Visual Computing

2024/2025

Class 4

2D Visualization and Transformations

deti departamento de electrónica, telecomunicações e informática

universidade de aveiro

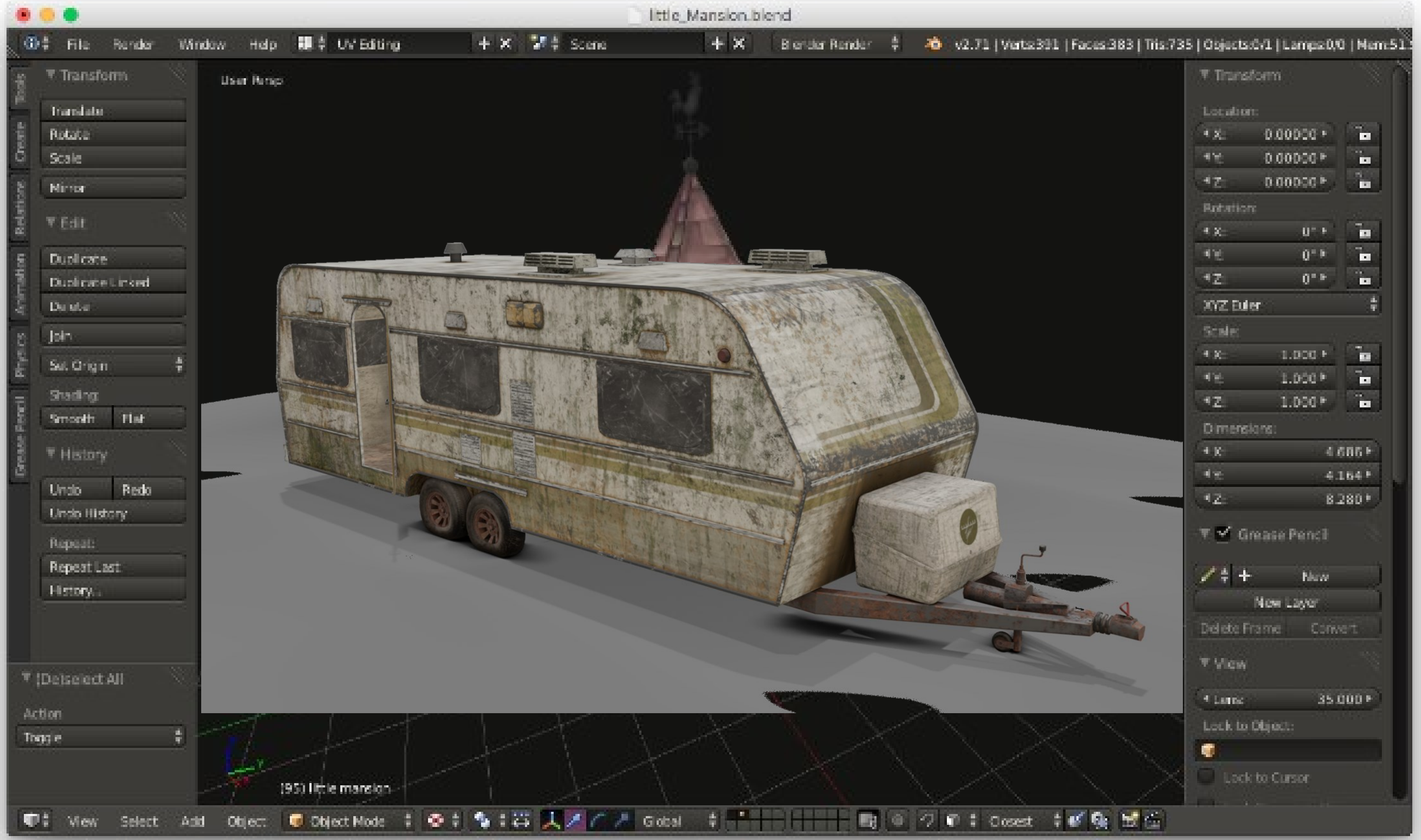
Today's Agenda

- 2D visualization pipeline
- 2D Transformations
- Translation / Rotation / Scaling
- Homogeneous Coordinates
- Concatenating Transformations

2D Visualization pipeline

From models to a scene viewed on screen

Modelling assets is just ONE stage



A World Needs to be Created

A Point of View needs to Be Chosen

2D transformations

2D Transformations

- **Position, orientation** and **scaling** for objects in XOY
- Basic transformations
  - **Translation / Displacement**
  - **Rotation** relative to the coordinates' origin
  - **Scaling** relative to the coordinates' origin
- Representation using **matrices**
  - **Homogeneous coordinates**
- Complex transformations
  - **Decompose** into a sequence of basic transformations

Basic 2D transformations

The basic transformations are:

Translation / Displacement

Scaling

Rotation

Some nomenclature:

$p = (x, y)$ → *original, given point*

$p' = (x', y')$ → *transformed point*

Column vector representation

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

2D translation

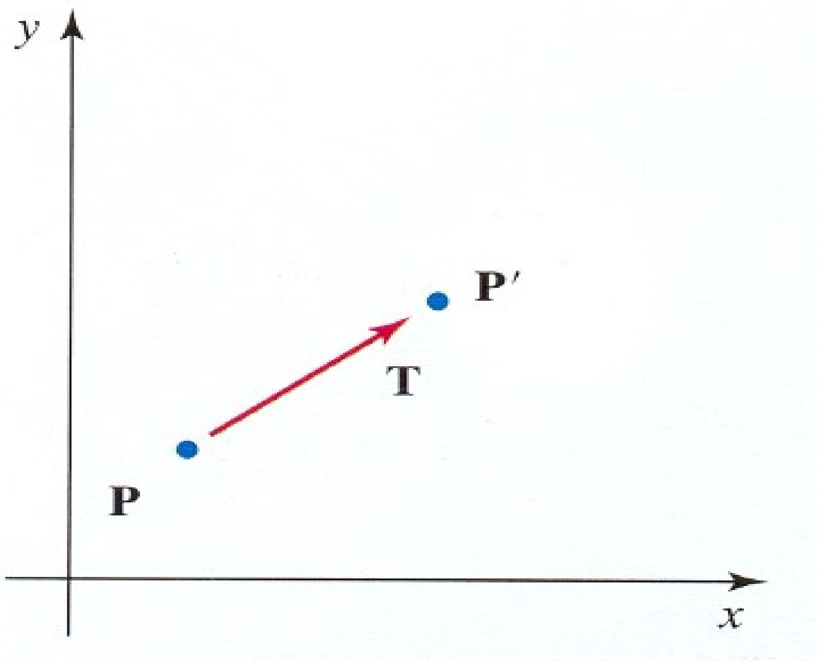
Translation

To translate a point we need the displacement values in *x* and *y*

$$x' = x + t_x, \qquad y' = y + t_y$$

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \qquad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \qquad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$\mathbf{P}' = \mathbf{P} + \mathbf{T}$$



