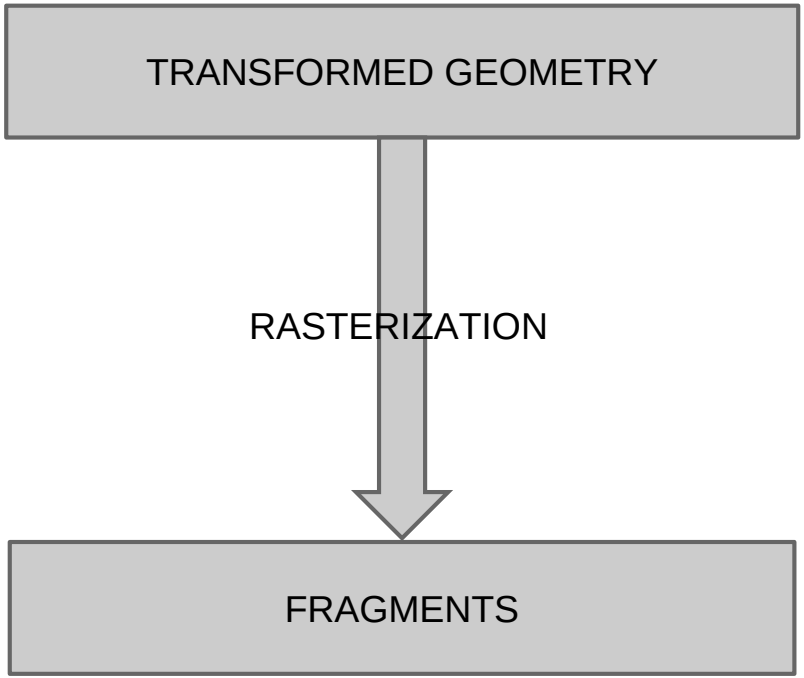
Rasterization

Pipeline



Primitives

- Points
- Line segments
- Triangles
- All other primitives are discretized to one of the above:
  - Curve → Chain of segments
  - Polygons are triangulated
  - Surface → Mesh of triangles

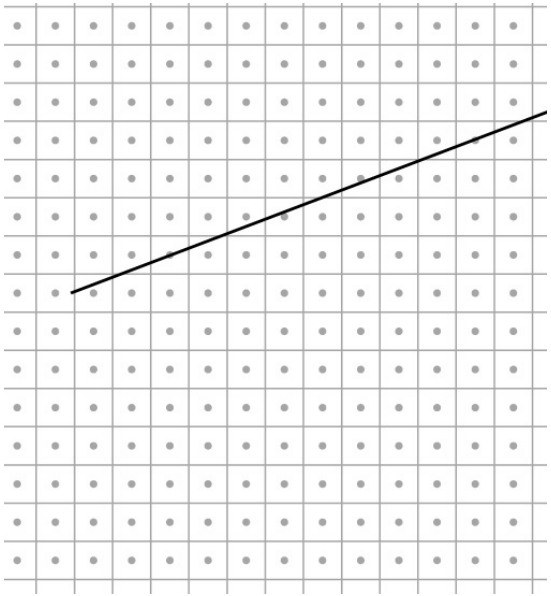
Rasterization

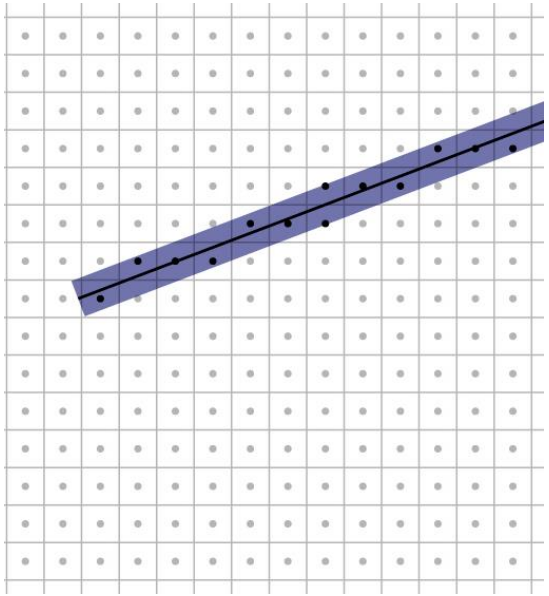
- Decide which pixels are covered by a primitive.
- Compute the value of an attribute at such pixels by interpolation of endpoints attributes.
- Output: a set of fragments, one for each pixel covered by the primitive.

