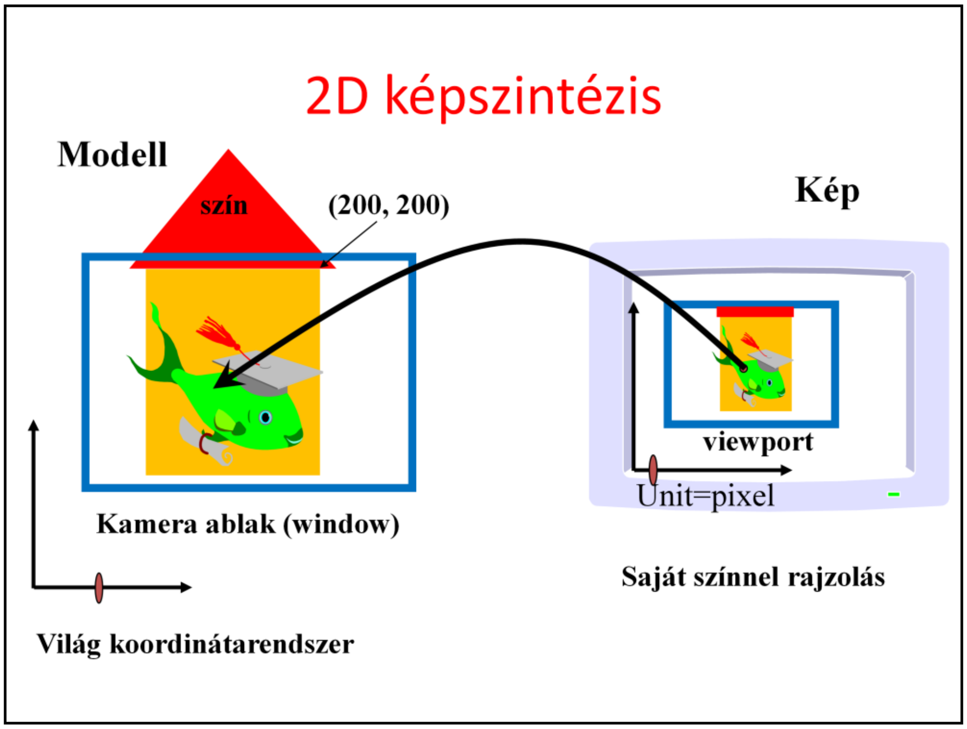
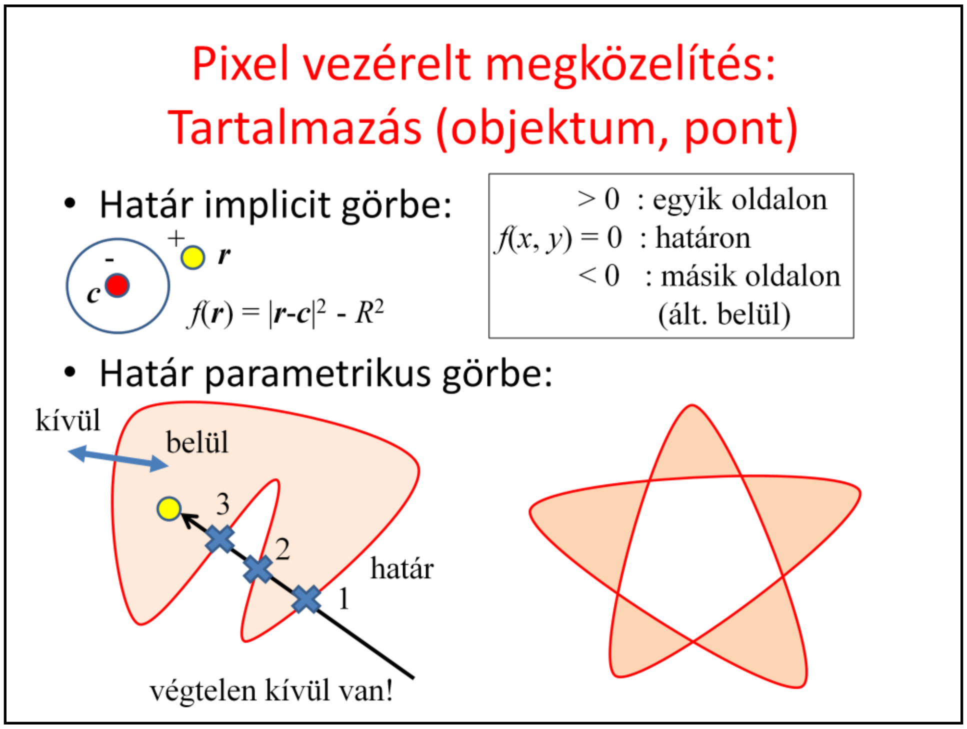
# 2D képszintézis

Szirmay-Kalos László



2D rendering takes a photo of the 2D scene with a virtual camera that selects an axis aligned rectangle from the scene. The photograph is placed into the viewport of the current application window.

A pixel driven solution of this problem would visit the pixels of the viewport one by one, transform the pixel center back to the 2D virtual world, and determine which object contains this point. If several objects contained this point, the one with the highest priority is retained. The pixel is colored with the color of the selected object.



## Pixel vezérelt megközelítés: Tartalmazás (objektum, pont)

- Határ implicit görbe:

$$f(x, y) \begin{cases} > 0 & \text{: egyik oldalon} \\ = 0 & \text{: határon} \\ < 0 & \text{: másik oldalon (ált. belül)} \end{cases}$$

-
+
*r*
*c*
$f(\mathbf{r}) = |\mathbf{r}-\mathbf{c}|^2 - R^2$

- Határ parametrikus görbe:

kívül
belül
3
2
1
határ
végtelen kívül van!

The core of a pixel driven algorithm is containment test, i.e. the determination whether a point is in the set of a 2D object. If the object is defined implicitly, this is usually equivalent to the check of the sign of the implicit equation.

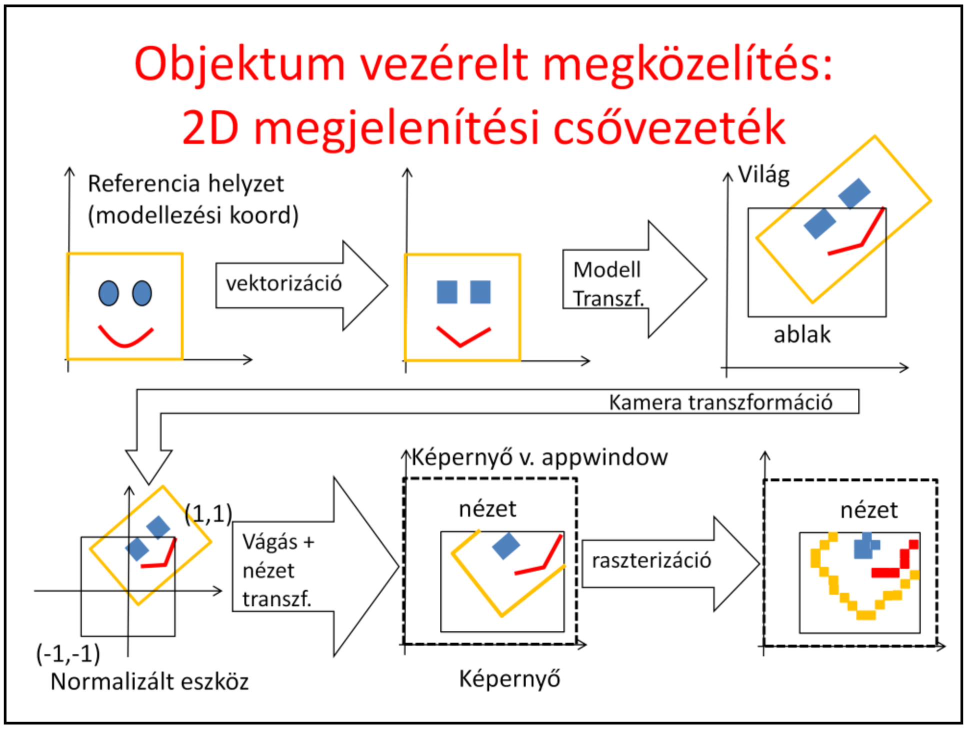
If the boundary of the object is defined, e.g. with parametric curve, whether or not a point is inside should be determined by counting how many times the boundary is crossed until infinity is reached from this point. If this is an odd number the point is inside, otherwise, outside.

## Pixel vezérelt rendering

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

## Pixel vezérelt rendering

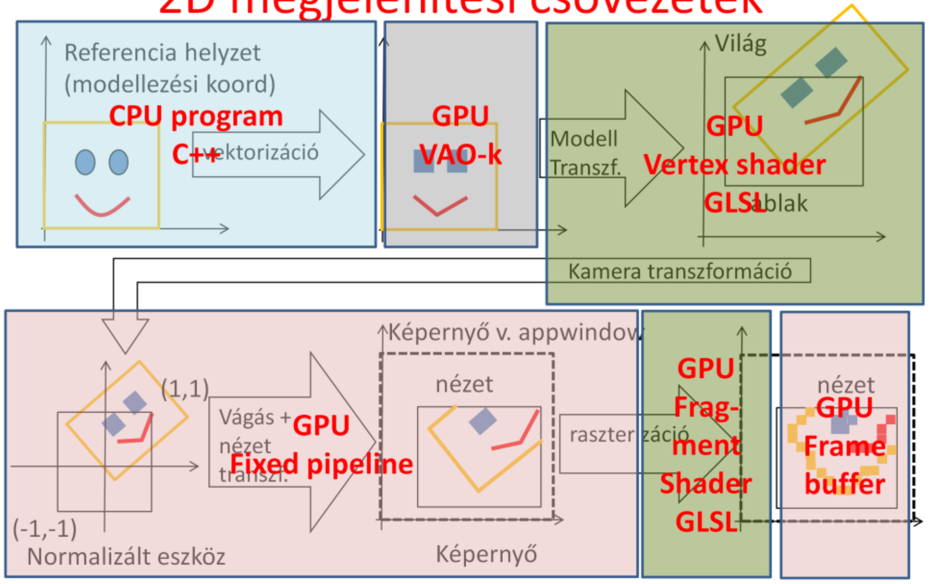
```
class Scene {                 // virtual world
   list<Object *> objs;       // objects with decreasing priority
   Object *picked = nullptr;  // selected for operation
public:
   void Add(Object * o) { objects.push_front(o); picked = o; }
   void Pick(int pX, int pY) {
      vec2 wPoint = Viewport2Window(pX, pY);
      picked = nullptr;
      for(auto o : objs) if (o->In(wPoint)) { picked = o; return; }
   }
   void BringToFront() {
      if (picked) {
         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);
         for(auto o : objs)
            if (o->In(wPoint)) { image[pY][pX] = o->color; break; }
      }
   }
};
```



2D rendering is a sequence, called pipeline, of computation steps. We start with the objects defined in their reference state, which can include points, parametric or implicit curves, 2D regions with curve boundaries. As we shall transform these objects, we they are vectorized, so curves are approximated by polylines and regions by polygons. The rendering pipeline thus processes only point, line (polyline) and triangle (polygon) primitives.

