

Canvas and CSS

Pushkar Joshi, Motorola Mobility

[HTML5 Canvas](#)

[Drawing Tool](#)

[Fill and Stroke Styles](#)

[Layout and Transformations](#)

[Image Editing](#)

[Animation](#)

[Comparison with SVG](#)

[CSS](#)

[Planes in Space](#)

[Animations](#)

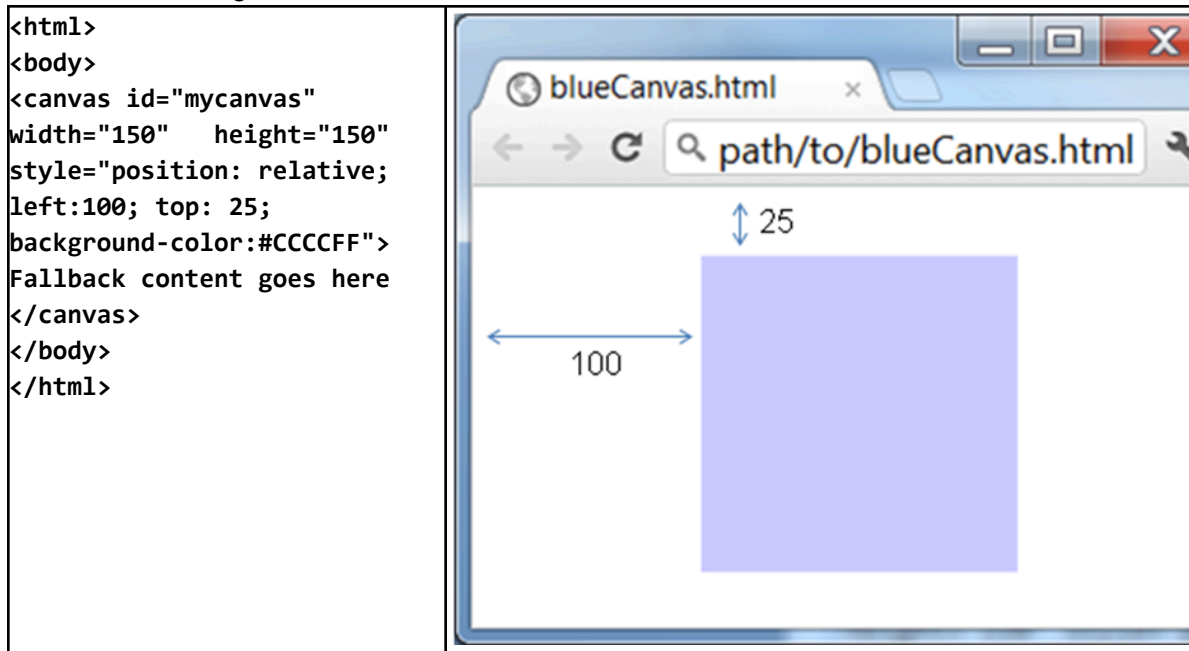
[Image Filters](#)

[General-Purpose CSS Shaders](#)

HTML5 Canvas

Once we are introduced to HTML, CSS, and Javascript, we are ready to learn about the canvas element that was introduced in HTML5. The canvas element is the easiest way to obtain an interactive drawing surface for a web page on a modern browser, and is currently extensively used for creating web-based games.

Similar to other block content tags like `<div>` or `<p>`, the `<canvas>` tag identifies a rectangular region of the browser window. Standard CSS operations (like setting the width, height, background color, and position) that can be performed on standard content tags like `<div>` can also be performed on the `<canvas>` tag. In case the browser cannot display the canvas (i.e. older browsers), we display a fallback message contained within the beginning `<canvas>` and end `</canvas>` tags.



The HTML document on the left produces a blue square canvas shown on the right. The canvas is offset by 100 pixels from the left and 25 pixels from the top, as specified by its style in the `<canvas>` tag above. Notice that with older browsers that do not support HTML5, the text “Fallback content goes here” will be displayed instead of the canvas.

Unlike the other HTML tags, the `<canvas>` tag offers a drawing context that can access and paint the individual pixels inside the canvas. People familiar with OpenGL or DirectX will be familiar with the notion of a drawing context. A drawing context is essentially the “surface” on which you can draw/paint your pixels. The standard method for accessing the drawing context is through Javascript. We have added some Javascript to the HTML document from earlier. This script queries the DOM and then calls the “`getContext()`” function of the canvas object:

```

<html>
<head>
<script type="application/javascript">
  function draw() {
    var canvas = document.getElementById("mycanvas");
    if (canvas.getContext) {
      var context = canvas.getContext("2d");
      //issue drawing commands here...
    }
  }
</script>
</head>
<body onload="draw();">
  <canvas id="mycanvas" width="150" height="150" style="position: relative; left:100px; top:
25px; background-color:#CCCCFF">
    Fallback content goes here
  </canvas>
</body>
</html>

```

Currently, two types of contexts are supported: a 2d context that offers the ability to manipulate the canvas like a bitmap, and a WebGL context that offers the ability to draw in 3D. In this section, we will cover the 2D drawing context, and WebGL will be covered in a separate section.

The coordinate system of the 2D context has its origin in the top left corner of the content of the box. The coordinates of any objects drawn in this context must lie in the range [0, canvas width] and [0, canvas height]. Any objects that do not lie within this range will be ignored and clipped.

