9 WebGL: 3D application development

This chapter covers

- Developing a WebGL engine
- Communicating with a graphics card
- Creating 3D shapes

Web developers have been trying for years to overcome 3D limitations to create better interactive games, education tools, and infographics. In the past, plug-ins such as Unity, Flash, and Quicksilver created Google Maps and online 3D explorations programs. Plug-ins can be useful, but they leave you at the browser vendor's mercy for updates, usually lack hardware acceleration, and are often proprietary. To solve these issues, the Khronos Group created a Web Graphics Library (WebGL). WebGL, as mentioned in chapter 1, gives you the ability to create awesome 3D applications like X-Wing, shown in figure 9.1, without plug-ins. Several developers have even used WebGL to make drawing interfaces that create 2D images and rotate those creations in 3D.

WARNING! You should be very familiar with Canvas and JavaScript object-oriented programming (OOP) before working through this chapter's sample application. If you aren't, please go through chapter 6 on 2D Canvas first, because the concepts we cover here build on chapter 6's application, mainly because WebGL builds on top of the Canvas API.

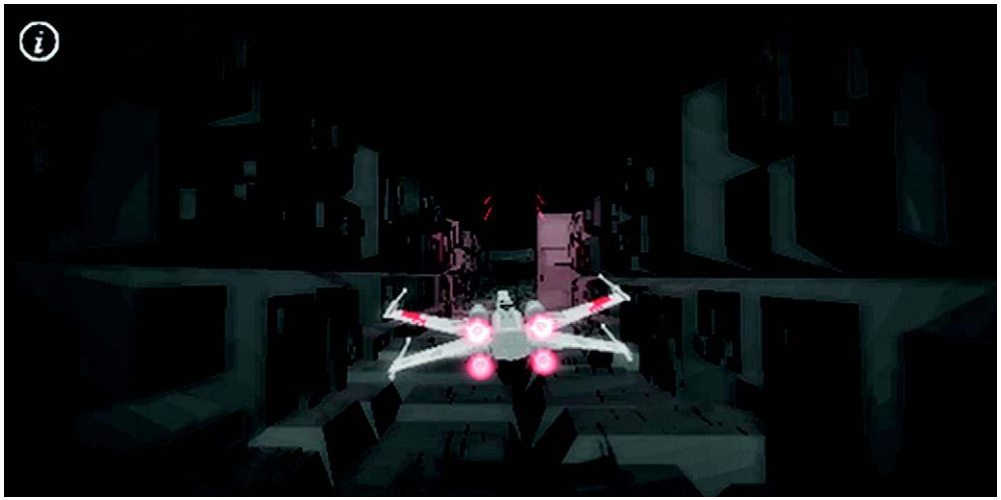

Figure 9.1 A simple WebGL application called X-Wing created by OutsideOfSociety. He worked on the popular WebGL project http://ro.me.

You could learn basic 3D programming elsewhere, but we've provided it all for you—all in one place—along with thorough explanations of 3D programming concepts, mathematics, diagrams, and more. We even teach you how to apply your new knowledge by walking you through the creation of a game: Geometry Destroyer!

Why build Geometry Destroyer?

Some online tutorials teach the basics of what you can do with WebGL. But this chapter's tutorial doesn't cover creating simple demos—you'll be creating a real application from the ground up. A few of the subjects you'll learn during the build include how to

- Create a reusable WebGL class
- Generate and maintain large numbers of WebGL entities
- Create different shape buffers with reusable code
- Work with assets in 2D and 3D space
- Handle 2D collision detection in 3D space with particle generation

In this chapter you'll first learn how to use WebGL to create an engine from scratch. Knowing how an engine works teaches you the fundamentals of managing 3D assets.

After you've built the engine's entity management to control visual objects, we'll walk you through making a request with WebGL, processing returned data, and displaying the resulting 3D shapes. For the last part of the lesson, we'll show you how to create your game's player and bullets with 2D shapes in 3D. We'll then expand on the

Need a prebuilt WebGL engine?

In a rush to get a WebGL application rolling? We recommend downloading CopperLicht for 3D gaming at http://www.ambiera.com/copperlicht/download.html. After you've downloaded the package, you should take a look at the documentation and demos at www.ambiera.com/copperlicht/documentation/ to get started. For any other projects (interactive data representations, architecture, animated videos, maps, and the like), grab a copy of Mr. Doob's three.js from GitHub at https://github.com/mrdoob/three.js. You'll find examples, documents, and usage guides to get you started at http://mng.bz/1iDu.

2D drawing ideas to create 3D rotating polygons that explode into cubes and squares when destroyed.

After completing this chapter, you'll understand how WebGL creates and manages 3D data. In addition, you'll walk away with a reusable basic WebGL engine and a fun game! Let's start by rolling out the engine's entity-management components.

9.1 Building a WebGL engine

Even though using a prebuilt engine can save a lot of time, it may cause problems if it doesn't support the functionality you need. We recommend rolling your own engine for JavaScript applications *when time permits*. You'll not only learn how to be a better programmer, you'll also create reusable code for future projects.

In this section, you'll learn the following reusable WebGL concepts:

- How to structure an engine that creates visual output
- How to create simple JavaScript inheritance with John Resig's script
- Where to get and how to use assets that make writing WebGL faster
- Methods for handling collisions, deletion, and other entity-management-related tasks

For example, the techniques you'll learn building Geometry Destroyer (figure 9.2) in this chapter will be transferable to other visual APIs such as Canvas and SVG.

WARNING: BUILDING AN ENGINE ISN'T EASY! If you don't want to copy and paste tons of JavaScript code to create the 3D engine, we recommend that you simply read along in sections 9.1 and 9.2 and then download the engine from Manning's source code. You can use that source code as your starting point and then write the game with us in section 9.3. Feeling adventurous and want to put your coding chops to work? Great! We invite you to build the engine from scratch by following and using the code listings.

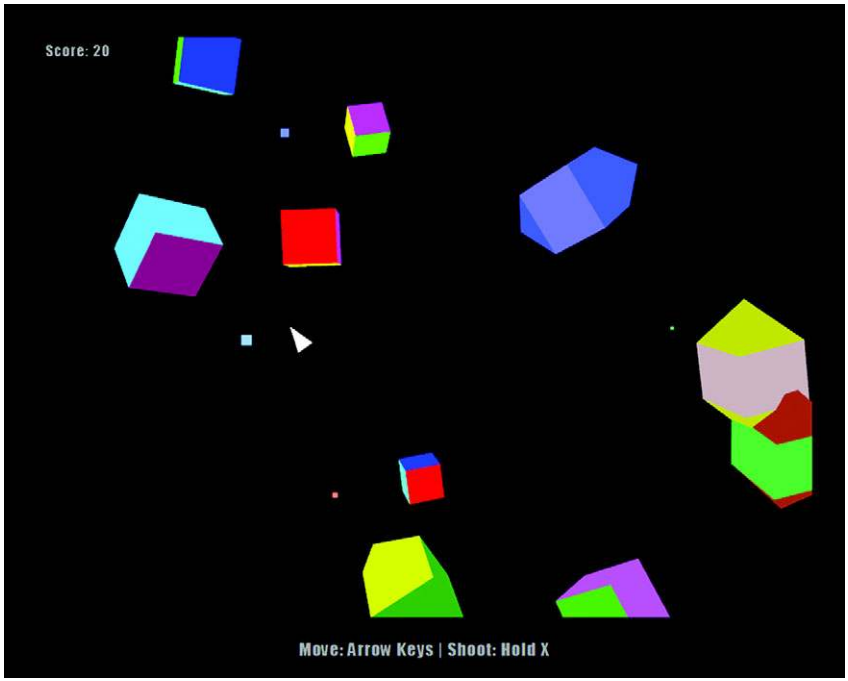

Figure 9.2 Get pumped to build your application by going to http://html5inaction.com/app/ch9/ and playing Geometry Destroyer before you build it. Download the source code from Manning's website at http://manning.com/crowther2/.

BROWSER NOTE: USE CHROME OR FIREFOX FOR THIS CHAPTER'S SAMPLE APPLICATION

Whether or not you're building the engine with us, we recommend that you use Google Chrome or Firefox's latest version. Other browsers may not support advanced 3D features or the necessary graphics acceleration. Although browsers may "support" WebGL, "support" doesn't mean that all features have been implemented.

WebGL for IE?

Want to enable WebGL in older versions of IE? Check out a plug-in called IEWebGL (http://iewebgl.com). It provides support for IE 6, 7, 8, 9, and 10. Because it's a downloaded executable, you can present it to users when they're using IE. Keep in mind that it doesn't work with our demo, but it works great with libraries like Three.js (see the site for a complete list).

WebGL	8	4	12	5.1

We've broken the engine-building work into seven steps to help you follow along and see the big picture:

- Step 1: Review/create the JavaScript code base and index.html.
- Step 2: Create style.css.
- Step 3: Implement time-saving scripts.
- Step 4: Create base engine logic.
- Step 5: Manage entity storage.
- Step 6: Create shape entities with 3D data.
- Step 7: Add reusable methods that speed up programming and make files easier to maintain.

Let's get started.

9.1.1 Setting up the engine's layout

Creating a WebGL engine requires several different developer tools and a file structure like the one you see in figure 9.3.

For now you can create an empty copy of each folder and file with the proper hierarchy ahead of time, or you can follow along and create each file and folder as we mention them. The JavaScript folder (named js) will house everything for your engine. Inside the JavaScript folder, place a run.js file and an engine folder. We're keeping engine's contents separate from everything else to keep things neatly organized.

GRAPHICS CARD WARNING Please note that not all graphics cards will support WebGL. If you're running the latest version of Chrome or Firefox and can't run the 3D files for this chapter on your hardware, the only solution we can think of is to try another computer. We apologize if you can't run WebGL; the lack of graphics card support has been frustrating for many developers.

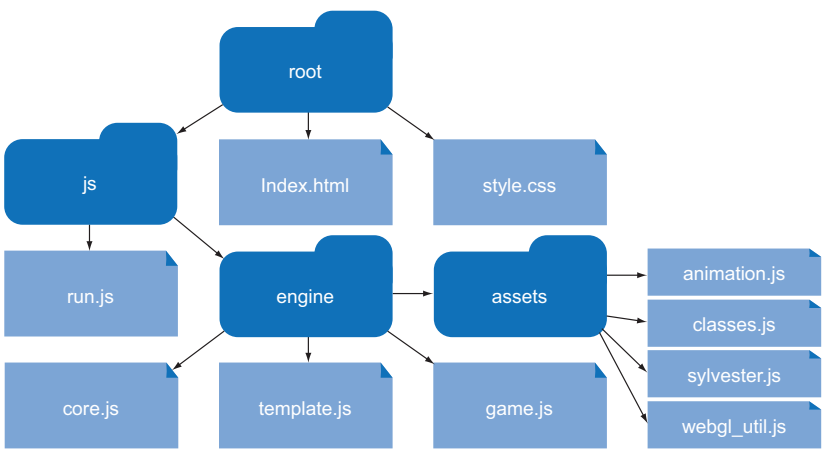

Figure 9.3 Your engine's file structure should be identical to this figure. We've organized it in a manner that's conducive to learning.

STEP 1: REVIEW/CREATE THE JAVASCRIPT CODE BASE AND INDEX.HTML

Create a file called index.html from the following listing, as a base for running all of your JavaScript code. You'll be including a `<canvas>` tag because WebGL runs on top of the Canvas API.

