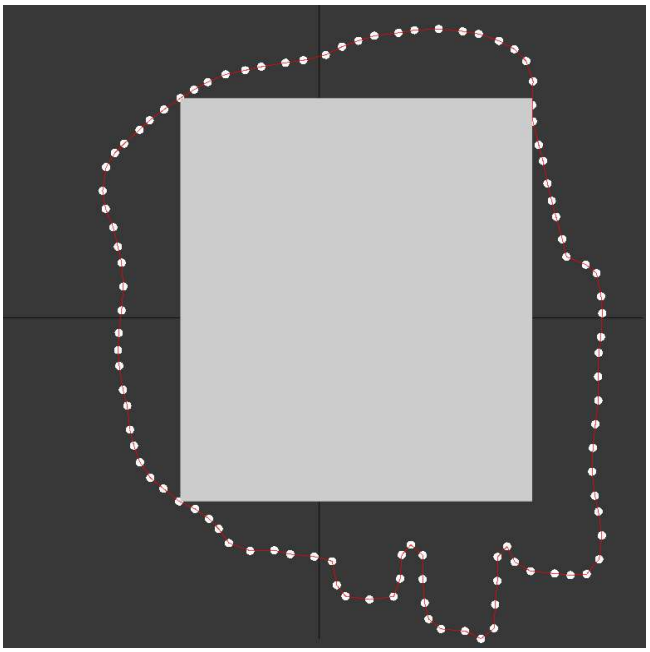
在一个简单的多边形中找到轴对齐的最大内部矩形

本文介绍了在简单多边形内找到轴对齐的最大内部矩形问题的解决方案。该解决方案是在 Unity 3D 中使用 C# 实现的。有一些数学、一些图片和一个完整工作实现的链接。

抽象的

简单多边形中最大的内部矩形是许多领域和应用中的有用信息。当我在寻找问题的解决方案时，我发现了与肉类工厂和 T 恤设计相关的文章，以及我试图解决的特定问题 - 我的 VR 房间边界内最大的矩形空间。据我所知，这个问题没有简单、快速的解决方案，所以 这个实现可能很有用。如果您确实发现它有用，并且您对其进行了改进，请分享改进！



一、简介

Unity 近期最大的变化之一是从与特定硬件（例如 Oculus 耳机）对话的内置代码转向使用“XR 插件”架构，该架构允许制造商创建和更新他们自己的无需 Unity 发布新版本的 Unity 编辑器的代码。

这种新架构背后的动机是好的——现在的实现几乎与以前的（遗留的，现在已弃用的）做事方式相当。然而，给我带来问题的一个领域是边界数据。

当耳机用户设置边界（或守护者）区域时，他们会在空间中绘制出一个可供他们安全玩耍的区域。这通常可以帮助玩家避免进入不安全的区域——例如，脸先撞墙，或者下楼梯，或者用控制器砸碎电视。这个边界区域通常有两种风格；原始“边界数据”（用户在安全区域外部绘制的一条不稳定的线）和内部“游戏数据”（通常是边界内的矩形区域）。