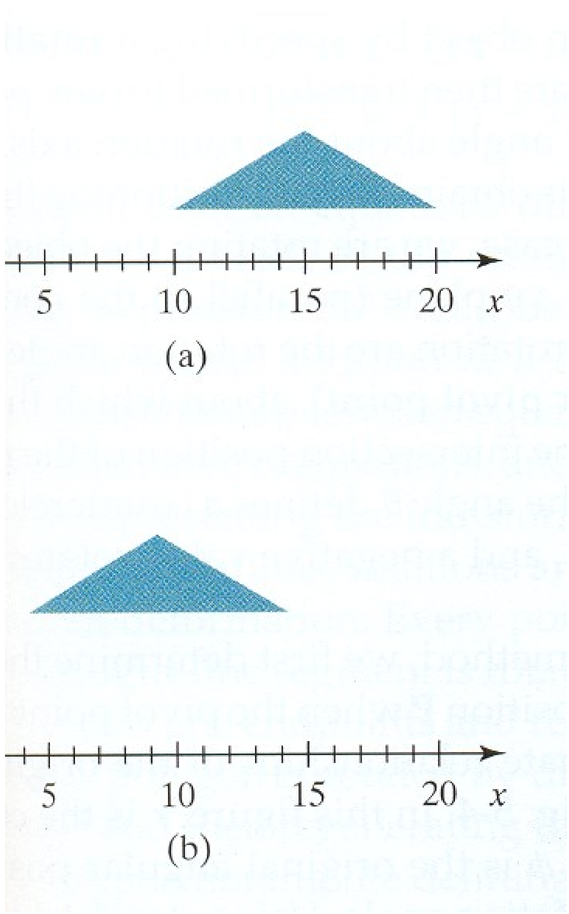
Translation

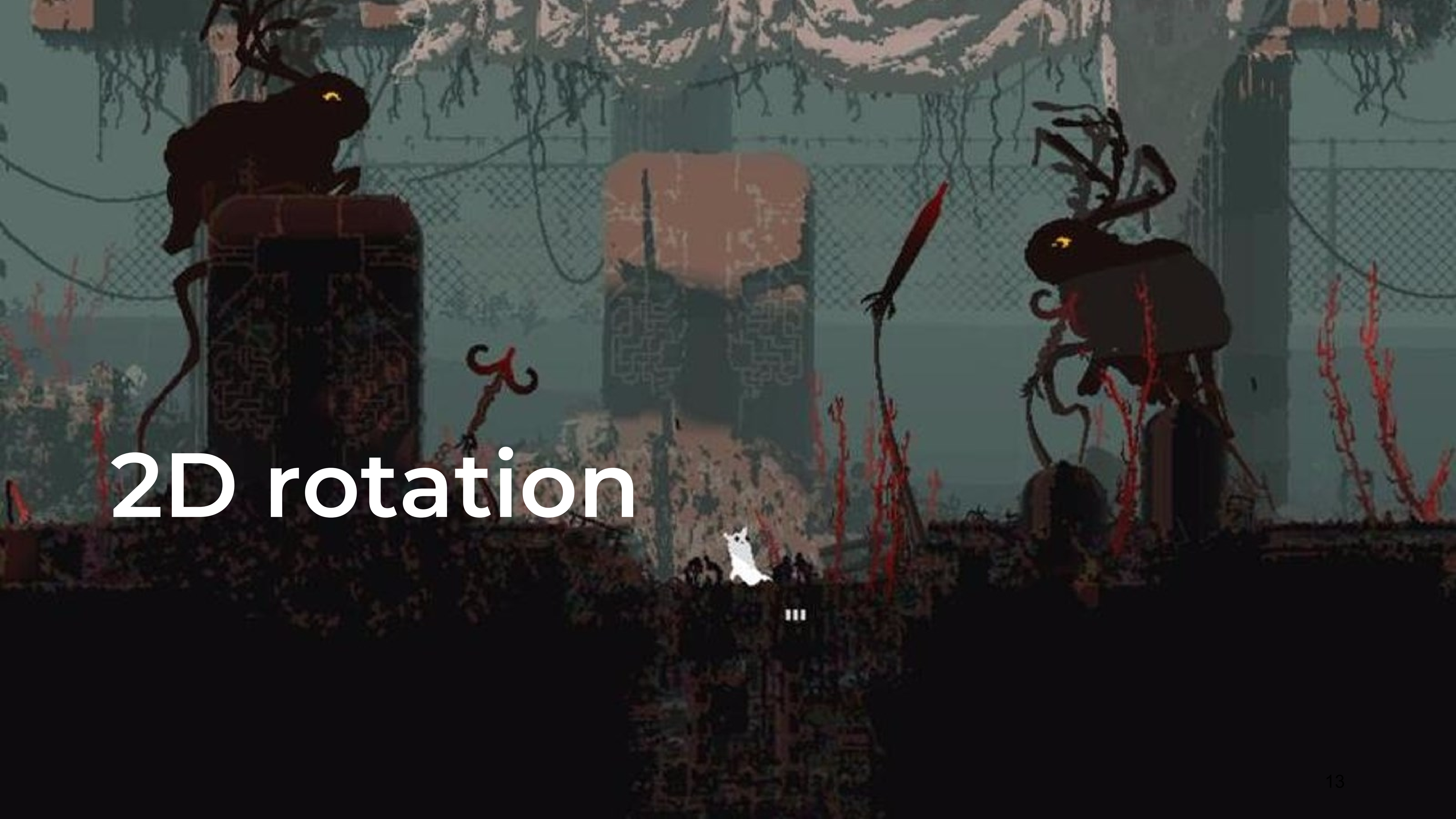
Each object is displaced without any deformation:

it is a **rigid-body** transformation

To displace a straight-line segment, apply the transformation to the **two end-points** and draw the resulting line segment.

To displace a polygon, apply the transformation to the polygon's **vertices**.





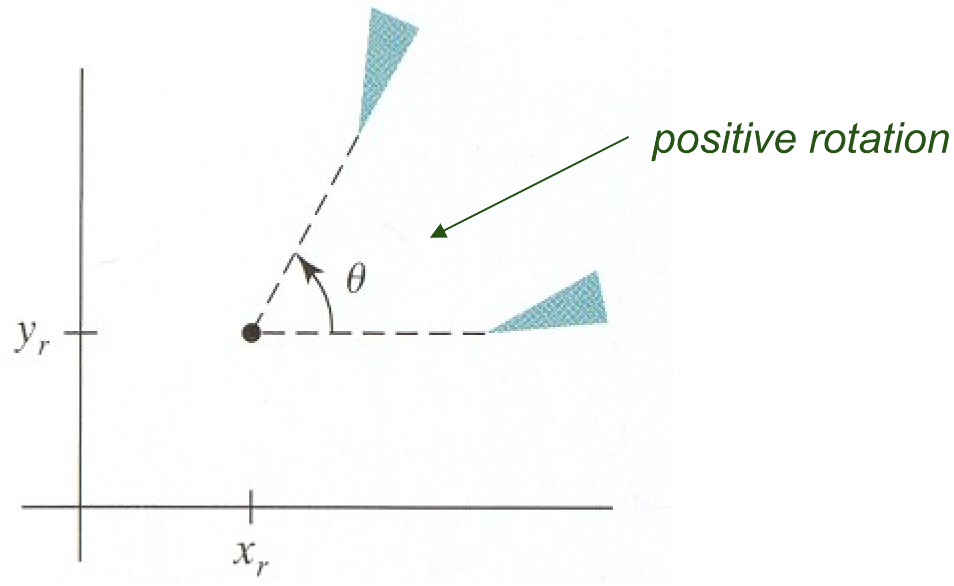
2D rotation

Rotation

To apply a rotation we need:

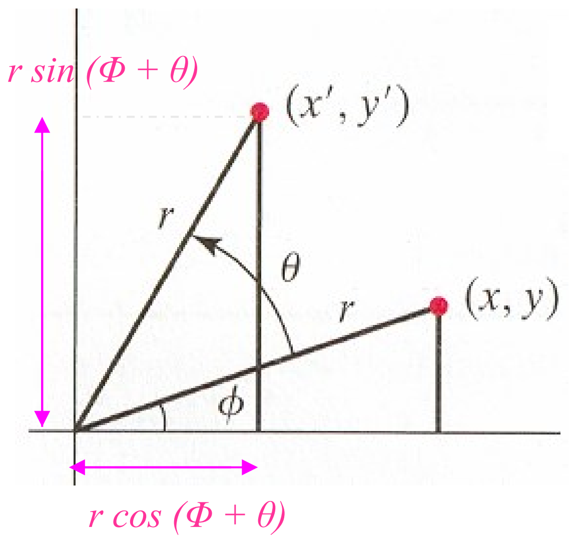
- a point: the **center of rotation** ***(x_r, y_r)***
  (intersection point between a perpendicular rotation axis and XOY)

- a **rotation angle** θ (positive, if counter-clockwise - CCW)



Rotation around the origin

Easier to determine the transformation representing a **rotation around (0,0):**



$$x' = r\cos(\Phi + \theta) = r\cos\Phi\cos\theta - r\sin\Phi\sin\theta$$

$$y' = r\sin(\Phi + \theta) = r\cos\Phi\sin\theta + r\sin\Phi\cos\theta$$

Original point coordinates in **polar coordinates**:

$$x = r\cos\Phi$$
$$y = r\sin\Phi$$

Replacing in the above equations, we get the desired result:

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

Rotation around the origin

$$x' = r\cos(\phi + \theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$y' = r\sin(\phi + \theta) = r\cos\phi\sin\theta + r\sin\phi\cos\theta$$

$$x = r\cos\phi, \qquad y = r\sin\phi$$

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

$$\mathbf{P}' = \mathbf{R}\cdot\mathbf{P}$$ with $$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

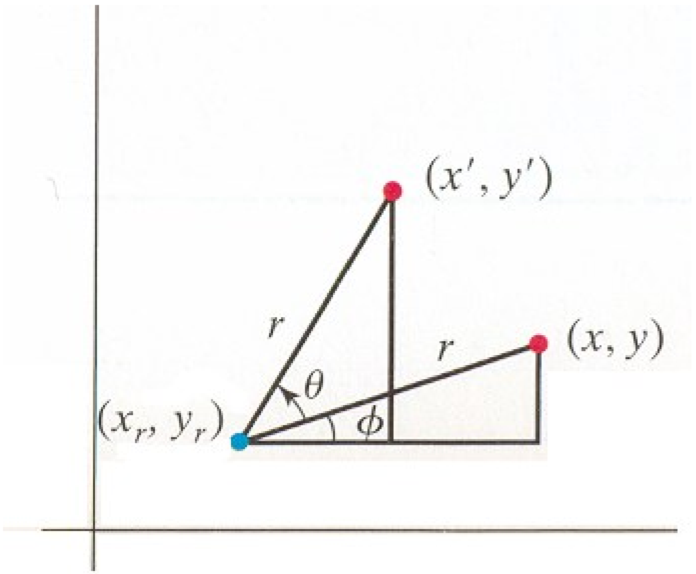
Matrix for rotation around the origin, with angle ***θ***

Rotation around an arbitrary point

Using the figure, the **rotation equations** are obtained as:

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$$

$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta$$



- Rotations are also **rigid-body transformations**
- To rotate a straight-line segment, transform its **end-points** and draw the line segment
- To rotate a polygon, transform its **vertices**

2D scaling

Scaling relative to the origin

The **scaling transformation** is applied to change the size of an object: s_x and s_y are the **scaling factors**.

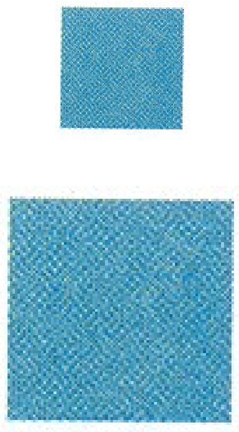
$$x' = x \cdot s_x$$

$$y' = y \cdot s_y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

transformation matrix

$$P' = S \cdot P$$



Obtaining a larger square through a scaling transformation, s_x=2, s_y=2

Scaling

Scaling factors are positive: ***s*** **>0**

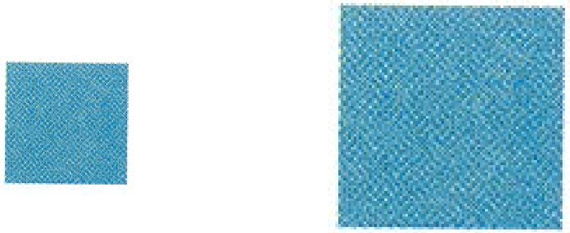
$$x' = x \cdot s_x$$

$$y' = y \cdot s_y$$

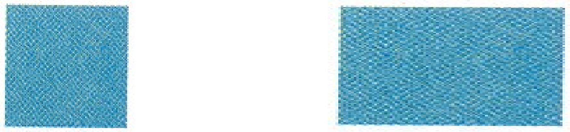
$s_x = s_y$ ⟶ **uniform** scaling

$s_x \neq s_y$ ⟶ **non-uniform** scaling

Obtaining a larger square through a scaling transformation, s_x=2, s_y=2



Transforming a square into a rectangle: the scaling has s_x=2, s_y=1

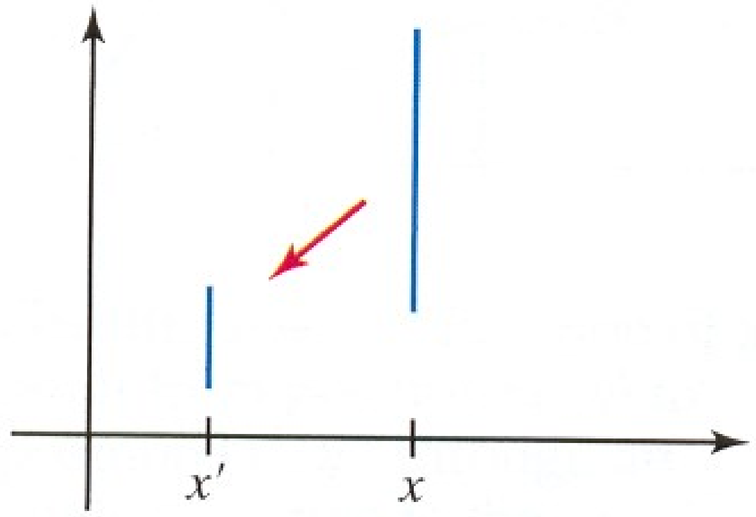


Scaling

The scaled objects are **reposioned** if **not** originally **centered** on the origin:

s < 1 → it will be **closer** to the origin

s > 1 → it will be **farther** from the origin



A straight-line segment becomes shorter and closer to the origin through the scaling $s_x = s_y = 0{,}5$

Efficient computation of transformations

- Most graphical applications apply **sequences of transformations**
    - The visualization transformation corresponds to sequences of translations and rotations to display a given scene
    - An animation might require that an object be displaced and rotated between consecutive frames
- To carry out **sequences of transformations** in an efficient way, each transformation is represented as a **matrix**

Homogeneous Coordinates

The three **basic transformations** can be **represented generally as**:

$$P' = M_1 \,.\, P + M_2$$

M_1 is a **2x2 matrix**

M_2 is a **column vector**, representing the translation vector

A more efficient representation uses **just one matrix** which can **represent all the transformations** in a sequence and is **applied just once** to every point

Such a representation uses **homogeneous coordinates**

homogeneous coordinates

Homogeneous Coordinates

- Every point in **2D** is now represented by **three coordinates**:

$$(x, y) \rightarrow (x_h, y_h, h), \quad h \neq 0$$

$$x = x_h / h \qquad y = y_h / h$$

$$(x.h, y.h, h)$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Homogeneous Coordinates

- **All transformations** are represented by a 3x3 matrix

- The third matrix column represents the displacement (additive) factors

$(x, y) \rightarrow (x_h, y_h, h), \quad h \neq 0$

$x = x_h / h \qquad y = y_h / h$

$(x.h, y.h, h)$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2D transformations using homogeneous coordinates

- When using homogeneous coordinates, **all transformations are carried out by matrix multiplication**

- 2D translation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \mathbf{P}' = \mathbf{T}(t_x, t_y) \cdot \mathbf{P}$$

2D transformations using homogeneous coordinates

- 2D rotation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \mathbf{P}' = \mathbf{R}(\theta) \cdot \mathbf{P}$$

- 2D scaling:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \mathbf{P}' = \mathbf{S}(s_x, s_y) \cdot \mathbf{P}$$

concatenation of transformations

Concatenation of transformations

We compute the matrix for a sequence of transformations by **multiplying the matrices representing the individual transformations**

The product of transformation matrices represents the **concatenation or composition of transformations**.

$$P' = M_2 \,.\, M_1 \,.\, P$$
$$= M \,.\, P$$

first transformation to be applied

The coordinates of the transformed point P' are computed with a **single matrix multiplication**

Concatenation of two translations

$$\begin{aligned}\mathbf{P}' &= \mathbf{T}(t_{2x}, t_{2y}) \cdot \{\mathbf{T}(t_{1x}, t_{1y}) \cdot \mathbf{P}\} \\ &= \{\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y})\} \cdot \mathbf{P}\end{aligned}$$

$$\begin{bmatrix} 1 & 0 & t_{2x} \\ 0 & 1 & t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{1x} \\ 0 & 1 & t_{1y} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{1x} + t_{2x} \\ 0 & 1 & t_{1y} + t_{2y} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y}) = \mathbf{T}(t_{1x} + t_{2x}, t_{1y} + t_{2y})$$

