Visual Computing I:

Interactive Computer Graphics and Vision

2D Transforms

Stefanie Zollmann

Quiz

Detect the red balloons in this photo.

Which color model is best to work in?

Why?

Original Image

Red Channel

Greyscale

Hue from HSV

2D Geometry

Points, Lines, etc.

- A point is a 2D location $(u,\ v)$
- Two points define a line

$$\left(u_0,\ v_0\right),\ (u_1, v_1)$$

- A polyline with k segments is a sequence of $k + 1$ points

$$\left(u_0, v_0\right),\ \left(u_1, v_1\right),\ \ldots,\ \left(u_k,\ v_k\right)$$

- A polygon is a polyline where

$$\left(u_0, v_0\right) = \left(u_k,\ v_k\right)$$

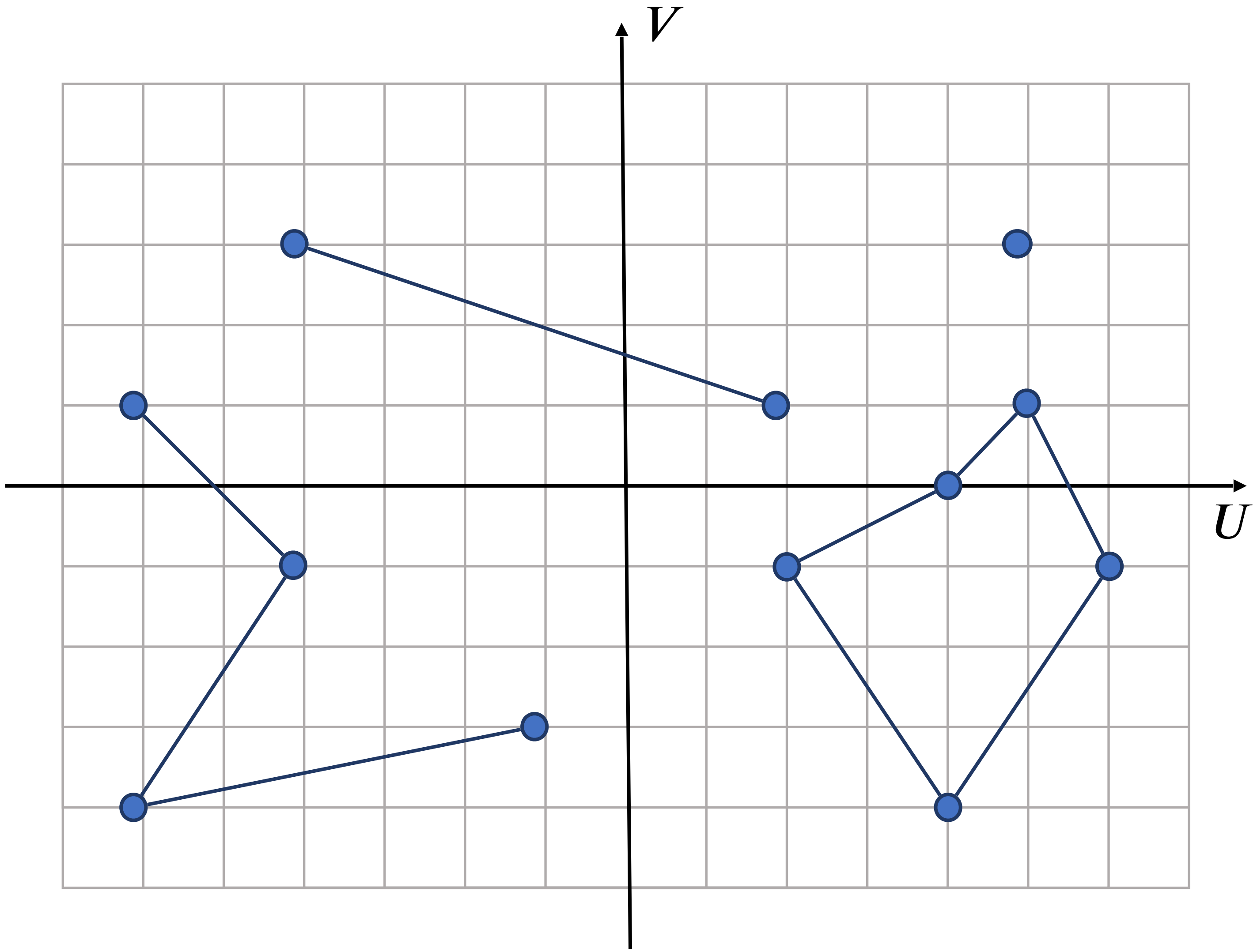
- We often omit the duplicate point

Points, Lines, etc.

- A point, $\left(5, 3\right)$
- A line, $(-4, 3),\ (2, 1)$
- A polyline,

  $\left(-6, 1\right),\ (-4,\ -1),$
  $\left(-6,\ -4\right),\ (-1,\ -3)$
- A polygon,

