MUEI/MUNR
ETSEIB

Course on Medical Imaging
2023-2024
Q1

Session 6
Vectorial images

Dani Tost

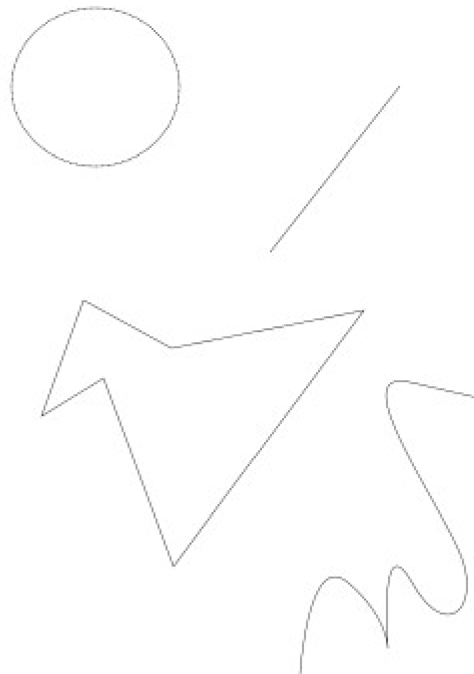
Vectorial images

Rasterization

The visualization pipeline

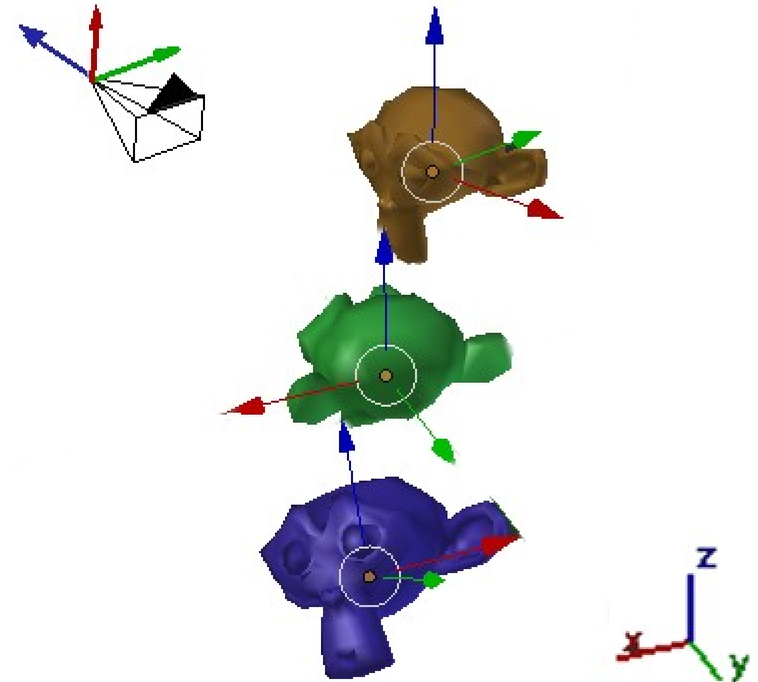
Vectorial images

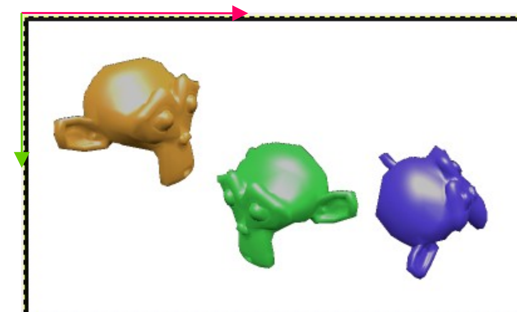
Vectorial images are made of points, lines, polygons and different types of curves. The coordinates of the points are generally expressed in a 2D cartesian coordinate system.



Coordinate systems

- World (global) coordinate system. It defines the global position and orientation of the objects
- Object (local) coordinate system. It defines the local position and orientation of an object.
- Camera coordinate system. It defines the coordinate system defined by the observer's posftion, the viewing direction (z) and the orientation of the area of projection
- Graphical area raster coordinate system: it defines the 2D coordinate system of the graphical area in which the scene is rendered. Measurements in pixels (int).





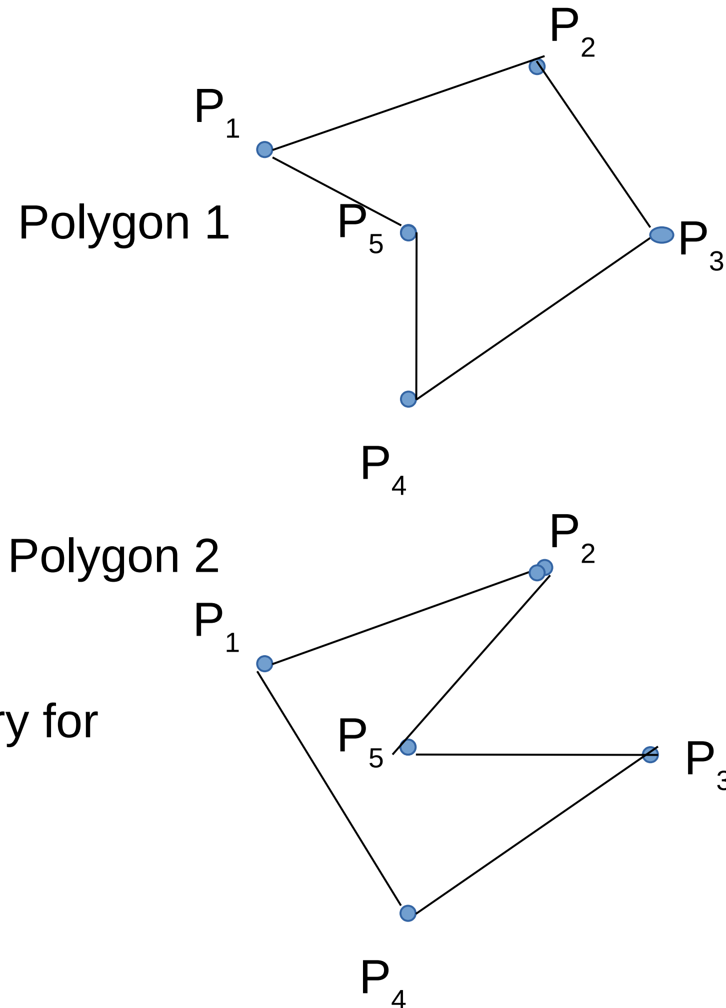
Topology and geometry

Geometry: point coordinates, normal vector components
Topology: connectivity and associativity between elements

Geometry

Vertices list

x1, y1
x2, y2
x3, y3
x4, y4
x5, y5



Same geometry for the different polygons

Edge list

Topology

Polygon1

P1	P2
P2	P3
P3	P4
P4	P5
P5	P1

Polygon2

P1	P2
P2	P5
P5	P3
P3	P4
P4	P1

Different edge lists for the different polygons

Vectorial image files

Open the file `test-vim2.json`. It contains an ascii description of a vectorial image made by a circle, a rectangle and a polygon. Try to understand it.

```
[
   {
        "type": "circle",
        "value": {
            "center": {"x":-4, "y":-4},
            "radius": 2,
            "color": [255, 255, 25]
         }
   },
   {
        "type": "rectangle",
        "value": {
           "center": {"x":4, "y":5},
           "width": 8,
           "height": 4,
           "color": [0, 255, 255]
         }
   },
    {
        "type": "polygon",
        "value": {
            "color": [230, 255, 155],
            "vertices": [{"x": 0, "y":0}, {"x":10, "y": -2}, {"x":5, "y":-4}]
        }
    }

]
```

The file is written in json format, a light format to read dictionaries without having to bother about text processing.