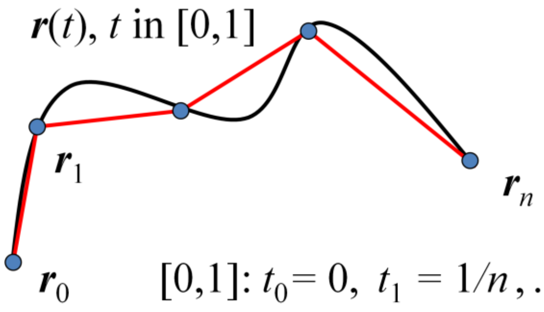
Modeling transformation places the object in world coordinates. This typically involves scaling, rotation and translation to set the size, orientation and the position of the object. In world, objects meet each other and also the 2D camera, which is the window AABB (axis aligned bounding box or rectangle). We wish to see the content of the window in the picture on the screen, called viewport. Thus, screen projection transforms the world in a way that the window rectangle is mapped onto the viewport rectangle. This can be done in a single step, or in two steps when first the window is transformed to a square of corners (-1,-1) and (1,1) and then from here to the physical screen. Clipping removes those objects parts that are outside of the camera window, or alternatively outside of the viewport in screen, or outside of  the square of corners (-1,-1) and (1,1) in normalized device space. The advantage of normalized device space becomes obvious now. Clipping here is independent of the resolution and of the window, so can be easily implemented in a fix hardware. Having transformed primitives onto the screen, where the unit is the pixel, they are rasterized. Algorithms find those sets of pixels which can provide the illusion of a line segment or a polygon.

## Objektum vezérelt megközelítés: 2D megjelenítési csővezeték



Vectorization is executed by our program running on the CPU since the GPU expects lines and triangles. The other steps are mapped onto the GPU stages.

## Vektorizáció (CPU)

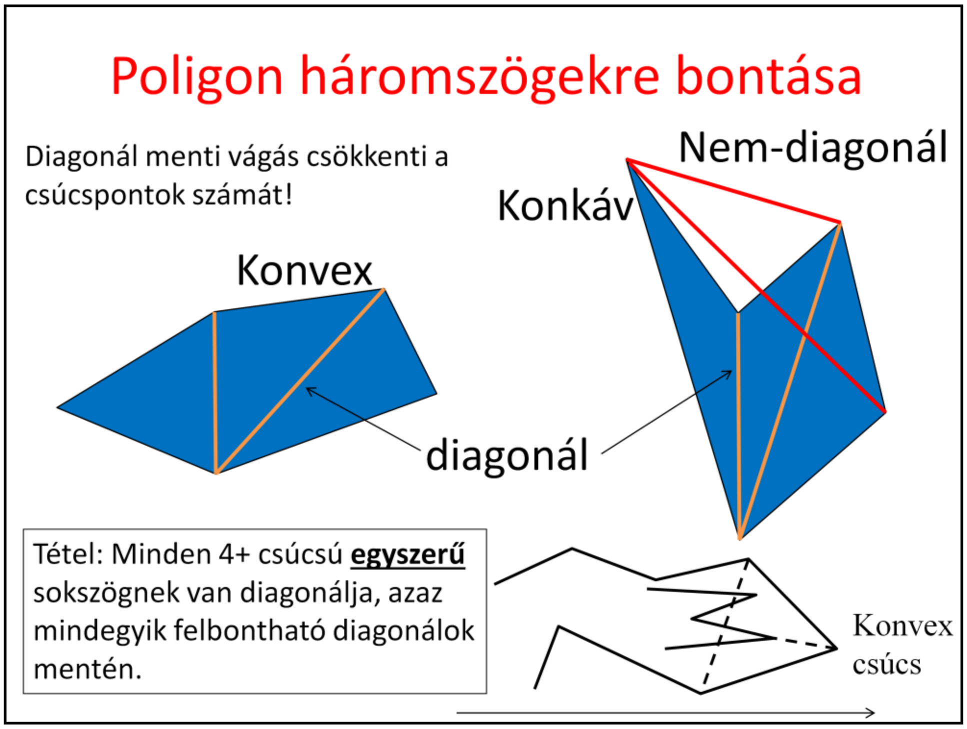


$[0,1]$: $t_0 = 0$, $t_1 = 1/n$ , ... , $t_n = 1$

$r_0 = r(t_0)$, $r_1 = r(t_0)$, … , $r_n = r(t_n)$

Vectorization is trivial for parametric curves. The parametric range is decomposed and increasing sample values are substituted into the equation of the curve, resulting in a sequence of points on the curve. Introducing a line segment between each subsequent pair of points, the curve is approximated by line segments.

If the curve is closed, using the same strategy, a polygon approximation of the region can be found.



## Poligon háromszögekre bontása

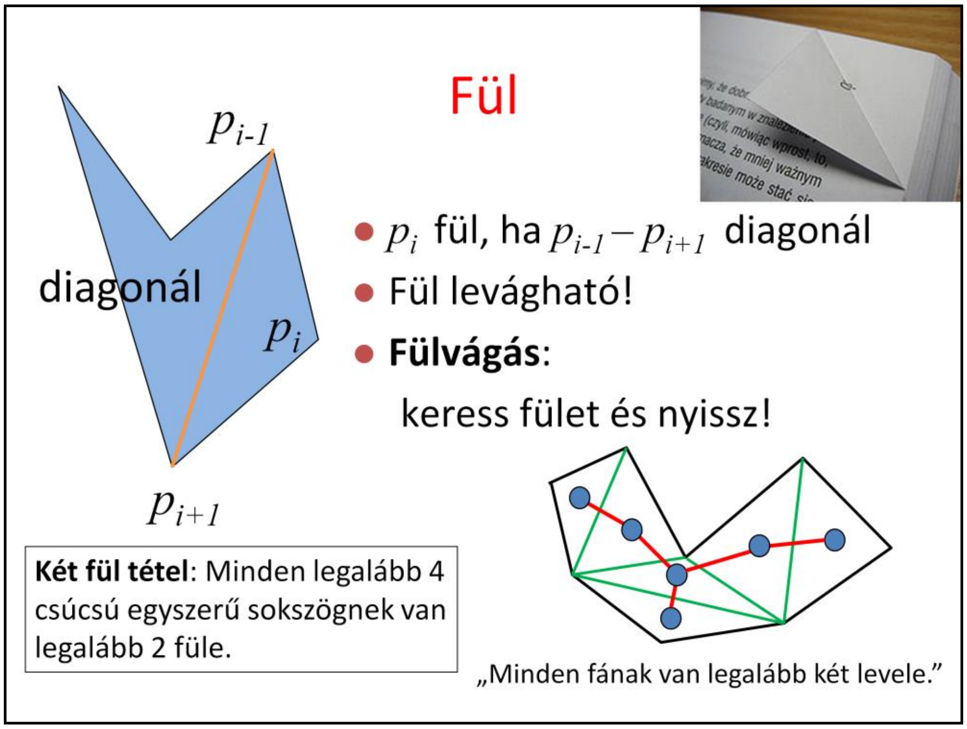
Diagonál menti vágás csökkenti a csúcspontok számát!

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

Polylines are often further decomposed to line segments and polygons to triangles. Such a decomposition has the advantage that the resulting data element has constant size (a line segment has 2 vertices a triangle has 3), and processing algorithms will be uniform and independent of the size of the data element.

Polylines can be easily decomposed to line segments. However, polygons are not so simple to decompose to triangles unless the polygon is convex. A polygon is broken down to smaller polygons and eventually to triangles by cutting them along diagonals. A diagonal is a line segment connecting two non-neighboring vertices, that is fully contained by the polygon. If the polygon is convex, then any line segment connecting two non-neighboring vertices is fully contained by the polygon (this is the definition of convexity), thus all of them are diagonals. This is not the case for concave polygons, when line segments connecting vertices can intersect edges or can fully be outside of the polygon.

The good news is that all polygons, even concave ones, have diagonals, so they can be broken to triangles by diagonals (prove it). An even better news is that any polygon of at least 4 vertices has special diagonals, that allow exactly one triangle to be cut.



## Fül

- $p_i$ fül, ha $p_{i-1} - p_{i+1}$ diagonál
- Fül levágható!
- **Fülvágás**:

  keress fület és nyissz!

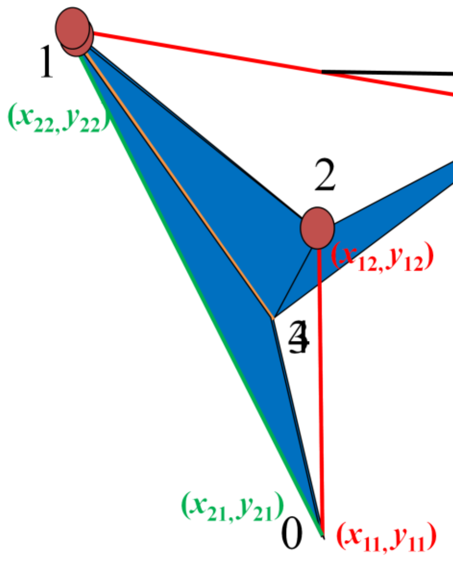
**Két fül tétel**: Minden legalább 4 csúcsú egyszerű sokszögnek van legalább 2 füle.

„Minden fának van legalább két levele.”

A vertex is an ear if the line segment between its previous and next vertices is a diagonal. According to the two ears theorem, every polygon of at least 4 vertices has at least two ears. So triangle decomposition should just search for ears and cut them until a single triangle remains.

The proof of the two ears theorem is based on the recognition that any polygon can be broken down to triangles by diagonals. Let us start with one possible decomposition, and consider triangles as nodes of a graph, and add edges to this graph where two triangles share a diagonal. This graph is a **tree** since it is connected (the polygon is a single piece) and cutting every edge, the graph falls apart, i.e. there is no circle in it. By induction, it is easy to prove that every tree of at least 2 nodes has at least two leaves, which correspond to two ears.

## Fülvágó algoritmus



Szakasz-szakasz metszés:

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

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

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

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

$x_1(t_1)= x_2(t_2)$

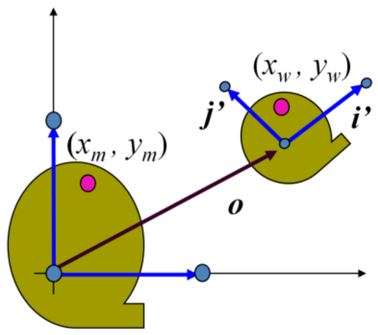
$y_1(t_1)= y_2(t_2)$

$?\ t_1,t_2\in (0,1)$

For every step, we check whether or not a vertex is an ear. The line segment of its previous and next vertices is tested whether it is a diagonal. This is done by checking whether the line segment intersects any other edge (if it does, it is not a diagonal). If there is no intersection, we should determine whether the line segment is fully outside. Selecting an arbitrary inner point, e.g. the middle, we check whether this point is inside the polygon. By definition, a point is inside if traveling from this point to infinity, the polygon boundary is intersected odd number of times.

