CONTENTS INCLUDE:

HTML 5 Canvas

A Web Standard for Dynamic Graphics

By Simon Sarris

INTRODUCTION TO CANVAS

The HTML <canvas> element allows for on-the-fly creation of graphs, diagrams, games, and other visual elements and interactive media. It also allows for the rendering of 2D and 3D shapes and images, typically via JavaScript.

```
<canvas id="canvas1" width="500" height="500"></canvas>

<script type="text/javascript">
var can = document.getElementById('canvas1');
var ctx = can.getContext('2d');

ctx.fillText("Hello World!", 50, 50);
</script>
```

Canvas is perhaps the most visible part of the new HTML5 feature set, with new demos, projects, and proofs of concept appearing daily.

Canvas is a very low-level drawing surface with commands for making lines, curves, rectangles, gradients and clipping regions built in. There is very little else in the way of graphics drawing, which allows programmers to create their own methods for several basic drawing functions such as blurring, tweening, and animation. Even drawing a dotted line is something that must be done by the programmer from scratch.

Canvas is an immediate drawing surface and has no scene graph. This means that once an image or shape is drawn to it, neither the Canvas nor its drawing context have any knowledge of what was just drawn.

For instance, to draw a line and have it move around, you need to do much more than simply change the points of the line. You must clear the Canvas (or part of it) and redraw the line with the new points. This contrasts greatly with SVG, where you would simply give the line a new position and be done with it.

Hot Tip You can visit the evolving specification for Canvas at the WHATWG site: http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html.

Browser Support and Hardware Acceleration

Canvas is supported by Firefox 1.5 and later; Opera 9 and later; and newer versions of Safari, Chrome, and Internet Explorer 9 and 10.

The latest versions of these browsers support nearly all abilities of the Canvas element. A notable exception is drawFocusRing, which no browser supports effects.

Hardware acceleration is supported in some variation by all current browsers, though the performance gains differ. It is difficult to benchmark between the modern browsers because they are changing frequently, but so far IE9 seems to consistently get the most out of having a good GPU. On a machine with a good video card it is almost always the fastest at rendering massive amounts of images or canvas-to-canvas draws.

Accelerated IE9 also renders fillRect more than twice as fast as the other major browsers, allowing for impressive 2D particle effects [1]. Chrome often has the fastest path rendering but can be inconsistent between releases. All browsers render images and rects much faster than paths or text, so it is best to use images and rects if you can regardless of which browsers you are targeting.

	Canvas	SVG
Support	• All modern versions of Chrome, Safari, Firefox, and Opera have at least some support. Internet Explorer 9+ has support. Almost all modern smart phones. • Internet Explorer 7 and 8 have limited support through the excanvas library. • Rapidly growing in popularity	• SVG support in all modern browsers. Almost all modern smart phones.
Statefulness	• Bitmapped, immediate drawing surface • Shapes are drawn and nothing is remembered about their state.	• Vector-based, retained drawing surface • Every drawn shape is a DOM object.
Other Considerations	• Generally faster • All event handling and statefulness must be programmed yourself. • Canvas will be effectively disabled (rendering nothing) if scripting is disabled.	• Generally slower, especially past 10,000 objects. • Since all SVG elements are DOM objects, statefulness is built in and event handling is much easier. • Easier for a designer to work with, many programs such as Illustrator can output SVG • SVG has built-in support for animation.
Accessibility	• Difficult to interface with other DOM objects • Working with text can be difficult. • Recreating text-based DOM element functionality is strongly advised against, even in the specification itself. • Cannot operate when scripting is disabled.	• All SVG objects are already DOM objects. • Text is searchable by the browser and web crawlers.

Best suited for	• Games, fast graphics • Very large amounts of objects • Creating high-performance content	• Accessibility oriented content or text-oriented content • Easily scalable shapes • Charts, graphs, and other mostly static data displays • Creating interactive content quickly.

What Canvas Can and Cannot Do

The specification advises against using Canvas to render static content. There are many reasons to not use Canvas if typical image and text elements will suffice. If scripting is disabled on the client, the Canvas will be useless. Text drawn on Canvas is not selectable, searchable, or crawlable by web spiders. For the same reason, Canvas makes web accessibility more difficult. If you are looking to simply stylize text or round off the edges of a text area, you should see if the desired effects (such as shadows or rounded corners) are possible with CSS3 before opting to use a Canvas.

