# Image Manipulation & Compression with Linear Algebra

tinyimg.amirghofran.com

### Image Transforms

Rotation -28°

Link Scale X/Y

Scale X 0.45x

Scale Y 0.50x

Shear X -0.49

Shear Y -0.40

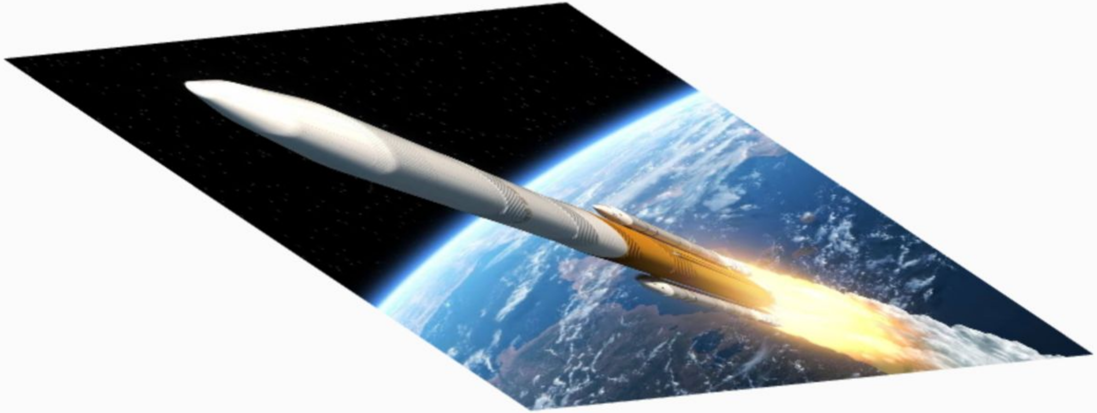
Translate X 0 px

Translate Y 0 px

Flip Horizontal (Off)

Flip Vertical (Off)

View on Github



### Processing & Export

Filters

Blur | Sharpen | Edge | Emboss

SVD Compression

Rank 50

Apply SVD

Transformed Area

**0.7236**

Formula: Area = 4 × |det(M)|

Transformation Matrices

Overall | Translate | Rotate | Scale | Shear

| | | | |
|---|---|---|---|
| 0.482 | -0.451 | 0.000 | 0.000 |
| 0.052 | 0.326 | 0.000 | 0.000 |
| 0.000 | 0.000 | 1.000 | 0.000 |
| 0.000 | 0.000 | 0.000 | 1.000 |

Download Image

# Overview

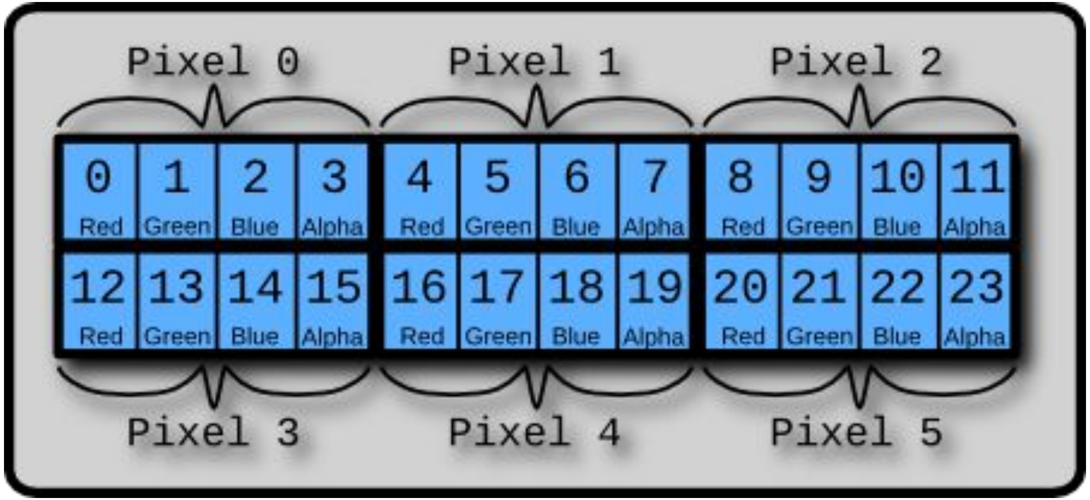
- Basic app for image manipulation
- Transformations: Rotation, Scale, Shear, Translate
- Filters: Blur, Sharpen, Edge Detection, Emboss
- Image Compression
- WebGL for image transformations
- Go logic for applying filters and image compression
- Compiled logic to WASM and loaded in the browser

# Matrix Representation of Images

When an image is uploaded each pixel is represented with 4 values following the RGBA color model

Instead of storing each color channel as four separate matrices we start with a Clamped Array, a flat single-dimensional array. This ensures:

- Memory efficiency
- Fast data transfer & compatibility

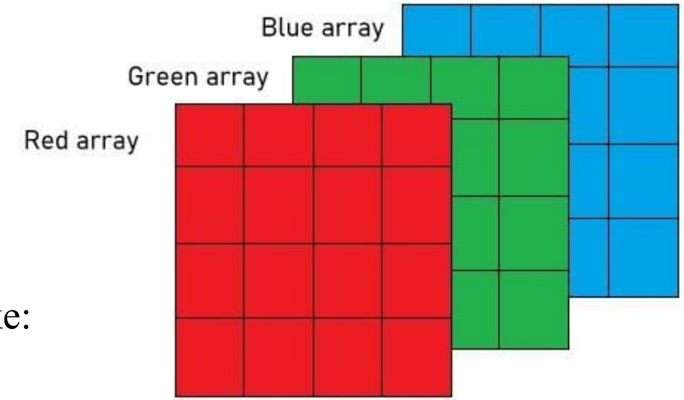


# Matrix Representation of Images: Processing

The backend splits the image into **4 color channels** — Red, Green, Blue, and Alpha — and turns each one into a **2D matrix**.

