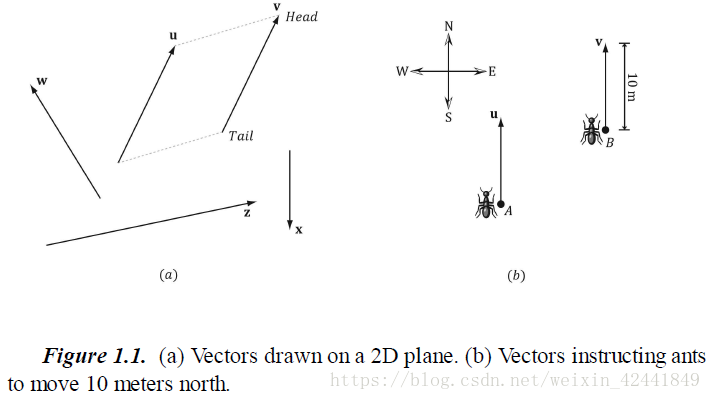
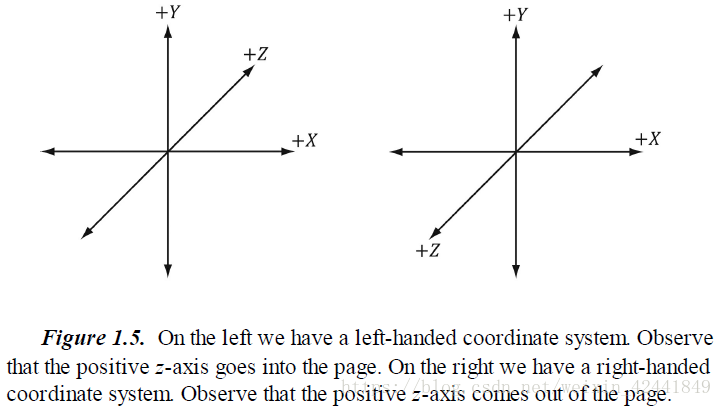
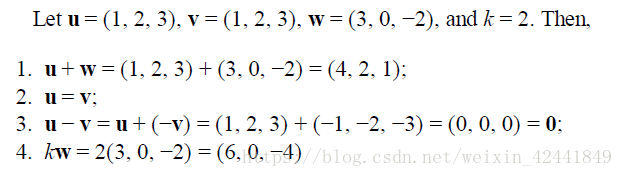
**第一章 向量**

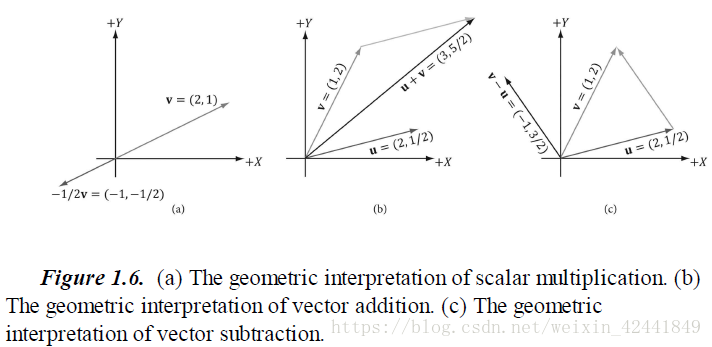
一个向量代表的是一个拥有大小和方向的量。类似力（拥有力的大小和方向）、位移（移动的方向和距离）、速度（速度的大小和方向）等。绘制向量的位置和向量本身无关，所以当且仅当两个向量的大小和方形相等时，两个向量相等

在计算机3D图形学中我们需要使用多个坐标系，我们要知道当前在哪一个坐标系下，并且熟悉坐标系之间的转换。

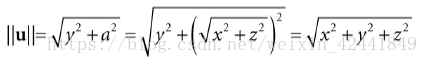
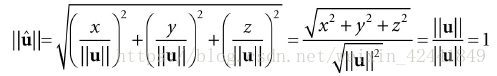
Direct3D 使用的是左手坐标系：如果你使用你的左手指向 X轴正方向，然后向Y轴正方向弯曲你的手指，此时你的大拇指指向的就是Z轴正方向，如图1.5 所示（右手坐标系类似，只不过替换为右手）