Concatenation of two scalings

$$\begin{bmatrix} s_{2x} & 0 & 0 \\ 0 & s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{1x} & 0 & 0 \\ 0 & s_{1y} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{1x} \cdot s_{2x} & 0 & 0 \\ 0 & s_{1y} \cdot s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{S}(s_{2x}, s_{2y}) \cdot \mathbf{S}(s_{1x}, s_{1y}) = \mathbf{S}(s_{1x} \cdot s_{2x}, \quad s_{1y} \cdot s_{2y})$$

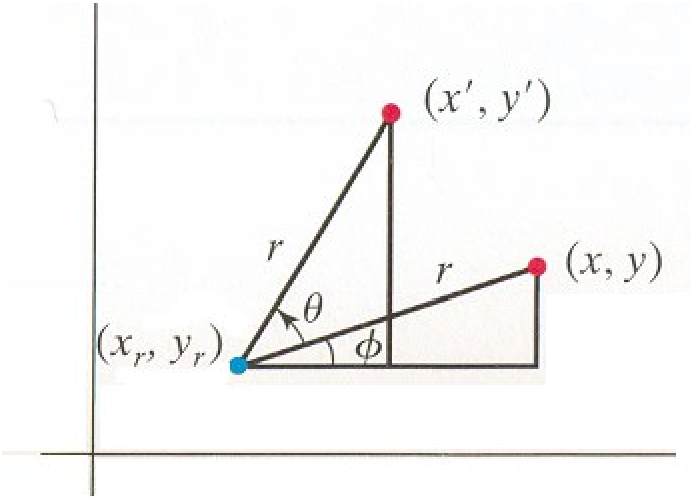
Doing the maths for the x' coordinate

$x' = r\cos(\theta + \Phi) + x_r \qquad x = r\cos\Phi + x_r$

$y' = r\,sen(\theta + \Phi) + y_r \qquad y = r\,sen\,\Phi + y_r$

$x' = r\cos\theta\ \cos\Phi - r\ sen\,\theta\ sen\,\Phi + x_r$

$y' = r\cos\theta\ sen\,\Phi + r\,sen\,\theta\ \cos\Phi + y_r$

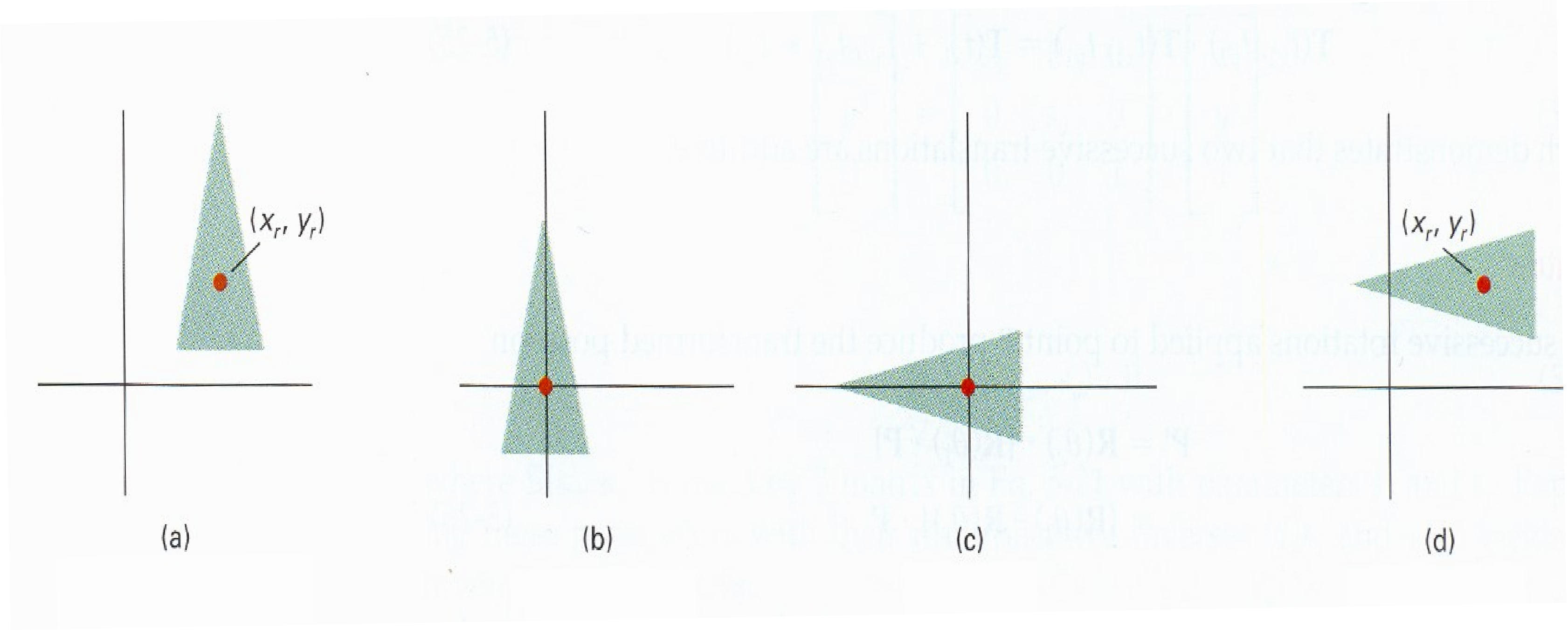


$x' = (x - x_r)\cos\theta - (y - y_r)\ sen\,\theta + x_r$

$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$

This seems to show that we first translate $-x_r$, rotate by θ, and then translate x_r

Rotation around an arbitrary point (x_r, y_r)



Original position of the triangle and point *(x_r, y_r)*

1- A translation moves point *(x_r, y_r)* to the origin

2- Rotation around the origin

3- Inverse translation moves the rotation center back to *(x_r, y_r)*

Rotation around an arbitrary point (x_r, y_r)

translation so that the arbitrary point moves to the origin

rotation around the origin

inverse translation to move the rotation center back to its original position

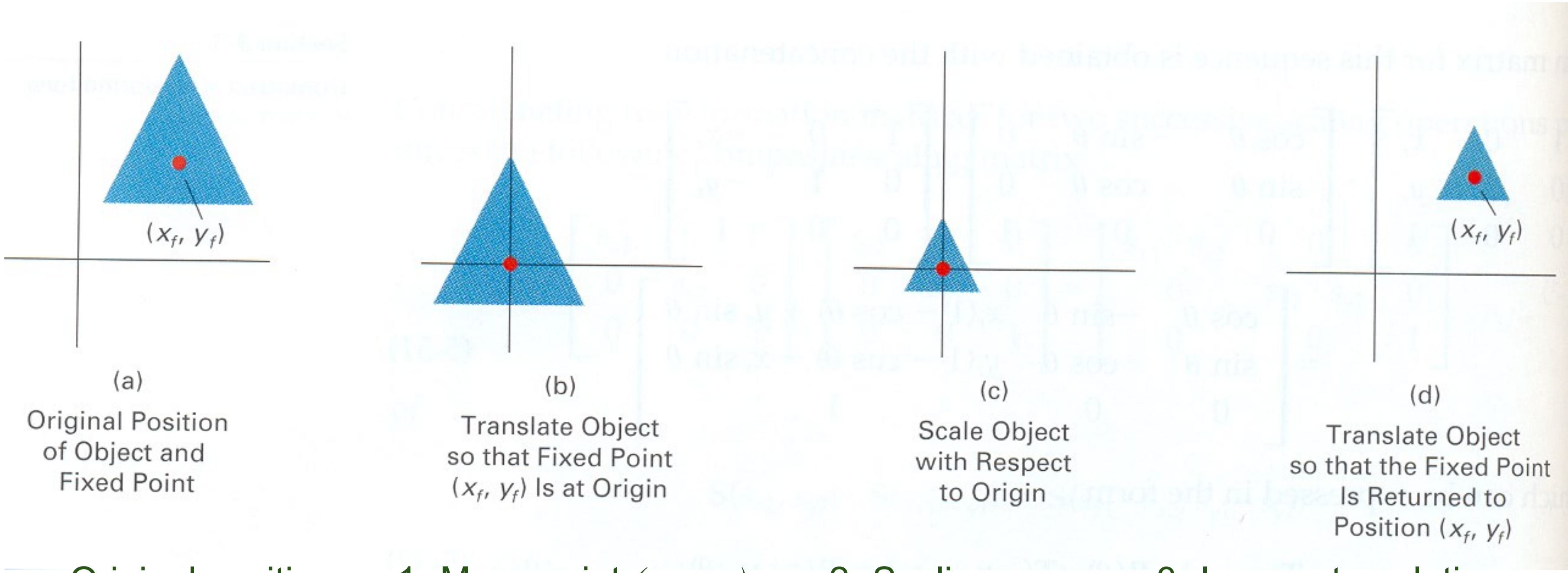
Inverse translation · θ degrees rotation · Moving to the origin

