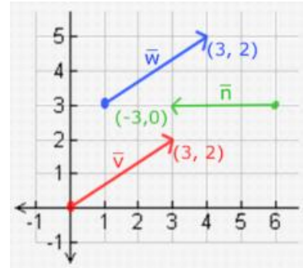
TRANSFORMATIONS



$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Using matrix objects is the best way to perform transformations (translation, rotation, and scaling) on objects.
It might be worth revisiting your linear algebra notes before looking into this section: Matrix Multiplication as Composition

Vectors

A vector is simply a representation of **direction** and **magnitude**.

- If a vector has 2 dimensions, it represents a direction on a plane, and if it has 3 dimensions, it can represent any direction in 3D space.
- Even if the origin of two vectors is different, they can still have the same direction and magnitude (e.g. $\vec{v}$ and $\overline{w}$).

If we want to visualize vectors as positions, we can imagine the origin of the direction vector to be $(0, 0, 0)$ and then point towards a certain direction that specifies the point, making it a **position vector**.

- e.g. The position vector $(3, 5)$ would point to $(3, 5)$ on the graph with an origin of $(0, 0)$.

Scalar Vector Operations

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + x \rightarrow \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} x \\ x \\ x \end{pmatrix} = \begin{pmatrix} 1+x \\ 2+x \\ 3+x \end{pmatrix}$$

A **scalar** is a constant that can be used to modify a vector by adding, subtracting, multiplying, or dividing.

Vector Negation (Inverse Vector)

$$-\bar{v} = -\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} -v_x \\ -v_y \\ -v_z \end{pmatrix}$$

Negating a vector results in a vector in the reversed direction.

- Can also be represented as a scalar-vector multiplication with a scalar value of -1.

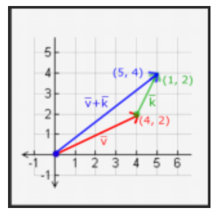
Vector Addition and Subtraction

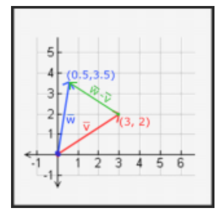
$$\bar{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \bar{k} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \rightarrow \bar{v} + \bar{k} = \begin{pmatrix} 1+4 \\ 2+5 \\ 3+6 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 9 \end{pmatrix}$$

$$\bar{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \bar{k} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \rightarrow \bar{v} + -\bar{k} = \begin{pmatrix} 1+(-4) \\ 2+(-5) \\ 3+(-6) \end{pmatrix} = \begin{pmatrix} -3 \\ -3 \\ -3 \end{pmatrix}$$

Addition of two vectors is defined as **component-wise** addition. That is, each component of one vector is added to the same component of the other vector.
Adding two vectors results in a new vector that points to the position where travelling both vectors would bring you.

- This is easily visualized with the **head-to-tail** method.





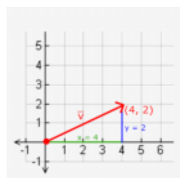
The result
Subtracting two vectors results in a new vector that points from the tail of the first vector to the tail of the second vector.

- Just like normal addition and subtraction, vector subtraction is the same as addition with a negated second vector.

Length

To retrieve the length/magnitude of a vector, we use the **Pythagoras theorem**.

- $|\vec{v}| = \sqrt{x^2 + y^2}$
  - Where |v| is the length of vector $\vec{v}$.
  - Based on the example to the right:
    $|\vec{v}| = \sqrt{4^2 + 2^2} = \sqrt{16 + 4} = \sqrt{20} = 4.47$



A unit vector is a vector whose length is 1.

- A unit vector, $\vec{n}$, can be calculated from any vector by dividing each of the vector's components by its length.
  - $\hat{n} = \dfrac{\vec{v}}{|\vec{v}|}$
  - Also known as **normalizing** a vector.
- Useful when you only care about the direction of a vector.

Vector-Vector Multiplication

Vector multiplication is split up into two different forms of multiplication: dot product and cross product.

Dot Product

$$\begin{pmatrix} 0.6 \\ -0.8 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = (0.6 * 0) + (-0.8 * 1) + (0 * 0) = -0.8$$

The **dot product** of two vectors is equal to the scalar product of their lengths times the cosine of the angle between them.

