Lecture #02

2D Graphics

Computer Graphics

Winter term 2020/21

Marc Stamminger

Rendering

- "Rendering":
  - fill frame buffer with shapes, text, 3D-content, …

- Examples:
  - render a rectangle → simple
  - render a circle with radius r and center (x, y) → ???
  - render a line from (x_1, y_1) to (x_2, y_2) → ???
  - fill a triangle with vertices (x_1, y_1), (x_2, y_2), (x_3, y_3) → ???
  - → next week "Rasterization"

Rendering

- Frame buffer is usually not written directly, but via a **Graphics API**

- **Today we will have a brief look into such APIs, but this is not the general topic of this lecture**

- Mostly, we learn about **the algorithms** for rendering
- In the exercises, we will also look at the **graphics APIs**

Rendering

- Command-based APIs:
  - a library containing functions that render primitives, such as lines, rectangles, circles or similar
  - oftentimes, this includes the interaction with the GPU (a special device on the computer that is solely responsible for rendering)

- Scenegraph-based APIs:
  - the scene to be rendered is defined in an abstract manner in a hierarchical (tree-like) structure, which is passed to the renderer as a whole
  - In HTML, this can be integrated into the normal document hierarchy (see later)

- 3D:
  - the primitives to be rendered are defined in 3D-space. A virtual camera is to be specified that defines the mapping of the 3D-world to the 2D image. Also occlusion must be considered.

Rendering APIs

looked at in the exercises

	2D	3D
command based	• JAVA graphics 2D • HTML canvas • Postscript • …	• OpenGL • DirectX • …
scenegraph based	• SVG • …	• X3D • Unreal • Unity

today: 2D

3D comes soon

Today: 2D Graphics APIs

- We look into graphics APIs provided by HTML
- HTML-elements that can be filled with 2D graphics:

- **Canvas** Element: Command-based API

- **SVG** Element: Scenegraph-based API

HTML

- To use these, we must know very basic HTML

Hello World | Hello Canvas | Hello SVG

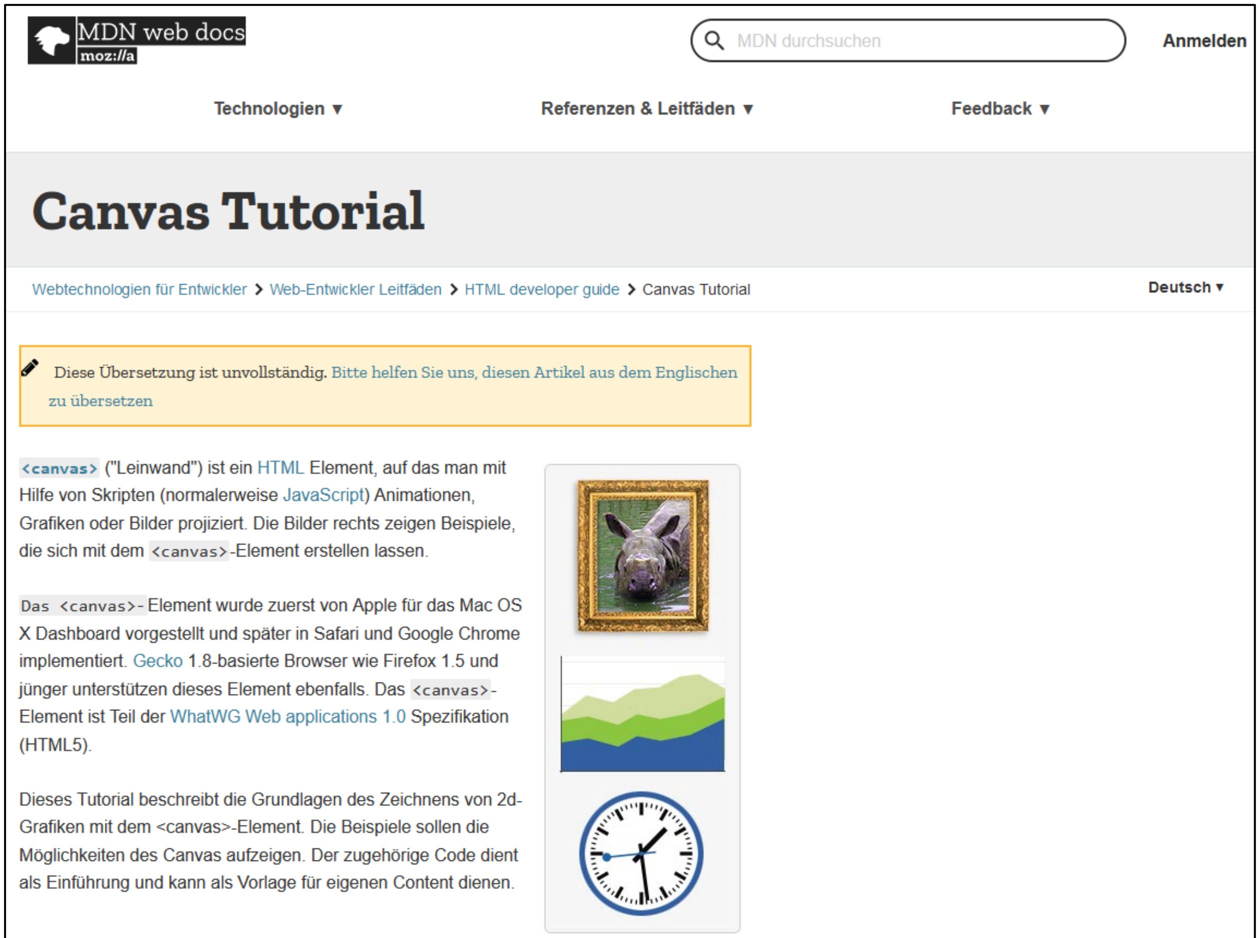
```
<body>
    <p>Hello World!</p>
</body>
```

>>>>

Hello World!

HTML5 Canvas

- We start with **canvas**. For more information see: https://developer.mozilla.org/de/docs/Web/Guide/HTML/Canvas_Tutorial



2D Graphics - Basics

- We start with the **canvas** element and its command-based API…
- … and then look into scene graphs

Most principles are the same for both API types:

- **Primitives**: Objects, from which an image is generated
- **Attributes** describe, how primitives are to be rendered (color, line width, …)
- **Transformations** are used to describe how objects are positioned within an image

Primitives

- Graphics are composed of *primitives* such as
  - lines
  - rectangles
  - circles / ellipses
  - triangles
  - polygons
  - curves
  - paths
- Each primitive has attributes such as
  - fill color
  - boundary color
  - line / boundary width
  - stipple pattern
  - …

Rectangles

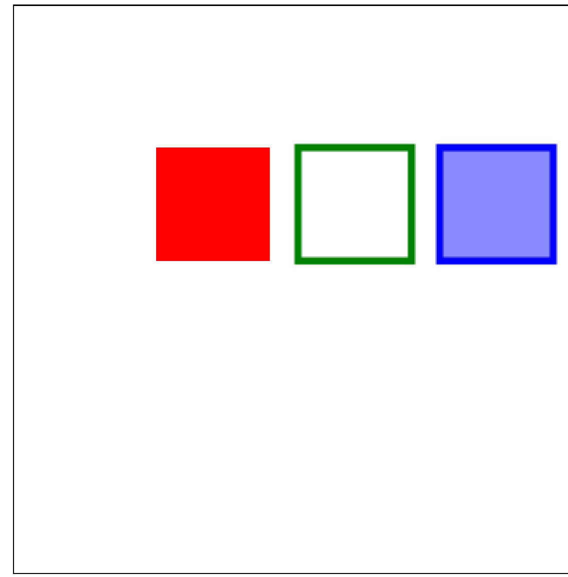
Rectangle

```
var context = canvas.getContext("2d");
context.fillStyle = 'red';
context.fillRect(100,100,80,80);

context.strokeStyle = 'green';
context.lineWidth = 5;
context.strokeRect(200,100,80,80)

context.fillStyle = '#88f';
context.strokeStyle = '#00f';
context.lineWidth = 5;
context.fillRect(300,100,80,80);
context.strokeRect(300,100,80,80);
```

>>>>



Attributes

- We have strokes (= lines) and fills (= areas)
- For strokes, we can define
  - its width → **linewidth**
  - its color → **strokeStyle**
  - line caps (shape of line ends)
  - line dash (stipple patterns)
  - …
- For fills, we can define
  - its color → **fillStyle**
  - fill patterns, fill gradients
  - …

- → for more information see https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Applying_styles_and_colors

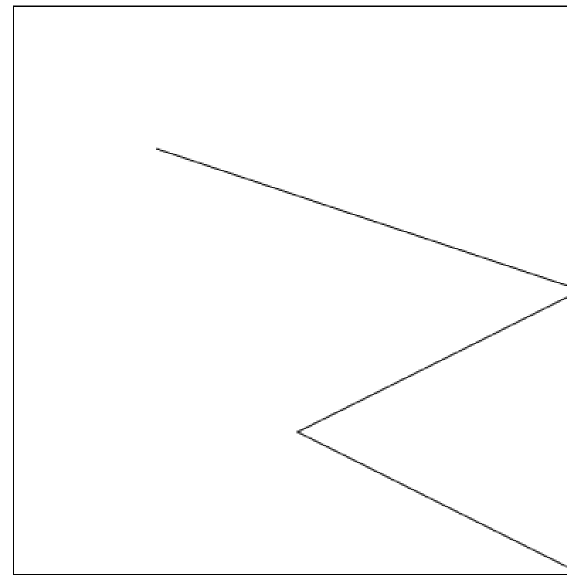
Paths

- With **canvas**, most 2D objects are defined as **paths**
- A path is a set of (joined) lines
- We can render the path as a stroke or fill it

Lines

```
var context = canvas.getContext("2d");
context.beginPath();
context.moveTo(100,100);
context.lineTo(400,200);
context.lineTo(200,300);
context.lineTo(400,400);
context.stroke();
```

>>>>



- Algorithms to render and fill line paths → next lecture(s)

