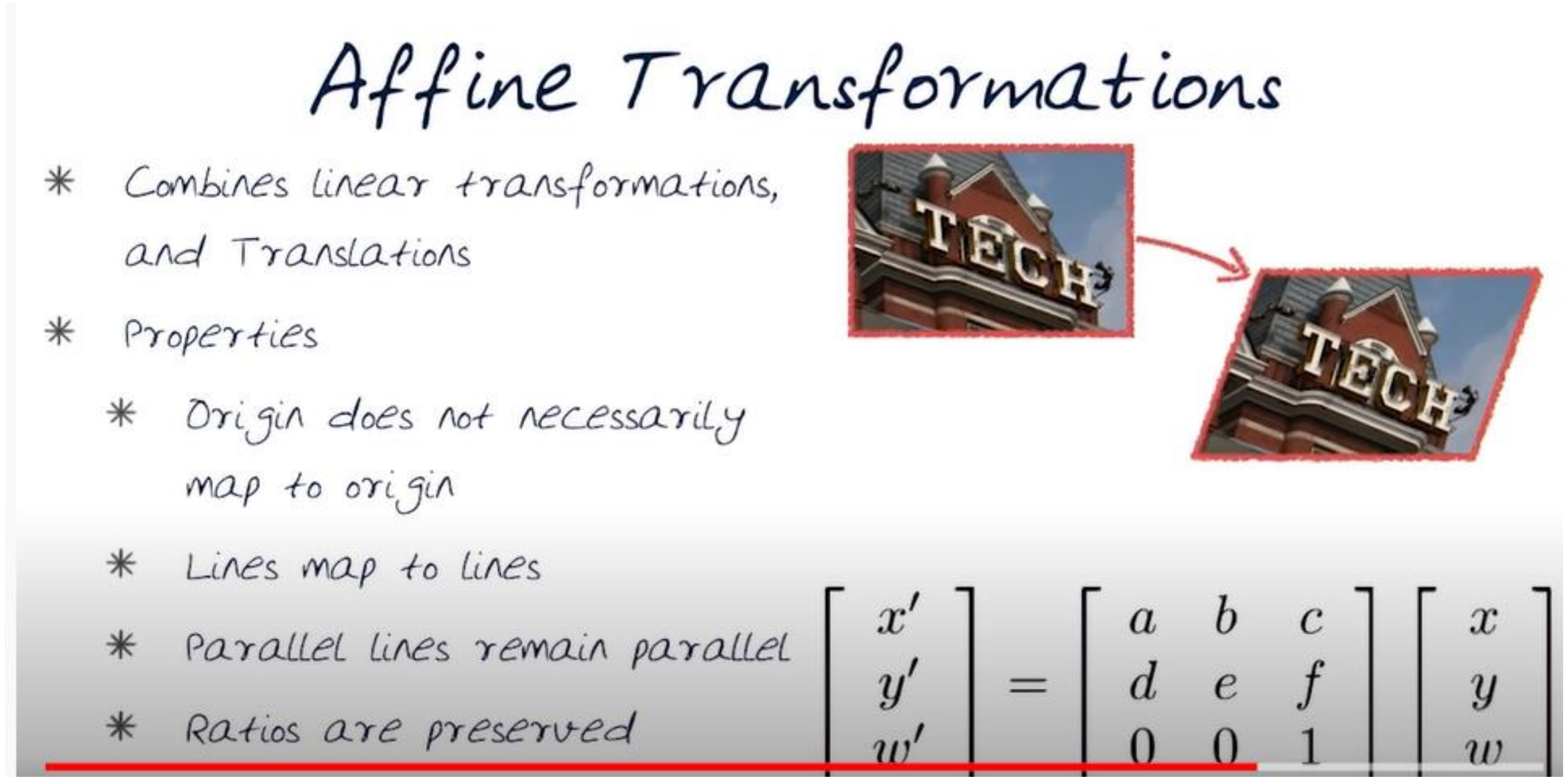
# Affine transformation

(An Affine Transform is a Linear Transform + a Translation Vector)

## Affine Transformation ?

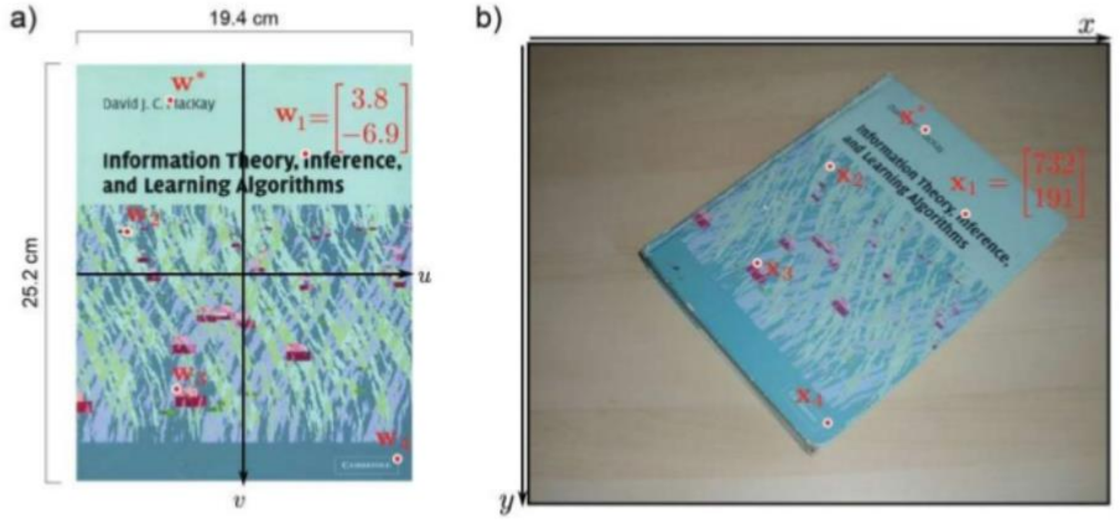


Affine Transformations

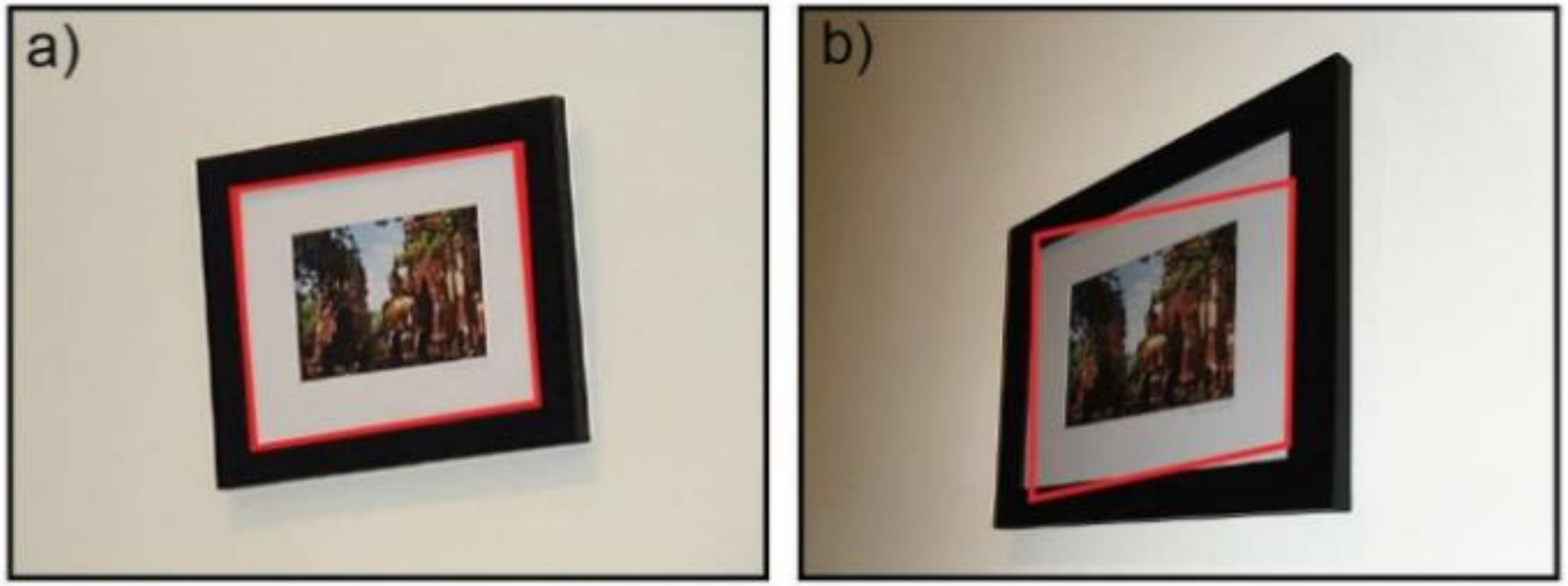
- Combines linear transformations, and Translations
- Properties
  - Origin does not necessarily map to origin
  - Lines map to lines
  - Parallel lines remain parallel
  - Ratios are preserved

