Texture mapping like it's 1995[1]

 **in Python[2]** 

1. Pixel color computation done on CPU; screen is just a big array of bytes

2. Using numpy (kinda cheating, I know)

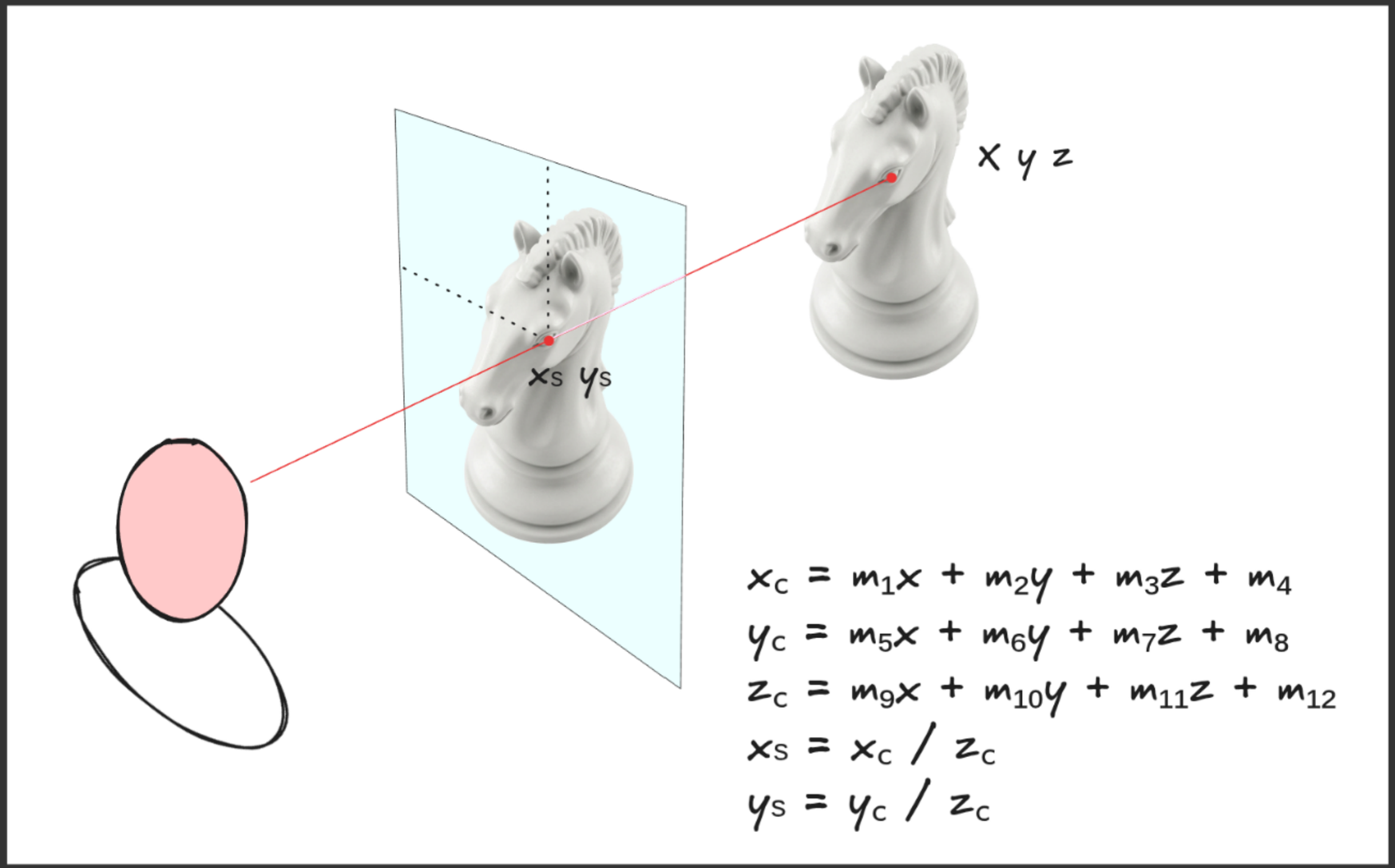
Who am I

- My name is Andrea "6502" Griffini
- I've been professionally programming for 40+ years in many different fields
- 3d graphics is one of my hobbies
- Back in 1995 it was my job

