1. 3D 그래픽 기초(3D Graphics Fundamentals)

(8) DirectX Math

① DirectXMath

과거의 DirectX SDK는 벡터와 행렬과 관련한 수학 연산을 위한 라이브러리를 D3DX 9 Math, D3DX 10 Math, XNA Math 등의 형태로 제공하였다. 윈도우 8 이후의 운영체제에서 DirectX 라이브러리에 포함되지 않은 DirectXMath 수학 라이브러리를 따로 제공한다.

- SIMD(Single Instruction Multiple Data)

  SIMD(Single Instruction Multiple Data) 컴퓨터는 SIMD 명령어(Instruction)를 실행할 수 있는 컴퓨터이다. SIMD 컴퓨터(프로세서: Processor, CPU)는 같은 연산을 여러 데이터에 대하여 동시에 실행할 수 있는 다중 처리 요소를 가진 컴퓨터이다. SIMD 프로세서는 같은 연산을 여러 데이터에 대하여 동시에 실행할 수 있는 SIMD 명령어들을 가진다. 기본적으로 SIMD 명령어는 동시에 4개의 32-비트 실수(정수)를 연산(계산)할 수 있다. 현재 대부분의 컴퓨터는 SIMD 프로세서를 가지고 있다.

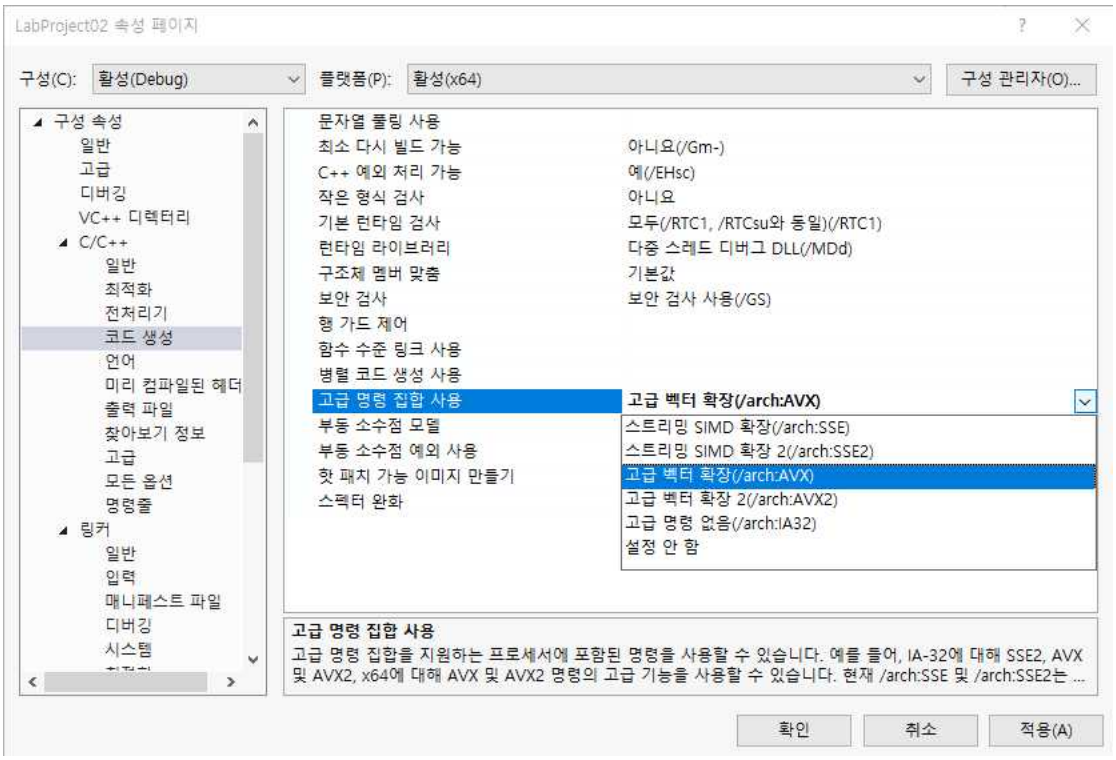
  두 개의 4-요소 벡터 $\boldsymbol{a}=(a_x, a_y, a_z, a_w)$와 $\boldsymbol{b}=(b_x, b_y, b_z, b_w)$를 더하는 경우를 생각해 보자. $\boldsymbol{a}+\boldsymbol{b}=(a_x+b_x, a_y+b_y, a_z+b_z, a_w+b_w)$를 계산하려면, SIMD 프로세서가 아닌 경우 32-비트 실수의 덧셈 명령을 4번 실행해야 한다. SIMD 프로세서는 두 개의 4-요소 벡터 $\boldsymbol{a}$와 $\boldsymbol{b}$의 덧셈을 한 번의 덧셈 명령을 실행하여 처리할 수 있다. 즉, SIMD 프로세서는 벡터의 연산(덧셈, 뺄셈, 스칼라 곱, 내적, 외적 등)과 행렬의 연산을 훨씬 빠르게 실행할 수 있다.

- DirectXMath

  윈도우 8 이후의 운영체제에서 윈도우즈 SDK(Windows SDK)는 DirectXMath 수학 라이브러리를 제공한다. DirectXMath는 SIMD 컴퓨터에서 SIMD 명령어를 사용할 수 있다. Visual Studio에서 Windows(x86/x64) 플랫폼에서 “고급 명령 집합 사용” 메뉴를 사용하여 SIMD 명령어를 사용하도록 프로그램을 컴파일할 수 있다. 최근의 대부분의 프로세서(예: Intel 또는 AMD CPU)는 벡터 연산을 위한 명령 집합 확장(Instruction set extension)을 제공한다.

  벡터 연산을 위한 명령 집합 확장을 사용하려면 Visual Studio에서 프로젝트 “속성 ▶ C/C++ ▶ 코드 생성 ▶ 고급 명령 집합 사용“에서 설정할 수 있다. 다음 그림과 같이 Visual Studio에서 벡터 연산을 위한 명령 집합 확장으로 ”SSE“, ”SSE2“, ”AVX“, ”AVX2“를 선택할 수 있다. 최근의 대부분의 프로세서는 기본적으로 ”SSE“(Streaming

SIMD Extensions) 또는 SSE2를 지원하며 "AVX"를 지원한다. Intel CPU의 경우 "펜티엄(Pentium) 4 모델부터 SSE2를 지원한다(아마도 2000년에 출시되었음).



