Linear Algebra Applications in Computer Graphics and Multimedia Systems

Project Report

Zewail City of Science, Technology and Innovation

Mathematics Department

MATH 201: Linear Algebra and Vector Geometry

Prepared by:

Mohammed Soliman [202402280]
Mahmoud Fady [202400514]
Abdelrhman Maniea [202400840]

December 31, 2025

Contents

1 Introduction

Linear algebra provides the mathematical foundation for modern computer graphics and multimedia systems. This comprehensive study examines how fundamental linear algebra concepts—from vector spaces and matrix transformations to orthogonal decompositions and quaternion algebra—enable sophisticated operations in digital media processing. The project is organized into three interconnected domains:

Image Processing: Demonstrating how matrices represent digital images and how linear transformations enable color manipulation, filtering, and compression techniques.

Audio Processing: Showing how audio signals inhabit vector spaces and how linear operations enable mixing, effects, and signal processing.

Transformations and Rotations: Exploring how matrix transformations form the core of geometric operations, from basic rotations to complete rendering pipelines.

Texture Mapping: Investegating how multimedia and computer graphics intersect in bulding virtual worlds.

Through mathematical formulations, practical examples, and implementation in both C++/OpenGL and C programming environments, this document illustrates the seamless translation of abstract linear algebra concepts into functional multimedia applications.

Project Structure

- **Section 2:** Matrix-based image representation and transformations
- **Section 3:** Vector space theory applied to audio signal processing
- **Section 4:** Geometric transformations in 3D computer graphics
- **Section 5:** Texture mapping, interpolation, and mipmapping
- **Section 6:** Practical implementations in C++/OpenGL and C

Key Linear Algebra Concepts Applied

- **Matrix Operations:** Image convolution, color transformations, DCT compression
- **Vector Spaces:** Audio signal representation, mixing, and effects
- **Linear Transformations:** 2D/3D graphics, projection operations
- **Orthogonal Decomposition:** Basis transformations in DCT
- **Homogeneous Coordinates:** Unified transformation framework
- **Quaternion Algebra:** Rotation representation without gimbal lock
- **Interpolation Techniques:** Bilinear interpolation for texture mapping
- **Pyramid Algorithms:** Mipmap construction and trilinear filtering

This integrated approach reveals the underlying mathematical unity between seemingly disparate multimedia domains, demonstrating how linear algebra serves as the common language of digital media processing.

2 Multimedia Systems

2.1 Image Processing as Matrix Operations

2.1.1 Image Representation as Matrices

A digital image is a discrete function $I : \mathbb{Z}^2 \to \mathbb{R}^k$ that can be represented as a matrix $\mathbf{I} \in \mathbb{R}^{m \times n}$ where:

- Each element I_{ij} represents the intensity at pixel (i, j)
- For grayscale: $k = 1$, $I_{ij} \in [0, 255]$
- For color (RGB): $k = 3$, represented as three matrices or a tensor

This matrix representation allows us to apply linear operators to manipulate images:

$$I' = \mathbf{A} \cdot I \cdot \mathbf{B}^T + C$$

where $\mathbf{A}, \mathbf{B}$ are transformation matrices and C is a constant matrix.

$$I = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mn} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

Example 1: Image as Matrix Element

A 3×3 grayscale image:

$$I = \begin{bmatrix} 120 & 80 & 200 \\ 150 & 100 & 50 \\ 90 & 180 & 110 \end{bmatrix}$$

Linear Algebra Interpretation:

- $I \in \mathbb{R}^{3 \times 3}$ is an element of the matrix space
- Each row can be viewed as a vector: $\mathbf{r}_1 = [120, 80, 200]^T \in \mathbb{R}^3$
- The image belongs to the vector space of all 3×3 matrices
- Dimension of this space: $\dim(\mathbb{R}^{3 \times 3}) = 9$

2.2 Color Transformations as Linear Maps

Color transformations are linear mappings $T : \mathbb{R}^3 \to \mathbb{R}^3$ defined by:

$$T(\mathbf{c}) = \mathbf{M}\mathbf{c} + \mathbf{b}$$

where:

- $\mathbf{c} = [r, g, b]^T \in \mathbb{R}^3$ is the input color vector
- $\mathbf{M} \in \mathbb{R}^{3 \times 3}$ is the transformation matrix
- $\mathbf{b} \in \mathbb{R}^3$ is the translation vector

Properties:

- Linearity: $T(\alpha\mathbf{c}_1 + \beta\mathbf{c}_2) = \alpha T(\mathbf{c}_1) + \beta T(\mathbf{c}_2)$ when $\mathbf{b} = 0$
- Affine transformation when $\mathbf{b} \neq 0$
- Preserves vector operations (scaling, addition)

Grayscale Conversion The standard RGB to grayscale conversion uses a weighted linear combination:

$$\begin{bmatrix} r' \\ g' \\ b' \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.299 & 0.587 & 0.114 \\ 0.299 & 0.587 & 0.114 \end{bmatrix} \begin{bmatrix} r \\ g \\ b \end{bmatrix}$$

Linear Algebra Interpretation: This is a projection onto a one-dimensional subspace spanned by $[0.299, 0.587, 0.114]^T$.

Example 2: Grayscale as Linear Transformation

Convert RGB$(255, 128, 64)$ to grayscale.
Solution:

$$\begin{aligned}\begin{bmatrix} r' \\ g' \\ b' \end{bmatrix} &= \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.299 & 0.587 & 0.114 \\ 0.299 & 0.587 & 0.114 \end{bmatrix} \begin{bmatrix} 255 \\ 128 \\ 64 \end{bmatrix} \\ &= \begin{bmatrix} 0.299(255) + 0.587(128) + 0.114(64) \\ 0.299(255) + 0.587(128) + 0.114(64) \\ 0.299(255) + 0.587(128) + 0.114(64) \end{bmatrix} \\ &= \begin{bmatrix} 76.245 + 75.136 + 7.296 \\ 158.677 \\ 158.677 \end{bmatrix} \approx \begin{bmatrix} 159 \\ 159 \\ 159 \end{bmatrix}\end{aligned}$$

Linear Algebra Verification:

- The transformation matrix has rank 1 (all rows identical)
- Column space: span$\{[1, 1, 1]^T\}$
- Projects all colors onto the gray diagonal in RGB space

Color Inversion as Affine Transformation Color inversion is an affine transformation: $T(\mathbf{c}) = -\mathbf{I}\mathbf{c} + \mathbf{1}$ where $\mathbf{I}$ is the identity matrix and $\mathbf{1} = [255, 255, 255]^T$.

