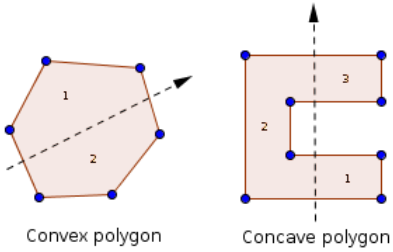
Divisione di un poligono arbitrario mediante una linea

21 marzo 2015di David | 13 commenti

In questo post presento un algoritmo per dividere un *poligono semplice* e arbitrario per una linea. La mia implementazione si basa sulle idee dell'articolo *Spatial Partitioning of a Polygon by a Plane* di George Vanecek, pubblicato nel libro *Graphics Gems* V. L'intenzione dietro questo post è coprire tutti i dettagli che l'articolo originale ha omesso. Si presuppone che tutti i poligoni siano orientati in senso antiorario. Puoi trovare il codice sorgente completo della mia implementazione in questo repository github .

introduzione

La divisione di un poligono tramite una linea è strettamente correlata al *ritaglio* di un poligono rispetto a una regione di ritaglio, come un rettangolo o un altro poligono. Dividere un poligono è più semplice e allo stesso tempo più difficile che ritagliarlo. È più semplice perché ritagli solo una singola linea ed è più difficile perché i poligoni risultanti non possono essere semplicemente scartati. Si scopre che dividere i poligoni *convessi* è molto più semplice che dividere i poligoni *concavi* . Il motivo è che la divisione dei poligoni convessi dà come risultato al massimo due nuovi poligoni, mentre la divisione dei poligoni concavi può dare come risultato molti nuovi poligoni arbitrari. Osservare la figura seguente per un'illustrazione.



Trovare un'implementazione corretta per i poligoni concavi è complicato, perché ci sono molti casi angolari che devono essere considerati. Di seguito viene innanzitutto delineato un algoritmo per la suddivisione dei poligoni convessi. Successivamente, l'algoritmo viene esteso ai poligoni concavi.

MESSAGGI RECENTI

- Advanced Octrees 4: ricerca dei nodi vicini 2 dicembre 2017
- Eliminatori personalizzati per puntatori intelligenti nel moderno C++ , 29 ottobre 2017
- Spinlock scalabili 1: basato su array , 3 dicembre 2016
- Il biglietto spinlock 9 aprile 2016
- Spinlock di prova e impostazione , 23 marzo 2016

CATEGORIE

C++ (7)
Architettura informatica (5)
Grafica computerizzata (8)
Strutture dati e algoritmi (10)
Banche dati (1)
Demoscena (4)
MySQL (1)
Sistemi operativi (5)
Ottimizzazione (3)
Programmazione parallela (4)
Qt (1)
Senza categoria (1)

Poligoni convessi

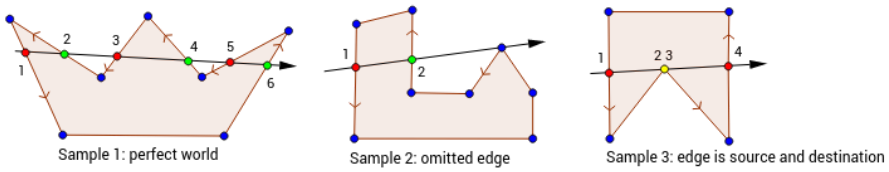
Dividere i poligoni convessi è piuttosto semplice. In sostanza, si scorre sui vertici del poligono, si controlla su quale lato della linea di divisione si trovano e li si aggiunge di conseguenza al poligono risultato sinistro o destro. Quando un bordo attraversa la linea di divisione, inoltre, il punto di intersezione deve essere aggiunto a entrambi i poligoni risultanti. È necessario prestare particolare attenzione quando più vertici successivi si trovano sulla linea di divisione. In tal caso, a causa della proprietà convessità, l'intero poligono si trova sul lato sinistro o destro del poligono. Quando il bordo di un poligono attraversa per la seconda volta la linea di divisione, sappiamo che la prima metà del poligono è già completamente divisa.

Poligoni concavi

A differenza della divisione dei poligoni convessi, la divisione dei poligoni concavi richiede uno sforzo algoritmico maggiore. Quando la linea divisa attraversa un poligono concavo per la seconda volta, non significa necessariamente che la prima metà del poligono sia ora completamente divisa. Al contrario, potrebbe darsi che si sia appena *aperto* un altro nuovo poligono da dividere . Per tenere conto di ciò, in primo luogo, ho provato a scorrere i bordi del poligono portando con me una pila di poligoni aperti. Ogni volta che mi imbattevo in un vertice che apparteneva al poligono aperto in cima allo stack, sapevo che era appena chiuso. Questo approccio ha funzionato bene finché non ho provato a tenere conto di tutti i casi limite. Alla fine, ho deciso di utilizzare il metodo dell'articolo menzionato nell'introduzione.

