

Week1

Advanced Graphics Programming

Hooman Salamat

Assessment

2x Assignments 30%



(8/10) Labs 20%



1x Midterm Exam 20%



1x Final Project 30%



Ground Rules

Make sure your phone is on silent.



Break after 50 mins of every hour.



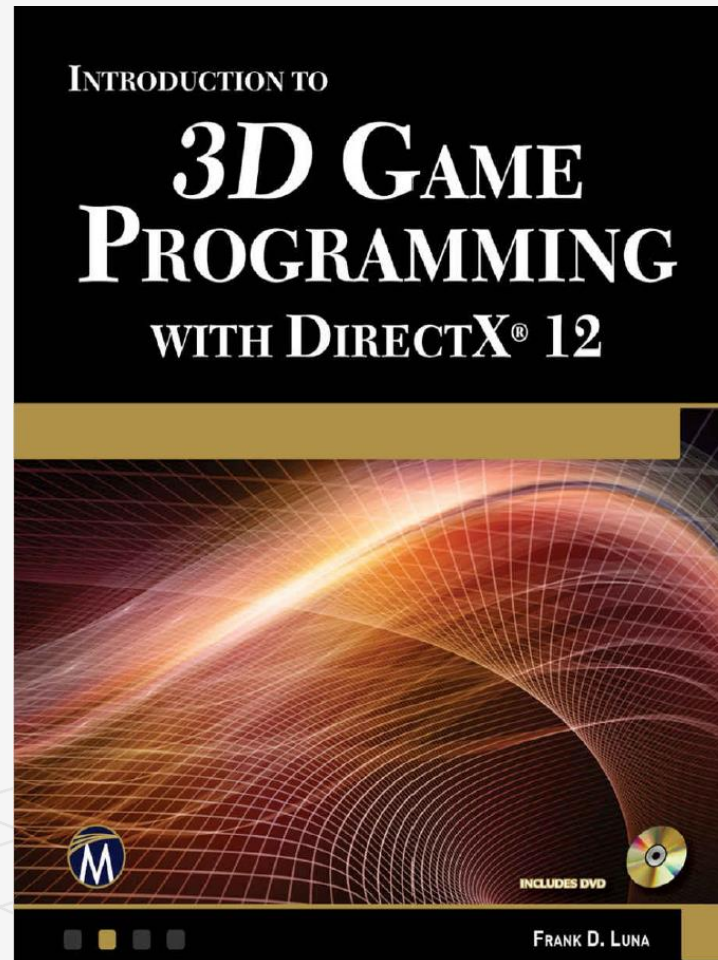
Using your laptop for taking notes is ok.



You can stop me anytime.



Textbook



DirectX Programming

Course Materials: <https://github.com/hsalamat/DirectX/>

DirectX 12 Programming Guide: <https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121%28v=vs.85%29.aspx>

Complete DirectX Framework: [https://msdn.microsoft.com/en-us/library/windows/desktop/dn643742\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn643742(v=vs.85).aspx)

Direct3D 12 sample programs that are available online: <https://github.com/Microsoft/DirectX-Graphics-Samples>

HLSL Programming Guide: <https://docs.microsoft.com/en-us/windows/win32/direct3dhls/dx-graphics-hls>

SharpDX: <http://sharpx.org/>

Book Source code: <https://github.com/d3dcoder/d3d12book>

Shader Playground: <http://shader-playground.timjones.io/>

<https://walbourn.github.io/getting-started-with-direct3d-12/>

Directx12 visual studio templates: <https://github.com/walbourn/directx-vs-templates>

<https://github.com/discosultan/dx12-game-programming>

Windows Programming

DirectX 12 Requirements:

- Windows 10
- Visual Studio 2015 or later
- A graphics card that supports Direct3D 12, e.g. Geforce GTX 760

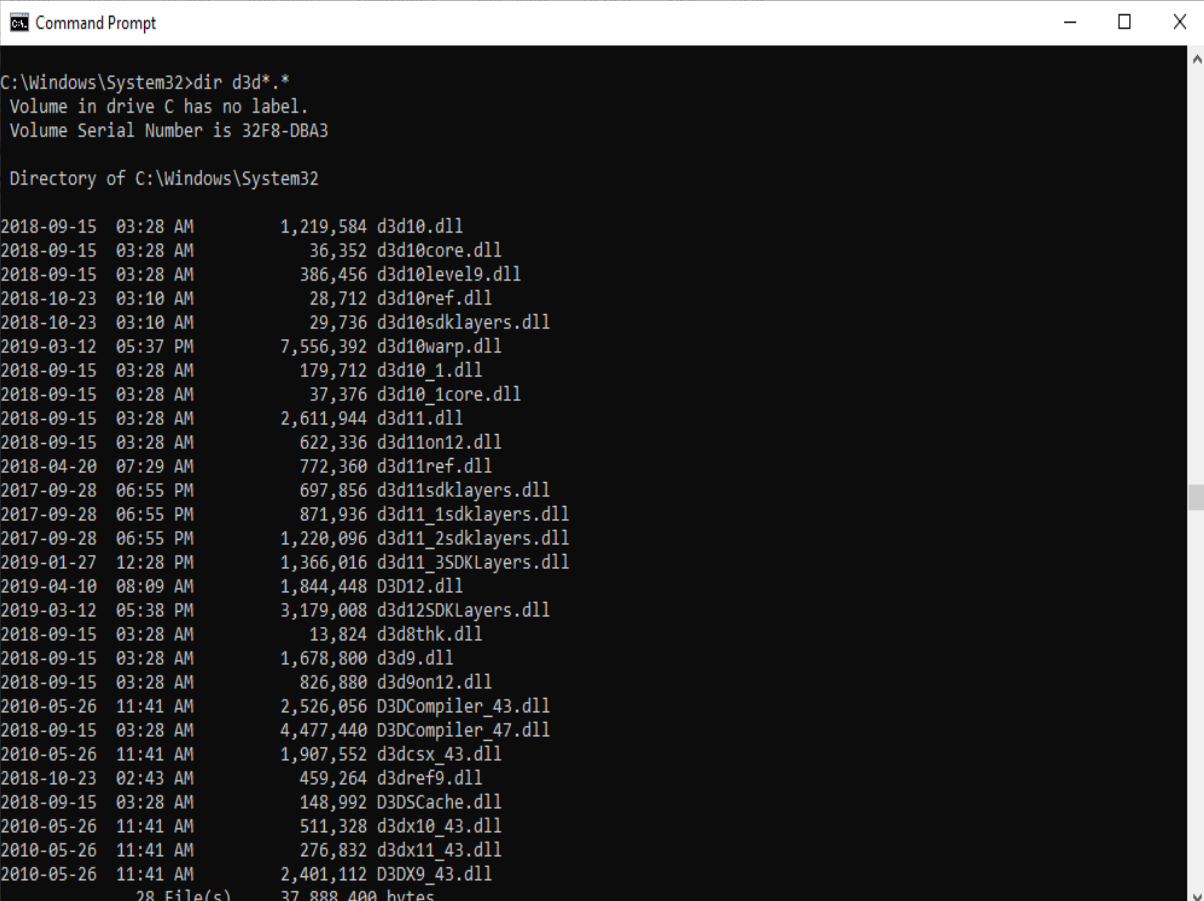
. D3d12.dll is under system32 directory or C:\Windows\SysWOW64

To use the [Direct3D 12 API](#), include D3d12.h and link to D3d12.lib, or query the entry points directly in D3d12.dll.

D3d12.h is under: C:\Program Files (x86)\Windows Kits\10\Include\10.0.17763.0\um

D3d12.lib is under C:\Program Files (x86)\Windows Kits\10\Lib\10.0.17763.0\um\x86

Note: your windows version might be different!



```
Command Prompt

C:\Windows\System32>dir d3d*.*
Volume in drive C has no label.
Volume Serial Number is 32F8-DBA3

Directory of C:\Windows\System32

2018-09-15  03:28 AM           1,219,584 d3d10.dll
2018-09-15  03:28 AM           36,352 d3d10core.dll
2018-09-15  03:28 AM          386,456 d3d10level9.dll
2018-10-23  03:10 AM           28,712 d3d10ref.dll
2018-10-23  03:10 AM           29,736 d3d10sdklayers.dll
2019-03-12  05:37 PM          7,556,392 d3d10warp.dll
2018-09-15  03:28 AM          179,712 d3d10_1.dll
2018-09-15  03:28 AM           37,376 d3d10_1core.dll
2018-09-15  03:28 AM          2,611,944 d3d11.dll
2018-09-15  03:28 AM           622,336 d3d11on12.dll
2018-04-20  07:29 AM           772,360 d3d11ref.dll
2017-09-28  06:55 PM          697,856 d3d11sdklayers.dll
2017-09-28  06:55 PM          871,936 d3d11_1sdklayers.dll
2017-09-28  06:55 PM          1,220,096 d3d11_2sdklayers.dll
2019-01-27  12:28 PM          1,366,016 d3d11_3SDKLayers.dll
2019-04-10  08:09 AM          1,844,448 D3D12.dll
2019-03-12  05:38 PM          3,179,008 d3d12SDKLayers.dll
2018-09-15  03:28 AM           13,824 d3d8thk.dll
2018-09-15  03:28 AM          1,678,800 d3d9.dll
2018-09-15  03:28 AM           826,880 d3d9on12.dll
2010-05-26  11:41 AM          2,526,056 D3DCompiler_43.dll
2018-09-15  03:28 AM          4,477,440 D3DCompiler_47.dll
2010-05-26  11:41 AM          1,907,552 d3dcsx_43.dll
2018-10-23  02:43 AM           459,264 d3dref9.dll
2018-09-15  03:28 AM           148,992 D3DSCache.dll
2010-05-26  11:41 AM           511,328 d3dx10_43.dll
2010-05-26  11:41 AM           276,832 d3dx11_43.dll
2010-05-26  11:41 AM          2,401,112 D3DX9_43.dll
                28 File(s)          37,888,400 bytes
```

D3d12.h

<https://docs.microsoft.com/en-us/windows/win32/api/d3d12/>

Interfaces

Functions

Structures

Enumerations

C++ is the only supported language for Direct3D 12 development, C# and other .NET languages are not supported.



D3dx12.h and DirectX Libraries

There are a number of helper structures that, in particular, make it easy to initialize a number of the D3D12 structures. These structures, and some utility functions, are in the header D3dx12.h.

Libraries

Library	Purpose	Documentation
DirectX Tool Kit for DirectX 12	A substantial collection of helper classes for writing Direct3D 12 C++ code for Universal Windows Platform (UWP) apps, Win32 desktop applications for Windows 10, and Xbox One exclusive apps.	DirectX12TK wiki
DirectXTex	Use this for reading and writing DDS files, and performing various texture content processing operations including resizing, format conversion, mip-map generation, block compression for Direct3D runtime texture resources, and height-map to normal-map conversion.	DirectXTex wiki
DirectXMesh	Use this for performing various geometry content processing operations including generating normals and tangent frames, triangle adjacency computations, and vertex cache optimization.	DirectXMesh wiki
DirectXMath	A large number of helper classes and methods to support vectors, scalars, matrices, quaternions, and many other mathematical operations.	DirectXMath documentation at MSDN
UVAtlas	Use this for creating and packing an isochart texture atlas.	UVAtlas wiki

Debug Layer

The header required to support the debugging layer, `D3D12SDKLayers.h`, is included by default from `d3d12.h`.

The debug layer provides warnings for many issues. For example:

- Forgot to set a texture but read from it in your pixel shader.

- Output depth but have no depth-stencil state bound.

- Texture creation failed with `INVALIDARG`.

Set the compiler define `D3DCOMPILER_DEBUG` to tell the HLSL compiler to include debug information into the shader blob.

```
#define D3DCOMPILER_DEBUG 1
```



Code Flow & Code Example for a simple app

<https://docs.microsoft.com/en-us/windows/win32/direct3d12/creating-a-basic-direct3d-12-component#code-flow-for-a-simple-app>

<https://docs.microsoft.com/en-us/windows/win32/direct3d12/creating-a-basic-direct3d-12-component#code-example-for-a-simple-app>

We will cover the details next week using a different framework!



Objectives

This week, you will:

- Learn how vectors are represented and discover the operations defined on them
- Become familiar with the vector functions and classes of the DirectX Math library
- Obtain an understanding of matrices and their operations
- Become familiar with the subset of classes and functions provided by the DirectX Math library used for matrix mathematics
- Understand how linear and affine transformations can be represented by matrices
- Learn the coordinate transformations for scaling, rotating, and translating geometry
- Become familiar with the subset of functions provided by the DirectX Math library used for constructing transformation matrices

Vectors Review

Vectors play a crucial role in computer graphics

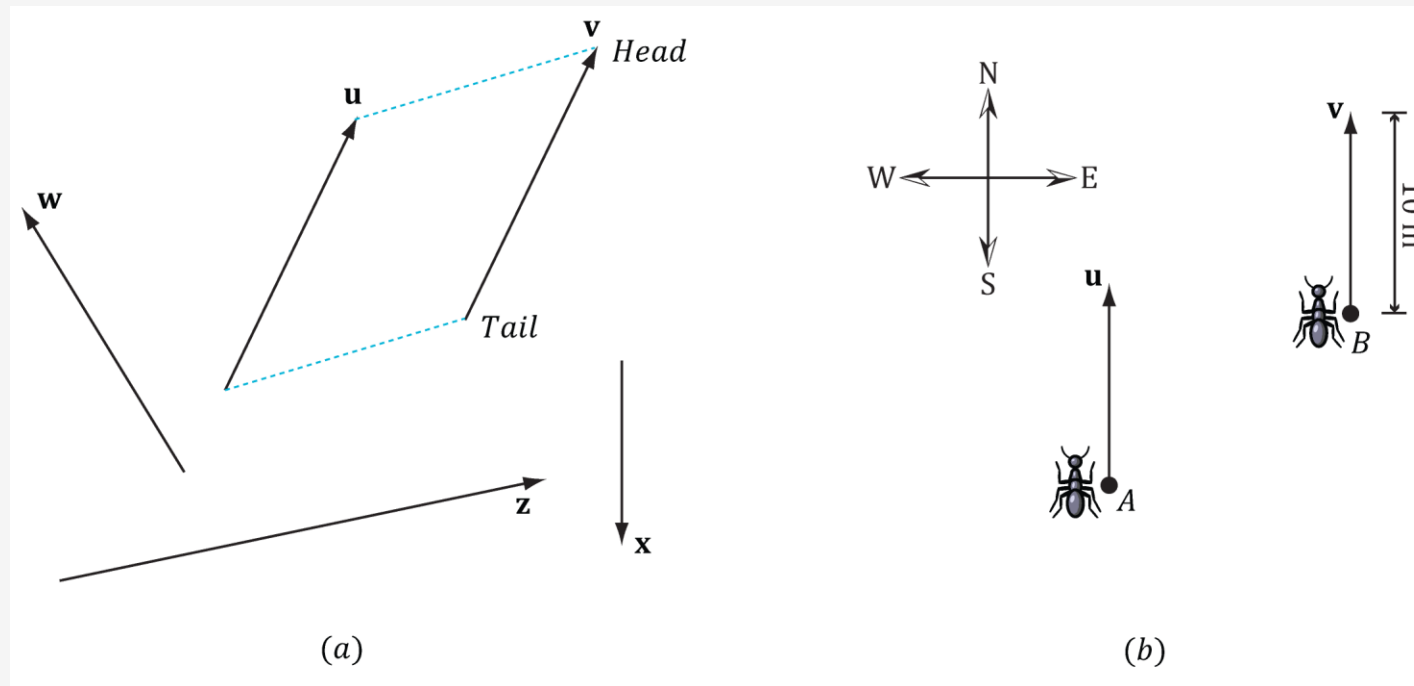
- Also collision detection and physical simulation
- Common components in modern video games
- A vector possesses both magnitude and direction
- Quantities that possess both magnitude and direction are called *vector-valued quantities*
- E.g.: forces, displacements (change) and velocities

