How To Create 3D Graphics Using Canvas

This topic covers a technique for manipulating a 3-dimensional object using HTML5's canvas element.

Note To view the sample presented in this topic, you'll need a browser, such as Windows Internet Explorer 9 or later, that supports the canvas element.

Suppose we want to visualize a 3-dimensional (3D) object by displaying various views of it on a display device, such as a computer monitor. Further, assume that the object to display is composed entirely of (*x*, *y*, *z*) points:

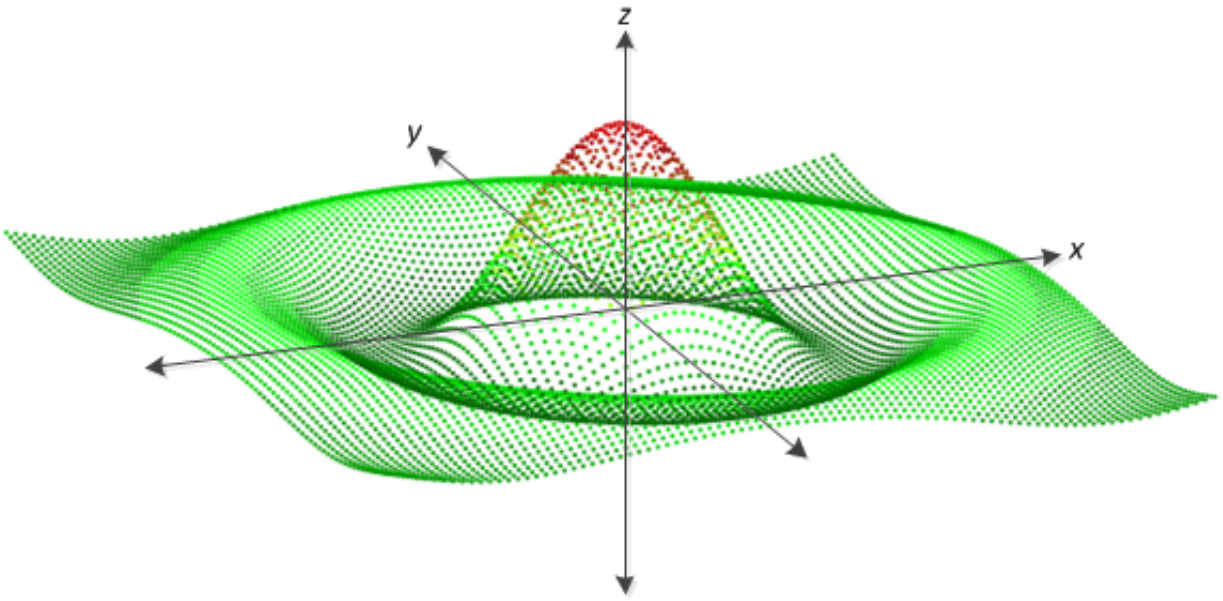

Figure 1

If we embed the object in an *xyz*-coordinate system such that its origin is at the center of the monitor and the *xy*-plane coincides with the plane of the monitor, an observer will see a projection of the view of the 3D object onto the 2D *xy*-plane or monitor:

