The Vector Graphics in Ariel

The topic is about how to implement a **graphics system** which draws **vector arts**. In this paper I will present all the steps to reach the goal, explain why some of the designs would be made like so, the purposes, and the traps you should avoid of.

Goals

Our goal was mainly about to implement a program that draws 2D vector paths into images, which was similar to a RIP(raster image processor).

The input of the paths was in a form like:

```
moveTo 100.320000, 159.139999;
cubicTo 41.509998, -15.280000, 224.860001, 71.500000, 287.140015, 27.680000;
lineTo 432.429993, 19.889999;
lineTo 329.510010, 224.000000;
lineTo 163.460007, 0.000000;
lineTo 0.000000, 261.190002;
cubicTo 206.699997, 87.930000, 299.239990, 163.750000, 448.859985, 115.029998;
cubicTo 313.660004, 68.320000, 296.940002, 217.080001, 100.320000, 159.139999;
```

In our implementation, the candidates of each path segment could be *moveTo, lineTo, quadTo, cubicTo*.

Furthermore, such a system usually contains various of filling and stroking styles, like *solid fill, gradient fill, pattern fill, picture fill*, etc. In our implementation, we will choose to implement solid fill, picture fill as they were representative of any other filling styles.

This was a summary of our goals, seems pretty simple. The details of each subject will be explained in the following chapters.

Clipping

As the paths were described in the way we stated above, we were actually dealing with arbitrary polygons, which was very unhandy. So the first step we need to take was to convert **arbitrary polygons** into **simple polygons**, according to some certain rules. This step was usually called clipping.

There were already a lot of former research papers on this subject, like the Vatti clipping algorithm[1], the Greiner-Hormann clipping algorithm[2], etc.

You could also take a look of the other libraries implemented such functions, like the QPainterPath in QT, or the Clipper[3], etc. The Clipper has an issue that it use integer to represent the path which I thought might cause precision problems. The QPainterPath has a method called *simplified()* does the same works. The Clipper cannot handle the curve segments in the path, while the *QPainterPath::simplified()* may convert the curve segments into line segments when they were intersected.

Since we are going to implement the Loop Blinn algorithm[4] to render the paths, we need to keep as many curve segments as we could in the paths. So I decided to implement a clipping algorithm based on the sweep line algorithm which can split curve segments on their intersections. In this way we could get a simple path has much curve detail.

This algorithm was in the following links:

https://github.com/lymastee/gslib/blob/master/code/pink/clip.h
https://github.com/lymastee/gslib/blob/master/code/pink/clip.cpp

1 A generic solution to polygon clipping. By Bala.R.Vatti 1992
2 Efficient clipping of arbitrary polygons. By Greiner.Günther & Kai.Hormann 1998
3 http://www.angusj.com/delphi/clipper.php
4 Resolution independent curve rendering using programmable graphics hardware. By Charles.Loop & Jim Blinn 2003

This algorithm was currently unfinished. Here was a list about things done or left undone below:

1. XOR clipping method, the most important clipping method.(√)
2. OR clipping method, combined polygons.(×)
3. AND clipping method, intersected polygons.(×)
4. pivot joint, which deals with the multiple intersections that coincidently happen.(×)

These clipping methods was corresponded to certain Boolean operations on 2D polygons, besides, on the self-intersected exception, which was called the complex polygons. A table about the connections of the *clipping methods* | *Boolean ops* | *filling rules* was below:

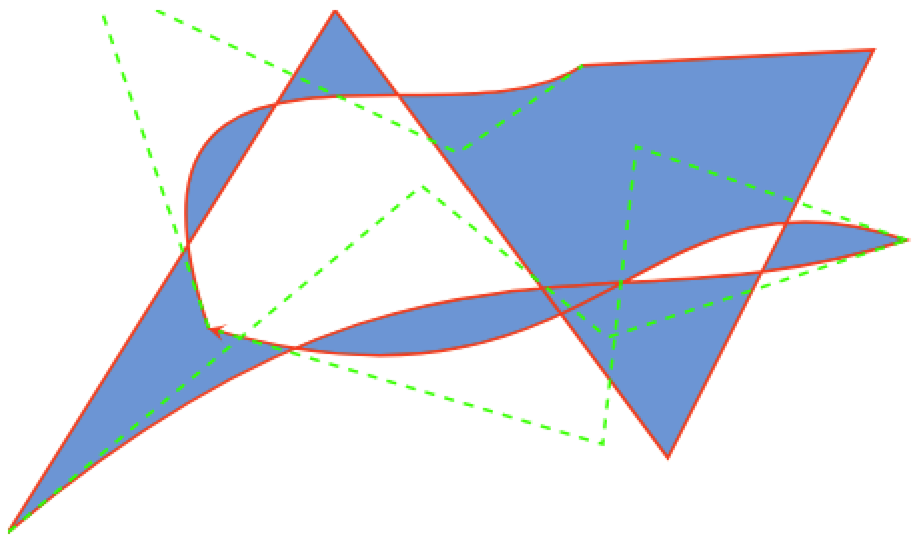
Clipping methods	*Boolean ops*	*Filling rules*
XOR	XOR | exclude	OddEven(windings start by 1)
OR	OR | union	OnePlus
AND	AND | intersect	OddEven(windings start by 0)

This algorithm was divided into 2 steps.

1. apply the sweep algorithm, create sweeper nodes on it.
2. assemble the output path

The main difference between these clipping methods was in the 2nd step. So we could easily implement the rest of the clipping methods in the future.

The following pictures show the difference before or after the clipping ops.