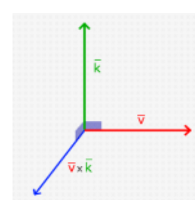
- $\vec{v} \cdot \vec{k} = |\vec{v}| \cdot |\vec{k}| \cdot \cos\theta$
- If $\vec{v}$ and $\vec{k}$ are unit vectors, then the formula is reduced to: $\hat{v} \cdot \hat{k} = 1 \cdot 1 \cdot \cos\theta = \cos\theta$
  - In this case, the dot product only defines the angle between both vectors.
  - Allows us to easily test if the two vectors are orthogonal or parallel to each other.

The dot product is calculated through component-wise multiplication where we add the results together.
Example: If the dot product of two vectors is -0.8, we use the inverse cosine function to calculate the angle (in degrees) between them, like so: $\theta = \cos^{-1}(-0.8) = \sim 143.13°$

The dot product is very useful for doing light calculations.

Cross Product

$$\begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} \times \begin{pmatrix} B_x \\ B_y \\ B_z \end{pmatrix} = \begin{pmatrix} A_y \cdot B_z - A_z \cdot B_y \\ A_z \cdot B_x - A_x \cdot B_z \\ A_x \cdot B_y - A_y \cdot B_x \end{pmatrix}$$



The **cross product** takes two non-parallel vectors as inputs and produces a third vector that is orthogonal to both the input vectors.

- **NOTE:** Cross products are only defined in 3D space.
- If both the input vectors are orthogonal to each other, then the cross product will result in 3 orthogonal vectors, which is useful (for some reason).

Matrices

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

A matrix is a rectangular array of numbers, symbols, and/or mathematical expressions.

- An individual item in a matrix is called an **element** of the matrix.
- Matrices are indexed (m, n).
- The **dimensions** of a matrix are denoted by **M x N**, where M is the number of rows, and N is the number of columns.

Addition and Subtraction

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 2 \\ 1 & 6 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 4-2 & 2-4 \\ 1-0 & 6-1 \end{bmatrix} = \begin{bmatrix} 2 & -2 \\ 1 & 5 \end{bmatrix}$$

Addition and subtraction between two matrices is done on a per-element basis.

- This means that addition and subtraction is only defined for matrices of the same dimensions.

Matrix-Scalar Products

$$2 \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 & 2 \cdot 2 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

A **matrix-scalar product** multiplies each element of the matrix by a scalar.

- A scalar *scales* the elements of a matrix by its value.

Matrix-Scalar Products

A **matrix-scalar product** multiplies each element of the matrix by a scalar.

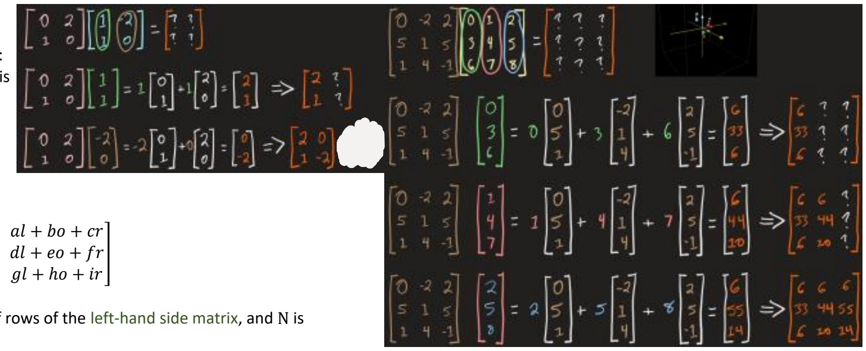
- A scalar *scales* the elements of a matrix by its value.

$$2 \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 & 2 \cdot 2 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

Matrix-Matrix Multiplication

There are two important things to remember when thinking about using matrix multiplication:

1. You can only multiply two matrices if the number of columns on the left-hand side matrix is equal to the number of rows on the right-hand side matrix.
2. Matrix multiplication is not commutative. This is, $AB \neq BA$.



2D matrix multiplication formula: $\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$

3D matrix multiplication formula: $\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} aj+bm+cp & ak+bn+cq & al+bo+cr \\ dj+em+fp & dk+en+fq & dl+eo+fr \\ gj+hm+ip & gk+hn+iq & gl+ho+ir \end{bmatrix}$

The resulting matrix of matrix multiplication has dimensions $\mathbf{M \times N}$, where M is the number of rows of the left-hand side matrix, and N is the number of columns of the right-hand side matrix.

Matrix-Vector Multiplication

In OpenGL, we usually work with 4x4 matrices. One of the reasons for that is because most of the vectors are of size 4.

The most simple transformation matrix is the **identity matrix**.

