A

Picking Your Graphics Technology

In this section, we will cover the following topics:

- Graphics with SVG
- Transitioning with CSS3 transitions
- Transforming things with CSS3 transforms
- Creating CSS timelines
- Getting started with OpenGL/WebGL

Introduction

Until recently, Flash was the only viable solution to create dynamic and interactive graphics; but with the new surge of supported technologies in HTML5, an appendix would be a good place to give you an overview of animation/drawing styles in HTML5. These days, HTML5 offers a large array of options that are becoming an alternative to Flash.

Although this book is mainly focused on creating graphics using canvas and open source tools, we wanted to explore alternative options in this section. This section is not an exhaustive consideration of these alternatives, but instead can be used as a guide to other options in creating content in HTML5.

Graphics with SVG

In this recipe we will introduce **Scalable Vector Graphics** (**SVG**). SVG is a large topic on its own and is now supported in HTML5. Let's see it in action. In our example, we will create a blue star with live text in it.

SVG is an XML-based vector language that enables drawing using XML nodes. The easiest way to work with SVG is using tools such as Inkscape, which enable you to output the XML format while working in a graphical environment. It is supported on all major modern browsers.

How to do it...

Let's just jump right into it:

1. We start with an empty HTML5 document:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
  <title>SVG Example</title>
  </head>
  <body style="background-color:#333333;">
    <svg xmlns="http://www.w3.org/2000/svg" version="1.1"
width='200' height="200">
  </svg>
  </body>
</html>
```

2. Integrate the SVG code into the HTML body:

```
  <body style="background-color:#333333;">
    <svg xmlns="http://www.w3.org/2000/svg" version="1.1"
width='200' height="200">
  </svg>
  </body>
```

3. Our first step was to define the SVG node and set its rules and version number, width, and height.
4. Let's start drawing:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" width='200'
height="200">
    <polygon points="100,10 40,180 190,60 10,60 160,180"
style="fill:blue;fill-rule:nonzero;"/>
    <text x="50" y="52" fill="white" transform="rotate(38
20,40)">Original Text</text>
  </svg>
```

How it works...

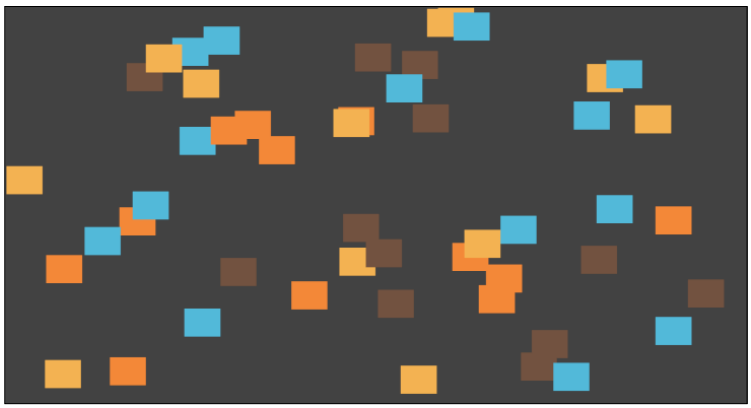
In this example, we draw a star providing its five 2D coordinates to create it, followed by creating a text element and rotating it to overlap our polygon.

SVG is that easy! Usually you wouldn't actually create the visuals via code directly, but would use a graphical interface.

Check out Inkscape's site; there are many very cool samples of what you can do with it, and it's all free. You can find it at `http://inkscape.org`.

Transitioning with CSS3 transitions

CSS3 introduced an array of animation options that enable us to create complex programming tasks with minimal code and sometimes no code at all. In this task we will spread boxes on the screen that will have rollover animations controlled via CSS.



How to do it...

Let's get started:

1. Create an empty HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
       <title>CSS Transforms Example</title>

  </head>
  <body>
  <div class="box"/>
</body>
</html>
```

2. Add style to the HTML header:

```
<style>
    .box {
        position:absolute;
        width: 30px;
        height: 30px;
        -webkit-transition:  all .3s;
        -moz-transition:  all .3s;
        -o-transition:  all .3s;
        -ms-transition: all .3s;
        transition: all .3s;
        background-color:#5E4130;
    }
    .box:hover {
      width:50px;
      height:50px;
    }

  </style>
```

3. Next, let's create a `div` element with the class `box`:

```
<body>
  <div class="box"/>
</body>
```

That's it! You've used CSS to create a transitioning box.

How it works...

Time to look into our CSS. In the CSS we created a `box` class and added a `hover` state to this box.