For an image that's **W×H** each channel becomes a matrix of size **H×W**, where each value represents the **intensity** of that channel at a specific pixel.

$$M_{\text{channel}} = \begin{pmatrix} p_{0,0} & p_{1,0} & \cdots & p_{W-1,0} \\ p_{0,1} & p_{1,1} & \cdots & p_{W-1,1} \\ \vdots & \vdots & \ddots & \vdots \\ p_{0,H-1} & p_{1,H-1} & \cdots & p_{W-1,H-1} \end{pmatrix}$$




Arrrays stacked over each other to form a Digital Image.

Once we have these matrices, we can apply **linear algebra techniques** like:

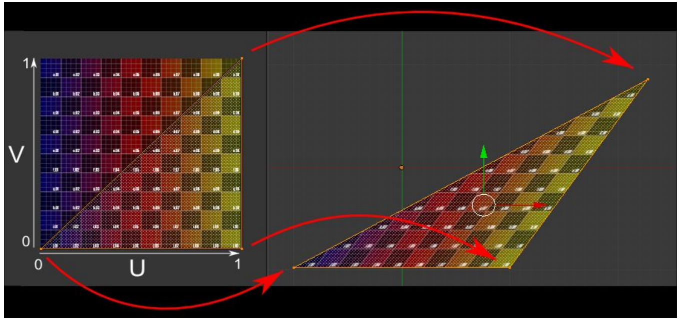
- **Convolution**
- **SVD**

# Matrix Representation of Images: Transformations

The image's **pixel data** (RGBA values) are **not changed**, only **how the image appears on screen** — its position, scale, rotation, and shape. To do so we follow the following steps:

1. A **quadrilateral** is drawn to fill the screen made up of **4 vertices**. These define **where** the image should be drawn.
2. WebGL requires the vertices to be in 4D (**homogeneous coordinates**). So we convert each 2D vertex into a 4D vector
3. These 2 extra coordinates are called **UV (texture) coordinates**, which map the image to its location in the **quad**.
4. Now the transformations can be applied to the two position coordinates, transforming the **quad** only, and the image is painted onto this.

$$\vec{v} = \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix}$$



# Linear Transformations

# Linear Transformation

A function that transforms vectors while keeping their structure.

- $T(\mathbf{u} + \mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$
- $T(k\mathbf{u}) = kT(\mathbf{u})$

# Matrix Form of Transformation

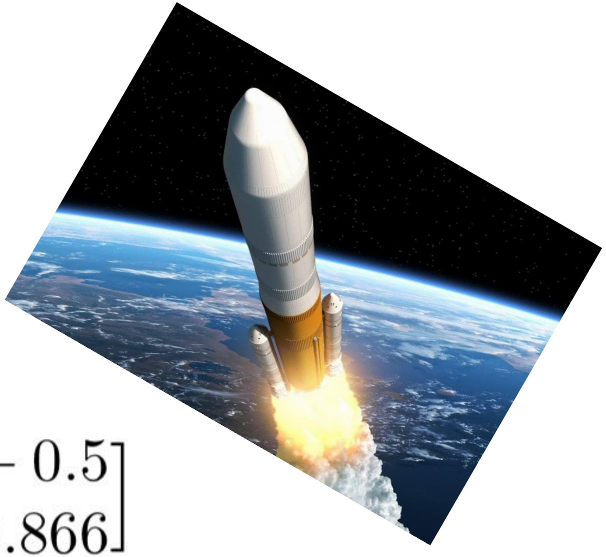
All linear transformations can be written as T(x) = Ax

$$A = \begin{bmatrix} 2 & -1 \\ 1 & 3 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} \quad \Rightarrow \quad T(\mathbf{x}) = \begin{bmatrix} 2x - y \\ x + 3y \end{bmatrix}$$

# Rotation

Rotation Matrix:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$



- Counterclockwise rotation by default
- Rotates every pixel of the image around the center

Equation for rotation:

$$\text{rotate}(30^\circ) = \begin{bmatrix} \cos(30^\circ) & -\sin(30^\circ) \\ \sin(30^\circ) & \cos(30^\circ) \end{bmatrix} \begin{bmatrix} 0.866 & -0.5 \\ 0.5 & 0.866 \end{bmatrix}$$

# Scaling

Scaling Matrix:

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

- Scales horizontally by $s_x$ and vertically by $s_y$
- Zoom in/out

Equation for scaling:

$$\text{scale}(2, 0.5) = \begin{bmatrix} 2 & 0 \\ 0 & 0.5 \end{bmatrix}$$

# Shearing

Shearing Matrix:

Horizontal: $$\begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$$

Vertical: $$\begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}$$

- Distorts the shape diagonally
- K controls how slanted the shear is

Equation for shearing:

$$\text{shear}_x(1.5) = \begin{bmatrix} 1 & 1.5 \\ 0 & 1 \end{bmatrix}$$

# Combining Transformations = Matrix Multiplication

$$T = R \cdot S = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} 0 & -0.5 \\ 2 & 0 \end{bmatrix}$$

# Calculating Area with Determinant

# Determinant

- In the project, we apply transformations like rotation, scaling, and shearing using 4x4 matrices.
- These transformations affect how the image is displayed on screen.
- The determinant of the matrix plays a key role in understanding how the image's area changes.