- DirectXMath 라이브러리 사용하기

  DirectXMath 라이브러리는 윈도우즈 SDK에서 제공하므로 윈도우즈 SDK를 설치하면 사용할 수 있다. DirectXMath 라이브러리를 사용하려면 먼저 다음 헤더 파일을 프로젝트의 "stdafx.h" 파일에 포함(#include)한다.

```
#include <DirectXMath.h>
#include <DirectXPackedVector.h>
#include <DirectXColors.h>
#include <DirectXCollision.h>
```

  DirectXMath 라이브러리의 함수, 구조체, 클래스 들은 다음의 네임스페이스를 통하여 제공되므로 헤더 파일을 포함한 부분 다음에 다음 문장을 추가한다.

```
using namespace DirectX;
using namespace DirectX::PackedVector;
```

  응용 프로그램에서 컴퓨터의 프로세서가 SIMD의 기능(벡터 연산을 위한 명령 집합 확장 기능)을 제공하는 가를 확인하려면 다음 함수를 호출한다.

```
bool XMVerifyCPUSupport();
```

▪ DirectXMath 라이브러리의 구성 요소

DirectXMath 라이브러리는 4개의 부분으로 나뉘어 구성되어 있는 C++ 라이브러리이다.

❶ <DirectXMath.h>

벡터와 행렬과 관련된 함수, 구조체, 클래스 등을 제공한다.

❷ <DirectXPackedVector.h>

벡터와 관련된 팩킹된(Packed) 구조체를 제공한다.

❸ <DirectXColors.h>

색상(Color)의 정의를 제공한다.

❹ <DirectXCollision.h>

충돌 검사(Collision Detection)와 관련된 함수, 구조체, 클래스 등을 제공한다.

DirectXMath 라이브러리를 구성하는 헤더 파일(.h)과 구현 파일(.inl)의 소스 코드가 윈도우즈 SDK 설치 폴더에 제공된다. 꼭 폴더를 열어 "DirectX"로 시작하는 파일들을 열어서 내용을 살펴보기 바란다. 특히, "DirectXCollision.h"와 "DirectXCollision.inl" 파일은 반드시 잘 살펴보기를 바란다(두고 두고 오랫동안 쭉).

```
C:\Program Files (x86)\Windows Kits\10\Include\10.0.17763.0\um
```

② DirectXMath 라이브러리 상수(Constant)

다음은 DirectXMath 라디안(Radian) 상수이다.

XM_PI	π
XM_2PI	2π
XM_1DIVPI	1/π
XM_1DIV2PI	1/2π
XM_PIDIV2	π/2
XM_PIDIV4	π/4

다음 함수는 라디안(Radian) 각도를 도(Degree)로 변환한다.

```
float XMConvertToDegrees(float fRadians);
```

다음 함수는 도(Degree)를 라디안(Radian) 각도로 변환한다.

```
float XMConvertToRadians(float fDegrees);
```

③ XMVECTOR와 XMMATRIX

"XMVECTOR"는 SIMD의 기능을 사용할 수 있는 4차원 벡터를 표현하는 자료형이고, "XMMATRIX"는 SIMD의 기능을 사용할 수 있는 (4×4) 행렬을 표현하는 자료형이다.

SIMD의 기능을 사용하기 위하여 “XMVECTOR”와 “XMMATRIX” 자료형의 변수는 반드시 16-바이트 정렬이 되어야 한다.

Visual Studio 컴파일러는 “XMVECTOR”와 “XMMATRIX” 자료형의 지역변수(Local variable), 전역변수(Global variable), 함수의 매개변수(Parameter)는 16-바이트 정렬을 하도록 컴파일을 한다. 그러나 “XMVECTOR”와 “XMMATRIX” 자료형에 대한 동적 메모리 할당을 할 때와 클래스 또는 구조체의 멤버로 “XMVECTOR”와 “XMMATRIX” 자료형을 사용하면 16-바이트 정렬이 보장되지 않는다. Visual Studio 컴파일러는 솔루션 플랫폼이 “x64”이면 모든 동적 메모리 할당을 16-바이트로 정렬되도록 컴파일을 한다. 여러분이 사용하는 대부분의 컴퓨터는 64-비트 CPU를 가지고 있으며, 64-비트 운영체제(Windows 10)를 사용하고 있을 것이다(구입한지 10년이 넘지 않았으면 아마도 64-비트 CPU를 장착하고 있을 것이다). 그러므로 Visual Studio에서 솔루션 플랫폼을 반드시 “x64”로 설정하기 바란다. 그러면 “XMVECTOR”와 “XMMATRIX” 자료형의 동적 메모리 할당에 대하여 컴파일러가 메모리 주소를 16-바이트 정렬이 되게 할 것이므로 SIMD 연산을 사용할 수 있다.

클래스 또는 구조체의 멤버로 “XMVECTOR”와 “XMMATRIX” 자료형을 사용하면 16-바이트 정렬이 보장되지 않기 때문에 클래스 또는 구조체의 멤버로 “XMVECTOR”와 “XMMATRIX” 자료형을 사용하지 않아야 한다.

- “XMVECTOR”

  “XMVECTOR”는 각 요소가 32-비트 실수인 4차원 벡터를 표현하는 자료형이다. SIMD 프로세서의 하드웨어 레지스터에 대응되는 128-비트 자료형이다. 32-비트 실수 또는 정수 요소들의 4차원 벡터를 표현하기 위한 자료형이다. 128-비트의 메모리의 첫 번째 32-비트(실수)가 벡터의 x-요소, 그 다음 32-비트(실수)가 y-요소, 그 다음 32-비트(실수)가 z-요소, 나머지 32-비트(실수)가 w-요소가 된다. 구조체가 아님에 주의하라. 실제 이 자료형에 대한 실제 구현은 플랫폼(Platform)에 따라 다소 다르다. 다음과 같이 정의된다.

```
#if defined(_XM_SSE_INTRINSICS_) && !defined(_XM_NO_INTRINSICS_)
typedef __m128 XMVECTOR;
#else
typedef __vector4 XMVECTOR;
#endif
```

  “XMVECTOR”는 128-비트 자료형(128-비트 메모리를 나타내는)이므로 벡터의 요소에 직접 접근할 수 없다. 즉, 4차원 벡터의 x, y, z, w의 요소 값을 직접 읽거나 설정할 수 없다. 128-비트에서 4차원 벡터의 x, y, z, w의 요소를 읽거나 설정하기 위하여 XMVectorGet...() 함수와 XMVectorSet...() 함수를 사용해야 한다(이 함수들을 벡

터 액세서(Vector accessor)라고 한다).

