

# HTML5 Canvas

Drawing on a Web page



POLITECNICO  
DI TORINO

Laura Farinetti - DAUIN



# Summary

- The canvas element
- Basic shapes
- Images
- Colors and gradients
- Pixel manipulation
- Animations

# Drawing on a Web page

- Possible just very recently
- SVG and canvas
  - Provide native drawing functionality on the Web
  - Completely integrated into HTML5 documents (part of DOM)
  - Can be styled with CSS
  - Can be controlled with JavaScript

# Canvas

- HTML5 element and plugin-free 2D drawing API that enables to dynamically generate and render graphics, charts, images, animation
- Scriptable bitmap canvas
  - Images that are drawn are final and cannot be resized
  - Can be manipulated with JavaScript and styled with CSS
  - 2D Context
  - 3D Context (Web GL)
- Canvas was originally introduced by Apple to be used in Mac OS

# Canvas

- The HTML5 <canvas> element is used to draw graphics, on the fly, via scripting (usually JavaScript)
- A canvas is a rectangular area, where it is possible to control every pixel
- The canvas element has several methods for drawing paths, boxes, circles, characters, and adding images
- The <canvas> element is only a container for graphics
  - You must use a script to actually draw the graphics

```
<canvas id="myCanvas" width="200" height="100">  
</canvas>
```

# Canvas features

- Canvas can draw text
  - Colorful text, with or without animation
- Canvas can draw graphics
  - Great features for graphical data presentation
- Canvas can be animated
  - Canvas objects can move: from simple bouncing balls to complex animations
- Canvas can be interactive
  - Canvas can respond to JavaScript events
  - Canvas can respond to any user action (key clicks, mouse clicks, button clicks, finger movement)
- HTML canvas can be used in games

# Canvas

- The canvas is initially blank
- To display something a script needs to access the rendering context and draw on it
- The canvas element has a DOM method called `getContext`, used to obtain the rendering context and its drawing functions
  - `getContext()` takes one parameter: the type of context (2D or 3D)
  - `getContext()` is a built-in HTML object, with properties and methods for drawing (paths, boxes, circles, characters, images, and more)

```
var canvas = document.getElementById('example');  
var ctx = canvas.getContext('2d');
```

# Canvas

- Skeleton template

```
<html>
  <head>
    <title>Canvas</title>
    <script type="application/javascript">
      function draw() {
        var canvas = document.getElementById('example');
        if (canvas.getContext) {
          var ctx = canvas.getContext('2d'); } }
    </script>
    <style type="text/css">
      canvas { border: 1px solid black; }
    </style>
  </head>
  <body onload="draw();" >
    <canvas id="example" width="150" height="150"></canvas>
  </body>
</html>
```



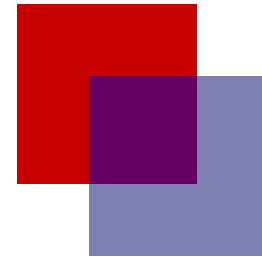
# Example: basic shape



- All drawing must be done in JavaScript

```
<html>
  <head>
    <script type="application/javascript">
      function draw() {
        var canvas=document.getElementById("canvas");
        if (canvas.getContext) {
          var ctx = canvas.getContext("2d");
          ctx.fillStyle="#FF0000";
          ctx.fillRect(0,0,150,75); } }
    </script>
  </head>
  <body onload="draw();" >
    <canvas id="canvas" width="200" height="100">
    </canvas>
  </body>
</html>
```

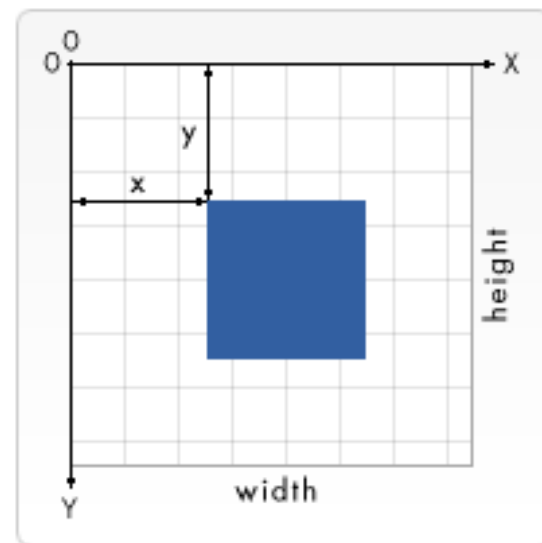
# Example



```
<html>
  <head>
    <script type="application/javascript">
      function draw() {
        var canvas=document.getElementById("canvas");
        if (canvas.getContext) {
          var ctx = canvas.getContext("2d");
          ctx.fillStyle = "rgb(200,0,0)";
          ctx.fillRect (10, 10, 55, 50);
          ctx.fillStyle = "rgba(0, 0, 200, 0.5)";
          ctx.fillRect (30, 30, 55, 50); } }
    </script>
  </head>
  <body onload="draw();" >
    <canvas id="canvas" width="150" height="150">
    </canvas>
  </body>
</html>
```

