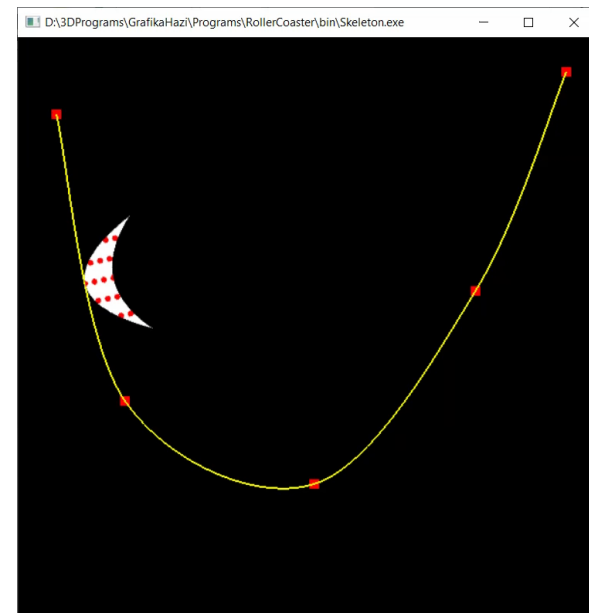


*"Photographers don't take pictures.
They create images."*

Mark Denman

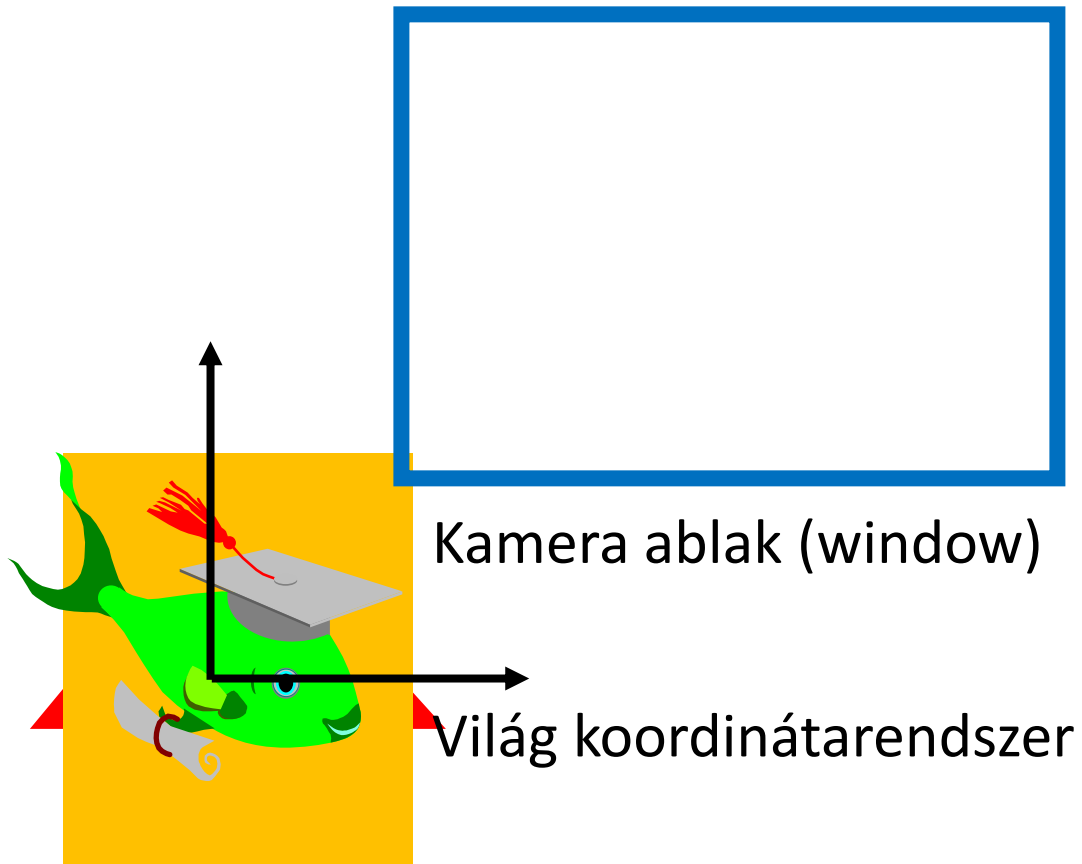
2D képszintézis

Szirmay-Kalos László

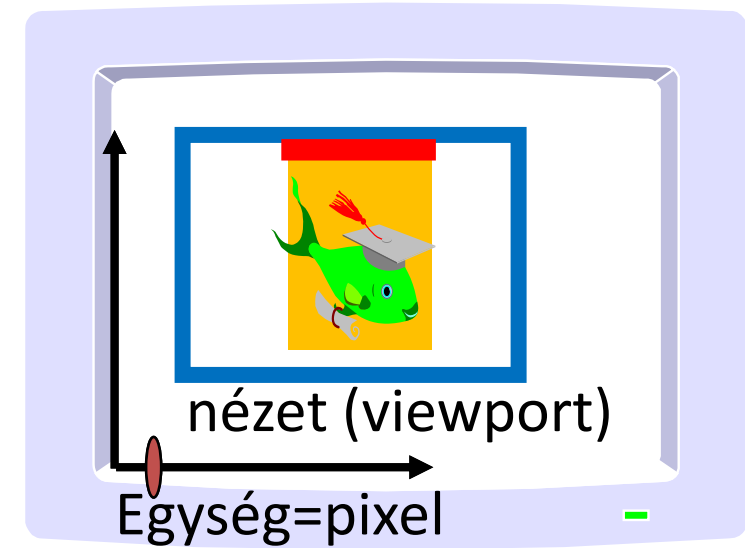


2D képszintézis

Modell

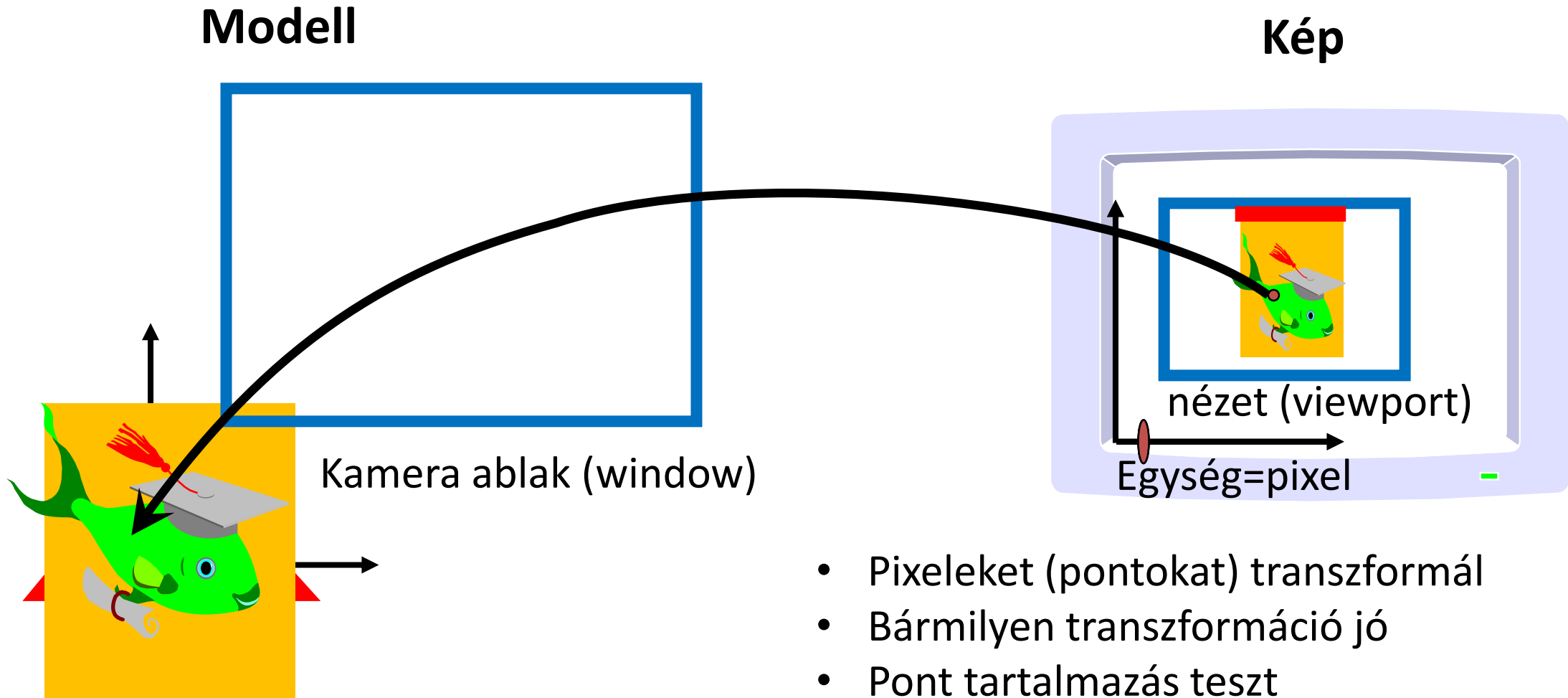


Kép



Saját színnel rajzolás

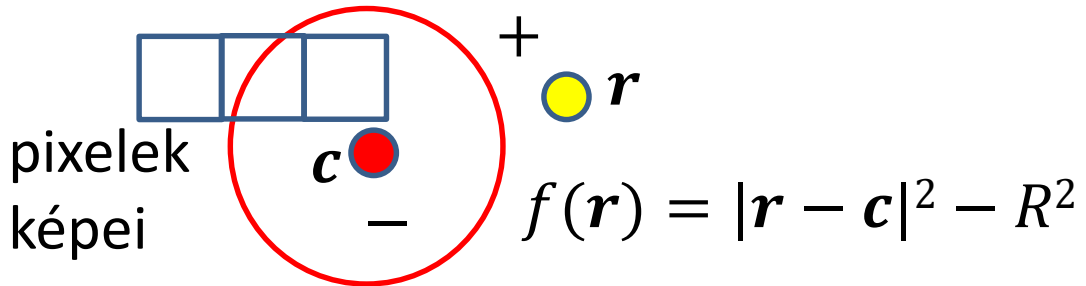
Pixel vezérelt 2D képszintézis



- Pixeleket (pontokat) transzformál
- Bármilyen transzformáció jó
- Pont tartalmazás teszt
- Lassú

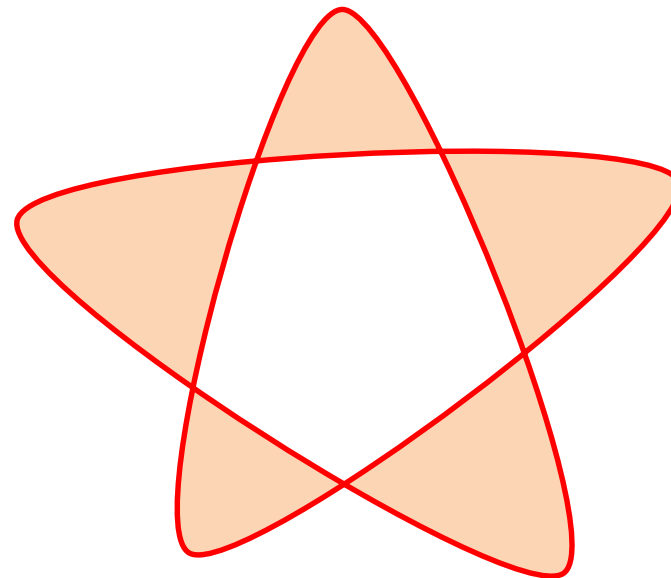
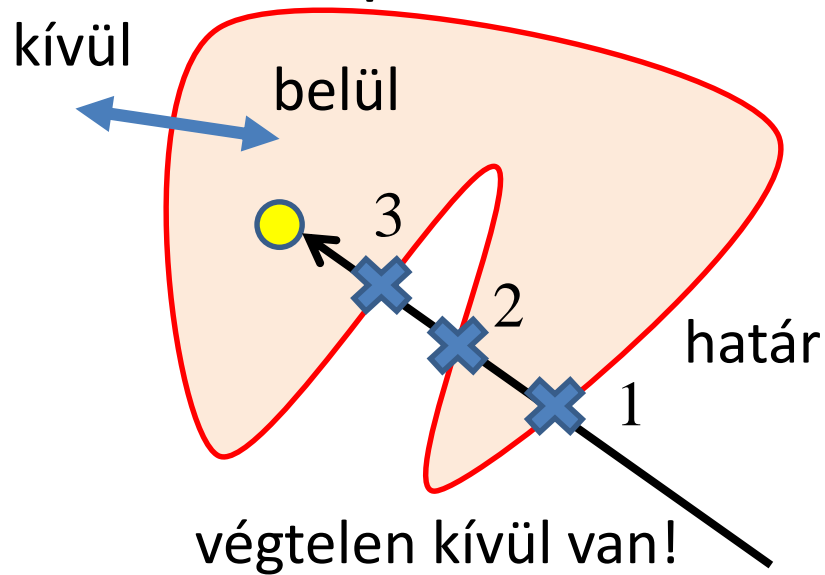
Pixel vezérelt megközelítés: Tartalmazás (objektum, pont)

- Határ implicit görbe:



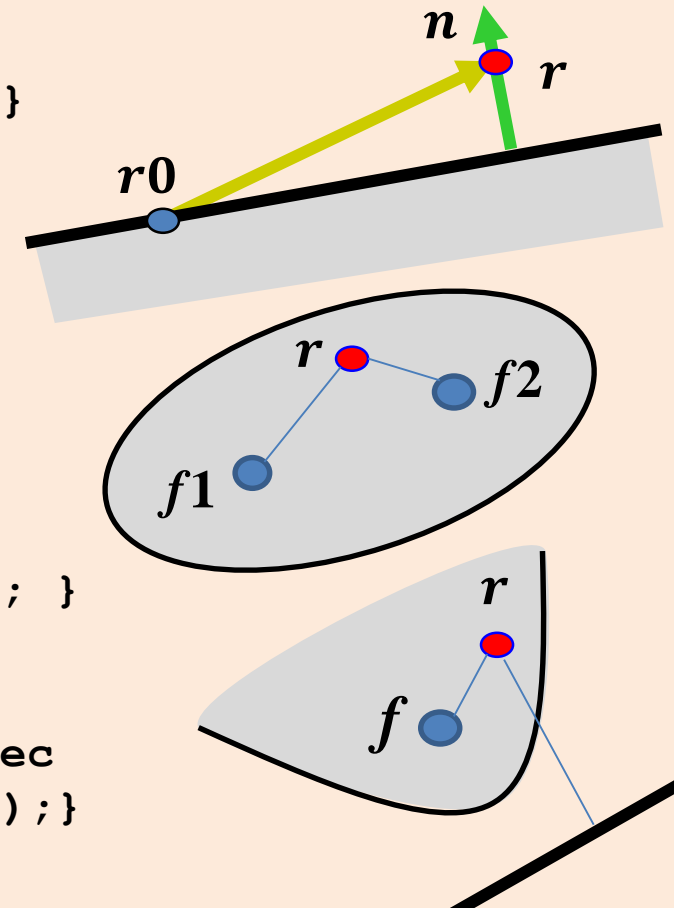
> 0 : egyik oldalon
 $f(x, y) = 0$: határon
 < 0 : másik oldalon
(ált. belül)