This combines:

1. Linear part: reflection through origin $(-\mathbf{I})$
2. Translation: shift by $(255, 255, 255)$

$$\begin{bmatrix} r' \\ g' \\ b' \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} r \\ g \\ b \end{bmatrix} + \begin{bmatrix} 255 \\ 255 \\ 255 \end{bmatrix}$$

Example 3: Color Inversion

Invert RGB$(200, 100, 50)$.
Solution:

$$\begin{aligned}\begin{bmatrix} r' \\ g' \\ b' \end{bmatrix} &= \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 200 \\ 100 \\ 50 \end{bmatrix} + \begin{bmatrix} 255 \\ 255 \\ 255 \end{bmatrix} \\ &= \begin{bmatrix} -200 \\ -100 \\ -50 \end{bmatrix} + \begin{bmatrix} 255 \\ 255 \\ 255 \end{bmatrix} = \begin{bmatrix} 55 \\ 155 \\ 205 \end{bmatrix}\end{aligned}$$

Verification: $r + r' = 200 + 55 = 255$

2.2.1 Convolution as Matrix-Vector Product

Image convolution can be expressed as matrix multiplication, revealing its structure as a linear transformation.

1D Convolution as Toeplitz Matrix:

For a 1D signal $\mathbf{x} = [x_0, x_1, x_2, x_3]^T$ and kernel $\mathbf{h} = [b_0, b_1, b_2]^T$, convolution becomes:

$$\mathbf{y} = \mathbf{H}\mathbf{x}$$

where $\mathbf{H}$ is a *Toeplitz matrix* (constant diagonals):

$$\mathbf{H} = \begin{bmatrix} b_0 & 0 & 0 & 0 \\ b_1 & b_0 & 0 & 0 \\ b_2 & b_1 & b_0 & 0 \\ 0 & b_2 & b_1 & b_0 \\ 0 & 0 & b_2 & b_1 \\ 0 & 0 & 0 & b_2 \end{bmatrix}$$

Each row is the kernel shifted by one position. This structure reveals:

- **Translation invariance:** Same kernel applied at every position
- **Sparsity:** Most entries are zero (efficient storage/computation)
- **Banded structure:** Non-zero entries concentrated near diagonal

2D Convolution:
For 2D images, convolution extends to:

$$\text{vec}(Y) = \mathbf{K} \cdot \text{vec}(X)$$

where $\mathbf{K}$ is a doubly block-Toeplitz matrix combining row and column convolutions.
Key Properties:

- **Linearity:** $\mathcal{L}(af + bg) = a\mathcal{L}(f) + b\mathcal{L}(g)$
- **Rank:** Kernel rank determines dimensionality of image space
- **Null space:** Contains patterns kernel cannot detect

Box Blur Filter

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

Linear Algebra Interpretation: Each output pixel is a convex combination of neighboring inputs—the kernel defines the weighting scheme.

Example 4: 1D Convolution Matrix

Apply kernel $\mathbf{h} = [0.25, 0.5, 0.25]^T$ to signal $\mathbf{x} = [4, 8, 6, 2]^T$.
Step 1: Construct Toeplitz matrix

$$\mathbf{H} = \begin{bmatrix} 0.25 & 0 & 0 & 0 \\ 0.5 & 0.25 & 0 & 0 \\ 0.25 & 0.5 & 0.25 & 0 \\ 0 & 0.25 & 0.5 & 0.25 \\ 0 & 0 & 0.25 & 0.5 \\ 0 & 0 & 0 & 0.25 \end{bmatrix}$$

Step 2: Multiply

$$\mathbf{y} = \mathbf{H}\mathbf{x} = \begin{bmatrix} 0.25 & 0 & 0 & 0 \\ 0.5 & 0.25 & 0 & 0 \\ 0.25 & 0.5 & 0.25 & 0 \\ 0 & 0.25 & 0.5 & 0.25 \\ 0 & 0 & 0.25 & 0.5 \\ 0 & 0 & 0 & 0.25 \end{bmatrix} \begin{bmatrix} 4 \\ 8 \\ 6 \\ 2 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 4.0 \\ 6.5 \\ 5.5 \\ 2.5 \\ 0.5 \end{bmatrix}$$

Verification (manual convolution):

$$\begin{aligned} y_0 &= 0.25(4) = 1.0 \\ y_1 &= 0.5(4) + 0.25(8) = 4.0 \\ y_2 &= 0.25(4) + 0.5(8) + 0.25(6) = 6.5 \\ y_3 &= 0.25(8) + 0.5(6) + 0.25(2) = 5.5 \quad \checkmark \end{aligned}$$

2D Example: Box Blur
For a 3×3 region:

$$X = \begin{bmatrix} 100 & 120 & 140 \\ 110 & 130 & 150 \\ 90 & 100 & 110 \end{bmatrix}$$

Center pixel after blur:

$$y_{\text{center}} = \frac{1}{9}(100 + 120 + 140 + 110 + 130 + 150 + 90 + 100 + 110) = \frac{1050}{9} \approx 117$$

As inner product: $y = \langle \mathbf{k}, \mathbf{p} \rangle$ where $\mathbf{k} = \frac{1}{9}\mathbf{1}_9$ (uniform weights) and $\mathbf{p} = \text{vec}(X)$.

Sharpening Filter

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

Example 5: Sharpening as Linear Operator

Apply sharpen kernel to center pixel of Example 4's matrix.
Solution:

$$\begin{aligned} p' &= 0(100) - 1(120) + 0(140) - 1(110) + 5(130) \\ &\quad - 1(150) + 0(90) - 1(100) + 0(110) \\ &= -120 - 110 + 650 - 150 - 100 = 170 \end{aligned}$$

Linearity Check: If we scale input by 2:

$$p'' = -240 - 220 + 1300 - 300 - 200 = 340 = 2 \times 170$$

2.2.2 Discrete Cosine Transform (DCT)

The DCT is an *orthogonal change of basis* that transforms images from spatial to frequency domain, demonstrating spectral decomposition and optimal compression.

Mathematical Foundation:
For an $N \times N$ block, the 2D-DCT applies an orthogonal transformation:

$$F = \mathbf{C} \cdot I \cdot \mathbf{C}^T$$

where $\mathbf{C}$ is the DCT basis matrix with entries:

$$C_{ij} = \sqrt{\frac{2}{N}}\alpha_i \cos\left(\frac{(2j+1)i\pi}{2N}\right), \quad \alpha_i = \begin{cases} \frac{1}{\sqrt{2}} & i = 0 \\ 1 & i > 0 \end{cases}$$

The inverse is $I = \mathbf{C}^T \cdot F \cdot \mathbf{C}$ (using orthogonality $\mathbf{C}^T\mathbf{C} = \mathbf{I}$).

Interpretation: DCT decomposes images as linear combinations of orthonormal cosine basis functions: $I = \sum_{i,j} F_{ij}\psi_{ij}$, where F_{ij} are projection coefficients. The sandwich product $\mathbf{C}I\mathbf{C}^T$ applies separable 1D transforms (rows then columns), reducing complexity from $O(N^4)$ to $O(N^3)$.