C++에서 "XMVECTOR"는 다음과 같은 연산자 확장을 제공한다(C++의 연산자 오버로드(Overload)에 대하여 기본적인 이해를 하면 사용하기에 문제가 없을 것이다). 즉, 벡터 자료형 변수에 대하여 사칙연산자(+, -, /, *)를 사용할 수 있다. DirectXMath 라이브러리는 벡터의 연산을 전역 함수들로도 제공하고 있다. 성능을 고려한다면 연산자 오버로드를 가급적 사용하지 말 것을 권장한다.

```
XMVECTOR& operator +=(XMVECTOR &v1, XMVECTOR v2);
XMVECTOR& operator -=(XMVECTOR &v1, XMVECTOR v2);
XMVECTOR& operator *=(XMVECTOR &v, float s);
XMVECTOR& operator *=(XMVECTOR &v1, XMVECTOR v2);
XMVECTOR& operator /=(XMVECTOR &v, float s);
XMVECTOR& operator /=(XMVECTOR &v1, XMVECTOR v2);

XMVECTOR operator *(float s, XMVECTOR v);
XMVECTOR operator *(XMVECTOR v, float s);
XMVECTOR operator *(XMVECTOR v1, XMVECTOR v2);
XMVECTOR operator /(XMVECTOR v, float s);
XMVECTOR operator /(XMVECTOR v1, XMVECTOR v2);
XMVECTOR operator +(XMVECTOR v1, XMVECTOR v2);
XMVECTOR operator -(XMVECTOR v);
XMVECTOR operator -(XMVECTOR v1, XMVECTOR v2);
```

확장된 벡터 연산자의 연산은 기본적으로 요소별 연산이다. 곱셈(*) 연산자는 두 개의 4차원 벡터의 x, y, z, w 요소끼리 서로 곱하는 것이다(result.x = v1.x * v2.x).

```
XMVECTOR operator *(XMVECTOR v1, XMVECTOR v2);
```

C++에서 "XMVECTOR"는 다음과 같이 연산자 확장을 사용할 수 있다.

```
{
   XMVECTOR a = XMVectorSet(1.0f, 2.0f, 3.0f, 4.0f);
   XMVECTOR b = XMVectorSet(1.0f, 0.0f, -1.0f, 1.0f);
   XMVECTOR c = XMVectorSet(1.0f, 1.0f, 1.0f, 1.0f);
   a += 2.0f * b + c * 3.0f;
}
```

"XMVECTOR"와 형 변환을 할 수 있는 "XMVECTORF32" 구조체를 다음과 같이 제공한다.

```
struct XMVECTORF32 {
    union {
        float f[4];
        XMVECTOR v;
    };
    inline operator XMVECTOR() const { return v; }
    inline operator const float*() const { return f; }
};
```

```
{
   XMVECTORF32 vf32 = { 1.0f, 2.0f, 3.0f, 4.0f };
   XMVECTOR v = vf32;
}
```

- "XMMATRIX"

"XMMATRIX"는 4개의 SIMD 하드웨어 레지스터에 대응되는 행렬 구조체(struct)이다. "XMMATRIX" 구조체는 "XMVECTOR" 4개의 배열(XMVECTOR r[4])을 포함한다. 4개의 "XMVECTOR"는 순서대로 행렬의 행(Row)이다. "XMMATRIX" 구조체는 (4×4) 행렬(행우선(Row major) 행렬)을 표현하는 구조체이다. 행우선 행렬의 의미는 행렬이 표현하는 16개의 실수가 순서대로 첫 번째 행, 두 번째 행, 세 번째 행, 네 번째 행을 나타낸다는 것이다(C-언어의 2차원 배열과 유사하다). "XMMATRIX"는 128-비트 자료형(128-비트 메모리를 나타내는)의 배열이므로 행렬의 각 요소에 직접 접근할 수 없다. 행렬의 각 요소를 읽거나 설정하기 위하여 "XMVECTOR" 4개의 배열의 원소를 통하여 XMVectorGet...() 함수와 XMVectorSet...() 함수를 사용해야 한다.

"XMMATRIX"는 생성자와 연산자 오버로드를 멤버로 제공한다. 즉, 행렬 자료형 변수에 대하여 사칙연산자(+, -, /, *)를 사용할 수 있다. 확장된 행렬 연산자의 +, -, / 연산은 기본적으로 요소별 연산이다. 그러나 곱셈(*) 연산자는 두 개의 행렬의 곱셈(Multiply) 연산을 의미한다. DirectXMath 라이브러리는 행렬의 연산을 전역 함수들로도 제공하고 있다. 성능을 고려한다면 연산자 오버로드를 가급적 사용하지 말 것을 권장한다.

```
struct XMMATRIX {
   XMVECTOR r[4];

   XMMATRIX();
   XMMATRIX(float *pArray);
   XMMATRIX(float m00, float m01, float m02, …, float m32, float m33);
   XMMATRIX(XMVECTOR r0, XMVECTOR r1, XMVECTOR r2, XMVECTOR r3);

   XMMATRIX operator +(XMMATRIX m);
   XMMATRIX operator -(XMMATRIX m);
   XMMATRIX operator *(XMMATRIX m);
   XMMATRIX operator *(float s);
   XMMATRIX operator /(float s);

   XMMATRIX operator +();
   XMMATRIX operator -();
   XMMATRIX& operator +=(XMMATRIX m);
   XMMATRIX& operator -=(XMMATRIX m);
   XMMATRIX operator *=(XMMATRIX &m);
   XMMATRIX operator *(XMMATRIX &m);
   XMMATRIX& operator *=(float s);
   XMMATRIX& operator /=(float s);
```

```
    XMMATRIX& operator =(XMMATRIX &m);
};
```

C++에서 “XMMATRIX”는 다음과 같이 연산자 확장을 사용할 수 있다.

```
{
    XMVECTOR a = XMVectorSet(1.0f, 2.0f, 3.0f, 4.0f);
    XMVECTOR b = XMVectorSet(1.0f, 0.0f, -1.0f, 1.0f);
    XMVECTOR c = XMVectorSet(1.0f, 1.0f, 1.0f, 1.0f);
    XMVECTOR d = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);

    XMMATRIX m(a, b, c, d);
    m += m * 2.0f;

    b = m.r[0];
}
```

④ DirectXMath 사용하기

16-바이트 정렬이 되어야 하는 “XMVECTOR”와 “XMMATRIX”를 함수의 입력 매개변수(Input parameter)로 사용할 때, 호환성을 향상시키고 데이터 레이아웃을 최적화하기 위하여 플랫폼에 따라 적당한 함수 호출 규칙(Calling convention)을 사용할 필요가 있다.

32-비트 Windows에서 “XMVECTOR” 자료형의 매개변수의 값(Argument)을 효율적으로 전달하기 위하여 2가지 함수 호출 규칙을 사용할 수 있다.

- “__fastcall”

  “XMVECTOR” 매개변수의 값을 처음 3개는 SSE/SSE2 레지스터로 전달하고 나머지는 스택(Stack)으로 전달한다. 32-비트 Windows에서 사용하는 기본 함수 호출 규칙이다.