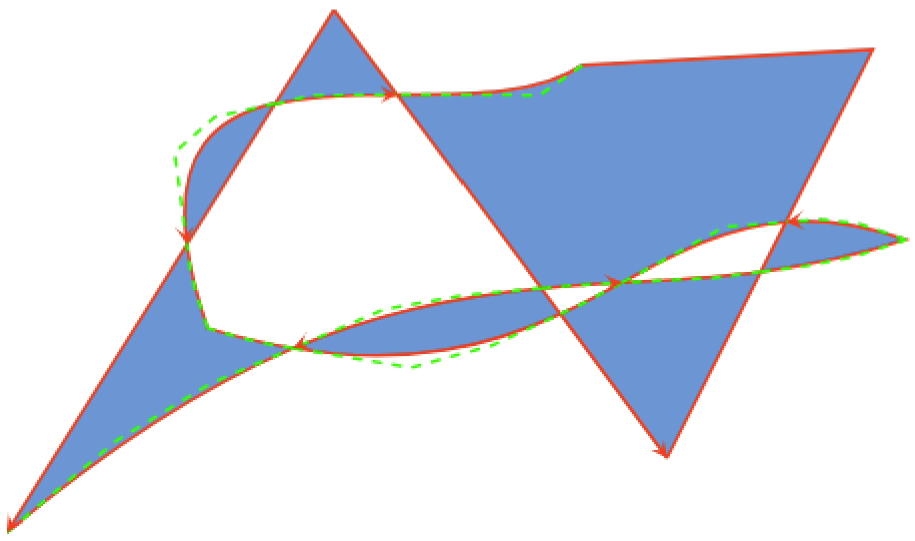
Before clipping

After we applied the clipping algorithm, the path we raised in the **Goals** chapter became:

```
moveTo 143.101517, 169.124496;
cubicTo 202.111404, 178.865891, 242.335800, 167.602325, 276.162872, 152.005249;
lineTo 266.593323, 139.130432;
cubicTo 228.411209, 142.622543, 188.602112, 149.664520, 143.101517, 169.124496;
moveTo 306.878296, 136.353683;
cubicTo 333.441528, 122.079879, 358.197174, 108.520424, 388.929291, 106.162598;
lineTo 432.429993, 19.889999;
lineTo 287.140015, 27.680000;
cubicTo 265.772797, 42.713894, 230.154938, 42.375397, 194.954712, 42.486526;
lineTo 266.649292, 139.201477;
cubicTo 280.166382, 137.889053, 293.533844, 137.096252, 306.878296, 136.353683;
moveTo 388.929291, 106.162598;
lineTo 376.430481, 130.944458;
cubicTo 399.479492, 128.026931, 423.339386, 123.340141, 448.859985, 115.029998;
cubicTo 425.969543, 107.121620, 406.475342, 104.816429, 388.929291, 106.162598;
moveTo 194.954712, 42.486526;
lineTo 163.460007, 0.000000;
lineTo 133.911316, 47.215355;
cubicTo 152.025711, 42.790951, 173.412003, 42.554535, 194.954712, 42.486526;
moveTo 329.510010, 224.000000;
lineTo 376.430481, 130.944458;
cubicTo 352.695862, 133.948761, 329.821106, 135.076995, 306.881287, 136.353210;
cubicTo 297.023040, 141.650970, 286.916199, 147.047073, 276.162872, 152.005249;
lineTo 329.510010, 224.000000;
```

```
moveTo 89.968872, 117.434502;
lineTo 134.158203, 46.822594;
cubicTo 103.929977, 53.643314, 83.773109, 71.326660, 89.968872, 117.434502;
moveTo 0.000000, 261.190002;
cubicTo 54.618332, 215.407837, 101.265717, 187.017089, 143.104340, 169.102432;
cubicTo 129.839050, 166.907532, 115.624512, 163.649933, 100.320000, 159.139999;
cubicTo 94.925667, 143.141357, 91.568726, 129.340317, 89.968872, 117.434502;
lineTo 0.000000, 261.190002;
```

If we drew it below, look to those arrows that showed the path segments:



After clipping

Unlike the QT, the raster engine in Ariel could handle both quadratic and cubic Bezier segments in the path, without having the quadratic Bezier curves converted into cubic ones, which could probably benefits the font renderings. All we have to do was to calculate the intersections in all the different situations, *linear – linear, linear – quad, linear – cubic, quad – quad, quad – cubic, cubic – cubic*. The implementation was in the following links:

https://github.com/lymastee/gslib/blob/master/code/pink/utility.h
https://github.com/lymastee/gslib/blob/master/code/pink/utility.cpp

Most of the situations we could deal with by solving **univariate quadratic | cubic | quartic** equations, which was very efficient. When dealing with the cubic – cubic intersection, I divide the curves by their inflections, then use the Newton-

Raphson iteration, which may have a **precision problem** here. I will improve the method in the future, or take consider in the Bezier clipping[5] method.

[5] Curve intersection using Bezier clipping. By T.W.Sederberg & T.Nishita

Loop Blinn Algorithm Part. 1

Loop Blinn algorithm was first raised by the loopblinn05[6] paper. Here I don't want to talk about its theory, but to implement it.

The principle was that a curve could be classified into **Loop | Cusp | Serpentine** categories by finding their klm coordinates. In which we could use the klm coordinates in pixel shader to render them. The algorithm was implemented in the following links:

https://github.com/lymastee/gslib/blob/master/code/pink/utility.h
https://github.com/lymastee/gslib/blob/master/code/pink/utility.cpp

See function: *get_cubic_klmcoords()*

The **quadratic** situation was very simple, the klm coordinates were below:

Quadratic	*C0*	*C1*	*C2*
k	0	0.5	1
l	0	0	1
m	1	1	1

To the **cubic** situation, denote the four control points as *C0, C1, C2, C3*, to get the intermediate vector *d1, d2, d3*, the pseudo-codes were below:

```
float3 b0(C0.xy, 1), b1(C1.xy, 1), b2(C2.xy, 1), b3(C3.xy, 1);
float a1 = dot(b0, cross(b3, b2));
float a2 = dot(b1, cross(b0, b3));
float a3 = dot(b2, cross(b1, b0));
float d1 = a1 – 2 * a2 + 3 * a3;
float d2 = –a2 + 3 * a3;
float d3 = 3 * a3;
d1, d2, d3 = normalize(float3(d1, d2, d3));
```

6 Resolution independent curve rendering using programmable graphics hardware. By Charles.Loop & Jim Blinn 2003

Then we could calculate the discriminant (denote as **discr**) by:

$$sbst = 3d2^2 - 4d1d3,\ discr = d1^2\ \times sbst$$

The categories of the curve types were below:

discr = 0		**discr** > 0	**discr** < 0
d1 = 0 & **d2** = 0 & **d3** = 0	line	serpentine	loop
d1 = 0 & **d2** = 0 & **d3** ≠ 0	quadratic		
d1 ≠ 0 & **sbst** = 0	cusp1		
d1 = 0 **& d2** ≠ 0	cusp2		
d1 ≠ 0 & **sbst** < 0	loop		
d1 ≠ 0 & **sbst** > 0	serpentine		

To the **Serpentine** case:

ls	$3d2 - \sqrt{9d2^2 - 12d1 \cdot d3}$	*ms*	$3d2 + \sqrt{9d2^2 - 12d1 \cdot d3}$
lt	$6d1$	*mt*	$6d1$

Serpentine	*C0*	*C1*	*C2*	*C3*
k	$ls \cdot ms$	$\frac{3ls \cdot ms - ls \cdot mt - lt \cdot ms}{3}$	$\frac{lt \cdot (mt - 2ms) + ls \cdot (3ms - 2mt)}{3}$	$(lt - ls) \cdot (mt - ms)$
l	ls^3	$-ls^2 \cdot (lt - ls)$	$(lt - ls)^2 \cdot ls$	$-(lt - ls)^3$
m	ms^3	$-ms^2 \cdot (mt - ms)$	$(mt - ms)^2 \cdot ms$	$-(mt - ms)^3$

If d1 < 0, then reverse the coordinates.

To the **Cusp** case:

ls	$d3$	*lt*	$3d2$

Cusp	*C0*	*C1*	*C2*	*C3*
k	ls	$ls - lt/3$	$ls - 2lt/3$	$ls - lt$
l	ls^3	$ls^2 \cdot (ls - lt)$	$(ls - lt)^2 \cdot ls$	$(ls - lt)^3$
m	1	1	1	1

To the **Loop** case:

ls	$d2 - \sqrt{4d1 \cdot d3 - 3d2^2}$	*ms*	$d2 + \sqrt{4d1 \cdot d3 - 3d2^2}$
lt	$2d1$	*mt*	$2d1$
ql	ls/lt	*qm*	ms/mt

Loop	*C0*	*C1*	*C2*	*C3*
k	$ls \cdot ms$	$\frac{-ls \cdot mt - lt \cdot ms + 3ls \cdot ms}{3}$	$\frac{lt \cdot (mt - 2ms) + ls \cdot (3ms - 2mt)}{3}$	$(lt - ls) \cdot (mt - ms)$
l	$ls^2 \cdot ms$	$\frac{ls \cdot (ls \cdot (mt - 3ms) + 2lt \cdot ms)}{-3}$	$\frac{(lt - ls) \cdot (ls \cdot (2mt - 3ms) + lt \cdot ms)}{3}$	$-(lt - ls)^2 \cdot (mt - ms)$
m	$ls \cdot ms^2$	$\frac{ms \cdot (ls \cdot (2mt - 3ms) + lt \cdot ms)}{-3}$	$\frac{(mt - ms) \cdot (ls \cdot (mt - 3ms) + 2lt \cdot ms)}{3}$	$-(lt - ls) \cdot (mt - ms)^2$

If $d1 > 0$, $ls \cdot ms < 0$ or $d1 < 0$, $ls \cdot ms > 0$, then reverse the coordinates.

