Game Graphic Programming

Kyung Hee University

Software Convergence

Prof. Daeho Lee

Graphics APIs

- What's the Difference Between Vulkan, OpenGL, and DirectX
  - **DirectX** is a proprietary graphics API developed by Microsoft for use on the **Windows operating system**. It is primarily used for developing games and other graphics-intensive applications.
  - **OpenGL** is a **cross-platform** graphics API that was developed by Silicon Graphics Inc. (SGI) in the early 1990s. It is widely used in a variety of applications, including video games, CAD software, and scientific visualization.
  - **Vulkan** is a **cross-platform** graphics API that was developed by the Khronos Group, a consortium of graphics hardware and software companies. It was designed to provide a more efficient, **lower-level alternative to OpenGL**, with the goal of minimizing the number of API calls required to draw a frame.
  - https://pcgazer.com/2023/02/01/whats-the-difference-between-vulkan-opengl-and-directx/
  - **Metal** is a modern, tightly integrated graphics and compute API coupled with a powerful shading language that is designed and optimized for **Apple platforms**. [https://developer.apple.com/metal/]

DirectX 11 vs. DirectX 12

- Between DirectX 11 and DirectX 12, the most important difference is that DirectX 11 is a high-level API, while DirectX 12 is a low-level API. There are various layers between your game and your hardware. Low-level APIs are closer to the hardware, while high-level APIs are further away and more generalized.
- DirectX 12 allows game developers to target optimizations closer to the hardware, reducing the overhead incurred from the API and graphics driver. In turn, it's also more difficult for developers to work with.
- A significant change in DirectX 12 is parallel computing. DirectX 11 handles serial operations, which means there's a single queue of operations that execute in order. Parallel computing opens up the option for developers to make multiple calls at the same time, vastly improving the efficiency of operations.
- https://www.digitaltrends.com/computing/directx-12-vs-directx-11/#dt-heading-graphics-apis-arent-equal

