Workshop: Advanced JSXGraph

Vol. 4

Alfred Wassermann

UNIVERSITÄT
BAYREUTH

24-02-2021

Contents

Preliminaries

Include JSXGraph

- JSXGraph skeleton page:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>JSXGraph template</title>
    <meta content="text/html; charset=utf-8" http-equiv="Content-Type">
    <link href="https://cdn.jsdelivr.net/npm/jsxgraph@1.2.1/distrib/
       jsxgraph.css" rel="stylesheet" type="text/css" />
    <script src="https://cdn.jsdelivr.net/npm/jsxgraph@1.2.1/distrib/
       jsxgraphcore.js" type="text/javascript" charset="UTF-8"></script>

    <!-- The next line is optional: MathJax -->
    <script src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-chtml.js"
       id="MathJax-script" async></script>
  </head>
  <body>

  <div id="jxgbox" class="jxgbox" style="width:500px; height:200px;"></div
     >

  <script>
    var board = JXG.JSXGraph.initBoard('jxgbox', {boundingbox: [-5, 2, 5,
       -2]});
  </script>

  </body>
</html>
```

- See JSXGraph handbook (in development): https://ipesek.github.io/jsxgraphbook/

Intersection, union, difference of paths

- Here, *path* can be a circle, curve (also function graph), or polygon.
- The easiest approach to visualize the *intersection* of paths is to work with `fillOpacity`, see https://jsfiddle.net/frkwzqh0/2/:

```
var c1 = board.create('circle', [[-1,1], 3], {
  fillColor: 'yellow', fillOpacity: 0.6,
  highlightFillColor: 'yellow', highlightFillOpacity: 0.2,
  hasInnerPoints: true});
```

```
var c2 = board.create('circle', [[1,-1], 3], {
  fillColor: 'red', fillOpacity: 0.6,
  highlightFillColor: 'red', highlightFillOpacity: 0.2,
  hasInnerPoints: true});
```

- However, if it is necessary to have access to the intersection / union / difference curve, the new (since v1.2.1) *clipping* methods of JSXGraph can be used. At the moment, this feature is implemented in a low level style. More comfortable methods might be added in the future.
- Clipping of paths is surprisingly complex. JSXGraph implements the algorithm by *Greiner-Hormann* (1998) and the improvements by *Foster, Hormann, Popa* (2019) to handle *degenerate intersections*.
- *Degenerate intersections* are vertices of one polygon which are on an edge of the other polygon. (Internally, JSXGraph treats a path as polygon).
- If you observe problems → please report at github or Google groups.
- **Example**: Create path of intersection of circles, https://jsfiddle.net/frkwzqh0/4/
  - Create an empty curve
  - Use the clipping methods from `JXG.Math.Clip` in the method `updateDataArray` of the curve
  - Call `board.update()`

```
var c1 = board.create('circle', [[-1,1], 3], {
  fillColor: 'yellow', fillOpacity: 0.6,
  highlightFillColor: 'yellow', highlightFillOpacity: 0.2,
  hasInnerPoints: true, strokeWidth: 1});
var c2 = board.create('circle', [[1,-1], 3], {
  fillColor: 'red', fillOpacity: 0.6,
  highlightFillColor: 'red', highlightFillOpacity: 0.2,
  hasInnerPoints: true, strokeWidth: 1});

var c3 = board.create('curve', [[], []], {
    strokeWidth: 4,
    fillColor: 'blue',
    layer: 7});

c3.updateDataArray = function() {
    var a = JXG.Math.Clip.intersection(c1, c2, this.board);

    this.dataX = a[0];
    this.dataY = a[1];
};
board.update();
```

- **Example**: intersection / union / difference of curve and circle, https://jsfiddle.net/47fq6dLy/

```
var f = board.create('functiongraph', ['x^2']);
```

```
var c = board.create('circle', [[0,0], 2]);

var c3 = board.create('curve', [[], []], {
    strokeWidth: 1, fillColor: 'yellow', fillOpacity: 0.4});

c3.updateDataArray = function() {
    var a = JXG.Math.Clip.intersection(f, c, this.board);

    this.dataX = a[0];
    this.dataY = a[1];
};
board.update();
```

- The method `JXG.Math.Clip.windingNumber()` may be useful to decide if a point is inside of a polygon. The answer is yes if and only if the winding number is odd. Here is an example, https://jsfiddle.net/jLk7v2xo/:

```
var p = board.create('polygon', [[-2,-3], [2,-1], [3,2], [-2,4], [1,1]]);
var A = board.create('point', [-2, 1]);
var txt = board.create('text', [1, -3, function() {
  var inside = JXG.Math.Clip.windingNumber(A.coords.usrCoords, p.vertices)
      % 2 === 1;

  if (inside) {
    return 'in';
  } else {
    return 'out';
  }
}], {fontSize: 24});
```

Transformations

Euclidean Geometry

- Use of the JSXGraph objects `reflection` or `mirrorelement`.
- Reflections, translations or rotations of objects in Euclidean Geometry are examples of *transformations*.
- Lines are mapped on to line, circles onto circles, …
- Angles stay the same under those transformations.
- Example, see https://jsfiddle.net/tgen2mbd/:

```
var line = board.create('line', [[-2, -2], [2, 2]], {
    point1: {visible:true, color:'red', layer:8},
    point2: {visible:true, color:'red', layer:8}});
```

```
var p1 = board.create('polygon', [[1,-3], [3,-3], [2, -1.5]], {
    fillColor:'yellow',
    hasInnerPoints:true});
var c1 = board.create('circumcircle', [
    p1.vertices[0],
    p1.vertices[1],
    p1.vertices[2]]);
var p2 = board.create('reflection', [p1, line]);
var c2 = board.create('reflection', [c1, line]);
```

- The same is possible for mirroring elements about a point, https://jsfiddle.net/tgen2mbd/3/

```
var P = board.create('point', [1, 0]);

var p1 = board.create('polygon', [[1,-3], [3,-3], [2, -1.5]], {
    fillColor:'yellow',
    hasInnerPoints:true});
var c1 = board.create('circumcircle', [
    p1.vertices[0],
    p1.vertices[1],
    p1.vertices[2]]);
var p2 = board.create('mirrorelement', [p1, P]);
var c2 = board.create('mirrorelement', [c1, P]);
```

Manipulate JSXGraph objects with transformation

- Objects to which a transformation can be applied are: point, line, circle, curve, polygon.
- There are three possibilities to use transformations:

  1) apply the transformation once
  2) bind the transformation to an object
  3) define an object as the transformed object of a base object.

Apply a transformation just once.

This Has the same effect as dragging the object or moving it with `moveTo`. This is done with `applyOnce`, see https://jsfiddle.net/tqf0j8pe/. Do not forget to update the board.

```
var sl = board.create('slider', [[0,3], [3,3], [0, Math.PI / 3, 2*Math.PI
   ]], {name:'&alpha;'});

var A = board.create('point', [2, 0]);
var T = board.create('transform', [()=>sl.Value()], {type: 'rotate'});
```

```
T.applyOnce(A);
board.update();
```

Bind a transformation to an object

The other possibility is to bind the transformation to an object. Then, this transformation is applied in every update of the board.

- **Example** for `bindTo`, https://jsfiddle.net/n7ef8pcr/

```
var sl = board.create('slider', [[0,3], [3,3], [0, 0, 2*Math.PI]], {name:'
    &alpha;'});

var A = board.create('point', [2, 0]);
// Rotate around (1, 1):
var T = board.create('transform', [()=>sl.Value()], {type: 'rotate'});
var T1 = board.create('transform', [1, 1], {type: 'translate'});
var T2 = board.create('transform', [-1, -1], {type: 'translate'});

T2.bindTo(A);
T.bindTo(A);
T1.bindTo(A);
```

- An object can receive several transformations.
- The transformations are stored in an array `object.transformations`.

Applying transformations is a *non-commutative* operation, i.e. the order in which the transformations are applied matters.

Create objects as images of transformations

- Point: https://jsfiddle.net/hynmL3dr/

```
var sl = board.create('slider', [[0,3], [3,3], [0, 0, 2*Math.PI]], {name:'
    &alpha;'});

var A = board.create('point', [2, 0]);
var T = board.create('transform', [()=>sl.Value()], {type: 'rotate'});
var T1 = board.create('transform', [1, 1], {type: 'translate'});
var T2 = board.create('transform', [-1, -1], {type: 'translate'});

// Rotate around (1,1)
var B = board.create('point', [A, [T2, T, T1]]);
```

- Polygon: https://jsfiddle.net/xn062m8z/