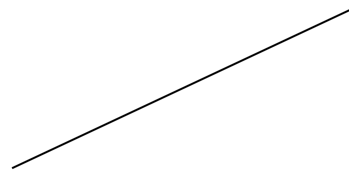
`Circle` and `Rectangle` are procedural models (you don't specify vertices but attributes)

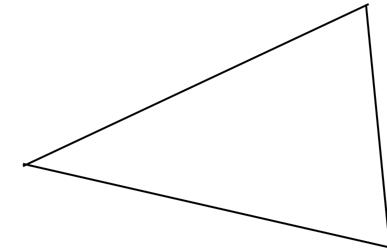
In `Polygon`, the topology is implicit.

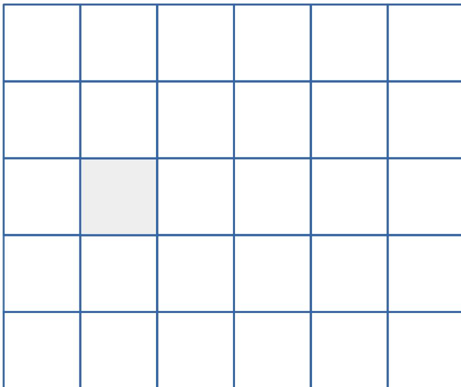
Rasterization

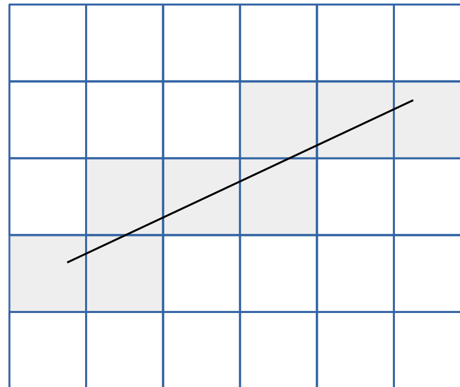
Rasterization consists of converting vectorial information into raster information: a raster image or a graphical area of a raster display.

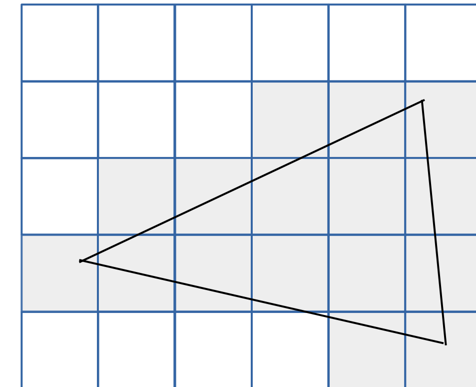
.









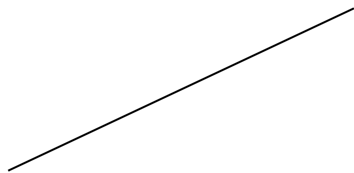


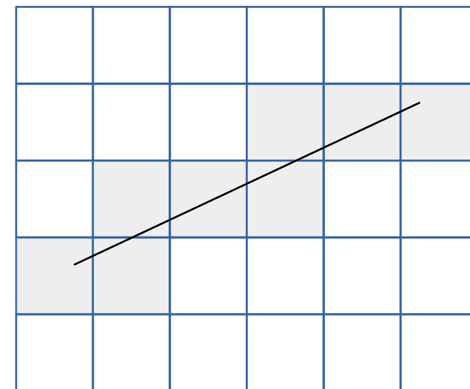
Rasterization

Rasterization of a point → a pixel

Rasterization of a line → the set of pixels that approximate better the line

Rasterization of a polygon → the set of pixels that are inside or on the rasterization of the edges of the polygon





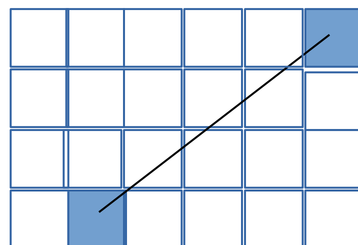
Rasterization

The principle of rasterization of lines is the DDA (Digital differential analyzer) or Breseenham's algorithm.

The rasterization is done by the GPU.

All graphical libraries provide methods to draw lines and polygons in raster coordinates

The `draw` module of `skimage` allows to rasterize geometry **expressed in the raster coordinate system**



Line by line wonder which pixels should be painted according to the line inclination

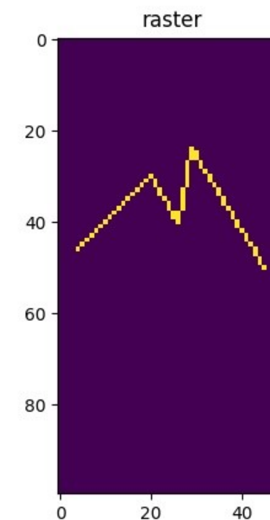
Rasterization

Rasterization consists of converting vectorial information into raster information.

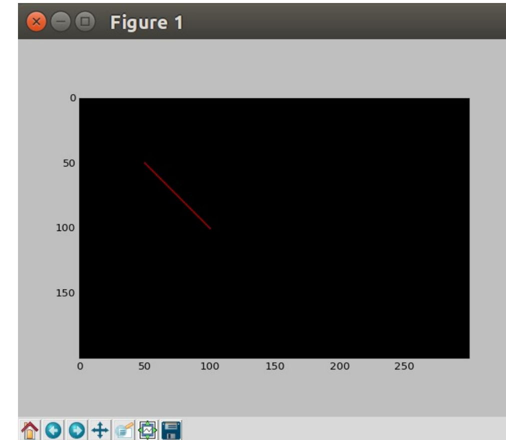
The module skimage.draw provides different functions to rasterize geometric primitives https://scikit-image.org/docs/dev/api/skimage.draw

Module: draw

Function	Description
`skimage.draw.line` (r0, c0, r1, c1)	Generate line pixel coordinates.
`skimage.draw.line_aa` (r0, c0, r1, c1)	Generate anti-aliased line pixel coordinates.
`skimage.draw.line_nd` (start, stop, *[, ...])	Draw a single-pixel thick line in n dimensions.
`skimage.draw.bezier_curve` (r0, c0, r1, c1, ...)	Generate Bezier curve coordinates.
`skimage.draw.polygon` (r, c[, shape])	Generate coordinates of pixels within polygon.
`skimage.draw.polygon_perimeter` (r, c[, ...])	Generate polygon perimeter coordinates.
`skimage.draw.ellipse` (r, c, r_radius, c_radius)	Generate coordinates of pixels within ellipse.
`skimage.draw.ellipse_perimeter` (r, c, ...[, ...])	Generate ellipse perimeter coordinates.
`skimage.draw.ellipsoid` (a, b, c[, spacing, ...])	Generates ellipsoid with semimajor axes aligned with grid dimensions on grid with specified *spacing*.
`skimage.draw.ellipsoid_stats` (a, b, c)	Calculates analytical surface area and volume for ellipsoid with semimajor axes aligned with grid dimensions of specified *spacing*.
`skimage.draw.circle` (r, c, radius[, shape])	Generate coordinates of pixels within circle.
`skimage.draw.circle_perimeter` (r, c, radius)	Generate circle perimeter coordinates.
`skimage.draw.circle_perimeter_aa` (r, c, radius)	Generate anti-aliased circle perimeter coordinates.
`skimage.draw.set_color` (image, coords, color)	Set pixel color in the image at the given coordinates.
`skimage.draw.random_shapes` (image_shape, ...)	Generate an image with random shapes, labeled with bounding boxes.
`skimage.draw.rectangle` (start[, end, extent, ...])	Generate coordinates of pixels within a rectangle.



