Efficient Handling of Polygon Data With The ES6 Proxy Object

Lightweight Technique for Data Conversion, Topological and Geometric Transformations.

Stefan Gössner[1]

[1]Dortmund University of Applied Sciences. Department of Mechanical Engineering

August 2021

Keywords: polygon, point array, ES6, iterator, generator, proxy, geometry

Abstract

There is no standard data format for polygon points used in geometry applications or JSON documents. In a real world application, you don't always have control over the data format provided by the user. A method is sought that returns points in a uniform, desired target format when accessing arrays with different data formats. Costly duplication of data is to be avoided. This paper presents an elegant method to achieve this goal using the JavaScript "Proxy" object. Here focusing on 2D points, this concept can be easily generalised to arrays of 3D points or other complex, homogeneous data types.

1. Introduction

2D points are normally described by Cartesian x- and y-coordinates. Mostly they are interpreted as polygon vertices, but this might be irrelevant here. The number of points is considered greater than one; they are usually stored in an Array.

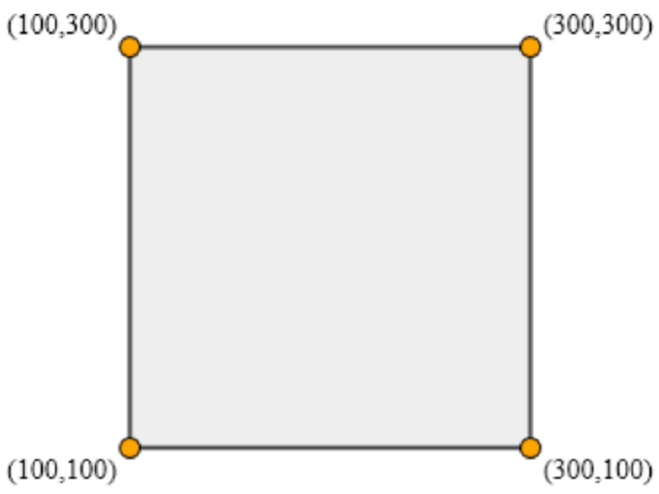

Fig 1: Four Points as Polygon Data.

Different representations used in practice range from separate arrays for x- and y-coordinates to individual objects for the points.

```
// separate arrays for x- and y-coordinates ... somewhat exotic
const xCoords = [100,300,300,100];
const yCoords = [100,100,300,300];

// flat array holding alternating x- and y-coordinates
const flat = [100,100,300,100,300,300,100,300];

// array holding point arrays
const arr = [[100,100],[300,100],[300,300],[100,300]];

// array holding point objects
const obj = [{x:100,y:100},{x:300,y:100},{x:300,y:300},{x:100,y:300}];
```

Listing 1: Different representations of 2D Points.

The aim is now to have unified access to those different point formats. Point access (read only) should ...

1. work the same way as used with arrays, i.e. `points[i]`.
2. return the point in a specific target format.
3. work with different loops provided by the language.

Additionally `Array` methods like `find`, `filter`, etc. should work as expected.

2. Concepts

Starting with ES6, Iterators, Generators and Proxies are official JavaScript features [1]. We want to examine them for suitability according to the three goals from above.

For this we start with two different Arrays for x- and y-coordinates and want to access points as `{x,y}` objects from them, i.e.

```
// separate arrays for x- and y-coordinates
const xCoords = [100,300,300,100];
const yCoords = [100,100,300,300];
// access by something called `points`, returning `{x,y}` objects ...
for (let i=0; i<points.length; i++)  // traditional loop
     console.log(points[i]);
for (let p of points)                // for..of loop
     console.log(p);
console.log(...points);              // spread operator
```

Listing 2: Unified Array Access Returning Point Objects.

Throughout this text we are allowed to assume for simplicity, that Arrays `xCoords` and `yCoords` are always of the same dimension.

2.1 Iterators

> Iterators bring the concept of iteration directly into the core language and provide a mechanism for customizing the behavior of `for...of` loops.
> – MDN [2]

We implement both, the iterable protocol and the iterator protocol [3].

```
function iterator(xarr, yarr) {
    let index = 0;
    return {
        next() {
            return index < xarr.length
                ? { value: { x:xarr[index], y:yarr[index++] }, done: false }
                : { done: true };
        },
        [Symbol.iterator]() { return this; }
    }
}
// usage ...
const points = iterator(xcoords, ycoords);
```

Listing 3: Iterator object returned by function.