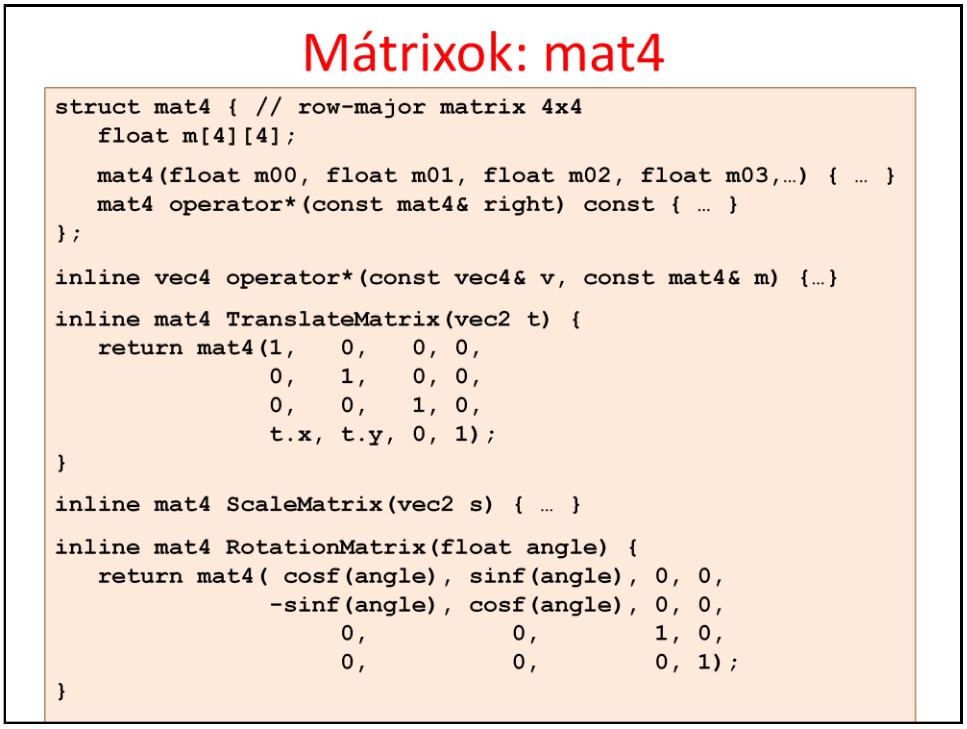
## Modellezési transzformáció
(Mátrixokat a CPU-n számítjuk, a transzformációt a GPU hajtja végre)

$$[x_w, y_w, 1] = [x_m, y_m, 1]\cdot\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ p_x & p_y & 1 \end{bmatrix}$$



$$(x_w, y_w) = \boldsymbol{o} + x_m\,\boldsymbol{i'} + y_m\,\boldsymbol{j'}$$

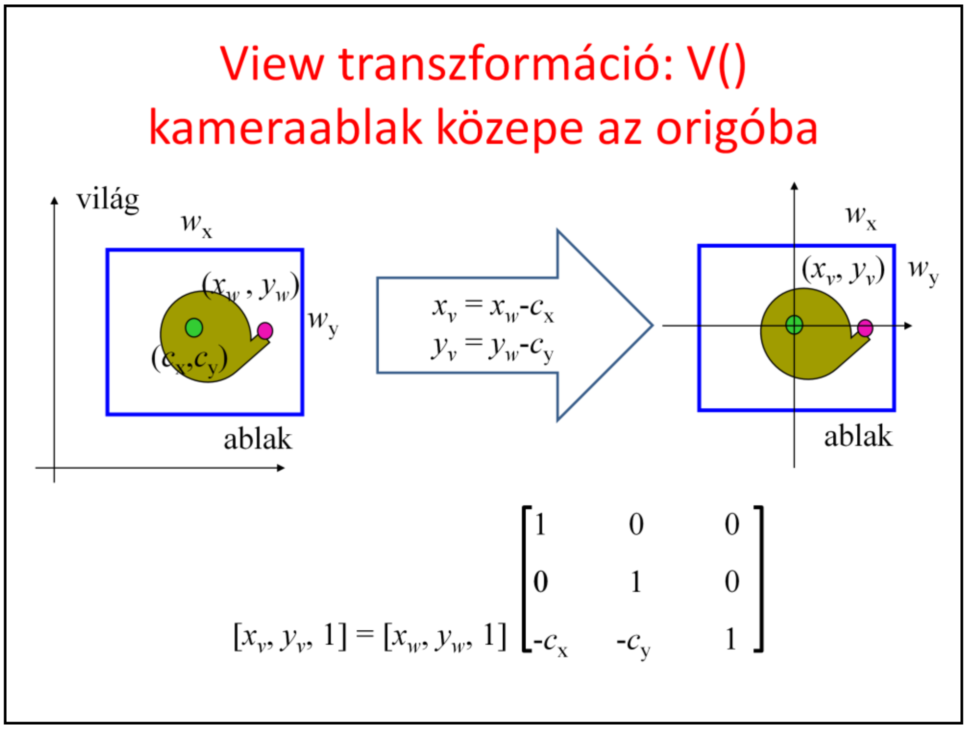
The first relevant step of rendering is placing the reference state primitives in world, typically scaling, rotating and finally translating its vertices. Recall that it is enough to execute these transformations to vertices, because points, lines and polygons are preserved by homogeneous linear transformations. These are affine transformations, and the resulting modeling transformation matrix will also be an affine transformation. If the third column is 0,0,1, then other matrix elements have an intuitive meaning, they specify what happens with basis vector i, basis vector j, and the origin itself.



## Mátrixok: mat4

```
struct mat4 { // row-major matrix 4x4
   float m[4][4];

   mat4(float m00, float m01, float m02, float m03,…) { … }
   mat4 operator*(const mat4& right) const { … }
};

inline vec4 operator*(const vec4& v, const mat4& m) {…}

inline mat4 TranslateMatrix(vec2 t) {
   return mat4(1,   0,   0, 0,
               0,   1,   0, 0,
               0,   0,   1, 0,
               t.x, t.y, 0, 1);
}

inline mat4 ScaleMatrix(vec2 s) { … }

inline mat4 RotationMatrix(float angle) {
   return mat4( cosf(angle), sinf(angle), 0, 0,
               -sinf(angle), cosf(angle), 0, 0,
                    0,           0,       1, 0,
                    0,           0,       0, 1);
}
```

To do transformations on the CPU, we use the following mat4 class. This type is built in GLSL as well.



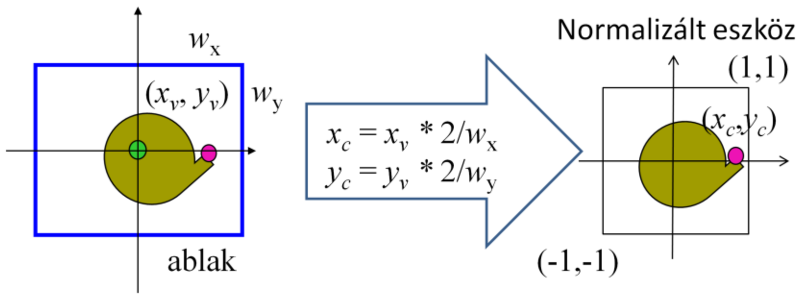
## View transzformáció: V()
## kameraablak közepe az origóba

$$[x_v, y_v, 1] = [x_w, y_w, 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -c_x & -c_y & 1 \end{bmatrix}$$

Screen projection maps the window rectangle, which is the camera in 2D, onto the viewport rectangle, which can be imagined as the photograph. This simple projection is usually executed in two steps, first transforming the window onto a normalized square, and then transforming the square to the viewport.

Transforming the window to a origin centered square of corners (-1,-1) and (1,1) is a sequence of two transformations: a translation that moves the center of the camera window to the origin; a scaling that modifies the window width and height to 2. These are affine transformations that can also be given as a matrix.

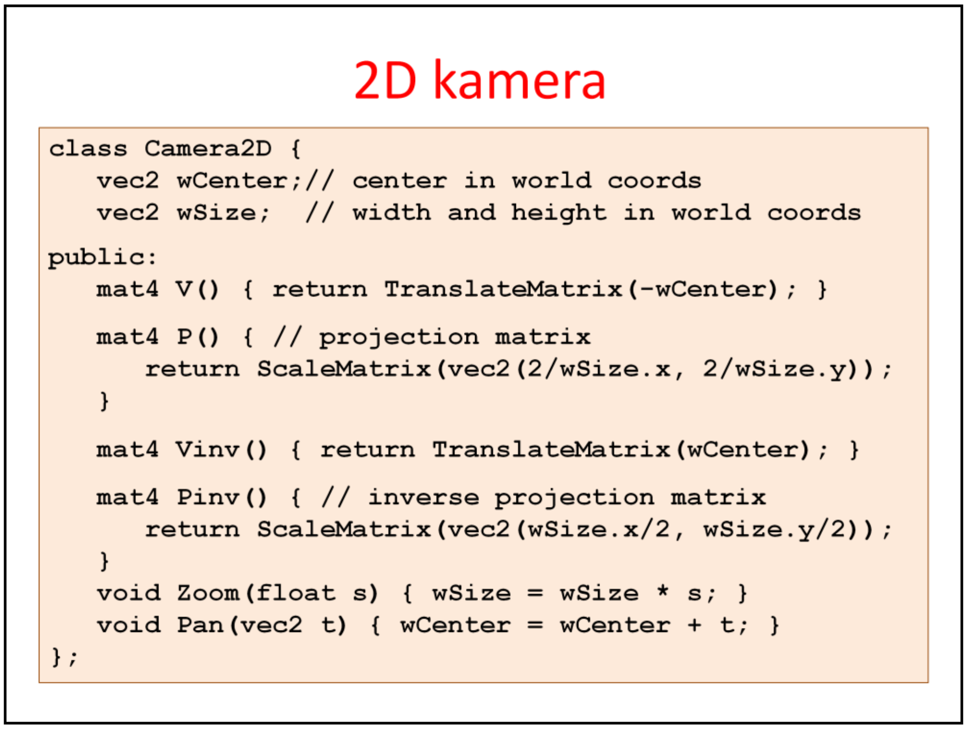
## Projekció: P()
## kameraablak a (-1, -1)-(1, 1) négyzetbe



$$x_c = x_v * 2/w_x$$
$$y_c = y_v * 2/w_y$$

$$[x_c, y_c, 1] = [x_v, y_v, 1] \begin{bmatrix} 2/w_x & 0 & 0 \\ 0 & 2/w_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

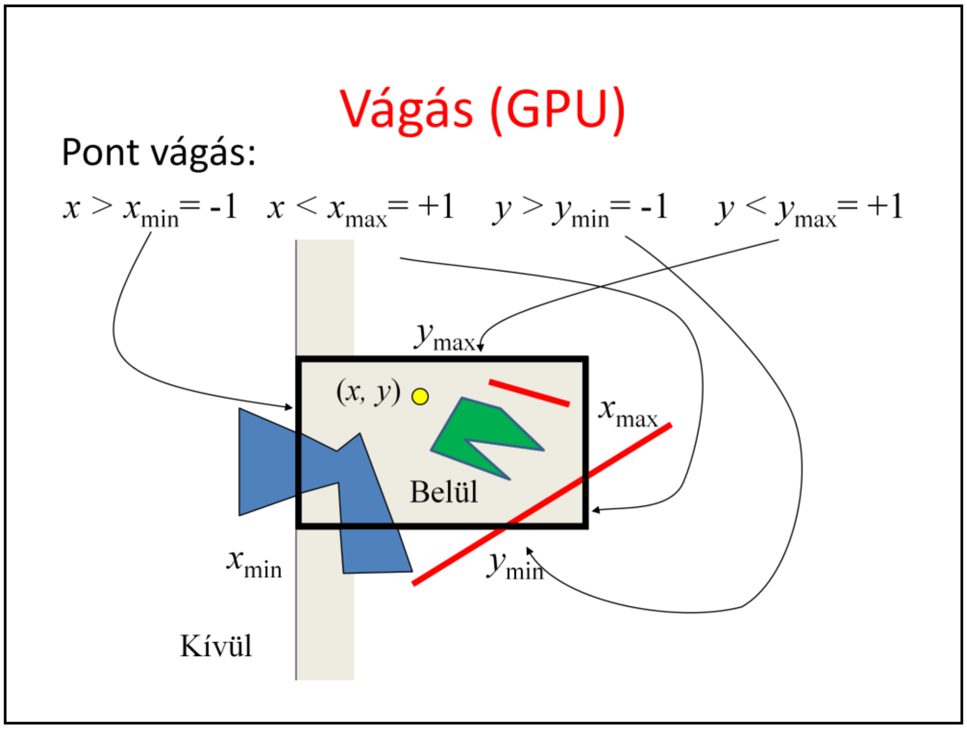
Projection is a scaling that makes the size of the window a square of edge length 2 while keeping the center in the origin. It might seem an over complication, but in 3D we use the same steps of view and projection transformations and they will be more complicated. The other advantage of separating view and projection transformations is that it is easy to invert the transformation in this form.



