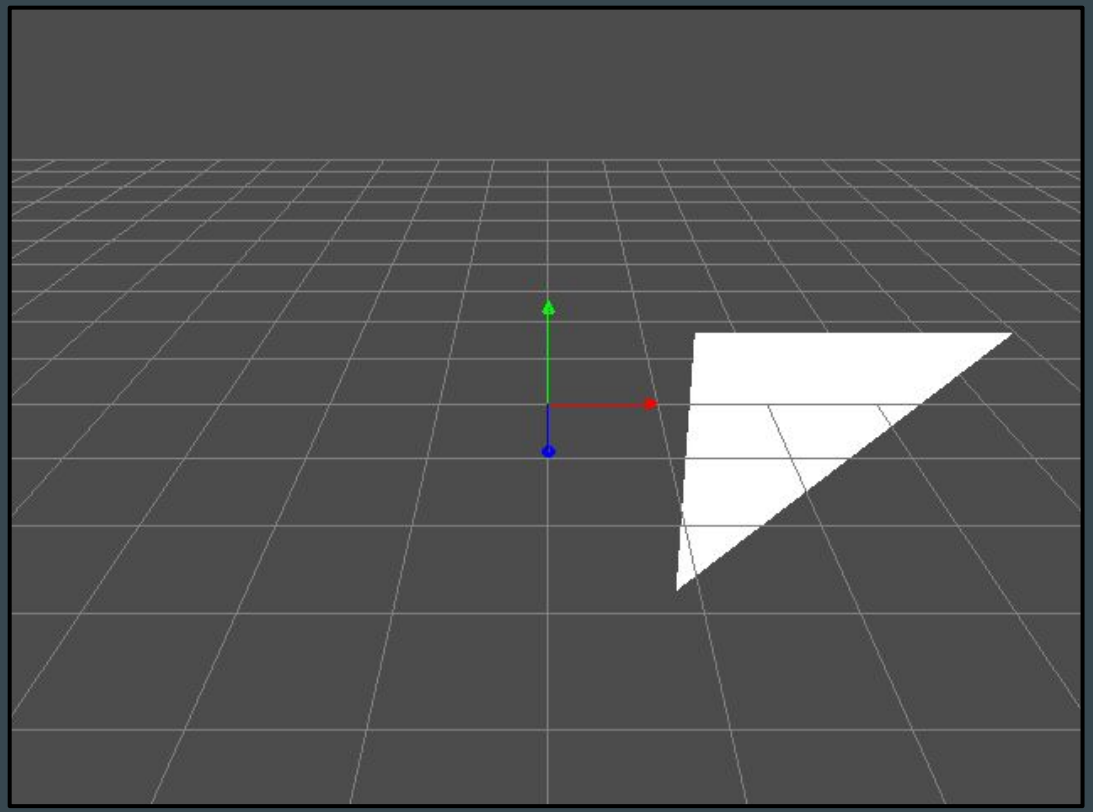
Transformations

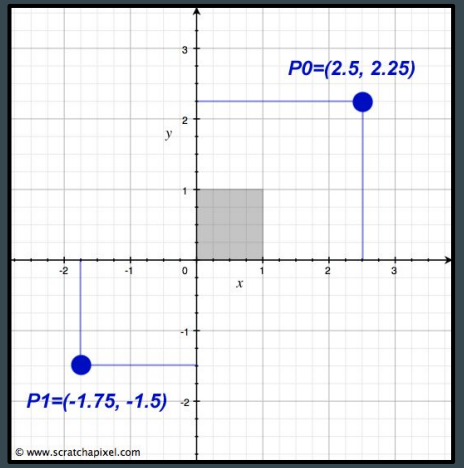
Carlos Fuentes



End Goal

Coordinate systems: definition

- We used to represent points as **tuples of numbers**
- For example (2.5, 2.25) or (-1.75, -1.5) tuples are points in 2 dimensions.
- This tuple have a meaning because of coordinates systems
- This tuple must be **the only** way to represent these point in that coordinate system.

