Handling user input in HTML5 Canvas-based games

Capture keyboard, mouse, and touch events for game development

Kevin Moot July 24, 2012

When stepping into the world of HTML5 games, it's easy to underestimate the complexity of managing keyboard, mouse, and touch-based input. This article explores essential techniques for handling user interaction in HTML Canvas-based games. Learn how to handle keyboard and mouse events, how to defeat the web browser's default event behaviors, and how to broadcast events to a logical representation of game objects. Also learn to handle device-agnostic input on mobile devices such as the iPhone and iPad.

Introduction

Developers with a background in Flash or Silverlight are often surprised that applications written for HTML5 Canvas offer no special accommodations in terms of handling user input. Essentially, HTML user input involves using an event-handling system built into browsers since the earliest days of JavaScript-enabled web browsers; there's nothing specific to HTML5 for detecting and handling user input. For example, the browser can provide low-level feedback indicating which coordinate (x,y) the user has clicked on, and that's about it.

Handling user interaction is no different than any other low-level game architecture. There are no built-in abstractions to notify you when the user has interacted with a specific object that has been rendered on the Canvas. This provides a great degree of low-level control over how you want to handle these events. As long as you can keep various browser quirks at bay, you can ultimately tune the handling of the events for maximum efficiency according to a unique application—rather than being tied to a specific implementation.

In this article, learn techniques for handling user interaction in HTML Canvas-based games. Examples illustrate how to handle keyboard, mouse, and touch-based events. Strategies for broadcasting events to game objects, and mobile compatability, are also covered.

You can download the source code for the examples used in this article.

Types of events

User interaction is handled entirely by the browser's traditional event listener model. There is nothing new with the advent of HTML5; it's the same event model that has been used since the early days of Netscape Navigator.

Essentially, think of an interactive application or game as a marriage between the browser event model for user input and Canvas for graphical output. There is no logical connection between the two unless you build it yourself.

You will take advantage of the fact that event listeners can be attached to the `<canvas>` element itself. Because the `<canvas>` element is simply a block-level element, as far as the browser is concerned this is no different than attaching event listeners to a `<div>` or any other block-level element.

Keyboard events

The simplest types of events to listen for and handle are keyboard events. They are not dependant on the Canvas element or the user's cursor position. Keyboard events simply require you to listen for the keydown, keyup, and keypress events at the document level.

Listening for keyboard events

The event listener model can vary depending upon the browser implementation, so the quickest way to get up and running is to use a library to normalize the handling of events. The following examples use jQuery to bind events. This is generally the easiest way to get started, but performance may suffer due to the level of cruft involved in jQuery's effort to be compatible with legacy browsers. Another popular library, specifically written for speedy cross-browser keyboard event handling, is Kibo (see Related topics).

Listing 1 illustrates listening for key events and taking an appropriate action based on which key was pressed.

Listing 1. Handling keyboard events

```
$(document.body).on('keydown', function(e) {
    switch (e.which) {
        // key code for left arrow
        case 37:
            console.log('left arrow key pressed!');
            break;

        // key code for right arrow
        case 39:
            console.log('right arrow key pressed!');
            break;
    }
});
```

If your application takes place in the environment of a web browser, it's important to keep sensible keyboard combinations in mind. While it may be technically possible to define behaviors for certain common key combinations that will override their default browser behaviors (such as control-r), this is highly frowned upon.

Mouse events

Mouse events are more complicated than keyboard events. You must be aware of the position of the Canvas element within the browser window as well as the position of the user's cursor.

Listening for mouse events

It's easy to get the position of the mouse relative to the entire browser window using the `e.pageX` and `e.pageY` properties. In this case, the origin of (0,0) would be located at the top left of the entire browser window.

You typically don't care too much about user input when the user's cursor is not positioned within the Canvas area. Therefore, it would be better to consider the origin of (0,0) to be located at the top left of the Canvas element. Ideally, you want to be working within the local coordinate system that's relative to the Canvas area rather than a global coordinate system that's relative to the entire browser window.

Mouse event strategies

Use the following steps to transform global window coordinates to local Canvas coordinates.

1. Calculate the (x,y) position of the Canvas DOM element on the page.
2. Determine the global position of the mouse in relation to the entire document.
3. To locate the origin (0,0) at the top left of the Canvas element, and effectively transform the global coordinates to relative coordinates, take the difference between the global mouse position calculated in step 2 and the Canvas position calculated in step 1.

