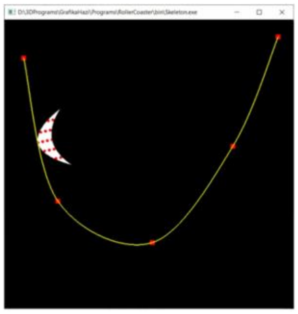
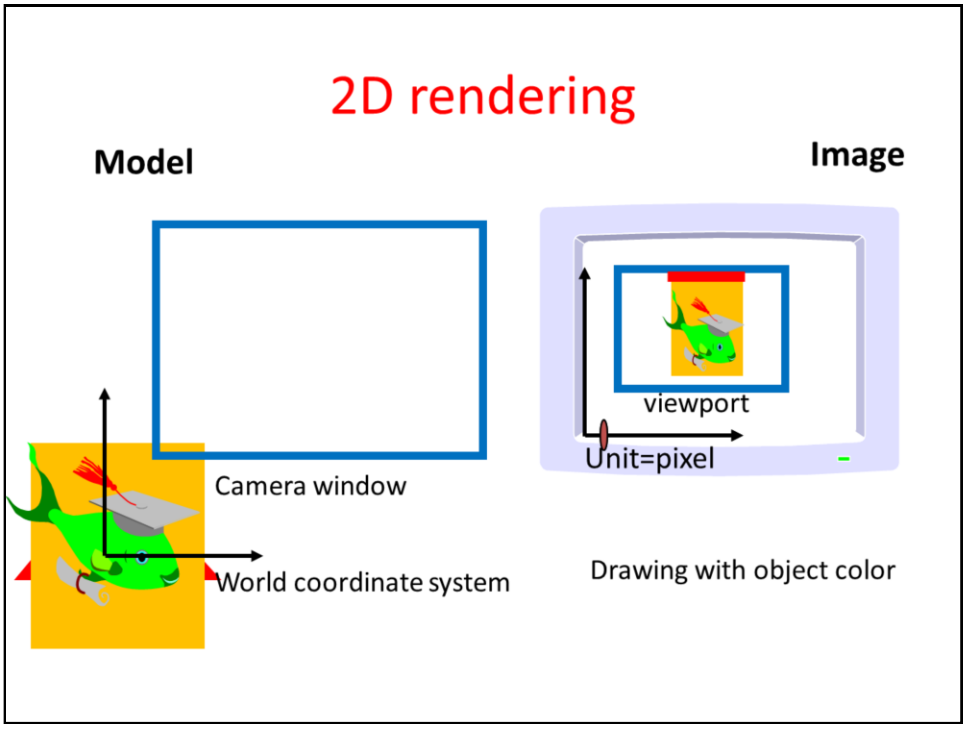
*"Photographers don't take pictures.
They create images."*
Mark Denman

2D rendering

Szirmay-Kalos László


D:\3DPrograms\GrafikaHazi\Programs\RollerCoaster\bin\Skeleton.exe



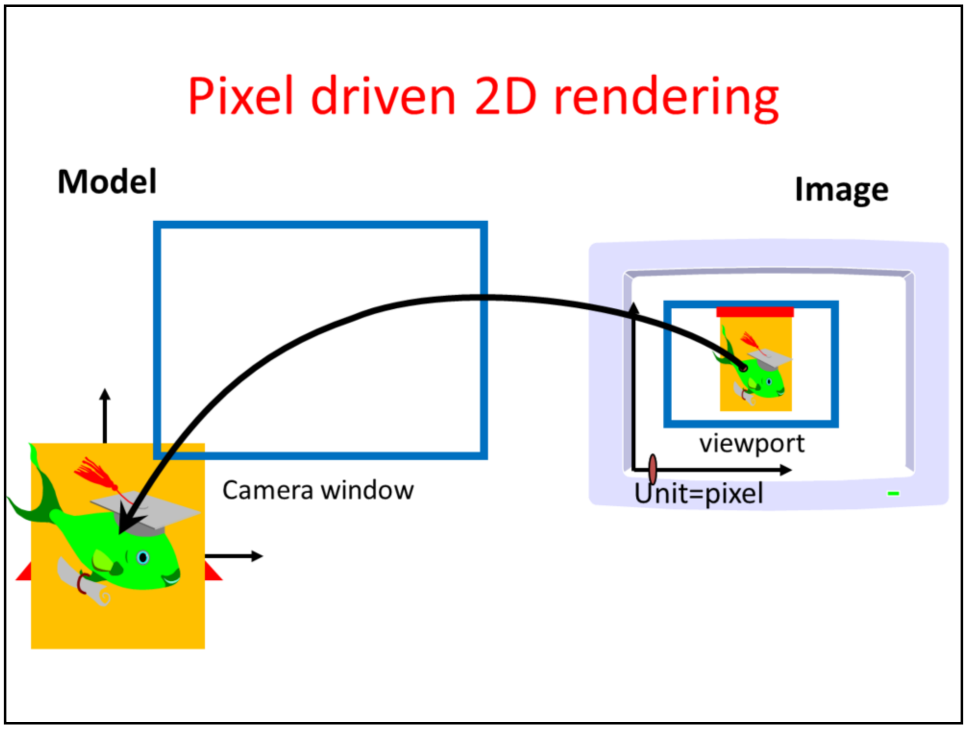
Let us consider the rendering problem when the virtual world is two dimensional, so objects are in a plane. The virtual world should be represented by numbers, for which we need a **world coordinate system**. A convenient reference system is a Cartesian coordinate system with an origin, two axes and also a unit. Using these every point of the plane can be specified by two numbers defining the distance traveled along the two axes and measured with respect to the unit.

With pairs of numbers, points can be defined, which can form primitives by adding topology information. For example, we can say that these three points define a triangle. Primitives are given material properties, which usually include the color.

The rendering process takes a photograph of the virtual world and presents the photo on the computer screen. The rectangle of the photo is called the **viewport**.

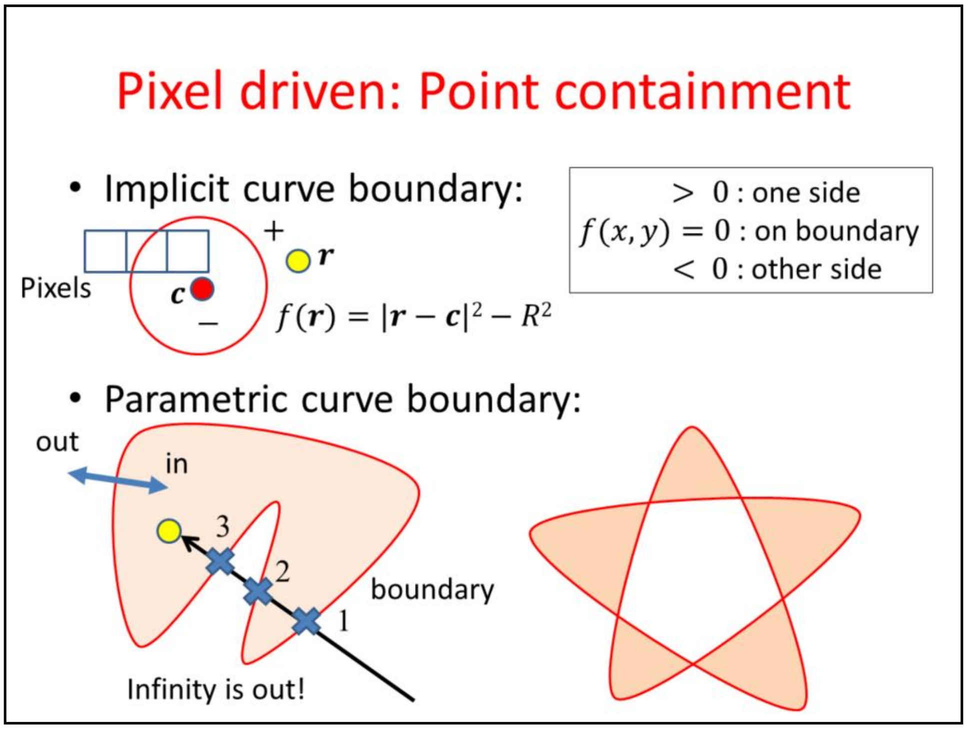
Pixels on the computer screen are identified in screen coordinates that identify the row and column of each pixel. In other words, the unit of the screen coordinate system is the pixel. To implement the photographing process, we introduce a camera in the virtual world. In 2D, the camera is just a rectangle, called the **camera window**. This rectangle has edges parallel with the coordinate axes. Similarly, the viewport edges are also parallel with the axes of the screen coordinates.

Rendering finds a correspondence between the pixels of the viewport and the objects of the virtual world. This correspondence can be established from two opposite directions. We can start the process in the virtual world, transforms objects one by one on the screen, and color pixels covered by the transformed objects. This approach is **object-driven**.



Alternatively, we can also start at the pixels of the viewport, when the method is called **pixel-driven**. Each pixel is transformed back to the world coordinate system and the object containing the transformed point is identified. The pixel color is then the color of the identified object. If more than one object contains the transformed point (in the Figure this is the case where the green fish and the yellow rectangle overlap), then that object is selected which has the higher priority.

Note that in popular drawing packages, actions line ”bring forward” or ”Send back” manipulate the priority of objects.



Pixel driven: Point containment

- Implicit curve boundary:

$$f(x,y) \begin{cases} > 0 : \text{one side} \\ = 0 : \text{on boundary} \\ < 0 : \text{other side} \end{cases}$$

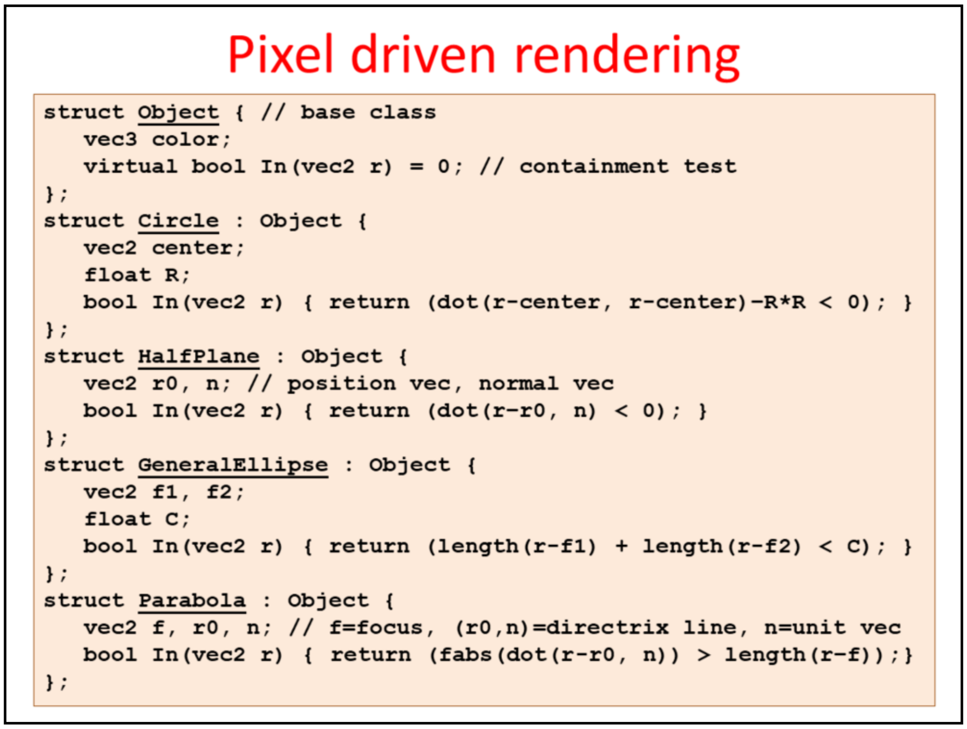
$$f(\boldsymbol{r}) = |\boldsymbol{r} - \boldsymbol{c}|^2 - R^2$$

- Parametric curve boundary:

Pixels
$+$
$\boldsymbol{r}$
$\boldsymbol{c}$
$-$
out
in
3
2
1
boundary
Infinity is out!