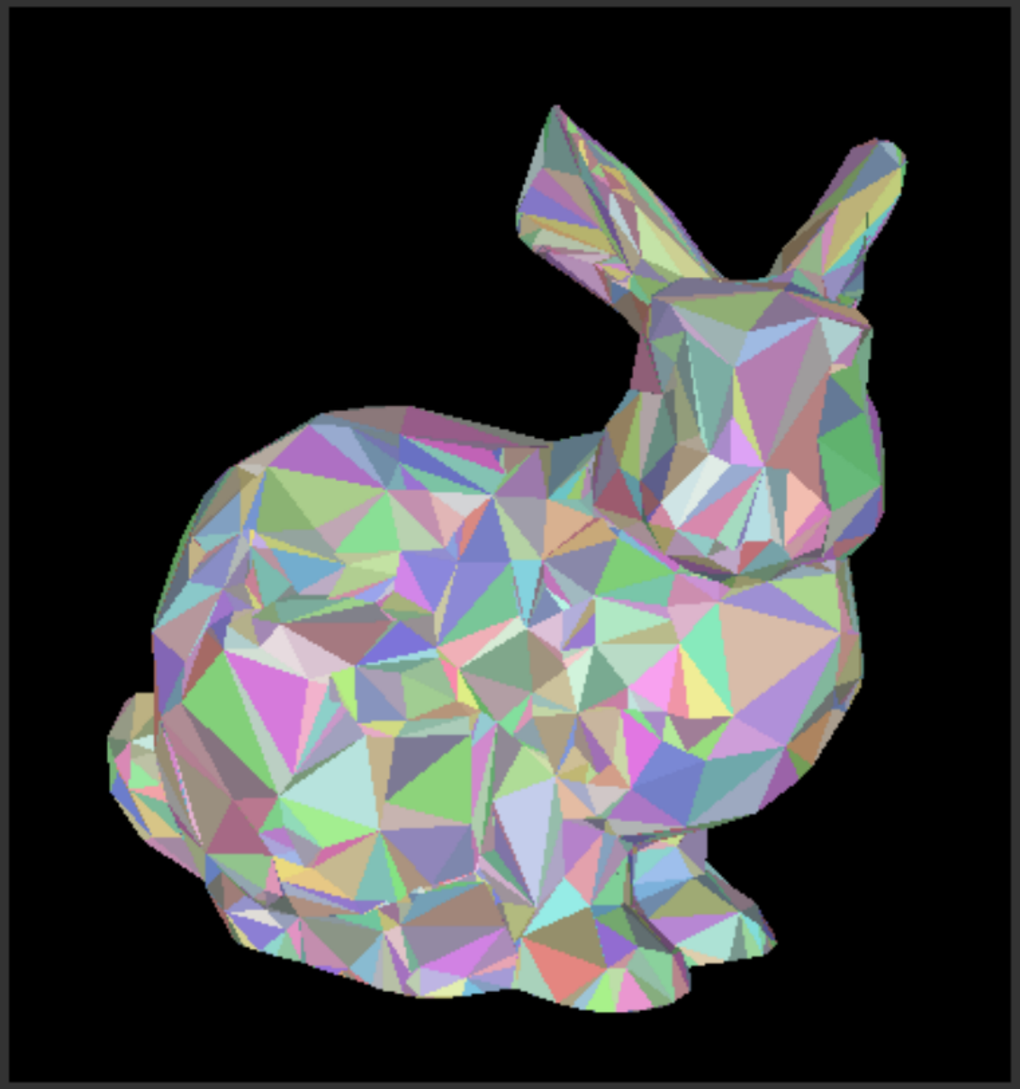
How it worked back in 1995

- No GPU
- Every pixel on the screen was computed by the CPU
- Single core... only partial parallelism on UV pipelines (required asm coding and the "new" Pentium processor)
- Many tricks were used

