image. If you ran the sketch with just this change, you'd see that it now acted as a mirror with the line moving to your left as you moved your hand that way and vice versa. Much more intuitive.

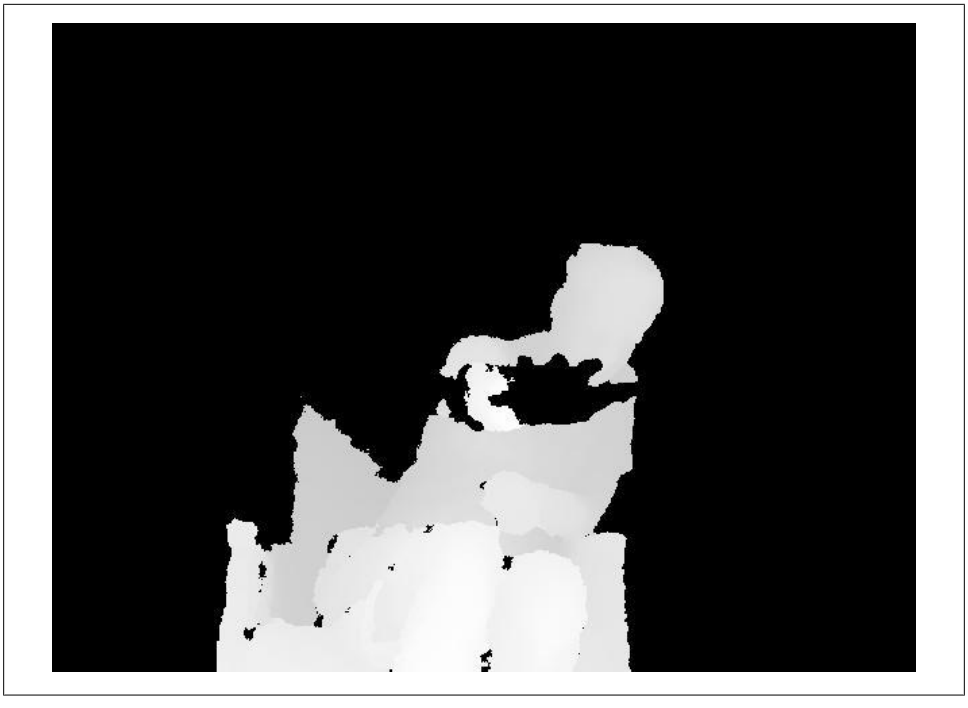


Figure 2-19. The depth image with its range limited to only show the part of the scene that's likely to include the user.

Now let's get to smoothing. As I mentioned above, there are actually two improvements in this version of the app that eliminate jitter. The first one is to only care about points within a certain depth. We know we only want the user to control the drawing, not bits of the wall or ceiling behind them, not cats that happen to walk by between them and the camera. Hence, if we make an assumption about how far away the user is likely to be from the Kinect, we can eliminate most of the points in the depth image from our closest point calculation. This will eliminate the occasional big jumps we detected in the basic version of the sketch.

But how to chose this range? We want it to be big enough that it's easy to access, but small enough that it limits the scene to the portion the user is most likely to occupy. After some experimentation, I found that a range of two to five feet was appropriate for this. To perform that experimentation, I wrote an sketch that only displays points from the depth image that are within a given range (you can see the code in the Appendix if you're interested). After trying some different values, Figure 2-19 shows what things looked like with the two to five foot range.

You can see that my hand is close enough that it's going black and the back of the chair is partially black as well. That should give you a sense of the size of the range: about the length of my outstretched arm.

Figure 2-20. The depth image range is calibrated to just capture my hand.

Having found this range, I moved my furniture around a bit so I could get just my arm into the range. When just my arm and hand started showing up, I knew I had things just right. Figure 2-20 shows the result.

Once I'd found the correct range, I went ahead and added it to our advanced drawing sketch. This range is enforced on this line (the `if` statement from the innermost `for` loop of `draw()`):

```
if(currentDepthValue > 610 && currentDepthValue < 1525
   && currentDepthValue < closestValue){
```

The first two criteria of that `if` statement are new. They say we should only consider a point as a potential closestValue if its depth reading is greater than 610 and less than 1525. I determined these numbers using what we learned in "Higher Resolution Depth Data" on page 74: these raw numbers are millimeters and there are 25.4 millimeters per inch. Hence these criteria will limit our search to the area between two (2 * 12 * 25.4 = 609.6) and five (5 * 12 * 25.4 = 1524.0) feet away from the Kinect. If the user positions themselves correctly, this will limit our search to just the points on their hand and arm eliminating some of the jitter we'd previously seen.