Idea dell'algoritmo

I punti di intersezione tra la linea di divisione e il poligono sono fondamentali per l'algoritmo di divisione. Accoppiando 2 tuple di punti di intersezione lungo la linea di divisione, è possibile determinare i bordi che chiudono i poligoni risultanti. Concettualmente, ogni due punti di intersezione lungo la linea di divisione formano una coppia di bordi, ciascuno dei quali chiude un lato di due poligoni risultanti vicini. In un mondo perfetto, i punti di intersezione potrebbero essere facilmente accoppiati dopo averli ordinati lungo la linea di divisione, rispetto al punto iniziale della linea di divisione (cfr. la figura sotto a sinistra). Sfortunatamente, nel mondo reale è necessario prestare particolare attenzione quando la linea di divisione non attraversa il bordo di un poligono al centro, ma tocca solo il vertice iniziale e/o finale. In questi casi l'abbinamento dei punti *di origine* (rosso) e dei punti *di destinazione* (verde) lungo la linea di divisione diventa più complicato (cfr. la figura al centro). Come puoi vedere, solo alcuni punti di intersezione si qualificano come sorgenti e destinazioni e alcuni punti di intersezione addirittura si qualificano come entrambi (giallo, cfr. figura sotto a destra). Fortunatamente, è possibile tenere conto di tutti questi casi, come vedremo più avanti.



Sample 1: perfect world | Sample 2: omitted edge | Sample 3: edge is source and destination

Calcolo dei punti di intersezione

Per facilitare il processo di suddivisione è utile rappresentare il poligono come una lista doppiamente collegata. Calcoliamo i punti di intersezione tra la linea di divisione e il poligono eseguendo un'iterazione su ciascun bordo del poligono `e` e determinando su quale lato della linea di divisione si trovano il vertice iniziale `e.Start` e il vertice finale del bordo `e.End`. Se `e.Start` si trova sulla linea di divisione `e` viene aggiunto all'array `EdgesOnLine`, che contiene tutti i bordi che iniziano sulla linea di divisione. Se `e.Start` e `e.End` si trovano su lati opposti della linea di divisione, un nuovo bordo che inizia nel punto di intersezione viene inserito nel poligono e aggiunto a `EdgesOnLine`. I bordi di questa matrice sono interessanti perché iniziano nei punti di intersezione con la linea di divisione. L'algoritmo è descritto di seguito.

```
for (auto &e : poly)
{
    const LineSide sideStart = GetLineSide(SplitLine, e.Start);
    const LineSide sideEnd = GetLineSide(SplitLine, e.End);

    if (sideStart == LineSide::On)
    {
        EdgesOnLine.push_back(e);
    }
    else if (sideStart != sideEnd && sideEnd != LineSide::On)
    {
        Point2 ip = IntersectLines(SplitLine, e);
        Edge *newEdge = poly.InsertEdge(e, ip);
        EdgesOnLine.push_back(newEdge);
    }
}
```

Ordinamento dei punti di intersezione

Dopo che tutti i bordi del poligono sono stati intersecati con la linea di divisione, i bordi nella `EdgesOnLine` serie vengono ordinati lungo la linea di divisione in base alla distanza tra i relativi punti iniziali e il punto iniziale della linea di divisione. Ordinare l' `EdgesOnLine` array è fondamentale per accoppiare i bordi di origine e di destinazione semplicemente eseguendo un'iterazione sull'array come vedremo nella sezione seguente.

```
std::sort(EdgesOnLine.begin(), EdgesOnLine.end(), [&](PolyEdge *e0, PolyEdge *e1)
{
    return (CalcSignedDistance(line, e0->StartPos) <
            CalcSignedDistance(line, e1->StartPos));
});
```

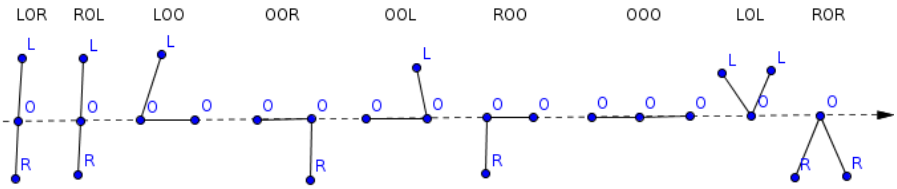
Come ha sottolineato Glenn Burkhardt in un commento, la distanza euclidea non può essere utilizzata come metrica della distanza perché non ha segno. La distanza euclidea può essere utilizzata solo finché il punto iniziale della linea divisa si trova all'esterno del poligono. In tal caso tutti i punti si trovano sullo stesso lato del punto iniziale della linea di divisione e la distanza euclidea fornisce un ordine rigoroso lungo la linea di divisione. Tuttavia, se il punto iniziale della linea divisa è all'interno del poligono, i punti su due lati diversi del punto iniziale della linea divisa avranno entrambi distanze positive, il che può comportare un ordinamento errato. Una formula per calcolare la distanza euclidea con segno tra due punti può essere derivata dalla formula per le proiezioni scalari . In caso di linee colineari (l'angolo tra i due vettori è `|1|`) il risultato è uguale alla distanza con segno lungo la linea di divisione.

```
static double CalcSignedDistance(const QLineF &line, const QPointF &p)
{
    return (p.x()-line.p1().x())*(line.p2().x()-line.p1().x())+
           (p.y()-line.p1().y())*(line.p2().y()-line.p1().y());
}
```