Key Properties:

- **Orthogonality:** $\mathbf{C}^T\mathbf{C} = \mathbf{I}$ preserves norms (Parseval: $\|I\|_F^2 = \|F\|_F^2$)
- **Energy compaction:** Most energy concentrates in low-frequency coefficients (top-left)
- **Compression via truncation:** Small coefficients can be zeroed (low-rank approximation)

Example 6: 2×2 DCT Transform

Calculate DCT for:

$$I = \begin{bmatrix} 100 & 120 \\ 140 & 160 \end{bmatrix}$$

Step 1: Construct DCT basis matrix ($N = 2$)

$$\mathbf{C} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \approx \begin{bmatrix} 0.707 & 0.707 \\ 0.707 & -0.707 \end{bmatrix}$$

Verify: $\mathbf{C}^T\mathbf{C} = \mathbf{I}$

Step 2: Apply transform $F = \mathbf{C}I\mathbf{C}^T$

$$\mathbf{C}I = \begin{bmatrix} 0.707 & 0.707 \\ 0.707 & -0.707 \end{bmatrix} \begin{bmatrix} 100 & 120 \\ 140 & 160 \end{bmatrix} = \begin{bmatrix} 169.7 & 197.9 \\ -28.3 & -28.3 \end{bmatrix}$$

$$F = \begin{bmatrix} 169.7 & 197.9 \\ -28.3 & -28.3 \end{bmatrix} \begin{bmatrix} 0.707 & 0.707 \\ 0.707 & -0.707 \end{bmatrix} = \begin{bmatrix} 260 & -20 \\ -40 & 0 \end{bmatrix}$$

Interpretation:

- $F(0,0) = 260$: DC coefficient (average intensity × 2)
- $F(0,1) = -20$: Horizontal frequency (left darker than right)
- $F(1,0) = -40$: Vertical frequency (top darker than bottom)

Energy distribution: $\frac{260^2}{260^2+20^2+40^2} = 97.1\%$ in DC alone! Demonstrates excellent energy compaction—JPEG compression exploits this by quantizing small coefficients more aggressively.

2.2.3 Image Scaling as Matrix Operations

Image downscaling is a *linear transformation* that demonstrates rank deficiency and information loss.

Mathematical Formulation:

Downscaling by averaging 2×2 blocks defines a linear map $T : \mathbb{R}^{m\times n} \to \mathbb{R}^{(m/2)\times(n/2)}$:

$$I'_{x,y} = \frac{1}{4}(I_{2x,2y} + I_{2x+1,2y} + I_{2x,2y+1} + I_{2x+1,2y+1})$$

In vectorized form:

$$\text{vec}(I') = \mathbf{A} \cdot \text{vec}(I)$$

where $\mathbf{A} \in \mathbb{R}^{(mn/4)\times mn}$ has four 1/4 entries per row (convex combination).
Linear Operator Properties:

- **Rank deficiency:** $\text{rank}(\mathbf{A}) = mn/4 < mn$ (many inputs map to same output)
- **Null space:** $\dim(\mathcal{N}(\mathbf{A})) = 3mn/4$ (high-frequency patterns that average to zero)
- **Non-invertible:** Upscaling is ill-posed (infinitely many preimages)
- **Separability:** $\mathbf{A}_{2D} = \mathbf{A}_{1D} \otimes \mathbf{A}_{1D}$ (Kronecker product structure)

Example 7: Downscaling as Linear Map

Downscale 4×4 to 2×2:

$$I = \begin{bmatrix} 100 & 120 & 140 & 160 \\ 110 & 130 & 150 & 170 \\ 90 & 100 & 110 & 120 \\ 80 & 90 & 100 & 110 \end{bmatrix}$$

Block averages:

$$I'(0,0) = \frac{100+120+110+130}{4} = 115, \quad I'(0,1) = \frac{140+160+150+170}{4} = 155$$

$$I'(1,0) = \frac{90+100+80+90}{4} = 90, \quad I'(1,1) = \frac{110+120+100+110}{4} = 110$$

Result: $I' = \begin{bmatrix} 115 & 155 \\ 90 & 110 \end{bmatrix}$
Matrix form: $\text{vec}(I') = \mathbf{A} \cdot \text{vec}(I)$
Analysis: $\text{rank}(\mathbf{A}) = 4$, $\dim(\mathcal{N}(\mathbf{A})) = 12 \to 75\%$ information loss. High-frequency patterns like checkerboards are in the null space, explaining why upscaling methods produce smooth results.

2.3 Audio Processing as Vector Operations

2.3.1 Audio Signals in Vector Spaces

Digital audio is a discrete-time signal represented as a vector in $\mathbb{R}^n$:

$$\mathbf{a} = [a_1, a_2, \ldots, a_n]^T \in \mathbb{R}^n$$

The set of all audio signals of length n forms a vector space with:

- Vector addition: $(\mathbf{a} + \mathbf{b})_i = a_i + b_i$ (mixing signals)
- Scalar multiplication: $(c\mathbf{a})_i = ca_i$ (amplification)
- Zero vector: silence $\mathbf{0} = [0, 0, \ldots, 0]^T$
- Dimension: $\dim(\mathbb{R}^n) = n$

Common operations are linear transformations on this space.

2.3.2 Amplification as Scalar Multiplication

Amplification is scalar multiplication in vector space:

$$\mathbf{a}' = k \cdot \mathbf{a}, \quad k \in \mathbb{R}^+$$

Decibel gain: $G_{dB} = 20 \log_{10}(k)$

Example 8: Volume Control

Amplify $\mathbf{a} = [100, -50, 200, -150]^T$ by factor $k = 2$.

Solution:

$$\mathbf{a}' = 2 \cdot \begin{bmatrix} 100 \\ -50 \\ 200 \\ -150 \end{bmatrix} = \begin{bmatrix} 200 \\ -100 \\ 400 \\ -300 \end{bmatrix}$$

Decibel gain:

$$G_{dB} = 20 \log_{10}(2) = 20(0.301) \approx 6.02 \text{ dB}$$

Linearity verification:

$$k(\alpha \mathbf{a}_1 + \beta \mathbf{a}_2) = \alpha(k\mathbf{a}_1) + \beta(k\mathbf{a}_2)$$

2.3.3 Audio Mixing as Linear Combination

Audio mixing is a linear combination of vectors:

$$\mathbf{m} = \sum_{i=1}^{N} w_i \mathbf{a}_i = w_1\mathbf{a}_1 + w_2\mathbf{a}_2 + \cdots + w_N\mathbf{a}_N$$

where $w_i \in \mathbb{R}$ are mixing weights. In matrix form:

$$\mathbf{m} = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_N \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = \mathbf{A}\mathbf{w}$$

Properties:

- Result lies in span of input signals: $\mathbf{m} \in \text{span}\{\mathbf{a}_1, \ldots, \mathbf{a}_N\}$
- If weights sum to 1, result is a convex combination
- Dimension of mixed signal space: $\dim(\text{span}) \leq N$

