

Appendix A

HTML5 and JavaScript Toolkits

This Appendix provides a light introduction and code samples for graphics-oriented JavaScript toolkits (such as `EaselJS` and `ThreeJS`) as well as JavaScript toolkits for developing HTML5 Web pages (such as `Backbone.js` and `Twitter Bootstrap`). The rationale for the inclusion of these technologies is simple: Web development is becoming increasingly sophisticated, and there is a distinct possibility that at least some of the technologies (or some variants) in this Appendix will become ubiquitous.

If you are a Web developer, portions of the material in this Appendix might be optional for you, so it's a good idea to skim through the list of topics (and graphics images) in this Appendix, and then decide which material is relevant for your needs. Knowledge of these toolkits will place you in a better position to compare emerging toolkits to those that are currently available and also assess them accordingly.

The first part of this Appendix briefly discusses `EaselJS` and `KineticJS`, which are JavaScript toolkits for HTML5 Canvas. These toolkits provides a layer of abstraction on top of HTML5, and you can easily create Android applications using these toolkits.

The second part of this Appendix discusses `D3`, which is a JavaScript-based toolkit (developed by Mike Bostock) that enables you to create SVG-based graphics quickly, and often with considerably less effort than doing so using plain JavaScript. This toolkit

bears some resemblance to jQuery, and its name is based on Document Driven Data.

This section presents D3 code samples, as well as a link to an open source project containing code samples using D3, and also how to combine D3 with CSS3 and SVG filters. You will also get a brief introduction to Google Go, along with a code sample written in Go that illustrates how to generate SVG-based graphics.

The third part of this Appendix discusses WebGL, which is not a formal part of HTML5, yet it is an upcoming technology for sophisticated graphics, animation, and games in browsers. If you are using Internet Explorer, this section provides information about obtaining a shim that adds WebGL support to IE. You will see some code examples that illustrate how to use the `Three.js`, which is a toolkit that provides a layer of abstraction on top of WebGL. This section also discusses CSG (Constructive Solid Geometry), which is another 3D toolkit that you can use in conjunction with `Three.js`.

The final section gives you a brief overview of Backbone.js (a JavaScript-based toolkit for developing Websites and applications) and Twitter Bootstrap (a very popular toolkit for developing Websites and applications).

There are a few points that you need to keep in mind as you read this Appendix. First, these toolkits have differing levels of complexity, and only a limited number of features are presented in this Appendix. However, the intent is to provide you with just enough details to pique your interest, and also to help you determine which toolkits (if any) might be relevant for your Web development. Although the toolkits in this Appendix serve different needs (e.g., D3 is extremely well-suited for data visualization), they usually require knowledge of JavaScript. Every technology in this Appendix works on major

browsers on desktops and (with the exception of WebGL) also on smart phones and tablets. In addition, IDEs such as Sencha Animator and Adobe Edge Animate (neither is covered here) can simplify the creation of Web pages with CSS3 with a minimum of custom code.

Second, this Appendix is *not* an introduction to game programming for desktop or mobile devices. There are entire books devoted to game programming, and if this topic interests you, then you can avail yourself of one of those books. The third point pertains to the code samples: virtually every sample renders an abstract image or pattern, sometimes accompanied by an open source project with additional code samples. These samples are included because you are unlikely to find them on any other Websites, and they can provide visual ideas that you can use to supplement your own Web pages.

JavaScript Toolkits for HTML5 Canvas

This Appendix covers the following JavaScript toolkits (the last two are covered in a cursory fashion) for HTML5 Canvas:

- EaselJS
- GWT Support for Canvas
- KineticJS
- FabricJS
- PaperJS

In addition, *ThreeJS* (discussed later in this Appendix) supports an HTML5 Canvas renderer.

EaselJS

EaselJS is an open source toolkit that provides a layer of abstraction over HTML5 Canvas, and its homepage is here:

<http://createjs.com>

EaselJS defines a hierarchy of shape objects “on top of” the bitmap-oriented APIs in HTML Canvas.

One programming style that is used in EaselJS in order to render HTML Canvas-based graphics involves the following sequence of six steps:

- obtain a reference to the HTML `<canvas>` element in a Web page
- create a “stage” object with the previously obtained reference
- create a new “container” object
- append the “container” object to the “stage” object
- append one or more Shape objects to the “container” object
- invoke the “update” method of the “stage” object

The last step in the previous list of six steps is the point at which the graphics is rendered in an HTML Web page.

As an example, the HTML page `LituusTransformDoubleEllipses1.html` in Listing A.1 uses `Easel.js` to render an abstract design based on a set of ellipses with gradients.

Listing A.1 LituusTransformDoubleEllipses1.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>EaselJS Lituus Ellipses</title>

<script src="easeljs-0.4.2.min.js"></script>
```

```

<script>
    var basePointX = 400, basePointY = 10;
    var currentX = 0, currentY = 0;
    var majorAxis = 80, minorAxis = 120;
    var deltaAngle = 3, maxAngle = 720;
    var Constant = 200, fillColor = "";
    var fillColors = ["#f00", "#ff0"];
    var stage, canvas, container, ellipse;

    function init() {
        canvas = document.getElementById("myCanvas");
        stage = new Stage(canvas);

        container = new Container();
        stage.addChild(container);

        displayGraphics();
    }

    function displayGraphics() {
        for(var angle=1; angle<maxAngle; angle++) {
            radius = Constant*Constant/angle;

            offsetX = radius*Math.cos(angle*Math.PI/180);
            offsetY = radius*Math.sin(angle*Math.PI/180);
            currentX = basePointX+offsetX;
            currentY = basePointY-offsetY;

            fillColor = fillColors[angle%fillColors.length];

            // create an ellipse at the current position
            ellipse = new Shape();
            ellipse.rotation = angle/20;

            ellipse.graphics.beginFill(fillColor).endStroke()
                .drawEllipse(currentX, currentY,
                    majorAxis, minorAxis, 8);
            container.addChild(ellipse);
        }
    }

```

```

        // create an adjacent ellipse
        ellipse = new Shape();
        ellipse.rotation = angle/10;

        ellipse.graphics.beginFill(fillColor).endStroke()
            .drawEllipse(currentX+majorAxis, currentY,
                majorAxis, minorAxis, 8);
        container.addChild(ellipse);
    }

    stage.update();
}
</script>
</head>

<body onload="init();">
    <div class="container">
        <canvas id="myCanvas" width="800" height="500"></canvas>
    </div>
</body>
</html>

```

The code in Listing A.1 follows the sequence of six steps that is listed at the beginning of this section. Notice that Step 5 is performed by a loop that adds a set of Shape objects that are ellipses, as shown here:

```

// create an ellipse at the current position
ellipse = new Shape();
ellipse.rotation = angle/20;

ellipse.graphics.beginFill(fillColor).endStroke()
    .drawEllipse(currentX, currentY,
        majorAxis, minorAxis, 8);
container.addChild(ellipse);

```

Listing A.1 creates another ellipse that is very similar to the preceding ellipse, except that its rotation attribute is set to `angle/10` instead of `angle/20`, and then this new ellipse is also appended to the `container` object.

Figure A.1 displays the graphics image that is rendered by the code in Listing A.1.

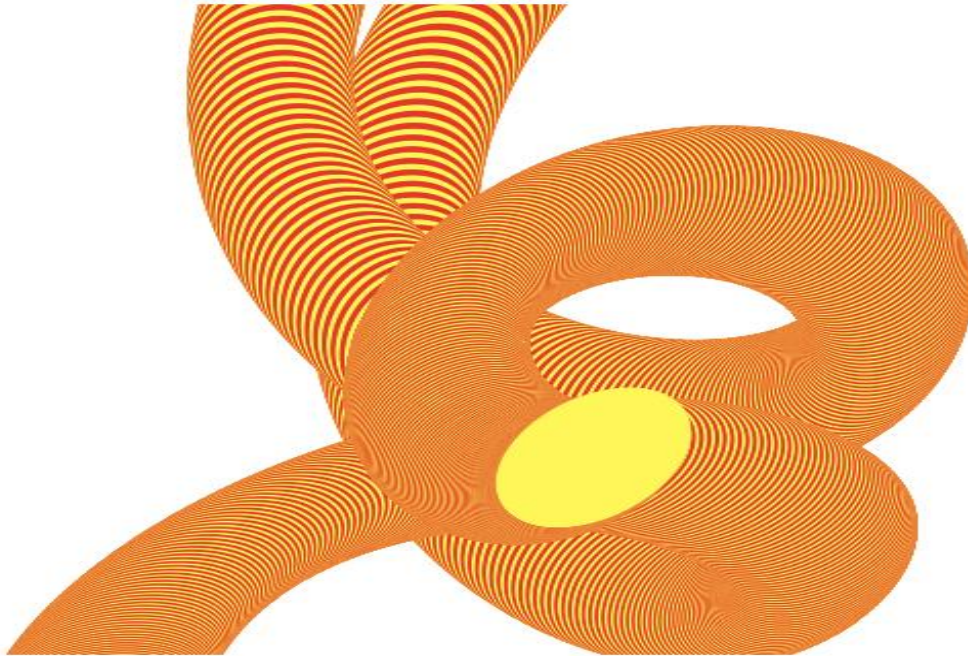


Figure A.1 Rendering a Set of Ellipses with Easel.js.

Despite the simplicity of the code in Listing A.1, the graphics image has a nice visual effect, and additional graphics-based code samples illustrating other features of `Easel.js` are here:

<http://code.google.com/p/easeljs-graphics>

Kinetic.js

KineticJS is another open source toolkit that provides a hierarchy of shape object above the bitmap-oriented APIs in HTML5 Canvas (just as EaselJS does) and its homepage is here:

<https://github.com/kinetic.js/>

KineticJS supports event bindings whose syntax is similar to jQuery. For example, the following code fragment shows you how to bind a JavaScript function to mouse event on an element called `container` (which is not defined here):