Ok so that's one source of jitter. But what about the other one? What about the constant back-and-forth scribbling? I said above that we were going to use interpolation to create a smooth line rather than just jumping straight between points like we had up to now.

Luckily for us, Processing has an interpolation function built right into it. Processing calls this function `lerp` for "linear interpolation". This function takes three arguments, the first two are the two numbers to interpolate between and the last is a float that tells you how much to move between the two points (with 1.0 being all the way to the second point and 0.0 being none of the way). You can see how we use `lerp()` to calculate interpolated values for both x and y on lines 56 and 57 of the sketch:

```
float interpolatedX = lerp(lastX, closestX, 0.3);
float interpolatedY = lerp(lastY, closestY, 0.3);
```

These interpolated values will be part of the way between our last position and the new closest point. Only going part of the way to each new point means that erratic values will have less of an impact on the line, introducing fewer jagged turns. As I explained above, interpolating is like constantly shifting your direction towards a new point while continuing to move in an unbroken line instead of just jumping to the new point like we'd previously been doing.

After we've calculated these interpolated values, we use them where we previously had used `closestX` and `closestY` for the rest of the sketch: as coordinates in `line()` and when we reset lastX and lastY after that.

With this change, our sketch is now smooth enough that you can actually use it to draw something intentionally. For example, as you can see in Figure 2-21, I wrote my name.

If you're still seeing erratic results, make sure you've got all of your furniture (and pets and roommates) moved out of the way. If you having a hard time telling if you've eliminated everything and if you're standing in the right spot you can use depth_range_limit.pde to see the thresholded image from your Kinect to get everything set up.

Once you've got everything working, try to add some more features. Can you change the color of the line based on the depth value that comes in? Can you add the ability to "lift the pen", i.e. turn drawing the line on and off with the mousePressed event so you don't have to only draw one continuous line? Can you think of something else fun you'd like the sketch to do?

Project 7: Minority Report Photos

This next project is going to build on what we just learned with the Invisible Pencil in order to implement an interface that helped defined the idea of gesture-controlled technology in popular culture. In the 2002 sci-fi movie Minority Report (based on a Philip K. Dick short story), Tom Cruise played a "pre-crime" policeman who uses the predictions of psychics to anticipate and prevent violent crimes. Cruise accesses these predictions as a series of images on a projected screen that he navigates with gestures. He drags his hands to pull in new images, spreads them apart to zoom in on a telling detail, pinches to zoom out to the big picture.

Figure 2-21. With smoothing and range limiting, I had enough control to be able to write my name by waving my hand around.

Watching the movie now, many of the gestures will seem familiar from contemporary touch screen interfaces that have come to market since the movie premiered. And now, with the Kinect, we can move beyond the touch screen and build a photo browsing application that looks almost exactly like the interface from the movie. With just Processing and the Kinect, we can make this sci-fi dream come true.

To do this, we're going to build on the code we've already written. Our advanced drawing example already smoothly tracks the user's hand. Now we're going to translate that motion into the position of photographs on the screen. Instead of drawing a line, we're going to let the user pick up and move around photographs.

Just like in the drawing app, we'll build two versions of this sketch. We'll start with a basic version that positions a single photo. Then we'll move on to an advanced version that displays multiple photos and lets us scale them up and down.

Basic Version: One Image

For the basic version of this app, we only need to make a few additions to our existing drawing app. We're still going to use all our same code for finding the closest point and interpolating its position as it moves. What we need to add is code related to images. We need to load an image from a file. We need to use the interpolated coordinates of

the closest point to position that image. And we need to give the user the ability to "drop" the image, to stop it moving by clicking the mouse.

Here's the code. It may look long, but it's actually mostly identical to our advanced drawing app. As usual, I've written comments on all the new lines.

```
inlcude::code/ex10_basic_minority_report/ex10_basic_minority_report.pde[]
```

To run this app, you'll need to add your own image file to it. Save your Processing sketch and give it a name. Then you'll be able to find the sketch's folder on your computer (Sketch→Show Sketch Folder). Move the image you want to play with into this folder and rename it "image1.jpg" (or change the second-to-last line in `setup()` to refer to your image's existing filename). Once you've added your image, run the sketch and you should see your image floating around the screen, following your outstretched hand.