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

# How to clip the book region?

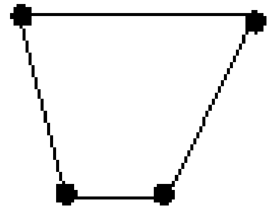
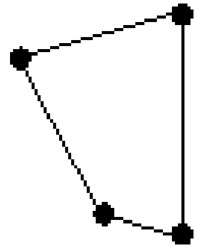
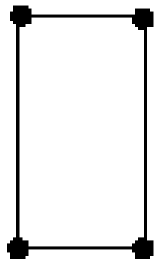
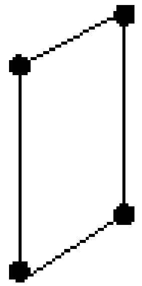
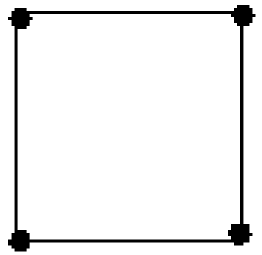
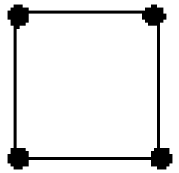
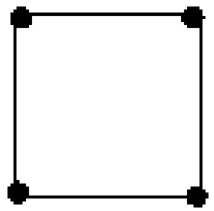
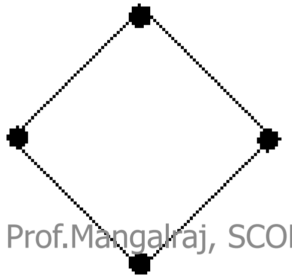


## How to clip the photo region?



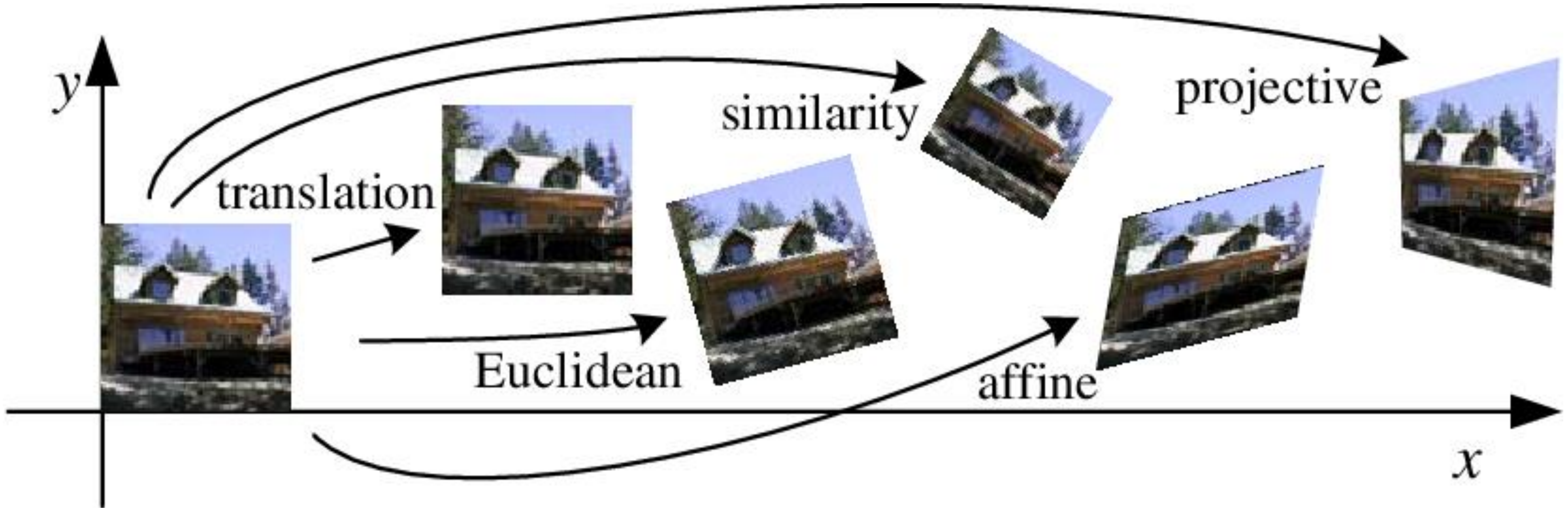
## Affine transformation

- Till now we have learned about lot of subsets (rotation, scaling, translation) of transformation.
- Affine transformation are transformation which preserve the parallelism between the lines in a image/model.
- Consider an example, if you apply affine transformation in an image and when you compare the output of the affine transformed image with the input image. You can notice that the parallelism between the image (input & output) is preserved.

| Transformation | Before | After |
|---|---|---|
| Projective |  |  |
| Affine |  |  |
| Similarity |  |  |
| Euclidean |  |  |

Euclidean transformations preserve length and angle measure. Moreover, the shape of a geometric object will not change. That is, lines transform to lines, planes transform to planes, circles transform to circles, and ellipsoids transform to ellipsoids. Only the position and orientation of the object will change.

Affine transformations are generalizations of Euclidean transformations. Under affine transformations, lines transforms to lines; but, circles become ellipses. Length and angle are not preserved.



## Affine transformation

- Affine transformation is a **linear** mapping method that preserves points, straight lines, and planes. Sets of parallel lines remain parallel after an affine transformation.

### Where is it applied or used?

- The affine transformation technique is typically used to correct for geometric distortions or deformations that occur with non-ideal camera angles.

## Performing affine transformation

- In order to calculate affine transform of any image or object model, you need a two set of points.

- Two set of points.
  - First set of points for input image.
  - Second set of points for output image.

- Affine transformation is performed by mapping these first set of points to the second set of points.

## Transforming points

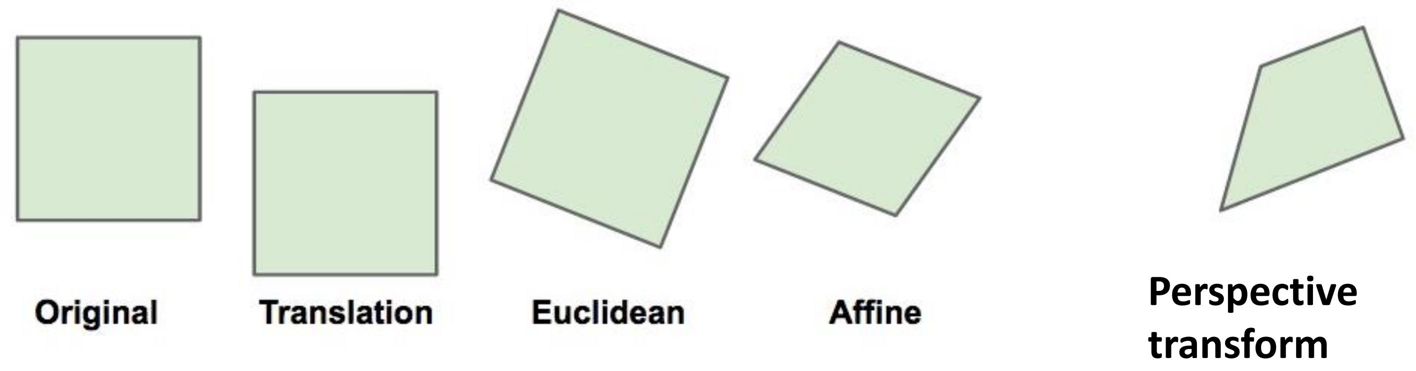
| Input image/Shape | Output image/shape |
|---|---|
| Point 1 | Point 1 |
| Point 2 | Point 2 |
| Point 3 | Point 3 |

## Types of transformation

- Affine
- Perspective or homography

# Affine transformation vs Perspective transformation

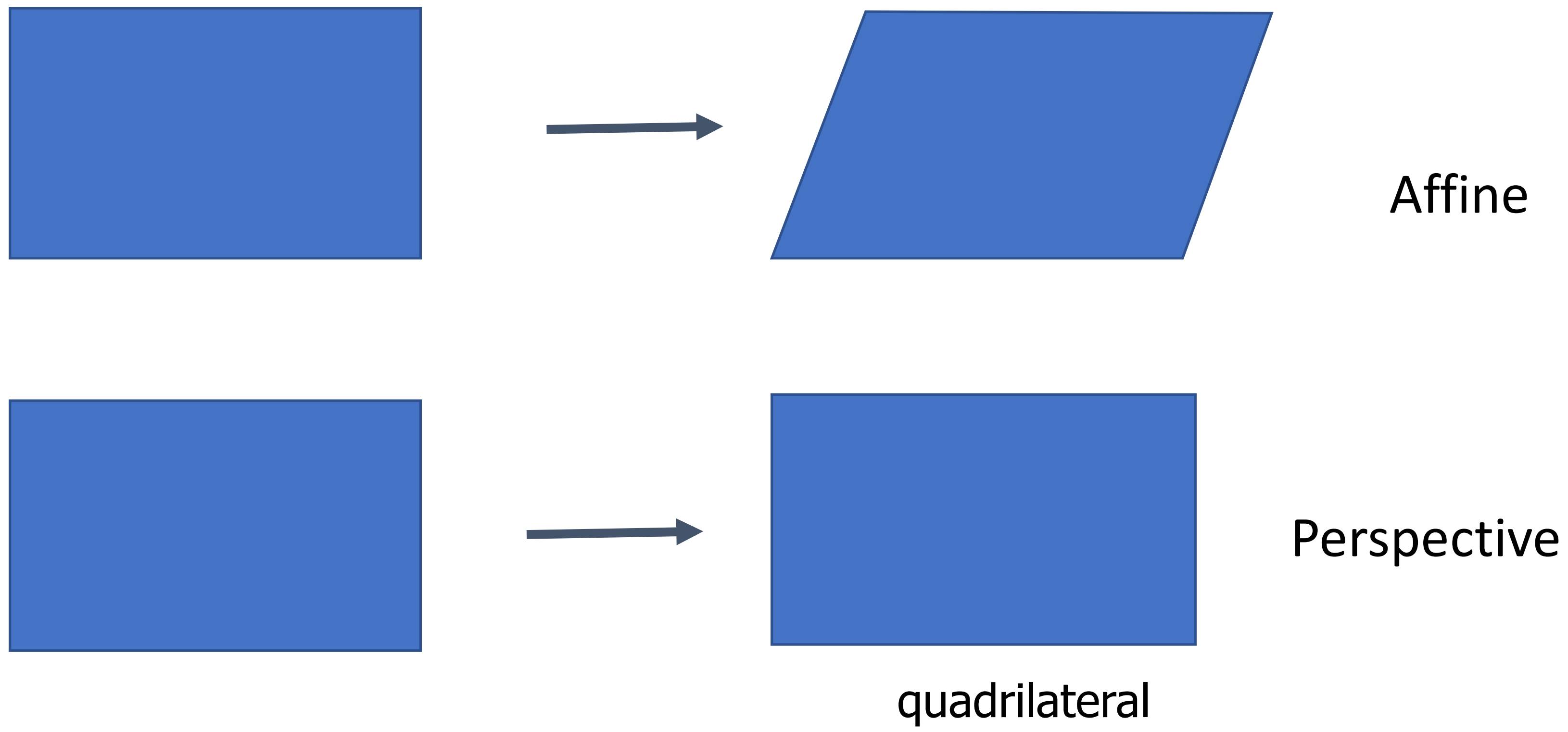
- Affine transformations *always* map parallel lines to parallel lines, while Perspective transformations *can* map parallel lines to intersecting lines