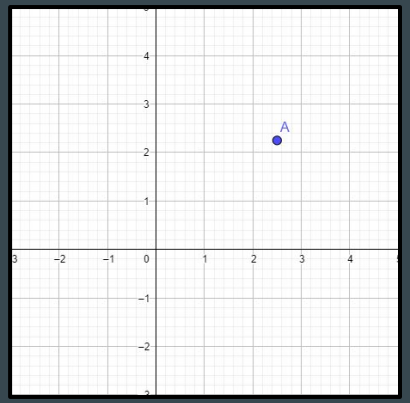


Coordinate systems: basis

- In linear algebra 2 **axis** (in 2D) form what we call **basis** of a coordinate system (3 axis in 3D)
- A basis is a set of **linearly independent vectors** that in a linear combination can represent any point in the coordinate system.
- Linearly independent means none of the vectors can be written as a **linear combination** of the others

Coordinate systems: basis

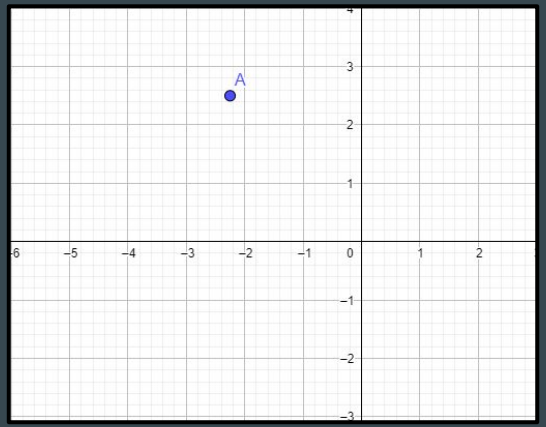
- Orthogonal (**perpendicular**) vectors are guaranteed to be **linearly independent** (inverse is not always true)
- 2D coordinate system (also known as **cartesian** coordinate system) has basis (1, 0) and (0, 1) → X_{axis} and Y_{axis} respectively.
- If we linearly combine it with tuple (2.5, 2.25) we get



$$X_{axis} * 2.5 + Y_{axis} * 2.25 = (1, 0) * 2.5 + (0, 1) * 2.25 = (2.5, 0) + (0, 2.25) = (2.5, 2.25)$$

Coordinate systems: basis

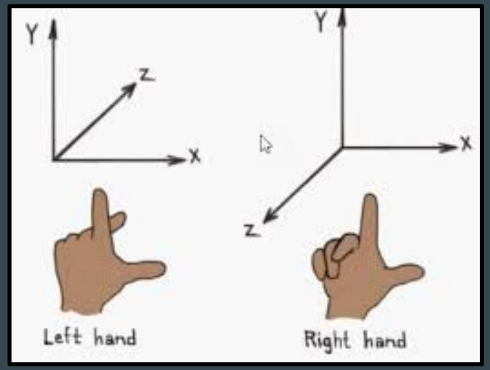
- **Changing the basis** is a common operation in computer graphics
- For example we can rotate 90 degrees both axis to get the basis axis (0, 1) and (-1, 0), so our previous example is this cartesian point



$$X_{axis} * 2.5 + Y_{axis} * 2.25 = (0, 1) * 2.5 + (-1, 0) * 2.25 = (0, 2.5) + (-2.25, 0) = (-2.25, 2.5)$$

Coordinate Systems: 3D

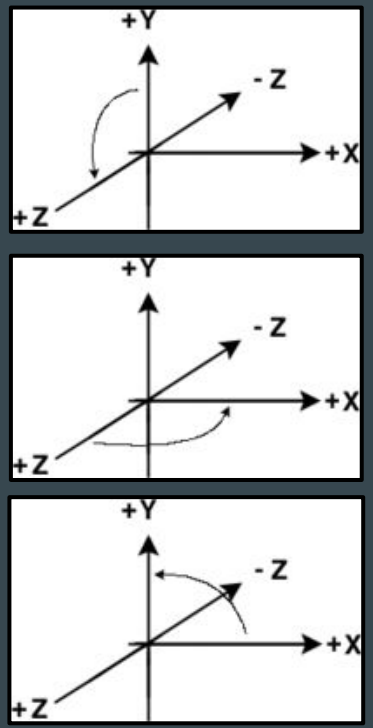
- 3 dimensions is an extension of 2 dimensions but adding a third orthogonal axis
- In Geometry a 3D coordinate system is also known as **Euclidean space**
- Unfortunately we have two ways of generate this new orthogonal axis that derives in 2 different euclidean space conventions: **right handed and left handed**.