## 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() { 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; }
};
```

A 2D camera is thus represented by the center and the size of the camera window and is associated with view and projection transformations, as well as their inverse. Zoom and pan are just the modifications of the size and center, respectively.



## Vágás (GPU)

Pont vágás:

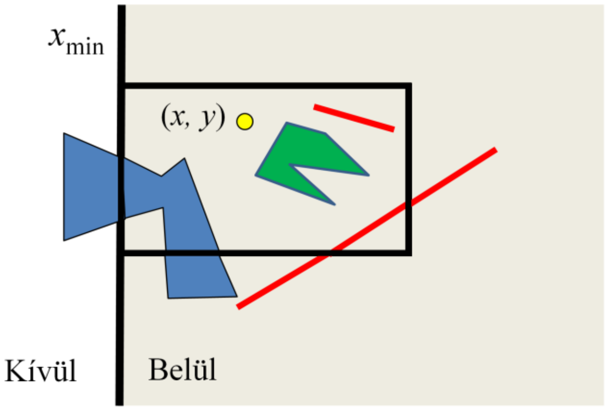
$x > x_{min} = -1 \quad x < x_{max} = +1 \quad y > y_{min} = -1 \quad y < y_{max} = +1$

Clipping is executed usually in normalized device space where x,y must be between -1 and 1. To be general, we denote the limits by xmin, xmax…

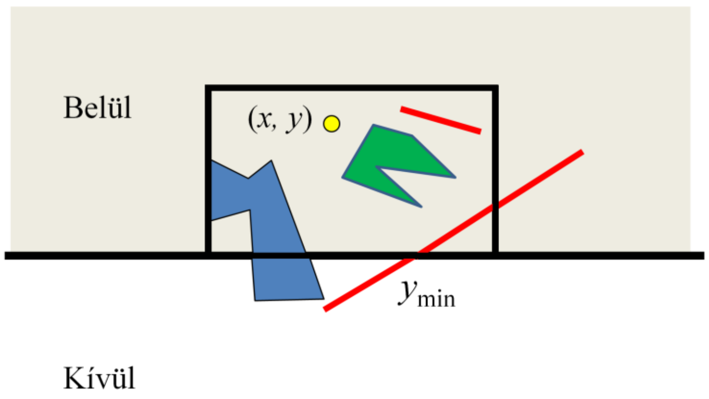
A point is preserved by clipping if it satisfies $x > x_{min} = -1$, $x < x_{max} = +1$, $y > y_{min} = -1$, $y < y_{max} = +1$. Let us realize that each of these inequalities is a clipping condition for a half-plane. A point is inside the clipping rectangle if it is inside all four half planes since the clipping rectangle is the intersection of the half planes.

This concept is very useful when line segments or polygons are clipped since testing whether or not the two endpoints of line segment or vertices of a polygon are outside the clipping rectangle cannot help to decide whether there is an inner part of the primitive.

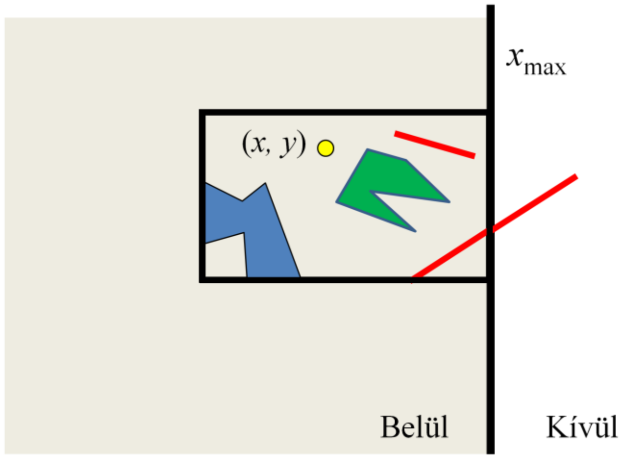
## Vágás



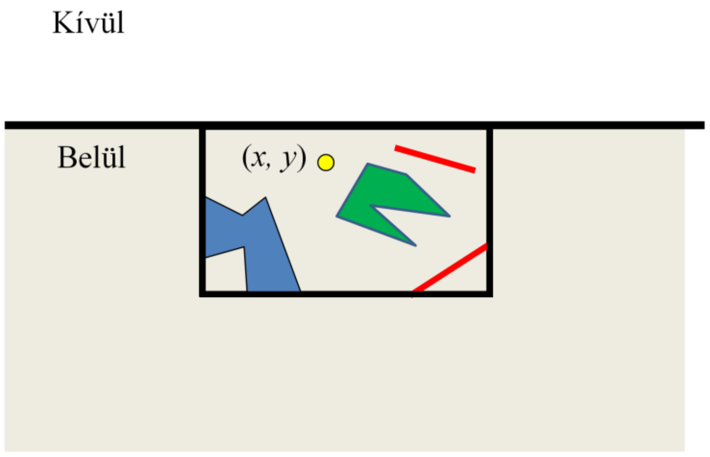
## Vágás

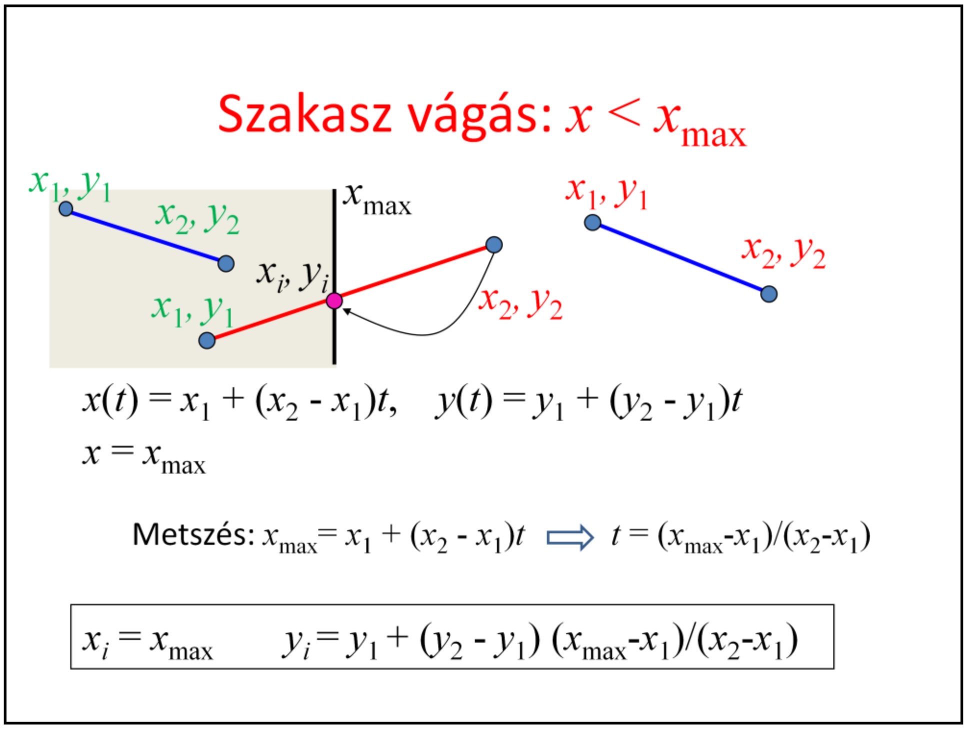


## Vágás

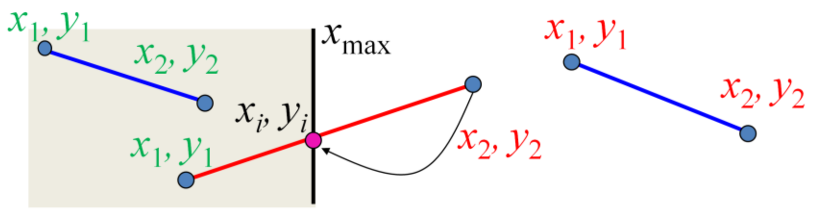


## Vágás





## Szakasz vágás: $x < x_{max}$

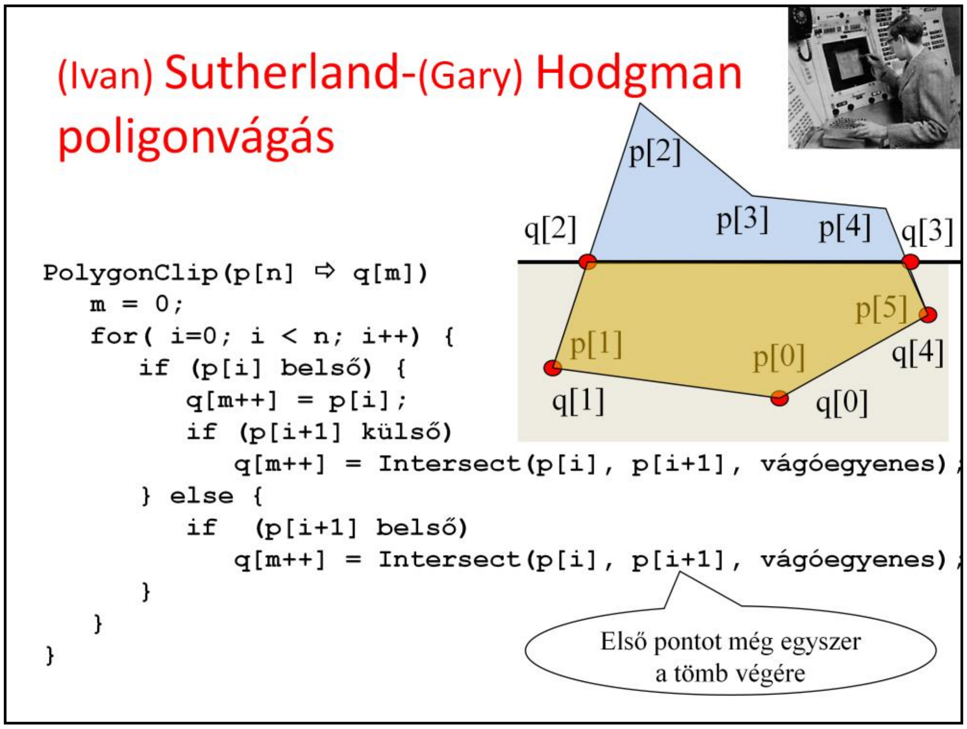


$$x(t) = x_1 + (x_2 - x_1)t, \quad y(t) = y_1 + (y_2 - y_1)t$$
$$x = x_{max}$$

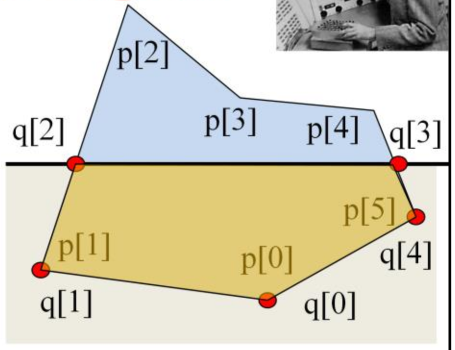
Metszés: $x_{max} = x_1 + (x_2 - x_1)t \Rightarrow t = (x_{max}-x_1)/(x_2-x_1)$

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

Let us consider a line segment and clipping on a single half plane. If both endpoints are inside, then the complete line segment is inside **since the inner region, the half plane, is convex**. If both endpoints are outside, then the line segment is completely outside, **since the outer region is also convex**. If one endpoint is inside while the other is outside, then the intersection of the line segment and the clipping line is calculated, and the outer point is replaced by the intersection.



## (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)
      }
   }
}
```