我的一些原型出于特定原因使用矩形区域（例如，将菜单锁定在墙上）。不幸的是，最新的 XR 插件不再给我这些数据——我只能访问完整的边界数据。

本文详细介绍了算法和实现细节。与此同时，我还写了一篇关于解决这个问题的过程的文章，你可能会觉得很有趣。

2. 背景

我要解决的问题是在 2D 平面中取一组点来描述一个*简单的多边形*，然后从这些点计算该简单多边形所包含的*最大内部矩形*。它是最大空矩形 问题的一个子类。

简单多边形是可以是凸面或凹面、不与自身相交且没有孔的多边形。

轴对齐的矩形是一个四边形，有两对平行的边，每对边的长度相同。其中一对平行于 x 轴，另一对平行于 y 轴。（这似乎很明显，但值得清楚一切的含义！）

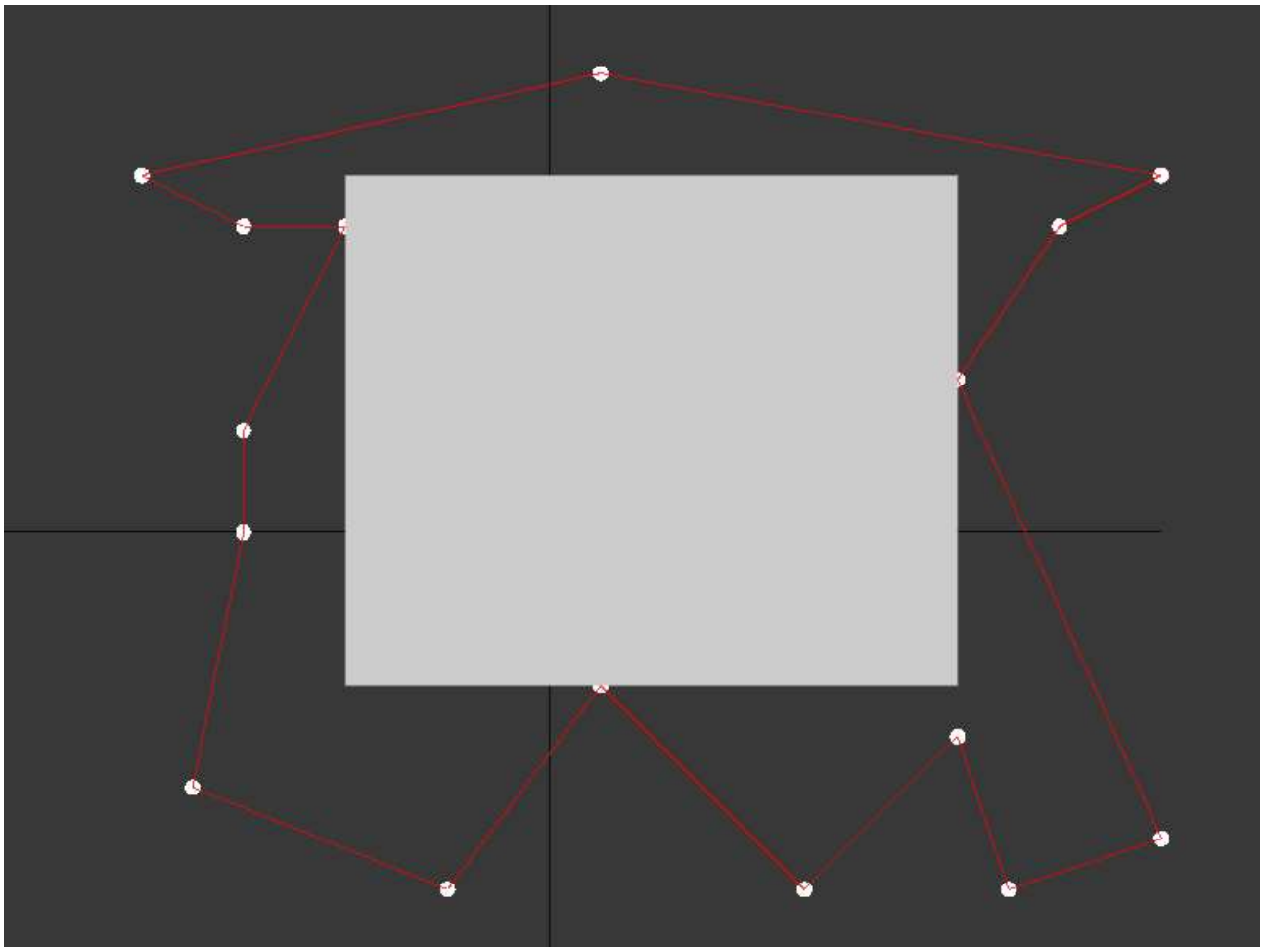


图 1：凹多边形中最大的内部矩形。

图 1 显示了一个凹多边形 - 一组点（显示为白点）通过边（显示为红线）连接以形成一个封闭的简单凹多边形。在多边形内部，有一个矩形区域显示最大的轴对齐内部矩形，该矩形不跨越多边形边缘或包含任何多边形点。

似乎有多种不同的算法可以解决这个问题的变体，我发现的相关工作可以追溯到 1970 年代。涵盖该问题的最佳论文之一是由 Karen Daniels、Victor Milenkovic 和 Dan Roth 撰写的，标题为在几类多边形中找到最大的矩形。然而，实施他们的解决方案所需的理解程度目前超出了我的能力范围。这篇论文我已经读了几遍了，还没有深入到必要的理解。

蛮力解决方案是可能的，但该过程将非常缓慢（因为它涉及针对某些属性的每种可能变化测试每个顶点与每个多边形边）。假设如果它是一个可行的解决方案，它将被记录和实施——我消化的所有论文都讨论了 O(n^4) 或 O(n^5) 方面的蛮力解决方案——换句话说，对于我希望使用的顶点数（最多 1000 个），我将进行数十亿或数万亿次计算。

Zahraa Marzeh、Maryam Tahmasbi 和 Narges Mirehi 撰写了一篇我确实设法理解的论文，也是我用作此实现基础的论文，题为Algorithm for find the maximum inscribed rectangle in polygon。以此为基础，我使用 C# 在 Unity 中创建了一个工作实现。

3. 实施

解决问题的一种方法是将多边形覆盖的空间划分为一个轴对齐的矩形网格，然后找到由这些子矩形组成的最大矩形。

第 1 步需要找到包含简单多边形的最小面积矩形。
第 2 步构造矩形（单元格）网格，使用多边形的每个顶点作为网格线的 x 和 y 坐标。
第 3 步可选地细化这些矩形（添加更多的矩形，因此对生成的网格有更多的粒度）。
步骤 4扫描这个矩形网格，并确定一个矩形是在多边形内部还是外部。
步骤 5确定哪些矩形在水平和垂直方向上链接到（相邻的）其他矩形。
第 6 步使用此邻接信息来计算每个矩形可以通过向上和向右移动构建的最大面积矩形。
步骤 7或者，迭代所有内部矩形，计算相邻矩形区域的面积，并跟踪最佳区域。

这些步骤中的每一个都很容易理解，并且所有步骤都可以以线性方式解决，将每个步骤的结果反馈到下一步。

3.0 预备知识和代码库

为了更好地促进实现的可见性，我已在 Github 上的 https://github.com/Evryway/lir上提供了它。随意阅读那里的代码。repo 上的文档相当稀少，但代码本身（连同本文）应该足以让您使用自己的实现。

大部分代码包含在 LargestInteriorRectangle.cs中，但 Extensions.cs中有一些辅助扩展，Bound2D.cs 中有一个Bound2D 类（表示非轴对齐的 2D 边界）。

您可以在LargestInteriorRectangleTests.cs中找到一些针对预期通过和失败案例的 测试。如果您不确定函数应该如何工作，请尝试在其中查找示例。

3.1 寻找面积最小的矩形

第一个过程是获取多边形的所有顶点（在本例中为 vs）并计算出唯一 x 和 y 值的列表。简单的解决方案是使用 LINQ，如下所示：

```
vs = Vector2[] { ... }

xs = vs.Select(v => v.x).OrderBy(x => x).Distinct().ToArray();
ys = vs.Select(v => v.y).OrderBy(y => y).Distinct().ToArray();
```

但这不是最快的方法（对于初学者，我们迭代 vs 数组两次），虽然“Distinct”确实确保了唯一性，但实际上最好使用 epsilon-check（小范围内的两个值）代替的平等。我使用的实际代码稍微复杂一些。首先，将 xs 和 ys 复制到两个列表中，并对它们进行排序。

```
var xsl = new List<float>(vs.Length);
var ysl = new List<float>(vs.Length);
for (int i = 0; i < vs.Length; i++)
{
    var v = vs[i];
    xsl.Add(v.x);
    ysl.Add(v.y);
}
xsl.Sort();
ysl.Sort();
```

然后，计算出最大范围（在 X 或 Y 中）并使用它来确定一个相对于比例的 epsilon - 小到不会任意对值进行别名，大到足以满足我们正在使用的任何输入范围. 唯一性是通过测试与前一个值一起出现的值来确保的——当列表被排序时，这是一个相当快的测试。使用 HashSet 之类的东西来确保唯一性意味着在插入时测试每个项目，这最终会相当慢，每次插入都要测试数百个项目。

```
float xmin = xsl[0];
float xmax = xsl[xsl.Count - 1];
float ymin = ysl[0];
float ymax = ysl[ysl.Count - 1];
```

```
float mmin = Mathf.Min(xmin, ymin);
float mmax = Mathf.Max(xmax, ymax);

var xsd = new List<float>(vs.Length) { xsl[0] };
var ysd = new List<float>(vs.Length) { ysl[0] };

float epsilon = (mmax - mmin) / (1024 * 1024);        // 1 millionth of the span.
for (int i = 0; i < vs.Length-1; i++)
{
    if (xsl[i + 1] - xsl[i] > epsilon) xsd.Add(xsl[i + 1]);
    if (ysl[i + 1] - ysl[i] > epsilon) ysd.Add(ysl[i + 1]);
}
xs = xsd.ToArray();
ys = ysd.ToArray();
```

此时，我们已经有了 xs 和 ys，以及轴对齐的边界（分别是最小和最大 x 和 y 值）。或者，您可以在此步骤之前显式搜索最小边界矩形。边界矩形可能不是轴对齐的，这将需要旋转多边形点以实现与轴对齐边界的对齐。这很可能（但不能保证）有助于接近最佳 LIR。这里需要更多的测试来确定成本是否值得。