- The identity matrix is an NxN matrix with only 0s except in its diagonal.
- Vector multiplication with an identity matrix has no effect on the vector.
- The identity matrix is usually used as a starting point for generating other transformation matrices.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 \\ 1 \cdot 2 \\ 1 \cdot 3 \\ 1 \cdot 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Scaling

Since we're working in either 2 or 3 dimensions, we can define a scaling vector of 2 or 3 scaling variables, each scaling one axis (x, y, or z).

- e.g. We can scale a vector $\vec{v} = (3, 2)$ by $(0.5, 2)$ and the resulting vector would be $\vec{v} = (3, 2) * (0.5, 2) = (1.5, 4)$.
  - This is known as a **non-uniform scale**, because the scaling factor is not the same for each axis.
  - If the scalar used was the same for all axes, the scaling performed would be a **uniform scale**.
- Scaling matrix on a vector (x,y,z): $\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 * x \\ S_2 * y \\ S_3 * z \\ 1 \end{pmatrix}$
  - **NOTE:** We keep the 4th scaling value as 1. The w component is used for other purposes.

Translation

Translation is the process of adding another vector on top of the original vector to return a new vector with a different position, thus *moving* the vector based on a translation vector.

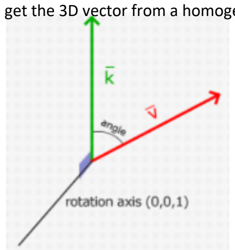
- Translation matrix on a vector (x,y,z): $\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$
  - **NOTE:** Because the w component of the vector is 1, we can add the translation values to the vector, and thus translate the vector. *We would not have been able to accomplish this with a 3x3 matrix.*

The w component of a vector is known as a homogeneous coordinate. To get the 3D vector from a homogeneous vector, you divide x, y, and z by its w coordinate.

Rotation

A rotation in 2D space is specified with just an **angle**.

- A 360 degree rotation is equivalent to 2π radians.
  - Most rotation functions require an angle in radians.
    - Degrees to Radians: d * (π / 180)
    - Radians to Degrees: r * (180 / π)
- In the image to the right, $\vec{v}$ is rotated 72 degrees from $\vec{k}$.



A rotations in 3D space is specified with an angle and a **rotation axis**.

- The angle specified rotates the object along the rotation axis.
- Rotation matrix around the x-axis on a vector (x,y,z): $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos\theta * y - \sin\theta * z \\ \sin\theta * y + \cos\theta * z \\ 1 \end{pmatrix}$
- Rotation matrix around the y-axis on a vector (x,y,z): $\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta * x + \sin\theta * z \\ y \\ -\sin\theta * x + \cos\theta * z \\ 1 \end{pmatrix}$
- Rotation matrix around the z-axis on a vector (x,y,z): $\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta * x - \sin\theta * y \\ \sin\theta * x + \cos\theta * y \\ z \\ 1 \end{pmatrix}$

To rotate around an arbitrary 3D axis, we can create a new transformation matrix by first rotating around the x-axis, then y-axis, then z-axis.

- This introduces the problem of **Gimbal Lock**, which is the loss of one degree of freedom in a 3D, three-gimbal mechanism that occurs when the axes of two of the three gimbals are driven into a parallel configuration, "locking" the system into a rotation in a degenerate 2D space.
  - Basically, the object can end up rotating incorrectly if all three axes are rotated upon simultaneously, in certain situations.

A better solution is to rotate around an arbitrary unit axis, e.g. (0.662, 0.2, 0.722) right away instead of combining the rotation matrices.

- Rotation matrix around an arbitrary axis (R_x, R_y, R_z): $\begin{bmatrix} \cos\theta + R_x^2(1-\cos\theta) & R_xR_y(1-\cos\theta) - R_z\sin\theta & R_xR_z(1-\cos\theta) + R_y\sin\theta & 0 \\ R_yR_x(1-\cos\theta) + R_z\sin\theta & \cos\theta + R_y^2(1-\cos\theta) & R_yR_z(1-\cos\theta) - R_x\sin\theta & 0 \\ R_zR_x(1-\cos\theta) - R_y\sin\theta & R_zR_y(1-\cos\theta) + R_x\sin\theta & \cos\theta + R_z^2(1-\cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

- **NOTE:** This rotation matrix does not *fully* prevent gimbal lock, but it makes it a lot less common. To truly prevent gimbal locks, we have to represent rotations using **quaternions**, which are not only safer, but also more computationally friendly.