Listing 9.1 index.html—Creating the engine HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>Geometry Destroyer</title>
    <link rel="stylesheet" type="text/css" href="style.css" />
</head>

<body>
    <div id="container">
        <canvas id="canvas" width="800" height="600">
            Download Chrome to experience the demo!
        </canvas>

        <span id="score">Score: <span id="count">0</span></span>

        <p id="title" class="strong screen">Geometry Destroyer</p>
        <p id="start" class="screen">Push <span class="strong">X</span> to
     Start</p>

        <p id="end" class="screen hide">
            <span class="strong">Game Over</span>
        </p>

        <p id="ctrls">Move: Arrow Keys | Shoot: Hold X</p>
    </div>

    <script type="text/javascript" src="js/engine/assets/sylvester.js"></
     script>
    <script type="text/javascript" src="js/engine/assets/webgl_util.js"></
     script>
    <script type="text/javascript" src="js/engine/assets/animation.js"></
     script>
    <script type="text/javascript" src="js/engine/assets/classes.js"></
     script>
    <script type="text/javascript" src="js/engine/core.js"></script>
    <script type="text/javascript" src="js/engine/game.js"></script>
    <script type="text/javascript" src="js/engine/template.js"></script>
    <script type="text/javascript" src="js/run.js"></script>
</body>
</html>
```

Canvas is required to run WebGL. Make sure you include a canvas tag when running it.

Score counter.

Initial text presented to a player.

Text presented at Game Over.

Include all of your engine's JavaScript files here.

Can I use 2D Canvas in WebGL?

Sadly, you can't use 2D Canvas and the WebGL API in the same context. The trick to getting around this is to use two `<canvas>` elements to create two different contexts and then sit one on top of the other via CSS.

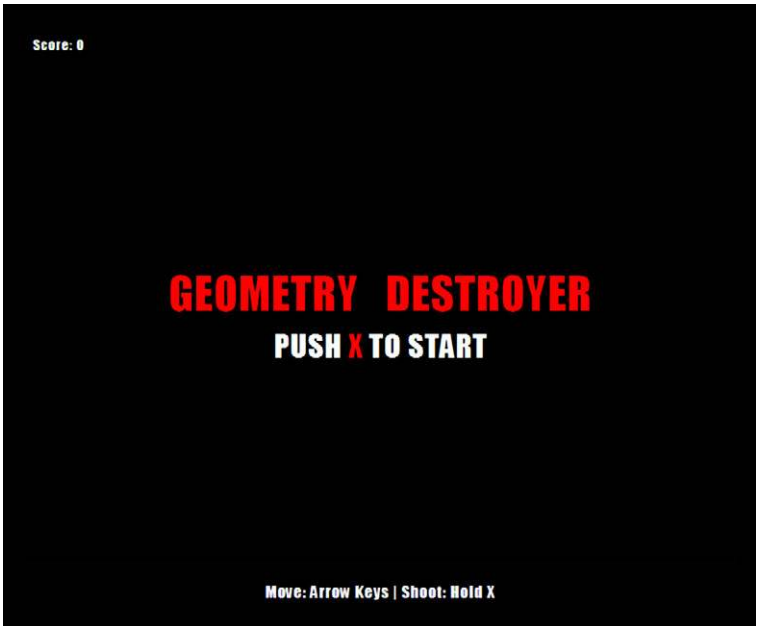

Figure 9.4 Result of running the index.html file with CSS and HTML only. In the final screen, the triangular player will appear between the words *Geometry* and *Destroyer*.

STEP 2: CREATE STYLE.CSS

Because creating text in WebGL isn't easy, you'll use text from HTML markup. We've included in the previous index.html listing an introduction and starting screen, but it needs some styling (see figure 9.4).

Place the next listing inside a new file called style.css. Put the file in the same folder that contains index.html.

Listing 9.2 style.css—Adding styling

```
body {
    background: #111;
    color: #aaa;
    font-family: Impact, Helvetica, Arial;
    letter-spacing: 1px;
}

#container {
    width: 800px;
    margin: 40px auto;
    position: relative;
}

#canvas {
    border: 1px solid #333;
}
```

```
#score {
    position: absolute;
    top: 5px;
    left: 8px;
    margin: 0;
    font-size: 15px;
}

.strong {
    color: #a00;
}

.screen {
    font-size: 34px;
    text-transform: uppercase;
    text-align: center;
    text-align: center;
    position: absolute;
    width: 100%;
    left: 0;
}

#title {
    top: 214px;
    font-size: 50px;
    word-spacing: 20px;
}

#start {
    top: 300px;
}

#end {
    top: 220px;
    display: none;
    font-size: 50px;
}

#ctrls {
    text-align: center;
    font-size: 18px;
}
```

STEP 3: IMPLEMENT TIME-SAVING SCRIPTS

Core API

Next, create a folder called js to house all of your JavaScript files. Inside create a file called run.js that will house all of your run code. Next to run.js create a folder called engine. Inside of engine create another folder called assets. You'll fill up the assets folder with four scripts that will save you time.

Getting your engine up and running requires several different external files. You'll need the following:

- Paul Irish's `requestAnimationFrame()` inside animation.js
- A slightly modified version of John Resig's Class Extension script called classes.js
- A transformation matrix library called sylvester.js
- Helpers from webgl_util.js

We'll explain exactly what each component does and how it aids your engine's functionality as we proceed.

PAUL IRISH'S REQUESTANIMATIONFRAME

Our goal is to equip your engine with best animation practices similar to those we discussed in chapter 6 on 2D Canvas. When we say "best animation practices," we mean

- Using `requestAnimationFrame()` instead of `setInterval` for mobile compatibility, to prevent updates when in another tab, and to prevent frame rate tearing
- Testing for the `requestAnimationFrame()` in other browsers with Paul Irish's polyfill and guaranteeing support for older browsers like IE8

To start building your dependencies, or the files your engine is dependent on, navigate to the assets folder. Inside create a file called animation.js using Paul Irish's `requestAnimationFrame()` shown in the following listing (http://mng.bz/h9v9).

Listing 9.3 animation.js—Requesting animation and intervals

```
window.requestAnimFrame = (function() {
    return window.requestAnimationFrame    ||
        window.webkitRequestAnimationFrame ||
        window.mozRequestAnimationFrame    ||
        window.oRequestAnimationFrame      ||
        window.msRequestAnimationFrame     ||
        function(callback) {
            window.setTimeout(callback, 1000 / 60);
        };
})();
```

JOHN RESIG'S SIMPLE JAVASCRIPT INHERITANCE

Because your engine requires you to create objects that can be modified, tweaked, and inherited on the fly, you need an extendable class. The problem is that classes usually require a robust library like prototype.js because JavaScript doesn't natively support them. To keep your engine's file size and dependencies limited, we're using a slightly modified version of John Resig's Simple JavaScript Inheritance script (http://ejohn.org/blog/simple-javascript-inheritance/). Insert a modified version of John Resig's script from the following listing into a file called classes.js in the assets folder.

Listing 9.4 classes.js—JavaScript inheritance

```
(function(){
    var initializing = false, fnTest = /xyz/.test(function(){xyz;}) ?
        /\b_super\b/ : /.*/;
    this.Class = function(){};

    Class.extend = function(prop) {
        var _super = this.prototype;

        initializing = true;
        var prototype = new this();
        initializing = false;
```

```
        for (var name in prop) {
            prototype[name] = typeof prop[name] == "function" &&
            typeof _super[name] == "function" && fnTest.test(prop[name]) ?
            (function(name, fn){
                return function() {
                    var tmp = this._super;

                    this._super = _super[name];

                    var ret = fn.apply(this, arguments);
                    this._super = tmp;

                    return ret;
                };
            })(name, prop[name]) :
            prop[name];
        }

        function Class() {}

        Class.prototype = prototype;
        Class.prototype.constructor = Class;
        Class.extend = arguments.callee;

        return Class;
    };
})();
```

The only piece of code we changed from the original inheritance script was removing a call to init() here. Originally, the script would automatically call init() if it were present on an object.

WANT MORE JAVASCRIPT?

If you want to learn more about JavaScript's prototype-based inheritance, pick up a copy of John Resig and Bear Bibeault's *Secrets of the JavaScript Ninja* (Manning, 2012). It's loaded with great techniques to help you work with libraries, create cross-browser solutions, and maintain your code.

SYLVESTER.JS

To create 3D shape objects, you also need to send the graphics card some packaged matrix information, such as `[0 1 3 0]`, but JavaScript doesn't have built-in tools for handling such information. You could write a matrix processing library for your engine from scratch, but it's quite a lot of work. Instead, you'll use sylvester.js to process everything. Get the latest version of the script from http://sylvester.jcoglan.com/, unzip it, and include the sylvester.js file in your assets folder.

WEBGL_UTIL.JS

The last asset you need is webgl_util.js, which contains lots of prewritten code to help with generating a perspective, processing matrixes, and more. We wish we could credit the author of this great script, but as Mozilla says, "Nobody seems entirely clear on where it came from." Grab the file at http://mng.bz/P7Vi and place it in assets.

> **Wait—didn't you say "custom rolled engine"?**
> Earlier we said that our WebGL tutorial centers on a built-from-scratch engine, which may lead you to ask, "Why are you making me use assets that aren't from scratch?" Truth is, we don't have time to custom roll everything; it would take at least 100 more pages to explain a complete engine step by step, so we thought that adding a few scripts to simplify everything was a good idea. We hope you agree!

9.1.2 Tools to create, alter, and delete objects

With your assets in place, let's get to work on the engine.

STEP 4: CREATE BASE ENGINE LOGIC

Core API

Use the following listing to create your first engine file, core.js, inside js/engine. With this listing, you are detecting WebGL support, setting up the base configuration for WebGL, creating a helper method to detect collisions, and creating placeholders for code in later listings.

Listing 9.5 core.js—Engine startup

```
var gd = gd || {};                                  // Inherits a previously existing gd variable or creates a new one. Great for accessing gd across multiple files.

gd.core = {
    canvas: document.getElementById("canvas"),

    size: function(width, height) {                 // WebGL requires you to set an aspect ratio; failure to do so will distort the correct aspect ratio of your canvas.
        this.horizAspect = width / height;
    },

    init: function(width, height, run) {
        this.size(width, height);

        if (!this.canvas.getContext) return alert('Please download ' +
            'a browser that supports Canvas like Google Chrome ' +
            'to proceed.');
        gd.gl = this.canvas.getContext("experimental-webgl");

        if (gd.gl === null || gd.gl === undefined)   // Manually check for WebGL support; some browsers return null and some undefined if getContext() fails.
            return alert('Uhhh, your browser doesn\'t support WebGL. ' +
                'Your options are build a large wooden badger ' +
                'or download Google Chrome.');

        gd.gl.clearColor(0.05, 0.05, 0.05, 1.0);    // Sets a clear color of slightly off-black for WebGL.
        gd.gl.enable(gd.gl.DEPTH_TEST);             // These two lines of code set up depth perception.
        gd.gl.depthFunc(gd.gl.LEQUAL);
        gd.gl.clear(gd.gl.COLOR_BUFFER_BIT | gd.gl.DEPTH_BUFFER_BIT);

        this.shader.init();
        this.animate();

        window.onload = run;                        // Fires the run code argument after everything has been set up.
    },

    animate: function() {
        requestAnimFrame(gd.core.animate);
```

```
        gd.core.draw();
    },

    shader: {
        init: function() {},
        get: function(id) {},
        store: function() {}
    },

    draw: function() {},

    overlap: function(
       x1, y1, width1, height1,
       x2, y2, width2, height2) {
         x1 = x1 - (width1 / 2);
         y1 = y1 - (height1 / 2);
         x2 = x2 - (width2 / 2);
         y2 = y2 - (height2 / 2);

         return x1 < x2 + width2 &&
            x1 + width1 > x2 &&
            y1 < y2 + width2 &&
            y1 + height1 > y2;
    }
};
```

Shaders will be covered later; this is a placeholder for now.

Drawing will be covered during graphic creation; this is currently a placeholder.

The gd.core.overlap() method is for detecting overlap between two squares.

WebGL objects are drawn from the center, and you need to calculate from the top left. You need to adjust the width and height calculations to account for that.

STEP 5: MANAGE ENTITY STORAGE

Now you need to manage entity storage and create a graveyard to handle cleaning out deleted entities. Add the following listing to complete core.js's entity management inside your existing `gd.core` object. These methods make maintaining entities significantly easier when you program the run.js file later.

Listing 9.6 core.js—Engine entity management