  $(4, 0),\ \left(5, 1\right),\ \left(6,\ -1\right),$
  $(4,\ -4),\ (2,\ -1)$

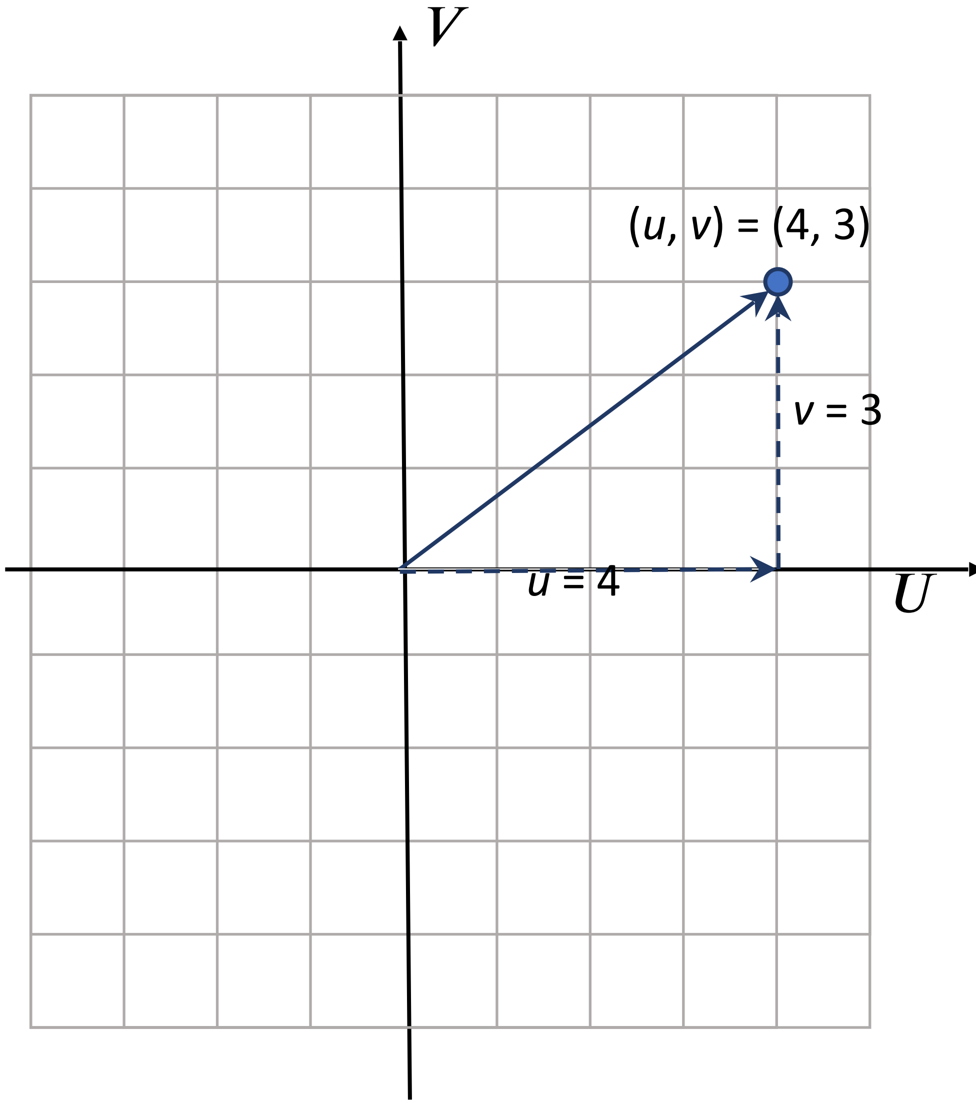


Points as Vectors

- We represent points as vectors

$$(u,\ v) \rightarrow \begin{bmatrix} u \\ v \end{bmatrix} = [u \quad v]^T$$

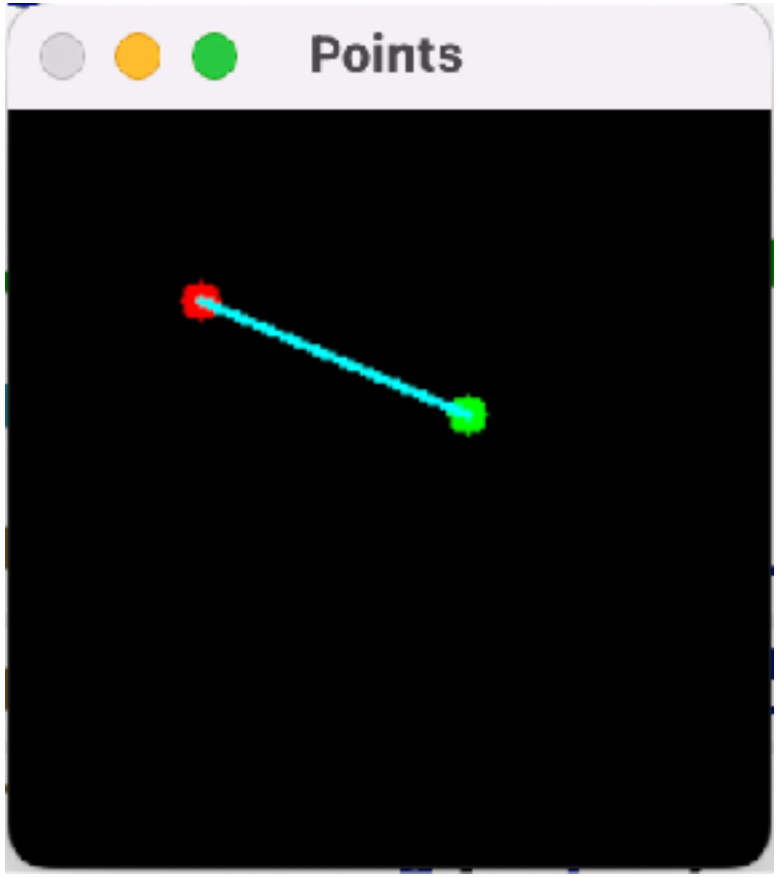
- We can think of this vector as:
    - The point $(u,\ v)$
    - A direction, or step of u units along one axis and v units along the other
    - The point is where you end up if you step along the vector from the origin



Example: 2D Points in OpenCV

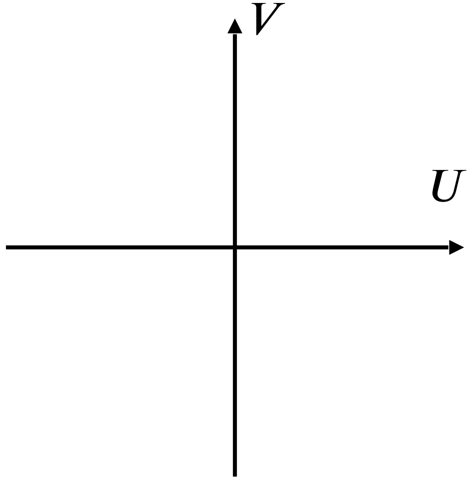
- In OpenCV, cv::Point (or cv::Point2f for floats) represents a 2D point

```
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace std;

int main() {
    // Create some points
    cv::Point p1(50, 50);           // integer coordinates
    cv::Point2f p2(120.5f, 80.3f); // float coordinates

    // Print points
    cout << "p1 = " << p1 << endl;
    cout << "p2 = " << p2 << endl;

    // Visualize on a blank image
    cv::Mat img = cv::Mat::zeros(200, 200, CV_8UC3);
    circle(img, p1, 5, cv::Scalar(0, 0, 255), cv::FILLED);  // red dot
    circle(img, p2, 5, cv::Scalar(0, 255, 0), cv::FILLED);  // green dot
    line(img, p1, p2, cv::Scalar(255, 255, 0), 2);      // yellow line

    cv::imshow("Points", img);
    cv::waitKey(0);
    return 0;
}
```



Choice of Coordinate Systems

Mathematical



- U: left -> right
- V: bottom -> top

Image-based

- U: left -> right
- V: top -> bottom

Matrix-based

C

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

R

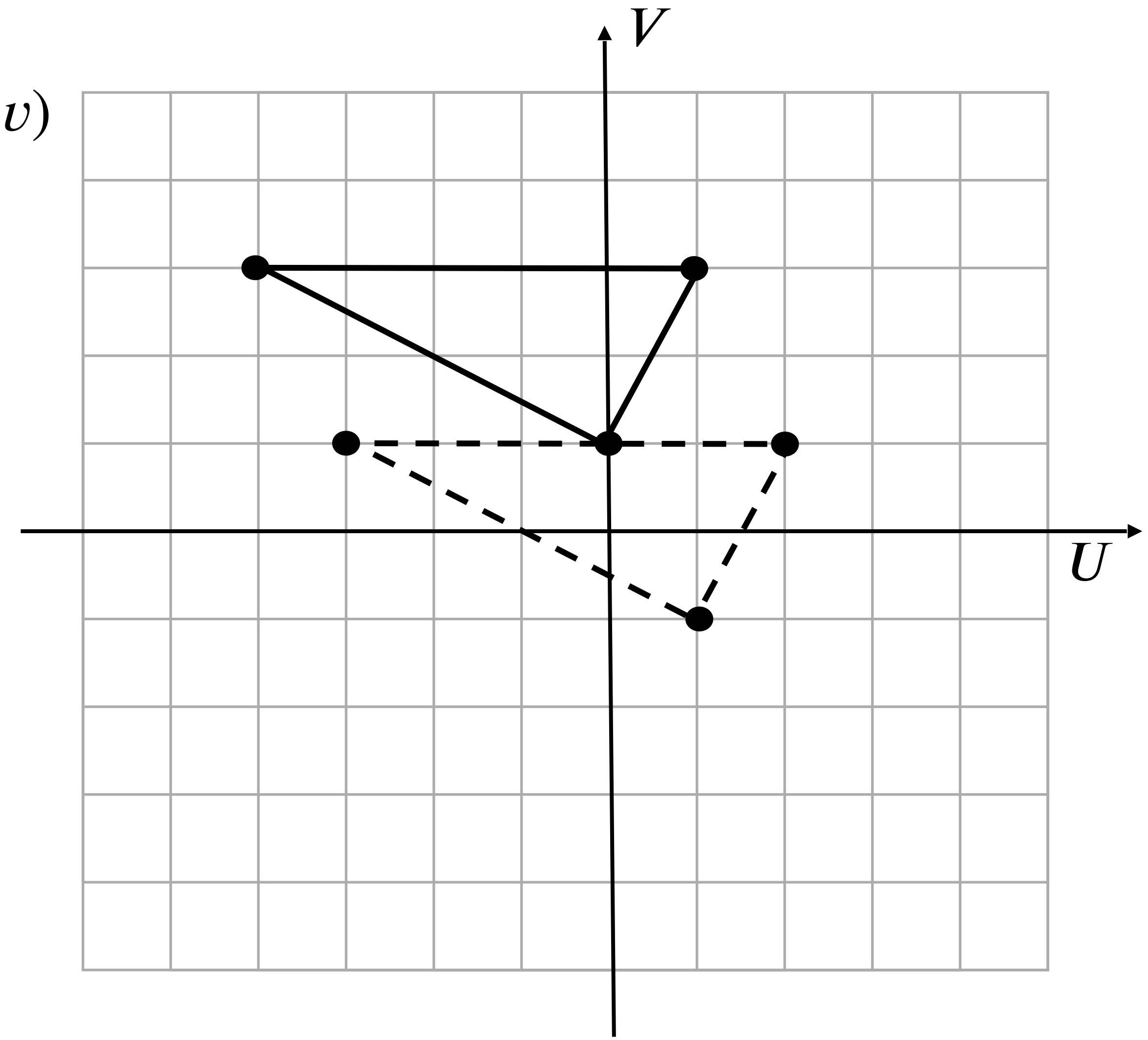
- R: top -> bottom
- C: left -> right

Transformations - Translation

- Shifting a point, $(u,\ v)$, by some offset, $(\Delta u,\ \Delta v)$
  $(u,\ v) \rightarrow (u + \Delta u,\ v + \Delta v)$

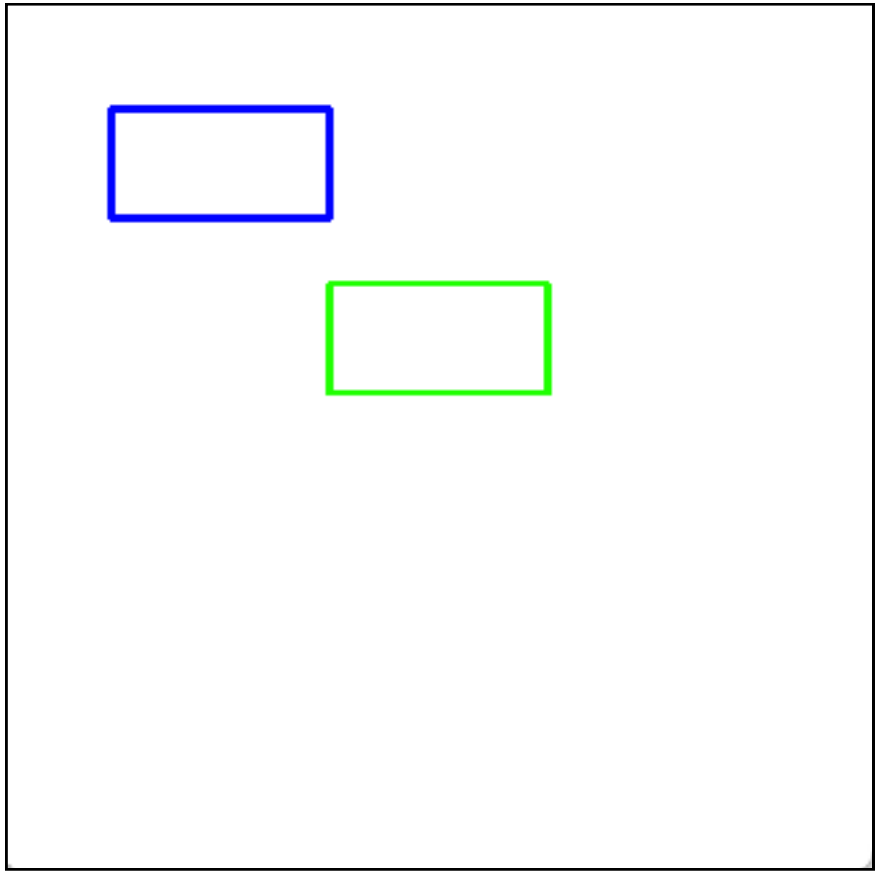
- In vector form:
  $$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix}$$