## Affine vs Perspective transformation when applied on rectangular shape

- Starting with a regular square, you can see that translational and Euclidean transformations (rotation, uniform scaling, and translation)
- keep the aspect ratio; the result on applying is still a square.
- However, affine transformations can squash the square into a rectangle in either direction, and it can also provide a shear/skew to the square.
- But notice that the shape after the affine transformation is applied is a parallelogram---the sides are still parallel.
- With a Perspective (homography) transformation, this need not be the case. The parallel lines could be warped so that they intersect. So the result of a rectangle transformed by an homography is a general quadrilateral.
- While the result of a rectangle transformed by an affine transformation is always a parallelogram.

## Affine vs Perspective transformation when applied on rectangular shape



## Affine transformation in Opencv

- Cv2. getAffineTransform()
- Cv2. warpAffine()

## Perspective transformation in Opencv

- Cv2. getPerspectiveTransform ()
- Cv2. warpPerspective()
- In addition to 3 points we have to add 4 points

## Any real time software in which you applied Affine and Perspective transformation?

- Word $\rightarrow$ shapes $\rightarrow$ Import any shapes $\rightarrow$ Edit points
- Camscanner

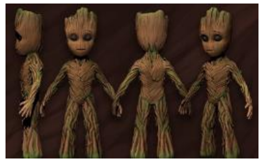
## Any real world examples where these transformation applied?

## Any real world examples where these transformation applied?

- Face morphing
- Cricket

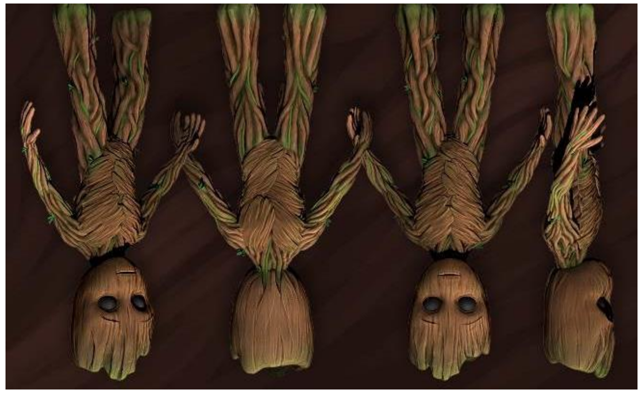
## Concatenation in transformation

- When two or more transformation is performed at a time.



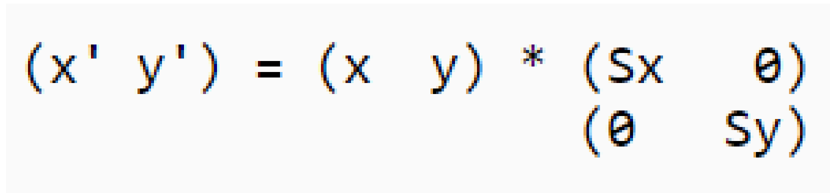
$$\xrightarrow{\hspace{3cm}}$$

**Concatenation :Rotation and scaling**



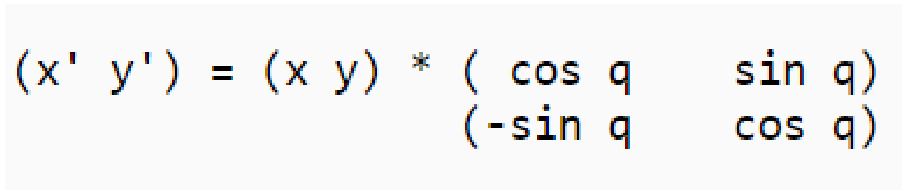
# Concatenation in transformation

- Scaling: x' = x * Sx, y' = y * Sy, which can be written as:



```
(x' y') = (x  y) * (Sx   0)
                   (0   Sy)
```

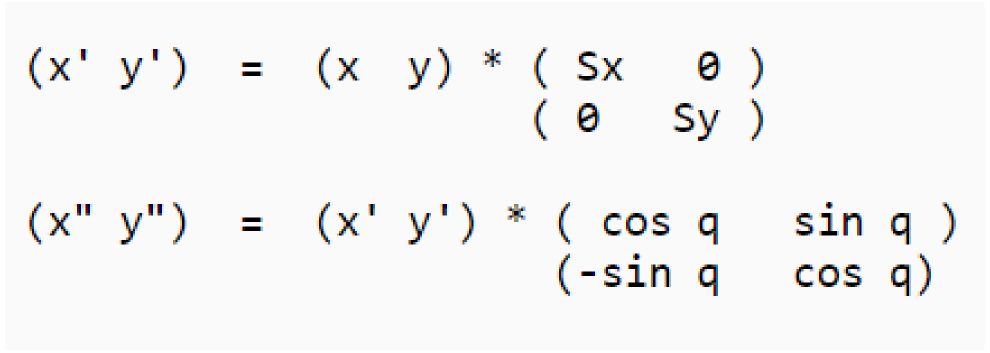
- Rotation: x' = x cos q - y sin q, y' = y cos q + x sin q, which can be written as:



```
(x' y') = (x y) * ( cos q   sin q)
                  (-sin q   cos q)
```

## Concatenation in transformation – cont’d

- So if we wanted to scale and then rotate the object we could do the following:



```
(x' y')  =  (x  y) * ( Sx   0 )
                     ( 0   Sy )

(x" y")  =  (x' y') * ( cos q   sin q )
                      (-sin q   cos q)
```

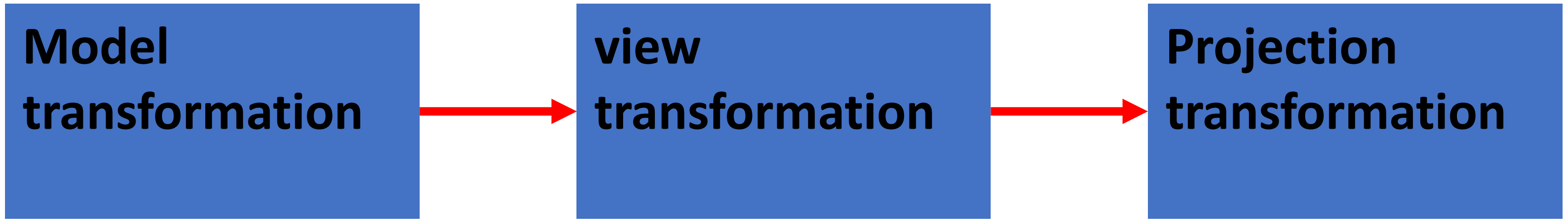
- Hence, we can concatenate the Scaling and Rotation matrices, and then multiply the old points by resultant matrix.

## OpenGL

- Open Graphics Library (**OpenGL**) **is** a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.

# OpenGL Vertex transformation

- The vertices that represent the objects forming a scene undergo several transformations through the OpenGL pipeline until they become pixels in the screen

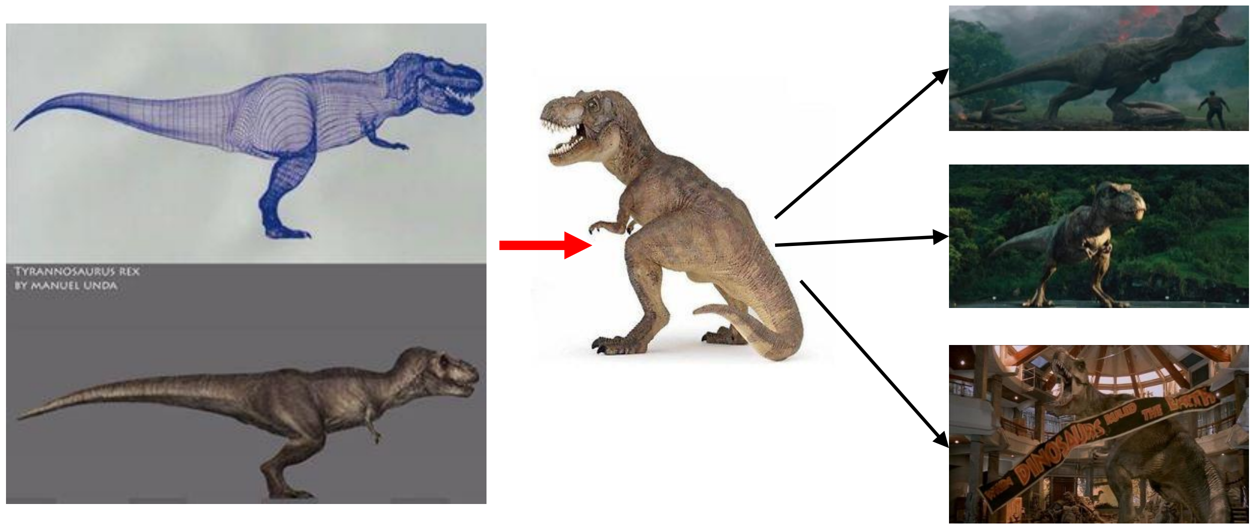


## Model transformation

- Each object is typically represented in its own coordinate system, known as its **model space** (or local space).
- In order to assemble a scene, it's necessary to transform the vertices from their local spaces to the **world space**, which is common to all the objects.
- So, the model transformation, represented by matrix called $M_{model}$, transforms each vertex of each object from its model space to the world space.