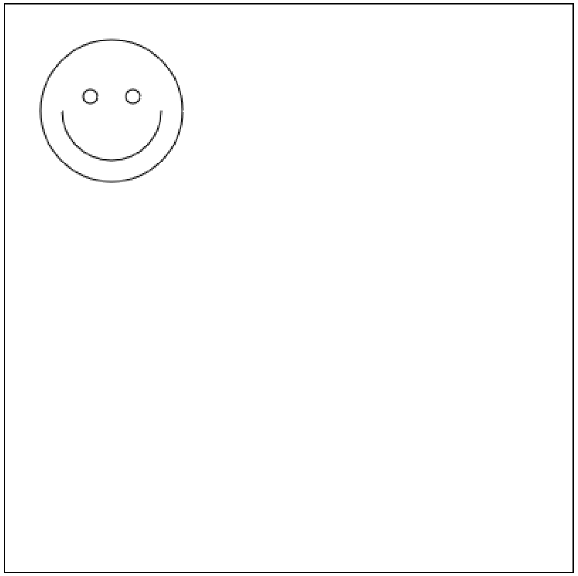
Primitives - Paths

- Paths can also contain circular (or elliptical) arcs

Arcs

```
var context = canvas.getContext("2d");
context.beginPath();
context.arc(75,75,50,0,Math.PI*2,true); // Outer circle
context.moveTo(110,75);
context.arc(75,75,35,0,Math.PI,false);  // Mouth (clockwise)
context.moveTo(65,65);
context.arc(60,65,5,0,Math.PI*2,true);  // Left eye
context.moveTo(95,65);
context.arc(90,65,5,0,Math.PI*2,true);  // Right eye
context.stroke();
```

>>>>



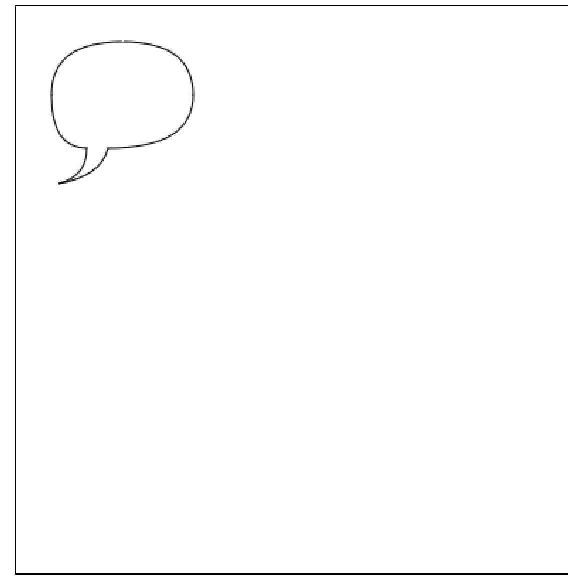
Primitives - Paths

- Paths can also contain Bezier curves

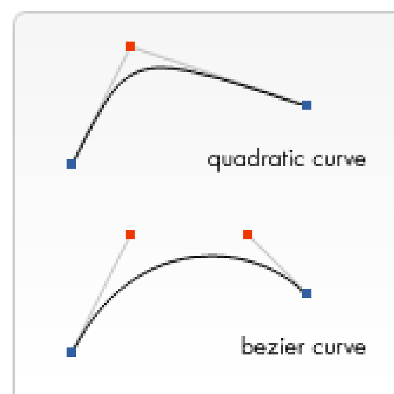
Beziér Curves

```
var context = canvas.getContext("2d");
context.beginPath();
context.moveTo(75,25);
context.quadraticCurveTo(25,25,25,62.5);
context.quadraticCurveTo(25,100,50,100);
context.quadraticCurveTo(50,120,30,125);
context.quadraticCurveTo(60,120,65,100);
context.quadraticCurveTo(125,100,125,62.5);
context.quadraticCurveTo(125,25,75,25);
context.stroke();
```

>>>>



- Bezier curves, conversion to line paths → lecture **“Geometric Modeling”**



Primitives – Filled Paths

- A path can also be filled (algorithm see next but one lecture)

Filled Path

```
var context = canvas.getContext("2d");
context.beginPath();
context.moveTo(75,25);
context.quadraticCurveTo(25,25,25,62.5);
context.quadraticCurveTo(25,100,50,100);
context.quadraticCurveTo(50,120,30,125);
context.quadraticCurveTo(60,120,65,100);
context.quadraticCurveTo(125,100,125,62.5);
context.quadraticCurveTo(125,25,75,25);

context.strokeStyle = "#000000";
context.lineWidth = 5;
context.stroke();

context.fillStyle = "pink";
context.fill();
```

>>>>

2D Transformations

- we can also apply transformations to objects:

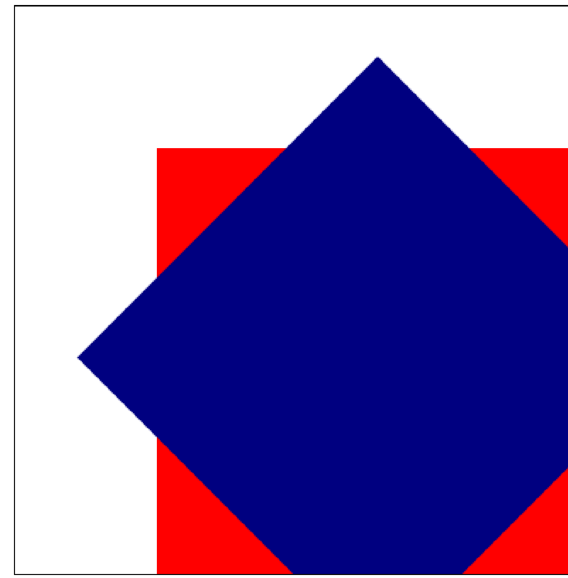
Basic

```
var context = canvas.getContext("2d");
context.resetTransform();

context.fillStyle = "red";
context.fillRect(100,100,300,300);

context.fillStyle = "navy";
context.translate(256,256);
context.rotate(Math.PI/4)
context.translate(-256,-256);
context.fillRect(100,100,300,300);
```

>>>>



Affine Transformations

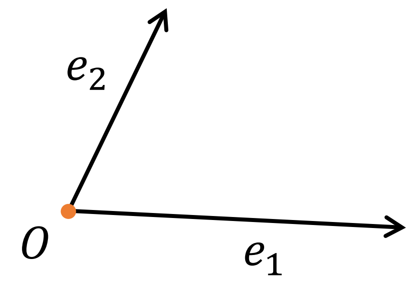
- Translate, rotate, and scale are **affine transformations**

- Important in CG
  - Positioning objects in a scene
  - Object Animations
  - Changing the shape of objects
  - Creation of multiple copies of objects

- Can be described easily using **Homogeneous Coordinates** and **Matrices**

Affine Transformations

- Coordinate Frames
  - Origin O (point)
  - Coordinate axes e_1, e_2 (vectors)