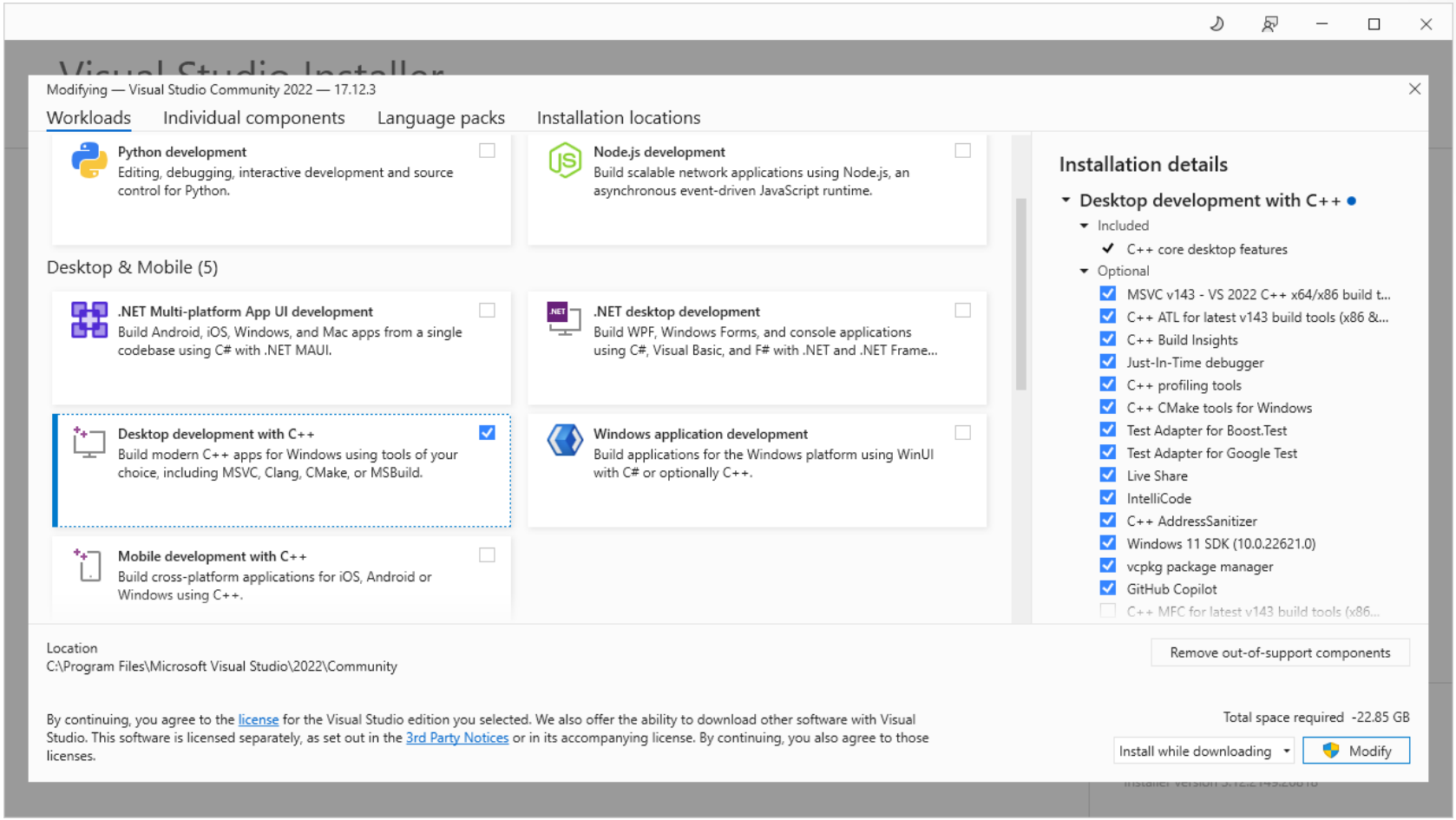
Text

- Introduction to 3D Game Programming with DirectX® 12
  - F. D. Luna, Mercury Learning
  - Sources
    - https://github.com/d3dcoder/d3d12book
    - https://github.com/yottaawesome/intro-to-dx12

- Reference
  - Direct3D 12 graphics
    - https://learn.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics

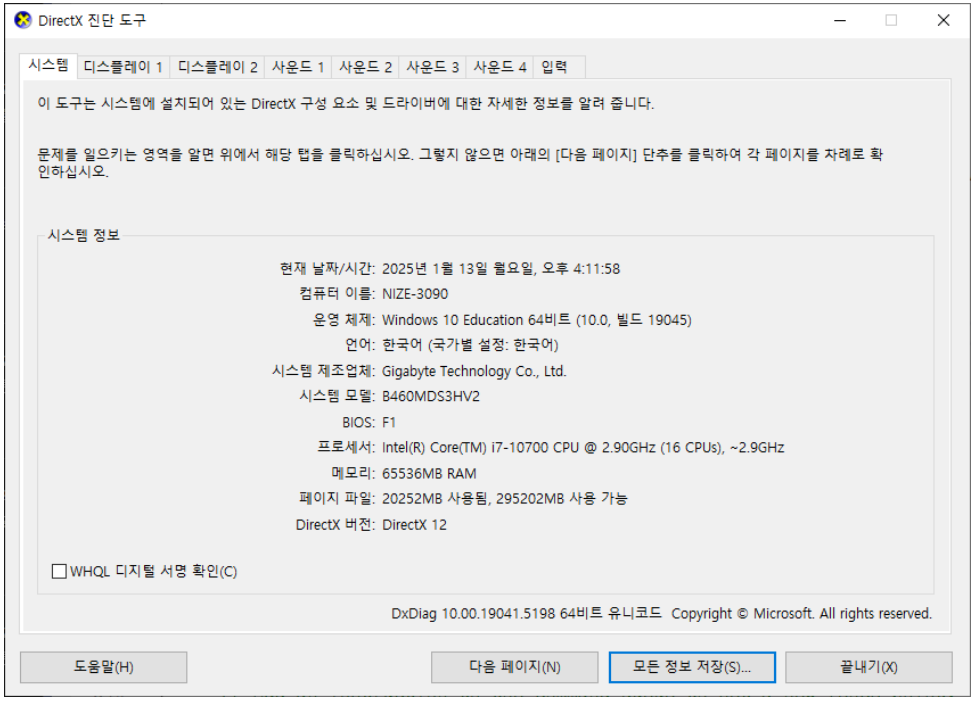
Visual Studio Installer

- [Desktop development with C++] – [Windows 11 SDK]



DxDiag

- DirectX Diagnostic
  - A diagnostics tool used to test DirectX functionality and troubleshoot video- or sound-related hardware problems.



I. Mathematical Prerequisites
1. Vector Algebra

Game Graphic Programming

Kyung Hee University

Software Convergence

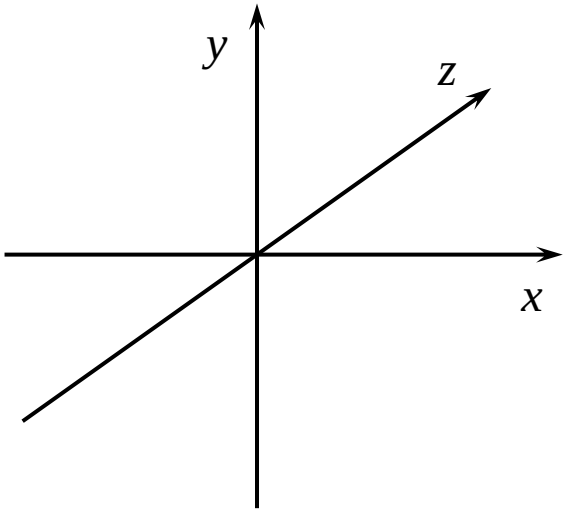
Prof. Daeho Lee

Vectors

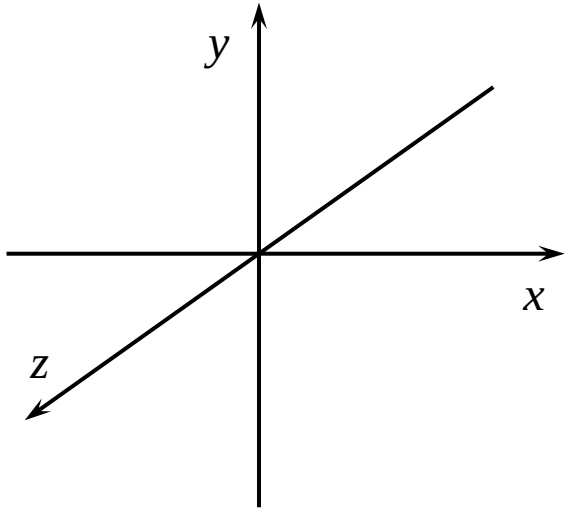
- A vector refers to a quantity that possesses both magnitude and direction.

$$\mathbf{v} = (x, y, z) \qquad \mathbf{v} = (x, y)$$

- Left-handed vs. right-handed coordinate systems
  - Direct3D uses a left-handed coordinate system.



