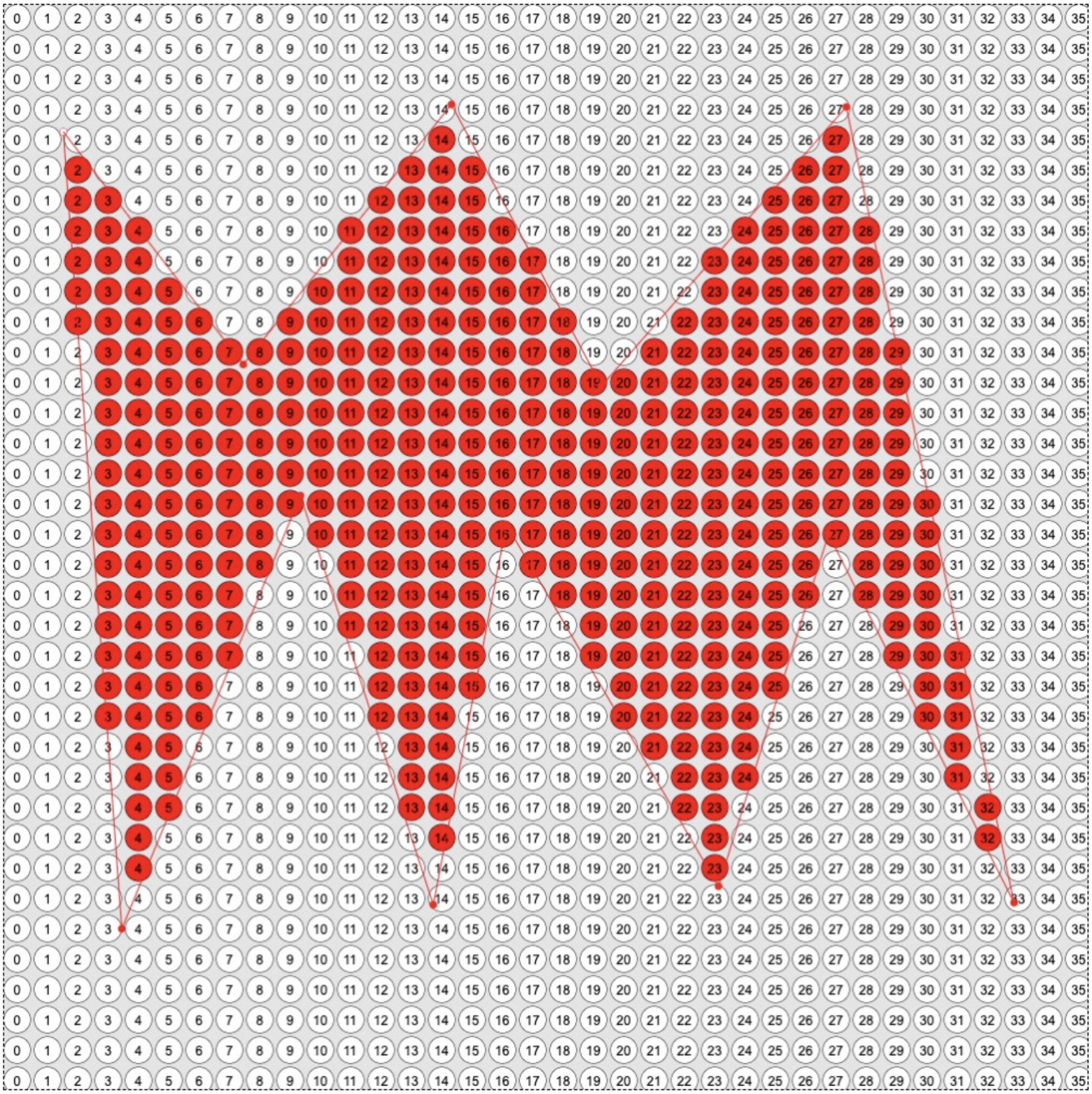
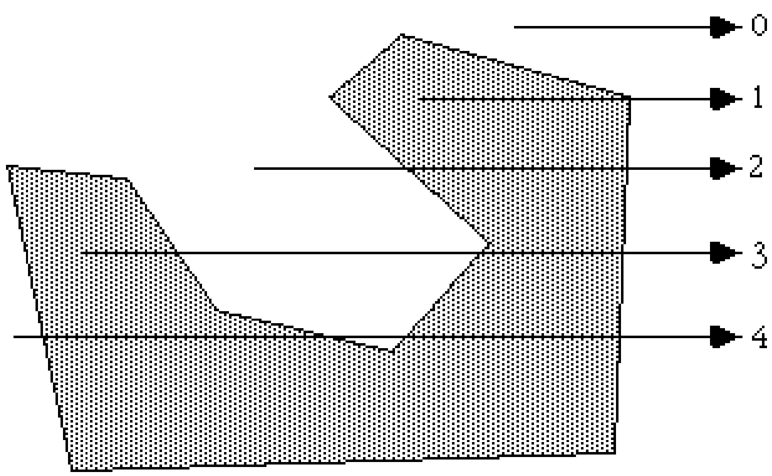
Javascript Vs Webassembly