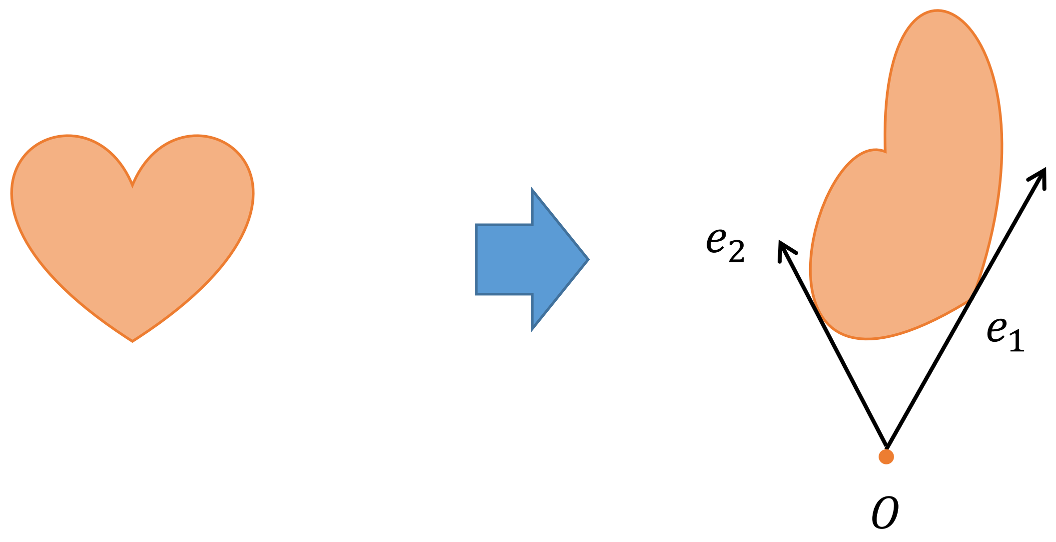


- Standard coordinate frame
  - $O = (0,0)$
  - $e_1 = (1,0), e_2 = (0,1)$

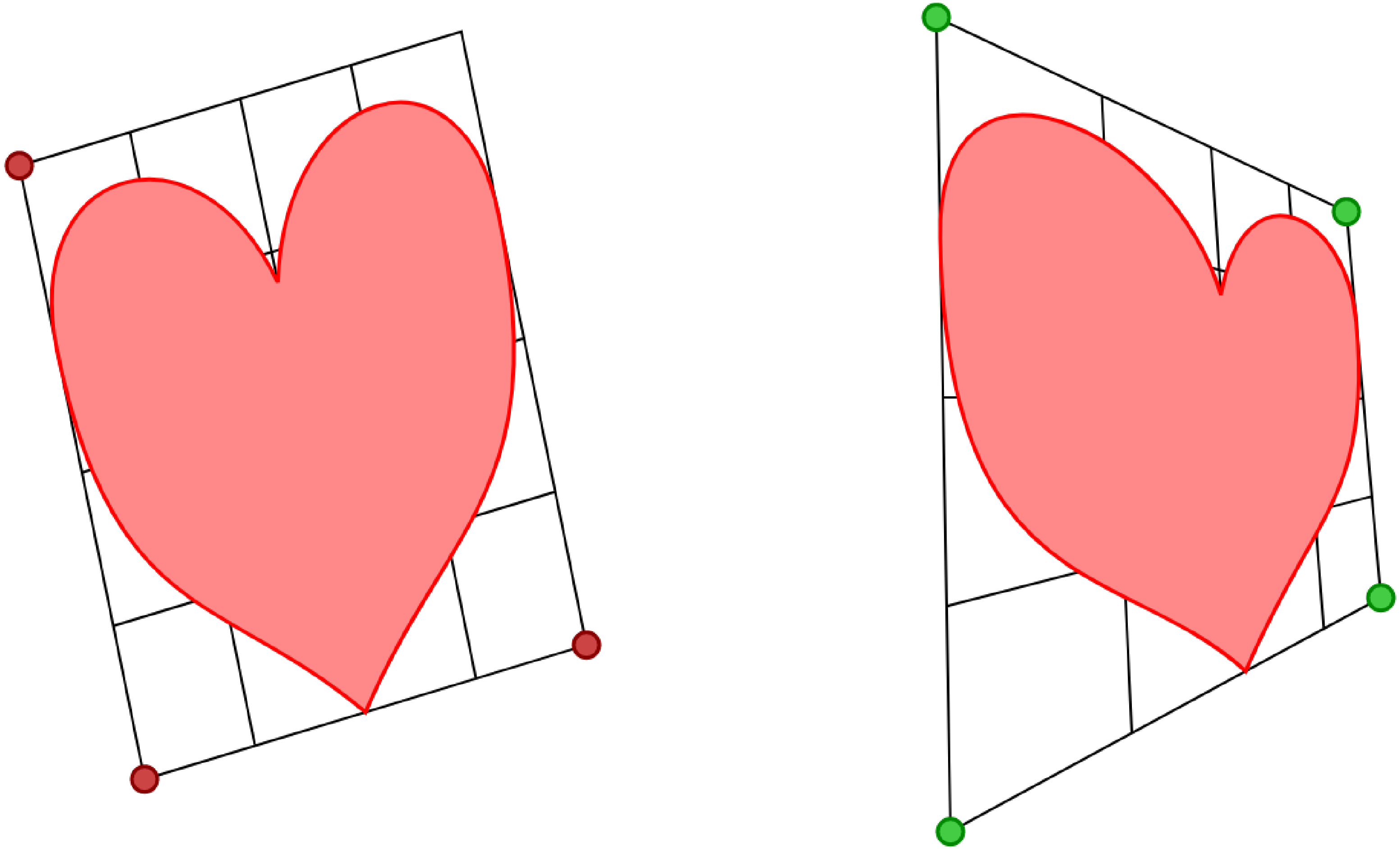
Affine Transformations

- Coordinate system change:

$$f(x,y) = O + xe_1 + ye_2$$



Affine Transformations



Affine Transformations

- We call such mappings *Affine Mappings*:

$$(x, y) \to \begin{pmatrix} e_1 & e_2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} O_1 \\ O_2 \end{pmatrix}$$

- more generally:

$$x \ \to Ax + t \qquad (A \in \mathbb{R}^{2\times 2}, t \in \mathbb{R}^2)$$

- concatenation results in another affine mapping:

$$x' = A_1 x + t_1$$
$$x'' = A_2 x' + t_2 = A_2(A_1 x + t_1) + t_2 = \underbrace{A_2 A_1}_{A_{concat}} x + \underbrace{A_2 t_1 + t_2}_{t_{concat}}$$

- we can apply a simple trick that allows us to also express affine transformations by a single matrix

  → homogeneous coordinates

Homogenous coordinates

- Add “1” as third homogeneous coordinate

$$\mathrm{x} = (x_1, \mathrm{x}_2) \rightarrow (x_1, \mathrm{x}_2, 1)$$

- To compute the mapping $Ax + t$ we apply a matrix of the form

$$\begin{pmatrix} A_{11} & A_{12} & t_1 \\ A_{21} & A_{22} & t_2 \\ 0 & 0 & 1 \end{pmatrix}$$

- $\begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} A_{11} & A_{12} & t_1 \\ A_{21} & A_{22} & t_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} = \begin{pmatrix} Ax + t \\ 1 \end{pmatrix}$

Homogenous coordinates

- If the last row of A is $(0,0,1)$ the mapping is affine
  → later we see how we can also use this row to express more general transformation

- Structure of a general affine transformation in homogeneous coordinates

linear part		translation
0	0	1

basis vectors after transf.

$\begin{pmatrix} e_{1,x} \\ e_{1,y} \end{pmatrix}, \begin{pmatrix} e_{2,x} \\ e_{2,y} \end{pmatrix}$		$\begin{pmatrix} t_x \\ t_y \end{pmatrix}$
0	0	1

Homogenous coordinates

- Transformation Rules & Matrix Operations

Multiplication $\equiv$ composition

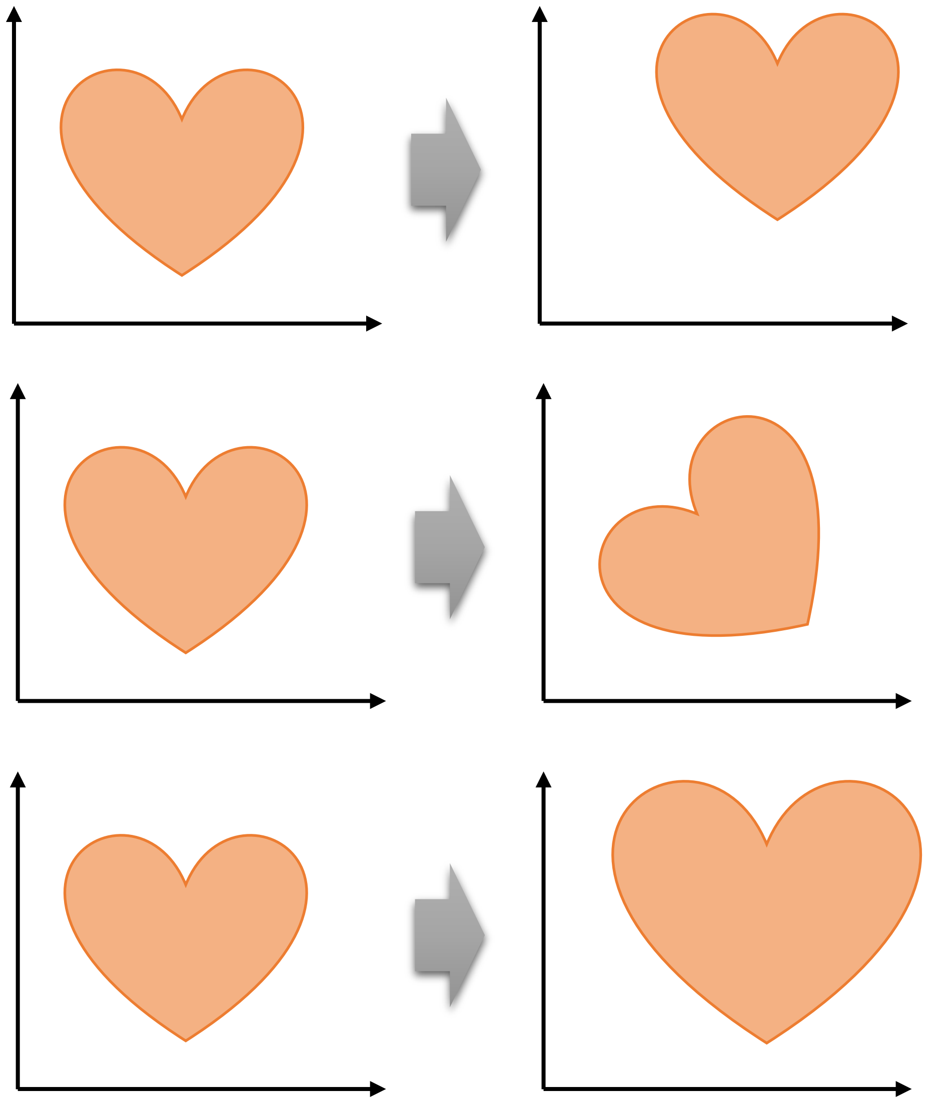
$$x \xrightarrow{T} Tx = y \xrightarrow{S} Sy = z \quad \equiv \quad x \xrightarrow{ST} STx = z$$