# Determinants

- Determinants help us understand how a transformation affects area.

For a 2D matrix:

$$\det \begin{pmatrix} a & c \\ b & d \end{pmatrix} = ad - bc$$

- det = ad - bc
- The absolute value of the determinant tells us the area scaling factor.
- Possible values:
  - |det| = 1 : Area unchanged (ex. rotation)
  - |det| > 1 : Area increases (ex. scaling up)
  - 0 < |det| < 1 : Area decreases (ex. slight compression)
  - |det| = 0 : Area collapses to a line or point (degenerate transformation)

# How is works:

In TinyIMG, we use 4x4 matrices, but only the top-left 2x2 submatrix affects area:

$$\det\begin{pmatrix} a & c & 0 & \mathrm{tx} \\ b & d & 0 & \mathrm{ty} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = ad - bc$$

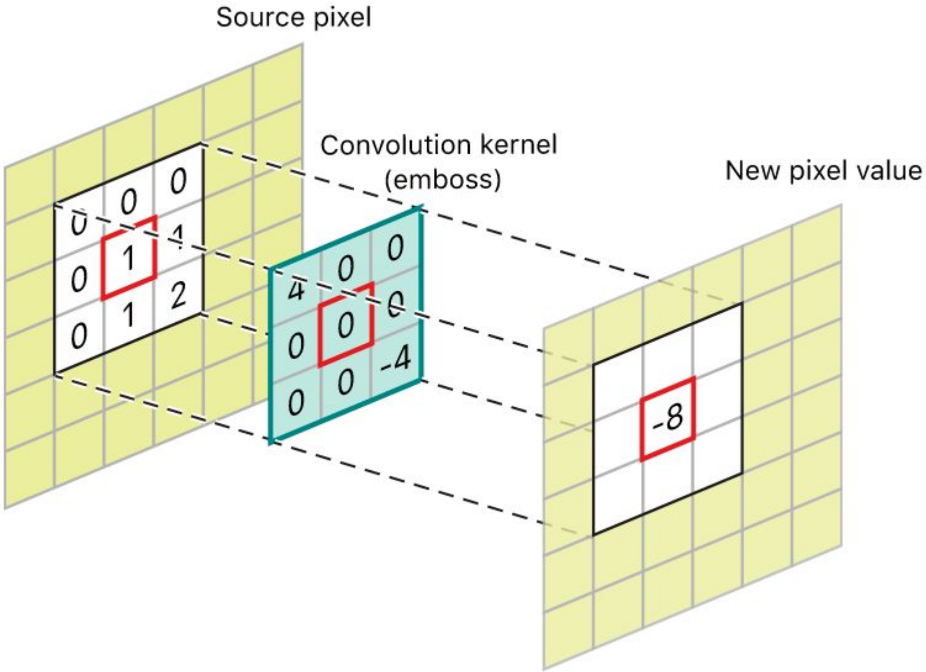
Area scaling factor = |ad - bc|

This value tells us how much larger or smaller the transformed image appears.

# Filters with Kernel Convolutions

# Convolution Basics

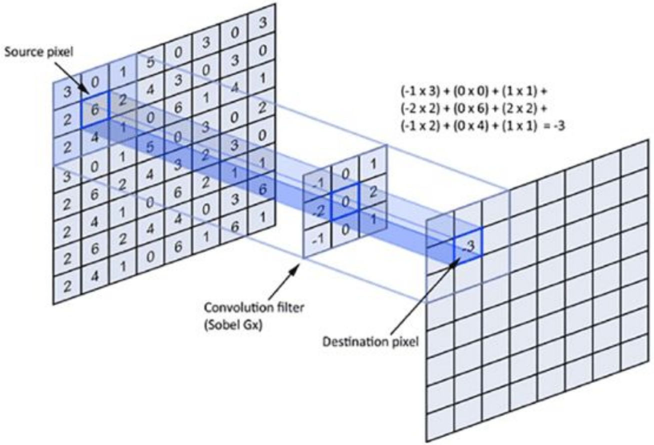
Convolution is a fundamental technique where a filter (or kernel) is applied to an image to modify its pixel values based on a weighted average of their neighbours.