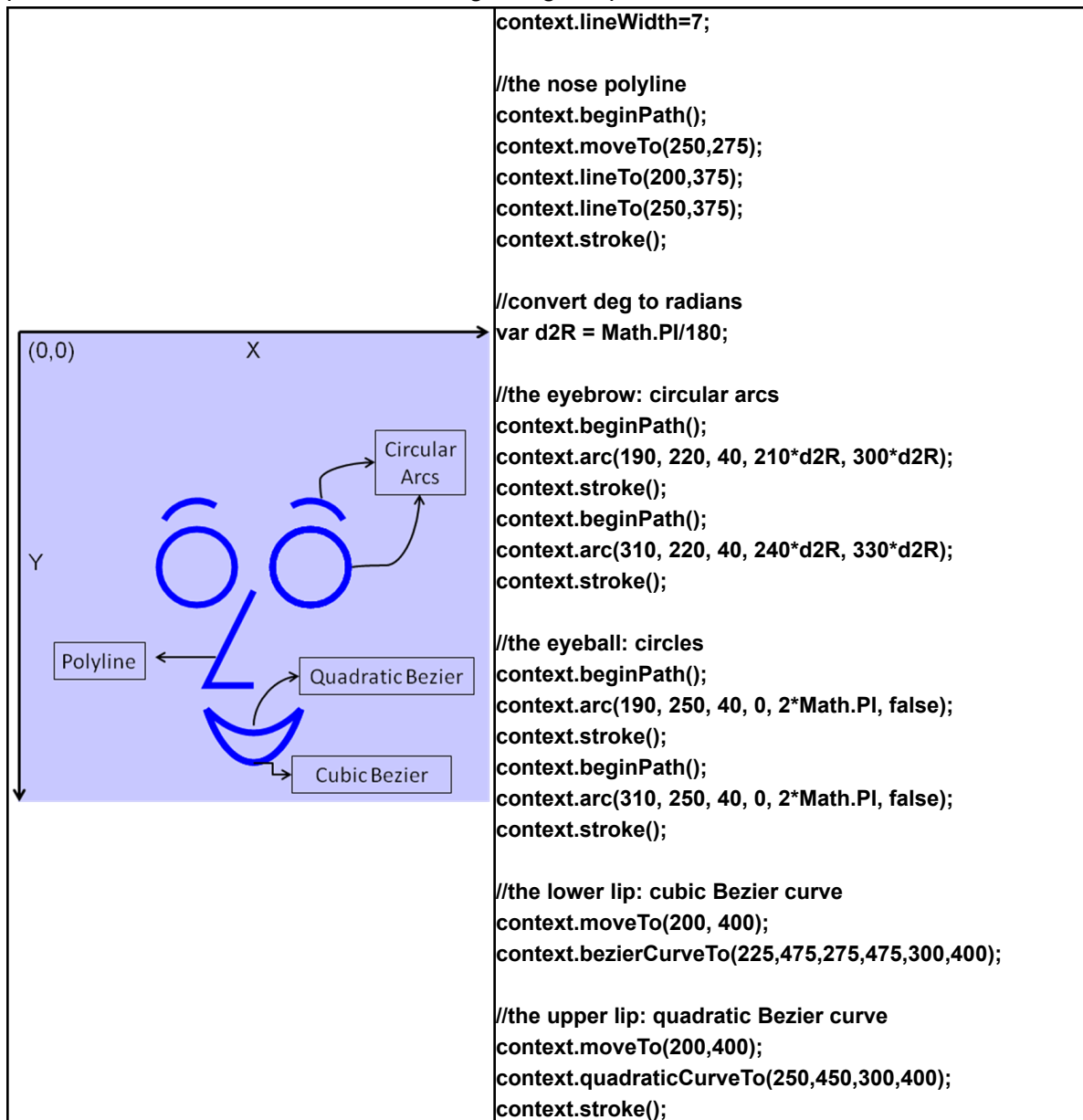
The HTML5 canvas uses the “immediate” mode of drawing: the drawing commands are executed immediately after being issued, and the system saves no information about what was just drawn. The only state of the canvas saved by the browser is the color of the pixels inside the canvas. Later, we contrast this with SVG, which uses the “declarative” or “retained” graphics mode. Unlike the canvas element, every SVG element can be referenced through the DOM and edited later on.

In the rest of this section, we will describe some of the functionality possible with the HTML5 Canvas API that is particularly relevant for graphics developers.

Drawing Tool

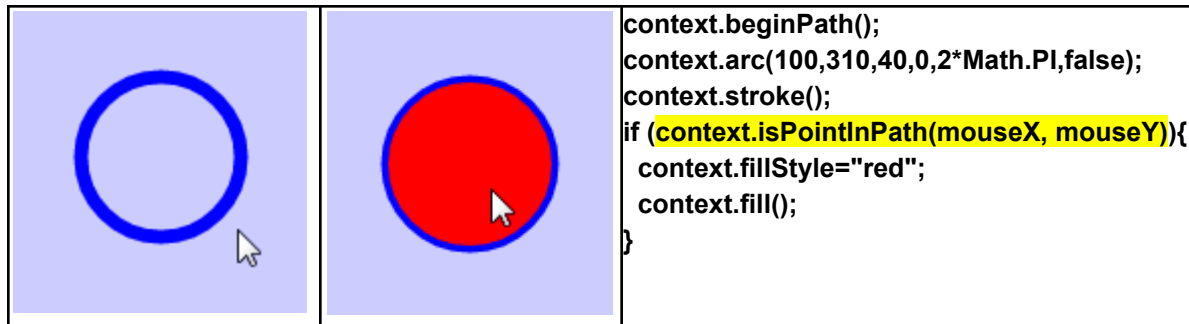
HTML5 introduced a quite powerful API for drawing and filling paths. Using this API, we can construct a vector design or sketching web application. See <http://mugtug.com/sketchpad/> for an example. If you are familiar with legacy drawing APIs like xLib or vector drawing APIs like PostScript, you will recognize the format of the path API. Similar to those APIs, we mimic the

pen and plotter interface where every new object is drawn by first lifting and moving the drawing pen to the start location and then tracing along the path to be drawn.