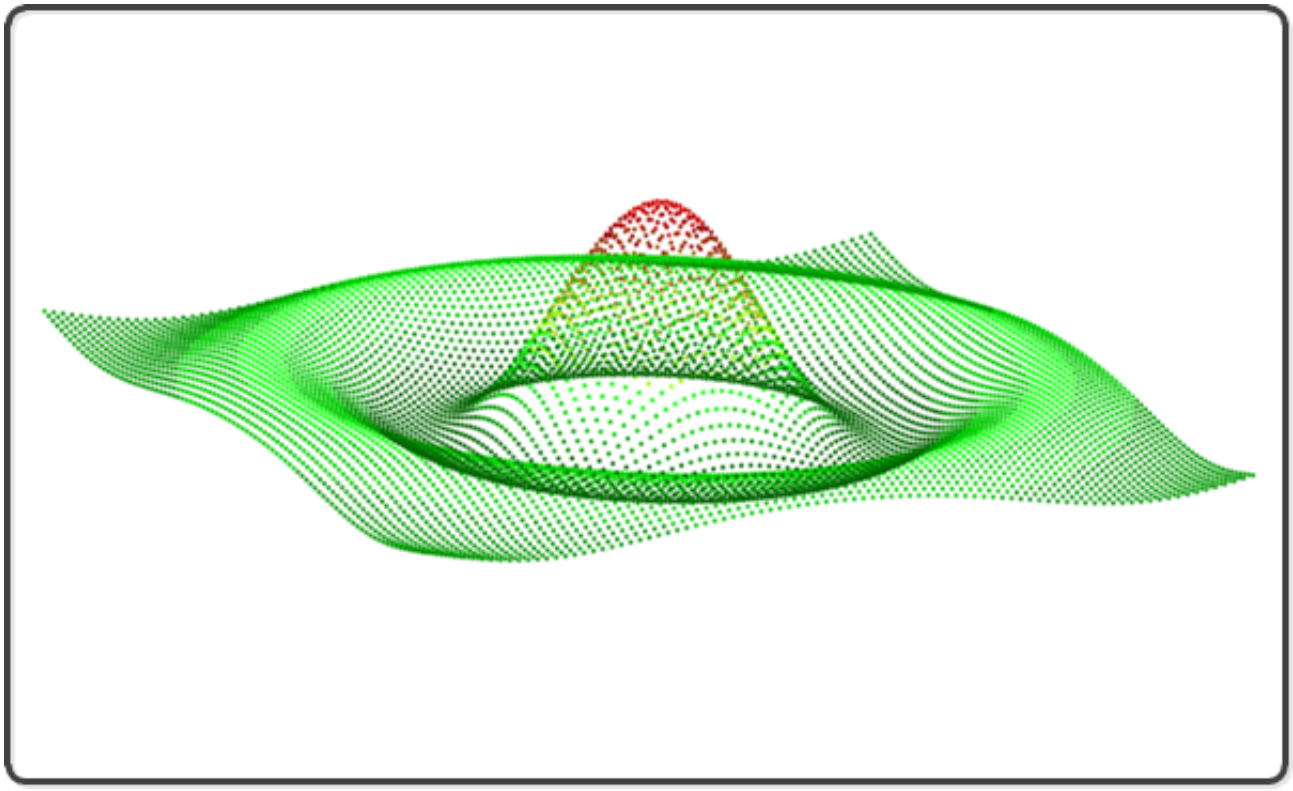

Figure 2

The object shown in Figure 1 is a point-based representation of the mathematical surface:

$$z = f(x,y) = \frac{4\sin\left(\sqrt{x^2+y^2}\right)}{\sqrt{x^2+y^2}}$$

To create the image in Figure 2, 8100 evenly spaced (*x*, *y*) points from the *xy*-plane were used to calculate the height *z* of the surface above the *xy*-plane using the given equation *f*(*x*, *y*). Stated another way, each surface point (*x*, *y*, *z*) in the *xyz*-coordinate system is given by (*x*, *y*, *f*(*x*, *y*)).

As an aside, the previous surface would appear "flat" when viewed from above. That is, when looking directly down the *z*-axis you'd see a contour plot of the surface:

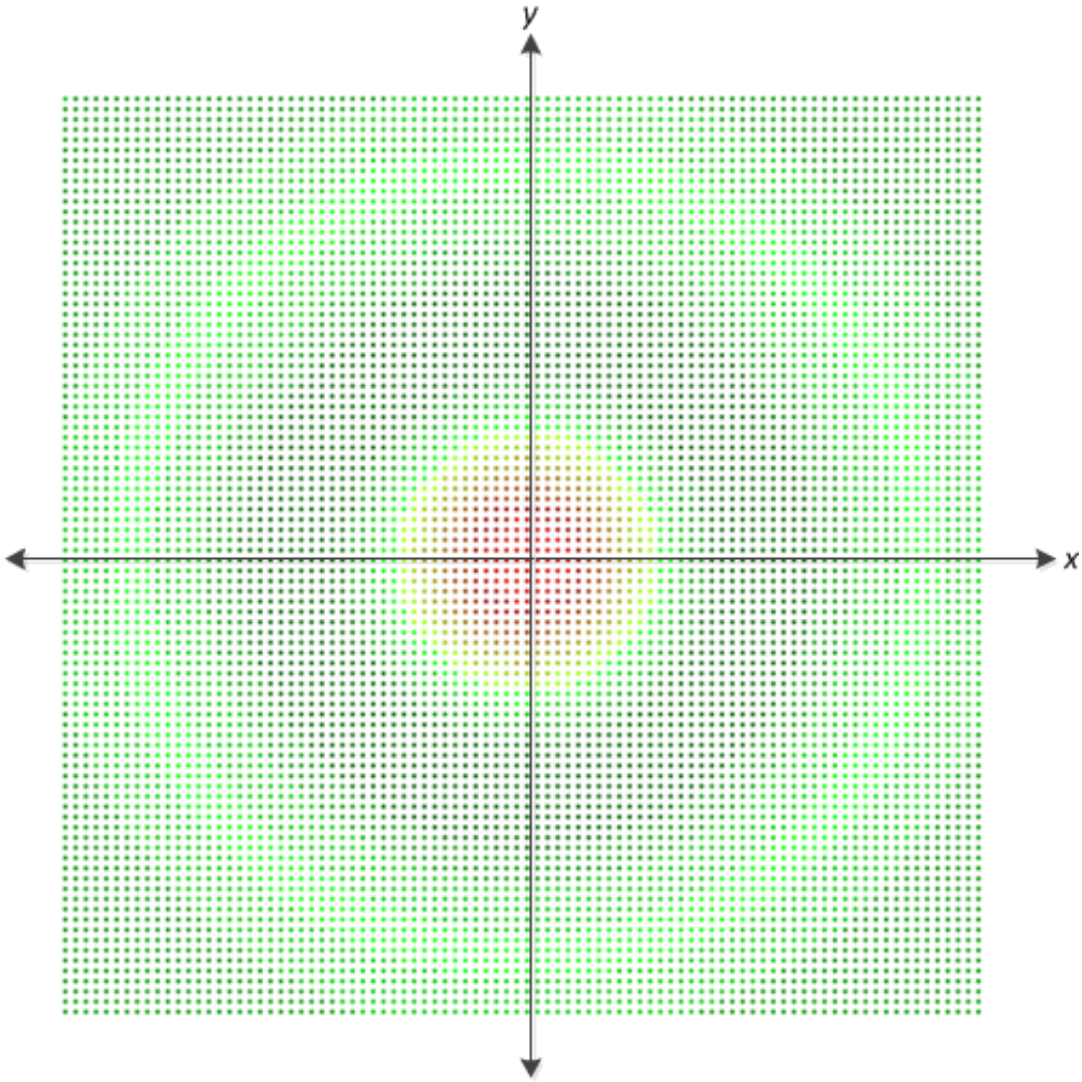

Figure 3

This contour plot of the surface shows the 8100 evenly spaced *xy*-points used to calculate the *z*-coordinates of the surface. A contour plot can definitely be useful in its own right (such as a topographic map) but it is only when you rotate these 3D points about the *x*-, *y*-, and *z*-axes that one can start to see the true nature of a 3D object (as suggested in Figures 1 and 2). This then begs the question – how do you display and rotate 3D points on an inherently 2D display device?

To answer this question, we will use the canvas element, which provides fast plotting of 2D points in an HTML5-compliant browser, such as Internet Explorer 9 or later. Be aware that the canvas element (at the time of this writing) only supports the manipulation of 2D objects. For example, the canvas **rotate** method is designed to rotate flat 2D objects only. Thus, we must write our own methods to plot and rotate 3D objects, such as the above surface $z = f(x, y)$. Fortunately, the first of these two tasks is trivial:

Plotting 3D points

For a given view of the surface, such as Figure 2, only the *x*- and *y*-coordinates of the points are needed by the display device to draw the view, as only the projection of the surface onto the *xy*-plane is displayed. We must, however, keep track of the *z*-coordinates to carry out certain transformations, as discussed next.