As of August 2011, Canvas does not look the same on all browsers. The implementations of anti-aliasing (or not) differ, and other quirks can cause gradients, text, and scaled objects to look dissimilar. For instance, until a few months ago, Chrome's handling of gradients would disregard opacity and take the last-known non-gradient color's opacity instead! Both the specification and implementations of Canvas should be considered slightly, yet constantly, evolving.

A Comparison with SVG

Canvas is a very flexible drawing surface but may not be appropriate for all projects. Most immediately relevant when planning a web-app is browser support. Internet Explorer 7 and 8 only support Canvas through the excanvas library. Excanvas performance degrades very quickly with animation, so any animated web-apps that must target Internet Explorer 7 and 8 should not use canvas. Note that excanvas is also no longer under active development.

The other large difference is the statefulness of each. Canvas is an immediate drawing surface whereas SVG is retained, meaning that the DOM remembers every drawn SVG element and each element has a fully-defined DOM object associated with it, event handlers included. This makes implementing interactivity with SVG much easier than Canvas, but it also introduces a large amount of overhead that is unsuitable for performance-needing applications.

The bottom line is that SVG is easier to program for from the get-go, but Canvas is more powerful. The decision to use either should rest upon what platforms you are targeting, how much performance will be needed, what libraries you wish to use, and ease of development based on your (team's) current knowledge and skillset.

For an in-depth comparison, see the IE team's blog post: http://blogs.msdn.com/b/ie/archive/2011/04/22/thoughts-on-when-to-use-canvas-and-svg.aspx

Canvas Performance

While the low-level of Canvas might make development slower, it outshines the other options when performance is crucial, especially when there are tens of thousands of objects to load and draw. While having each SVG object be a DOM object makes events and object modification easier to code, the overhead involved makes SVG unsuitable for complex, interactive apps. Creating 10,000 shapes in Canvas is a very fast process, while creating the same shapes in SVG means creating tens of thousands of SVG DOM nodes, resulting in a much slower process.

Canvas can be very fast, but it is up to the programmer to keep it that way. Many of the optimizations that might be taken care of by more advanced drawing frameworks must be done by the programmer. Speed becomes important, and any serious Canvas developer should familiarize himself with the typical concepts of graphics performance, such as invalidations and viewports.

Additionally, different drawing operations are faster or slower than others, and different methods of accomplishing the same task can take wildly different times. I will go over a few of them in the final section.

CREATING A CANVAS

The tag syntax for a Canvas is as follows:

```
<canvas id="canvas1" width="500" height="500">
This text is displayed if you do not have a canvas-capable browser.
</canvas>
```

Note how "width" and "height" are attributes just like "id". The CSS width and height are distinct and are not used for sizing the Canvas.

Canvas Attributes

Name/Method	Description
width	Default 300. Also a tag attribute, sets the width of the Canvas in pixels.
height	Default 150. Also a tag attribute, sets the height of the Canvas in pixels.
toDataURL([type, ...])	Returns a data:URL for the image in the Canvas. The first optional argument controls the type of the image to be returned (e.g., PNG or JPEG). The default is image/png; that type is also used if the given type isn't supported. The other arguments are specific to the type, and control the way that the image is generated, as given in the table below.
toBlob(callback [, type, ...])	Creates a Blob object representing a file containing the image in the Canvas and invokes a callback with a handle to that object. The second optional argument controls the type of the image to be returned (e.g., PNG or JPEG). The default is image/png; that type is also used if the given type isn't supported. The other arguments are specific to the type, and control the way that the image is generated, as given in the table below. This is an extremely new (May 2011) method for getting the contents of a Canvas. Note that it is also asynchronous.
getContext(in DOMString contextID, in any... args)	Returns an object that exposes an API for drawing on the Canvas. The first argument specifies the desired API. Subsequent arguments are handled by that API. Returns null if the given context ID is not supported or if the Canvas has already been initialized with some other (incompatible) context type (e.g. trying to get a "2d" context after getting a "webgl" context).

It is important to note that unlike many HTML Elements, width and height are attributes of the Canvas element itself and not style attributes. Setting the style width and height of the Canvas will scale the Canvas instead.