A more complete reference to video game mathematics:

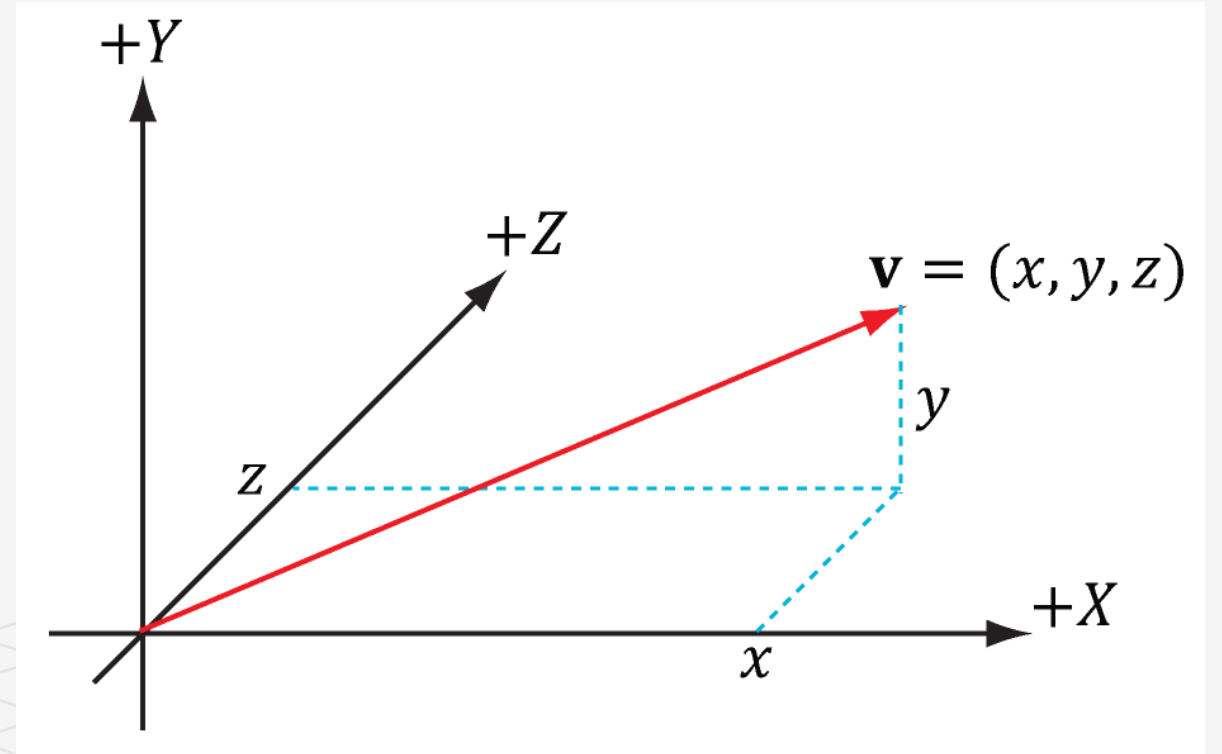
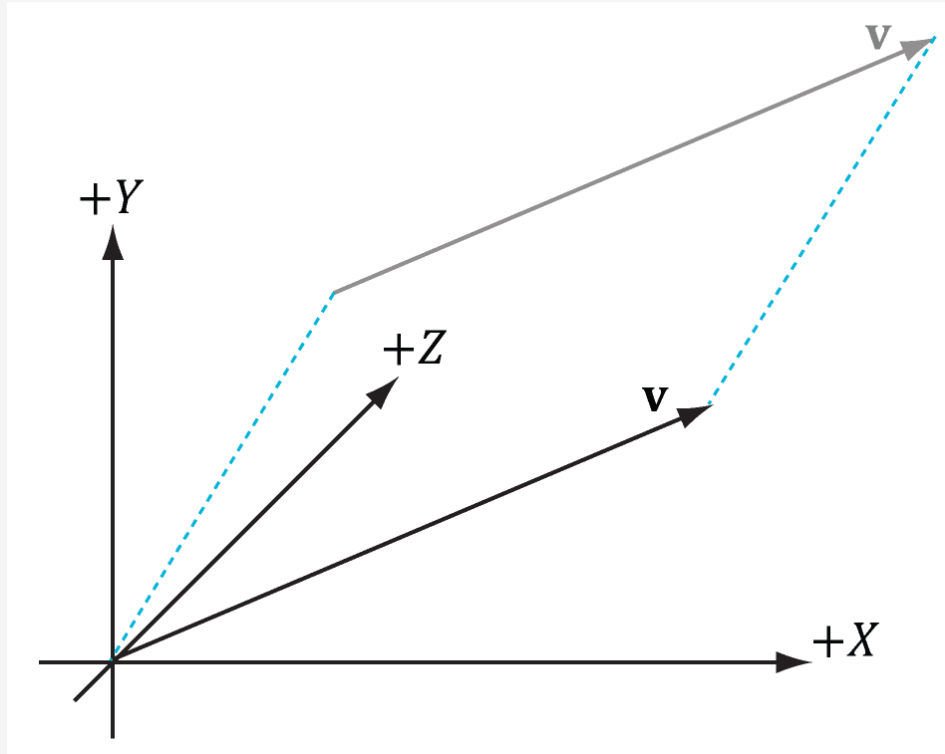
[Verth04] Verth, James M. van, and Lars M. Bishop. *Essential Mathematics for Games & Interactive Applications: A Programmer's Guide*. Morgan Kaufmann Publishers (www.mkp.com), 2004.

[Lengyel02] Lengyel, Eric, *Mathematics for 3D Game Programming and Computer Graphics*. Charles River Media, Inc., 2002.

2D Vectors



2D vs. 3D Vectors

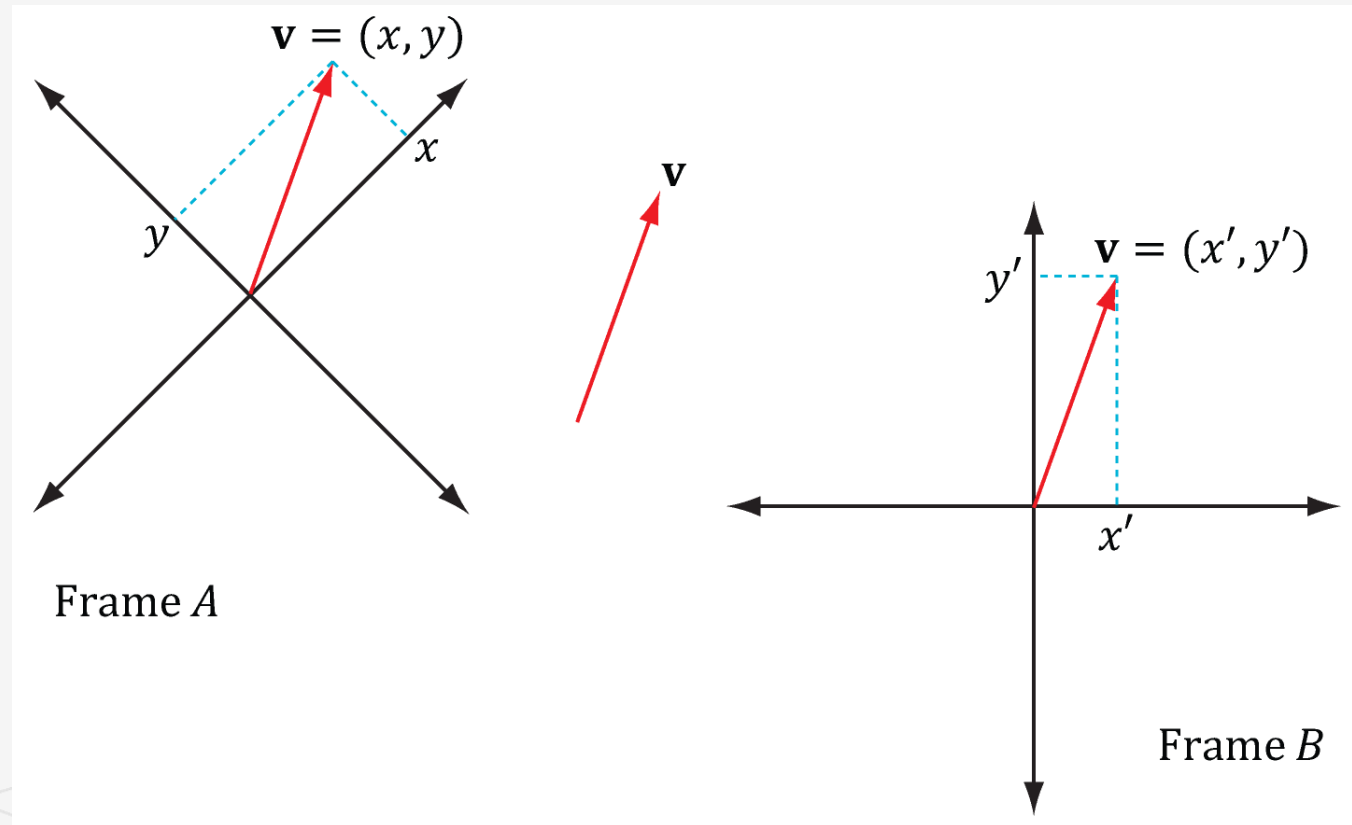


A vector **in** two frames in space

The same vector \mathbf{v} has different coordinates when described relative to different frames.

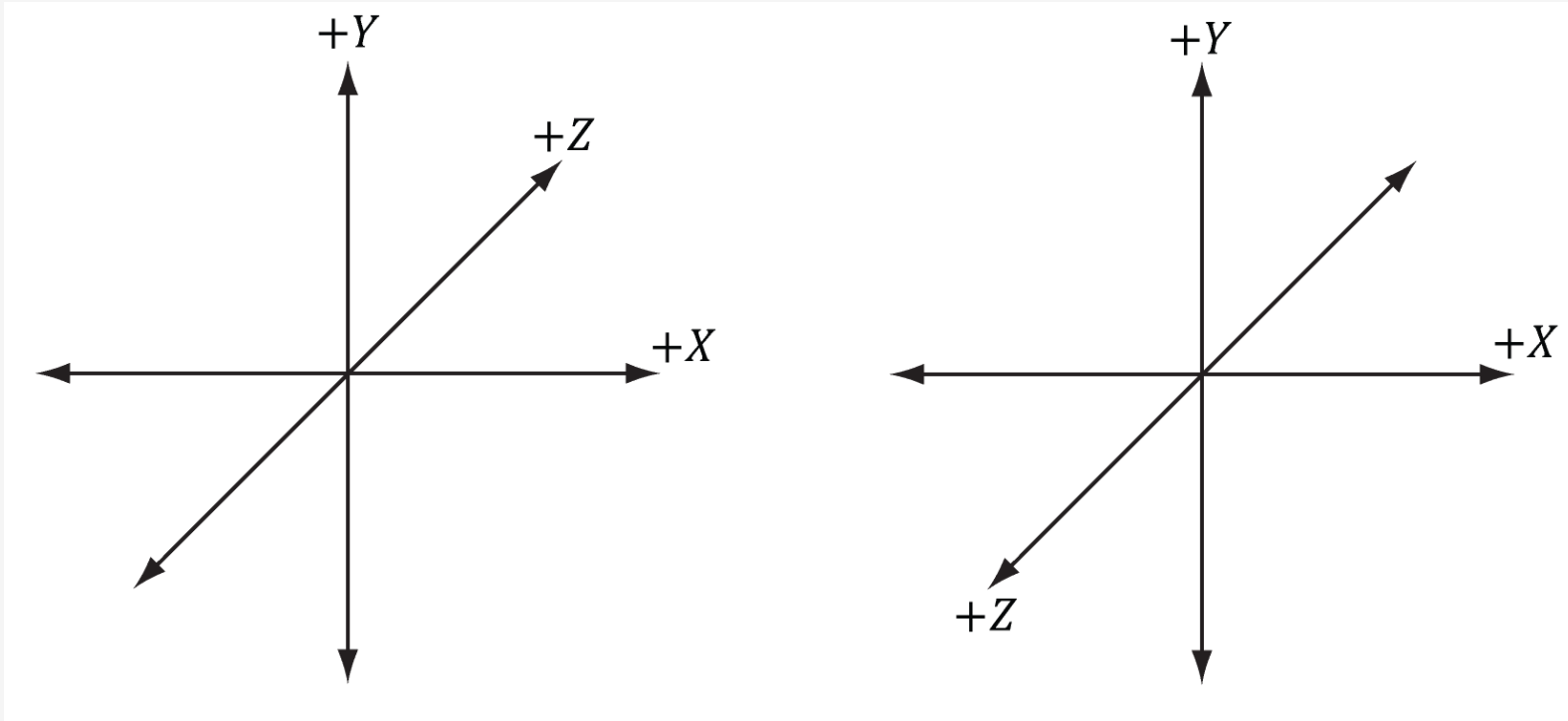
The idea is analogous to, say, temperature. Water boils at 100° Celsius or 212° Fahrenheit.

The physical temperature of boiling water is the *same* no matter the scale (i.e., we can't lower the boiling point by picking a different scale), but we assign a different scalar number to the temperature based on the scale we use.



Vectors in Direct3D

- Direct3D uses a so-called left-handed coordinate system. In other words, the +Z goes into the page.