left-handed coordinate system

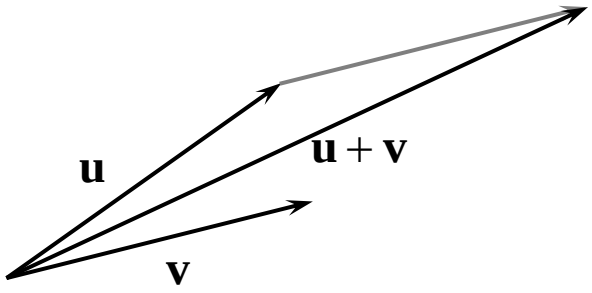


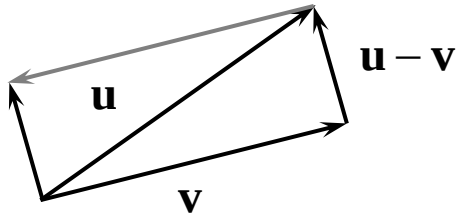
right-handed coordinate system

Basic Vector Operations

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

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





Length and Unit Vectors

- The magnitude of a vector is the length of the directed line segment.
  - The magnitude is denoted by double vertical bars ||.
- A unit vector is a vector that has a magnitude equal to 1.
  - The unit vectors are denoted by the cap symbol ^.
  - The unit vector can be calculated by dividing the vector by its magnitude.
    - Normalizing

$$\mathbf{v} = (x, y, z)$$

$$\|\mathbf{v}\| = \sqrt{x^2 + y^2 + z^2}$$

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

$$\|\hat{\mathbf{v}}\| = 1$$

Dot Product

- The dot product is a form of vector multiplication that results in a scalar value, that is, the sum of the products of the corresponding components.

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

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z$$

$$= \|\mathbf{u}\| \|\mathbf{v}\| \cos\theta$$

  - Geometric properties
    - If $\mathbf{u} \cdot \mathbf{v} = 0$, then $\mathbf{u} \perp \mathbf{v}$
    - If $\mathbf{u} \cdot \mathbf{v} > 0$, then $\theta < 90°$
    - If $\mathbf{u} \cdot \mathbf{v} < 0$, then $\theta > 90°$
  - $\mathbf{v}$ and unit vector $\hat{\mathbf{n}}$

$$\mathbf{p} = (\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} = \left(\|\mathbf{v}\| \cos\theta\right)\hat{\mathbf{n}}$$

$$= \text{proj}_{\hat{\mathbf{n}}}(\mathbf{v})$$

    - **Orthogonal projection** of $\mathbf{v}$ on $\hat{\mathbf{n}}$

$$\mathbf{p} = \text{proj}_{\mathbf{n}}(\mathbf{v}) = \left(\mathbf{v} \cdot \frac{\mathbf{n}}{\|\mathbf{n}\|}\right)\frac{\mathbf{n}}{\|\mathbf{n}\|} = \frac{(\mathbf{v} \cdot \mathbf{n})}{\|\mathbf{n}\|^2}\mathbf{n}$$

Perpendicular to $\mathbf{n}$

$$\text{perp}_{\mathbf{n}}(\mathbf{v}) = \mathbf{v} - \mathbf{p}$$

Orthogonalization

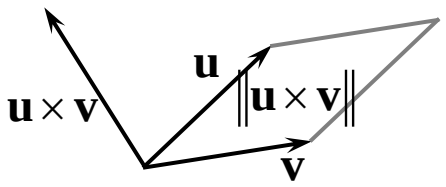
- A set of vectors is called orthonormal if the vectors are mutually orthogonal (every vector in the set is orthogonal to every other vector in the set) and unit length.
- **Orthogonalization** is the process of converting a set of basis vectors into an orthonormal set by ensuring that the vectors are mutually orthogonal and have unit length.
- Gram-Schmidt orthogonalization process: $\{\mathbf{v}_0, \cdots, \mathbf{v}_{n-1}\} \rightarrow \{\mathbf{w}_0, \cdots, \mathbf{w}_{n-1}\}$
  - Set $\mathbf{w}_0 = \mathbf{v}_0$
  - For $1 \le i \le n-1$, set $\mathbf{w}_i = \mathbf{v}_i - \sum_{j=0}^{i-1} \text{proj}_{\mathbf{w}_j}(\mathbf{v}_i)$
  - Set $\mathbf{w}_i = \frac{\mathbf{w}_i}{\|\mathbf{w}_i\|}$

Cross Product

- The cross product is defined as a vector that is perpendicular (orthogonal) to two vectors, with a direction given by the right-hand rule (right-handed coordinate) and a magnitude equal to the area of the parallelogram that the vectors span.