So, how does this sketch work? The first few additions, declaring and loading an image (lines 23 and 36), should be familiar to you from your previous work in Processing:

```
PImage image1;

void setup()
{
  ... some code omitted
  image1 = loadImage("image1.jpg");
```

At the top of the sketch, we also declare a few other new variables: `image1X` and `image1Y` which will hold the position of our image and a boolean called `imageMoving`, which will keep track of whether or not the user has "dropped" the image.

At the very bottom of the sketch, we also rewrote our `mousePressed()` function. Now it simply toggles that `imageMoving` variable. So if `imageMoving` is true, clicking the mouse will set it to false and vice versa. That way the mouse button will act to drop the image if the user is currently moving it around and to start it moving around again if it's dropped.

The real action here is at the end of the `draw()` function, after we've calculated `interpolatedX` and `interpolatedY`:

```
// only update image position
// if image is in moving state
if(imageMoving){
    image1X = interpolatedX;
    image2Y = interpolatedY;
}

//draw the image on the screen
image(image1,image1X,image1Y);
```

If our `imageMoving` variable is true, we update our image's x-y coordinates based on `interpolatedX` and `interpolatedY`. And then we draw the image using those x-y coordinates. Actually we draw the image using those coordinates whether or not it is cur-

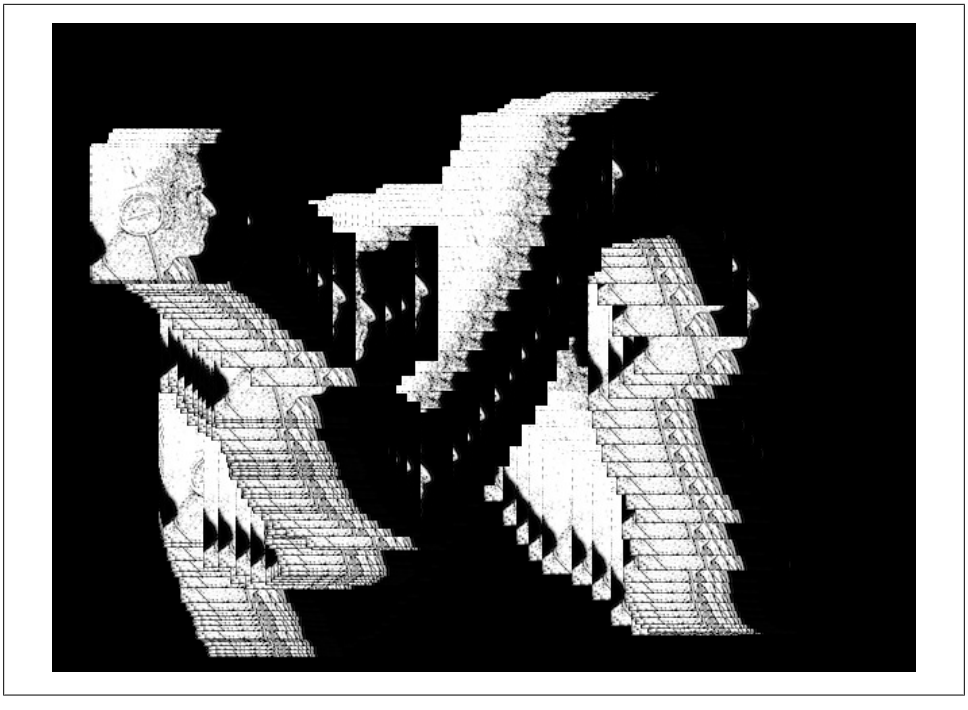


Figure 2-22. If we don't clear the background to black when moving an image around, the result will be a smeary mess.

rently being moved. If the image is being moved `image1X` and `image1Y` will always be set to the most recent values of `interpolatedX` and `interpolatedY`. The image will move around the screen tracking your hand. When you click the mouse and set `imageMov ing` to false `image1X` and `image1Y` will stop updating from the interpolated coordinates. However, we'll still go ahead and draw the image using the most recent values of `image1X` and `image1Y`. In other words we still display the image, we just stop changing its position based on our tracking of the closest point. It's like we've dropped the image onto the table. It will stay still no matter how you move around in front of the Kinect.