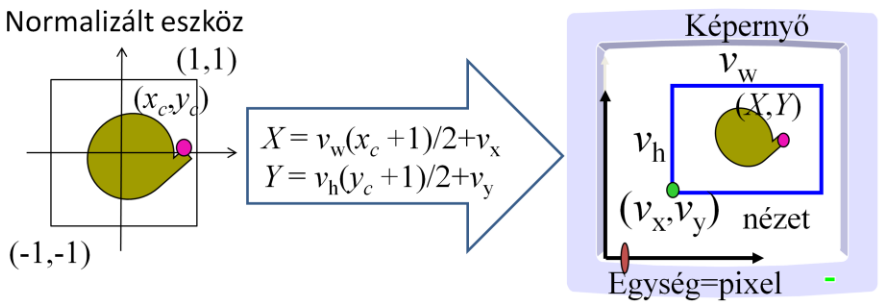
Első pontot még egyszer a tömb végére

Polygon clipping is traced back to line clipping. We consider the edges of the polygon one-by-one. If both endpoints are in, the edge will also be part of the clipped polygon. If both of them are out, the edge is ignored. If one is in and the other is out, the inner part of the segment is computed and added as an edge of the clipped polygon.

The input of this implementation is an array of vertices p and number of points n. The output is an array of vertices q and number of vertices m.

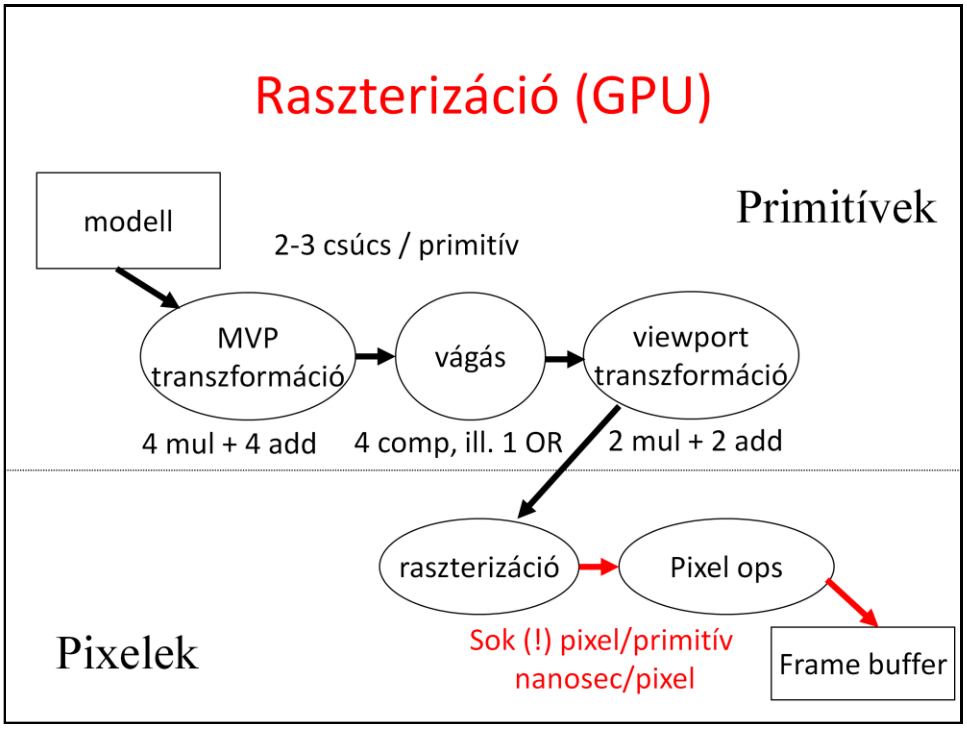
Usually, we can assume that the ith edge has endpoints p[i] and p[i+1]. However, the last edge is an exception since its endpoints are p[n-1] and p[0].

## Viewport transzformáció: Normalizáltból képernyő koordinátákba (GPU)

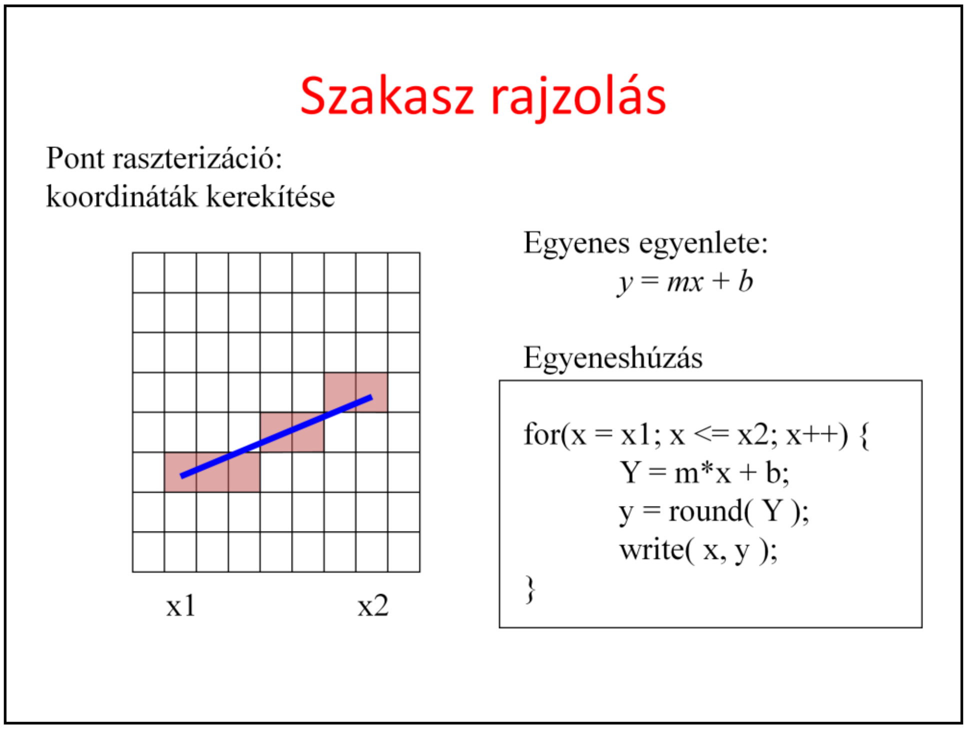


```
glViewport(vx, vy, vw, vh);
```

The transformation from normalized device space to screen space is a scaling and a translation. This transformation is executed by the fixed function stage of the GPU. With **glViewport**, the parameters of the transformation can be supplied.

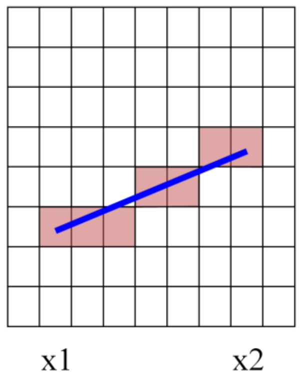


Before starting the discussion of rasterization it is worth looking at the pipeline and realizing that rasterization uses a different data element, the pixel, while phases discussed so far work with geometric primitives. A primitive may be converted to many pixels, thus the performance requirements become crucial at this stage. In order to maintain real-time frame rates, the process should output a new pixel in every few nanoseconds. It means that only those algorithms are acceptable that can deliver such performance.



## Szakasz rajzolás

Pont raszterizáció:
koordináták kerekítése



Egyenes egyenlete:

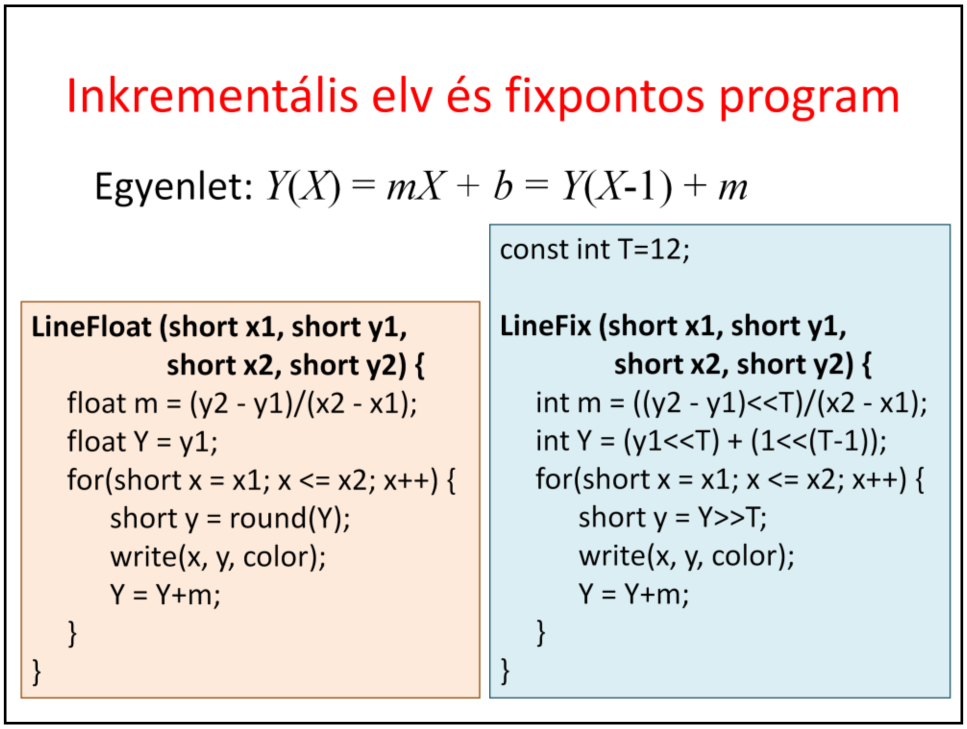
$y = mx + b$

Egyeneshúzás

```
for(x = x1; x <= x2; x++) {
        Y = m*x + b;
        y = round( Y );
        write( x, y );
}
```

Line drawing should provide the illusion of a line segment by coloring a few pixels. A line is thin and connected, so pixels should touch each other, should not cover unnecessary wide area and should be close to the geometric line. If the slope of the line is moderate, i.e. x is the faster growing coordinate, then it means that in every column exactly one pixel should be drawn (connected but thin), that one where the pixel center is closest to the geometric line. The line drawing algorithm iterates on the columns, and in a single column it finds the coordinate of the geometric line and finally obtains the closest pixel, which is drawn.