```
container.on("mousedown"), function() {
    // do something here
});
```

Listing A.2 displays the contents of `KineticCanvas1.html`, which illustrates how to handle touch events in `Kinetic.js`.

Listing A.2 KineticCanvas1.html

```
<!DOCTYPE html>

<html>

  <head>

    <meta name="viewport"
      content="width=device-width, initial-scale=0.552, user-
scalable=no"/>

    <script src="kinetic-v3.5.2.js">

  </script>

  <script>

    function writeMessage(messageLayer, message) {
      var context = messageLayer.getContext();
      messageLayer.clear();
      context.font = "18pt Calibri";
      context.fillStyle = "black";
      context.fillText(message, 10, 25);
    }
  }
</script>

</html>
```



```

window.onload = function(){
    var stage = new Kinetic.Stage("container", 578, 200);
    var shapesLayer = new Kinetic.Layer();
    var messageLayer = new Kinetic.Layer();

    var triangle = new Kinetic.Shape(function(){
        var context = this.getContext();
        context.beginPath();
        context.lineWidth = 4;
        context.strokeStyle = "black";
        context.fillStyle = "#00D2FF";
        context.moveTo(120, 50);
        context.lineTo(250, 80);
        context.lineTo(150, 170);
        context.closePath();
        context.fill();
        context.stroke();
    });

    triangle.on("touchmove", function(){
        var touchPos = stage.getTouchPosition();
        var x = touchPos.x - 120;
        var y = touchPos.y - 50;
        writeMessage(messageLayer, "x: " + x + ", y: " + y);
    });

    shapesLayer.add(triangle);

    var circle = new Kinetic.Shape(function(){
        var canvas = this.getCanvas();
        var context = this.getContext();
        context.beginPath();
        context.arc(380, canvas.height / 2, 70, 0,
            Math.PI * 2, true);
        context.fillStyle = "red";
        context.fill();
        context.lineWidth = 4;
        context.stroke();
    });

```

```

    });

    circle.on("touchstart", function(){
        writeMessage(messageLayer, "Touchstart circle");
    });
    circle.on("touchend", function(){
        writeMessage(messageLayer, "Touchend circle");
    });

    shapesLayer.add(circle);

    stage.add(shapesLayer);
    stage.add(messageLayer);
};
</script>
</head>

<body>
    <div id="container"></div>
</body>
</html>

```

In high-level terms, Listing A.2 contains a JavaScript variable `stage` that serves as an outer container object for the `shapesLayer` object (which contains dynamically created shapes later in the code listing) and also the `messageLayer` object, which “styles” the rendered text.

When the HTML Web page in Listing A.2 is loaded, a JavaScript function is executed that creates a triangle shape, adds a `touchmove` handler to the triangle, and then adds the triangle to the `shapesLayer` object, as shown here:

```

var triangle = new Kinetic.Shape(function(){. . . });
triangle.on("touchmove", function(){. . . });
shapesLayer.add(triangle);

```

The next block of code creates a circle object, adds a `touchstart` and a `touchend` handler to the circle, and then adds the circle to the `shapesLayer` object. The last bit of

code adds the `shapeLayer` object and the `messageLayer` object to the `stage` object.

Figure A.2 displays the graphics image that is rendered by Listing A.2.

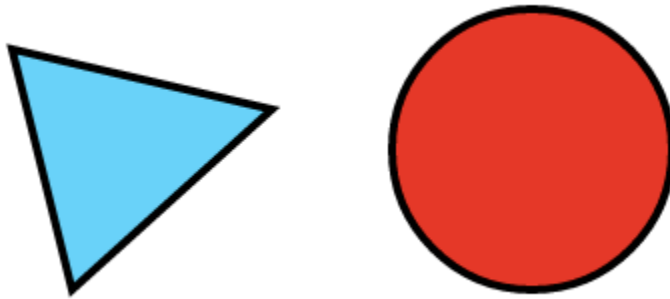


Figure A.2 Rendering 2D Shapes with KineticJS.

Creating Android Applications with KineticJS

Create an Android project called `KineticDnd1` for Android 2.x or higher (with a namespace of your choice), and then perform the following three steps:

- create the directory `$PROJECT_TOP/assets/www`
- place a copy of the file `KineticDnd1.html` in the preceding directory
- place a copy of `kinetic-3.7.4.js` in the same subdirectory

Listing A.3 displays the contents of `KineticDnd1.html`, which illustrates how to handle drag-and-drop events in an Android application that uses `Kinetic.js`.

Listing A.3 KineticDnD1.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta name="viewport"
content="width=device-width, initial-scale=0.552, user-scalable=no"/>
```

```
<script src="kinetic-v3.7.4.js"></script>
```

```
<script>
```

```
function writeMessage(stage, message){
    var context = stage.getContext();
    stage.clear();
    context.font = "18pt Calibri";
    context.fillStyle = "black";
    context.fillText(message, 10, 25);
}
```

```
window.onload = function(){
    var stage = new Kinetic.Stage("container", 578, 200);
    var canvas = stage.getCanvas();
    var rectX = canvas.width / 2 - 50;
    var rectY = canvas.height / 2 - 25;

    var box = new Kinetic.Shape(function(){
        var context = this.getContext();
        context.beginPath();
        context.rect(rectX, rectY, 100, 50);
        context.lineWidth = 4;
        context.strokeStyle = "black";
        context.fillStyle = "#00D2FF";
        context.fill();
        context.stroke();
        context.closePath();
    });

    // enable drag and drop
    box.draggable(true);

    // write out drag and drop events
    box.on("dragstart", function(){
        writeMessage(stage, "dragstart");
    });
    box.on("dragend", function(){
        writeMessage(stage, "dragend");
    });
};
```

```

        stage.add(box);
    };
</script>
</head>

<body onmousedown="return false;">
    <div id="container">
        </div>
</body>
</html>

```

The code in Listing A.3 has a similar structure, except that instead of creating a circle and a triangle, Listing A.3 creates a rectangular object called `box`, makes the `box` object draggable, adds `dragstart` and `dragend` event listeners, and then adds the `box` object to the `stage` object, as shown here:

```

var box = new Kinetic.Shape(function(){. . .});
box.draggable(true);
box.on("dragstart", function(){
    writeMessage(stage, "dragstart");
});
box.on("dragend", function(){
    writeMessage(stage, "dragend");
});
stage.add(box);

```

Listing A.4 displays the contents of `KineticDnd1Activity.html`, which is the Android Activity for the `KineticDnD1` Android project.

Listing A.4 `KineticDnD1Activity.java`

```

package com.iquarkt.kinetic.canvas;

import android.app.Activity;
import android.os.Bundle;
import android.view.KeyEvent;
import android.webkit.WebView;

```

```

import android.webkit.WebViewClient;

public class KineticCanvas1Dnd1Activity extends Activity
{
    WebView mWebView;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        mWebView = (WebView) findViewById(R.id.webview);
        mWebView.setWebViewClient(new ChangeURLClient());
        mWebView.getSettings().setJavaScriptEnabled(true);
        mWebView.getSettings().setDomStorageEnabled(true);
        mWebView.loadUrl(
            "file:///android_asset/www/KineticCanvas1Dnd1.html");
    }

    public boolean onKeyDown(int keyCode, KeyEvent event) {
        if ((keyCode == KeyEvent.KEYCODE_BACK) && mWebView.canGoBack())
        {
            mWebView.goBack();
            return true;
        }
        return super.onKeyDown(keyCode, event);
    }

    private class ChangeURLClient extends WebViewClient {
        public boolean shouldOverrideUrlLoading(WebView view,
                                                String url)
        {
            System.out.println("Destination URL: " + url);
            view.loadUrl("javascript:changeLocation('" + url + "')");
            return true;
        }
        return super.onKeyDown(keyCode, event);
    }
}

```

In this example, the Android application launches the HTML5 Web page in Listing A.3 in an Android `WebView`.