The one other detail worth noting here is this line from `draw()`: `background(0)`. This clears the whole sketch to black. If we didn't do that, we'd end up seeing trails of our image as we moved it around. Remember, Processing always just draws on top of whatever is already there. If we don't clear our sketch to black, we'll end up constantly displaying our image on top of old copies of itself in slightly different positions. This will make a smeary mess (or a cool psychedelic effect depending on your taste). Figure 2-22 shows what my version of the sketch looks like without that line. And Figure 2-23 shows what it looks like with the line back in.

Advanced Version: Multiple Images and Scale

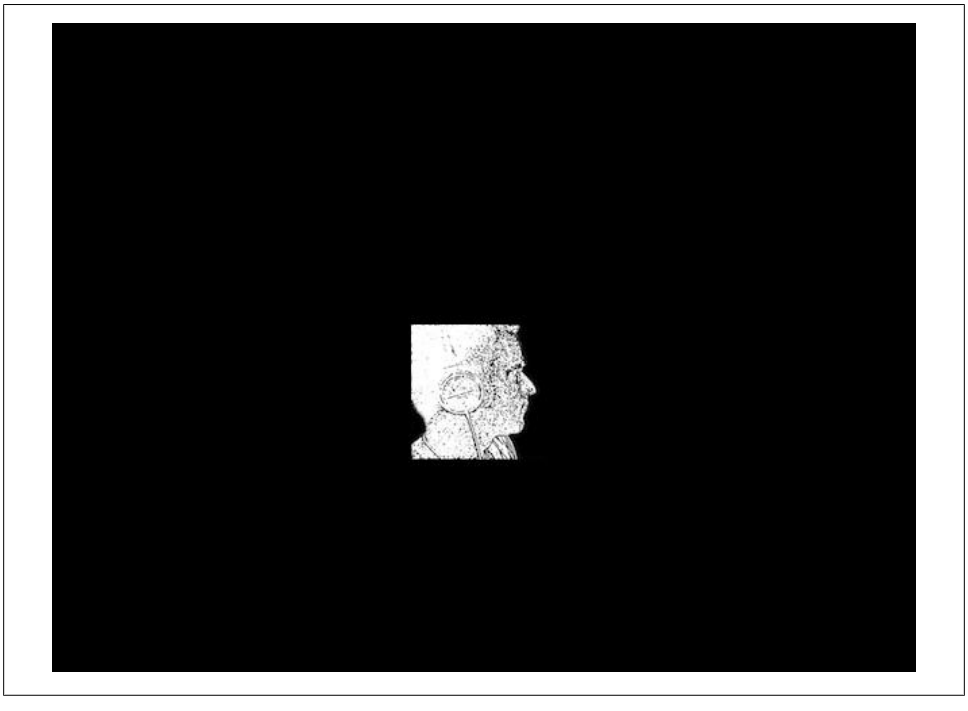


Figure 2-23. Clearing the sketch's background to black prevents redrawing the image every time and creating a smeary mess.

That's the basic version. There really wasn't a lot to it beyond the smooth hand tracking we already had working from our drawing example. Let's move on to the advanced version. This version of the sketch is going to build on what we have in two ways. First, it's going to control multiple images. That change is not going to introduce any new concepts, but will simply be a matter of managing more variables to keep track of the location of all of our images and remembering which image the user is currently controlling. The second change will be more substantial. We're going to give the user the ability to scale each image up and down by moving their hand closer to or further from the Kinect. In order to do this, we'll need to use `closestValue`, the actual distance of the closest point detected in the image. Up to this point, we've basically been ignoring `closestValue` once we've found the closest point, but in this version of the sketch it's going to become part of the interface: its value will be used to set the size of the current image.