Example 9: Two-Track Mixing

Mix signals with weights $w_1 = 0.6$, $w_2 = 0.4$:

$$\mathbf{a}_1 = [100, 200, 300]^T$$
$$\mathbf{a}_2 = [50, 150, 250]^T$$

Solution:

$$\begin{aligned} \mathbf{m} &= 0.6\mathbf{a}_1 + 0.4\mathbf{a}_2 \\ &= 0.6 \begin{bmatrix} 100 \\ 200 \\ 300 \end{bmatrix} + 0.4 \begin{bmatrix} 50 \\ 150 \\ 250 \end{bmatrix} \\ &= \begin{bmatrix} 60 \\ 120 \\ 180 \end{bmatrix} + \begin{bmatrix} 20 \\ 60 \\ 100 \end{bmatrix} = \begin{bmatrix} 80 \\ 180 \\ 280 \end{bmatrix} \end{aligned}$$

Verification: $\mathbf{m}$ is in $\text{span}\{\mathbf{a}_1, \mathbf{a}_2\}$

2.3.4 Echo Effect as Time-Delayed Sum

Echo is implemented using a shift operator and scalar multiplication:

$$\mathbf{y} = \mathbf{x} + \alpha \mathbf{S}^D \mathbf{x}$$

where:

- $\mathbf{S}$ is the shift matrix (delays by 1 sample)
- D is the delay in samples
- $\alpha \in [0, 1]$ is attenuation factor

The shift matrix $\mathbf{S}$ is:

$$\mathbf{S} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

This is a linear operator: $\mathbf{y} = (\mathbf{I} + \alpha \mathbf{S}^D)\mathbf{x}$

Example 10: Echo Implementation

Apply echo with $D = 3$ samples, $\alpha = 0.5$ to $\mathbf{x} = [100, 200, 300, 400, 500, 600]^T$.
Solution using operator:

$$\mathbf{y} = \mathbf{x} + 0.5\mathbf{S}^3\mathbf{x}$$

$$\begin{aligned} y[0] &= 100 + 0.5(0) = 100 \\ y[1] &= 200 + 0.5(0) = 200 \\ y[2] &= 300 + 0.5(0) = 300 \\ y[3] &= 400 + 0.5(100) = 450 \\ y[4] &= 500 + 0.5(200) = 600 \\ y[5] &= 600 + 0.5(300) = 750 \end{aligned}$$

Result: $\mathbf{y} = [100, 200, 300, 450, 600, 750]^T$
Matrix form:

$$\begin{bmatrix} 100 \\ 200 \\ 300 \\ 450 \\ 600 \\ 750 \end{bmatrix} = \left(\mathbf{I} + 0.5 \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \right) \begin{bmatrix} 100 \\ 200 \\ 300 \\ 400 \\ 500 \\ 600 \end{bmatrix}$$

3 Computer Graphics

3.1 Rotation Representations

Euler Angles Euler angles represent 3D rotations as three successive rotations about coordinate axes:
Euler angles decompose a rotation matrix $\mathbf{R} \in \mathbb{R}^{3\times3}$ as a product of elementary rotations:

$$\mathbf{R} = \mathbf{R}_z(\psi) \cdot \mathbf{R}_y(\theta) \cdot \mathbf{R}_x(\phi)$$

where:

$$\mathbf{R}_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}, \quad \mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \quad \mathbf{R}_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Properties and Issues:

- The decomposition is not unique due to gimbal lock
- Each rotation matrix $\mathbf{R}_i$ is orthogonal: $\mathbf{R}_i^{-1} = \mathbf{R}_i^T$
- Composition preserves orthogonality: $\mathbf{R}\mathbf{R}^T = \mathbf{I}$
- Determinant: $\det(\mathbf{R}) = 1$ (proper rotation)

Example 30: Euler Angles Rotation and Gimbal Lock

Rotate point $P(1, 0, 0)$ by $90°$ about the z-axis using ZYX Euler angles.

Step 1: Define Euler angles For a pure rotation about the z-axis:

$$\psi = 90° = \pi/2, \quad \theta = 0°, \quad \phi = 0°$$

Step 2: Construct rotation matrix

$$\mathbf{R} = \mathbf{R}_z(90°) \cdot \mathbf{R}_y(0°) \cdot \mathbf{R}_x(0°) = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \mathbf{I} \cdot \mathbf{I} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Step 3: Apply rotation

$$P' = \mathbf{R}P = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Result: $P'(0, 1, 0)$

Step 4: Demonstrate Gimbal Lock When $\theta = 90°$, the x- and z-axes align:

$$\mathbf{R}_y(90°) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

Now observe:

$$\mathbf{R}_z(\psi)\mathbf{R}_y(90°)\mathbf{R}_x(\phi) = \mathbf{R}_z(\psi + \phi)\mathbf{R}_y(90°)$$

Linear Algebra Explanation:

- Rotations $\mathbf{R}_x$ and $\mathbf{R}_z$ become linearly dependent
- The transformation loses one degree of freedom
- The rotation matrix becomes singular in Euler angle space
- This causes problems in animation interpolation

Quaternions Quaternions provide an elegant solution to rotation representation without gimbal lock:
A quaternion is an extension of complex numbers: $\mathbb{H} = \{w+xi+yj+zk\}$ where $i^2 = j^2 = k^2 = ijk = -1$.

In vector form: $\mathbf{q} = (w, \vec{v})$ where $w \in \mathbb{R}$ and $\vec{v} = (x, y, z) \in \mathbb{R}^3$.

Quaternion to Matrix Conversion:

The rotation matrix derived from a unit quaternion $\mathbf{q} = (w, x, y, z)$ is:

$$\mathbf{R} = \begin{bmatrix} 1 - 2(y^2 + z^2) & 2(xy - wz) & 2(xz + wy) \\ 2(xy + wz) & 1 - 2(x^2 + z^2) & 2(yz - wx) \\ 2(xz - wy) & 2(yz + wx) & 1 - 2(x^2 + y^2) \end{bmatrix}$$

This matrix emerges from expanding the quaternion rotation formula $P' = \mathbf{q}P\mathbf{q}^{-1}$ using quaternion multiplication rules. The diagonal terms $(1 - 2y^2 - 2z^2, 1 - 2x^2 - 2z^2, 1 - 2x^2 - 2y^2)$ represent how much each axis preserves itself during rotation, while off-diagonal terms combine cross-products of quaternion components, encoding the axis-angle rotation geometry. The factor of 2 appears because the half-angle $\theta/2$ in quaternions gets doubled during the $\mathbf{q} \cdot \mathbf{q}^{-1}$ operation.

Rotation Properties:

- Unit quaternions: $\|\mathbf{q}\| = \sqrt{w^2 + x^2 + y^2 + z^2} = 1$ represent rotations
- Rotation by angle θ about axis $\hat{n}$: $\mathbf{q} = \cos(\theta/2) + \sin(\theta/2)\hat{n}$
- Conjugate (inverse): $\mathbf{q}^{-1} = (w, -\vec{v})$ for unit quaternions
- Rotation formula: $P' = \mathbf{q}P\mathbf{q}^{-1}$

Advantages:

- No gimbal lock (smooth interpolation)
- Compact representation (4 values vs. 9 for matrices)
- Efficient composition: $\mathbf{q}_{total} = \mathbf{q}_2 \cdot \mathbf{q}_1$

Example 29: Constructing Rotation Matrix from Quaternion

Construct the rotation matrix for a 90 rotation about the z-axis using quaternions.

