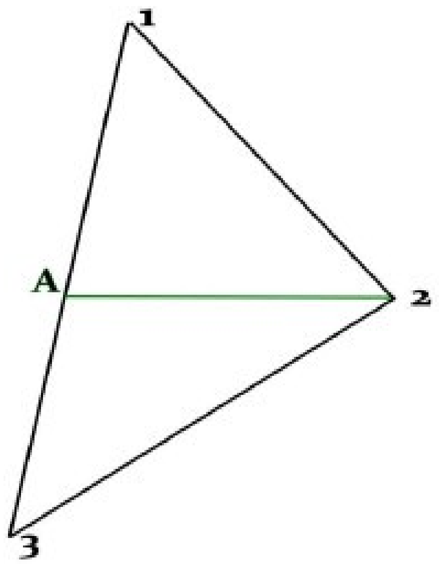
Triangles - Rendering.

1994-2023



Draw X.

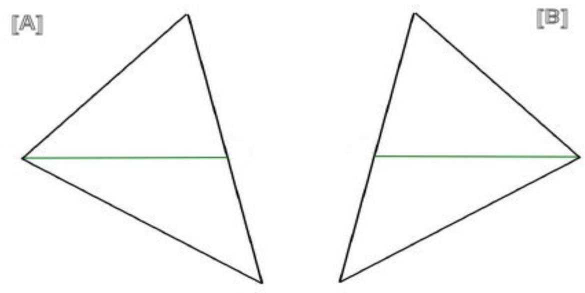
For a component c about triangular form we have the solutions for rasterisations :

$$Dc = \frac{(c2-c1)(y3-y1)-(c3-c1)(y2-y1)}{(x2-x1)(y3-y1)-(x3-x1)(y2-y1)}$$

Dc12=(c2-c1)/(y2-y1)
Dc13=(c3-c1)/(y3-y1)
Dc23=(c3-c2)/(y3-y2)

...and 1/Z is on linear way on a scanline.
I first think about such an artefact in july 1994 with no documentation and it makes my intellectual property, but I got no time to release more than my draft. Thanks to Karl, LCA, Zeb, Gandalf, Kroc, for the purpose.
More informations: Algorithm.

We fill triangle in rasterisation from up to down, with left and right edges calculation.
There is two cases to fill a triangle [A] and [B] :



Draw Y.

Here a gouraud-zbuffer triangle rendering:

```
void triangle(p0,p1,p2)
{

	mini=vertex[p0].Ya; conf=0;

	if (mini>vertex[p1].Ya) { mini=vertex[p1].Ya; conf=1;}

	if (mini>vertex[p2].Ya) { mini=vertex[p2].Ya; conf=2;}

	switch (conf)
	{
	case 0:
		memcpy(&a,&vertex[p0],sizeof(point));
		memcpy(&b,&vertex[p1],sizeof(point));
		memcpy(&c,&vertex[p2],sizeof(point));
		break;
	case 1:
		memcpy(&a,&vertex[p1],sizeof(point));
		memcpy(&b,&vertex[p2],sizeof(point));
		memcpy(&c,&vertex[p0],sizeof(point));
		break;
	case 2:
		memcpy(&a,&vertex[p2],sizeof(point));
		memcpy(&b,&vertex[p0],sizeof(point));
		memcpy(&c,&vertex[p1],sizeof(point));
		break;
	};

	float sd=1.0f/((b.Xa-a.Xa)*(c.Ya-a.Ya) – (c.Xa-a.Xa)*(b.Ya-a.Ya));

	int inccoul=256*((b.coul-a.coul)*(c.Ya-a.Ya) – (c.coul-a.coul)*(b.Ya-a.Ya))*sd;

	int incz=256*((b.Za-a.Za)*(c.Ya-a.Ya) – (c.Za-a.Za)*(b.Ya-a.Ya))*sd;

	if (a.Ya<height)
	{
		if (b.Ya>c.Ya)
		{
			if (b.Ya>0)
			{
				// [ A ]

				calc_seg_incr(&sg1,&a,&b);
				calc_seg_incr(&sg2,&a,&c);
				calc_seg_incr(&sg3,&c,&b);

				g=d=a;
				for (n=0;n<sg2.len;n++)
				{
					fill(&g,&d,inccoul,incz);

					g.Xa+=sg2.incx;
					g.coul+=sg2.inccoul;
					g.Za+=sg2.incz;
					d.Xa+=sg1.incx;
					d.coul+=sg1.inccoul;
					d.Za+=sg1.incz;
					g.Ya++;
					d.Ya++;
				}
				g=c;
				for (n=0;n<sg3.len;n++)
				{
					fill(&g,&d,inccoul,incz);
					g.Xa+=sg3.incx;
					g.coul+=sg3.inccoul;
					g.Za+=sg3.incz;
					d.Xa+=sg1.incx;
					d.coul+=sg1.inccoul;
					d.Za+=sg1.incz;
					g.Ya++;
					d.Ya++;
				}
			}
		}
		else
		{
```

```
                if (c.Ya>0)
                {
                        // [ B ]

                        calc_seg_incr(&sg1,&a,&b);
                        calc_seg_incr(&sg2,&a,&c);
                        calc_seg_incr(&sg3,&b,&c);

                        g=d=a;
                        for (n=0;n<sg1.len;n++)
                        {
                                fill(&g,&d,inccoul,incz);
                                g.Xa+=sg2.incx;
                                g.coul+=sg2.inccoul;
                                g.Za+=sg2.incz;
                                d.Xa+=sg1.incx;
                                d.coul+=sg1.inccoul;
                                d.Za+=sg1.incz;
                                g.Ya++;
                                d.Ya++;
                        }
                        d=b;
                        for (n=0;n<sg3.len;n++)
                        {
                                fill(&g,&d,inccoul,incz);
                                g.Xa+=sg2.incx;
                                g.coul+=sg2.inccoul;
                                g.Za+=sg2.incz;
                                d.Xa+=sg3.incx;
                                d.coul+=sg3.inccoul;
                                d.Za+=sg3.incz;
                                g.Ya++;
                                d.Ya++;
                        }
                }
            }
        }
}
```