# Rectangles functions

- `fillRect(x,y,width,height)`
  - draws a filled rectangle
- `strokeRect(x,y,width,height)`
  - draws a rectangular outline
- `clearRect(x,y,width,height)`
  - clears the specified area and makes it fully transparent
- Canvas coordinate space



# Path functions

- `beginPath()`:
  - creates a path (list of lines, arcs, ...)
- `closePath()`
  - closes the path by drawing a straight line from the current point to the start
- `stroke()`
  - draws an outlined shape
- `fill()`
  - paints a solid shape
- `moveTo(x,y)`
  - move the pencil to the x and y coordinates
- `lineTo(x,y)`
  - draws a straight line to the specified ending point
- `arc(x, y, radius, startAngle, endAngle, anticlockwise)`
  - draws an arc using a center point (x, y), a radius, a start and end angle (in radians), and a direction flag (false for clockwise, true for counter-clockwise)
  - to convert degrees to radians: `var radians = (Math.PI/180)*degrees`

# Example: path



```
function drawShape() {
  var canvas = document.getElementById('tutorial');
  if (canvas.getContext){
    // use getContext to use the canvas for drawing
    var ctx = canvas.getContext('2d');
    // Draw shapes
    ctx.beginPath();
    ctx.arc(75,75,50,0,Math.PI*2,true);
    ctx.moveTo(110,75);
    ctx.arc(75,75,35,0,Math.PI,false);
    ctx.moveTo(65,65);
    ctx.arc(60,65,5,0,Math.PI*2,true);
    ctx.moveTo(95,65);
    ctx.arc(90,65,5,0,Math.PI*2,true);
    ctx.stroke(); }
  else { ... }
}
```

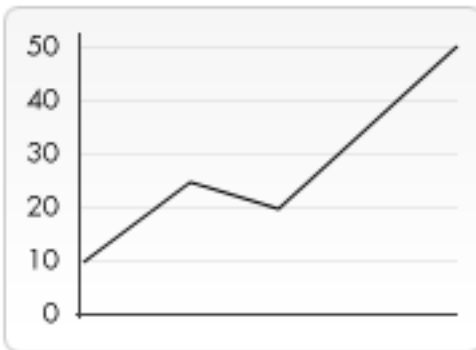
# Examples: image

- `drawImage(image, x, y)`
  - renders an image



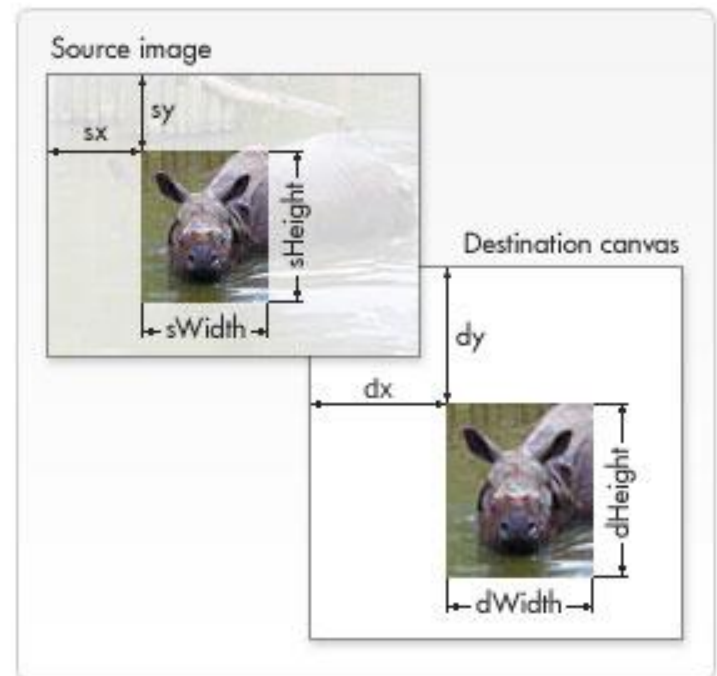
```
function draw() {  
  var ctx = document.getElementById('canvas').getContext('2d');  
  var img = new Image();  
  img.src = "img_flwr.png";  
  img.onload = function(){  
    ctx.drawImage(img,0,0);  
  }  
}
```

```
function draw() {  
  var ctx = document.getElementById('canvas').getContext('2d');  
  var img = new Image();  
  img.src = 'images/backdrop.png';  
  img.onload = function(){  
    ctx.drawImage(img,0,0);  
    ctx.beginPath();  
    ctx.moveTo(30,96);  
    ctx.lineTo(70,66);  
    ctx.lineTo(103,76);  
    ctx.lineTo(170,15);  
    ctx.stroke(); }  
}
```