Vector Algebra

For definitions of $\mathbf{u} = (u_x, u_y, u_z)$ and $\mathbf{v} = (v_x, v_y, v_z)$

a) Multiplication with a scalar k with \mathbf{u} is $k\mathbf{u} = (ku_x, ku_y, ku_z)$

b) Addition is $\mathbf{u} + \mathbf{v} = (u_x + v_x, u_y + v_y, u_z + v_z)$

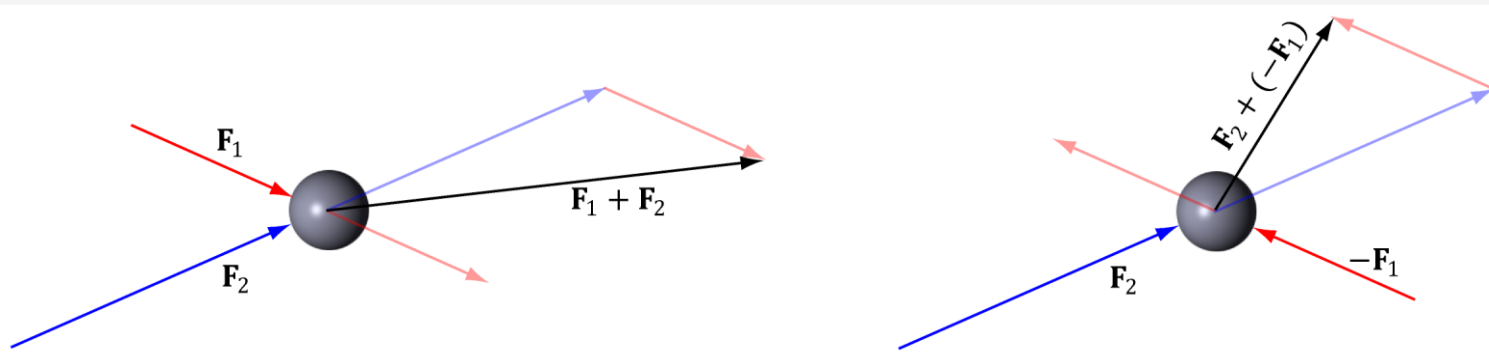
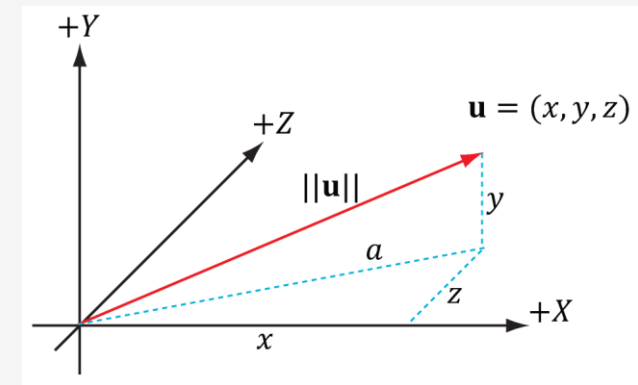
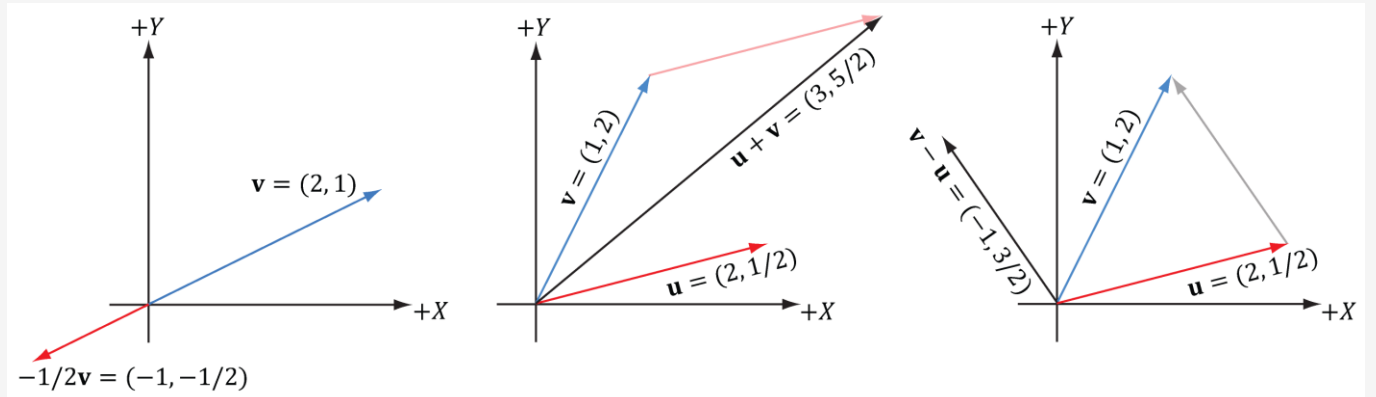
c) Subtraction of $\mathbf{u} - \mathbf{v} = \mathbf{u} + (-\mathbf{v})$ or $(u_x - v_x, u_y - v_y, u_z - v_z)$

d) Magnitude of a vector $\|\mathbf{u}\|$ is

$$\|\mathbf{u}\| = \sqrt{y^2 + a^2} = \sqrt{y^2 + (\sqrt{x^2 + z^2})^2} = \sqrt{x^2 + y^2 + z^2}$$

e) A normalized vector is

$$\hat{\mathbf{u}} = \frac{\mathbf{u}}{\|\mathbf{u}\|} = \left(\frac{x}{\|\mathbf{u}\|}, \frac{y}{\|\mathbf{u}\|}, \frac{z}{\|\mathbf{u}\|} \right)$$



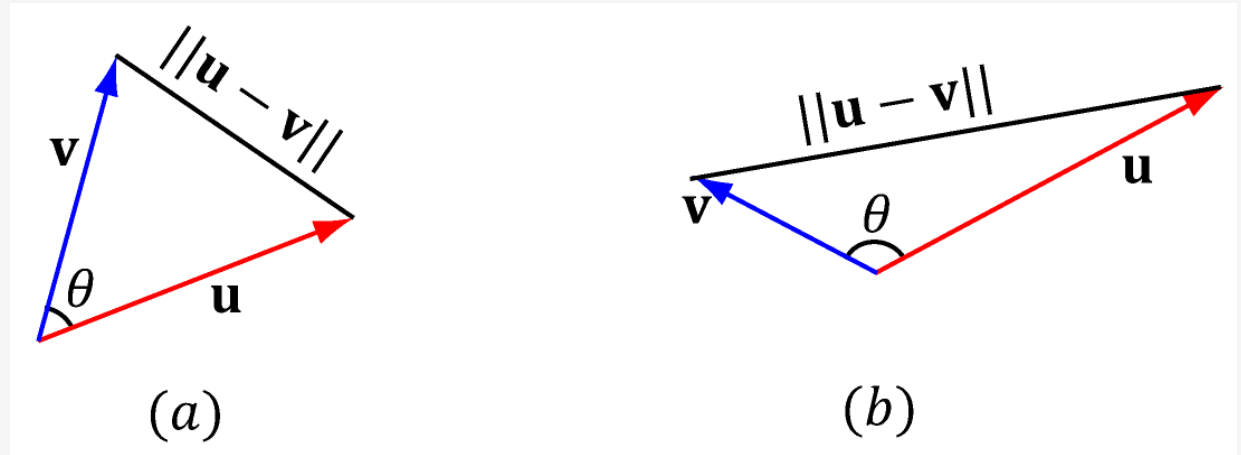
Dot product

- A form of vector multiplication that results in a scalar value
- Sometimes referred to as the *scalar product*
- Defined as $\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z$
- Given the law of cosines, we can get $\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta$

In the left figure, the angle θ between \mathbf{u} and \mathbf{v} is an acute angle.

In the right figure, the angle θ between \mathbf{u} and \mathbf{v} is an obtuse angle.

When we refer to the angle between two vectors, we always mean the smallest angle, that is, the angle θ such that $0 \leq \theta \leq \pi$.



How to find an angle between 2 vectors

Find the angle between \mathbf{u} and \mathbf{v} where $\mathbf{u} = (1, 2, 3)$ and $\mathbf{v} = (-4, 0, -1)$

$$1) \quad \mathbf{u} \cdot \mathbf{v} = (1, 2, 3) \cdot (-4, 0, -1) = -4 - 3 = -7$$

$$2) \quad \|\mathbf{u}\| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$$

$$3) \quad \|\mathbf{v}\| = \sqrt{(-4)^2 + 0^2 + (-1)^2} = \sqrt{17}$$

$$4) \quad \cos \theta = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|} = \frac{-7}{\sqrt{14} \sqrt{17}}$$

$$5) \quad \theta = \cos^{-1} \frac{-7}{\sqrt{14} \sqrt{17}} \approx 117^\circ$$

Vector Projection

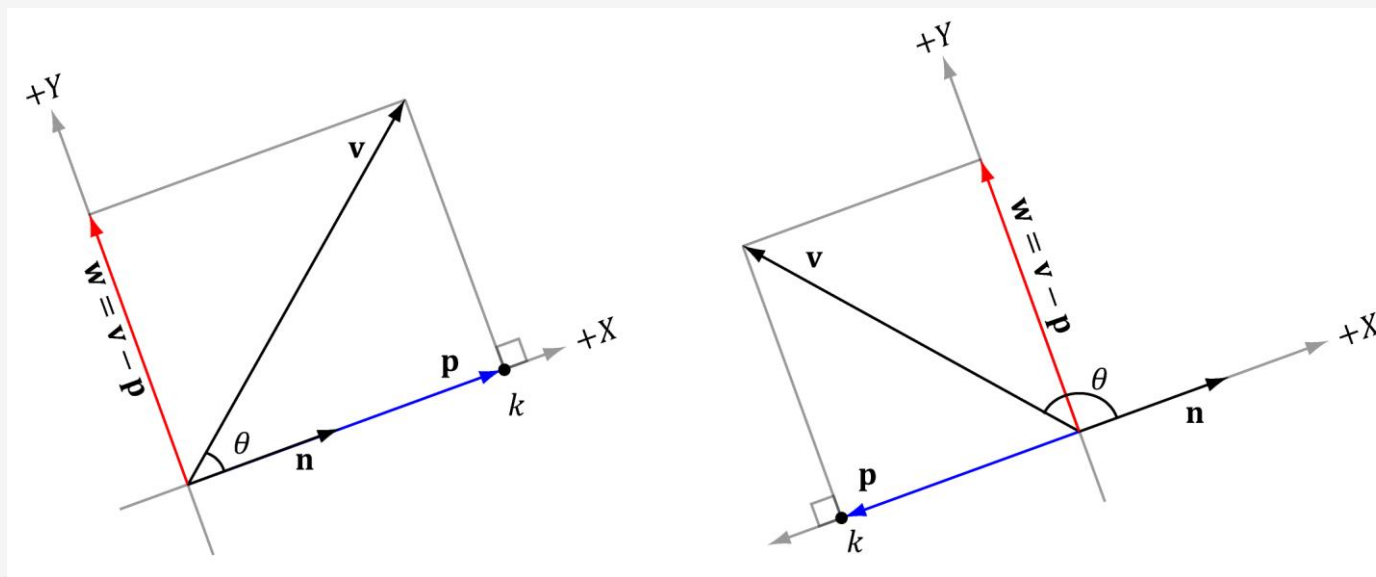
We have a scalar k such that:

$$\mathbf{p} = k\mathbf{n} = (\|\mathbf{v}\| \cos\theta)\mathbf{n}$$

▪ Let's assume that \mathbf{n} is a unit vector, so we have the following result: $\mathbf{p} = (\|\mathbf{v}\| \cos\theta)\mathbf{n} = (\|\mathbf{v}\| \cdot 1 \cos\theta)\mathbf{n} = (\|\mathbf{v}\| \|\mathbf{n}\| \cos\theta)\mathbf{n} = (\mathbf{v} \cdot \mathbf{n})\mathbf{n}$

▪ We call \mathbf{p} the *orthogonal projection* of \mathbf{v} on \mathbf{n} , and can be expressed as: $\mathbf{p} = \text{proj}_{\mathbf{n}}(\mathbf{v})$

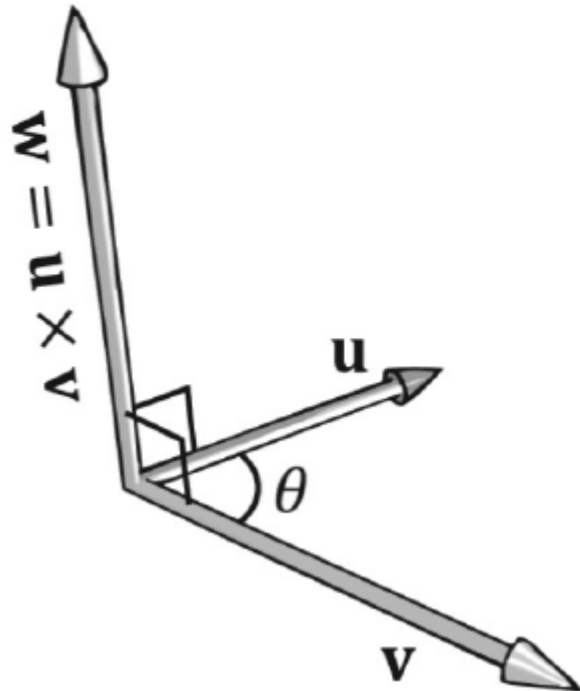
▪ And finally, we get: $\mathbf{p} = \text{proj}_{\mathbf{n}}(\mathbf{v}) = \left(\mathbf{v} \cdot \frac{\mathbf{n}}{\|\mathbf{n}\|} \right) \frac{\mathbf{n}}{\|\mathbf{n}\|} = \frac{(\mathbf{v} \cdot \mathbf{n})}{\|\mathbf{n}\|^2} \mathbf{n}$



Cross Product

Given $\mathbf{u} = (2, 1, 3)$ and $\mathbf{v} = (2, 0, 0)$, compute $\mathbf{w} = \mathbf{u} \times \mathbf{v}$

$$\mathbf{w} = \mathbf{u} \times \mathbf{v} = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x)$$



$$\begin{aligned}\mathbf{w} &= \mathbf{u} \times \mathbf{v} \\ &= (2, 1, 3) \times (2, 0, 0) \\ &= (1 \cdot 0 - 3 \cdot 0, 3 \cdot 2 - 2 \cdot 0, 2 \cdot 0 - 1 \cdot 2) \\ &= (0, 6, -2)\end{aligned}$$