Accoppiamento dei bordi

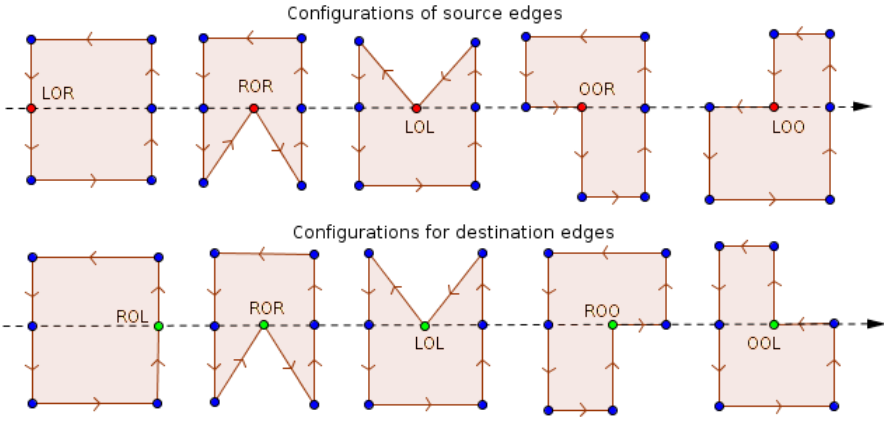
Come abbiamo visto prima, tutti gli archi nell'array `EdgesOnLine` sono bordi di origine, bordi di destinazione o entrambi. I bordi devono essere classificati di conseguenza, per poter determinare tra quali coppie di bordi deve essere creato un ponte per dividere il poligono. La classificazione di origine/destinazione è la parte più importante dell'algoritmo e la più difficile da ottenere correttamente a causa di numerosi casi limite.

Il punto iniziale `e.Start` di qualsiasi bordo `e` della `EdgesOnLine` matrice si trova sulla linea di divisione. Ogni bordo `e` ha un bordo predecessore `e.Prev` e un bordo successore `e.Next`. Possono trovarsi sul lato *sinistro* (`L`), sul lato *destro* (`R`) o *sulla* linea di divisione (`O`). Ci sono quindi un totale di nove possibilità di configurazione che devono essere valutate per scoprire se un bordo è un'origine, una destinazione o entrambi. Tutte e nove le configurazioni sono illustrate nella figura seguente. L'ordine dei vertici è mostrato nella didascalia di ciascuna configurazione.

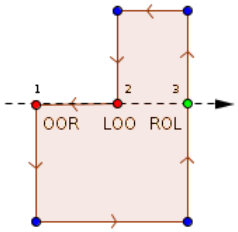


Ho semplicemente esaminato gli esempi per classificare ciascuna configurazione come origine, destinazione o entrambe. Gli ordini di avvolgimento delle configurazioni sono fondamentali per la classificazione. Immagina di muoverti lungo la linea di divisione dal punto iniziale al punto finale. L'ordine di avvolgimento nel punto di intersezione attuale indica se ci troviamo attualmente all'interno del poligono oppure no. Prendiamo come

esempio i casi `LOR` e `ROL`. Il `LOR` caso si verifica solo quando si passa dall'esterno all'interno di un poligono. Quindi, è una configurazione di origine. Al contrario, il `ROL` caso si verifica solo quando si passa dall'interno del poligono all'esterno. Quindi, è una configurazione di destinazione. Di seguito sono riportati alcuni poligoni di esempio raffigurati per la classificazione di ciascuna sorgente e ciascuna configurazione di destinazione.



Ne consegue che `LOR`, `ROR`, `LOL`, `OOR` e `LOO` sono configurazioni di origine e `ROL`, `ROR`, `LOL`, `ROO` e `OOL` sono configurazioni di destinazione. Si noti che questi due insiemi non sono disgiunti: `ROR` e `LOL` sono configurazioni di origine e destinazione. Tuttavia, queste due configurazioni possono essere origini solo se in precedenza sono servite come destinazioni! Un altro caso d'angolo si verifica quando due punti iniziali successivi del bordo giacciono sulla linea di divisione. Osserva l'esempio illustrato nella figura seguente. Ci sono due bordi sorgente (rossi) di cui il secondo deve essere utilizzato come sorgente, anche se è il secondo nell'array `EdgesOnLine`.