3.2 构建Cell网格

给定一个 xs 数组和一个 ys 数组，步骤 2 涉及创建单元格网格。实际上，这很简单。对于 xs (x[i] 到 x[i+1]) 和 ys (y[j] 到 y[j+1]) 中的每对条目，您都有一个单元格 cell[i,j] 是一个矩形。我们已经有了 xs 和 ys，所以不需要复制它们；只需创建一个单元格数组，即可使用。cells[i,j] 中的每个单元格表示横轴从 x[i] 到 x[i+1] 的矩形，纵轴表示从 y[j] 到 y[j+1] 的矩形。

```
var xc = xs.Length - 1;
var yc = ys.Length - 1;

cells = new int[xc,yc];
```

需要注意的一点是，在 C# 中，二维数组的执行速度可能比一维数组慢。示例代码使用 2D 数组，但将 2D 索引转换为 1D 索引相当简单 - 而不是使用

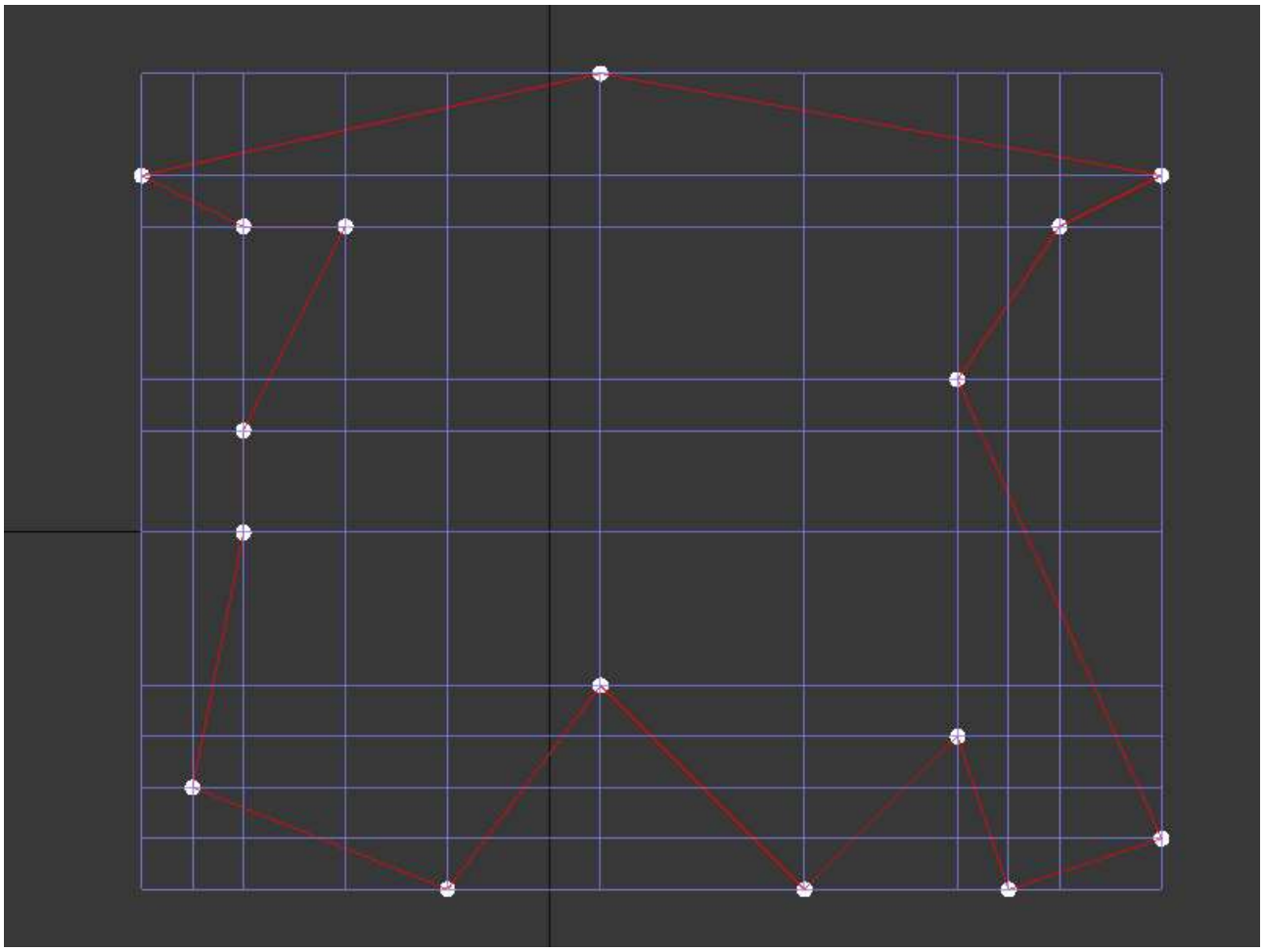
```
var q = cells[i,j];
```

您可以像这样索引一维数组：

```
var q = cells[(j*xc) + i];
```

阅读难度稍大，性能稍快，可能值得权衡。

此处显示了生成的单元格 - 您可以轻松查看每个顶点如何生成 x 线和 y 线，并延伸到多边形的边界框。在此示例中，一些顶点与其他顶点共享 X 和 Y 中的线 - 这没关系。



3.3（可选）细化网格

如果您的简单多边形中没有足够的顶点，您最终可能会得到不靠近多边形边界的大像元。这里的一种选择是检测多边形边缘和每个顶点的 X/Y 线之间的每个交点，然后将这些新点插入到顶点数组中（需要重新排序，然后重新生成单元格）。我发现有了足够的分数，这并不是真正必要的，但它会给出更准确的结果（以潜在的更多工作为代价）。请注意图 4 中的示例，它显然不够完善。

另一种选择是缩小单元格的数量（例如，通过将 xs 和 ys 数组减少到更少的值）。同样，我没有使用过这种方法——我发现简单地减少原始多边形中的点数就可以为我的用例获得完全足够的结果。如果您的源多边形在顶点密度方面严重偏向特定方向，您可能会发现单元缩减比简单地从多边形中剥离顶点更可取。