- Example:
  $$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} -1 \\ 2 \end{bmatrix}$$



Transformations - Translation

```
cv::Mat img2(400, 400, CV_8UC3, cv::Scalar(255, 255, 255)); // white BGR

// Original rectangle points
std::vector<cv::Point> pts = { {50,50}, {150,50}, {150,100}, {50,100} };

// Draw original rectangle (blue)
 cv::polylines(img2, std::vector<std::vector<cv::Point>>{pts}, true,
 cv::Scalar(255,0,0), 2);
cv::imshow("2DTransform", img2);
cv::waitKey(0);

// Translation vector
int tx = 100, ty = 80;

// Apply translation
std::vector<cv::Point> translated_pts;
for (auto &p : pts) {
    translated_pts.push_back(cv::Point(p.x + tx, p.y + ty));
}

// Draw translated rectangle (green)
 cv::polylines(img2, std::vector<std::vector<cv::Point>>{translated_pts},
 true, cv::Scalar(0,255,0), 2);

cv::imshow("2DTransform", img2);
cv::waitKey(0);
```

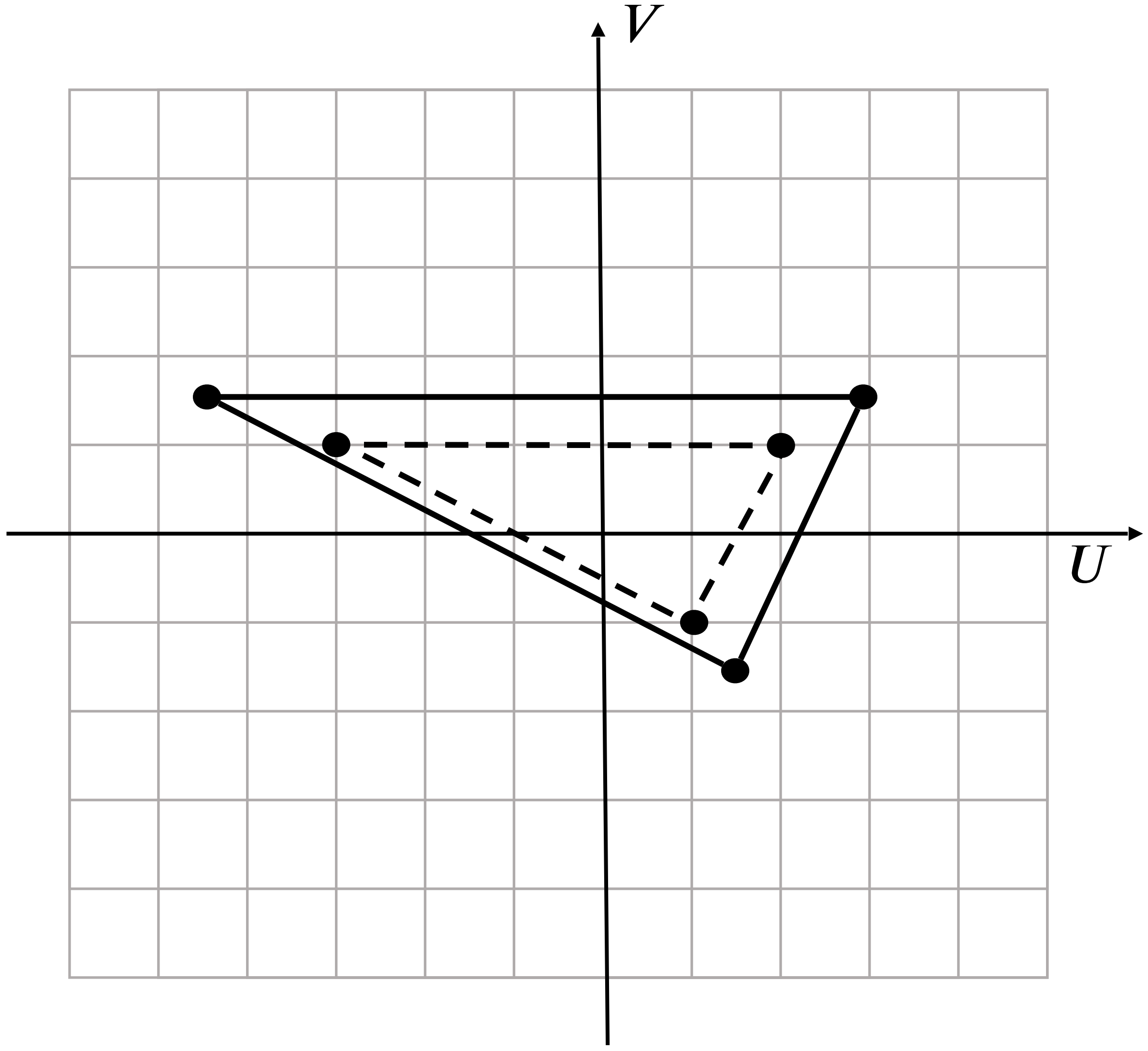


Transformations - Scaling

- Scaling points by factor, s
  $(u, v) \rightarrow (su,\ sv)$

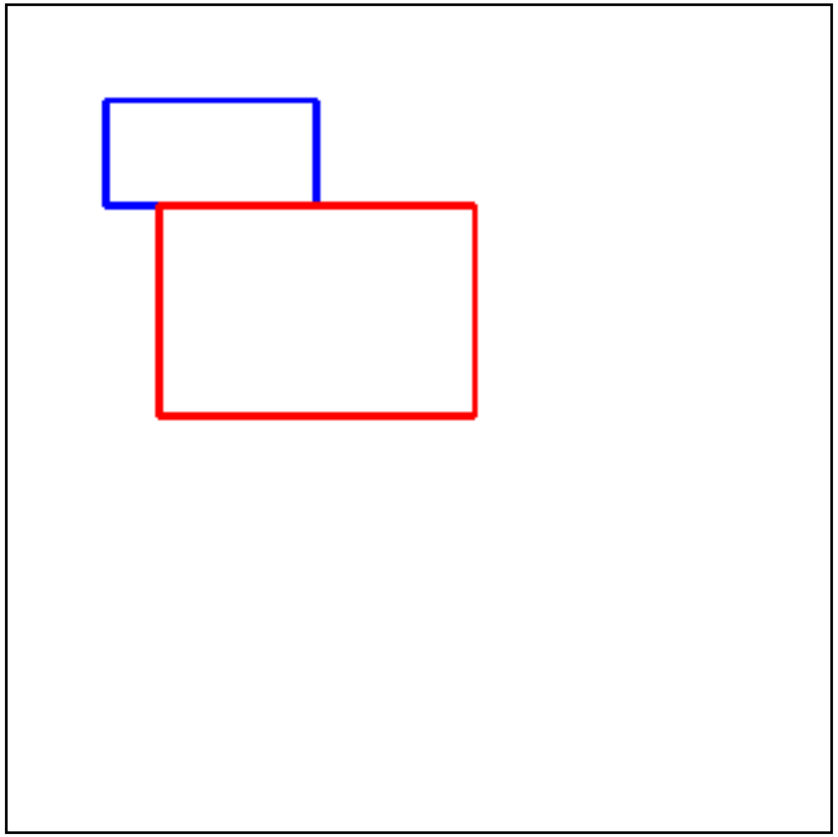
- In vector terms
  $$\begin{bmatrix} u' \\ v' \end{bmatrix} = s \begin{bmatrix} u \\ v \end{bmatrix}$$

- Example:
  $$\begin{bmatrix} u' \\ v' \end{bmatrix} = 1.5 \begin{bmatrix} u \\ v \end{bmatrix}$$



Transformations - Scaling

```
cv::Mat canvas(400, 400, CV_8UC3, cv::Scalar(255, 255, 255)); // white BGR

std::vector<cv::Point2f> pts = { {50,50}, {150,50}, {150,100}, {50,100} };

// Draw original rectangle (blue)
std::vector<cv::Point> pts_int;
 for(auto &p: pts) pts_int.push_back(cv::Point(cvRound(p.x),
 cvRound(p.y)));
cv::polylines(canvas, pts_int, true, cv::Scalar(255,0,0), 2);
cv::imshow("Scale", canvas);
cv::waitKey(0);
float sx = 1.5f, sy = 2.0f;

std::vector<cv::Point> scaled_pts;
for(auto &p: pts) {
    scaled_pts.push_back(cv::Point(cvRound(p.x * sx), cvRound(p.y * sy)));
}

cv::polylines(canvas, scaled_pts, true, cv::Scalar(0,0,255), 2);
cv::imshow("Scale", canvas);
cv::waitKey(0);
```