- “__vectorcall”

  “XMVECTOR” 매개변수의 값을 처음 6개까지 SSE/SSE2 레지스터로 전달하고 나머지는 스택으로 전달한다. “XMMATRIX” 매개변수의 값을 SSE/SSE2 레지스터로 전달할 수 있다.

64-비트 Windows에서 “XMVECTOR” 자료형의 매개변수를 효율적으로 전달하기 위하여 2가지 함수 호출 규칙을 사용할 수 있다.

- “__fastcall”

  모든 “XMVECTOR” 매개변수의 값을 스택으로 전달한다. 64-비트 Windows에서 사용하는 기본 함수 호출 규칙이다.

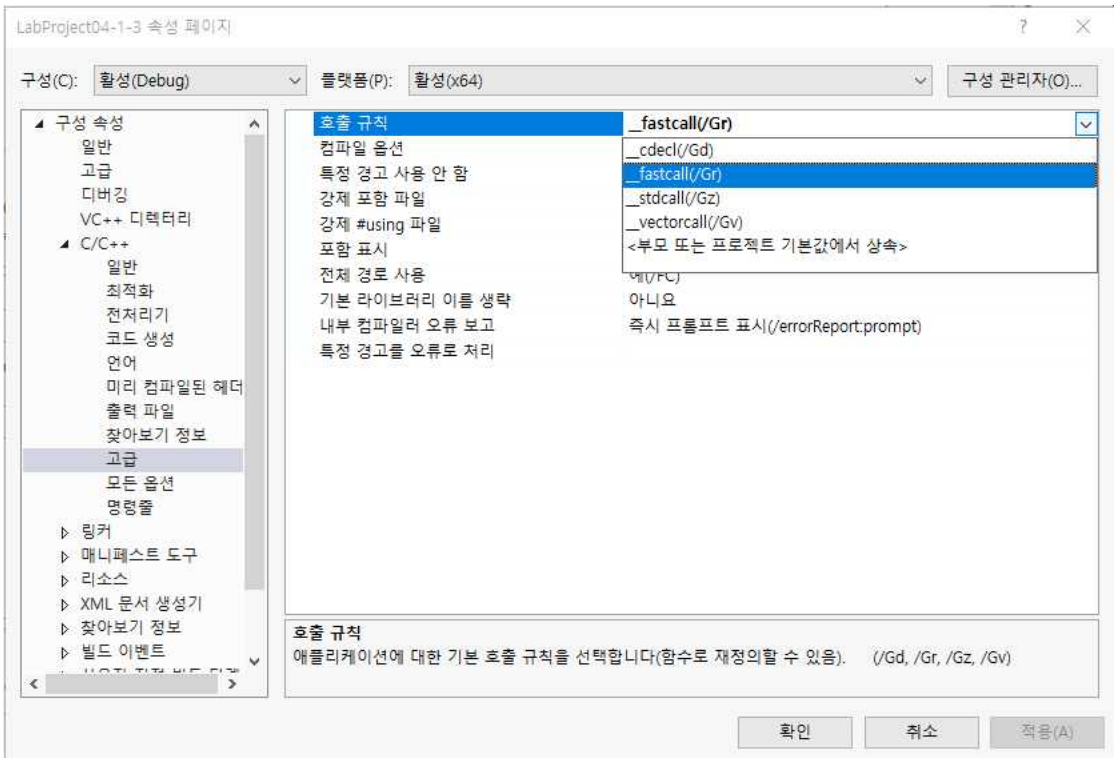
- “__vectorcall”

  ”XMVECTOR” 매개변수의 값을 처음 6개 까지 SSE/SSE2 레지스터로 전달하고 나

머지는 스택으로 전달한다. “XMMATRIX” 매개변수의 값을 SSE/SSE2 레지스터로 전달할 수 있다.

여러분은 대부분(실제로 모두) 64-비트 Windows를 사용하고 있으므로 Visual Studio에서 프로젝트의 “솔루션 플랫폼(Solution Platform)”을 “x64”로 설정하고, 함수 호출 규칙을 “__vectorcall”로 설정하기 바란다.

Visual Studio에서 프로젝트의 함수 호출 규칙을 설정하려면 프로젝트 “속성 ▶ C/C++ ▶ 고급 ▶ 호출 규칙“에서 설정할 수 있다.



DirectXMath는 이러한 함수 호출 규칙을 지원하기 위하여 64-비트 윈도우 응용 프로그램에서 사용할 수 있는 다음 자료형을 제공한다.

```
typedefconst XMVECTOR  FXMVECTOR;
typedefconst XMVECTOR  GXMVECTOR;
typedefconst XMVECTOR  HXMVECTOR;
typedefconst XMVECTOR& CXMVECTOR;
typedefconst XMMATRIX  FXMMATRIX;
typedefconst XMMATRIX& CXMMATRIX
```

DirectXMath 라이브러리의 대부분의 함수는 인라인(Inline) 함수로 선언되어 있으므로 대부분의 경우 DirectXMath 라이브러리의 함수 호출에는 이러한 함수 호출 규칙이 적용되지 않을 것이다.

“XMVECTOR” 또는 “XMMATRIX”를 매개변수를 통하여 전달받는 함수(인라인 함수가 아닌 경우)를 작성할 때, 다음과 같은 함수 호출 규칙을 사용하여 함수를 작성할 것을 권장한다.

- 함수가 적절한 함수 호출 규칙을 사용할 수 있도록 XM_CALLCONV을 사용하라. 다만, “__vectorcall”의 제약 때문에 C++의 생성자에는 XM_CALLCONV을 사용하지 않는다.

```
XMVECTOR XM_CALLCONV XMVector3Transform(FXMVECTOR V, FXMMATRIX M);
```

- 함수에서 “XMVECTOR”를 매개변수의 값으로 전달할 때, 처음 3개를 전달하기 위해 “XMVECTOR” 대신에 “FXMVECTOR” 자료형을 사용하라.
- 4번째 “XMVECTOR”를 전달하기 위해 “GXMVECTOR” 자료형을 사용하라.
- 5번째와 6번째 “XMVECTOR”를 전달하기 위해 “HXMVECTOR” 자료형을 사용하라. 나머지 “XMVECTOR”를 전달하기 위해 “CXMVECTOR” 자료형을 사용하라.
- “__vectorcall”의 제약 때문에 “GXMVECTOR”와 “HXMVECTOR” 자료형을 사용하지 말고 “FXMVECTOR”와 “CXMVECTOR” 자료형을 사용할 것을 권장한다.
- 생성자의 처음 3개의 파라메터에 “FXMVECTOR”를 사용하고, 나머지는 CXMVECTOR 사용하라.
- 출력 매개변수(Output parameter)에 항상 “XMVECTOR *” 또는 “XMVECTOR&” 자료형을 사용하라.
- 함수에서 “XMMATRIX”를 처음 매개변수의 값으로 전달할 때, “XMMATRIX” 대신에 “FXMMATRIX” 자료형을 사용하라. 나머지 “XMMATRIX”를 매개변수의 값으로 전달하기 위해 “CXMMATRIX” 자료형을 사용하라.
- “__vectorcall”의 제약 때문에 생성자에 “FXMMATRIX” 자료형을 사용하지 말고 “CXMMATRIX” 자료형을 사용할 것을 권장한다.