3.4 扫描网格的内部和外部单元格

接下来，我们必须查看每个单元格，看看它是在多边形内部（内部）还是多边形外部（外部）。任何与多边形边相交的像元都被归类为外部像元。

天真的方法是找到每条边的起始顶点和结束顶点的 xs 和 ys 索引。然后，对于该跨度中的每个单元格，测试该单元格以查看它是在边缘内部、交叉还是外部。

我的实现做了一些性能改进。首先，因为我（可能）在 xs 和 ys 中对一些 x 和 y 值进行了别名，所以不是多边形中的每个顶点都会有一个精确匹配 - 但我知道如果我迭代边缘，我总是从前一个边的结束（所以我已经知道该顶点的 xs 和 ys 的索引）。其次，我可以使用边缘在 X 和 Y 轴上移动的方向来确定我需要在 xs 和 ys 数组中搜索哪个方向，因此能够找到正确的索引 - 并且只要我等于 (或大于！）顶点坐标，我有一个跨度匹配。

```
var v0 = vs[0];
var six = -1;
var siy = -1;
var eix = 0; while (xs[eix] < v0.x && eix < xs.Length-1) eix++;
var eiy = 0; while (ys[eiy] < v0.y && eiy < ys.Length-1)  eiy++;
```

最初的实现只是通过使用 xs.IndexOf(v0.x) 和 ys.IndexOf(v0.y) 找到开始和结束索引。这在初始情况下并不慢（在上面找到 eix 和 eiy），但是因为我可能会由于 epsilon 检查而跳过 xs 和 ys 中的值，所以在某些情况下永远找不到确切的 x 或 y 坐标。如果没有找到确切的值，while 循环通过简单地取下一个较大的值来确保我们接近（足够接近）。

相同的过程用于查找每个后续边的结束索引 - 但是，边可能在多边形内向左或向下移动，在这种情况下，我们需要向后搜索。

这种优化对算法的运行时间产生了很大的影响。

```
for (int i = 0; i < vc; i++)
{

    var s = vs[i];
    var e = vs[(i + 1) % vc];
    var edge = e - s;

    // get the indices for the start - it should be the end vertex of the previous edge.
    six = eix;
    siy = eiy;

    // could possibly binary search, but if the edge lengths are short, a linear scan
    // should be fairly fast anyway.
    int tx = edge.x >= 0 ? 1 : - 1;          // -1 or 1
    int ty = edge.y >= 0 ? 1 : - 1;          // -1 or 1
```

```
if (tx > 0) { while (xs[eix] < e.x && eix < xs.Length-1) eix++; }
      else { while (xs[eix] > e.x && eix > 0) eix -= 1; }
if (ty > 0) { while (ys[eiy] < e.y && eiy < ys.Length-1) eiy++; }
      else { while (ys[eiy] > e.y && eiy > 0) eiy -= 1; }

// we now have a span.
var span_x_start = Mathf.Min(six, eix);
var span_y_start = Mathf.Min(siy, eiy);

var span_x_end = Mathf.Max(six, eix);
var span_y_end = Mathf.Max(siy, eiy);
```

现在，鉴于我的跨度，我知道哪些细胞接触（或接近）该特定边缘。

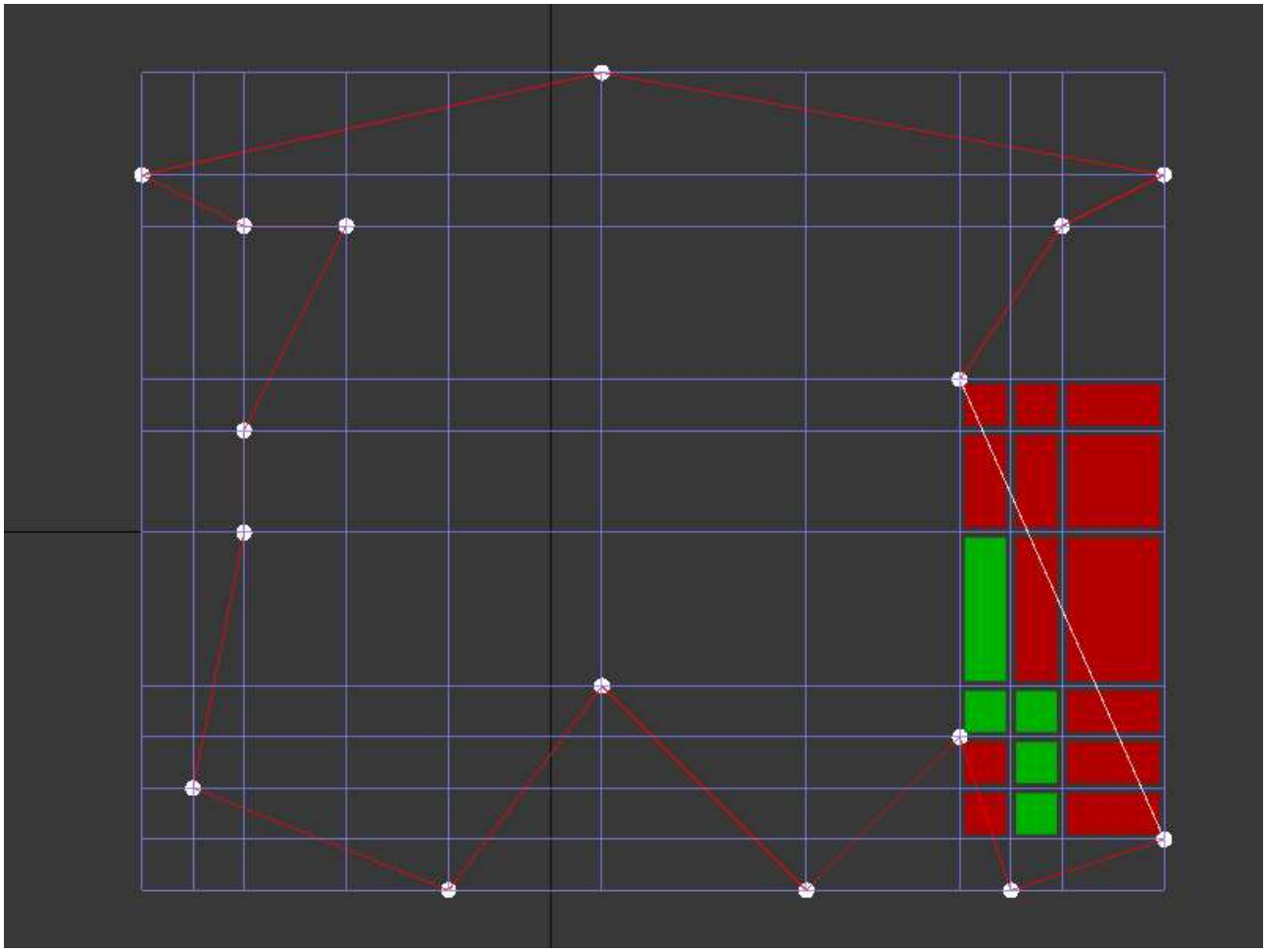


图 3 显示了由最右下边缘跨越的单元格（以白色绘制）。外部或交叉单元格标记为红色。内部单元格标记为绿色。左下角的两个单元格被标记为红色 - 它们位于我们当前正在测试的边缘的内部，但它们已经被标记为外部，因为它们与之前的边缘交叉。