$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}(x_r, y_r) \cdot \mathbf{R}(\theta) \cdot \mathbf{T}(-x_r, -y_r) = \mathbf{R}(x_r, y_r, \theta)$$

Scaling relative to a fixed point (x_f, y_f)



Original position of the triangle and point (x_f, y_f)

1- Move point (x_f, y_f) to the origin

2- Scaling

3- Inverse translation back to (x_f, y_r)

Concatenation of transformations
Properties

- Matrix multiplication is **associative**

- Given any three matrices M_1, M_2, M_3, their product can be computed multiplying first M_3 by M_2, or multiplying first M_2 by M_1

$$M_3 \, . \, M_2 \, . \, M_1 = (M_3 \, . \, M_2) \, . \, M_1 = M_3 \, . \, (M_2 \, . \, M_1)$$

- In general, matrix multiplication is **not commutative**:

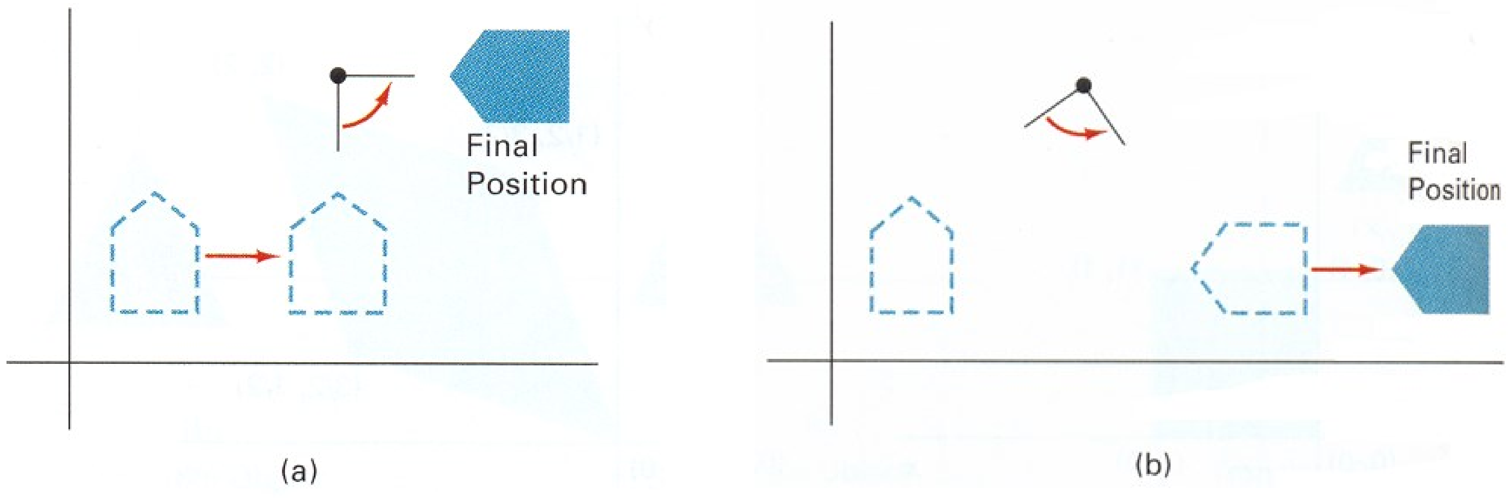
$$M2 \, . \, M1 \neq M1 \, . \, M2$$

- For instance, to apply a **rotation and a translation** to an object, care is needed to carry out the multiplication in the **appropriate order**

Concatenation of transformations

Order is Important

Changing the order of a transformation sequence **might affect the final result**.



In **particular cases**, matrix **multiplication is commutative**, e.g., two successive rotations, or two successive scalings, or two successive translations.

Concatenation of transformations

Efficiency

A 2D transformation representing a concatenation of transformations (translations, rotations, scalings) can be expressed as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} rs_{xx} & rs_{xy} & trs_x \\ rs_{yx} & rs_{yy} & trs_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

rotations and scalings

Translation distances, or rotation center, or fixed-point for scaling

Concatenation of transformations

Efficiency

Example: **scaling** followed by a **rotation**, both relative to an object's **center** *(x_c, y_c)* , followed by **translation**:

$$\mathbf{T}(t_x, t_y) \cdot \mathbf{R}(x_c, y_c, \theta) \cdot \mathbf{S}(x_c, y_c, s_x, s_y)$$

$$= \begin{bmatrix} s_x \cos\theta & -s_y \sin\theta & x_c(1 - s_x \cos\theta) + y_c s_y \sin\theta + t_x \\ s_x \sin\theta & s_y \cos\theta & y_c(1 - s_y \cos\theta) - x_c s_x \sin\theta + t_y \\ 0 & 0 & 1 \end{bmatrix}$$

The **transformed coordinates** are given by:

$$x' = x \cdot rs_{xx} + y \cdot rs_{xy} + trs_x, \qquad y' = x \cdot rs_{yx} + y \cdot rs_{yy} + trs_y$$

Concatenation of transformations Efficiency

- For any sequence of transformations, represented by one **global transformation matrix**, we need only:

  - 4 multiplications

  - 4 additions

- If each transformation was independently applied, the number of multiplications and additions would be larger

- Use the **single, global transformation matrix** resulting from the concatenation of the individual transformation

coordinate systems

Model Transform

- Defining where the objects are in the world

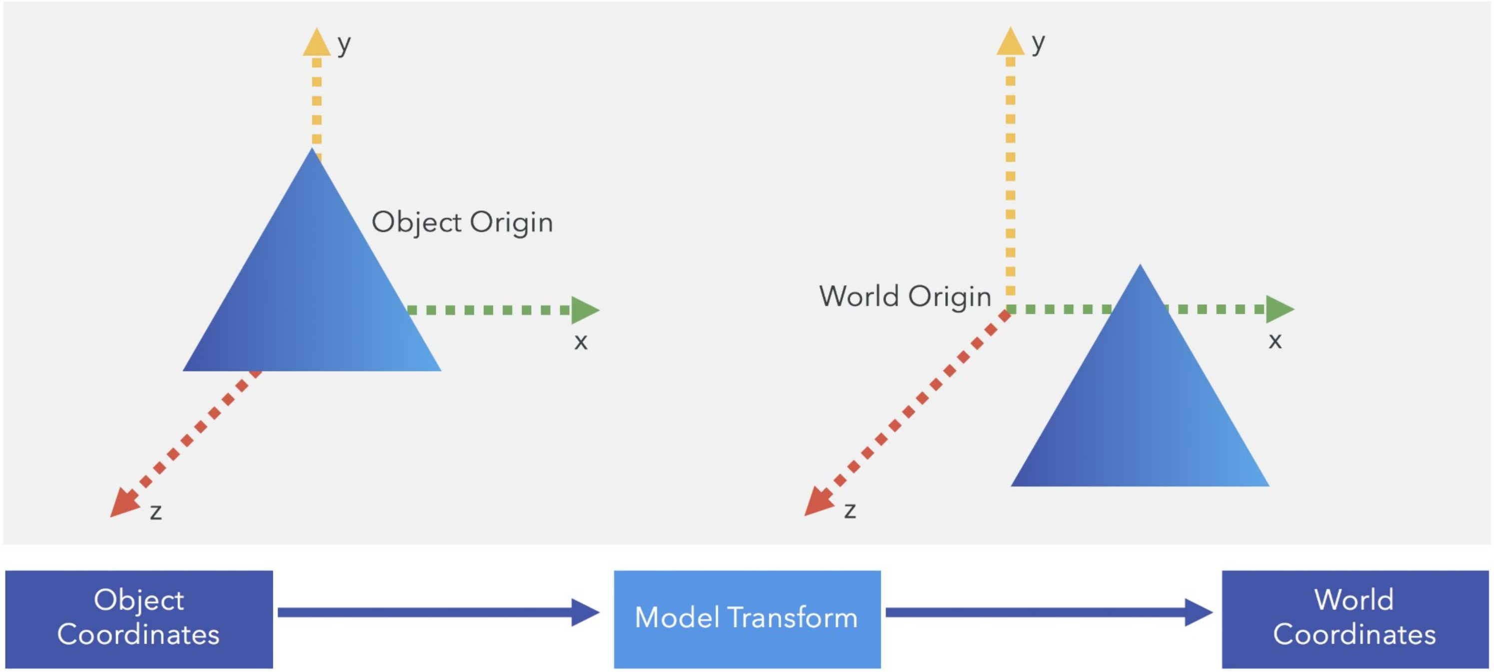


Image from Ghayour, Cantor, “Real-time 3D Graphics with WebGL 2”

View Transform

- Define where the eye (i.e., the camera) is.