## Model transformation ($M_{model}$)

- $M_{model}$ :performs the appropriate coordinate change (rotation, translation, scale) to each vertex.



$$\begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{pmatrix} = M_{model} \cdot \begin{pmatrix} x_{object} \\ y_{object} \\ z_{object} \\ 1 \end{pmatrix}$$

## View transformation

- The next step is to transform the vertices from the world space to the the **eye or camera space**.
- We need other matrix ($M_{view}$) to apply this transformation.

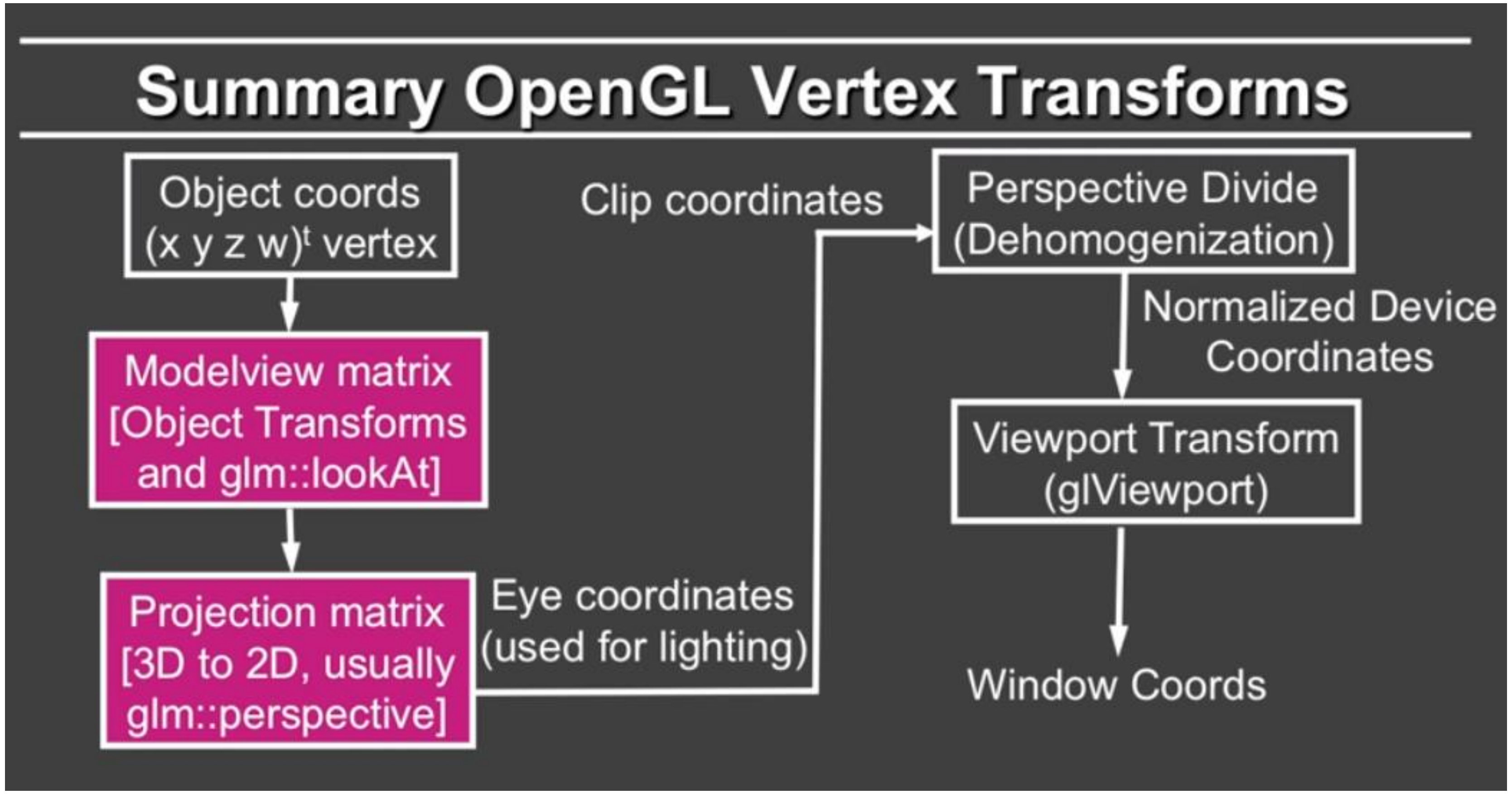
$$\begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} = M_{modelView} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix} = M_{view} \cdot M_{model} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix}$$

## Projection transformation

- Here we define, how the objects are projected onto the screen.
- This is done by specifying a viewing volume or *clipping volume* and selecting a projection mode/view.
- $M_{projection}$: To transform the vertices into clip-coordinates.

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = M_{projection} \cdot \begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix}$$

## OpenGL vertex transform


Summary OpenGL Vertex Transforms
Object coords (x y z w)$^t$ vertex
Modelview matrix [Object Transforms and glm::lookAt]
Projection matrix [3D to 2D, usually glm::perspective]
Eye coordinates (used for lighting)
Clip coordinates
Perspective Divide (Dehomogenization)
Normalized Device Coordinates
Viewport Transform (glViewport)
Window Coords

## Day 12
## MATRIX STACK

## Matrix stack

- The pushMatrix() function **saves the current coordinate system** to the stack and popMatrix() **restores the prior coordinate system**.
- pushMatrix() and popMatrix() are used in conjunction with the other transformation functions and may be embedded to control the scope of the transformations.

## Matrix stack

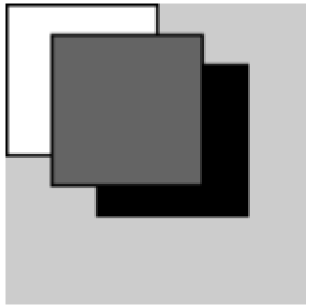
- Glpushmatrix() or pushmatrix()
- Glpopmatrix() or popmatrix()

## Model view matrix

- Model + world transformation → Model matrix
- Camera + View transformation → View matrix
- Projection transformation → Projection matrix

**Name** pushMatrix()

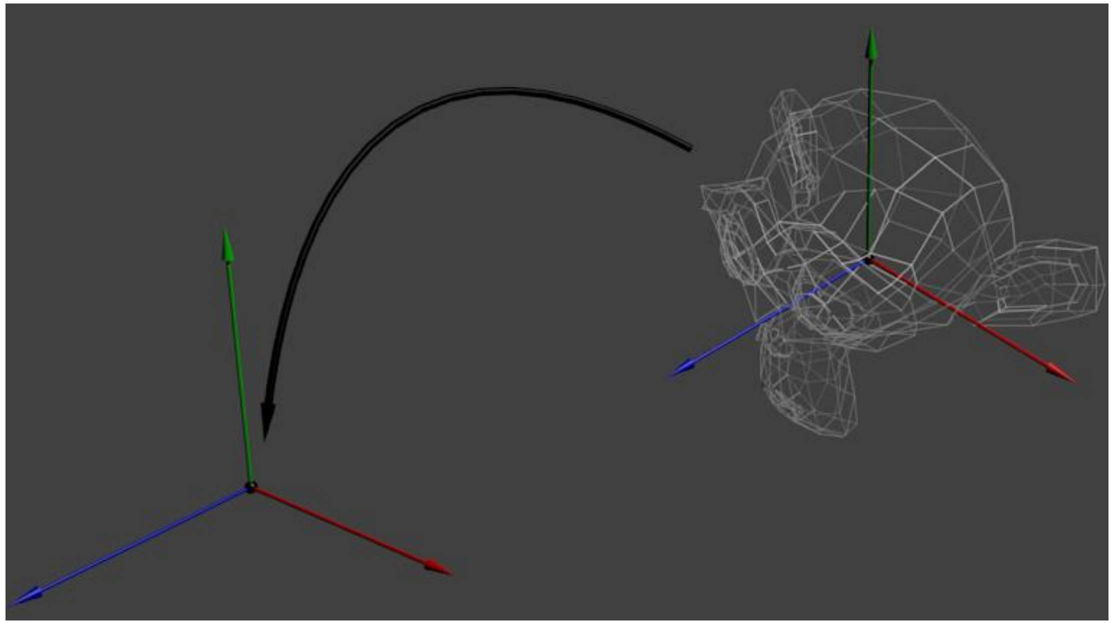
**Examples**



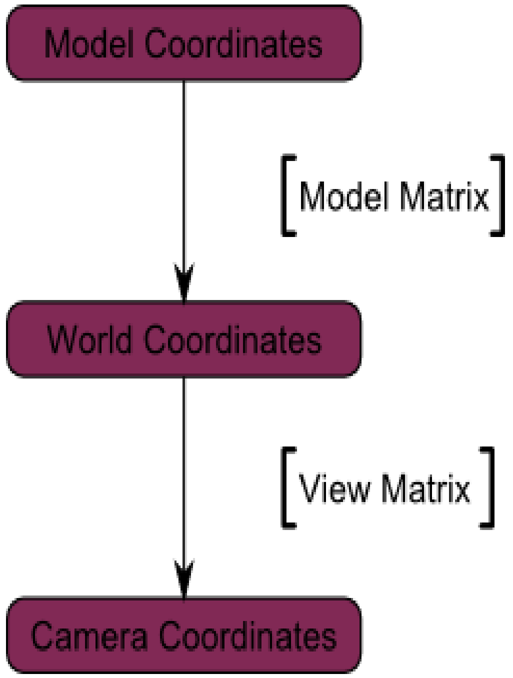
```
fill(255);
rect(0, 0, 50, 50);  // White rectangle

pushMatrix();
translate(30, 20);
fill(0);
rect(0, 0, 50, 50);  // Black rectangle
popMatrix();

fill(100);
rect(15, 10, 50, 50);  // Gray rectangle
```