This works, but a floating point multiplication, addition and a rounding operation is needed in a single cycle, which is too much for a few nanoseconds. So we modify this algorithm preserving its functionality but getting rid of the complicated operations.



## Inkrementális elv és fixpontos program

Egyenlet: $Y(X) = mX + b = Y(X\text{-}1) + m$

```
const int T=12;

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

```
LineFloat (short x1, short y1,
            short x2, short y2) {
    float m = (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;
    }
}
```

The algorithm transformation is based on the incremental concept, which realizes that a linear function (the explicit equation of the line) is evaluated for an incremented X coordinate. So when X is taken, we already have the Y coordinate for X-1. The fact is that it is easier to compute Y(X) from its previous value than from X. The increment is m, the slope of the line, thus a single addition is enough to evaluate the line equation. This single addition can be made faster if we used fixed point number representation and not floating point format. As these numbers are non integers (m is less than 1), the fixed point representation should use fractional bits as well. It means that an integer stores the Tth power of 2 multiple of the non-integer value. Such values can be added as two integers.

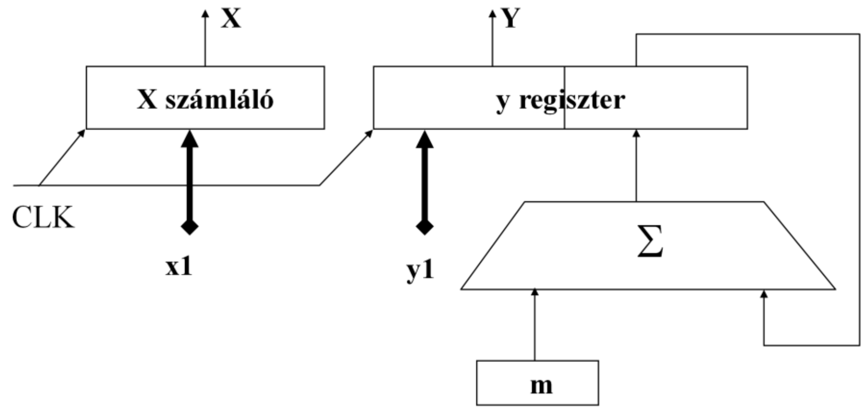
The number of fractional bits can be determined from the requirement that even the longest iteration must be correct. If the number of fractional bits is T, the error caused by the finite fractional part is 2^{-T} in a single addition. If errors are accumulated, the total error in the worst case is N 2^{-T} where N is the number of additions. N is the linear resolution of the screen, e.g. 1024. In screen space the unit is the pixel, so the line will be correctly drawn if the total error is less than 1. It means that T=10, for example, satisfies all requirements.

The line drawing algorithm based on the incremental concepts is as follows. First the slope of the line is computed. The y value is set according to the end point. This y stores the precise location of the line for a given x, so it is non integer. In a for cycle, the closest integer is found, the pixel is written, and – according to the incremental concept – the new y values for the next column is obtained by a single addition.

Rounding can be replaced by simple truncation if 0.5 is added to the y value.

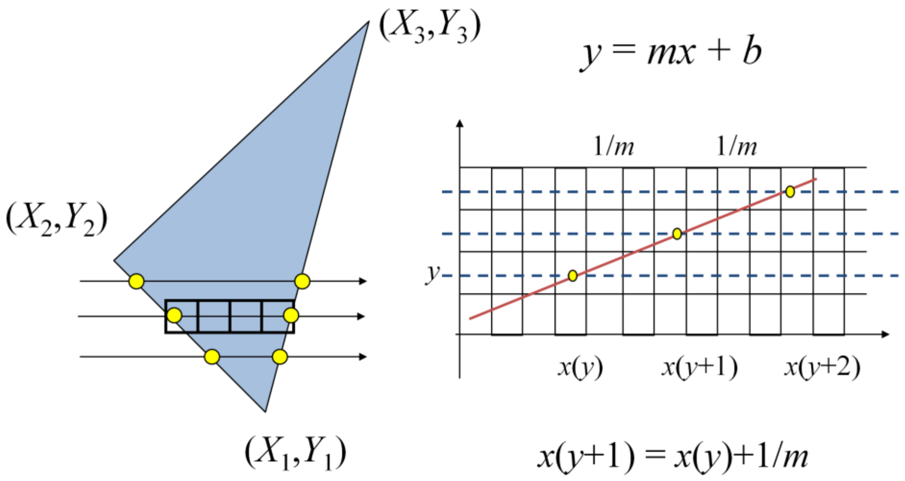
If fixed point representation is used, we shift m and y by T number of bits and rounding ignores the low T bits.

## DDA szakaszrajzoló hardver



This algorithm can be implemented in hardware with a simple counter that generates increasing x values for every clock cycle. For y we use a register that stores both its fractional and integer parts. The y coordinate is incremented by m for every clock cycle.

## Háromszög kitöltés



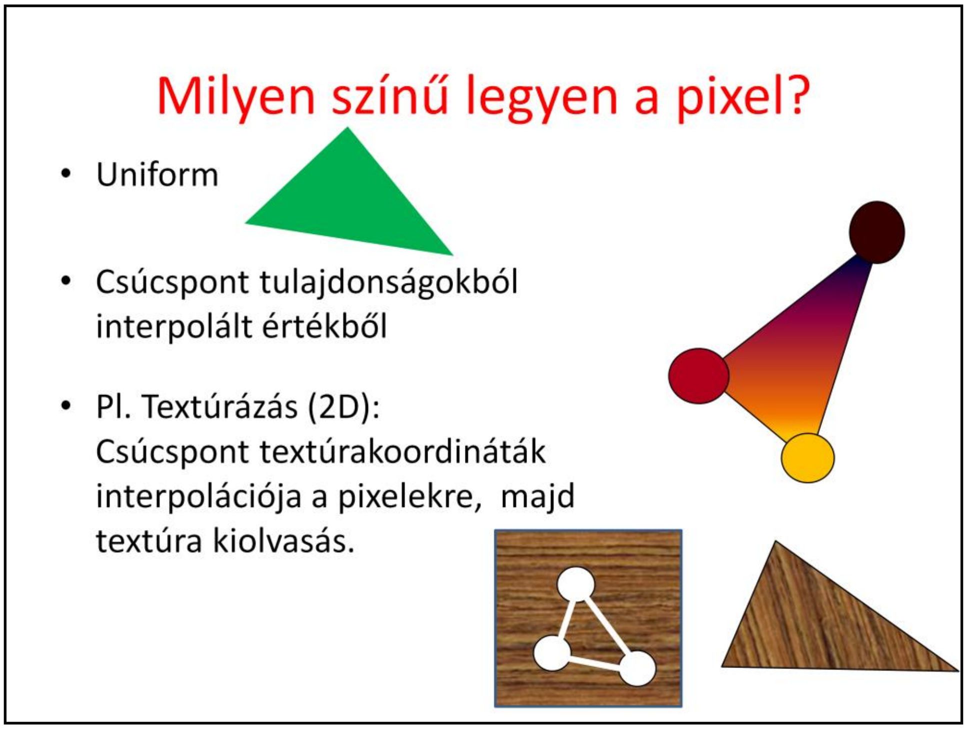
$$y = mx + b$$

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

For triangle rasterization, we need to find those pixels that are inside the triangle and color them. The search is done along horizontal lines of constant y coordinate. These lines are called scan lines and rasterization as scan conversion. For a single scan line, the triangle edges are intersected with the scan line and pixels are drawn between the minimum and maximum x coordinates.

The incremental principle can also be applied to determine scan-line and edge intersections. Note that while y is incremented by 1, the x coordinate of the intersection grows with the inverse slope of the line, which is constant for the whole edge, and thus should be computed only once.

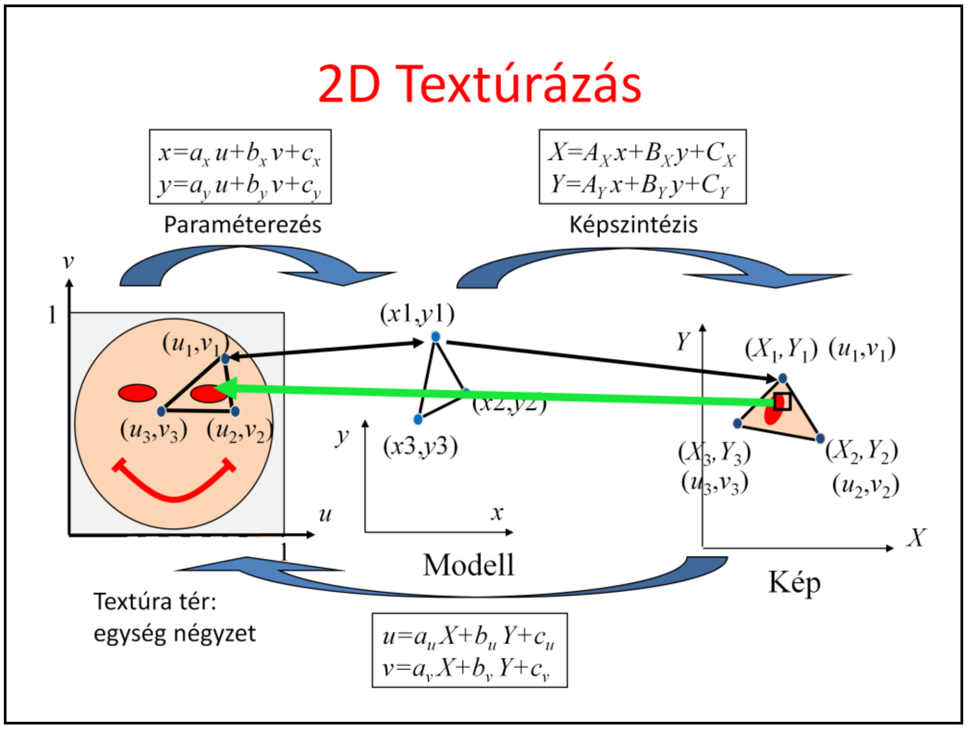
Again, we have an algorithm that uses just increments and integer additions.



## Milyen színű legyen a pixel?

- Uniform
- Csúcspont tulajdonságokból interpolált értékből
- Pl. Textúrázás (2D): Csúcspont textúrakoordináták interpolációja a pixelekre, majd textúra kiolvasás.

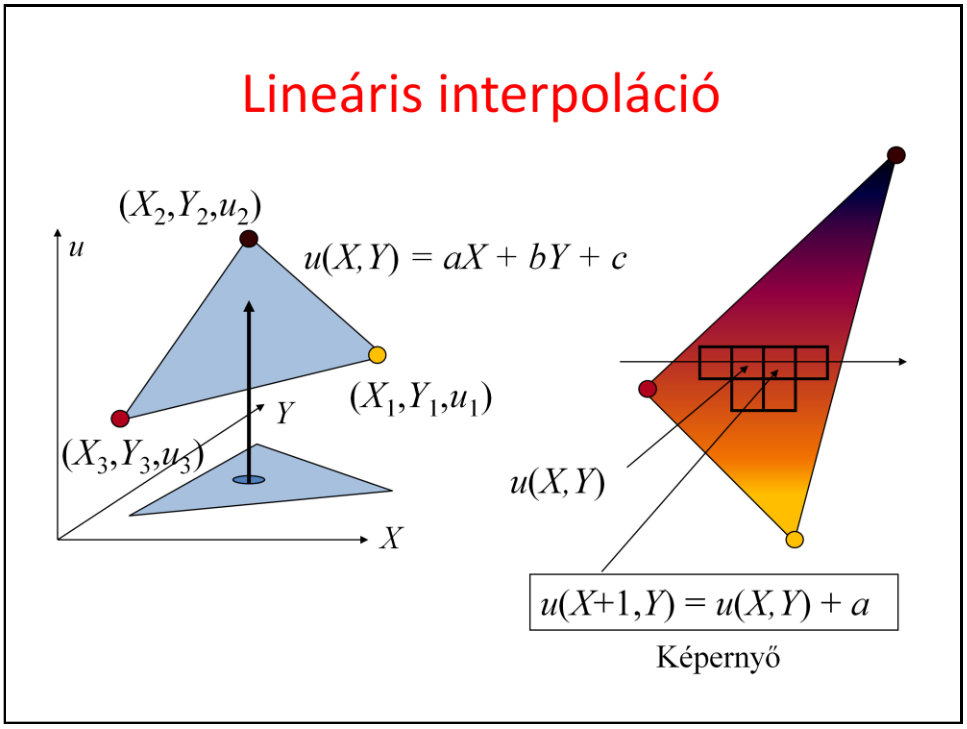