Rotating 3D points

To start, we place all n surface points (n = 8100) in a 3 x n matrix P, known as the coordinate matrix of the view. That is,

there are *n* columns in *P* such that each column represents a surface point:

$$P = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \\ z_1 & z_2 & \cdots & z_n \end{bmatrix}$$

In order to rotate this surface about the *x*-, *y*-, or *z*-axes, we multiple *P* with an appropriate 3 x 3 rotation matrix *R*:

$P' = RP$

Here *P* is the original view of the surface and *P′* is the view after being rotated about one of the coordinate axes as defined by *R*. Because there are three coordinate axes, there are three rotational matrices for *R*:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If, for example, you wanted to rotate the surface shown in Figure 3 by π/12 radians (15°) about the *x*-axis, R_x becomes (approximately):

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.966 & -0.259 \\ 0 & 0.259 & 0.966 \end{bmatrix}$$

To produce the new rotated view *P′* of the surface, one simply performs the matrix multiplication R_xP as follows:

$$P' = R_xP = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.966 & -0.259 \\ 0 & 0.259 & 0.966 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \\ z_1 & z_2 & \cdots & z_n \end{bmatrix} =$$

$$\begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ 0.966y_1 - 0.259z_1 & 0.966y_2 - 0.259z_2 & \cdots & 0.966y_n - 0.259z_n \\ 0.259y_1 + 0.966z_1 & 0.259y_2 + 0.966z_2 & \cdots & 0.259y_n + 0.966z_n \end{bmatrix}$$

Then, to display the surface, the resulting numeric (*x*, *y*) points from *P*' are plotted on the display device. The following section provides a concrete example of this technique.

3D Canvas sample

This section discusses the canvas3dRotation.html sample. To view the source code associated with this sample, use the view source feature of your browser. For example, in Windows Internet Explorer, right-click the webpage whose source code you want to view and click **View source**. Make sure that you have the source code available while reading the remainder of this document.

The first step is to choose an interesting surface *f*(*x*, *y*) and an appropriate range for *x* and *y*. As in Figure 1, we let:

$$f(x,y) = \frac{4\sin\sqrt{x^2+y^2}}{\sqrt{x^2+y^2}}$$

An acceptable range for f is:

$$-9 \le x \le 9$$

$$-9 \le y \le 9$$

The next step is to decide upon a reasonable number of 3D points to plot over this 18 x 18 *xy*-plane region. If we allow five points per unit, we have a total of 5·18 x 5·18 = 8100 surface points of the form (*x*, *y*, *z*) = (*x*, *y*, *f*(*x*, *y*)):