Coordinate Systems: 3D

- Handness is also important for **rotation** and **cross product** interpretation
- Right-handed systems rotations are **counterclockwise** and left-handed systems rotations are **clockwise**.
- **OpenGL** is right-handed. Positive x and y axis points to right and up and negative z points forward

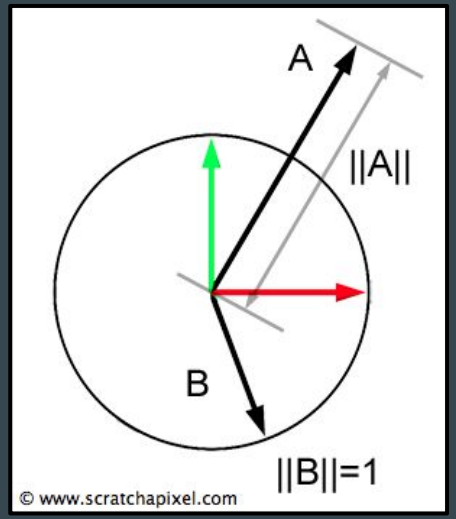
OpenGL positive x, y, z rotations



Vectors

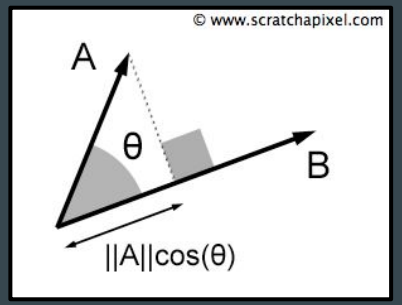
- Vectors represents **points** but also the **direction** from point A to B and also the distance (also called **norm** or **magnitude**).
- A normalized vector is a vector whose magnitude is 1. Is constructed by dividing the vector from its length if it's greater than 0:

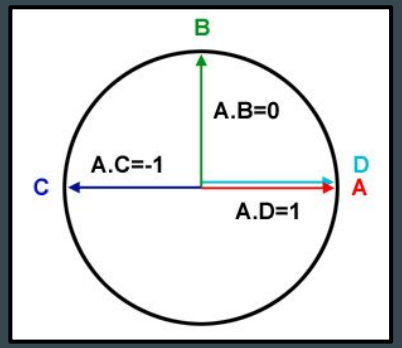
$$\frac{V}{\|V\|}$$



Vectors: dot product

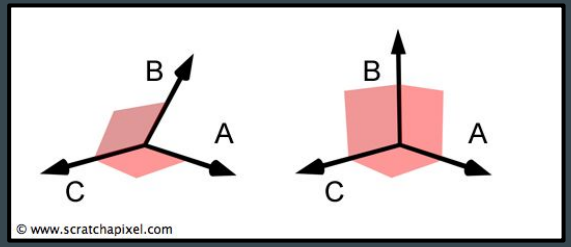
- The **dot product** between two vectors can be seen as the projection of one onto the other.
- Dot product is related to the **cosine** of the angle of both vectors ⇒ is used to compute **angle**
- Is used also to test whether both vectors are pointing to the **same,** the **opposite direction**, or are **orthogonals**.