Variable `points` is now holding an Iterator object returned by the `iterator` function. Reusing it in the example from Listing 2 shows, that `for..of` loop as well as spread operator works as expected. Single element access is possible by `points.next()` method, but standard array element access `points[i]` and with it the traditional `for(;;)` loop is not supported.

If we want to reuse `points` multiple times, we simply rewrite `{ done: true }` to `{ done: !(index = 0) }`.

An Iterator is no Array, so Array methods – except static `Array.from()` function – do not work with it.

2.2 Generators

Generator objects are very similar to Iterators and even somewhat more elegant. Generators conform to both the iterable protocol and the iterator protocol. They cannot be instantiated directly, instead they are returned from a generator function [4].

```
function* generator(xarr, yarr) {
    for (let i=0; i<xarr.length; i++)
        yield {x: xarr[i], y: yarr[i]};
}
// usage ...
const points = generator(xcoords, ycoords);
```

Listing 4: Generator object implicitly returned by function.

Variable `points` from Listing 4 holds a Generator object implicitly returned by the `generator` function. Reusing it in Listing 2 example shows behavior identical to Iterators. Only `for..of` loop and spread operator work as expected.

Please note, that generators cannot be reset to initial state. We need to call `generator(xcoords, ycoords)` each time.

2.3 Proxy Object

The `Proxy` object enables us to create a proxy for another object, which can intercept and redefine fundamental operations for that object. We create a `Proxy` object with `new Proxy(target, handler)`. Herein `target` is the original object and `handler` an object with a predefined – not extensible – interface [5].

In our context we don't have a single object (array) to create a proxy for, but two. So it reads:

```
const proxy = function(xarr, yarr) {
    return new Proxy([], {
        get: (pts, key) => {
            if (!isNaN(+key))
                return {x:xarr[+key],y:yarr[+key]}
            else
                return xarr[key];
        }
    });
}
// usage ...
const points = proxy(xcoords, ycoords);
```

Listing 5: Proxy object returned by function.

Variable `points` is now holding a Proxy object returned by the `proxy` function. Reusing it in the example from Listing 2 shows, that usual array access `points[i]`, dimension property `length` and with it the traditional `for(;;)` loop works as expected. `Array` methods are also supported – at least the non-modifyable ones get correct results due to read-only access of the proxy.

On the other hand `Proxy` object does not support the iteration protocols, so `for..of` loop and spread operator is not working. Applying `console.log(...points)` results in `"TypeError: can't convert symbol to number"`.

2.4 Intermediate Results

Applying the `Proxy` object to 2D point data provides promising results and is clearly superior to `Iterator` and `Generator` objects. It behaves as if it were an array itself.

Table 1: Comparison of Iterator, Generator and Proxy object behavior.

Apply	Iterator	Generator	Proxy
`arr[i]`	–	–	✓
`arr.length`	–	–	✓
`for(;;)`	–	–	✓
`Array.isArray`	–	–	✓
`for..of`	✓	✓	–
`Array.from`	✓	✓	–
`JSON.stringify`	–	–	✓

The only missing thing is lack of support of the iteration protocols.

2.5 Proxy Object Enhanced

The `Proxy` object does not support the iterable protocol and the iterator protocol natively. Let us try to implement them to make the example in Listing 2 work as a whole.

```
const proxy = function(xarr, yarr) {
    const points = new Array(~~xarr.length);
    points[Symbol.iterator] = function() {
        let index = 0;
        return {
            next() {
                return index < xarr.length
                    ? { value: { x:xarr[index], y:yarr[index++] },
                        done: false }
                    : { done: true };
            }
        }
    }
    return new Proxy(points, {
        get: (pts, key) => {
            if (key === Symbol.iterator)
                return points[Symbol.iterator].bind(points);
            else if (!isNaN(+key))
                return {x:xarr[key],y:yarr[key]};
            else
                return xarr[key];
        }
    });
}
```

Listing 6: Proxy object implementing iteration protocols.

In our `proxy` wrapper function in Listing 6 we add a helper array `points` with correct `length` property and implement a custom `[Symbol.iterator]` method on it. In the `Proxy`'s `get` method the `Symbol.iterator` key is handled explicitly then.

Please note, that on line 2 of Listing 6 in `new Array(~~xarr.length)` the double `~~` operator efficiently works like `Math.floor` and the `Array` constructor only reserves empty slots and should not allocate memory [6].

These additions are sufficient to enable our `points` object from Listing 2 to deal with `for..of` loops and spread operator correctly.

It is perfectly mimicking an array now – restricted to read access only though. So `points.find((p)=>Math.hypot(p.x,p.y) > 400)` correctly yields the point `{x:300,y:300}`.