套索(Lasso)选座工具下有一个很依赖CPU的算法，碰撞座位是否在用户画的不规则polygon内，如图:



其中碰撞算法采用 Jordan曲线定理的射线法，js 实现：

```javascript
function isPointInPoly (poly, pt) {
  for(var c = false, i = -1, l = poly.length, j = l - 1; ++i < l; j = i)
    ((poly[i].y <= pt.y && pt.y < poly[j].y) || (poly[j].y <= pt.y && pt.y < poly[i].y))
  && (pt.x < (poly[j].x - poly[i].x) * (pt.y - poly[i].y) / (poly[j].y - poly[i].y) + poly[i].x)
  && (c = !c);
  return c;
}
```



原理很简单：**从测试点水平运行半无限光线（增加x，固定y），并计算它穿过的边数。 在每个交叉点，射线在内部和外部之间切换。 这被称为Jordan曲线定理。**
每当水平光线穿过任何边缘时，变量c从0切换到1并且从1切换到0。 所以基本上它是跟踪交叉的边数是偶数还是奇数。 0表示偶数，1表示奇数。

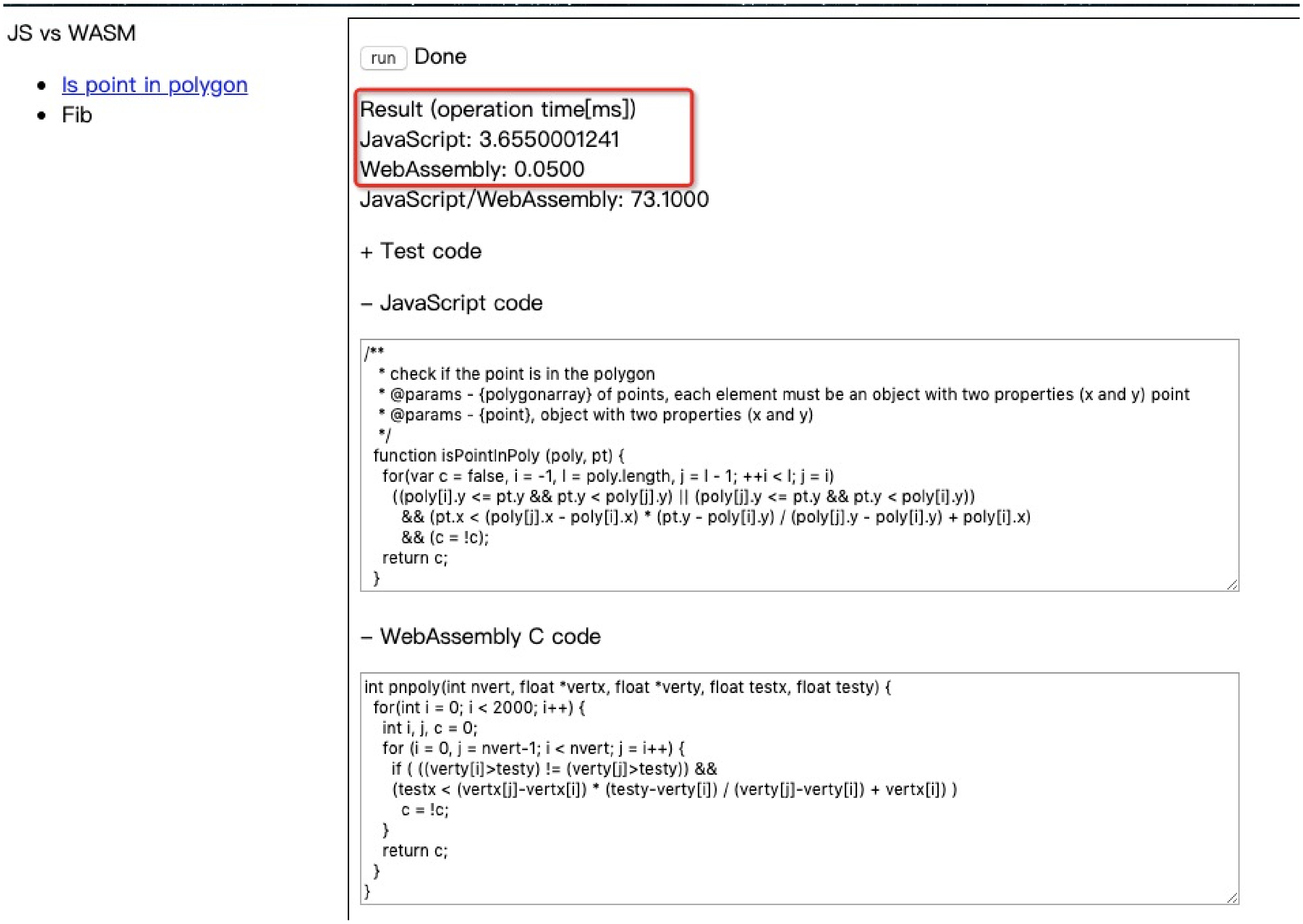
考虑到当准备被碰撞的座位很大 这一实际场景时，顾虑到这个算法可能会占据太多scripting time，减少CPU的计算，或者提高运算效率，首先考虑到的就是 **webassembly**~