```
gd.core = {
    id: {
        count: 0,
        get: function() {
            return this.count++;
        }
    },

    storage: {
        all: [],
        a: [],
        b: []
    },

    graveyard: {
        storage: [],
        purge: function() {
            if (this.storage) {
                for (var obj = this.storage.length; obj--;) {
                    this.remove(this.storage[obj]);
                }
                this.graveyard = [];
```

Gives new entities a unique ID identifier. Speeds up searching for and deleting objects.

Storage container for holding all the objects you generate. The A and B containers are used to cut down on collision-detection comparisons by placing friendlies in A, enemies in B.

Used to destroy entities at the end of your update loop to prevent accidentally referencing a nonexistent entity.

```
            }
        },
        remove: function(object) {
            var obj;
            for (obj = gd.core.storage.all.length; obj--;) {
                if (gd.core.storage.all[obj].id === object.id) {
                    gd.core.storage.all.splice(obj, 1);
                    break;
                }
            }

            switch (object.type) {
                case 'a':
                    for (obj = gd.core.storage.a.length; obj--;) {
                        if (gd.core.storage.a[obj].id === object.id) {
                            gd.core.storage.a.splice(obj, 1);
                            break;
                        }
                    }
                    break;
                case 'b':
                    for (obj = gd.core.storage.b.length; obj--;) {
                        if (gd.core.storage.b[obj].id === object.id) {
                            gd.core.storage.b.splice(obj, 1);
                            break;
                        }
                    }
                    break;
                default:
                    break;
            }

            gd.gl.deleteBuffer(object.colorStorage);
            gd.gl.deleteBuffer(object.shapeStorage);
        }
    }
};
```

JavaScript's garbage cleanup is subpar. You need to manually purge 3D data from entities to prevent your application from slowing down.

STEP 6: CREATE SHAPE ENTITIES WITH 3D DATA

Core API

You need to set up an extendable class to create entities that contain 3D data. You'll use John Resig's Simple JavaScript Inheritance script that you added earlier in combination with a template object. Think of templates as molds for all of your game's reusable visual assets, such as players, enemies, and particles. Add the next listing in a file right next to core.js called template.js.

Listing 9.7 template.js—Entity default template

```
var gd = gd || {};

gd.template = {
    Entity: Class.extend({
        type: 0,

        x: 0,
        y: 0,
        z: 0,
```

Set the collision detection to a string of "a" = friendly, "b" = enemy, and "0" = passive. Friends and enemies will collide, but passive entities won't during collision detection.

Z-axis makes elements 3D; we'll cover this in more detail later.

We're using zoom to create an artificial camera in WebGL. Normally, a good chunk of extra programming is required, so it's kind of a hack to speed up programming.

```
        zoom: -80,

        position: function() {
            return [this.x, this.y, this.z + this.zoom];
        },

        width: 0,
        height: 0,

        update: function() {},

        collide: function() {
            this.kill();
        },

        kill: function() {
            gd.core.graveyard.storage.push(this);
        },

        rotate: {
            angle: 0,
            axis: false
        }
    })
};
```

Assembles and returns a position in a WebGL editable format.

update() is always called before an entity is drawn.

Collisions fire the kill method.

Send the entity to the graveyard for deletion before cp.core.draw() can run again.

Rotation will be used later to configure unique angles for entities.

STEP 7: ADD REUSABLE METHODS THAT SPEED UP PROGRAMMING AND MAKE FILES EASIER TO MAINTAIN

Core API

We know that the previous code doesn't directly create any 3D graphics, but it makes working with 3D much easier. Bear with us for one more code snippet, and we'll cover WebGL right after.

Let's create the last file, game.js, which will have several generic methods to speed up programming. These methods will slim down your run.js file and make it easier to maintain. Populate the game.js file in the engine directory with the following listing.

Listing 9.8 game.js—Entity helper methods

```
var gd = gd || {};

gd.game = {
    spawn: function(name, params) {
        var entity = new gd.template[name];

        entity.id = gd.core.id.get();

        gd.core.storage.all.push(entity);
        switch (entity.type) {
            case 'a':
                gd.core.storage.a.push(entity);
                break;
            case 'b':
                gd.core.storage.b.push(entity);
                break;
            default:
                break;
        }
```

gd.game.spawn() will generate any entity template when given a name with type String. It'll also pass any additional parameters to your init() method if you declared them.

Pushes the newly created entity into storage.

```
        if (arguments.length > 1 && entity.init) {
            var args = [].slice.call(arguments, 1);
            entity.init.apply(entity, args);
        } else if (entity.init) {
            entity.init();
        }
    },

    boundaries: function(obj, top, right, bottom, left, offset) {
        if (offset === undefined)
            offset = 0;

        if (obj.x < - this.size.width - offset) {
            return left.call(obj);
        } else if (obj.x > this.size.width + offset) {
            return right.call(obj);
        } else if (obj.y < - this.size.height - offset) {
            return bottom.call(obj);
        } else if (obj.y > this.size.height + offset) {
            return top.call(obj);
        }
    },

    rotate: function(obj) {
        var currentTime = Date.now();
        if (obj.lastUpdate < currentTime) {
            var delta = currentTime - obj.lastUpdate;

            obj.rotate.angle += (30 * delta) / obj.rotate.speed;
        }
        obj.lastUpdate = currentTime;
    },

    random: {
        polarity: function() {
            return Math.random() < 0.5 ? -1 : 1;
        },
        number: function(max, min) {
            return Math.floor(Math.random() * (max - min + 1) + min);
        }
    }
};
```

If you added additional arguments to init(), they'll be passed in via the currying technique of prefilling function arguments. John Resig blogs about curring in JavaScript.[1]

Allows you to easily set logic for leaving the game's play area. You'll need to manually set the game's width and height later because 3D environment units are subjective. Most 3D engines allow you to set measurements because none exist by default.

Rotation method will allow you to move an object around its center point (originally taken from Mozilla's WebGL tutorial).[2]

Random number generation helpers.

If everything was set up correctly, you can run index.html, and your browser's console will only inform you of no errors or that run.js doesn't exist. If you happened to create the run.js file earlier, it won't fire the error shown in figure 9.5.

Now that your engine's mechanics are set up, you need to complete it by sending your object's 3D data to a user's graphics card, then displaying the returned information.

---

[1] John Resig blog, "Partial Application in JavaScript," last updated February 2008, http://mng.bz/6SU0.

[2] "Animating objects with WebGL," Mozilla Developer Network, last updated Aug 7, 2012, http://mng.bz/O5Z2.

Figure 9.5 If you load up index.html and take a look at your console, it will display no errors or that run.js is missing. Know that if you've created a run.js file already, it won't fire the shown error.

9.2 Communicating with a graphics card

While a war rages on to establish online standards, so does another for computer graphics. OpenGL and Direct X are two heavily competing graphics API libraries for 3D applications. Although the two have many differences between them, you mainly need to know that OpenGL is open source and Direct X is proprietary. Because of OpenGL's open source nature, support for its internet baby, WebGL, has grown significantly.

> **NOTE** We're deeply indebted to Mozilla's WebGL tutorials (https://developer.mozilla.org/en/WebGL) and Learning WebGL's lessons (http://learningwebgl.com) for the code you'll be using in this section. Thanks, Mozilla and WebGL!

Core API

OpenGL is a cross-platform library for Mac OS X, Unix/Linux, and Windows. It allows for graphics hardware control at a low level. WebGL is based on OpenGL ES (OpenGL for Embedded Systems), which is a subset of OpenGL for mobile devices. Although WebGL's ability to render 3D data via browser seems great, it's also violating the internet's security model of not letting web pages access hardware. The good news, though, is that browsers integrate extra security features to "hopefully" prevent someone from setting your graphics card on fire, stealing graphic memory, and/or launching DoS attacks (more details at http://www.contextis.com/resources/blog/webgl2/). We're going to be optimistic here and assume those things won't happen.

In this section, you'll learn how

- WebGL processes data inside a computer
- To create shader data and store
- To create and store shape data with buffers
- To manipulate matrices to output assembled 3D data on a screen
- To use a few scripts that make writing matrices easier

Let's start by looking at how WebGL renders data before you see it.

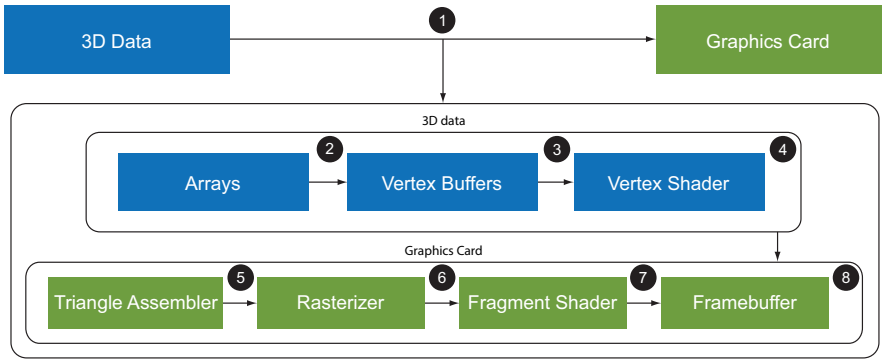

Figure 9.6 A clean version of the rendering pipeline. Although not a be-all-end-all explanation, it explains the basic steps WebGL goes through as it processes 3D data from start to finish.

9.2.1 Graphics cards: a quick primer

Consider the game you're creating: How will a user's browser process and display the 3D data for your objects? Take a look at figure 9.6.

What figure 9.6 shows you is that when sending over the 3D data ❶ for entities to a *graphics card*, the data starts as *arrays* ❷ (computer data) and gets processed by the GPU (graphics processing unit) into *vertex buffers* ❸ (more data). During this rendering stage, additional information is required to assemble your 3D shapes (such as buffer variables). After processing vertex buffers, the data runs through a *vertex shader* ❹ to generate screen positioning and color information. 3D data is then further processed by the GPU into triangle segments through the *triangle assembler* ❺ and then sent to a *rasterizer* ❻ that removes unnecessary visual data from shapes, generates pixel fragments, and smooth's out color surfaces. Shape data then flows through a *fragment shader* ❼, which outputs color and depth for each pixel. Lastly, everything is drawn onto a user's screen by a *framebuffer* ❽.

> **3D graphics and triangles? I don't get it.**
> When you're learning to create shapes with a 2D surface, you usually create a rectangle first. But it isn't the simplest of shapes, and you can't easily fit a bunch of tiny rectangles together to create a person's face or a ball. On the other hand, tiny triangles can fit together to easily create almost any shape imaginable. For a great overview of triangles in 3D programming, see Rene Froeleke's article "Introduction to 3D graphics" at http://mng.bz/STHc.

If you need more detailed information on how WebGL processes data, we recommend reading Opera's explanation at http://mng.bz/4Lao. Our version is quick and simple, because we don't want to put you to sleep.

MEANWHILE, BACK AT THE ENGINE

Your engine currently doesn't communicate with a graphics card. To do so, you'll follow two groups of steps:

Group 1—Creating shaders and buffers	Group 2—Working with matrices and drawing shapes
▪ Step 1: Create and configure color, vertex, and shape shaders via OpenGL ES. ▪ Step 2: Set up shader retrieval from the DOM. ▪ Step 3: Pull shader data from the DOM. ▪ Step 4: Create shape, color, and dimension buffers for entities.	▪ Step 1: Use matrices and buffers to visually output information. ▪ Step 2: Bind and draw shapes. ▪ Step 3: Detect overlap and remove entities. ▪ Step 4: Add matrix helpers to simplify matrix interaction. ▪ Step 5: Add Vlad Vukićević's WebGL helpers for rotation.

Once you've completed these tasks, you'll be ready to program the game.

9.2.2 Creating shaders for 3D data

Before you begin with the Group 1 set of tasks, pick up Jacob Seidelin's helpful WebGL Cheat Sheet at http://blog.nihilogic.dk/2009/10/webgl-cheat-sheet.html. It breaks down all of the methods for WebGL's context into categories such as shaders, buffers, and more, which will help as you move through these next few sections.

WHAT ARE SHADERS AGAIN?

We're throwing "shaders" around like it's a hip word. A long time ago it may have meant shading in shapes with color, but now it means much more than that. Today's shaders program the GPU for transformations, pixel shading, and special effects such as lighting.

STEP 1: CREATE AND CONFIGURE COLOR, VERTEX, AND SHAPE SHADERS VIA OPENGL ES

Core API

To start up your shaders, `gd.core.shader.init()` needs to call `gd.core.shader.get()` and `gd.core.shader.store()` to retrieve shading data. In addition, you'll need to write a little bit of code in a mystery language—OpenGL ES (see the sidebar on OpenGL ES for more information)—and place that code in your HTML document. Add the following listing inside index.html right before your JavaScript files. Note that if you put it anywhere other than right before your JavaScript files, your game will probably fail to load.

Listing 9.9 index.html—Color, vertex, and shape shaders