Ok, let's see the code.

```
import SimpleOpenNI.*;
SimpleOpenNI kinect;

int closestValue;
int closestX;
```

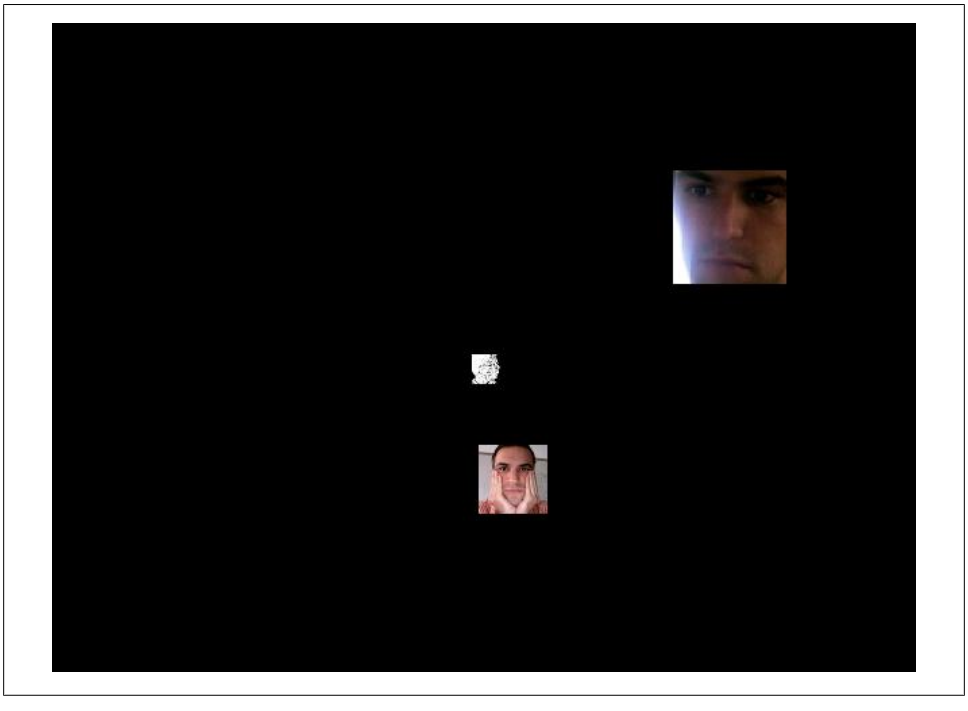


Figure 2-24. Controlling the position and size of three images, one at a time with our closest point.

```
int closestY;

float lastX;
float lastY;

float image1X;
float image1Y;
// declare variables for
// image scale and dimensions
float image1scale;
int image1width = 100;
int image1height = 100;

float image2X;
float image2Y;
float image2scale;
int image2width = 100;
int image2height = 100;

float image3X;
float image3Y;
float image3scale;
int image3width = 100;
int image3height = 100;
// keep track of which image is moving
```

```
int currentImage = 1;

// declare variables
// to store the images
PImage image1;
PImage image2;
PImage image3;


void setup()
{
  size(640, 480);
  kinect = new SimpleOpenNI(this);
  kinect.enableDepth();

  // load the images
  image1 = loadImage("image1.jpg");
  image2 = loadImage("image2.jpg");
  image3 = loadImage("image3.jpg");

}

void draw(){
  background(0);

  closestValue = 8000;

  kinect.update();

  int[] depthValues = kinect.depthMap();

    for(int y = 0; y < 480; y++){
      for(int x = 0; x < 640; x++){

        int reversedX = 640-x-1;
        int i = reversedX + y * 640;
        int currentDepthValue = depthValues[i];

        if(currentDepthValue > 610 && currentDepthValue < 1525 && currentDepthValue < closestValue){

          closestValue = currentDepthValue;
          closestX = x;
          closestY = y;
        }
      }
    }

   float interpolatedX = lerp(lastX, closestX, 0.3);
   float interpolatedY = lerp(lastY, closestY, 0.3);

   // select the current image
   switch(currentImage){
     case 1:
       // update its x-y coordinates
       // from the interpolated coordinates
```

```
        image1X = interpolatedX;
        image1Y = interpolatedY;

        // update its scale
        // from closestValue
        // 0 means invisible, 4 means quadruple size
        image1scale = map(closestValue, 610,1525, 0,4);
      break;
      case 2:
        image2X = interpolatedX;
        image2Y = interpolatedY;
        image2scale = map(closestValue, 610,1525, 0,4);

      break;
      case 3:
        image3X = interpolatedX;
        image3Y = interpolatedY;
        image3scale = map(closestValue, 610,1525, 0,4);
      break;
    }

    // draw all the image on the screen
    // use their saved scale variables to set their dimensions
    image(image1,image1X,image1Y, image1width * image1scale, image1height * image1scale);
    image(image2,image2X,image2Y, image2width * image2scale, image2height * image2scale);
    image(image3,image3X,image3Y, image3width * image3scale, image3height * image3scale);

    lastX = interpolatedX;
    lastY = interpolatedY;
}

void mousePressed(){
  // increase current image
  currentImage++;
  // but bump it back down to 0
  // if it goes above 3
  if(currentImage > 3){
    currentImage = 1;
  }
    println(currentImage);
}
```