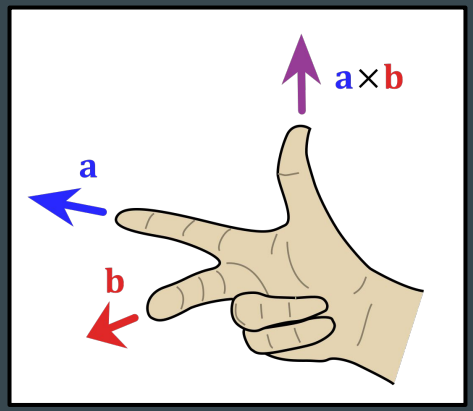




Vectors: cross product

- Cross product returns a vector that is perpendicular to the other two.
- This is useful for creating a coordinate system
- Cross product is anticommutative AxB = -BxA
- Cross product returns always the same value
- When using it to find the third basis axis we have to take care of **handedness** and **cross product sign**





Matrices

- Matrices are used to **transform points** by multiplying vectors by them
- Matrices can be used to combine three basic **geometric transformations** (rotation, translation and scale) into a very fast, easy and compact way using matrix **product**.
- Matrix product is **not commutative** $M_1 \times M_2 \neq M_2 \times M_1$
- In graphics we use square matrices (have the same number of rows and columns), and particularly **3x3 and 4x4 matrices**

Matrices: transformations

- A **matrix and point product** is the same as a matrix and matrix product because we can write a vector like a 1xn matrix (1x3 for a 3D point)
- Matrix point product is not commutative and transforms depends on left or right point multiplication (we will consider **right multiplication**)
- A point multiplying by **Identity** matrix remains unmodified

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} = \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix}$$

Matrices: transformations

- Multiplying the **Scaling** matrix by a point returns the point scaled by the factors.

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} = \begin{pmatrix} P_x * S_x \\ P_y * S_y \\ P_z * S_z \end{pmatrix}$$

- **Rotation** matrices around x, y and z axis for a given angle θ

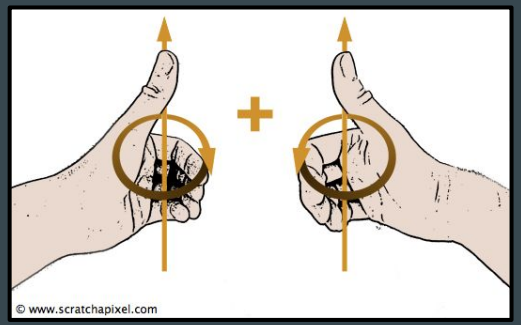
$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{pmatrix} \quad R_y(\theta) = \begin{pmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{pmatrix} \quad R_z(\theta) = \begin{pmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

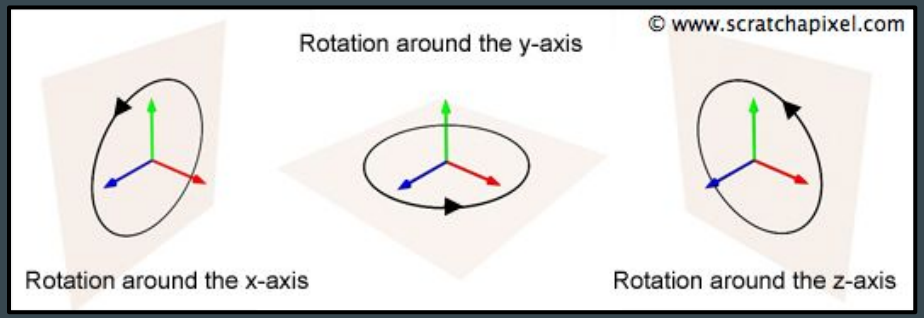
Matrices: transformations

- Rotations around axis can be combined into one matrix

$$R_{xy}(\theta) = R_x(\theta) * R_y(\theta)$$

- The direction of rotation depends on handedness

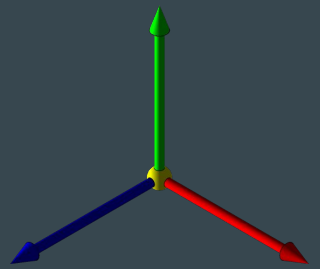




Matrices: transformations

- Rotation matrices, each column represents the **axis of a coordinate system**.

$$\begin{pmatrix} Xx & Yx & Zx \\ Xy & Yy & Zy \\ Xz & Yz & Zz \end{pmatrix}$$



- Rotation matrices are **orthogonal** → each column is a perpendicular vector.
- The inverse of orthogonal matrix is equal to **transpose** (set columns as rows)
- Transpose is a lot more cheap operation than inverse.