- Határ parametrikus görbe:



Pixel vezérelt rendering

```
struct Object { // base class
    vec3 color;
    virtual bool In(vec2 r) = 0; // containment test
};
struct Circle : Object {
    vec2 center;
    float R;
    bool In(vec2 r) { return (dot(r-center, r-center) < R*R); }
};
struct HalfPlane : Object {
    vec2 r0, n; // position vec, normal vec
    bool In(vec2 r) { return (dot(r-r0, n) < 0); }
};
struct GeneralEllipse : Object {
    vec2 f1, f2;
    float C;
    bool In(vec2 r) { return (length(r-f1) + length(r-f2) < C); }
};
struct Parabola : Object {
    vec2 f, r0, n; // f=focus, (r0,n)=directrix line, n=unit vec
    bool In(vec2 r) { return (fabs(dot(r-r0, n)) > length(r-f)); }
};
```



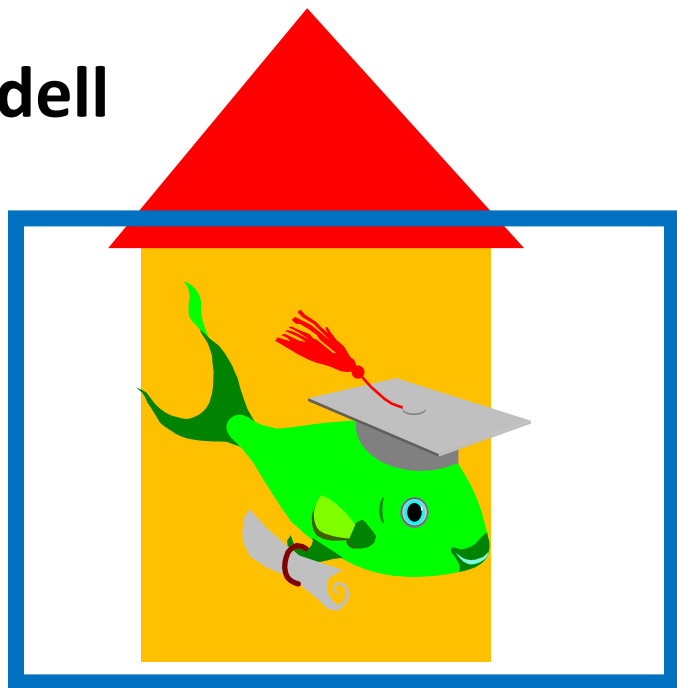
Pixel vezérelt rendering



```
class Scene { // virtual world
    list<Object *> objs; // objects with decreasing priority
    Object *picked = nullptr; // selected for operation
public:
    void Add(Object * o) { objects.push_front(o); picked = o; }
    void Pick(int pX, int pY) { // pX, pY: pixel coordinates
        vec2 wPoint = Viewport2Window(pX, pY); // transform to world
        picked = nullptr;
        for(auto o : objs) if (o->In(wPoint)) { picked = o; return; }
    }
    void BringToFront() {
        if (picked) { // move to the front of the priority list
            objs.erase(find(objs.begin(), objs.end(), picked));
            objs.push_front(picked);
        }
    }
    void Render() {
        for(int pX = 0; pX < xmax; pX++) for(int pY = 0; pY < ymax; pY++) {
            vec2 wPoint = Viewport2Window(pX, pY); // wPoint.x = a * pX + b * pY + c
            for(auto o : objs) if (o->In(wPoint)) { image[pY][pX] = o->color; break; }
        }
    }
};
```

Objektum vezérelt 2D képszintézis

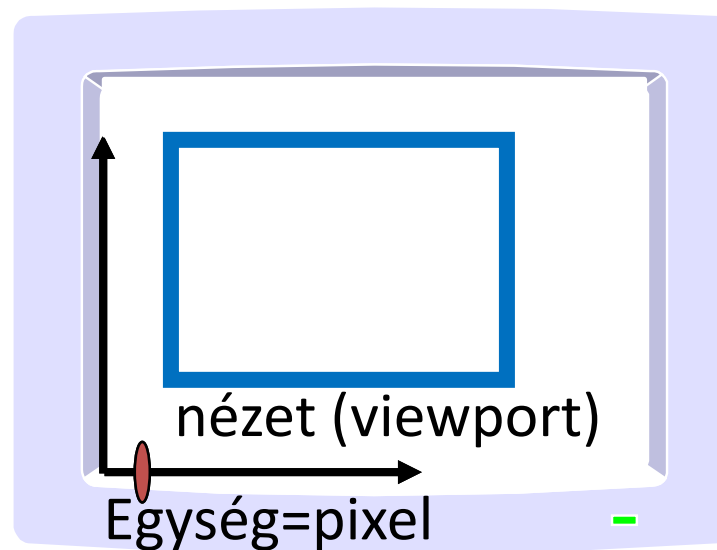
Modell



Kamera ablak (window)

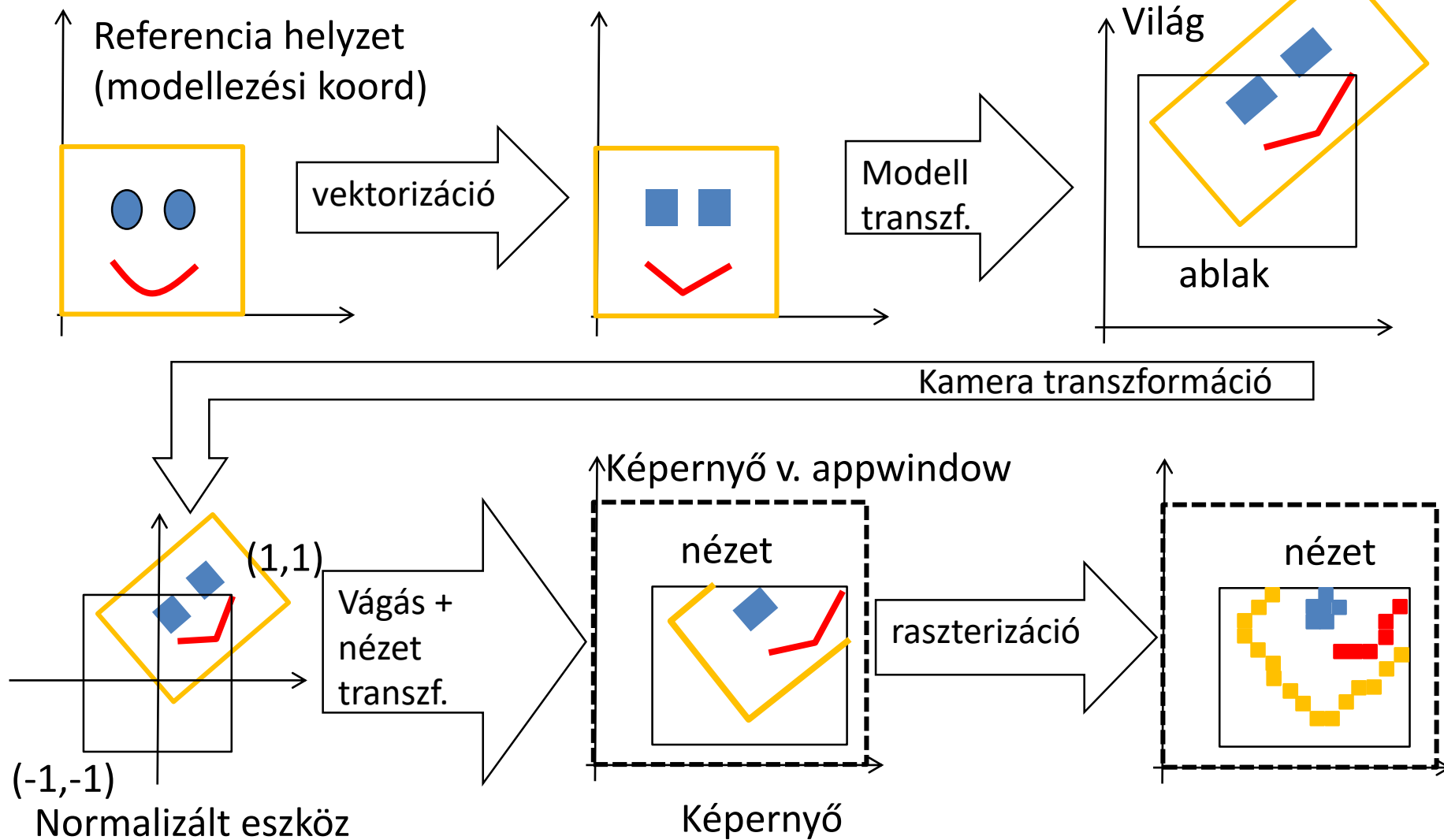
Világ koordinátarendszer

Kép



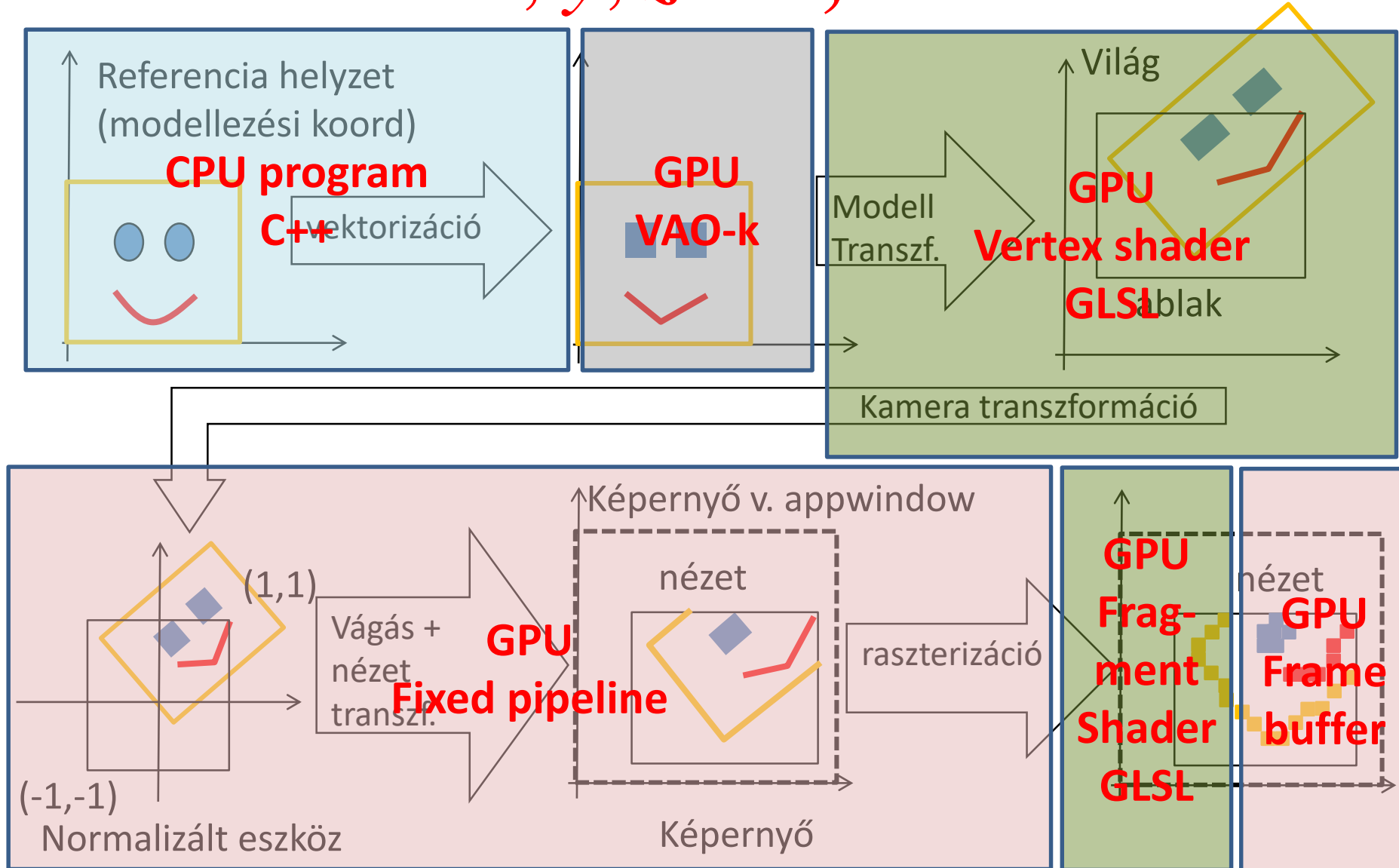
Saját színnel rajzolás
a kis prioritásúakkal kezdve