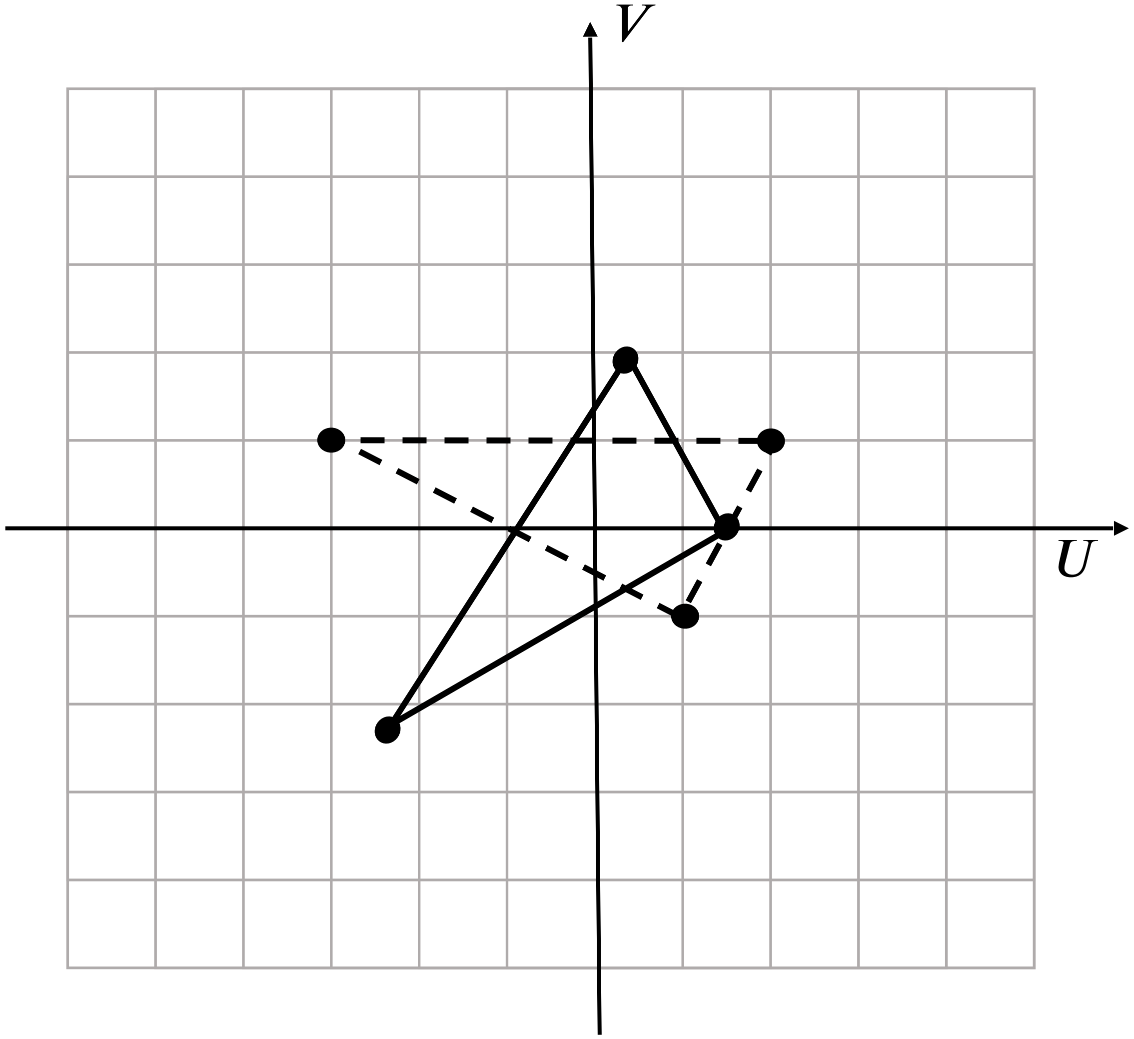
Transformations - Rotation

- Rotate by some angle, θ, about the origin

- Rotation from the U axis towards the V axis
    - NOT clockwise/anticlockwise

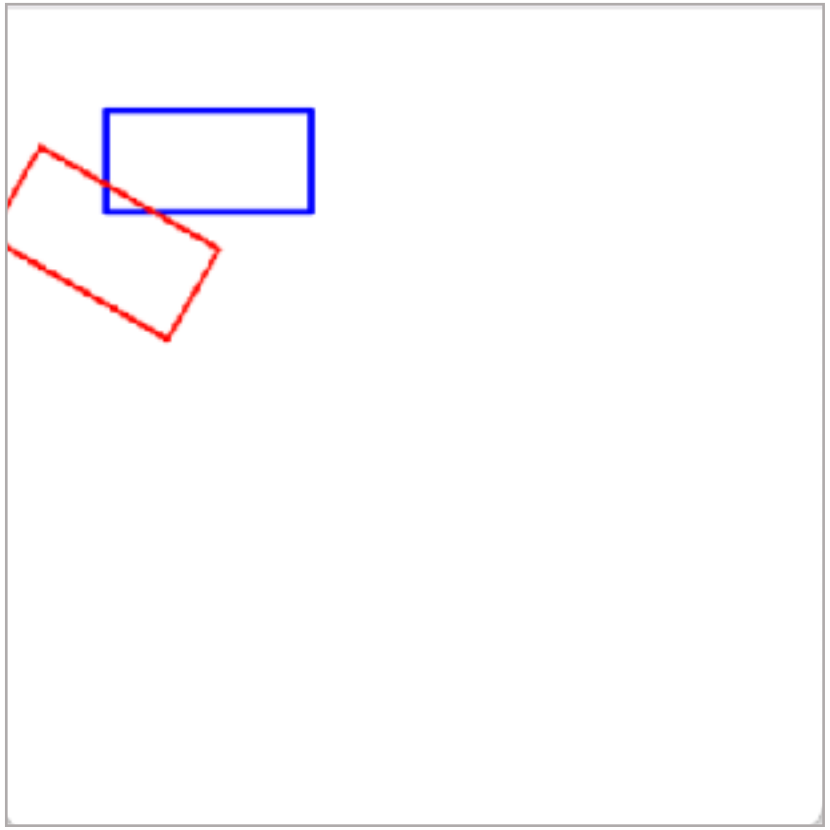
- In vector terms

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$



Transformations - Rotation

```
std::vector<cv::Point2f> pts = { {50,50}, {150,50}, {150,100},
{50,100} };

// Draw original rectangle (blue)
std::vector<cv::Point> pts_int;
for(auto &p: pts) pts_int.push_back(cv::Point(cvRound(p.x),
cvRound(p.y)));
cv::polylines(canvas, pts_int, true, cv::Scalar(255,0,0), 2);

// Rotation angle
double theta = CV_PI / 6; // 30 degrees

std::vector<cv::Point> rotated_pts;
for(auto &p: pts) {
    float x_new = p.x * cos(theta) - p.y * sin(theta);
    float y_new = p.x * sin(theta) + p.y * cos(theta);
    rotated_pts.push_back(cv::Point(cvRound(x_new),
    cvRound(y_new)));
}

cv::polylines(canvas, rotated_pts, true, cv::Scalar(0,0,255), 2);
```