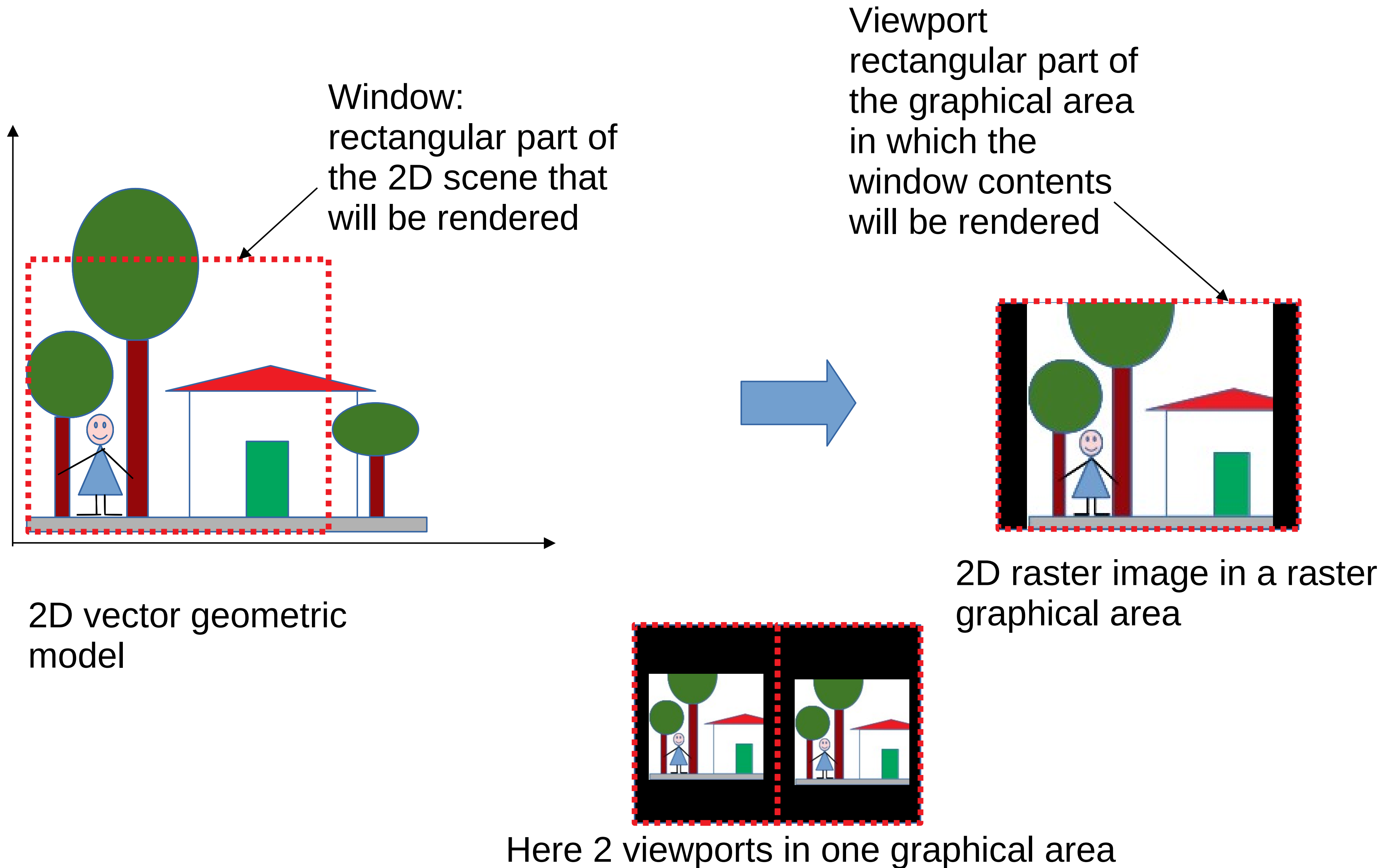
Rasterization



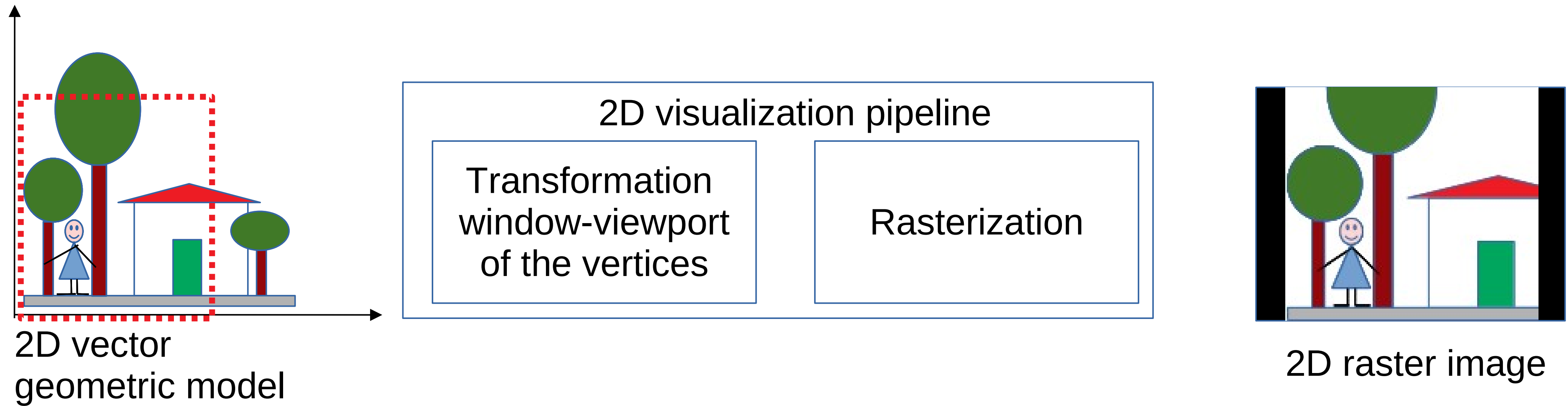

```
>>> import numpy as np
>>> a = np.zeros((200, 300, 3), dtype = np.uint8
>>> import skimage
>>> from skimage.draw import line
>>> line = skimage.draw.line(50, 50, 100, 100)
>>> line
(array([ 50,  51,  52,  53,  54,  55,  56,  57,  58,  59,  60,  61,  62,
     63,  64,  65,  66,  67,  68,  69,  70,  71,  72,  73,  74,  75,
     76,  77,  78,  79,  80,  81,  82,  83,  84,  85,  86,  87,  88,
     89,  90,  91,  92,  93,  94,  95,  96,  97,  98,  99, 100]), array([ 50,  51,  52,  53,  54,  55,  56,  57,  58,  59,  60,
  61,  62,
     63,  64,  65,  66,  67,  68,  69,  70,  71,  72,  73,  74,  75,
     76,  77,  78,  79,  80,  81,  82,  83,  84,  85,  86,  87,  88,
     89,  90,  91,  92,  93,  94,  95,  96,  97,  98,  99, 100]))  It is a doble array (x, y) of the coordinate of the
  rasterized pixels
>>> a[line[0], line[1] = (255, 0, 0)
```