Inverse matrix $\equiv$ Inverse transformation

- Note order of multiplication: ST means: first T, then S

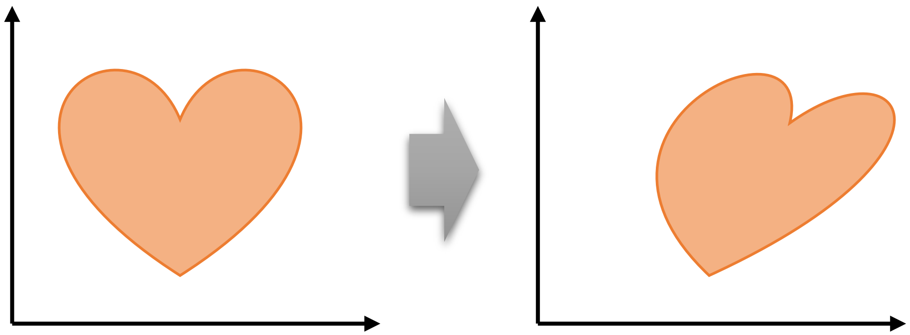
Affine Transformations

- Special cases
  - Translations
  - Rotations
  - Scalings

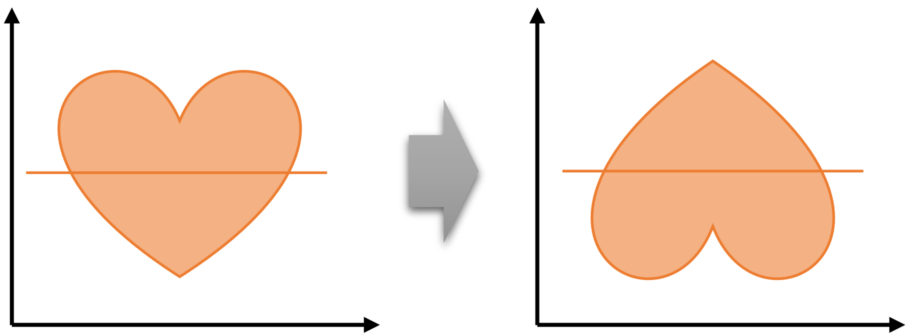


Affine Transformations

- Special cases
  - Shearings



  - Reflections

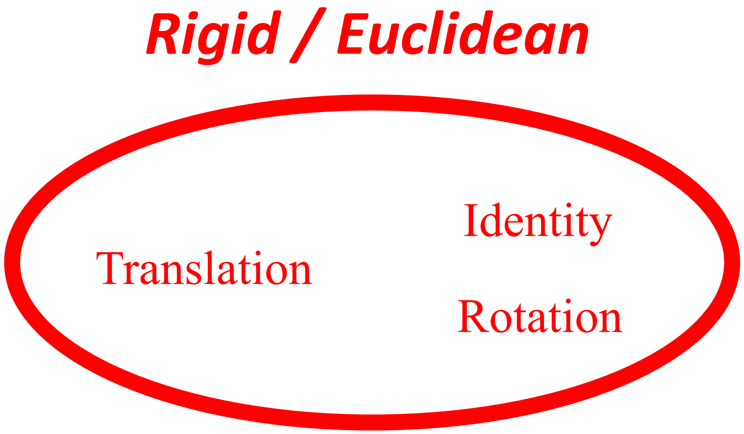


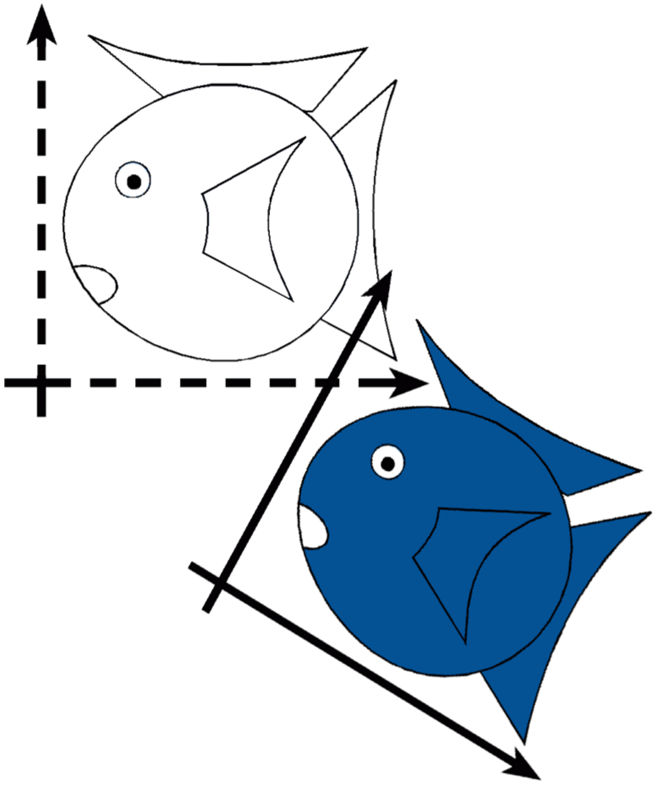
Affine Transformations

- Classes of Affine Transformations
  - Rigid
  - Similarity
  - Linear

Affine Transformations

- Rigid Transformation (Euclidean Transform)
  - Preserves distances
  - Preserves angles

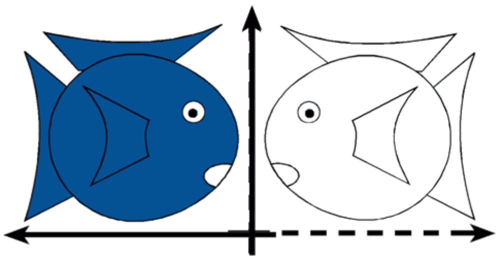




Affine Transformations

- Rigid transformation:
  e_1 and e_2 are orthonormal and have unit length
- $x \rightarrow Ax + t$ with A orthogonal and $\det(A) > 0$
- Application of multiple rigid transformations is a rigid transformation again (also true for following classes)

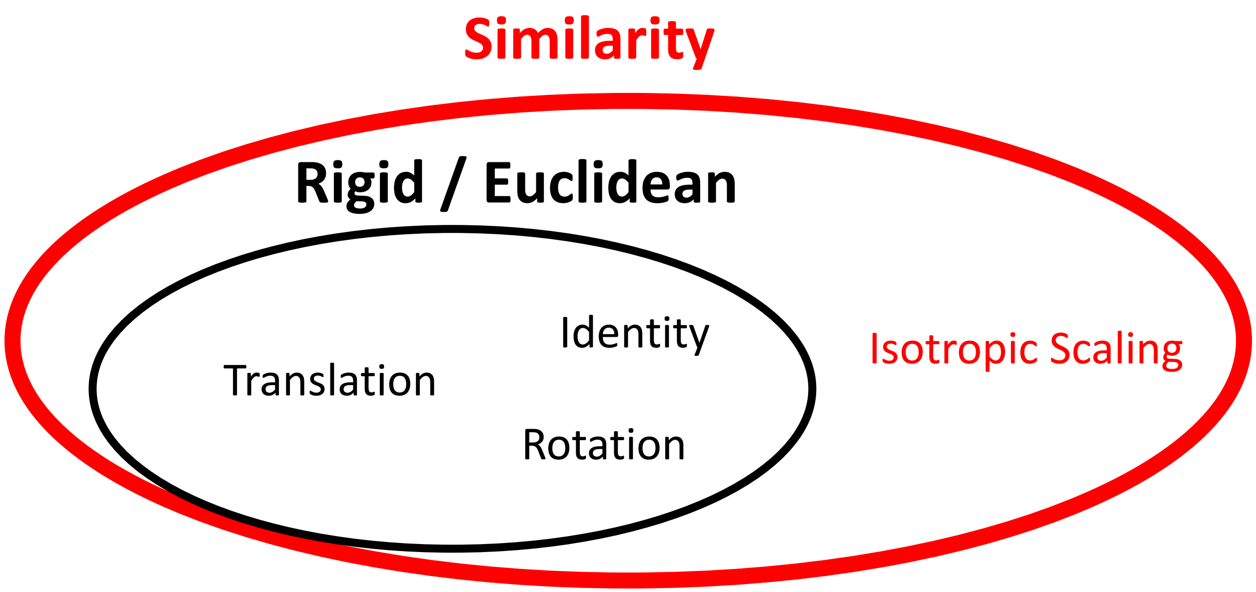
- If $\det(A) < 0$, A contains a reflection, which is not rigid

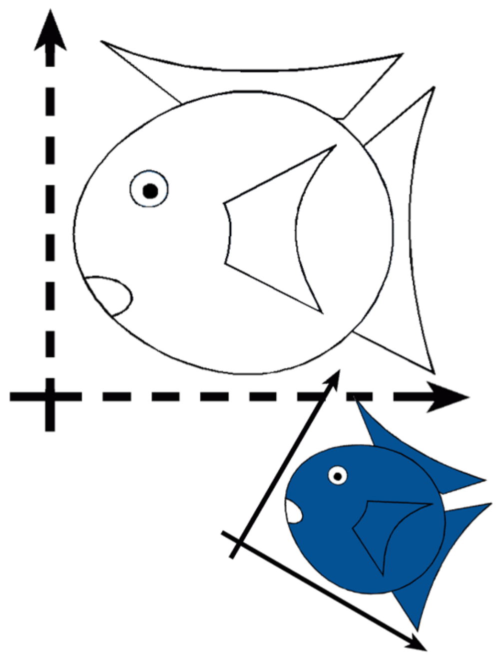


Affine Transformations

- Similarity Transforms
  - Preserves angles, but changes distances
  - Rigid + (isotropic) scaling + reflection

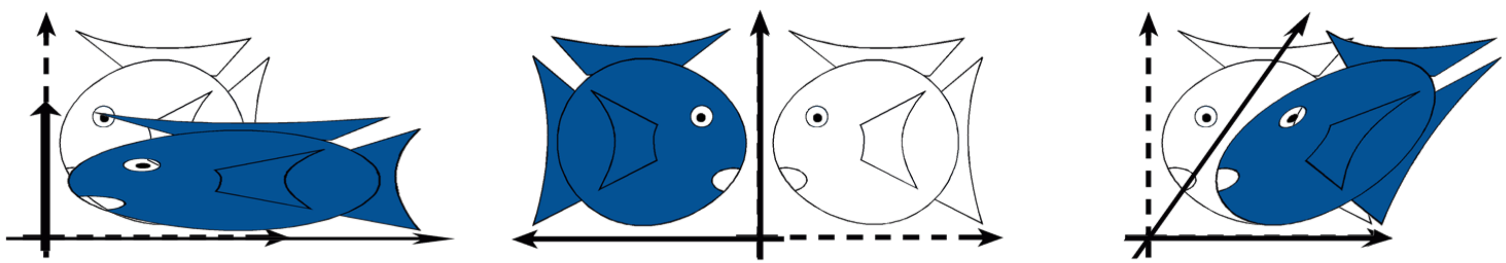
$x \rightarrow cAx + t$ with $c \in \mathbb{R}$ and A orthonormal



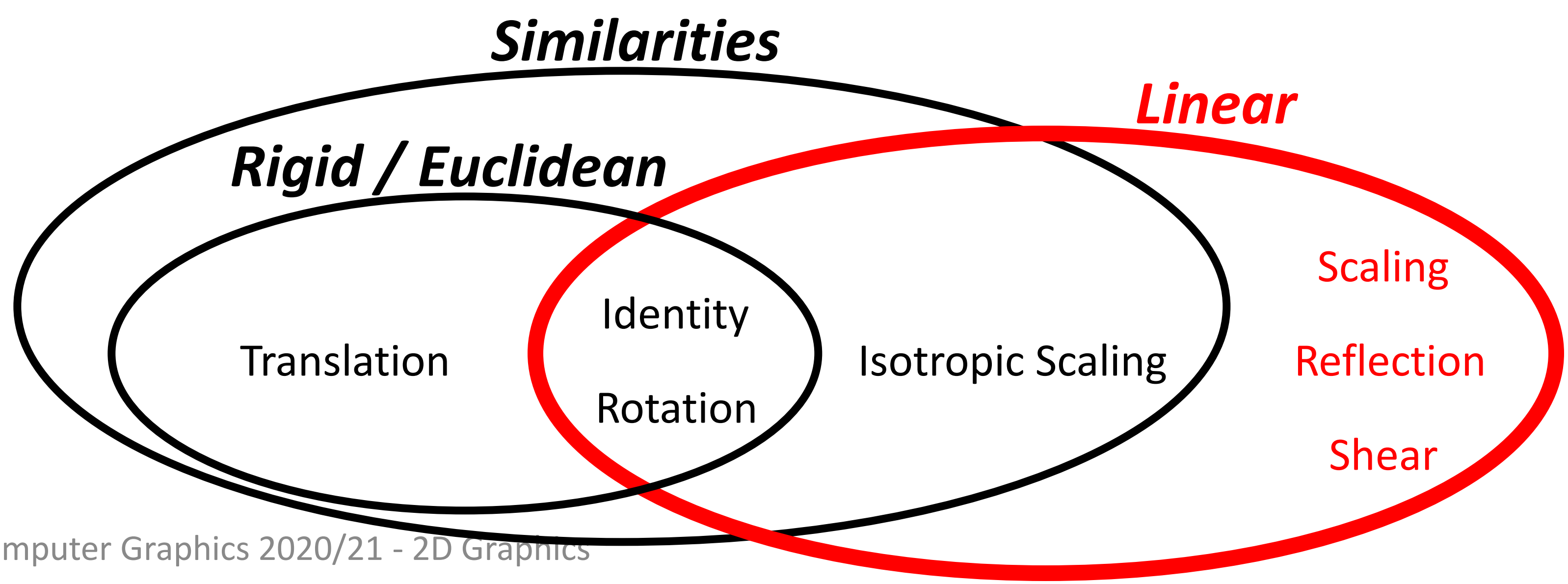


Affine Transformations

- General Linear Transformations = affine without translation
- Origin (0,0) is always mapped to origin