To run this code you'll need to use three images of your own. Just like with the basic example, you'll have to save your sketch so that Processing will create a sketch folder for it. Then you can move your three images into that folder so that your sketch will be able to find them. Make sure they're named "image1.jpg", "image2.jpg", and "image3.jpg" so that our code will be able to find them.

Make sure that you tell the sketch about the dimensions of the images you're using. I'll explain the process in detail below, but in order to scale your images this sketch needs to know their starting size. Look through the top of the sketch for six variables: `image1width`, `image1height`, `image2width`, `image2height`, `image3width`, and `image3height`. Set each of those to the appropriate value based on the real size of your images before running your sketch.

Once you've setup your images, you'll be ready to run this sketch. Set your Kinect up so that you're three or four feet away from it and there's nothing between it and you. Just like the last few examples, we'll be tracking the closest point and we want that to be your outstretched hand. When you first run the sketch you should see one image moving around, following the motions of your hand just like before. However, this time try moving your hand closer and further from the Kinect. You'll notice that the image grows as you get further away and shrinks as you approach. Now, click your mouse. The image you were manipulating will freeze in place. It will hold whatever size and position it had at the moment you clicked and your second image will appear. It will also follow your hand, growing and shrinking with your distance from the Kinect. A second click of the mouse will bring out the third image for you to scale and position. A fourth will cycle back around to the first image, and so on.

We'll break our analysis of this sketch up into two parts. First, we'll look at how this sketch works with multiple images. We'll see how it remembers where to position each image and how it decides which image should be controlled by your current movements. Then, we'll move on to looking at how this sketch uses the distance of the `closestPoint` to scale the images.

The changes involved in controlling multiple images start at the top of the sketch. The first thing we need is new variables for our new images. In the old version of this sketch we declared two variables for the position of the image: `image1X` and `image1Y`. Now we have two more pairs of variables to keep track of the location of the other two images: `image2X`, `image2Y`, `image3X`, and `image3Y`. In the basic version we simply assigned `image1X` and `image1Y` to `closestX` and `closestY` whenever we wanted to update the position of the image to match the user's movement. Now, the situation is a little bit more complicated. We need to give the user the ability to move any of the three images without moving the other two. This means that we need to decide which of the pairs of image position x-y variables to update based on which image is currently being moved. We use a variable called `currentImage` to keep track of this. At the top of the sketch we initialize that variable to one so that the user controls the first image when the sketch starts up.

`currentImage` gets updated whenever the user clicks the mouse. To make this happen we use the `mousePressed` callback function at the bottom of the sketch. Let's take a look at that function to see how it cycles through the images, letting our sketch control each one in turn. Here's the code for `mousePressed`:

```
void mousePressed(){
  currentImage++;
  if(currentImage > 3){
    currentImage = 1;
  }
}
```

We only have three images and `currentImage` indicates which one we're supposed to be controlling. So the only valid values for `currentImage` are: one, two, or three. If `currentImage` ended up as zero or any number higher than three, our sketch would end up controlling none of our images. The first line of `mousePressed` increments the value of `currentImage`. Since we initialized `currentImage` to one, the first time the user clicks the mouse it will go up to two. Two is less than three so the `if` statement here won't fire and `currentImage` will stay as two. The next time `draw()` runs we'll be controlling the second image and we'll keep doing so until the next time the user clicks the mouse. Shortly we'll examine how `draw()` uses `currentImage` to determine which image to control, but first lets look at what happens when the user clicks the mouse a couple of more times. A second click will increment `currentImage` again, setting it to three and again skipping the `if` statement. Now our third image appears and begins moving. On the third click, however, incrementing `currentImage` leaves its value as four. We have no fourth image to move, but thankfully the `if` statement here kicks in and we reset the value of `currentImage` back to one. The next time `draw()` runs, our first image will move around again for a second time.