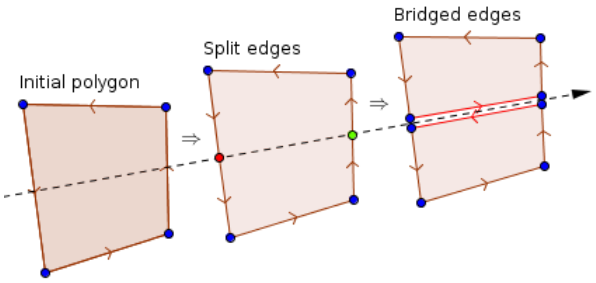


L' `EdgesOnLine` array contiene tre bordi: due bordi di origine (rosso) e un bordo di destinazione (verde). Quando si divide il poligono è fondamentale accoppiare il secondo e il terzo bordo, non il primo e il terzo. Quindi, come possiamo capire quale bordo di origine selezionare quando si esegue l'iterazione sull'array `EdgesOnLine`? Risulta che è possibile utilizzare la distanza tra i punti iniziali dei due successivi bordi sorgente in questione e il punto iniziale della linea di divisione. Un bordo di origine `OOR` e `LOO` viene selezionato solo se

il bordo candidato successivo non è più lontano dall'inizio della linea di divisione e quindi più vicino alla destinazione successiva.

Ricablare il poligono

Dopo aver capito come funziona la classificazione dei bordi di origine e di destinazione, possiamo finalmente formulare l'algoritmo per dividere i poligoni concavi. Tutto quello che dobbiamo fare è scorrere l' `EdgesOnLine` array accoppiando i bordi di origine e di destinazione. Ogni coppia indica un lato del poligono che deve essere diviso. A questo punto è utile utilizzare una rappresentazione a lista concatenata per il poligono. Il lato corrente del poligono viene diviso inserendo due nuovi bordi tra il bordo di origine e quello di destinazione. Pertanto, iteriamo sull'array `EdgesOnLine`, otteniamo il successivo bordo di origine e quello di destinazione e inseriamo tra loro due nuovi bordi orientati in modo opposto. Questa operazione *di bridging* è illustrata nella figura seguente.



Il ciclo principale del processo di ricablaggio è illustrato di seguito. Per ragioni di brevità viene mostrata solo la struttura complessiva. Per maggiori dettagli esamina l'implementazione completa nel repository github .

```
for (size_t i=0; i<EdgesOnLine.size(); i++)
{
    // find source edge
    PolyEdge *srcEdge = useSrc;

    for (; !srcEdge && i<EdgesOnLine.size(); i++)
    {
        // test conditions if current edge is a source
    }

    // find destination edge
    PolyEdge *dstEdge = nullptr;

    for (; !dstEdge && i<EdgesOnLine.size(); )
    {
        // test conditions if current edge is a destination
    }

    // bridge source and destination edges
    if (srcEdge && dstEdge)
        CreateBridge(srcEdge, dstEdge);
}
```

Raccolta dei poligoni risultanti

Il passaggio finale, dopo che tutti i bordi del ponte sono stati inseriti, è raccogliere i poligoni risultanti. A tale scopo utilizzo l' `Visited` attributo nella struttura dati del bordo per indicare se un bordo è già stato visitato e quindi è già stato assegnato a uno dei poligoni risultanti.

Categorie: Computer grafica , Strutture dati e algoritmi | *Tag:* poligono concavo , bordo , algoritmo geometrico | Collegamento permanente .

---

13 PENSIERI SU " DIVIDERE UN POLIGONO ARBITRARIO CON UNA LINEA "

lascia un commento

Glenn Burkhardt
20 gennaio 2016 alle 18:19

Ho giocato con il tuo divisore di poligoni. Credo di aver trovato un set di dati per il quale fallisce. Prova il poligono:

```
QVector{{-595.000000,-1044.000043},{823.000000, -369.000043},{334.000000, 107
```

e le linee divise:

```
{{{106.0, 185.0}, {23.0 ,92.0}}, &TestPolys[8]},
{{{23.0, 92.0}, {106.0, 185.0}}, &TestPolys[8]},
{{{-300, -269.91566265060237}, {-200, -157.86746987951807}}, &TestPolys[8]},
{{{-200, -157.86746987951807}, {-300, -269.91566265060237}}, &TestPolys[8]},
```

La linea divisa è la stessa linea definita da diversi segmenti di linea e ciascun test produce un risultato diverso. Solo il primo è corretto.

Sarei felice di continuare questa conversazione via e-mail: gbburkhardt@gmail.com . Grazie.

Rispondere

David
11 febbraio 2016 alle 12:20

Grazie per avermi informato di questo bug. Lo esaminerò. Ci hai già provato? Hai quale potrebbe essere il problema?