```
var line = board.create('line', [[-2, -2], [2, 2]], {
    point1: {visible:true, color:'red', layer:8},
    point2: {visible:true, color:'red', layer:8}});

var T = board.create('transform', [line], {type: 'reflect'});

var p1 = board.create('polygon', [[1,-3], [3,-3], [2, -1.5]], {
  fillColor:'yellow',
  hasInnerPoints:true});
var p2 = board.create('polygon', [p1, T]);
```

- Other objects: curves, lines, images, texts and circles
  - images: see https://jsxgraph.org/wiki/index.php/Images_and_Transformations for an elaborated example.
  - texts: see https://jsxgraph.org/wiki/index.php/Texts_and_Transformations and https://jsxgraph.org/wiki/index.php/Texts_and_Transformations_II

Available transformation types

- `translate`
- `scale`
- `reflect`
- `rotate`
- `shear`
- `generic`

In general, the image of a circle under a transformation is not longer a circle. Therefore, the transformed circle is of JSXGraph class *curve*.

Background on transformations

Transformations are realized as matrix multiplications of coordinate vectors. For example, a JSXGraph point has coordinates `[z, x, y]`. The z-coordinate is always equal to 1, except if the point is an infinite point. These types of coordinates are called *homogeneous coordinates*. Every transformation matrix has size 3×3.

Since matrix multiplication is non-commutative, it is now also clear why the order matters in which the transformations are applied.

In SVG or canvas, transformations are realized in the same way with matrices – but the coordinates of the elements are ordered `[x, y, z]`. For the matrix, this means the first row and first column of a JSXGraph transformation matrix is now the last row and last column of the SVG matrix, where entry $(0,0)$ becomes entry $(3,3)$ in SVG.

The JSXGraph matrices belonging to the transformations have the following form:

- **Translation** matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ b & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} z \\ x \\ y \end{bmatrix} = \begin{bmatrix} z \\ x+a \\ y+b \end{bmatrix}$$

- **Scale** matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & b \end{bmatrix} \cdot \begin{bmatrix} z \\ x \\ y \end{bmatrix} = \begin{bmatrix} z \\ ax \\ by \end{bmatrix}$$

- **Rotation** matrix with angle a (in Radians) around $(0,0)$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(a) & -\sin(a) \\ 0 & \sin(a) & \cos(a) \end{bmatrix} \cdot \begin{bmatrix} z \\ x \\ y \end{bmatrix} = \begin{bmatrix} z \\ \cos(a)x - \sin(a)y \\ \sin(a)x + \cos(a)y \end{bmatrix}$$

- **Shear** matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & a \\ 0 & b & 1 \end{bmatrix} \cdot \begin{bmatrix} z \\ x \\ y \end{bmatrix} = \begin{bmatrix} z \\ x+ay \\ y+bx \end{bmatrix}$$

- **Generic** transformation matrix:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \cdot \begin{bmatrix} z \\ x \\ y \end{bmatrix}$$

Advanced example: offside in soccer

This elaborated example uses *generic transformations* to display the "offside" line in a soccer game. The *camera image* is "distorted" by the viewing angle and must be "rectified" into a rectangle in the *model view*. In the model view it is easy to draw vertical lines or similar objects. Finally, the whole figure has to be transformed back into the view of the camera. This is what the watchers finally see on *TV*.

The 3×3 transformation matrix can be computed by solving a linear system with 13 unknowns and 13 equations. Thanks to Roman Hašek for the inspiration.

Example: https://jsfiddle.net/5pxedfzh/

```
    var board1 = JXG.JSXGraph.initBoard('jxgbox1', {boundingbox: [-10, 10,
        10, -10]});
    var board2 = JXG.JSXGraph.initBoard('jxgbox2', {boundingbox: [-10, 10,
        10, -10]});
    var board3 = JXG.JSXGraph.initBoard('jxgbox3', {boundingbox: [-10, 10,
        10, -10]});

    board1.addChild(board2);
    board1.addChild(board3);


    // Polygon in camera image
    var p1 = board1.create('polygon', [[-8,-6], [9,-6], [5,6.5], [-6,7]],
       {fillColor: 'green'});

    // Polygon in model view
    var p2 = board2.create('polygon', [[-5,-4], [5,-4], [5,4], [-5,4]], {
       fillColor: 'green',
        vertices: {visible: false, fixed: true}});