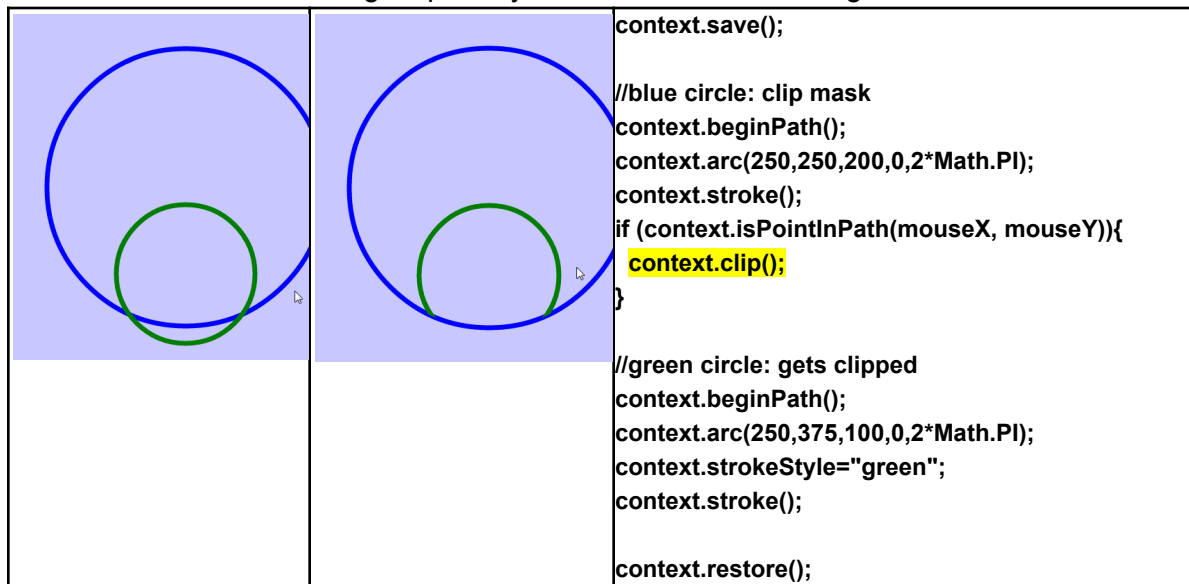
In the figure above, the code on the right produces the line drawing on the left for a 500x500 pixel canvas. This example demonstrates the ability to draw polylines, circular arcs (including full circles), cubic Bezier paths, and quadratic Bezier paths. Notice the use of the “beginPath()” function to indicate that a new path is being drawn for cases where calling “moveTo()” would be more complicated. Prior to “beginPath()” we must call the “stroke()” function to render the previous path.

The path API includes two geometric functions that are commonly needed for graphics tasks, so they are worth mentioning here.



Tracking when the mouse pointer enters a path by using the `isPointInPath()` function using the current mouse pointer coordinates. If the mouse is detected to be inside the path, we fill the path with a solid red color.

The “`isPointInPath(x,y)`” function returns true if the input (x,y) position is inside the path, assuming a non-zero winding number rule (i.e. same rule used for filling the path). This can be useful for intersection testing, especially for collision detection in games.



When the mouse pointer enters the blue path, we specify the drawing context to clip all subsequent drawing calls against the blue path. The clipping can be turned off by calling the context “`restore()`” function.

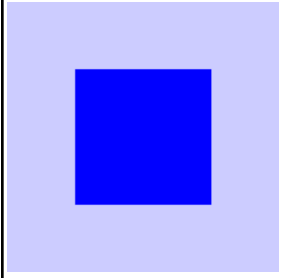
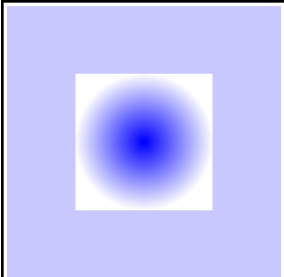
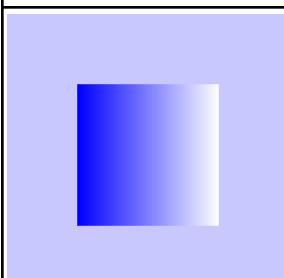
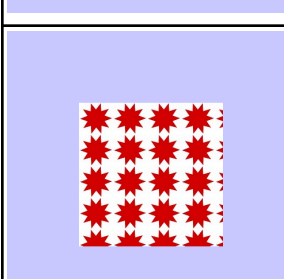
The “`clip()`” method is used to indicate that only the part of the canvas that’s inside the path will be rendered to the canvas. Make sure to include the “`save()`” function prior to calling the “`clip()`” method, so the clipping can be turned off by calling the corresponding “`restore()`” function.