The core of a pixel driven algorithm is the **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 driven 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 < 0); }
};
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));}
};
```

Here we show the implementation of a simple pixel-driven 2D drawing package (the complete code can be downloaded from the cg.iit.bme.hu web page).

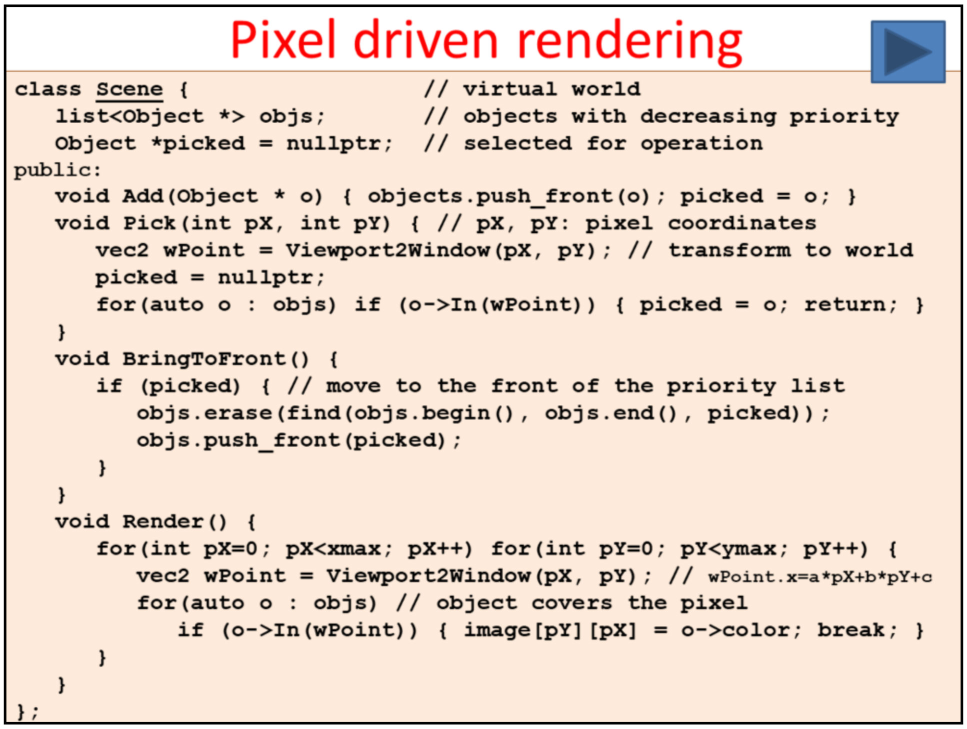
Object is the abstract base class of all object. It has a color, which will be the pixel color if this particular object is visible in that pixel. Additionally, it has a pure virtual function **In** that checks whether point r given in world coordinates is contained by this object.

The implementation of the In function depends on the type of the object. Therefore, different types are derived from the base class Object, and In is given a particular implementation, for which type dependent parameters may be needed. For example, a **Circle** is defined by a center and a radius R. Based on these parameters, the In function checks whether the square distance of point r from the center is smaller than the squared of the radius. If it is smaller, the point is in. In case of equality, the point is on the boundary. If the difference is positive, the point is outside.

Halfplane has a line boundary defined by position vector r0 and normal vector n, and we assume that normal vector n point outwards. Thus, substituting a point r into the equation of the line, the sign of the result tells us whether the point is in the outer section (dot product is positive), on the boundary line (dot product is zero), or inside (dot product is negative).

GeneralEllipse is based on the geometric definition of the ellipse: set of points for which the sum of distance from the two focal points is less than constant C.

Parabola is also the direct implementation of the definition: set of points for which the distance from a line called directrix is greater than the distance from its focal point.



Pixel driven 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) // object covers the pixel
            if (o->In(wPoint)) { image[pY][pX] = o->color; break; }
      }
   }
};
```

The **Scene** is a heterogeneous collection of objects. We use the list since deletion and priority management require the dynamic erase and insertion of elements. We have a special data member to show the picked or selected object that is the target of the future operations.

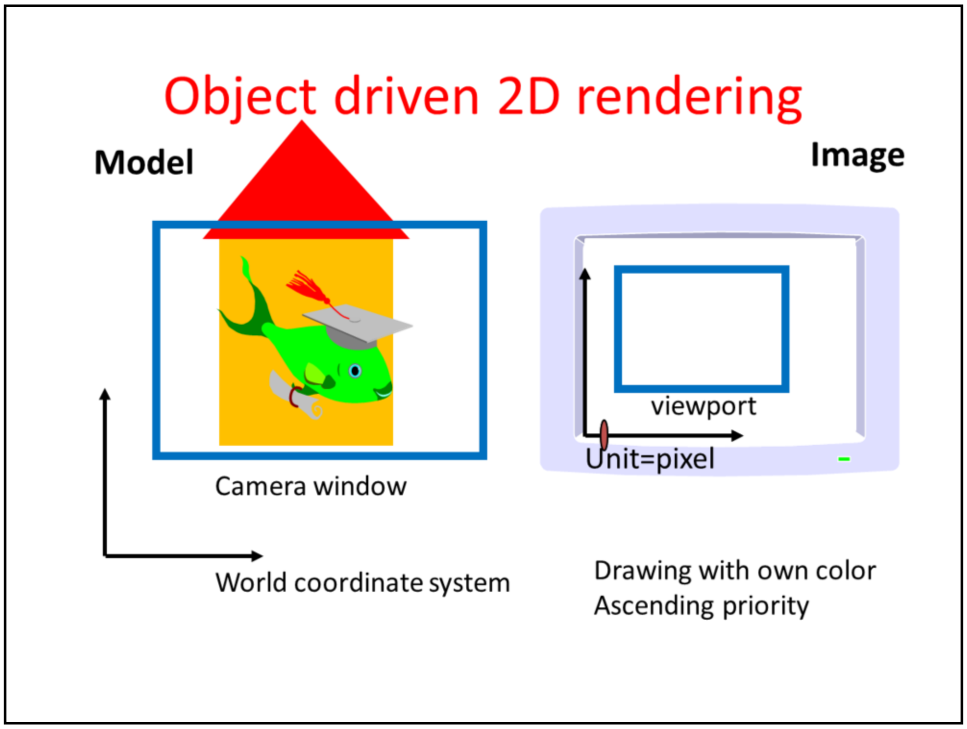
Add adds a new object to the list and the new object becomes the selected one (this is the typical strategy in drawing packages).

Pick takes the pixel coordinate, transforms it to the world coordinate system, and tests objects for containment of this point in the order of the priority. If an object is found, it will be the picked one and terminate testing since we need that object to the picked which is in front of others.

BringToFront moves the picked object to the front of the list, i.e. gives the highest priority to it. Send to back and Delete are not shown here, because their implementation is very similar. Sent to back would move the picked object to the end of the list, Delete would simply erase it.

Render computes the two dimensional image array by computing the colors of pixels. Each pixel is visited in a double loop. The pixel is transformed to world coordinates. Objects are tested for containment of the transformed point selecting the one with maximum priority if more than one object contains the point. The selected object’s color is written into the pixel.

The Pixel-driven solution is simple and elegant. However, it is two slow, interactive frame rates are impossible even with moderate number of objects. Therefore, we need another strategy.



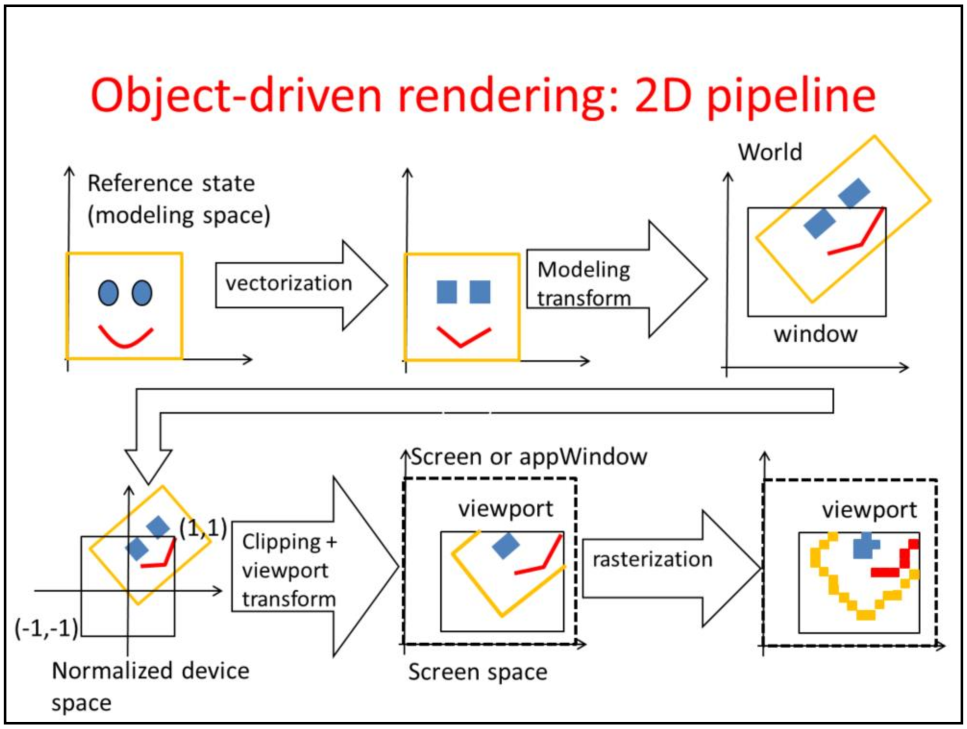
Let us consider the rendering problem when the virtual world is two dimensional, so objects are in a plane. The virtual world should be represented by numbers, for which we need a **world coordinate system**. A convenient reference system is a Cartesian coordinate system with an origin, two axes and also a unit. Using these every point of the plane can be specified by two numbers defining the distance traveled along the two axes and measured with respect to the unit.

With pairs of numbers, points can be defined, which can form primitives by adding topology information. For example, we can say that these three points define a triangle. Primitives are given material properties, which usually include the color.

The rendering process takes a photograph of the virtual world and presents the photo on the computer screen. The rectangle of the photo is called the **viewport**.

Pixels on the computer screen are identified in screen coordinates that identify the row and column of each pixel. In other words, the unit of the screen coordinate system is the pixel. To implement the photographing process, we introduce a camera in the virtual world. In 2D, the camera is just a rectangle, called the **camera window**. This rectangle has edges parallel with the coordinate axes. Similarly, the viewport edges are also parallel with the axes of the screen coordinates.