Other JavaScript Toolkits for HTML5 Canvas

There are various other JavaScript toolkits that look promising, and after evaluating their supported features, you might decide that they are better suited for your needs than some of the toolkits that you learned about earlier in this Appendix.

One JavaScript toolkit for HTML5 Canvas is `Fabric.js`, whose homepage is here:

<http://fabricjs.com>

`FabricJS` has an “interactive object model” that is a layer above the HTML5 Canvas element, and it is also a parser for SVG-to-Canvas as well as Canvas-to-SVG. With `Fabric.js` you can create objects (such as rectangles, circles, ellipses, polygons, and so forth), and also apply transforms (such as scale, rotate, translate, and skew) to these objects. Moreover, you can apply filters to images, and also serialize the entire canvas.

Some `FabricJS` code samples are available here:

<https://github.com/kangax/fabric.js/>

`PaperJS` is another JavaScript toolkit for HTML5 Canvas, and its homepage is here:

<http://paperjs.org>

`Paper.js` provides a Document Object Model (also called a Scene Graph) populate it with layers, groups, paths, rasters, and so forth. Groups and layers can contain other items as well as other groups. A collection of `PaperJS` code samples is here:

<http://paperjs.org/examples/>

You might also be interested in reading a slightly older review of various Canvas-based toolkits (including jCanvasScript, CAKE, and doodle.js) is here:

<http://www.suburban-glory.com/blog?page=141>

JavaScript Toolkits for SVG

In addition to JavaScript toolkits for HTML5 Canvas, there are some very good JavaScript toolkits available for SVG, and this Appendix covers the D3 and Raphael, both of which are JavaScript toolkits that provide a layer of abstraction over SVG.

D3 (Document Driven Data)

Mike Bostock created the open source toolkit `Protovis`, and then he created the D3 toolkit, which is a JavaScript-based open source project for creating very appealing data visualization, and its homepage is here:

<http://mbostock.github.com/d3/>

In December of 2011, D3 was named the data visualization project of the year (by `Flowing Data!`), which is not surprising when you see the functionality that is available in D3.

D3 provides a layer of abstraction that generates underlying SVG code. D3 enables you to create a surprisingly rich variety of data visualizations. If you need to generate graphics-oriented Web pages, and you prefer to work with JavaScript instead of working with “raw” SVG, then you definitely ought to consider using D3.

Two key aspects of D3 involve tools for reading data in multiple formats and also the ability to transform the data and render the data in many forms.

D3 supports the following features:

- creation of SVG-based 2D shapes
- 2D graphics and animation effects
- method chaining

D3 has an extensive collection of “helper methods”, such as `select()`, `append()`, `data()`, and `attr()`, along with many other helper methods. Read the online documentation about these and other D3 methods.

The next section shows you how to use D3 in order to render a set of ellipses following the path of a polar equation.

Rendering a Cardioid-based Set of Gradient Ellipses with D3

Listing A.5 displays the contents of `CardioidEllipses1Grad2.html`, which is an HTML5 Web page that illustrates how to render a set of Cardioid-based gradient ellipses with D3.

Listing A.5 CardioidEllipses1Grad2.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cardioid Ellipses</title>
    <meta charset="utf-8" />
    <script src="d3.js"></script>
  </head>

  <body>
    <script type="text/javascript">
```

```

var basePointX    = 200, basePointY    = 250,
    minorAxis     = 120, majorAxis     = 80,
    Radius        = 120, petalCount    = 1,
    maxAngle      = 360, stripCount    = 20,
    stripWidth    = Math.floor(maxAngle/stripCount),
    rVal          = 0, gVal           = 0,
    bVal          = 0;

var w = 800, h = 500, p = 30,
    eColors = ['#f00', '#0f0', '#00f'];

var data = d3.range(maxAngle).map(function(a) {
    return {angle: a,
        r:      Radius*(1.0+Math.cos(petalCount*a*Math.PI/180)),
        x:      Math.cos(a*Math.PI/180),
        y:      Math.sin(a*Math.PI/180)}
});

var vis = d3.select("body")
    .append("svg:svg")
    .data([data])
    .attr("width", w + p * 2)
    .attr("height", h + p * 2)
    .append("svg:g")
    .attr("transform", "translate(" + p + "," + p + ")");

vis.selectAll("text")
    .data(data)
    .enter().append("svg:ellipse")
    .attr("fill", function(d) {
        rVal = Math.floor(
            (d.angle%stripWidth)*255/stripWidth);
        return "rgb("+rVal+","+gVal+","+bVal+")"; })
    .attr("cx", function(d) { return basePointX+d.r*d.x; })
    .attr("cy", function(d) { return basePointY+d.r*d.y; })
    .attr("rx", function(d) { return majorAxis; })
    .attr("ry", function(d) { return minorAxis; });
</script>
</body>

```

</html>

Listing A.5 contains HTML markup and also initializes some JavaScript variables. Next, the definition of the `data` variable is a collection of objects, each of which provides the formulas for calculating the values of the variables `r`, `x`, and `y` for a given value of `angle` (which varies between 0 and `maxAngle`), as shown here:

```
var data = d3.range(maxAngle).map(function(a) {
    return {angle: a,
            r:      Radius*(1.0+Math.cos(petalCount*a*Math.PI/180)),
            x:      Math.cos(a*Math.PI/180),
            y:      Math.sin(a*Math.PI/180)}
    });
```

Notice that `maxAngle` (shown in bold) is specified as the value for the `D3 range()` function in the first line of the preceding code block.

The `vis` variable creates an `<svg>` element and uses method chaining to set various attributes of this `<svg>` element. The final block of code in Listing A.6 creates a set of SVG `<ellipse>` elements (and also sets the attributes of each ellipse) and appends every `<ellipse>` element to the `<svg>` element via an implicit loop.

For example, the calculation the `cx` attribute and the `cy` attribute of each ellipse is shown here:

```
.attr("cx",    function(d) { return basePointX+d.r*d.x; })
.attr("cy",    function(d) { return basePointY+d.r*d.y; })
```

The `d` object that is passed into the anonymous function in the preceding two lines of code is one of the elements of the `data` object defined earlier in Listing A.5, which means that you can reference the values `d.r`, `d.x`, and `d.y`. The attributes `rx` and `ry`

are constants, and the calculation for the `fill` attribute uses `d.angle` (which is not used for any of the other attributes).

Figure A.3 displays the graphics image that is rendered by the code in Listing A.5.

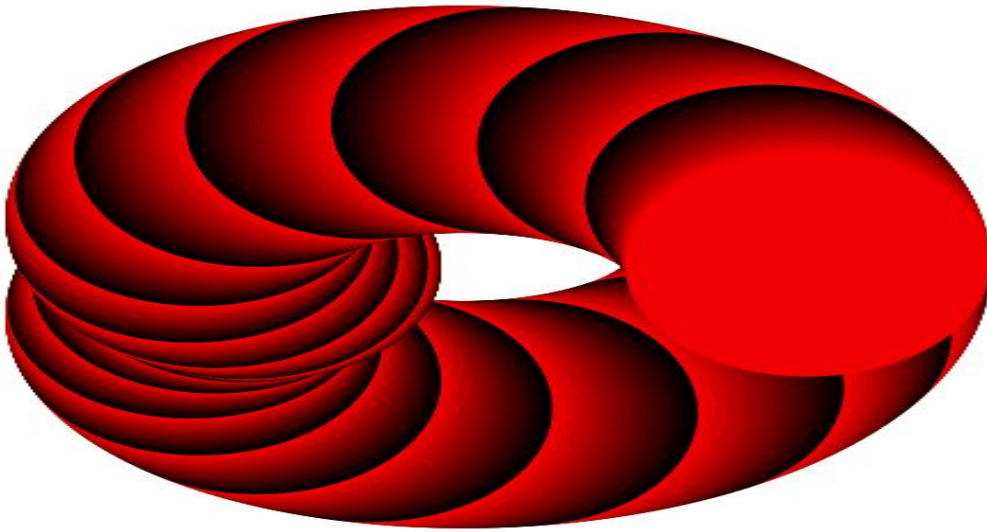


Figure A.3 Rendering a Set of Cardioid-based Gradient Ellipses.

An open source project with many similar D3-based code samples is available here:

<http://code.google.com/p/d3-graphics>

Raphael.js

`Raphael.js` is a JavaScript toolkit that creates SVG-based graphics, and its homepage is here:

<http://dmitrybaranovskiy.github.io/raphael/>

Listing A.6 displays the contents of `LituusDoubleEllipses1Grad1.html`, which illustrates how to render a set of ellipses using Raphael.

Listing A.6 LituusDoubleEllipses1Grad1.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <title>Lituus Spiral Graphics</title>

    <script src="raphael.js" type="text/javascript" charset="utf-
8"></script>

    <script>
        function init() {

            var cWidth      = 800, cHeight      = 500;
            var basePointX  = 350, basePointY  = 250;
            var currentX    = 0,   currentY    = 0;
            var offsetX     = 0,   offsetY     = 0;
            var radius      = 0,   Constant    = 200;
            var angle       = 0,   deltaAngle  = 1;
            var maxAngle    = 721, majorAxis   = 80;
            var minorAxis   = 40;

            var hexArray    = new Array('0','1','2','3','4','5','6','7',
                                         '8','9','a','b','c','d','e','f');

            var paper = Raphael("canvas", cWidth, cHeight);
            paper.rect(0, 0, cWidth, cHeight)
                .attr({fill: "#fff", stroke: "none"});

            for(angle=1; angle<maxAngle; angle+=deltaAngle) {
                radius    = Constant*Constant/angle;
                offsetX   = radius*Math.cos(angle*Math.PI/180);
                offsetY   = radius*Math.sin(angle*Math.PI/180);
                currentX  = basePointX+offsetX;
                currentY  = basePointY-offsetY;

                // red gradient
                fill = '#' + hexArray[angle%16] +'00';
```

```

        // draw the current shape
        var ellipse1 = paper.ellipse(currentX, currentY,
                                      majorAxis, minorAxis);
        ellipse1.attr({fill: fill, stroke: "none"});

        var ellipse2 = paper.ellipse(currentX+majorAxis,
                                      currentY+minorAxis,
                                      majorAxis, minorAxis);

        ellipse2.attr({fill: fill, stroke: "none"});
    }
};
</script>
</head>

<body onload="init()">
    <div id="canvas"></div>
</body>