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

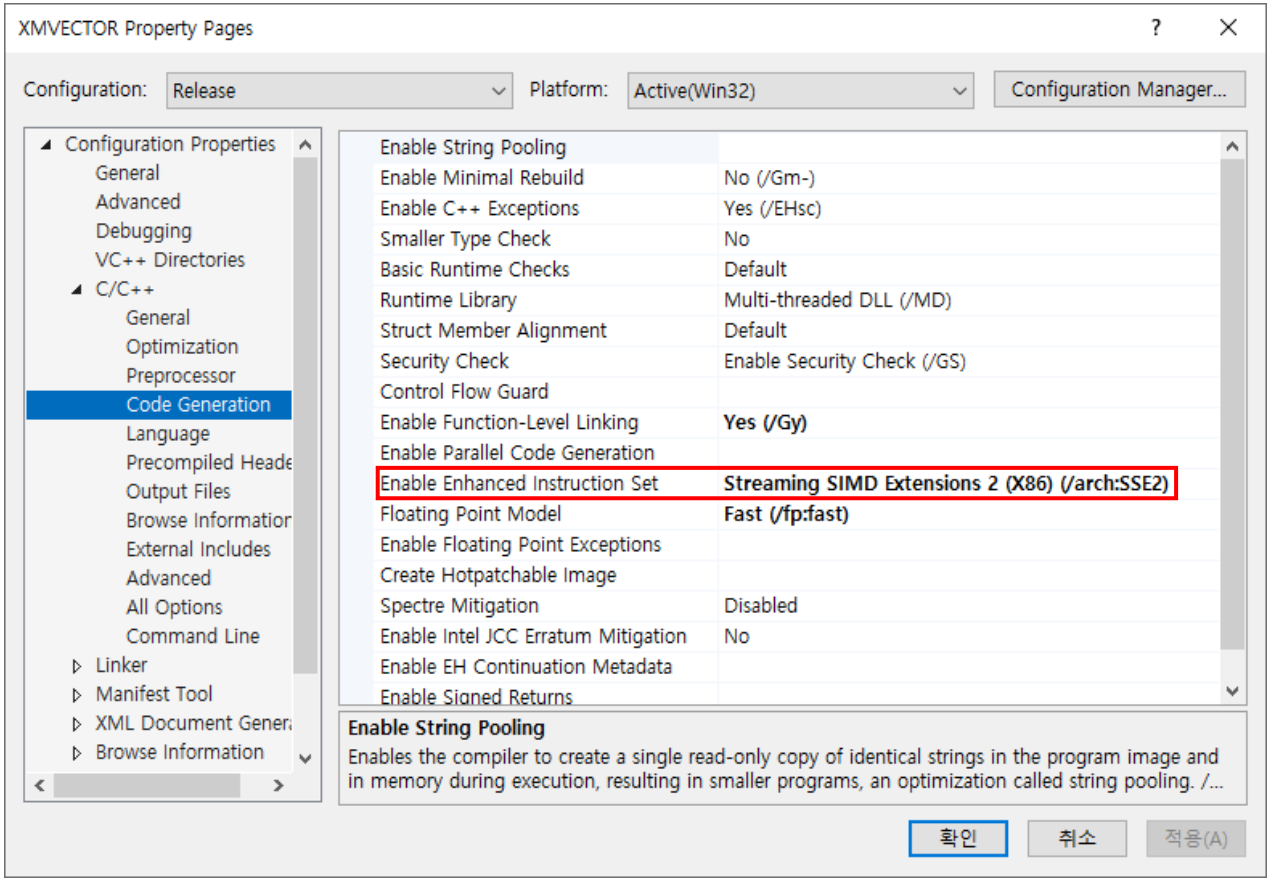
$$\mathbf{u} \times \mathbf{v} = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x)$$
$$= \left(\|\mathbf{u}\| \|\mathbf{v}\| \sin\theta \right) \hat{\mathbf{n}}$$