# Convolution Calculation

The convolution operation for a pixel at (x, y) and a channel c is mathematically defined as:

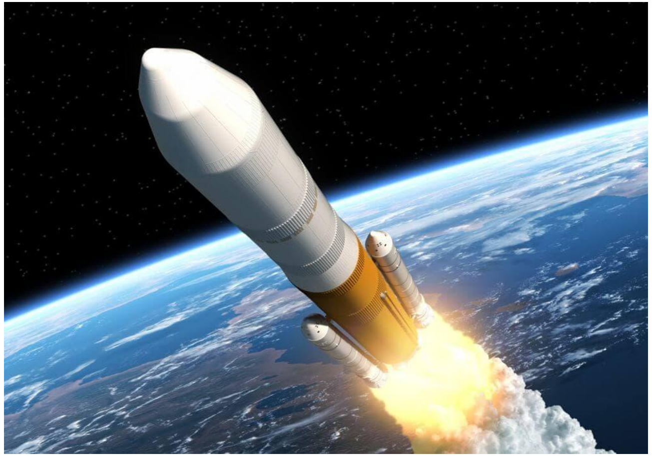
$$P'_{\text{out}}(x, y, c) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} K(i, j) \cdot P_{\text{in}}(x + i, y + j, c)$$



# Blur

Averages neighbor pixels.

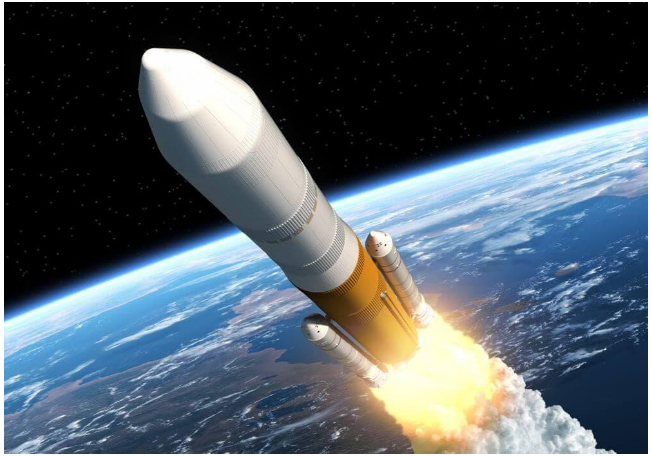
$$K_{\text{blur}} = \frac{1}{9}\begin{pmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{pmatrix}$$



# Sharpen

Enhances edges by emphasizing differences with neighbors.

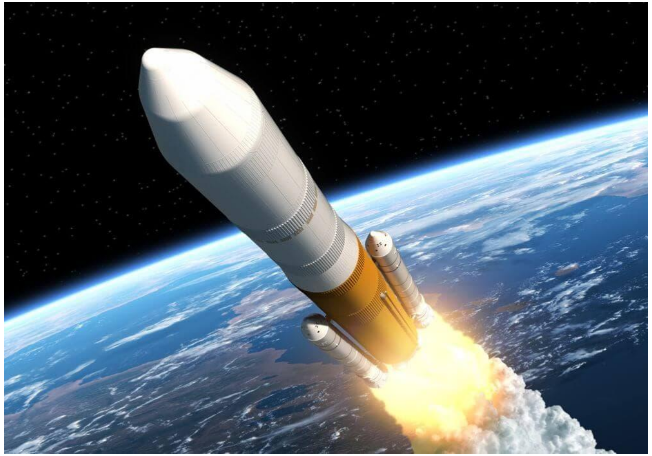
$$K_{\text{sharpen}} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

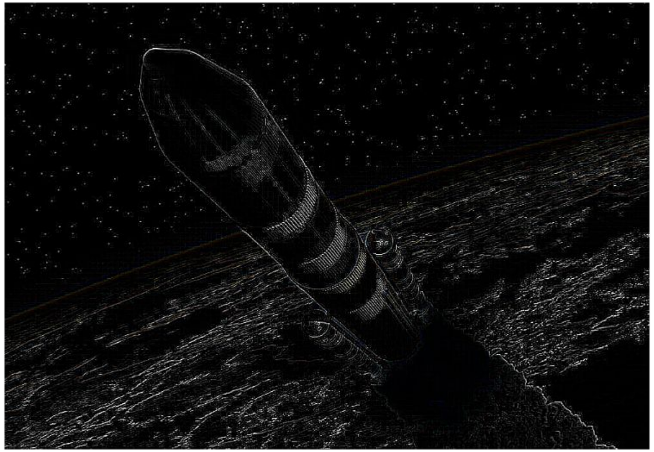


# Edge Detection

Highlights areas of rapid intensity change.