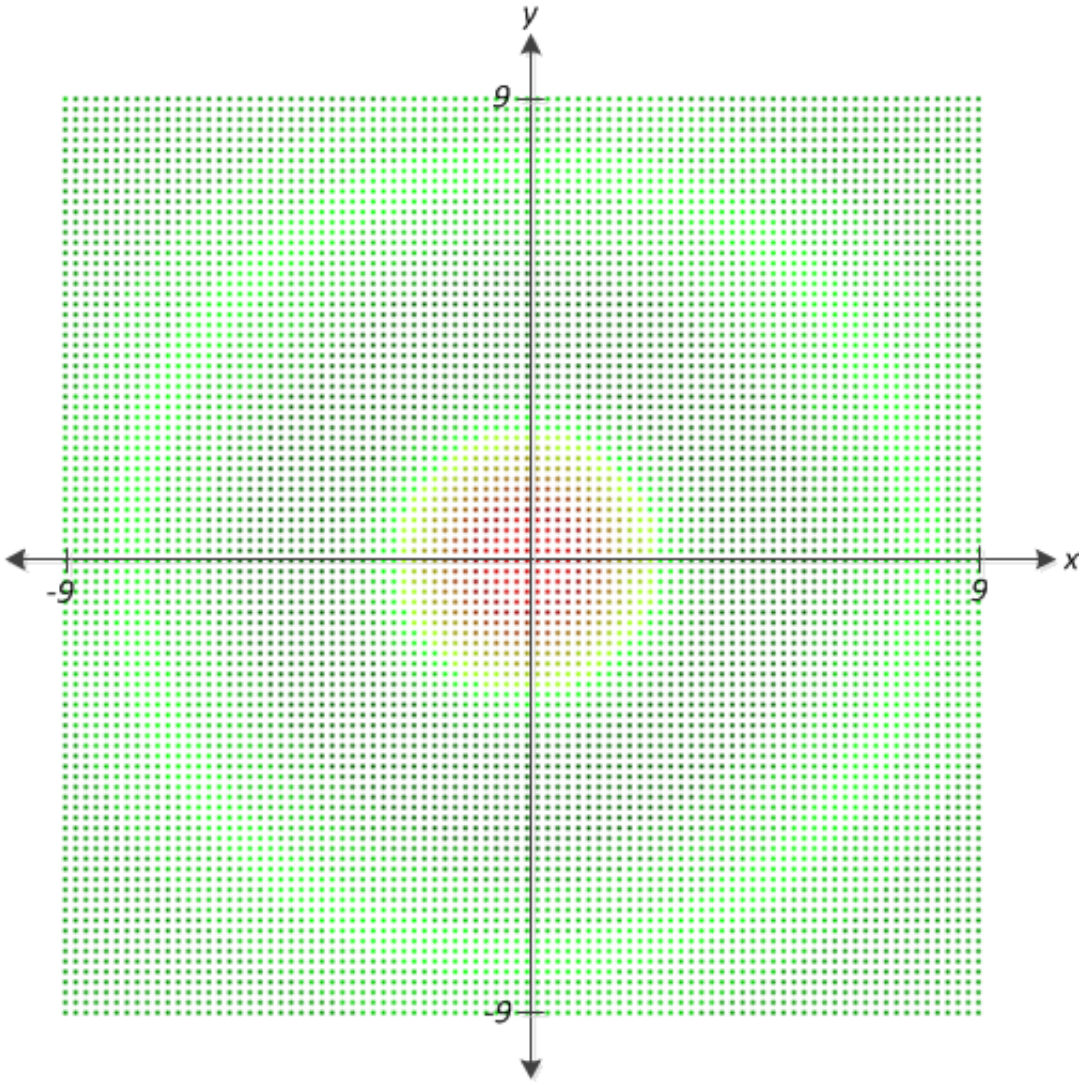

Figure 4

This range related information is contained in the global `constants` variable, as indicated by the following JavaScript comments:

JavaScript

```
var constants = {
```

```
    canvasWidth: 600,
    canvasHeight: 600,
    leftArrow: 37,
    upArrow: 38,
    rightArrow: 39,
    downArrow: 40,
    xMin: -9, // RANGE RELATED
    xMax: 9, // RANGE RELATED
    yMin: -9, // RANGE RELATED
    yMax: 9, // RANGE RELATED
    xDelta: 0.2, // RANGE RELATED
    yDelta: 0.2, // RANGE RELATED
    colorMap: ["#060", "#090", "#0C0", "#0F0", "#9F0", "#9C0", "#990", "#960", "#930", "#900
    pointWidth: 2,
    dTheta: 0.05,
    surfaceScale: 24
};
```

The 8100 points are stored in a JavaScript array of the form [[x_0, y_0, z_0], [x_1, y_1, z_1], ..., [x_{8099}, y_{8099}, z_{8099}]]. The array is initialized as follows:

JavaScript

```
Surface.prototype.generate = function()
{
  var i = 0;

  for (var x = constants.xMin; x <= constants.xMax; x += constants.xDelta)
  {
    for (var y = constants.yMin; y <= constants.yMax; y += constants.yDelta)
    {
      this.points[i] = point(x, y, this.equation(x, y));
      ++i;
    }
  }
}
```

Here `this.equation(x, y)` is equivalent to:

$$f(x,y) = \frac{4\sin\sqrt{x^2+y^2}}{\sqrt{x^2+y^2}}$$

And the `this.points` array is analogous to the matrix *P* above (that is, a list of 3 x 1 column vectors, each of which represents a 3D surface point). In the array, `this.points[0][0]` would access the *x*-coordinate of the first surface point and `this.points[2][2]` would access the *z*-coordinate of the third point.

Once all surface points have been generated, the colors for the points are chosen based upon the *z*-coordinate of the point. That is, a point's color is based upon the "height" of the point "above" the *xy*-plane. There are 11 such "height" colors (contained in the `constants.colorMap` array), which are assigned as follows:

JavaScript

```
Surface.prototype.color = function()
{
  var z;

  this.zMin = this.zMax = this.points[0][Z];
  for (var i = 0; i < this.points.length; i++)
  {
    z = this.points[i][Z];
    if (z < this.zMin) { this.zMin = z; }
    if (z > this.zMax) { this.zMax = z; }
  }

  var zDelta = Math.abs(this.zMax - this.zMin) / constants.colorMap.length;

  for (var i = 0; i < this.points.length; i++)
  {
    this.points[i].color = constants.colorMap[ Math.floor( (this.points[i][Z]-this.zMin)/zI
  }
}
```