Here is the clean view of the proceeding of triangle rendering and countin instructions cycles involves a simple optimisation :

1. The expression "(x2-x1)(y3-y1)-(x3-x1)(y2-y1)" is a cross-product allowing CW and CCW traitment and require 2 MULs et 4 SUBs.
2. The expression "(c2-c1)(y3-y1)-(c3-c1)(y2-y1)" counts the same and allows to proceed the interpolation increment like gouraud shading or mapping coordinates, etc.
3. Evidence give that from the two case [A] et [B], the horizontal position of A (Draw X) defines the orientation too.

So, a simplest proceeding of interpolation increments :

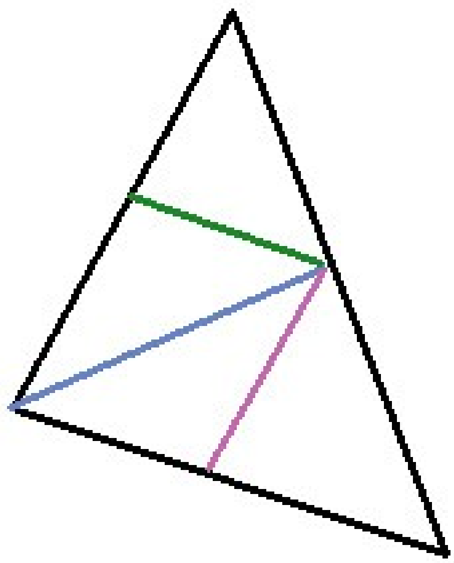
1 is the top point, (reordination cycling included), if vertical position of point 2 is greater than vertical position of point 3, we consider case [A], else case [B].

So the coordinates of point A (Draw X) : A = (1)+(y2-y1)*Dc13

And interpolation increment (uv(2)-uv(A))/(x2-xA) !

The advantage of this form is using precalculation tables for calculating Dc12, Dc13 and Dc23, by range of screen. (K*X/Y)

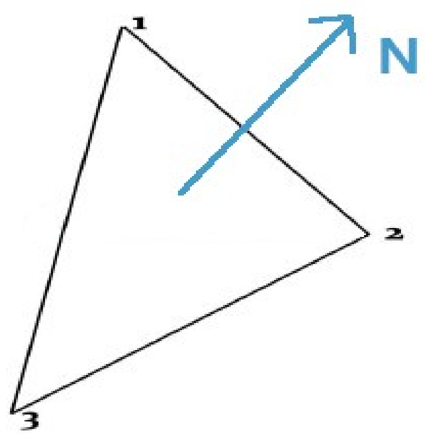
Optimisations for small triangles :



Taking recursively the longest edge, subdividing by 2, and so on, until pixels are close, is faster for rendering a small triangle. Maybe also, perspective corrections are not necessary for this kind of set.

Better perspective mapping :

Projection of the triangle normal on plane, gives the case if triangles must be drawn vertically or horizontally, an avoid segmentation of perspective mapping with better precision, by an angulus of 45°



See my fractal cubes demo to observe the poor perspective mapping applied even on modern cards.
Perspective mapping is lowless by this way by calculating perspective projection only on edges.