DirectX Math Vectors

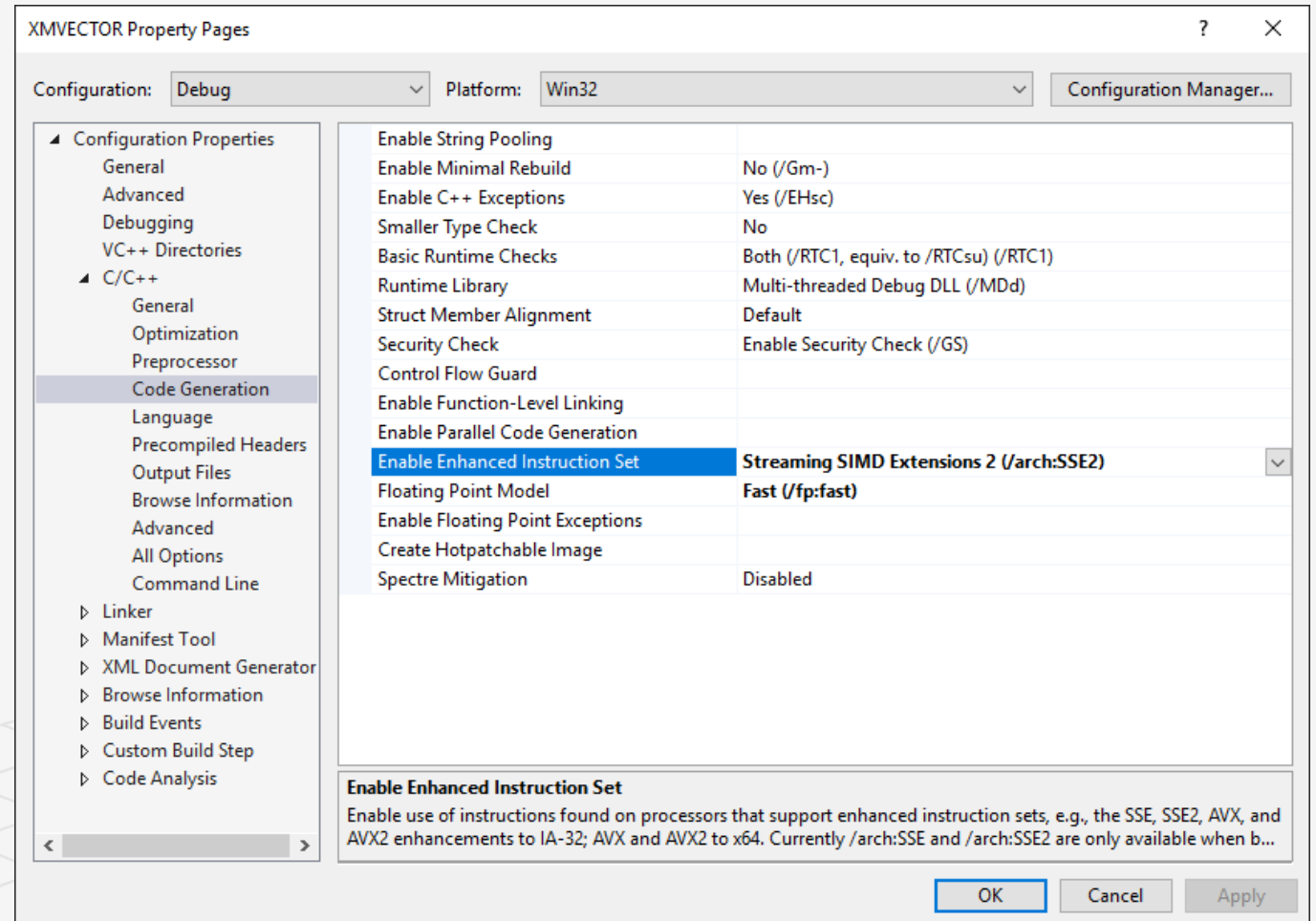
For Windows 8+, DirectX Math is library for Direct3D

- Uses Streaming SIMD Extensions 2 (SSE2) instruction set
 - 128-bit single instruction multiple data registers (SIMD)
 - Operates on four 32-bit (floats or ints) with one instruction
- Requirements

- `#include <DirectXMath.h>`
- `#include <DirectXPackedVector.h>`
- `DirectXMath.h` resides in `DirectX` namespace
- `DirectXPackedVector` resides in `DirectX::PackedVector` ns

If using X86 platform, you need to enable SSE2 in

- Project Properties->Configuration Properties->C/C++->Code Generation->Enable Enhanced Instruction Set
- For all platforms, enable fast floating point model `/fp:fast`
- Project Properties->Configuration Properties->C/C++->Code Generation->Floating Point Model



First Example

```
#include <windows.h> // for XMVerifyCPUSupport
#include <DirectXMath.h>
#include <DirectXPackedVector.h>
#include <iostream>
using namespace std;
using namespace DirectX;
using namespace DirectX::PackedVector;

int main()
{
    // Check support for SSE2 (Pentium4, AMD K8, and above).
    if (!XMVerifyCPUSupport())
    {
        cout << "directx math not supported" << endl;
        return 0;
    }

    return 0;
}
```

DirectX Math Vectors

DirectX Vector Types

- In DirectX Math, the core vector type is XMVECTOR and it defined as `typedef _m128 XMVECTOR`
- 1. This is a 128-bit type that can process four 32-bit floats with a single SIMD instruction.
- 2. `_m128` is a special SIMD type
- Use this XMVECTOR for local or global variables
- XMVECTOR needs to be 16-byte aligned
- For class data members use XMFLOAT2, XMFLOAT3 and XMFLOAT4 for 2D, 3D and 4D respectively
- Loading and storage functions convert between the types
- Do calculations with XMVECTOR instances
- The definitions of the XMFLOATs are shows in the next few slides

DirectX::XMFLOAT2

```
// 2D Vector; 32 bit floating point components
struct XMFLOAT2
{
    float x;
    float y;

    XMFLOAT2() = default;

    XMFLOAT2(const XMFLOAT2&) = default;
    XMFLOAT2& operator=(const XMFLOAT2&) = default;

    XMFLOAT2(XMFLOAT2&&) = default;
    XMFLOAT2& operator=(XMFLOAT2&&) = default;

    XM_CONSTEXPR XMFLOAT2(float _x, float _y) : x(_x), y(_y) {}
    explicit XMFLOAT2(_In_reads_(2) const float *pArray) : x(pArray[0]), y(pArray[1]) {}
};
```

`XMFLOAT2(float _x, float _y)` initializes a new instance of `XMFLOAT2` from two float arguments.

DirectX::XMFLOAT3

```
// 3D Vector; 32 bit floating point components
struct XMFLOAT3
{
    float x;
    float y;
    float z;

    XMFLOAT3() = default;

    XMFLOAT3(const XMFLOAT3&) = default;
    XMFLOAT3& operator=(const XMFLOAT3&) = default;

    XMFLOAT3(XMFLOAT3&&) = default;
    XMFLOAT3& operator=(XMFLOAT3&&) = default;

    XM_CONSTEXPR XMFLOAT3(float _x, float _y, float _z) : x(_x), y(_y), z(_z) {}
    explicit XMFLOAT3(_In_reads_(3) const float *pArray) : x(pArray[0]), y(pArray[1]), z(pArray[2]) {}
};
```

DirectX::XMFLOAT4

```
// 4D Vector; 32 bit floating point components
struct XMFLOAT4
{
    float x;
    float y;
    float z;
    float w;

    XMFLOAT4() = default;

    XMFLOAT4(const XMFLOAT4&) = default;
    XMFLOAT4& operator=(const XMFLOAT4&) = default;

    XMFLOAT4(XMFLOAT4&&) = default;
    XMFLOAT4& operator=(XMFLOAT4&&) = default;

    XM_CONSTEXPR XMFLOAT4(float _x, float _y, float _z, float _w) : x(_x), y(_y), z(_z), w(_w) {}
    explicit XMFLOAT4(_In_reads_(4) const float *pArray) : x(pArray[0]), y(pArray[1]), z(pArray[2]), w(pArray[3]) {}
};
```

XMFLOATn to XMVECTOR

If we use XMFLOATn types directly for calculations, we will not take advantage of SIMD. In order to use SIMD, we need to convert the instances of XMFLOATn types to XMVECTOR. Conversely, DirectX Math provides storage functions which are used to convert data from XMVECTOR into the XMFLOATn

```
XMVECTOR    XM_CALLCONV    XMLoadFloat2(_In_ const XMFLOAT2* pSource);
```

```
XMVECTOR    XM_CALLCONV    XMLoadFloat3(_In_ const XMFLOAT3* pSource);
```

```
XMVECTOR    XM_CALLCONV    XMLoadFloat4(_In_ const XMFLOAT4* pSource);
```

```
void        XM_CALLCONV    XMStoreFloat2(_Out_ XMFLOAT2* pDestination, _In_ FXMVECTOR V);
```

```
void        XM_CALLCONV    XMStoreFloat3(_Out_ XMFLOAT3* pDestination, _In_ FXMVECTOR V);
```

```
void        XM_CALLCONV    XMStoreFloat4(_Out_ XMFLOAT4* pDestination, _In_ FXMVECTOR V);
```

XMVectorGetX and XMVectorSetX getter and setter

We can use the following getters and setters if we just want to get or set one component of an XMVECTOR

```
float XM_CALLCONV XMVectorGetX(FXMVECTOR V);
```

```
float XM_CALLCONV XMVectorGetY(FXMVECTOR V);
```

```
float XM_CALLCONV XMVectorGetZ(FXMVECTOR V);
```

```
float XM_CALLCONV XMVectorGetW(FXMVECTOR V);
```

```
XMVECTOR XM_CALLCONV XMVectorSetX(FXMVECTOR V, float x);
```

```
XMVECTOR XM_CALLCONV XMVectorSetY(FXMVECTOR V, float y);
```

```
XMVECTOR XM_CALLCONV XMVectorSetZ(FXMVECTOR V, float z);
```

```
XMVECTOR XM_CALLCONV XMVectorSetW(FXMVECTOR V, float w);
```

SSE/SSE2 registers

SSE2 (Streaming SIMD Extensions 2) is one of the Intel [SIMD](#) (Single Instruction, Multiple Data) [processor supplementary instruction](#) sets first introduced by [Intel](#) with the initial version of the [Pentium 4](#) in 2000. It extends the earlier [SSE](#) instruction set, and is intended to fully replace [MMX](#). Intel extended SSE2 to create [SSE3](#) in 2004. SSE2 added 144 new instructions to SSE, which has 70 instructions.

Single instruction, multiple data (SIMD) is a class of [parallel computers](#). It describes computers with [multiple processing elements](#) that perform the same operation on multiple data points simultaneously.

For efficiency purposes, XMVECTOR values can be passed as arguments to functions in SSE/SSE2 registers instead of on the stack. The number of arguments depends on the platform (32-bit, 64-bit, Windows RT) and compiler.

To be platform independent, we use the types: **FXMVECTOR, GXMVECTOR, HXMVECTOR, CXMVECTOR**

- The first three XMVECTOR parameters should be of type FXMVECTOR
- The fourth XMVECTOR should be of type GXMVECTOR
- The fifth and sixth XMVECTOR parameter should be of type HXMVECTOR
- Any additional XMVECTOR parameters should be of type CXMVECTOR

// 32-bit Windows `__fastcall` passes first 3 XMVECTOR arguments via registers, the remaining on the stack.

```
typedef const XMVECTOR FXMVECTOR;  
typedef const XMVECTOR& GXMVECTOR;  
typedef const XMVECTOR& HXMVECTOR;  
typedef const XMVECTOR& CXMVECTOR;
```


// 32-bit Windows `__vectorcall` passes first 6 XMVECTOR arguments via registers, the remaining on the stack.

```
typedef const XMVECTOR FXMVECTOR;  
typedef const XMVECTOR GXMVECTOR;  
typedef const XMVECTOR HXMVECTOR;  
typedef const XMVECTOR& CXMVECTOR;
```

Example

This function takes 6 XMVECTOR parameters, but following the parameter passing rules, it uses FXMVECTOR for the first three parameters, GXMVECTOR for the fourth, and HXMVECTOR for the fifth and sixth.

```
inline XMATRIX XM_CALLCONV XMMatrixTransformation(  
  
    FXMVECTOR ScalingOrigin,  
  
    FXMVECTOR ScalingOrientationQuaternion,  
  
    FXMVECTOR Scaling,  
  
    GXMVECTOR RotationOrigin,  
  
    HXMVECTOR RotationQuaternion,  
  
    HXMVECTOR Translation);
```



DirectX::XMVECTORF32 and XMVECTORU32

Constant XMVECTOR instances should use the XMVECTORF32 and XMVECTORU32 type.

```
static const XMVECTORF32 g_vHalfVector = { 0.5f, 0.5f, 0.5f, 0.5f };
static const XMVECTORF32 g_vZero = { 0.0f, 0.0f, 0.0f, 0.0f };
static const XMVECTORU32 vGrabY = {0x00000000,0xFFFFFFFF,0x00000000,0x00000000};
```

Essentially, we use XMVECTORF32 whenever we want to use initialization syntax. XMVECTORF32 is a 16-byte aligned structure with a XMVECTOR conversion operator; it is defined as follows:

```
__declspec(align(16)) struct XMVECTORF32
{
    union
    {
        float f[4];
        XMVECTOR v;
    };

    inline operator XMVECTOR() const { return v; }
    inline operator const float*() const { return f; }
#ifdef _XM_NO_INTRINSICS_ && defined(_XM_SSE_INTRINSICS_)
    inline operator __m128i() const { return _mm_castps_si128(v); }
    inline operator __m128d() const { return _mm_castps_pd(v); }
#endif
};
```


Overloaded Operators in DirectXMath.h

The DirectX Math library provides useful overloaded operators and utility functions to perform a variety of operations. Examples:

```
XMVECTOR      XM_CALLCONV  operator+ (FXMVECTOR V);
XMVECTOR      XM_CALLCONV  operator- (FXMVECTOR V);

XMVECTOR&     XM_CALLCONV  operator+= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR&     XM_CALLCONV  operator-= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR&     XM_CALLCONV  operator*= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR&     XM_CALLCONV  operator/= (XMVECTOR& V1, FXMVECTOR V2);

XMVECTOR&     operator*= (XMVECTOR& V, float S);
XMVECTOR&     operator/= (XMVECTOR& V, float S);

XMVECTOR      XM_CALLCONV  operator+ (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR      XM_CALLCONV  operator- (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR      XM_CALLCONV  operator* (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR      XM_CALLCONV  operator/ (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR      XM_CALLCONV  operator* (FXMVECTOR V, float S);
XMVECTOR      XM_CALLCONV  operator* (float S, FXMVECTOR V);
XMVECTOR      XM_CALLCONV  operator/ (FXMVECTOR V, float S);
```

Pi any one?

