Further Event-Handling Topics: Key events and Timed Functions

Key events

Key events occur when the user presses a key. With the growth of in-browser applications resembling traditional desktop applications, handling key events is becoming more useful. Key events include ***keydown*** (when the user presses a key down), ***keypress*** (when the user presses a key down and up immediately as you normally would when typing) and ***keyup*** (when the user releases a key). Here is an example of using key events:

```
<!DOCTYPE html>
<html>
<head>
<style type='text/css'>
#canvas1 { background-color: #ffffc0; }
</style>
<script type='text/javascript'>

function init()
{
    var canvas = document.getElementById('canvas1');
    canvas.addEventListener ("keyup", handleKeyEvent);
}

function handleKeyEvent(e)
{
    status("status","keyCode=" + e.keyCode + " shift? '
}

function status(id,txt)
{
    document.getElementById(id).innerHTML = txt;
}


</script>
</head>
<body onload='init()'>
<p>
<canvas id='canvas1' width='400' height='400' tabindex=
```

```
</canvas>
</p>
<div id='status'>status</div>
</body>
</html>
```

Note how we make use of two properties of the event object relating to key events, namely ***keyCode*** and ***shiftKey***. ***keyCode*** gives us a numerical code (not necessarily the ASCII code) for each key while ***shiftKey*** gives us a boolean (true/false) value representing whether the Shift key was pressed or not. ***keyCode*** relates to the actual key, so pressing A or Shift/A will return the same value (even though lower case 'a' and capital 'A' have different ASCII codes); we use ***shiftKey*** to determine whether Shift was pressed at the same time. You can also use ***ctrlKey*** and ***altKey*** in the same way, but use of CTRL and Alt during in browser applications is generally not used, as these keys have default actions in the browser window itself.

Note also the use of ***tabindex*** in the canvas tag. (Ref: Jonathan Snook). Normally, you cannot focus a canvas by clicking the mouse or tabbing to it, because (unlike text fields, for example) they were not intended for text input. Adding the attribute ***tabindex*** with a value of ***0*** (i.e. tabindex=0) to the canvas allows it to be focused by clicking the mouse on it or tabbing to it. It also allows you to force focus in code using the ***focus()*** method, for example:

```
canvas.focus();
```

We can also use tabindex=-1; in this case, it will not be possible to tab to the canvas, but it is still possible to force the canvas to take keyboard focus in code or via a mouse click.

Timed functions

In the kind of event-driven programming that we use with JavaScript, we might want to schedule a particular action to occur at a fixed time interval. Imagine, for example, an in-browser game. The hero might move in response to the arrow keys, but we need to make the monsters move independently and continuously without some sort of trigger by the user (if the hero stands still the monsters will still need to chase). This can be done with the ***setInterval()*** function. ***setInterval()*** is very easy to use: it takes two arguments, the function we would like to run, and the interval to run it in milliseconds. For example:

```
<!DOCTYPE html>
<html>
<head>
<style type='text/css'>
```

```
#canvas1 { background-color: #ffffc0; }
</style>
<script type='text/javascript'>


var canvas, index;
var styles = ['black','blue','red','magenta','green','c


function init()
{
    canvas = document.getElementById('canvas1');
    setInterval( doUpdate, 2000);
    index = 0;
}


function doUpdate()
{
    var ctx = canvas.getContext("2d");
    ctx.fillStyle = styles[index];
    ctx.fillRect(0,0,400,400);

    index++;
    if (index==styles.length)
    {
        index = 0;
    }
}

</script>
</head>
<body onload='init()'>
<p>
<canvas id='canvas1' width='400' height='400' tabindex=
</canvas>
</p>
</body>
</html>
```

Here we define a ***doUpdate()*** function. It fills the canvas with the current colour from the array ***styles*** and then increases the index variable "count" by one, so that it moves on to the next colour and once it's reached the end, it

resets "count" to 0 again. We schedule this function to be called in the ***doUpdate()*** function every 2000 milliseconds using ***setInterval()***:

```
setInterval( doUpdate, 2000);
```

Starting and stopping a scheduled function

At some point we will probably want to stop our scheduled function. We might also want to stop the user trying to set a scheduled function going twice. To do these tasks we need to make use of the ***return value*** of ***setInterval()***, which is a "handle" on the function. We can then use ***clearInterval()***, passing the handle in as an argument, to stop the function being scheduled. Also, we can test whether the "handle" already exists, to prevent the function being scheduled twice. The next example demonstrates this:

```
<!DOCTYPE html>
<html>
<head>
<style type='text/css'>
#canvas1 { background-color: #ffffc0; }
</style>
<script type='text/javascript'>

var canvas, index;
var styles = ['black','blue','red','magenta','green','c
var handle;

function init()
{
    canvas = document.getElementById('canvas1');
    document.getElementById("startbtn").addEventListene
    document.getElementById("endbtn").addEventListener
}

function start()
{
    if(handle)
    {
        alert("Function already scheduled!");
    }
    else
    {
        handle=setInterval(doUpdate, 2000);
        index = 0;
    }
```

```
}

function end()
{
    // Clear the function and set "handle" to null so t
    // in "start" will work and we'll be able to schedu
    // again.
    if(handle)
    {
        clearInterval(handle);
        handle=null;
    }
}

function doUpdate(n)
{
    var ctx = canvas.getContext("2d");
    ctx.fillStyle = styles[index];
    ctx.fillRect(0,0,400,400);

    index++;
    if (index==styles.length)
    {
        index = 0;
    }
}

</script>
</head>
<body onload='init()'>
<p>
<canvas id='canvas1' width='400' height='400' tabindex=
</canvas>
</p>
<input type='button' id='startbtn' value='Start!' />
<input type='button' id='endbtn' value='End!' />
</body>
</html>
```

Note how we have two buttons, one to schedule the function and another to cancel it. Note also how we test for the existence of the handle before starting the function, to prevent the function being scheduled more than once which, if done enough times, might crash the browser!

setTimeout()

Very similar to ***setInterval()*** is ***setTimeout()***. The only difference is that it schedules a function to run just once, not multiple times.

Further exercise

Write code to move an image round the canvas. The user should use the arrow keys to move it. (key codes for the cursor keys: 37=left, 38=up, 39=right, 40=down). If you do not have an image use this one: hero.png

Advanced exercises

- Develop your application above (in which you moved an image around the screen using the cursor keys) into a simple game, in which the player has to avoid the chasing enemy for as long as possible. To do this, you will need to write a function to update the enemy's position to move it closer to the player, and schedule it with ***setInterval()***.
  To give you further practice with object-oriented programming in JavaScript, you should create an object to represent a game character (player or enemy). The object should have properties including x and y position, and, if you are using an image, an appropriate Image object.
- Develop an in-browser application which resembles Microsoft Paint. Users should be able to scribble on a canvas and change colour (via a series of coloured s to allow the user to pick a colour). Users should be able to save their image to the server. If you have time, try allowing users to pick a drawing tool (freehand, lines, rectangles, text, etc).