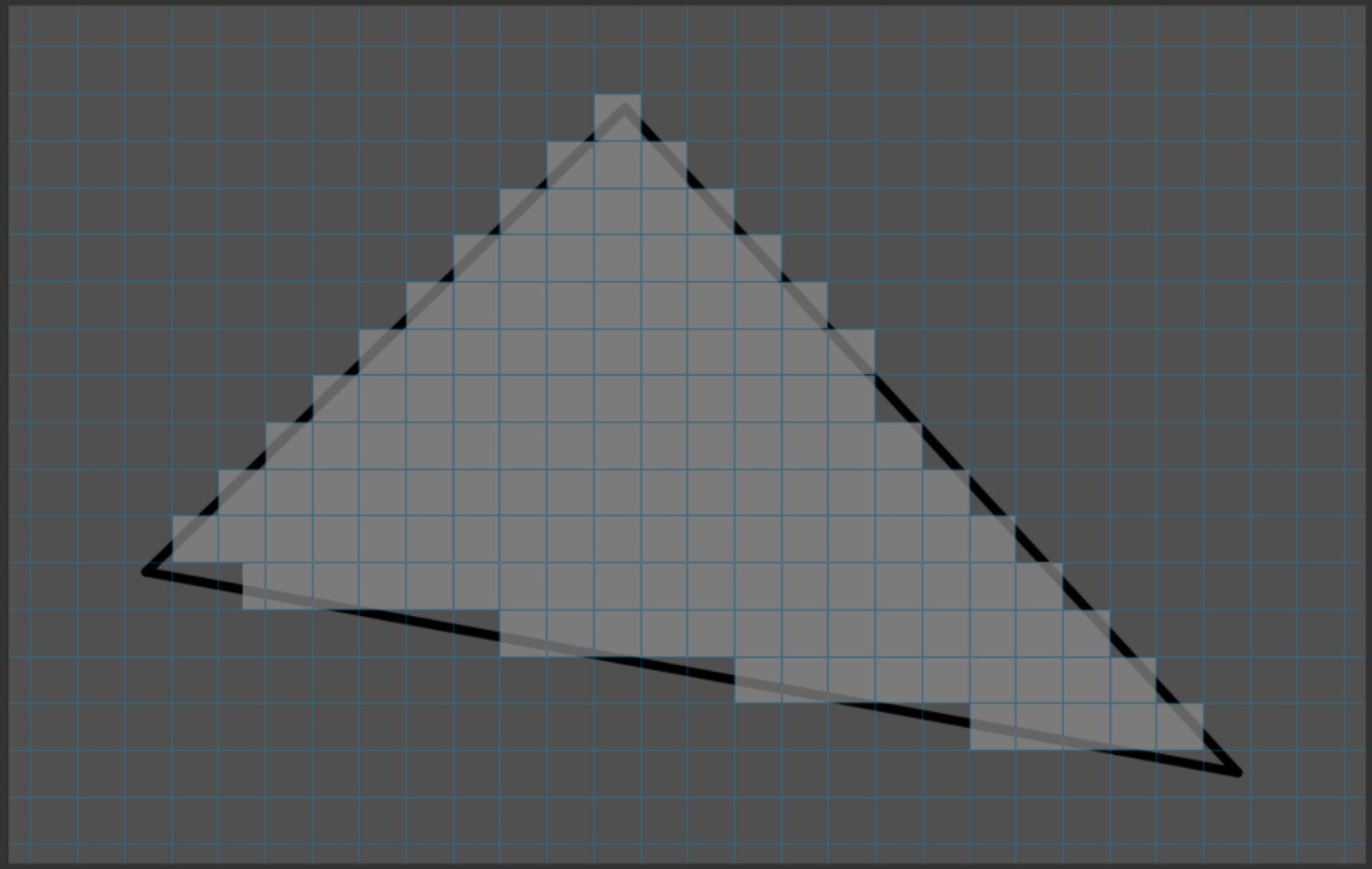
3d → 2d camera mapping



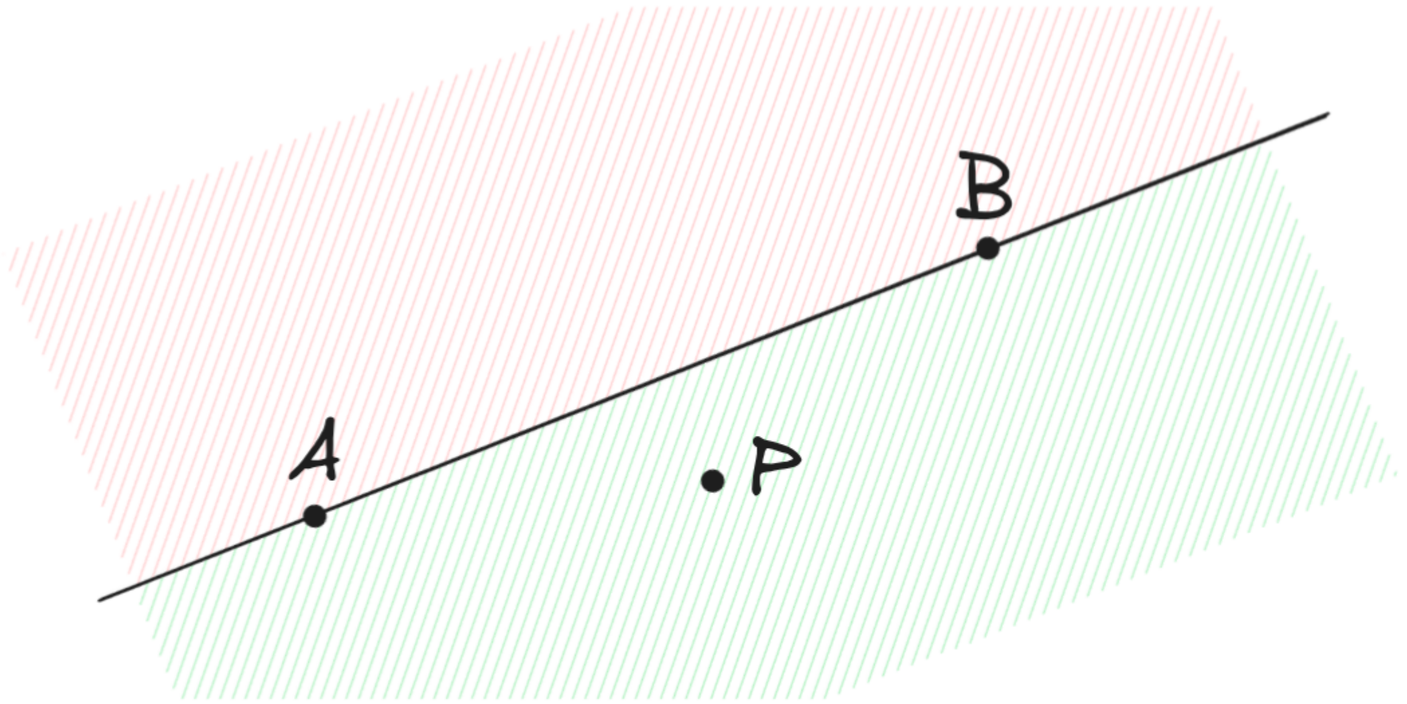
Triangulation



Triangle rasterization



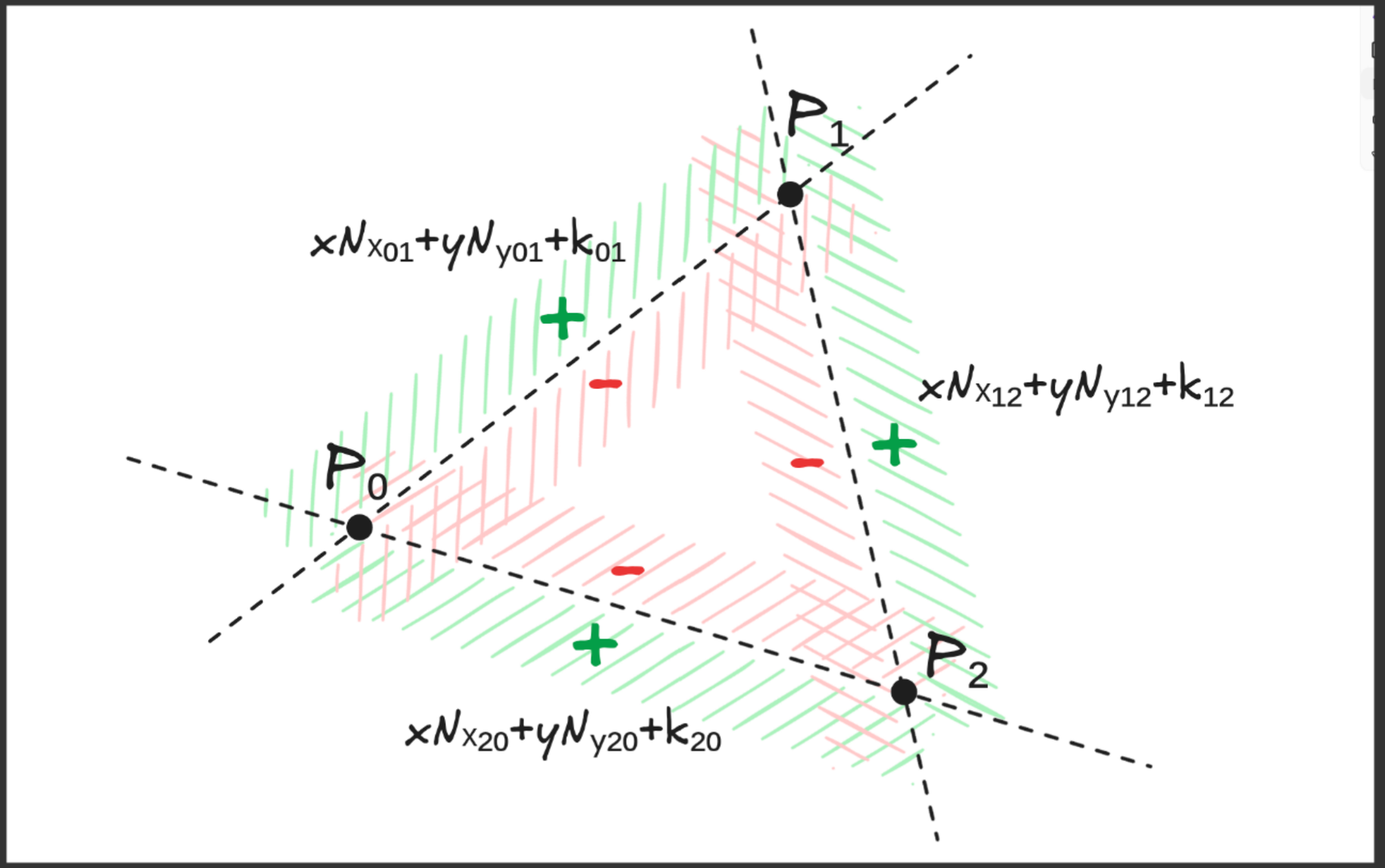
Semiplane equation



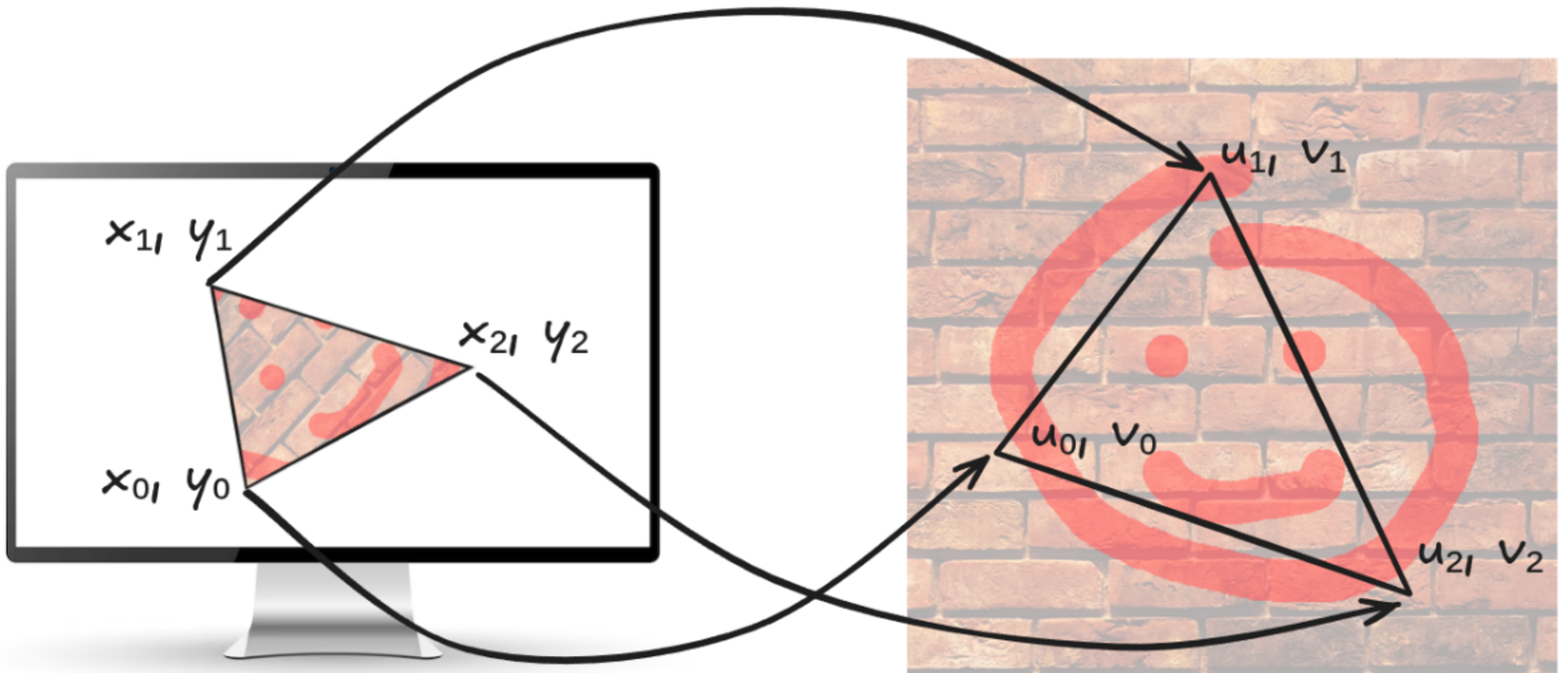
$$(P_x - A_x)(B_y - A_y) - (P_y - A_y)(B_x - A_x) > 0$$

$$N_x P_x + N_y P_y + K > 0$$

Triangle pixels selection



Linear texture mapping



screen space
x y

texture space
u v

$$u = xu_x + yu_y + u_k$$
$$v = xv_x + yv_y + v_k$$

Triangle drawing algorithm

- Project triangle to screen coordinates
- Compute N_x, N_y and k coefficients for each triangle side ($p_0 \rightarrow p_1$, $p_1 \rightarrow p_2$, $p_2 \rightarrow p_0$)