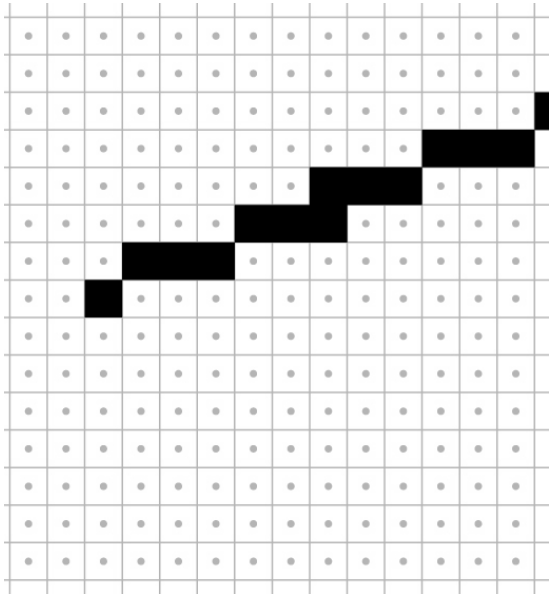
Line drawing

Input: two endpoints in screen coordinates.
Draw reasonable set of pixels approximating the line between these two points.
Example of algorithms: Midpoint algorithm, Bresenham algorithm. Both produce same results.
Use an implicit representation of the line.

Line drawing







Line drawing

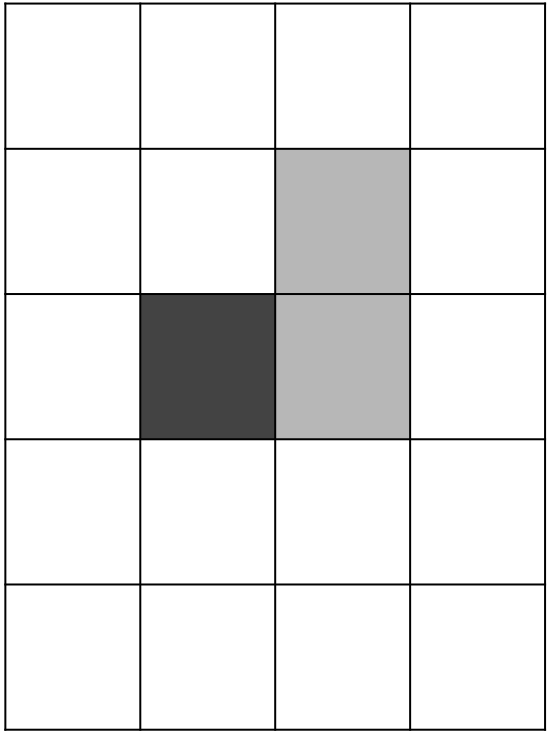
Given endpoints: (x0,y0) and (x1,y1).
Implicit line equation is:
$f(x,y) \equiv (y0-y1)x+(x1-x0)y+x0y1-x1y0=0.$
Assume x0 < x1. If not swap the points.
Assume that the slope of the line is in (0,1].
Other cases are obtained by symmetry.
$slope=(y1-y0)/(x1-x0).$

Line drawing algorithm

Go from left endpoint to right.
When progressing to next pixel, two choices:

- Draw the next pixel at the same height
- Or draw it one higher.

Invariant: in each column, always one pixel.



Basic algorithm

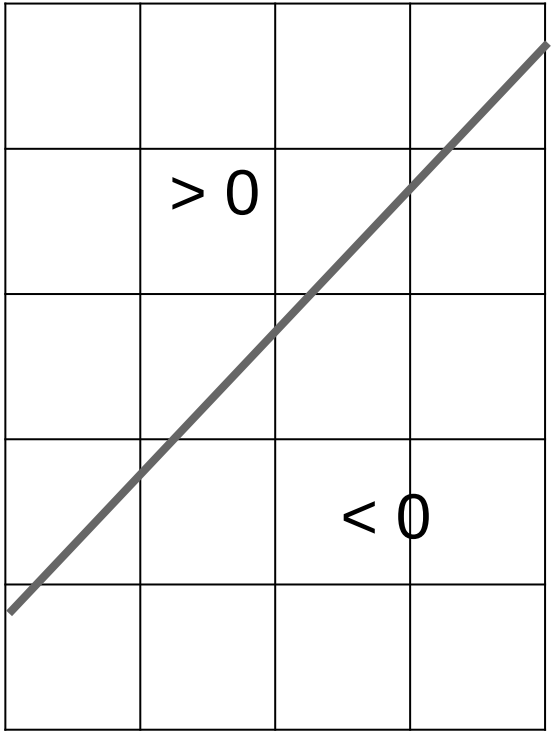
```
y = y0;
for x = x0 to x1 {
  draw_pixel(x,y);
  if <condition> {
    y = y + 1;
  }
}
```