- Orthogonalization with the cross product: $\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\} \rightarrow \{\mathbf{w}_0, \mathbf{w}_1, \mathbf{w}_2\}$
  - Set $\mathbf{w}_0 = \dfrac{\mathbf{v}_0}{\|\mathbf{v}_0\|}$
  - Set $\mathbf{w}_2 = \dfrac{\mathbf{w}_0 \times \mathbf{v}_1}{\|\mathbf{w}_0 \times \mathbf{v}_1\|}$
  - Set $\mathbf{w}_1 = \mathbf{w}_2 \times \mathbf{w}_0$

Project Setting

- Project setting
  - [C/C++] – [Code Generation] – [Enable Enhanced Instruction Set] – SSE2



XMVECTOR Type

```
// XMVECTOR
#if defined(_XM_NO_INTRINSICS_)
    struct __vector4 {
        union {
            float       vector4_f32[4];
            uint32_t    vector4_u32[4];
        };
    };
#endif
#if defined(_XM_SSE_INTRINSICS_) && !defined(_XM_NO_INTRINSICS_)
    using XMVECTOR = __m128;    // SSE
#elif defined(_XM_ARM_NEON_INTRINSICS_) \
&& !defined(_XM_NO_INTRINSICS_) // for ARM SIMD
    using XMVECTOR = float32x4_t;
#else
    using XMVECTOR = __vector4; // _XM_NO_INTRINSICS_
#endif
```

Calling Conventions

- Calling conventions
  - To enhance portability and optimize data layout, you need to use the appropriate calling conventions for each platform supported by the DirectXMath Library. Specifically, when you pass XMVECTOR objects as parameters, which are defined as aligned on a 16-byte boundary, there are different sets of calling requirements, depending on the target platform.

  - Use the FXMVECTOR alias to pass up to the first three instances of XMVECTOR used as arguments to a function.
  - Use the GXMVECTOR alias to pass the 4th instance of an XMVECTOR used as an argument to a function.
  - Use the HXMVECTOR alias to pass the 5th and 6th instances of an XMVECTOR used as an argument to a function.
  - Use the CXMVECTOR alias to pass any further instances of XMVECTOR used as arguments.

  - **`#define XM_CALLCONV __vectorcall`**
    - The **`__vectorcall`** calling convention specifies that arguments to functions are to be passed in registers when possible.

struct XMFLOAT3 (1)

```
// 3D Vector; 32-bit floating point components
// For class data members, it is recommended to
//   use XMFLOAT2 (2D), XMFLOAT3 (3D), and XMFLOAT4 (4D) instead.
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;
    /* … */
};
```

struct XMFLOAT3 (2)

```
// Stores an XMVECTOR in an XMFLOAT3
void XM_CALLCONV XMStoreFloat3
    (XMFLOAT3 *pDestination, FXMVECTOR V) noexcept;

// Loads an XMFLOAT3 into an XMVECTOR
XMVECTOR XM_CALLCONV XMLoadFloat3
    (const XMFLOAT3 *pSource) noexcept;
```

DirectX Math Vector Code (1)

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

// Overload the  "<<" operators so that we can use cout to
// output XMVECTOR objects.
ostream& XM_CALLCONV operator << (ostream& os, FXMVECTOR v) {
    XMFLOAT3 dest;
    XMStoreFloat3(&dest, v);