```

Listing A.6 defines a JavaScript `onload` function that executes when the Web page is loaded. After initializing some JavaScript variables, this JavaScript function defines the variable `paper` that is a container object that “holds” the 2D shapes that are constructed later in the code. The main part of the code consists of a loop that initializes a set of ellipses and adds them to the `paper` object using the following code:

```

var ellipse1 = paper.ellipse(currentX, currentY,
                              majorAxis, minorAxis);
ellipse1.attr({fill: fill, stroke: "none"});

```

Figure A.4 displays the graphics image that is rendered by the code in Listing A.6.

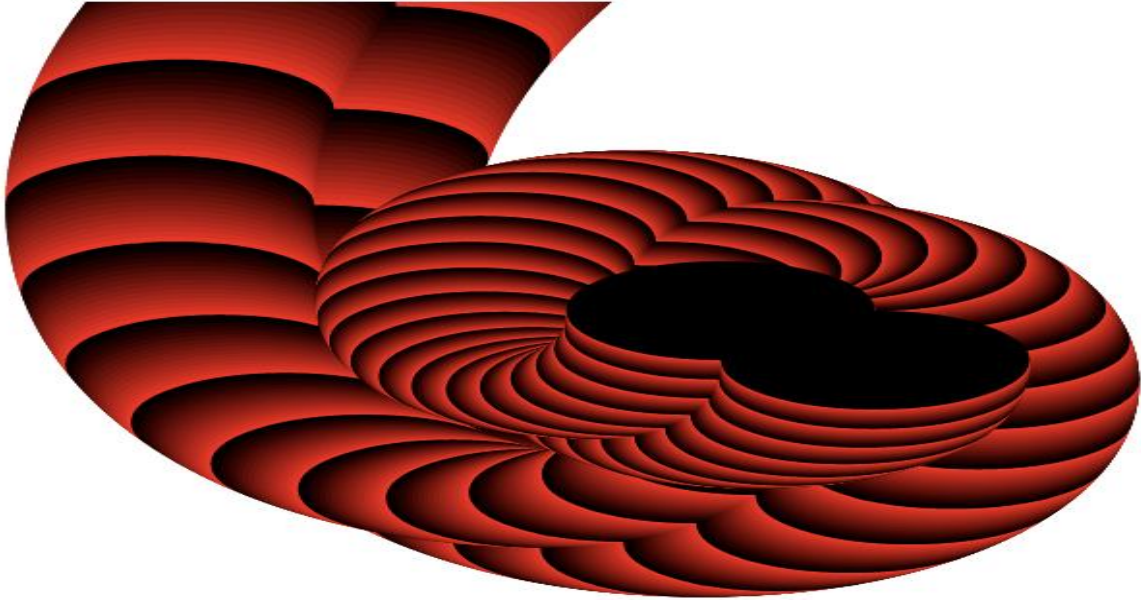


Figure A.4 Rendering a Set of Cardioid-based Gradient Ellipses.

You can find an assortment of code samples using Raphael here:

<http://code.google.com/p/raphael-graphics>

Google Go

Google Go is a programming language that is available as an open source project, and you can find the source code, demos, and tutorials on the Google Go homepage:

<http://code.google.com/p/go/>

Although the syntax of Go resembles the C programming language, there are differences: line-ending semicolons are optional, and variable declarations are usually optional. Type conversions must be made explicit, and new `go` and `select` control keywords have been introduced to support concurrent programming. New built-in types include maps, Unicode strings, array slices, and channels for inter-thread communication. The Go

language requires garbage collection, but Go does not include type inheritance, generic programming, assertions, method overloading, or pointer arithmetic.

Listing A.7 displays the contents of `GGArchEllipsesRGrad1Mod1.go`, which illustrates how to render a set of Archimedean-based radial gradient ellipses with Google Go.

Listing A.7 GGArchEllipsesRGrad1Mod1.go

```
package main

import (
    "math"
    "os"
    "github.com/ajstarks/svggo"
)

func main() {
    var offsetX, offsetY, radius float64

    basePointX    := 240
    basePointY    := 250
    majorAxis     := 60
    minorAxis     := 120
    currentX      := 0
    currentY      := 0
    Constant      := 0.25
    deltaAngle    := 1
    maxAngle      := 721
    screenWidth   := 600
    screenHeight  := 500
    //strokeStyle  := ""

    radial1 := []svg.Offcolor{
        {10, "#00cc00", 1},
        {30, "#006600", 1},
        {70, "#cc0000", 1},
    }
```



```

{90, "#000099", 1}}

g := svg.New(os.Stdout)
g.Start(screenWidth, screenHeight)
g.Title("Radial Gradients")
g.Rect(0, 0, screenWidth, screenHeight, "fill:white")
g.Def()
    g.RadialGradient("radial1", 50, 50, 100, 25, 25, radial1)
g.DefEnd()

for angle := 0; angle<maxAngle; angle+=deltaAngle {
    radius    = Constant*float64(angle);
    offsetX   = radius*math.Cos(float64(angle)*math.Pi/180);
    offsetY   = radius*math.Sin(float64(angle)*math.Pi/180);
    currentX  = basePointX+int(offsetX);
    currentY  = basePointY-int(offsetY);

    // draw the current ellipse
    g.Ellipse(currentX, currentY,
               angle % majorAxis, angle % minorAxis,
               "fill:url(#radial1)")
}

g.End()
}

```

Notice how Listing A.7 starts with a package statement, an import statement, and then the declaration of the main function using the following syntax:

```

func main() {
// code goes here
}

```

The previous function contains definitions of a set of variables using “=:” (and not “=”), followed by the variable `radial1` that consists of an array of color stops that are used for filling the ellipses later in the code listing.

The next section of code performs some SVG-related initialization and also defines a radial gradient, as shown here:

```
g.Def()
  g.RadialGradient("radial1", 50, 50, 100, 25, 25, radial1)
g.DefEnd()
```

Finally, the main loop in Listing A.7 creates and then renders a set of ellipses with the following code:

```
g.Ellipse(currentX, currentY,
          angle % majorAxis, angle % minorAxis,
          "fill:url(#radial1)")
```

Generate the SVG code by issuing the following command from the command line:

```
run go GGArchEllipses1RGrad1.go > GGArchEllipses1RGrad1.svg
```

Listing A.8 displays a truncated portion of the generated SVG document, but the accompanying CD contains the complete code listing.

Listing A.8 GGArchEllipsesRGrad1Mod1.svg

```
<?xml version="1.0"?>
<!-- Generated by SVGGo -->
<svg width="600" height="500"
      xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink">
<title>Linear Gradients</title>
<rect x="0" y="0" width="600" height="500" style="fill:white"/>
<defs>
<radialGradient id="radial1" cx="50%" cy="50%" r="100%" fx="25%"
fy="25%">
<stop offset="10%" stop-color="#00cc00" stop-opacity="1.00"/>
<stop offset="30%" stop-color="#006600" stop-opacity="1.00"/>
<stop offset="70%" stop-color="#cc0000" stop-opacity="1.00"/>
<stop offset="90%" stop-color="#000099" stop-opacity="1.00"/>
</radialGradient>
</defs>
<ellipse cx="240" cy="250" rx="0" ry="0" style="fill:url(#radial1)"/>
```

```

<ellipse cx="240" cy="250" rx="1" ry="1" style="fill:url(#radial1)"/>
<ellipse cx="240" cy="250" rx="2" ry="2" style="fill:url(#radial1)"/>
// 700 similar lines omitted for brevity
<ellipse cx="419" cy="256" rx="58" ry="118"
        style="fill:url(#radial1)"/>
<ellipse cx="419" cy="253" rx="59" ry="119"
        style="fill:url(#radial1)"/>
<ellipse cx="420" cy="250" rx="0" ry="0" style="fill:url(#radial1)"/>
</svg>

```

The code in Listing A.8 consists of standard SVG elements that we have covered in one of the book chapters, so we will skip their discussion.