Notice that the Loop case may have artifacts, take the denotation *ql, qm* above, the situations were:

$ql > 0$ & $ql < 1$	$qm > 0$ & $qm < 1$
split at **ql**…	split at **qm**…

Sometimes the divided curves may still have artifacts, and kept having artifacts all the time, this recursion may cause stack overflow. So if the curve segment was short enough, then convert the curve segment into line segment.

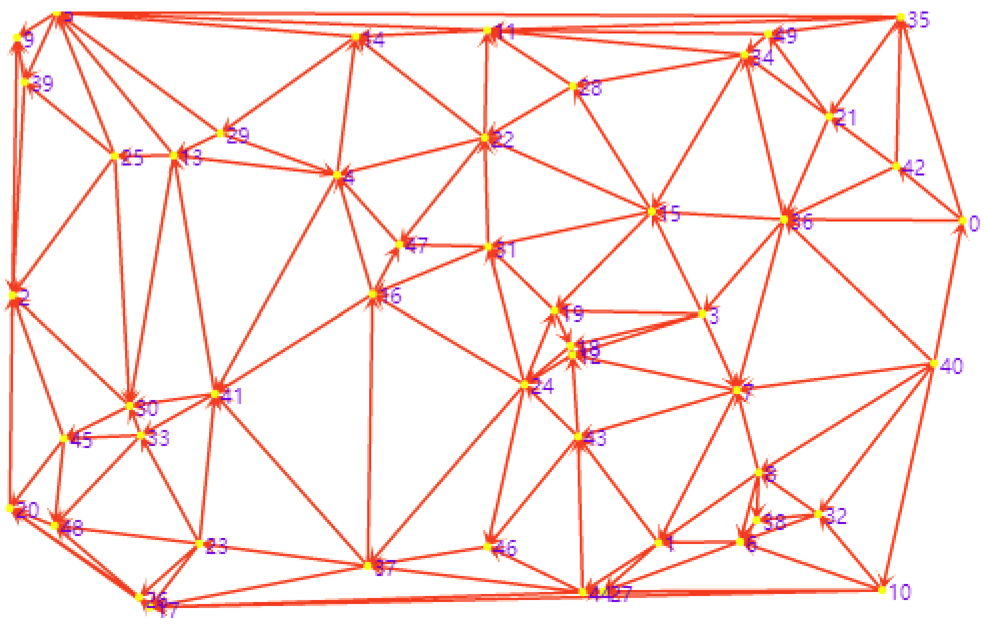
Loop Blinn Algorithm Part. 2

This chapter I will talk about how to tessellate the path. Currently the tessellation work was done on the CPU side. The main procedure was a constrained Delaunay triangulation, divide and conquer[7] method.

The constrained Delaunay triangulation was implemented in the following links:

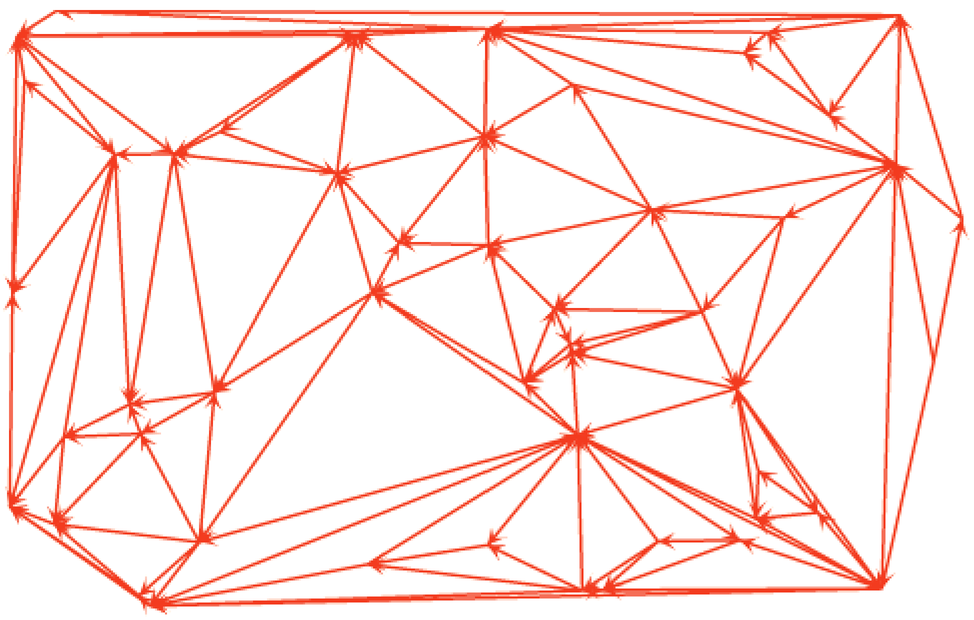
https://github.com/lymastee/gslib/blob/master/code/ariel/delaunay.h
https://github.com/lymastee/gslib/blob/master/code/ariel/delaunay.cpp

There was a project named “cdt” under the solution could be used to test this algorithm. The results of a few random tests were below:

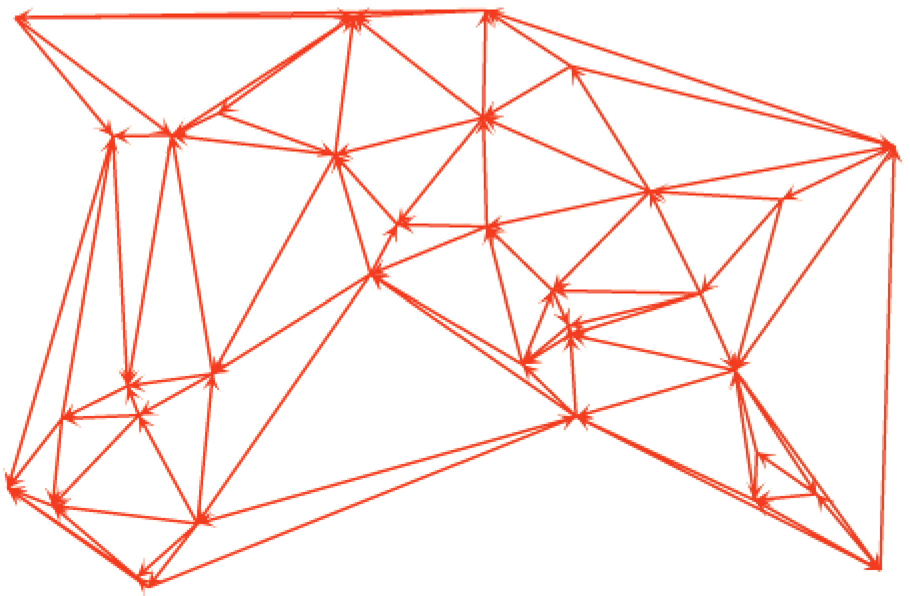


After input constraints *9,25,20,17,43,10,42,11,9* it became:

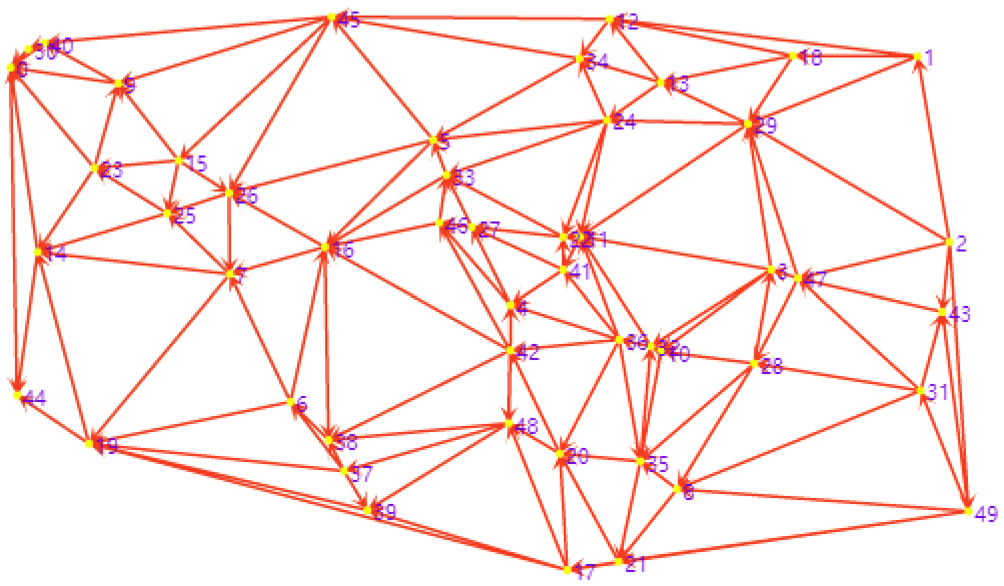
[7] Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. By Guibas.L & Stolfi.J 1985



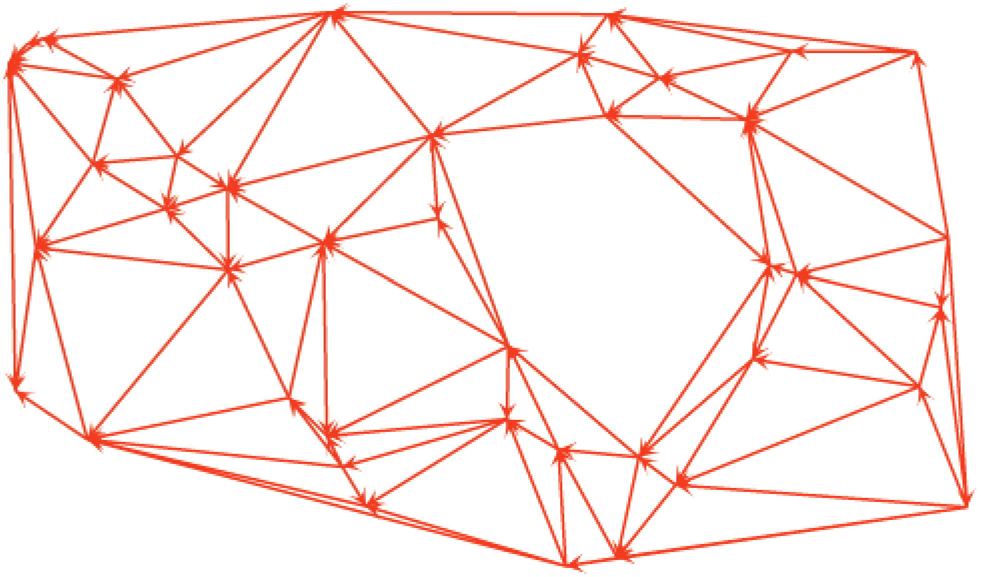
After trimming it became:



Input constraints counter-clockwise(considered the *depath* tool was y-axis down, so it seems like clockwise) to make a hole.



After input constraints *5,24,3,35,42,5* and trimming, it became:



Back to the topic, the implementation of the tessellation for Loop Blinn rendering was in the following links:

https://github.com/lymastee/gslib/blob/master/code/ariel/loopblinn.h
https://github.com/lymastee/gslib/blob/master/code/ariel/loopblinn.cpp

The main steps of the tessellation were:

```
void loop_blinn_processor::proceed(const painter_path& path)
{
	flattening(path);			// 1.flattening
	if(_polygons.empty())
		return;
	for(auto* p : _polygons) {		// 2.check overlays
		check_boundary(p);
		check_holes(p);
		check_rtree(p);
	}
	calc_klm_coords();			// 3.calculate klm coordinates
	for(auto* p : _polygons)		// 4.build constrained Delaunay triangulation
		p->build_cdt();
}
```

The explanations of each step were:

1. Flattening:

   After the flattening step, the path will be organized into the form *boundary + holes*.

2. Check boundary:

   This step would try to avoid the situation that the control path of the boundary runs outside the boundary itself.

3. Check holes:

   Unlike the boundary, every control points on the hole would have a chance to cross the boundary. This step would try to avoid it.

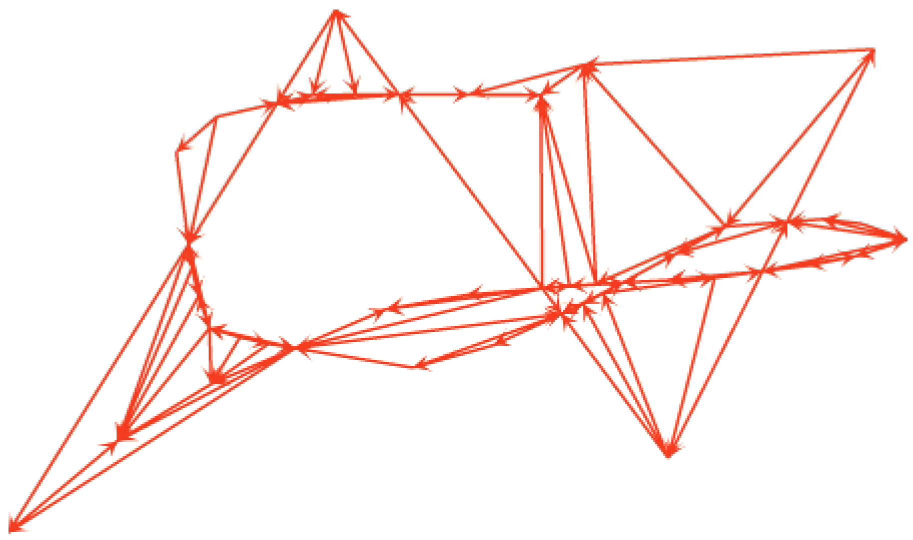
4. Check r-tree:

   If the polygon had holes, then each hole has a chance to overlap another. Here we use the r-tree structure to determine if two holes overlapped.
   https://github.com/lymastee/gslib/blob/master/code/gslib/rtree.h

5. Build cdt:

   For each polygon, build up the constrained Delaunay triangulation structure.

After the tessellation, the path we given in the **Goals** chapter now became:



Batching

The batching for the rendering was to reduce the draw calls. As the pixel shader usually have a heavy load, we cannot put them in a single batch. So we need to make a proper design for the batching.

The principle was:

1. The input of the IA stage should be as simple as possible.
2. The area of the primitives which have a complex shader process should be as small as possible.