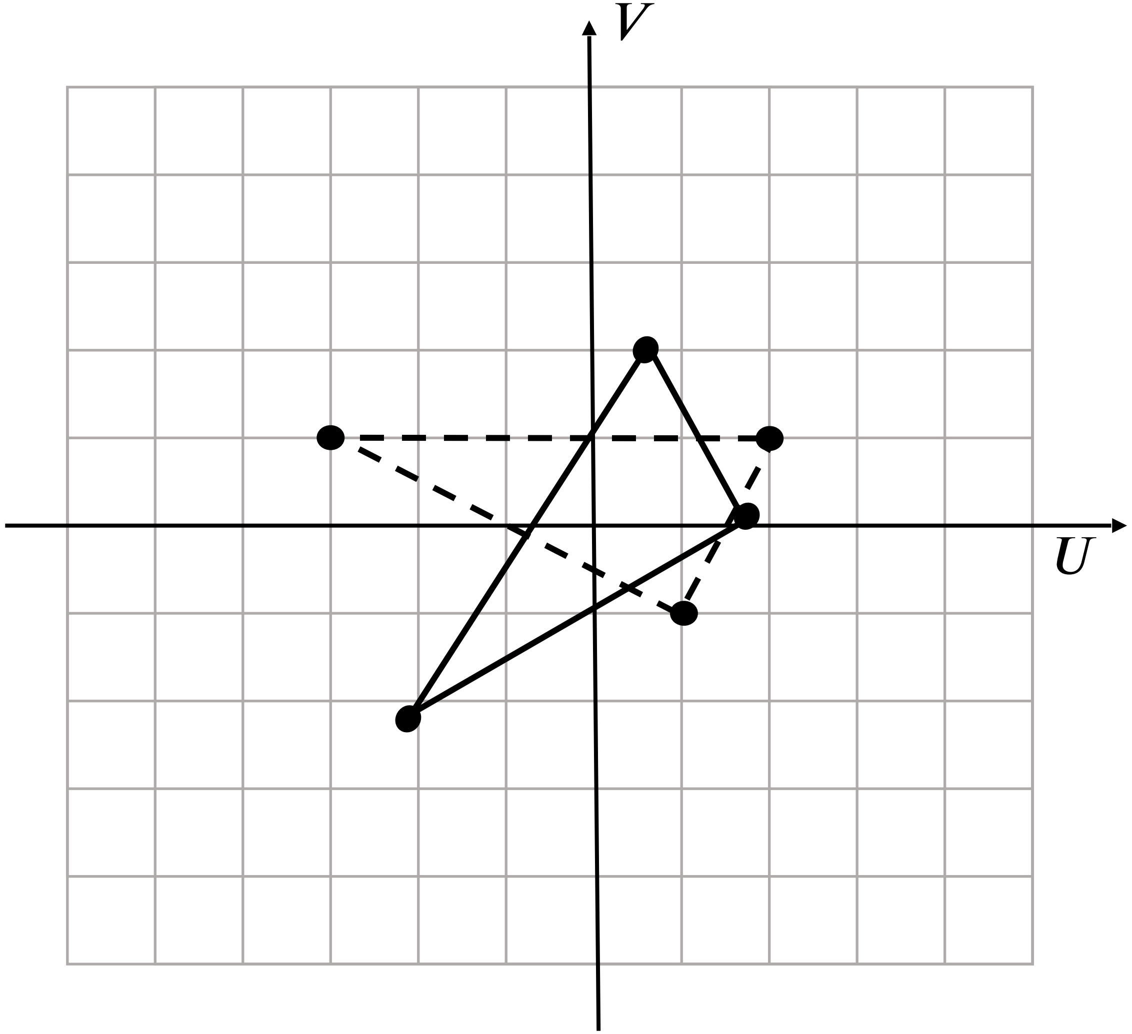
Transformations - Rotation

- Example rotation around 45 degrees

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$R\left(\frac{\pi}{4}\right) = \begin{bmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \cdot 2 + (-\frac{1}{\sqrt{2}}) \cdot 1 \\ \frac{1}{\sqrt{2}} \cdot 2 + \frac{1}{\sqrt{2}} \cdot 1 \end{bmatrix}$$



Transformations - Rotation

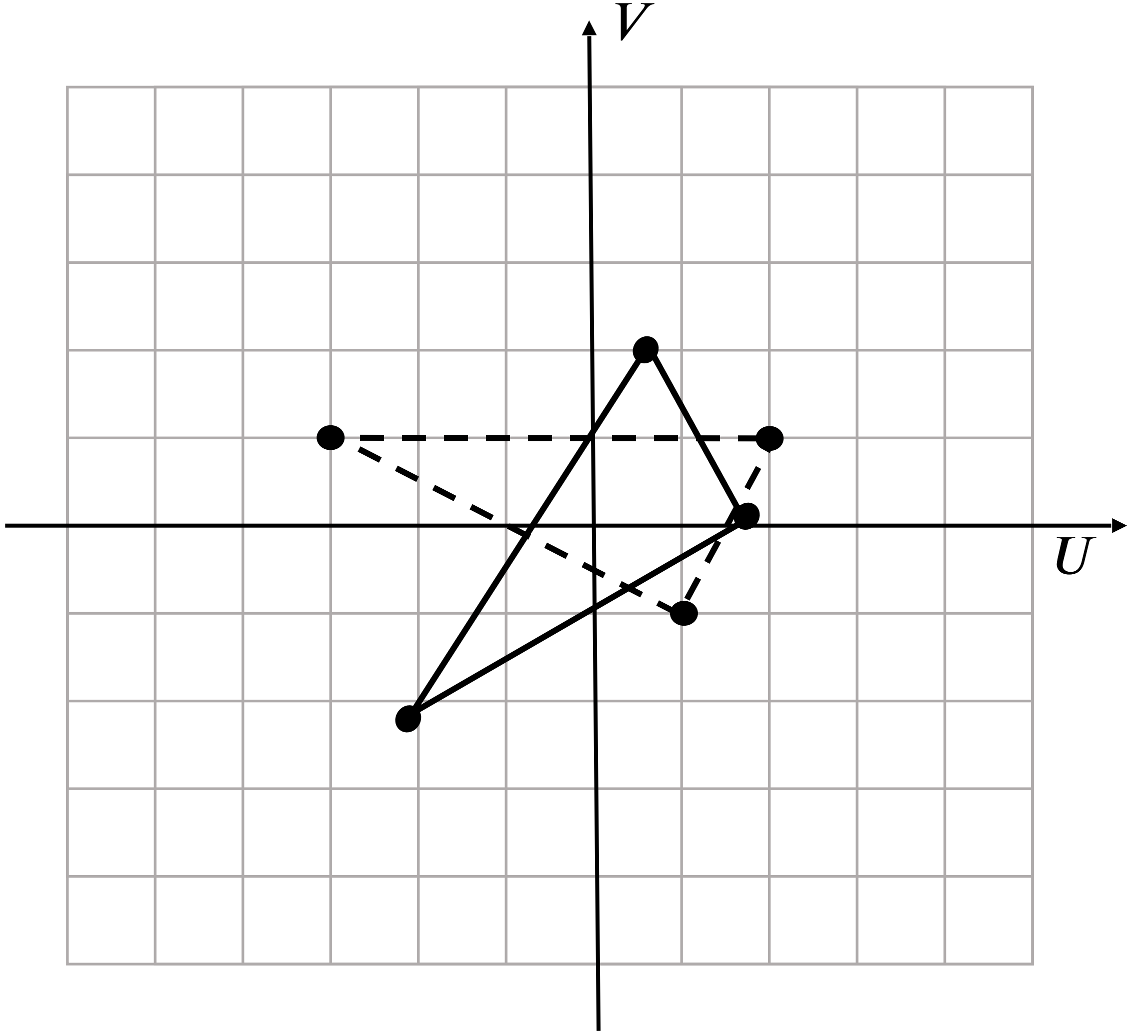
- Example rotation around 45 degrees

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$R\left(\frac{\pi}{4}\right) = \begin{bmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{3}{\sqrt{2}} \end{bmatrix}$$