The most complex aspect of this method is the following `for` loop:

JavaScript

```
for (var i = 0; i < this.points.length; i++)
{
  this.points[i].color = constants.colorMap[ Math.floor( (this.points[i][Z]-this.zMin)/zDe
}
```

To help understand this loop, recall that `constants.colorMap` contains the following array literal:

JavaScript

```
["#060", "#090", "#0C0", "#0F0", "#9F0", "#9C0", "#990", "#960", "#930", "#900", "#C00"]
```

In other words, this `for` loop is functionally equivalent to the following (much less elegant) loop:

JavaScript

```
for (var i = 0; i < this.points.length; i++)
{
  if (this.points[i][Z] <= this.zMin + zDelta) {this.points[i].color = "#060";}
  else if (this.points[i][Z] <= this.zMin + 2*zDelta) {this.points[i].color = "#090";}
  else if (this.points[i][Z] <= this.zMin + 3*zDelta) {this.points[i].color = "#0C0";}
  else if (this.points[i][Z] <= this.zMin + 4*zDelta) {this.points[i].color = "#0F0";}
  else if (this.points[i][Z] <= this.zMin + 5*zDelta) {this.points[i].color = "#9F0";}
  else if (this.points[i][Z] <= this.zMin + 6*zDelta) {this.points[i].color = "#9C0";}
  else if (this.points[i][Z] <= this.zMin + 7*zDelta) {this.points[i].color = "#990";}
  else if (this.points[i][Z] <= this.zMin + 8*zDelta) {this.points[i].color = "#960";}
  else if (this.points[i][Z] <= this.zMin + 9*zDelta) {this.points[i].color = "#930";}
  else if (this.points[i][Z] <= this.zMin + 10*zDelta) {this.points[i].color = "#900";}
  else {this.points[i].color = "#C00";}
}
```

As can be seen above, the color representing the lowest *z*-coordinates is `#060` (a darkish green). `#060` is then "increased" towards `#C00` (a medium red), which represents the highest *z*-coordinates. Stated another way, the minimum and maximum *z*-coordinates (`this.zMin` and `this.zMax`) are used to linearly distribute (in the vertical sense) the 11 "height" colors amongst the points based upon each point's *z*-coordinate.

The `points` array now contains all the information required to plot a view of the surface. In order to do so, we programmatically create a canvas element as shown here:

JavaScript

```
function appendCanvasElement()
{
  var canvasElement = document.createElement('canvas');

  canvasElement.width = constants.canvasWidth;
  canvasElement.height = constants.canvasHeight;
  canvasElement.id = "myCanvas";

  canvasElement.getContext('2d').translate(constants.canvasWidth/2, constants.canvasHeight

  document.body.appendChild(canvasElement);
}
```

The `translate(constants.canvasWidth/2, constants.canvasHeight/2)` method is used to center the *xyz*-coordinate system in the middle of the canvas.

Now that we have created an appropriate canvas element, we can draw (or plot) each point to it:

JavaScript

```
Surface.prototype.draw = function()
{
  var myCanvas = document.getElementById("myCanvas");
  var ctx = myCanvas.getContext("2d");

  this.points = surface.points.sort(surface.sortByZIndex);

  for (var i = 0; i < this.points.length; i++)
  {
    ctx.fillStyle = this.points[i].color;
    ctx.fillRect(this.points[i][X] * constants.surfaceScale, this.points[i][Y] * constants
  }
}
```

In other words, for each 3 x 1 point in the `points` array, we create a small colored rectangle (which is must faster than rendering a small circle) and position it using the point's *x*- and *y*-coordinates multiplied by an empirically derived constant. The critical line is repeated here:

JavaScript

```
ctx.fillRect(this.points[i][X] * constants.surfaceScale, this.points[i][Y] * constants.sur
```

The (empirically derived) constant `constants.surfaceScale` is used to scale the surface such that the surface is guaranteed to visually fit on the canvas, for all possible views.