The implementation of the batching, along with other supportive functions were in the following links:

https://github.com/lymastee/gslib/blob/master/code/ariel/rose.h
https://github.com/lymastee/gslib/blob/master/code/ariel/rose.cpp
https://github.com/lymastee/gslib/blob/master/code/ariel/rosed3d11.cpp
https://github.com/lymastee/gslib/blob/master/code/ariel/rose.hlsl

The batches were currently divided into *fill_cr, fill_klm_cr, stroke_coef_cr, fill_klm_tex, stroke_coef_tex*. Here we list their input structures (layout) below:

```
struct vertex_info_cr
{
        vec2                    pos;
        vec4                    cr;
};
struct vertex_info_klm_cr
{
        vec2                    pos;
        vec3                    klm;
        vec4                    cr;
};
struct vertex_info_coef_cr
{
        vec2                    pos;
        vec4                    coef;           /* float3 coef & float tune */
```

```
	vec4			cr;
};
struct vertex_info_klm_tex
{
	vec2			pos;
	vec3			klm;
	vec2			tex;
};
struct vertex_info_coef_tex
{
	vec2			pos;
	vec4			coef;
	vec2			tex;
};
```

1. The reason why *fill_klm_cr, fill_cr* would be separated was klm shader for loop blinn rendering was more complex than normal fill.

2. The reason why the stroke batches (like *stroke_coef_cr, stroke_coef_tex*) would be separated was the stroke batches were also used for anti-aliasing, the pixel shader of them were more complex than normal fill.

3. The reason why I put *fill_tex* (there's no *fill_tex*) together with *fill_klm_tex* was the texture batching was more complex, and maybe more expensive.

4. The reason why I didn't put the color into the constant buffers was it's easy to batch solid fill with different colors together, and it's easy to implement a batched gradient fill in the future.

Seeing the following links for details:

https://github.com/lymastee/gslib/blob/master/code/ariel/batch.h
https://github.com/lymastee/gslib/blob/master/code/ariel/batch.cpp

The kernel of the batching process was still an r-tree. Each layer of the batching process was an individual r-tree, each triangle of different type was tested by the r-tree, if not overlapping, then insert it into the layer; otherwise create a new layer (batch) and insert it into the new layer.

The texture batching was a bit special. When dealing with the picture fill batch, we should always prefer texture batching result than geometry relationships. To reuse the texture batching result, there would be a *stroke_coef_tex* batch (for anti-aliasing use) after the *fill_klm_tex* batch always.

Next chapter we will talk about how to deal with texture batching (IE. To generate a texture atlas by runtime).

Bin Packing

The texture batching problem was actually a bin packing problem. There were a lot of former research works on this subject, too. For example the rectangle bin pack[8] algorithm:

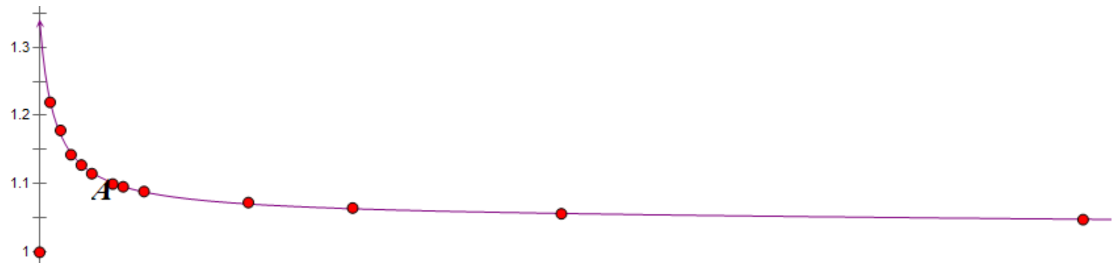
https://github.com/juj/RectangleBinPack

This problem was NP-hard. So there's no final conclusion on this subject. As we needed to pack the bin in the runtime, we would always prefer efficiency than packing quality.

I wrote the algorithm based on this link:

http://blackpawn.com/texts/lightmaps/default.html

The basic idea was to predict the **area expand ratio** so that most of the time the prediction could succeed in the first guess. I got this result by curve fit:



In this case there may be some wastes on the texture atlas, but not too much. This procedure was called *pack_compactly()*.

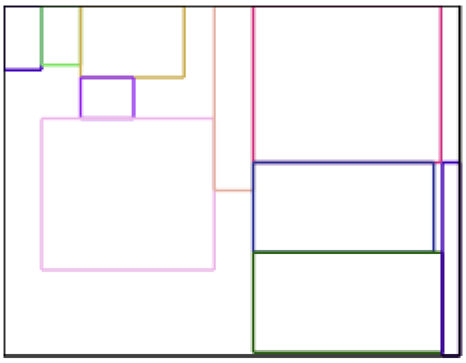
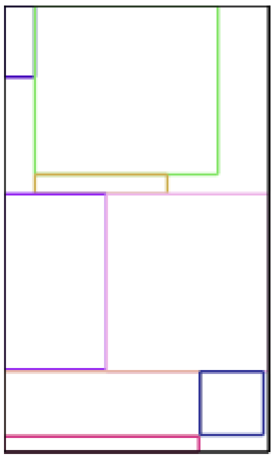
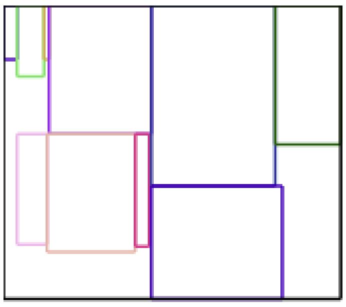
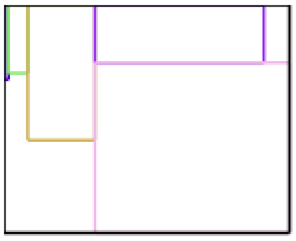
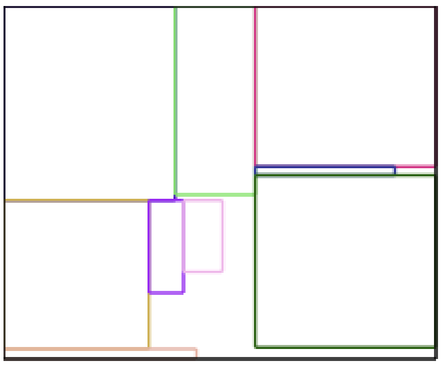
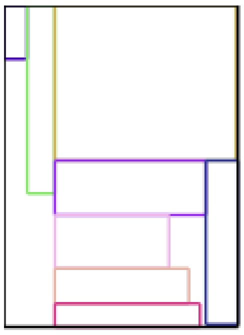
When the input rectangles less than 10 (that means very less), the prediction was either too wasteful or too inaccurate. So I wrote a new algorithm called *pack_dynamically()* to handle this.