Rispondere

Glenn Burkhardt
17 febbraio 2016 alle 1:00

Credo di aver trovato il problema, ovvero con la funzione di confronto utilizzata per ordinare l' `EdgesOnLine` elenco. La chiave sta nella descrizione che George Vanecek fornisce di ciò che l'ordinamento dovrebbe fare:

> The edges are ordered by their source vertex in increasing distance along a cut direction (e.g., left to right)…
> Note that Dr. Vanecek used the dot product of `cutDir` with a vector with origin `refP` to the edge source point. The `cutDir` is the cross product of the unit normal vectors of the cut plane unit and the polygon `plane`.

So, in the `PolySplitter` code, the comparison function doesn't give a signed value. Thus, the order of the on edge points with respect to cut direction is lost.

I think the two points that define the cut line need to be treated like a vector, and each source vertex used with one of those two points to form a second vector, and the dot product (or scalar product) to get the value on which to sort. I think that one can safely leave out the division by the cut line vector magnitude (saving some cycles) for the purposes of your code.

```
1  double dotProduct(const QLineF &line, const QPointF &p)
2  {
3       return (p.x() - line.p1().x()) * (line.p2().x() - line.p1
4              (p.y() - line.p1().y()) * (line.p2().y() - line.p1
5  }
6
7  void PolySplitter::SortEdges(const QLineF &line)
8  {
9      // sort edges by start position relative to
10     // the start position of the split line
11     std::sort(EdgesOnLine.begin(), EdgesOnLine.end(), [&](Poly
12     {
13         return dotProduct(line, e0->StartPos) < dotProduct(lir
14     }
15 }
```

I also pulled the original code by Dr. Venecek, and found several errors. I got the code here: http://www.realtimerendering.com/resources/GraphicsGems/ and have submitted the patches to Eric Haines at the ACM, who is listed as the maintainer. Probably no one has been using that code since it was written 20 years ago, or they never bothered to submit a patch.

I was really interesting to read the original implementation in 3D, and your 2D implementation in modern C++. I haven't done a lot work with 3D

analytical geometry, and it certainly adds another (more difficult) dimensior

I was trying to use his constructor to generate a plane from three points, an discovered that there's an issue of "orientation" of the cut plane. For his unit he had explicitly defined the plane using a unit vector and distance (`Plane(Vector(0,1,0),-3)`). I wanted to try his code using 2 points to defin I defined three points to define the cut plane using the constructor he provi

```
Point p1(0, 3, 0);
Point p2(10, 3, 0);
Point p3(p1.x(), p1.y(), -10);
Point pts[] = {p1, p2, p3};
Plane cutPlane(3, pts);
```

One would think that changing `p3` to `Point p3(p1.x(), p1.y(), 10)` that c get the same plane, but for the purposes of the algorithm, one doesn't. He's Hessian normal form to represent the plane, and here's where my understa math gets fuzzy. My first guess is that these two representations should be t plane:

```
Plane(Vector(0,1,0), -3);
Plane(Vector(0,-1,0), 3);
```

but clearly for the purposes of the algorithm they are not. The code fails to c split the polygon when the second definition of the plane is used. If you can light on this issue, I'd appreciate it. You mention in your blog that your code several cases that the original Graphic Gems code did not. I'd be interested what those are.

In any case, thanks for posting your code and the explanation in your blog.

David

February 18, 2016 at 1:30 pm

Thank you very much for the detailed information. I've applied your fix already to the source code on Github. Additionally, I've investigated a bit more. Here are my findings.

I agree that the `SortEdges()` function is broken. `PointDistance()` always returns a value >= 0, because it computes the Euclidean distance between the two point arguments. This only works correctly as long as the split line's starting point, `line.p1()`, lies outside the polygon. In that case all points are on the same side of the split line's starting point and the Euclidean distance provides a strict order along the split line. If the split line's starting point is inside the polygon, points on two different sides of the split line's starting point will both have positive distances, which can result in wrong sort orders. I didn't find this issue in my tests and applications, because the split lines I use always start outside the polygon.

As you pointed out, we need a signed distance metric to fix this problem. I did some research my-self and eventually came up with the same solution you've described: basically, it's a scalar projection along the split line. In

case of co-linear lines (angle between the two vectors is `|1|`) the result equ
signed distance along the split line. Starting with the formula from Wikipedi
simply derive your solution, except that you're not normalizing `b^`. This has
effect on the sort order, as you mentioned in your comment as well already

```
s = a * b^
s = (curPoint - refPoint) * normalize(splitLineEnd - splitLineS
s = (p - splitLineStart) * normalize(splitLineEnd - splitLineSt
```

Taking this equation without the normalization gives:

```
s = (p.x - splitLineStart.x) * (splitLineEnd.x - splitLineStart
```

Which is equal to your `dotProduct()` implementation.