Step 1: Create rotation quaternion

- Axis: $\hat{n} = (0, 0, 1)$, Angle: $\theta = 90° = \pi/2$
- $\mathbf{q} = \cos(\theta/2) + \sin(\theta/2)\hat{n}$
- $\mathbf{q} = \cos(45°) + \sin(45°)(0i + 0j + 1k)$
- $\mathbf{q} = \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}k = (w, x, y, z) = \left(\frac{\sqrt{2}}{2}, 0, 0, \frac{\sqrt{2}}{2}\right)$

Step 2: Apply quaternion-to-matrix formula With $w = \frac{\sqrt{2}}{2}$, $x = 0$, $y = 0$, $z = \frac{\sqrt{2}}{2}$:

$$\mathbf{R} = \begin{bmatrix} 1 - 2(0^2 + z^2) & 2(0 \cdot 0 - wz) & 2(0 \cdot z + w \cdot 0) \\ 2(0 \cdot 0 + wz) & 1 - 2(0^2 + z^2) & 2(0 \cdot z - w \cdot 0) \\ 2(0 \cdot z - w \cdot 0) & 2(0 \cdot z + w \cdot 0) & 1 - 2(0^2 + 0^2) \end{bmatrix}$$

$$= \begin{bmatrix} 1 - 2\left(\frac{1}{2}\right) & -2\left(\frac{1}{2}\right) & 0 \\ 2\left(\frac{1}{2}\right) & 1 - 2\left(\frac{1}{2}\right) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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

Verification: This is indeed the standard rotation matrix for 90 counterclockwise rotation about the z-axis. Applying it to point $(1, 0, 0)$ gives $(0, 1, 0)$, as expected.

3.2 Transformation Pipeline

The 3D graphics pipeline consists of successive linear transformations:

$$P_{\text{screen}} = \mathbf{M}_{\text{viewport}} \cdot \mathbf{M}_{\text{projection}} \cdot \mathbf{M}_{\text{view}} \cdot \mathbf{M}_{\text{model}} \cdot P_{\text{local}}$$

Properties:

- Each $\mathbf{M}_i$ is a 4×4 matrix (homogeneous coordinates)
- The pipeline preserves linearity: overall transformation is linear
- Model-view-projection matrix: $\mathbf{M}_{\text{MVP}} = \mathbf{PVM}$

3.2.1 Transformation Pipeline

1. **Model Matrix:** Local space → World space (transformation hierarchy)
2. **View Matrix:** World space → Camera space (change of basis)
3. **Projection Matrix:** Camera space → Clip space (perspective/orthographic)
4. **Perspective Division:** Clip space → NDC (w-division)
5. **Viewport Transform:** NDC → Screen coordinates (scaling and translation)

3.2.2 View Matrix Construction

The view matrix performs a change of basis from world space to camera space:

$$\mathbf{V} = \begin{bmatrix} \hat{u}_x & \hat{u}_y & \hat{u}_z & -\hat{u} \cdot \mathbf{c} \\ \hat{v}_x & \hat{v}_y & \hat{v}_z & -\hat{v} \cdot \mathbf{c} \\ \hat{w}_x & \hat{w}_y & \hat{w}_z & -\hat{w} \cdot \mathbf{c} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where:

- $\hat{w} = \text{normalize}(\text{target} - \text{eye})$ (look direction)
- $\hat{u} = \text{normalize}(\hat{w} \times \text{up})$ (right direction)
- $\hat{v} = \hat{w} \times \hat{u}$ (up direction)
- $\mathbf{c}$ = camera position in world space

This is an orthogonal matrix (in the 3×3 block) representing a rotation and translation.

Example 26: Construct View Matrix

Camera at position $\mathbf{c} = (0, 0, 5)$, looking at origin, up$= (0, 1, 0)$.

Step 1: Calculate orthonormal basis

$$\hat{w} = \frac{\mathbf{c} - \text{target}}{\|\mathbf{c} - \text{target}\|} = \frac{(0,0,5) - (0,0,0)}{5} = (0,0,1)$$
$$\hat{u} = \frac{\text{up} \times \hat{w}}{\|\text{up} \times \hat{w}\|} = \frac{(0,1,0) \times (0,0,1)}{\|(1,0,0)\|} = (1,0,0)$$
$$\hat{v} = \hat{w} \times \hat{u} = (0,0,1) \times (1,0,0) = (0,1,0)$$

Step 2: Construct view matrix

$$\mathbf{V} = \begin{bmatrix} 1 & 0 & 0 & -\hat{u} \cdot \mathbf{c} \\ 0 & 1 & 0 & -\hat{v} \cdot \mathbf{c} \\ 0 & 0 & 1 & -\hat{w} \cdot \mathbf{c} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Step 3: Verify Point $(0,0,0)$ in world space becomes $(0,0,-5)$ in camera space.

3.2.3 Projection Matrices

Projection Matrix:

The standard perspective projection matrix is:

$$\mathbf{P}_{\text{persp}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where n, f are near/far planes, and l, r, b, t define frustum bounds.

Mathematical Properties:

- **Homogeneous linearity:** Non-linear in 3D but linear in $\mathbb{P}^3$ (projective space)
- **Perspective division:** Final step $\mathbf{x}_{\text{NDC}} = \mathbf{x}_{\text{clip}}/w$ creates foreshortening
- **Depth non-linearity:** z component uses rational function for depth buffer precision
- **Frustum mapping:** Transforms view frustum to NDC cube $[-1,1]^3$

the -1 in row 4? Copies $-z$ to w coordinate, so perspective division by $w = -z$ makes distant objects smaller (larger divisor). This encodes the key perspective property: apparent size $\propto 1/\text{distance}$.

Example 28: Perspective Projection

Project point $P(1,0,-10)$ with FOV= 90, aspect= 1.0, $n = 0.1$, $f = 100$.