内部/外部检查相当简单 - 计算出边缘“进入”方向（基本上是边缘法线，在多边形内部的方向上）。然后我们选择“最佳”单元角进行测试——如果“最佳”角在多边形内，其他三个角也将在多边形内。

水平和垂直边缘是特殊情况，因为它们实际上不跨越任何单元格（因此跨度在该轴上的长度为零）。在这种情况下，我们只需识别内侧，并直接标记相关的多边形。

```
var into = new Vector2(-edge.y, edge.x);
int rx = into.x >= 0 ? 0 : 1;
int ry = into.y >= 0 ? 0 : 1;

// for -x,-y edges, the into direction is +X, -Y - use the TL cell vertex to test. (0,1)
// for +x,-y edges, the into direction is +X, +Y - use the BL cell vertex to test. (0,0)
// for -x,+y edges, the into direction is -X, -Y - use the TR cell vertex to test. (1,1)
// for +x,+y edges, the into direction is -X, +Y - use the BR cell vertex to test. (1,0)

// it is possible for two edges to span the same cell, so we check to see if a cell
// has already been marked exterior - and if so, continue.

// this covers all of the cells that the edge crosses.
// if the edge is vertical or horizontal, the span width in that axis should be 0.
if (span_x_end - span_x_start == 0)
{
    // vertical edge. pick the cells on the interior side.
    var p = span_x_start - rx;
    for (int q = span_y_start; q < span_y_end; q++)
    {
        if (cells[p, q] < 0) continue;
        cells[p,q] = 1;
    }
    continue;
}
else if (span_y_end - span_y_start == 0)
{
     // horizontal edge. pick the cells on the interior side.
    var q = span_y_start - ry;
    for (int p = span_x_start; p < span_x_end; p++)
    {
        if (cells[p, q] < 0) continue;
        cells[p,q] = 1;
    }
    continue;
}
for (int q = span_y_start; q < span_y_end; q++)
{
```

```
    for (int p = span_x_start; p < span_x_end; p++)
    {
        // if we've already marked this as exterior, then skip it.
        // it's possible to be interior to another edge, but still exterior
        // to this one, so continue the check in that case.
        if (cells[p, q] < 0) continue;
        // based on the edge direction, pick the correct corner to test against.
        var v = new Vector2(xs[p + rx], ys[q + ry]);
        var sv = v - s;
        var d = Vector2.Dot(sv, into);

        // mark the cell either exterior (-1) or interior (1)
        cells[p,q] = d < 0 ? -1 : 1;
    }
}
```

我们现在已经测试了与边缘相交或与水平或垂直边缘相邻的每个单元格。然而，这并不是所有的单元格——我们需要一种标记所有外部单元格的方法，以及一种标记所有内部单元格的方法。

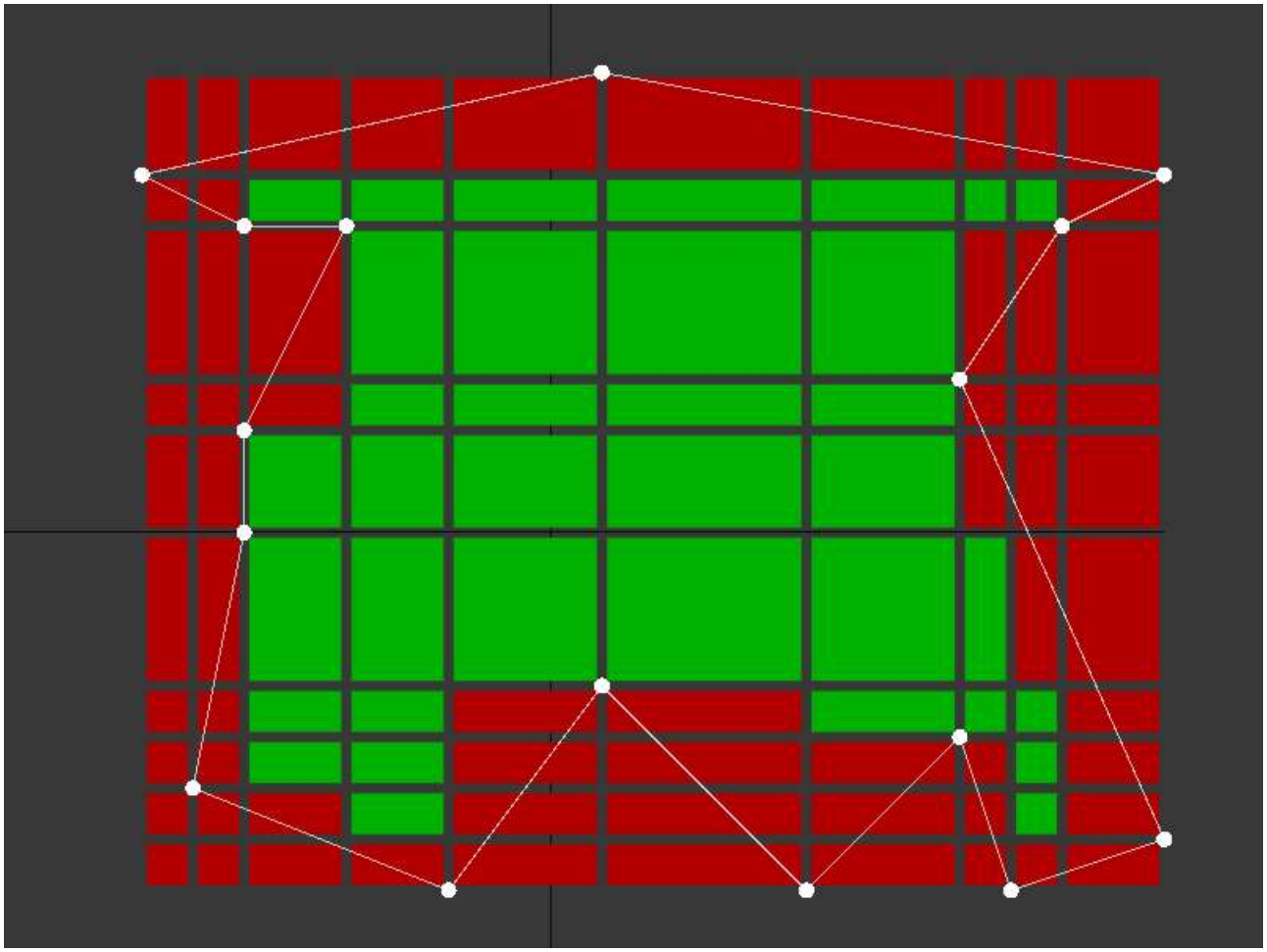
在这里，我们从每行（和每列）单元格的外边缘开始，然后向内走，直到找到之前标记的单元格。任何尚未标记的都是外部的，直到我们碰到内部单元格为止。