Objektum vezérelt megközelítés: 2D megjelenítési csővezeték



GPU megjelenítési csővezeték

2D: $x, y, z = 0, w = 1$

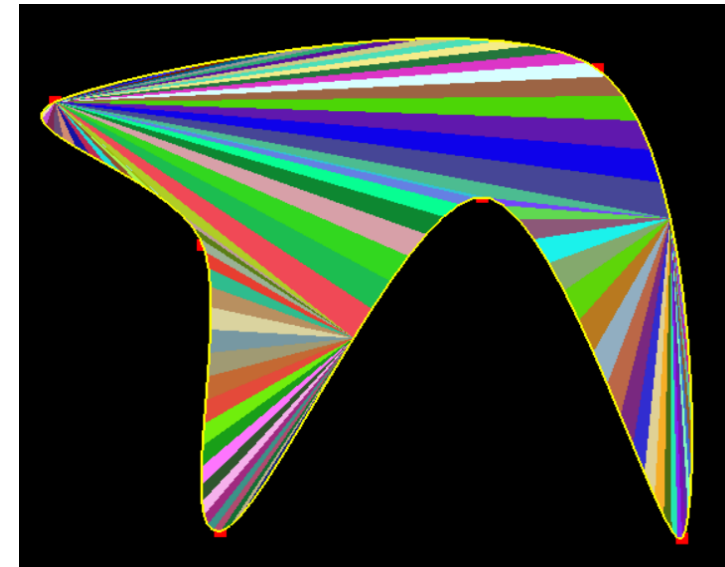


"μή μου τοὺς κύκλους τάραττε."
Ἀρχιμήδης

2D képszintézis

2. Vektorizáció és háromszögesítés

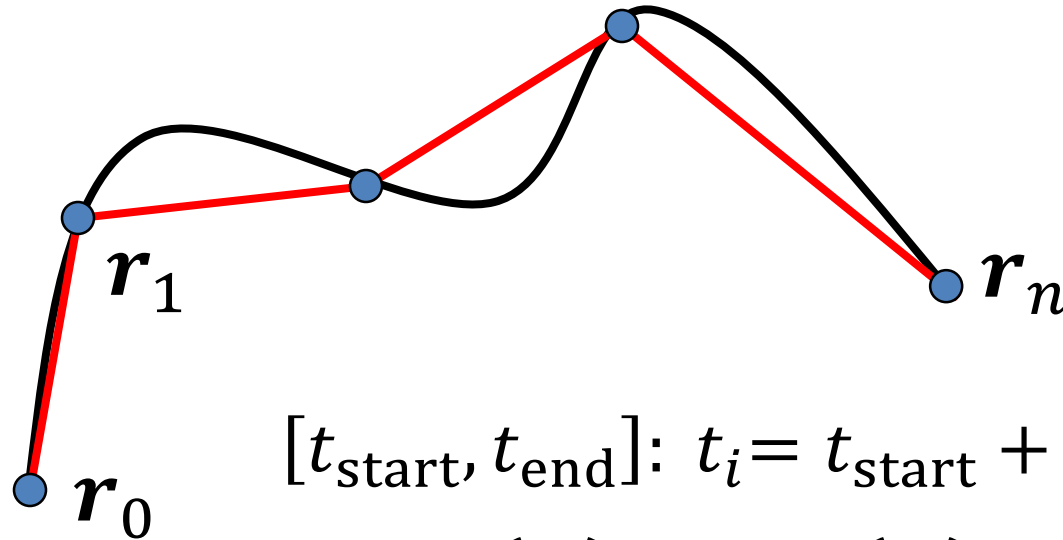
Szirmay-Kalos László



Vektorizáció (CPU)

$$\mathbf{r}(t), t \in [t_{\text{start}}, t_{\text{end}}]$$

GL_LINE_STRIP
GL_LINE_LOOP



$$[t_{\text{start}}, t_{\text{end}}]: t_i = t_{\text{start}} + (t_{\text{end}} - t_{\text{start}})i/n$$

$$\mathbf{r}_0 = \mathbf{r}(t_0), \mathbf{r}_1 = \mathbf{r}(t_1), \dots, \mathbf{r}_n = \mathbf{r}(t_n)$$

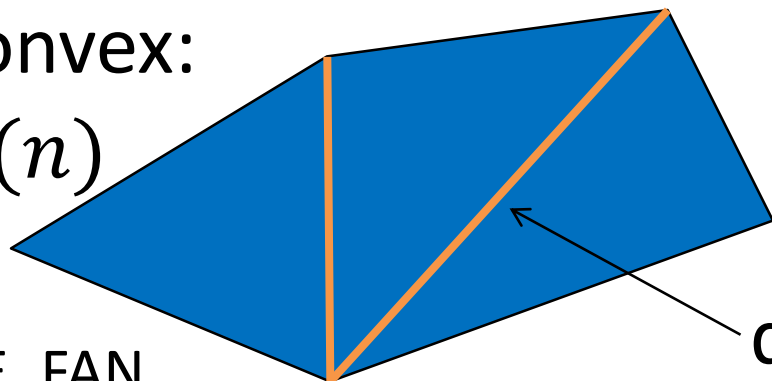
Görbe → nyílt töröttvonal → szakaszok
Terület határa → zárt töröttvonal = poligon → háromszögek

Hw érdekében

Poligon háromszögekre bontása

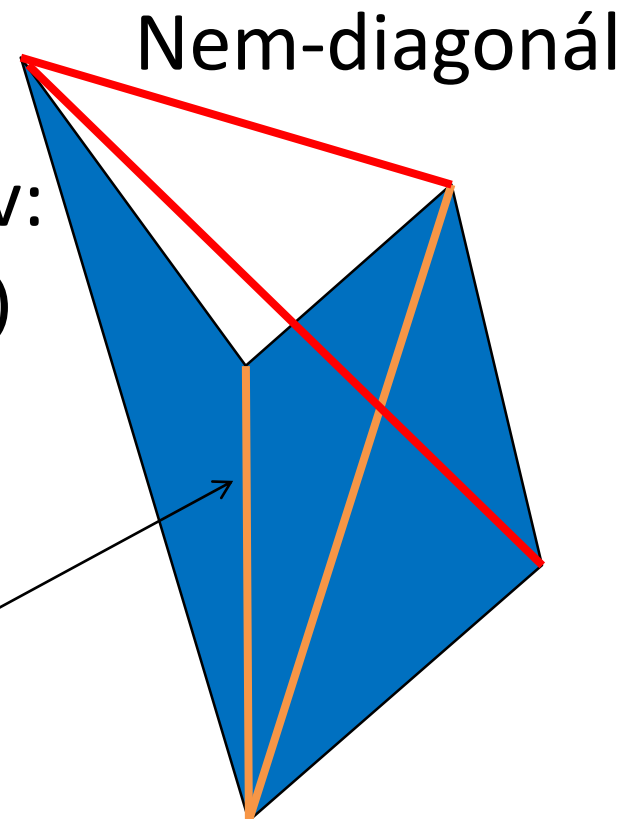
Diagonál mentén értelmes vágni, mert az csökkenti a csúcspontok számát!

Konvex:
 $O(n)$



GL_TRIANGLE_FAN

Konkáv:
 $O(n^4)$



Tétel: Minden 4+ csúcsú egyszerű sokszögnek van diagonálja, azaz mindegyik felbontható diagonálok mentén.

Diagonál me-
csökkenti a cs

Ko
00



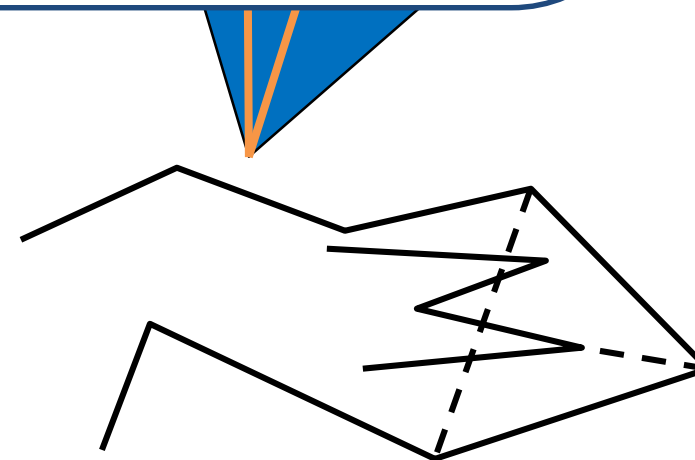
Egyszerű sokszög



Nem egyszerű sokszög



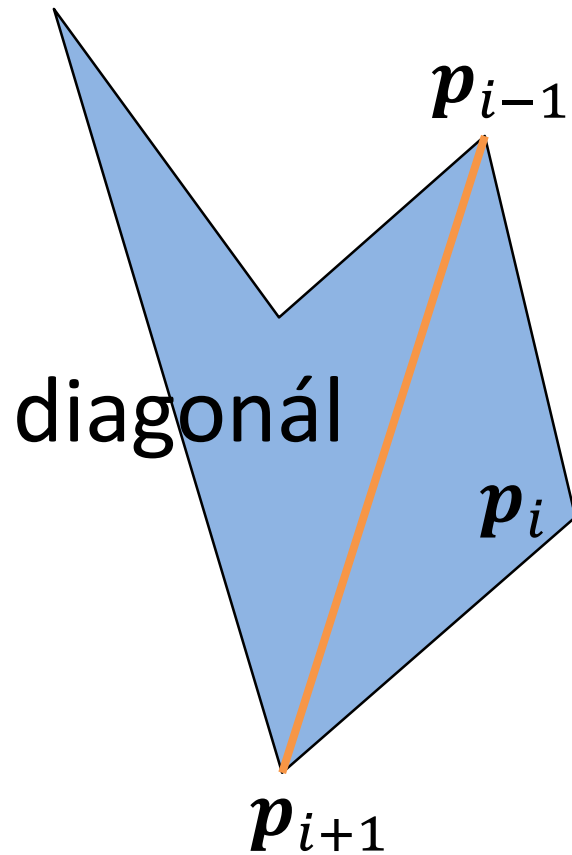
Tétel: Minden 4+ csúcsú egyszerű sokszögnek van diagonálja, azaz mindegyik felbontható diagonálok mentén.



Konvex
csúcs

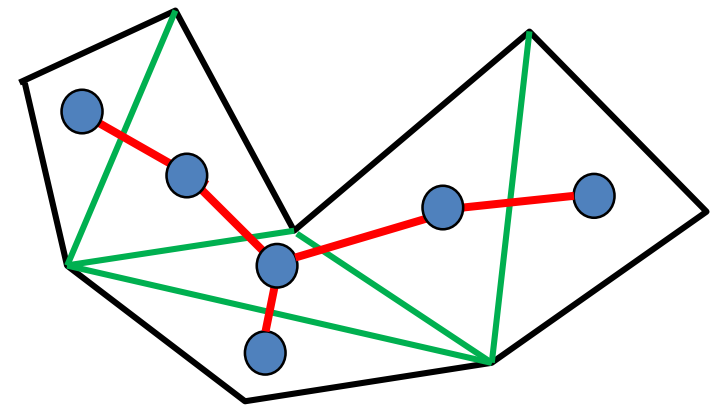


Fül



- p_i fül, ha $p_{i-1} \leftrightarrow p_{i+1}$ diagonál
- Fül levágható!
- **Fülvágás:** keress fület és nyissz!
- $O(n^3)$

Két fül tétele: Minden legalább 4 csúcsú egyszerű sokszögnek van legalább 2 füle.

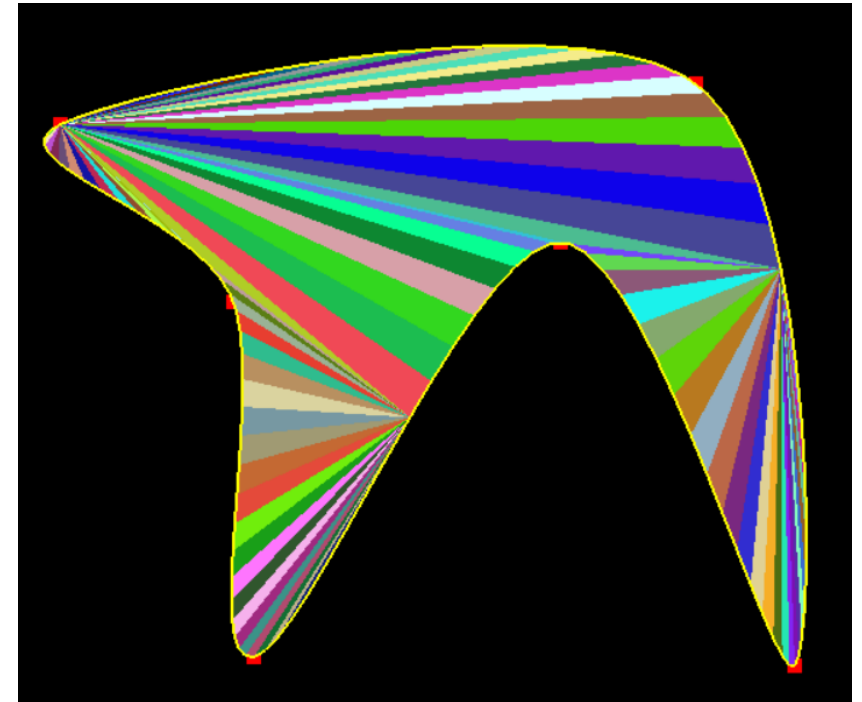
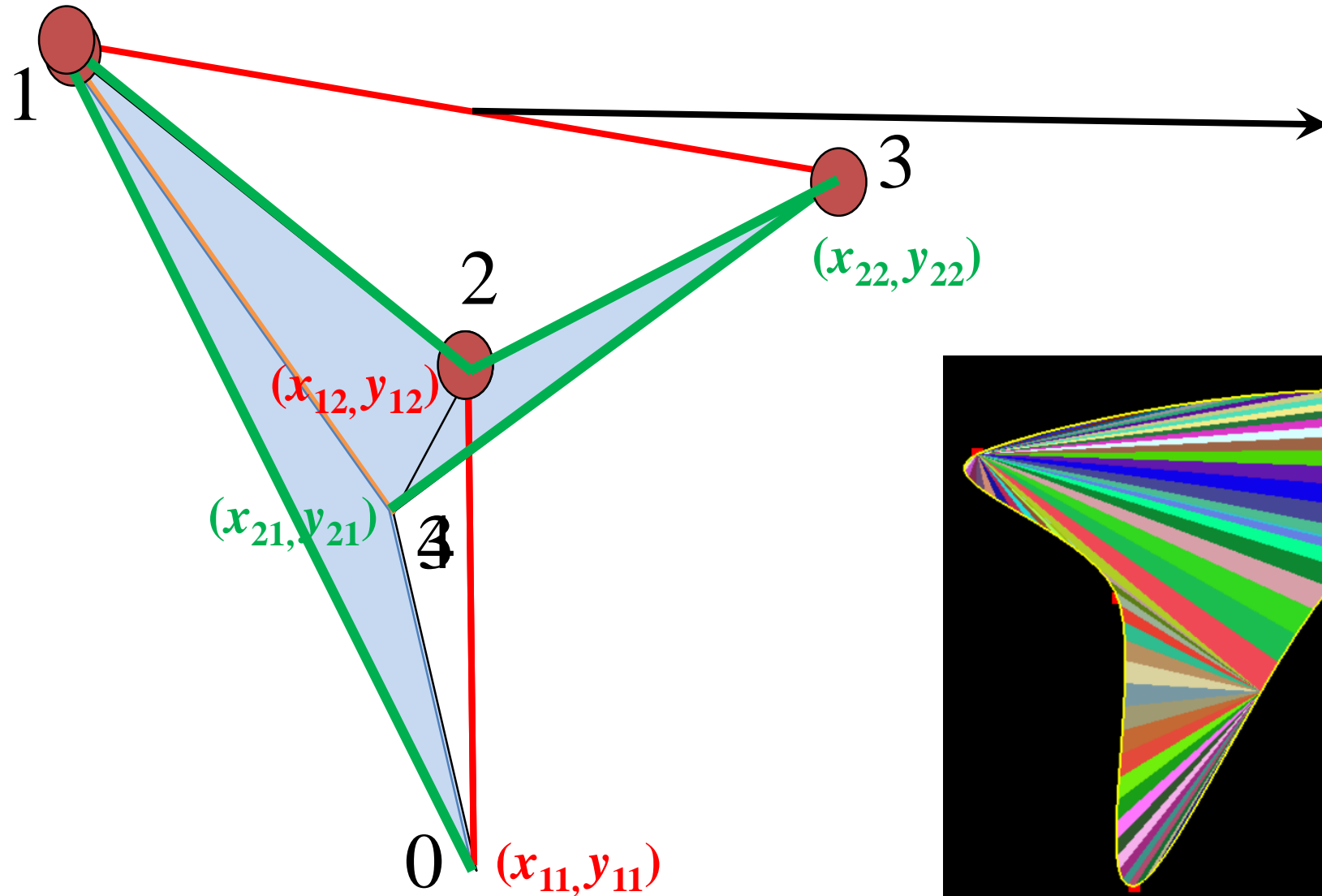


„Minden fának van legalább két levele.”