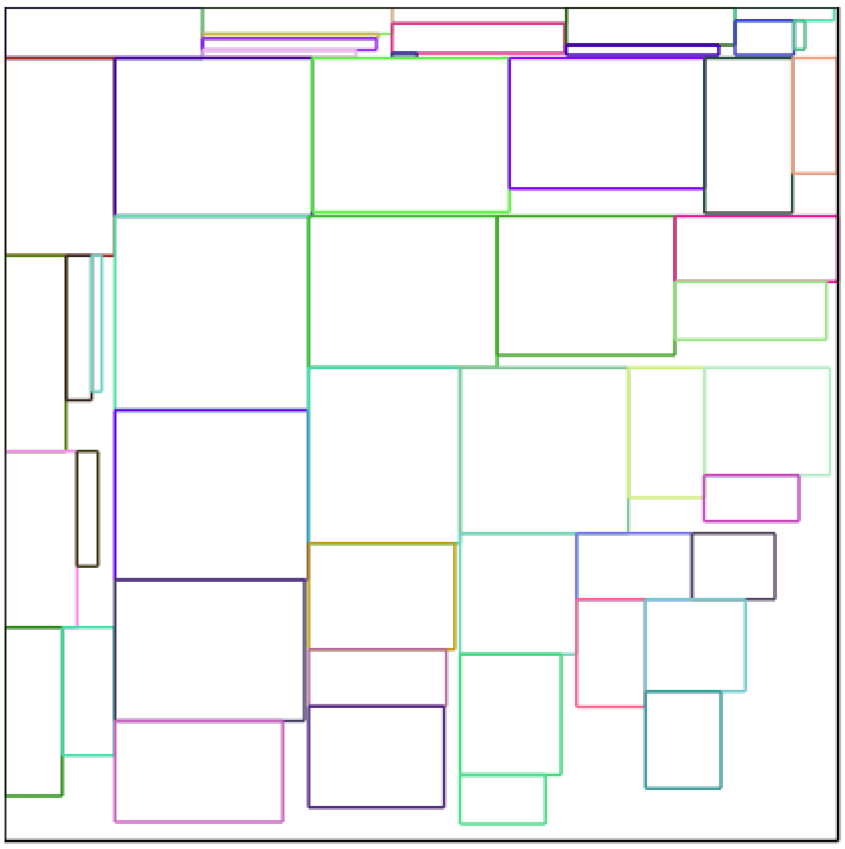
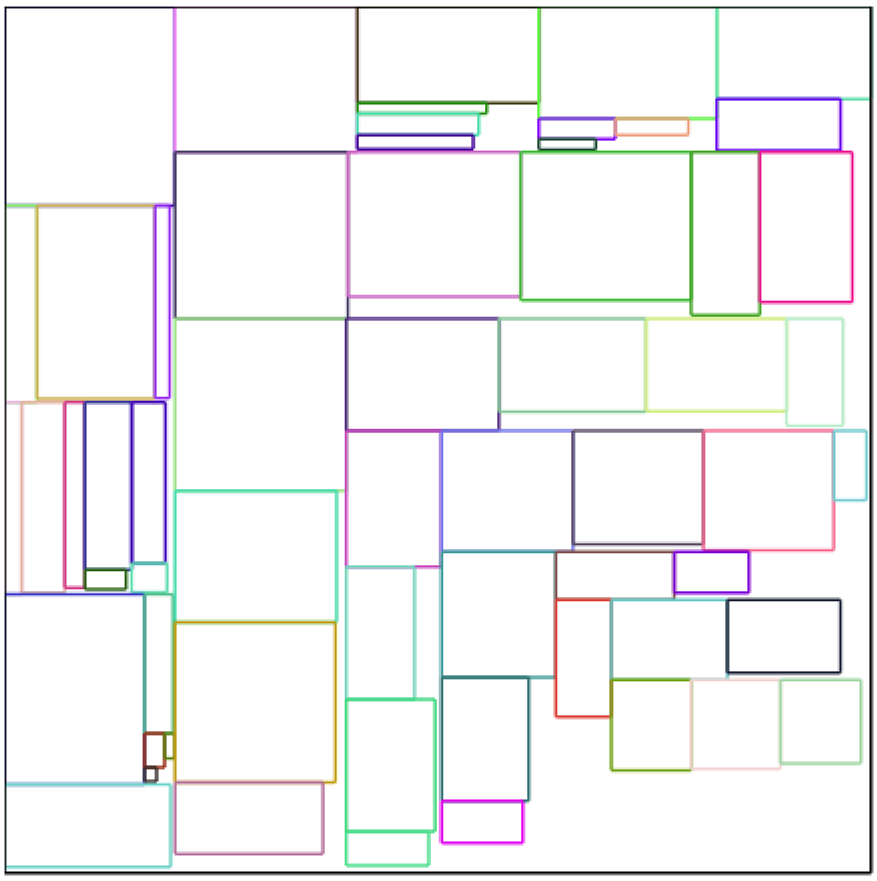
This algorithm was implemented in the following links:

[8] A Thousand Ways to Pack the Bin – A Practical Approach to Two-Dimensional Rectangle Bin Packing – By Jukka. Jylänki 2010

https://github.com/lymastee/gslib/blob/master/code/ariel/rectpack.h
https://github.com/lymastee/gslib/blob/master/code/ariel/rectpack.cpp
https://github.com/lymastee/gslib/blob/master/code/ariel/texbatch.h
https://github.com/lymastee/gslib/blob/master/code/ariel/texbatch.cpp

There was a project called *rectpack* in the test folder for testing the algorithm. Here was a few random test results below:

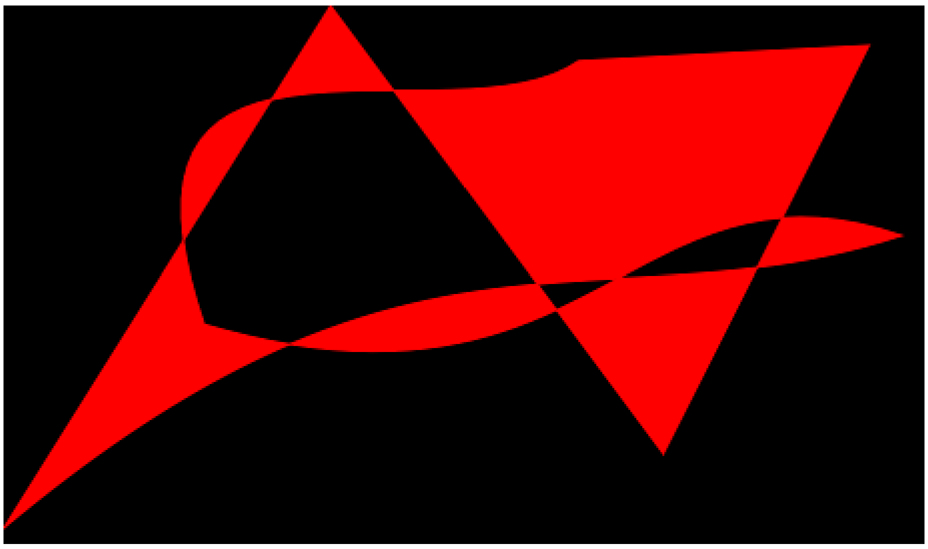







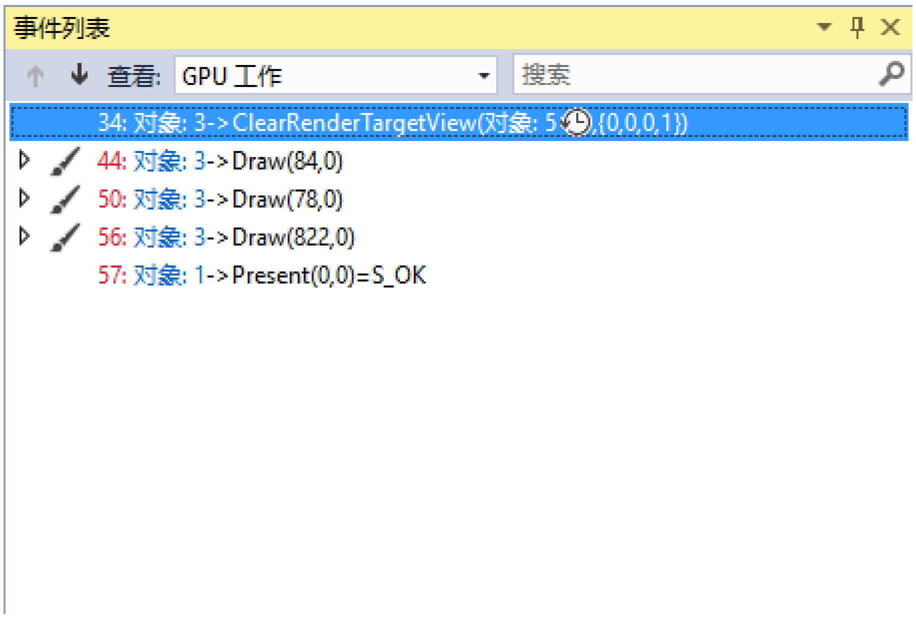
Results

Solid fill: RGB(255,0,0) | Stroke: none

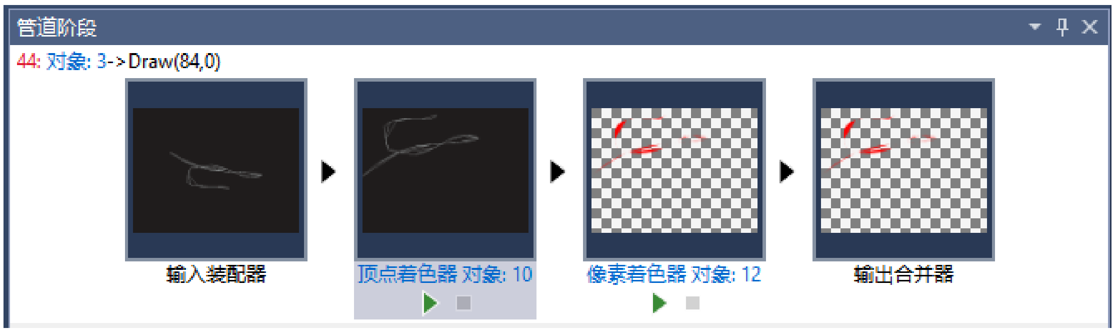
Final result:

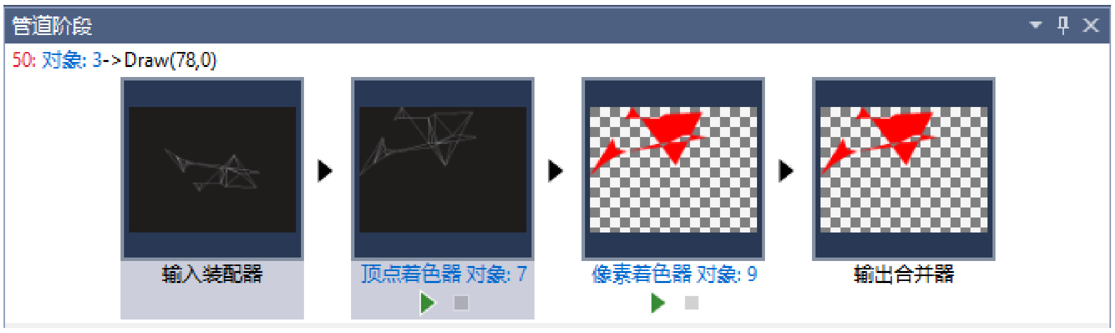


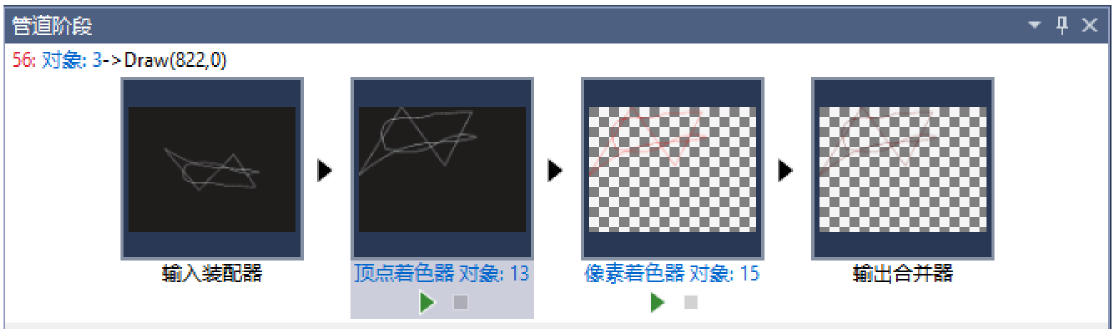
3 Batches: LoopBlinn batch, common batch, anti-aliasing batch



```
34: 对象: 3->ClearRenderTargetView(对象: 5,{0,0,0,1})
44: 对象: 3->Draw(84,0)
50: 对象: 3->Draw(78,0)
56: 对象: 3->Draw(822,0)
57: 对象: 1->Present(0,0)=S_OK
```

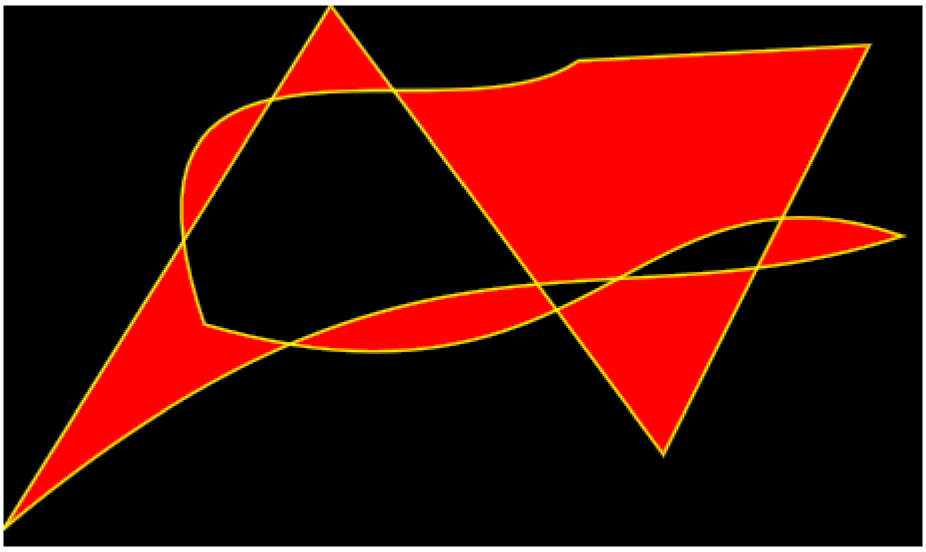




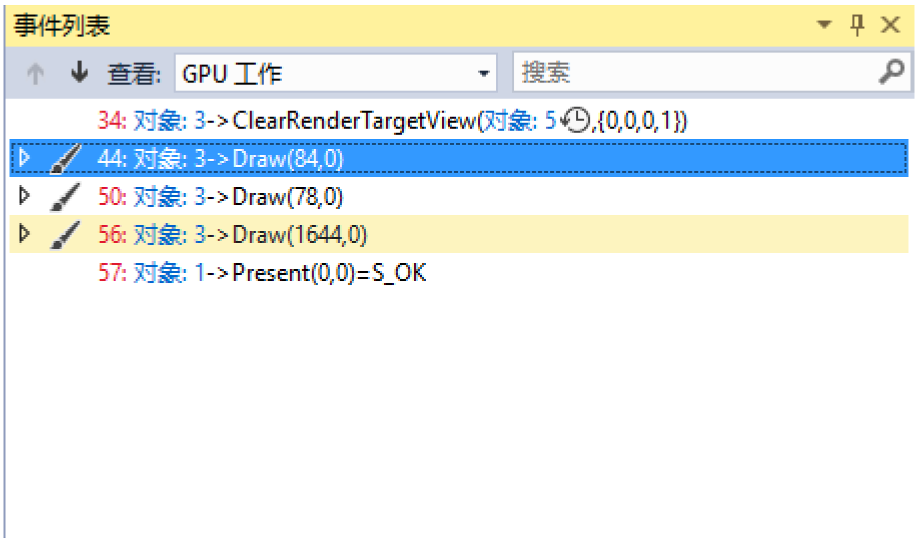


Solid fill: RGB(255,0,0) | Solid stroke: RGB(255,255,0)