Fill and Stroke Styles

Whatever shape has been added to the path so far will be filled when you issue the `fill()` command. Even open paths can be filled – for the purpose of the fill, the path is assumed to be

closed by connecting the last point to the first point. The fill rule for complex (self-intersecting) paths is the non-zero winding number rule – the region of the path that has a non-zero winding number is filled. Obviously, this is independent of the orientation of the path.

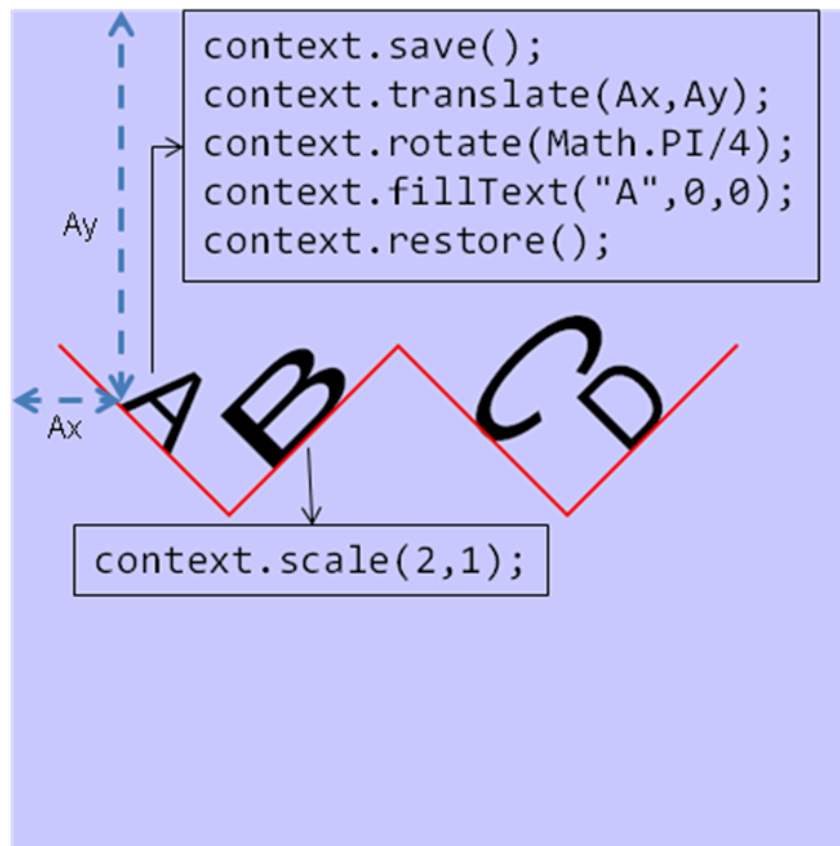
The shape can be filled with a solid color, a gradient, or a pattern (tiled images), as shown below:

	<pre>context.fillStyle="blue";</pre>
	<pre>var gradient = context.createRadialGradient(w/2,h/2,0, w/2,h/2,w/2); gradient.addColorStop(0, "blue"); gradient.addColorStop(1, "rgba(255,255,255,1.0)"); context.fillStyle = gradient;</pre>
	<pre>var gradient = context.createLinearGradient(w/4,h/2,w*0.75,h/2); gradient.addColorStop(0, "blue"); gradient.addColorStop(1, "rgba(255,255,255,1.0)"); context.fillStyle = gradient;</pre>
	<pre>var img = new Image(); img.src = 'star.jpg'; var myPattern = context.createPattern(img,'repeat'); context.fillStyle = myPattern;</pre>

The same rules that apply for “fillStyle()” also apply for “strokeStyle()”. That is, the stroke region for a path can be filled with a solid color, a gradient (radial or linear), or a repeating image pattern.

Layout and Transformations

Any shape drawn on the canvas can be transformed in order to position it anywhere within the canvas coordinate space. In this way the Canvas API can be used to layout 2D graphical elements on the screen.



Let us consider the task of typing the letters A, B, C, D along a zigzag path, as shown in the figure above. While we could use the path API to draw the letters, we'll take the simpler option of using the text api (i.e. the 'fillText()' or 'strokeText()' functions) built into the 2D drawing context.

The transformations affect the coordinate system of the drawing context. See the code block above for the letter a typical call to translate and rotate the letter A in order to place it in the proper position. Except in few cases, the order of the transformations is important. For example, calling the rotate function before the translate function would not have rotated the letter in place as above.

Experienced graphics developers will see the similarity between this transformation model and that present in prevalent graphics APIs like OpenGL or DirectX. Similar to those graphics API, we can concatenate the entire transformation and specify it as one homogeneous (3x3) matrix via the "transform()" and "setTransform()" functions in the Canvas API. Both functions take six arguments (the number of degrees of freedom available for 2D transforms). The difference between the "transform()" and "setTransform()" function is that the former concatenates the

specified transformation to the current transformation, while the latter sets the specified transformation as the only transformation (we lose the history of the previous transforms).

You may have noticed calls to “`save()`” and “`restore()`.” The “`save`” function pushes the current drawing context state (including the transformation, along with some other state variables) onto a stack. Subsequent calls to change the state (e.g. via additional transformations) will concatenate to the current state, but the original state can be recovered by the “`restore()`” function that will pop the top of the stack and set the current state to the popped off value. Again, experienced graphics developers will notice the similarity between “`save()`” and `glPushMatrix()` and “`restore()`” and `glPopMatrix()` in OpenGL.

Image Editing

The Canvas API allows random access to the byte-level RGBA values of the individual pixels within the coordinate space of the canvas. Therefore, we can set the colors of any part of the canvas on a pixel-by-pixel level. We can also load arbitrary images into the canvas and manipulate their pixel values. Given these functions, we can implement a comprehensive set of image editing features using the canvas API.