- Compute u_x, u_y, u_k, v_x, v_y, v_k for the triangle
- Find all pixels inside the triangle
- Draw the texture mapped triangle mapping from **(x, y)** to **(u, v)** to find the texel to use for each pixel

NumPy to the rescue

1. Can compute the same formula **"in parallel"** on all elements of arrays at the same time
2. Loops are done in C (**~100** times faster than Python loops)
3. Allows indirect access using an array and an array of indices

u_x, u_y, u_k computation

$$\begin{cases} u_x x_0 + u_y y_0 + u_k = u_0 & \text{I} \\ u_x x_1 + u_y y_1 + u_k = u_1 & \text{II} \\ u_x x_2 + u_y y_2 + u_k = u_2 & \text{III} \end{cases}$$

$$\begin{cases} u_x(x_1 - x_0) + u_y(y_1 - y_0) = u_1 - u_0 & \text{II} - \text{I} \\ u_x(x_2 - x_0) + u_y(y_2 - y_0) = u_2 - u_0 & \text{III} - \text{I} \end{cases}$$

$$\Delta = det \begin{pmatrix} x_1 - x_0 & y_1 - y_0 \\ x_2 - x_0 & y_2 - y_0 \end{pmatrix} = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

$$u_x = \frac{det \begin{pmatrix} u_1 - u_0 & y_1 - y_0 \\ u_2 - u_0 & y_2 - y_0 \end{pmatrix}}{\Delta} \qquad u_y = \frac{det \begin{pmatrix} x_1 - x_0 & u_1 - u_0 \\ x_2 - x_0 & u_2 - u_0 \end{pmatrix}}{\Delta} \qquad u_k = u_0 - u_x x_0 - u_y y_0$$