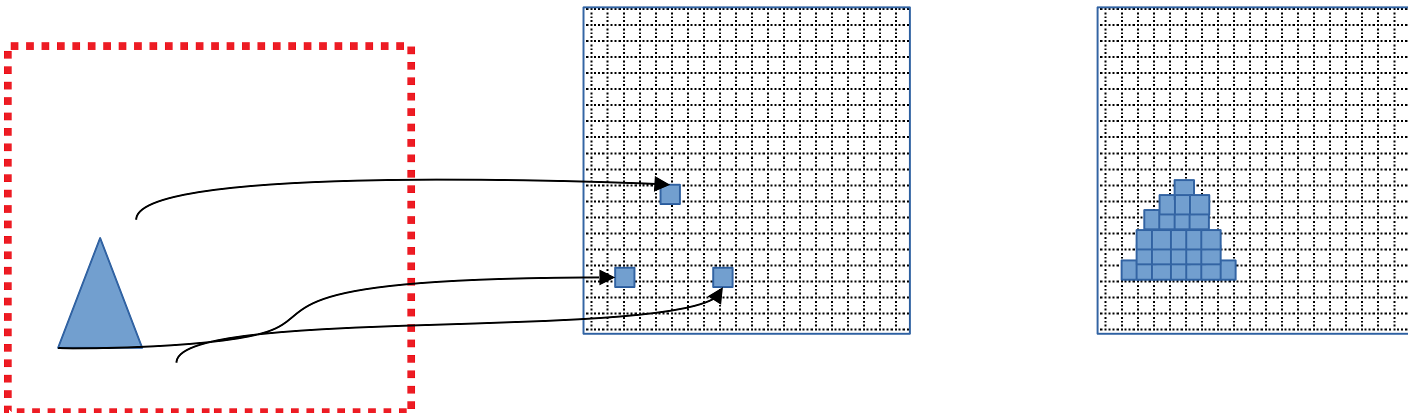
Observe that the geometry is expressed in the raster coordinate system

2D visualization pipeline



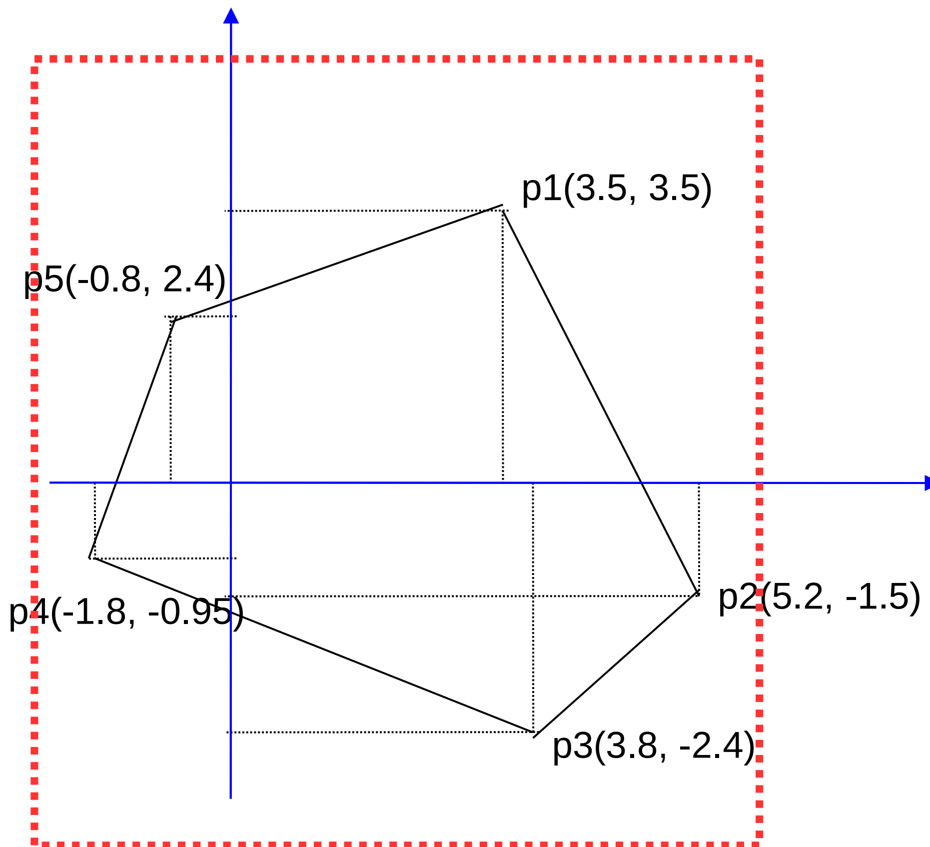
2D visualization pipeline

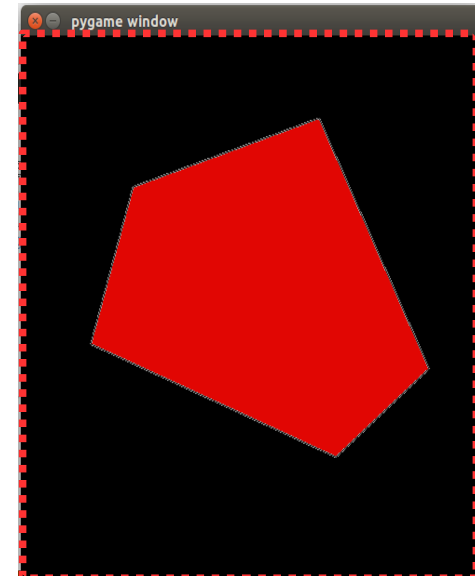




Fill the pixels “inside” the geometry

2D visualization pipeline





- Define the geometric model of the polygon: geometry, topology
- Specify what part of the plane we would like to render: window

- Create the graphical window
- Define the graphical area: viewport
- Transform the vertices to pixels
- Fill the polygon: rasterize

Rasterization of points

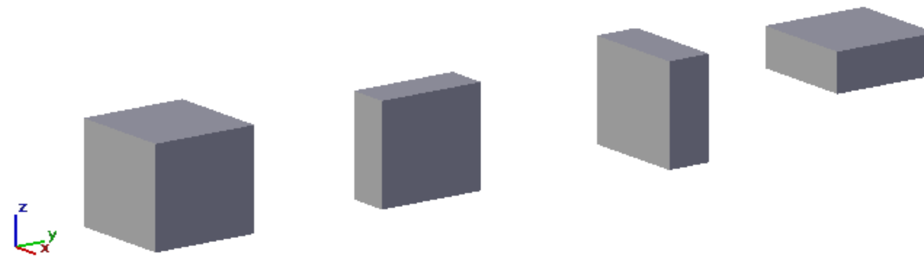
The first step to rasterize thus is to rasterize the vertices of the vectorial geometry. After, we'll the inside.

To rasterize vertices we will need to apply geometrical transformations as explained next.