    // Polygon on TV screen (copy of p1)
    var p3 = board3.create('polygon', [
             [()=>p1.vertices[0].X(), ()=>p1.vertices[0].Y()],
             [()=>p1.vertices[1].X(), ()=>p1.vertices[1].Y()],
             [()=>p1.vertices[2].X(), ()=>p1.vertices[2].Y()],
             [()=>p1.vertices[3].X(), ()=>p1.vertices[3].Y()]
           ], {fillColor: 'green', vertices: {visible: false}});

    //
    // Compute a projective transformation which maps the polygon p1 to
       the polygon p2.
    //

    // Two global variables containing the transformation matrix (in
       vector and in matrix form)
    var x_global = [],
```

```
        x2_global = [],
        mat_global = [[0,0,0], [0,0,0], [0,0,0]],
        mat2_global = [[0,0,0], [0,0,0], [0,0,0]];

    // This function computes the transformation matrix
    var updateTransformationMatrix = function() {
        var i, j, k, M = [];

        // Initialise a 13x13 matrix to zero.
        for (i = 0; i < 13; i++) {
            M.push([0,0,0,0,0, 0,0,0,0,0, 0,0,0]);
        }

        // Set up the system of linear equations:
        // 12 equations and 13 unknowns for the matrix
        // mat_global such that
        // mat_global * p1 - p2 * (i, j, k, l)^T = 0
        for (i = 0; i < 3; i++) {
            for (j = 0; j < 3; j++) {
                for (k = 0; k < 4; k++) {
                    M[i * 4 + k][i * 3 + j] = p1.vertices[k].coords.
                        usrCoords[j];
                }
            }
        }
        for (i = 0; i < 3; i++) {
            for (k = 0; k < 4; k++) {
                M[i * 4 + k][9 + k] = -p2.vertices[k].coords.usrCoords[i];
            }
        }
        // Equation 13: set mat_global[0][0] = 1.
        // Remember that in JSXGraph the coordinates are ordered by (z, x,
            y)
        M[12][0] = 1;

        // Right hand side vector
        var b = [0,0,0,0,0, 0,0,0,0,0, 0,0,1];

        // Solve the system
        x_global = JXG.Math.Numerics.Gauss(M, b);

        // Convert the solution vector into matrix form
        for (i = 0; i < 3; i++) {
            for (j = 0; j < 3; j++) {
                mat_global[i][j] = x_global[i * 3 + j];
            }
        }

        // Invert the matrix to get the inverse transform
        mat2_global = JXG.Math.inverse(mat_global);
```

```
        // Convert the matrix into vector form
        for (i = 0; i < 3; i++) {
            for (j = 0; j < 3; j++) {
                x2_global[i * 3 + j] = mat2_global[i][j];
            }
        }
    };

    // First computation of the transformation matrix
    updateTransformationMatrix();

    // Store the transformation vectors in functions
    // in order to make the JSXGraph transformation dynamic
    var x_fcts = [], x2_fcts = [];
    for (let i = 0; i < 9; i++) {
        x_fcts[i]  = () => x_global[i];
        x2_fcts[i] = () => x2_global[i];
    }

    // Create the transform from p1 to p2 and its inverse.
    var transform  = board1.create('transform', x_fcts, {type: 'generic'})
        ;
    var transform2 = board2.create('transform', x2_fcts, {type: 'generic'
        });

    // Whenever a point of p1 is dragged, the transformation matrix and
        its
    // inverse will be updated.
    for (let i = 0; i < 4; i++) {
        p1.vertices[i].on('drag', updateTransformationMatrix);
    }

    var player = { fillColor: 'white', strokeColor: 'black', size:6 };
    // Set a point in the camera image
    var A = board1.create('point', [-2.5, 0], player);
    // Create its image and a vertical line through the point in the model
    var A2 = board2.create('point', [A, transform], player);
    var line1 = board2.create('segment', [
            [()=>A2.X(), -4],
            [()=>A2.X(), 4]], {strokeColor:'yellow'});

    // Transform the point and the line into the TV screen
    var A3 = board3.create('point', [A2, transform2], player);
    var line2 = board3.create('segment', [line1, transform2], {strokeColor
        :'yellow'});
```

Discussion and suggestion of further topics

Next webinar

The next webinar will be **Wednesday, March 24th, 2021 at 4 pm CET**