Fülvágó algoritmus: $O(n^3)$



Szakasz-szakasz metszés

Algebrai megoldás:

$$x_1(t_1) = x_{11}t_1 + x_{12}(1 - t_1)$$

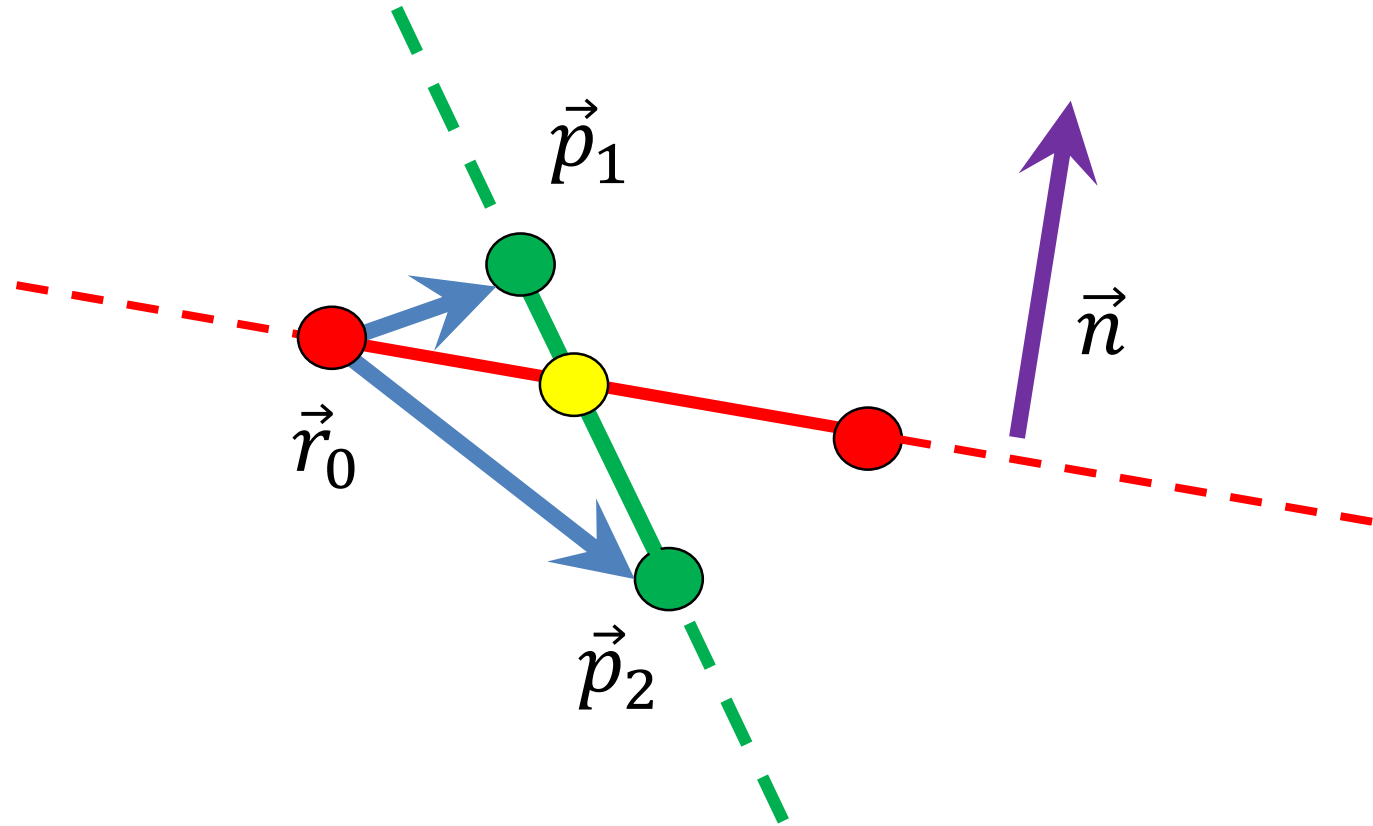
$$y_1(t_1) = y_{11}t_1 + y_{12}(1 - t_1), t_1 \in (0,1)$$

$$x_2(t_2) = x_{21}t_2 + x_{22}(1 - t_2)$$

$$y_2(t_2) = y_{21}t_2 + y_{22}(1 - t_2), t_2 \in (0,1)$$

$$x_1(t_1) = x_2(t_2)$$

$$y_1(t_1) = y_2(t_2) \quad ? t_1, t_2 \in (0,1)$$



$$(\vec{n} \cdot (\vec{p}_1 - \vec{r}_0)) (\vec{n} \cdot (\vec{p}_2 - \vec{r}_0)) < 0$$

“For geometry, you know, is the gate of science, and the gate is so low and small that we can only enter it as a little child.”

William Kingdon Clifford

2D képszintézis

3. Transzformációk és vágás

Szirmay-Kalos László



Modellezési transzformáció

- Mátrixokat a CPU-n számítjuk, a transzformációt a GPU hajtja végre
- Homogén lineáris transzformáció:

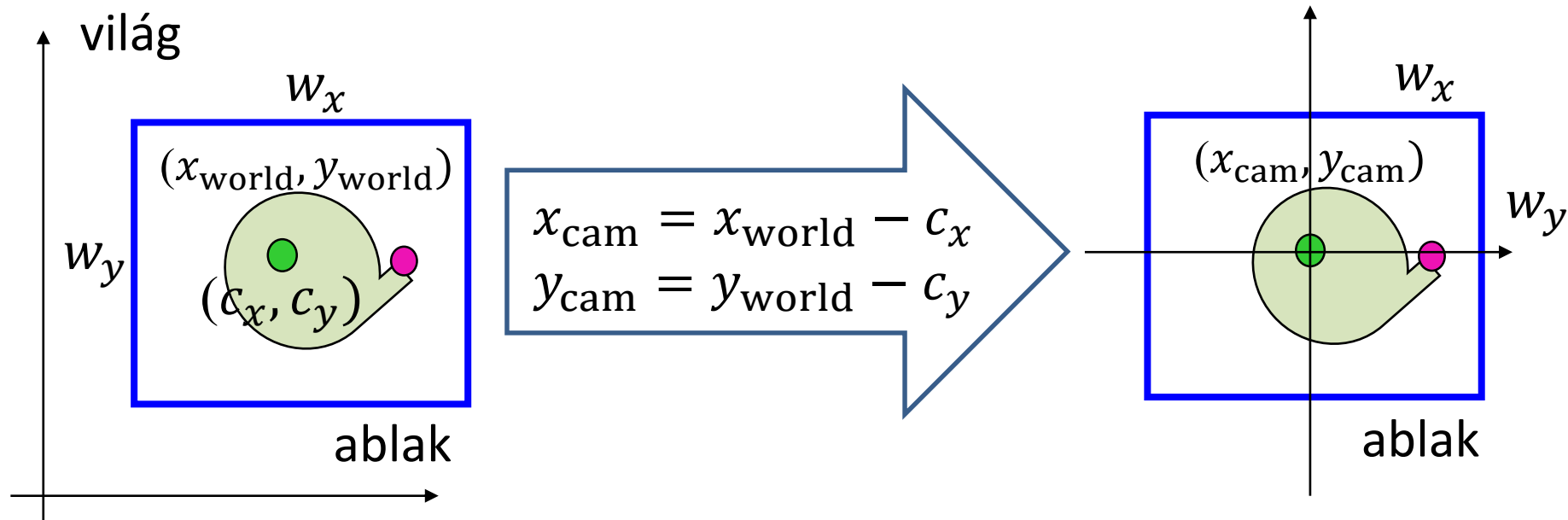
$$[x_{\text{world}}, y_{\text{world}}, z_{\text{world}}, 1] = [x_{\text{model}}, y_{\text{model}}, z_{\text{model}}, 1] \cdot \mathbf{T}_{4 \times 4}$$

- Példa: skálázás, forgatás, eltolás:

$$\mathbf{T}_{4 \times 4} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & * & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\varphi) & \sin(\varphi) & 0 & 0 \\ -\sin(\varphi) & \cos(\varphi) & 0 & 0 \\ 0 & 0 & * & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & * & 0 \\ p_x & p_y & 0 & 1 \end{bmatrix}$$

View transzformáció: $V()$

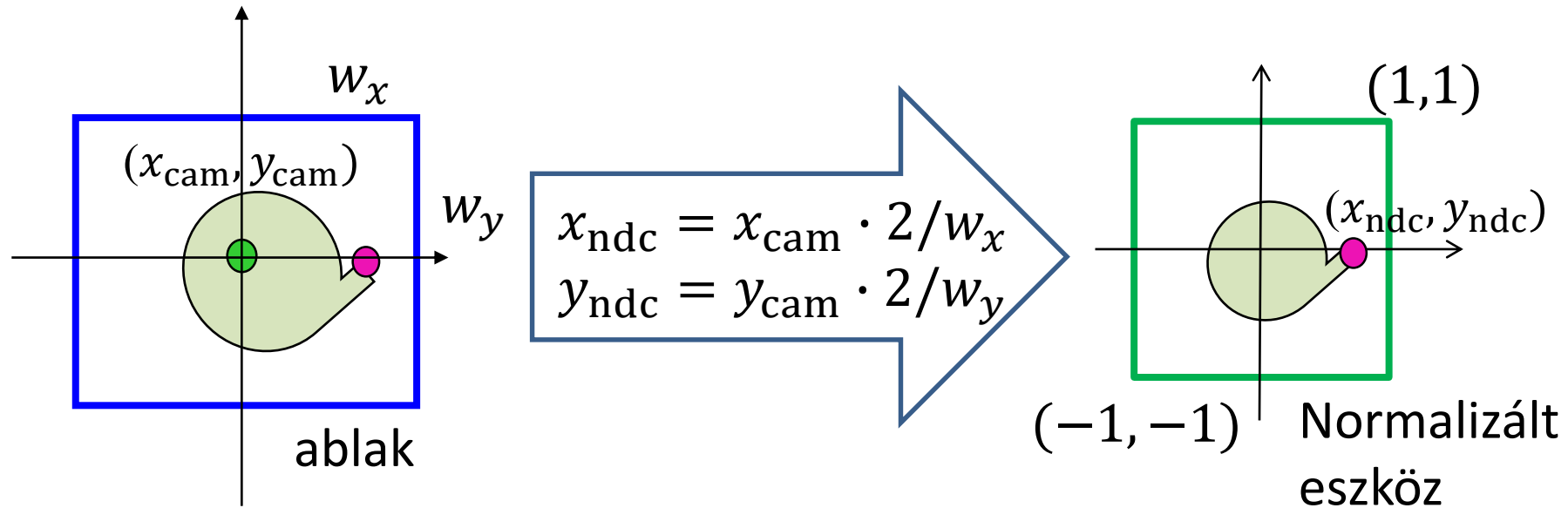
Kameraablak közepe az origóba



$$[x_{\text{cam}}, y_{\text{cam}}, z_{\text{cam}}, 1] = [x_{\text{world}}, y_{\text{world}}, z_{\text{world}}, 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -c_x & -c_y & 0 & 1 \end{bmatrix}$$

Projekció: P()

Kameraablak a $(-1, -1)$ - $(1, 1)$ négyzetbe



$$[x_{ndc}, y_{ndc}, z_{ndc}, 1] = [x_{cam}, y_{cam}, z_{cam}, 1] \begin{bmatrix} 2/w_x & 0 & 0 & 0 \\ 0 & 2/w_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2D kamera

```
class Camera2D {  
    vec2 wCenter; // center in world coords  
    vec2 wSize;  // width and height in world coords  
  
public:  
    mat4 V() { return TranslateMatrix(-wCenter); }  
  
    mat4 P() { // projection matrix  
        return ScaleMatrix(vec2(2/wSize.x, 2/wSize.y));  
    }  
  
    mat4 Vinv() { // inverse view matrix  
        return TranslateMatrix(wCenter);  
    }  
  
    mat4 Pinv() { // inverse projection matrix  
        return ScaleMatrix(vec2(wSize.x/2, wSize.y/2));  
    }  
  
    void Zoom(float s) { wSize = wSize * s; }  
    void Pan(vec2 t) { wCenter = wCenter + t; }  
};
```

2D Object

```
class Object {
    unsigned int vao = 0, vbo = 0, nVtx = 0; // GPU
protected:
    vec2  scale = vec2(1, 1), pos = vec2(0, 0);
    float phi = 0;           // rotation
public:
    void Create() {
        glGenVertexArrays(1, &vao); glBindVertexArray(vao);
        glGenBuffers(1, &vbo); glBindBuffer(GL_ARRAY_BUFFER, vbo);
        glEnableVertexAttribArray(0); glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, NULL);
    }
    virtual vector<vec2> GenVertexData() = 0; // vectorization, ear clipping, etc.
    void UpdateGPU() { // CPU -> GPU
        glBindVertexArray(vao); glBindBuffer(GL_ARRAY_BUFFER, vbo);
        vector<vec2> vtx = GenVertexData(); nVtx = vtx.size();
        glBufferData(GL_ARRAY_BUFFER, nVtx * sizeof(vec2), &vtx[0], GL_STATIC_DRAW);
    }
    void Draw(int type, vec3 color) {
        if (nVtx == 0) return;
        mat4 M = ScaleMatrix(scale) * RotationXYMatrix(phi) * TranslateMatrix(pos);
        mat4 MVP = M * camera.V() * camera.P();
        gpuProgram.setUniform(MVP, "MVP");
        gpuProgram.setUniform(color, "color");
        glBindVertexArray(vao); glDrawArrays(type, 0, nVtx);
    }
};
```

Példa: Bézier görbe

```
const int nTessVertices = 100;

class BezierCurve : public Object {
    vector<vec2> cps; // control pts
    float B(int i, float t) {
        float choose = 1;
        for(int j = 1; j <= i; j++) choose *= (float)(cps.size()-j)/j;
        return choose * pow(t, i) * pow(1-t, cps.size()-1-i);
    }
public:
    void AddControlPoint(vec2 cp) { cps.push_back(cp); UpdateGPU(); }
    vec2 r(float t) {
        vec2 rt(0, 0);
        for(int i = 0; i < cps.size(); i++) rt += cps[i] * B(i,t);
        return rt;
    }
    vector<vec2> GenVertexData() {
        vector<vec2> vertices;
        for(int i = 0; i <= nTessVertices; ++i) {
            float t = (float)i / nTessVertices;
            vertices.push_back(r(t));
        }
        return vertices;
    }
};
```