Launch a browser session with the SVG document in Listing A.8 and you will see the image that is displayed in Figure A.5.

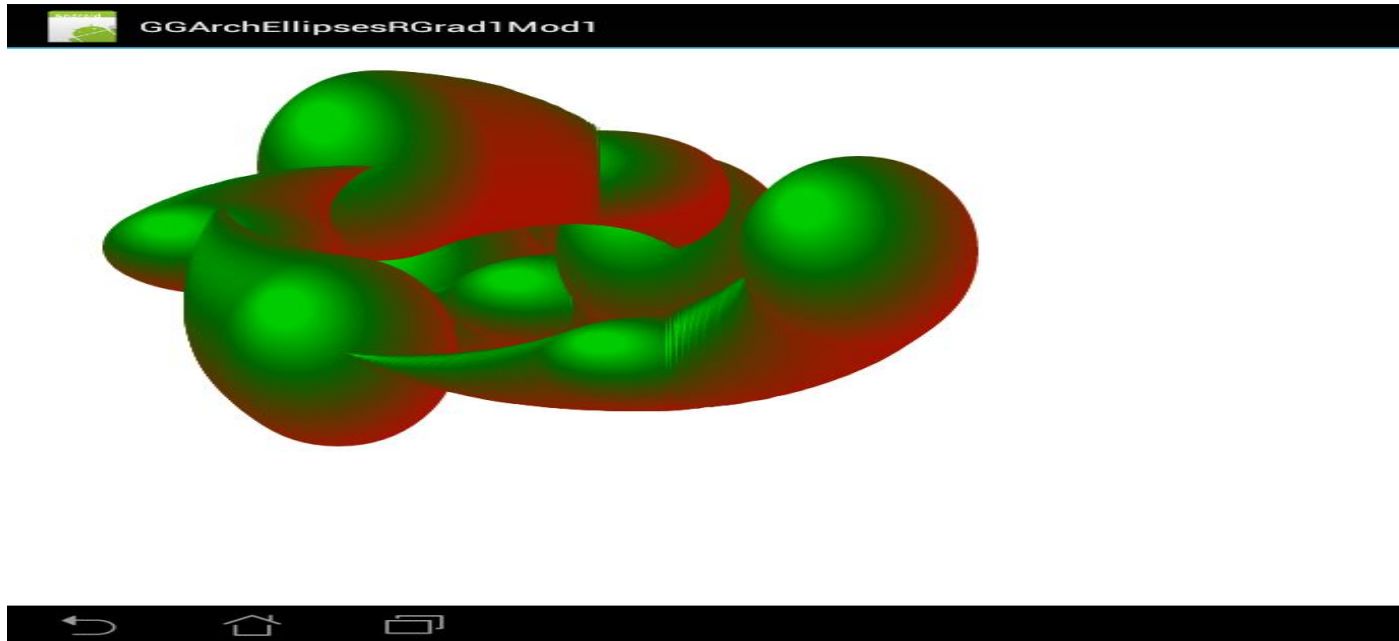


Figure A.5 SVG Generated by Go Rendered on an Asus Prime Tablet with Android ICS.

WebGL

WebGL is an implementation of OpenGL (a cross-platform 3D drawing standard created in 1992) that provides JavaScript bindings for OpenGL ES 2 in order to access OpenGL functionality. WebGL is moving toward standardization, and the process involves The Khronos Group (which is responsible for OpenGL) and browser vendors.

WebGL has a pipeline-oriented architecture, along with “fragment shaders” for textures and “vertex shaders” for managing vertex buffers. The programming language is GLSL (GL Shading Language) that is similar to the C programming language.

Currently you need to invoke the `getContext()` method with vendor prefixes, such as `moz-webgl` and `webkit-3d`, in order to obtain a 3D context. Although WebGL is arguably more complicated than drawing lines, rectangles, and shapes in HTML5 Canvas, WebGL enables you to render much richer graphics and animation effects.

You can determine whether or not your machine supports WebGL by navigating to this Website:

<http://get.webgl.org/>

There are several JavaScript-based toolkits available that provide a layer of abstraction on top of OpenGL APIs that “shield” you from having to learn the lower-level details, including ThreeJS, PhiloGL, and SceneJS.

As this book goes to print, `Three.js` has the largest number of users on its forum (more than the next two combined), which is a reasonable indicator of the level of activity surrounding `Three.js`. However, keep in mind that the pace of development of `Three.js` also means that APIs change frequently, and backward compatibility might not be available. In addition, do not discount the value of other toolkits; it’s certainly possible that they provide functionality that is closer to your needs.

One unfortunate fact is that currently there is little support for WebGL on mobile devices, but that is destined to change (hopefully soon!). You can find the status of API support for mobile devices here:

<http://mobilehtml5.org/>

Three.js

The `Three.js` open source project is a JavaScript-based toolkit that provides a layer of abstraction on top of WebGL that makes it easier for you to create rich graphics and animation effects. The source code for `Three.js` is here:

<https://github.com/mrdoob/three.js/>

<http://learningthreejs.com/blog/2012/01/17/dom-events-in-3d-space/>

Currently the `Three.js` documentation is sparse, but hopefully that will improve over time. You can find some code samples by performing an Internet search, and if you encounter difficulties, www.stackoverflow.com is a very good place to search for answers to your questions.

If you plan to use `Three.js` in your applications, the following Website contains useful boilerplate code for `Three.js`:

<http://jeromeetienne.github.com/threejsboilerplatebuilder/>

In brief, there are three objects that you must always create in `Three.js` in order to render graphics objects in an HTML page with `Three.js`:

- 1) a scene (to put things that you want people to see)
- 2) a camera (which can be moved around)
- 3) a renderer (for drawing shapes).

After creating a camera, you can move it around to see the objects in the scene from different points of view.

`Three.js` also supports three different renderers for creating graphics effects using HTML5 Canvas, WebGL, and SVG.

Some people might find `Three.js` straightforward and intuitive, and others might experience a steeper learning curve (it depends on whether or not you already have experience in 3D graphics programming).

The following online tool shows you the results of changing the attributes of a camera (rotation and position) and the position of the light source, along with the coordinates of two cubes:

<http://hotblocks.nl/tests/three/cubes.html>

Although `Three.js` provides built-in support only for cubes, cylinders, and spheres, which is not as extensive as other 2D toolkits, you can create very rich 3D visual effects with `Three.js`. You can also create 3D animation effects in `Three.js`, as shown later in this section.

Rendering a Sphere and a Cylinder using a WebGL Renderer

The example in this section is in lieu of the traditional “Hello World” application, and the code illustrates the basic sequence of steps that you need to perform in Web pages that use `Three.js`.

Listing A.9 displays the contents of `Sphere1Cylinder1.html`, which illustrates how to render a sphere and a cylinder in `Three.js`.