다음은 위와 같은 함수 호출 규칙을 따라 선언된 함수들의 예이다.

```
XMMATRIX XM_CALLCONV XMMatrixLookAtLH(FXMVECTOR eye, FXMVECTOR at,
FXMVECTOR up);

void XM_CALLCONV XMVectorSinCos(XMVECTOR* pSin, XMVECTOR* pCos,
FXMVECTOR V);

ContainmentType XM_CALLCONV ContainedBy(FXMVECTOR Plane0, FXMVECTOR
Plane1, FXMVECTOR Plane2, CXMVECTOR Plane3, CXMVECTOR Plane4,
CXMVECTOR Plane5);

XMVECTOR XM_CALLCONV XMVector3Transform(FXMVECTOR V, FXMMATRIX M);

XMMATRIX XM_CALLCONV XMMatrixMultiplyTranspose(FXMMATRIX M1,
CXMMATRIX M2);
```

```
XMMATRIX(FXMVECTOR r0, FXMVECTOR r1, FXMVECTOR r2, CXMVECTOR r3);
```

⑤ XMVECTOR 또는 XMMATRIX의 로드(Load)와 저장(Store)

클래스 또는 구조체의 멤버로 "XMVECTOR"와 "XMMATRIX" 자료형을 사용하면 16-바이트 정렬이 보장되지 않기 때문에 클래스 또는 구조체의 멤버로 "XMVECTOR"와 "XMMATRIX" 자료형을 사용하지 않아야 한다.

클래스 또는 구조체의 멤버 변수가 벡터 또는 행렬일 때 사용할 수 있는 "XMFLOAT3", "XMFLOAT4", "XMFLOAT4X4" 등의 32-비트 실수들의 벡터 또는 행렬 구조체 자료형을 제공한다.

DirectXMath 라이브러리는 "XMINT4", "XMUINT4"와 같이 32-비트 정수들의 벡터 또는 행렬 구조체 자료형과 관련된 함수들을 제공한다. 우리는 32-비트 정수들의 벡터 또는 행렬 구조체를 거의 사용하지 않을 것이므로, 32-비트 실수들의 벡터 또는 행렬 구조체 자료형과 관련 함수들에 대해서만 이해하도록 한다.

"XMFLOAT3"는 실수 3개로 구성되는 3-차원 벡터를 나타내는 구조체이다. 구조체이므로 멤버(x, y, z)들에 대한 직접 접근이 당연히 가능하다. "XMFLOAT2"는 2-차원 벡터를 나타내는 구조체이며, "XMFLOAT4"는 4-차원 벡터를 나타내는 구조체이다. XMFLOAT3A, XMFLOAT4A와 같이 구조체 이름이 A로 끝나는 구조체의 변수들은 16-바이트 정렬이 된다는 의미이다.

```
struct XMFLOAT3
{
   float x; float y; float z;
   XMFLOAT3();
   XMFLOAT3(float *pArray);
   XMFLOAT3(float x, float y, float z);
   XMFLOAT3& operator =(const XMFLOAT3 &v);
};

struct XMFLOAT4
{
   float x; float y; float z; float w;
   XMFLOAT4();
   XMFLOAT4(float *pArray);
   XMFLOAT4(float x, float y, float z , float w);
   XMFLOAT4& operator =(const XMFLOAT4 &v);
};
```

"XMFLOAT4X4"는 (4×4) 행렬(행우선(Row major) 행렬)을 표현하는 구조체이다. 즉, m[2][3]는 3행 4열의 원소이고, 12번째 실수에 해당한다(2 * 4 + 4 = 12). 행렬이 표현하는 16개의 실수가 순서대로 열을 나타낸다면 열우선(Column major) 행렬이라고 한다.

"XMFLOAT3X3", "XMFLOAT4X3", "XMFLOAT2X2", "XMFLOAT3X3A" 등의 구조체들도 제공된다. 행렬에 대한 자료형 이름에서 'X'는 대문자이다.

```
struct XMFLOAT4X4
{
   union {
      struct {
         float _11; ... float _44;
      };
      float m[4][4];
   };
   XMFLOAT4X4();
   XMFLOAT4X4(float *pArray);
   XMFLOAT4X4(float m00, ..., float m33);
   XMFLOAT4X4& operator =(XMFLOAT4X4 &m);
   float& operator ()(size_t row, size_t column);
   float operator ()(size_t row, size_t column);
};
```

그러나 벡터와 행렬에 대한 연산을 할 때 SIMD 연산의 장점을 사용해야 한다. SIMD 연산은 "XMVECTOR" 또는 "XMMATRIX" 자료형에 대해서만 적용할 수 있다. 클래스 또는 구조체의 멤버 변수가 벡터 또는 행렬일 때, 벡터 또는 행렬에 대한 연산은 다음과 같은 과정으로 수행한다.

❶ 클래스 또는 구조체의 멤버를 XMVECTOR 또는 XMMATRIX 자료형의 지역 변수로 복사(Load)한다. 이러한 과정을 위하여 XMLoadFloat...() 함수들이 제공된다.

❷ XMVECTOR 또는 XMMATRIX 지역 변수를 사용하여 계산(SIMD 연산)한다.

❸ XMVECTOR 또는 XMMATRIX 지역 변수를 클래스 또는 구조체의 멤버 변수로 복사(Store)한다. 이러한 과정을 위하여 XMStoreFloat...() 함수들이 제공된다.