Csúcspon t és pixel árnyalók

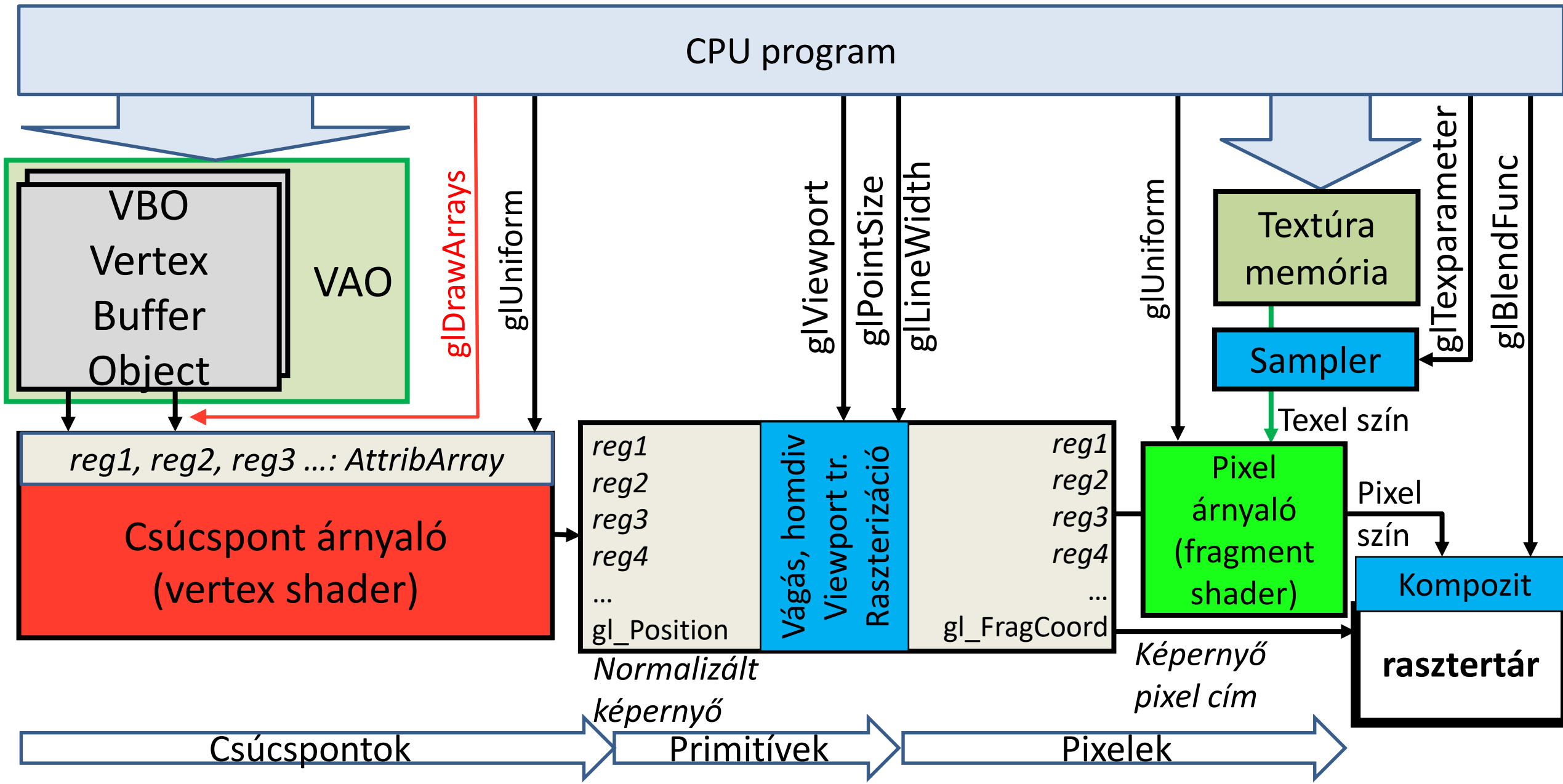
Vertex shader:

```
uniform mat4 MVP;  
layout(location = 0) in vec2 vertexPosition;  
  
void main() {  
    gl_Position = vec4(vertexPosition, 0, 1) * MVP;  
}
```

Fragment shader:

```
uniform vec3 color;  
out vec4 fragmentColor;  
  
void main() {  
    fragmentColor = vec4(color, 1);  
}
```

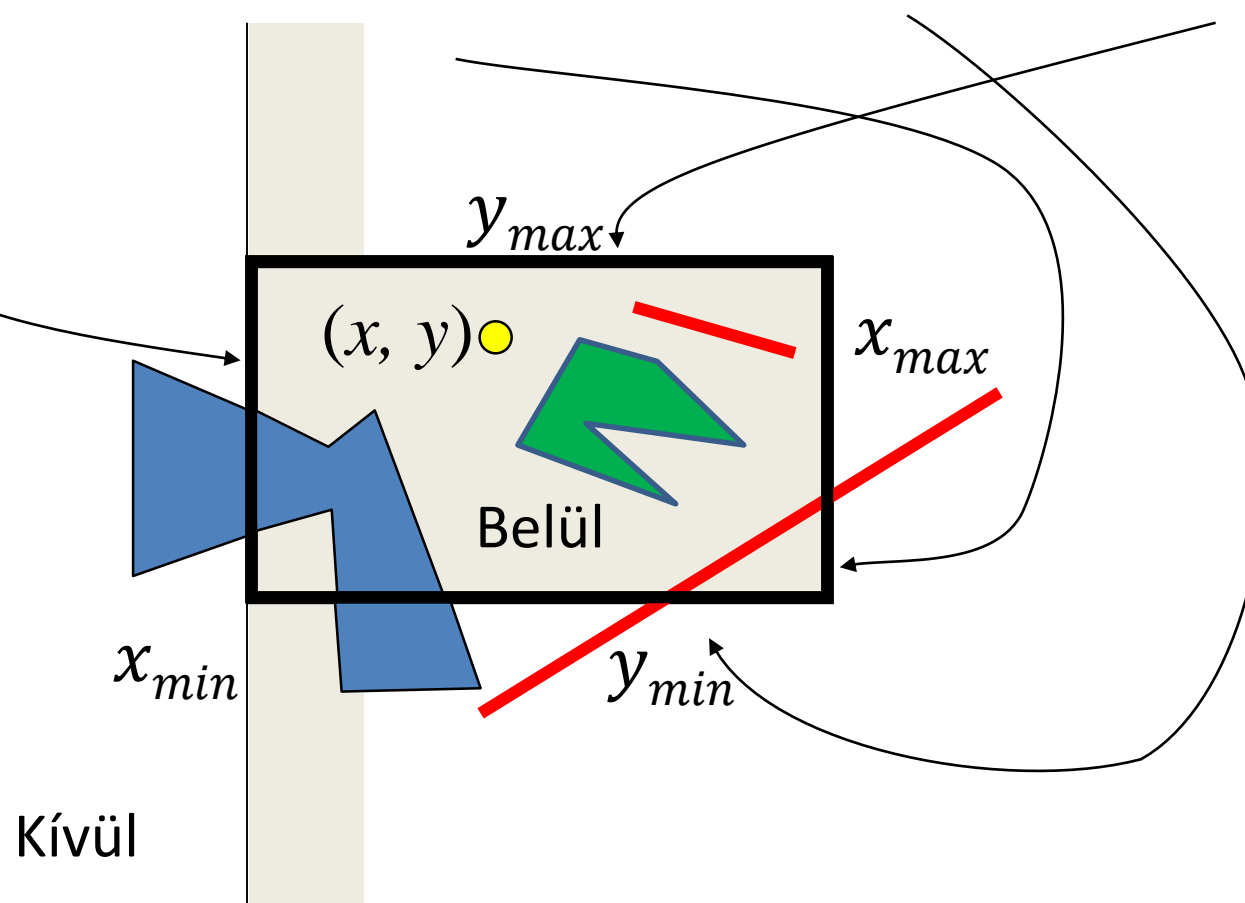

OpenGL 3.3 ... 4.6 (Modern OpenGL)



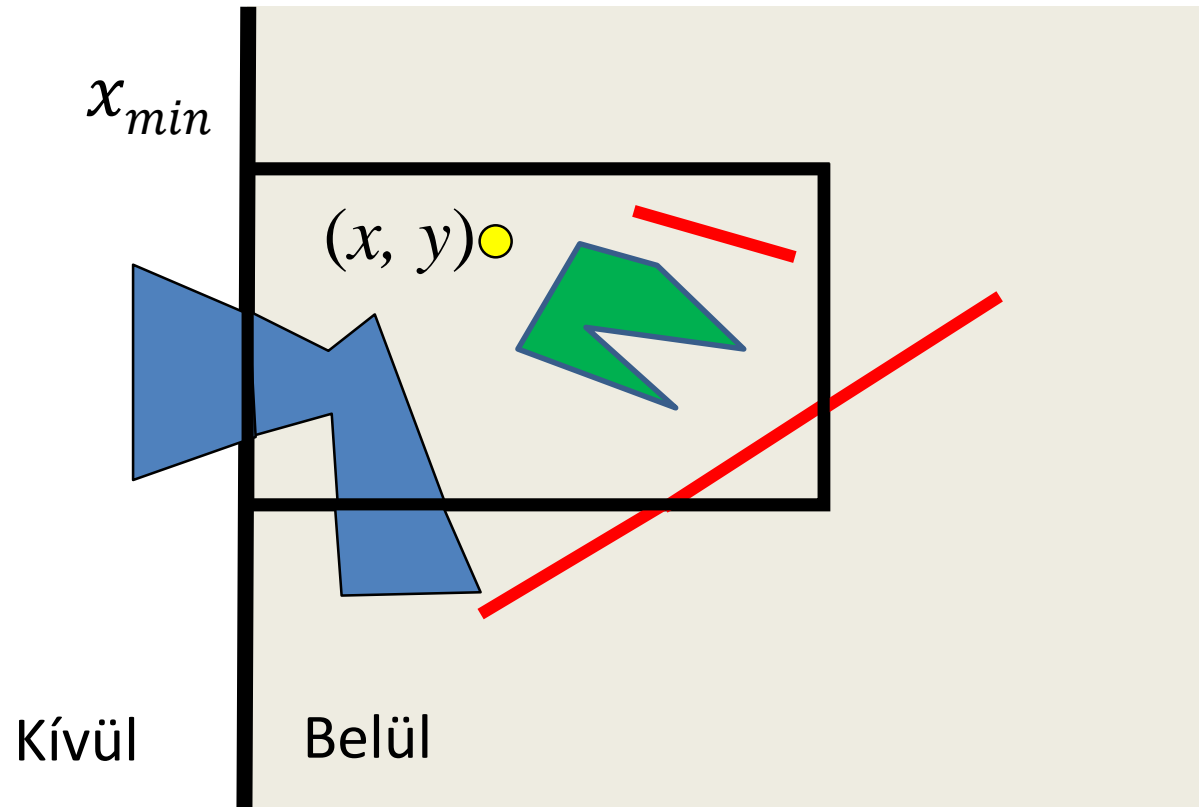
2D vágás

Pont vágás:

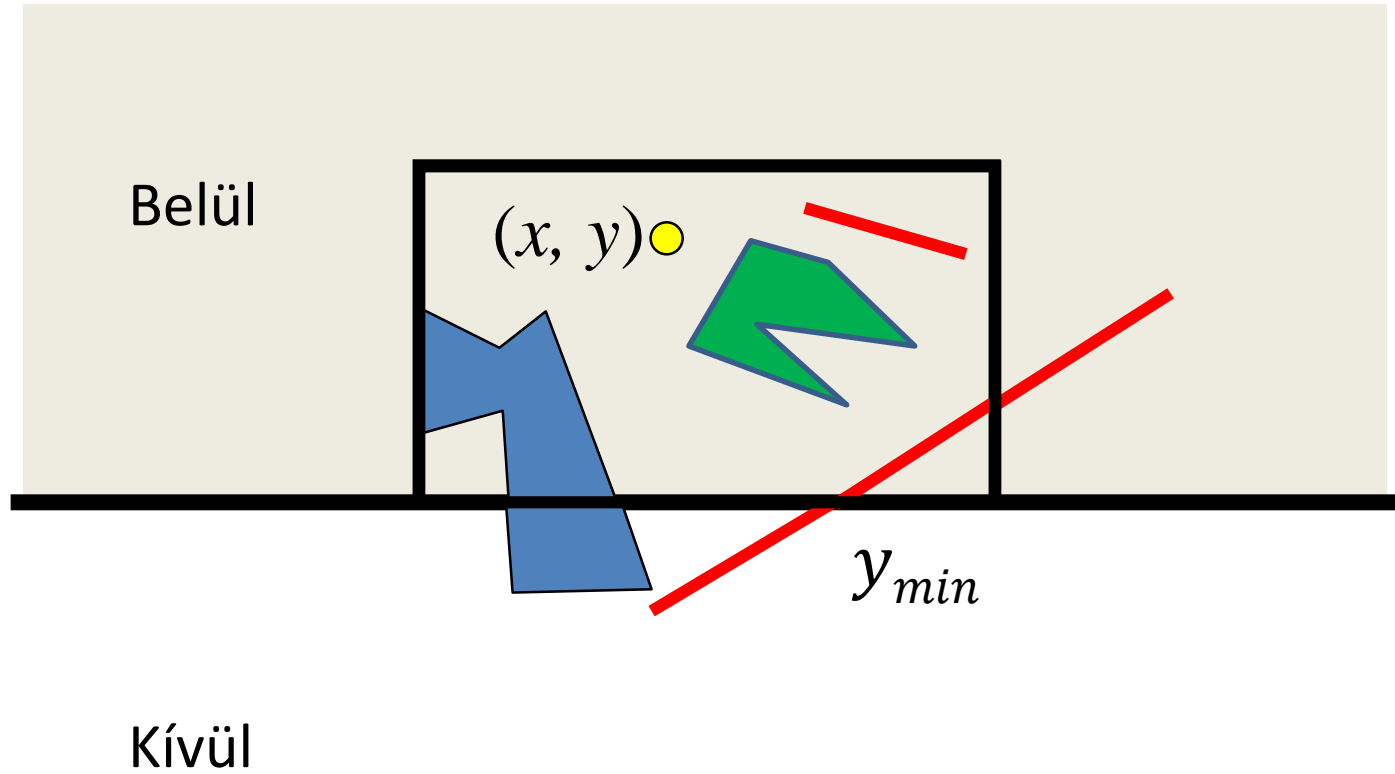
$$x > x_{min} = -1, x < x_{max} = +1, y > y_{min} = -1, y < y_{max} = +1$$