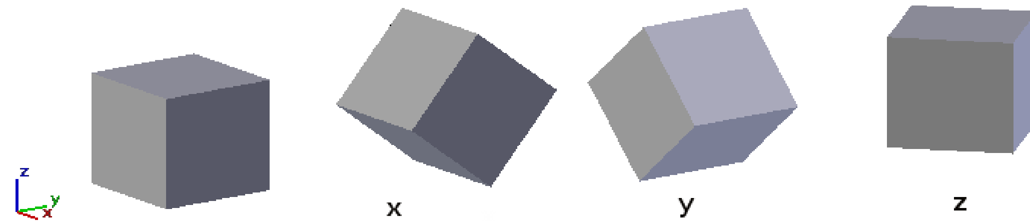
Geometric transformations

Geometric transformations are needed to animate the objects and project them for rendering.

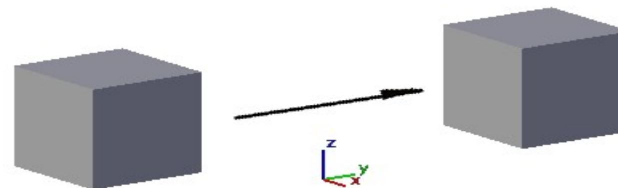
Scale



Rotation



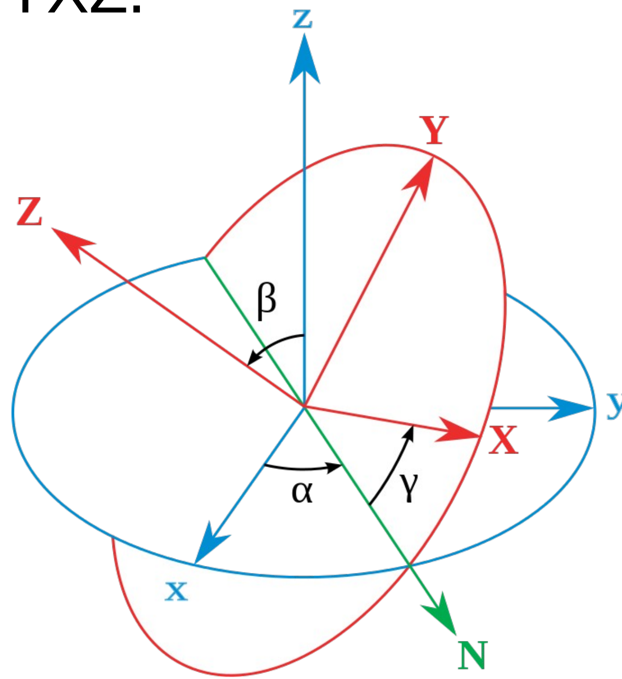
Translation



Geometric transformations

Arbitrary rotations can be expressed as a composition of elementary rotations around the axis x, y, z of an Eulerian coordinate system.

In general, arbitrary rotations are defined by three angles in the world coordinate system. The composition is not commutative: the order matters! In general YXZ.

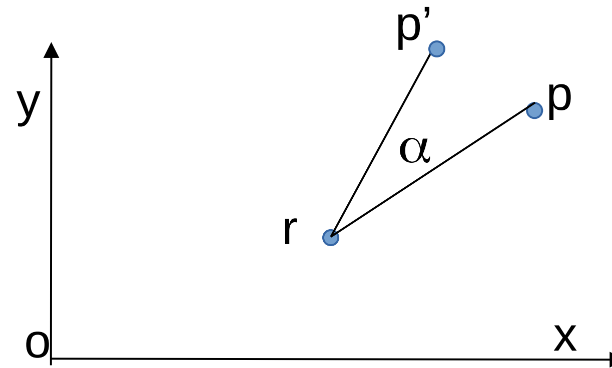


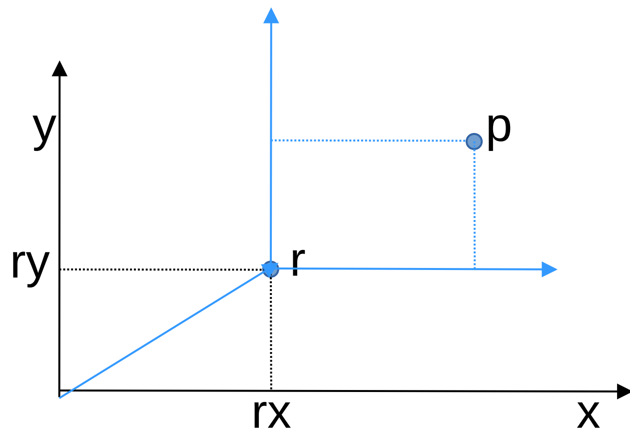
From wikipedia, https://en.wikipedia.org/wiki/Euler_angles

Geometrical transformations

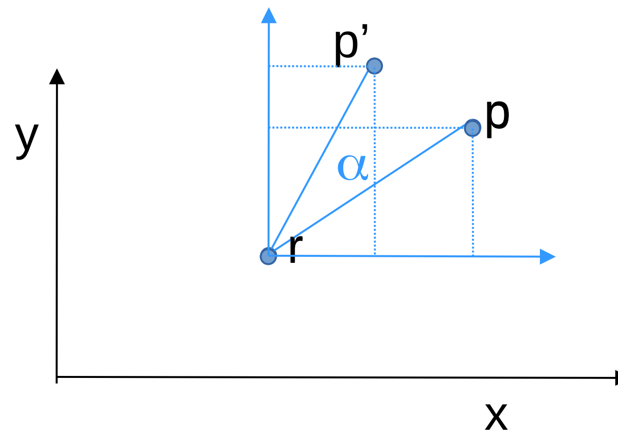
Complex transformations can be expressed as a sequence of simple transformations.

Example: rotation of α of point p with center r

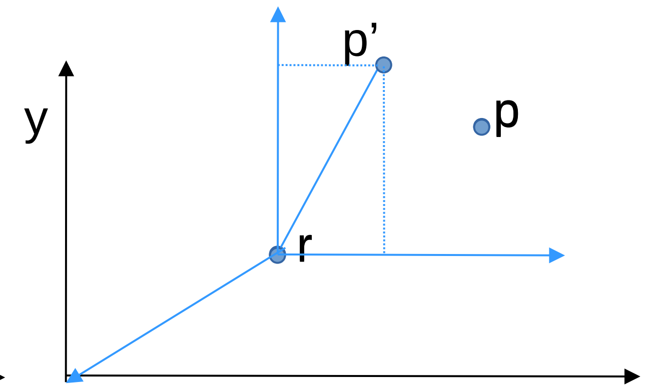




Step 1
Change of coordinate system center from o to r. Equivalent to Translation of (-rx, -ry)



Step 2
Rotation of a with center (0, 0) in the new coordinate system



Step 3
Change of coordinate system center from r to o. Equivalent to Translation of (rx, ry)

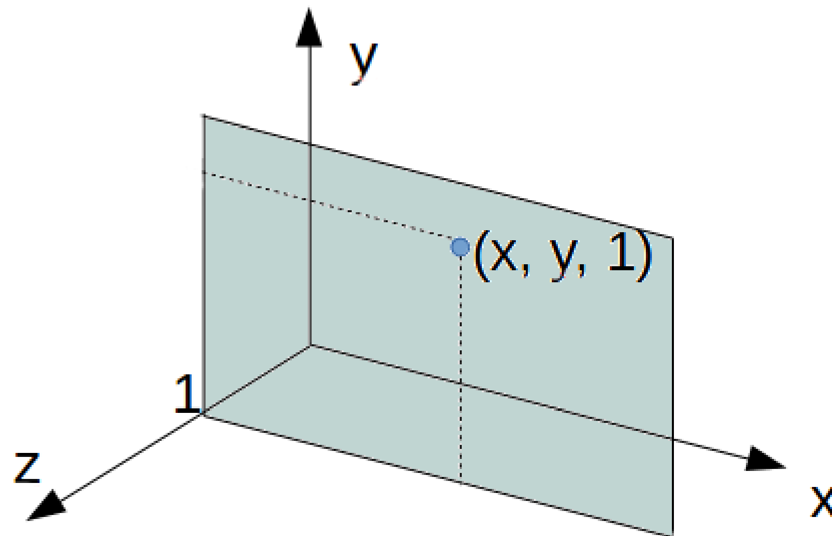
Matrices

Geometric transformations can be expressed as matrices. In order to handle all kinds of transformations, matrices are expressed in a (+1) higher dimension that the workspace (3x3 matrices for 2D (4x4 matrices for 3D).