综合社区内对 webassembly 的适用场景的总结：**单次做复杂运算，和JS线程通讯次数很低的场景，** webassembly的效率完胜javascript ，所以尝试采用相同算法用C语言实现，编译成wasm对比下效率：

测试Case：
传入一个10个节点的polygon，和一个固定测试Point: {x: 100, y: 100}，循环调用function N次
翻译为实际场景就是，用户在一个N个座位的看台内画了一个有10个节点的 polygon，判断多少个座位在polygon内

javascript: 循环调用isPointInPoly方法2000次。
C: pnpoly function内自循环2000次，减少和JS线程的通信次数。

如下为测试结果： **wasm完胜**



JS vs WASM

- Is point in polygon
- Fib

run Done

Result (operation time[ms])
JavaScript: 3.6550001241
WebAssembly: 0.0500
JavaScript/WebAssembly: 73.1000

+ Test code

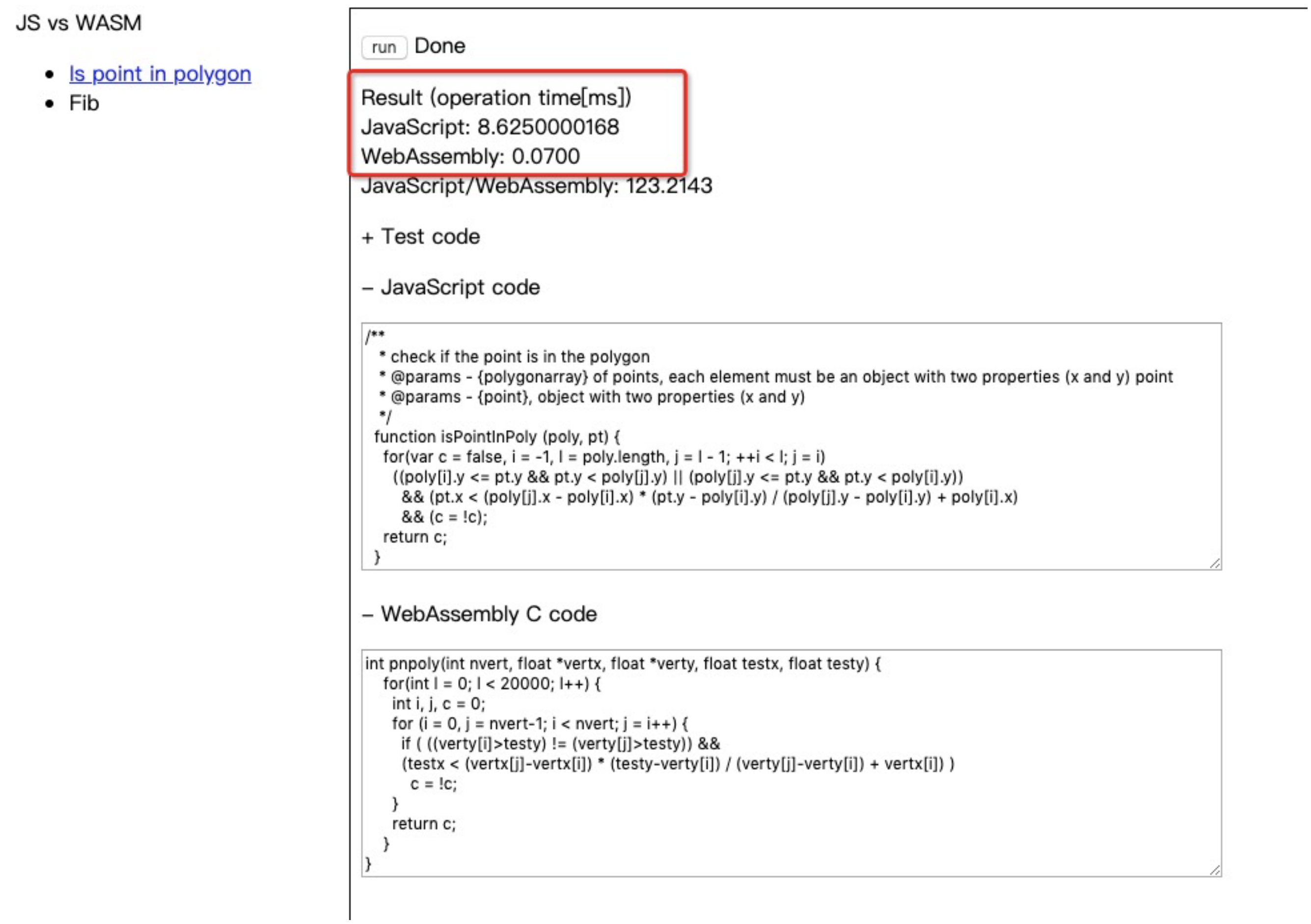
– JavaScript code

```
/**
  * check if the point is in the polygon
  * @params - {polygonarray} of points, each element must be an object with two properties (x and y) point
  * @params - {point}, object with two properties (x and y)
  */
 function isPointInPoly (poly, pt) {
  for(var c = false, i = -1, l = poly.length, j = l - 1; ++i < l; j = i)
   ((poly[i].y <= pt.y && pt.y < poly[j].y) || (poly[j].y <= pt.y && pt.y < poly[i].y))
    && (pt.x < (poly[j].x - poly[i].x) * (pt.y - poly[i].y) / (poly[j].y - poly[i].y) + poly[i].x)
    && (c = !c);
  return c;
 }
```