Condition

Decide between: `(x+1,y)` and `(x+1,y+1)`.
Look at the midpoint: `(x+1,y+0.5)`.
If the line passes below the midpoint, select pixel `(x+1,y)`.
Otherwise, select pixel `(x+1,y+1)`.
Position of midpoint wrt line is obtained by the sign of:
`f(x+1,y+0.5)`.
`<condition> := (f(x+1,y+0.5) < 0)` which corresponds to the midpoint being under the line.

Condition

Implicit line equation:
$f(x,y)\equiv(y0-y1)x+(x1-x0)y+x0y1-x1y0=0$.
Since $(x1-x0)>0$ the term $(x1-x0)y$ increases as y increases. So the part above the line is positive.

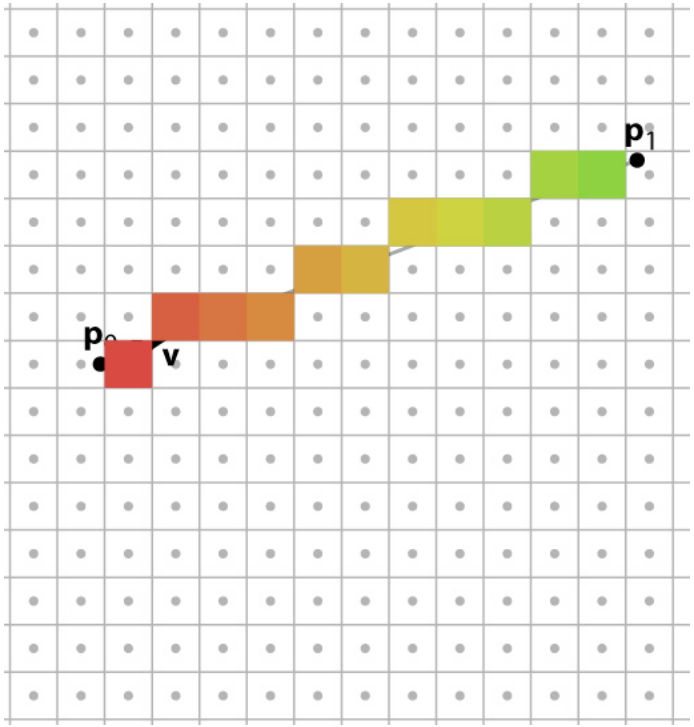


Incremental algorithm

```
y = y0;
d = f(x0+1,y0+0.5);
for x = x0 to x1 {
 draw(x,y);
 if d < 0 {
  y = y+1;
  d = d + (x1-x0) + (y0-y1);
 } else {
  d = d + (y0-y1);
 }
}
```

Interpolation

Given attributes (colors) at endpoints, compute the attribute at each pixel by interpolation:
`a=(1-w)a0 + w a1`
where `a0` is the attribute at `p0`,
`a1` the attribute at `p1`.
And `w` is approximated by:
`(p-p0).v/L`.
`v`: unit direction of the line
`L`: length of the line
`.` is the scalar (inner) product



Triangle rasterization

Given 2D triangle, defined by vertices:
`p0=(x0,y0), p1=(x1,y1), p2=(x2,y2).`
`p0,p1,p2` in screen coordinates (after transformation, clipping and projection).
Draw this 2D triangle.
Also interpolate color or other attribute from its vertices.