$$K_{\text{edge}} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

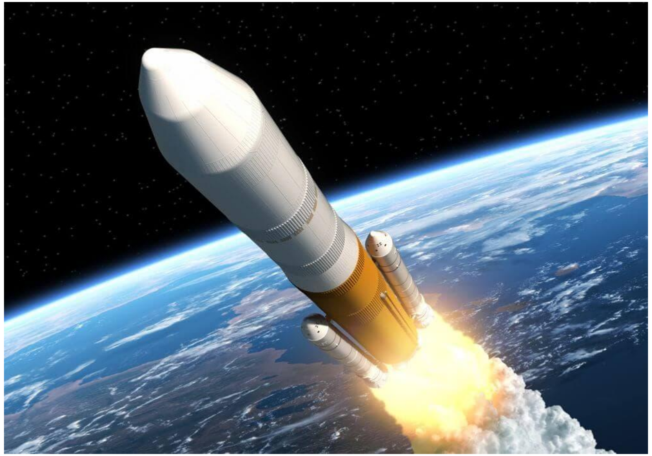




# Emboss

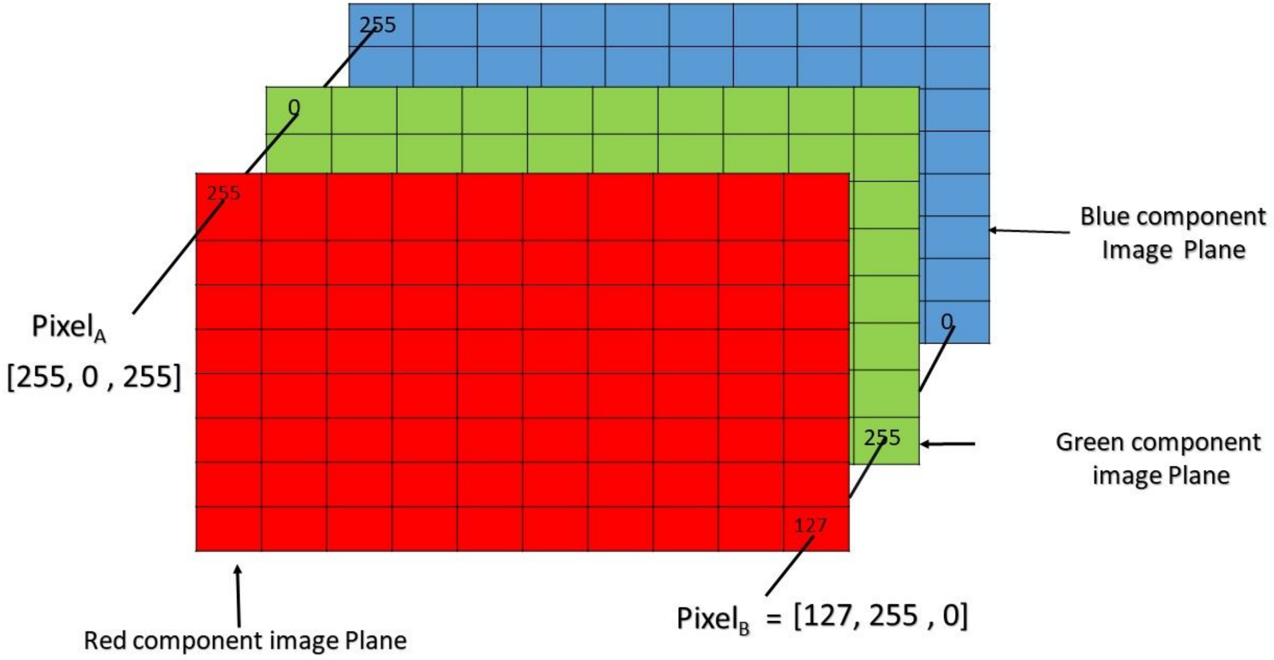
Creates a raised or lowered effect based on edge direction.

$$K_{\text{emboss}} = \begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$



# Compression with SVD

# Separating Different Color Channels




Pixel of an RGB image are formed from the corresponding pixel of the three component images

# Calculating Singular Value Decomposition $\quad \mathrm{A} = \mathrm{U}\,\Sigma V^T$

- **Left Singular Vectors (U)**: The columns of the matrix $U$ are the eigenvectors of the matrix $AA^T$. These vectors form an orthonormal basis for the codomain space.

- **Singular Values ($\Sigma$)**: The diagonal entries of the matrix $S$ are the singular values of $A$, arranged in descending order. As mentioned above, these singular values ($\sigma_i$) are the square roots of the eigenvalues ($\lambda_i$) of $A^TA$ (or $AA^T$).

$$\sigma_i = \sqrt{\lambda_i(A^TA)} = \sqrt{\lambda_i(AA^T)}$$

- **Right Singular Vectors (V)**: The columns of the matrix $V$ are the eigenvectors of the matrix $A^TA$. These vectors form an orthonormal basis for the domain space.

# Calculating Singular Value Decomposition

$$\mathrm{A} = \mathrm{U}\,\Sigma V^T$$

$$\mathbf{A}_{m\times n} = \left[\begin{array}{c|c|c|c} \vec{u}_1 & \vec{u}_2 & \dots & \vec{u}_m \end{array}\right] \begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_{\min(m,n)} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} \left[\begin{array}{c|c|c|c} \vec{v}_1 & \vec{v}_2 & \dots & \vec{v}_n \end{array}\right]^T$$

$$\mathrm{m} \times \mathrm{m} \qquad\qquad \mathrm{m} \times \mathrm{n} \qquad\qquad \mathrm{n} \times \mathrm{n}$$

# Reconstruction with k ranks

We reconstruct the approximation image (matrix) by multiplying the SVD decomposition components.

$$A_k = U_k \Sigma_k V_k^T.$$

$$\mathbf{A}_k \approx \begin{bmatrix} \vec{u}_1 & \vec{u}_2 & \dots & \vec{u}_k \end{bmatrix} \quad \begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_k \end{bmatrix} \quad \begin{bmatrix} \vec{v}_1 & \vec{v}_2 & \dots & \vec{v}_k \end{bmatrix}^T$$