– WebAssembly C code

```
int pnpoly(int nvert, float *vertx, float *verty, float testx, float testy) {
 for(int i = 0; i < 2000; i++) {
  int i, j, c = 0;
  for (i = 0, j = nvert-1; i < nvert; j = i++) {
   if ( ((verty[i]>testy) != (verty[j]>testy)) &&
   (testx < (vertx[j]-vertx[i]) * (testy-verty[i]) / (verty[j]-verty[i]) + vertx[i]) )
    c = !c;
  }
  return c;
 }
}
```

把N改成2w，再对比一下较大数据量的差异：


JS vs WASM

- Is point in polygon
- Fib

run Done

Result (operation time[ms])
JavaScript: 8.6250000168
WebAssembly: 0.0700
JavaScript/WebAssembly: 123.2143

+ Test code

– JavaScript code

```
/**
  * check if the point is in the polygon
  * @params - {polygonarray} of points, each element must be an object with two properties (x and y) point
  * @params - {point}, object with two properties (x and y)
  */
 function isPointInPoly (poly, pt) {
   for(var c = false, i = -1, l = poly.length, j = l - 1; ++i < l; j = i)
     ((poly[i].y <= pt.y && pt.y < poly[j].y) || (poly[j].y <= pt.y && pt.y < poly[i].y))
       && (pt.x < (poly[j].x - poly[i].x) * (pt.y - poly[i].y) / (poly[j].y - poly[i].y) + poly[i].x)
       && (c = !c);
   return c;
 }
```

– WebAssembly C code

```
int pnpoly(int nvert, float *vertx, float *verty, float testx, float testy) {
   for(int l = 0; l < 20000; l++) {
     int i, j, c = 0;
     for (i = 0, j = nvert-1; i < nvert; j = i++) {
       if ( ((verty[i]>testy) != (verty[j]>testy)) &&
       (testx < (vertx[j]-vertx[i]) * (testy-verty[i]) / (verty[j]-verty[i]) + vertx[i]) )
         c = !c;
     }
     return c;
   }
}
```

可以看到，wasm 的计算性能在这个场景下远超于javascript。
回归到业务场景，虽然webassembly 的性能高于纯JS几十倍，甚至上百倍，但是这个算法在纯JS引擎下毕竟还是毫秒级别的，并且上面的Demo仅是一个测试，还不是实际业务场景，还没有处理wasm的内存分配、js传参，这个复杂度并不低。

JS VS WASM的项目地址: http://gitlab.alibaba-inc.com/aseat/aseat-benchmark