```
<script id="shader-vertex" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    attribute vec4 aVertexColor;

    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;
```

Configuration for position and color in your shaders.

Uniform declares this is a constant variable, and mat4 references a 4-by-4 float matrix.

Varying declares color data will vary over the surface of a primitive shape.

Stores your data inside appropriate variables.

```
    varying lowp vec4 vColor;

    void main(void) {
        gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
        vColor = aVertexColor;
    }
</script>

<script id="shader-fragment" type="x-shader/x-fragment">

    varying lowp vec4 vColor;

    void main(void) {
        gl_FragColor = vColor;
    }
</script>
```

shader-vertex handles position and vertex info; shader-fragment handles color assignment.

OpenGL ES shading language cheat sheet

OpenGL ES is a subset of OpenGL aimed at embedded systems such as mobile phones, game consoles, and similar devices. The Khronos Group has compiled a PDF for WebGL that contains a cheat sheet on OpenGL ES Shading Language. It significantly helps with writing your own custom shader scripts. Pick up your copy at http://mng.bz/1TA3.

Step 2: Set up shader retrieval from the DOM

With your shader scripts configured, you need to process them via JavaScript. Replace `gd.core.shader.init()` with the following listing in core.js.

Listing 9.10 core.js—Shader setup

Pulls shader programs from the DOM. Notice that shader-fragment and shader-vertex reference the two shader scripts you wrote.

Creates a "program" for your shader (holds one fragment and vertex shader).

Links your shaders and newly created "program" together.

Failsafe in case shaders crash as they're loading.

Stores the shader data for later use.

Clears out leftover shader data so it doesn't sit uselessly in memory. You could delete these shaders manually by waiting for JavaScript's garbage collector, but this gives more control.

```
gd.core = {
    shader: {
        init: function() {
            this.fragments = this.get('shader-fragment');
            this.vertex = this.get('shader-vertex');

            this.program = gd.gl.createProgram();

            gd.gl.attachShader(this.program, this.vertex);
            gd.gl.attachShader(this.program, this.fragments);
            gd.gl.linkProgram(this.program);

            if (!gd.gl.getProgramParameter(this.program, gd.gl.LINK_STATUS)) {
                return alert("Shaders have FAILED to load.");
            }

            gd.gl.useProgram(this.program);

            this.store();

            gd.gl.deleteShader(this.fragments);
            gd.gl.deleteShader(this.vertex);
            gd.gl.deleteProgram(this.program);
        }
    }
};
```

STEP 3: PULL SHADER DATA FROM THE DOM

In the previous listing, `gd.core.shader.init()` accesses the shader-vertex and shader-fragment scripts you put in index.html. `gd.core.shader.get()` retrieves and processes your shader by pulling it from the DOM, sending back a compiled package of data or an error. `gd.core.shader.init()` continues processing and attaches your DOM results to a program. The program sets up vertices, fragments, and color in a store method. Lastly, all the leftover graphics data is deleted. Replace `gd.core.shader.get()` and `gd.core.shader.store()` with the next listing in core.js to complete loading your shaders.

Listing 9.11 core.js—Shader retrieval

```
gd.core = {
    shader: {
        get: function(id) {
            this.script = document.getElementById(id);

            if (!this.script) {
                alert('The requested shader script was not found ' +
                    'in the DOM. Make sure that gd.shader.get(id) ' +
                    'is properly setup.');
                return null;
            }

            this.source = "";
            this.currentChild = this.script.firstChild;

            while (this.currentChild) {
                if (this.currentChild.nodeType ===
➥ this.currentChild.TEXT_NODE) {
                    this.source += this.currentChild.textContent;
                }
                this.currentChild = this.currentChild.nextSibling;
            }

            if (this.script.type === 'x-shader/x-fragment') {
                this.shader = gd.gl.createShader(gd.gl.FRAGMENT_SHADER);
            } else if (this.script.type === 'x-shader/x-vertex') {
                this.shader = gd.gl.createShader(gd.gl.VERTEX_SHADER);
            } else {
                return null;
            }

            gd.gl.shaderSource(this.shader, this.source);
            gd.gl.compileShader(this.shader);

            if (!gd.gl.getShaderParameter(this.shader,
➥ gd.gl.COMPILE_STATUS)) {
                alert('Shader compiling error: ' +
                    gd.gl.getShaderInfoLog(this.shader));
                return null;
            }

            return this.shader;
        },
```

No shader script in the DOM? Return nothing and an error.

Returns the compiled shader data after being collected via a while loop.

Tests what kind of shader is being used (fragment or vertex) and processes it based on the results.

Takes all of your shader data and compiles it together.

Compile success? If not, fire an error.

```
        store: function() {
            this.vertexPositionAttribute =
                gd.gl.getAttribLocation(
                    this.program, "aVertexPosition");
            gd.gl.enableVertexAttribArray(this.vertexPositionAttribute);

            this.vertexColorAttribute = gd.gl.getAttribLocation(
                this.program, "aVertexColor");
            gd.gl.enableVertexAttribArray(this.vertexColorAttribute);
        }
    }
};
```

Retrieves vertex data from your shader program for rendering 3D objects later.

Color data retrieval from shader program.

9.2.3 Creating buffers for shape, color, and dimension

With all that shader data present, you now need to create buffers for shape, color, and dimension. One interesting fact about buffer data is that each object will have its own independent set of buffers.

STEP 4: CREATE SHAPE, COLOR, AND DIMENSION BUFFERS FOR ENTITIES

Core API

To buffer your data, open template.js and append `gd.template.Entity.shape()`, `gd.template.Entity.color()`, and `gd.template.Entity.indices()` to the `Entity` object with the following listing.

Listing 9.12 template.js—Buffer configuration

```
gd.template = {
    Entity: Class.extend({
        shape: function(vertices) {
            this.shapeStorage = gd.gl.createBuffer();
            gd.gl.bindBuffer(gd.gl.ARRAY_BUFFER, this.shapeStorage);
            gd.gl.bufferData(gd.gl.ARRAY_BUFFER,
                new Float32Array(vertices), gd.gl.STATIC_DRAW);

            this.shapeColumns = 3;
            this.shapeRows = vertices.length / this.shapeColumns;
        },

        color: function(vertices) {
            this.colorStorage = gd.gl.createBuffer();

            if (typeof vertices[0] === 'object') {

                var colorNew = [];

                for (var v = 0; v < vertices.length; v++) {
                    var colorLine = vertices[v];
                    for (var c = 0; c < 4; c++) {
                        colorNew = colorNew.concat(colorLine);
                    }
                }

                vertices = colorNew;
            }

            gd.gl.bindBuffer(gd.gl.ARRAY_BUFFER, this.colorStorage);
            gd.gl.bufferData(gd.gl.ARRAY_BUFFER,
                new Float32Array(vertices), gd.gl.STATIC_DRAW);
```

When creating a shape you'll pass in vertices, and this method will take care of everything else.

Creates buffer data.

Stores created buffer data so you can use it.

Uses float32 to change the array into a WebGL editable format.

At the end of each method you need to record information about the passed array because your dependency sylvester.js requires extra array details.

A helper to disassemble large packages of color data.

```
            this.colorColumns = 4;
            this.colorRows = vertices.length / this.colorColumns;
        },

        indices: function(vertices) {
            this.indicesStorage = gd.gl.createBuffer();
            gd.gl.bindBuffer(gd.gl.ELEMENT_ARRAY_BUFFER,
                this.indicesStorage);
            gd.gl.bufferData(gd.gl.ELEMENT_ARRAY_BUFFER,
                new Uint16Array(vertices), gd.gl.STATIC_DRAW);

            this.indicesCount = vertices.length;
        }
    })
};
```

Indices is plural for index. In WebGL buffers are used to assemble triangles into a single shape. By using indices you can define the location of a pair of triangles, instead of just one at a time.

To use the buffer methods you created, you'll need to manually call `this.shape()`, `this.color()`, and possibly `this.indices()` when you create a new entity. More on how to use these new methods when you program run.js later in this chapter. In order to output the created buffer data, you'll need to configure `gd.core.draw()` next.

9.2.4 Displaying shape data on a screen

Using `gd.core.draw()`, you'll loop through all of the current entities in `gd.core.storage.all`. For each entity, you'll use a three-step process that spans three code listings, which means you need to make sure each of the next three listings continues from the previous one or the code won't work. Note also that we're now working through the second group of steps.

- Group 2—Working with matrices and drawing shapes
  - Step 1: Use matrices and buffers to visually output information.
  - Step 2: Bind and draw shapes.
  - Step 3: Detect overlap and remove entities.
  - Step 4: Add matrix helpers to simplify matrix interaction.
  - Step 5: Add Vlad Vukićević's WebGL helpers for rotation.

STEP 1: USE MATRICES AND BUFFERS TO VISUALLY OUTPUT INFORMATION

Core API

Let's start step 1 by opening core.js and replacing `gd.core.draw()` with listing 9.13. The listing will clear out the canvas's previous draw data and set the current perspective to draw all entities currently in storage. For all of the entities, it will run their update and rotation logic if it's configured. Be careful with the `for` loop in this listing, because it's continued for two more listings (up to listing 9.15).

Listing 9.13 core.js—Drawing shapes

```
gd.core = {
    draw: function() {
        gd.gl.clear(gd.gl.COLOR_BUFFER_BIT | gd.gl.DEPTH_BUFFER_BIT);

        this.perspectiveMatrix = makePerspective(45, this.horizAspect,
            0.1, 300.0);
```

Wipes your WebGL viewport clean to draw a brand-new frame.

Sets the viewing perspective from 1 to 300 units of distance (prevents aspect ratio distortion).