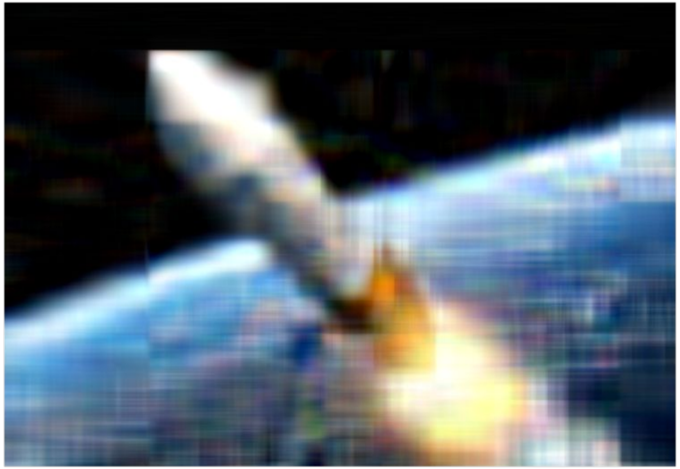
$$\mathrm{m} \times \mathrm{k} \qquad\qquad \mathrm{k} \times \mathrm{k} \qquad\qquad \mathrm{k} \times \mathrm{n}$$

Original

Rank 70

Rank 40

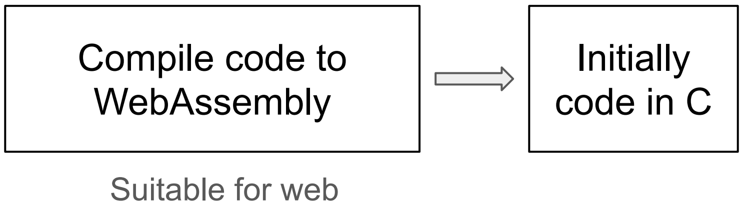
Rank 10

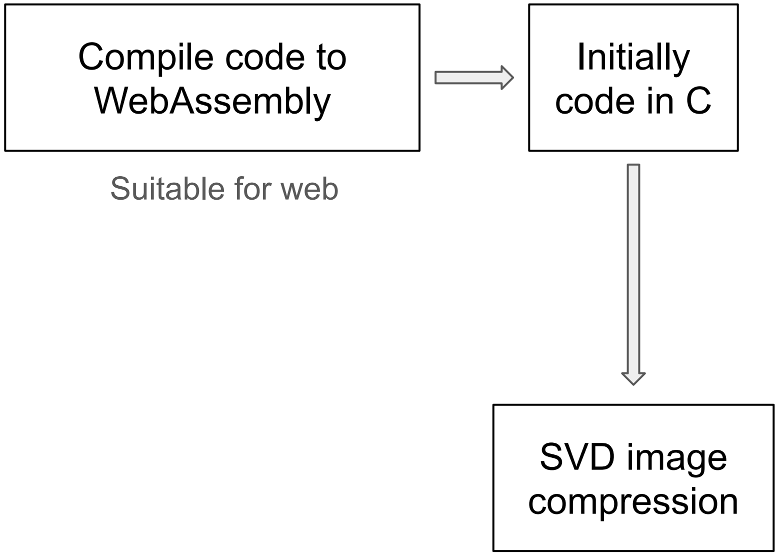


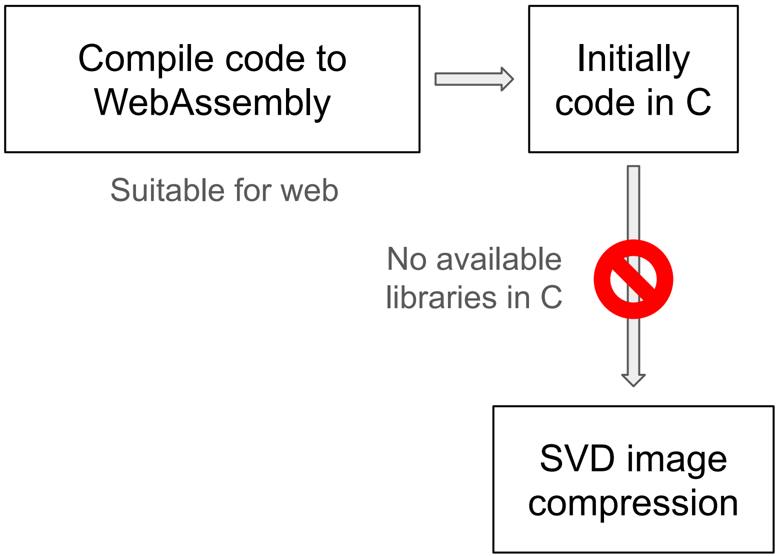
# Challenges

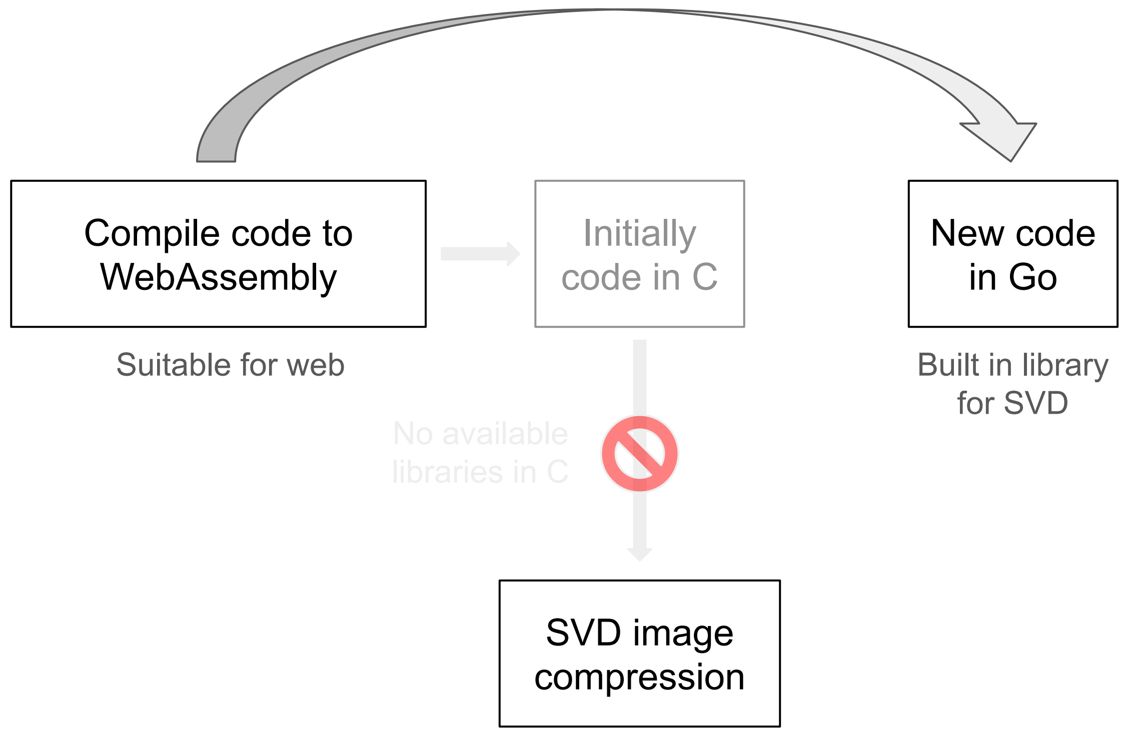
Compile code to WebAssembly

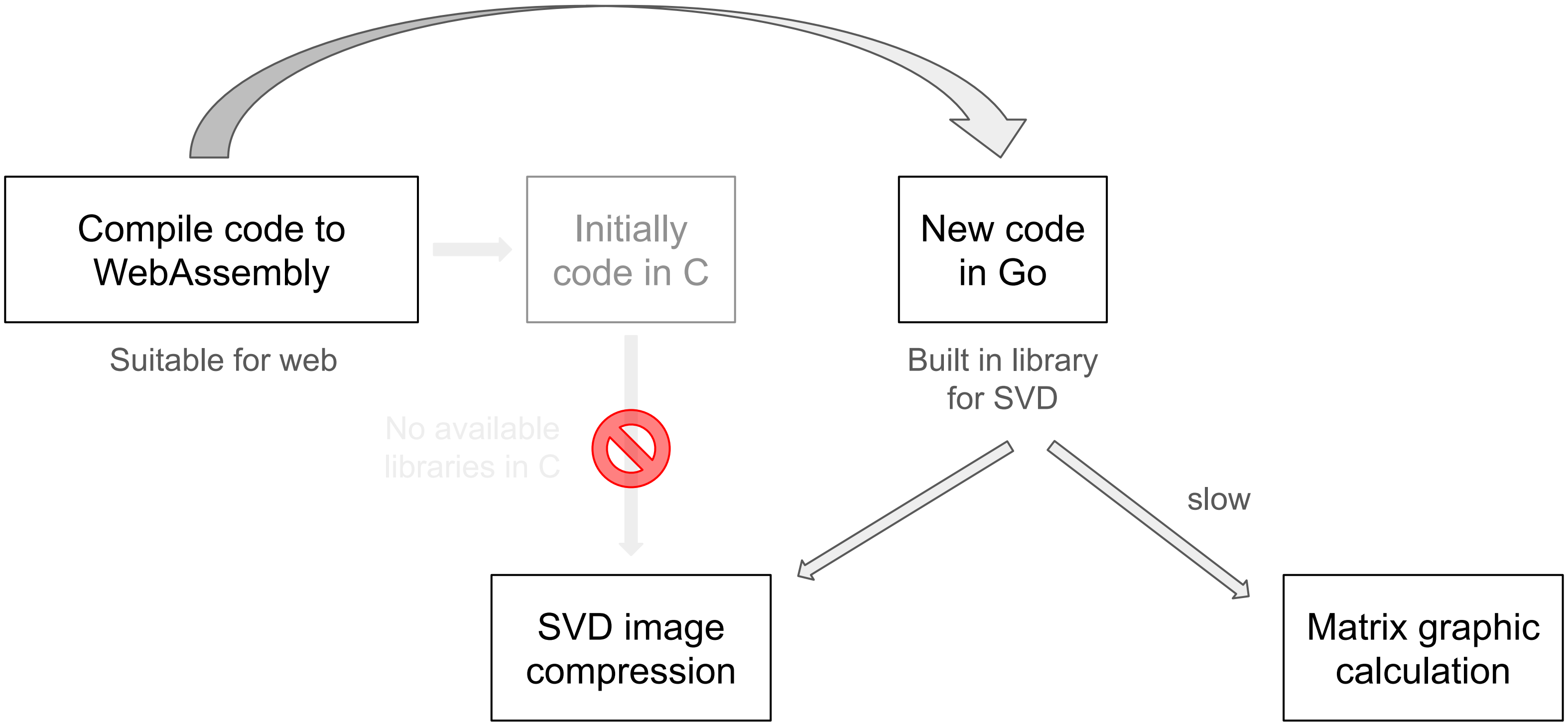