The DirectX Math library defined the following constants for approximating pi:

```
XM_CONST float XM_PI      = 3.141592654f;
```

```
XM_CONST float XM_2PI     = 6.283185307f;
```

```
XM_CONST float XM_1DIVPI  = 0.318309886f;
```

```
XM_CONST float XM_1DIV2PI = 0.159154943f;
```

```
XM_CONST float XM_PIDIV2  = 1.570796327f;
```

```
XM_CONST float XM_PIDIV4  = 0.785398163f;
```

Matrices Review

An $m \times n$ matrix is a rectangular array of real numbers with m rows and n columns

- The product of the number of rows and columns gives its dimensions
- The numbers in a matrix are called elements or entries

□ Examples: The matrix **A** is a 4×4 matrix; the matrix **B** is a 3×2 matrix; the matrix **u** is a 1×3 matrix; and the matrix **v** is a 4×1 matrix.

$$\mathbf{A} = \begin{bmatrix} 3.5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 2 & -5 & \sqrt{2} & 1 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{bmatrix} \quad \mathbf{u} = [u_1, u_2, u_3] \quad \mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ \sqrt{3} \\ \pi \end{bmatrix}$$

Matrices have some nice algebraic properties

$\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A} \rightarrow$ Commutative law of addition

$(\mathbf{A} + \mathbf{B}) + \mathbf{C} = \mathbf{A} + (\mathbf{B} + \mathbf{C}) \rightarrow$ Associative law of addition

$r(\mathbf{A} + \mathbf{B}) = r\mathbf{A} + r\mathbf{B} \rightarrow$ Scalar distribution over matrices

$(r + s)\mathbf{A} = r\mathbf{A} + s\mathbf{A} \rightarrow$ Matrix distribution over scalars

Matrix operations:

- Two matrices must have the same number of rows and columns in order to be compared
- We add two matrices by adding their corresponding elements so it only makes sense to add matrices that the same number of rows and columns
- We multiply a scalar and a matrix by multiplying the scalar with every element in the matrix
- We define subtraction in terms of matrix addition and scalar multiplication, e.g. $A - B = A + (-1 B) = A + (-B)$

Given: $\mathbf{A} = \begin{bmatrix} 1 & 5 \\ -2 & 3 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 6 & 2 \\ 5 & -8 \end{bmatrix}, \mathbf{C} = \begin{bmatrix} 1 & 5 \\ -2 & 3 \end{bmatrix}, \mathbf{D} = \begin{bmatrix} 2 & 1 & -3 \\ -6 & 3 & 0 \end{bmatrix}$

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} 1 & 5 \\ -2 & 3 \end{bmatrix} + \begin{bmatrix} 6 & 2 \\ 5 & -8 \end{bmatrix} = \begin{bmatrix} 1+6 & 5+2 \\ -2+5 & 3+(-8) \end{bmatrix} = \begin{bmatrix} 7 & 7 \\ 3 & -5 \end{bmatrix}$$

$$3\mathbf{D} = 3 \begin{bmatrix} 2 & 1 & -3 \\ -6 & 3 & 0 \end{bmatrix} = \begin{bmatrix} 3(2) & 3(1) & 3(-3) \\ 3(-6) & 3(3) & 3(0) \end{bmatrix} = \begin{bmatrix} 6 & 3 & -9 \\ -18 & 9 & 0 \end{bmatrix}$$

$$\mathbf{A} - \mathbf{B} = \begin{bmatrix} 1 & 5 \\ -2 & 3 \end{bmatrix} - \begin{bmatrix} 6 & 2 \\ 5 & -8 \end{bmatrix} = \begin{bmatrix} 1-6 & 5-2 \\ -2-5 & 3-(-8) \end{bmatrix} = \begin{bmatrix} -5 & 3 \\ -7 & 11 \end{bmatrix}$$

Vector-Matrix Multiplication

$$\mathbf{uA} = [x, y, z] \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = [x, y, z] \begin{bmatrix} \uparrow & \uparrow & \uparrow \\ \mathbf{A}_{*,1} & \mathbf{A}_{*,2} & \mathbf{A}_{*,3} \\ \downarrow & \downarrow & \downarrow \end{bmatrix}$$

$$\begin{aligned} \mathbf{uA} &= [\mathbf{u} \cdot \mathbf{A}_{*,1} \quad \mathbf{u} \cdot \mathbf{A}_{*,2} \quad \mathbf{u} \cdot \mathbf{A}_{*,3}] \\ &= [xA_{11} + yA_{21} + zA_{31}, \quad xA_{12} + yA_{22} + zA_{32}, \quad xA_{13} + yA_{23} + zA_{33}] \\ &= [xA_{11}, xA_{12}, xA_{13}] + [yA_{21}, yA_{22}, yA_{23}] + [zA_{31}, zA_{32}, zA_{33}] \\ &= x[A_{11}, A_{12}, A_{13}] + y[A_{21}, A_{22}, A_{23}] + z[A_{31}, A_{32}, A_{33}] \\ &= x\mathbf{A}_{1,*} + y\mathbf{A}_{2,*} + z\mathbf{A}_{3,*} \end{aligned}$$

$$\mathbf{uA} = x\mathbf{A}_{1,*} + y\mathbf{A}_{2,*} + z\mathbf{A}_{3,*}$$

$$[u_1, \dots, u_n] \begin{bmatrix} A_{11} & \dots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{n1} & \dots & A_{nm} \end{bmatrix} = u_1\mathbf{A}_{1,*} + \dots + u_n\mathbf{A}_{n,*}$$

Matrix multiplication Example

$$\mathbf{A} = \begin{bmatrix} -1 & 5 & -4 \\ 3 & 2 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} 2 & 1 & 0 \\ 0 & -2 & 1 \\ -1 & 2 & 3 \end{bmatrix}$$


$$\begin{aligned} \mathbf{AB} &= \begin{bmatrix} -1 & 5 & -4 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 0 & -2 & 1 \\ -1 & 2 & 3 \end{bmatrix} \\ &= \begin{bmatrix} (-1,5,-4) \cdot (2,0,-1) & (-1,5,-4) \cdot (1,-2,2) & (-1,5,-4) \cdot (0,1,3) \\ (3,2,1) \cdot (2,0,-1) & (3,2,1) \cdot (1,-2,2) & (3,2,1) \cdot (0,1,3) \end{bmatrix} \\ &= \begin{bmatrix} 2 & -19 & -7 \\ 5 & 1 & 5 \end{bmatrix} \end{aligned}$$

Matrix multiplication

Matrix multiplication has some nice algebraic properties:

$$\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB} + \mathbf{AC}$$

$$(\mathbf{A}+\mathbf{B})\mathbf{C} = \mathbf{AC} + \mathbf{BC}$$

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$


Matrix Transpose

Interchange the rows and columns of the matrix!

Given these matrices:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 8 \\ 3 & 6 & -4 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}, \mathbf{C} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Their matrix transpose is found by swapping rows and columns:

$$\mathbf{A}^T = \begin{bmatrix} 2 & 3 \\ -1 & 6 \\ 8 & -4 \end{bmatrix}, \mathbf{B}^T = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}, \mathbf{C}^T = [1 \quad 2 \quad 3 \quad 4]$$

Matrix Transpose

Transpose has some useful properties

$$(\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$$

$$(\mathbf{cA})^T = \mathbf{cA}^T$$

$$(\mathbf{A}^T)^T = \mathbf{A}$$

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$$

$$(\mathbf{A}^{-1})^T = (\mathbf{A}^T)^{-1}$$

Matrix Identity

The identity matrix is a square matrix that has zeros for all elements except for ones along the main diagonal:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Let $\mathbf{u} = [-1, 2]$ and let $\mathbf{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Verify that $\mathbf{uI} = \mathbf{u}$

$$\mathbf{uI} = [-1, \ 2] \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = [(-1, 2) \cdot (1, 0), \ (-1, 2) \cdot (0, 1)] = [-1, \ 2]$$

Matrix Minors

Given an $n \times n$ matrix A , the minor matrix is the $(n - 1) \times (n - 1)$ matrix found by deleting the i th row and j th column of A

Find the minor matrices \bar{A}_{11} , \bar{A}_{22} , and \bar{A}_{13} of the following matrix:

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

For \bar{A}_{11} we eliminate the first row and first column to obtain:

$$\bar{A}_{11} = \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix}$$

For \bar{A}_{22} we eliminate the second row and second column to obtain:

$$\bar{A}_{22} = \begin{bmatrix} A_{11} & A_{13} \\ A_{31} & A_{33} \end{bmatrix}$$

For \bar{A}_{13} we eliminate the first row and third column to obtain:

$$\bar{A}_{13} = \begin{bmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$$

Matrix Determinant

The determinant is a special function which inputs a square matrix and outputs a real number

- The determinant of a square matrix A is commonly denoted by $\det A$
- Determinants are used to solve systems of linear equations
- Determinants give us an explicit formula for finding the inverse of a matrix
- Matrix A is invertible if and **only** if $\det A \neq 0$

The determinant of a matrix is defined recursively

- the determinant of a 4×4 matrix is defined in terms of the determinant of a 3×3 matrix
- the determinant of a 3×3 matrix is defined in terms of the determinant of a 2×2 matrix
- the determinant of a 2×2 matrix is defined in terms of the determinant of a 1×1 matrix
- the determinant of a 1×1 matrix $A = [A_{11}]$ is trivially defined to be $\det [A_{11}] = A_{11}$

Matrix Determinant

For 2x2 matrices, we have the formula:

$$\det \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = A_{11} \det[A_{22}] - A_{12} \det[A_{21}] = A_{11}A_{22} - A_{12}A_{21}$$

For 3x3 matrices, we have the formula:

$$\det \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = A_{11} \det \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix} - A_{12} \det \begin{bmatrix} A_{21} & A_{23} \\ A_{31} & A_{33} \end{bmatrix} + A_{13} \det \begin{bmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$$

Matrix determinant for 4x4 matrices

$$\det \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} = A_{11} \det \begin{bmatrix} A_{22} & A_{23} & A_{24} \\ A_{32} & A_{33} & A_{34} \\ A_{42} & A_{43} & A_{44} \end{bmatrix} - A_{12} \det \begin{bmatrix} A_{21} & A_{23} & A_{24} \\ A_{31} & A_{33} & A_{34} \\ A_{41} & A_{43} & A_{44} \end{bmatrix} \\ + A_{13} \det \begin{bmatrix} A_{21} & A_{22} & A_{24} \\ A_{31} & A_{32} & A_{34} \\ A_{41} & A_{42} & A_{44} \end{bmatrix} - A_{14} \det \begin{bmatrix} A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} \end{bmatrix}$$

Determinant Example

Find the determinant of

$$\mathbf{A} = \begin{bmatrix} 2 & -5 & 3 \\ 1 & 3 & 4 \\ -2 & 3 & 7 \end{bmatrix}$$

$$\det \mathbf{A} = A_{11} \det \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix} - A_{12} \det \begin{bmatrix} A_{21} & A_{23} \\ A_{31} & A_{33} \end{bmatrix} + A_{13} \det \begin{bmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$$

$$\begin{aligned} \det \mathbf{A} &= 2 \det \begin{bmatrix} 3 & 4 \\ 3 & 7 \end{bmatrix} - (-5) \det \begin{bmatrix} 1 & 4 \\ -2 & 7 \end{bmatrix} + 3 \det \begin{bmatrix} 1 & 3 \\ -2 & 3 \end{bmatrix} \\ &= 2(3 \cdot 7 - 4 \cdot 3) + 5(1 \cdot 7 - 4 \cdot (-2)) + 3(1 \cdot 3 - 3 \cdot (-2)) \\ &= 2(9) + 5(15) + 3(9) \\ &= 18 + 75 + 27 \\ &= 120 \end{aligned}$$

Matrix Adjoint

With a matrix adjoint, we can find an explicit formula for computing matrix inverses

Given an $n \times n$ matrix A , the product $C_{ij} = (-1)^{i+j} \det \bar{A}_{ij}$

is called the cofactor of A_{ij}

After computing C_{ij} and placing it in a matrix C_A as follows:

$$C_A = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1n} \\ C_{21} & C_{22} & \cdots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nn} \end{bmatrix}$$

The adjoint of A is a transpose of C_A which we can denote by

$$\mathbf{A}^* = \mathbf{C}_A^T$$

Matrix Inverses

Only square matrices have inverses

- Not every square matrix has one though
- The inverse of an $n \times n$ matrix M is an $n \times n$ matrix denoted by M^{-1}
- The inverse is unique when it exists
- Multiplying a matrix with its inverse results in the identity matrix: $MM^{-1} = M^{-1}M = I$
- The formula for finding inverses can be given as follows:

$$\mathbf{A}^{-1} = \frac{\mathbf{A}^*}{\det \mathbf{A}}$$

DirectX Matrices

We'll focus on 4 x 4 matrices

In DirectX math, we have the XMATRIX class defined in the DirectXMath.h header

- Uses four XMVECTORs to represent the matrix for SIMD
- Initialized by specifying four row vectors
- Has several overloaded operators for arithmetic
- Can be created using the XMMatrixSet as follows:

```
XMATRIX XM_CALLCONV XMMatrixSet(float m00, float m01, float m02, float m03,  
                                float m10, float m11, float m12, float m13,  
                                float m20, float m21, float m22, float m23,  
                                float m30, float m31, float m32, float m33);
```

DirectX Matrices

Use XMFLOAT4X4 to store matrices as class data members

Has several useful matrix related functions such as

- XMMatrixIdentity()
- XMMatrixIsIdentity()
- XMMatrixMultiply()
- XMMatrixTranspose()
- XMMatrixDeterminant()
- XMMatrixInverse()



Examples

```
XMMATRIX A(1.0f, 0.0f, 0.0f, 0.0f,  
0.0f, 2.0f, 0.0f, 0.0f,  
0.0f, 0.0f, 4.0f, 0.0f,  
1.0f, 2.0f, 3.0f, 1.0f);
```

```
XMMATRIX B = XMMatrixIdentity();
```

```
XMMATRIX C = A * B;
```

```
//or you can call xmmultiply  
XMMATRIX C1 = XMMatrixMultiply(A, B);
```

```
XMMATRIX D = XMMatrixTranspose(A);
```

```
XMVECTOR det = XMMatrixDeterminant(A);  
XMMATRIX E = XMMatrixInverse(&det, A);
```

```
XMMATRIX F = A * E;
```

```
cout << "A = " << endl << A << endl;  
cout << "B = " << endl << B << endl;  
cout << "C = A*B = " << endl << C << endl;  
cout << "C1 = XMMatrixMultiply(A,B) = " << endl << C1 << endl;  
cout << "D = transpose(A) = " << endl << D << endl;  
cout << "det = determinant(A) = " << det << endl << endl;  
cout << "E = inverse(A) = " << endl << E << endl;  
cout << "F = A*E = " << endl << F << endl;
```

```
ostream& XM_CALLCONV operator << (ostream& os, FXMVECTOR v)  
{  
    XMFLOAT4 dest;  
    XMStoreFloat4(&dest, v);
```

```
    os << "(" << dest.x << ", " << dest.y << ", " << dest.z << ", " << dest.w  
    << ")";  
    return os;  
}
```

```
ostream& XM_CALLCONV operator << (ostream& os, FXMMATRIX m)  
{  
    for (int i = 0; i < 4; ++i)  
    {  
        os << XMVectorGetX(m.r[i]) << "\t";  
        os << XMVectorGetY(m.r[i]) << "\t";  
        os << XMVectorGetZ(m.r[i]) << "\t";  
        os << XMVectorGetW(m.r[i]);  
        os << endl;  
    }  
    return os;  
}
```

Transformations

Matrix equations can transform points and vectors in 3D.

We will begin with linear transformations:

Consider the mathematical “Tau” function: $\tau(\mathbf{v}) = \tau(x, y, z) = (x', y', z')$.