Listing A.9 Sphere1Cylinder1.html

```
<!DOCTYPE html>
```

```

<html lang="en">
  <head>
    <title>Sphere and Cylinder</title>
    <script src="Three45.js">
    </script>

  <script>
    function draw() {
      var renderer = new THREE.WebGLRenderer();
      renderer.setSize(window.innerWidth, window.innerHeight);
      document.body.appendChild(renderer.domElement);

      // create a camera
      var camera = new THREE.PerspectiveCamera(
        45,
        window.innerWidth/window.innerHeight,
        1, 1000);

      camera.position.z = 800;

      // create a scene
      var scene = new THREE.Scene();

      // create a sphere
      var sphere = new THREE.Mesh(
        new THREE.SphereGeometry(100, 20, 20),
        new THREE.MeshLambertMaterial({color: 0x0000ff}));

      sphere.overdraw = true;
      sphere.scale.y = 0.5;
      scene.add(sphere);

      var cylinder = new THREE.Mesh(
        new THREE.CylinderGeometry(
          100,100,200,16,4,false),
        new THREE.MeshLambertMaterial(
          {color: 0x2D303D,
           wireframe: true,
           shading: THREE.FlatShading})
      )
    }
  </script>

```

```

        );

scene.add(cylinder);

// add some ambient lighting
var ambientLight = new THREE.AmbientLight(0x555555);
scene.add(ambientLight);

// add a directional light source
var directionalLight = new THREE.DirectionalLight(0xffffff);
directionalLight.position.set(1, 1, 1).normalize();
scene.add(directionalLight);

renderer.render(scene, camera);
}
</script>
</head>

<body onload="draw()">
</body>
</html>

```

Listing A.9 contains a JavaScript function `draw()` that is invoked when the HTML Web page is launched in a browser. As you can see, this function creates a `renderer`, a `camera`, and a `scene`, which are the three objects that are required for rendering and visualizing shapes (as you learned in the previous section) with the following code:

```

var camera = new THREE.PerspectiveCamera(. . . );
camera.position.z = 800;
var scene = new THREE.Scene();
var sphere = new THREE.Mesh(

```

The next portion of Listing A.9 creates a sphere and a cylinder and adds them to the `scene` object:

```

var sphere = new THREE.Mesh(. . . );
scene.add(sphere);
var cylinder = new THREE.Mesh(. . . );

```



```
scene.add(cylinder);
```

The final portion of Listing A.9 creates an ambient light and a directional light, and after adding them to the scene object, the sphere and the cylinder in the scene are rendered with the camera:

```
var ambientLight = new THREE.AmbientLight(0x555555);
scene.add(ambientLight);
var directionalLight = new THREE.DirectionalLight(0xffffffff);
scene.add(directionalLight);
renderer.render(scene, camera);
```

Figure A.6 displays the graphics image that is rendered by the code in Listing A.9.

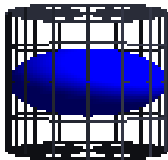


Figure A.6 Rendering A Sphere and a Cylinder with ThreeJS.

Rendering a Sphere and a Cylinder using a WebGL Renderer

Now that you understand how to render a sphere, the following code sample shows you how to render multiple spheres.

Listing A.10 displays the contents of `ThreeSpheres1.html`, which illustrates how to render three spheres using a WebGL renderer in `Three.js`.

Listing A.10 ThreeSpheres1.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Sample Spheres in Three.js</title>
```

```

<style>
  #container {
    background: #000;
    width: 600px;
    height: 400px;
  }
</style>
</head>

<body>
  <div id="container"> </div>
</body>

<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.5.1/jquery.min.js"><
/script>
<script src="Three.js"></script>

<script type="text/javascript">
  // set the scene size
  var WIDTH = 600, HEIGHT = 400;

  // set some camera attributes
  var VIEW_ANGLE = 45,
      ASPECT = WIDTH / HEIGHT,
      NEAR = 0.1,
      FAR = 10000;

  // set up the sphere vars
  var radius = 80, segments = 16, rings = 16;

  // get the DOM element to attach to
  // assume we've got jQuery to hand
  var $container = $('#container');

  // create a WebGL renderer, camera and add a scene
  var renderer = new THREE.WebGLRenderer();
  var camera = new THREE.PerspectiveCamera(VIEW_ANGLE,
                                          ASPECT,

```

```

NEAR,
FAR);

var scene = new THREE.Scene();

// the camera starts at 0,0,0 so pull it back
camera.position.z = 300;

// start the renderer
renderer.setSize(WIDTH, HEIGHT);

// attach the render-supplied DOM element
$container.append(renderer.domElement);

// create material for three spheres
var sphereMaterial1 = new THREE.MeshLambertMaterial(
    { color: 0xCC0000 });

var sphereMaterial2 = new THREE.MeshLambertMaterial(
    { color: 0xCCCC00 });

var sphereMaterial3 = new THREE.MeshLambertMaterial(
    { color: 0x0000CC });

// create a new mesh with sphere geometry
// we will cover the sphereMaterial next
var sphere1 = new THREE.Mesh(
    new THREE.SphereGeometry(radius, segments, rings),
    sphereMaterial1);

var sphere2 = new THREE.Mesh(
    new THREE.SphereGeometry(radius/2, segments, rings),
    sphereMaterial2);

var sphere3 = new THREE.Mesh(
    new THREE.SphereGeometry(radius/4, segments, rings),
    sphereMaterial3);

// move the spheres so that they are visible
sphere1.position.x = -80;

```

```

sphere1.position.y = -40;

sphere2.position.x = 100;
sphere2.position.y = 0;

sphere3.position.x = 20;
sphere3.position.y = 80;

// add the sphere to the scene
scene.add(sphere1); // largest
scene.add(sphere2); // middle
scene.add(sphere3); // smallest

// create a point light
var pointLight = new THREE.PointLight( 0xFFFFFF );

// set its position
pointLight.position.x = 10;
pointLight.position.y = 50;
pointLight.position.z = 130;

// add to the scene
scene.add(pointLight);

// draw the spheres
renderer.render(scene, camera);
</script>
</html>

```

Listing A.10 starts with the creation of a renderer, a camera, and a scene (which are the minimum requirements), followed by the creation of three meshes that are used during the creation of three spheres. For example, the first sphere is created with this code:

```

var sphere1 = new THREE.Mesh(
    new THREE.SphereGeometry(radius, segments, rings),
    sphereMaterial1);

```

Positional values are specified for the three spheres, and after they are added to the scene object, a point light is created and added to the scene, which is used during the rendering of the three spheres in the scene:

```
var pointLight = new THREE.PointLight( 0xFFFFFF );  
// code omitted  
scene.add(pointLight);  
renderer.render(scene, camera);
```

Note that if you want to use a Canvas renderer instead of a WebGL renderer, replace this line of code:

```
var renderer = new THREE.WebGLRenderer();
```

with the following line of code:

```
var renderer = new THREE.CanvasRenderer();
```

Figure A.7 displays the three spheres that are rendered by the code in Listing A.10.

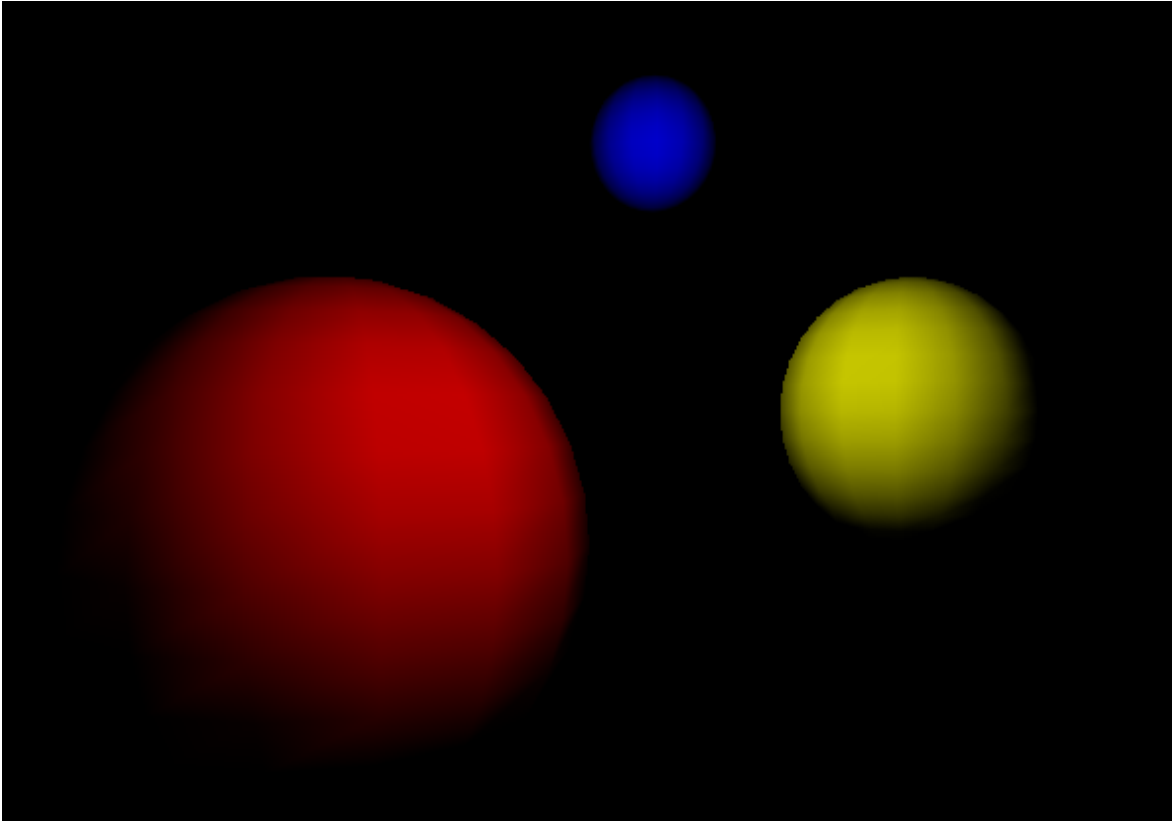


Figure A.7 Rendering Three Spheres in ThreeJS.

The tQuery jQuery Plugin for Three.js

The `tQuery` plugin for jQuery provides a layer of abstraction over the `Three.js` toolkit, and its homepage is here:

<http://jeromeetienne.github.com/tquery/>

Listing A.11 displays the contents of `Torus1.html` that illustrates the ease with which you can render a torus using a minimal amount of `tquery` code.

Listing A.11 Torus1.html