接下来，我们将剩余的所有内容标记为内部。**这不是正确的行为，因为有些情况应该是外部的，而这些情况将被标记为内部。**我在写这篇文章时才意识到这一点，这表明代码审查是一个美妙的过程。正确的行为可以通过计算边缘交叉来实现，这会更复杂。

```
// outside region sweep.
// some cells may not be spanned by edges.
// start on the outside of the region, and mark everything as exterior, until
// we come across a cell that has been explicitly marked.

for (int q = 0; q < yc; q++)
{
    // from the left edge.
    for (int x = 0; x < xc; x++ )
    {
        if (cells[x,q] != 0) break;
        cells[x,q] = -1;
    }
    // from the right edge.
    for (int x = xc-1; x >= 0; x--)
```

```
        {
            if (cells[x,q] != 0) break;
            cells[x,q] = -1;
        }
    }

    for (int p = 0; p < xc; p++)
    {
        // from the bottom edge.
        for (int y = 0; y < yc; y++)
        {
            if (cells[p,y] != 0) break;
            cells[p,y] = -1;
        }
        // from the top edge.
        for (int y = yc-1; y >= 0; y--)
        {
            if (cells[p,y] != 0) break;
            cells[p,y] = -1;
        }
    }


    // sweep for interior (untested) cells.
    // in fact, mark everything.
    for (int j = 0; j < yc; j++)
    {
        for (int i = 0; i < xc; i++)
        {
            // anything that was -1 goes to 0.
            // anything that was 0 or 1 goes to 1.
            // this ensures any un-tested cells are classed as interior.
            cells[i,j] = cells[i,j] < 0 ? 0 : 1;
        }
    }
```

此扫描的结果应该是每个单元格都被标记为内部或外部。图 4 显示了我们的示例多边形。



您可以在此图像中清楚地看到，内部单元格的顶行实际上并未接触多边形的顶部边缘，并且可以向上移动。**我的实现没有执行这个调整，它可能是一个值得改进的地方**。但是，多边形中的顶点越多，接触就越准确（或接近边缘）。

3.5 确定小区邻接

接下来我用作参考的论文描述了确定小区邻接性。我将描述这个过程，因为我已经实现了它——但我发现了一个稍微快一点的机制，我用它来帮助计算我接下来要描述的区域。

第一个动作是创建两个邻接数组——一个用于水平邻接，一个用于垂直邻接。接下来，扫描每一行（和每一列）以计算每个单元格的最大邻接长度。

每个单元格都知道有多少个单元格可以向右遍历（+X） - 以及有多少个单元格可以向上遍历（+Y）。如果一个单元格没有内部邻居，它将没有邻接关系。

```
var adjacency_horizontal = new int[xc, yc];
var adjacency_vertical = new int[xc, yc];

// calculate horizontal adjacency, row by row
for (int y = 0; y < yc; y++)
{
```

```
    int span = 0;
    for (int x = xc-1; x >= 0; x--)
    {
        if (cells[x,y] > 0) span++; else span = 0;
        adjacency_horizontal[x, y] = span;
    }
}

// calculate vertical adjacency, column by column.
for (int x = 0; x < xc; x++)
{
    int span = 0;
    for (int y = yc-1; y >= 0; y--)
    {
        if (cells[x,y] > 0) span++; else span = 0;
        adjacency_vertical[x, y] = span;
    }
}
```

算法的下一步是检查每个单元格，并创建两个向量（称为 H 和 V）来描述每个单元格在根单元格垂直上方可以水平遍历多少个单元格（这些值进入 H 向量）；以及根单元右侧的每个单元可以垂直遍历多少个单元（这些值进入 V 向量）。