基本的向量运算





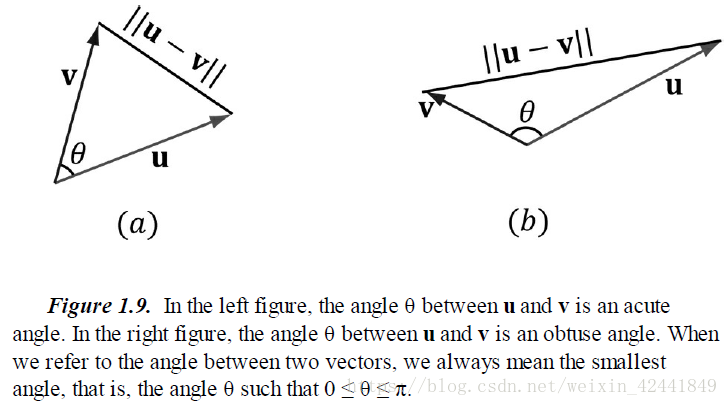
**长度和单位向量**

向量长度计算公式：  
  
单位向量计算公式：  
这里写图片描述  
为了证明单位向量计算公式，我们可以计算单位向量的长度：  


3 向量的点积

向量的点积是一种结果为数量值的乘法形式，其定义和计算公式为：https://img-blog.csdn.net/20180903172124112?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3dlaXhpbl80MjQ0MTg0OQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

向量的点积的定义并没有提出明显的几何定义，利用余弦定理，我们可以找到几何关系：https://img-blog.csdn.net/20180903172316466?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3dlaXhpbl80MjQ0MTg0OQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70



如图1.9，θ是向量v和u的夹角，再根据上面的公式，我们可以得出向量点积的一些有用的几何属性：

当两个向量点积为0时，两个向量垂直；

当两个向量点积大于0时，两个向量夹角小于90度；

当两个向量点积小于0时，两个向量夹角大于90度。

### 给出向量v和单位向量n，使用点积公式求出向量p： 这里写图片描述

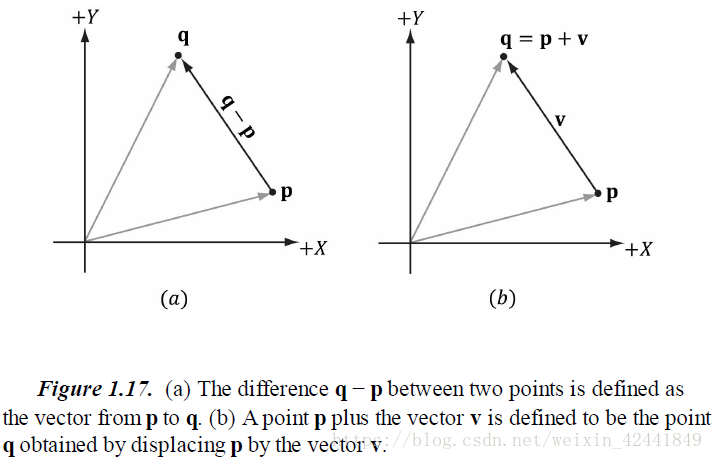
### 正交化

当一组向量相互之间都垂直，并且都是单位向量时，我们称他们为标准正交。在3D计算机图形学中，我们刚开始可能会有一组标准正交的向量，但是由于变量精度的问题，这些向量会在进行一系列计算后开始变得相互不垂直。所以我们的目标就是主要考虑在3D和2D情况下手动正交化。

## 向量的叉积

两个向量的叉积（叉积不支持2D）结果为另外一个同时垂直于他俩的向量，叉积的运算公式为：  
这里写图片描述  
叉积不支持交换律，其交换后的结果是相反的，即：u \* v = - v \* u。