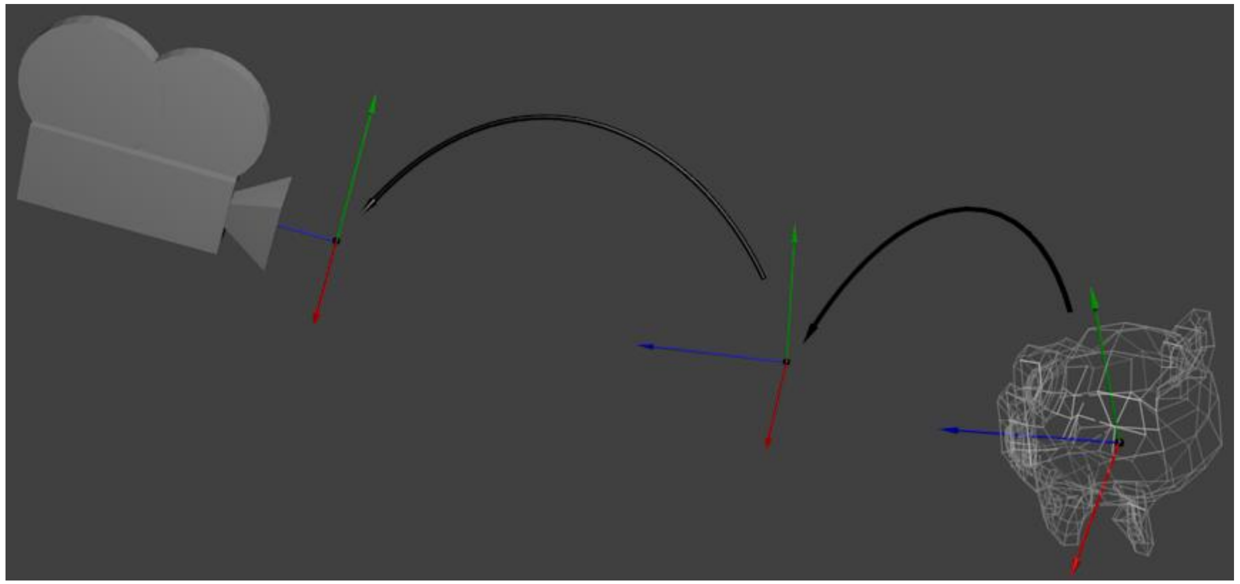
## Model matrix

$$\begin{tikzcd}\boxed{\text{Model Coordinates}} \arrow[d, "\left[\text{Model Matrix}\right]"] \\ \boxed{\text{World Coordinates}}\end{tikzcd}$$

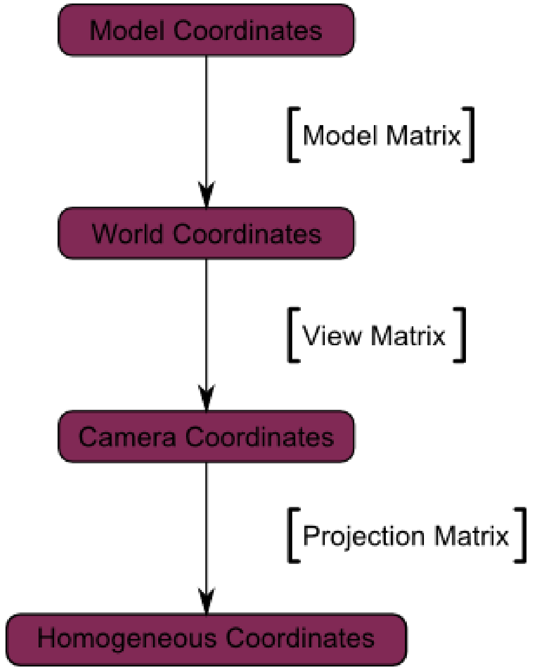


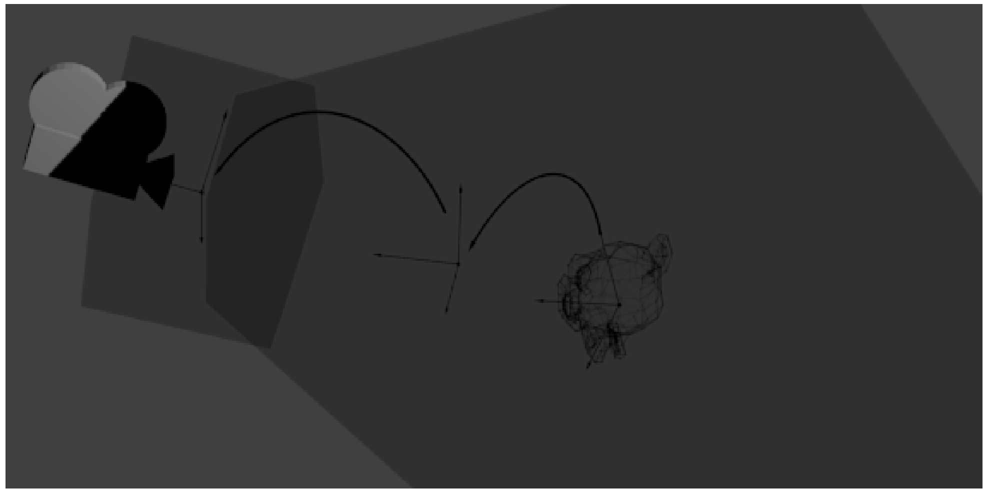
## View matrix

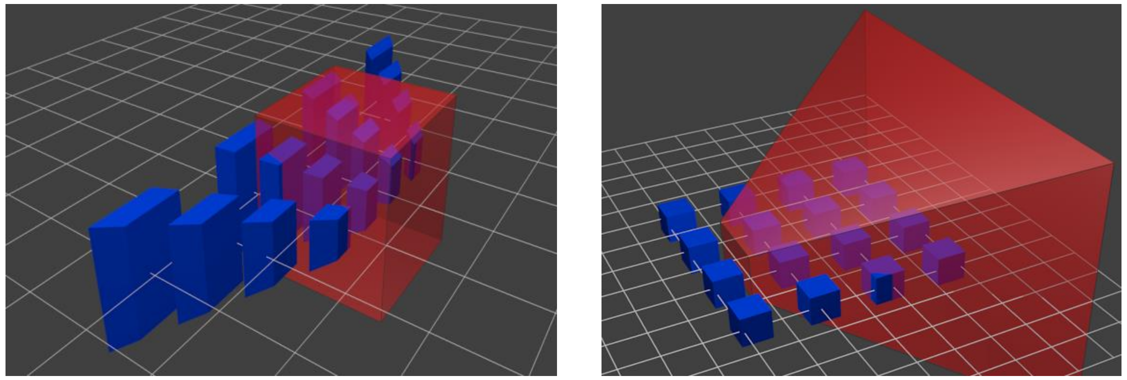

Model Coordinates
Model Matrix
World Coordinates
View Matrix
Camera Coordinates



# Projection matrix







## World transformation matrix vs. View matrix

- The **world transformation matrix** is the matrix that determines the position and orientation of an object in 3D space.

- The **view matrix** is used to transform a model's vertices from world-space to view-space.