Figure 1 shows an example of the information you need to capture in terms of the global coordinate system.

Figure 1. Mouse position, global coordinates

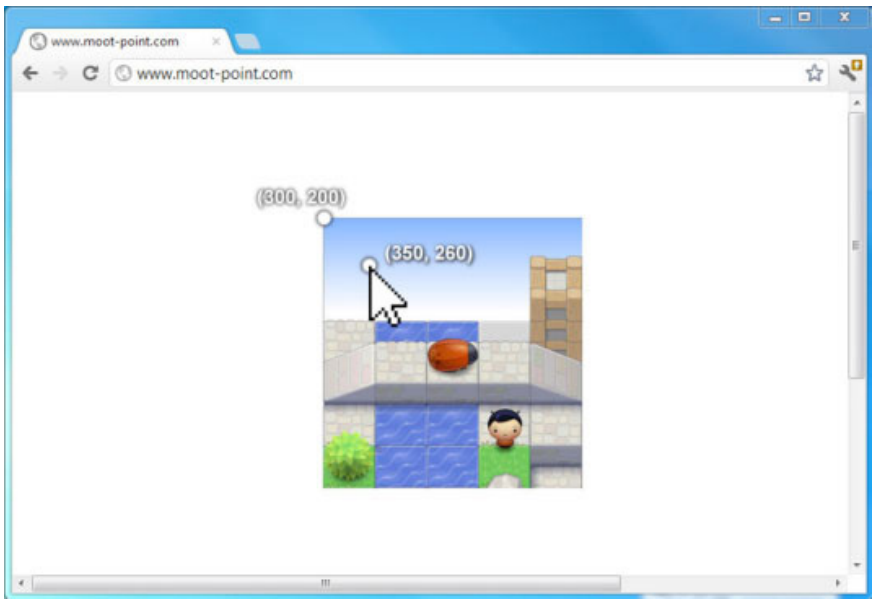
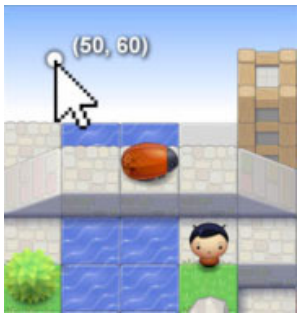


Figure 2 shows the result after transforming the mouse position into local coordinates.

Figure 2. Mouse position after transformation into local coordinates



Listing 2 shows the method of determining local mouse coordinates. It is assumed you've defined a Canvas element in the markup, as follows: `<canvas id="my_canvas"></canvas>`.

Listing 2. Handling mouse events

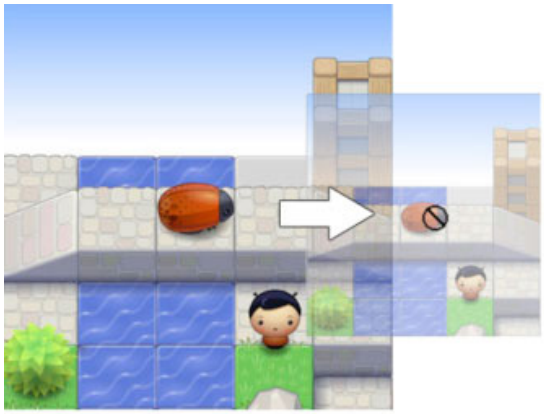
```
var canvas = $('#my_canvas');

// calculate position of the canvas DOM element on the page

var canvasPosition = {
    x: canvas.offset().left,
    y: canvas.offset().top
};

canvas.on('click', function(e) {

    // use pageX and pageY to get the mouse position
    // relative to the browser window

    var mouse = {
        x: e.pageX - canvasPosition.x,
        y: e.pageY - canvasPosition.y
    }

    // now you have local coordinates,
    // which consider a (0,0) origin at the
    // top-left of canvas element
});
```

Undesired browser behaviors

In a computer game, you typically don't want any default browser behaviors interfering with your actions. For instance, you don't want a drag of the mouse to perform text selection, a right-click of the mouse to open context menus, or a scroll of the mouse wheel to move the page up and down.

Figure 3 shows an example of what can occur if a user clicks and drags an image in the browser. Although the default browser behavior makes total sense for drag and drop applications, it is not a behavior you would want in your game.