DJ
November 25, 2016 at 5:46 pm

It looks like the algorithm fails when I change the first test polygon (`TestPolys[0]`) with it's reverse:

```
// QVector{{-50, 50}, {-50, -50}, { 50, -50}, {50,  50}},
    QVector{{50,  50}, { 50, -50}, {-50, -50}, {-50, 50}},
```

Reply

Glenn Burkhardt
December 13, 2016 at 1:47 am

You can find a good test for CCW polygon orientation on Dan Sunday's site:
http://geomalgorithms.com/a01-_area.html
Search for `orientation2D_Polygon`.
Note that those algorithms depend on a polygon with `n` vertices having `V[0] == V[n]`. Not everyone does it that way, but it makes the code cleaner.

Reply

Glenn Burkhardt
December 4, 2016 at 4:40 pm

Please see comment in introductory paragraph "All polygons are assumed to be oriented counter clockwise.".

Reply

David
December 4, 2016 at 5:10 pm

Glenn found it! The polygon must be oriented counter clockwise for the algorithm to work. Otherwise, the checks for determining if an edge is a source/destination edge don't work anymore. In contrast, the orientation of the split line doesn't matter, because it's only used to test if the start and end point of a polygon edge lies on two different sides of the split line.

Reply

DJ
December 6, 2016 at 8:26 pm

Thanks for pointing this out. I obviously missed that very important property input polygons must satisfy.

C
May 17, 2019 at 10:53 am

Really impressive stuff. This algorithm seems to work well.
Could it be adapted to treat the line as a line segment? ie. only cut parts that fully overlap the line.
I've tried modifying it but it's a bit beyond me.

Reply

David
May 17, 2019 at 9:08 pm

I don't understand what you want to achieve. Could you describe a little more?

Reply

C
May 18, 2019 at 10:55 am

So I **think** I have something figured out, though how full-proof it is I don't know because I don't really know what I'm doing, but so far it seems to do ok.

What I mean is something like this:
https://imgur.com/a/lUQu72R

The red line is the line used to cut the split the polygon. In the current implementation it's treated like an infinite ray and it doesn't matter where the beginning and end points are (left side of the picture).

The right side is what I want to happen. If the line does not fully overlap a portion of the polygon, intersections that don't produce a full split should be ignored. eg. in the picture below, only the middle two intersection will