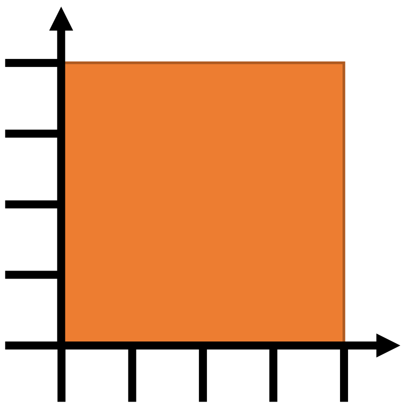


Scaling Reflection Shear

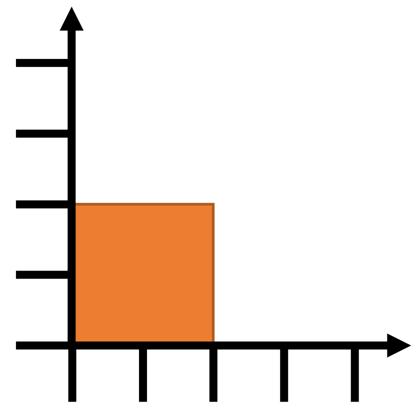


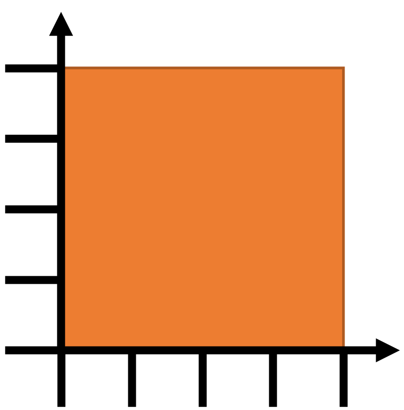
Scaling

- Examples

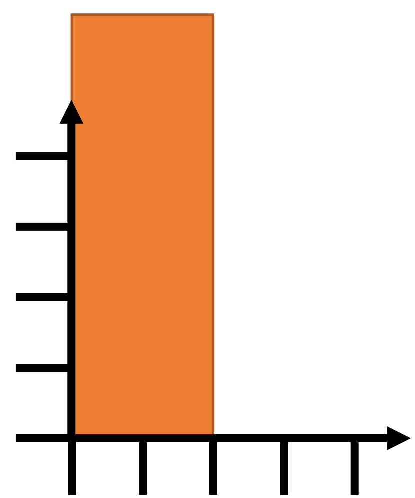


$$\begin{pmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$





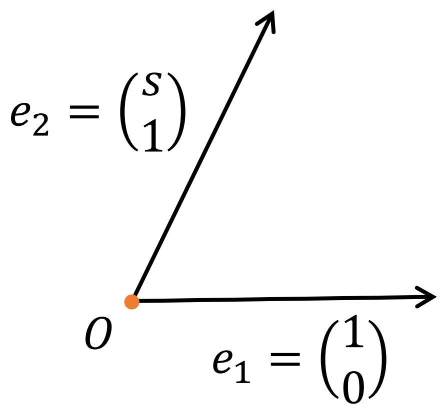
$$\begin{pmatrix} 0.5 & 0 & 0 \\ 0 & 1.5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



Shearing

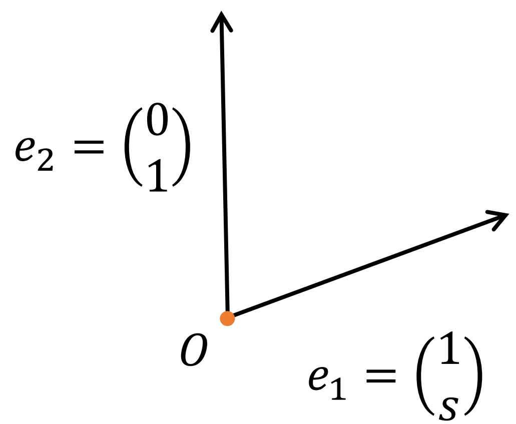
- Shearing
  - Pushing things sideways *(compare deck of cards)*
  - Horizontal (y-coordinate will not change)

$$shear_x(s) = \begin{pmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



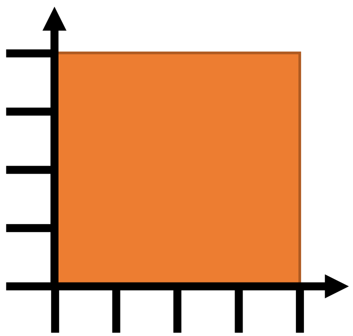
  - Vertical (x-coordinate will not change)

$$shear_y(s) = \begin{pmatrix} 1 & 0 & 0 \\ s & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

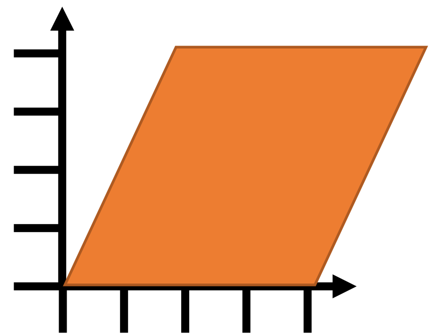


Shearing

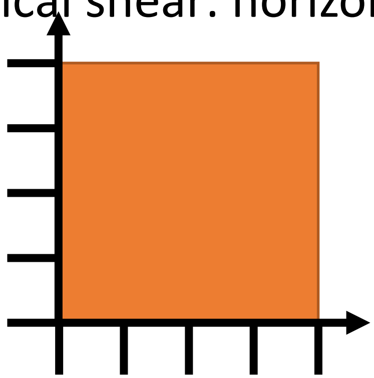
- Examples
  - Horizontal shear: vertical lines → 45°to the right



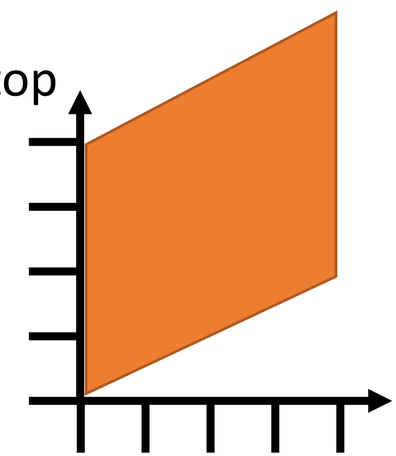
$$\begin{pmatrix} 1 & 0.5 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



  - Vertical shear: horizontal lines → 45°to the top

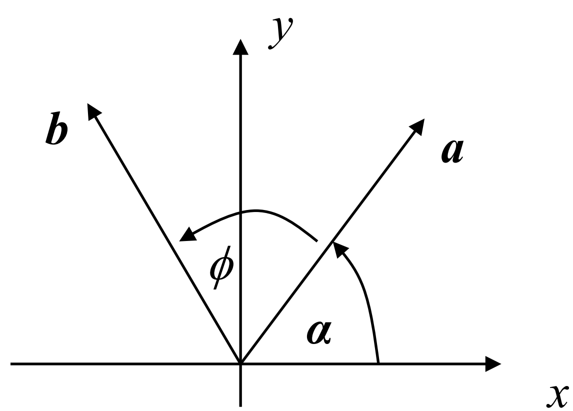


$$\begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



Simple Rotation in 2D

- Rotation
  - Vector $\boldsymbol{a} = (a_x, a_y)$, angle α with x-axis
  - Length $r = \sqrt{a_x^2 + a_y^2}$
  - By definition: $$a_x = r\cos\alpha, \quad a_y = r\sin\alpha$$
  - Rotation by an angle ϕ counter-clockwise:
$$b_x = r\cos(\alpha + \phi) = r\cos\alpha\cos\phi - r\sin\alpha\sin\phi$$
$$b_y = r\sin(\alpha + \phi) = r\sin\alpha\cos\phi + r\cos\alpha\sin\phi$$

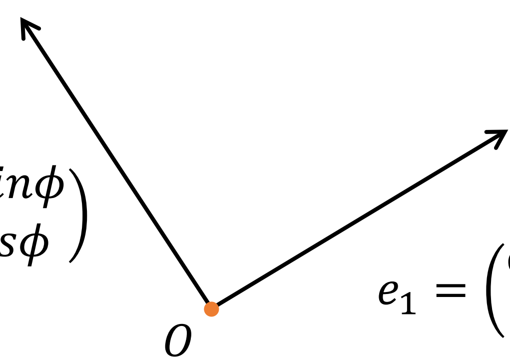


Simple Rotation in 2D

- After substitution
  - $b_x = a_x \cos\phi \, - \, a_y \sin\phi$
  - $b_y = a_y \cos\phi \, + \, a_x \sin\phi$

- Matrix form taking a to b

$$rotate(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

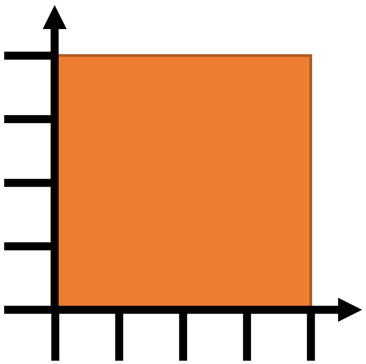


$$e_2 = \begin{pmatrix} -sin\phi \\ cos\phi \end{pmatrix}$$

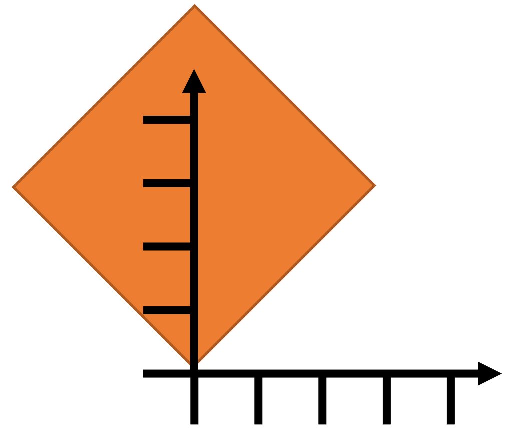
$$e_1 = \begin{pmatrix} \cos\phi \\ \sin\phi \end{pmatrix}$$