# Images

- Images can be scaled...
  - `drawImage(image, x, y, width, height)`
- ... and sliced
  - `drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight)`



# Example



```
function draw() {  
  var ctx = document.getElementById('canvas').getContext('2d');  
  var img = new Image();  
  img.src = 'images/rhino.jpg';  
  img.onload = function(){  
    for (i=0;i<4;i++) {  
      for (j=0;j<3;j++) {  
        ctx.drawImage(img,j*50,i*38,50,38);  
      }  
    }  
  }  
}
```





# Example



```
function draw() {  
  var ctx = document.getElementById('canvas').getContext('2d');  
  ctx.drawImage(document.getElementById('source'),  
                33,71,104,124,21,20,87,104);  
  ctx.drawImage(document.getElementById('frame'),0,0);  
}
```



# Example: colors and interaction



Click on the buttons below to change the color of the rectangle.



```
<body>
  <canvas id="drawing" style="" > </canvas>
  <p>Click on the buttons below to change the color of the
    rectangle. </p>

  <input type="button" value="Blue" id="blue" onclick="BlueRect()" />
  <input type="button" value="Green" id="green" onclick="GreenRect()" />
  <input type="button" value="Yellow" id="yellow" onclick="YellRect()" />
  <input type="button" value="Red" id="red" onclick="RedRect()" />
  <input type="button" value="Click to clear canvas" id="clear"
    onclick="ImgClr()" />
</body>
```

# Example: colors and interaction

```
<script type="text/javascript">
var canvas=null; var context=null;
window.onload = function() {
    canvas=document.getElementById("drawing");
    context=canvas.getContext("2d");
    // Border
    context.beginPath(); //This initiates the border
    context.rect(100,60,175,70);
    context.fillStyle="#ffffff";
    context.fill();
    // Border width
    context.lineWidth=1; //This sets the width of the border
    // Border color
    context.strokeStyle="#000000";
    context.stroke(); }
function BlueRect () {
    context.fillStyle="#701be0"; // Changes the rectangle to blue
    context.fill();
    context.strokeStyle="#000000";
    context.stroke(); }
function GreenRect () { ... }
function ImgClr () {
    context.clearRect(0,0, canvas.width, canvas.height); }
</script>
```

# Gradients

- `createLinearGradient(x1,y1,x2,y2)`
  - starting point (x1,y1) and end point (x2,y2) of the gradient
- `createRadialGradient(x1,y1,r1,x2,y2,r2)`
  - starting circle (x1,y1, r1) and end circle (x2,y2, r2)
- `addColorStop(position,color)`
  - position: a number between 0.0 and 1.0 that defines the relative position of the color in the gradient
  - color: string

```
var lineargradient = ctx.createLinearGradient(0,0,150,150);  
lineargradient.addColorStop(0,'red');  
lineargradient.addColorStop(1,'green');
```



# Example: linear gradients

```
function draw() {  
  var ctx = document.getElementById('canvas').getContext('2d');  
  // Create gradient  
  var lingrad = ctx.createLinearGradient(0,0,0,150);  
  lingrad.addColorStop(0, '#00ABEB');  
  lingrad.addColorStop(0.5, '#fff');  
  lingrad.addColorStop(0.5, '#66CC00');  
  lingrad.addColorStop(1, '#fff');  
  // assign gradients to fill style  
  ctx.fillStyle = lingrad;  
  // draw shape  
  ctx.fillRect(10,10,130,130);  
}
```