Notice that before we draw the points (rectangles) to the display device, we sort the points by their relative z-axis locations. This ensures that the points farthest from the viewer's eyes are drawn first and those closest are drawn last. When the point widths (`pointWidth`) are small (1 to 2 pixels), this sorting effect isn't to noticeable. However, try commenting out `this.points = surface.points.sort(surface.sortByZIndex);` and increasing `pointWidth` to 5 - when you rotate the surface, odd visualizations start to occur. Now remove the comment and refresh the page - the surface rotates as expected.

Now that we are able to plot a view of the surface, we can change the view through a rotation, as shown here:

JavaScript

```
Surface.prototype.multi = function(R)
{
  var Px = 0, Py = 0, Pz = 0;
  var P = this.points;
  var sum;
```

```
    for (var V = 0; V < P.length; V++)
    {
      Px = P[V][X], Py = P[V][Y], Pz = P[V][Z];
      for (var Rrow = 0; Rrow < 3; Rrow++)
      {
        sum = (R[Rrow][X] * Px) + (R[Rrow][Y] * Py) + (R[Rrow][Z] * Pz);
        P[V][Rrow] = sum;
      }
    }
  }
```

This method performs the matrix multiplication *RP* to produce the new view *P'* (as described previously). More precisely, this method performs *P* = *RP*, where "=" indicates the JavaScript assignment operator.

Recall that *R* describes a rotation about the *x*-, *y*-, or *z*-axis. That is, the *R* passed into the `multi` method is either R_x, R_y, or R_z (as defined previously). R_x, R_y, and R_z are implemented via the `xRotate`, `yRotate`, and `zRotate` methods, respectively. For example, R_y is implemented as follows:

JavaScript

```
Surface.prototype.yRotate = function(sign)
{
  var Ry = [ [0, 0, 0],
             [0, 0, 0],
             [0, 0, 0] ];

  Ry[0][0] = Math.cos( sign*constants.dTheta );
  Ry[0][1] = 0;
  Ry[0][2] = Math.sin( sign*constants.dTheta );
  Ry[1][0] = 0;
  Ry[1][1] = 1;
  Ry[1][2] = 0;
  Ry[2][0] = -Math.sin( sign*constants.dTheta );
  Ry[2][1] = 0;
  Ry[2][2] = Math.cos( sign*constants.dTheta );

  this.multi(Ry);
  this.erase();
  this.draw();
}
```

When called, `yRotate` rotates the surface about the *y*-axis by the small angular amount `constants.dTheta`. That is, `yRotate` first builds `Ry` and then calls `this.multi(Ry)` to perform R_yP, thus rotating the surface about the *y*-axis by `sign*constants.dTheta` radians. The `sign` parameter is used to rotate either clockwise or counterclockwise about the *y*-axis.

As you may have guessed, there are six possible rotations – clockwise or counterclockwise about the *x*-, *y*-, or *z*-axis. Thus, and as described on the instructions.html page, there are six arrow key combinations used to signal the desired rotation:

JavaScript

```
function processKeyDown(evt)
{
  if (evt.ctrlKey)
  {
    switch (evt.keyCode)
    {
      case constants.upArrow:
        // No operation other than preventing the default behavior of the arrow key.
        evt.preventDefault();
        break;
      case constants.downArrow:
        // No operation other than preventing the default behavior of the arrow key.
        evt.preventDefault();
        break;
      case constants.leftArrow:
        surface.zRotate(-1);
        evt.preventDefault();
        break;
      case constants.rightArrow:
        surface.zRotate(1);
        evt.preventDefault();
        break;
    }
    return;
  }

  switch (evt.keyCode)
  {
    case constants.upArrow:
      surface.xRotate(1);
      evt.preventDefault();
      break;
    case constants.downArrow:
      surface.xRotate(-1);
      evt.preventDefault();
      break;
    case constants.leftArrow:
      surface.yRotate(-1);
      evt.preventDefault();
      break;
    case constants.rightArrow:
      surface.yRotate(1);
      evt.preventDefault();
      break;
  }
}
```

For example, if the control key and left arrow key are pressed simultaneously, the following code is executed:

JavaScript

```
case constants.leftArrow:
  surface.zRotate(-1);
  evt.preventDefault();
  break;
```

`surface.zRotate(-1)` rotates the surface about the *z*-axis in the counterclockwise direction. `evt.preventDefault()` prevents the default behavior of the arrow key, which is to scroll the browser's window when scrollbars are present. Note that a user can still scroll the window with the mouse.