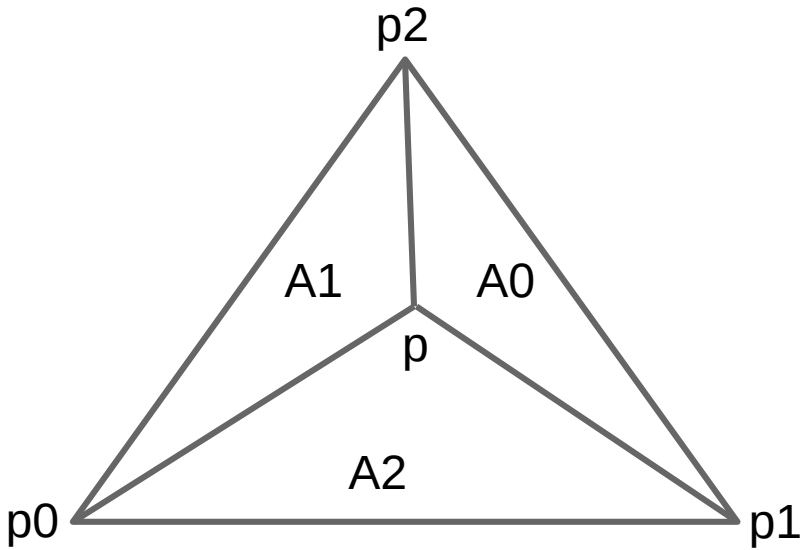
Gouraud interpolation

Given vertex colors: `c0`, `c1`, `c2`; the color `c` at a point in a triangle `p` with barycentric coordinates `(u, v, w)` is computed by interpolation:

```
c = u c0 + v c1 + w c2.
```

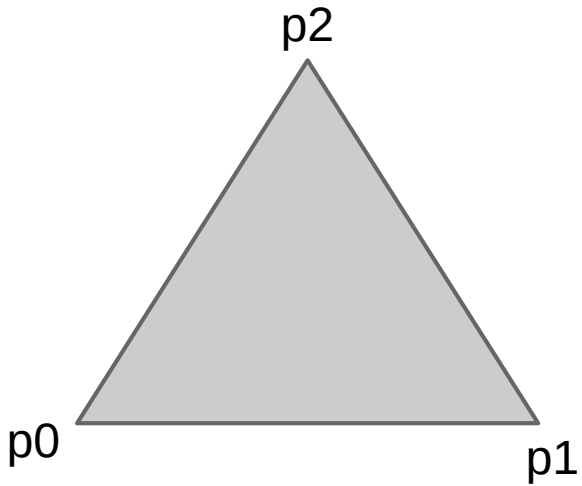
Barycentric coordinates

`u=A0/A;v=A1/A;w=A2/A`
where: `A` is the signed area of the outer triangle,
and `A0`, `A1`, `A2` are the signed areas of the inner triangles.
p is in the triangle iff u, v, and w in [0,1].



Triangle area

Signed area of triangle is:
$S = 1/2(p_1 - p_0) \times (p_2 - p_0)$
with $\times$ the cross-product.
`S>0` if p_0, p_1, p_2 are in counterclockwise order.
`S<0` otherwise.



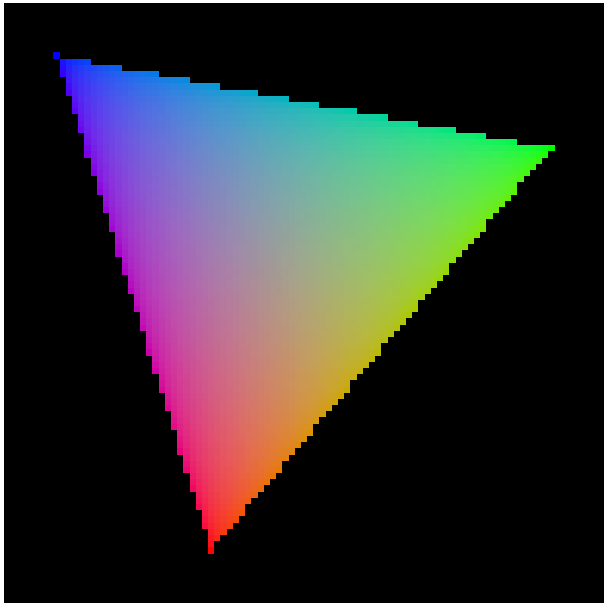
Brute force algorithm

```
for x=0 to width {
 for y=0 to height {
  compute barycentric coordinates (u,v,w) for (x,y);
  if (u in [0,1] and v in [0,1] and w in [0,1]) {
    /* pixel (x,y) is in the triangle */
    c = u c0 + v c1 + w c2;
    set fragment (x,y) with color c;
  }
 }
}
```

Bounding rectangle optimization

```
xmin = min(x0,x1,x2);ymin = min(y0,y1,y2);
xmax = max(x0,x1,x2);ymax = max(y0,y1,y2);
for x=xmin to xmax {
 for y=ymin to ymax {
  compute barycentric coordinates (u,v,w) for (x,y);
  if (u in [0,1] and v in [0,1] and w in [0,1]) {
    c = u c0 + v c1 + w c2;
    set fragment (x,y) with color c;
  }
 }
}
```