Rasterization has selected those pixels that belong to an object. The only remaining task is to obtain a color and write into the selected pixel. There are different options to find the color. It can be uniform for all points. Colors or any property from which the color is computed can be assigned to the vertices. Then the colors of internal pixels are generated by interpolation. Finally, we can also define the object vertices on a pattern image, called texture. The pattern is then mapped or wallpapered onto the object.



2D texture mapping can be imagined as wallpapering. We have a wallpaper that defines the image or the function of a given material property. This image is defined in texture space as a unit rectangle. The wallpapering process will cover the 3D surface with this image. This usually implies the distortion of the texture image. To execute texturing, we have to find a correspondence between the region and the 2D texture space. After tessellation, the region is a triangle mesh, so for each triangle, we have to identify a 2D texture space triangle, which will be painted onto the model triangle. The definition of this correspondence is called **parameterization.**

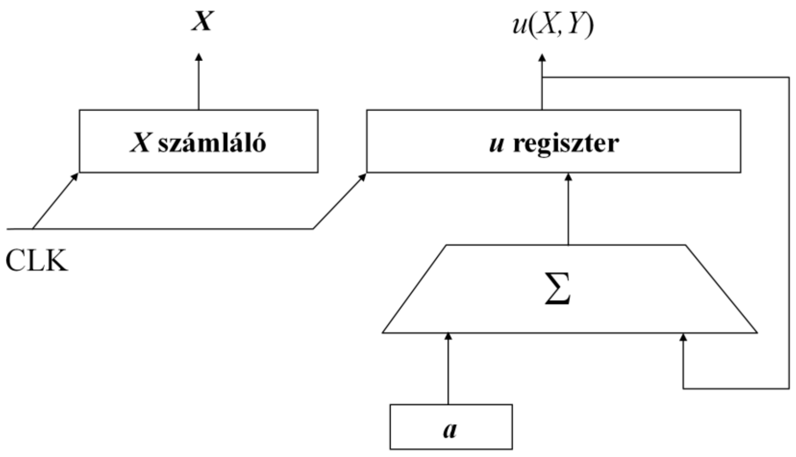
A triangle can be parameterized with an affine transformation (x,y are linear functions of u, v). Screen space coordinates are obtained with affine transformation from x,y. Thus the mapping between texture space and screen space is also an affine linear transformation:

The triangle is rasterized in screen space. When a pixel is processed, texture coordinate pair u,v must be determined from pixel coordinates X,Y.

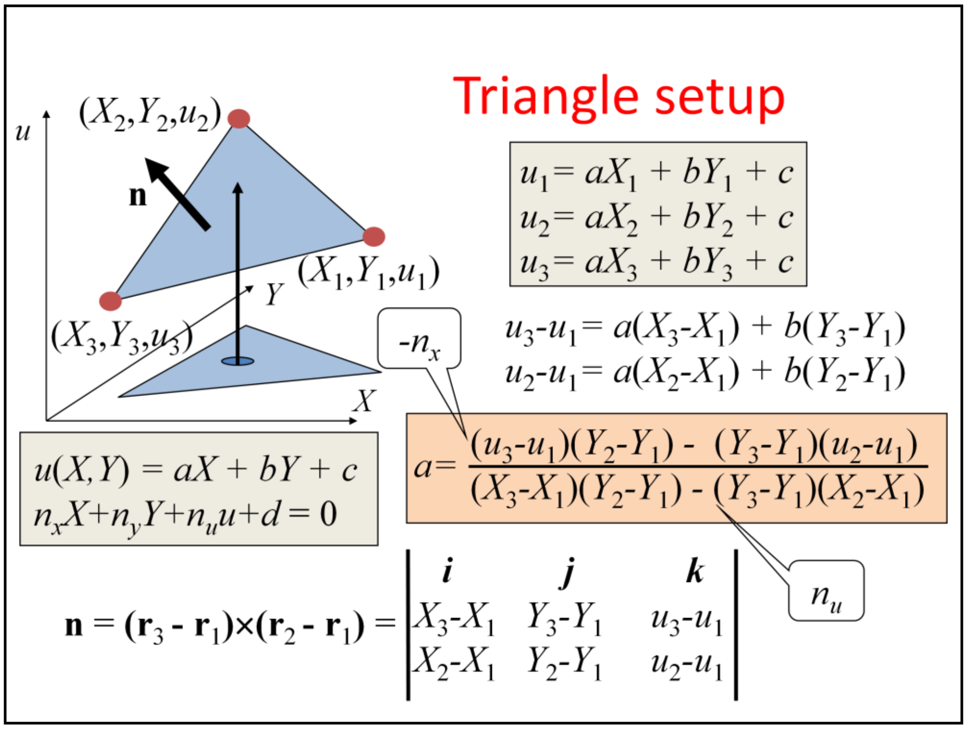


The crucial problem is then the interpolation of the color, other property or texture coordinates for internal pixels. This interpolation must be fast as we have just a few nanoseconds for each pixel. Let us denote the interpolated property by u (it can be the red intensity or anything else). Linear interpolation means that u is a linear function of pixel coordinates X, Y. This linear function evaluation would require two multiplications and two additions. This can be reduced to a single addition if we use the incremental concept and focus on the difference between the u values of the current and previous pixels in this scanline.

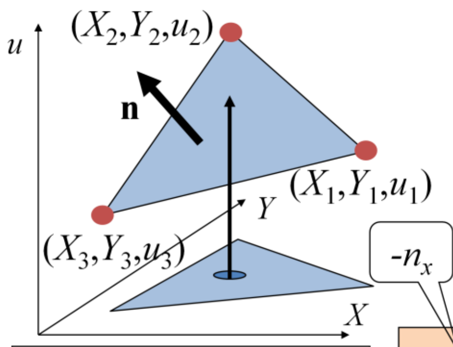
## Interpolációs hardver



Such an incremental algorithm is easy to be implemented directly in hardware. A counter increments its value to generate coordinate X for every clock cycle. A register that stores the actual u coordinate in fixed point, non-integer format (note that increment *a* is usually not an integer). This register is updated with the sum of its previous value and *a* for every clock cycle.



## Triangle setup



$$u_1 = aX_1 + bY_1 + c$$
$$u_2 = aX_2 + bY_2 + c$$
$$u_3 = aX_3 + bY_3 + c$$

$$u_3 - u_1 = a(X_3 - X_1) + b(Y_3 - Y_1)$$
$$u_2 - u_1 = a(X_2 - X_1) + b(Y_2 - Y_1)$$

$$u(X,Y) = aX + bY + c$$
$$n_x X + n_y Y + n_u u + d = 0$$

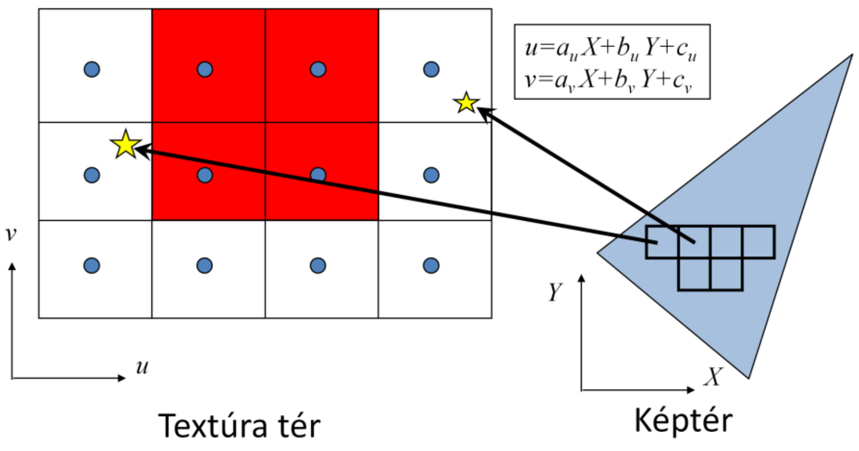
$$a = \frac{\overbrace{(u_3-u_1)(Y_2-Y_1) - (Y_3-Y_1)(u_2-u_1)}^{-n_x}}{\underbrace{(X_3-X_1)(Y_2-Y_1) - (Y_3-Y_1)(X_2-X_1)}_{n_u}}$$

$$\mathbf{n} = (\mathbf{r}_3 - \mathbf{r}_1)\times(\mathbf{r}_2 - \mathbf{r}_1) = \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ X_3-X_1 & Y_3-Y_1 & u_3-u_1 \\ X_2-X_1 & Y_2-Y_1 & u_2-u_1 \end{vmatrix}$$

The final problem is how increment *a* is calculated. One way of determining it is to satisfy the interpolation constraints at the three vertices.

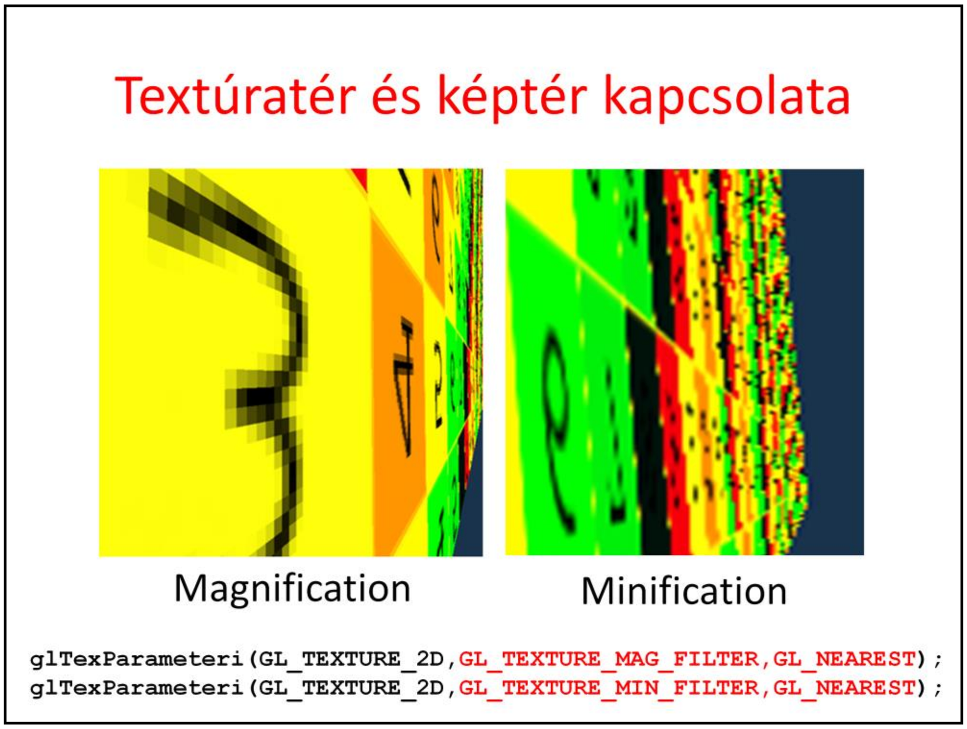
The other way is based on the recognition that we work with the plane equation where X,Y,u coordinates are multiplied by the coordinates of the plane’s normal. The normal vector, in turn, can be calculated as a cross product of the edge vectors.

## Textúra szűrés



Rasterization visits pixels inside the projection of the triangle and maps the center of the pixel from screen space to texture space to look up the texture color. This mapping will result in a point that is in between the texel centers. More importantly, this mapping may be a magnification, which means that a single step in screen space results in a larger step in texture space, so we may skip texels, and the result will be a mess or noise.

From signal processing point of view, in this case, the texture is a high frequency signal which is sampled too rarely, resulting in sampling artifacts.



## Textúratér és képtér kapcsolata

Magnification

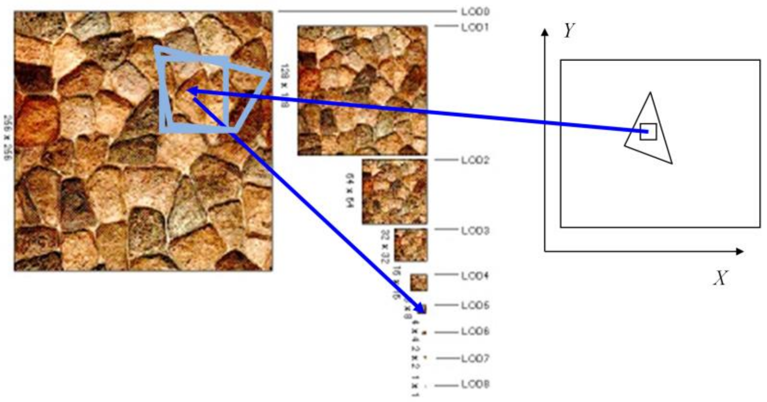
Minification

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
```