## Camera transformation matrix

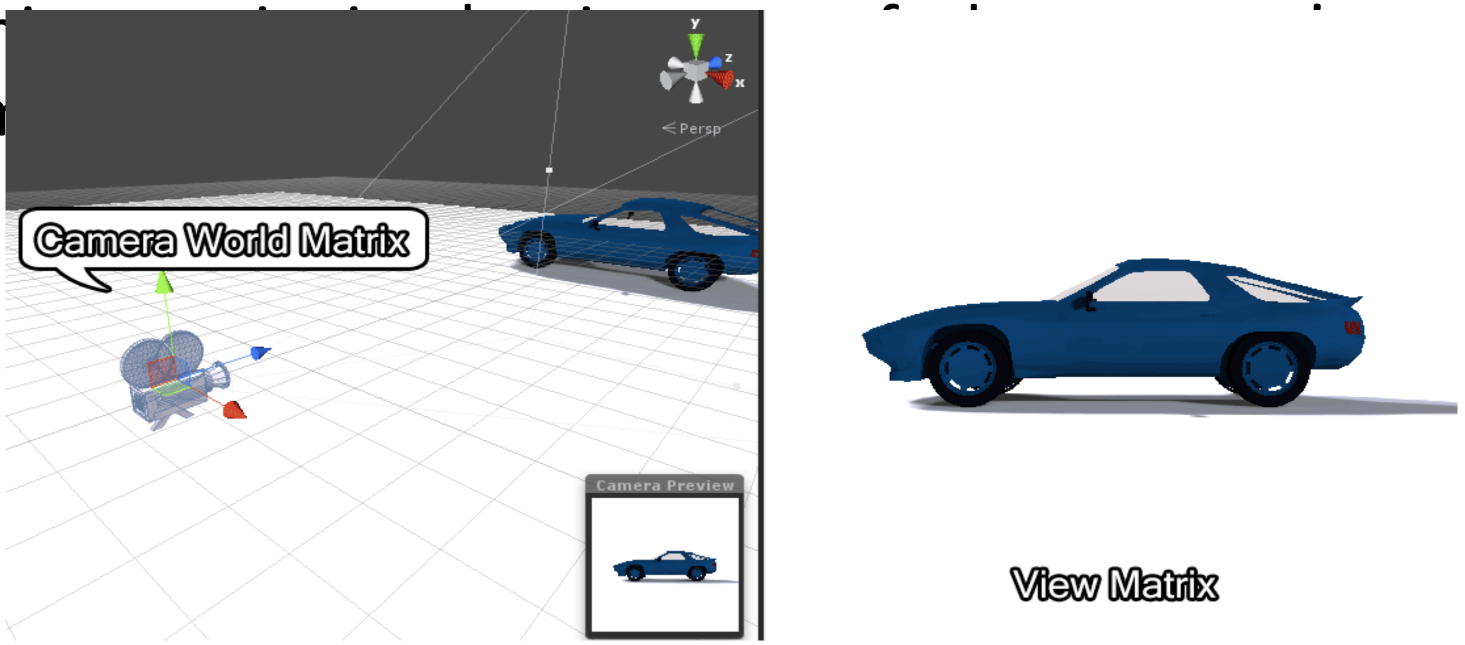
- The transformation that places **the camera in the correct position and orientation in world space** (this is the transformation that you would apply to a 3D model of the camera if you wanted to represent it in the scene).

## The View Matrix

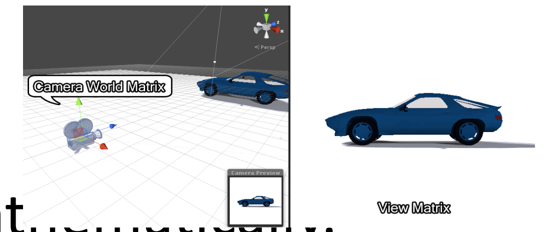
- This matrix will transform **vertices from world-space to view-space**.

Note: Th

transforn



# Mathematical convention (Model-view matrix)



- Consider this example and represent the matrices mat[illegible]y.
- Let us consider matrices to be column major. That is, in a **4×4 homogeneous transformation matrix**,
  - the first column represents the "right" vector (X),
  - the second column represents the "up" vector (Y),
  - the third column represents the "forward" vector (Z) and
  - the fourth column represents the translation vector **(origin or position) (W)**

## 4×4 homogeneous transformation matrix

$$V = \begin{bmatrix} right_x & up_x & forward_x & position_x \\ right_y & up_y & forward_y & position_y \\ right_z & up_z & forward_z & position_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Mathematical convention

- That is, in order to transform a vector **v** by a transformation matrix **M,** we would need to pre-multiply the column vector **v** by the matrix **M** on the left.

$$\begin{aligned} \mathbf{v}' &= \mathbf{M}\mathbf{v} \\ \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} &= \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \end{aligned}$$

# Model Transformation – mathematical convention

- When applying affine transformations such as Scale, rotate, translate.

$$\mathbf{v}' = (\mathbf{T}(\mathbf{R}(\mathbf{S}\mathbf{v})))$$

- This transformation can be stated in words as "first scale, then rotate, then translate".
- Finally, model matrix can be represented as

$$\mathbf{M} = \mathbf{TR}$$

- If **R** represents the orientation of the camera, and **T** represents the translation of the camera in world space

## View Transformation

- The view matrix on the other hand is used to transform vertices from world-space to view-space.
- If **M** represents the object's world matrix (or model matrix), and **V** represents the view matrix, and **P** is the projection matrix, then the concatenated world (or model), view, projection can be represented by **MVP** simply by multiplying the three matrices together:

$$\mathbf{MVP = M * V * P}$$

- And finally a vertex **v** an be transformed to clip-space by multiplying by the combined matrix MVP:

$$\mathbf{V' = MVP * v}$$

## Matrix Stack Ex.

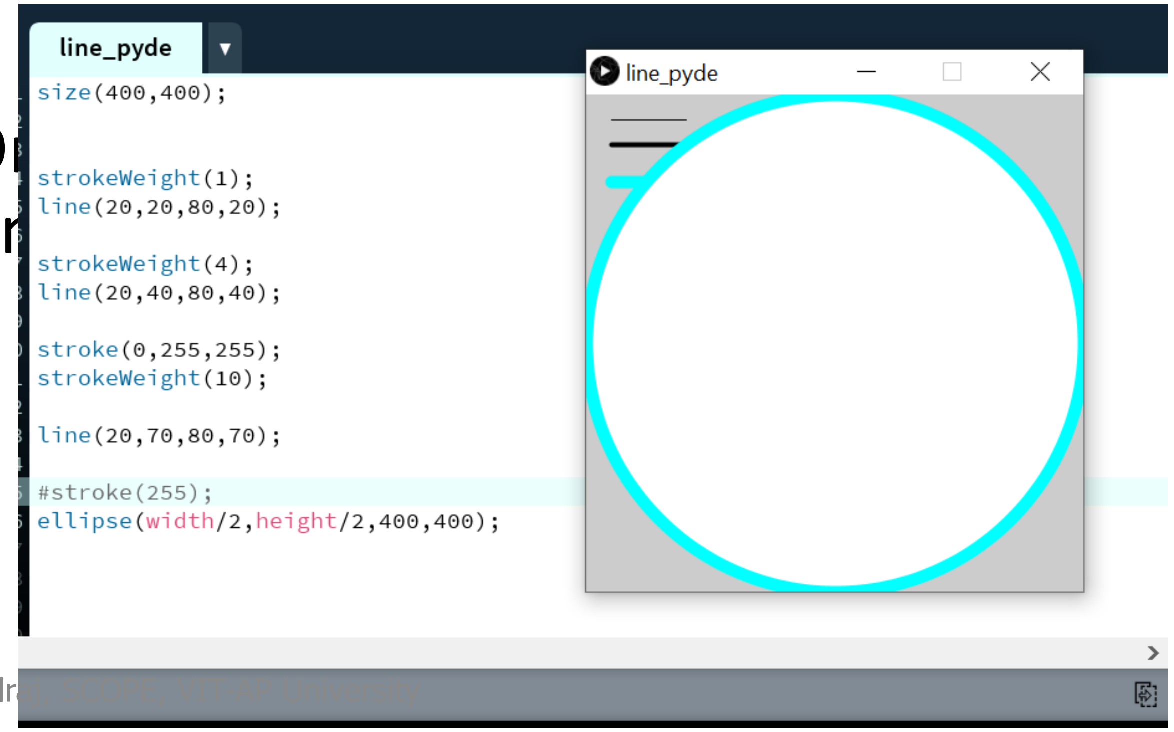
- **Logic:** When multiple objects there in canvas, if you perform any transformation like T,R,S etc then all the objects will be moved. Inthis case I don’t want to do transformation on all objects.

- I want only one object to translated/rotated etc. the you must pass that object coord. Into homogeneous system(w). So that the object which you want to translate will be stored in Homog.point and it will be perform specific function.

- This can be achieved thru push matrix and pop matrix.