$$\approx \begin{bmatrix} 0.7071 \\ 2.1213 \end{bmatrix}$$



Transformations - Rotation

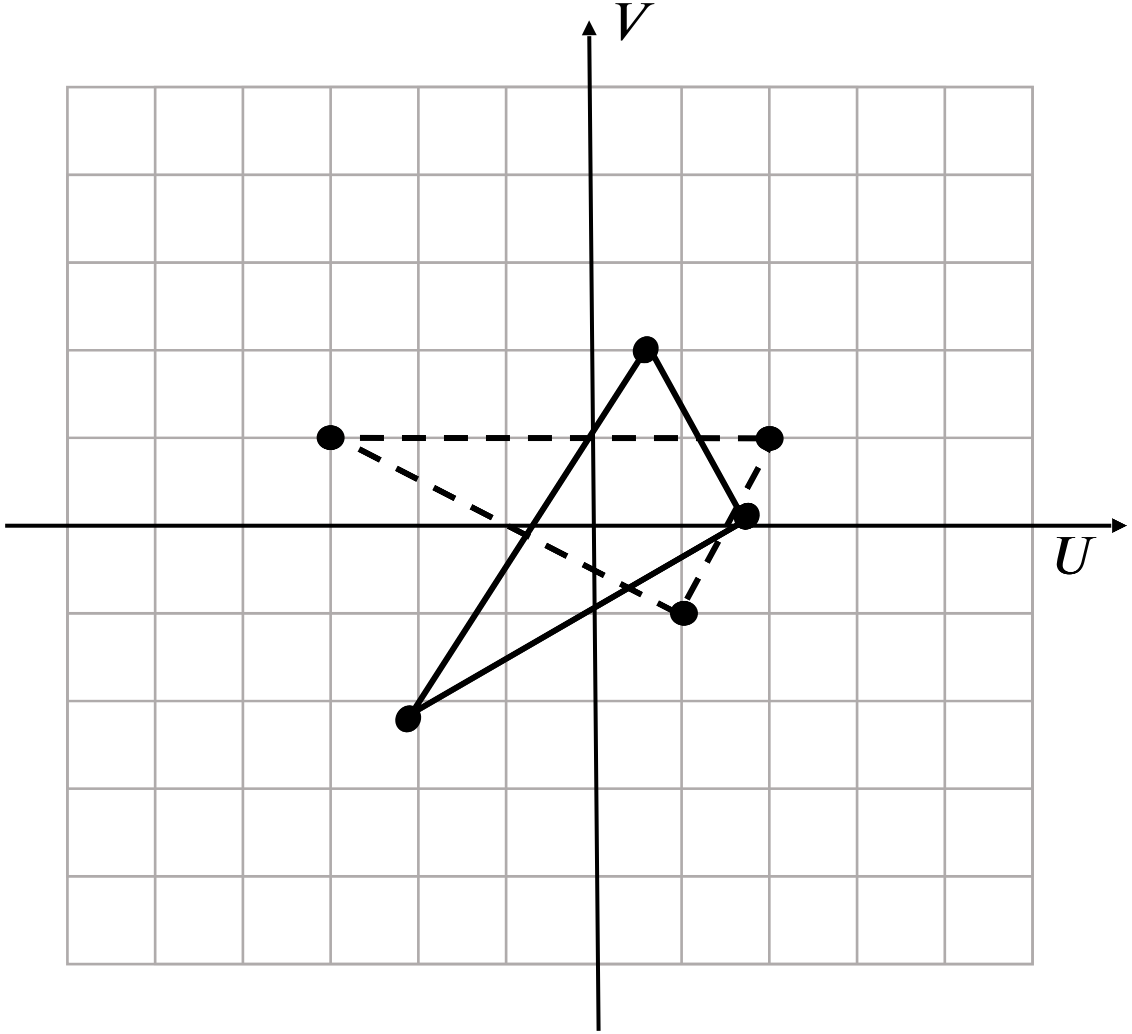
- Example rotation around 45 degrees

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$R\left(\frac{\pi}{4}\right) = \begin{bmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix} R\left(\frac{\pi}{4}\right) = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{3}{\sqrt{2}} \end{bmatrix}$$

$$\approx \begin{bmatrix} 0.7071 \\ 2.1213 \end{bmatrix}$$



Transformations - Inverse Transforms

- Transforms can be undone geometrically
  - Inverse of shifting by $(\Delta u, \Delta v)$ is shifting by $(-\Delta u, \ -\Delta v)$
  - Inverse of scaling by s is scaling by $\frac{1}{s}$
  - Inverse of rotating by θ is rotating by $-\theta$

- Inverse of a rotation matrix is its transpose

$$R_{-\theta} = \begin{bmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} = R_\theta^T$$

  - This is a special property of rotations – not of matrices in general

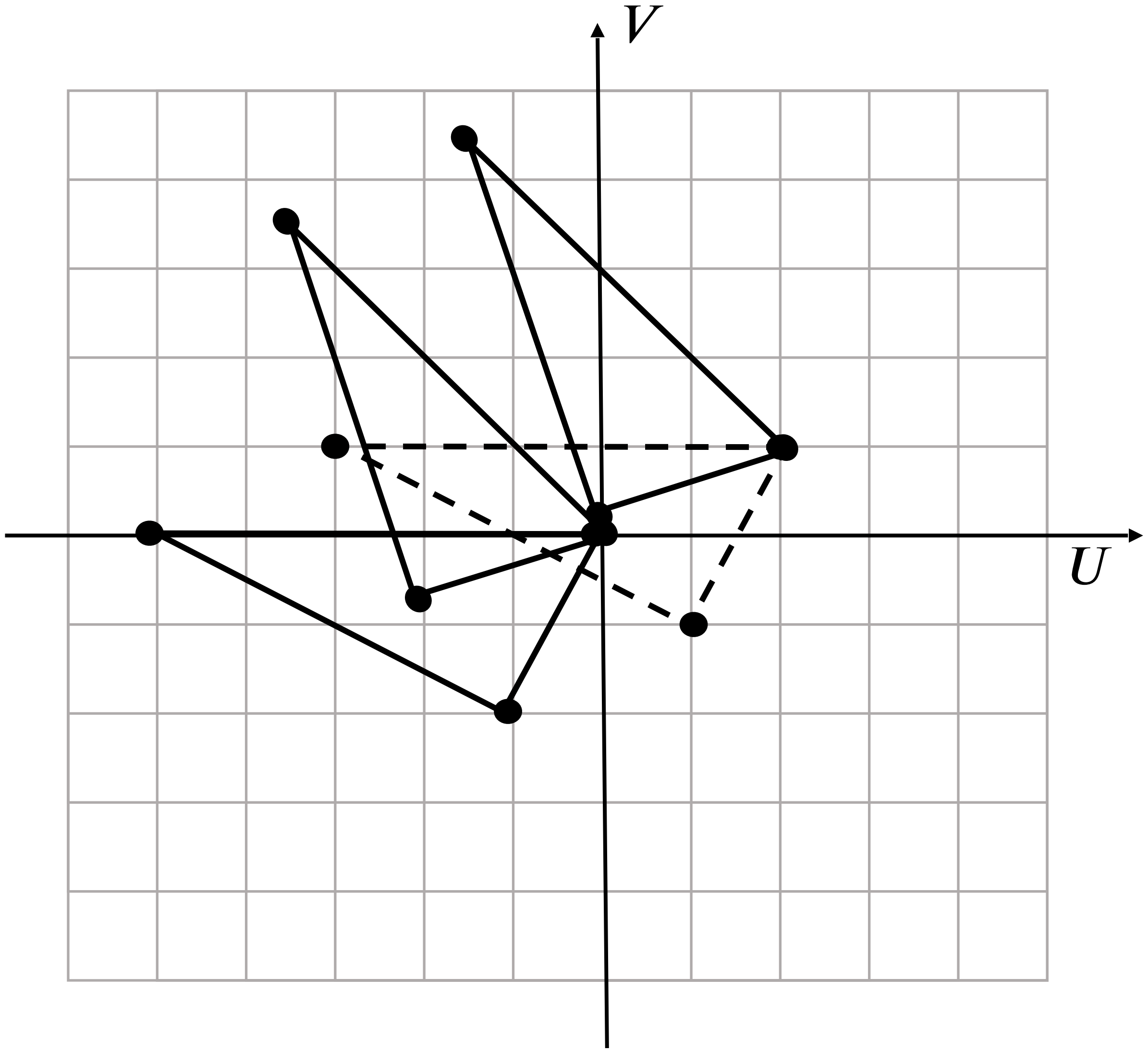
Transformations - Combining Transforms

- Rotate $-45°$ about $(2,1)$
  1. Shift by $(-2, -1)$
  2. Rotate by $-45°$
  3. Shift by $(2, 1)$

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} -2 \\ -1 \end{bmatrix}$$

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = R_{-45°}\left(\begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} -2 \\ -1 \end{bmatrix} \right)$$

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = R_{-45°}\left(\begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} -2 \\ -1 \end{bmatrix} \right) + \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$



Easier Way?

Homogenous Coordinates

- The solution is counter-intuitive

- 2D points become sets of 3-vectors

$$\begin{bmatrix} u \\ v \end{bmatrix} \rightarrow k \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, \quad \forall k \neq 0$$

- The vector $[a \quad b \quad c]^T$ corresponds to the point $\left(\frac{a}{c}, \frac{b}{c} \right)$

- These are homogeneous coordinates

- All (linear) transformations become 3×3 matrices

Homogenous Coordinates

https://szollmann.github.io/LectureExamples/tutorials/transformations/Transformations2DTutorial

Homogenous Transforms

- Translation by $(\Delta u,\ \Delta v)$ becomes

$$\begin{bmatrix} 1 & 0 & \Delta u \\ 0 & 1 & \Delta v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} u + \Delta u \\ v + \Delta v \\ 1 \end{bmatrix}$$

- Rotation by an angle θ becomes

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \cos(\theta)u - \sin(\theta)v \\ \sin(\theta) + \cos(\theta)v \\ 1 \end{bmatrix}$$

Homogenous Transforms

- Scaling by a factor of s becomes

$$\begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} su \\ sv \\ 1 \end{bmatrix} \equiv \begin{bmatrix} u \\ v \\ \frac{1}{s} \end{bmatrix} \equiv \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \frac{1}{s} \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

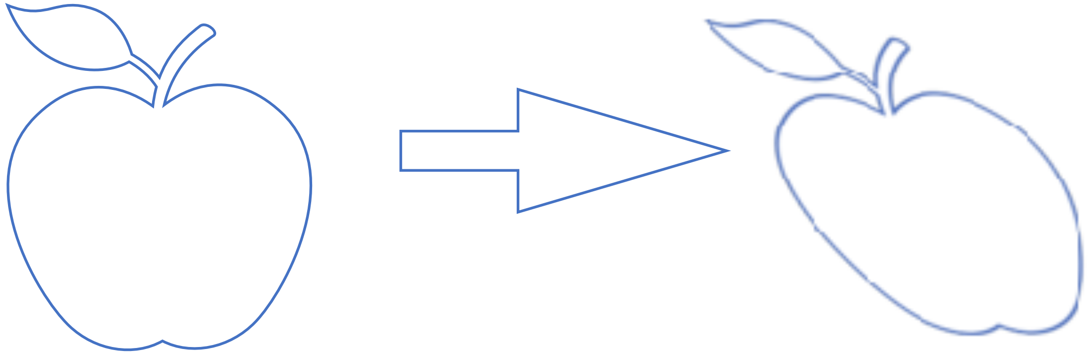
- Can also do non-uniform scaling:

$$\begin{bmatrix} s_u & 0 & 0 \\ 0 & s_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} s_u u \\ s_v v \\ 1 \end{bmatrix}$$