In case of oversampling or undersampling artifacts occur.

## Mip-map (multum in parvo) Minification-ra jó

Mip-map is van: ☺
Az a default: ☹

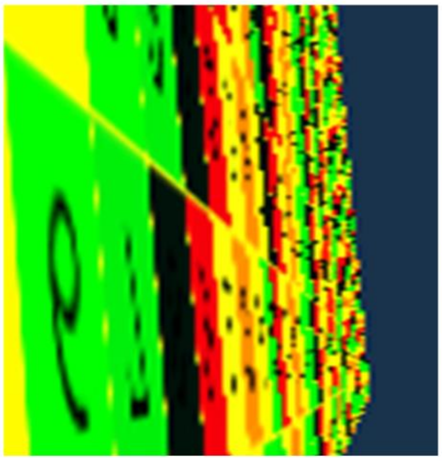


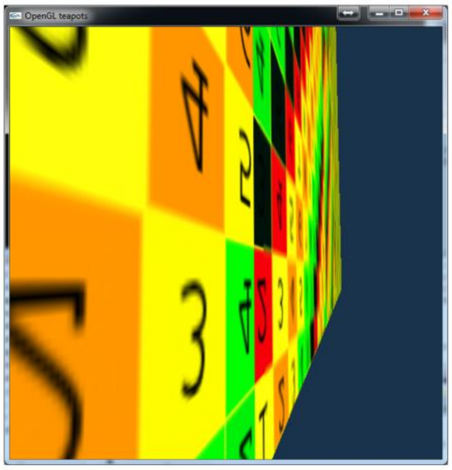
```
a) glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_NEAREST); // Mip-mapping,

b) glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_LINEAR); // Tri-linear filtering
```

The solution for such sampling problems is filtering. Instead of mapping just the center of the pixel, the complete pixel rectangle must be mapped to texture space at least approximately, and the average of texels in this region should be returned as a color. However, it would be two time consuming.

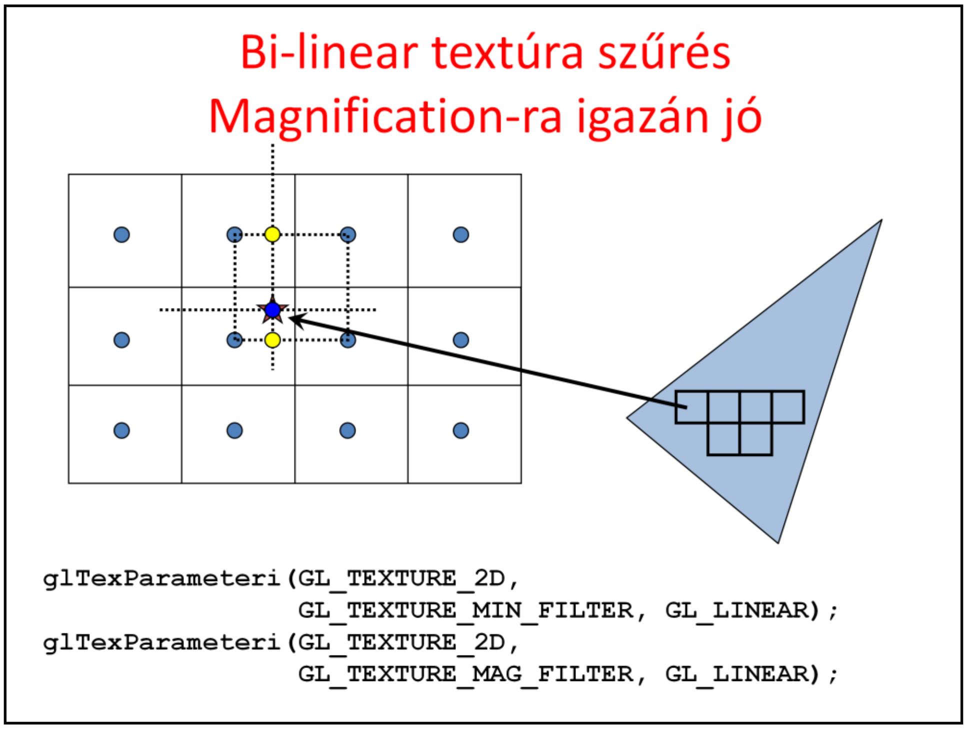
One efficient approximation is to prepare the texture not only in its original resolution, but also in half resolution, quarter resolution, etc. where a texel represents the average color of a square of the original texel. During rasterization, OpenGL estimates the magnification factor, and looks up the appropriate version of filtered, downsample texture. The collection of the original and downsampled textures is called mip-map.

## Mip-map

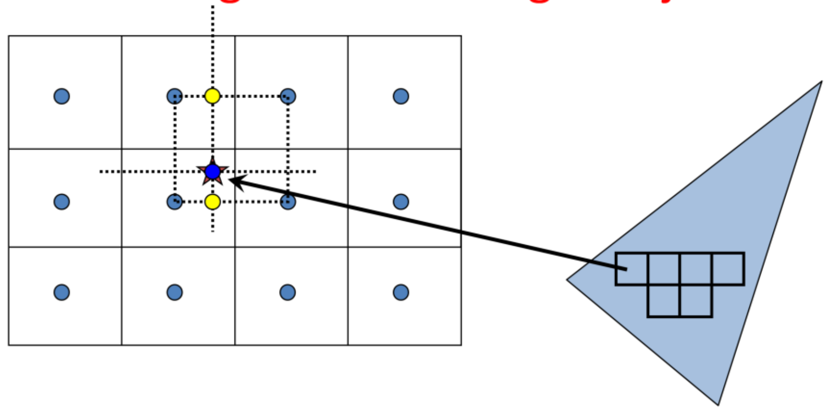

GL_NEAREST



GL_LINEAR_MIPMAP_NEAREST



## Bi-linear textúra szűrés
## Magnification-ra igazán jó

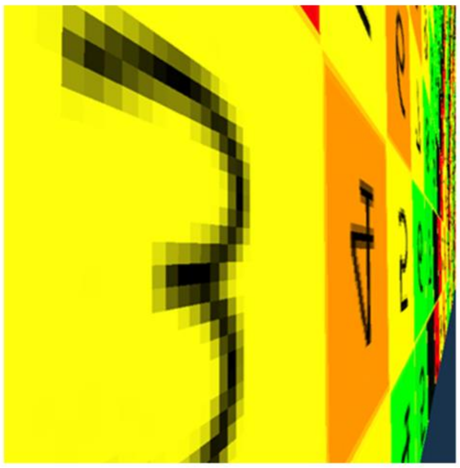


```
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

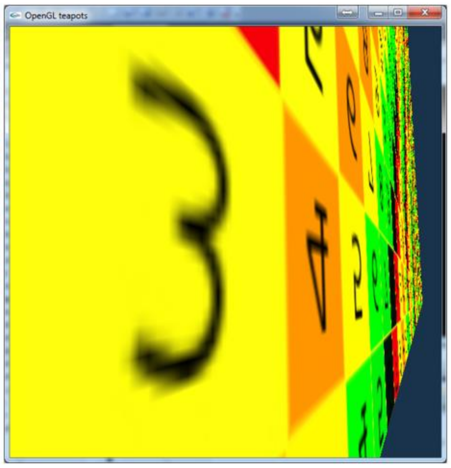
If we do not like to prepare our texture with reduced resolution, there is another simpler filtering scheme. When a pixel center is mapped to texture space, not only the closest texel is obtained but the four closest ones, and the filtered color is computed as the bi-linear interpolation of their colors.

The filtering method can be set separately when the pixel to texture space is a magnification or when it is a minification.

## Bi-linear filtering



**`GL_NEAREST`**



**`GL_LINEAR`**

## Ellenőrző kérdések

- Bizonyítsa be, hogy bármely 4+ csúcsú sokszögnek van diagonálja!
- Bizonyítsa be a kétfül tételt!
- Van értelme egy kört raszterizáló algoritmusnak?
- Írjon sokszögkitöltő algoritmust, amely nem egyszerű (határ önmagát metszi és több határ is van) sokszögeket is ki tud tölteni.
- Implementálja a vágás és raszterizálás algoritmusait!
- Írjon programot, amely eldönti, hogy egy koordináta-tengelyekkel párhuzamos téglalap tartalmaz-e egy szakaszból vagy egy sokszögből valamennyit?
- Adja meg egy 2D szerkesztő (pl. egyszerűsített Powerpoint) osztálydiagramját.
- Mi az értelme a normalizált eszköz-koordinátarendszer bevezetésének?