## Special case

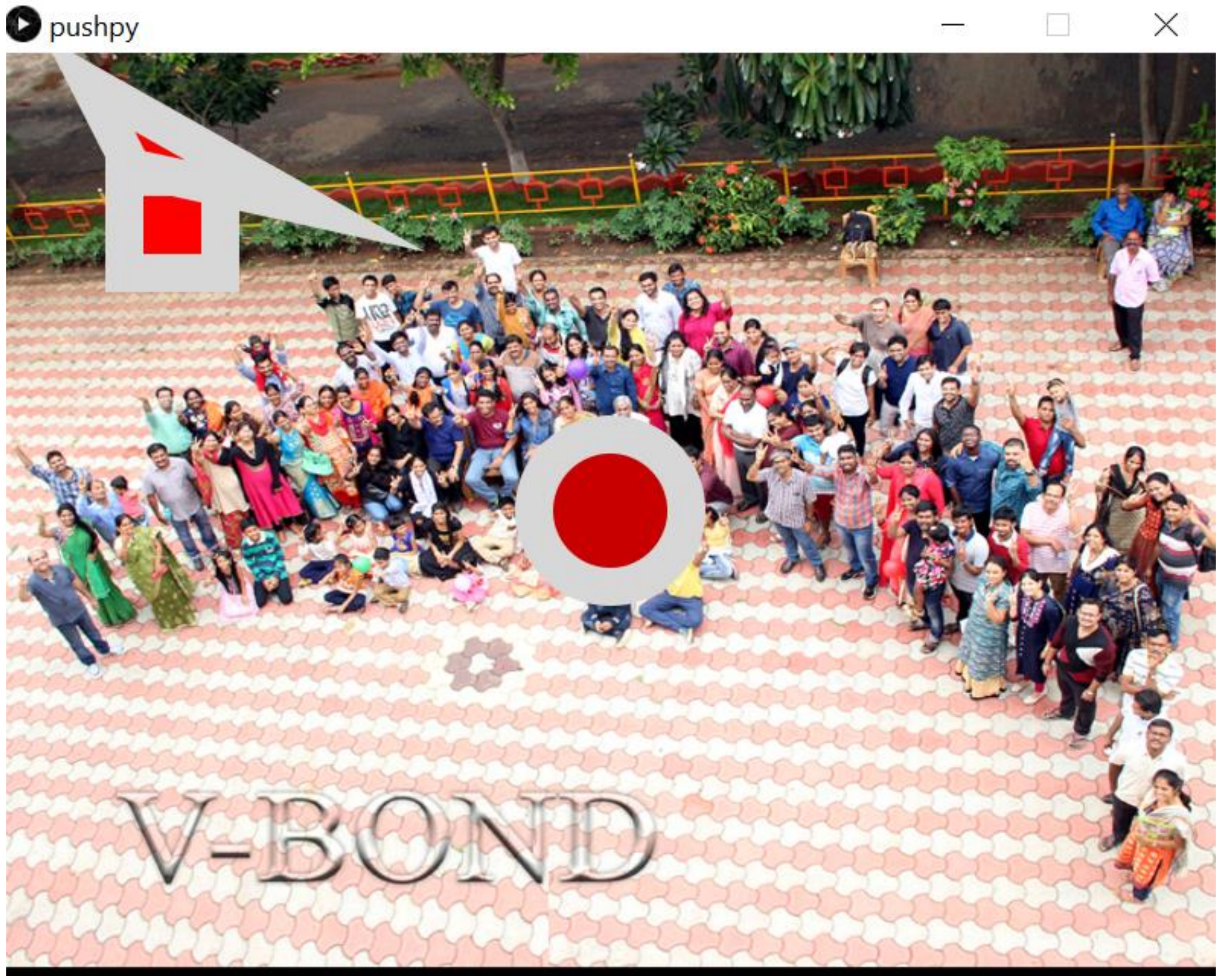
- Matrix is not applicable to colors.
- You can restrict of applying colors to only specific objects alot in canvas.
- Not all (lines) are blue color. O
  color. So no need to use push



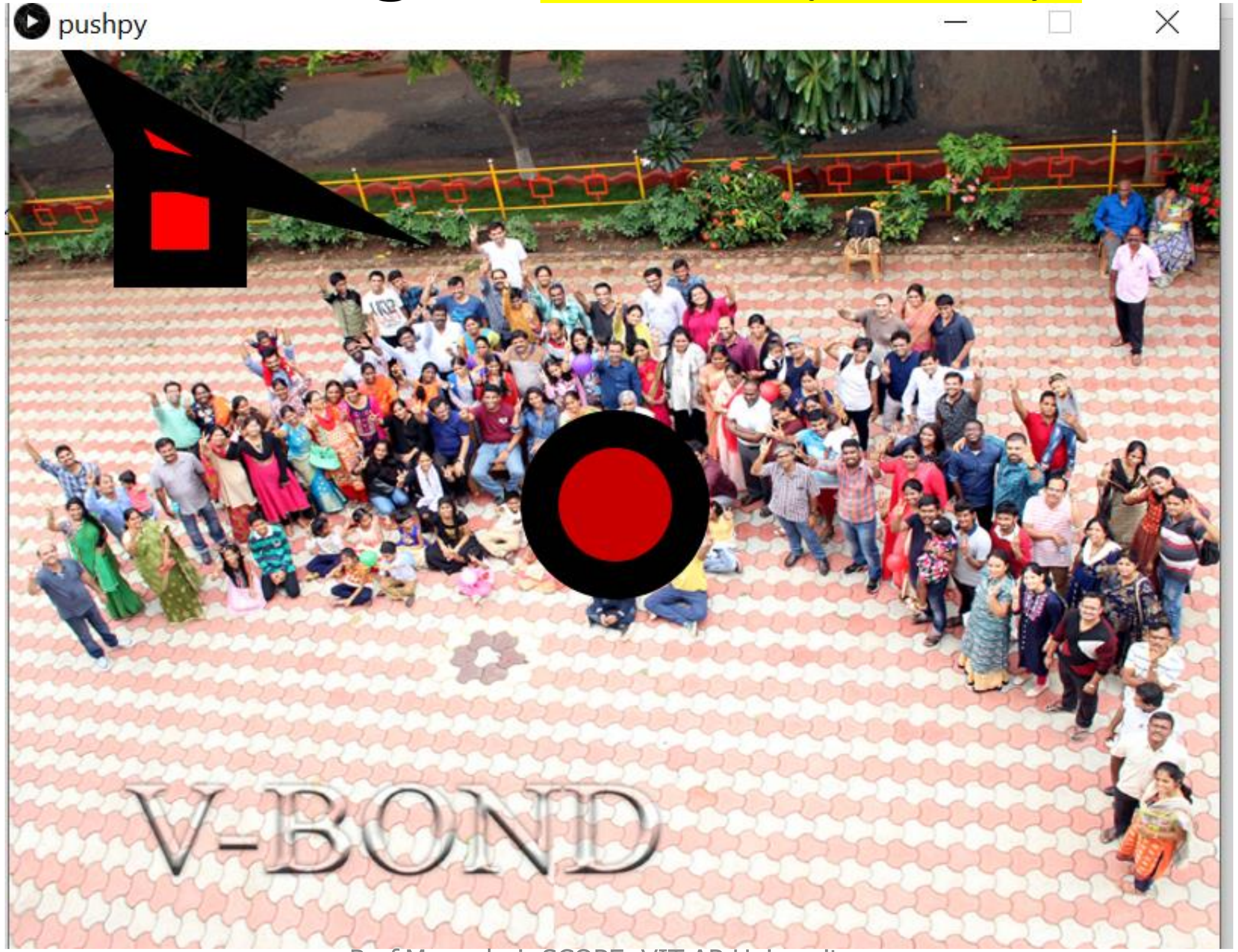
```
size(400,400);


strokeWeight(1);
line(20,20,80,20);

strokeWeight(4);
line(20,40,80,40);

stroke(0,255,255);
strokeWeight(10);

line(20,70,80,70);

#stroke(255);
ellipse(width/2,height/2,400,400);
```

```
#global grid;
#rotat = 0.0;
def setup():
    global grid;
    size(800, 600);
    grid = loadImage('V:\VBond.jpg’);
  def draw():
    background(255);
    #colorMode(RGB,100,500,10,255);
    image(grid,0,0,800, 600);
    #pushMatrix();
    #noFill();
    ##rotate(30);
    strokeWeight(25);
    #translate(200,100);
    #translate(mouseX,mouseY);
    stroke(215);
    fill(200,0,0);
    ellipse(400,300,100,100);
    #400+200=600,300+100=400,circle alone move
now.
    # translate(300,300);
    #popMatrix();
    #pushMatrix();
    fill(255,0,0);
    #translate(mouseX,mouseY);
    # textSize(32);
    # text("sibi", 300,200);
    #popMatrix();
    # noStroke();
    rect(81, 81, 63, 63);
    #pushMatrix()
    #rotate (2)
    triangle(200, 100, 58, 20, 86, 75);
    #popMatrix()
```

# Stroke used



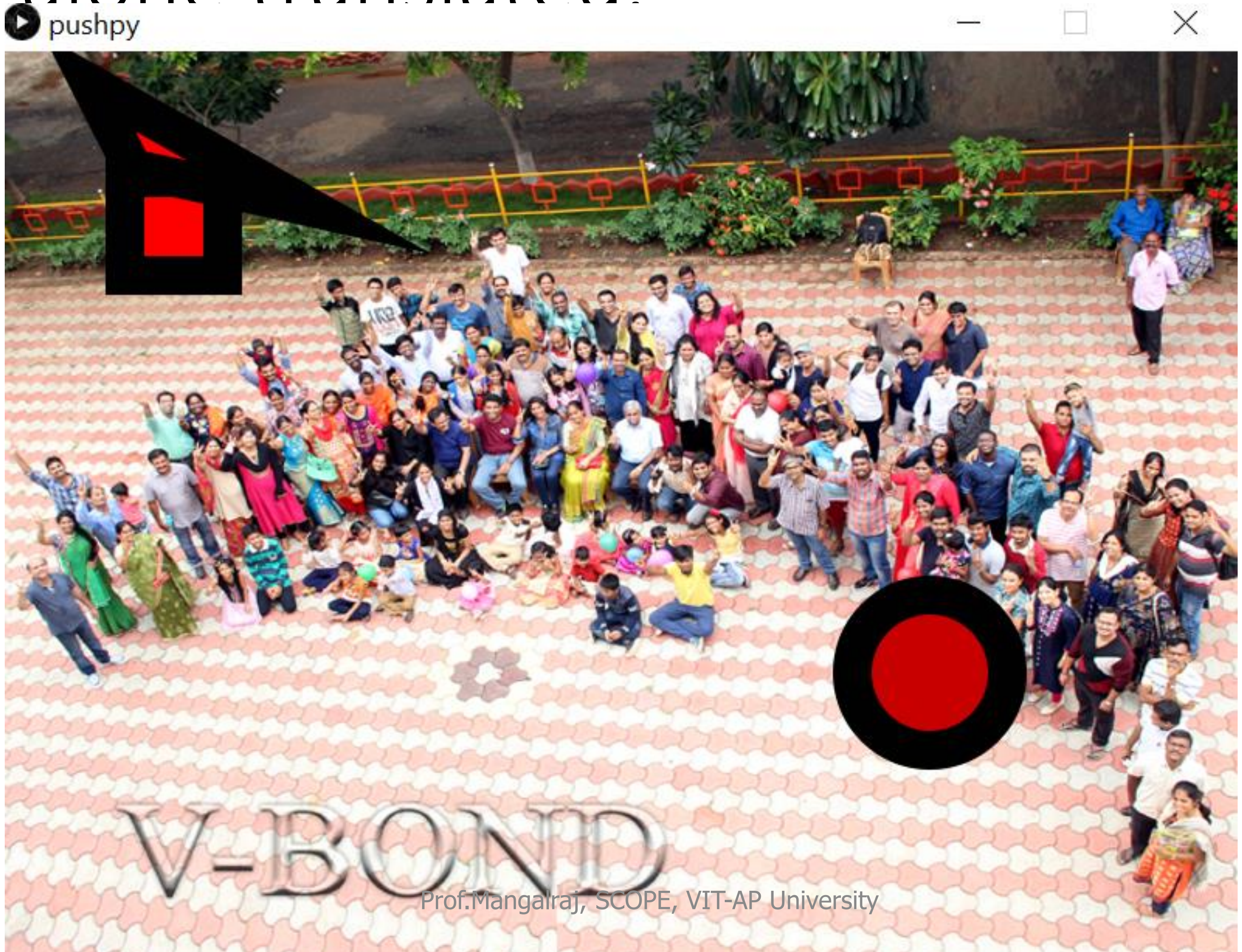
## Stroke color changed stroke(1,1,1);