- 벡터 로드(Vector Load) 함수

  "XMFLOAT#" 자료형 변수의 값을 "XMVECTOR" 자료형 변수에 복사하기 위한 함수들을 벡터 로드 함수라고 한다(#은 1, 2, 3, 4의 값을 가질 수 있다). 로드 함수의 이름은 XMLoadFloat#, 함수의 반환 자료형은 "XMVECTOR", 함수의 파라메터는 "XMFLOAT# *" 자료형이다. 다음은 벡터 로드 함수의 예이다.

```
XMVECTOR XMLoadFloat(const float *pSource);
XMVECTOR XMLoadFloat3(const XMFLOAT3 *pSource);
XMVECTOR XMLoadFloat3A(const XMFLOAT3A *pSource);
XMVECTOR XMLoadFloat4(const XMFLOAT4 *pSource);
```

  다음의 예에서 벡터(XMVECTOR) $\boldsymbol{a}$의 x-요소는 1.0이고, y-요소는 2.0, z-요소는 3.0, w-요소는 4.0이 된다.

```
{
   XMFLOAT4 source(1.0f, 2.0f, 3.0f, 4.0f);
   XMVECTOR a = XMLoadFloat4(&source);
```

```
}
```

파라메터 자료형의 실수 개수가 4개 보다 작으면 0으로 채워진다. 다음의 예에서 벡터(XMVECTOR) $\boldsymbol{a}$의 x-요소는 3이고, y-요소, z-요소, w-요소는 0이 된다.

```
{
   float source = 3;
   XMVECTOR a = XMLoadFloat(&source);
}
```

- 행렬 로드(Matrix Load) 함수

"XMFLOAT#X#" 자료형 변수의 값을 "XMMATRIX" 자료형 변수에 복사하기 위한 함수들을 행렬 저장 함수라고 한다(#은 1, 2, 3, 4의 값을 가질 수 있다). 로드 함수의 이름은 XMLoadFloat#x#, 함수의 반환 자료형은 "XMMATRIX", 함수의 파라메터는 "XMFLOAT#X# *" 자료형이다. 다음은 벡터 로드 함수의 예이다. 행렬 로드 함수 이름에서 'x'는 소문자이다.

```
XMMATRIX XMLoadFloat4x4(XMFLOAT4X4 *pSource);
XMMATRIX XMLoadFloat3x3(XMFLOAT3X3 *pSource);
XMMATRIX XMLoadFloat4x4A(XMFLOAT4X4A *pSource);
XMMATRIX XMLoadFloat4x3(XMFLOAT4X3 *pSource);
```

파라메터 자료형의 실수 개수가 16개 보다 작으면 아핀 행렬이 되도록 채워진다. 다음의 예에서 행렬(XMMATRIX) $\boldsymbol{a}$의 4 번째 열은 (0, 0, 0, 1), 4 번째 행이 (0, 0, 0, 1)이 된다.

```
{
   XMFLOAT3X3 source;
   ...
   XMMATRIX a = XMLoadFloat3x3(&source);
}
```

- 벡터 저장(Vector Store) 함수

"XMVECTOR" 자료형 변수의 값을 "XMFLOAT#" 자료형 변수에 복사하기 위한 함수들을 벡터 저장 함수라고 한다. 저장 함수의 이름은 XMStoreFloat#, 함수의 반환 자료형은 void, 함수의 파라메터는 "XMFLOAT# *"와 "XMVECTOR" 자료형이다. 다음은 벡터 저장 함수의 예이다.

```
void XMStoreFloat(float *pDestination, XMVECTOR v);
void XMStoreFloat3(XMFLOAT3 *pDestination, XMVECTOR v);
void XMStoreFloat4(XMFLOAT4 *pDestination, XMVECTOR v);
void XMStoreFloat4A(XMFLOAT4A *pDestination, XMVECTOR v);
```

저장 함수는 *pDestination 변수의 실수 개수만큼 벡터(XMVECTOR)의 x-요소, y-요소, z-요소, w-요소를 순서대로 복사한다. 다음 예에서 source에 벡터 $\boldsymbol{a}$의 x-요소가 복사된다.

```
{
   float source = 0;
   XMVECTOR a;
   XMStoreFloat(&source, a);
}
```

- 행렬 저장(Matrix Store) 함수

  "XMMATRIX" 자료형 변수의 값을 "XMFLOAT#X#" 자료형 변수에 복사하기 위한 함수들을 행렬 저장 함수라고 한다. 저장 함수의 이름은 XMStoreFloat#x#, 함수의 반환 자료형은 void, 함수의 파라메터는 "XMFLOAT#X# *"와 "XMMATRIX" 자료형이다. 행렬 저장 함수는 *pDestination 변수의 자료형에 대응되는 행렬(XMVECTOR)의 요소를 복사한다. 다음은 행렬 저장 함수의 예이다.

```
void XMStoreFloat4x3(XMFLOAT4X3 *pDestination, XMMATRIX m);
void XMStoreFloat4x3A(XMFLOAT4X3A *pDestination, XMMATRIX m);
void XMStoreFloat4x4(XMFLOAT4X4 *pDestination, XMMATRIX m);
```

- 색상 로드와 저장(Color Load/Store) 함수

  색상은 (r, g, b, a) 형태의 벡터로 취급할 수 있고, 색상에 대한 연산(예: 두 색상을 더하기)도 벡터의 연산으로 계산할 수 있으므로 SIMD 벡터 연산을 할 수 있다. 클래스 또는 구조체의 멤버 변수가 색깔을 나타내는 경우 "XMCOLOR" 구조체를 사용할 수 있다. "XMCOLOR" 자료형의 색상은 A8R8G8B8의 정규화된 정수의 형태로 32-비트로 팩킹되어 저장된다. 최상위 비트에서부터 알파(Alpha), R, G, B 요소의 순서이다.

```
[32] aaaaaaaa rrrrrrrr gggggggg bbbbbbbb [0]
```

```
struct XMCOLOR {
   union {
      struct {
         uint32_t b;
         uint32_t g;
         uint32_t r;
         uint32_t a;
      };
      uint32_t c; //A8R8G8B8
   };
   XMCOLOR();
   XMCOLOR(uint32_t Color);
   XMCOLOR(float *pArray);
   XMCOLOR(float r, float g, float b, float a);
   uint32_t& operator uint32_t();
   XMCOLOR& operator = (const XMCOLOR &color);
};
```

"XMCOLOR" 자료형 변수의 값을 "XMVECTOR" 자료형 변수에 복사하기 위한 함수의 이름은 XMLoadColor, 함수의 반환 자료형은 "XMVECTOR", 함수의 파라메터는

"XMCOLOR *" 자료형이다. "XMVECTOR" 자료형 변수의 값을 "XMCOLOR" 자료형 변수에 복사하기 위한 함수의 이름은 XMStoreColor, 함수의 반환 자료형은 void, 함수의 파라메터는 "XMCOLOR *"와 "XMVECTOR" 자료형이다.

```
XMVECTOR XMLoadColor(XMCOLOR *pSource);
void XMStoreColor(XMCOLOR *pDestination, XMVECTOR v);
```

"Colors" 네임스페이스를 통하여 표준 색상을 표현하는 이름을 제공한다. 각 이름의 자료형은 "XMVECTORF32" 구조체(float [4] 배열을 멤버로 가지고 있다)이다.

```
Colors::Black, Colors::Beige, Colors::Blue, Colors::DarkViolet, …
```

"XMVECTORF32" 구조체는 "XMVECTOR" 자료형과 형변환을 할 수 있으므로 "XMVECTOR" 변수에 표준 색상 이름을 다음과 같이 대입할 수 있다.

```
XMVECTOR c = Colors::AliceBlue;
```