Step 1: Calculate frustum bounds

$$t = n\tan(\text{FOV}/2) = 0.1\tan(45) = 0.1, \quad b = -0.1, \quad r = 0.1, \quad l = -0.1$$

Step 2: Construct matrix

$$\mathbf{P} = \begin{bmatrix} \frac{2(0.1)}{0.2} & 0 & 0 & 0 \\ 0 & \frac{2(0.1)}{0.2} & 0 & 0 \\ 0 & 0 & -\frac{100.1}{99.9} & -\frac{20}{99.9} \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1.002 & -0.2002 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Step 3: Apply transformation

$$\mathbf{P}\begin{bmatrix}1\\0\\-10\\1\end{bmatrix} = \begin{bmatrix}1\cdot 1\\1\cdot 0\\-1.002(-10)-0.2002\\-1(-10)\end{bmatrix} = \begin{bmatrix}1\\0\\10.02-0.2002\\10\end{bmatrix} = \begin{bmatrix}1\\0\\9.82\\10\end{bmatrix}$$

Step 4: Perspective division (normalize by w)

$$\mathbf{P}_{\text{NDC}} = \frac{1}{10}\begin{bmatrix}1\\0\\9.82\\10\end{bmatrix} = \begin{bmatrix}0.1\\0\\0.982\\1\end{bmatrix}$$

Interpretation: Point at distance 10 is scaled by factor 1/10 in x, y (foreshortening). The z-value $0.982 \in [-1, 1]$ encodes depth for the depth buffer. Objects farther away have z closer to 1.

3.3 Texture Mapping and Interpolation

3.3.1 Bilinear Interpolation

Bilinear interpolation computes texture values between texels using a *bilinear form*—a function that is linear in each variable separately. This is a fundamental example of multilinear algebra applied to computer graphics.

Theoretical Foundation:

A bilinear form is a map $B : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ that satisfies:

- Linearity in first argument: $B(\alpha\mathbf{u}_1 + \beta\mathbf{u}_2, \mathbf{v}) = \alpha B(\mathbf{u}_1, \mathbf{v}) + \beta B(\mathbf{u}_2, \mathbf{v})$
- Linearity in second argument: $B(\mathbf{u}, \alpha\mathbf{v}_1 + \beta\mathbf{v}_2) = \alpha B(\mathbf{u}, \mathbf{v}_1) + \beta B(\mathbf{u}, \mathbf{v}_2)$

Every bilinear form can be represented as a matrix "sandwich": $B(\mathbf{u}, \mathbf{v}) = \mathbf{u}^T\mathbf{M}\mathbf{v}$, where $\mathbf{M}$ encodes the interaction between dimensions.

Tensor Product Structure:

Bilinear interpolation decomposes as:

$$f(u, v) = \ell_u(u) \otimes \ell_v(v)$$

where ℓ_u and ℓ_v are linear interpolation operators along each axis. This *tensor product* structure means we interpolate along u first, then along v (or vice versa—the order doesn't matter).

Basis Function Representation:

Define the basis functions:

$$\phi_0(\alpha) = 1 - \alpha, \quad \phi_1(\alpha) = \alpha$$

These are the *hat functions* that satisfy $\phi_i(\alpha_j) = \delta_{ij}$ (Kronecker delta). The interpolation becomes:

$$f(u, v) = \sum_{i=0}^{1}\sum_{j=0}^{1} \phi_i(\alpha)\phi_j(\beta)P_{ij}$$

This is a weighted sum where weights form a *partition of unity*: $\sum_{i,j} \phi_i(\alpha)\phi_j(\beta) = 1$.

Standard Form:

Given texture coordinates (u, v) with $u_0 = \lfloor u \rfloor$, $u_1 = u_0 + 1$, $v_0 = \lfloor v \rfloor$, $v_1 = v_0 + 1$, and texels $P_{00}, P_{10}, P_{01}, P_{11}$:

$$f(u, v) = (1-\alpha)(1-\beta)P_{00} + \alpha(1-\beta)P_{10} + (1-\alpha)\beta P_{01} + \alpha\beta P_{11}$$

where $\alpha = u - u_0$ and $\beta = v - v_0$ are the local coordinates within the texel.

Matrix Form (Bilinear Form Representation):

$$f(u,v) = \begin{bmatrix} 1-\alpha & \alpha \end{bmatrix} \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix} \begin{bmatrix} 1-\beta \\ \beta \end{bmatrix} = \boldsymbol{\phi}^T(\alpha)\mathbf{P}\boldsymbol{\phi}(\beta)$$

The matrix $\mathbf{P}$ contains the texel values, while the vectors $\boldsymbol{\phi}(\alpha) = [1-\alpha, \alpha]^T$ and $\boldsymbol{\phi}(\beta) = [1-\beta, \beta]^T$ contain the basis function weights. This form reveals the *separability*: we can compute row interpolations first, then interpolate the results.

Geometric Interpretation:

The coefficients $(1-\alpha)(1-\beta)$, $\alpha(1-\beta)$, $(1-\alpha)\beta$, $\alpha\beta$ represent *normalized areas* in the unit square. Point (u, v) divides the unit texel into four rectangles, and each texel's contribution is proportional to the area of the opposite rectangle—these are *barycentric coordinates* for the bilinear case.

Properties:

- **Bilinearity:** Linear in each dimension separately (but not globally linear)
- **Continuity:** C^0 continuous across texel boundaries
- **Interpolation property:** $f(i,j) = P_{ij}$ for integer coordinates
- **Convex combination:** Output is always within range of input values
- **Affine invariance:** Preserved under affine transformations

Example 31: Bilinear Interpolation

Given a 2×2 texture:

$$T = \begin{bmatrix} 100 & 150 \\ 200 & 250 \end{bmatrix}$$

with texel values: $P_{00} = 100$, $P_{10} = 200$, $P_{01} = 150$, $P_{11} = 250$.
Compute the interpolated value at $(u, v) = (0.3, 0.7)$.

Solution: Step 1: Compute local coordinates

$$\alpha = u - \lfloor u \rfloor = 0.3, \quad \beta = v - \lfloor v \rfloor = 0.7$$

Step 2: Compute basis function weights

$$\boldsymbol{\phi}(\alpha) = \begin{bmatrix} 1-0.3 \\ 0.3 \end{bmatrix} = \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix}, \quad \boldsymbol{\phi}(\beta) = \begin{bmatrix} 1-0.7 \\ 0.7 \end{bmatrix} = \begin{bmatrix} 0.3 \\ 0.7 \end{bmatrix}$$

Step 3: Apply bilinear form (matrix sandwich)

$$f(0.3, 0.7) = \begin{bmatrix} 0.7 & 0.3 \end{bmatrix} \begin{bmatrix} 100 & 150 \\ 200 & 250 \end{bmatrix} \begin{bmatrix} 0.3 \\ 0.7 \end{bmatrix}$$

Step 4: First multiply (interpolate along rows)

$$\begin{bmatrix} 0.7 & 0.3 \end{bmatrix} \begin{bmatrix} 100 & 150 \\ 200 & 250 \end{bmatrix} = \begin{bmatrix} 0.7(100) + 0.3(200) & 0.7(150) + 0.3(250) \end{bmatrix} = \begin{bmatrix} 130 & 180 \end{bmatrix}$$

This gives interpolated values at $(0.3, 0)$ and $(0.3, 1)$.