Figure 3. Default browser behavior when dragging an image



In all event handlers, add a `preventDefault()` line and return false from the function. The code in Listing 3 will do the trick as far as preventing both the default action and event bubbling from occurring.

Listing 3. Preventing default behaviors

```
canvas.on('click', function(e) {
    e.preventDefault();

    var mouse = {
        x: e.pageX - canvasPosition.x,
        y: e.pageY - canvasPosition.y
    }

    //do something with mouse position here

    return false;
});
```

Even with the code in Listing 3, you could still encounter several undesirable side effects when the user initiates a drag event on a DOM element, such as the appearance of the I-beam cursor, text selection, and so on. The drag event issue is traditionally more common with images, but it's a good idea to also apply it to the Canvas element to prevent dragging and selections. Listing 4 shows a CSS rule to prevent selection side-effects by sprinkling in a bit of CSS.

Listing 4. Recommended styles to prevent selection

```
image, canvas {
    user-select: none;
    -ms-user-select: none;
    -webkit-user-select: none;
    -khtml-user-select: none;
    -moz-user-select: none;
    -webkit-touch-callout: none;
    -webkit-user-drag: none;
}
```

Overriding desktop behaviors

It is generally a good idea to override drag and selection events to ensure that the browser's default drag and selection behavior does not rear its ugly head.

The code in Listing 5 intentionally does not use jQuery for attaching events. jQuery does not properly handle the `ondragstart` and `onselectstart` events (if attached using jQuery, the event handlers may never fire).

Listing 5. Canceling drag and selection events

```
var canvasElement = document.getElementById('my_canvas');

// do nothing in the event handler except canceling the event
canvasElement.ondragstart = function(e) {
    if (e && e.preventDefault) { e.preventDefault(); }
    if (e && e.stopPropagation) { e.stopPropagation(); }
    return false;
}

// do nothing in the event handler except canceling the event
canvasElement.onselectstart = function(e) {
    if (e && e.preventDefault) { e.preventDefault(); }
    if (e && e.stopPropagation) { e.stopPropagation(); }
    return false;
}
```

Overriding mobile behaviors

On mobile devices, it is often critical that you prevent the user from zooming and panning the browser window (zooming and panning are often the mobile browser's default behavior for touch gestures).

You can prevent the zoom behavior by adding `user-scalable=no` to the `viewport` meta-tag. For example:

```
<meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1" />
```

To disable all movement of the document or window using gestures, attach the event listeners in Listing 6 to the document.body file. This will essentially cancel all default browser behaviors if the user happens to tap anywhere outside of the Canvas or game area.

Listing 6. Canceling mobile window movement

```
document.body.ontouchstart = function(e) {
    if (e && e.preventDefault) { e.preventDefault(); }
    if (e && e.stopPropagation) { e.stopPropagation(); }
    return false;
}

document.body.ontouchmove = function(e) {
    if (e && e.preventDefault) { e.preventDefault(); }
    if (e && e.stopPropagation) { e.stopPropagation(); }
    return false;
}
```

Broadcasting to game objects

You need to attach only one event listener to the Canvas for each type of event you want to capture. For example, if you need to capture click and mousemove events, simply attach a single click event listener and a single mousemove event listener to the Canvas. These event listeners

only need to be attached once, so it's typical to attach these events during the initialization of the application.

If you need any useful information that is captured by the event listeners to propagate down to objects rendered on the Canvas, you must build your own logic for the system. In this example, such a system would be responsible for broadcasting the click or mousemove event to all game objects that are concerned with handling one of those events.

When each game object learns of one of these events, the game object would first need to identify whether the click or mousemove event concerns them. If so, the game object would then need to determine whether the mouse coordinates are positioned within its own boundaries.

Broadcasting strategies

Your exact strategy will vary based on the type of game. For instance, a 2D tileset may have a different strategy than a 3D world.

The following steps outline a naive implementation that can work well for a simple 2D application.

1. Detect the coordinates of the user's mouse click within the Canvas area.
2. Notify all game objects that a click event has occurred at the given set of coordinates.
3. For each game object, perform a hit test between the mouse coordinates and the bounding box of the game object to determine whether the mouse coordinates collide with that object.