⑥ 벡터 초기화(Initialize) 함수

"XMVECTOR"와 형 변환을 할 수 있는 "XMVECTORF32" 구조체를 사용하여 "XMVECTOR" 변수를 다음과 같이 초기화 할 수 있다.

```
XMVECTORF32 v1 = { 1.0f, 2.0f, 3.0f, 4.0f };
XMVECTOR v2 = v1;
```

DirectXMath는 미리 정의된 전역 상수들을 제공한다. 이 상수들은 XMVECTORF32 자료형이므로 "XMVECTOR" 변수를 초기화하기 위하여 사용할 수 있다. 다음은 이러한 상수들의 예이다. 나머지 상수들은 DirectXMath.h 헤더 파일을 참고하라.

```
XMGLOBALCONST XMVECTORF32 g_XMOne = { 1.0f, 1.0f, 1.0f, 1.0f};
XMGLOBALCONST XMVECTORF32 g_XMOne3 = { 1.0f, 1.0f, 1.0f, 0.0f};
XMGLOBALCONST XMVECTORF32 g_XMZero = { 0.0f, 0.0f, 0.0f, 0.0f};
XMGLOBALCONST XMVECTORF32 g_XMPi = {XM_PI, XM_PI, XM_PI, XM_PI};
```

"XMVECTOR" 자료형 변수의 값을 초기화하기 위한 함수는 다음과 같다.

다음 함수는 영벡터 (0, 0, 0, 0)를 반환한다.

```
XMVECTOR XMVectorZero();
```

다음 함수는 벡터 (x, y, z, w)를 반환한다.

```
XMVECTOR XMVectorSet(float x, float y, float z, float w);
```

다음 함수는 벡터 (value, value, value, value)를 반환한다.

```
XMVECTOR XMVectorReplicate(float value);
```

다음 함수는 벡터 (*pValue, *pValue, *pValue, *pValue)를 반환한다.

```
XMVECTOR XMVectorReplicatePtr(const float *pValue);
```

다음 함수는 벡터 (1.0f, 1.0f, 1.0f, 1.0f)를 반환한다.

```
XMVECTOR XMVectorSplatOne();
```

다음은 벡터 초기화 함수의 사용 예이다.

```
{
   XMVECTOR zero = XMVectorZero();
   XMVECTOR one = XMVectorSet(1.0f, 1.0f, 1.0f, 1.0f);
   XMVECTOR two = XMVectorReplicate(2.0f);
   float value = 3.0f;
   XMVECTOR zero = XMVectorZero(&value);
}
```

다음과 같이 벡터 접근자 함수를 사용하여 "XMVECTOR" 자료형 변수의 값을 초기화할 수 있다.

```
{
   XMVECTOR zero = XMVectorZero();
   XMVECTOR v = XMVectorSetW(zero, 1.0f);
}
```

- 벡터 컴포넌트(Component-Wise) 함수
  다른 "XMVECTOR" 자료형 변수의 요소로부터 새로운 벡터를 생성(초기화)할 수 있다.

  다음 함수 XMVectorSplatX()는 입력 벡터(v) (v_x, v_y, v_z, v_w)의 x-요소를 반복하여 벡터 (v_x, v_x, v_x, v_x)를 반환한다. 함수 XMVectorSplatY()는 입력 벡터(v)의 y-요소를 반복하여 벡터 (v_y, v_y, v_y, v_y)를 반환한다. 함수 XMVectorSplatZ()는 입력 벡터(v)의 z-요소를 반복하여 벡터 (v_z, v_z, v_z, v_z)를 반환한다. 함수 XMVectorSplatW()는 입력 벡터(v)의 w-요소를 반복하여 벡터 (v_w, v_w, v_w, v_w)를 반환한다.

```
XMVECTOR XMVectorSplatX(XMVECTOR v);
```

  다음 함수 XMVectorPermute()는 두 개의 벡터(v1, v2)의 요소로부터 새로운 벡터를 생성하여 반환한다. X, Y, Z, W는 0 ~ 7 또는 XM_PERMUTE_0X ~ XM_PERMUTE_1W의 값을 가질 수 있다. 0 ~ 3은 벡터 v1의 x-요소, y-요소, z-요소, w-요소를 선택하고, 4 ~ 7은 벡터 v2의 x-요소, y-요소, z-요소, w-요소를 선택하여 벡터의 값으로 설정한다.

```
const uint32_t XM_PERMUTE_0X = 0;
const uint32_t XM_PERMUTE_0Y = 1;
const uint32_t XM_PERMUTE_0Z = 2;
const uint32_t XM_PERMUTE_0W = 3;
const uint32_t XM_PERMUTE_1X = 4;
const uint32_t XM_PERMUTE_1Y = 5;
const uint32_t XM_PERMUTE_1Z = 6;
const uint32_t XM_PERMUTE_1W = 7;
```

```
template<uint32_t X, Y, Z, W> XMVECTOR XMVectorPermute(XMVECTOR v1,
v2);
```

다음은 벡터 $\boldsymbol{a} = (a_x, a_y, a_z, a_w)$와 벡터 $\boldsymbol{b} = (b_x, b_y, b_z, b_w)$에서 요소를 선택하여 벡터 $\boldsymbol{v} = (a_x, a_y, b_x, b_y)$를 반환한다.

```
v = XMVectorPermute<0, 1, 4, 5>(a, b);
v = XMVectorPermute<XM_PERMUTE_0X, XM_PERMUTE_0Y, XM_PERMUTE_1X,
XM_PERMUTE_1Y>(a, b);
```

다음 XMVectorSwizzle() 함수는 벡터(v)의 요소로부터 새로운 벡터를 생성하여 반환한다. SwizzleX, SwizzleY, SwizzleZ, SwizzleW는 0 ~ 3 또는 XM_SWIZZLE_X ~ XM_SWIZZLE_W의 값을 가질 수 있다. 0은 벡터 v의 x-요소, 1은 벡터 v의 y-요소, 2는 벡터 v의 z-요소, 3은 벡터 v의 w-요소를 선택한다.

```
const uint32_t XM_SWIZZLE_X = 0;
const uint32_t XM_SWIZZLE_Y = 1;
const uint32_t XM_SWIZZLE_Z = 2;
const uint32_t XM_SWIZZLE_W = 3;

template<uint32_t SwizzleX, SwizzleY, SwizzleZ, SwizzleW> XMVECTOR
XMVectorSwizzle(XMVECTOR v);
```

다음은 벡터 $\boldsymbol{v} = (v_x, v_y, v_z, v_w)$에서 요소를 선택하여 벡터 $\boldsymbol{r} = (v_w, v_w, v_x, v_z)$를 반환한다.

```
XMVECTOR r = XMVectorSwizzle<3, 3, 0, 2>(v);
```

⑦ 벡터 접근자(Accessor) 함수

"XMVECTOR" 자료형의 요소를 직접 접근할 수 없기 때문에 각 요소에 접근할 수 있는 함수(접근자: Accessor)들을 제공한다. x-요소, y-요소, z-요소, w-요소는 0.0 ~ 1.0의 값을 가진다.