```
        for (var i in this.storage.all) {
            this.loadIdentity();
            this.storage.all[i].update();

            this.mvTranslate(this.storage.all[i].position());
            this.mvPushMatrix();

            if (this.storage.all[i].rotate.axis) {
                this.mvRotate(
                    this.storage.all[i].rotate.angle,
                    this.storage.all[i].rotate.axis);
            }
        }
    };
```

Loops through every entity in storage and draws it. The for statement doesn't end in this listing because it's continued in the next two.

Resets and creates a matrix that has 1s diagonally and 0s everywhere else[3].

Run the update() before outputting shapes to prevent new entities from showing up in the wrong location for a split second.

Grabs x, y, and z coordinates from your entity to clarify a draw location and pushes it into an array.

If rotate data is present, it will be run here.

Standardized method for pushing the current matrix item to the top of the matrix stack.

STEP 2: BIND AND DRAW SHAPES

With the matrix set up properly and rotation applied, you need to output the buffer information for the current 3D object. Do this by binding 3D data and then outputting it through `gd.gl.vertexAttribPointer()`, which passes along bound buffer data. Use the next listing to continue your `gd.core.draw()` method.

Listing 9.14 core.js—Drawing shapes (continued)

```
gd.core = {
    draw: function() {
            gd.gl.bindBuffer(
                gd.gl.ARRAY_BUFFER,
                this.storage.all[i].shapeStorage);
            gd.gl.vertexAttribPointer(
                this.shader.vertexPositionAttribute,
                this.storage.all[i].shapeColumns,
                gd.gl.FLOAT,
                false, 0, 0);

            gd.gl.bindBuffer(
                gd.gl.ARRAY_BUFFER,
                this.storage.all[i].colorStorage);
            gd.gl.vertexAttribPointer(
                this.shader.vertexColorAttribute,
                this.storage.all[i].colorColumns,
                gd.gl.FLOAT,
                false, 0, 0);

            this.setMatrixUniforms();

            if (this.storage.all[i].indicesStorage) {
                gd.gl.drawElements(
                    gd.gl.TRIANGLES,
                    this.storage.all[i].indicesCount,
```

Binds ARRAY_BUFFER to your shapeStorage object.

Defines an array of generic vertex attribute data.

Depending on whether or not indices were used, the buffer data needs to be output differently.

Pushes your matrix data from JavaScript to WebGL so the shaders can be properly seen.

[3] Weisstein, Eric W., "Identity Matrix," MathWorld, a Wolfram Web Resource, http://mng.bz/CO1M.

```
                    gd.gl.UNSIGNED_SHORT,
                    0);
            } else {
                gd.gl.drawArrays(
                    gd.gl.TRIANGLE_STRIP,
                    0,
                    this.storage.all[i].shapeRows);
            }

            this.mvPopMatrix();
    }
};
```

Removes an item from the current matrix stack.

NOTE We know it's frustrating that you can't see 3D models by simply refreshing your browser. Bear with us to output 3D models through the engine's `draw` loop, and we'll show you the awesome result of what you've created.

STEP 3: DETECT OVERLAP AND REMOVE ENTITIES

You've completed your output for 3D objects, but you need to append one more chunk of code to `cp.core.draw()` with the following listing. It will add optimized collision detection to properly monitor `a` (friendly) to `b` (enemy) overlap and clean up your graveyard.

Listing 9.15 core.js—Drawing shapes (continued)

```
gd.core = {
    draw: function() {
            if (this.storage.all[i].type === 'a') {
                for (var en = this.storage.b.length; en--;) {
                    if (this.overlap(
                    this.storage.all[i].x,
                    this.storage.all[i].y,
                    this.storage.all[i].width,
                    this.storage.all[i].height,
                    this.storage.b[en].x,
                    this.storage.b[en].y,
                    this.storage.b[en].width,
                    this.storage.b[en].height)) {
                        this.storage.all[i].collide(this.storage.b[en]);
                        this.storage.b[en].collide(this.storage.all[i]);
                    }
                }
            }

        this.graveyard.purge();
    }
};
```

Collision detection compares `a` type and `b` type entities to minimize logic.

Closes the `for` statement from two listings back.

Deleted elements are dumped out of the graveyard. This is accomplished here instead of in the loop to prevent accidentally referencing a nonexistent entity.

PROGRESS CHECK!

Now is a good time to check your browser's console for errors other than run.js being missing. If so, you're good to move on to the next section.

STEP 4: ADD MATRIX HELPERS TO SIMPLIFY MATRIX INTERACTION.

Core API

For `gd.core.draw()` you'd normally have to write some extremely complex logic to handle matrices for colors and shapes. Instead, you're going to use some prewritten helpers for modelview (http://3dengine.org/Modelview_matrix), perspective (http://mng.bz/VitL), and identity matrices (http://en.wikipedia.org/wiki/Identity_matrix). Append listing 9.16 to your `gd.core` object. Like webgl_util.js, the following chunk of code comes from an unknown source, but you'll find that Mozilla's WebGL tutorials, Learning WebGL, and many other online lessons make use of it.

Listing 9.16 core.js—Matrix helpers

```
gd.core = {
    loadIdentity: function() {
        mvMatrix = Matrix.I(4);
    },
    multMatrix: function(m) {
        mvMatrix = mvMatrix.x(m);
    },
    mvTranslate: function(v) {
        this.multMatrix(Matrix.Translation($V([v[0], v[1],
            v[2]])).ensure4x4());
    },
    setMatrixUniforms: function() {
        var pUniform = gd.gl.getUniformLocation(
            this.shader.program, "uPMatrix");
        gd.gl.uniformMatrix4fv(pUniform, false,
            new Float32Array(this.perspectiveMatrix.flatten()));

        var mvUniform = gd.gl.getUniformLocation(
            this.shader.program, "uMVMatrix");
        gd.gl.uniformMatrix4fv(
            mvUniform, false, new Float32Array(mvMatrix.flatten()));
    }
};
```

Loads up an identity matrix, which is a series of 1s surrounded by 0s.

Multiplies a matrix[4].

Runs matrix multiplication and then translation[5].

Sets the perspective and model view matrix.

STEP 5: ADD VLAD VUKIĆEVIĆ'S WEBGL HELPERS FOR ROTATION.

The code in listing 9.17 comes from Mozilla's site at http://mng.bz/BU9f. Mozilla tells us that "these routines were borrowed from a sample previously written by Vlad Vukićević," whose blog you can find at http://blog.vlad1.com. Vlad has created a couple of tools to help with rotation and with pushing and popping data. Append his rotation logic to `gd.core` with the following code.

Listing 9.17 core.js—Vlad Vukićević utilities

```
gd.core = {
    mvMatrixStack: [],

    mvPushMatrix: function(m) {
        if (m) {
```

Your stack will be used to manipulate matrix data with the following methods.

Moves given data to the top of the stack.

[4] "Matrix multiplication," Wikipedia, last modified April 8, 2013, http://mng.bz/yo4D.

[5] "Translation (geometry)," Wikipedia, last modified Feb. 21, 2013, http://mng.bz/2dbB.

```
            this.mvMatrixStack.push(m.dup());
            mvMatrix = m.dup();
        } else {
            this.mvMatrixStack.push(mvMatrix.dup());
        }
    },

    mvPopMatrix: function() {
        if (! this.mvMatrixStack.length) {
            throw("Can't pop from an empty matrix stack.");
        }

        mvMatrix = this.mvMatrixStack.pop();
        return mvMatrix;
    },

    mvRotate: function(angle, v) {
        var inRadians = angle * Math.PI / 180.0;

        var m = Matrix.Rotation(inRadians, $V([v[0], v[1],
     v[2]])).ensure4x4();
        this.multMatrix(m);
    }
};
```

Pop in JavaScript refers to an array method that removes the last element from an array and returns that value to the caller. Here, mvPopMatrix() is returning an error or removing and returning the last item.

This is the method that fires rotation in cp.core.draw().

PROGRESS CHECK!

Run index.html now and check your browser's console. You should see the screen shown in figure 9.7, possibly without the missing-file error. If you get additional errors or have trouble with your engine's code as you proceed, you might find it easier and less frustrating to replace the engine files with chapter 9's source code instead of debugging files. Debugging WebGL is a bit of a nightmare because browsers don't have easily accessible graphic monitoring tools.

With the last of the utility helpers in place, you should now feel somewhat comfortable with graphics card communication, comfortable enough to write basic 3D output for a WebGL application at least. Next, we'll take the foundation you created and use it to build your interactive 3D game: Geometry Destroyer.

Figure 9.7 Your code should output the displayed error of "run.js is missing" or no errors at all when running index.html. If you have trouble with the engine files as you proceed, just replace them with the source files from Manning's website. It's a nightmare to debug WebGL because of browsers not having easily accessible graphic monitoring tools.

9.3 Putting it all together: creating Geometry Destroyer

Creating 3D shapes is tough, but you just created (or read through as we created) a 3D engine that will significantly simplify the process. You can create new entities and attach 3D data via matrices; the engine will take care of outputting all the data for you. The engine will also take care of cleaning data out of memory whenever you need to.

In this section, you'll build a cool game as you learn to

- Write a simple matrix to output shape and color in 3D space
- Create 3D rotation data and use it with a controller to indicate direction in 2D
- Create and control entity generations for enemies and particles
- Use indices to turn triangles into squares for easy matrix creation
- Draw simple 2D shapes in 3D, plus unique polygons and cubes

As you understand how to create entities, you'll learn about 3D modeling and efficient OOP programming. If you don't have any knowledge about creating 3D shapes or entity management, don't worry; we'll guide you along the way.

Prereqs: play the game, grab the code, and test your engine

If you haven't done so already, head over to http://html5inaction.com/app/ch9/ and play the game. And make sure you pick up the game's files from http://www.manning.com/crowther2/ by downloading *HTML5 in Action*'s source files.

The work in this section is bundled into three groups of steps:

Group 1—Making your player	Group 2—Outputting enemies	Group 3—Generating particles
▪ Step 1: Capture user input. ▪ Step 2: Program the heads-up display. ▪ Step 3: Create the 2D player entity. ▪ Step 4: Animate the player entity. ▪ Step 5: Create the player's bullets.	▪ Step 1: Create a 3D polygon enemy. ▪ Step 2: Create a complex 3D model. ▪ Step 3: Generate random enemy properties. ▪ Step 4: Resolve enemy collisions. ▪ Step 5: Spawn enemies in a controlled manner.	▪ Step 1: Create a 3D cube particle. ▪ Step 2: Add color, rotation, and index data for cubes. ▪ Step 3: Add size, type, and other cube metadata. ▪ Step 4: Generate square particles.

Let's dive in to the first group and make your player.

9.3.1 Creating a game interface and control objects

The first thing we'll focus on is setting up the intro screen's non-3D logic, the result of which appears in figure 9.8.

Figure 9.8 The first thing you'll do is set up the intro screen logic. After that, you'll create the triangular player between the words *Geometry* and *Destroyer*, which you haven't seen previously.

STEP 1: CAPTURE USER INPUT

In your js folder, create and/or open run.js in the text editor of your choice. You should notice that it's completely blank. Set up the game's basic input monitor and methods by inserting everything into a self-executing function with the following listing in run.js. Make sure to place all code from here on out in this self-executing function to prevent variables from leaking into the global scope.

Listing 9.18 run.js–Initial game setup

```
(function() {
    gd.core.init(800, 600, function() {
        Ctrl.init();
        Hud.init();
        gd.game.spawn('Player');
    });

    gd.game.size = {
        width: 43,
        height: 32
    };

    var Ctrl = {
        init: function() {
            window.addEventListener('keydown', this.keyDown, true);
            window.addEventListener('keyup', this.keyUp, true);
        },
```

Declares width, height, and game setup logic to fire after loading your engine.

Place all code from here on out inside the self-executing function to prevent variables from leaking into the global scope.

The width and height of the play area in 3D units. Everything is measured from the middle with a Cartesian graph, so this is only half the width and height.

Controller for user input.