```
position:absolute;
width: 30px;
height: 30px;
```

The box is set to have an absolute positioning and a width and height of 30 pixels.

```
-webkit-transition:  all .3s;
-moz-transition:  all .3s;
-o-transition:  all .3s;
-ms-transition: all .3s;
transition: all .3s;
```

The next five lines are all basically the same line. HTML5 is still evolving and changing and hasn't been finalized and so the features in the browsers are updating and getting modified over time. As such the browsers provide their labs features with an extra extension to the keyword. The way to incorporate experimental features is by prepending to the feature the browser code, for example `-mos-` stands for Mozilla Firefox. In the future, all these will be represented directly within transition when it's the final working product. This enables us to create modifications based on the browsers used as well. In our case, we are calling the `transition` property and setting it to animate any property that is changed by giving it a 0.3 seconds animation time. We can list out properties, set an ease, and so on, but for our example let's keep it simple.

Next we define what will happen when a user rolls over the `box` class:

```
.box:hover {
  width:50px;
  height:50px;
}
```

All we are doing in this state is making our box bigger, from 30x30 to 50x50. Run the application now and you will find a box on the screen that animates and expands when you roll over it.

The magic is literally in the one CSS line, `transition`, and its counterparts, which enable everything to move around. So easy! But this isn't enough. Let's integrate some JavaScript into this next.

There's more...

Let's revisit our code and make it more interactive and dynamic:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
  <title>CSS Transforms Example</title>
  <script src="01.03.css.js"></script>
  <style>
    .box {
        position:absolute;
        width: 30px;
        height: 30px;
        -webkit-transition:  all .3s;
        -moz-transition:  all .3s;
        -o-transition:  all .3s;
        -ms-transition: all .3s;
        transition: all .3s;
        background-color:#5E4130;
    }

    .box:hover {
      width:50px;
      height:50px;
    }
  </style>
  </head>
  <body onLoad="init();" style="background-color:#333333;">
      <div class="box" />
 </body>
</html>
```

Before we can start coding, we need to remove a few things. Let's remove the background color from our element, as we want to create this dynamically in JavaScript and we want to remove the `div` element from our code body as well (I've highlighted these lines).

```
<script src="01.03.css.js"></script>
```

The next step is to add the external JavaScript and start our project.

```
var BOX_COUNT = 50;
var ANIM_AREA_WIDTH = 600;
var ANIM_AREA_HEIGHT = 400;
```

```
var aColors=["#43A9D1","#EFA63B","#EF7625","#5E4130"];
var COLORS_LENGTH = aColors.length;
function init(){
  var box;
  for(var i=0; i<BOX_COUNT;i++){
    box = document.createElement('div');
    box.setAttribute('id', 'box_'+i);
    box.setAttribute('class', 'box');
    updateIt(box);
    document.body.appendChild(box);

  }
}
function updateIt(it){
  it.style.top=Math.floor(Math.random()*ANIM_AREA_HEIGHT)+'px';
  it.style.left=Math.floor(Math.random()*ANIM_AREA_WIDTH)+'px'
it.style.backgroundColor=aColors[Math.floor(Math.random()*COLORS_
LENGTH)];
}
```

Okay, let's break this down a bit. Our first steps are to set up our global variables:

```
var BOX_COUNT = 50;
var ANIM_AREA_WIDTH = 600;
var ANIM_AREA_HEIGHT = 400;
var aColors=["#43A9D1","#EFA63B","#EF7625","#5E4130"];
var COLORS_LENGTH = aColors.length;
```

Our `BOX_COUNT` variable will define how many boxes we create, while the `ANIM_AREA_WIDTH` and `ANIM_AREA_HEIGHT` variables will define the boundaries of the boxes we create. We already met our `aColors` array in previous tasks (an array of colors) and the `COLORS_LENGTH` variable that we create to cut down the processing time of fetching the length of our array, as we will not change it after it's created.