Here are some code snippets that you will need in order to perform any image editing functionality:

Loading and displaying images

```
var imageObject = new Image();
imageObject.src = "<path_to_image_file>";
imageObject.onload = function() {
    context.drawImage(imageObject,0,0);
}
```

In the above code, we first create an image object and specify the path to the actual image file. Also, “`context`” is the standard Canvas 2D context, the “`0,0`” in the `drawImage` call specifies the top left corner of the image (at the canvas origin in this case), and the image is drawn only when it is fully loaded (i.e. is in the `onload` event handler for the image object).

Access pixels

Now that the image is drawn on the canvas, we get a pointer to its pixels. At any time we can obtain a 1D array of pixels that contain the current color values of the canvas element by using the `getImageData` function. We can get all or a subset of the canvas pixels. The returned 1D array of pixels is ordered left to right, followed by top to bottom.

```
var imageData = context.getImageData(0,0,canvaswidth,canvasheight);
```

Changing pixels

Once we have access to the pixel array, we can modify its values. The following example converts the colored pixels into a grayscale representation.


```

var pixels = imageData.data;
for (var i = 0, n = pixels.length; i < n; i += 4) {
    //gray = 30% red + 59% green + 11% blue
    var gray = (pixels[i]*0.3) + (pixels[i+1]*0.59) + (pixels[i+2]*0.11);
    pixels[i] = gray;
    pixels[i+1] = gray;
    pixels[i+2] = gray;
}

```

Note that each pixel actually takes four spots in the pixel array (one for each of Red, Green, Blue, and Alpha values), which is why we increment our array iterator by 4.

Updating the canvas

After modifying the pixels, we need to update the image displayed by the canvas. We do so by replacing the current value of the pixels by the modified value:

```
context.putImageData(imageData, 0, 0);
```

As before, we can position the new pixels anywhere within the canvas, and in the example above, we have positioned it at the canvas origin.

A common image editing operation performed using the Canvas API is the implementation of image filters. See the following link for a demo. of some Canvas API image filters, including some convolution (sharpen, Laplace, etc.) filters:

<http://www.html5rocks.com/en/tutorials/canvas/imagefilters/>

If your image editing application is limited to filters where you will perform the *same* operation for every pixel (e.g. you wish to perform a fixed stencil convolution over the image), we recommend you limit the use of HTML5 Canvas only for prototyping, and use WebGL or CSS filters (described later) for the actual release code. The latter option will prevent the expensive Javascript loop over all the pixels, and can instead be executed in parallel as GPU shaders.

Animation

Remember that the HTML5 canvas operates in “immediate” mode, and does not save any information about the objects drawn on it. Given this immediate mode, animation is performed by modifying the position of the animated object and simply redrawing the region of the canvas that has changed. In most cases, the modified region includes the entire canvas.

<http://bomomo.com/>

<http://www.blobsallad.se/>

Performance Improvement

Traditionally, the code for animation using Javascript utilizes a timer, where some code for updating the position of elements in the browser is invoked at regular intervals (using the “setTimeout(<callback>, <timeout_interval>)” function call). The problem with that approach is that complex animations can slow down the browser and produce an undesirable user experience. The new, preferred method is to use the new requestAnimationFrame

function that indicates that we wish to animate the contents of the browser window at 60Hz (ideally) or as fast as possible for the browser (if not 60Hz). This allows the browser to optimize code for us, and also prevents the host computer from being unnecessarily slowed down due to our animation.

Therefore, we can use the following code structure (originally from <http://paulirish.com/2011/requestanimationframe-for-smart-animating/>):

```
//define the correct requestAnimationFrame function first
window.requestAnimationFrame = window.webkitRequestAnimationFrame ||
window.mozRequestAnimationFrame || window.msRequestAnimationFrame;

//invoke the requestAnimationFrame function in the render callback
function animateCanvas(time) {
    //specify that we wish to update the canvas at 60Hz
    window.requestAnimationFrame(animateCanvas, canvas);

    //display the entire canvas (includes both the changed and unchanged
    items)
    drawCanvas();
}
```

The following link describes the use of requestAnimationFrame and other hints for improving performance on HTML5 Canvas:

<http://www.html5rocks.com/en/tutorials/canvas/performance/>

Comparison with SVG

Some readers will have observed the similarity between the path drawing and filling capabilities of the HTML5 Canvas and those of the Scalable Vector Graphics (SVG) specification commonly implemented on modern web browsers. In many aspects, the HTML5 Canvas is similar to SVG. In this sub-section, we point out the differences between the two, especially those that are relevant for graphics programmers.

The main difference between the HTML5 Canvas and SVG is their rendering mode: HTML5 Canvas uses immediate mode, while SVG uses declarative mode. By using declarative mode, we can access individual SVG elements through the DOM and modify their properties (such as position, color, visibility) without needing to re-draw the drawing area. The re-draw is handled by the browser. In this sense, an SVG element is similar to an HTML element --- we can change the individual element properties dynamically and the web page is re-drawn automatically.

The declarative mode of SVG also includes another feature: grouping. Vector artwork can be

combined into one groups, several of which can be further combined into another group, and so on. Such a hierarchical organization of artwork in semantically relevant groups allows us to specify regions of influence of local transformations (useful for adding details to existing, animated vector artwork, for example).