create a split.
https://imgur.com/a/byVLTBU

C
May 18, 2019 at 11:26 am

I've ported it to Javascript, here's the code with my changes. It's almost 1 to 1, so it should be pretty straight forward to understand. Let me know what you think, if you care to. .gist table { margin-bottom: 0; } This file contains bidirectional Unicode text that may be interpreted or compiled differently than what appears below. To review, open the file in an editor that reveals hidden Unicode characters. Learn more about bidirectional Unicode characters Show hidden characters (function() { /** * @type {Edge[]} */ const SplitPoly = []; /** * @type {Edge[]} */ const EdgesOnLine = []; const DebugIntersections = []; /** * @param {Polygon} poly * @param {Line} line * @param thickness * @param constrainToLine * @return {*} */ function Cut(poly, line, thickness = 0, constrainToLine = false) { if(poly.points.length < 3) { return null; } SplitPoly.length = 0; EdgesOnLine.length = 0; DebugIntersections.length = 0; if(thickness <= 0) { SplitEdges(poly, line, constrainToLine); if(!EdgesOnLine.length) { poly.side = GetSideOfLine(line, poly.points[0]); return null; } SortEdges(line); SplitPolygon(constrainToLine); return CollectPolys(constrainToLine); } else { const p1 = line.p1; const p2 = line.p2; const dx = p1.x – p2.x; const dy = p1.y – p2.y; const length = Math.sqrt(dx * dx + dy * dy); const nx = dx / length * thickness; const ny = dy / length * thickness; line = new Line(new Point(p1.x – ny, p1.y + nx), new Point(p2.x – ny, p2.y + nx),); const line2 = new Line(new Point(p2.x + ny, p2.y – nx), new Point(p1.x + ny, p1.y – nx),); let results; SplitEdges(poly, line); if(EdgesOnLine.length) { SortEdges(line); SplitPolygon(); results = CollectPolys(false); } else { poly.side = GetSideOfLine(line, poly.points[0]); results = [poly]; } const newResults = []; for(let resultPoly of results) { if(resultPoly.side === LineSide.Right) { SplitPoly.length = 0; EdgesOnLine.length = 0; let line2Results; SplitEdges(resultPoly, line2); if(EdgesOnLine.length) { SortEdges(line2); SplitPolygon(); line2Results = CollectPolys(false); } else { resultPoly.side = GetSideOfLine(line2, resultPoly.points[0]); line2Results = [resultPoly]; } for(let line2ResultPoly of line2Results) { if(line2ResultPoly.side === LineSide.Left) { line2ResultPoly.updateBounds(); newResults.push(line2ResultPoly); } } } else { resultPoly.updateBounds(); newResults.push(resultPoly); } } return newResults.length ? newResults : null; } } /** * @param line * @return {boolean} */ function IntersectionsAreOneLine(line) { const firstEdge = EdgesOnLine[0]; const lineDelta = line.p2.subtract(line.p1); const edgeDelta = firstEdge.start.subtract(line.p1); if(lineDelta.dot(edgeDelta) < 0) { return false; } const firstEdgeDist = PointDistance(line.p1, firstEdge.start); return firstEdgeDist + EdgesOnLine[EdgesOnLine.length – 1].distOnLine <= line.length(); } /** * @param {Line} line * @param {Point} pt * @return {LineSide} * @constructor */ function GetSideOfLine(line, pt) { const d = (pt.x – line.p1.x) * (line.p2.y – line.p1.y) – (pt.y – line.p1.y) * (line.p2.x – line.p1.x); return (d > 0.1 ? LineSide.Left : (d < -0.1 ? LineSide.Right :

```
LineSide.On)); } /** * @param {Point} pt0 * @param {Point} pt1 * @return {
@constructor */ function PointDistance(pt0, pt1) { const dx = pt0.x – pt1.x; c
pt0.y – pt1.y; return Math.sqrt(dx * dx + dy * dy); } /** * @param {Line} line
{Point} p * @return {number} * @constructor */ function CalcSignedDistanc
scalar projection on line. in case of co-linear // vectors this is equal to the si
distance. return (p.x – line.p1.x) * (line.p2.x – line.p1.x) + (p.y – line.p1.y) * (l
line.p1.y); } /** * @param {Polygon} poly * @param {Line} line * @param
constrainToLine * @return {*} */ function SplitEdges(poly, line, constrainToL
const points = poly.points; const pointCount = points.length; for(let i = 0; i <
i++) { const polyEdge = new Line(points[i], points[(i + 1) % pointCount]); cons
edgeStartSide = GetSideOfLine(line, polyEdge.p1); const edgeEndSide =
GetSideOfLine(line, polyEdge.p2); const newEdge = new Edge(points[i], edge
SplitPoly.push(newEdge); if(edgeStartSide === LineSide.On) { // This edge is
intersection but a vertex on the source polygon, mark it as such
newEdge.isSourceVertex = true; if(constrainToLine) { const t = GetEdgeLineT(
line); const isOnLine = t >=0 && t <= 1; // Only add this vertex as on edge if it
line, and between the line's start and end points if(isOnLine) {
DebugIntersections.push(newEdge.start); EdgesOnLine.push(newEdge); } //
line but not inside it, make distOnLine negative so it will pass LOO and OOR
edge // during SplitPolygon else { newEdge.distOnLine = -1; } } else {
DebugIntersections.push(newEdge.start); EdgesOnLine.push(newEdge); } } e
if(edgeStartSide !== edgeEndSide && edgeEndSide !== LineSide.On) { const
polyEdge.intersect(line, !constrainToLine); // If constrainToLine == true, the
be null for intersection that lie outside the line's // start and end points
if(!constrainToLine) { console.assert(result != null); } if(result) { const edge =
Edge(result, LineSide.On); DebugIntersections.push(result); SplitPoly.push(e
EdgesOnLine.push(edge); } } } // connect doubly linked list, except // first->p
>next for(let i = 0; i < SplitPoly.length – 1; i++) { const current = SplitPoly[i]; c
SplitPoly[i + 1]; current.next = next; next.prev = current; } // connect first->pr
>next const front = SplitPoly[0]; const back = SplitPoly[SplitPoly.length – 1]; b