Time to look into the `init` function:

```
function init(){

  var box;
  for(var i=0; i<BOX_COUNT;i++){
    box = document.createElement('div');
    box.setAttribute('id', 'box_'+i);
    box.setAttribute('class', 'box');
    updateIt(box);

    document.body.appendChild(box);

  }
}
```

In this function we loop through our `BOX_COUNT` variable and create 50 boxes. We give an ID to each as `'box_'+i` (box_0, box_1 and so on) and set each box to use the class `box`. We then call the `updateIt` function and send a `box` parameter to it (we will see what happens there next) and add our new `box` parameter into our `document.body` element using the `appendChild` method.

It's time to explore what the `updateIt` function does:

```
function updateIt(it)
{
  it.style.top=Math.floor(Math.random()*ANIM_AREA_HEIGHT)+'px';
  it.style.left=Math.floor(Math.random()*ANIM_AREA_WIDTH)+'px';
  it.style.backgroundColor=aColors[Math.floor(Math.random()*COLORS_
LENGTH)];

}
```

The `updateIt` function takes in a `box` parameter and then manipulates its position (top, left) and its background color. It randomly selects the position by using the `Math.random()` method and multiplies it by the width and height respectively.

Last but not the least, we define the background color by randomly selecting a color from the `aColors` array. Before we set this, our box has no color as we removed its base color from the CSS earlier.

Time to run the application! You will find 50 boxes spread on the screen and if you roll over them, they will animate and open up.

Adding more complexity

I wanted to really showcase how creative you can be using this power, by adding a small random animation that would change the color and positions of a few of the boxes that are visible. Let's revisit the JavaScript and add the highlighted code:

```
function init(){

  var box;
  for(var i=0; i<BOX_COUNT;i++){
    box = document.createElement('div');
    box.setAttribute('id', 'box_'+i);
    box.setAttribute('class', 'box');
    updateIt(box);

    document.body.appendChild(box);

  }
  setInterval(updateThings,2000);
}
```

```
function updateThings(){
  var trgt;
  for(var i=0;i<10;i++){
    trgt = "box_" + Math.floor(Math.random()*BOX_COUNT);
    updateIt(document.getElementById(trgt));

  }
}
```

We added into our `init` function an interval that will call the `updateThings` function once every two seconds. In the `updateThings` function, we randomly select up to 10 boxes and change their values (I say "up to" because we might select the same box a few times; but that's okay, as it adds something to the random feeling of this sample). We select a random box and then send it again to the `updateIt` function to update its color and position. Now, when you run the application you will see that once every two seconds elements start moving around and change their color.

See also

- The *Transforming with CSS3 transforms* and *Creating CSS timelines* recipes in this section.

Transforming with CSS3 transforms

How to create CSS3 transitions is now in our tool set and it's time for us to integrate this new learning with another really cool CSS3 powerhouse. Transforms enable us to take any HTML element and transform its scale, rotation, and so on. In this sample we will keep it simple, but in the future (maybe now is the future) you will be able to create Z animations (also known as sudo 3D) in a 2D context; I can't wait for that to be reality!

In this task we will use what we built in the previous example and add into it sequential animations and transforms using JavaScript, to control the CSS3 property.

How to do it

Let's start by creating our base HTML5 file.

1. Create the HTML file with some slides:

```
<!DOCTYPE html>
<html>

  <head>
    <title>CSS Transforms Example</title>
  </head>

  <body>

    <div class="main">
         <div id="slide1" class="slide" >
             <img src="images/slide1.jpg" />
           </div>
           <div id="slide2" class="slide" >
             <img src="images/slide2.jpg" />
           </div>

           <div id="slide3" class="slide" >
              <img src="images/slide3.jpg" />
           </div>

           <div id="slide4" class="slide" >
             <img src="images/slide4.jpg" />
           </div>
      </div>
  </body>

</html>
```