```
<!doctype html>
<html>
  <head>
    <title>Minimal tQuery Page</title>
    <meta src="utf-8" />
```

```
<script src="./tquery-all.js"></script>
<head>

<body>
  <script>
    var world  = tQuery.createWorld().boilerplate().start();
    var object = tQuery.createTorus().addTo(world);
  </script>
</body>
</html>
```

As you can see, Listing A.11 consists of very basic HTML markup, a reference to the JavaScript file `tquery-all.js`, and a mere *two lines* of JavaScript code in a `<script>` element that renders a torus (could the code be any simpler than this?)

Figure A.8 displays the torus that is rendered by the code in Listing A.11.

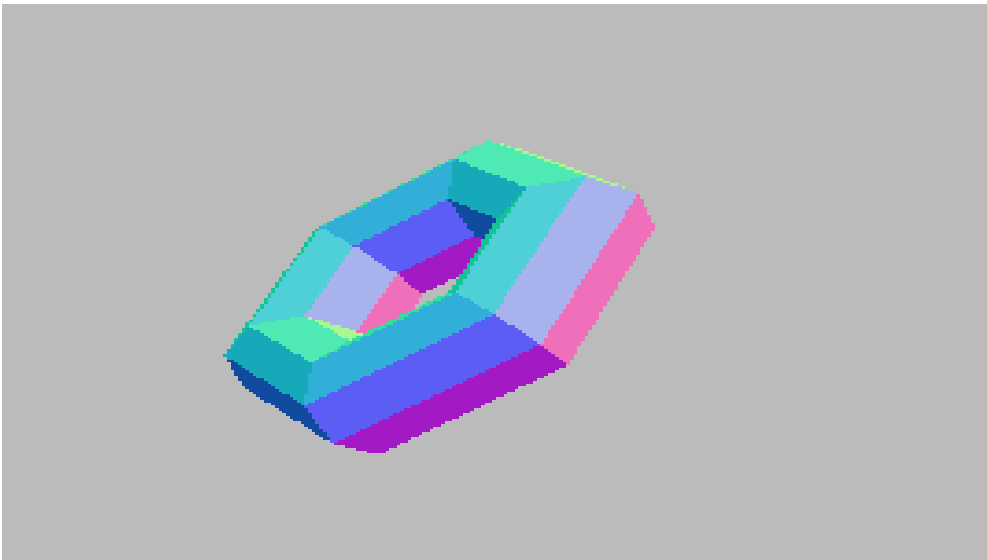


Figure A.8 Rendering a Torus with tQuery.

CSG (Constructive Solid Geometry)

This open source toolkit provides a layer of functionality on top of WebGL, and its

homepage is here:

<https://github.com/evanw/csg.js/>

Rendering Spheres and Cylinders

Listing A.12 displays most of the contents of `SphereCylinder1.html`, which illustrates how to render a sphere and a cylinder in CSG, and also how to perform set-theoretic operations, such as union, intersection, and difference.

Listing A.12 MultipleSolids1.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Multiple Solids</title>
    <meta src="utf-8" />

    <script src="lightgl.js"></script>
    <script src="csg.js"></script>
    <script src="viewer.js"></script>
  </script>
</html>

<script>
  function draw() {
    // create a cube, a sphere, and three cylinders
    var a = CSG.cube();
    var b = CSG.sphere({ radius: 1.35, stacks: 12 });
    var c = CSG.cylinder({radius: 0.7,start: [-1,0,0],end: [1,0,0]});
    var d = CSG.cylinder({radius: 0.7,start: [0,-1,0],end: [0,1,0]});
    var e = CSG.cylinder({radius: 0.7,start: [0,0,-1],end: [0,0,1]});

    // color the solids
```



```

a.setColor(1, 0, 0);
b.setColor(1, 1, 0);
c.setColor(0, 1, 0);
d.setColor(0, 0, 1);
e.setColor(1, 0, 1);

// set-theoretic operations
var operations = [
  a,
  b,
  a.inverse(),
  b.inverse(),
  a.union(b),
  b.union(a),
  a.subtract(b),
  b.subtract(a),
  a.intersect(b),
  b.intersect(a),
  c,
  d,
  c.inverse(),
  d.inverse(),
  c.union(d),
  d.union(c),
  c.subtract(d),
  d.subtract(c),
  c.intersect(d),
  d.intersect(c)
];

// render the graphics using the "id" value of <div> elements
Viewer.lineOverlay = true;
for (var i = 0; i < operations.length; i++) {
  addViewer(new Viewer(operations[i], 250, 250, 5));
}
}
</script>
</head>

```

```

<body onload="draw()">
  <div id="outer" >
    <div id="0" > </div>
    <div id="1" > </div>
    <div id="2" > </div>
    // elements omitted for brevity
    <div id="19" > </div>
    <div id="20" > </div>
  </div>
</body>
</html>

```

Listing A.12 defines a JavaScript function `draw()` that is executed when the Web page is loaded into a browser, and this method creates a cube, a sphere, and three cylinders and applies colors to these solids. The JavaScript variable `operations` is a JavaScript array containing a set of set-theoretic operations (such as union, intersect, subtract, and so forth) to perform on these solids. Note that the length of the `operations` array matches the number of `<div>` elements in the `<body>` element.

Launch Listing A.12 in a browser and notice how the objects move as you move your mouse around the screen.

Figure A.9 displays the graphics image that is rendered by the CSG code in Listing A.12.

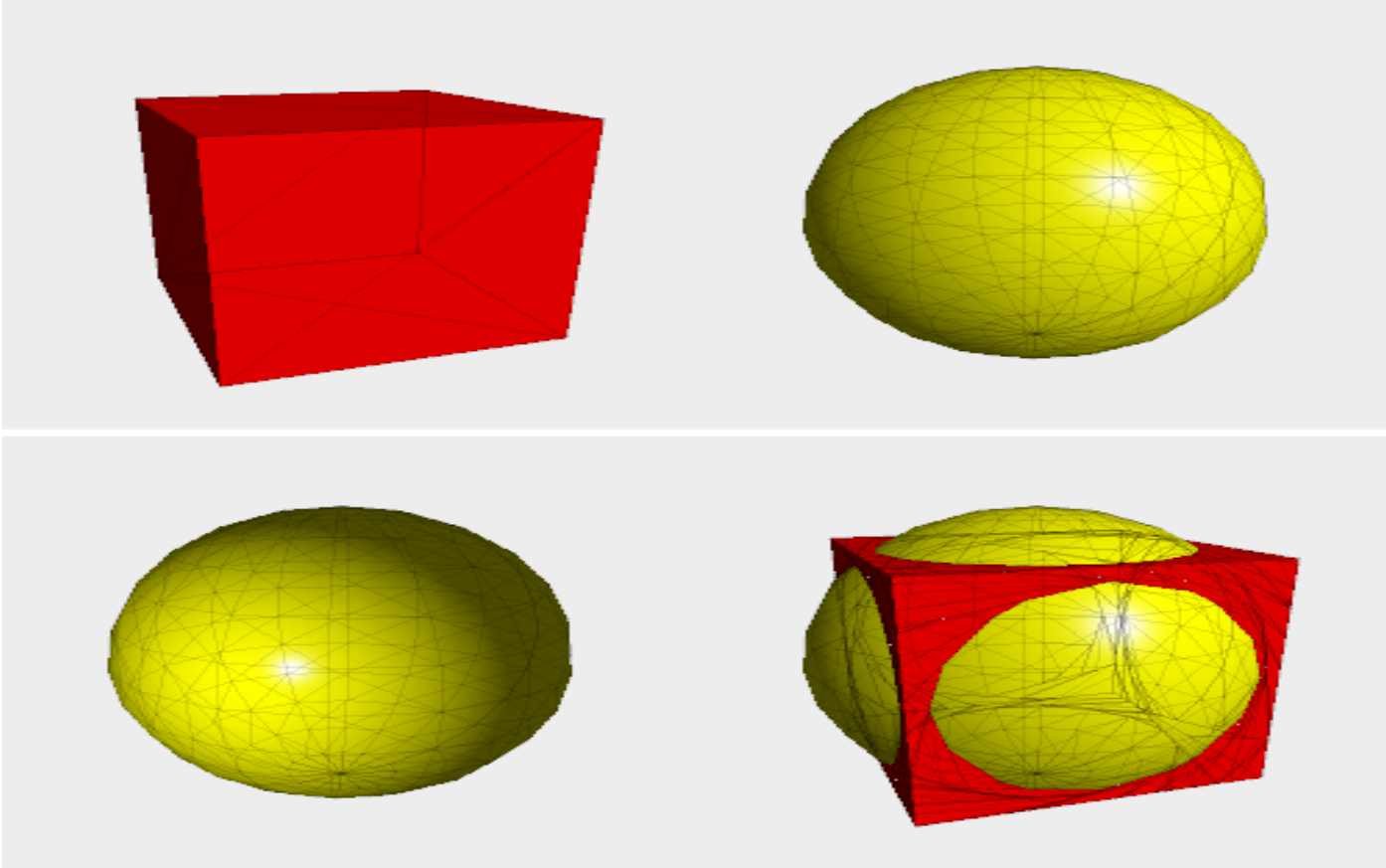


Figure A.9 Rendering 3D Objects Using Set-Theoretic Operations in tQuery.