# Example 14

## Radial gradients



```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');

  // Create gradients
  var radgrad = ctx.createRadialGradient(45,45,10,52,50,30);
  radgrad.addColorStop(0, '#A7D30C');
  radgrad.addColorStop(0.9, '#019F62');
  radgrad.addColorStop(1, 'rgba(1,159,98,0)');

  var radgrad2 = ctx.createRadialGradient(105,105,20,112,120,50);
  radgrad2.addColorStop(0, '#FF5F98');
  radgrad2.addColorStop(0.75, '#FF0188');
  radgrad2.addColorStop(1, 'rgba(255,1,136,0)');
  ...
}
```

# Example: radial gradients



```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  // Create gradients
  var radgrad = ctx.createRadialGradient(45,45,10,52,50,30);
  radgrad.addColorStop(0, '#A7D30C');
  radgrad.addColorStop(0.9, '#019F62');
  radgrad.addColorStop(1, 'rgba(1,159,98,0)');
  var radgrad2 = ctx.createRadialGradient(105,105,20,112,120,50);
  radgrad2.addColorStop(0, '#FF5F98');
  radgrad2.addColorStop(0.75, '#FF0188');
  radgrad2.addColorStop(1, 'rgba(255,1,136,0)');
  var radgrad3 = ctx.createRadialGradient(95,15,15,102,20,40);
  radgrad3.addColorStop(0, '#00C9FF');
  radgrad3.addColorStop(0.8, '#00B5E2');
  radgrad3.addColorStop(1, 'rgba(0,201,255,0)');
  var radgrad4 = ctx.createRadialGradient(0,150,50,0,140,90);
  radgrad4.addColorStop(0, '#F4F201');
  radgrad4.addColorStop(0.8, '#E4C700');
  radgrad4.addColorStop(1, 'rgba(228,199,0,0)');
  // draw shapes
  ctx.fillStyle = radgrad4; ctx.fillRect(0,0,150,150);
  ctx.fillStyle = radgrad3; ctx.fillRect(0,0,150,150);
  ctx.fillStyle = radgrad2; ctx.fillRect(0,0,150,150);
  ctx.fillStyle = radgrad; ctx.fillRect(0,0,150,150);
}
```

# Example: save and restore

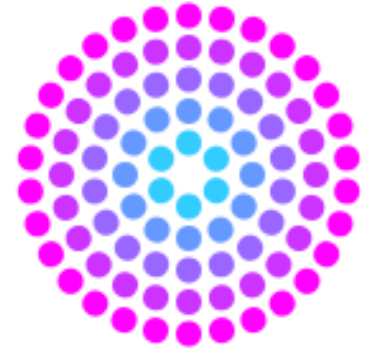


- Canvas states can be stored on a stack
- Methods `save()` and `restore()`

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  ctx.fillRect(0,0,150,150); // default settings
  ctx.save(); // Save the default state
  ctx.fillStyle = '#09F' // Make changes to the settings
  ctx.fillRect(15,15,120,120); // Draw with new settings
  ctx.save(); // Save the current state
  ctx.fillStyle = '#FFF' // Make changes to the settings
  ctx.globalAlpha = 0.5;
  ctx.fillRect(30,30,90,90); // Draw with new settings
  ctx.restore(); // Restore previous state
  ctx.fillRect(45,45,60,60); // Draw with restored settings
  ctx.restore(); // Restore original state
  ctx.fillRect(60,60,30,30); // Draw with restored settings
}
```



# Example: transformations



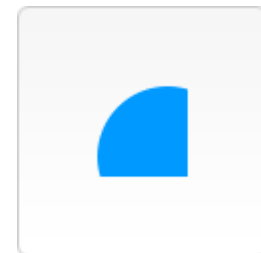
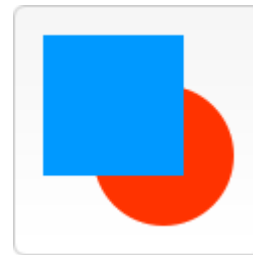
- translate(x,y)
- rotate(angle)
- scale(x,y)
- transform(m11,m12,m21,m22,dx,dy)