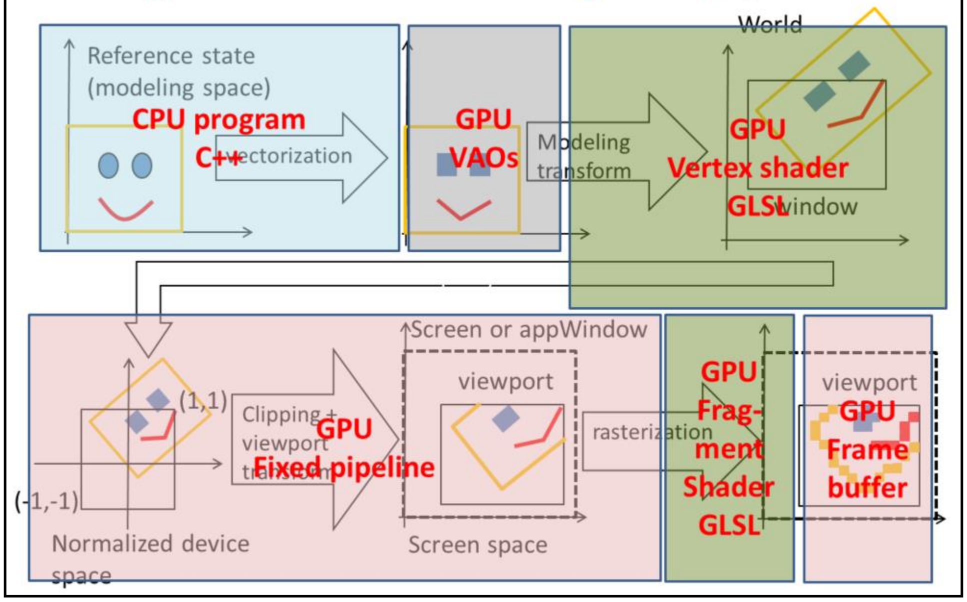
Rendering finds a correspondence between the pixels of the viewport and the objects of the virtual world. This correspondence can be established from two opposite directions. We can start the process in the virtual world, transforms objects one by one on the screen, and color pixels covered by the transformed objects. This approach is **object-driven**.



Object-driven 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, 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.

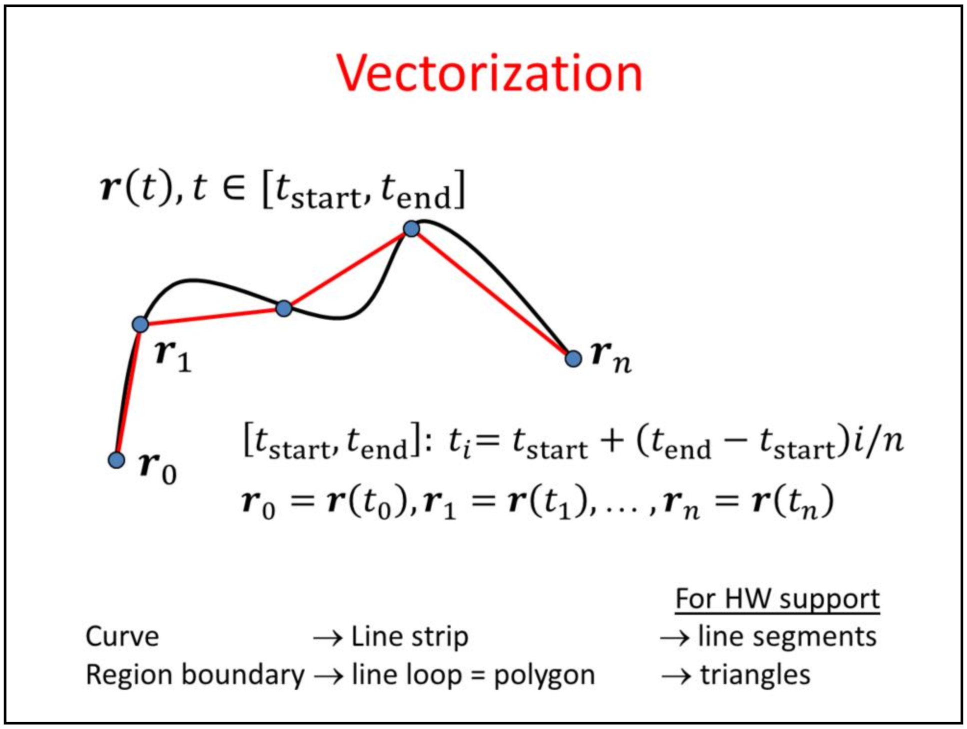
Modeling transformation places the objects in world coordinates. This typically involves scaling, rotation and translation to set the size, orientation and the position of the object. In world, objects meet each other and also the 2D camera, which is the window rectangle or AABB (axis aligned bounding 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 object 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 of the viewport or the size of the window, so can be easily implemented in a fixed 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.

Object-driven rendering: 2D pipeline



Object-driven rendering is implemented with the help of the GPU. This figure shows which hardware is responsible for different stages of the rendering pipeline. The management of the virtual world and the vectorization of the objects are done by our CPU program written typically in C++. Vectorized point, line and triangle primitives are copied to the GPU where they are organized in **Vertex Array Objects** or **VAO**s for short. A VAO is a collection of arrays storing vertices and vertex properties. These arrays are called **vertex buffer objects** or **VBO**s. The GPU feeds the **vertex shader processor** by the vertices and their attributes. A vertex shader processor gets one vertex with its attributes at a time and is responsible for computing the location of the vertex in normalized device coordinates. Additionally, the vertex shader may modify the attributes. Vertices output by the vertex shader form primitives like points, lines or triangles, which are processed by the **fixed algorithm pipeline** of the GPU. This hardware clips the primitives, transforms them to screen space and rasterizes them to produce a sequence of pixels. During these steps vertex attributes are interpolated to generate attributes for each pixel that gradually change inside the primitive. For each pixel of this sequence, the **fragment shader processor** can determine the color of the pixel based on the interpolated attributes. The output of the fragment shader goes into the frame buffer.

Vertex shader and fragment shader processors are programmed in the GLSL (OpenGL Shading Language). The fixed algorithm pipeline cannot be programmed as it is hardwired.



Vectorization

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

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

$\boldsymbol{r}_0 = \boldsymbol{r}(t_0), \boldsymbol{r}_1 = \boldsymbol{r}(t_1), \ldots, \boldsymbol{r}_n = \boldsymbol{r}(t_n)$

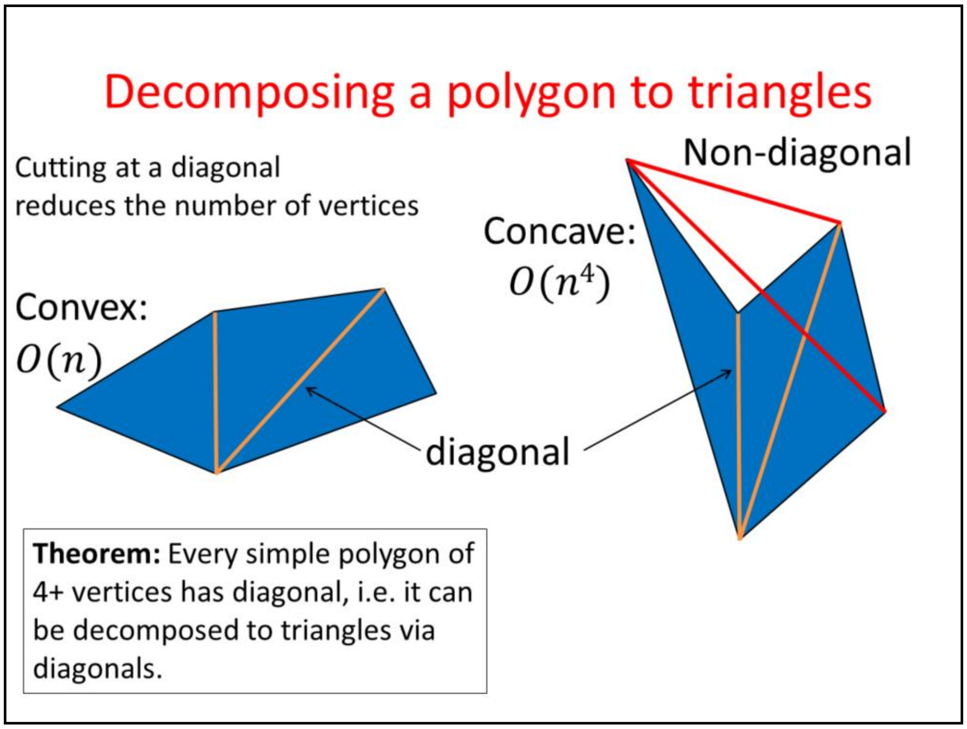
		For HW support
Curve	$\rightarrow$ Line strip	$\rightarrow$ line segments
Region boundary	$\rightarrow$ line loop = polygon	$\rightarrow$ triangles

Vectorization approximates model objects with points, lines (or polylines), and triangles (or polygons). The main reason is that homogeneous linear transformations preserve only these types (and for the same reason, OpenGL accepts only these primitive types). A positive side effect of this decision is that pipeline stages like clipping and rasterization should be solved only for points, lines and triangles.

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.

If we had a software implementation of the output pipeline, we would work with line strips or polygons or arbitrary number of vertices. However, in GPU implementation, the hardware prefers records of fixed size, therefore polylines should be converted to line segments and polygons to triangles. The conversion of line strips to line segments is straightforward. However, the conversion of polygons to triangles is not.



Decomposing a polygon to triangles

Cutting at a diagonal reduces the number of vertices

Convex: $O(n)$

Concave: $O(n^4)$

Non-diagonal
diagonal

Theorem: Every simple polygon of 4+ vertices has diagonal, i.e. it can be decomposed to triangles via diagonals.

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. We need to clip along diagonals since only this clipping guarantees that the resulting polygons have less number of vertices than the original polygon, thus the method will terminate in finite step.

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.

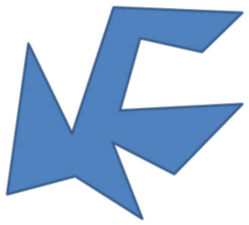
This is not the case for **concave polygons**, when line segments connecting vertices can intersect edges or can fully be outside of the polygon. The good news is that all **simple polygons**, even concave ones, have diagonals, so they can be broken to triangles by diagonals. A polygon is said to be simple if its boundary is a single polyline that does not intersect itself.

The sketch of the proof is shown in the right bottom corner. Let us find a vertex that is extremal in one direction, for instance, that has the maximum x coordinate. Consider the triangle defined by this vertex, the previous and the next vertices. If this triangle does not contain vertices, then the previous and next vertices form a diagonal, so the theorem is proven for this case. If the triangle has vertices, let us find the one with the maximum x coordinate. Connecting this point to the original extremal point, the line segment is inside and cannot intersect any edge, so it is a diagonal.

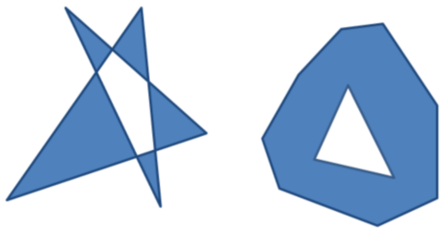
An algorithm based on this idea would search for diagonals, and having found one, the polygon would be cut into two, for which the same algorithm is executed recursively until all polygons are triangles.

This approach has cubic complexity in terms of the number of vertices. Fortunately, there is a simpler and more efficient algorithm for polygon triangulation.