## 点

点在3D图形学中要来表示位置，在坐标系中，一个向量可以表示一个位置。  
一方面，我们对向量的运算不能应用到点上（比如两个点相加是没有意义的）；另一方面，我们可以把这些运算扩展到点上。  


DIRECTX MATH 中的向量

Direct Math是Direct3D应用中的一个数学库，并且已经内置到了Window 8以上的操作系统。

该数学库使用 SSE2(Streaming SIMD Extensions 2) 系统指令，支持128位的 SIMD(single instruction multiple data) 寄存器。SIMD 指令可以在一条指令中运算4个32位的浮点数和整数。这对于向量的计算非常有用。

比如做4D向量的相加，我们不需要使用4条标量相加指令，而是1条SIMD指令即可，对于2D和3D向量也可以使用SIMD，我们可以无视不使用的坐标系。

如果想要了解DirectX Math的全部细节，推荐阅读DirectX Math的在线文档；

如果想要知道SIMD向量库如何优化开发，或者了解它为何如此设计，推荐阅读文章：Designing Fast Cross-Platform SIMD Vector Libraries by [Oliveira2010]。

使用DirectX Math时，需要的所有头文件：

#include <DirectXMath.h> // namespace: DirectX DirectX 数学库

#include <DirectXPackedVector.h>

// namespace:DirectX::PackedVector一些额外附加数据类型

对于X86系统，需要开启SSE2(Project Properties > Configuration Properties > C/C++ > Code Generation > Enable Enhanced Instruction Set)；

对于所有系统，还需要开启快速浮点数模式\*\*(Project Properties > Configuration Properties > C/C++ > Code Generation > Floating Point Model\*\*)；

对于64位系统不需要开启SSE2，因为所有64位CPU都支持SSE2(http://en.wikipedia.org/wiki/SSE2)。

向量的类型

DirectX Math 的核心类型是映射到SIMD硬件寄存器的 XMVECTOR，它是一个128位，可以使用单个指令计算4个32位浮点数的类型。对于X86和64位系统中，它的定义如下：

typedef \_\_m128 XMVECTOR;

\_\_m128是SIMD专用的类型。当我们计算的时候，向量必须声明位该类型才能利用SIMD的优点。

XMVECTOR的局部和全部变量会自动被16位对其；对于类的成员变量，使用XMFLOAT2 (2D)，XMFLOAT3 (3D)，和XMFLOAT4 (4D) 来替换；

struct XMFLOAT2

{

float x;

float y;

XMFLOAT2() {}

XMFLOAT2(float \_x, float \_y) : x(\_x), y(\_y) {}

explicit XMFLOAT2(\_In\_reads\_(2) const float \*pArray) : x(pArray[0]), y(pArray[1]) {}

XMFLOAT2& operator= (const XMFLOAT2& Float2) { x = Float2.x; y = Float2.y; return \*this; }

};

struct XMFLOAT3

{

float x;

float y;

float z;

XMFLOAT3() {}

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]) {}

XMFLOAT3& operator= (const XMFLOAT3& Float3) { x = Float3.x; y = Float3.y; z = Float3.z; return \*this; }

};

struct XMFLOAT4

{

float x;

float y;

float z;

float w;

XMFLOAT4() {}

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]) {}

XMFLOAT4& operator= (const XMFLOAT4& Float4) { x = Float4.x; y = Float4.y; z = Float4.z; w = Float4.w; return \*this; }

};

如果直接利用这些类型进行计算，就无法利用SIMD的优点，所以我们需要进行类型的转换；DirectX Math中提供了Loading函数可以将XMFLOATn类型数据加载到XMVECTOR；Storage函数可以将XMVECTOR类型数据保存到XMFLOATn。