Homogenous Transforms - Shearing

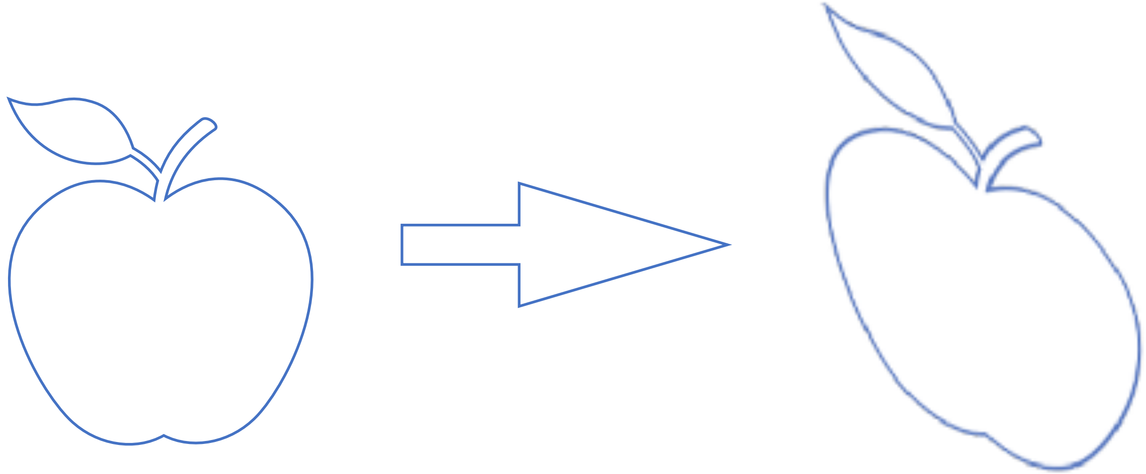
- Distorts the shape by shifting one axis relative to the other
- Example: Turning a rectangle into a parallelogram
- Shearing by a factor of sh_x in x direction:

$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



- Shearing by a factor of sh_y in your direction:

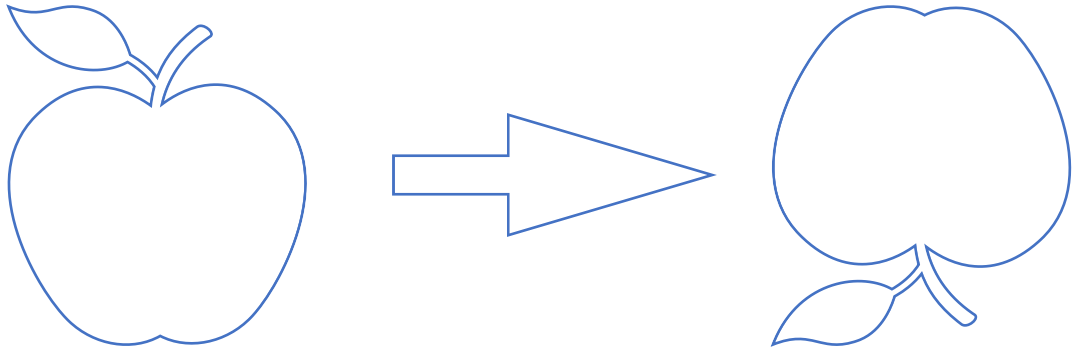
$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Homogenous Transforms - Reflection

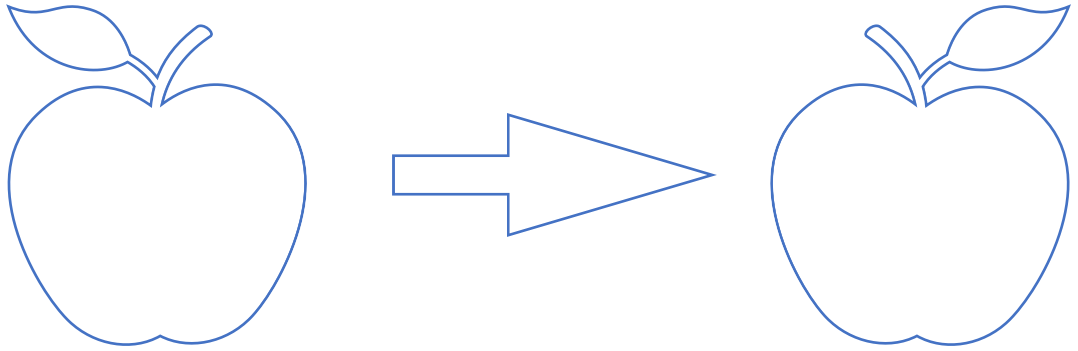
- Flips points across a line (e.g., the x-axis, y-axis, or an arbitrary line)
- Reflection across the x-axis:

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



- Reflection across the y-axis:

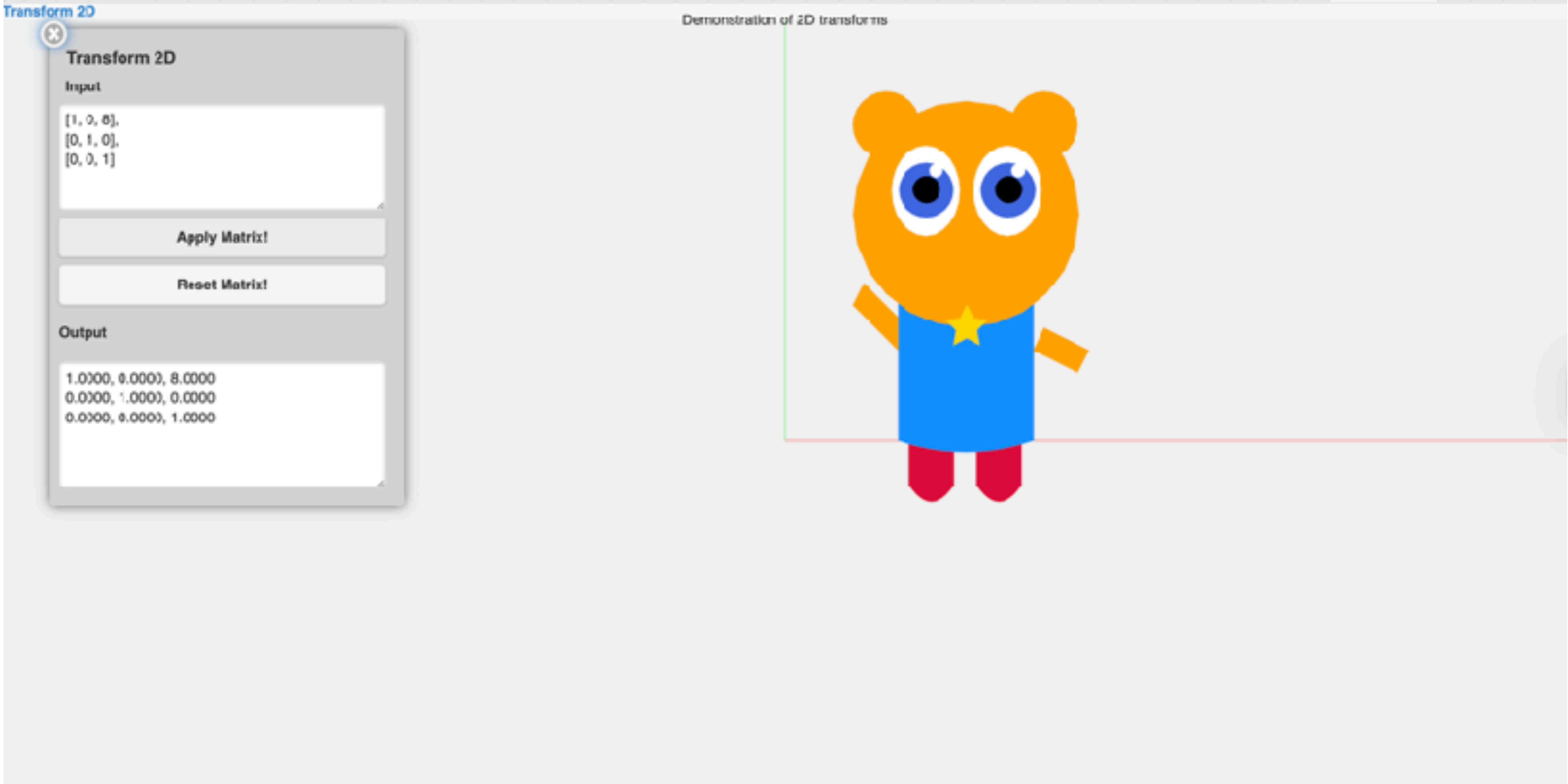
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Combining Transforms

- Because all transforms are matrices they combine easily
- Suppose we have a series of transforms, $T_1, T_2, \ldots, \ T_k$
- Applying them to a point, $\mathbf{p}$, in order gives
  $\mathbf{p}' = T_k\left(T_{k-1}\left(\ldots\left(T_2\left(T_1\mathbf{p}\right)\right)\right)\right)$
- Because matrix multiplication is associative, we have
  $\mathbf{p}' = \left(T_kT_{k-1}\ldots T_2T_1\right)\mathbf{p}$
- Can combine transforms once and apply to a set of points

Try yourself



https://szollmann.github.io/LectureExamples/tutorials/transformations/Transformations2DTutorial

Additional Examples Combining Transforms

Inverse Transforms

- Inverse of shifting by $(\Delta u, \Delta v)$ is shifting by $(-\Delta u, \ - \Delta v)$

$$\begin{bmatrix} 1 & 0 & \Delta u \\ 0 & 1 & \Delta v \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & -\Delta u \\ 0 & 1 & -\Delta v \\ 0 & 0 & 1 \end{bmatrix}$$

- Inverse of scaling by s is scaling by $\frac{1}{s}$

$$\begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1/s & 0 & 0 \\ 0 & 1/s & 0 \\ 0 & 0 & 1 \end{bmatrix} \equiv \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & s \end{bmatrix}$$

Inverse Transforms

- Inverse of rotating by θ is rotating by $-\theta$

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

- If we have a sequence of transforms, $T_k T_{k-1} \ldots T_2 T_1$, then

$$\left(T_k T_{k-1} \ldots T_2 T_1\right)^{-1} = T_1^{-1} T_2^{-1} \ldots T_{k-1}^{-1} T_k^{-1}$$

Recap: What have we learned so far?