Using this reset-to-one method, we've ensured that the user can cycle through the images and control each one in turn. However this means that one of the images will always be moving. What if we wanted to give the user the option to freeze all three of the images in place simultaneously once they've gotten them positioned how they want? If we change the line inside our `if` statement from `currentImage = 1` to `currentImage = 0` that will do the trick. Now, when the user hits the mouse for the third time, no image will be selected. There's no image that corresponds to the number zero so all the images will stay still. When they hit the mouse again `currentImage` will get incremented back to one and they'll be in control again. Go ahead and make that change and test out the sketch to see for your self.

But how does our `draw()` function use `currentImage` to decide which image to control? Just keeping `currentImage` set to the right value doesn't do anything by itself. We need to use its value to change the position of the corresponding image. To do this, we use a new technique called a switch-statement. A switch-statement is a tool for controlling the flow of our sketch much like an `if` statement. `If` statements decide whether or not to take some particular set of actions based on the value of a particular variable. Switch-statements, on the other hand, choose between a number of different options. With an `if` statement we can decide whether or not to reset of `currentImage` variable as we just saw in our `mousePressed` function. With a switch-statement we can choose which image position to update based on the value of our `currentImage` variable. Let's take a look at the switch-statement in this sketch. I'll explain the basic anatomy of a switch-statement

and then show you how we use this one in particular to give our user control of all three images.

```
switch(currentImage){
  case 1:
    image1X = interpolatedX;
    image1Y = interpolatedY;
    image1scale = map(closestValue, 610,1525, 0,4);
  break;
  case 2:
    image2X = interpolatedX;
    image2Y = interpolatedY;
    image2scale = map(closestValue, 610,1525, 0,4);
  break;
  case 3:
    image3X = interpolatedX;
    image3Y = interpolatedY;
    image3scale = map(closestValue, 610,1525, 0,4);
  break;
}
```

A switch-statement has two parts: the `switch()` which sets the value the statement will examine and the cases which tell Processing what to do with each different value that comes into the switch. We start off by passing `currentImage` to `switch()`, that's the value we'll be using to determine what to do. We want to set different variables based on which image the user is currently controlling. After calling `switch()` we have three `case` statements, each one determining a set of actions to take for a different value of `currentImage`. Each instance of `case` takes an argument in the form of a possible value for `currentImage`: `1`, `2`, or `3`. The code for each `case` will run when `currentImage` is set to its argument. For example, when `currentImage` is set to one, we'll set the value of `image1X`, `image1Y`, and `image1scale`. Then we'll `break+—we'll exit the switch-statement. None of the other code will run after the +break`. We won't update the positions or scales of any of the other images. That's how the switch-statement works to enforce our `currentImage` variable: it only lets one set of code run at a time depending on the variable's value.

Now, let's look inside each of these cases at how this switch-statement actually sets the position of the image once we've selected the right one. Inside of each case we use the interpolated value of the closest point to set the x- and y-values of the selected image. Before this point in the sketch we found the closest point for this run of the sketch and interpolated it with the most recent value to create a smoothly moving position. This code is just the same as we've seen throughout the basic version of this project and the entirety of our Invisible Pencil project. Now, we simply assign these `interpolatedX` and `interpolatedY` values to the correct variables for the current image: `image1X` and `image1Y`, `image2X` and `image2Y`, or `image3X` and `image3Y`. Which one we chose will be determined by which case of our switch-statement we entered.

The images that aren't current will have their x- any y-coordinates unchanged. Then, a little lower down in the sketch, we display all three images, using the variables with their x- and y-coordinates to position them:

```
image(image1,image1X,image1Y, ...);
image(image2,image2X,image2Y, ...);
image(image3,image3X,image3Y, ...);
```

The image that's currently selected will get set to its new position and the other two will stay where they are, using whatever value their coordinates were set to the last time the user controlled them. The result will be one image that follows the user's hand and two that stay still wherever the user last left them.