Vertices are expressed in homogeneous coordinates:

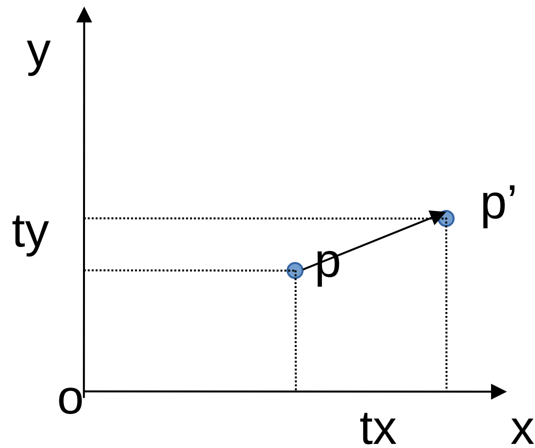
2D: $(x, y) \rightarrow (x, y, 1)$

3D: $(x, y, z) \rightarrow (x, y, z, 1)$



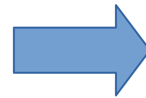
Matrices

Example 2D: translation of point p of (tx, ty)



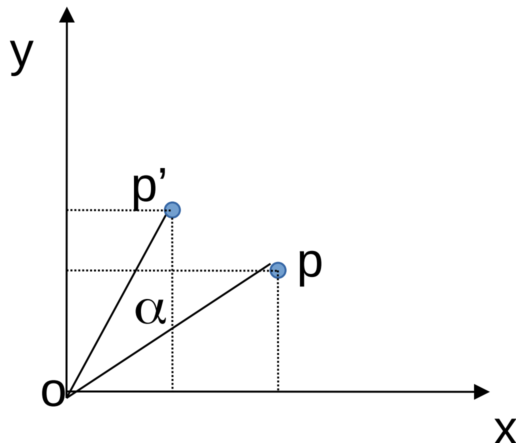
p(x, y)
p'(x', y')

x' = x + tx
y' = y + ty



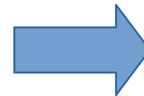
$$\begin{pmatrix} x' \\ y' \\ h \end{pmatrix} = \begin{pmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Example 2D: rotation of α of point p with centre (0, 0)



p(x, y)
p'(x', y')

x' = x cosα - y sinα
y' = x sinα + y cosα



$$\begin{pmatrix} x' \\ y' \\ h \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

2D Matrices

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Identity

$$\begin{pmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{pmatrix}$$

Translation

$$\begin{pmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Scale

$$\begin{pmatrix} \cos(d) & -\sin(d) & 0 \\ \sin(d) & \cos(d) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Rotation

Matrices

A sequence of transformations can expressed as a one transformation with a matrix result of the concatenation of matrices of the simple transformations

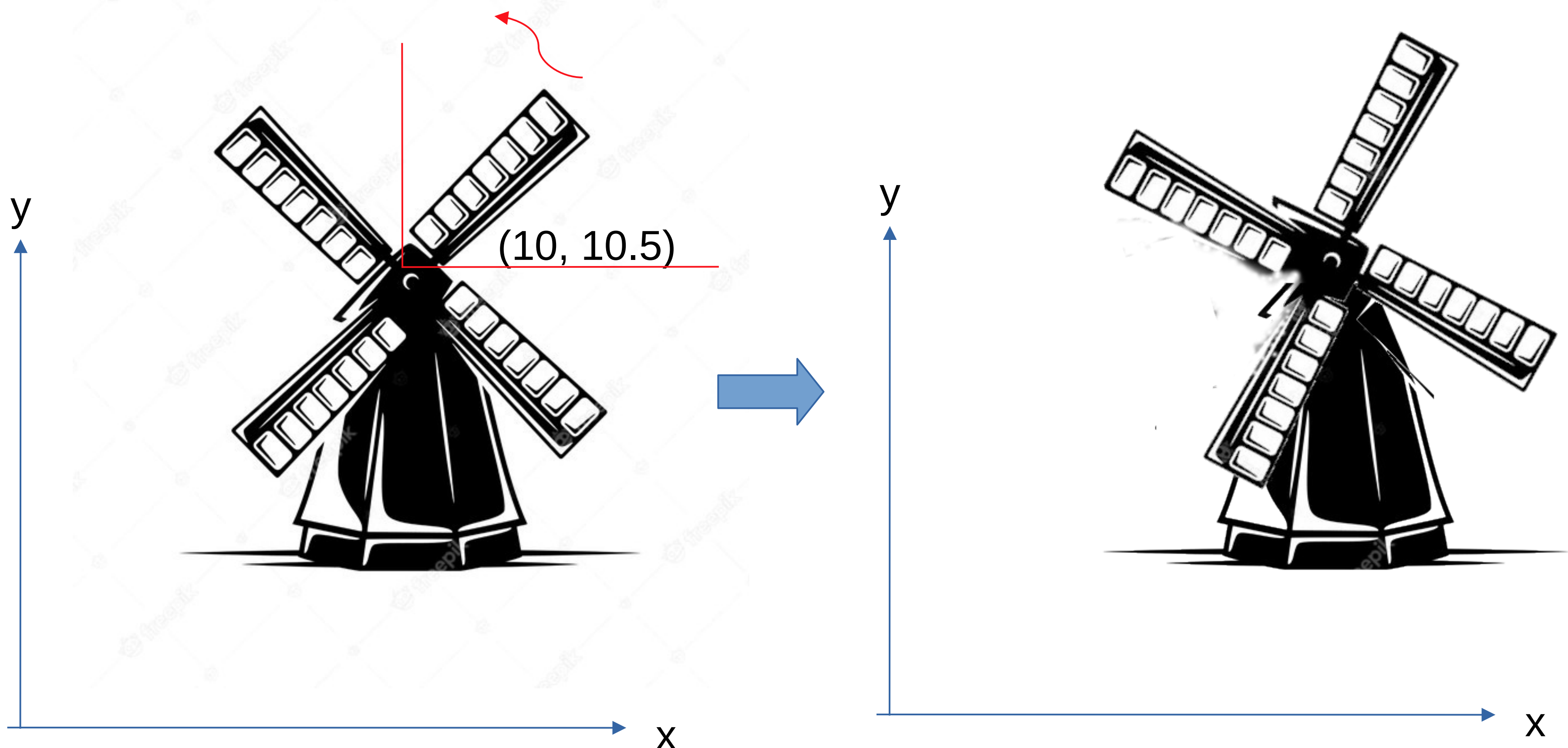
Example: rotation of α of point p with center r

$$\begin{pmatrix} x' \\ y' \\ h \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & -rx \\ 0 & 1 & -ry \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & rx \\ 0 & 1 & ry \\ 0 & 0 & 1 \end{pmatrix}} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