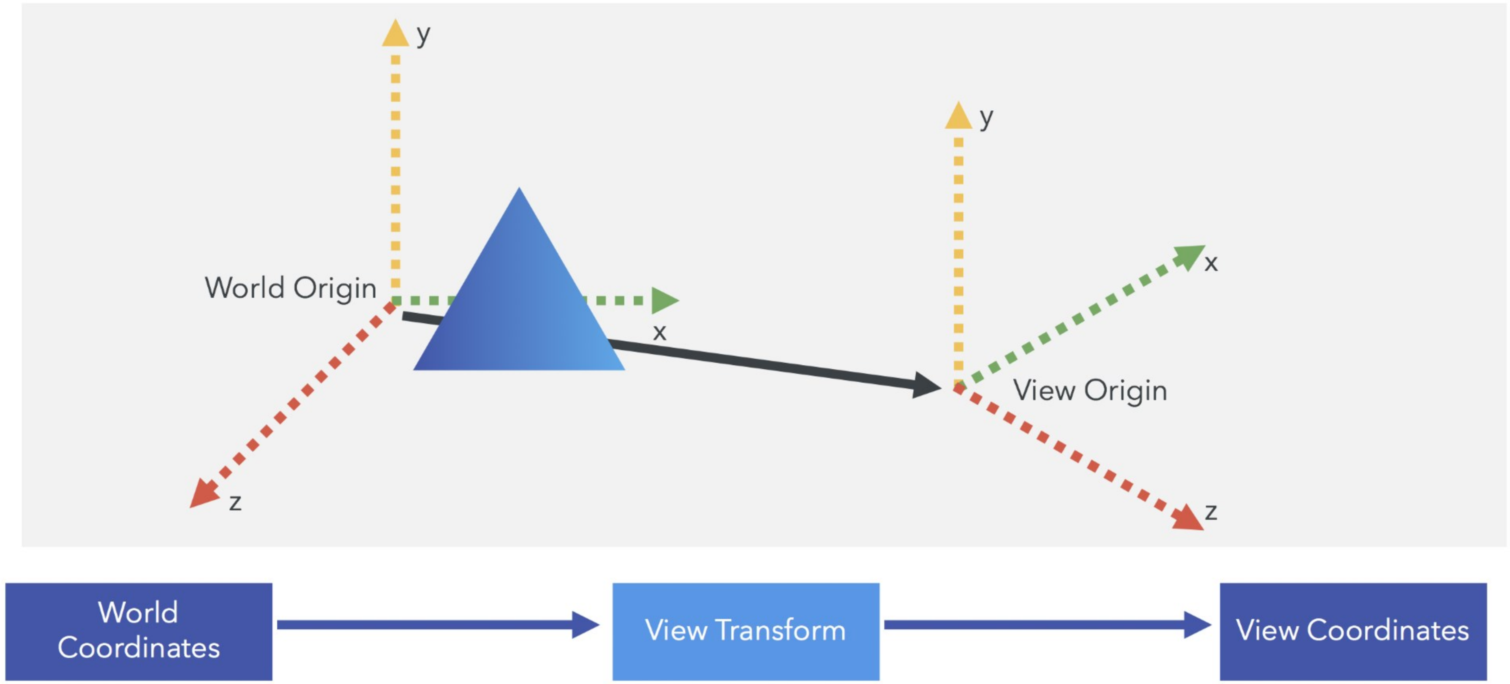


Image from Ghayour, Cantor, “Real-time 3D Graphics with WebGL 2”

Projection Transform

- How much of the world do I see? And how do I see it mapped to screen

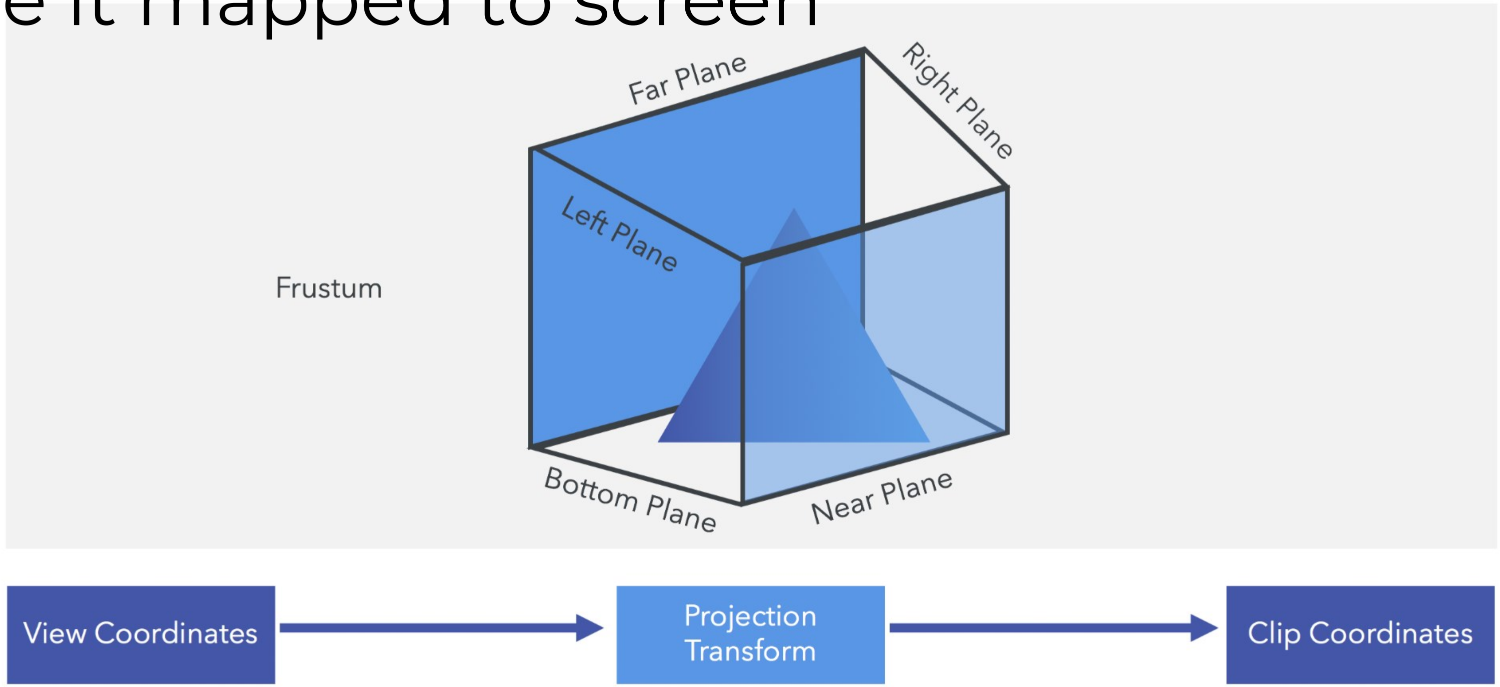


Image from Ghayour, Cantor, “Real-time 3D Graphics with WebGL 2”