Another difference between SVG and HTML5 Canvas is the use of filter effects in SVG. While SVG does not offer direct byte-level access to pixels like HTML5 Canvas does, a fixed set of image filters can be applied directly to the SVG elements, without needing to write them in (slower) Javascript. See this site for an example:

http://ie.microsoft.com/testdrive/Graphics/hands-on-css3/hands-on_svg-filter-effects.htm

We shall explore SVG filter effects in more detail in the next section on CSS3, when we describe CSS shaders.

CSS

Remember that after building the DOM tree, the browser performs a layout process. For the purpose of this course, think of the layout as the placement of rectangular regions (one for each HTML block element) within the 2D space of the browser window. As mentioned before, the rules used by the browser for displaying each visual element (e.g. the background color of the rectangular region) are provided by the CSS style associated with that element.

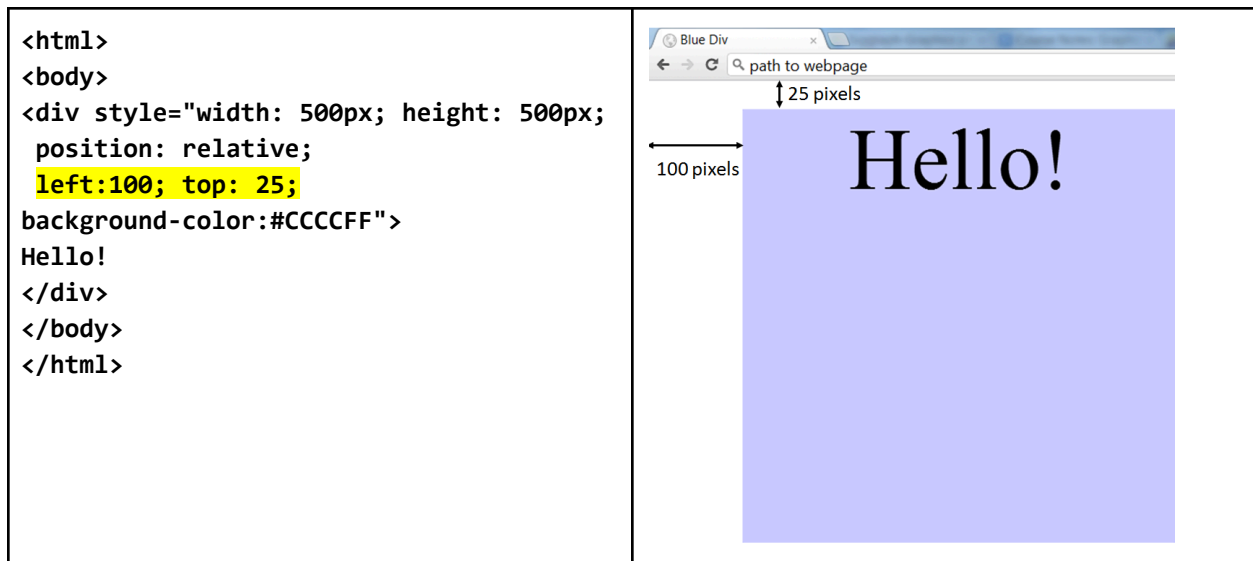
In this section, we describe some of the new features in CSS that are relevant for graphics programmers. These new features significantly improve the ability to dynamically change the display styles of HTML elements, most notably the position and orientation of those elements. CSS allows us to easily produce animations without needing any Javascript code to update positions programmatically. Since this transfers the animation functionality from non-native (Javascript) code to native and carefully optimized (browser) byte-code, it usually produces significant performance improvements.

Similar to the HTML5 Canvas section, we introduce the graphics capabilities of CSS by showing how CSS can be used to implement functionality commonly needed by graphics applications. We shall consider two applications: planes in space, and image filters.