다음 함수는 인덱스(Index)로 "XMVECTOR" 자료형의 요소를 반환한다. 인덱스(index)가 0이면 x-요소, 1이면 y-요소, 2이면 z-요소, 3이면 w-요소를 반환한다.

```
float XMVectorGetByIndex(XMVECTOR v, size_t index);
```

다음 함수는 인덱스(Index)로 "XMVECTOR" 자료형의 요소를 설정한다. 입력 벡터(v)의 인덱스(index)에 해당하는 요소를 실수(f)로 바꾸어 반환한다.

```
XMVECTOR XMVectorSetByIndex(XMVECTOR v, float f, size_t index);
```

다음 함수 XMVectorGetX()는 "XMVECTOR" 자료형의 x-요소를 반환한다. 함수 XMVectorGetY()는 y-요소, 함수 XMVectorGetZ()는 z-요소, 함수 XMVectorGetW()는

w-요소를 반환한다.

```
float XMVectorGetX(XMVECTOR v);
```

다음 함수 XMVectorSetX()는 입력 벡터(v)의 x-요소를 실수(x)로 바꾸어 반환한다. 함수 XMVectorSetY()는 y-요소, 함수 XMVectorSetZ()는 z-요소, 함수 XMVectorSetW()는 w-요소를 바꾸어 반환한다.

```
XMVECTOR XMVectorSetX(XMVECTOR v, float x);
```

다음 함수 XMVectorGetXPtr()는 "XMVECTOR" 자료형의 x-요소를 포인터(pX)를 통하여 반환한다. 함수 XMVectorGetYPtr()는 y-요소, 함수 XMVectorGetZPtr()는 z-요소, 함수 XMVectorGetWPtr()는 w-요소를 반환한다.

```
void XMVectorGetXPtr(float *pX, XMVECTOR v);
```

다음 함수 XMVectorSetXPtr()는 입력 벡터(v)의 x-요소를 실수(*pX)로 바꾸어 반환한다. 함수 XMVectorSetYPtr()는 y-요소, 함수 XMVectorSetZPtr()는 z-요소, 함수 XMVectorSetWPtr()는 w-요소를 바꾸어 반환한다.

```
XMVECTOR XMVectorSetXPtr(XMVECTOR v, float *pX);
```

⑧ 색상(Color) 함수

"XMVECTOR" 자료형이 색상을 표현할 때 x-요소, y-요소, z-요소, w-요소는 0.0 ~ 1.0의 값을 가진다. x-요소가 Red, y-요소가 Green, z-요소가 Blue, w-요소가 Alpha를 표현한다.

다음 함수들은 두 개의 색상을 비교(Comparison)하는 함수이다.

```
bool XMColorLess(XMVECTOR c1, XMVECTOR c2);
bool XMColorLessOrEqual(XMVECTOR c1, XMVECTOR c2);
bool XMColorEqual(XMVECTOR c1, XMVECTOR c2);
bool XMColorNotEqual(XMVECTOR c1, XMVECTOR c2);
bool XMColorGreater(XMVECTOR c1, XMVECTOR c2);
bool XMColorGreaterOrEqual(XMVECTOR c1, XMVECTOR c2);
```

다음 함수는 두 개의 색상을 요소별로 곱한 색상 (c1.x*c2.x, c1.y*c2.y, c1.z*c2.z, c1.w*c2.w)를 반환한다.

```
XMVECTOR XMColorModulate(XMVECTOR c1, XMVECTOR c2);
```

다음 함수는 색상의 보색(Complementary color) (1.0-c.x, 1.0-c.y, 1.0-c.z, 1.0-c.w)를 반환한다.

```
XMVECTOR XMColorNegative(XMVECTOR c);
```

다음 XMColorAdjustContrast() 함수는 입력 색상(c)의 콘트라스트(Contrast: 색상 대

비)를 조절하여 반환한다. 반환되는 색상(result)은 다음과 같다.

```
   result.x = (c.x - 0.5f) * contrast + 0.5f;
   result.y = (c.y - 0.5f) * contrast + 0.5f;
   result.z = (c.z - 0.5f) * contrast + 0.5f;
   result.w = c.w
```

```
XMVECTOR XMColorAdjustContrast(XMVECTOR c, float contrast);
```

다음 XMColorAdjustSaturation() 함수는 입력 색상(c)의 채도(Saturation)를 조절하여 반환한다. 반환되는 색상(result)은 다음과 같다.

```
XMVECTOR XMColorAdjustSaturation(XMVECTOR c, float saturation);
   float fLuminance = 0.2125f * c.x + 0.7154f * c.y + 0.0721f * c.z;
   result.x = (c.x - fLuminance) * saturation + fLuminance;
   result.y = (c.y - fLuminance) * saturation + fLuminance;
   result.z = (c.z - fLuminance) * saturation + fLuminance;
   result.w = c.w
```

다음 XMColorRGBToSRGB() 함수는 RGB 색상을 sRGB 색상으로 변환한다.

```
XMVECTOR XMColorRGBToSRGB(XMVECTOR rgb);
```

다음 XMColorSRGBToRGB() 함수는 sRGB 색상을 RGB 색상으로 변환한다.

```
XMVECTOR XMColorSRGBToRGB(XMVECTOR srgb);
```

다음 XMColorRGBToHSL() 함수는 RGB 색상을 HSL(Hue Saturation Luminance) 색상으로 변환한다.

```
XMVECTOR XMColorRGBToHSL(XMVECTOR rgb);
```

다음 XMColorHSLToRGB() 함수는 HSL 색상을 RGB 색상으로 변환한다.

```
XMVECTOR XMColorHSLToRGB(XMVECTOR hsl);
```

다음 XMColorRGBToHSV() 함수는 RGB 색상을 HSV(Hue Saturation Value) 색상으로 변환한다.

```
XMVECTOR XMColorRGBToHSV(XMVECTOR rgb);
```

다음 XMColorHSVToRGB() 함수는 HSV 색상을 RGB 색상으로 변환한다.

```
XMVECTOR XMColorHSVToRGB(XMVECTOR hsv);
```

다음 XMColorRGBToXYZ() 함수는 RGB 색상을 XYZ 색상으로 변환한다.

```
XMVECTOR XMColorRGBToXYZ(XMVECTOR rgb);
```

다음 XMColorXYZToRGB() 함수는 RGB 색상을 HSV 색상으로 변환한다.

```
XMVECTOR XMColorXYZToRGB(XMVECTOR hsv);
```