2. Add CSS styles into the header:

```
<style>
  .main{
    width:100%;
    height:500px;
    position:absolute;
    overflow:hidden;
  }
  .slide{
    display: block;
    height: 210px;
    position:absolute;
```

```
    width: 720px;
    top:500px;
    -webkit-transition: all 1s ease-in-out;
    -moz-transition: all 1s ease-in-out;
    -o-transition: all 1s ease-in-out;
    -ms-transition: all 1s ease-in-out;
    transition: all 1s ease-in-out;
  }


.main{
    width:100%;
    height:500px;
    position:absolute;
    overflow:hidden;
  }
</style>
```

3. Integrate the external JavaScript file:

```
<head>
    <title>CSS Transforms Example</title>
    <script src="01.04.transform.js"></script>
  <style>
  …
```

4. Before we close down our HTML file and start working on the JavaScript, let's add an `init` function call as soon as the document is done loading:

```
<body onLoad="init();">
```

5. Add new JavaScript code into the file `01.04.transform.js`:

```
var slide1 ;
var slide2 ;
var slide3 ;
var slide4 ;

function init(){
  slide1 = document.getElementById('slide1');
  slide1.style.top = "10px";
  slide1.style["MozTransform"] = "scale(0.2)";

  setTimeout(slide1_step2,1500);

}

function slide1_step2(){
  slide1.style["MozTransform"] = "rotate(120deg)";
```

```
    slide1.style.opacity = 0;
    slide2_step1();
  }

  function slide2_step1(){
    slide2 = document.getElementById('slide2');
    slide2.style.top = "30px";
    slide2.style.left = "50px";
    slide2.style["MozTransform"] = "scale(0.22) rotate(-6deg)";

  }
```

That's it! Now, when you go to your browser you will find our sample working.

How it works...

Let's start by looking into the CSS. The idea behind the class `main` is simple. Our first step is to create our main area. We define our main area to be 100 percent of the width of the screen, and limit it to the top 500 px of the screen so content that is out of this area will be hidden.

```
.main{
    width:100%;
    height:500px;
    position:absolute;
    overflow:hidden;
  }
```

While our slide is a bit more detailed, our slides will always stay within our `main` function. Each of them will have a base size of 720 x 210, their positions will be absolute, and we will set a transition (same as we learned about in the previous task) to animate all properties. It will take one second and it will have an ease-in-out effect. Last but not the least, when starting up the application we don't want anything to be visible, so we are pushing all content out of the visible area of the screen by setting the top value to be 500 px (as the main height is 500 px).

In the next few lines in the JavaScript file, we are going to create the steps of our animation. I've only created a few of the steps and left it open to you to continue and expand on it.

The first step is to create the global variables so we don't need to redefine them in each function. For that we create our variables `slide1` through `slide4`:

```
var slide1 ;
var slide2 ;
var slide3 ;
var slide4 ;
```

Our `init` function is the first to be called:

```
function init(){
  slide1 = document.getElementById('slide1');
  slide1.style.top = "10px";
  slide1.style["MozTransform"] = "scale(0.2)";


}
```

The first step is to fetch our `slide1` variable from the `div` element, and the next step is to bring our element onto the stage by setting its top position to be 10 px (it was 500 px before). The next line is really the heart of this sample:

```
slide1.style["MozTransform"] = "scale(0.2)";
```

In this line, we set a new transform. Notice that we are using `"MozTransform"`. This property only works on a Mozilla Firefox browser. If we were building a production-ready application, we would then need to accommodate all the browsers and not focus only on the `Moz` option; but for the sake of this example, let's assume our application will only run on Firefox. We then set the transform value to be `"scale(0.2)"`, letting our application know that we want to scale the `div` layer to be 0.2 of its original value.

If you run the application now without adding in the new `setTimeout` function, you will find the image will be animated into the screen (it will take one second).