我花了一点时间来理解这两个向量的值，所以这是一张显示特定单元格数字的图像。

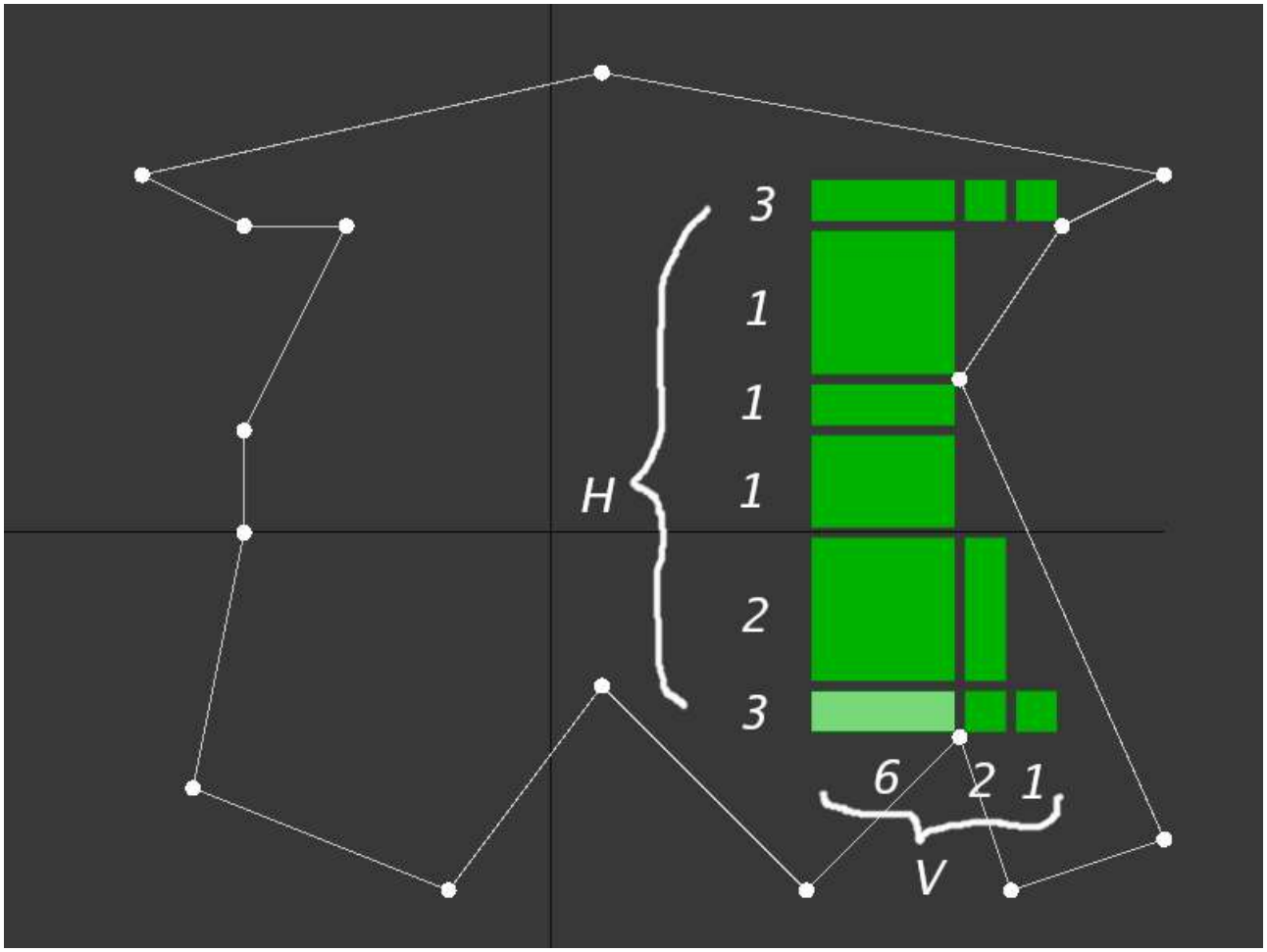
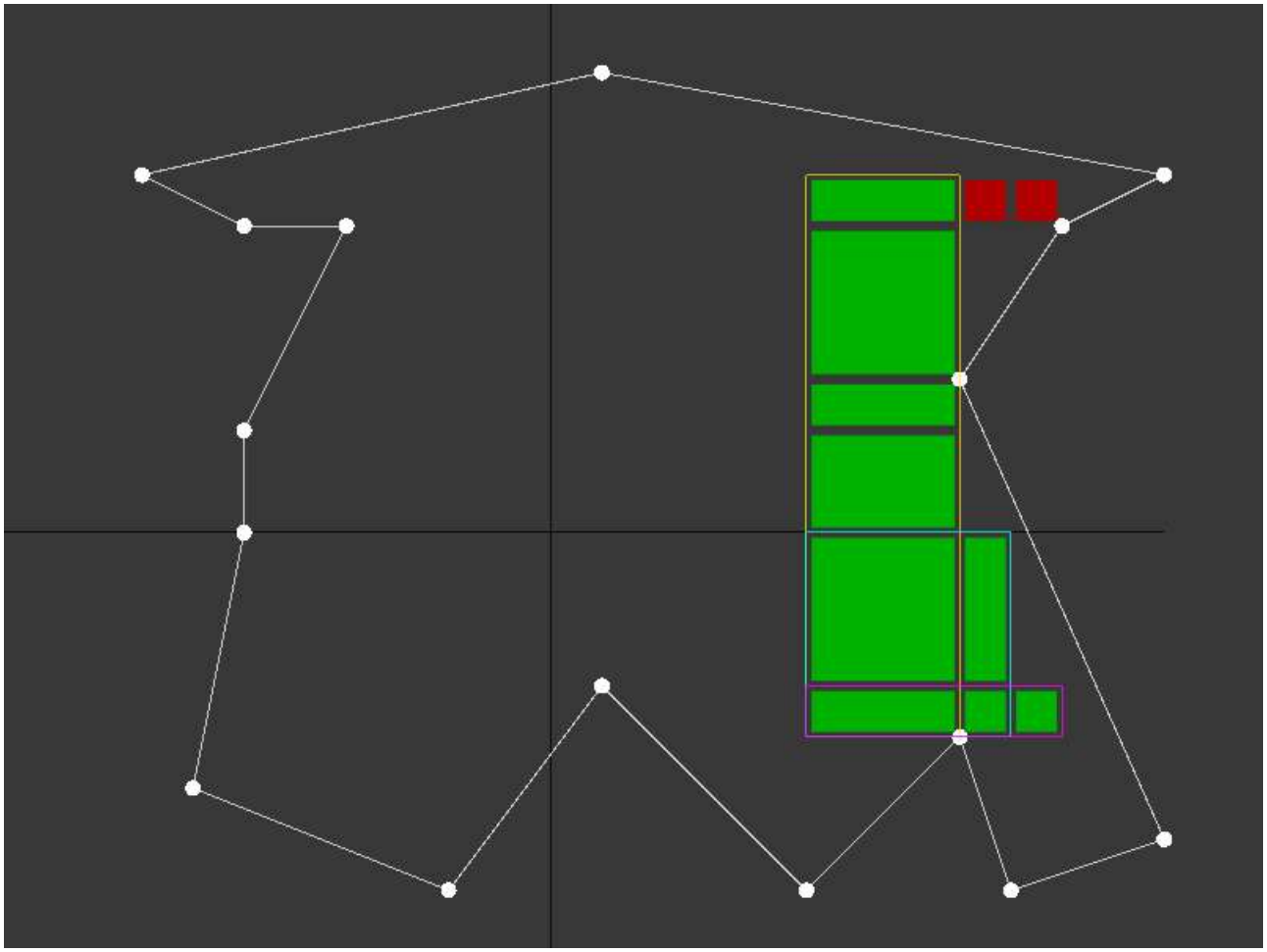


图 5 显示了一个作为根的示例单元格。H 向量描述了可以水平遍历多少个单元格，从根单元格开始每一步向上 - 在这种情况下，3，然后 2，然后 1、1、1，然后 3。V 向量描述可以垂直遍历多少个单元格，从根单元格向右的每一步 - 在本例中是 6，然后是 2，然后是 1。

这给了我们 [3,2,1,1,1,3] 的 H 和 [6,2,1] 的 V。

但是，如果我们试图使最大的矩形成为可能，我们会受到相关行下方所有行以及相关列左侧所有列的最大遍历的限制——这意味着最高值H 向量需要被限制为 1，将 H 向量设为 [3,2,1,1,1,1]。



在本例中组合 V 和 H 向量的结果给出了可以从单元格延伸的三个可能的大矩形 - 一个是 (1 x 6)，一个是 (2 x 2)，一个是 (3 x 1) ）。在图 6 中，这些显示为黄色、青色和洋红色轮廓。

顶行的固定单元格显示为红色 - 很明显，您不能创建一个仅包含包含它们的内部单元格的矩形，因为下面的行限制向右移动。

```
// generate H vector - this is horizontal adjacency for each step up.
clir_hvec.Clear();
// look at horizontal adjacency.
// step up from our initial cell, and look right.

var h = adjacency_horizontal[x, y];
clir_hvec.Add(h);
for (int q = y+1; q < ayc; q++)
{
    if (cells[x, q] != 1) break;
    // each row can only be as large as the previous - a rectangle cannot push
    // further out than a lower row.
    h = Mathf.Min(adjacency_horizontal[x, q], h);
    clir_hvec.Add(h);
```

```
}

// generate V vector. This is vertical adjacency for each step right.
clir_vvec.Clear();
// look at vertical adjacency.
// step right from our initial cell, and look up.

var v = adjacency_vertical[x, y];
clir_vvec.Add(v);
for (int p = x+1; p < axc; p++)
{
    if (cells[p, y] != 1) break;
    // each column can only be as large as the previous - a rectangle cannot push
    // further up than a previous column.
    v = Mathf.Min(adjacency_vertical[p, y], v);
    clir_vvec.Add(v);
}
```