front; front.prev = back; } /** * @param {Line} line * @return {*} */ function
SortEdges(line) { // sort edges by start position relative to // the start positio
split line EdgesOnLine.sort((e0, e1) => { // it's important to take the signed d
here, // because it can happen that the split line starts/ends // inside the po
that case intersection points // can fall on both sides of the split line and tak
unsigned distance metric will result in wrongly // ordered points in EdgesOn
CalcSignedDistance(line, e0.start) – CalcSignedDistance(line, e1.start); }); // c
distance between each edge's start // position and the first edge's start posi
count = EdgesOnLine.length; if(count) { const firstEdge = EdgesOnLine[0]; fo
count; i++) { const edge = EdgesOnLine[i]; edge.distOnLine = PointDistance(e
firstEdge.start); } } } /** * Returns the percentage (0-1) along the line where
lies * Edges outside of the line will be outside the range of 0 – 1 * @param {
* @param {Line} line */ function GetEdgeLineT(edge , line) { const lineDelta
line.p2.subtract(line.p1); const edgeLineDelta = edge.start.subtract(line.p1);
edgeLineDelta.x /= lineDelta.x; edgeLineDelta.y /= lineDelta.y; return
!Number.isNaN(edgeLineDelta.x) ? edgeLineDelta.x : edgeLineDelta.y; } func
SplitPolygon(constrainToLine = false) { /** @type {Edge} */ let useSrc = null;
= EdgesOnLine.length; for(let i = 0; i < count; i++) { // find source let srcEdge
```

```
useSrc = null; for(; !srcEdge && i < count; i++) { let curEdge = EdgesOnLine[i];
curSide = curEdge.side; const prevSide = curEdge.prev.side; const nextSide =
curEdge.next.side; console.assert(curSide === LineSide.On); if((prevSide ===
LineSide.Left && nextSide === LineSide.Right) || (prevSide === LineSide.Left
nextSide === LineSide.On && curEdge.next.distOnLine < curEdge.distOnLine
(prevSide === LineSide.On && nextSide === LineSide.Right && curEdge.prev.
< curEdge.distOnLine) ) { srcEdge = curEdge; srcEdge.isSrcEdge = true; } // In
cases LOL or ROR can be considered sources if the edges formed by prev > c
are concave else if(constrainToLine) { const isOutside =
CalculateTriangleArea(curEdge.prev.start, curEdge.start, curEdge.next.start)
(prevSide === LineSide.Left && nextSide === LineSide.Left && isOutside) || (
=== LineSide.Right && nextSide === LineSide.Right && isOutside) ) { srcEdge
srcEdge.isSrcEdge = true; } } } // find destination /** @type {Edge} */ let dstE
for(; !dstEdge && i < count;) { let curEdge = EdgesOnLine[i]; const curSide =
curEdge.side; const prevSide = curEdge.prev.side; const nextSide = curEdge.
console.assert(curSide === LineSide.On); if((prevSide === LineSide.Right &&
=== LineSide.Left) || (prevSide === LineSide.On && nextSide === LineSide.Le
(prevSide === LineSide.Right && nextSide === LineSide.On) || (prevSide ===
LineSide.Right && nextSide === LineSide.Right) || (prevSide === LineSide.Le
nextSide === LineSide.Left)) { dstEdge = curEdge; dstEdge.isDstEdge = true; }
} // bridge source and destination if(srcEdge && dstEdge) { // Mark these edg
of a bridge so we know to keep them during CollectPolys srcEdge.isBridge =
dstEdge.isBridge = true; CreateBridge(srcEdge, dstEdge); VerifyCycles(); // is
configuration in which a vertex // needs to be reused as source vertex?
if(srcEdge.prev.prev.side === LineSide.Left) { useSrc = srcEdge.prev; useSrc.is
true; } else if(dstEdge.next.side === LineSide.Right) { useSrc = dstEdge; useSr
= true; } } } } /** * signed area of a triangle * @param {Point} p * @param {P
@param {Point} r * @return {number} */ function CalculateTriangleArea(p, q
(q.y – p.y) * (r.x – q.x) – (q.x – p.x) * (r.y – q.y); } /** * @param {Edge} srcEdg
{Edge} dstEdge */ function CreateBridge(srcEdge, dstEdge) { const a = new
Edge(srcEdge.start, srcEdge.side); const b = new Edge(dstEdge.start, dstEdge
SplitPoly.push(a); SplitPoly.push(b); a.isBridge = srcEdge.isBridge; b.isBridge
dstEdge.isBridge; a.isSourceVertex= srcEdge.isSourceVertex; b.isSourceVerte
dstEdge.isSourceVertex; a.next = dstEdge; a.prev = srcEdge.prev; b.next = src
b.prev = dstEdge.prev; srcEdge.prev.next = a; srcEdge.prev = b; dstEdge.prev
dstEdge.prev = a; } function VerifyCycles() { const splitPolyCount = SplitPoly.l
for(let edge of SplitPoly) { let curSide = edge; let count = 0; do { if(count >
splitPolyCount) { throw new Error('count > splitPolyCount'); } // console.asse
splitPolyCount); curSide = curSide.next; count++; } while(curSide !== edge); }
CollectPolys(updateBounds = true, constrainToLine = false) { const resPolys =
console.log(SplitPoly); for(const e of SplitPoly) { if(!e.visited) { const splitPoly
Polygon(); splitPoly.side = e.side; let curSide = e; do { if(splitPoly.side === Lin
&& curSide.side !== LineSide.On) { splitPoly.side = curSide.side; } curSide.vis
// Ignore edges that do not form a complete split in the polygon if(!constrair
curSide.side !== LineSide.On || curSide.isSourceVertex || curSide.isBridge)
splitPoly.points.push(curSide.start); } curSide = curSide.next; } while(curSide
if(updateBounds) { splitPoly.updateBounds(); } resPolys.push(splitPoly); } } re
resPolys; } window.PolyCut = { Cut: Cut, DebugIntersections: DebugIntersect
```

visualizza PolySplitter.js raw ospitato con ♥ da GitHub Non riesco a contras
righe su github, quindi elencherò le modifiche qui. (Ho aggiunto commenti
spiegano) – Ho aggiunto un'opzione vincolaToLine. – In SplitEdges: — 199-2
230-243 – SplitPolygon — righe 344-357 — righe 392-394 – CreateBridge —
441 – CollectPolys — righe 500-504

lascia un commento

← Articolo precedente Articolo successivo →