Final result:



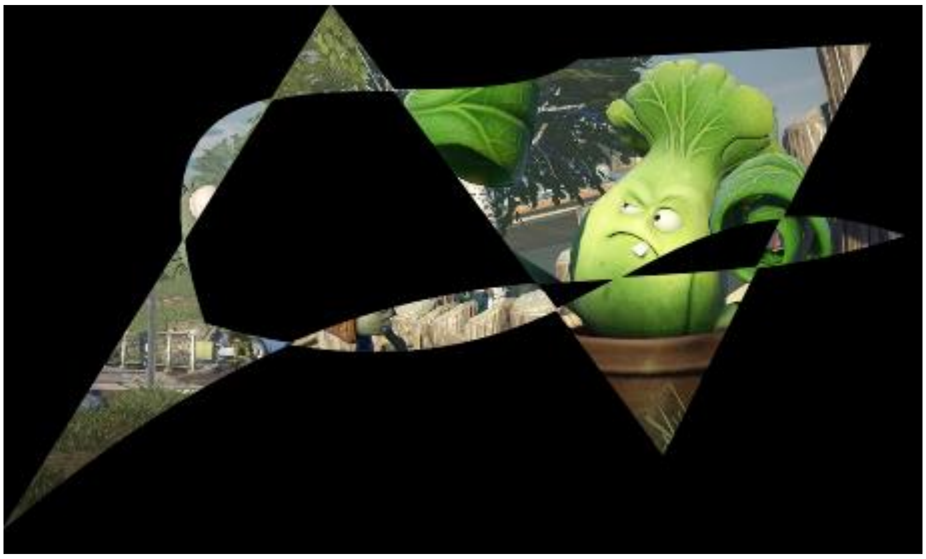
3 Batches: LoopBlinn batch, common batch, stroke & anti-aliasing batch



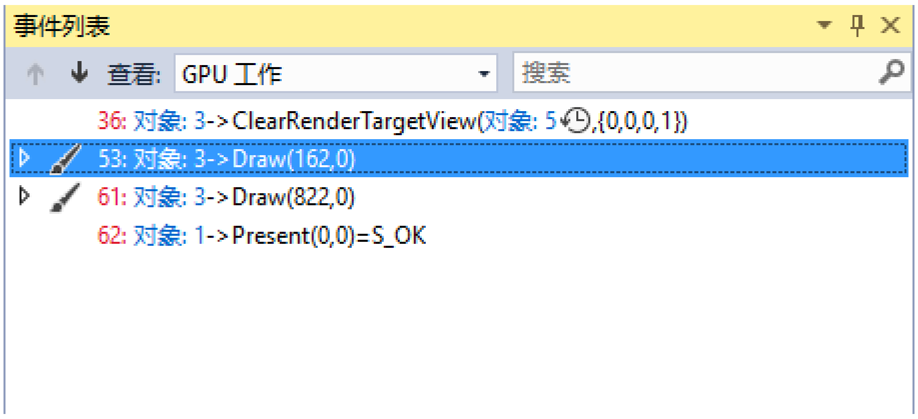
```
34: 对象: 3->ClearRenderTargetView(对象: 5,{0,0,0,1})
44: 对象: 3->Draw(84,0)
50: 对象: 3->Draw(78,0)
56: 对象: 3->Draw(1644,0)
57: 对象: 1->Present(0,0)=S_OK
```

Picture fill | Stroke: none

Final result:



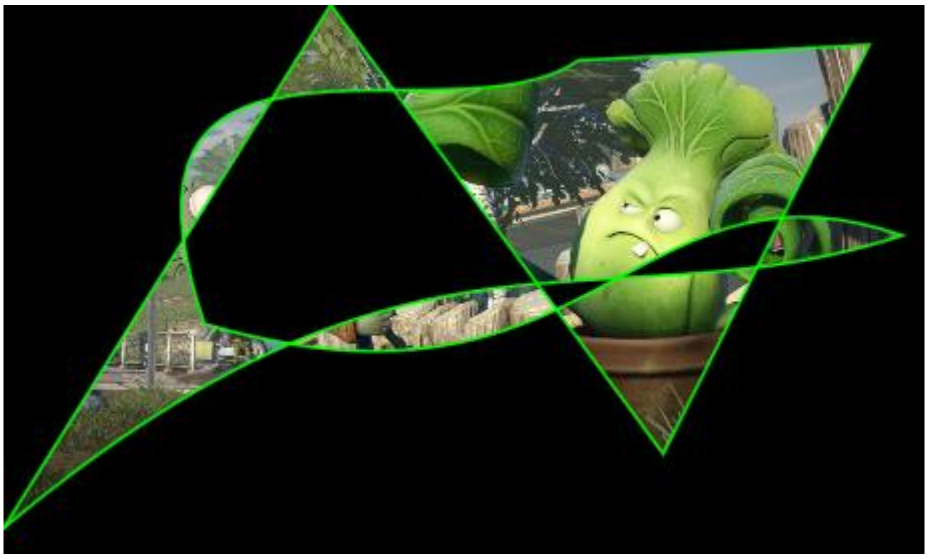
2 Batches: LoopBlinn & common batch(merged for better texture batching), anti-aliasing batch



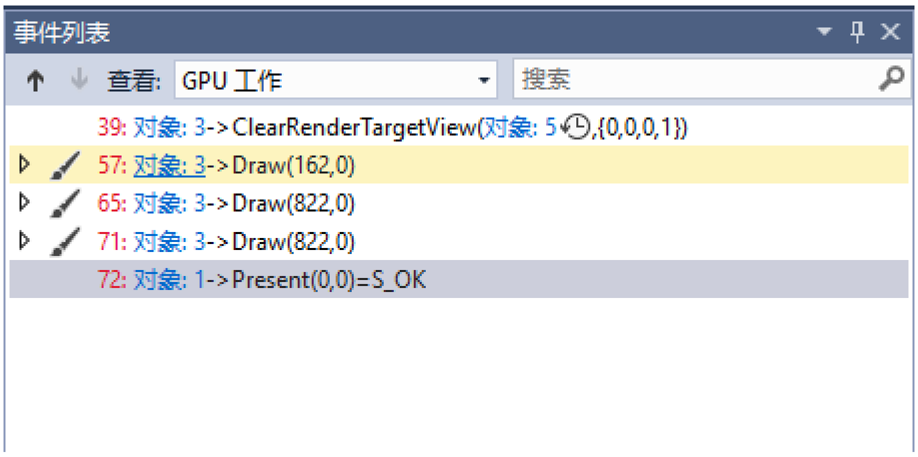
36: 对象: 3->ClearRenderTargetView(对象: 5,{0,0,0,1})
53: 对象: 3->Draw(162,0)
61: 对象: 3->Draw(822,0)
62: 对象: 1->Present(0,0)=S_OK

Picture fill | Solid stroke: RGB(0,255,0)

Final result:



3 Batches: LoopBlinn & common batch, anti-aliasing(half-edge picture stroke) batch, solid stroke batch



39: 对象: 3->ClearRenderTargetView(对象: 5,{0,0,0,1})
57: 对象: 3->Draw(162,0)
65: 对象: 3->Draw(822,0)
71: 对象: 3->Draw(822,0)
72: 对象: 1->Present(0,0)=S_OK