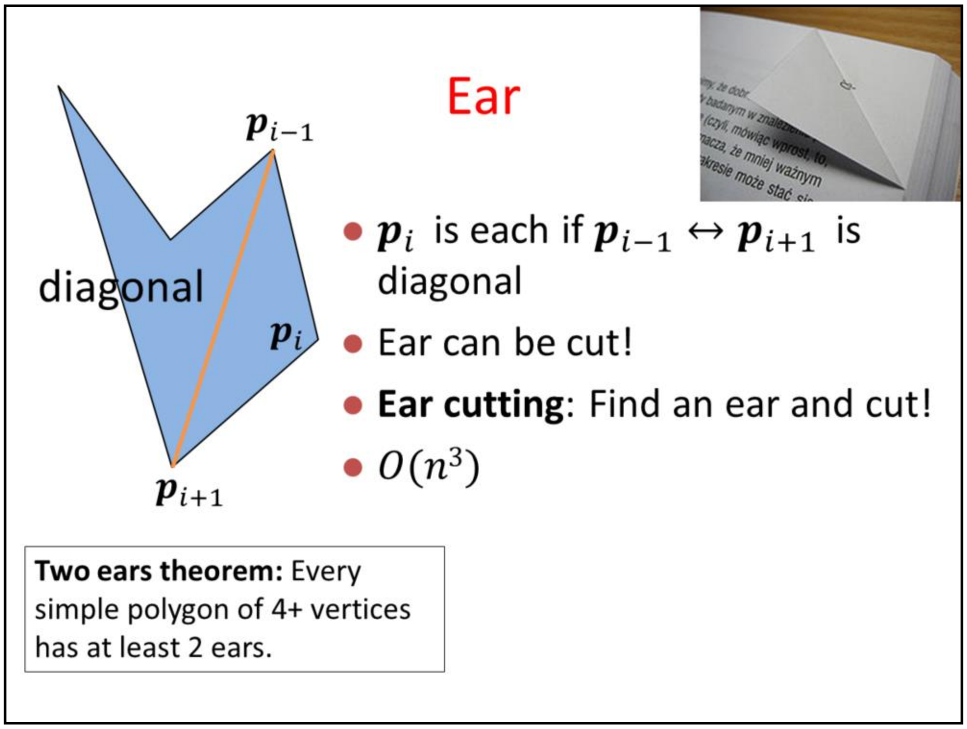
Simple polygons



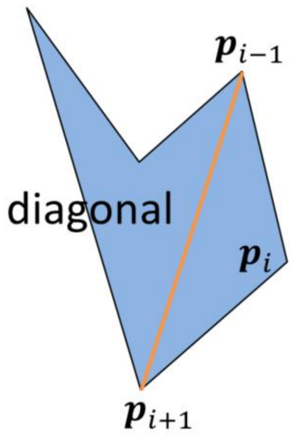
Simple polygon



Non-simple polygon



Ear

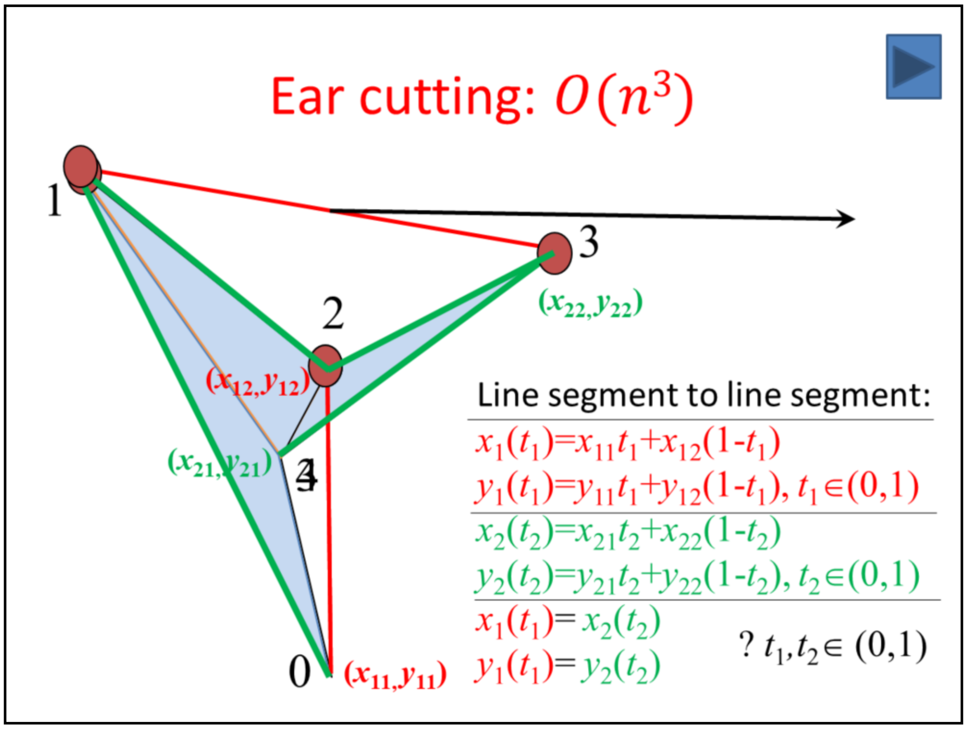


- $\boldsymbol{p}_i$ is each if $\boldsymbol{p}_{i-1} \leftrightarrow \boldsymbol{p}_{i+1}$ is diagonal
- Ear can be cut!
- **Ear cutting**: Find an ear and cut!
- $O(n^3)$

Two ears theorem: Every simple polygon of 4+ vertices has at least 2 ears.

This triangulation algorithm searches for special vertices called **ears**. 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 **simple 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 any edge, the graph falls apart, so 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.



Ear cutting: $O(n^3)$

Line segment to line segment:

$x_1(t_1)=x_{11}t_1+x_{12}(1\text{-}t_1)$
$y_1(t_1)=y_{11}t_1+y_{12}(1\text{-}t_1),\ t_1\in(0,1)$

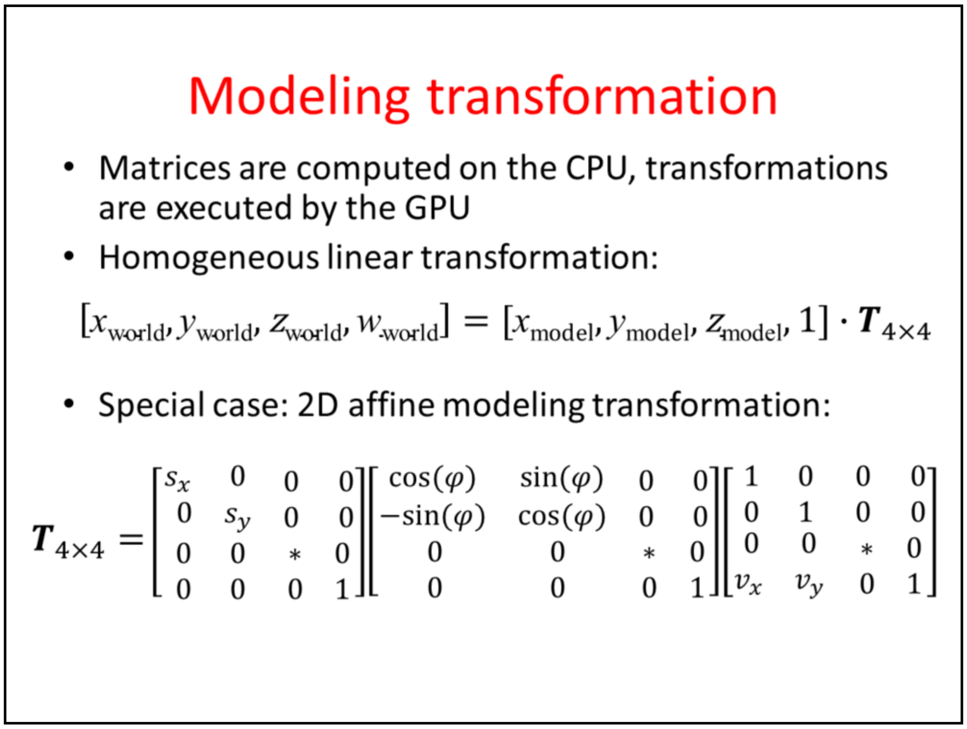
$x_2(t_2)=x_{21}t_2+x_{22}(1\text{-}t_2)$
$y_2(t_2)=y_{21}t_2+y_{22}(1\text{-}t_2),\ t_2\in(0,1)$

$x_1(t_1)=x_2(t_2)$
$y_1(t_1)=y_2(t_2)$? $t_1,t_2\in(0,1)$

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 inside (if it does, it is not a diagonal). We can skip those edges that share a vertex with the tested diagonal candidate since there cannot be a second intersection between two lines. One way of testing two line segments for intersection is the solution of the system of linear equations stating that the intersection point is in both line segments. A line segment is the convex combination of its two endpoints. Solving the linear system, we should check whether the intersection is inside the line segment, i.e. t_1 and t_2 are in (0, 1). The test of a diagonal candidate should go on for all edges.

If there is no intersection, we should additionally 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.

In the above example, vertex 1 is not an ear because candidate 0-2 is not a diagonal as it intersects edge 3-4. Neither Vertex 2 is an ear since its middle point is outside of the polygon. Vertex 3 is an ear, so a triangle 2-3-4 can be cut from the polygon, and we proceed with the remaining simpler polygon until a single triangle is left.



Modeling transformation

- Matrices are computed on the CPU, transformations are executed by the GPU
- Homogeneous linear transformation:

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

- Special case: 2D affine modeling transformation:

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

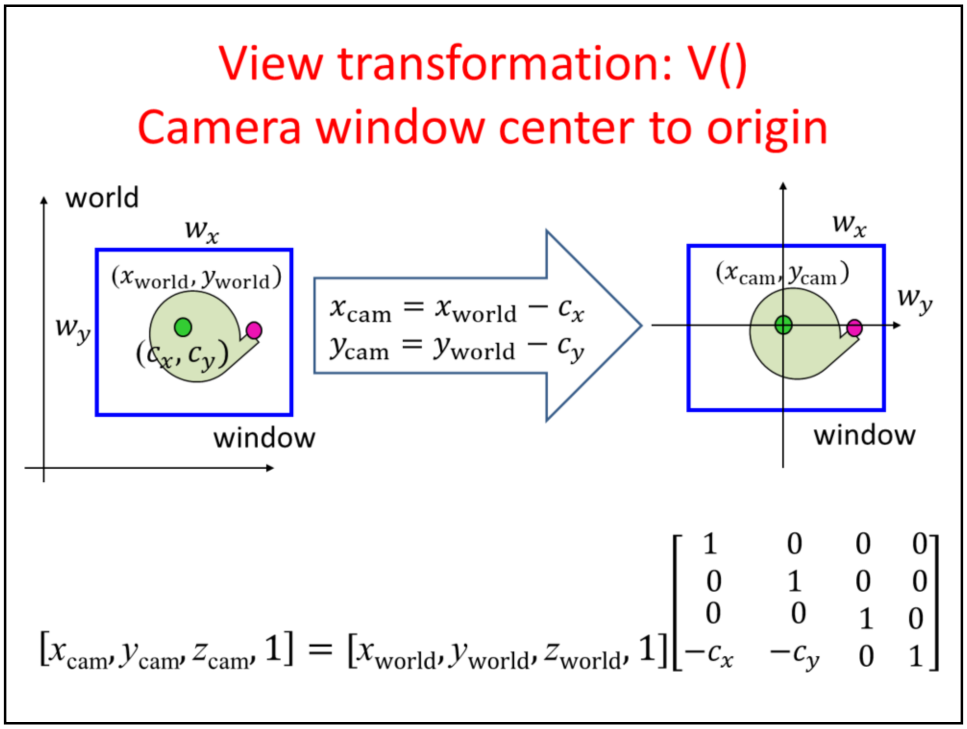
After vectorization, the first relevant step of rendering is placing the reference state primitives in world, typically scaling, rotating and finally translating its vertices. Recall that it is enough to execute these transformations to vertices, because points, lines and polygons are preserved by homogeneous 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.

mat4 class

```
struct mat4 { // row-major matrix 4x4
   vec4 rows[4];

   mat4(vec4& it, vec4& jt, vec4& kt, vec4& ot) {
      rows[0]=it; rows[1]=jt; rows[2]=kt; rows[3]=ot;
   }
   vec4& operator[](int i) { return rows[i]; }
};

inline vec4 operator*(vec4& v, mat4& m) {
   return v.x*m[0] + v.y*m[1] + v.z*m[2] + v.w*m[3];
}

inline mat4 operator*(mat4& ml, mat4& mr) {
   mat4 res;
   for (int i = 0; i < 4; i++)
       res.rows[i] = ml.rows[i] * mr;
   return res;
}
```

mat4 constructors

```
inline mat4 TranslateMatrix(vec2 t) {
       return mat4( vec4(1,   0,   0,  0),
                    vec4(0,   1,   0,  0),
                    vec4(0,   0,   1,  0),
                    vec4(t.x, t.y, 0,  1));
}
inline mat4 ScaleMatrix(vec2 s) {
       return mat4( vec4(s.x, 0,   0,  0),
                    vec4(0,   s.y, 0,  0),
                    vec4(0,   0,   1,  0),
                    vec4(0,   0,   0,  1));
}
inline mat4 RotationMatrix(float fi) {
       return mat4( vec4( cos(fi), sin(fi), 0,  0),
                    vec4(-sin(fi), cos(fi), 0,  0),
                    vec4(0,        0,       1,  0),
                    vec4(0,        0,       0,  1));
}
```