State and Transformations

Method	Description
save()	Pushes the current context state onto the stack.
restore()	Pops the top state on the stack, restoring the context to that state.
scale(x, y)	Applies a scaling transformation to the current transformation matrix
rotate(angle)	Applies a rotation transformation to the current transformation matrix. The angle is in radians.
transform(m0, m1, m2, m3, m4, m5)	Applies the supplied transformation matrix to the current transformation matrix.
setTransform(m0, m1, m2, m3, m4, m5)	Replaces the current transformation matrix with the given transformation matrix.

The transformation matrix is an important part of drawing complex shapes on Canvas, though complete understanding of how one works is not immediately necessary. Using the rotate, scale, and translate methods will modify the matrix. It can be modified directly (but never retrieved) with transform and setTransform.

To not interfere with objects drawn after such transformations, save and restore are typically called before and after each transformed element is drawn to the Canvas.

While Canvas does not remember shapes and images drawn, it does keep track of several drawing rules that comprise its state. The methods save and restore will push or pop the state onto or off of a stack, saving and restoring not only the transformation matrix, but also the current clipping region, as well as all of the stateful attributes:

• strokeStyle	• lineWidth	• miterLimit
• fillStyle	• lineCap	• shadowOffsetX
• globalAlpha	• lineJoin	• shadowOffsetY
• shadowBlur	• shadowColor	• globalCompositeOperation
• font	• textAlign	• textBaseline

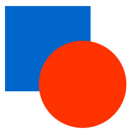
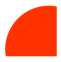
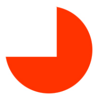
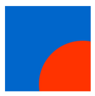
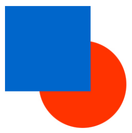
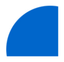
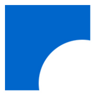
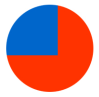
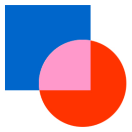
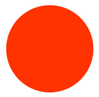
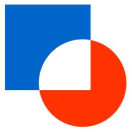
Compositing

Property	Description
globalAlpha	Gets or sets the alpha value applied to all drawing operations. Default is 1.0
globalCompositeOp-eration	Gets or sets the current composite operation. Default is "source-over".

All drawing operations are affected by the two compositing attributes. In the following images, a blue square is drawn to represent shapes already existing on a canvas, then the globalCompositeOperation is set to the specified value and a red circle is drawn.

In the following, “existing content” is defined as any pixels that were already drawn and not previously transparent.

globalCompositeOperation

Result	Description
	**source-over** The default. New content is drawn over existing content.
	**source-in** New content is only drawn where existing content was non-transparent.
	**source-out** New content is drawn only where there was transparency.
	**source-atop** New content is drawn only where its overlap existing content.
	**destination-over** Opposite of source-over. It acts as if new content is drawn “behind” existing content.
	**destination-in** Opposite of source-in. Existing content is drawn only where new content is non-transparent.
	**destination-out** Opposite of source-out. Existing content is drawn only where new content is transparent. Acts as if existing content is drawn everywhere except the where the new content is.
	**destination-atop** Opposite of source-atop. New content is drawn, and then old content is drawn only where it overlaps with new content.
	**lighter** Where new content overlaps old content, color is determined by adding the color values.
	**copy** New content replaces all old content.
	**xor** New content is drawn where old content is transparent. Where the content of both old and new are not transparent, transparency is drawn instead.

Colors and Styles

Method/Property	Description
strokeStyle	Gets or sets the current style used for stroking shapes. Can be a string, CanvasGradient or CanvasPattern. The string must be parsed as a CSS color value. Once set, changes to the CanvasGradient or CanvasPattern that was used will reflect upon newly drawn content. Default is “#00000”
fillStyle	Gets or sets the current style used for filling shapes. Follows the same rules as fillStyle. Default is "#00000".

The CanvasGradient interface defines the methods for creating linear and radial gradients. Once a gradient is created, stops are placed along it to define the color distribution. With no stops, the gradient is simply black.

CanvasGradient Methods

Method/Property	Description
addColorStop(offset, color)	Adds a color stop to the gradient at the given offset. The offset goes from 0 to 1. The color is a string representation of a valid CSS color value.
context.	Creates and returns a CanvasGradient object that represents a linear gradient that paints along the coordinates given. Is called on an instance of a Canvas2DContext.