Triangle drawing



Issues and possible improvements

- Integer computations
- Integer overflow
- Sub-pixel precision
- Fill-rules
- Speed

Integer computations

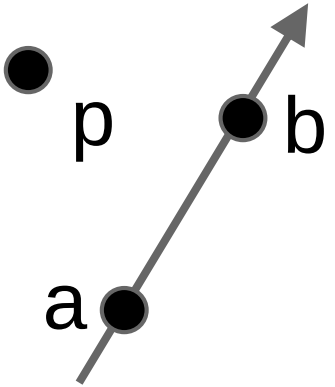
No need to check for `u,v,w` in `[0,1]`.
Sufficient to check that `u,v,w>0` to decide if a point is in the triangle.
Therefore: no need to compute normalized barycentric coordinates. The sign of the (signed) area of each inner triangle is sufficient. It corresponds to an orientation test.
Assuming colors are encoded as integers, all computations can be done with integers.

Orientation test

a,b,p is in counterclockwise order if orient2D(a,b,p)>0.
They are in clockwise order if orient2D(a,b,p)<0.
They are collinear if orient2D(a,b,p) = 0.

```
struct Point2D_ { int x, y; };
typedef struct Point2D_ Point2D;

int orient2D(Point2D a, Point2D b, Point2D p)
{
  return (b.x-a.x)*(p.y-a.y)-(b.y-a.y)*(p.x-a.x);
}
```



Triangle drawing algorithm

```
xmin = min(x0,x1,x2);ymin = min(y0,y1,y2);
xmax = max(x0,x1,x2);ymax = max(y0,y1,y2);
for x=xmin to xmax {
 for y=ymin to ymax {
  w0=orient2D(p1,p2,p); w1=orient2D(p2,p0,p);
  w2=orient2D(p0,p1,p);
  if (w0>=0 and w1>=0 and w2>=0) {
    c = (w0 c0 + w1 c1 + w2 c2)/(w0+w1+w2);
    set fragment (x,y) with color c;
  }
 }
}
```

Fill-rules

Set of tie-breaking rules to guarantee that when two non-overlapping triangles share an edge, every pixel covered by the two triangles is set only once.
Essentially: dealing with pixels on triangle edges.
Different rules exist. Any rule is fine as long as it is consistently applied.

Examples of fill-rules

Approach 1: select an offscreen point and draw the edge as part of the triangle lying on the same side as the offscreen point.

Approach 2: draw a pixel falling on an edge if the edge is a top or left edge.

Top edge: In a counterclockwise triangle, a top edge is an horizontal edge going toward the left (end point is left to start point).

Left edge: in a counterclockwise triangle, a left edge is an edge that goes down (end point is strictly below start point).

Speed improvement

- Incremental approach: similar to the line drawing approach, go to the next step by adding a fixed increment rather than recomputing the barycentric coordinates each time
- Process multiple pixels at once. Either with dedicated hardware or with SIMD instructions (SSE2) in software.

Minimal 3D pipeline

The rasterizer described so far assume vertices to be in screen space.
In 3D, transformation from world space to screen space is obtained by applying: the product of the modeling, camera, projection and viewport matrices, to: incoming vertices.
Screen space lines and triangles are then drawn using the previous algorithms.
Occlusion problem: nearer objects should be drawn over further objects.

Depth buffer

Depth buffer also called z-buffer.
Requires an additional buffer same size as the color buffer (same size as screen).
This buffer stores the depth of the pixel currently in the color buffer.
Z-buffer initialized with the maximum depth.
If a new fragment’s depth is closer, both its color and depth overwrite the current values in the color and depth buffers.

Depth buffer

```
if (w0>=0 and w1>=0 and w2>=0) {
    z = (w0 z0 + w1 z1 + w2 z2)/(w0+w1+w2);
    if (z < depth(x,y)) {
      c = (w0 c0 + w1 c1 + w2 c2)/(w0+w1+w2);
      set fragment (x,y) with color c;
      set entry (x,y) in depth-buffer with z;
    }
}
```