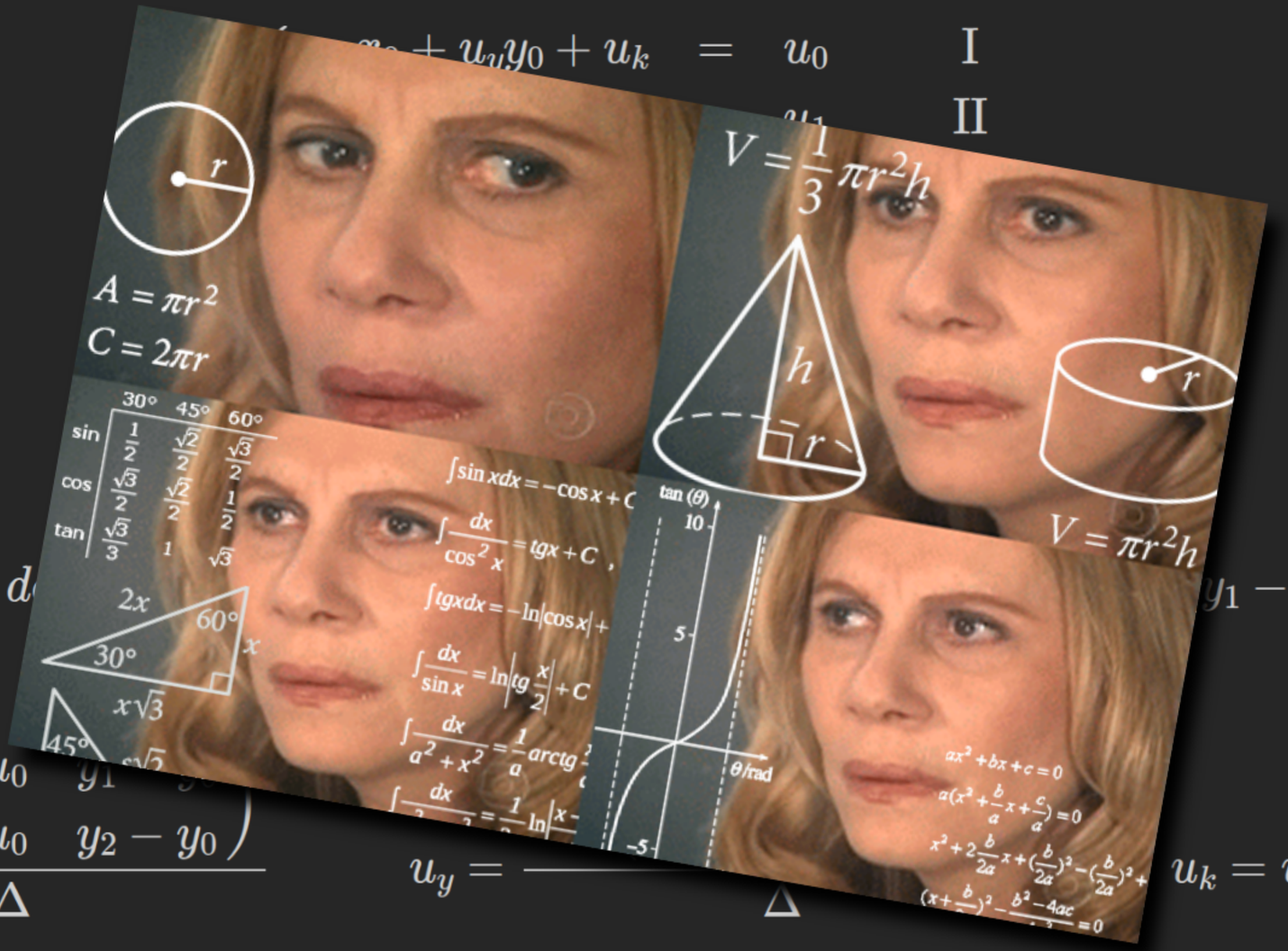
u_x, u_y, u_k computation

$$u_0 + u_y y_0 + u_k = u_0 \quad \text{I}$$

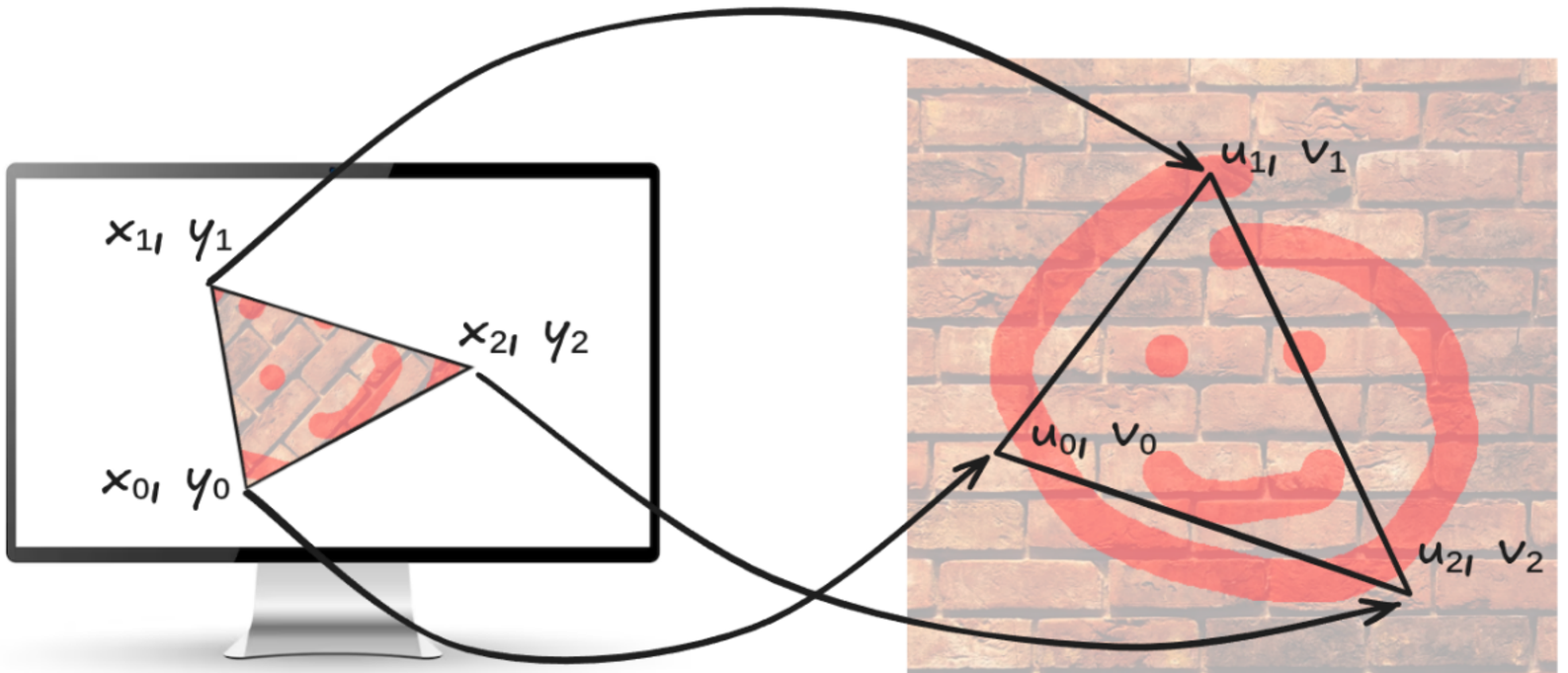
$$\text{II}$$

$$\Delta = d \qquad y_1 - y_0)$$

$$u_x = \frac{det \begin{pmatrix} u_1 - u_0 & y_1 \\ u_2 - u_0 & y_2 - y_0 \end{pmatrix}}{\Delta} \qquad u_y = \qquad \Delta \qquad u_k = u_0 - u_x x_0 - u_y y_0$$



Linear texture mapping



screen space
x y

texture space
u v

$$u = xu_x + yu_y + u_k$$
$$v = xv_x + yv_y + v_k$$

u_x, u_y, u_k computation

$$\begin{cases} u_x x_0 + u_y y_0 + u_k = u_0 & \text{I} \\ u_x x_1 + u_y y_1 + u_k = u_1 & \text{II} \\ u_x x_2 + u_y y_2 + u_k = u_2 & \text{III} \end{cases}$$

$$\begin{cases} u_x(x_1 - x_0) + u_y(y_1 - y_0) = u_1 - u_0 & \text{II} - \text{I} \\ u_x(x_2 - x_0) + u_y(y_2 - y_0) = u_2 - u_0 & \text{III} - \text{I} \end{cases}$$

$$\Delta = det \begin{pmatrix} x_1 - x_0 & y_1 - y_0 \\ x_2 - x_0 & y_2 - y_0 \end{pmatrix} = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

$$u_x = \frac{det \begin{pmatrix} u_1 - u_0 & y_1 - y_0 \\ u_2 - u_0 & y_2 - y_0 \end{pmatrix}}{\Delta} \qquad u_y = \frac{det \begin{pmatrix} x_1 - x_0 & u_1 - u_0 \\ x_2 - x_0 & u_2 - u_0 \end{pmatrix}}{\Delta} \qquad u_k = u_0 - u_x x_0 - u_y y_0$$

v_x, v_y, v_k computation

- Identical... but with **v** instead of **u**
- Also **Δ** is the same as for **u**
  (**Δ** depends only on **x**/**y**, not **u** or **v**)
- **Δ=0** means you've a bad triangle
  (vertices are aligned)