Simple broadcasting example

The click event handler might look something like Listing 7. The example assumes that you've already set up some kind of structure to track all of the game objects in the world. The position and dimensions of all game objects are stored in a variable called `gameObjectArray`.

Listing 7. Click event handler broadcasting to game objects

```
// initialize an array of game objects
// at various positions on the screen using
// new gameObject(x, y, width, height)

var gameObjectArray = [
 new gameObject(0, 0, 200, 200),
 new gameObject(50, 50, 200, 200),
 new gameObject(500, 50, 100, 100)
];

canvas.on('click', function(e) {
    var mouse = {
        x: e.pageX - canvasPosition.x,
        y: e.pageY - canvasPosition.y
    }

    // iterate through all game objects
    // and call the onclick handler of each

    for (var i=0; i < gameObjectArray.length; i++) {
        gameObjectArray[i].handleClick(mouse);
    }
});
```

The next step is to ensure that each game object is able to perform hit testing to determine whether the mouse coordinate collides within the bounding box region of the game object. Figure 4 shows an example of an unsuccessful hit test.

Figure 4. Out of bounds click--hit test is unsuccessful

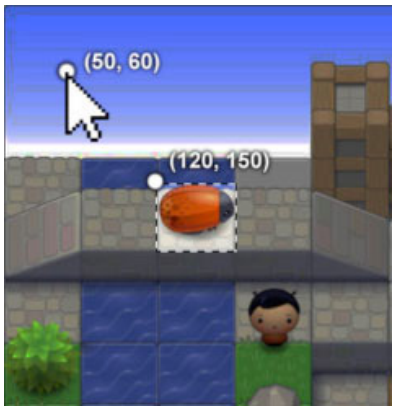
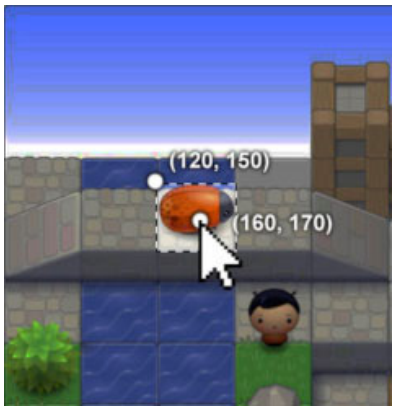


Figure 5 shows a successful hit test.

Figure 5. In-bounds click--hit test is successful



You can define a class for game objects, as in Listing 8. A hit test is performed within the `onclick()` function, which tests for a collision between the rectangular bounding box of the object and the mouse coordinates that are passed in as a parameter.

Listing 8. Game object class and hit testing

```
function gameObject(x, y, width, height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;

    // mouse parameter holds the mouse coordinates
    this.handleClick = function(mouse) {

        // perform hit test between bounding box
        // and mouse coordinates

        if (this.x < mouse.x &&
            this.x + this.width > mouse.x &&
```

```
            this.y < mouse.y &&
            this.y + this.height > mouse.y) {

            // hit test succeeded, handle the click event!
            return true;
        }

        // hit test did not succeed
        return false;
    }
}
```

Improving broadcasting efficiency

In many cases, it's possible to build a more efficient implementation. For example, in a game with thousands of game objects, you definitely want to avoid testing every game object in the scene against every time an event is fired.

The following example uses jQuery custom events to fire off a synthetic event. The synthetic event is only handled by those game objects that are listening for that particular event. For the example:

1. Handle the mouse-click event as before, and perform any necessary transformations (such as transforming the mouse position in terms of local coordinates).
2. Fire off a synthetic event that contains the transformed mouse coordinates as a parameter.
3. Any game object concerned with handling a click event would set up a listener to listen for the synthetic event.

The mouse click event handler is modified to simply trigger a custom event. The custom event can be given any arbitrary name. In Listing 9, it's called `handleClick`.

Listing 9. Triggering a custom event

```
canvas.on('click', function(e) {
    var mouse= {
        x: e.pageX - canvasPosition.x,
        y: e.pageY - canvasPosition.y
    }

    //fire off synthetic event containing mouse coordinate info
    $(canvas).trigger('handleClick', [mouse]);
});
```

As shown in Listing 10, the game object class is also modified. Instead of defining an `onclick` function, simply listen for the `handleClick` event. Any time the `handleClick` event is triggered, any game objects that are listening for that event will fire their corresponding event handlers.