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

DirectX Math Vector Code (2)

```
int main() {
    cout.setf(ios_base::boolalpha);

    // Check support for SSE2 (Pentium4, AMD K8, and above).
    if (!XMVerifyCPUSupport()) {
        cout << "directx math not supported" << endl;
        return 0;
    }
    XMVECTOR n = XMVectorSet(1.0f, 0.0f, 0.0f, 0.0f);
    XMVECTOR u = XMVectorSet(1.0f, 2.0f, 3.0f, 0.0f);
    XMVECTOR v = XMVectorSet(-2.0f, 1.0f, -3.0f, 0.0f);
    XMVECTOR w = XMVectorSet(0.707f, 0.707f, 0.0f, 0.0f);

    // Vector addition: XMVECTOR operator +
    XMVECTOR a = u + v;

    // Vector subtraction: XMVECTOR operator -
    XMVECTOR b = u - v;

    // Scalar multiplication: XMVECTOR operator *
    XMVECTOR c = 10.0f*u;
```

DirectX Math Vector Code (3)

```
    // ||u||
    XMVECTOR L = XMVector3Length(u);
// XMVECTOR XM_CALLCONV XMVector3Length(FXMVECTOR V) noexcept;
// It returns a vector. The length of V is replicated into
// each component.

    // d = u / ||u||
    XMVECTOR d = XMVector3Normalize(u);

    // s = u dot v
    XMVECTOR s = XMVector3Dot(u, v);

    // e = u x v
    XMVECTOR e = XMVector3Cross(u, v);
```

DirectX Math Vector Code (4)

```
    // Find proj_n(w) and perp_n(w)
    XMVECTOR projW;
    XMVECTOR perpW;
    XMVector3ComponentsFromNormal(&projW, &perpW, w, n);

    // Does projW + perpW == w?
    bool equal = XMVector3Equal(projW + perpW, w) != 0;
    bool notEqual = XMVector3NotEqual(projW + perpW, w) != 0;

    // The angle between projW and perpW should be 90 degrees.
    XMVECTOR angleVec = XMVector3AngleBetweenVectors(projW, perpW);
    float angleRadians = XMVectorGetX(angleVec);
    float angleDegrees = XMConvertToDegrees(angleRadians);
// XMVECTOR XM_CALLCONV XMVector3AngleBetweenVectors
//    (FXMVECTOR V1, FXMVECTOR V2) noexcept;
// It returns a vector.
// The radian angle between V1 and V2 is replicated to each of
// the components.
```

DirectX Math Vector Code (5)

```
    cout << "u                      = " << u << endl;
    cout << "v                      = " << v << endl;
    cout << "w                      = " << w << endl;
    cout << "n                      = " << n << endl;
    cout << "a = u + v              = " << a << endl;
    cout << "b = u - v              = " << b << endl;
    cout << "c = 10 * u             = " << c << endl;
    cout << "d = u / ||u||          = " << d << endl;
    cout << "e = u x v              = " << e << endl;
    cout << "L  = ||u||             = " << L << endl;
    cout << "s = u.v                = " << s << endl;
    cout << "projW                  = " << projW << endl;
    cout << "perpW                  = " << perpW << endl;
    cout << "projW + perpW == w     = " << equal << endl;
    cout << "projW + perpW != w     = " << notEqual << endl;
    cout << "angle                  = " << angleDegrees << endl;

    return 0;
}
```

DirectX Math Vector Code (6)

```
u                       = (1, 2, 3)
v                       = (-2, 1, -3)
w                       = (0.707, 0.707, 0)
n                       = (1, 0, 0)
a = u + v               = (-1, 3, 0)
b = u - v               = (3, 1, 6)
c = 10 * u              = (10, 20, 30)
d = u / ||u||           = (0.267261, 0.534522, 0.801784)
e = u x v               = (-9, -3, 5)
L  = ||u||              = (3.74166, 3.74166, 3.74166)
s = u.v                 = (-9, -9, -9)
projW                   = (0.707, 0, 0)
perpW                   = (0, 0.707, 0)
projW + perpW == w      = true
projW + perpW != w      = false
angle                   = 90
```