```
        keyDown: function(event) {
            switch(event.keyCode) {
                case 38: Ctrl.up = true; break;
                case 40: Ctrl.down = true; break;
                case 37: Ctrl.left = true; break;
                case 39: Ctrl.right = true; break;
                case 88: Ctrl.x = true; break;
                default: break;
            }
        },

        keyUp: function(event) {
            switch(event.keyCode) {
                case 38: Ctrl.up = false; break;
                case 40: Ctrl.down = false; break;
                case 37: Ctrl.left = false; break;
                case 39: Ctrl.right = false; break;
                case 88: Ctrl.x = false; break;
                default: break;
            }
        }
    };
}());
```

Up arrow.

Down arrow.

Left arrow.

Right arrow.

x keyboard key.

STEP 2: PROGRAM THE HEADS-UP DISPLAY

Controller input is now detectable, and the game engine will launch as expected. But you still need to create the heads-up display (HUD) to manage score and initial setup. You also need the player, but let's start with the HUD by creating a new variable called `Hud` below `Ctrl` with the following listing.

Listing 9.19 run.js—Heads-up display (HUD)

```
var Hud = {
    init: function() {
        var self = this;

        var callback = function() {
            if (Ctrl.x) {
                window.removeEventListener('keydown', callback, true);
                PolygonGen.init();
                self.el.start.style.display = 'none';
                self.el.title.style.display = 'none';
            }
        };

        window.addEventListener('keydown', callback, true);
    },

    end: function() {
        var self = this;
        this.el.end.style.display = 'block';
    },

    score: {
        count: 0,
        update: function() {
```

Begins polygon generation when a players presses X.

Ends the game by displaying the Game Over screen.

Simple method that increments and tracks a player's score.

```
            this.count++;
            Hud.el.score.innerHTML = this.count;
        }
    },
    el: {
        score: document.getElementById('count'),
        start: document.getElementById('start'),
        end: document.getElementById('end'),
        title: document.getElementById('title')
    }
};
```

Captures and stores alterable elements for easy reference.

9.3.2 Creating 2D shapes in 3D

With your HUD and controller built, you can program the player entity, a simple white triangle that can move when certain keyboard keys are pressed. You'll also make it generate bullets whenever a player presses the X key. Figure 9.9 shows the white, triangular player and a single red bullet.

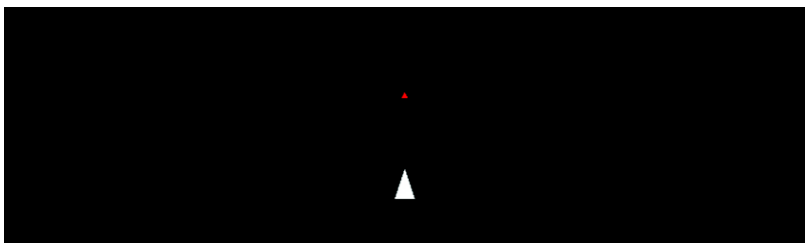

Figure 9.9 Displays the player's ship firing a bullet. Notice that both shapes are 2D but drawn in a 3D environment.

STEP 3: CREATE THE 2D PLAYER ENTITY

Core API

Append the next listing after your `Hud` object to create all of the data required to initialize your player. Most of the initializing information will be stored in variables at the top, so you can easily tweak the player's data in the future.

Listing 9.20 run.js—Player creation

```
gd.template.Player = gd.template.Entity.extend({
    type: 'a',
    x: -1.4,
    width: 1,
    height: 1,
    speed: 0.5,
    shoot: true,
    shootDelay: 400,
    rotate: {
        angle: 0,
        axis: [0, 0, 1],
        speed: 3
    },
```

Offsets player to line up nicely with text.

All width and height measurements are equal to one player unit.

A variable we'll use to decide how fast a player's position increments.

Can be a value from 0 to 360.

Allows you to only rotate the player in 2D.

```
    init: function() {
        this.shape([
            0.0,  2.0,  0.0,
           -1.0, -1.0,  0.0,
            1.0, -1.0,  0.0
        ]);
        this.color([
            1.0, 1.0, 1.0, 1.0,
            1.0, 1.0, 1.0, 1.0,
            1.0, 1.0, 1.0, 1.0
        ]);
    },
    boundaryTop: function() { this.y = gd.game.size.height; },
    boundaryRight: function() { this.x = gd.game.size.width; },
    boundaryBottom: function() { this.y = -gd.game.size.height; },
    boundaryLeft: function() { this.x = -gd.game.size.width; },
    kill: function() {
        this._super();
        PolygonGen.clear();
        Hud.end();
    }
});
```

Creates a triangle by plotting and connecting three different points from the passed array data. Each line of the array plots a point in the format of x, y, and z.

Outputs white for all three points you created with the shape method.

Creates a color for each point you created with the shape method. Each line of this array outputs a color as red, green, blue, alpha.

When the player is destroyed, the HUD and polygon generator (set up later) will be shut down.

3D DRAWING BASICS

Core API

The most confusing part of creating players is probably the `shape()` and `color()` methods. The `shape()` method assembles the triangle in figure 9.10, and the `color()` method fills it in with white.

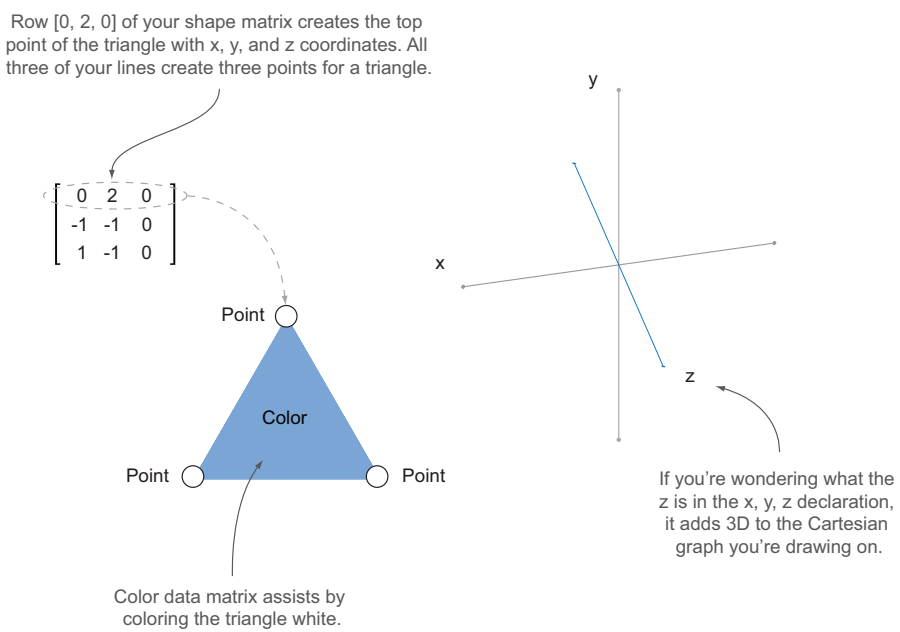

Figure 9.10 Diagram on the left shows a triangle comprising three points from the player's matrix data. The right diagram shows a Cartesian coordinate system with x, y, and z.

The single form of the word *vertices* is *vertex*. In math, a vertex of an angle is an endpoint where two line segments meet. Declaring three vertices, you created a triangle, as shown in the previous figure. Adding one more vertex to the triangle creates the square shown in figure 9.11, as you probably guessed.

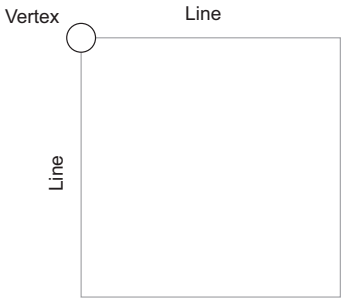

Figure 9.11 Demonstrates where a vertex is located on a square

STEP 4: ANIMATE THE PLAYER ENTITY

Getting back to your `Player` entity, you need to append an `update()` method with the following listing to complete it with movement, rotation, and shooting controls via the keyboard. You're already generating keyboard properties from the `Ctrl` object you integrated earlier.

Listing 9.21 run.js—Player update

```
gd.template.Player = gd.template.Entity.extend({
    update: function() {
        var self = this;

        if (Ctrl.left) {
            this.rotate.angle += this.rotate.speed;
        } else if (Ctrl.right) {
            this.rotate.angle -= this.rotate.speed;
        }

        if (Ctrl.up) {
            this.x -= Math.sin(this.rotate.angle * Math.PI / 180)
                * this.speed;
            this.y += Math.cos(this.rotate.angle * Math.PI / 180)
                * this.speed;
        } else if (Ctrl.down) {
            this.x += Math.sin(this.rotate.angle * Math.PI / 180)
                * this.speed;
            this.y -= Math.cos(this.rotate.angle * Math.PI / 180)
                 * this.speed;
        }

        gd.game.boundaries(this, this.boundaryTop, this.boundaryRight,
            this.boundaryBottom, this.boundaryLeft);

        if (Ctrl.x && this.shoot) {
            gd.game.spawn('Bullet', this.rotate.angle, this.x, this.y);

            this.shoot = false;
            window.setTimeout(function() {
                self.shoot = true;
            }, this.shootDelay);
        }
    }
});
```

Update logic fires every time a new frame is drawn.

When pushing left or right, rotation will be triggered for the player. Rotation is automatically applied by the cp.core.draw method you set up earlier.

Updates the player's position using the current angle.

Prevents the player from going out of the game's boundaries.

Generates a bullet from ship's current location and moves it at its current angle.

Figure 9.12 You should be able to move your player around the screen now. We've moved him from between "Geometry Destroyer" to the upper-left corner. Be warned: You can't shoot bullets with X yet.

PROGRESS CHECK!

At this point, you should be able to move your ship around the page without errors, as shown in figure 9.12. If you press X on the keyboard, though, your application will explode because bullets haven't been configured yet. Let's fix that.

STEP 5: CREATE THE PLAYER'S BULLETS

Create bullets to shoot by appending the following listing after your `Player` entity. Your player will shoot small triangles that destroy enemy entities on collision. `Bullets` will spawn at the `Player`'s position when you pass in parameters through the `init()` method.

Listing 9.22 run.js—Making bullets

```
gd.template.Bullet = gd.template.Entity.extend({
    type: 'a',
    width: 0.6,
    height: 0.6,
    speed: 0.8,
    angle: 0,

    init: function(angle, x, y) {
        this.shape([
            0.0,  0.3,  0.0,
           -0.3, -0.3,  0.3,
            0.3, -0.3,  0.3
        ]);

        var stack = [];
        for (var line = this.shapeRows; line--;)
            stack.push(1.0, 0.0, 0.0, 1.0);
        this.color(stack);

        this.angle = angle;
        this.x = x;
        this.y = y;
    },

    update: function() {
        gd.game.boundaries(this, this.kill, this.kill, this.kill, this.kill);
```

Angle is used to determine the movement direction (0 to 360 degrees).

Notice how init() allows the bullet to spawn at an x and y location and then move at the player's current angle.

Alternative method for creating a color matrix. Useful when creating a massive number of points that have the same color value.