This function inputs a 3D vector and outputs a 3D vector.

We say that τ is a **linear transformation** if and only if the following properties hold. where $\mathbf{u} = (u_x, u_y, u_z)$ and $\mathbf{v} = (v_x, v_y, v_z)$ are any 3D vectors, and k is a scalar.

$$\tau(\mathbf{u} + \mathbf{v}) = \tau(\mathbf{u}) + \tau(\mathbf{v}) \text{ \& } \tau(k\mathbf{u}) = k\tau(\mathbf{u})$$

Define the function $\tau(x, y, z) = (x^2, y^2, z^2)$; for example, $\tau(1, 2, 3) = (1, 4, 9)$. This function is not linear!

Because: for $k = 2$ and $\mathbf{u} = (1, 2, 3)$

we have: $\tau(k\mathbf{u}) = \tau(2, 4, 6) = (4, 16, 36)$ but $k\tau(\mathbf{u}) = 2(1, 4, 9) = (2, 8, 18)$

If τ is linear, then it follows that:

$$\tau(a\mathbf{u} + b\mathbf{v} + c\mathbf{w}) = \tau(a\mathbf{u} + (b\mathbf{v} + c\mathbf{w})) = a\tau(\mathbf{u}) + \tau(b\mathbf{v} + c\mathbf{w}) = a\tau(\mathbf{u}) + b\tau(\mathbf{v}) + c\tau(\mathbf{w})$$

Matrix Representation

Let $\mathbf{u} = (x, y, z)$. Observe that we can always write this as:

$$\mathbf{u} = (x, y, z) = x\mathbf{i} + y\mathbf{j} + z\mathbf{k} = x(1, 0, 0) + y(0, 1, 0) + z(0, 0, 1)$$

The vectors $\mathbf{i} = (1, 0, 0)$, $\mathbf{j} = (0, 1, 0)$, and $\mathbf{k} = (0, 0, 1)$, which are unit vectors that aim along the working coordinate axes, respectively, are called the **standard basis vectors** for \mathbf{R}^3 . (\mathbf{R}^3 denotes the set of all 3D coordinate vectors (x, y, z)).

Now let τ be a linear transformation; by linearity, we have:

$$\tau(\mathbf{u}) = \tau(x\mathbf{i} + y\mathbf{j} + z\mathbf{k}) = x\tau(\mathbf{i}) + y\tau(\mathbf{j}) + z\tau(\mathbf{k})$$

$$= \mathbf{uA} = \begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} \leftarrow \tau(\mathbf{i}) \rightarrow \\ \leftarrow \tau(\mathbf{j}) \rightarrow \\ \leftarrow \tau(\mathbf{k}) \rightarrow \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

where $\tau(\mathbf{i}) = (A_{11}, A_{12}, A_{13})$, $\tau(\mathbf{j}) = (A_{21}, A_{22}, A_{23})$, and $\tau(\mathbf{k}) = (A_{31}, A_{32}, A_{33})$.

We call the matrix \mathbf{A} the matrix representation of the linear transformation τ .

Scaling

The left pawn is the original object. The middle pawn is the original pawn scaled 2 units on the y -axis making it taller. The right pawn is the original pawn scaled 2 units on the x -axis making it fatter.

We define the scaling transformation by: $S(x, y, z) = (s_x x, s_y y, s_z z)$. This scales the vector by s_x units on the x -axis, s_y units on the y -axis, and s_z units on the z -axis. S is a linear transformation:

$$S(\mathbf{u} + \mathbf{v}) = (s_x(u_x + v_x), s_y(u_y + v_y), s_z(u_z + v_z))$$

$$= (s_x u_x + s_x v_x, s_y u_y + s_y v_y, s_z u_z + s_z v_z)$$

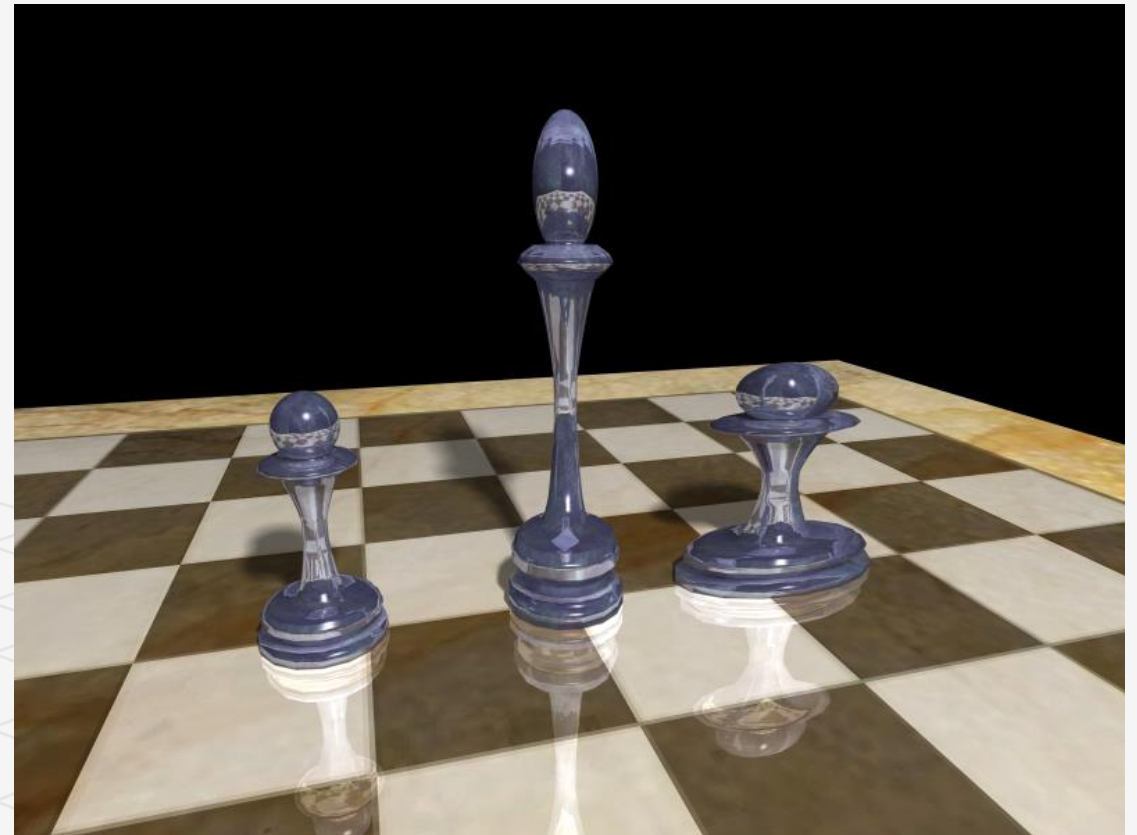
$$= (s_x u_x, s_y u_y, s_z u_z) + (s_x v_x, s_y v_y, s_z v_z)$$

$$= S(\mathbf{u}) + S(\mathbf{v})$$

$$S(k\mathbf{u}) = (s_x k u_x, s_y k u_y, s_z k u_z)$$

$$= k(s_x u_x, s_y u_y, s_z u_z)$$

$$= kS(\mathbf{u})$$



Scaling Matrix

To find the matrix representation, we just apply $S(x, y, z) = (s_x x, s_y y, s_z z)$ to each of the standard basis vectors ($\mathbf{i} = (1, 0, 0)$, $\mathbf{j} = (0, 1, 0)$, and $\mathbf{k} = (0, 0, 1)$):

$$S(\mathbf{i}) = (S_x \cdot 1, S_y \cdot 0, S_z \cdot 0)$$

$$S(\mathbf{j}) = (S_x \cdot 0, S_y \cdot 1, S_z \cdot 0)$$

$$S(\mathbf{k}) = (S_x \cdot 0, S_y \cdot 0, S_z \cdot 1)$$

Thus the matrix representation of S is:

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

And the inverse is:

$$\mathbf{S}^{-1} = \begin{bmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1/s_z \end{bmatrix}$$

Scaling Example

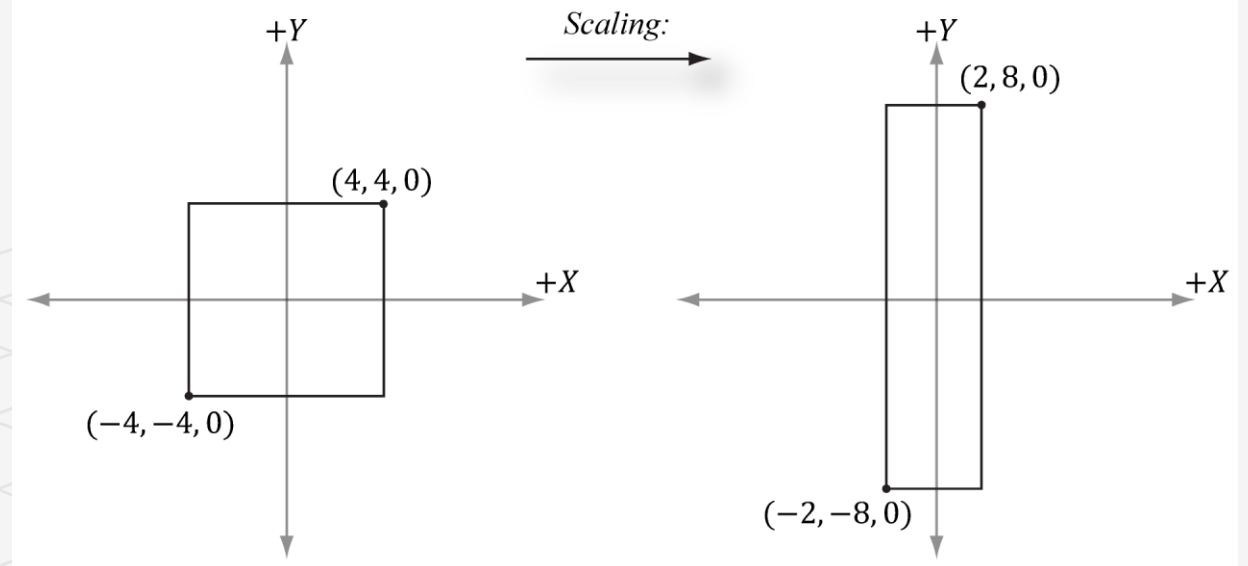
Suppose we have a square defined by a minimum point $(-4, -4, 0)$ and a maximum point $(4, 4, 0)$. Suppose now that we wish to scale the square 0.5 units on the x-axis, 2.0 units on the y-axis, and leave the z-axis unchanged. The corresponding scaling matrix is:

$$S = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -4 & -4 & 0 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -2 & -8 & 0 \end{bmatrix} \quad \begin{bmatrix} 4 & 4 & 0 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 8 & 0 \end{bmatrix}$$

Now to actually scale (transform) the square, we multiply both the minimum point and maximum point by this matrix.

Note that when looking down the negative z-axis, the geometry is basically 2D since $z = 0$.



Scaling Example using DirectX Math

```
XMMATRIX S(0.5f, 0.0f, 0.0f, 0.0f,  
            0.0f, 2.0f, 0.0f, 0.0f,  
            0.0f, 0.0f, 1.0f, 0.0f,  
            0.0f, 0.0f, 0.0f, 0.0f);
```

```
XMVECTOR minPoint = XMVectorSet(-4.0f, -4.0f, 0.0f, 0.0f);
```

```
XMVECTOR maxPoint = XMVectorSet(4.0f, 4.0f, 0.0f, 0.0f);
```

```
XMVECTOR scaledMinPoint = XMVector4Transform(minPoint , S);
```

```
XMVECTOR scaledMaxPoint = XMVector4Transform(maxPoint , S);
```

```
cout << "scaledMinPoint = minPoint x S = " << endl << scaledMinPoint << endl;
```

```
cout << "scaledMinPoint = maxPoint x S = " << endl << scaledMaxPoint << endl;
```

Rotation

we describe rotating a vector \mathbf{v} about an axis \mathbf{n} by an angle θ ; Note that we measure the angle clockwise when looking down the axis \mathbf{n} ; we assume $\|\mathbf{n}\| = 1$.

First, decompose \mathbf{v} into two parts: one part parallel to \mathbf{n} and the other part orthogonal to \mathbf{n} . The parallel part is just $\text{proj}_{\mathbf{n}}(\mathbf{v})$ the orthogonal part is given by

$$\mathbf{v}_{\perp} = \text{perp}_{\mathbf{n}}(\mathbf{v}) = \mathbf{v} - \text{proj}_{\mathbf{n}}(\mathbf{v}), \text{ since } \mathbf{n} \text{ is a unit vector,}$$

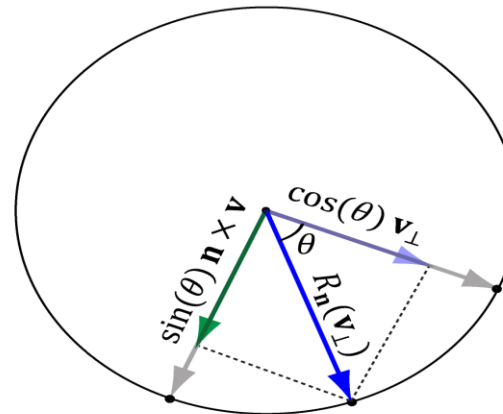
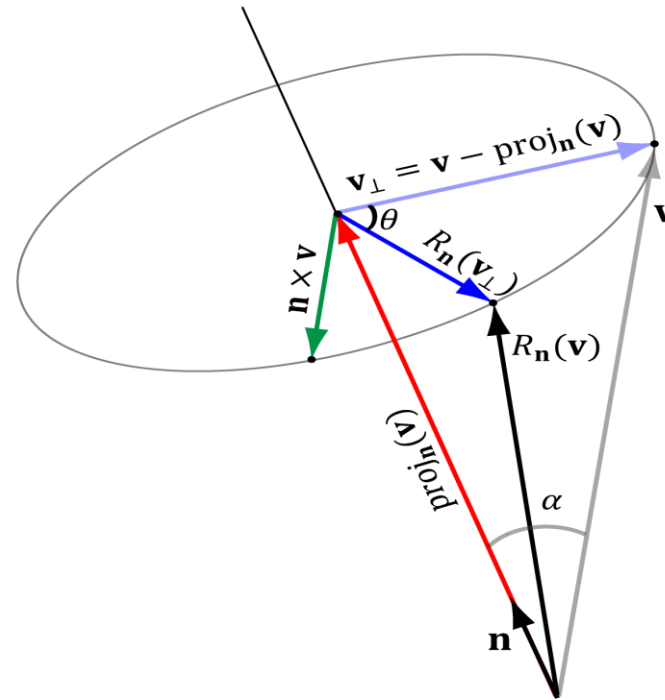
$$\text{we have } \text{proj}_{\mathbf{n}}(\mathbf{v}) = (\mathbf{n} \cdot \mathbf{v})\mathbf{n}.$$

The key observation is that :

the part $\text{proj}_{\mathbf{n}}(\mathbf{v})$ that is parallel to \mathbf{n} is invariant under the rotation,

so we only need to figure out how to rotate the orthogonal part.

$$\text{That is, the rotated vector } R_{\mathbf{n}}(\mathbf{v}) = \text{proj}_{\mathbf{n}}(\mathbf{v}) + R_{\mathbf{n}}(\mathbf{v}_{\perp})$$



Rotation Matrix

The rotation matrices have an interesting property.