Backbone.js

`Backbone.js` is an open source JavaScript toolkit for developing better structured Web applications, and its homepage is here:

<http://Backbone.js.org/>

Although `Backbone.js` is not an MVC (Model-View-Controller) framework, there are some similarities. `Backbone.js` provides *models* with key-value binding and custom events, *collections* with an API of enumerable functions, and *views* with declarative event handling, all of which is connected to your existing API over a RESTful JSON interface. `Backbone.js` is an open-source component of DocumentCloud, and it's available on

GitHub under the MIT software license, where you can find the source code, an online test suite, an example application, a list of tutorials, and real-world projects that use `Backbone.js`.

Download `Backbone.js` from its GitHub repository:

<https://github.com/documentcloud/backbone/>

A Brief Introduction of Backbone.js

The following subsections give you a simple overview of models, views, collections, and routers in Backbone.js. After you have read this section, you can search for online tutorials that show you examples of Backbone.js applications.

What is a Model?

According to the authors of backbone:

"Models are the heart of any JavaScript application, containing the interactive data as well as a large part of the logic surrounding it: conversions, validations, computed properties, and access control."

Whenever you create a new instance of a model, the `initialize()` method is invoked (which is also the case when you instantiate a new view in Backbone.js). A simple example is shown in the following code block:

```
Vehicle = Backbone.Model.extend({
  initialize: function() {
    alert("I am a vehicle");
  }
});
var vehicle = new Vehicle;
```

You can set parameters in a model in one of two ways. One way is to use a constructor, as shown here:

```
var vehicle = new Vehicle({model: "Ford", make: "Mustang"});
```

Another way is to use the set method, as shown here:

```
vehicle.set({make: "Ford", model: "Mustang"});
```

After setting property/value pairs, you can retrieve their values as follows:

```
var make = vehicle.get("make"); // "Mustang"
var model = vehicle.get("model"); // "Ford"
```

You can also set default values in the following manner:

```
Vehicle = Backbone.Model.extend({
  defaults: {
    make: "Ford",
    model: "Mustang"
  },
  initialize: function(){
    alert("I am a vehicle");
  }
});
```

Model Changes

You can listen for changes to the model and execute code whenever a change is detected.

For example, here the new contents of the initialize method:

```
initialize: function(){
  alert("I am a vehicle");

  this.bind("change:model", function(){
    var model = this.get("model"); // 'Stewie Griffin'
    alert("Changed my model to " + model);
  });
}
```

There are various other features available for manipulating models that you can read in the documentation.

What is a View?

```

SearchView = Backbone.View.extend({
  initialize: function(){
    alert("A view of a vehicle.");
  }
});

// The initialize function is like a constructor in that
// it is always called when you instantiate a Backbone View
var search_view = new SearchView;

```

What is a Collection?

A `Collection` in `Backbone` is an ordered set of models, and they are useful because you can include methods inside of collections to fetch data from a server, prepare that data before returning the collection, and set sample collections for debugging/testing.

Moreover, you can add event listeners and attach views to collections, which is not the case for simple JavaScript arrays.

As a simple example, we can use the `Model Vehicle` that we created in the previous section in order to create a `Collection` in `Backbone` as follows:

```

var Cars = Backbone.Collection.extend({
  model: Vehicle
});

var vehicle1 = new Vehicle({ make: "Ford",  model: "Mustang"  });
var vehicle2 = new Vehicle({ make: "GM",    model: "Camaro"   });
var vehicle3 = new Vehicle({ make: "GM",    model: "Corvette" });

var myCars = new Cars([ vehicle1, vehicle2, vehicle3]);
console.log( myCars.models ); // [vehicle1, vehicle2, vehicle3]

```

What is a Router?

Backbone routers are used for routing the URLs in your applications when using hash tags (“#”), which makes them useful for applications that need URL routing and history capabilities. Defined routers should contain at least one route and a function to map to that particular route. Keep in mind that routes interpret anything after “#” tag in the URL, and all links in your application should target either “#/action” or “#action”.

As a simple example, the following code block defines a Backbone router:

```
var MyAppRouter = Backbone.Router.extend({
  routes: {
    // matches http://example.com/#you-can-put-anything-here
    "*actions": "defaultRoute"
  },
  defaultRoute: function( actions ){
    // The variable passed in matches the variable
    // that is in the route definition "actions"
    alert( actions );
  }
});

// Initiate the router
var appRouter = new MyAppRouter;

// Start Backbone history (required for bookmarkable URLs)
Backbone.history.start();
```

This concludes a bare-bones introduction to Backbone, and the next section provides some useful links with more detailed information.

Useful Links

A collection of Backbone .js tutorials is here:

<http://backbonetutorials.com/>

An example of using Backbone.js with jQuery Mobile:

<http://coenraets.org/blog/2012/03/using-backbone-js-with-jquery-mobile/>

<http://weblog.bocoup.com/organizing-your-backbone-js-application-with-modules/>

Twitter Bootstrap

Currently Twitter Bootstrap is the most watched and forked repository in GitHub, and it provides simple and flexible HTML, CSS, and JavaScript for user interface components and interactions, you can download Twitter Bootstrap here:

<http://twitter.github.com/bootstrap/>

<http://blog.getbootstrap.com/>

Bootstrap was designed primarily as a style guide to document best practices, and also to be for people diverse skill levels. Bootstrap supports new HTML5 elements and syntax, and you can use Bootstrap as a complete kit or to start something more complex.

Bootstrap was initially created for modern browsers, but it has expanded its support to include all major browsers (including IE7). In addition, Bootstrap 2 supports tablets and smartphones.

Bootstrap 2 supports tablets and smartphones, and its responsive design means that its components are scaled according to a range of resolutions and devices, thereby providing

a consistent experience. Moreover, Bootstrap provides custom-built jQuery plugins, and it's built on top of the LESS toolkit. Some features of Bootstrap 2.0 are tooltips, styled stateful buttons, more table and form styling, and an improved structure for its CSS source code (multiple files instead of a single monolithic file).

Bootstrap handles layouts and provides various components, as well as support for popovers, dropdown menus, carousel content, modals, and other functionality. In addition, Bootstrap 2.0 supports responsive design for mobile devices. If you prefer, you can create your own custom download if you do not want to download the full toolkit.

Bootstrap supports styling for various widgets, including buttons and button groups, tabs, navigation bars, form controls, navigation lists, labels, and others.

One interesting Bootstrap feature is its support for progress bars using the CSS classes `.bar` and `.progress` that are available in Bootstrap. As an illustration, the following code block shows how to render bars with different colors based on custom attributes that start with the string `progress-`, as shown here:

```
<div class="progress progress-info" style="margin-bottom: 9px;">
  <div class="bar" style="width: 20%"></div>
</div>
<div class="progress progress-success" style="margin-bottom: 9px;">
  <div class="bar" style="width: 40%"></div>
</div>
<div class="progress progress-warning" style="margin-bottom: 9px;">
  <div class="bar" style="width: 60%"></div>
</div>
<div class="progress progress-danger" style="margin-bottom: 9px;">
  <div class="bar" style="width: 80%"></div>
</div>
```

Useful Links

A sample application using Twitter Bootstrap is here:

<https://dev.twitter.com/blog/say-hello-to-bootstrap-2>

Tutorials and videos:

<http://webdesign.tutsplus.com/tutorials/workflow-tutorials/twitter-bootstrap-101-tabs-and-pills/>

Twitter Bootstrap examples:

<http://twitter.github.com/bootstrap/examples.html>

An example of Backbone .js with Twitter Bootstrap:

<http://coenraets.org/blog/2012/02/sample-app-with-backbone-js-and-twitter-bootstrap/>

Summary

This Appendix provided an overview of several JavaScript toolkits that are available in various categories. You learned about toolkits that provide a layer of abstraction above HTML5 Canvas and SVG. In particular, you learned about the following toolkits:

- EaselJS
- FabricJS
- PaperJS
- D3
- Raphael
- CSG
- Backbone.js
- Twitter Bootstrap