The remaining implementation details are well described by the large number of detailed comments contained within the sample.

Exercises

In this section, we suggest two core exercises. The first involves reducing the size and number of points for the surface; the second involves implementing two additional surface transformations.

Surface reduction

The number of surface points in the sample is relatively large – 8100. For non-performant hardware, this can cause the rendering of the surface to appear slow, choppy, or both. In such situations, reducing the total number of points will improve perceived performance. This can be accomplished by adjusting the following constants, as indicated by the following JavaScript comments:

JavaScript

```
var constants = {
  canvasWidth: 600, // ADJUST
  canvasHeight: 600, // ADJUST
  leftArrow: 37,
  upArrow: 38,
  rightArrow: 39,
  downArrow: 40,
  xMin: -9,
  xMax: 9,
  yMin: -9,
  yMax: 9,
  xDelta: 0.2, // ADJUST
  yDelta: 0.2, // ADJUST
  colorMap: ["#060", "#090", "#0C0", "#0F0", "#9F0", "#9C0", "#990", "#960", "#930", "#900
  pointWidth: 2,
  dTheta: 0.05,
```

```
  surfaceScale: 24 // ADJUST
};
```

To reduce the size of the canvas, decrease `canvasWidth` and `canvasHeight`. To reduce the total number of points, increase `xDelta` and `yDelta`. After these adjustments have been made, empirically reduce `surfaceScale` such that the rendered surface fits nicely on the canvas. For example, the following set of constants produce a surface that fits nicely on a 400 x 400 canvas with a total of 3721 points (see JavaScript comments):

JavaScript

```
var constants = {
  canvasWidth: 400, // 600 TO 400
  canvasHeight: 400, // 600 TO 400
  leftArrow: 37,
  upArrow: 38,
  rightArrow: 39,
  downArrow: 40,
  xMin: -9,
  xMax: 9,
  yMin: -9,
  yMax: 9,
  xDelta: 0.3, // 0.2 TO 0.3
  yDelta: 0.3, // 0.2 TO 0.3
  colorMap: ["#060", "#090", "#0C0", "#0F0", "#9F0", "#9C0", "#990", "#960", "#930", "#900
  pointWidth: 2,
  dTheta: 0.05,
  surfaceScale: 16 // 24 TO 16
};
```

Try experimenting with these constants; possibly including `pointWidth` and `dTheta` (`dTheta` is the amount the surface is rotated about any given axis per key press). To adjust the "zoom factor" for the surface, experiment with `xMin`, `xMax`, `yMin`, and `yMax`.

Scaling and translation

As described previously, to rotate the surface about an axis, we perform:

$P' = RP$

Similarly, to scale the surface, we can perform the same matrix multiplication but with a different 3 x 3 transformation matrix *S*:

$$P' = SP, \text{ where } S = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & \gamma \end{bmatrix}$$

Here, *S* scales a view of the surface along the *x*, *y*, and *z* directions by factors of *α*, *β*, and *γ*, respectively. For example, to double the size of the surface in the *x* direction, use:

$$S = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The last transformation to consider is translation (or displacement). That is, moving the surface to a new position on the display device. To translate a view of the surface, we displace each surface point (x_i, y_i, z_i) to a new point by adding constants (one or more of which could be zero) to each point coordinate:

$(x_i + d_x, y_i + d_y, z_i + d_z)$

Here, d_x, d_y, and d_z represent the desired displacements in the *x*, *y*, and *z* directions, respectively. For example, to move the surface three units in the positive *x* direction, we let $d_x = 3$, $d_y = 0$, and $d_z = 0$. That is, we simply add three to each surface point's *x*-coordinate.

From a matrix perspective, translation can be accomplished using matrix addition and a 3 x *n* translation matrix *T* as follows:

$$P' = P + T, \text{ where } T = \begin{bmatrix} d_x & d_x & \cdots & d_x \\ d_y & d_y & \cdots & d_y \\ d_z & d_z & \cdots & d_z \end{bmatrix}$$

As the last suggested exercised, try adding scaling and translation options to the sample. For example, various buttons could be clicked to scale and translate the surface.

Related topics

HTML5 Graphics
HTML5 Canvas