Each row vector is unit length and the row vectors are mutually orthogonal (verify).

Therefore the row vectors are *orthonormal* (i.e., mutually orthogonal and unit length).

A matrix whose rows are orthonormal is said to be an **orthogonal matrix**.

An orthogonal matrix has the attractive property that its inverse is actually equal to its transpose.

let $c = \cos\theta$ and $s = \sin\theta$.

$$\mathbf{R}_n = \begin{bmatrix} c + (1-c)x^2 & (1-c)xy + sz & (1-c)xz - sy \\ (1-c)xy - sz & c + (1-c)y^2 & (1-c)yz + sx \\ (1-c)xz + sy & (1-c)yz - sx & c + (1-c)z^2 \end{bmatrix}$$

$$\mathbf{R}_n^{-1} = \mathbf{R}_n^T = \begin{bmatrix} c + (1-c)x^2 & (1-c)xy - sz & (1-c)xz + sy \\ (1-c)xy + sz & c + (1-c)y^2 & (1-c)yz - sx \\ (1-c)xz - sy & (1-c)yz + sx & c + (1-c)z^2 \end{bmatrix}$$

Rotation Matrix

if we choose the x-, y-, and z-axes for rotation (i.e., $\mathbf{n} = (1, 0, 0)$, $\mathbf{n} = (0, 1, 0)$, and $\mathbf{n} = (0, 0, 1)$, respectively), then we get the following rotation matrices which rotate about the x-, y-, and z-axis, respectively. Suppose we have a square defined by a minimum point $(-1, 0, -1)$ and a maximum point $(1, 0, 1)$.

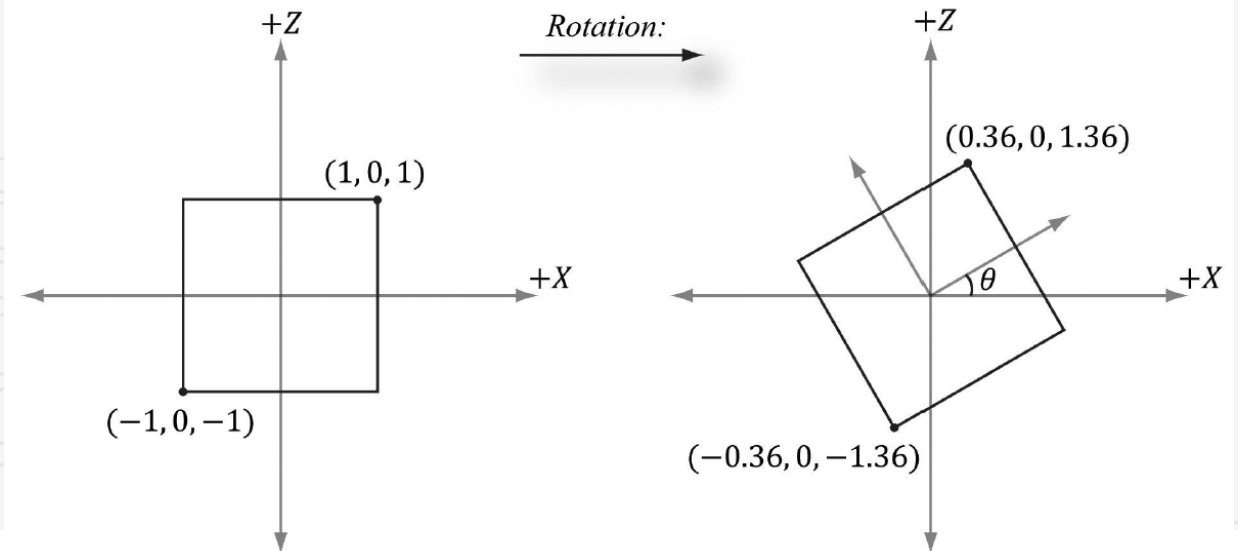
Suppose now that

we wish to rotate the square -30° clockwise

about the y-axis, then $R_y =$

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

$$\begin{bmatrix} \cos(-30^\circ) & 0 & -\sin(-30^\circ) \\ 0 & 1 & 0 \\ \sin(-30^\circ) & 0 & \cos(-30^\circ) \end{bmatrix}$$



RotatingMatrix Example using DirectX Math

```
XMVECTOR minPoint = XMVectorSet(-1.0f, 0.0f, -1.0f, 0.0f);
```

```
XMVECTOR maxPoint = XMVectorSet(1.0f, 0.0f, 1.0f, 0.0f);
```

```
XMMATRIX Ry = XMMatrixRotationY(-XM_PI / 6.0f);
```

```
XMVECTOR rotatedMinPoint = XMVector4Transform(minPoint, Ry);
```

```
XMVECTOR rotatedMaxPoint = XMVector4Transform(maxPoint, Ry);
```

```
cout << "rotatedMinPoint = minPoint x Ry = " << endl << rotatedMinPoint << endl;
```

```
cout << "rotatedMaxPoint = maxPoint x Ry = " << endl << rotatedMaxPoint << endl;
```

Translation Matrix

The translation matrix is represented as:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ b_x & b_y & b_z & 1 \end{bmatrix}$$

With its inverse as:

$$\mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -b_x & -b_y & -b_z & 1 \end{bmatrix}$$

Translation Example

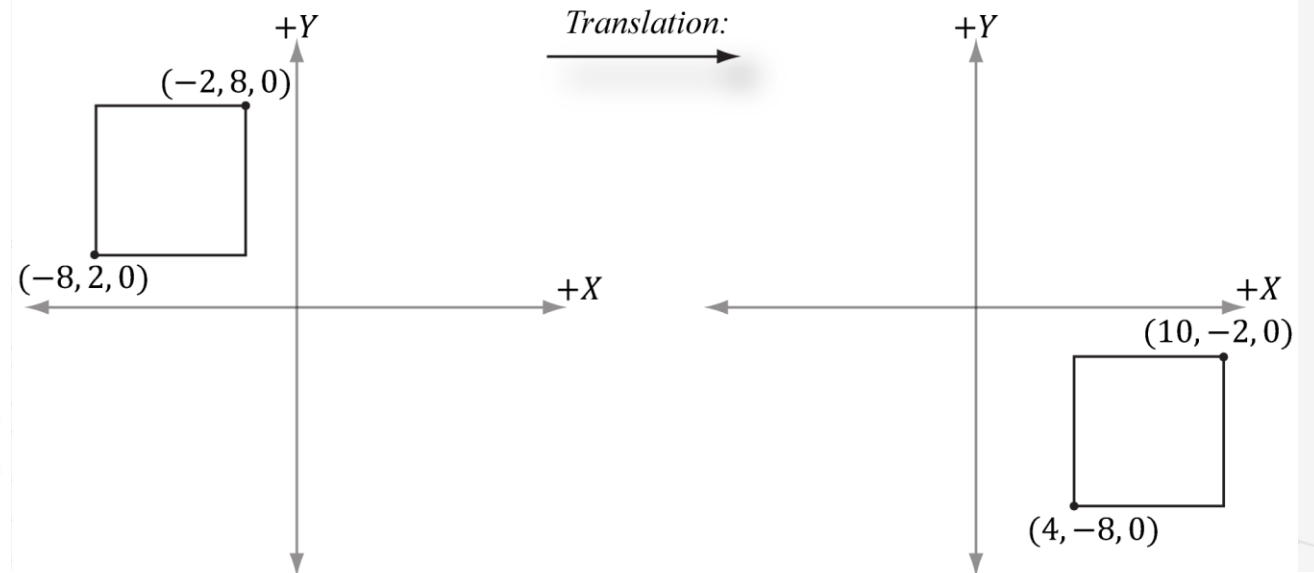
Suppose we have a square defined by a minimum point $(-8, 2, 0)$ and a maximum point $(-2, 8, 0)$.

Suppose now that we wish to translate the square 12 units on the x -axis, -10.0 units on the y -axis, and leave the z -axis unchanged.

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 12 & -10 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -8 & 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 12 & -10 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 4 & -8 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -2 & 8 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 12 & -10 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 10 & -2 & 0 & 1 \end{bmatrix}$$



Translation Code

```
//make sure w = 1.0 for points!
XMVECTOR minPoint = XMVectorSet(-8.0f, 2.0f, 0.0f, 1.0f);
XMVECTOR maxPoint = XMVectorSet(-2.0f, 8.0f, 0.0f, 1.0f);

//Suppose now that we wish to translate the square 12 units on the x-axis, -10.0 units on the y - axis,

XMMATRIX T(1.0f, 0.0f, 0.0f, 0.0f,
            0.0f, 1.0f, 0.0f, 0.0f,
            0.0f, 0.0f, 1.0f, 0.0f,
            12.0f, -10.0f, 0.0f, 1.0f);

XMVECTOR translatedMinPoint = XMVector4Transform(minPoint, T);
XMVECTOR translatedMaxPoint = XMVector4Transform(maxPoint, T);

cout << "translatedMinPoint = minPoint x T = " << endl << translatedMinPoint << endl;
cout << "translatedMaxPoint = maxPoint x T = " << endl << translatedMaxPoint << endl;

//The inverse of the translation matrix is given by :
XMMATRIX TI(1.0f, 0.0f, 0.0f, 0.0f,
            0.0f, 1.0f, 0.0f, 0.0f,
            0.0f, 0.0f, 1.0f, 0.0f,
            -12.0f, 10.0f, 0.0f, 1.0f);

cout << "Let's apply the Inverse of Translation Matrix TI to new translated square: " << endl;

XMVECTOR inversedMinPoint = XMVector4Transform(translatedMinPoint, TI);
XMVECTOR inversedMaxPoint = XMVector4Transform(translatedMaxPoint, TI);

cout << "inversedMinPoint = minPoint x TI = " << endl << inversedMinPoint << endl;
cout << "inversedMaxPoint = maxPoint x TI = " << endl << inversedMaxPoint << endl;
```

Affine Transformation

An affine transformation is a **linear transformation** combined with a **translation**

- Vectors should be unchanged under translations
- Translations should only be applied to points

Homogeneous coordinates provide a convenient notation for us to handle points and vectors uniformly:

1. $(x, y, z, 0)$ for vectors
2. $(x, y, z, 1)$ for points

An affine transformation is a linear transformation plus a translation vector **b**; **A** is the matrix representation of a linear transformation.

$$\alpha(\mathbf{u}) = \tau(\mathbf{u}) + \mathbf{b}$$

Or in matrix notation:

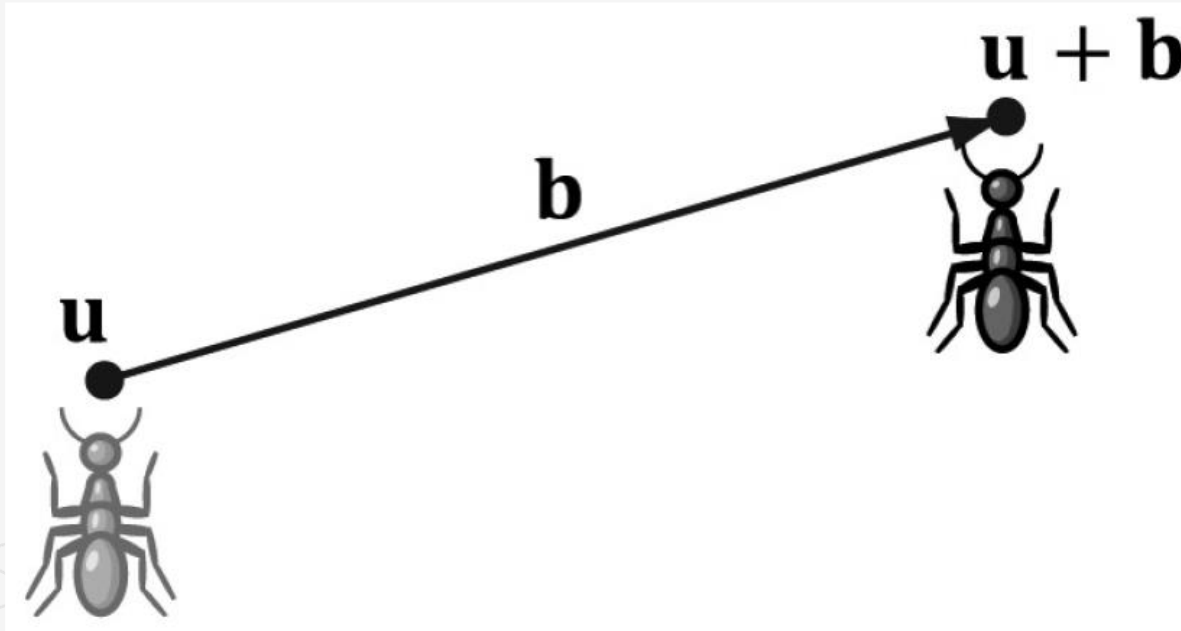
$$\alpha(\mathbf{u}) = \mathbf{uA} + \mathbf{b} = \begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} + \begin{bmatrix} b_x & b_y & b_z \end{bmatrix} = \begin{bmatrix} x' & y' & z' \end{bmatrix}$$

Translation

We define the translation transformation to be the affine transformation whose linear transformation is the identity transformation

$$\tau(\mathbf{u}) = \mathbf{u}\mathbf{I} + \mathbf{b} = \mathbf{u} + \mathbf{b}$$

We simply displace \mathbf{u} by \mathbf{b}



Geometric Interpretation of an Affine Transformation Matrix

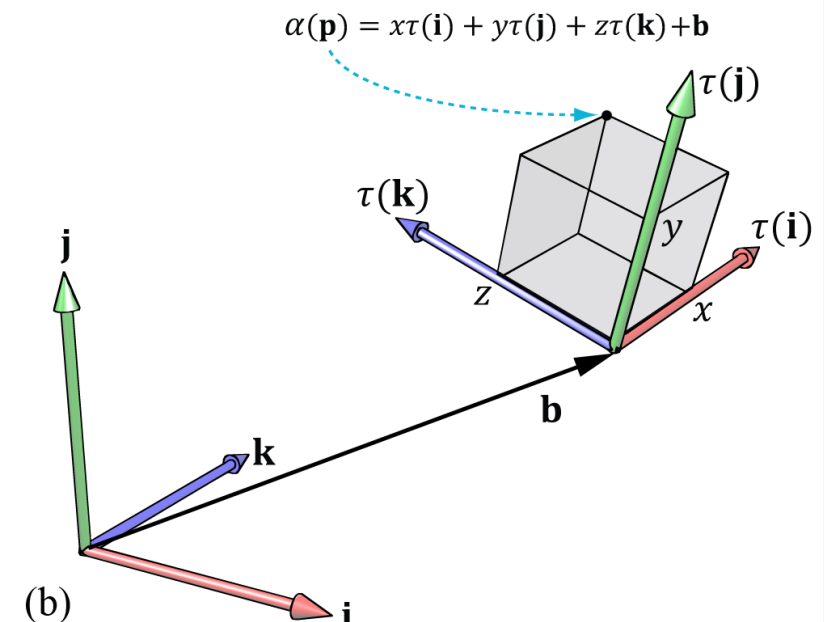
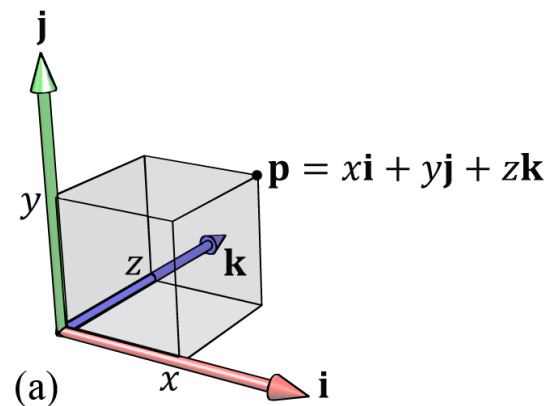
A real world example of a rigid body transformation might be picking a book off your desk and placing it on a bookshelf; during this process you are translating the book from your desk to the bookshelf, but also very likely changing the orientation of the book in the process (rotation).