Coding time

```
def triangle(x0, y0, u0, v0,
             x1, y1, u1, v1,
             x2, y2, u2, v2,
             L, tx):
    delta = (x1-x0)*(y2-y0) - (x2-x0)*(y1-y0)
    if delta == 0: return
    ux = ((u1-u0)*(y2-y0) - (u2-u0)*(y1-y0)) / delta
    uy = ((x1-x0)*(u2-u0) - (x2-x0)*(u1-u0)) / delta
    uk = u0 - x0*ux - y0*uy
    vx = ((v1-v0)*(y2-y0) - (v2-v0)*(y1-y0)) / delta
    vy = ((x1-x0)*(v2-v0) - (x2-x0)*(v1-v0)) / delta
    vk = v0 - x0*vx - y0*vy

    nx01, ny01 = y1-y0, x0-x1
    nx12, ny12 = y2-y1, x1-x2
    nx20, ny20 = y0-y2, x2-x0
    k01 = -(x0*nx01 + y0*ny01)
    k12 = -(x1*nx12 + y1*ny12)
    k20 = -(x2*nx20 + y2*ny20)

    th, tw = tx.shape[:2]
    xa = max(0, int(min(x0, x1, x2)))
    ya = max(0, int(min(y0, y1, y2)))
    xb = min(w, int(max(x0, x1, x2))+1)
    yb = min(h, int(max(y0, y1, y2))+1)
    Xb = X[ya:yb, xa:xb]
    Yb = Y[ya:yb, xa:xb]
    aab = aa[ya:yb, xa:xb]
    mask = (np.uint8(Xb*nx01 + Yb*ny01 + k01 <= 0) &
            np.uint8(Xb*nx12 + Yb*ny12 + k12 <= 0) &
            np.uint8(Xb*nx20 + Yb*ny20 + k20 <= 0))
    xya = aab[mask == 1]
    xx = X.ravel()[xya]
    yy = Y.ravel()[xya]
    uv = np.int32(xx*vx + yy*vy + vk) & (th-1)
    uv *= tw
    uv += np.int32(xx*ux + yy*uy + uk) & (tw-1)
    img.reshape((w*h, 3))[xya] = np.int32(tx.reshape((tw*th,3))[uv] * L)
```

Coding time

```
def triangle(x0, y0, u0, v0,
             x1, y1, u1, v1,
             x2, y2, u2, v2,
             L, tx):
```

Code: xy → uv mapping coefficients

```
delta = (x1-x0)*(y2-y0) - (x2-x0)*(y1-y0)
if delta == 0: return

ux = ((u1-u0)*(y2-y0) - (u2-u0)*(y1-y0)) / delta
uy = ((x1-x0)*(u2-u0) - (x2-x0)*(u1-u0)) / delta
uk = u0 - x0*ux - y0*uy

vx = ((v1-v0)*(y2-y0) - (v2-v0)*(y1-y0)) / delta
vy = ((x1-x0)*(v2-v0) - (x2-x0)*(v1-v0)) / delta
vk = v0 - x0*vx - y0*vy
```

Code: sides semiplane equations

```
nx01, ny01 = y1-y0, x0-x1
nx12, ny12 = y2-y1, x1-x2
nx20, ny20 = y0-y2, x2-x0
k01 = -(x0*nx01 + y0*ny01)
k12 = -(x1*nx12 + y1*ny12)
k20 = -(x2*nx20 + y2*ny20)
```

Code: triangle pixels bounding box

```
xa = max(0, int(min(x0, x1, x2)))
ya = max(0, int(min(y0, y1, y2)))
xb = min(w, int(max(x0, x1, x2))+1)
yb = min(h, int(max(y0, y1, y2))+1)
```

Code: numpy arrays for X, Y and pixel address

```
Xb = X[ya:yb, xa:xb]
Yb = Y[ya:yb, xa:xb]
aab = aa[ya:yb, xa:xb]
```

Code: find all pixels inside triangle

```
mask = (np.uint8(Xb*nx01 + Yb*ny01 + k01 <= 0) &
        np.uint8(Xb*nx12 + Yb*ny12 + k12 <= 0) &
        np.uint8(Xb*nx20 + Yb*ny20 + k20 <= 0))
```

Code: find their x/y/address

```
xya = aab[mask == 1]
xx = X.ravel()[xya]
yy = Y.ravel()[xya]
```

Code: xy → uv & texels address

```
uv = np.int32(xx*vx + yy*vy + vk) & (th-1)
uv *= tw
uv += np.int32(xx*ux + yy*uy + uk) & (tw-1)
```

Code: draw triangle pixels

```
img.reshape((w*h, 3))[xya] = (
    np.int32(tx.reshape((tw*th,3))[uv] * L)
)
```

Demo time

Questions?

Andrea "6502" Griffini

https://andreagriffini.com

@agriffini

@AndreaGriffini

pyit26@andreagriffini.com