Combining Matrices

You can combine multiple transformations into a single matrix using matrix-matrix multiplication.

$$Trans.Scale = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 2x+1 \\ 2y+2 \\ 2z+3 \\ 1 \end{bmatrix}$$

In the example above, we first do a scale transformation, and then a translation.

- **NOTE:** When multiplying matrices, we read right-to-left. Meaning, the right-most transformation is applied first and the left-most transformation is applied last.

It is advised to do transformations in the following order:

1. Scaling
2. Rotations
3. Translations

Performing transformations in a different order than the above might cause the transformations to negatively affect each other.

- Example: If you first perform a translation, and then a scale, the scaling matrix would also scale the translation.

In Practice

OpenGL does not have any form of matrix or vector knowledge built in, so we have to define our own mathematics classes and functions. Instead, we'll use the GLM library.

GLM

OpenGL Mathematics (GLM) is a header-only library with easy-to-use and tailored-for-OpenGL mathematics operations.

- [GLM Download Link]
- Copy the root directory of the header files into your includes folder.

Create a vector (1, 0, 0) and translate it (1, 1, 0) using a transformation matrix.

- [Translating a Vector]

Scale the container object down by 50% and rotate it 90 degrees counter-clockwise.

- [Scaling and Rotating an Object]

Rotate the container object over time and move it to the bottom-right of the window.

- [Rotating an Object Over Time]

EXERCISES

1. Using the last transformation on the container, try switching the order around by first rotating and then translating. See what happens and try to reason why this happens.
2. Try drawing a second container with another call to glDrawElements, but place it at a different position using transformations <u>only</u>. Make sure this second container is placed at the top-left of the window and instead of rotating, scale it over time (using the sin function is useful here; note that sin will cause the object to invert as soon as a negative scale is applied).

Translating a Vector

Tuesday, March 22, 2022 10:30 PM

```
// Constructs a vector of (1, 0, 0)
glm::vec4 vec = { 1.0f, 0.0f, 0.0f, 1.0f };
std::cout << vec.x << vec.y << vec.z << std::endl;
// Constructs an identity matrix
glm::mat4 trans = glm::mat4(1.0f);
// Sets up the translation matrix for a (1, 1, 0) translation
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
// Translates the vector using the translation matrix
vec = trans * vec;
std::cout << vec.x << vec.y << vec.z << std::endl;
```

Scaling and Rotating an Object

Tuesday, March 22, 2022 10:52 PM

Add a uniform to your vertex shader for the transformation matrix and apply it to the vertex position.

```
uniform mat4 transform;

void main() {
   gl_Position = transform * vec4(aPos, 1.0);
   vertexColor = aColor;
   texCoord = aTexCoord;
}
```

Construct the transformation matrix and send it to the uniform.

```
// Constructs an identity matrix
glm::mat4 trans = glm::mat4(1.0f);
// Creates the transformation matrix for a 90 degree rotation around the z-axis
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0f, 0.0f, 1.0f));
// Creates the tranformation matrix for a 0.5x uniform scale, which then rotates 90 degrees around the z-axis
trans = glm::scale(trans, glm::vec3(0.5f, 0.5f, 0.5f));

// Sends the transformation matrix data to the transform uniform
glUniformMatrix4fv(glGetUniformLocation(shaderProgram.id, "transform"), 1, GL_FALSE, glm::value_ptr(trans));
```

For glUniformMatrix4fv, the 3rd parameter, *transpose*, asks us if we want to swap the columns and rows.

- OpenGL developers often use an internal matrix layout called **column-major ordering**, which is the default matrix layout in GLM so there is no need to transpose the matrices.

For glUniformMatrix4fv, the 4th parameter, *value*, is the matrix data.

- GLM stores their matrices' data in a way that doesn't always match OpenGL's expectations, so we must convert the data with GLM's built-in function, value_ptr.

Rotating an Object Over Time

Tuesday, March 22, 2022 11:17 PM

Since the object is being rotated over time, we have to put (at least) the rotation transformation and the uniform setter function in the render loop.

```
// Constructs an identity matrix
glm::mat4 trans = glm::mat4(1.0f);
// Creates the transformation matrix for a translation to the bottom-right of the window
trans = glm::translate(trans, glm::vec3(0.5f, -0.5f, 0.0f));
// Creates the tranformation matrix for a counter-clockwise rotation over time, which is then translated to the
// bottom-right of the window
trans = glm::rotate(trans, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));

// Sends the transformation matrix data to shader
glUniformMatrix4fv(glGetUniformLocation(shaderProgram.id, "transform"), 1, GL_FALSE, glm::value_ptr(trans));
```