```
function draw() {  
  var ctx = document.getElementById('canvas').getContext('2d');  
  ctx.translate(75,75);  
  for (i=1;i<6;i++){  
    ctx.save();  
    ctx.fillStyle = 'rgb('+ (51*i) +',' + (255-51*i) +',255)';  
    for (j=0;j<i*6;j++){  
      ctx.rotate(Math.PI*2/(i*6));  
      ctx.beginPath();  
      ctx.arc(0,i*12.5,5,0,Math.PI*2,true);  
      ctx.fill(); }  
    ctx.restore(); }  
}
```

# Compositing

```
globalCompositeOperation = type
```

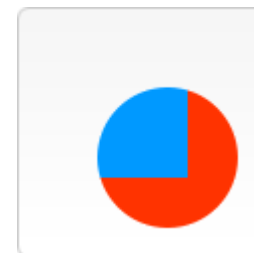
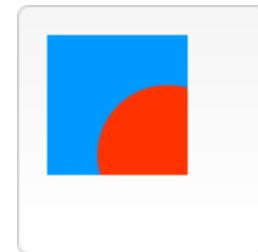
- source-over (default)
  - draws new shapes on top of the existing canvas content
- destination-over
  - new shapes are drawn behind the existing canvas content
- source-in
  - the new shape is drawn only where both the new shape and the destination canvas overlap; everything else is transparent
- destination-in
  - the existing canvas content is kept where both the new shape and existing canvas content overlap; everything else is transparent



# Compositing

```
globalCompositeOperation = type
```

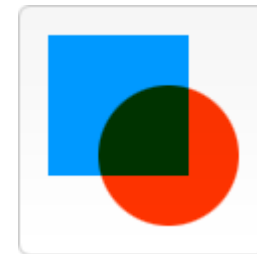
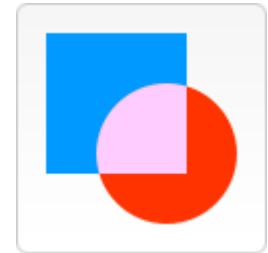
- source-out
  - the new shape is drawn where it doesn't overlap the existing canvas content
- destination-out
  - the existing content is kept where it doesn't overlap the new shape
- source-atop
  - the new shape is only drawn where it overlaps the existing canvas content
- destination-atop
  - the existing canvas is only kept where it overlaps the new shape; the new shape is drawn behind the canvas content



# Compositing

```
globalCompositeOperation = type
```

- **lighter**
  - where both shapes overlap the color is determined by adding color values
- **darker (unimplemented)**
  - where both shapes overlap the color is determined by subtracting color values
- **xor**
  - shapes are made transparent where both overlap and drawn normal everywhere else
- **copy**
  - only draws the new shape and removes everything else



# Example: clipping path



```
function draw() {  
  var ctx = document.getElementById('canvas').getContext('2d');  
  ...  
  
  // Create a circular clipping path  
  ctx.beginPath();  
  ctx.arc(0,0,60,0,Math.PI*2,true);  
  ctx.clip();  
  
  ...  
}
```

# Canvas pixel manipulation

- It is possible to access the individual pixels of a canvas, by using the ImageData object
- The ImageData object has three properties: width, height and data
  - The width and height properties contain the width and height of the graphical data area
  - The data property is a byte array containing the pixel values
- The 2D context API provides three methods to draw pixel-by-pixel
  - `createImageData()`
  - `getImageData()`
  - `putImageData()`

# Pixel manipulation

- Example: create a 100 x 100 pixels ImageData object

```
var canvas = document.getElementById("ex1");  
var context = canvas.getContext("2d");  
var width = 100;  
var height = 100;  
var imageData = context.createImageData(width, height);
```

- Each pixel in the data array consists of 4 bytes values: one value for the red color, green color and blue color, and a value for the alpha channel
  - Each of the red, green, blue and alpha values can take values between 0 and 255
- Example: sets the color and alpha values of the first pixel

```
var pixelIndex = 0;  
imageData.data[pixelIndex ] = 255;    // red color  
imageData.data[pixelIndex + 1] = 0;    // green color  
imageData.data[pixelIndex + 2] = 0;    // blue color  
imageData.data[pixelIndex + 3] = 255;  // alpha
```

# Pixel manipulation

- Once you have finished manipulating the pixels, you can copy them onto the canvas using the function `putImageData()`