Step 5: Second multiply (interpolate along column)

$$\begin{bmatrix} 130 & 180 \end{bmatrix} \begin{bmatrix} 0.3 \\ 0.7 \end{bmatrix} = 130(0.3) + 180(0.7) = 39 + 126 = 165$$

Verification using direct formula:

$$\begin{aligned} f(0.3, 0.7) &= (1-0.3)(1-0.7) \cdot 100 + 0.3(1-0.7) \cdot 200 \\ &\quad + (1-0.3) \cdot 0.7 \cdot 150 + 0.3 \cdot 0.7 \cdot 250 \\ &= 0.7 \cdot 0.3 \cdot 100 + 0.3 \cdot 0.3 \cdot 200 + 0.7 \cdot 0.7 \cdot 150 + 0.3 \cdot 0.7 \cdot 250 \\ &= 21 + 18 + 73.5 + 52.5 = 165 \end{aligned}$$

Geometric interpretation: The point $(0.3, 0.7)$ creates four rectangular areas with weights: 0.21 (opposite P_{00}), 0.09 (opposite P_{10}), 0.49 (opposite P_{01}), 0.21 (opposite P_{11}). Note: $0.21 + 0.09 + 0.49 + 0.21 = 1$ (partition of unity).

3.3.2 Mipmaps

Mipmaps are a precomputed pyramid of texture resolutions used for efficient texture filtering. Each level l is $\frac{1}{2}$ the dimensions of level $l-1$.

Construction: For a $2^k \times 2^k$ texture, level l ($0 \leq l \leq k$) has dimensions $2^{k-l} \times 2^{k-l}$.

Each texel at level l is the average of 4 texels from level $l-1$:

$$T_l(i,j) = \frac{1}{4}\left(T_{l-1}(2i, 2j) + T_{l-1}(2i+1, 2j) + T_{l-1}(2i, 2j+1) + T_{l-1}(2i+1, 2j+1)\right)$$

Matrix Representation: If T_{l-1} is vectorized as $\mathbf{t}_{l-1} \in \mathbb{R}^{4m}$ (where m is number of texels at level l), then:

$$\mathbf{t}_l = \mathbf{A} \cdot \mathbf{t}_{l-1}$$

where $\mathbf{A}$ is an $m \times 4m$ averaging matrix with each row containing four $\frac{1}{4}$ entries.

Trilinear Interpolation: Combines bilinear interpolation within a mipmap level with linear interpolation between levels:

$$T_{\text{final}} = (1-\lambda)T_l + \lambda T_{l+1}$$

where λ is the fractional mipmap level.

Example 32: Mipmap Construction and Usage

Given a 4×4 texture (level 0):

$$T_0 = \begin{bmatrix} 100 & 120 & 140 & 160 \\ 110 & 130 & 150 & 170 \\ 90 & 100 & 110 & 120 \\ 80 & 90 & 100 & 110 \end{bmatrix}$$

Step 1: Construct mipmap level 1 (2×2)

$$\begin{aligned} T_1(0,0) &= \frac{1}{4}(100 + 120 + 110 + 130) = \frac{460}{4} = 115 \\ T_1(0,1) &= \frac{1}{4}(140 + 160 + 150 + 170) = \frac{620}{4} = 155 \\ T_1(1,0) &= \frac{1}{4}(90 + 100 + 80 + 90) = \frac{360}{4} = 90 \\ T_1(1,1) &= \frac{1}{4}(110 + 120 + 100 + 110) = \frac{440}{4} = 110 \end{aligned}$$

$$T_1 = \begin{bmatrix} 115 & 155 \\ 90 & 110 \end{bmatrix}$$

Step 2: Construct mipmap level 2 (1×1)

$$T_2(0,0) = \frac{1}{4}(115 + 155 + 90 + 110) = \frac{470}{4} = 117.5$$

Step 3: Trilinear interpolation example Sample at mipmap level $\lambda = 0.3$ (between level 1 and 2).
First, sample T_1 at $(0.3, 0.7)$ using bilinear interpolation:

$$\begin{aligned} T_1(0.3,0.7) &= (1-0.3)(1-0.7)\cdot 115 + 0.3(1-0.7)\cdot 155 \\ &\quad + (1-0.3)\cdot 0.7\cdot 90 + 0.3\cdot 0.7\cdot 110 \\ &= 24.15 + 13.95 + 44.1 + 23.1 \\ &= 105.3 \end{aligned}$$

Then interpolate between levels:

$$T_{\text{final}} = (1-0.3)\cdot 105.3 + 0.3\cdot 117.5 = 73.71 + 35.25 = 108.96$$

4 Simulation and Implementation

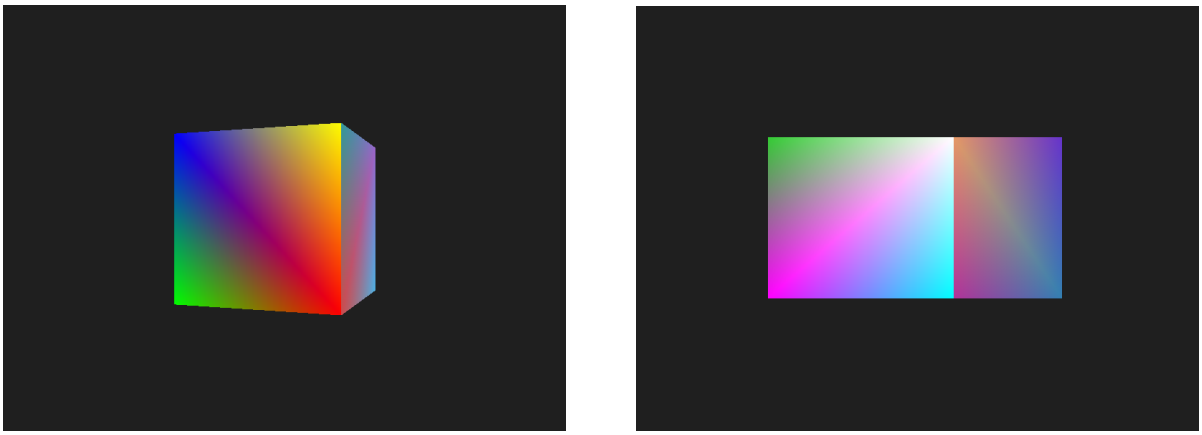
4.1 C++/OpenGL Simulation

```cpp
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/quaternion.hpp>
#include <glm/gtc/type_ptr.hpp>

const char* vertexShaderSource = R"(
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform int useOrthographic;
out vec3 ourColor;
void main()  {
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    ourColor = aColor;
}
)";

const char* fragmentShaderSource = R"(
#version 330 core
in vec3 ourColor;
out vec4 FragColor;
void main()  {
    FragColor = vec4(ourColor, 1.0);
}
)";

// 1. Quaternion rotation using matrix conversion
glm::quat rotateQuaternion(glm::quat q, float angle, glm::vec3 axis) {
    glm::quat rotation = glm::angleAxis(glm::radians(angle), axis);
    return rotation * q;
}

// 2. DIRECT QUATERNION ROTATION
```

```
glm::vec3 rotatePointByQuaternion(const glm::vec3& point, const glm::quat& q) {
    glm::quat p(0.0f, point.x, point.y, point.z);  // Represent point as pure quaternion: p = (point, 0)
    glm::quat q_inv = glm::inverse(q);    // Calculate inverse quaternion
    glm::quat p_rotated = q * p * q_inv;   // Apply rotation
    return glm::vec3(p_rotated.x, p_rotated.y, p_rotated.z);  // Extract vector part
}

// 3. ORTHOGRAPHIC PROJECTION MATRIX
glm::mat4 constructOrthographicMatrixManual(float left, float right, float bottom, float top, float near, float far) {
    glm::mat4 ortho = glm::mat4(1.0f);
    // As shown in Example 27
    ortho[0][0] = 2.0f / (right - left);
    ortho[1][1] = 2.0f / (top - bottom);
    ortho[2][2] = -2.0f / (far - near);
    ortho[3][0] = -(right + left) / (right - left);
    ortho[3][1] = -(top + bottom) / (top - bottom);
    ortho[3][2] = -(far + near) / (far - near);
    return ortho;
}
```