Listing 10. Handling a custom event

```
function gameObject(x, y, width, height) {
    var self = this;
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;

    $(canvas).on('handleClick', function(e, mouse) {
```

```
        // perform hit test between bounding box
        // and mouse coordinates

        if (self.x < mouse.x &&
            self.x + self.width > mouse.x &&
            self.y < mouse.y &&
            self.y + self.height > mouse.y) {

            // hit test succeeded, handle the click event!

        }
    });
}
```

Advanced hit testing

It's important to consider what will happen when several game objects are layered on top of one another. If the user clicks on a point where several game objects are layered, you'll need to determine how to handle the behavior. For instance, you would typically expect only the nearest object's event handler to be fired off and the other objects below it ignored.

To handle such layering, you need to know the order, or depth, of each layered game object. Canvas does not expose any logical representation of depth, so again you need to take the reigns and produce the necessary logic to handle this situation.

To introduce the concept of depth, assigning a z-index for all game objects to represent their depth is necessary. Listing 11 shows an example.

Listing 11. Adding a z-index to the Game object

```
function gameObject(x, y, zIndex, width, height) {
    var self = this;
    this.x = x;
    this.y = y;
    this.zIndex = zIndex;
    this.width = width;
    this.height = height;

    //...
}
```

To facilitate depth testing, you need to perform sorting. In Listing 12, the example structure for storing game objects is sorted such that the game object with the highest z-index occurs first in the list.

Listing 12. Sorting the game object array

```
// sort in order such that highest z-index occurs first
var sortedGameObjectArray = gameObjectArray.sort(function(gameObject1, gameObject2) {
    if (gameObject1.zIndex < gameObject2.zIndex) return true;
    else return false;
})
```

Finally, in the `click` function, switch things up to iterate through all game objects in the sorted array.

As soon as you encounter a positive result from a game object's hit test, immediately break so the click does not continue to propagate. If you don't halt the testing, as in Listing 13, the undesirable behavior of game objects at deeper depths handling the `click` event will continue.

Listing 13. Breaking on successful hit test

```
canvas.on('click', function(e) {
    var mouse = {
        x: e.pageX - canvasPosition.x,
        y: e.pageY - canvasPosition.y
    }

    for (var i=0; i < sortedGameObjectArray.length; i++) {
        var hitTest = sortedGameObjectArray[i].onclick(mouse);

        // stop as soon as one hit test succeeds
        if (hitTest) {
            break; // break out of the hit test
        }
    }
});
```

Irregular game object boundaries

Though it's often simplest and most efficient to perform hit testing against rectangular bounding boxes, in many cases that's not sufficient. If the game object has a more irregular shape, it might make more sense to test against a triangular or polygonal bounding area. In such cases, you would need to swap out the hit test logic in the game object's event handler for the more advanced form of hit detection. Typically, you would refer to the realm of game collision physics for the appropriate logic.

The Canvas API offers an interesting function called `IsPointInPath()` that can perform polygonal collision tests. Essentially, `IsPointInPath(x, y)` lets you test whether the given (x,y) point falls within an arbitrary path (basically a polygonal boundary). It will return true if the provided (x,y) coordinate falls within the current path, which is defined within Canvas context.

Using `isPointInPath()`

Figure 6 shows a situation where it would be necessary to test the mouse coordinates against a non-rectangular path. In this case, it is a simple triangular path.

Figure 6. Clicking within the bounds of a triangular path

The filled path is visualized for illustration purposes only. Because the path does not have to be physically rendered on-screen for `IsPointInPath()` to return a useful result, it is sufficient to define the path without ever making a call to `fill()` or `stroke()` to actually draw the path. Listing 14 shows the details.

Listing 14. Using `isPointInPath` for hit detection

```
$(canvas).on('handleClick', function(e, mouse) {

    // first, define polygonal bounding area as a path
    context.save();
    context.beginPath();
    context.moveTo(0,0);
    context.lineTo(0,100);
    context.lineTo(100,100);
    context.closePath();

    // do not actually fill() or stroke() the path because
    // the path only exists for purposes of hit testing
    // context.fill();

    // perform hit test between irregular bounding area
    // and mouse coordinates
    if (context.isPointInPath(mouse.x, mouse.y)) {
        // hit test succeeded, handle the click event!

    }
    context.restore();
});
```

Although it is often more efficient to write the collision algorithms yourself rather than using `IsPointInPath()`, it can be a good tool for prototyping and rapid development.