View transformation: V()
Camera window center to origin

world
w_x
w_y
$(x_{\text{world}}, y_{\text{world}})$
(c_x, c_y)
window
w_x
w_y
$(x_{\text{cam}}, y_{\text{cam}})$
window

$$x_{\text{cam}} = x_{\text{world}} - c_x$$
$$y_{\text{cam}} = y_{\text{world}} - c_y$$

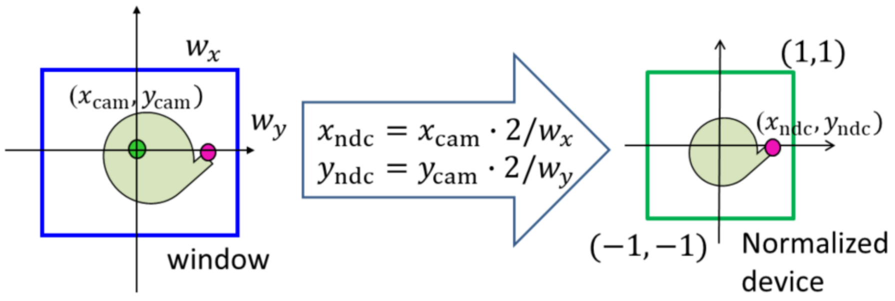
$$[x_{\text{cam}}, y_{\text{cam}}, z_{\text{cam}}, 1] = [x_{\text{world}}, y_{\text{world}}, z_{\text{world}}, 1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -c_x & -c_y & 0 & 1 \end{bmatrix}$$

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, execute clipping, and then transforming the square to the viewport.

Transforming the window to normalized device space, i.e. an origin centered square of corners (-1,-1) and (1,1) is also a sequence of two transformations:

The first is the View transformation denoted by V: a translation that moves the center of the camera window to the origin. The translation happens with the negative center of the camera window, so after the translation, the camera window center will be in the origin.

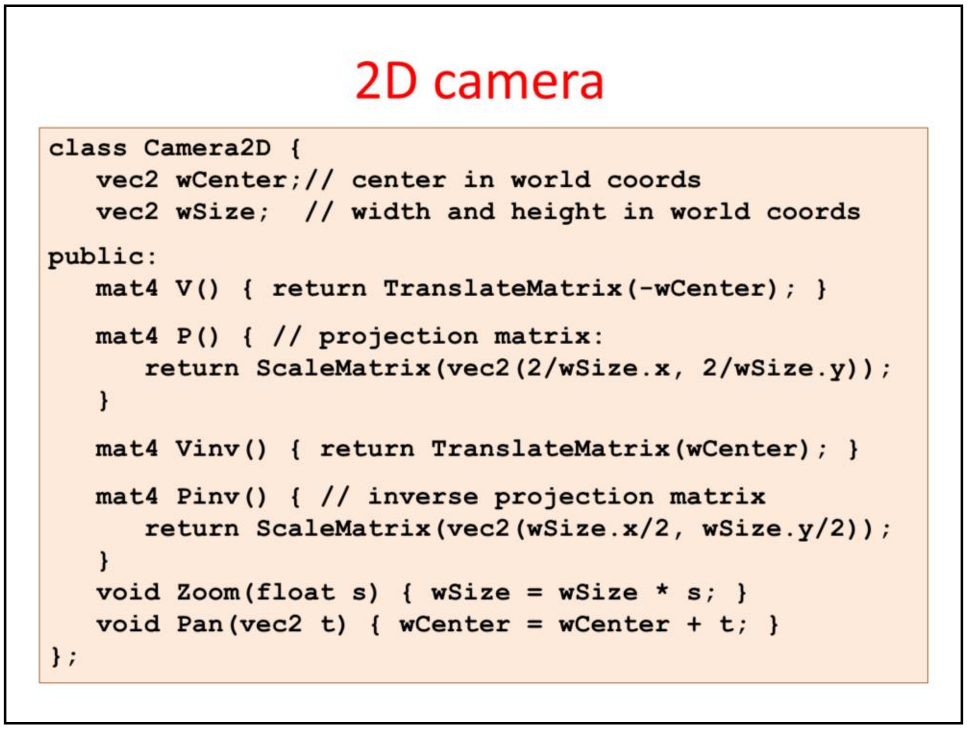
Projection P(): Camera window to a square of corners (-1, -1) and (1, 1)


w_x
$(x_{\text{cam}}, y_{\text{cam}})$
w_y
window
$x_{\text{ndc}} = x_{\text{cam}} \cdot 2/w_x$
$y_{\text{ndc}} = y_{\text{cam}} \cdot 2/w_y$
(1,1)
$(x_{\text{ndc}}, y_{\text{ndc}})$
$(-1,-1)$
Normalized device

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

The second transformation is the Projection denoted by P: a scaling that modifies the window width and height to 2.

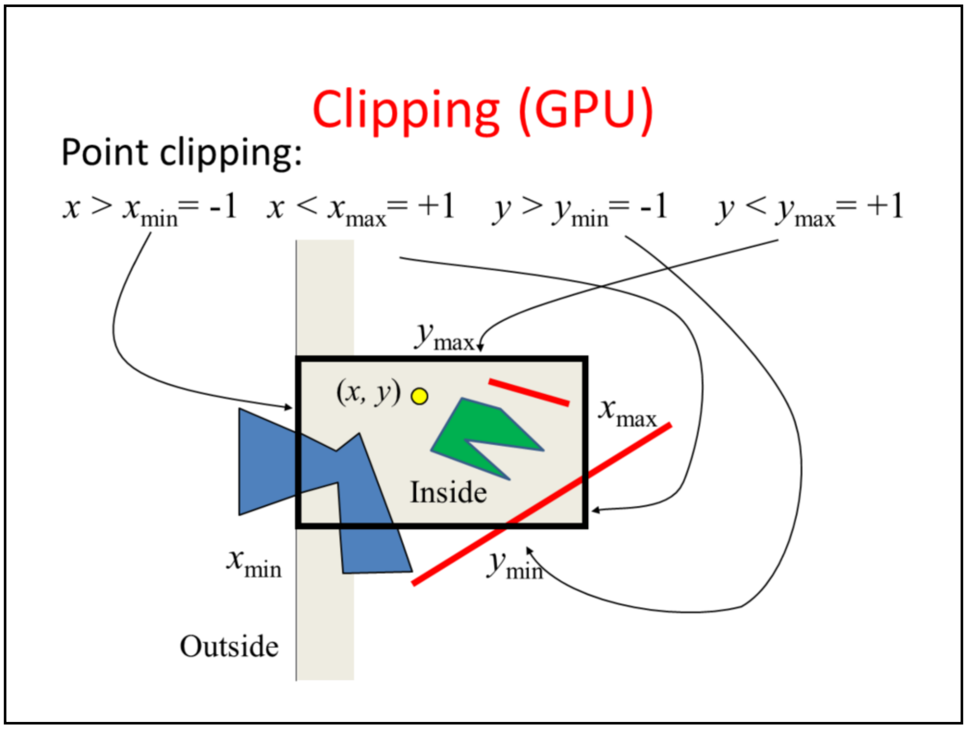
Projection transformation scales by $2/w_x$ in direction x and by $2/w_y$ in direction y where (w_x, w_y) are the width and height of the camera window to make sure that after scaling the camera window will be a square of edge length 2.