Listing 1: 3D Transformation Pipeline Implementation Sample

Implementation Results:



(a) Perspective Projection

(b) Orthographic Projection

Figure 1: Transformation operations

4.2 C Implementation for Image Files

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#pragma pack(push,1)
typedef struct {
    uint16_t bfType;      // 'BM' = 0x4D42
    uint32_t bfSize;
    uint16_t bfReserved1;
    uint16_t bfReserved2;
    uint32_t bfOffBits;
} BITMAPFILEHEADER;
#pragma pack(pop)

#pragma pack(push,1)
typedef struct {
```

```
    uint32_t biSize;          // size of this header (40)
    int32_t  biWidth;
    int32_t  biHeight;
    uint16_t biPlanes;
    uint16_t biBitCount;
    uint32_t biCompression;
    uint32_t biSizeImage;
    int32_t  biXPelsPerMeter;
    int32_t  biYPelsPerMeter;
    uint32_t biClrUsed;
    uint32_t biClrImportant;
} BITMAPINFOHEADER;
#pragma pack(pop)

// Grayscale transformation matrix
float grayscaleMatrix[3][3] = {
    {0.212655f, 0.715158f, 0.072187f},
    {0.212655f, 0.715158f, 0.072187f},
    {0.212655f, 0.715158f, 0.072187f}
};

// Sepia transformation matrix
float sepiaMatrix[3][3] = {
    {0.3588f, 0.7044f, 0.1368f},
    {0.2990f, 0.5870f, 0.1140f},
    {0.2392f, 0.4696f, 0.0912f}
};

float blurKernel[3][3] = {
    {1.0f / 3.0f, 1.0f / 3.0f, 1.0f / 3.0f},
    {1.0f / 3.0f, 1.0f / 3.0f, 1.0f / 3.0f},
    {1.0f / 3.0f, 1.0f / 3.0f, 1.0f / 3.0f}
};

// Image convolution operation (3x3 kernel)
void applyConvolution(uint8_t* image, int width, int height, float kernel[3][3], float
    factor) {
    if (!image || width <= 2 || height <= 2 || !kernel) return;

    size_t bytes = (size_t)width * (size_t)height * 3;
    uint8_t* temp = (uint8_t*)malloc(bytes);
    if (!temp) return;

    memcpy(temp, image, bytes);

    for (int y = 1; y < height - 1; y++) {
        for (int x = 1; x < width - 1; x++) {
            float sum_r = 0.0f, sum_g = 0.0f, sum_b = 0.0f;

            for (int ky = -1; ky <= 1; ky++) {
                for (int kx = -1; kx <= 1; kx++) {
                    size_t idx = ((size_t)(y + ky) * (size_t)width + (size_t)(x + kx)) * 3;
                    float k = kernel[ky + 1][kx + 1];

                    sum_b += (float)temp[idx] * k;
                    sum_g += (float)temp[idx + 1] * k;
                    sum_r += (float)temp[idx + 2] * k;
                }
            }

            size_t idx = ((size_t)y * (size_t)width + (size_t)x) * 3;
            image[idx] = (uint8_t)fmaxf(0.0f, fminf(255.0f, sum_b * factor));
            image[idx + 1] = (uint8_t)fmaxf(0.0f, fminf(255.0f, sum_g * factor));
            image[idx + 2] = (uint8_t)fmaxf(0.0f, fminf(255.0f, sum_r * factor));
        }
    }
    free(temp);
```

```
}
```

Listing 2: BMP Image Processing Operations Sample

Implementation Results:

(a) Original (b) Grayscale (c) Invert (d) Sepia

(e) Blur (f) Sharpen (g) Ridge (h) BC1

(i) DCT (j) Downscale (k) Upscale

Figure 2: Image processing operations

5 Conclusion

This comprehensive project has demonstrated the central role of linear algebra as the unifying mathematical framework for computer graphics and multimedia systems. Through both theoretical analysis and practical implementation, we have established how:

Core Mathematical Principles

- **Matrix representations** enable efficient image operations, from simple color transformations to sophisticated convolution filters
- **Vector space theory** provides the foundation for audio signal processing, allowing operations like mixing and echo effects to be expressed as linear combinations
- **Linear transformations** form the backbone of 3D graphics pipelines, enabling complex geometric manipulations through simple matrix multiplications
- **Quaternion algebra** solves practical limitations in rotation representation, eliminating gimbal lock while maintaining computational efficiency

Integration of Advanced Techniques

The project successfully bridges multiple domains:

- **Bilinear interpolation** applies linear algebra to texture mapping, ensuring smooth visual transitions in rendered graphics
- **Mipmap construction** demonstrates how hierarchical averaging (a linear operation) optimizes texture sampling in real-time rendering
- **Orthogonal transformations** in DCT compression illustrate the power of basis changes for data reduction

Practical Implementation Outcomes

The C++/OpenGL and C implementations confirm several key insights:

- Linear algebra concepts translate directly into efficient code
- Matrix operations enable hardware acceleration through optimized linear algebra libraries
- The mathematical elegance of transformations simplifies complex visual and audio effects
- Quaternion rotations provide superior animation quality compared to Euler angles

Future Implications

The principles established in this project have far-reaching applications:

- **Real-time rendering systems** will continue to leverage matrix transformations for VR/AR applications
- **Multimedia compression** algorithms will evolve from the orthogonal decomposition principles demonstrated here
- **Machine learning** in graphics increasingly relies on linear algebra for neural network-based rendering
- **Real-time signal processing** builds upon the vector space concepts shown in audio manipulation

6 References

1. Fundamentals of Multimedia, 1st Edition – Ze-Nian Li and Mark S. Drew
2. Fundamentals of Computer Graphics, 3rd Edition – Peter Shirley and Steve Marschner
3. Real-time Rendering, 4th Edition – Tomas Akenine-Möller, Eric Haines, and Naty Hoffman
4. Gonzalez, R. C., & Woods, R. E. (2018). Digital Image Processing. Pearson.