Matrices: transformations

- **Translation** is just adding and x,y,z value to a point
- This operation can't be done using 3x3 matrix multiplications
- Translation can be done using **4x4 matrix** and converting our points to

  homogeneous coordinates (x,y,z,w) where w = 1 $\Rightarrow$ (P_x, P_y, P_z, 1)

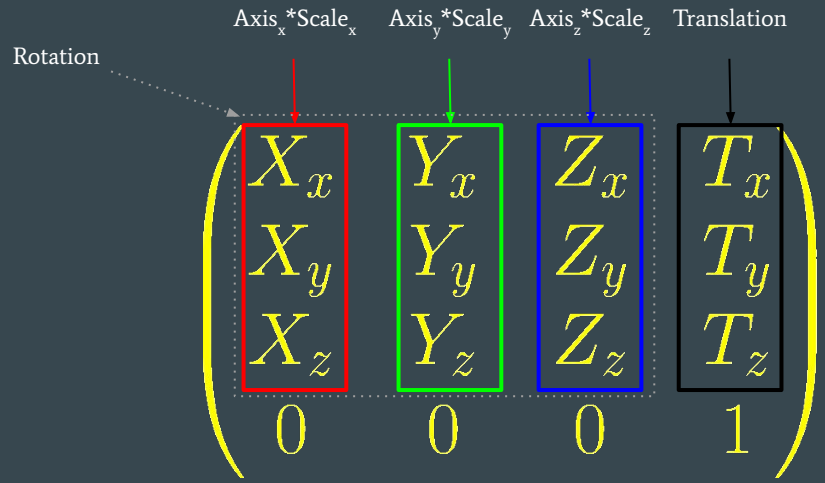
$$\begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix} = \begin{pmatrix} P_x + T_x \\ P_y + T_y \\ P_z + T_z \\ 1 \end{pmatrix}$$

Matrices: transformations

- Homogeneous coordinates is also important for **perspective projection** (allows depth scaling of objects modifying the w component)
- All previous transformations (scale, rotation, translation) are **affine transformations**
- **Affine transformations** are transformations that preserve parallel lines.
- Projective transformations like perspective aren't necessary affine transforms

Model matrix

- **model matrix** is an affine transform result of compositing translation, rotation and scale $\Rightarrow model_{4x4} = T_{x,y,z} * R_{x,y,z}(\theta_1, \theta_2, \theta_3) * S_{x,y,z}$
- Its resulting matrix has this form



Model matrix

- Vectors are used for represent **points or directions**.
- Directions can be rotated or scaled but can't be translated.
- When we use model matrix to **transform directions** we must extract 3x3 rotation matrix from our 4x4 model matrix and apply the transformation.

$$\begin{pmatrix} X_x & Y_x & Z_x \\ X_y & Y_y & Z_y \\ X_z & Y_z & Z_z \end{pmatrix}$$

Model matrix

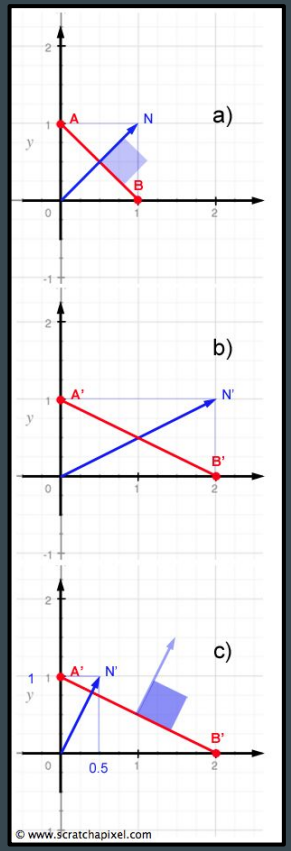
- Vectors usually represent not just directions but also **normals**
- When we apply model matrix to a normal with uniform scale the result normal is scaled ⇒ Re-normalize !!!
- A normal has the property of being **perpendicular to the tangent** of a surface
- Normal resulting of applying model matrix with **non-uniform scale** is not perpendicular to surface tangent ⇒ Wrong result!!!!