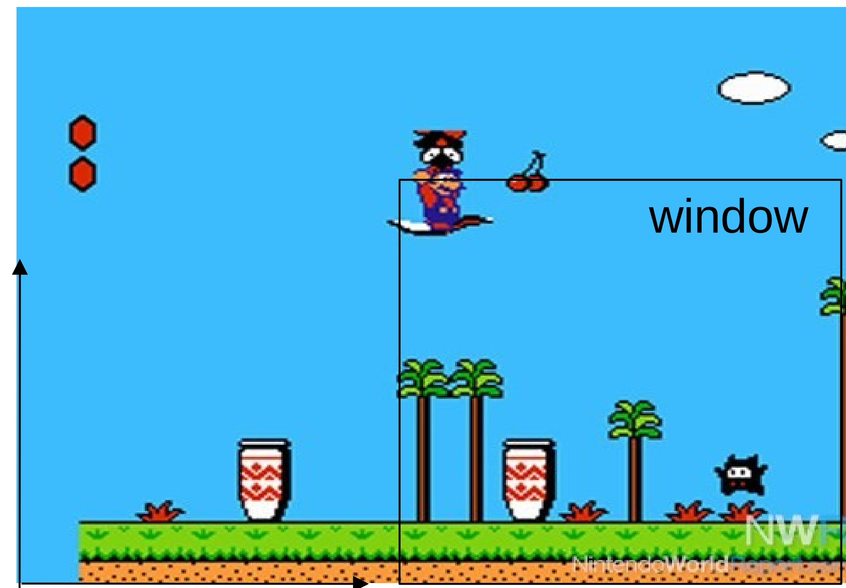
$$\begin{pmatrix} x' \\ y' \\ h \end{pmatrix} = \begin{pmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31 & m32 & m33 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Example

Write the matrices needed to make a rotation of 30º of the wings of the windwill around their center.



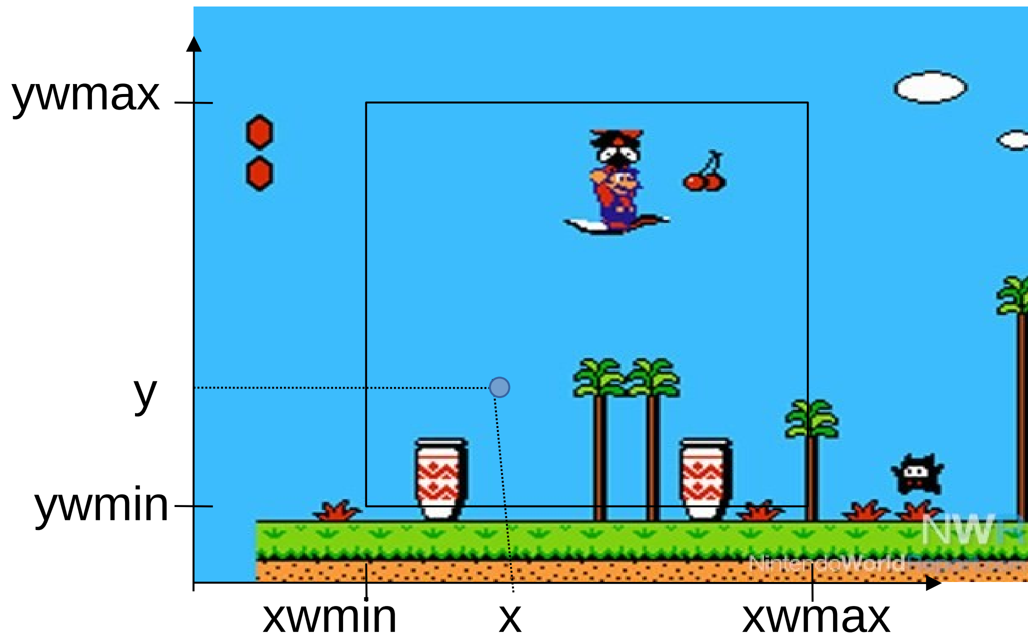
The 2D visualization pipeline
The scene and the window

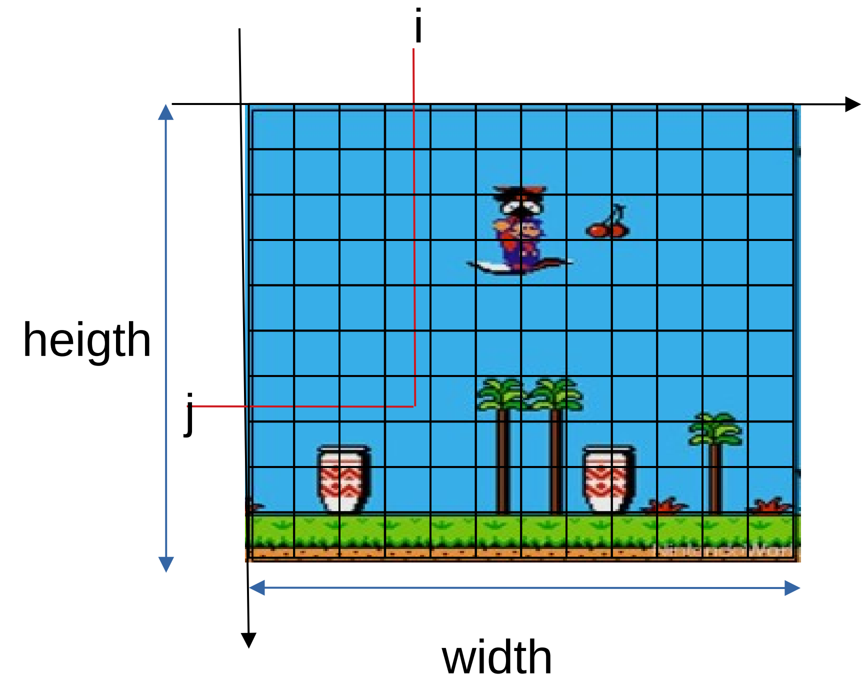


2D World coordinate system

To rasterize a scene, you should first define **the window**: the bounding box of the piece of 2D world you want to render. Then, you should define the **window-to-viewport transformation** and convert all the vertices into the raster coordinate system. Then you can rasterize using `skimage.draw` (or whatever needed).