That concludes the code needed to control the location of the images and the decision about which image the user controls. But what about the images' size? We saw when we ran the sketch that the current image scaled up and down based on the user's distance from the Kinect. How do we make this work? Processing's `image()` function lets us set the size to display each image by passing in a width and a height. So, our strategy for controlling the size of each image will be to create two more variables for each image to store the image's width and height. We'll set these variables at the start of our sketch to correspond to each image's actual size. Then, when the user is controlling an image, we'll use the depth of the closest pixel to scale these values up and down. We'll only update the scale of the image that the user is actively controlling so the other images will stick at whatever size the user left them. Finally, when we call `image()` we'll pass in the scaled values for each image's width and height to set them to the right size. And voila: scaled images controlled by depth.

Let's take a look at the details of how this actually works in practice. We'll start at the top of the sketch with variable declarations.

We declare three additional variables for each image: `image1width`, `image1height`, and `image1scale` are the examples for image 1, there are parallel width, height, and scale variables for images 2 and 3 as well. We initialize the width and height variables to the actual sizes of the images we'll be using. In my case, I chose three images that are each 100 pixels square. So I set the widths and heights of all of the images to be 100. You should set these to match the dimensions of the images you're actually using. These values will never change throughout our sketch. They'll just get multiplied by our scale values to determine the size at which we'll display each image. Let's look at how those scale variables get set.

We've actually already seen where this happens: inside of our switch-statement. In addition to setting the x- and y-coordinates of our current image, we also set the scale in each `case` statement:

```
case 1:
  image1X = interpolatedX;
  image1Y = interpolatedY;
  image1scale = map(closestValue, 610,1525, 0,4);
break;
```

You can see from that example controlling `image1` that we use `map()` to scale `closestValue` from zero to four. The incoming range of depth values we're looking for here, 610 to 1525, were determined experimentally. I printed out `closestValue` using `println()`, waved my hand around in front of the Kinect, and examined the numbers that resulted. I chose these values as a reasonable minimum and maximum based on that experiment. So, when the `closestValue` seen by the Kinect was around 610, the image will scale down to nothing and as the closest point moves further away, the image will grow towards four times its original size. Just like with our position variables, the case-statement will ensure that only the scale of the current image is altered. Other images will retain the scale set by the user until the next time they become current.

But, again, just setting the value of `image1scale` (or `image2scale` or `image3scale`) is not enough. We have to use it when we call `image()` to determine the actual size at which each image is displayed. Let's look again at the arguments we pass to `image()`:

```
image(image1,image1X,image1Y, image1width * image1scale, image1height * image1scale);
image(image2,image2X,image2Y, image2width * image2scale, image2height * image2scale);
image(image3,image3X,image3Y, image3width * image3scale, image3height * image3scale);
```

For the width and height values for each image, we multiply their scale by their original width and height. The result will proportionally scale each image based on the value we just set from the user's distance to the Kinect. Now the image that the user controls will scale up and down as they move their hand in front of the Kinect and each image will freeze at its current size whenever the user hits the mouse button to cycle along to the next image.

This completes our Minority Report project. You now have hands-free control over the position and size of three images. You've created a sophisticated application that uses the depth data from the Kinect in multiple ways at once. You found the closest pixel to the Kinect and used its x- and y-coordinates as a control point for the user. You used the distance of this closest pixel to scale the size of images up and down. And you wrapped it all within a complex control flow that has multiple states and keeps track of a bunch of data to do it.

You're now ready to move on to the next chapter. There we'll start to tackle working with the data from the Kinect in 3D. We'll learn how to navigate and draw in three dimensions and we'll learn some techniques for making sketches interactive based on the user's position in space.

Exercises

Here are some exercises you can do to extend and improve this project. Some of them assume advanced skills that you might not have yet. If that's the case, don't worry. These exercises are just suggestions for things you could do to expand the project and practice your skills.

- Give all of the images starting positions so that they're visible when the sketch starts up.
- Add the ability to capture a screen grab of the current position of the images using Processing's `keyPressed` callback.
- Write a `ScalableImage` class that will remembers the position, size, and scale of each image. Using multiple instances of your class should dramatically clean up the repetitive variables in the project as it currently exists and make it easier to add multiple images.

About the Author

After a decade as a musician, web programmer, and startup founder, Greg Borenstein recently moved to New York to become an artist and teacher. His work explores the use of special effects as an artistic medium. He is fascinated by how special effects techniques cross the boundary between images and the physical objects that make them: miniatures, motion capture, 3D animation, animatronics, and digital fabrication. He is currently a grad student at NYU,Äôs Interactive Telecommunications Program.