Model matrix

- To preserve **normal-tangent perpendicularity**, when using a model matrix with **non-uniform scale**, we have to apply the transpose of the inverse of our 3x3 rotation/scale matrix.

$$transpose(inverse(model_{3x3})) * normal$$

- Resulting matrix is the same model matrix but **inverse scaled**.
- Here a good demonstration



Camera matrix

- Camera matrix is like **model matrix** but without scales ⇒ **Rigid transform**.
- OpenGL camera points towards **Z negative** axis so camera matrix has this form

$$\begin{pmatrix} right_x & up_x & -forward_x & position_x \\ right_y & up_y & -forward_y & position_y \\ right_z & up_z & -forward_z & position_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Camera matrix is usually constructed from **LookAt function** using eye (camera position), target (target position), and up (up vector) parameters:

$$forward = normalized(target - eye)$$
$$right = normalized(cross(forward, up));$$
$$up = normalized(cross(right, forward));$$
$$position = eye;$$

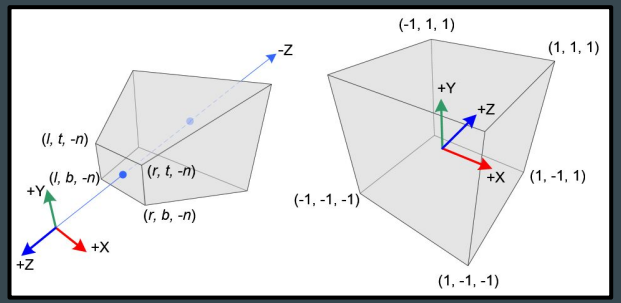
View matrix

- View matrix transforms points from world space relative to camera/view space
- View matrix is the **inverse of camera matrix**
- Camera matrix is the composition of a rotation matrix (**orthonormal**) and a translation matrix so, its inverse is

$$\begin{pmatrix} right_x & right_y & right_z & 0 \\ up_x & up_y & up_z & 0 \\ -forward_x & -forward_y & -forward_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & -T_x \\ 0 & 1 & 0 & -T_y \\ 0 & 0 & 1 & -T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} right_x & right_y & right_z & -dot(T_{x,y,z}, right_{x,y,z}) \\ up_x & up_y & up_z & -dot(T_{x,y,z}, up_{x,y,z}) \\ -forward_x & -forward_y & -forward_z & dot(T_{x,y,z}, forward_{x,y,z}) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

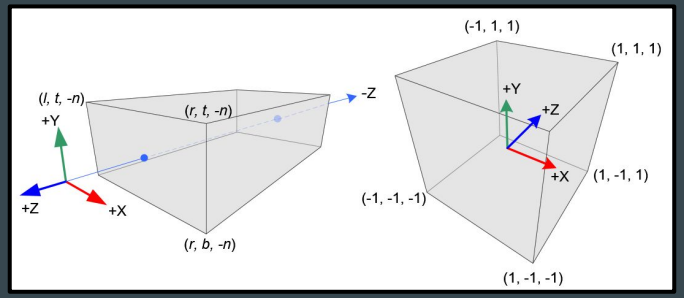
Projection matrix: perspective

- Perspective projection maps a point in a **pyramid to a unit cube.**
- Due to this mapping **farther objects appears smaller**.
- This is accomplished modifying w component of homogeneous coordinates.
- Projection transforms points to **Clipping Space**, this points are then divided by w (perspective corrected) to get **Normalized Device Coordinates**.