```
            this.x -= Math.sin( this.angle * Math.PI / 180 ) * this.speed;
            this.y += Math.cos( this.angle * Math.PI / 180 ) * this.speed;
        },

        collide: function() {
            this._super();
            Hud.score.update();
        }
});
```

Armed with bullets, you should be able to run the game and fly your ship around. Try it out if you'd like. You'll notice that once you fire a bullet, the game fails because you haven't yet created the enemy assets. Let's create those targets next.

9.3.3 Creating 3D shapes and particles

Enemies in Geometry Destroyer are complex and robust because of their dynamic color and spawning points. As you can see in figure 9.13, they explode on contact, shattering into cubes and rectangle particles to create an interesting effect.

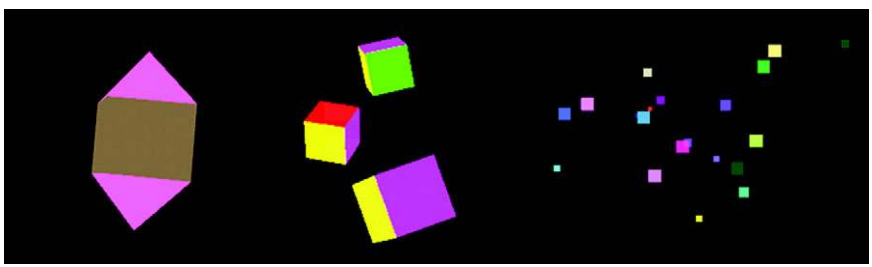

Figure 9.13 Enemies in the game have three major components. First is the large shape shown on the far left. When destroyed, it spawns the next two components: cubes (middle) and particles (far right).

Let's get started with the second group of tasks:

- Group 2—Outputting enemies
  - Step 1: Create a 3D polygon enemy.
  - Step 2: Create a complex 3D model.
  - Step 3: Generate random enemy properties.
  - Step 4: Resolve enemy collisions.
  - Step 5: Spawn enemies in a controlled manner.

STEP 1: CREATE A 3D POLYGON ENEMY

Set up the large `Polygon` first by adding it below `gd.template.Bullet` with the following listing. You're only going to create its base right now; you'll configure its 3D data in the next listing.

Listing 9.23 run.js—Polygon base

```
gd.template.Polygon = gd.template.Entity.extend({
    type: 'b',
```

```
        width: 7,
        height: 9,

        init: function() {
            this.randomSide();
            this.randomMeta();

            var stack = [];
            for (var v = 0; v < this.shapeRows * this.shapeColumns; v += 3) {
                if (v > 108 || v <= 36) {
                    stack.push(this.colorData.pyramid[0],
    this.colorData.pyramid[1], this.colorData.pyramid[2], 1);
                } else {
                    stack.push(this.colorData.cube[0], this.colorData.cube[1],
    this.colorData.cube[2], 1);
                }
            }
            this.color(stack);
        }
    });
```

Width is the measurement of the shape's span of vertices from left to right, whereas `height` is top to bottom.

Tests if a triangle is being drawn instead of a square.

Because you have an insane number of points that need to be colored, you'll have to dynamically create a map of colors instead of writing them by hand.

Step 2: Create a complex 3D model

Core API

You need to add a massive amount of vertex data to finish `gd.template.Polygon.init()` from the previous listing. It comprises a pyramid on the top and bottom, with a cube in the middle. You'll notice a massive array of data is needed to create the 3D model. We recommend copying and pasting this from the downloaded source code; if you don't have that option, we sincerely apologize. Prepend `this.shape()` call from the following listing to the top of `gd.template.Polygon.init()`'s existing code from the previous listing.

Listing 9.24 run.js—Polygon shape `init()` prepend

```
gd.template.Polygon = gd.template.Entity.extend({
    init: function() {
        this.shape([
             0.0,  7.0,  0.0,
            -4.0,  2.0,  4.0,
             4.0,  2.0,  4.0,

             0.0,  7.0,  0.0,
             4.0,  2.0,  4.0,
             4.0,  2.0, -4.0,

             0.0,  7.0,  0.0,
             4.0,  2.0, -4.0,
            -4.0,  2.0, -4.0,

             0.0,  7.0,  0.0,
            -4.0,  2.0, -4.0,
            -4.0,  2.0,  4.0,

            -4.0,  2.0,  4.0,
            -4.0, -5.0,  4.0,
            -4.0, -5.0, -4.0,
```

Top pyramid's front.

Top pyramid's right.

Top pyramid's back.

Top pyramid's left.

Each middle plate section comprises a side of the polygon's cubic body. The sections comprised two triangles drawn together, which creates a square plate.

```
                -4.0,  2.0,  4.0,
                -4.0,  2.0, -4.0,
                -4.0, -5.0, -4.0,

                -4.0,  2.0, -4.0,
                -4.0, -5.0, -4.0,
                 4.0, -5.0, -4.0,
                -4.0,  2.0, -4.0,
                 4.0,  2.0, -4.0,
                 4.0, -5.0, -4.0,

                 4.0,  2.0,  4.0,
                 4.0,  2.0, -4.0,
                 4.0, -5.0, -4.0,
                 4.0,  2.0,  4.0,
                 4.0, -5.0,  4.0,
                 4.0, -5.0, -4.0,

                -4.0,  2.0,  4.0,
                 4.0,  2.0,  4.0,
                 4.0, -5.0,  4.0,
                -4.0,  2.0,  4.0,
                -4.0, -5.0,  4.0,
                 4.0, -5.0,  4.0,

                 0.0, -10.0, 0.0,
                -4.0, -5.0,  4.0,
                 4.0, -5.0,  4.0,

                 0.0, -10.0, 0.0,
                 4.0, -5.0,  4.0,
                 4.0, -5.0, -4.0,

                 0.0, -10.0, 0.0,
                 4.0, -5.0, -4.0,
                -4.0, -5.0, -4.0,

                 0.0, -10.0, 0.0,
                -4.0, -5.0, -4.0,
                -4.0, -5.0,  4.0
            ]);
        }
    };
```

Each middle plate section comprises a side of the polygon's cubic body. The sections comprised two triangles drawn together, which creates a square plate.

Bottom pyramid parallels the drawing format of the top pyramid, except it's drawn pointing down instead of up.

STEP 3: GENERATE RANDOM ENEMY PROPERTIES

With your polygon's 3D data built, you need to generate speed, rotation, color, and a spawning point so it functions properly. Append `randomMeta()` and `cube()` methods to `gd.template.Polygon` with the next listing.

Listing 9.25 run.js—Polygon shape `init()` prepend

```
gd.template.Polygon = gd.template.Entity.extend({
    randomMeta: function() {
        this.rotate = {
            speed: gd.game.random.number(400, 100),
            axis: [
                gd.game.random.number(10, 1) / 10,
```

Responsible for creating random details about rotation, speed, and color.

```
                gd.game.random.number(10, 1) / 10,
                gd.game.random.number(10, 1) / 10
            ],
            angle: gd.game.random.number(250, 1)
        };

        this.speed = {
            x: gd.game.random.number(10, 4) / 100,
            y: gd.game.random.number(10, 4) / 100
        };

        this.colorData = {
            pyramid: [
                gd.game.random.number(10, 1) / 10,
                gd.game.random.number(10, 1) / 10,
                gd.game.random.number(10, 1) / 10
            ],
            cube: [
                gd.game.random.number(10, 1) / 10,
                gd.game.random.number(10, 1) / 10,
                gd.game.random.number(10, 1) / 10
            ]
        };
    }
});
```

Generates random color details for pyramids and cubes. Data is processed and arranged by methods in Polygon.init() you already created.

STEP 4: RESOLVE ENEMY COLLISIONS

The last step to create the `gd.template.Polygon` requires you to add methods for generating shape data from a random side and cube particles when it's destroyed. You also need to update logic and collision information. Append your remaining methods to `gd.template.Polygon` with the following listing.

Listing 9.26 run.js—Polygon side, update, and collide

```
gd.template.Polygon = gd.template.Entity.extend({
    randomSide: function() {
        var side = gd.game.random.number(4, 1);

        if (side === 1) {
            this.angle = gd.game.random.number(200, 160);
            var range = gd.game.size.width - this.width;
            this.x = gd.game.random.number(range, -range);
            this.y = gd.game.size.height + this.height;
        } else if (side === 2) {
            this.angle = gd.game.random.number(290, 250);
            var range = gd.game.size.height - this.height;
            this.x = (gd.game.size.width + this.width) * -1;
            this.y = gd.game.random.number(range, -range);
        } else if (side === 3) {
            this.angle = gd.game.random.number(380, 340);
            var range = gd.game.size.width - this.width;
            this.x = gd.game.random.number(range, -range);
            this.y = (this.height + gd.game.size.height) * -1;
        } else {
            this.angle = gd.game.random.number(110, 70);
```

Determines from which side to randomly spawn a polygon.

```
            var range = gd.game.size.height - this.height;
            this.x = gd.game.size.width + this.width;
            this.y = gd.game.random.number(range, -range);
        }
    },

    update: function() {
        gd.game.boundaries(this, this.kill, this.kill, this.kill, this.kill,
            (this.width * 2));

        this.x -= Math.sin( this.angle * Math.PI / 180 ) * this.speed.x;
        this.y += Math.cos( this.angle * Math.PI / 180 ) * this.speed.y;

        gd.game.rotate(this);
    },

    collide: function() {
        if (gd.core.storage.all.length < 50) {
            for (var p = 15; p--;) {
                gd.game.spawn('Particle', this.x, this.y);
            }
        }

        var num = gd.game.random.number(2, 4);
        for (var c = num; c--;) {
            gd.game.spawn('Cube', this.x, this.y);
        }

        this.kill();
    }
});
```

Uses randomly generated rotate data to make the polygon slowly rotate.

Creates a number of particles at the center of a polygon upon destruction. Only occurs if the storage isn't too full to prevent hogging memory.

Generates a random number of cubes at the center of a polygon upon destruction.

STEP 5: SPAWN ENEMIES IN A CONTROLLED MANNER

Although you now have a class for polygon entities, you'll need a separate object to generate them. You can create this with a new object called `PolygonGen` right below `gd.template.Polygon` with the next listing.

Listing 9.27 run.js—Polygon generator

```
var PolygonGen = {
    delay: 7000,
    limit: 9,

    init: function() {

        var self = this;

        this.count = 1;
        gd.game.spawn('Polygon');

        this.create = window.setInterval(function() {
            if (gd.core.storage.b.length < self.limit) {
                if (self.count < 3)
                    self.count++;

                for (var c = self.count; c--;) {
                    gd.game.spawn('Polygon');
                }
            }
```

Initiates polygon generation by creating an interval.

Failsafe to prevent too many objects spawning and potentially crashing the browser.