If we wanted we could have stopped in our tracks, but we wanted to outline the idea of controlling time-based animations using CSS and JavaScript. Our next step is to add a delay so our image will be visible for a bit before continuing. We will use the `setTimeout` function to control when the next step in our timeline animation will start:

```
function init(){
  slide1 = document.getElementById('slide1');
  slide1.style.top = "10px";
  slide1.style["MozTransform"] = "scale(0.2)";

  setTimeout(slide1_step2,1500);

}
```

As the animation is one second long, we are letting the image linger on the screen for an extra 500 milliseconds before the function `slide1_step2` will trigger.

```
function slide1_step2(){
  slide1.style["MozTransform"] = "rotate(120deg)";
  slide1.style.opacity = 0;
  slide2_step1();
}
```

We then, in the next function, update the rotation of our element to go to 120 degrees and fade out of the screen, and at the same time we trigger the call to the next step. If we wanted, we could have added a delay using a `setTimeout` function. In this case we want the animations to happen at the same time.

```
function slide2_step1(){
  slide2 = document.getElementById('slide2');
  slide2.style.top = "30px";
  slide2.style.left = "50px";
  slide2.style["MozTransform"] = "scale(0.22) rotate(-6deg)";

}
```

In this step we fetch the next slide, `slide2`, and position it to go to the point (50, 30), and add a transform to it to scale to 22 percent and rotate by only 6 degrees anti-clockwise.

At this stage, you should be comfortable working with CSS3 transitions.

There's more...

Before I wrap up this demo, I want to point out a few important things for you to consider.

Why break down to so many functions

It's very common while working with creatives that they want to make small tweaks like changing the speed of animations when something starts going in and out, and breaking down the steps makes it easier than to skip steps or to modify the transition times.

Avoid creating separate objects over and over

At the start of the application you could locate in which browser you are running and based on it pass a dynamic value into the `style` object, like the following example:

```
slide1.style["MozTransform"]
```

This way, you only need to figure out the property to be used and then you can reuse it as many times as you need throughout your application.

```
var sTrans;
if(slide1.style["MozTransform"])  sTrans = "MozTransform";
...
slide1.style[sTrans]
```

See also

- The *Transitioning with CSS3 transitions* and *Creating CSS timelines* recipes in this section.

Creating CSS timelines

In the previous two recipes we learned about transitions and transforms; to wrap up this topic, we need to visit timelines—another really cool and very powerful feature of CSS3.

Getting ready

We start our application from a tweaked version of the previous sample. We will start over, but use the same baseline. This time around we have only two images and we have manually set their positions and their rotation values.

How to do it...

This time we are going to do everything in CSS without JavaScript support.

1. We start by setting up our HTML file by removing the JavaScript references and leaving only two images:

```
<!DOCTYPE html>
<html>

  <head>
    <title>CSS Timeline Example</title>
   <style>
  .main{
    width:100%;
    height:500px;
    position:absolute;
    overflow:hidden;
  }
  .slide{
    display: block;
    position:absolute;
    width: 720px;
    top:50px;
    -webkit-transition: all 1s ease-in-out;
    -moz-transition: all 1s ease-in-out;     -o-transition: all 1s
ease-in-out;
    -ms-transition: all 1s ease-in-out;
  }
  .slide img{ width:266px; height:200px}

  #slide1{
    left:100px;
    -moz-transform: rotate(-15deg);
  }
  #slide2{
    left: 450px;
    -moz-transform: rotate(15deg);
  }


    </style>
  </head>
  <body>

    <div class="main">
                <div id="slide1" class="slide" >
```

```
                    <img src="images/slide1.jpg" />
                  </div>
                  <div id="slide2" class="slide" >
                    <img src="images/slide2.jpg" />
                  </div>

          </div>
  </body>

</html>
```

2. In the CSS, we want to add a timeline or, as it's called in CSS3, we want to set our **keyframes**. It works in a very simple way:

```
@keyframes spin{
    0% { transform:rotate(0) scale(1) }
    25% { transform:rotate(360deg) scale(.9) }
    50% { transform:rotate(720deg) scale(.8) }
    75% { transform:rotate(720deg) scale(.9) }
    100% { transform:rotate(720deg) scale(1) }

  }
```

3. Keyframes aren't widely supported as a final product, yet; we need to define separate values for each browser. So, for example, for Firefox we would write it in the following way:

```
@-moz-keyframes spin {
  0% { -moz-transform:rotate(0) scale(1) }
  25% { -moz-transform:rotate(360deg) scale(.9) }
  50% { -moz-transform:rotate(720deg) scale(.8) }
  75% { -moz-transform:rotate(360deg) scale(.9) }
  100% { -moz-transform:rotate(0deg) scale(1) }

  }
```

4. Now define in CSS what will animate and for how long:

```
.slide img:hover{
    -moz-animation-name:spin;
     -moz-animation-duration:5000ms;
      -moz-animation-iteration-count:2;
      -moz-animation-timing-function: ease-in;
  }
```

When you open the browser window you will find our images animating.