Simple Rotation in 2D

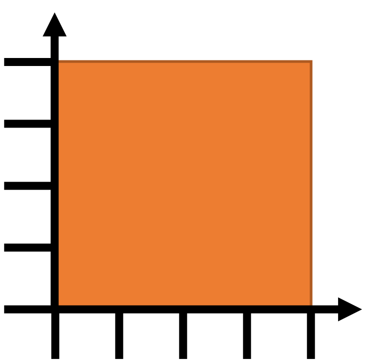
- Rotation by 45°counter-clockwise



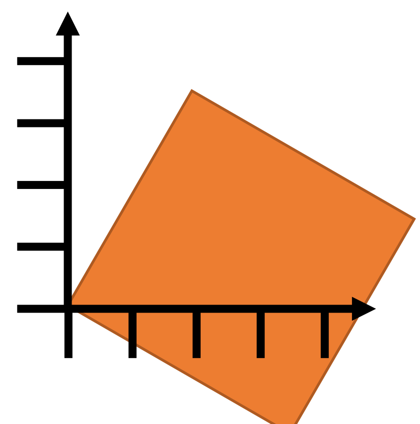
$$\begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



- Rotation by 30° clockwise

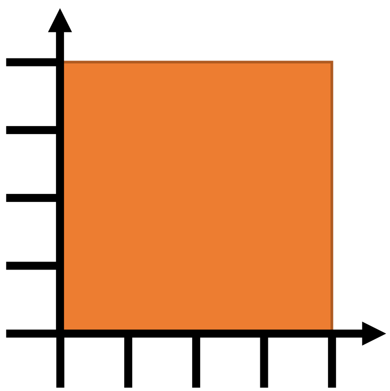


$$\begin{pmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} & 0 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

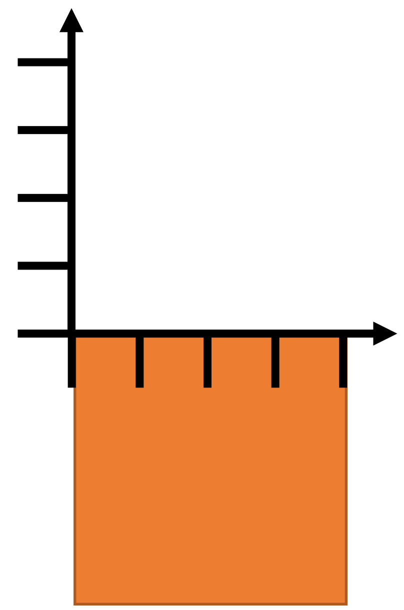


Reflection

- Reflection
  - Reflect a vector across either of the coordinate axes
  - Determinant of a reflection is negative
  - About x-axis (multiply y by -1):

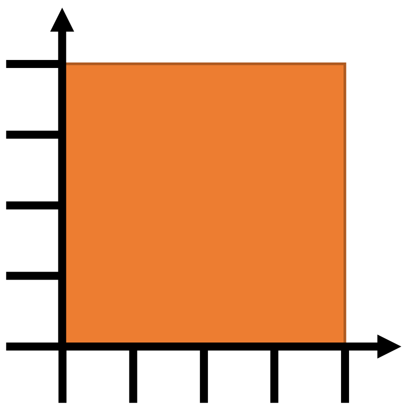


$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

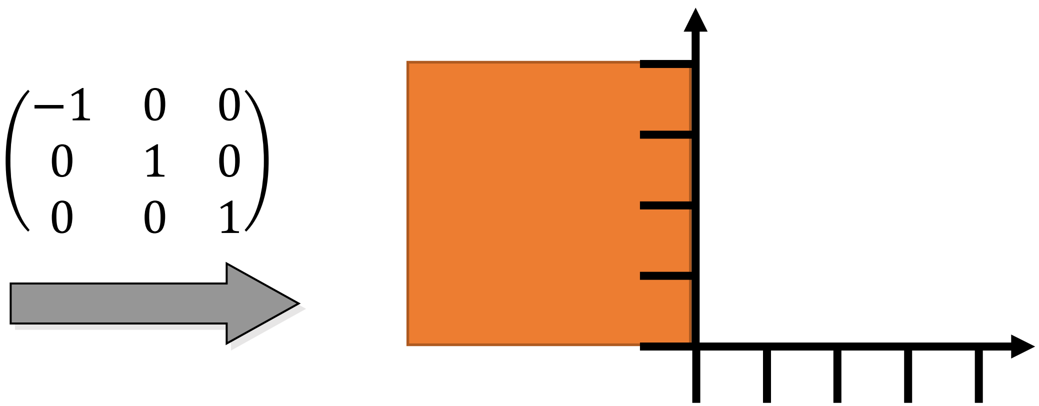


Reflection

- Across y-axis (multiply x coordinates by -1)

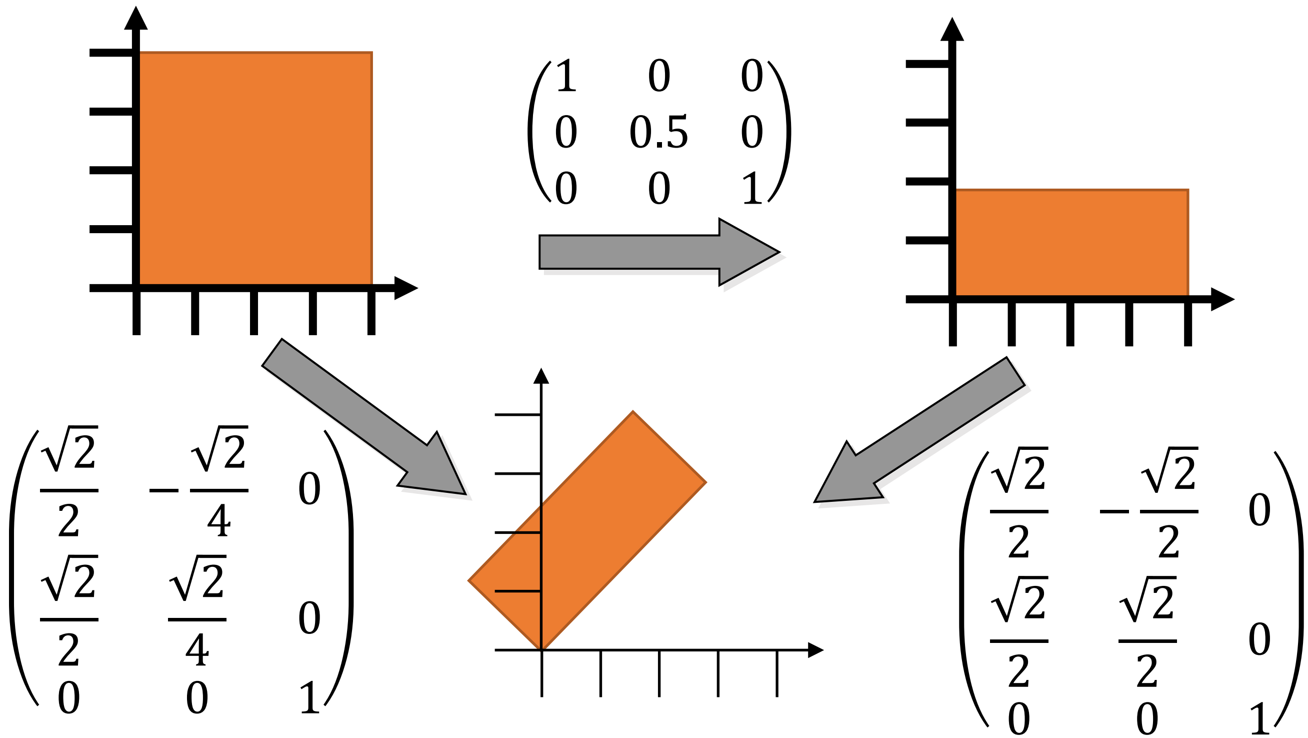


$$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



Linear Transformations

- Compositing of 2D transformations
  - First $v_2 = Sv_1$ then $v_3 = Rv_2$
  - Equivalently $v_3 = R(Sv_1) = (RS)v_1$



$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{4} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{4} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Linear Transformations

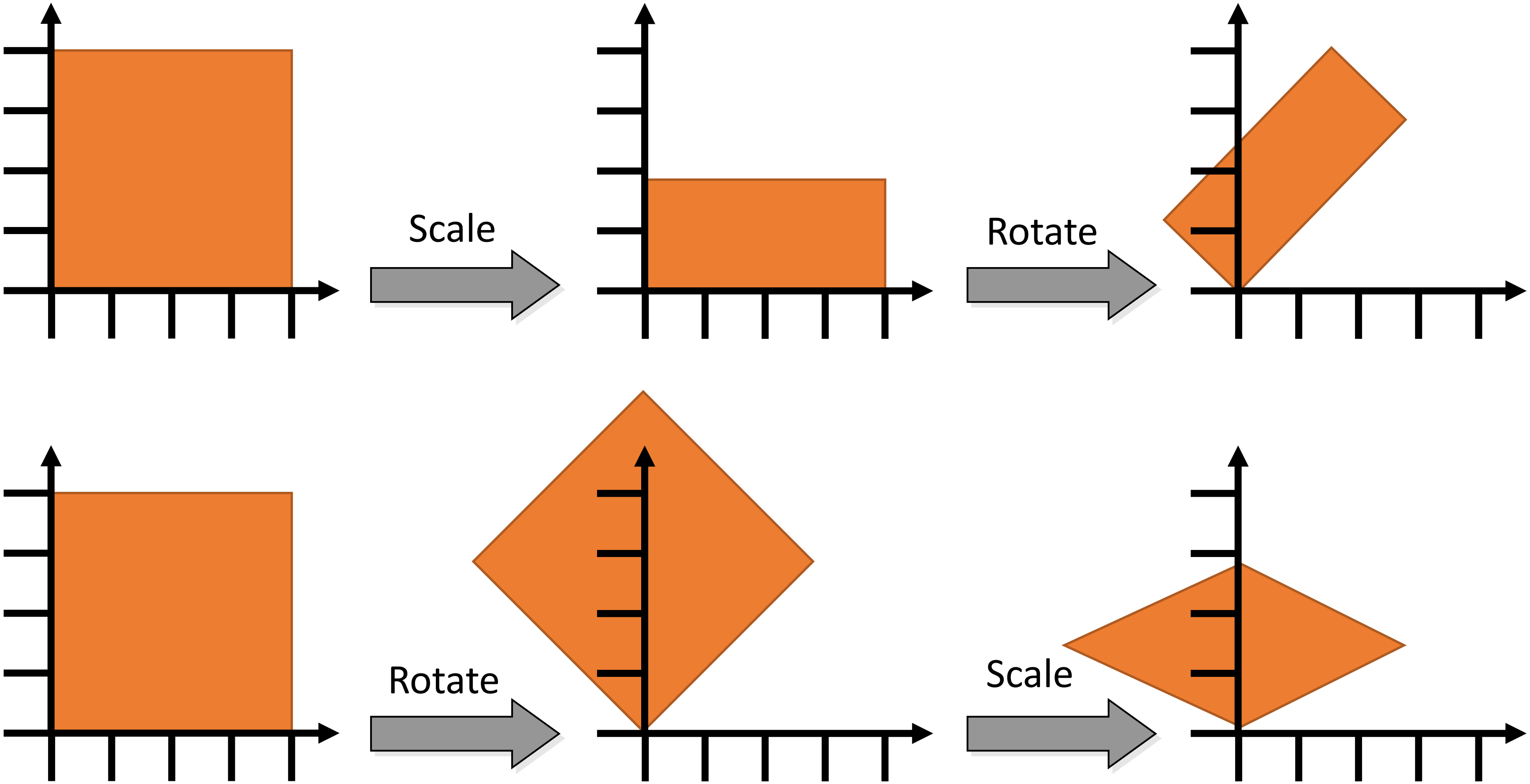
- Matrix multiplications are associative:

$$(RS)T = R(ST) \rightarrow v_3 = (RS)v_1 = Mv_1 \text{ with } M = RS$$

- Matrix multiplications are **not** commutative
  - **The order** of transformations **does matter** !!!
  - Note the difference
    - Scaling then rotating
    - Rotating then scaling