Modifying the array doesn't work, as our proxy handler does not implement the `set` method. In fact, we do not want that either, as it is considered bad practice. Modifications should be made transparently to the original array.

3. Practical Applications

First we want to move to a more realistic polygon structure contained in a single array. This requires a modification of our proxy implementation.

```
const polygon = [100,100,300,100,300,300,100,300]; // flat coordinates array

const proxy = function(poly) {
    const pnts = new Array(~~(poly.length/2));
    pnts[Symbol.iterator] = function() {
        ...
    }
    return new Proxy(poly, {
        get: (pts, key) => {
            if (key === Symbol.iterator)
                return pnts[Symbol.iterator].bind(pnts);
            else if (!isNaN(+key))
                return {x:pts[key*2],y:pts[key*2+1]};
            else if (key === 'length')
                return  ~~(pts.length/2);
            else
                return pts[key];
        }
    });
}

const points = proxy(polygon);
// [{"x":100,"y":100},{"x":300,"y":100},{"x":300,"y":300},{"x":100,"y":300}]
```

Listing 7: Proxy object for a flat points array.

Analogously to this we can adapt the implementation in Listing 7 for an array holding point arrays as well. An array containing point objects `{x,y}` needs no proxy at all, as it already contains points in our target format.

3.1 Transformations

Our `proxy`'s can not only handle various user-side polygon data formats, but also manage more advanced transformations. Mostly only the change of one line in Listing 7 is necessary.

```
// Reverse points ...
      ...
      else if (!isNaN(+key)) {
         const i = pts.length - 1 - key;
         return {x:pts[i].x, y:pts[i].y};
      }
      ...

// Polar coordinates ...
      ...
      else if (!isNaN(+key)) {
         const p = pts[+key];
         return {r:Math.hypot(p.x,p.y), w:Math.atan2(p.y,p.x)};
      }
      ...

// Rotate polygon points about `x0,y0` by angle `w` in radians ...
const rotPoly = function(poly,w,x0=0,y0=0) {
    ...
    const sw = Math.sin(w), cw = Math.cos(w);
    const x = (1-cw)*x0 + sw*y0, y = -sw*x0 + (1-cw)*y0;
    return new Proxy(poly, {
        get: (pts, key) => {
            ...
            else if (!isNaN(+key)) {
                const p = pts[i];
                return {x: p.x*cw - p.y*sw + x, y: p.x*sw + p.y*cw + y};
            }
            ...
```

The charming fact with this approach is, that there is no mutation or copying of geometry data necessary. So the reverse proxy might be even a better solution than applying native `Array.reverse` method, since the latter transposes array elements in place.

3.2 Concatenation

Another good thing is the possibility of serial combination of transformations. Let

- `flat` be the proxy handling a flat points array,
- `rev` be the proxy delivering points in reverse order,
- `rot` be the rotation of points by angle `w` about center `x0,y0`,

then

```
rot(rev(flat(poly)),w,x0,y0);
```

will deliver rotated polygon points in reversed order and in object format `{x,y}` from its original flat coordinates array in a time and memory efficient way.

4. Conclusion

Users polygon data format cannot always be controlled by the application. So different approaches for transforming points to a unified target format in a memory and time efficient way is discussed in this paper.

The comparison of ES6 Iterator, Generator and Proxy object shows the clear superiority of `Proxy` for this task. After equipping the `Proxy` object with the iterable protocol and the iterator protocol, it behaves identically to a JavaScript `Array`.

The lightweight and easy to implement `Proxy` technique described is not only useful for data conversion, but also for topological (point order) and geometric (rotate, scale, translate) transformations.

The source code to Listing 2...6 can be found on GitHub [7]. The HTML page to this paper [8] is generated by Markdown+Math, which is documented [9] and available on GitHub [10].

References

[1] ECMA-262 6th Edition (https://262.ecma-international.org/6.0/)
[2] MDN - Iterators and generators (tinyurl.com/2d3v9sbr)
[3] MDN - Iteration protocols (https://tinyurl.com/2a54tedb)
[4] MDN - Generator (https://tinyurl.com/e8xyz2vy)
[5] MDN - Proxy (https://tinyurl.com/3ywxevcy)
[6] MDN - Array constructor (https://tinyurl.com/4scaba6e)
[7] polygon-data (https://github.com/goessner/polygon-data)
[8] Polygon Data (https://goessner.github.io/polygon-data/)
[9] Markdown+Math (https://goessner.github.io/mdmath/)
[10] mdmath (https://github.com/goessner/mdmath)