CanvasPattern Methods

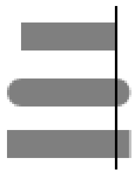
Method/Property	Description
context. createPattern(image, repetition)	Creates and returns a CanvasPattern object that uses the given image and repeats in the directions defined by the repetition argument. The first argument must be an Image, a Canvas, or a Video element. The allowed values for the second argument are the strings "repeat", "repeat-x", "repeat-y", and "no-repeat". The default is "repeat". Like the gradients, this method is called on an instance of a Canvas2DContext.

Note that the CanvasGradient and CanvasPattern are distinct objects from the Canvas, but these gradients and patterns can only be created using an instance of the Canvas2DContext.Example syntax:

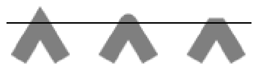
```
// creating a gradient that will display black to white along the linear path
from (0,0) to (0,150)
// where context is an instance of Canvas2DContext
var gradient =  context.createLinearGradient(0,0,0,150);
gradient.addColorStop(0, '#000000');
gradient.addColorStop(1, '#FFFFFF');
ctx.fillStyle = gradient; // assignment to the context's fillStyle
```

Line Styles

Method/Property	Description
lineWidth	Gets or sets the width of lines to be drawn. Default 1.0.
lineCap	Gets or sets how the end of lines are to be drawn. Valid values are "butt", "round", and "square". Default "butt".
lineJoin	Gets or sets how corners are drawn when two lines meet. Valid values are "bevel", "round", and "miter". Default "miter".
miterLimit	Gets or sets the current miter limit ratio. Default 10.0.



Three gray lines drawn to the thin black line. From top to bottom, the lineCap property of each is "butt", "round", and "square".

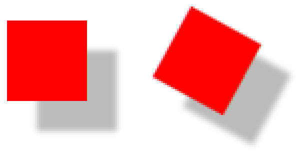


Three paths, each drawn up to the thin black line and back down. From left to right, the lineJoin property of each is "miter", "round", and "bevel".

Shadows

Method/Property	Description
shadowColor	Gets or sets the color of the shadow. Accepts any valid CSS color string. Default is transparent black.
shadowOffsetX	Gets or sets the X offset. Pushes the shadow farther to the right. Default 0.
shadowOffsetY	Gets or sets the Y offset. Default 0.
shadowBlur	Gets or sets the level of blurring effect. The lower the value, the sharper the edge of the shadow. Default 0.

Shadows are "smart", emulating the drawn pixels precisely so that if text is drawn, the shadow will look just as the text does. Additionally, shadows are not affected by the transformation matrix.



A square drawn with a shadowOffsetX and shadowOffsetY of 15. The shadow is drawn evenly 15 pixels offset in both axes. To the right is the same square drawn with the same shadowOffsetX and Y, but the context was rotated about the square's center.

This means that if you were to rotate the square 180 degrees, the shadow would be in the exact same place as if the square were not rotated at all. There is an advantage to having shadows drawn unaffected by the transformation matrix. If you draw several objects, some rotated and some not, you typically want the shadows to all stay in the same direction, giving the proper illusion of a light-source. If the shadows were affected by the transformation matrix, it would look as if there is no singular light source but shadows going in every direction!

Paths

Method/Property	Description
beginPath()	Erases all current subpaths in preparation for drawing a new path.
closePath()	Closes the subpath by drawing a straight line from the current point to the initial point.
moveTo(x, y)	Creates a new subpath at the given point. Typically used to place the starting point, or to draw unconnected paths.
lineTo(x, y)	Draw a line to the given point.
quadraticCurveTo(cpx, cpy, x, y)	Draw a quadratic curve with the given control point (cpx, cpy) to the given point (x, y)
bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)	Draw a bezier curve described by the two given sets of control points and a given end point.
arcTo(x1, y1, x2, y2, radius)	Draw an arc with the given control points and radius, connected to the previous point via a straight line.
arc(in double x, y, radius, startAngle, endAngle, [anticlockwise])	Draw an arc described by the circumference of the circle described by the given arguments, starting at the startAngle and ending at the endAngle , going in the given direction (defaulting to clockwise if the last argument is left blank)
rect(x, y, w, h)	Add the closed subpath in the shape of a rectangle.
fill()	Fills all described subpaths with the current fillStyle.
stroke()	Strokes all described subpaths with the current strokeStyle.