Linear Transformations

- Note that the order of transformations is important



Linear Transformations

- Decomposition of transformations
  - Write some transformation M as the product of certain classes of matrices

- In 2D: Decomposition of any linear 2D transform into product: rotation → scale → rotation = $R_2 S R_1$
  - From existence of singular value decomposition (SVD) (Singulärwertzerlegung, Ausgleichsprobleme)
  - Note that the scale can have negative entries

Linear Transformations

- Example: shearing
  - σ_i singular values, R_1 and R_2 rotations

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = R_2 \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} R_1$$

$$= \begin{pmatrix} 0.851 & -0.526 \\ 0.526 & 0.851 \end{pmatrix} \begin{pmatrix} 1.618 & 0 \\ 0 & 0.618 \end{pmatrix} \begin{pmatrix} 0.526 & 0.851 \\ -0.851 & 0.526 \end{pmatrix}$$

R_1 SR_1 R_2SR_1

Linear Transformations

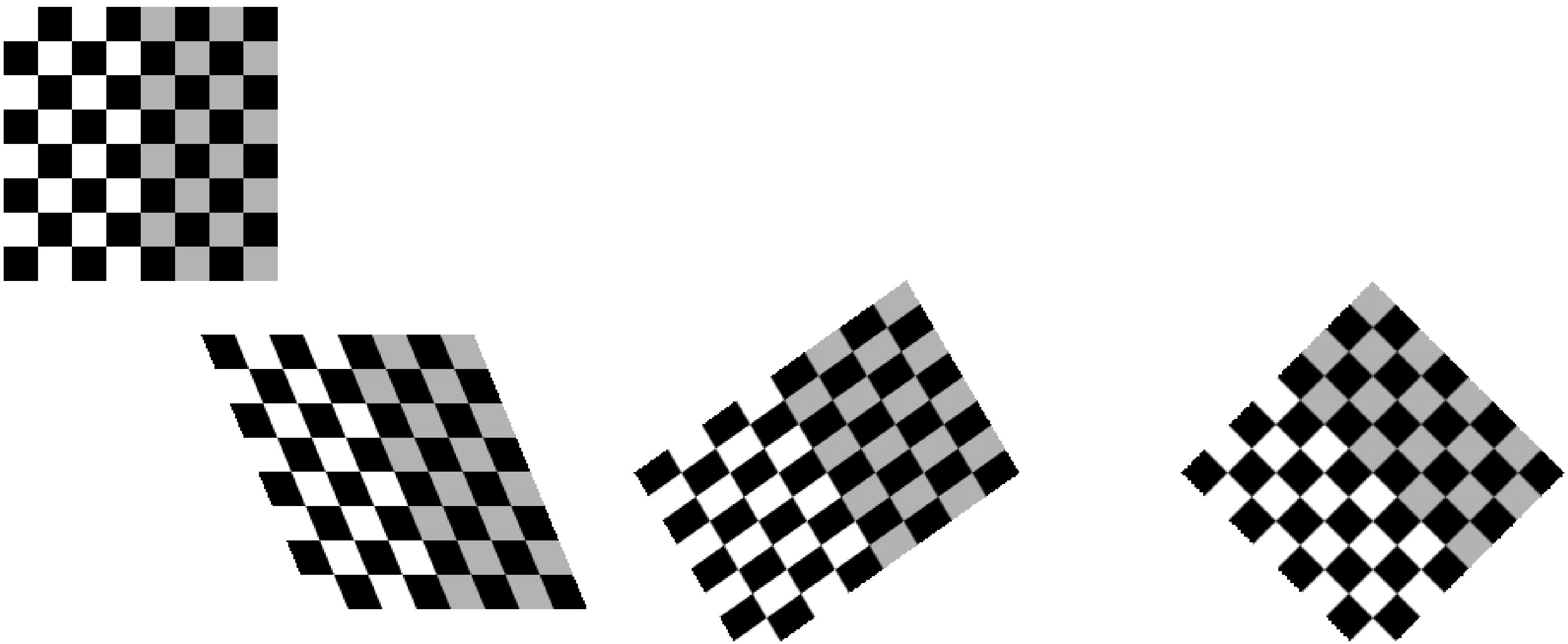
- Matrix decomposition: represent rotations with shears

$$\begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} = \begin{pmatrix} 1 & \dfrac{\cos\phi - 1}{\sin\phi} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \sin\phi & 1 \end{pmatrix} \begin{pmatrix} 1 & \dfrac{\cos\phi - 1}{\sin\phi} \\ 0 & 1 \end{pmatrix}$$

- Useful for raster rotation
  - Very efficient raster operation for images: only column-wise and row-wise operations!
  - Introduces some jaggies but no holes

Linear Transformations

- $rotate\left(\frac{\pi}{4}\right) = S_3 S_2 S_1 = \begin{pmatrix} 1 & 1-\sqrt{2} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \frac{\sqrt{2}}{2} & 1 \end{pmatrix} \begin{pmatrix} 1 & 1-\sqrt{2} \\ 0 & 1 \end{pmatrix}$



S_1 S_2S_1 $S_3S_2S_1$

Linear Transformations

- Images – simple raster rotation
  - Take raster position (i, j) and apply horizontal shear

$$\begin{pmatrix} 1 & s \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i + sj \\ j \end{pmatrix}$$

  - Round sj to nearest integer: in every row a constant shift
  - Move each row sideways by a different amount
  - Resulting image has no gaps

Example

- Affine Transformations with the HTML Canvas

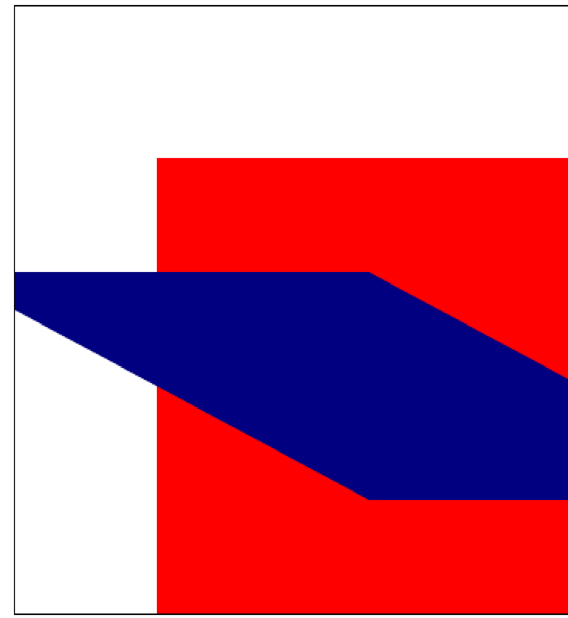
Matrices

```
var context = canvas.getContext("2d");
context.resetTransform();

context.setTransform(1,0,0,1,0,0);
context.fillStyle = "red";
context.fillRect(100,100,300,300);

context.setTransform(1,0,1,.5,-250,125);
context.fillStyle = "navy";
context.fillRect(100,100,300,300);
```

>>>>



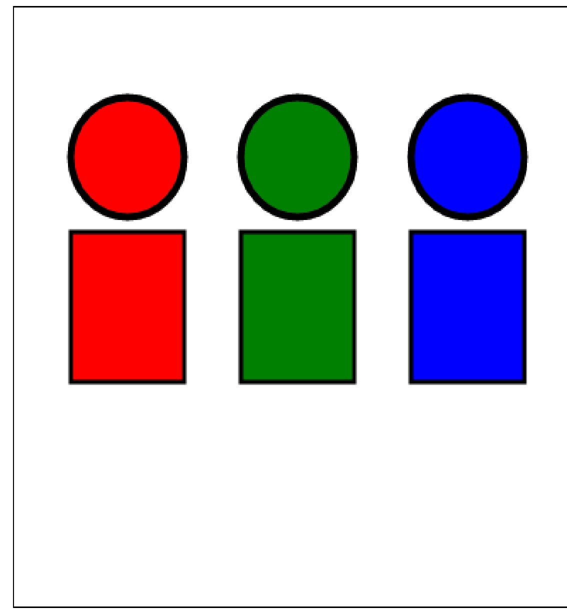
HTML5 SVG (**S**calable **V**ector **G**raphics)

- **Scene Graph based** Graphics APIs
- contains primitives as children, including their attributes (note the slightly different attribute names)

Circles and rectangles

```
<circle cx="80" cy="100" r="40" stroke="black" stroke-width="5" fill="red"></circ
<circle cx="200" cy="100" r="40" stroke="black" stroke-width="5" fill="green"></c
<circle cx="320" cy="100" r="40" stroke="black" stroke-width="5" fill="blue"></ci
<rect x="40" y="150" width="80" height="100" stroke="black" stroke-width="3" fill
<rect x="160" y="150" width="80" height="100" stroke="black" stroke-width="3" fil
<rect x="280" y="150" width="80" height="100" stroke="black" stroke-width="3" fil
```