Let τ be a rotation transformation describing how we want to rotate an object and let \mathbf{b} define a displacement vector describing how we want to translate an object. This rigid body transform can be described by the affine transformation:

$$\alpha(x, y, z) = \tau(x, y, z) + \mathbf{b} = x\tau(\mathbf{i}) + y\tau(\mathbf{j}) + z\tau(\mathbf{k}) + \mathbf{b}$$

In matrix notation, using homogeneous coordinates ($w = 1$ for points and $w = 0$ for vectors so that the translation is not applied to vectors), this is written as:

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \begin{bmatrix} \leftarrow \tau(\mathbf{i}) \rightarrow \\ \leftarrow \tau(\mathbf{j}) \rightarrow \\ \leftarrow \tau(\mathbf{k}) \rightarrow \\ \leftarrow \mathbf{b} \rightarrow \end{bmatrix} = \begin{bmatrix} x' & y' & z' & w \end{bmatrix}$$



Inverses and Change of Coordinate Matrices

Suppose that we are given \mathbf{p}_B (the coordinates of a vector \mathbf{p} relative to frame B),

We are given the change of coordinate matrix \mathbf{M} from frame A to frame B ; that is, $\mathbf{p}_B = \mathbf{p}_A \mathbf{M}$.

We want to solve for \mathbf{p}_A . In other words, instead of mapping from frame A into frame B , we want the change of coordinate matrix that maps us from B into A .

To find this matrix, suppose that \mathbf{M} is invertible (i.e., \mathbf{M}^{-1} exists). We can solve for \mathbf{p}_A like so:

$$\mathbf{p}_B = \mathbf{p}_A \mathbf{M}$$

$$\mathbf{p}_B \mathbf{M}^{-1} = \mathbf{p}_A \mathbf{M} \mathbf{M}^{-1}$$

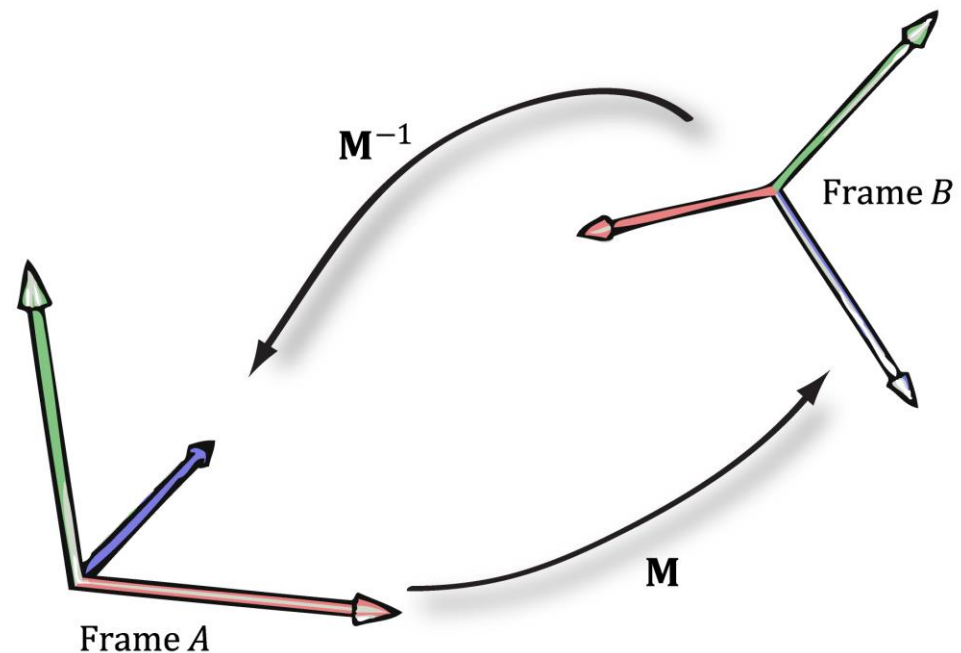
$$\mathbf{p}_B \mathbf{M}^{-1} = \mathbf{p}_A \mathbf{I}$$

$$\mathbf{p}_B \mathbf{M}^{-1} = \mathbf{p}_A$$

Multiplying both sides of the equation by \mathbf{M}^{-1}

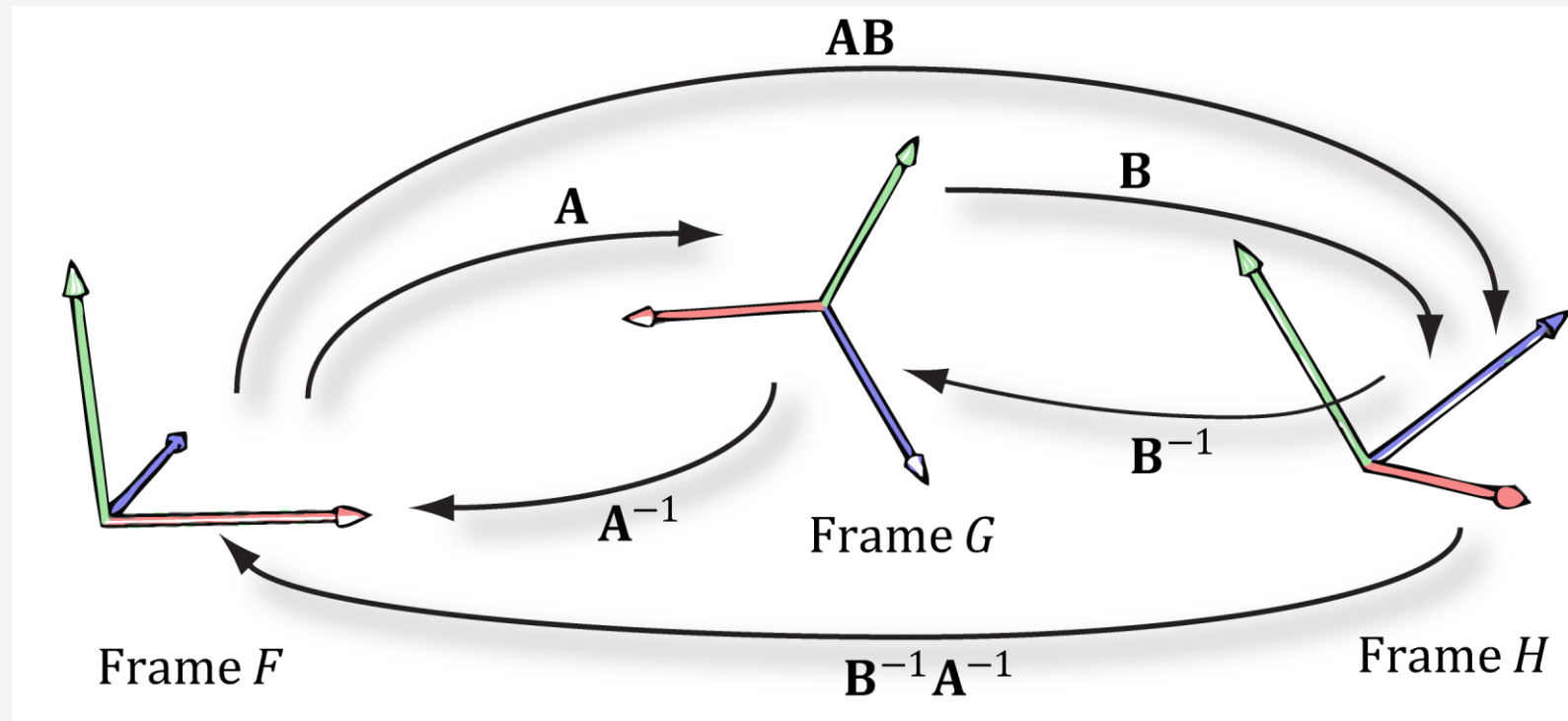
$\mathbf{M} \mathbf{M}^{-1} = \mathbf{I}$, by definition of inverse.

$\mathbf{p}_A \mathbf{I} = \mathbf{p}_A$, by definition of the identity matrix.



Multiple Frames Scenario

A maps from F into G , **B** maps from G into H , and **AB** maps from F directly into H . **B**⁻¹ maps from H into G , **A**⁻¹ maps from G into F and **B**⁻¹**A**⁻¹ maps from H directly into F .



TRANSFORMATION MATRIX VERSUS CHANGE OF COORDINATE MATRIX

So far we have distinguished between “active” transformations (scaling, rotation, translation) and change of coordinate transformations. Mathematically, the two are equivalent, and an active transformation can be interpreted as a change of coordinate transformation, and conversely.

We see that $\mathbf{b} = \mathbf{Q}, \tau(\mathbf{i}) = \mathbf{u}, \tau(\mathbf{j}) = \mathbf{v}, \text{ and } \tau(\mathbf{k}) = \mathbf{w}$.

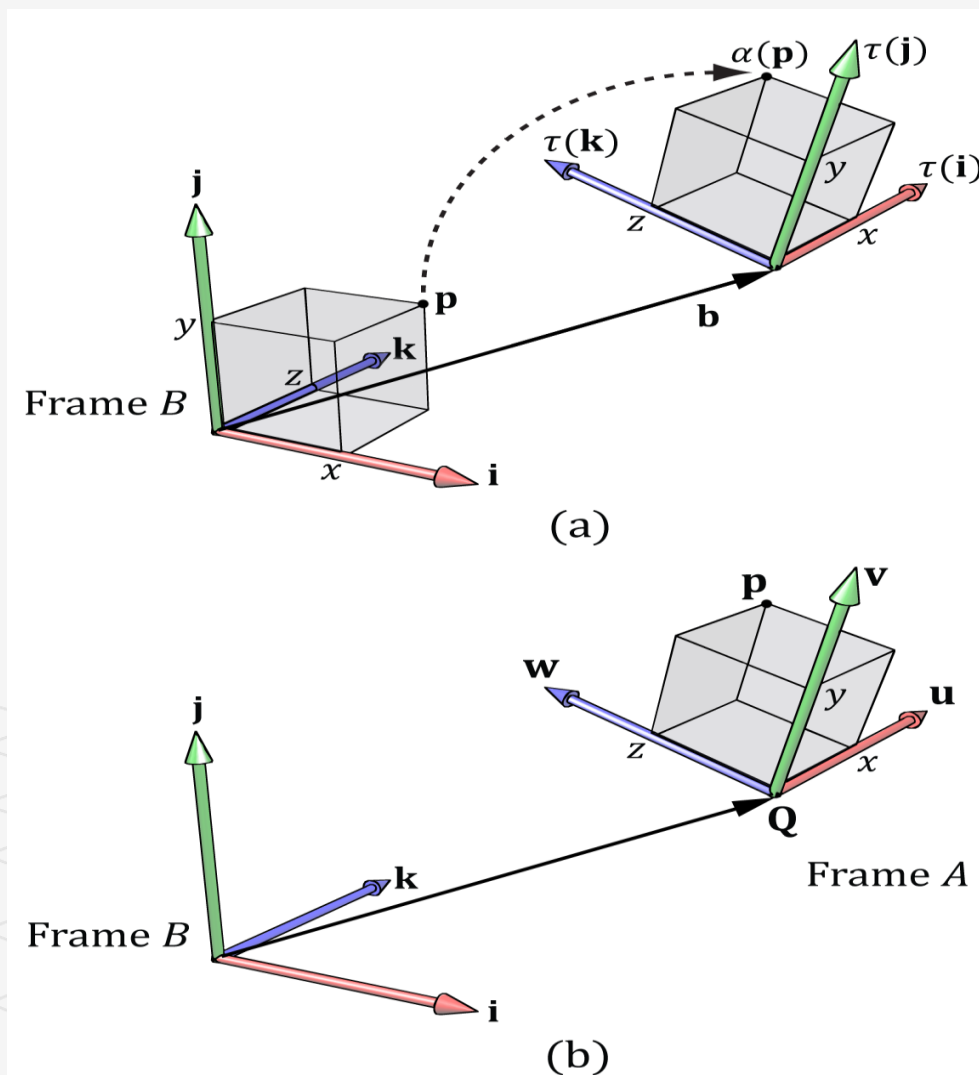
(a) We work with one coordinate system, call it frame B, and we apply an affine transformation to the cube to change its position and orientation relative to frame B:

$$\alpha(x, y, z, w) = x\tau(\mathbf{i}) + y\tau(\mathbf{j}) + z\tau(\mathbf{k}) + w\mathbf{b}.$$

(b) We have two coordinate systems called frame A and frame B. The points of the cube relative to frame A can be converted to frame B coordinates by the formula

$$\mathbf{p}_B = x\mathbf{u}_B + y\mathbf{v}_B + z\mathbf{w}_B + w\mathbf{Q}_B, \text{ where } \mathbf{p}_A = (x, y, z, w).$$

In both cases, we have $\alpha(\mathbf{p}) = (x', y', z', w) = \mathbf{p}_B$ with coordinates relative to frame B.



DirectX Functions

DirectX Math has the following transformation functions:

- XMMatrixScaling()
- XMMatrixScalingFromVector()
- XMMatrixRotationX()
- XMMatrixRotationY()
- XMMatrixRotationZ()
- XMMatrixRotationAxis()
- XMMatrixTranslation()
- XMMatrixTranslationFromVector()
- XMVector3TransformCoord()
- XMVector3TransformNormal()