总结如下：

**对于局部或者全局变量，使用XMVECTOR；**

**对于类的成员变量，使用XMFLOATn；**

**使用Loading和Storage函数对数据进行加载和保存；**

**计算的时候使用XMVECTOR类型；**

**参数传递**

为了优化性能为目的，XMVECTOR可以作为函数参数直接传递到SSE/SSE2寄存器中（而不是堆栈内存），参数传递的数量依赖于平台（例如：32/64位 Windows，Windows RT）和编译器。所以根据不同平台/编译器，我们使用**FXMVECTOR**，**GXMVECTOR**，**HXMVECTOR** 和 **CXMVECTOR**类型来传递**XMVECTOR**参数；此外，在函数名前要指明调用注释**XM\_CALLCONV**

**XMVECTOR类型参数传递规则如下：**

前三个参数类型要定义为FXMVECTOR；

第四个要定义为GXMVECTOR；

第五个和第六个要定义为HXMVECTOR；

其他参数要定义为CXMVECTOR。

### 常量向量

常量向量的实例需要使用XMVECTORF32类型

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 };

XMVECTORF32 vRightTop = {

vViewFrust.RightSlope,

vViewFrust.TopSlope,

1.0f,1.0f

};

XMVECTORF32 vLeftBottom = {

vViewFrust.LeftSlope,

vViewFrust.BottomSlope,

1.0f,1.0f

};

其实所有类似的初始化操作都可以使用XMVECTORF32类型，它是一个16位对齐并带有XMVECTOR转换的结构体,定义如下：

// Conversion types for constants

\_\_declspec(align(16)) struct XMVECTORF32

{

union

{

float f[4];

XMVECTOR v;

};

inline operator XMVECTOR() const { return v; }

inline operator const float\*() const { return f; }

#if !defined(\_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

};

你也可以使用**XMVECTORU32**创建XMVECTOR整形常量：

static const XMVECTORU32 vGrabY = { 0x00000000,0xFFFFFFFF,0x00000000,0x00000000 };

### 重载运算符

XMVECTOR有几个重载运算符来计算向量的加减和量乘法：

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);

### 其它

DirectX Math定义了一些有用的常量来近似的表现和 π 相关的值：

const float XM\_PI = 3.141592654f;

const float XM\_2PI = 6.283185307f;

const float XM\_1DIVPI = 0.318309886f;

const float XM\_1DIV2PI = 0.159154943f;

const float XM\_PIDIV2 = 1.570796327f;

const float XM\_PIDIV4 = 0.785398163f;

另外，还定义了下面的内敛函数用以在转换角度和弧度：

inline float XMConvertToRadians(float fDegrees) { return fDegrees \* (XM\_PI / 180.0f); }

inline float XMConvertToDegrees(float fRadians) { return fRadians \* (180.0f / XM\_PI); }

还定义了min/max函数：

template<class T> inline T XMMin(T a, T b) { return (a < b) ? a : b; }

template<class T> inline T XMMax(T a, T b) { return (a > b) ? a : b; }

### Setter 函数

DirectX Math提供了下面函数用来修改XMVECTOR的值：

// Returns the zero vector 0

XMVECTOR XM\_CALLCONV XMVectorZero();

// Returns the vector (1, 1, 1, 1)

XMVECTOR XM\_CALLCONV XMVectorSplatOne();

// Returns the vector (x, y, z, w)

XMVECTOR XM\_CALLCONV XMVectorSet(float x, float y, float z, float w);

// Returns the vector (s, s, s, s)

XMVECTOR XM\_CALLCONV XMVectorReplicate(float Value);

// Returns the vector (vx, vx, vx, vx)

XMVECTOR XM\_CALLCONV XMVectorSplatX(FXMVECTOR V);

// Returns the vector (vy, vy, vy, vy)

XMVECTOR XM\_CALLCONV XMVectorSplatY(FXMVECTOR V);

// Returns the vector (vz, vz, vz, vz)

XMVECTOR XM\_CALLCONV XMVectorSplatZ(FXMVECTOR V);