Transforms

- Translation
- Rotation
- Scaling
- Shear
- Reflection
- These can all be written as matrices in homogeneous coordinates
- Instead of thinking of each transform separately, we can compose them

Affine Transforms

- General form (homogeneous 3×3 matrix, bottom row fixed):

$$\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- Includes: translation, rotation, scaling, shear, reflection
- Properties:
    - Preserve straightness of lines
    - Preserve parallelism
    - Ratios of lengths along the same line are preserved
- Examples: skewing a photo, mapping a rectangle to a parallelogram

Limits of Affine Transforms

- Cannot capture perspective effects
- Example: a photo of a square table looks like a trapezoid
- In reality, parallel edges seem to meet at a vanishing point
- Affine transformations cannot model this

Projective Transforms

- To capture perspective, we generalize to Projective Transforms
- Affine = special case where: $h_{31} = 0, h_{32} = 0, h_{33} = 0$
- Properties:
  - Lines stay straight
  - Parallelism not preserved
  - Can map any quadrilateral to any other quadrilateral
  - Models perspective projection

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$$

$$H = \begin{bmatrix} 2 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.5 & 0.3 & 1 \end{bmatrix},$$

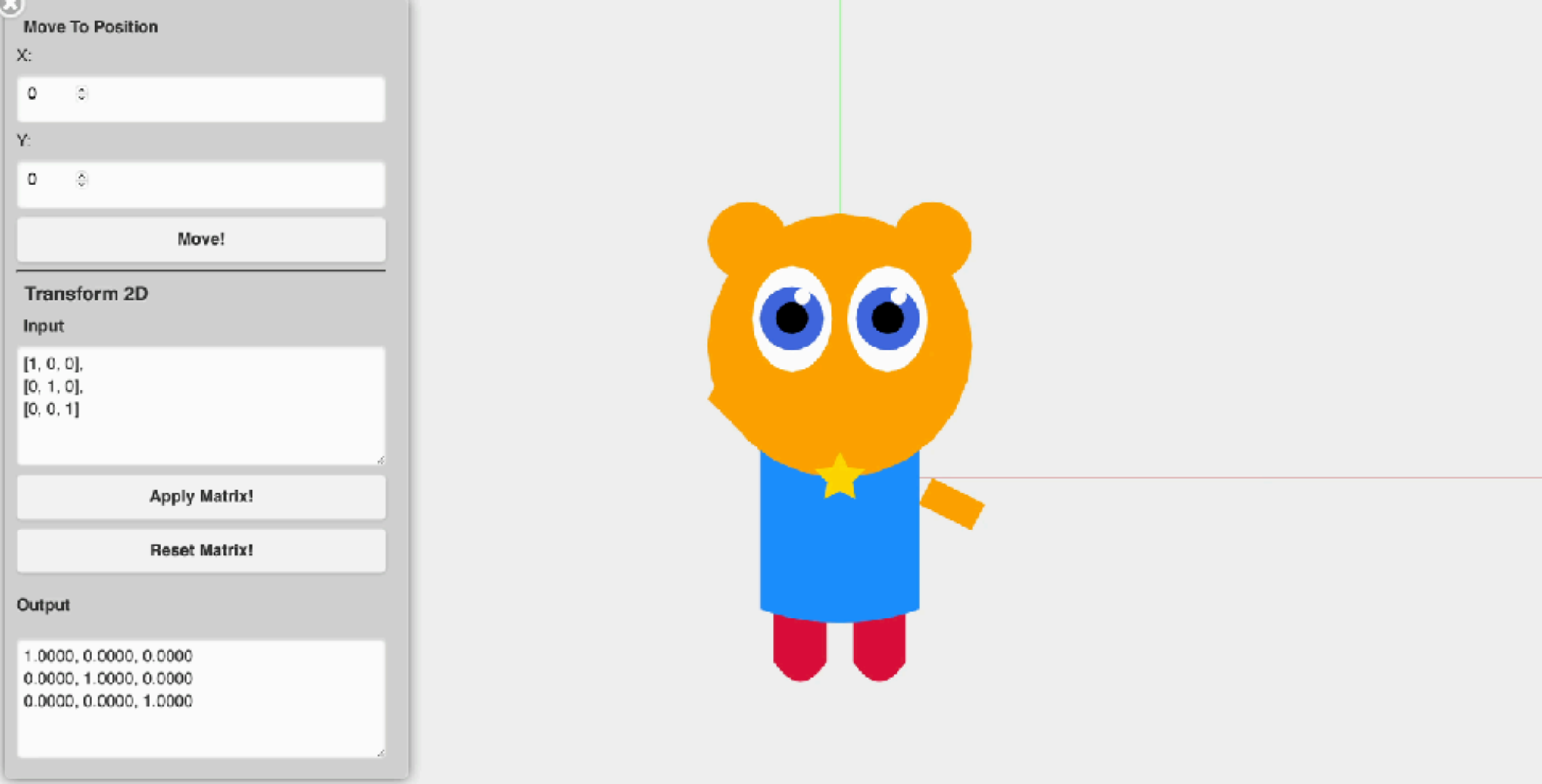
Applications

- Affine:
  - Geometric modeling
  - Simple graphics transforms
- Projective:
  - Image stitching
  - AR plane tracking
  - Perspective correction in images

2D Animations

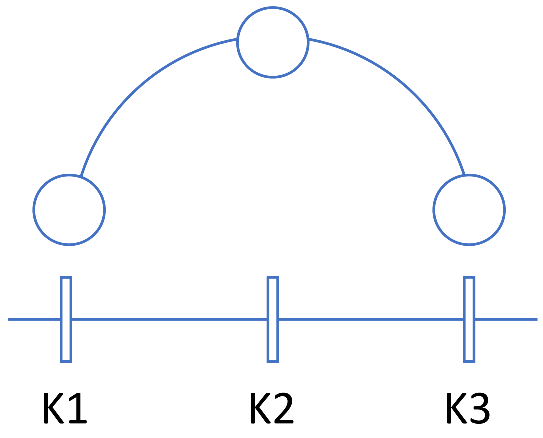
2D Animations

- Animation = applying transformations over time
- Common techniques:
  - Translation over time -> moving an object
  - Scaling over time -> growing/shrinking
  - Rotation over time -> spinning
- Achieved by updating transformation parameters frame by frame



https://szollmann.github.io/LectureExamples/tutorials/transformations/2DAnimations

Principles 2D Animations



- Keyframes: define positions/transform states at specific times
- Interpolation (tweening): smooth transitions between keyframes
- Combining transforms: e.g., rotate + translate for circular motion
- Applications:
  - Game character motion
  - UI transitions
  - Data visualization (dynamic charts)

Example 2D Animation

```
// Animation loop
    for (int i = 0; i < total_frames; i++) {
        double t = (double)i / (total_frames - 1);
        // goes 0 → 1
        int dx = (int)(t * shift_x);
        int dy = (int)(t * shift_y);

        // Create a black frame
        cv::Mat frame = cv::Mat::zeros(h, w, img.type());

        // Copy pixels manually with translation
        for (int y = 0; y < h; y++) {
            for (int x = 0; x < w; x++) {
                int nx = x + dx; // new x position
                int ny = y + dy; // new y position
                if (nx >= 0 && nx < w && ny >= 0 && ny < h) {
                    frame.at<cv::Vec3b>(ny, nx) = img.at<cv::Vec3b>(y, x);
                }
            }
        }
```

Example 2D Animation Composite

```
// Animation loop
for (int i = 0; i < total_frames; i++) {
    double t = (double)i / (total_frames - 1); // 0 → 1

    // Background
    cv::Mat frame = cv::Mat::zeros(height, width, CV_8UC3);

    // Circle parameters
    int x = (int)lerp(50, width - 50, t);                   // move left → right
    int y = height / 2 + (int)(50 * std::sin(t * 6.28 * 2)); // wiggle up/down
    int r = (int)lerp(20, 80, t);                           // radius grows

    cv::circle(frame, cv::Point(x, y), r, cv::Scalar(0, 255, 255), -1);

    // Show and save
    cv::imshow("Circle Animation", frame);
    if (cv::waitKey(30) == 27) break; // ESC to stop

    writer.write(frame);
}
```

Summary: 2D geometry manipulation

How about image content manipulation?

Next time :)

The end!