Normalized Device Coordinates

- Coordinates of objects in normalized 2D screen space

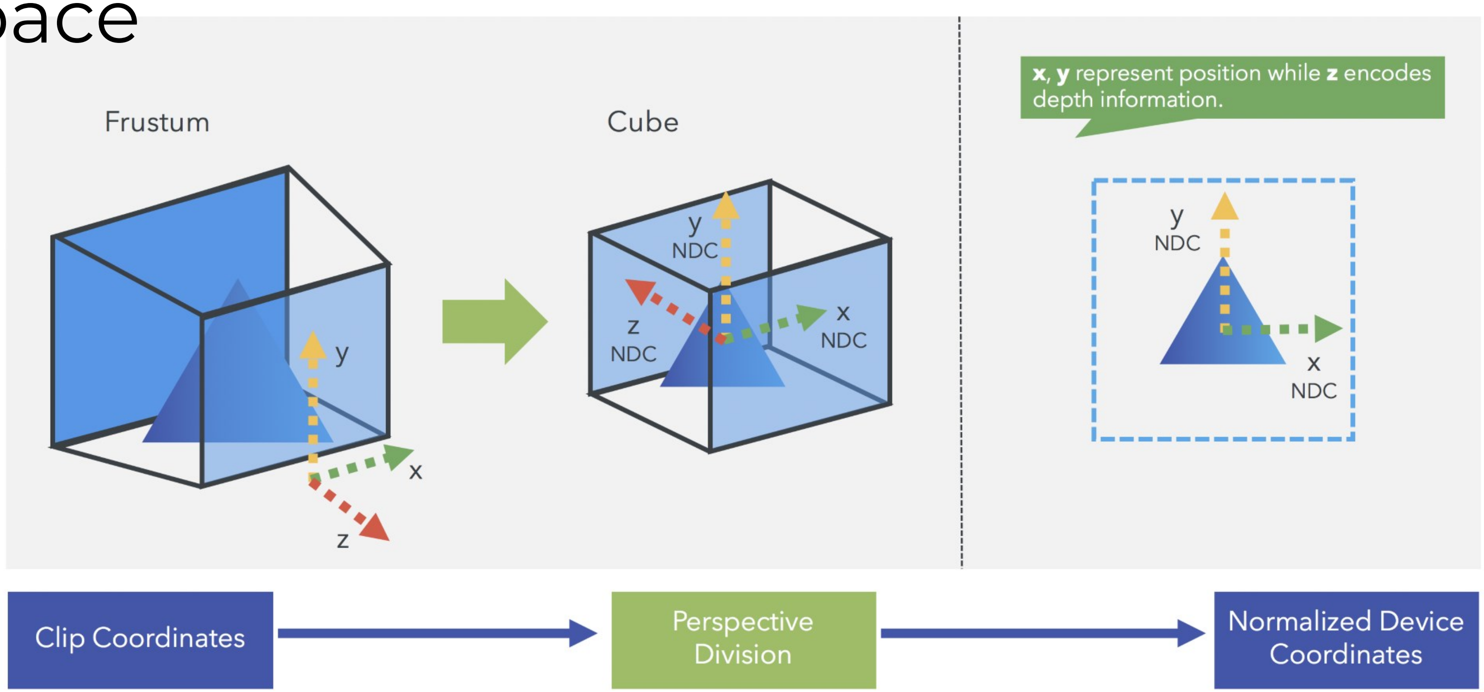


Image from Ghayour, Cantor, “Real-time 3D Graphics with WebGL 2”

Viewport Transform

- Scene is adjusted to the viewing window

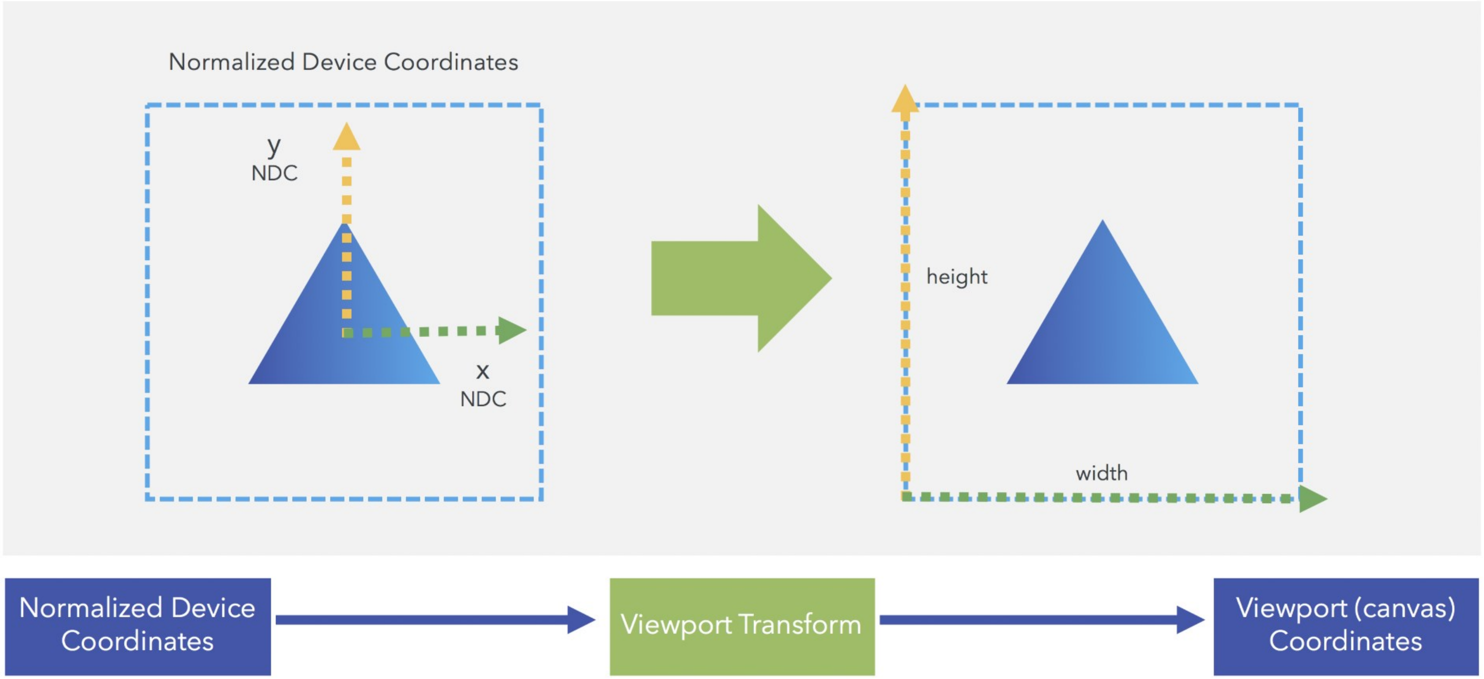
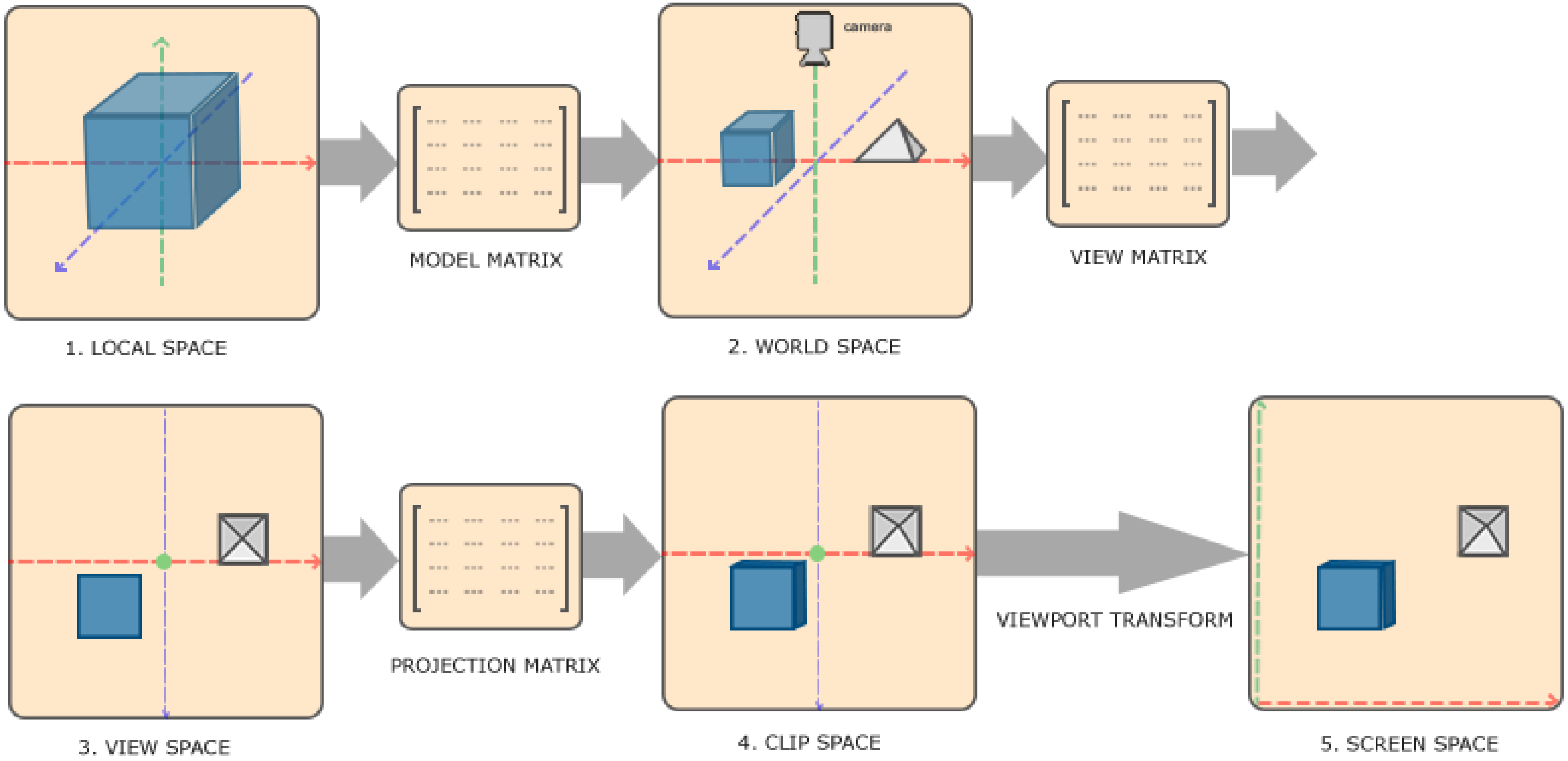