```
var canvasX = 25;  
var canvasY = 25;  
context.putImageData(imageData, canvasX, canvasY);
```

- It is also possible to grab a rectangle of pixels from a canvas into an `ImageData` object, by using the `getImageData()` function
  - `x` and `y`: coordinates of the upper left corner of the rectangle to grab from the canvas
  - `width` and `height`: width and height of the rectangle to grab from the canvas

```
var x = 25;  
var y = 25;  
var width = 100;  
var height = 100;  
var imageData2 = context.getImageData(x, y, width, height);
```



# Example: pixel manipulation



```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  var imgd = false;
  var w = 50, var h = 50, x = 0, y = 0;

  imgd = ctx.createImageData(w, h);
  var pix = imgd.data;

  // Loop over each pixel
  for (var i = 0, n = pix.length; i < n; i += 4) {
    pix[i] = 255; // the red channel
    pix[i+3] = 127; // the alpha channel
  }

  // Draw the ImageData object.
  ctx.putImageData(imgd, x, y);
}
```

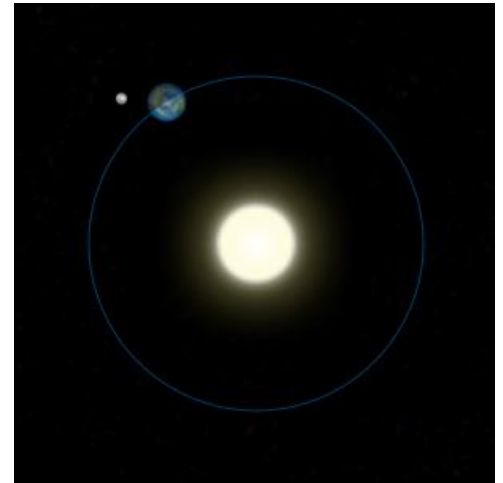
# Example: pixel manipulation

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  var x = 0, y = 0;
  // Create a new image
  var img = new Image();
  img.src = 'tree.jpg';
  // Draw the image on canvas
  img.onload = function(){
    ctx.drawImage(img,0,0);
    // Get the pixels
    var imgd = ctx.getImageData(x, y, this.width, this.height);
    var pix = imgd.data;
    // Loop over each pixel and invert the color.
    for (var i = 0, n = pix.length; i < n; i += 4) {
      pix[i ] = 255 - pix[i ]; // red
      pix[i+1] = 255 - pix[i+1]; // green
      pix[i+2] = 255 - pix[i+2]; // blue
      // i+3 is alpha (the fourth element)
    }
    // Draw the ImageData object
    ctx.putImageData(imgd, x, y);
  }
}
```



# Animations

- Since scripts can control canvas elements, it's easy to make animations
- Unfortunately there are limitations: once a shape gets drawn it stays that way
  - To move it we have to redraw it and everything that was drawn before it
- It takes a lot of time to redraw complex frames and the performance depends highly on the speed of the computer it's running on



# Basic animation steps

- Clear the canvas
  - Unless the shapes you'll be drawing fill the complete canvas (for instance a backdrop image), you need to clear any shapes that have been drawn previously
  - The easiest way to do this is using the `clearRect` method
- Save the canvas state
  - If you're changing any setting (styles, transformations, etc) which affect the canvas state and want to make sure the original state is used each time a frame is drawn, you need to save it
- Draw animated shapes
  - The step where you do the actual frame rendering
- Restore the canvas state
  - If you've saved the state, restore it before drawing a new frame

# Controlling animations

- Two ways
  - Execute the drawing functions over a period of time

```
// execute every 500 milliseconds  
setInterval(animateShape,500);  
  
// execute once after 500 milliseconds  
setTimeout(animateShape,500);
```

- User input: by setting eventListeners to catch user interaction

- Examples



# License

- This work is licensed under the Creative Commons “Attribution-NonCommercial-ShareAlike Unported (CC BY-NC-SA 3,0)” License.
- You are free:
  - to Share - to copy, distribute and transmit the work
  - to Remix - to adapt the work
- Under the following conditions:
  - Attribution - You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
  - Noncommercial - You may not use this work for commercial purposes.
  - Share Alike - If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.
- To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/>