3.6 确定每个小区的最大潜在跨越面积

你也可以看到，每次你在一个轴上“踩”，你必须同时在另一个轴上“踩”。可以在每个方向上跳跃多个单元格，但您必须至少移动一个单元格（因为，例如，如果 H 向量是 [3,3,1,1,1,1]，那么等效的 V矢量将是 [6,2] - 生成的跨度矩形将是 (1 x 6) 和 (3 x 2)。

这种洞察力意味着您实际上在任何一种情况下都只使用唯一值 - 所以回到我们原来的 H 向量，它会导致 [3,2,1] 与 [6,2,1] 的 V 相对。反转任一向量可以让您通过压缩向量来创建跨度，例如，将 V 反转为 [1,2,6] 会给出一组 (3 x 1)、(2 x 2) 和 (1 x 6) 的跨度。这就是我们的跨度设置！

此代码不会将 H 向量仅减少为唯一值 - 但它确实检查跨度本身是否唯一，这实际上给出了相同的结果。

```
spans.Clear();

// generate the set of valid spans.
int2 span_last = new int2(-1, -1);
for (int i = 0; i < clir_hvec.Count; i++)
{
    int p = hvec[i];
    int q = vvec[p-1];
```

```
        int2 span = new int2(p, q);
        if (span.x != span_last.x && span.y != span_last.y)
        {
            spans.Add(span);
            span_last = span;
        }
    }
```

给定我们的跨度集，我们可以轻松计算出每个跨度的面积，因为我们知道每个矩形的尺寸。跟踪最佳区域使我们能够返回整个单元集中找到的最佳区域。

```
    for (int i = 0; i < spans.Count; i++)
    {
        var span = clir_spans[i];
        var xstart = xs[x];
        var xend = xs[x + span.x];
        var ystart = ys[y];
        var yend = ys[y + span.y];
        var xsize = xend - xstart;
        var ysize = yend - ystart;
        var area = xsize * ysize;
        if (area > best_area)
        {
            best_area = area;
            best_span = span;
            best_origin = new int2(x, y);
        }
    }
```

3.7 直接从像元边长而不是跨度来迭代和计算面积

注意到所有潜在的最大内部矩形导致 H 和 V 向量的长度相等，并且在生成跨度之前所有距离都是已知的，这意味着我们实际上可以直接使用跨度长度（而不是计算跨度然后返回长度）。

该代码与 3.5 和 3.6 中的跨度扫描代码非常相似。首先，计算跨度的长度（而不是邻接）。

```
    var lengths_horizontal = new float[xc,yc];
```

```
var lengths_vertical = new float[xc,yc];


for (int y = 0; y < yc; y++)
{
    float span = 0;
    for (int x = xc - 1; x >= 0; x--)
    {
        span = (cells[x, y] <= 0) ? 0 : span + xs[x + 1] - xs[x];
        lengths_horizontal[x,y] = span;
    }
}

for (int x = 0; x < xc; x++)
{
    float span = 0;
    for (int y = yc - 1; y >= 0; y--)
    {
        span = (cells[x, y] <= 0) ? 0 : span + ys[y + 1] - ys[y];
        lengths_vertical[x,y] = span;
    }
}
```

然后，直接使用垂直和水平长度迭代每个单元格。

```
for (int y = 0; y < yc; y++)
{
    for (int x = 0; x < xc; x++)
    {
        var iv = cells[x,y];
        if (iv == 0) continue;

        var h = lengths_horizontal[x,y];
        var v = lengths_vertical[x,y];

        // if the best POSSIBLE area (which may not be valid!)
        // is smaller than the best area, then we don't need to run any further tests.
        if (h * v < best_area) continue;

        // generate H vector - this is horizontal spans for each step up.
```

```
hspans.Clear();
// look at horizontal spans.
// step up from our initial cell, and look right.

hspans.Add(h);
for (int q = y+1; q < yc; q++)
{
    if (cells[x,q] == 0) break;
    var h2 = lengths_horizontal[x,q];
    if (h2 >= h) continue;
    h = h2;
    hspans.Add(h);
}

// generate V vector. This is vertical spans for each step right.
vspans.Clear();
// look at vertical spans.
// step right from our initial cell, and look up.

vspans.Add(v);
for (int p = x+1; p < xc; p++)
{
    if (cells[p,y] == 0) break;
    var v2 = lengths_vertical[p,y];
    if (v2 >= v) continue;
    v = v2;

    vspans.Add(v);
}

// reverse the v spans list - this lets us trivially combine the correct
// spans for each rectangle combination with the same list index.
vspans.Reverse();

for (int i = 0; i < hspans.Count; i++)
{
    float hl = hspans[i];
    float vl = vspans[i];
    float area = hl * vl;
    if (area > best_area)
    {
```

```
                best_area = area;
                best_origin = new Vector2(xs[x], ys[y]);
                best_span = new Vector2(hl, vl);
            }
        }

    }
}
```

我发现的一个问题是生成 H 和 V 向量偶尔会给出向量长度不匹配的结果（例如，H 将包含 16 个值，V 将包含 15 个）。在将结果与 Adjacency Span 版本进行匹配后，我发现这是由于数值不准确，并在生成初始 xs 和 ys 数组时引入了 epsilon 检查。那里的值非常接近，可能会导致上述扫描代码中的长度相同。在向量不匹配的情况下，可以选择使用邻接数据。

最后，检查最大可能区域是否已经小于最佳区域，这对算法的运行时间产生了巨大影响。那里可能还有其他好的优化。

4。结论

你有它 - Axis Aligned Largest Internal Rectangle。我希望有人能像我一样发现这很有用，而且我很乐意收到有关代码和文章的任何反馈。

参考

参考 1：在几类多边形中找到最大的矩形（Karen Daniels、Victor Milenkovic、Dan Roth） 参考 2：在多边形中找到最大内接矩形的算法（Zahraa Marzeh、Maryam Tahmasbi、Narges Mirehi）

写于 2020 年 9 月 15 日

« 两个问题的故事 - 解决问题的解决方案
»

ALSO ON **EVRYWAY.COM**

How to build a holodeck, week 19

6 years ago • 1 comment

This week has been a bust, from the point of view of progress on the …

Dev Blog - How to build a holodeck, …

6 years ago • 5 comments

Welcome to my “How to build a holodeck - follow along at home with sticky …

How to holodec

6 years ag

This weel my R200 capturing