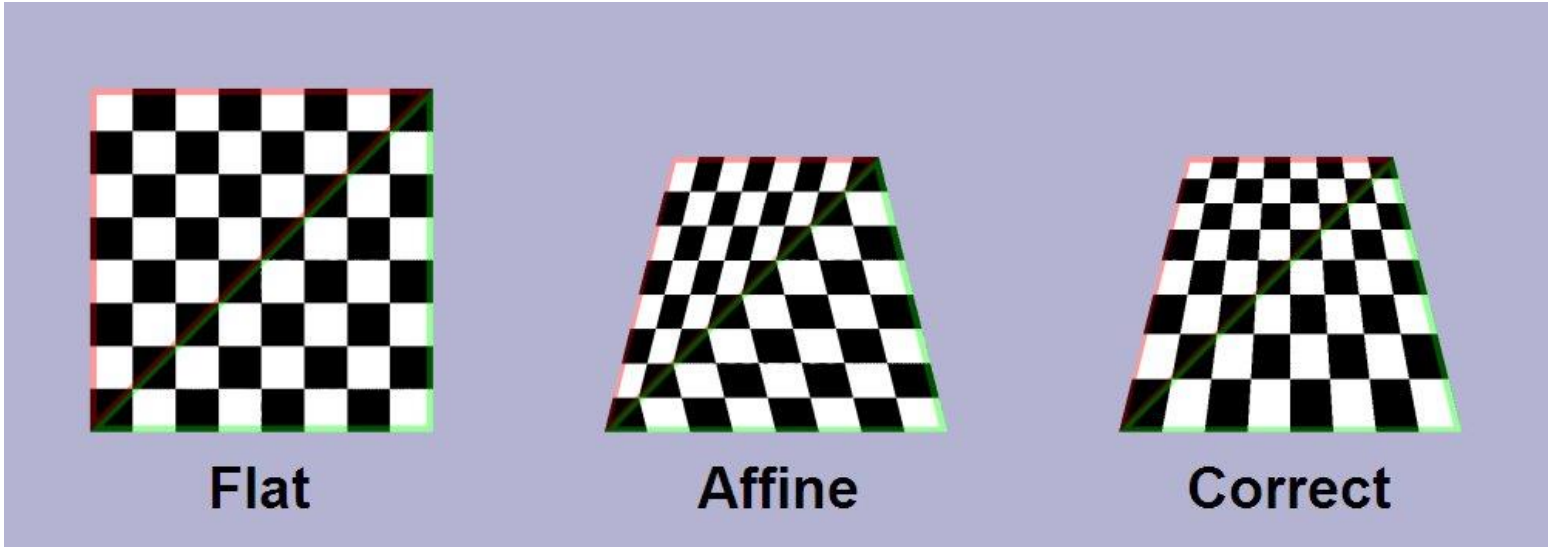
Perspective-corrected barycentric coordinates

We can modify our rasterizer for texture mapping by interpolating texture coordinates:

```
for all x
 for all y
  compute barycentric coordinates (a0,a1,a2) for (x,y);
  if (a0 in [0,1] and a1 in [0,1] and a2 in [0,1]) {
   // … compute u and v by interpolation
   set fragment (x,y) with texture (u,v);
  }
```

Perspective-corrected barycentric coordinates

Unfortunately this approach does not work when a perspective projection is used.
Compare middle image (this approach) to the right image (perspective correct result).



Perspective-corrected barycentric coordinates

The problem appears because the barycentric coordinates in screen space are *non-linearly* distorted by the homogeneous division (in the case of perspective projection).
Instead, we would like to interpolate in world space.
Fortunately, barycentric coordinates can be corrected to account for the perspective projection.

Perspective-corrected barycentric coordinates

Given vertices $t_i = (x_i, y_i, z_i)$, where $i = 0,1,2$, after modeling, camera and projection transformation but before homogenization.
In the case of perspective projection, $h_i = z_i/n$, where `n` is the distance to the near plane.
Compute the barycentric coordinates in screen space (as before). Then apply the perspective correction.

Perspective-corrected barycentric coordinates

```
compute bounds for xi/hi and yi/hi
for x=xmin to xmax {
 for y=ymin to ymax {
  compute barycentric coordinates (w0,w1,w2) for (x,y);
  if (w0>=0 and w1>=0 and w2>=0) {
    d=h1*h2 + h2*w1*(h0-h1) + h1*w2*(h0-h2);
    w1c=h0*h2*w1/d;
    w2c=h0*h1*w2/d;
    w0c=1-w1c-w2c;
    u=w0c*u0+w1c*u1+w2c*u2;
    v=w0c*v0+w1c*v1+w2c*v2;
    set pixel (x,y) with texture (u,v);
  }
 }
}
```