2D camera

```
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 wCenter and the size wSize of the camera window. We use w initials since these data are interpreted in world coordinates. A 2D camera 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.



Clipping (GPU)

Point clipping:

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

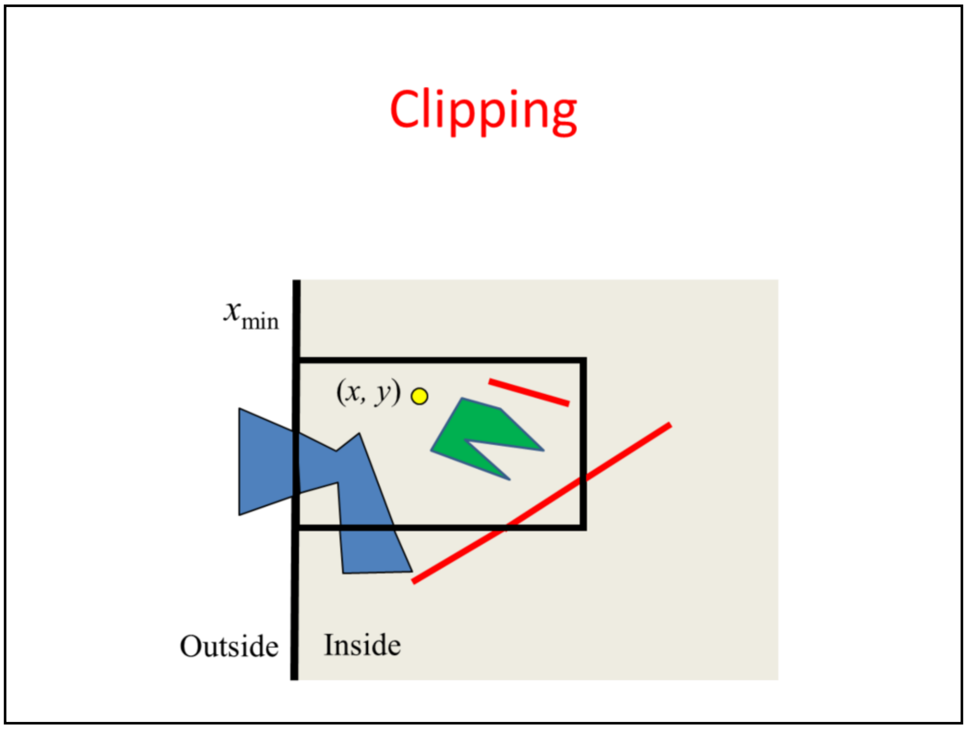
Having transformed the objects to normalized device space, the next step is clipping that removes object parts outside of the camera window or the viewport.

Clipping is executed usually in normalized device space where the camera window or the viewport is a square of corner points (-1,-1) and (1, 1),

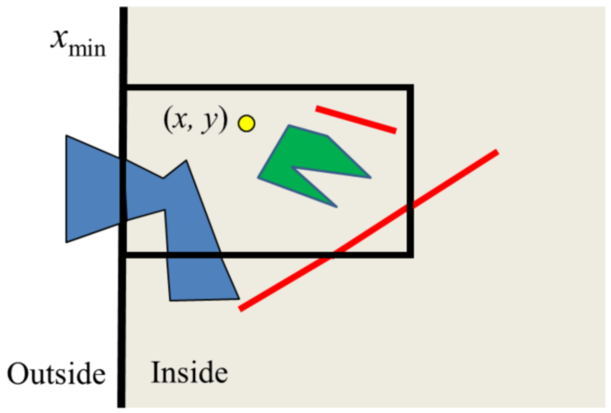
therefore for a point to survive, its x, y coordinates must be between -1 and 1. To be general, we denote the limits by $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$.

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.

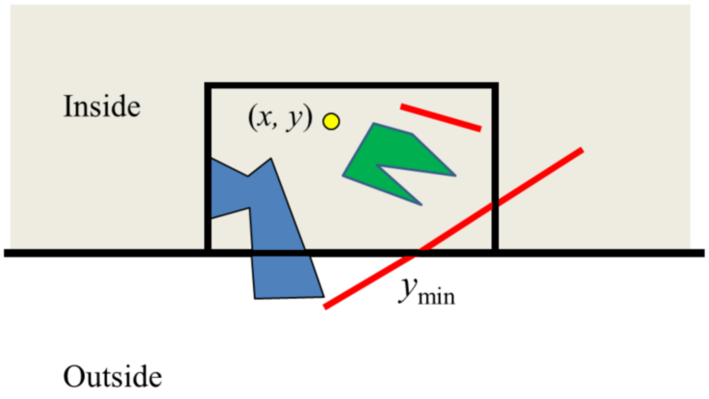


Clipping

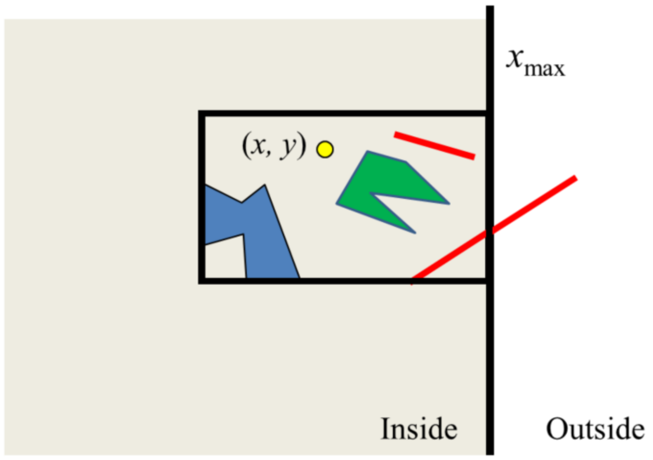


Thus clipping on a rectangle is replaced by the sequence of four clipping steps on four half planes.

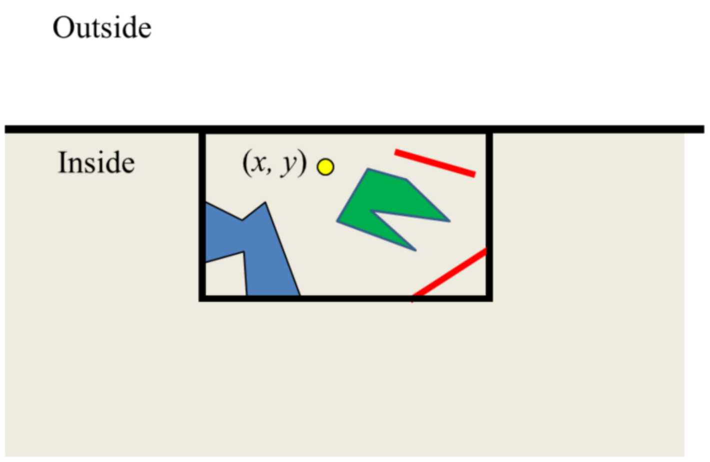
Clipping

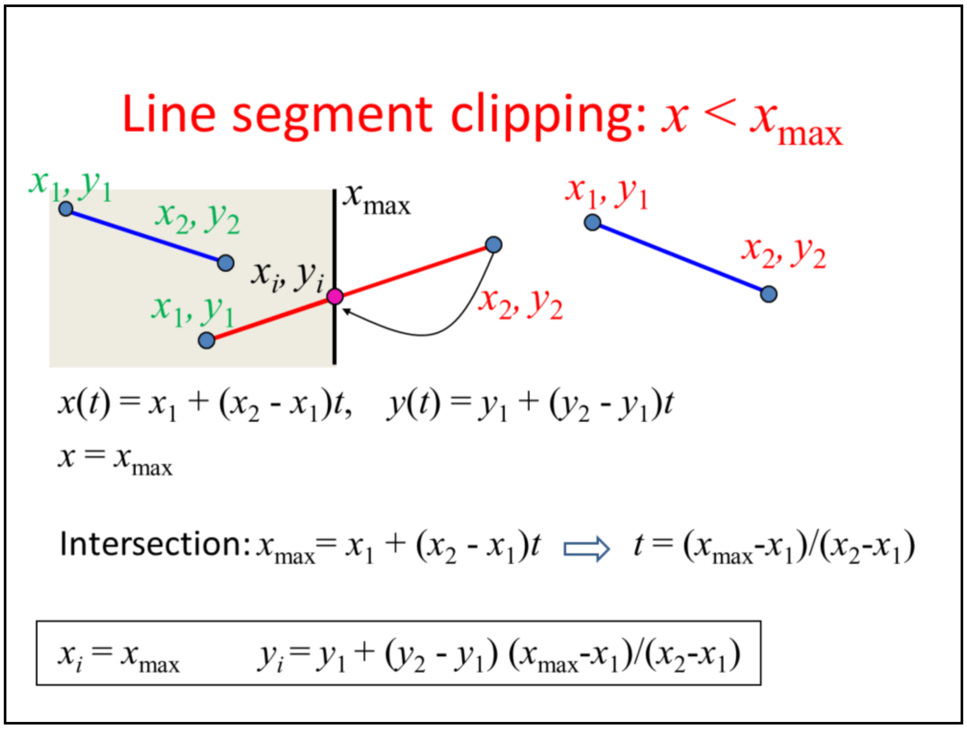


Clipping



Clipping





Line segment clipping: $x < x_{\max}$

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

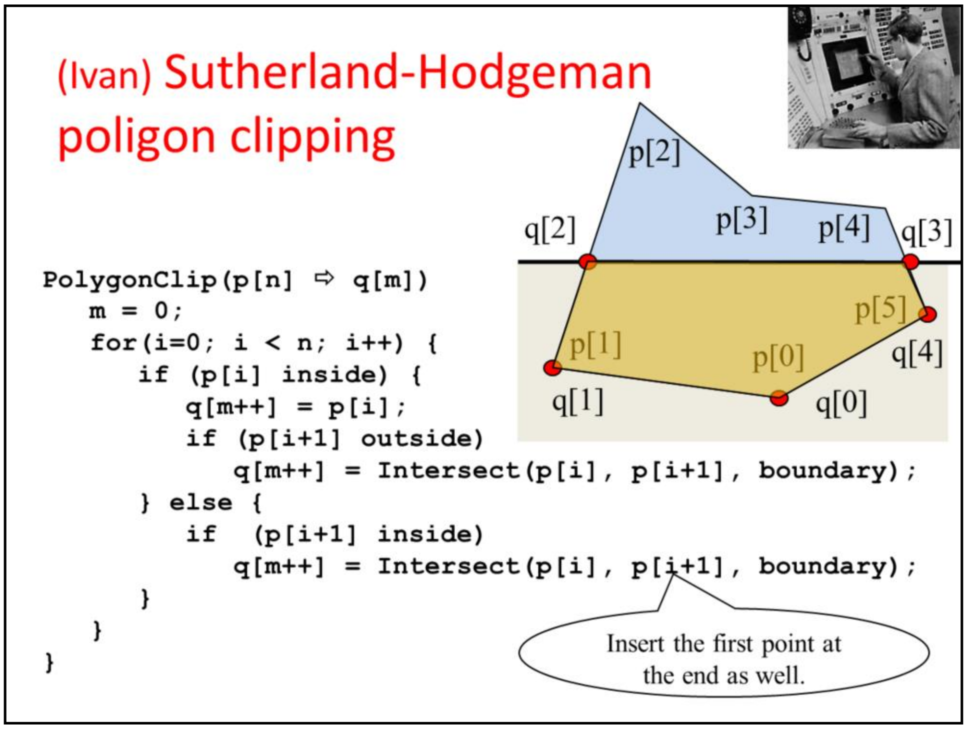
$$x = x_{\max}$$

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

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

Let us consider a line segment and its clipping on a single half plane, for example, the plane of $x < x_{\max}$. If both endpoints are inside, then the complete line segment is inside **since the inner region, which is a half plane, is convex**. If both endpoints are outside, then the line segment is completely outside, **since the outer region, which is also a half plane, 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.

For the intersection calculation, we take the parametric equation of the line segment, which expresses the line segment points as the convex combination of the two endpoints, and also the equation of the clipping boundary, e.g. $x = x_{\max}$. This system of linear equations is solved and we obtain the intersection point.



(Ivan) Sutherland-Hodgeman poligon clipping