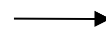
The 2D visualization Pipeline Window-Viewport transformation





Mapping equation in x:

$$\frac{x - xwmin}{xwmax - xwmin} = \frac{i}{width}$$



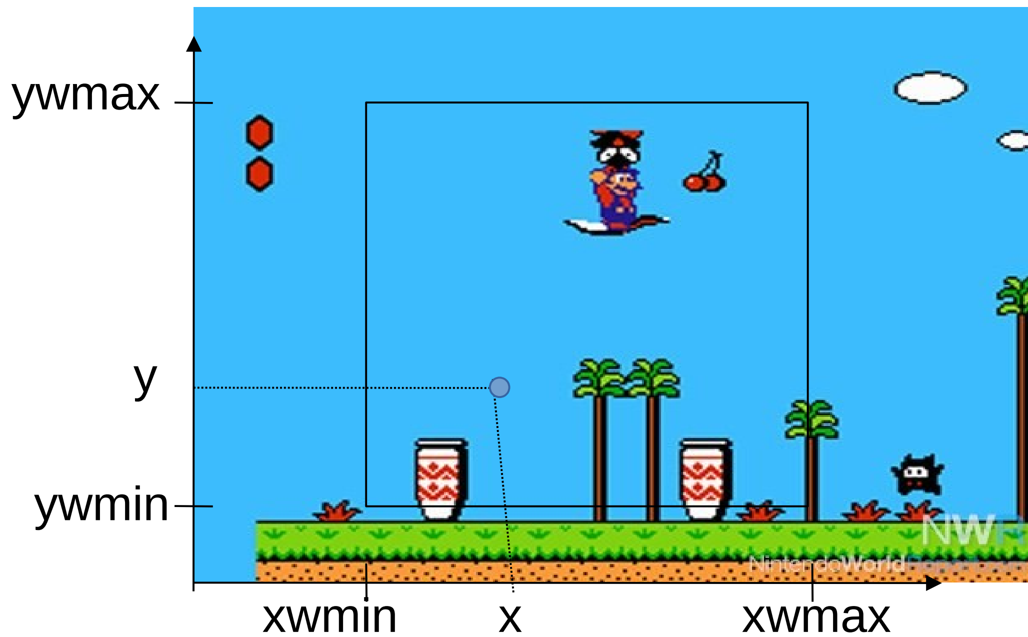
$$i = \frac{width}{xwmax - xwmin}(x - xwmin)$$

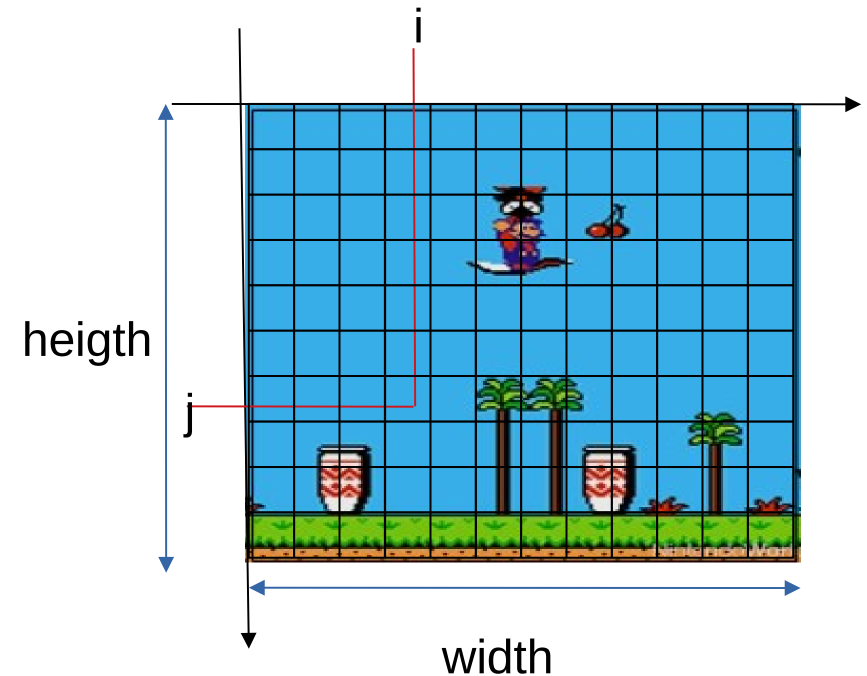
Transformation: $$i = (integer)(Sx \cdot x + Tx)$$

$$Sx = \frac{width}{xwmax - xwmin}$$

$$Tx = -Sx \cdot xwmin$$

The 2D visualization Pipeline Window-Viewport transformation

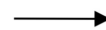




Mapping equation in y:

$$\frac{y - ywmin}{ywmax - ywmin} = \frac{height - j}{height} \longrightarrow j = height - \frac{height}{ywmax - ywmin} * (y - ywmin)$$

Transformation: $j = (integer)(Sy \cdot y + Ty)$



$$Sy = \frac{-\,height}{ywmax - ywmin}$$

$$Ty = height - Sy \cdot ywmin$$

Matrices

The window-to-viewport transformation can be expressed as a matrix:

Mapping equation:

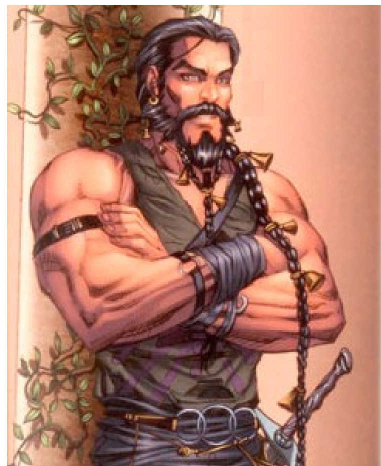
$$\frac{x - xwmin}{xwmax - xwmin} = \frac{i - xvpmin}{xvpmax - xvpmin} \quad \text{and} \quad \frac{y - ywmin}{ywmax - ywmin} = \frac{(yvpmax - j) - yvpmin}{yvpmax - yvpmin}$$

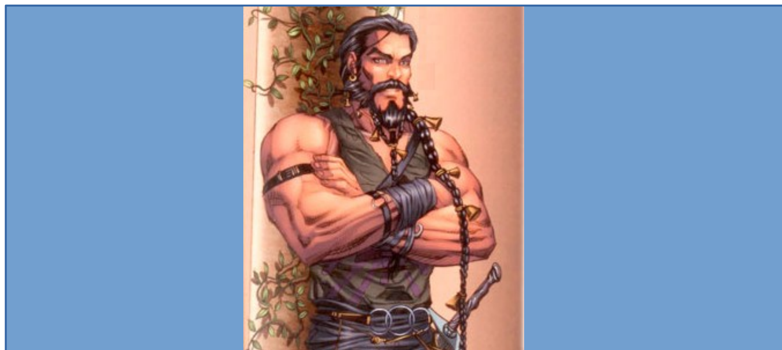
Transformation: $i = int(Sx\ x + Tx)$ $j = int(Sy\ y + Ty)$

$$MWV = \begin{pmatrix} Sx & 0 & Tx \\ 0 & Sy & Ty \\ 0 & 0 & 1 \end{pmatrix}$$

2D Pipeline
Window-Viewport transformation

Deformation





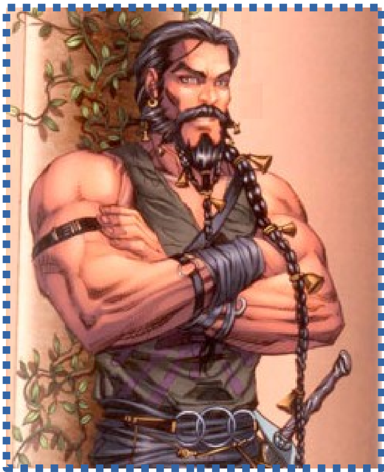
Aspect ratio = width/height

If the aspect ratio of the window is not equal to that of the viewport a deformation can occur

2D Pipeline
Window-Viewport transformation

Deformation

To avoid deformations, the window size should be recomputed to match the viewport aspect ratio and still fit the area that we want to see: make it wider o larger.



Desired window

Viewport

Computed window: same aspect ratio as the viewport but still fits all the scene adding margins in one or the other direction

Next week ... 3D