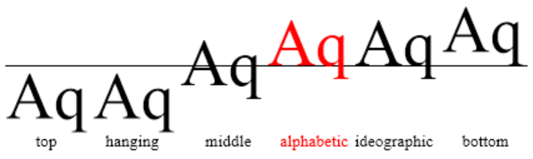
Filling a path will fill all of the subpaths of the current path. It fills them according to the non-zero winding number rule. When a fill is called, all open subpaths get implicitly closed.

Text and Images

Method/Property	Description
font	Gets or sets the current font settings. Must be a string that will be parsed as a CSS font property.
textAlign	Gets or sets the text alignment. Possible values are "start", "end", "left", "right", and "center". Default is "start".
textBaseline	Gets or sets the baseline setting. Can be "top", "hanging", "middle", "alphabetic", "ideographic", or "bottom". Default is "alphabetic".
fillText(text, x, y, [, maxWidth])	Fills the given text at a given position using the fillStyle. Note that the position at the textBaseline.
strokeText(text, x, y, [, maxWidth])	Strokes the text at a given position using the strokeStyle.
drawImage(image, dx, dy)	Draws a given image to the context at the given position. See below.
drawImage(image, dx, dy, dw, dh)	Draws a given image to the context at the given position, scaling the image to the given width and height.

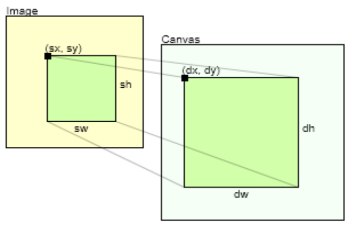
drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)	Draws a section of a given image that is described by the point and size given by (sx, sy, sw, sh), which is in turn drawn to the context at the point and size given by (dx, dy, dw, dh).

Drawing and measuring text are among some of the slowest operations on Canvas, so it is among the first places you should start fiddling around for more performance. It can often be the case that storing the result of a drawText in a PNG or a separate in-memory Canvas will improve performance.



The six different baselines, all drawn on the same Y-value, represented by the black line. "Alphabetic" shown in red is the default.

All three drawImage functions take either an image, Canvas, or video element. Even if you never think of your app asdrawing any images onto your Canvas, these methods can come in very handy. For performance reasons, you may find yourself making Canvases in memory and drawing portions of your screen to them or vice versa. If you want a miniature overlay above your complex Canvas, you can call drawImage on your Canvas context and pass in its own Canvas to create such a mini-map.



The optional arguments to drawImage allow us to paint a portion of an image onto the canvas and force it to scale into the width and height dw and dh.

Pixel Manipulation

Method/Property	Description
createImageData(sw, sh)	Returns an ImageData object with the given width and height. The ImageData's pixels are filled with transparent black.
createImageData(imagedata)	Returns an ImageData object with the same width and height as the given ImageData. All of the returned ImageData's pixels are transparent black.
putImageData(imagedata, dx, dy,[, dirtyX, dirtyY, dirtyWidth, dirtyHeight])	Paints a given ImageData onto the Canvas. If a "dirty" rectangle is provided, only the pixels from that rectangle are painted.

The pixels painted by putImageData are precise; globalAlpha and globalCompositeOperation are not taken into account during their painting. Because of this, you do not want to use putImageData to paint part of a Canvas to another Canvas as the transparent region will be carried over, clobbering whatever was there before.

Another reason to always consider drawImage before using putImageData is the performance difference. Working with ImageData is very slow compared to calls to drawImage, so only use it if you absolutely need to do per-pixel operations (such as making an eyedropper tool).

Also note that the optional "dirty" rectangle arguments for putImageData are not implemented on all browsers. If you use it, make sure all browsers targeted have the functionality.

ImageData Properties

Property	Description
width	Gets the width of the ImageData object.
height	Gets the height of the ImageData object.
data	Returns a one-dimensional array containing the data in RGBA order, as integers in the range 0 to 255.

The ImageData.data property returns an array where each pixel is represented by four indices. The first pixel's R, G, B, and Alpha values are thus indicated by data[0], data[1], data[2], data[3]. The second pixel is indicated by the indices 4 to 7 and so on. The order of the pixels is from left to right, top to bottom, just like reading text in left-to-right languages.