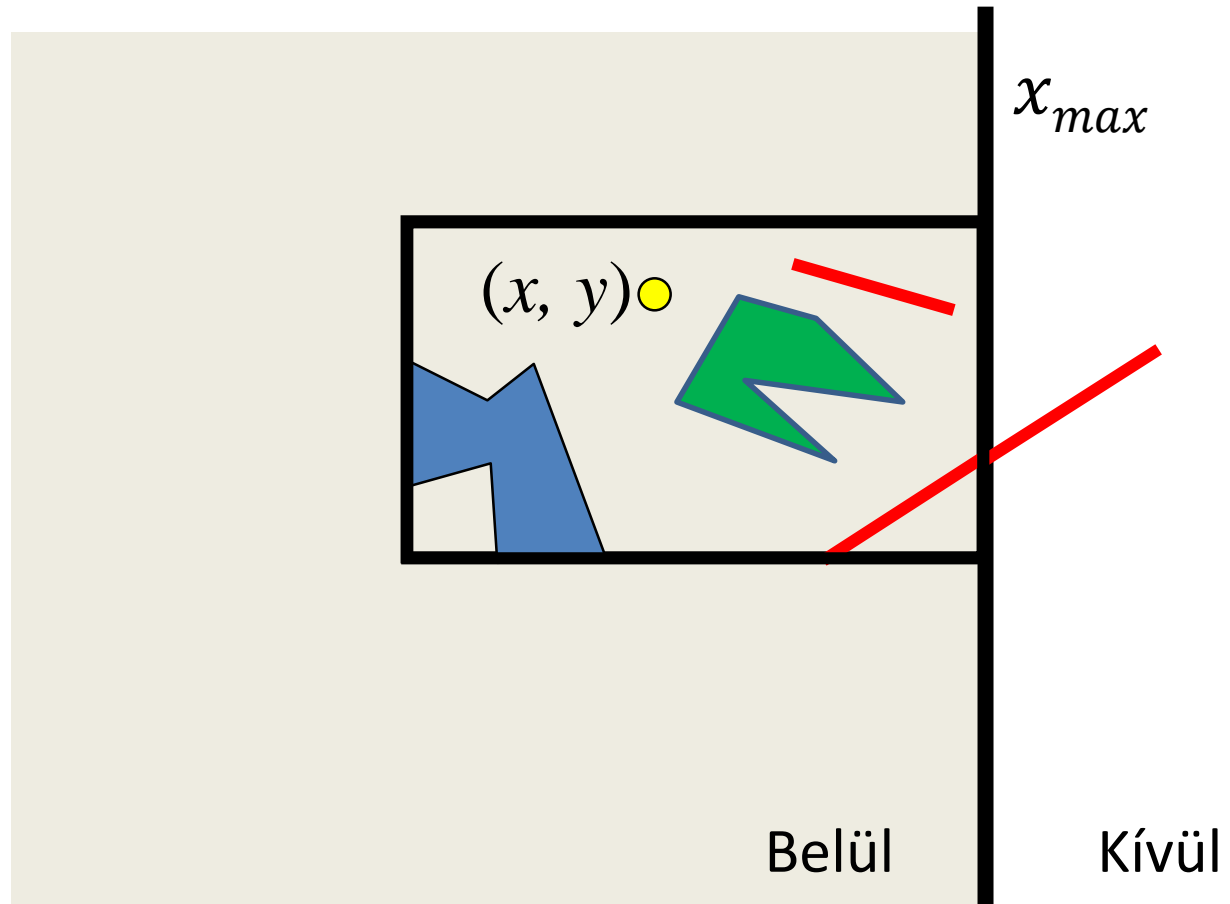
Vágás



Vágás

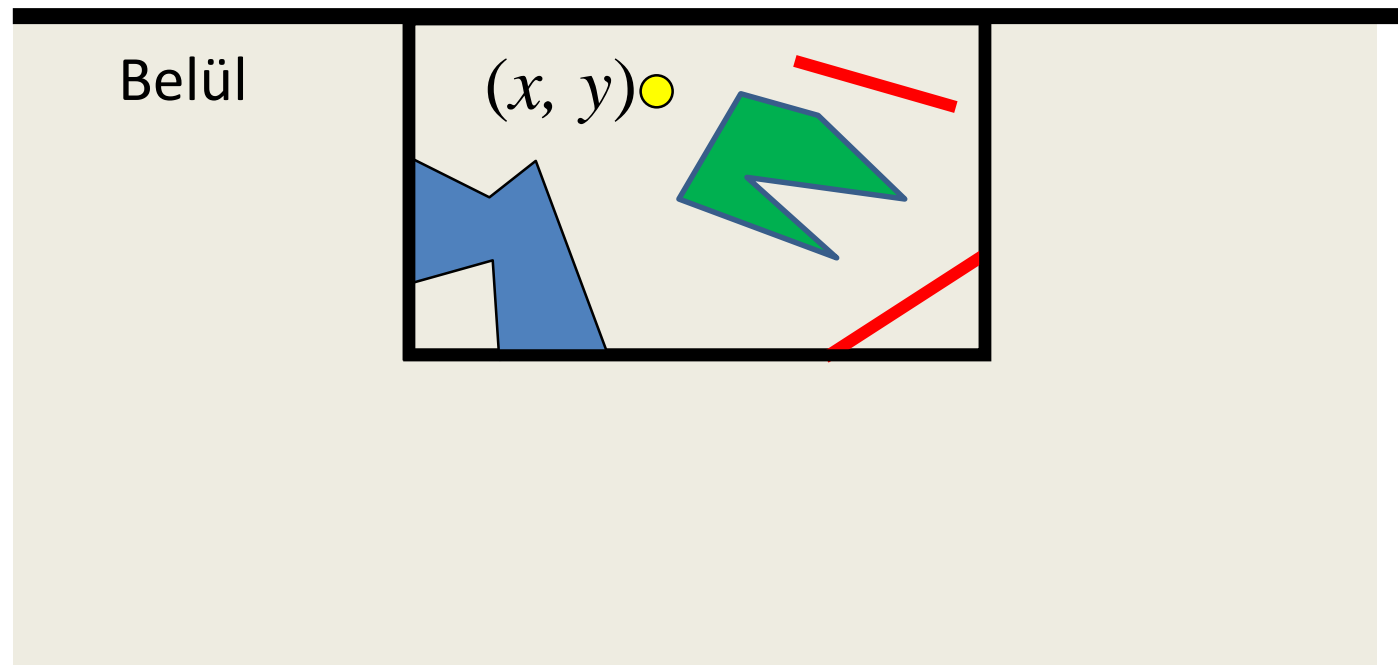


Vágás

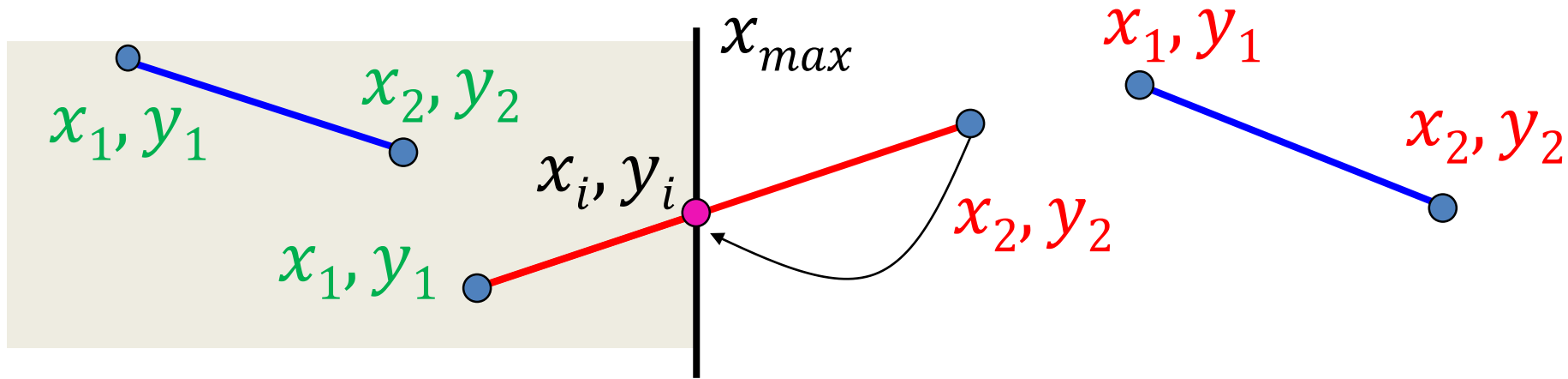


Vágás

Kívül



2D szakasz vágás: $x < x_{max}$



$$x(t) = x_1 + (x_2 - x_1)t, \quad y(t) = y_1 + (y_2 - y_1)t$$

$$x = x_{max}$$

$$\text{Metszés: } x_{max} = x_1 + (x_2 - x_1)t \Rightarrow t = (x_{max} - x_1)/(x_2 - x_1)$$

$$x_i = x_{max} \quad y_i = y_1 + (y_2 - y_1) (x_{max} - x_1)/(x_2 - x_1)$$

(Ivan) Sutherland-(Gary) Hodgman poligonvágás



```
PolygonClip(p[n] ⇒ q[m])
```

```
  m = 0;
```

```
  for(i=0; i < n; i++) {
```

```
    if (p[i] belső) {
```

```
      q[m++] = p[i];
```

```
      if (p[i+1] külső)
```

```
        q[m++] = Intersect(p[i], p[i+1], vágóegyenes);
```

```
    } else {
```

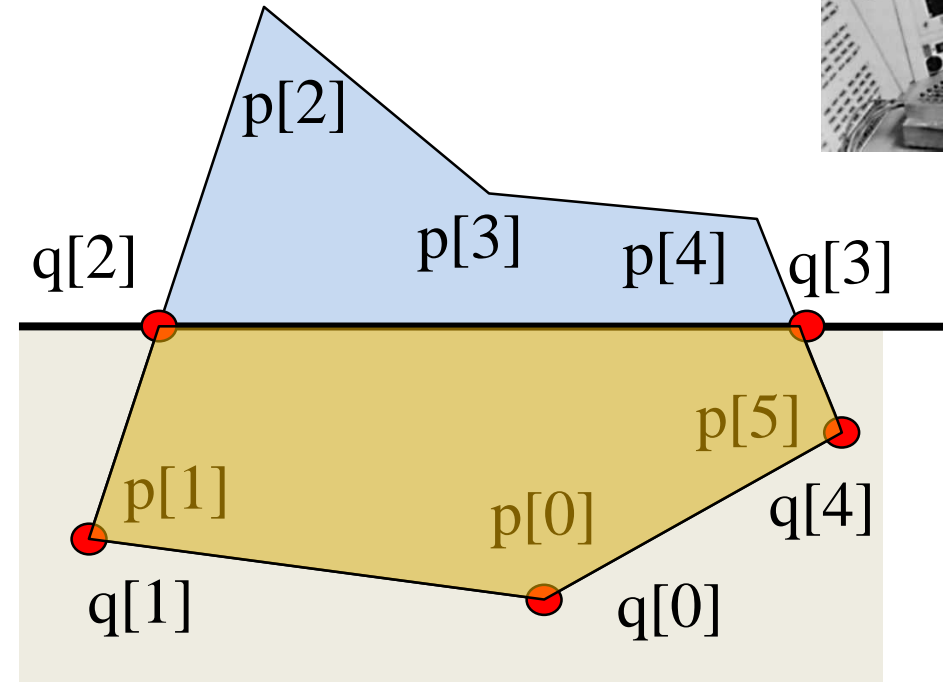
```
      if (p[i+1] belső)
```

```
        q[m++] = Intersect(p[i], p[i+1], vágóegyenes);
```

```
    }
```

```
  }
```

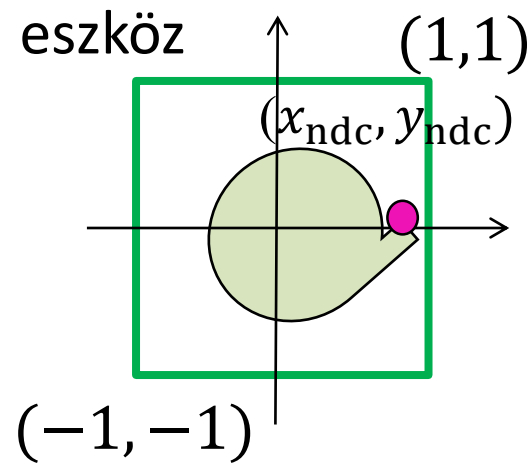
```
}
```



Első pontot még egyszer
a tömb végére

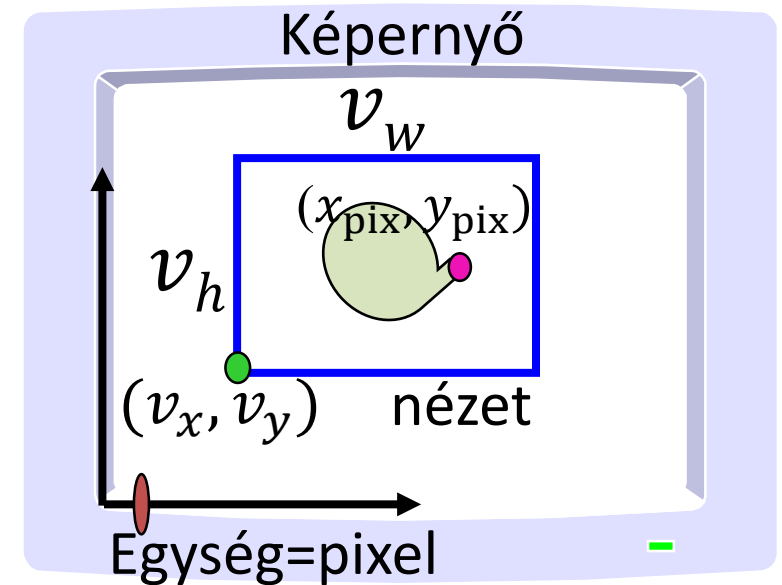
Viewport transzformáció: Normalizáltból képernyő koordinátákba (GPU)

Normalizált



$$x_{pix} = v_w(x_{ndc} + 1)/2 + v_x$$
$$y_{pix} = v_h(y_{ndc} + 1)/2 + v_y$$

$$z_{pix} = (z_{ndc} + 1)/2$$



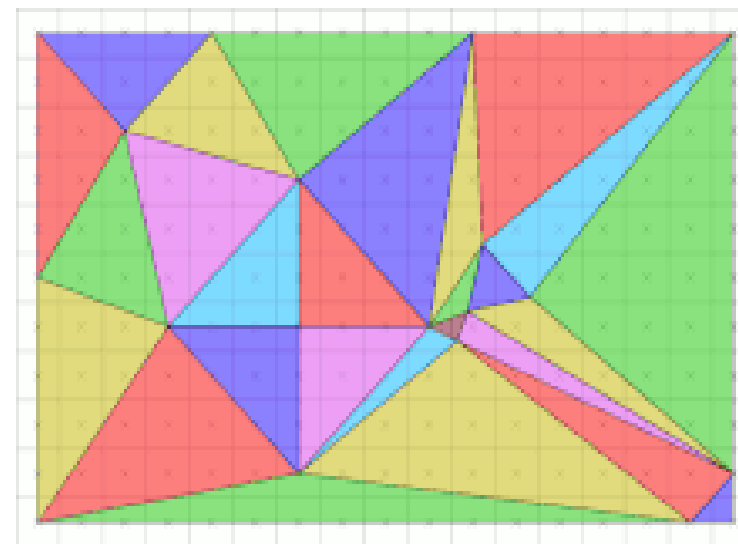
`glViewport(vx, vy, vw, vh)`

*“The computer was born to solve
problems that did not exist before.”
Bill Gates*