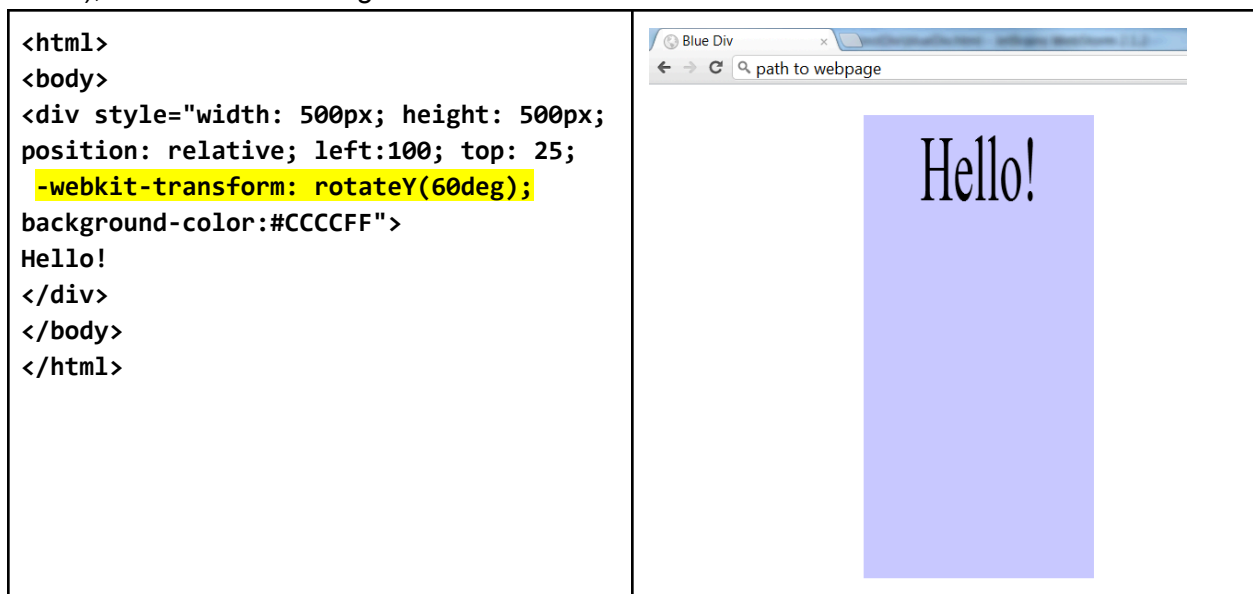
Planes in Space

The term “planes in space” refers to the ability to place 2D planar polygons at any position and orientation in 3D space. Previously we could place any HTML block element at any position within the 2D window. Typically this was done by changing the “left” and “top” style attributes for that element.

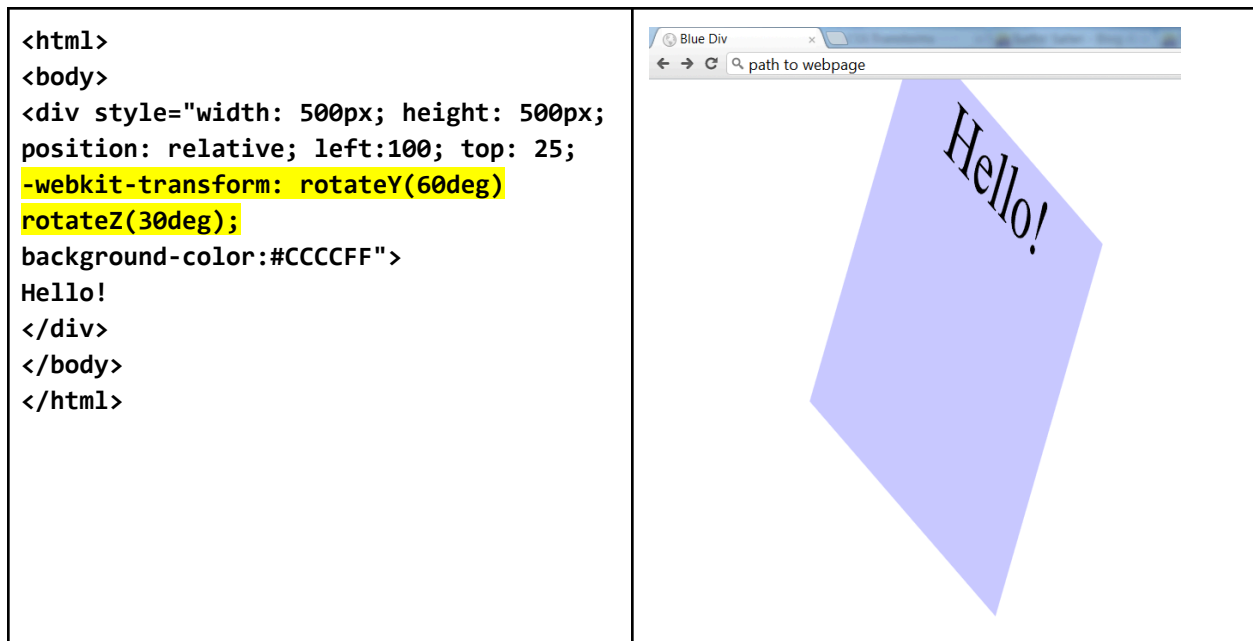
Consider the task of placing a `<div>` element at a horizontal distance 100 pixels and a vertical distance of 25 pixels from the top-left corner of the browser window. We use the following code:



We can add a 3D effect to the 2D element by making it rotate by 60 degrees about the vertical (Y) axis. If we use a browser using the Webkit layout engine (e.g. Google Chrome, Apple Safari), we use the following code:



The transforms can be concatenated:



We can specify most common types of 3D transformations (rotate, translate, scale, skew, and perspective) on any block element. We can even specify directly the 4x4 transformation matrix to be applied to the element. See <http://dev.w3.org/csswg/css3-transforms/> for the API currently proposed for CSS transforms. Also see <http://desandro.github.com/3dtransforms/> for an excellent explanation of the CSS transforms in depth.

The benefit of applying 3D transforms to a block element is that after transforming the element, the browser continues to interact with contents of the element as before. Text in the element remains selectable, links still work, and videos or images are displayed with correct transformations. This makes the CSS3 transforms very useful for creating 3D interactive user interfaces. The links below give some examples:

A simple image flip on mouse over:

<http://www.webkit.org/blog-files/3d-transforms/image-flip.html>

More advanced image effects using CSS:

<https://developer.mozilla.org/en-US/demos/detail/3d-image-transitions/launch>

Placing webpage elements in 3D space:

<http://www.webkit.org/blog-files/3d-transforms/morphing-cubes.html>

Note: You may be wondering why we needed to use the prefix “-webkit-” for style attributes like transform. The newer CSS styles are not yet finalized by the W3C. In order to allow browsers to support the non-finalized feature or an incomplete implementation of the feature, individual browsers support the prefixed forms of the style attribute. Browsers with the Webkit

layout engine (e.g. Google Chrome, Apple Safari) will respect the transform attribute with the “-webkit-” prefix. Similarly, Mozilla-based Firefox needs the “-moz-” prefix, Internet Explorer the “-ms-” prefix, and Opera the “-o-” prefix. This is a temporary solution until the CSS specification is finalized by the W3C and universally adopted by all browsers. Until then, you need to specify all the prefixes for the new style attributes so your code may run on all browsers. You can use tools like “Prefix free” (<http://leaverou.github.com/prefixfree/>) to produce the prefixed versions of the CSS attributes automatically at script runtime.