How it works...

We set a keyframe using the `@keyframes` attribute and then give the keyframe a name. In the selection area (between the opening and closing braces), we outline points in time and define within them we will describe the CSS rules we want to apply in that given time.

Again, in this sample, we are focusing only on Firefox, but you understand how to make this work for any browser that supports CSS3 keyframes.

Our next step is to define what and how things animate:

```
.slide img:hover{
    -moz-animation-name:spin;
     -moz-animation-duration:5000ms;
      -moz-animation-iteration-count:2;
      -moz-animation-timing-function: ease-in;
  }
```

The steps are very simple; we first define a rollover state for our images and set `spin` as the animation name. We now connect the animation and the image. The last steps left are to define the duration of the animation, duration count, and ease type (animation-timing-function).

That's all we needed to do. No JavaScript, all done via CSS!

Getting started with OpenGL/WebGL

Don't get too excited about this recipe as it's really just aimed to let you know about this really amazing option. OpenGL is a way to create really amazing 3D graphics in HTML5. As this topic is so vast and complicated, in this sample we will just see how to start up a project and point you to some resources to continue from that point.

Getting ready

Don't worry, this sample is going to be simple; but to actually develop graphics in 3D will take a bit more investment than this recipe could do justice to. So breathe and let's start!

How to do it...

Our application on the HTML end looks just about the same as our earlier samples. We set a canvas area. While in more advanced samples there is extra information that would be needed in the HTML area, we will not cover that information in this book.

1. Start by creating the HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
<script src="01.02.opengl.js"></script>
   <title>WebGL Basic Example</title>
   </head>
<body onLoad="init();" style="margin:0px">
  <canvas id="myCanvas" width="500" height="500"/>
   </body>
</html>
```

2. We will create one variable `gl` to contain the WebGL context:

```
var gl;
```

3. Create the `init` function (all the code in the next steps will be inside the `init` function):

```
function init(){
   //all code in next steps here
}
```

4. Get a reference to the canvas element:

```
var c = document.getElementById('myCanvas');
```

5. Grab the WebGL context and set it into the `gl` variable:

```
gl = c.getContext("webgl") || c.getContext("experimental-webgl");
```

6. Time to set a default color:

```
gl.clearColor(0.99,0.92,0.72,1.0);
```

7. Last but not the least, let's fill our canvas with our `clear` color:

```
   gl.clear(gl.COLOR_BUFFER_BIT);
```

You might think this is too complex. I agree. To create a simple background is a lot of work, but as you move into real 3D applications, this complication is needed. Probably our sample would be better if it were to be done in a real-world application on a 2D canvas.

How it works...

We deliberately kept the JavaScript simple in this example, as any more complexity than this would take a nice chunk of pages to break into.

Our first step is to get our canvas element:

```
var c = document.getElementById('myCanvas');
```

Our next step is to fetch the 3D context, also known as the WebGL. In this case we test to see if WebGL is present; if not, we go for the experimental one:

```
gl = c.getContext("webgl") || c.getContext("experimental-webgl");
```

Now that we have our 3D context, the first thing we want to do with it is to set its default clear color:

```
gl.clearColor(0.99,0.92,0.72,1.0);
```

The first three parameters are the red, green, and blue (with values between 0 and 1), while the last parameter is to define the alpha channel.

```
gl.clear(gl.COLOR_BUFFER_BIT);
```

This line is the line that will render our page in the color we requested in the parameter `gl.COLOR_BUFFER_BIT`. This is just a way to tell OpenGL to use the color we defined as the color for this operation.

There's more...

There is actually a lot more than the background we just created in this sample. Going deeper into the 3D world of OpenGL is just out of the scope of this book, but it was important to outline this language. If you're interested in getting deeper into building in 3D, please check a few of the resources attached in the downloadable code pack.

See also

For more information about OpenGL, see *OpenGL 4.0 Shading Language Cookbook*, *David Wolff*, *Packt Publishing Limited*.