2D képszintézis

4. Raszterizáció

Szirmay-Kalos László

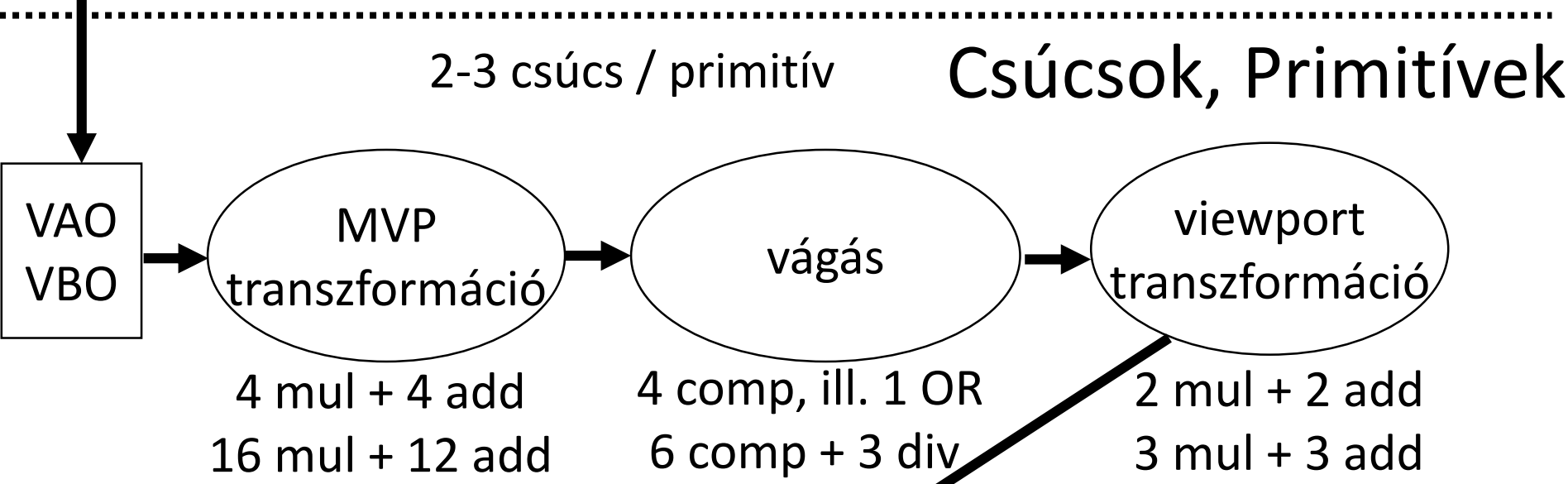


Raszterizáció (GPU)

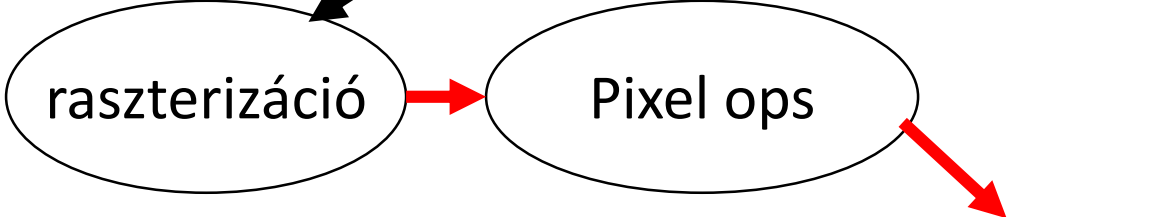
CPU



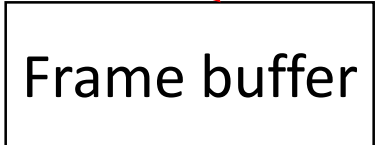
GPU



Pixelek



Sok (!) pixel/primitív
nanosec/pixel

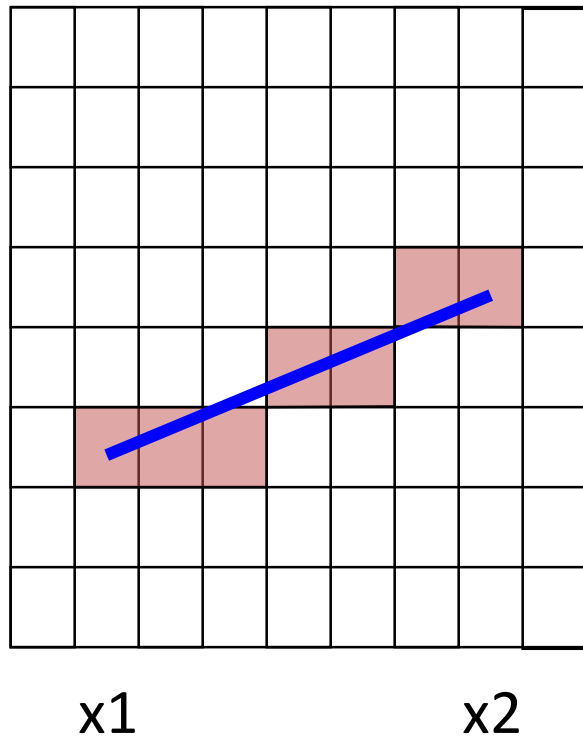


Pont rajzolás

A pont „**kicsi**” és van jellemző helye:

- A kiszínezett tartomány is legyen kicsi
- A legkisebb dolog, amit át lehet színezni, a pixel
- Színezzünk ki egy (vagy néhány) pixelt, amely legközelebb van a ponthoz
- Pixelkoordináták egészek
- **Legközelebbi pixel** = koordináták kerekítése

Szakasz rajzolás



Egyenes „**vékony**” és **összefüggő**.
Pontjai kielégítik az egyenletét:

$$y = mx + b$$

$x_2 > x_1$, $|x_2 - x_1| \geq |y_2 - y_1|$
típusú szakasz rajzolása:

```
float m = (float)(y2-y1)/(x2-x1);  
for(int x = x1; x <= x2; x++) {  
    float y = m*x + b;  
    int Y = round( y );  
    write( x, Y );  
}
```

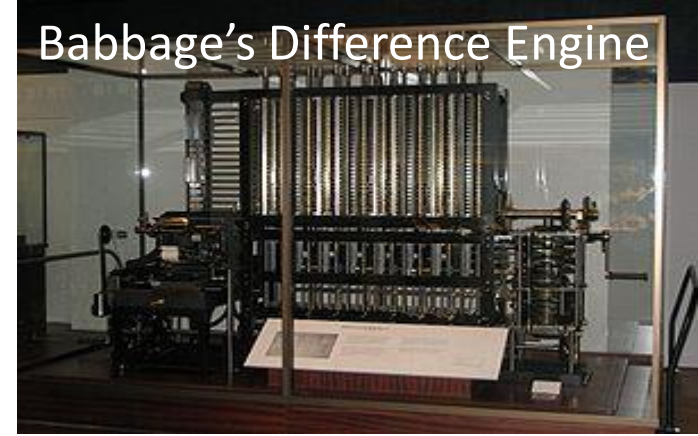
Inkrementális elv és fixpontos számítás

$$y(x) = mx + b = y(x - 1) + m$$

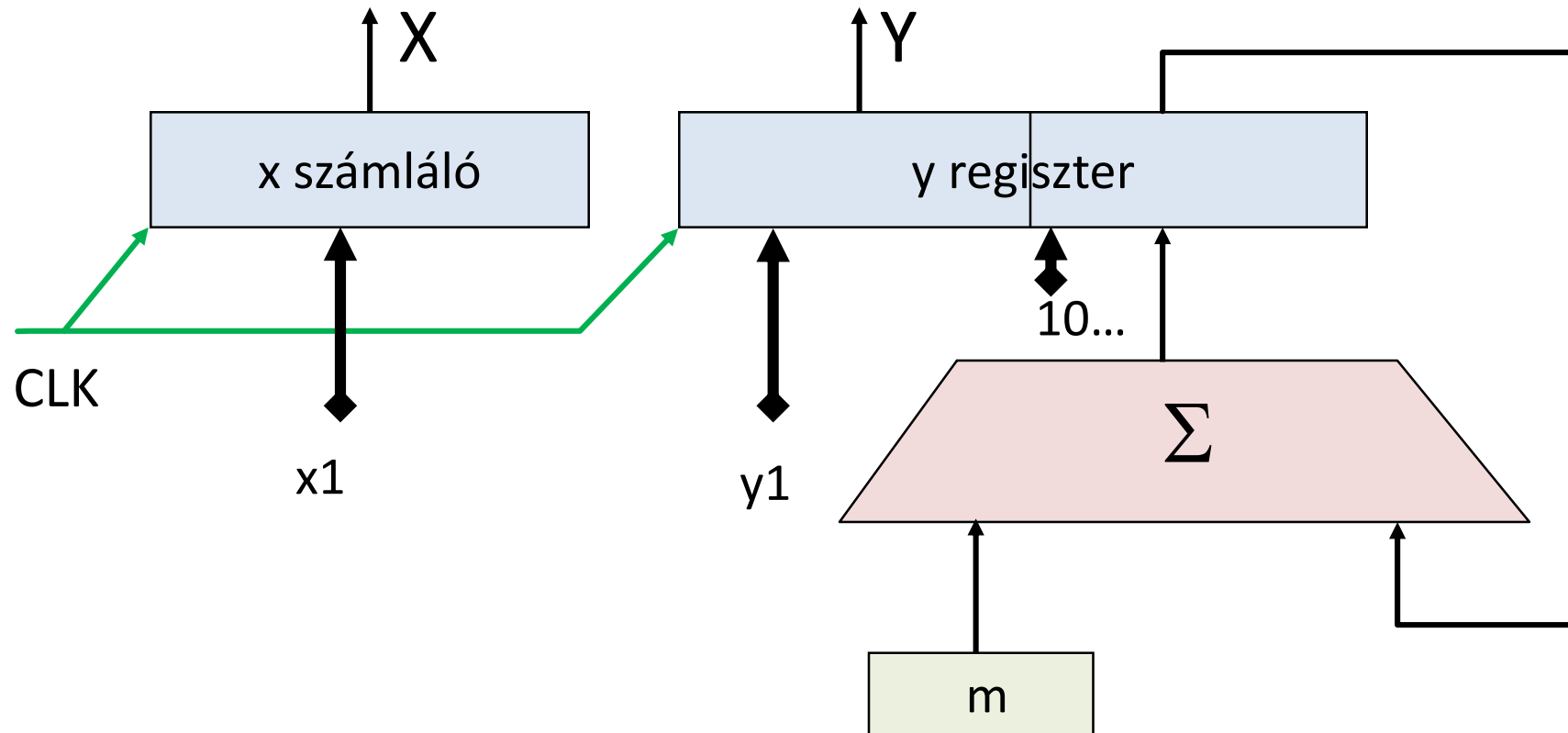
```
LineFloat (short x1, short y1,  
           short x2, short y2) {  
    float m = (float)(y2-y1)/(x2-x1);  
    float y = y1;  
    for(short x = x1; x <= x2; x++) {  
        short Y = round(y);  
        write(x, Y, color);  
        y = y+m;  
    }  
}
```

```
const int T=12; // fractional bits
```

```
LineFix (short x1, short y1,  
         short x2, short y2) {  
    int m = ((y2 - y1)<<T)/(x2 - x1);  
    int y = (y1<<T)+(1<<(T-1)); // +0.5  
    for(short x = x1; x <= x2; x++) {  
        short Y = y >> T; // trunc  
        write(x, Y, color);  
        y = y+m;  
    }  
}
```