Projection matrix: orthographic

- Orthographic projection is simpler than perspective
- It just needs to map a rectangular volume into a cube scaling it.
- With orthographic projection there is no Z scaling of objects
- Here an article deriving both projections



Matrices: row major vs column major

- Matrices can be stored in memory in 2 different memory layouts:
  - **Row major** means in memory we have [Row_0 Row_1 Row_2 Row_3 ....]
  - **Column major** means in memory we have [$Column_0$ $Column_1$ $Column_2$ $Column_3$ ...]
- **GLSL** matrix are column major
- **MathGeoLib** is row major
- **Transpose** matrix is equivalent to change from row to column major.

Using transforms in shaders: Uniforms

- **Vertex shader** is responsible of transform vertices to clipping space using **model, view and projection** transforms.
- We need a way to publish this transforms as variables of the shader: **Uniforms**
- Uniforms are global variables assigned to a program (vertex and fragment shader)
- Uniform values **remains equal for all invocations** of shaders inside a **draw call**
    - For each vertex, vertex shader is invoked with different vertex attributes but the same uniforms
    - For each fragment, fragment shader is invoked with different fragments but the shame uniforms

Using transforms in shaders: Uniforms

- How to declare a uniform into our vertex shader

```
#version 330
layout(location=0) in vec3 my_vertex_position;

uniform mat4 proj;
uniform mat4 view;
uniform mat4 model;

void main()
{
    gl_Position = proj*view*model*vec4(my_vertex_position, 1.0);
}
```

Using transforms in shaders: Uniforms

- How to set Uniform values for each draw call:
  - glGetUniformLocation retrieves an index location for a uniform variable name
  - glUniform * (glUniform1f, glUniform2f, ..., glUniformMatrix4f, etc) assigns a value

```
void RenderTriangle()
{
    float4x4 model, view, projection;
    // TODO: retrieve model view and projection

    glUseProgram(program);
    glUniformMatrix4fv(glGetUniformLocation(program, “model”), 1, GL_TRUE, &model[0][0])
    glUniformMatrix4fv(glGetUniformLocation(program, “view”), 1, GL_TRUE, &view[0][0])
    glUniformMatrix4fv(glGetUniformLocation(program, “proj”), 1, GL_TRUE, &projection[0][0])
    // TODO: bind buffer and vertex attributes

    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

Exercise

- Modify our triangle rendering example so we can multiply each vertex by model and view and projection matrices in our vertex shader
- Use Frustum class in MathGeoLib to get projection matrices

```
Frustum frustum;
frustum.type = FrustumType::PerspectiveFrustum;

frustum.pos = float3::zero;
frustum.front = -float3::unitZ;
frustum.up = float3::unitY;

frustum.nearPlaneDistance = 0.1f;
frustum.farPlaneDistance = 100.0f;
frustum.verticalFov = math::pi/4.0f;
frustum.horizontalFov = 2.f * atanf(tanf(frustum.verticalFov * 0.5f) *aspect;

float4x4 proj = frustum.ProjectionMatrix();
```

Exercise

- Use float4x4 class in MathGeoLib to get model and view matrix
- Note: For view matrix you can create your own LookAt function or use MathGeoLib LookAt.

```
float4x4 model =  float4x4::FromTRS(float3(2.0f, 0.0f, 0.0f),
                                    float4x4::RotateZ(pi / 4.0f),
                                    float3(2.0f, 1.0f, 0.0f));


float4x4 view  = LookAt(float3(0.0f, 4.0f, 8.0f), float3(0.0f, 0.0f, 0.0f), float3::unitY);
```

Debug Draw

- In debug_draw.zip you can find debug primitives for drawing axis, grids, etc. To integrate into your project:
  a. Add source and header files to your project
  b. Create **ModuleDebugDraw** in Application class
  c. Call drawing primitives each render
     - **dd::axisTriad**(float4x4::identity, 0.1f, 1.0f);
     - **dd::xzSquareGrid**(-10, 10, 0.0f, 1.0f, dd::colors::Gray);
  d. Call **ModuleDebugDraw::Draw** method passing projection, view, and screen width and height as arguments
  e. Note: Draw method must be called after drawing all primitives