I want to perform translation only on circle(black coloured) not on all objects. Then give circle translation func between push & pop matrix. Matrixstack is not suitable for colour.

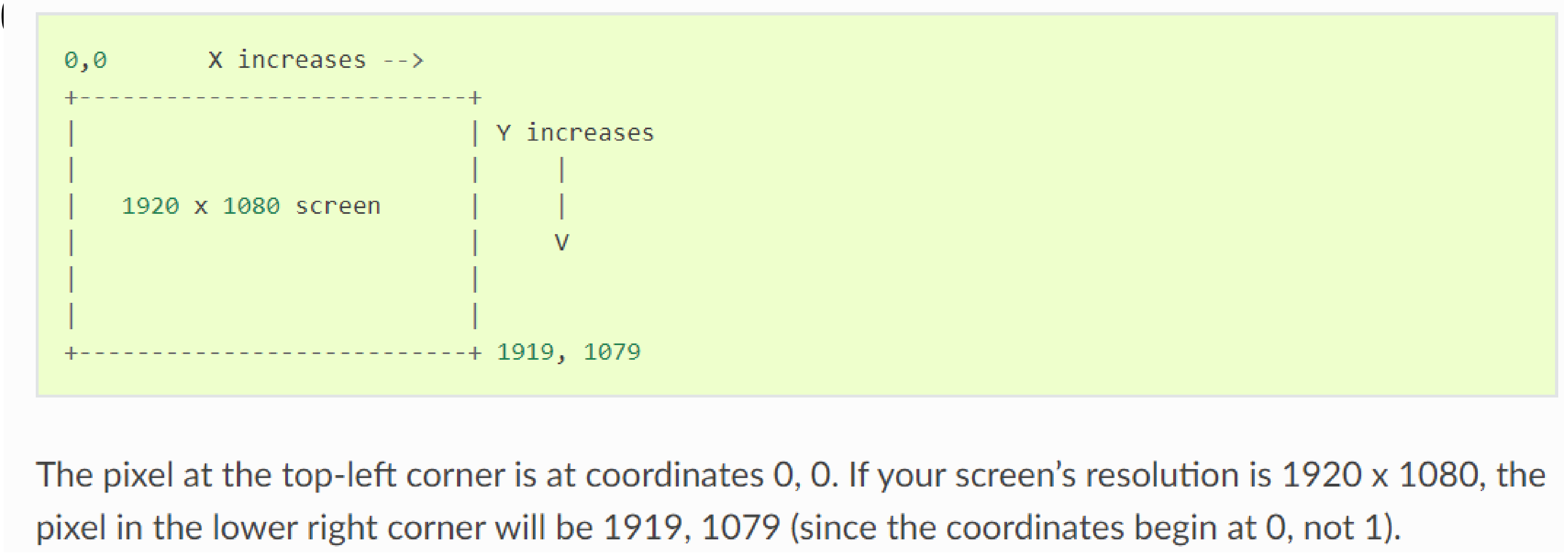
- def draw():
-     background(255);
-     #colorMode(RGB,100,500,10,255);
-     image(grid,0,0,800, 600);
-     pushMatrix();
-     strokeWeight(25);
-     translate(200,100);
- 
-     stroke(1,1,1);
-     fill(200,0,0);
-     ellipse(400,300,100,100);
-      #400+200=600,300+100=400,circle alone move now to (600,400)
-      popMatrix();

## Circle alone translated.



# The Screen and Mouse Position

- Locations on your screen are referred to by X and Y Cartesian coordinates. The X coordinate starts at 0 on the left side and increases going right. Unlike in mathematics, the Y coordinate starts at



```
0,0       X increases -->
+-------------------------+
|                         | Y increases
|                         |     |
|   1920 x 1080 screen    |     |
|                         |     V
|                         |
|                         |
+-------------------------+ 1919, 1079
```

The pixel at the top-left corner is at coordinates 0, 0. If your screen's resolution is 1920 x 1080, the pixel in the lower right corner will be 1919, 1079 (since the coordinates begin at 0, not 1).

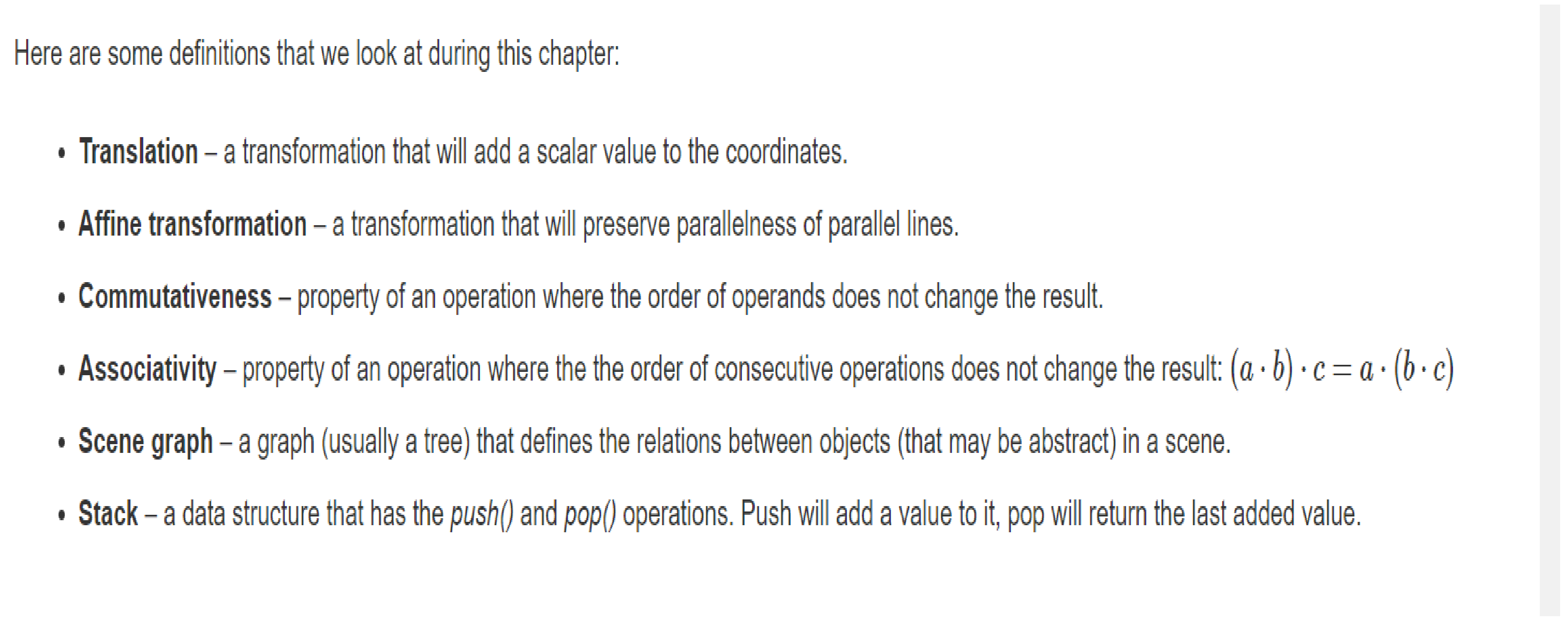
## Summary

- Geometric transformations I
  - Linear Transformation
  - Translation, Rotation, Scaling

- Geometric transformations II
  - Multiple transformations
  - Matrix Stack
  - Rotation revisted

http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

https://cglearn.codelight.eu/pub/computer-graphics/geometry-and-transformations-ii



Here are some definitions that we look at during this chapter:

- **Translation** – a transformation that will add a scalar value to the coordinates.
- **Affine transformation** – a transformation that will preserve parallelness of parallel lines.
- **Commutativeness** – property of an operation where the order of operands does not change the result.
- **Associativity** – property of an operation where the the order of consecutive operations does not change the result: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- **Scene graph** – a graph (usually a tree) that defines the relations between objects (that may be abstract) in a scene.
- **Stack** – a data structure that has the *push()* and *pop()* operations. Push will add a value to it, pop will return the last added value.

## References

- http://www.ecartouche.ch/content_reg/cartouche/graphics/en/html/Transform_learningObject1.html

- https://www.gatevidyalay.com/2d-transformation-in-computer-graphics-translation-examples/

- http://csis.pace.edu/~marchese/CG/Lect6/cg_l6_part1.htm