```
        }, self.delay);
    },
    clear: function() {
        window.clearInterval(this.create);
        this.count = 0;
        this.delay = 7000;
    }
};
```

Shuts down polygon generation.

Polygons will now generate after you press X on a keyboard for the first time. If you shoot them, they'll fire an error because the game tries to use nonexistent entity templates for cubes and particles. You'll set up those with the next set of tasks:

- Group 3—Generating particles
  - Step 1: Create a 3D cube particle.
  - Step 2: Add color, rotation, and index data for cubes.
  - Step 3: Add size, type, and other cube metadata.
  - Step 4: Generate square particles.

Issues with requestAnimationFrame() and other timers

Your method in `gd.core.animate()` that fires `requestAnimationFrame()` stops running when a user leaves a tab open in the background, unlike JavaScript's traditional timers `setInterval()` and `setTimeout()`, which keep on running. This means coupling animation with traditional timers is generally not a good idea, because traditional timers keep on running in the background. There used to be polyfills that relied on a frame counter in the `draw()` loop, but some implementations of `requestAnimationFrame()` still update a frame after a couple seconds when a user navigates away from a tab. The most bulletproof way to use traditional and nontraditional timers is to build a custom timer script that checks elapsed time and fires in your `draw` loop. But this subject is complicated, and we don't have the time to cover it here. Instead, we've given the `polygonGen` object a limit to how many enemies it can spawn for a quick patch.

STEP 1: CREATE A 3D CUBE PARTICLE

Core API

Create a new `gd.template.Cube` entity below `PolygonGen` with this listing.

Listing 9.28 run.js—Cube shape

```
gd.template.Cube = gd.template.Entity.extend({
    init: function(x, y) {
        this.x = x;
        this.y = y;

        this.meta();

        this.shape([
            -this.s, -this.s,  this.s,
             this.s, -this.s,  this.s,
             this.s,  this.s,  this.s,
            -this.s,  this.s,  this.s,
```

Sets position for x and y with the parameters passed at spawn.

Our shape declaration is using a much more efficient method than our polygon to create rectangles by using four points instead of six. The catch is we need to provide a set of indices.

Front plate; this.s is a reference to a random size generated later in gd.template.Cube.meta().

```
            -this.s, -this.s, -this.s,
            -this.s,  this.s, -this.s,
             this.s,  this.s, -this.s,
             this.s, -this.s, -this.s,

            -this.s,  this.s, -this.s,
            -this.s,  this.s,  this.s,
             this.s,  this.s,  this.s,
             this.s,  this.s, -this.s,

            -this.s, -this.s, -this.s,
             this.s, -this.s, -this.s,
             this.s, -this.s,  this.s,
            -this.s, -this.s,  this.s,

             this.s, -this.s, -this.s,
             this.s,  this.s, -this.s,
             this.s,  this.s,  this.s,
             this.s, -this.s,  this.s,

            -this.s, -this.s, -this.s,
            -this.s, -this.s,  this.s,
            -this.s,  this.s,  this.s,
            -this.s,  this.s, -this.s
        ]);
    }
});
```

Back plate.

Top plate.

Bottom plate.

Right plate.

Left plate.

STEP 2: ADD COLOR, ROTATION, AND INDEX DATA FOR CUBES

You now need to append the `gd.template.Cube.init()` method with color, rotation, and indices data from the next listing. If you're wondering what indices are, they allow you to draw the sides of a square with four points. Normally, a square's side requires six points to create two triangles—this cuts down on code and makes it easier to maintain.

Listing 9.29 run.js—Cube indices and color

```
gd.template.Cube = gd.template.Entity.extend({
    init: function(x, y) {
        this.indices([
             0,  1,  2,    0,  2,  3,
             4,  5,  6,    4,  6,  7,
             8,  9, 10,    8, 10, 11,
            12, 13, 14,   12, 14, 15,
            16, 17, 18,   16, 18, 19,
            20, 21, 22,   20, 22, 23
        ]);

        this.color([
            [1, 0, 0, 1],
            [0, 1, 0, 1],
            [0, 0, 1, 1],
            [1, 1, 0, 1],
            [1, 0, 1, 1],
            [0, 1, 1, 1]
        ]);
```

Each row of indices assembles the shape coordinates of two triangles into a plate. Each number here represents an index to an indice, not x, y, z coordinates.

We're passing an array of indices for the colors; you previously set up the color method in your template.js file to output large amounts of color data for indices.

```
        if (this.rotate)
            this.rotate = {
                axis: [
                    gd.game.random.number(10, 1) / 10,
                    gd.game.random.number(10, 1) / 10,
                    gd.game.random.number(10, 1) / 10],
                angle: gd.game.random.number(350, 1),
                speed: gd.game.random.number(400, 200)
            };
    }
});
```

STEP 3: ADD SIZE, TYPE, AND OTHER CUBE METADATA

Before `gd.template.Cube` is complete, you need to add metadata, such as size, type, and other details. Append the following listing to your existing `Cube` object.

Listing 9.30 run.js—Cube metadata

```
gd.template.Cube = gd.template.Entity.extend({
    type: 'b',
    size: {
        max: 3,
        min: 2,
        divider: 1
    },
    pressure: 50,

    meta: function() {
        this.speed = {
            x: (gd.game.random.number(this.pressure, 1) / 100)
                * gd.game.random.polarity(),
            y: (gd.game.random.number(this.pressure, 1) / 100)
                * gd.game.random.polarity()
        };

        this.angle = gd.game.random.number(360, 1);

        this.s = gd.game.random.number(this.size.max, this.size.min)
            / this.size.divider;
        this.width = this.s * 2;
        this.height = this.s * 2;
    },

    update: function() {
        gd.game.boundaries(this, this.kill, this.kill, this.kill,
            this.kill, this.width);

        this.x -= Math.sin( this.angle * Math.PI / 180 ) * this.speed.x;
        this.y += Math.cos( this.angle * Math.PI / 180 ) * this.speed.y;

        if (this.rotate)
            gd.game.rotate(this);
    }
});
```

You'll use a size object and the meta method to randomly generate a cube's size. This makes size changes easy for when you extend this entity for particles later.

Pressure will be used to generate how much speed a cube has after exploding out of a polygon.

STEP 4: GENERATE SQUARE PARTICLES

Core API

Finish your game by adding `gd.template.Particle` right after `gd.template.Cube` with the following listing. For awesome special effects, you can turn up the number of particles and turn off the particle limiter in `Polygon.collide()`. Keep in mind that generating lots of particles can cause memory issues and frame-rate drops.

Listing 9.31 run.js—Particle generation

```
gd.template.Particle = gd.template.Cube.extend({
    pressure: 20,
    type: 0,
    size: {
        min: 2,
        max: 6,
        divider: 10
    },

    init: function(x, y) {
        this.x = x;
        this.y = y;

        this.meta();

        this.shape([
             this.s,  this.s,  0.0,
            -this.s,  this.s,  0.0,
             this.s, -this.s,  0.0,
            -this.s, -this.s,  0.0
        ]);

        var r = gd.game.random.number(10, 0) / 10,
        g = gd.game.random.number(10, 0) / 10,
        b = gd.game.random.number(10, 0) / 10;
        this.color([
            r, g, b, 1,
            r, g, b, 1,
            r, g, b, 1,
            r, g, b, 1
        ]);

        var self = this;
        this.create = window.setTimeout(function() {
            self.kill();
        }, 5000);
    }
});
```

Extends the cube logic instead of writing a new particle entity from scratch.

Creates a flat rectangle shape with four points.

Randomly generates a red, green, blue color with a constant alpha level.

Cleans the particle out of memory after five seconds to prevent memory hogging.

Boot up the completed application in your browser, and everything should work correctly. You did it! You created a real 3D game—a basic WebGL engine—and learned foundational 3D programming concepts at the same time. With these tools, you can start using WebGL in your JavaScript projects immediately to create logos, illustrations, and more—especially with robust 3D libraries like three.js.

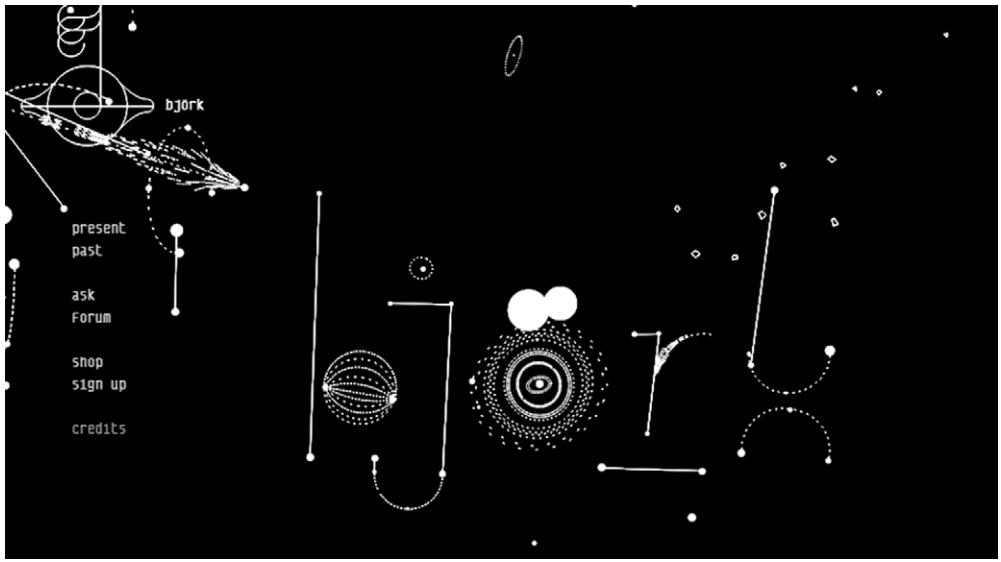

Figure 9.14 Almost every illustration on the bjork.com home page is drawn in a 2D fashion. When they're moved, you can tell that all the illustrations are 3D.

9.4 Summary

The words *3D application* evoke thoughts of video games and animation that illuminate the mind's eye. Even though you can use WebGL for entertainment purposes, this function makes up a small percentage of what you can do. Some authors have created 3D simulations for various scenarios, such as walking through architecture and operating vehicles. Uses for 3D in-browser can also transcend Canvas's 2D space limitations. For instance, Bjork's website (bjork.com) uses 2D shapes in a 3D environment for an amazing effect (shown in figure 9.14).

Various websites and companies are investing big money in WebGL. It's too powerful to ignore, and as support improves, it will drastically change how websites and mobile devices are programmed, mostly because WebGL will eventually give mobile developers the ability to write one 3D application with graphics acceleration for multiple devices. Therefore, we think it's important for developers to learn more about it now by playing with demos and tutorials.

You'll also be glad to know that WebGL isn't the only API that's evolving the Net; we'll talk about several others, such as the Full-Screen, Orientation, and Pointer Lock APIs in appendix I.