```
PolygonClip(p[n] ⇨ q[m])
   m = 0;
   for(i=0; i < n; i++) {
      if (p[i] inside) {
         q[m++] = p[i];
         if (p[i+1] outside)
            q[m++] = Intersect(p[i], p[i+1], boundary);
      } else {
         if  (p[i+1] inside)
            q[m++] = Intersect(p[i], p[i+1], boundary);
      }
   }
}
```

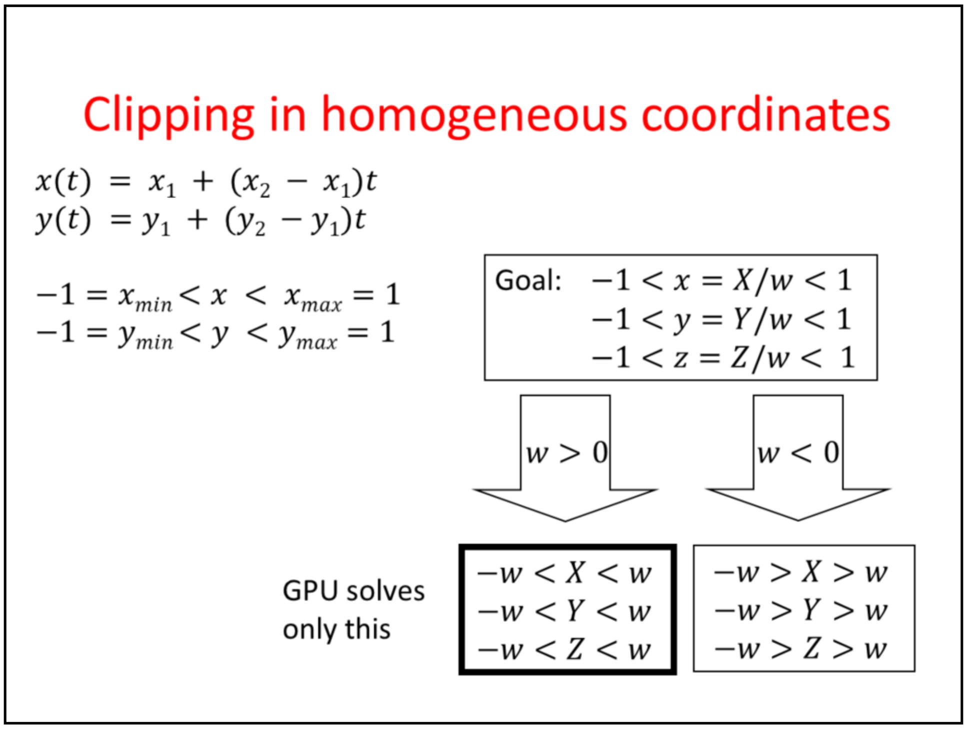
Insert the first point at the end as well.

p[2]
p[3]
p[4]
q[2]
q[3]
p[5]
p[1]
p[0]
q[4]
q[1]
q[0]

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 another array of vertices q and number of vertices in it m.

Usually, we can assume that the edge i 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]. Either the last point should be handled in a special way, or we can store the first element once again at the end to avoid overindexing the array.



Clipping in homogeneous coordinates

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

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

Goal: $-1 < x = X/w < 1$
$-1 < y = Y/w < 1$
$-1 < z = Z/w < 1$

$w > 0$ → $-w < X < w$, $-w < Y < w$, $-w < Z < w$

$w < 0$ → $-w > X > w$, $-w > Y > w$, $-w > Z > w$

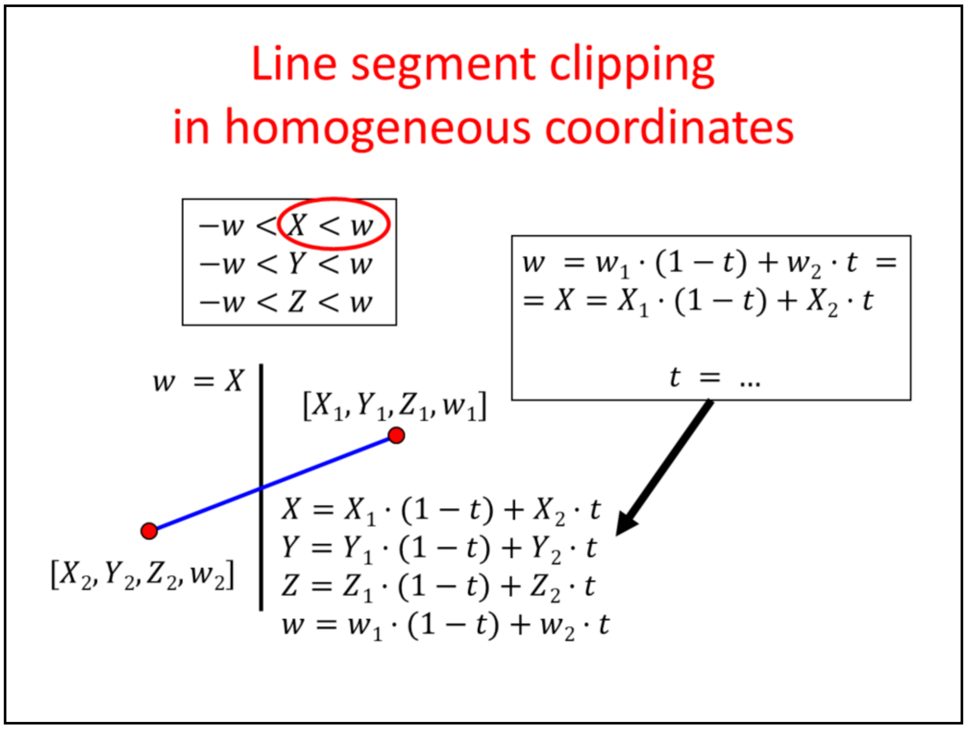
GPU solves only this

In Cartesian coordinates, the limits of the clipping are -1 and 1. We shall adopt this requirement for the third coordinate as well when clipping is extended to 3D. Transformation to normalized device space is a homogeneous linear transformation, which may be non affine, so it may happen that the result is in real homogeneous form where the forth homogeneous coordinate is not 1 anymore. To prepare for this general case, clipping operation is executed in homogeneous coordinates. So we should find the equation of the clipping volume in homogeneous coordinates. Substituting Cartesian coordinate X by X_h/h, etc. these equations can be obtained. To make it simpler we wish to multiply both sides by the fourth homogeneous coordinate h. However, an inequality cannot be multiplied by an unknown variable since should this variable be negative, the relations must be negated. The further calculations should be done separately for the positive h and for the negative h case.

Alternatively, we add requirement h>0 and consider only one case, which simplifies the problem and OpenGL also applies this simplification.

This requirement seems to be arbitrary but is typically true in 2D graphics when homogeneous coordinates are obtained by extending Cartesian coordinates with an extra value 1, i.e. h is indeed positive. In 3D, h>0 corresponds to the requirement the volume must be in front of the eye, so we can accept this extra requirement in 3D too.

The collection of six inequalities defines a cube. A point is inside the cube if all inequalities are met. We clip onto 6 half-spaces one after the other. The intersection of these half-spaces is the cubical view frustum. Each half space is associated with a single inequality and the border plane of the half-space is defined by the equation where < is replaced by =.



Line segment clipping in homogeneous coordinates

$$\begin{array}{l} -w < \boxed{X < w} \\ -w < Y < w \\ -w < Z < w \end{array}$$

$$w = w_1 \cdot (1-t) + w_2 \cdot t = \\ = X = X_1 \cdot (1-t) + X_2 \cdot t$$

$$t = \dots$$

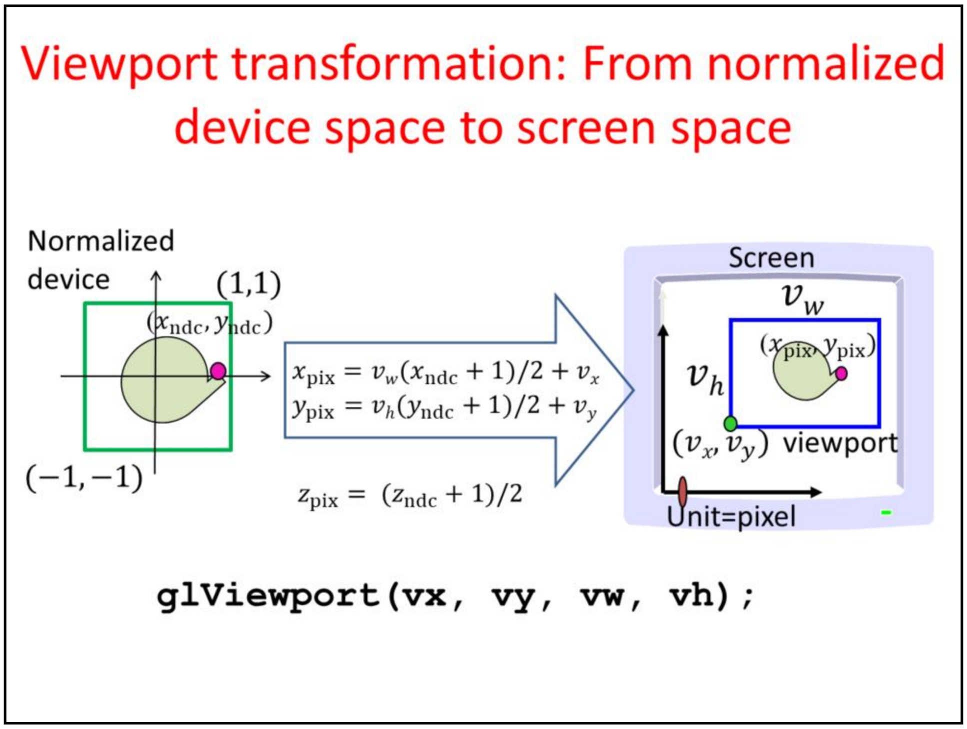
$w = X$

$[X_1, Y_1, Z_1, w_1]$

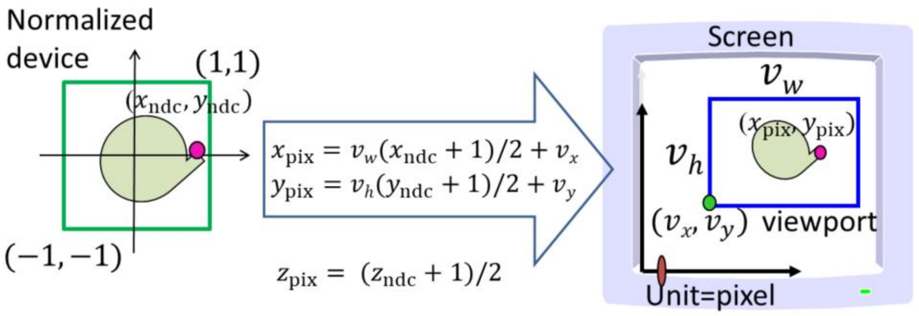
$[X_2, Y_2, Z_2, w_2]$

$$\begin{array}{l} X = X_1 \cdot (1-t) + X_2 \cdot t \\ Y = Y_1 \cdot (1-t) + Y_2 \cdot t \\ Z = Z_1 \cdot (1-t) + Z_2 \cdot t \\ w = w_1 \cdot (1-t) + w_2 \cdot t \end{array}$$

We consider here just one half-space of inequality $X_h < h$, whose boundary is the plane of equation $X_h = h$. The half-space inequality is evaluated for both endpoints. If both of them are in, the line segment is completely preserved. If both of them are out, the line segment is completely ignored. If one is in and the other is out, we consider the equation of the boundary plane ($X_h = h$) and the equation of the line segment (a line segment is the convex combination of its two endpoints), and solve this for unknown combination parameter *t*. Substituting the solution back to the equation of the line segment, we get the homogeneous coordinates of the intersection point. This intersection point replaces the endpoint that has been found outside.



Viewport transformation: From normalized device space to screen space



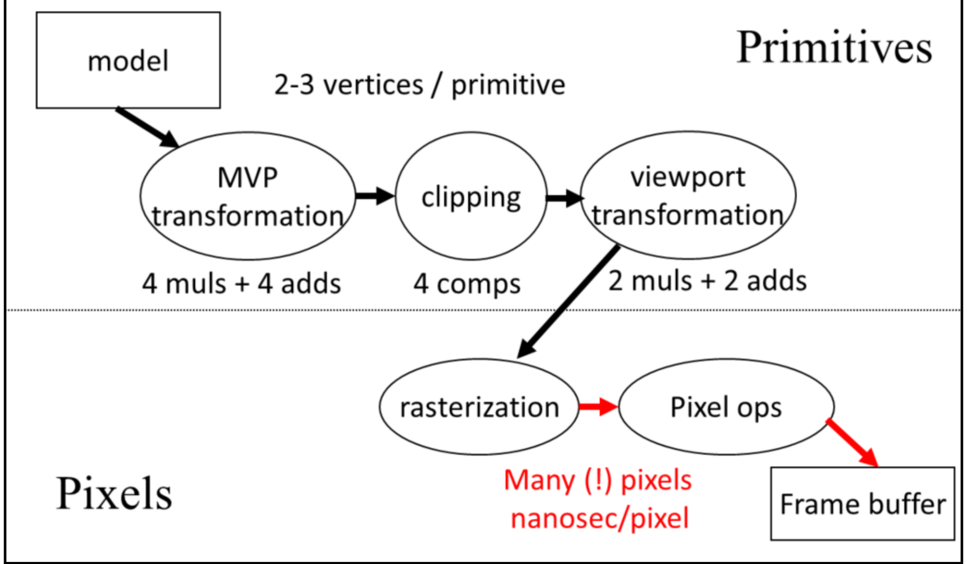
$$x_{\mathrm{pix}} = v_w(x_{\mathrm{ndc}} + 1)/2 + v_x$$
$$y_{\mathrm{pix}} = v_h(y_{\mathrm{ndc}} + 1)/2 + v_y$$
$$z_{\mathrm{pix}} = (z_{\mathrm{ndc}} + 1)/2$$

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

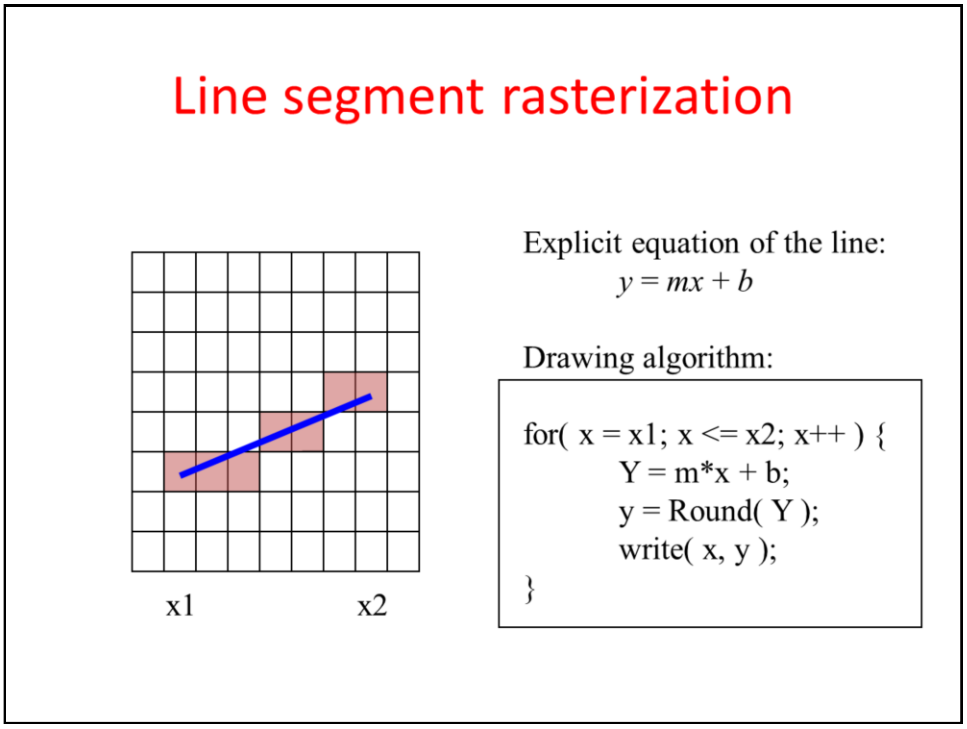
Having executed the clipping step homogenous coordinates are converted to Cartesian coordinates. Because of clipping, all three Cartesian coordinates are in [-1, 1]. The last transformation maps this normalized device space to real pixel space so that (-1,-1) goes to the lower left corner of the viewport and (1,1) to the upper right corner. OpenGL transforms the third coordinate as $Z = (z_c + 1)/2$, i.e. the [-1,1] interval is mapped onto [0,1].

The viewport lower left corner, width and height are specified by the **glViewport** OpenGL function.

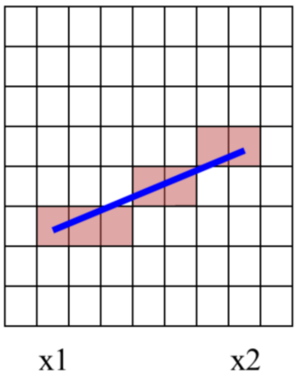
Rasterization (GPU)



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.



Line segment rasterization



Explicit equation of the line:

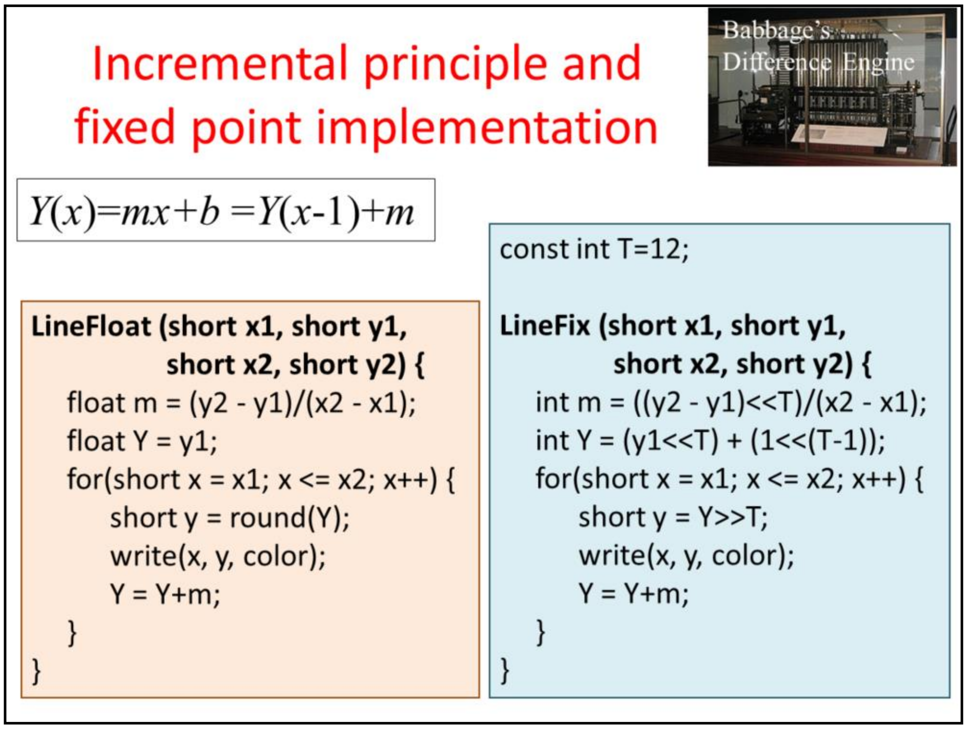
$$y = mx + b$$

Drawing algorithm:

```
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 are too much for a few nanoseconds. So we modify this algorithm preserving its functionality but getting rid of the complicated operations.



Incremental principle and fixed point implementation

$$Y(x)=mx+b=Y(x\text{-}1)+m$$

```
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;
    }
}
```

```
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;
    }
}
```

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 rather than directly 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 use 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.

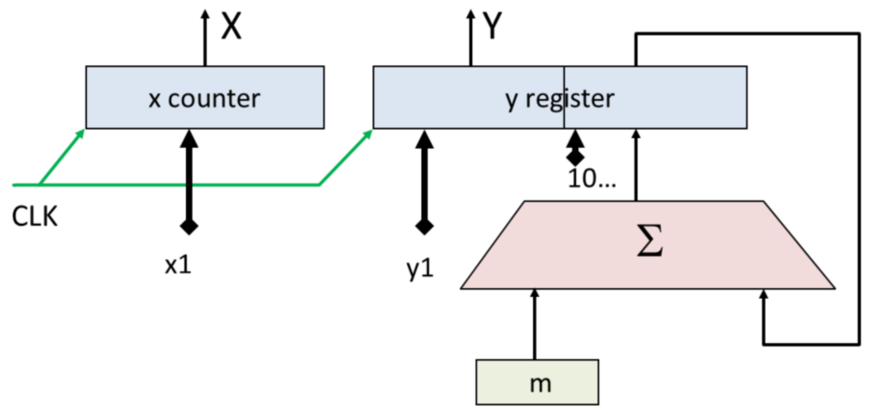
The number of fractional bits can be determined from the requirement that even the longest iteration must be correct. If the number of fractional bits is T, the error caused by the finite fractional part is 2^{-T} in a single addition. If errors are accumulated, the total error in the worst case is N 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 concept 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 value 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.

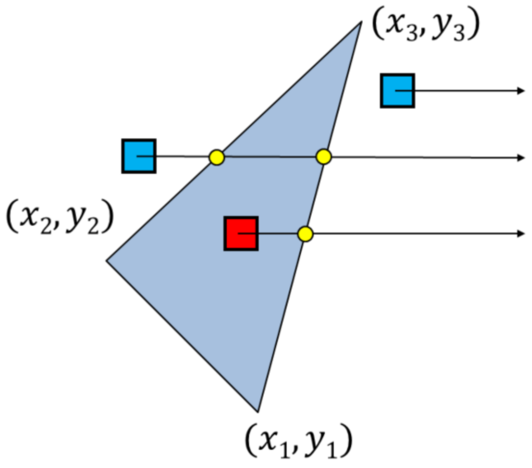
If fixed point representation is used, we shift m and y by T number of bits and rounding ignores the low T bits.

DDA line rasterization hardware

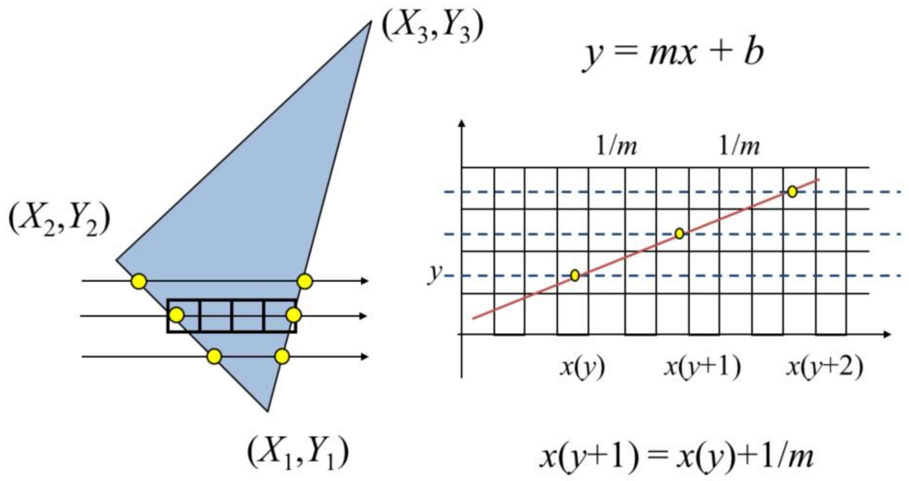


This algorithm can be implemented in hardware with a simple counter that generates increasing x values for every clock cycle. For y we use a register that stores both its fractional and integer parts. The y coordinate is incremented by m for every clock cycle.

Naive triangle fill



Incremental triangle fill



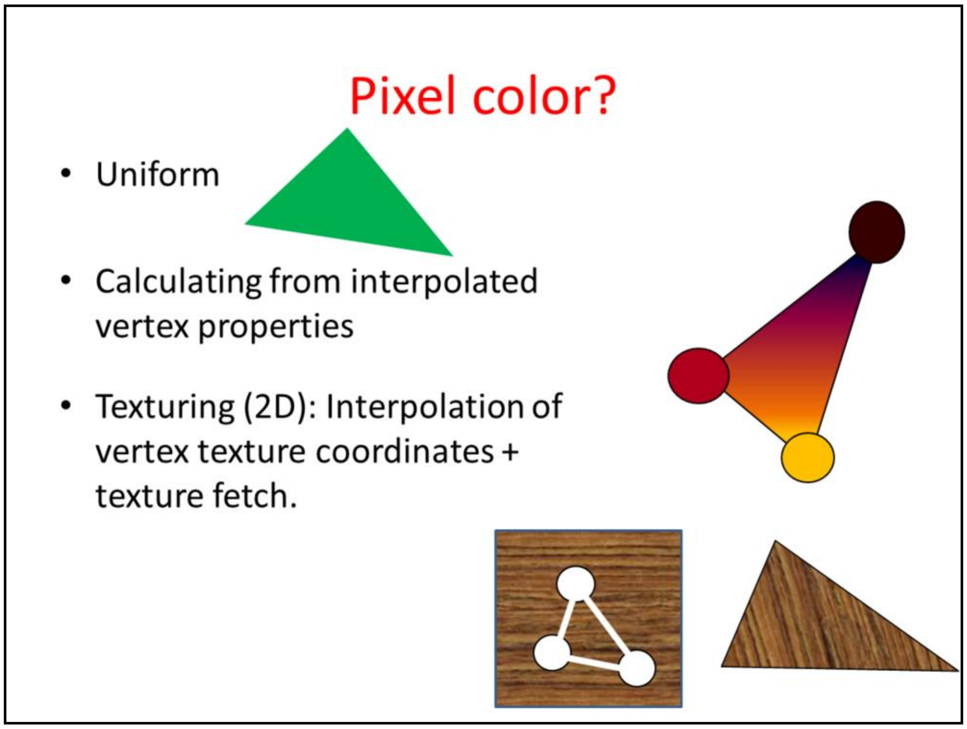
$$y = mx + b$$

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

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 the y coordinate 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.



Pixel color?

- Uniform
- Calculating from interpolated vertex properties
- Texturing (2D): Interpolation of vertex texture coordinates + texture fetch.

Rasterization selects those pixels that belong to an object. The only remaining task is to obtain a color and write it into the selected pixel. There are different options to find the color. It can be uniform for all points.

Alternatively, 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.

Excercises

- Prove that any polygon of at least 4 vertices has a diagonal!
- Prove the two ears theorem!
- Does it make sense to develop a circle rasterization algorithm?
- Write a polygon filling algorithm that can handle even non-simple polygons (the boundary can be built of multiple polylines and may intersect itself).
- Implement the learnt clipping and rasterization algorithms!
- Write a program that decides whether an axis aligned window may contain any primitive.
- Give the class diagram of a PowerPoint like program.
- What is the reason behind the introduction of normalized device space?