TIPS FOR YOUR CANVAS APP

Web developers are no strangers to quirks, and Canvas has its own laundry list of idiosyncrasies. Below are a few things to keep in mind as you make Canvas apps.

- There are several valid ways to clear a Canvas, but some are much faster than others. Below are three common ways with their differences described:

```
can.width = can.width;
```

Setting the Canvas' width attribute equal to itself will not only clear the Canvas but also clear the entire state (see State and transformations section for what this entails). This is useful if you want a full reset, but it is typically slow.

```
ctx.clearRect(0, 0, can.width, can.height);
```

The above merely clears the pixels on the screen; but if the transformation is not identity, it may not work as intended.

```
ctx.save();
ctx.setTransform(1, 0, 0, 1, 0, 0);
ctx.clearRect(0, 0, can.width, can.height);
ctx.restore();
```

This is a good way to clear the Canvas while keeping all the state data intact. The performance difference between the use of clearRect and setting Canvas width equal to itself varies wildly between browsers. You should benchmark often on the platforms you are targeting.

- There are different ways of doing operations that have different performance effects on different browsers. For example, in the Canvas-clearing example above, it is always faster to not set Canvas width equal to itself with the lone exception of Safari 5, in which that method is two orders of magnitude faster. If you are targeting a specific browser, especially a phone or tablet, you should be sure to tailor your performance computations around it.

- In a draw loop, it is often beneficial to try and draw only the objects that have changed and (perhaps) any objects that intersect their bounds. However, depending on the application, this is not always the case. If nothing in the scene has changed, it is of course always the case that skipping out on drawing entirely is much faster.

- Use requestAnimationFrame on browsers that support it. It will ensure that animation is not occurring on non-active tabs with Canvas elements, which can save battery life on phones and tablets.

- Draw and hit test only what is on the screen.

- Hit testing only what is on the screen.

- If you are to draw the same background every time, make the background a PNG and set the Canvas' style's "background-image" attribute.

- Depending on the app, using multiple Canvases atop each other (for background, foreground, and middle ground) can increase performance.

- Keep as much out of the draw loop as possible and touch DOM objects as little as possible.

- Minimize the setting of styles. If you know that you are to draw 400,000 blue objects, set the fillStyle to blue only once at the beginning instead of before each object.

- Always draw images on integer coordinates. If you draw on non-integer coordinates the browser will have to interpolate the image (sub-pixel anti-aliasing the whole thing), which is often slower and can sometimes look quite different. A quick way to make your javascript vars into integers is to bitwise OR your vars with zero, like so:

```
ctx.drawImage(yourImage, x | 0, y | 0);
```

- Drawing images is almost always faster than drawing text or paths. If you have a repeatedly drawn string or shape that never changes and does not need to scale, consider making it into an image.

- This will "floor" any 32-bit integer quickly. Beware that when you bitwise-OR zero a number larger than a 32-bit integer, meaning 2147483648 or higher, you will get junk.

[1]. Any performance notes were based on browsers tested in the set from Firefox 3.6-7.0, Opera 11.x, IE9, Chrome 12-16 (beta versions), and Safari 5.x. Some of these varied wildly as they progressed. To get up to date performance stats I suggest creating your own tests using www.jsperf.com. Remember that simply reading the aggregate results from jsperf are not useful for cross-browser comparison unless the same hardware is used in all cases.

ABOUT THE AUTHOR

Simon Sarris is a core developer for web development at Northwoods Software. At Northwoods he is developing a fully-featured diagramming library for HTML5 Canvas. In his spare time he is an avid game programmer and hobbyist. His passion for web development is echoed in his game development projects as well as his Canvas articles and tutorials. He is among the top contributors on StackOverflow for Canvas. Recent publications are on his blog (simonsarris.com).

RECOMMENDED BOOK

No matter what platform or tools you use, the HTML5 revolution will soon change the way you build web applications, if it hasn't already. HTML5 is jam-packed with features, and there's a lot to learn. This book gets you started with the Canvas element, perhaps HTML5's most exciting feature. Learn how to build interactive multimedia applications using this element to draw, render text, manipulate images, and create animation.

ISBN-13: 978-1-936502-40-0
ISBN-10: 1-936502-40-2
50795
9 781936 502400
$7.95

Version 1.0