Mobile compatibility

To make the example game compatible with mobile devices, you'll need to work with touch events rather than mouse events.

Although a tap of the finger can also be interpreted by the mobile browser as a click event, it is generally not a good approach to rely on listening to only click events on mobile browsers. A better approach is to attach listeners for specific touch events in order to guarantee the best responsiveness.

Detecting touch events

You can write a helper function that first detects whether the device supports touch events and then returns either mouse coordinates or touch coordinates accordingly. This lets calling functions agnostically process input coordinates regardless of whether you're on a desktop or mobile platform.

Listing 15 shows an example of a device-agnostic function to capture mouse and touch events and to normalize the response.

Listing 15. Normalizing mouse and touch events

```
function getPosition(e) {
    var position = {x: null, y: null};

    if (Modernizr.touch) { //global variable detecting touch support
        if (e.touches && e.touches.length > 0) {
            position.x = e.touches[0].pageX - canvasPosition.x;
            position.y = e.touches[0].pageY - canvasPosition.y;
        }
    }
    else {
        position.x = e.pageX - canvasPosition.x;
        position.y = e.pageY - canvasPosition.y;
    }

    return position;
}
```

For detection of touch support, the example uses the Modernizr library (see Related topics). The Modernizr library makes detection of touch support simply a matter of testing the variable Modernizr.touch, which returns true if the device supports touch events.

Device-agnostic event handlers

During the initialization of the application, you can replace the earlier code to define the event listeners with a separate branch for touch-supporting devices and mouse input. It is quite straightforward to map mouse events to an equivalent touch event. For instance, `mousedown` is replaced with `touchstart`, and `mouseup` is replaced with `touchend`.

Listing 16 shows an example of using Modernizr to map equivalent mouse/touch events. It also uses the `getPosition()` function defined in Listing 15.

Listing 16. Using normalized mouse/touch events

```
var eventName = Modernizr.touch ? 'touchstart' : 'click';

canvas.on(eventName, function(e) {
    e.preventDefault();

    var position = getPosition(e);
    //do something with position here

    return false;
});
```

Unless you need to handle more advanced actions, such as pinching and swiping, this approach generally works well when doing a direct port of mouse events from a desktop application. A single-touch system is assumed; if multi-touch detection is needed, it would require some additional code (which is outside the scope of this article).

Conclusion

In this article, you learned how to handle keyboard and mouse events, and how to cancel undesired browser behavior. The article also discussed strategies for broadcasting events to

game objects, and reviewed more advanced considerations for hit testing and a simple method of addressing mobile compatibility. Although the scope of user input goes far beyond this article, the typical user input scenarios offer a jumping-off point toward creating a robust, device-agnostic library to handle user input for your HTML5 application.

Downloadable resources

Description	Name	Size
Article code listings	article.listings.zip	5KB

Related topics

- Develop and deploy your next app on the IBM Bluemix cloud platform.
- "Create great graphics with the HTML5 canvas" (developerWorks, February 2011): Read how to enhance your web pages with Canvas, a simple HTML5 element that packs a punch.
- "HTML5 fundamentals, Part 4: The final touch - The Canvas" (developerWorks, July 2011): Learn about the HTML5 Canvas element with several examples that demonstrate functions.
- Canvas Pixel Manipulation: This demo from the Safari Dev Center is an excellent example of managing the canvas to develop effective visual assets.
- WHATWG: Explore this community of developers working with the W3C to fine-tune HTML5.
- Canvas tutorial: The Mozilla developers show how to implement the <canvas> element in your HTML pages.
- HTML5 Canvas Reference: Check out W3Schools.com's useful exercises to help you hone your canvas knowledge.
- jQuery Events API: Learn about methods used to register behaviors to take effect when the user interacts with the browser and to further manipulate those registered behaviors.
- developerWorks Web development zone: Find articles covering various web-based solutions. See the Web development technical library for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- jQuery: Download this popular JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development.
- Modernizr: Get the open-source JavaScript library that helps you build the next generation of HTML5 and CSS3-powered websites.
- Kibo: Use this popular, simple JavaScript library for handling keyboard events.
- IBM product evaluation versions: Download or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.