>>>>



- for more information see: https://developer.mozilla.org/de/docs/Web/SVG

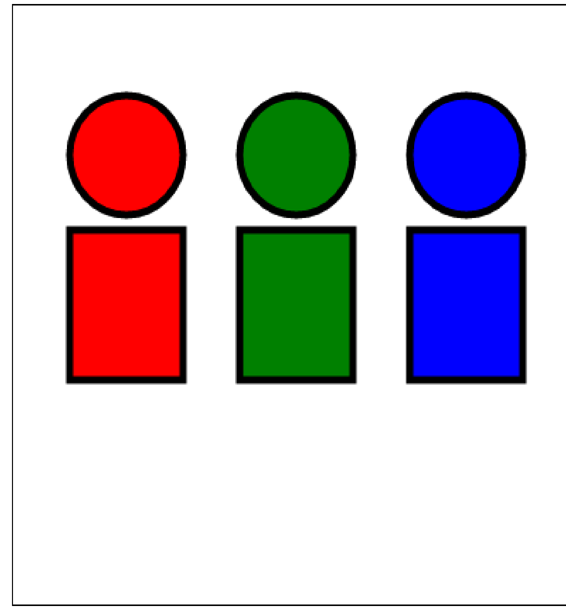
HTML5 SVG (**S**calable **V**ector **G**raphics)

- primitives can be grouped using a group node with tag “**g**”
  - nodes form a tree
  - attributes from inner nodes are valid for entire subtree

Groups

```
<g stroke="black" stroke-width="5">
    <g fill="red">
        <circle cx="80" cy="100" r="40"></circle>
        <rect x="40" y="150" width="80" height="100"></rect>
    </g>
    <g fill="green">
        <circle cx="200" cy="100" r="40"></circle>
        <rect x="160" y="150" width="80" height="100"></rect>
    </g>
    <g fill="blue">
        <circle cx="320" cy="100" r="40"></circle>
        <rect x="280" y="150" width="80" height="100"></rect>
    </g>
</g>
```

>>>>



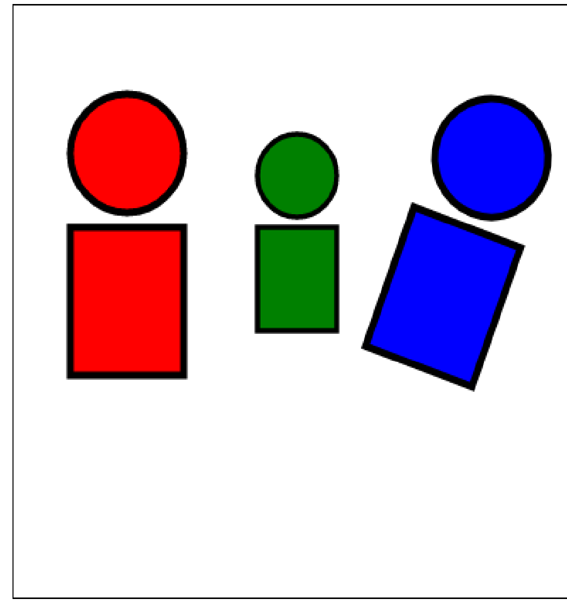
HTML5 SVG (**S**calable **V**ector **G**raphics)

- nodes can be transformed using an attribute “**transform**”

Transformations

```
<g stroke="black" stroke-width="5">
    <g fill="red" transform="translate(80,150)">
        <circle cy="-50" r="40"></circle>
        <rect x="-40" width="80" height="100"></rect>
    </g>
    <g fill="green" transform="matrix(0.7 0 0 0.7 200 150)">
        <circle cy="-50" r="40"></circle>
        <rect x="-40" width="80" height="100"></rect>
    </g>
    <g fill="blue" transform="translate(320,150) rotate(20)">
        <circle cy="-50" r="40"></circle>
        <rect x="-40" width="80" height="100"></rect>
    </g>
</g>
```

>>>>



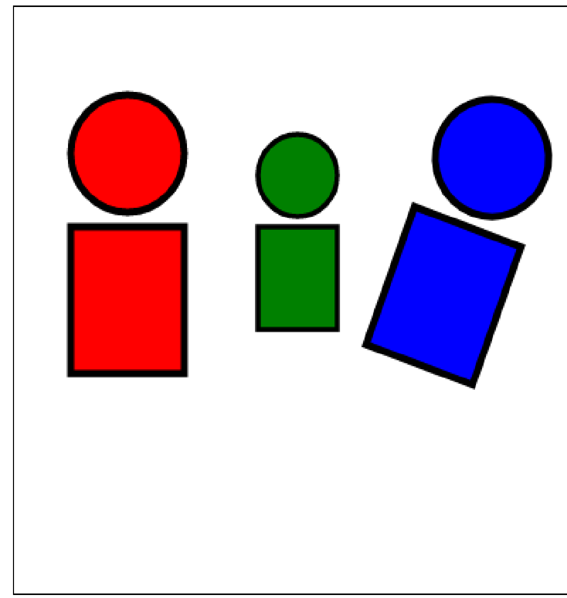
HTML5 SVG (**S**calable **V**ector **G**raphics)

- In the previous example the leaf nodes are identical
- reuse one instance multiple times → **use** elements

Scene Graph

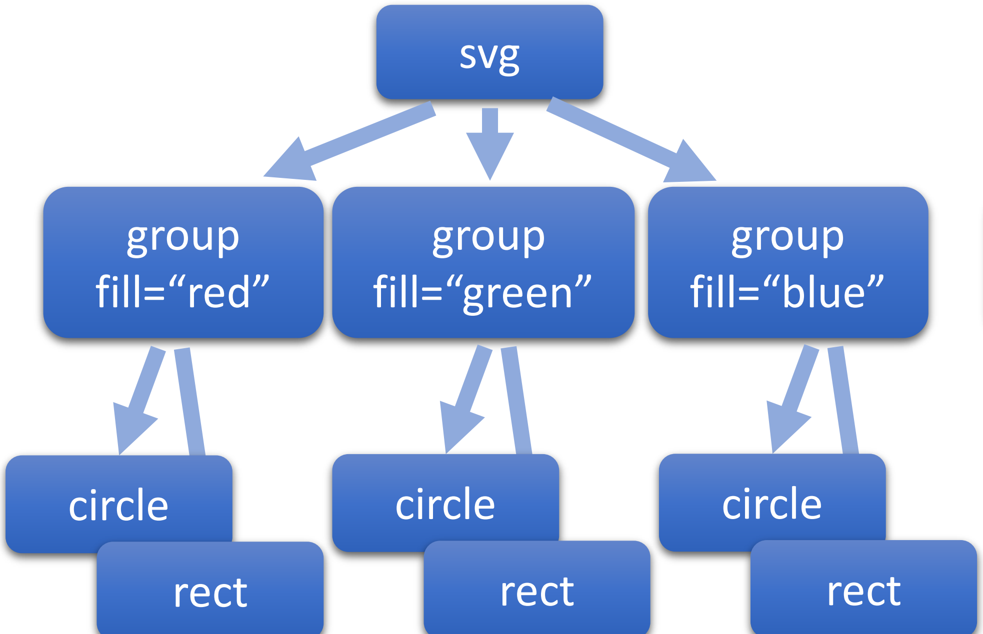
```
<defs>
    <g id="shape">
        <circle cy="-50" r="40"></circle>
        <rect x="-40" width="80" height="100"></rect>
    </g>
</defs>
<g stroke="black" stroke-width="5">
    <g fill="red" transform="translate(80,150)">
        <use xlink:href="#shape"></use>
    </g>
    <g fill="green" transform="matrix(0.7 0 0 0.7 200 150)">
        <use xlink:href="#shape"></use>
    </g>
    <g fill="blue" transform="translate(320,150) rotate(20)">
        <use xlink:href="#shape"></use>
    </g>
</g>
```

>>>>

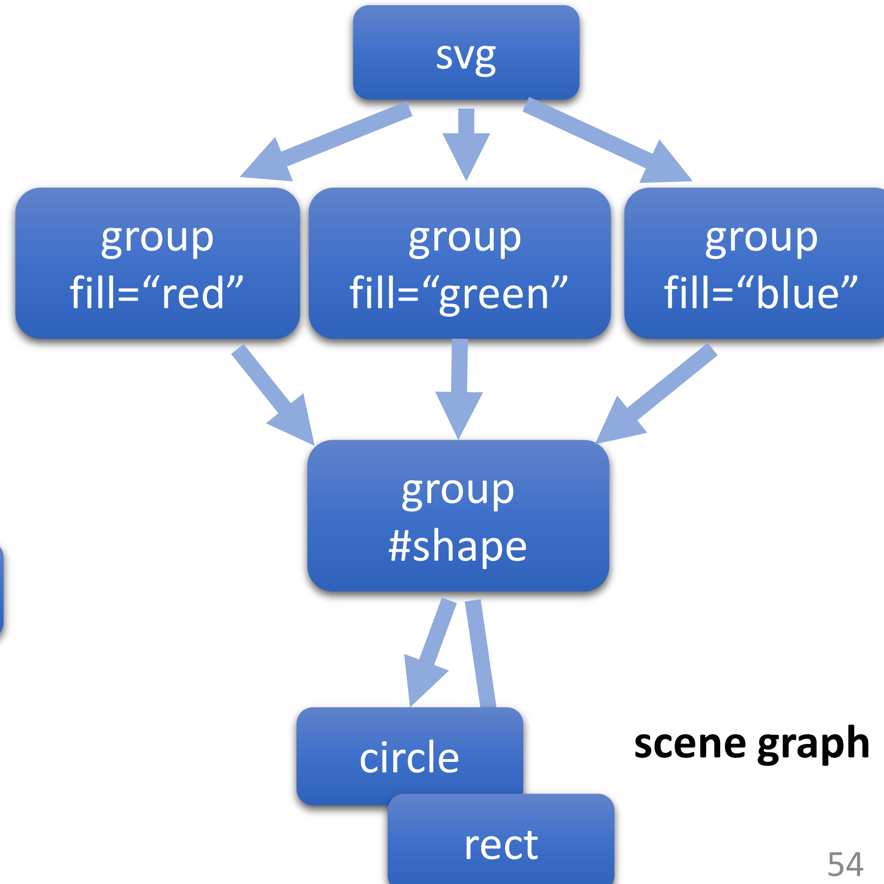


Scene Graph

- reusing nodes turns the **scene tree** into a **scene graph**
- more precisely, a directed acyclic graph = DAG
- such a graph can be traversed just like a tree



scene tree



scene graph

Scene Graph

- universal data structure to describe scenes
  → hierarchical modeling
- to render such a scene graph, we have to
  - traverse graph depth first
  - remember current attributes
  - accumulate transformations
  - rasterize leaf nodes with these attributes and transformations

- We will come back to scene graphs later on

Next lectures …

- Rasterization of lines and Polygons