Suitable for web

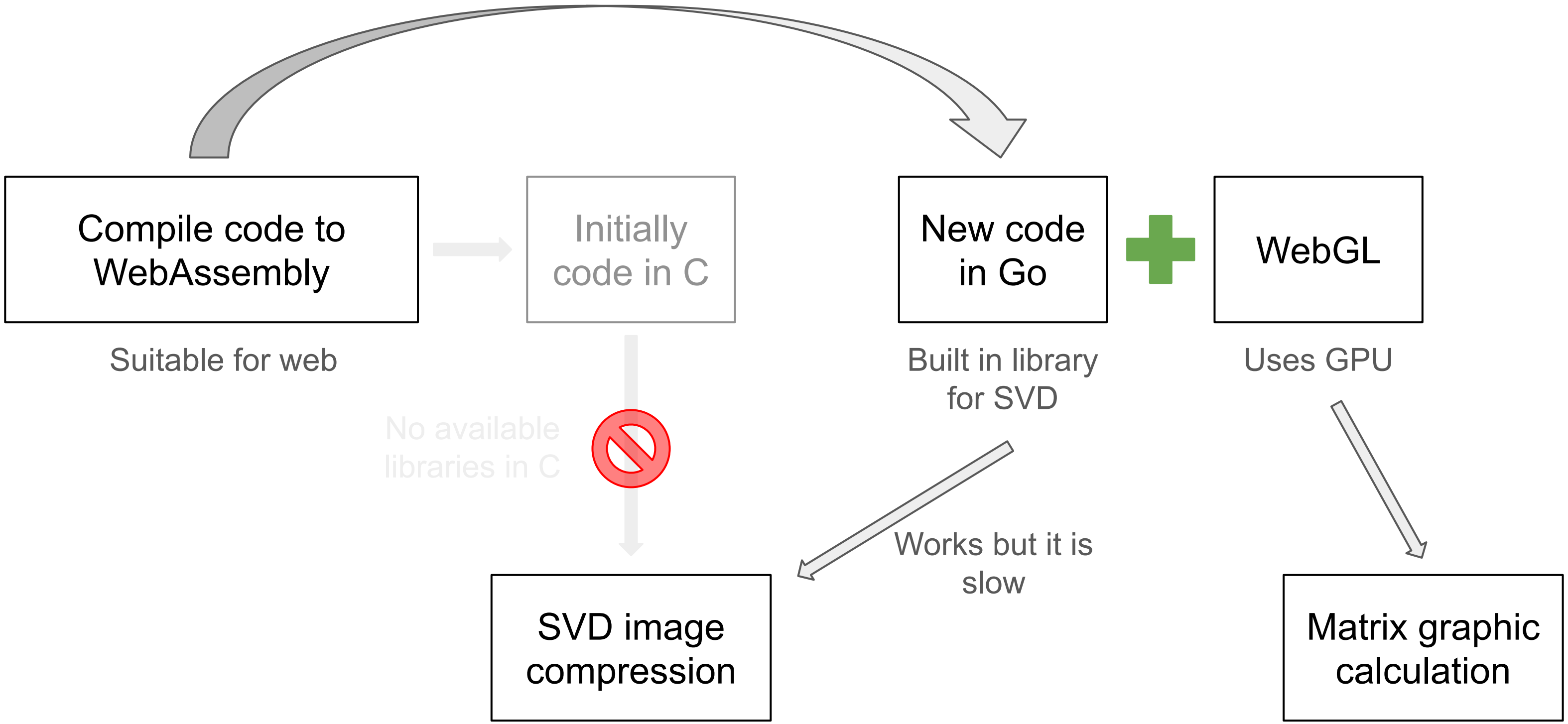

Compile code to WebAssembly
Initially code in C
Suitable for web


Compile code to WebAssembly
Suitable for web
Initially code in C
SVD image compression


Compile code to WebAssembly
Initially code in C
Suitable for web
No available libraries in C
SVD image compression


Compile code to WebAssembly
Initially code in C
New code in Go
Suitable for web
Built in library for SVD
No available libraries in C
SVD image compression


Compile code to WebAssembly
Initially code in C
New code in Go
Suitable for web
Built in library for SVD
No available libraries in C
slow
SVD image compression
Matrix graphic calculation


Compile code to WebAssembly
Suitable for web
Initially code in C
No available libraries in C
New code in Go
Built in library for SVD
WebGL
Uses GPU
SVD image compression
Works but it is slow
Matrix graphic calculation

Thank you