Image from Ghayour, Cantor, “Real-time 3D Graphics with WebGL 2”

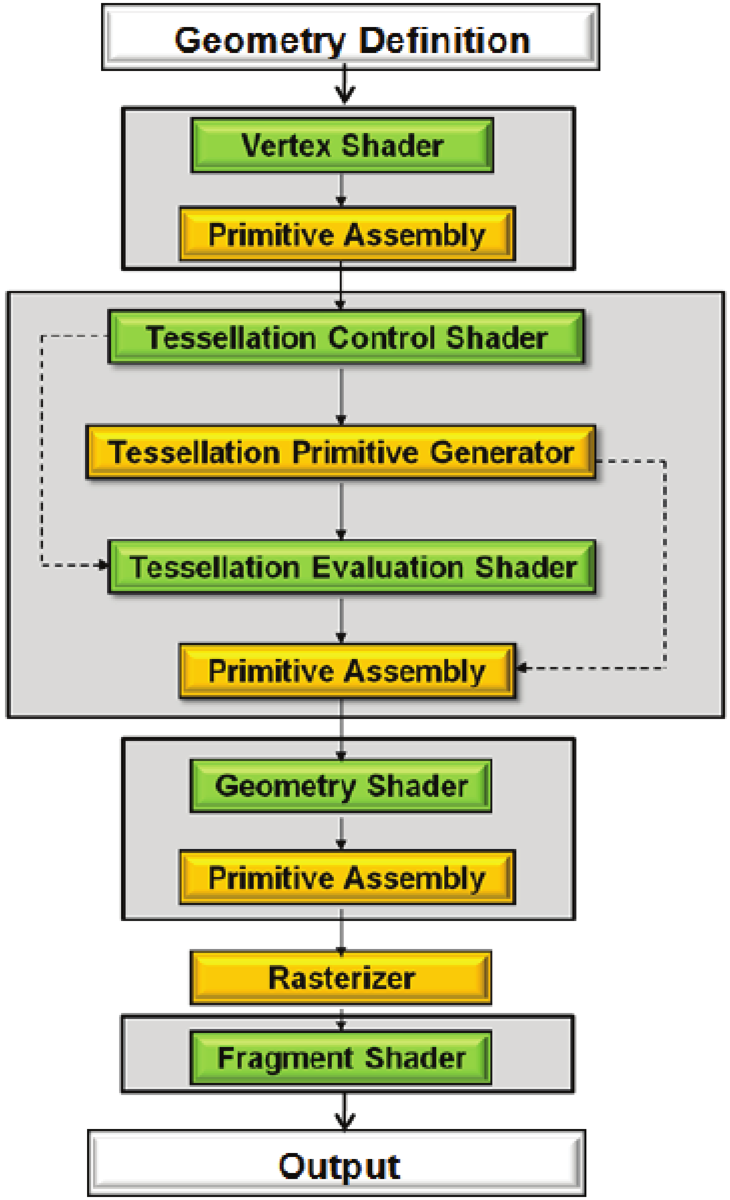
Model, View, Projection



learnopengl.com

Graphics Pipeline

- Transformations are applied on the **vertex shader**



Applying the transformations

In the **vertex shader**, we need to transform the vertex coordinates from local coordinates (model) to clip coordinates (projection):

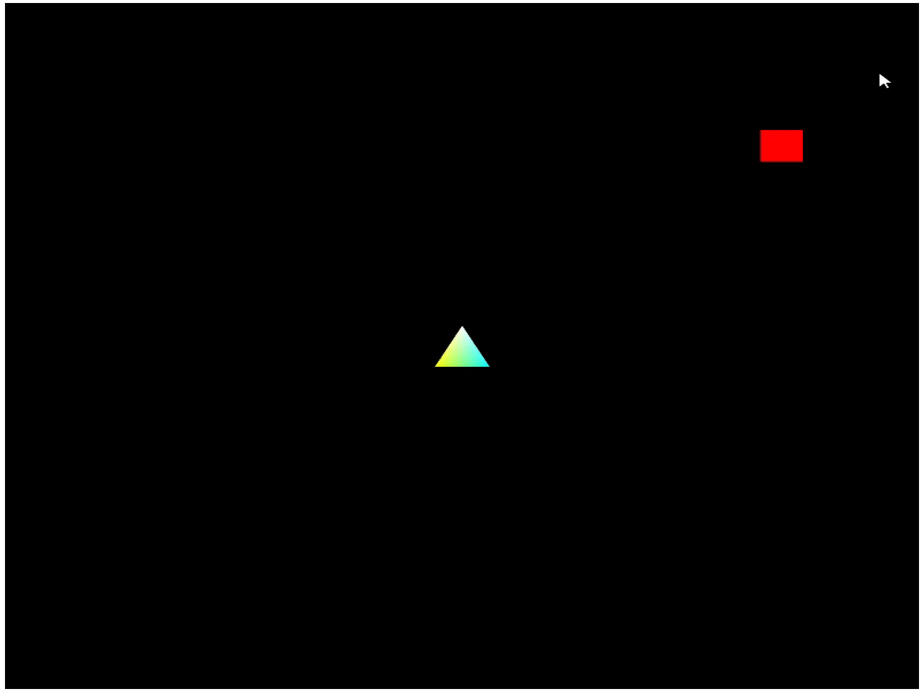
- Vfinal = **Mproj** * **Mview** * **Mmodel** * Vinicial

Pay attention to the order!
The first applied is the one closest to *Vinicial*

*Note that **Mproj** and **Mview** are the same for the whole world.*

*Only **Mmodel** may vary with different models.*

Hands On 02




bibliography

Bibliography

- S Marschner, P Shirley, *Fundamentals of Computer Graphics,* , A K Peters / CRC Press, 4th ed., 2018 https://learning.oreilly.com/library/view/fundamentals-of-computer/9781482229417/

- D. Hearn and M. P. Baker, *Computer Graphics with OpenGL*, 3rd Ed., Addison-Wesley, 2004

- Coordinate Systems, in https://learnopengl.com/Getting-started/Coordinate-Systems

pyGLM

pyGLM methods translate, rotate, and scale, apply the transformation by right-multiplying

glm.translate(myMatrix, glm.vec3(tx,ty,tz))

= myMatrix * T

SO, the last transformation to be applied is the first to be coded... T R S * P => glm.translate ... glm.rotate... glm.scale