Animations

In the demos above, elements placed by CSS are animated. We can control the manner in which they move from one position to the other. A simple method to bring about animation is via CSS transitions. For example, our example above for rotating the <div> element can be animated by applying a transition on the transform style attribute. The transform style attribute itself is modified when the user clicks on the element.

```
<html>
<body>
<div style="width: 500px; height: 500px; position: relative; left:100; top: 25;
background-color:#CCCCFF;
-webkit-transition: -webkit-transform 3s ease-in;"
onclick="this.style.webkitTransform='rotateY(60deg) rotateZ(30deg)'">
Hello!
</div>
</body>
</html>
```

In the above example, we set the style of the “-webkit-transition” to be “-webkit-transform 3s ease-in”. As you can guess, this means that we are adding a transition to the -webkit-transform attribute that is 3 seconds long, and we ease-in from the old value to the new one. We can specify transitions on all/none/some of the style attributes of the element, for any duration, and with different timing functions (ease-in, ease-out, linear, etc.). The complete specification for the transitions is provided here:

<http://www.w3.org/TR/css3-transitions/>.

If animations using transitions are not sufficient, we can also create explicit animation of the style attributes using keyframes.

```
<html>
<head>
<style type="text/css">
@-webkit-keyframes wobble {
    0% {
```

```

        -webkit-transform: scale3d(1,1,1) rotateZ(0deg);
    }
    33% {
        -webkit-transform: scale3d(1.2,1.2,1) rotateZ(30deg);
    }
    66% {
        -webkit-transform: scale3d(1.2,1.2,1) rotateZ(-30deg);
    }

    100% {
        -webkit-transform: scale3d(1,1,1) rotateZ(-deg);
    }
}
</style>
</head>
<body>
<div style="width: 50px; height: 50px; position: relative; left:100; top: 25;
background-color:#CCCCFF;
-webkit-animation-name: wobble;
-webkit-animation-duration: 3s;
-webkit-animation-iteration-count: infinite;
-webkit-animation-direction: alternate;
-webkit-animation-timing-function: linear;">
Hello!
</div>
</body>
</html>

```

In the above example, we first create some keyframes (at 0%, 33%, 66% and 100% of the animation duration). At each keyframe, we specify values of the style attributes that need to change. The animation keyframes are then specified as the value of the animation-name attribute, along with the duration of each cycle, the number of complete cycles (infinite for continuous looping animation), whether to step forwards, backwards or forward and backward, and the timing function (ease-in, ease-out, linear, etc.). The complete specifications for CSS animations is available here: <http://www.w3.org/TR/css3-animations/>.

A final note on the “planes in space” functionality is that soon browsers will be able to support *curved* planes in space for placing block tags. Curved planes will be possible through the use of CSS shaders, which are described in the next section.

More resources

General-purpose CSS animation: <http://tinyurl.com/csswalk>

Traditional animation principles implemented as CSS:

<http://coding.smashingmagazine.com/2011/09/14/the-guide-to-css-animation-principles-and-examples/>

Image Filters

Similar to the HTML5 Canvas, we can implement image filters using CSS. The image filters in CSS (called “filter effects”) are essentially the same as those for SVG: a fixed set of image filters applied to every pixel of the image. The proposed specification for CSS filter effects is <https://dvcs.w3.org/hg/FXTF/raw-file/tip/filters/index.html>.

Like SVG, CSS filter effects support some hard-coded filter functions (e.g. “brighten”, “sepia”, “grayscale”) that accept a few user parameters. The syntax is as simple as specifying the filter attribute, like:

```
-webkit-filter: blur(2px) grayscale(1);
```

Note that multiple filter functions can be specified, and the order of the filter functions is important.

The following link shows all the filters in action:

<http://html5-demos.appspot.com/static/css/filters/index.html>

Similar to SVG filters, the benefit of using filter effects through CSS instead of HTML5 Canvas is improved performance: we do not need to iterate over every pixel in the image using Javascript, and can exploit the parallel computation ability of modern GPUs. For this reason, if you wish to use one of the filter effects in the specifications and CSS filter effects are supported by your target browser, we recommend you use CSS filter effects instead of Javascript-coded HTML5 Canvas image manipulation.

General-Purpose CSS Shaders

The exciting development of CSS filter effects is that in addition to the fixed filters, the CSS filter effects specification allows us to specify a *custom* shader as a filter. This is exciting because it has the potential to open up the massive parallel computation available on most GPUs for general-purpose computing via simple shader programs. These general-purpose programs need not have anything to do with graphics, and the CSS shaders can be used for large-scale parallel computation. Since this is a rather new specification, browser implementations that support general purpose CSS shaders are not yet available (at the time of this writing). This article gives a more in-depth description of CSS shaders:

<http://www.adobe.com/devnet/html5/articles/css-shaders.html>

As the above link illustrates, with CSS shaders we can apply vertex shaders to the grid of polygons that is overlaid on the HTML element. The contents of the HTML element are rendered as a texture map (filtered appropriately) onto grid. Through these shaders, we can specify a filter that changes the position of the grid vertices, thereby creating *curved* planes in

space, as was mentioned before.