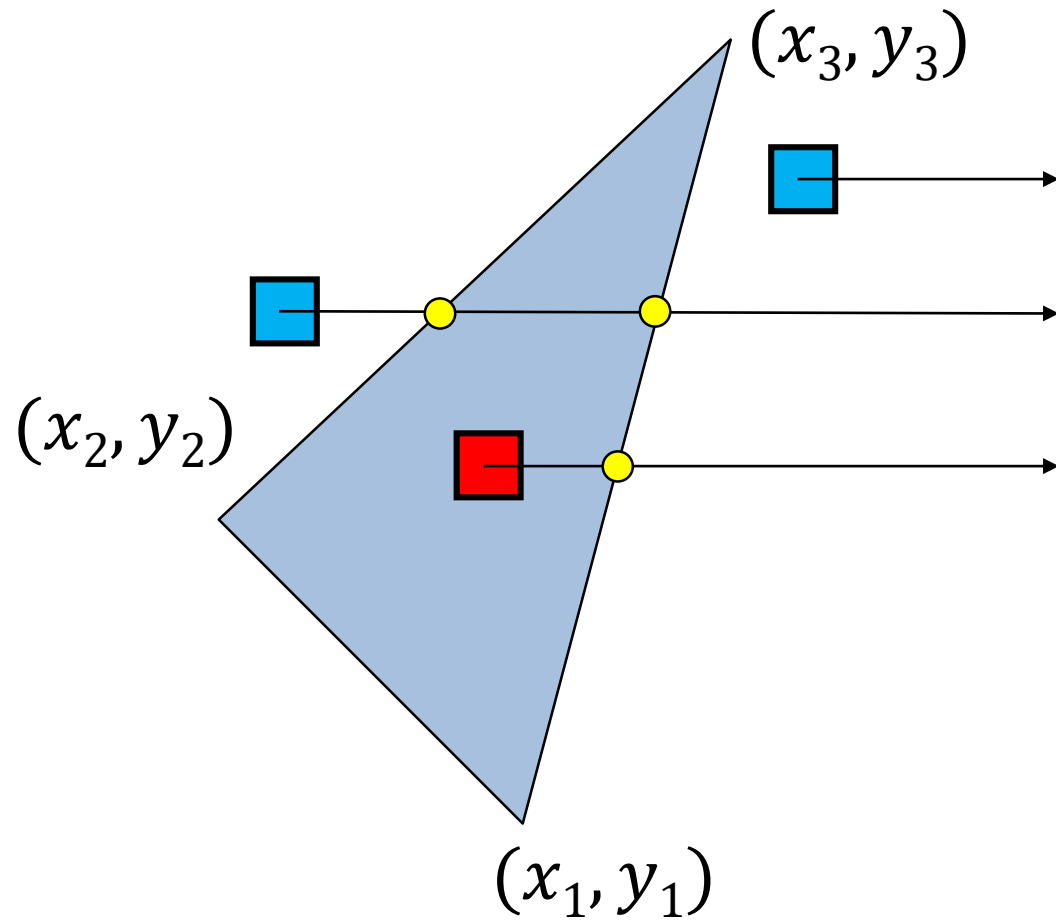
DDA szakaszrajzoló hardver



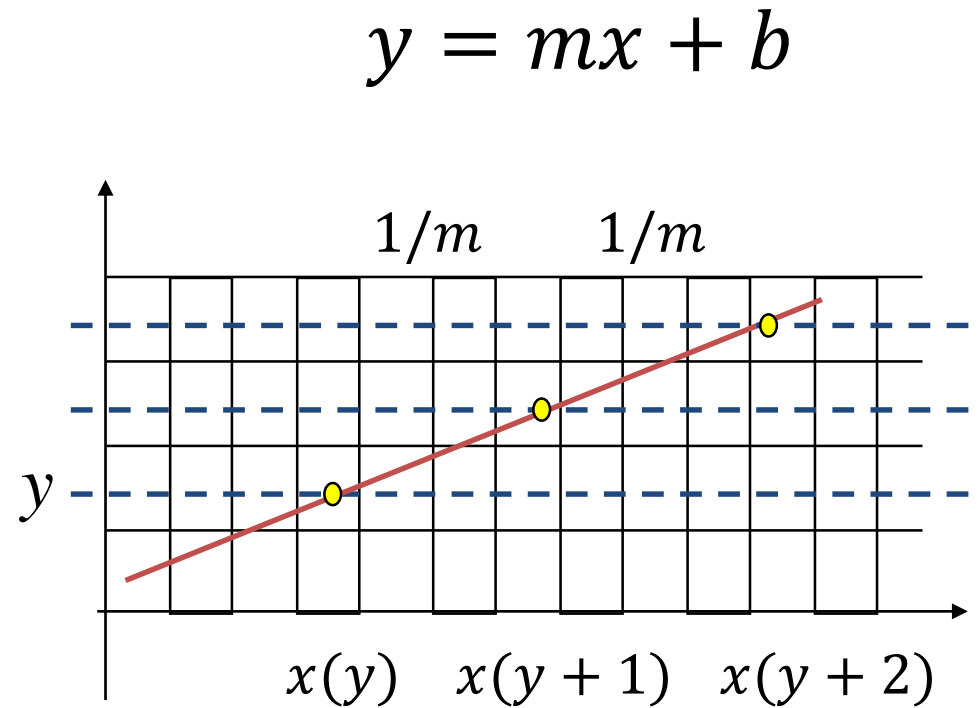
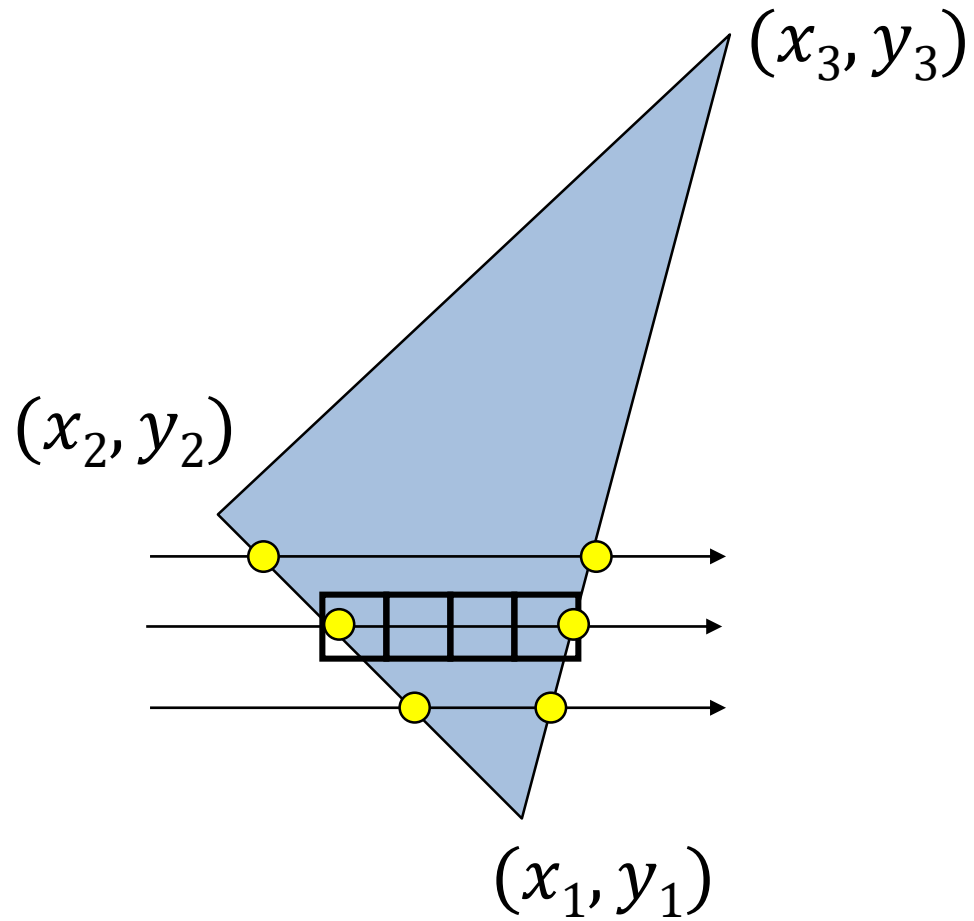
Algoritmikus fotográfia



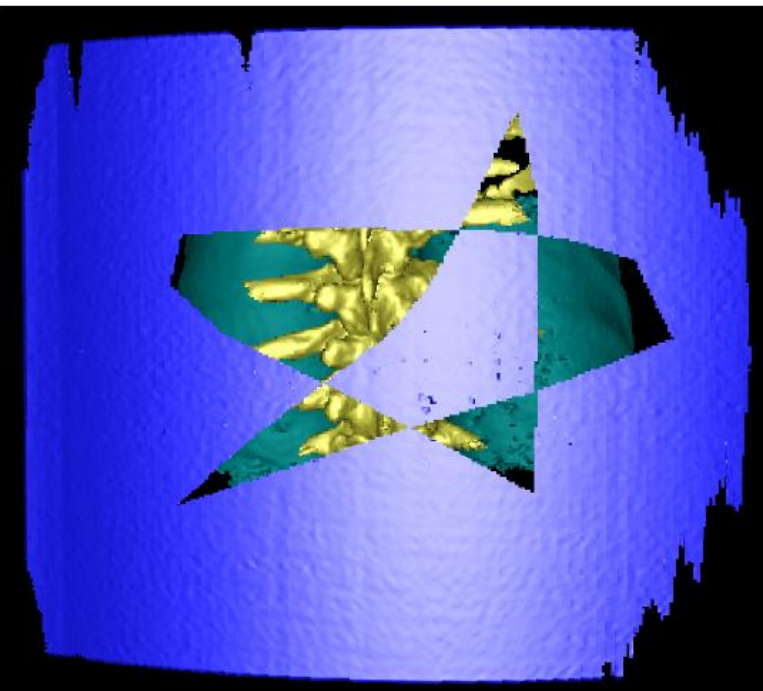
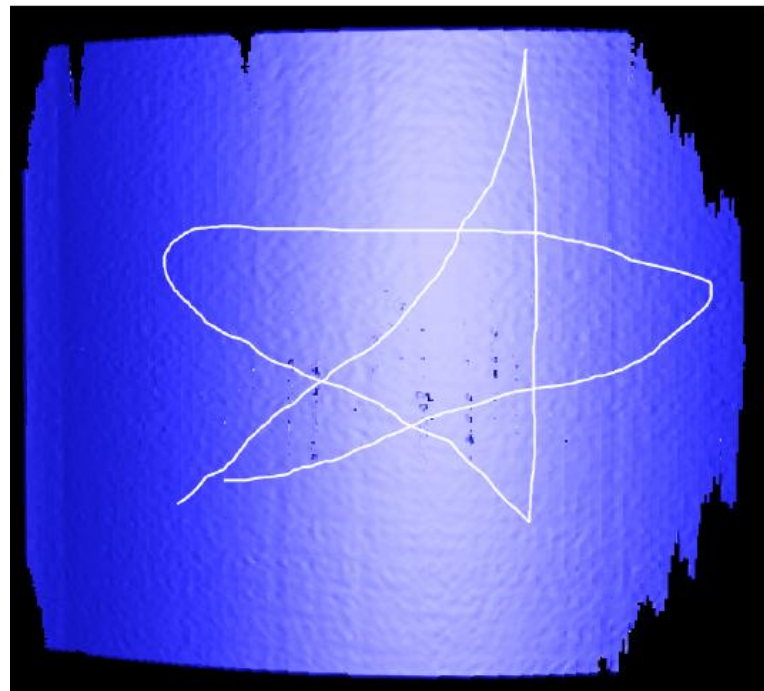
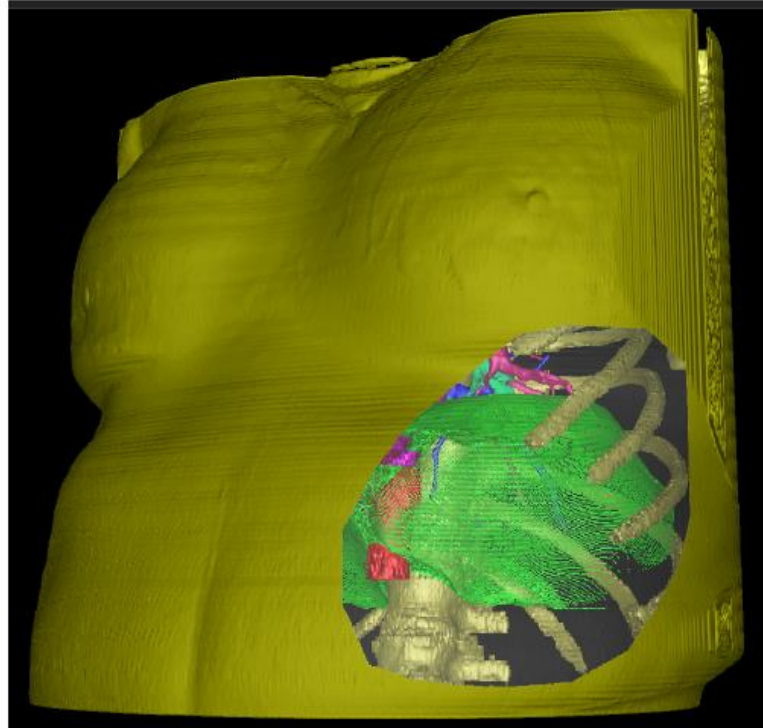
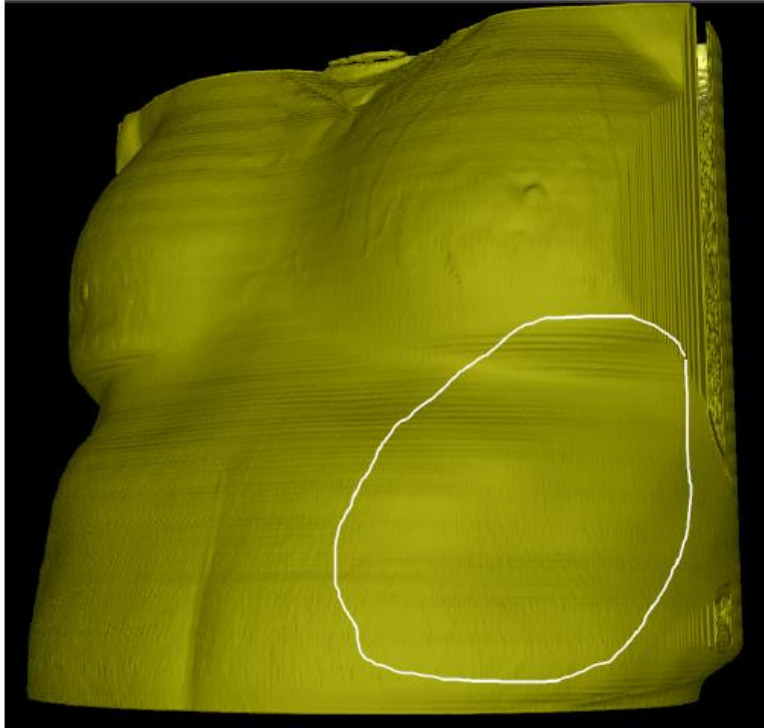
Naív háromszög kitöltés



Koherencia és inkrementális elv

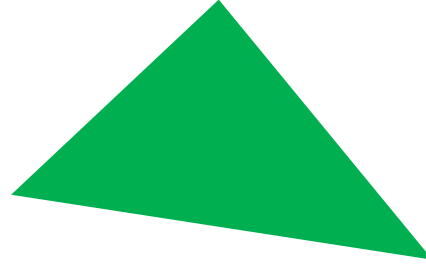


$$x(y + 1) = x(y) + 1/m$$

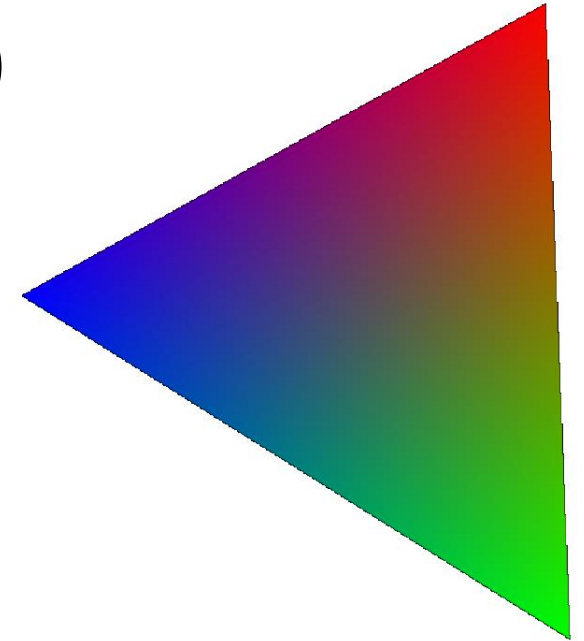
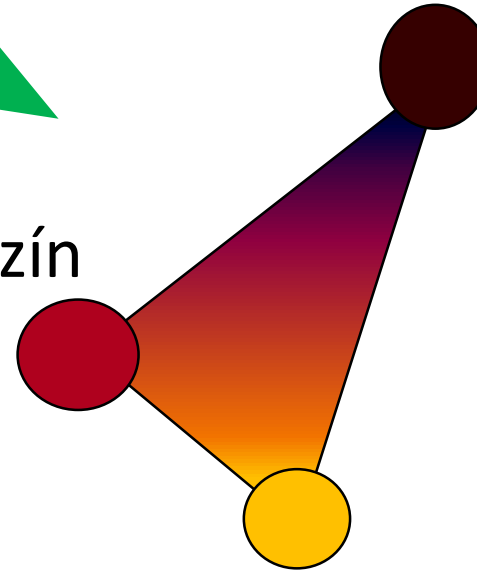


Milyen színű legyen a pixel?

- Uniform az egész objektum



- Csúcspontról tulajdonságokból interpolált szín



- Pl. Textúrázás (2D): Csúcspontról textúrákoordináták interpolációja a pixelekre, majd textúra kiolvasás.

