Chapter 16

D3DX Concrete Types

"The study of mathematics, like the Nile,
begins in minuteness, but
ends in magnificence."
C. C. Colton: *Lacon*, 1820

16.1 Overview

D3DX provides some concrete C++ datatypes and routines for manipulating them. `D3DXCOLOR` represents RGBA colors as 4 `float`s. `D3DXVECTOR2`, `D3DXVECTOR3`, and `D3DXVECTOR4` provide two, three and four dimensional vectors of `float`s. `D3DXMATRIX` provides a homogeneous 4x4 coordinate transformation matrix as 16 `float`s. `D3DXPLANE` represents the equation of a plane in three dimensions as 4 `float`s. `D3DXQUATERNION` provides orienting transformations with a quaternion as 4 `float`s.

Each of these data types has constructors, conversion operators, arithmetic operators and functions for the manipulation of the data type. Some of the datatypes derive directly from Direct3D core data types, when appropriate. This allows you to use the D3DX type directly in methods using the core data types.

We will explore each type with a description of its function, a listing of its C++ class definition and an examination of its class members. Following that, any global functions for manipulating the type will be described.

16.2 Colors

D3DX represents an RGBA color as an instance of the `D3DXCOLOR` class. Its concrete representation is four `float` values, one per color channel, in the public data members `r`, `g`, `b` and `a`. Through type conversion operators, `D3DXCOLOR`

is interchangeable with the D3DCOLOR type alias for DWORD, the D3DCOLORVALUE structure and an array of floats. D3DXCOLOR does not derive from D3DCOLOR-VALUE, so be sure to use its conversion operators to obtain the correct type depending on the context. The structure is listed below and the members are summarized in table 16.1.

```
typedef struct D3DXCOLOR
{
public:
    D3DXCOLOR() {}
    D3DXCOLOR(DWORD argb);
    D3DXCOLOR(const float *);
    D3DXCOLOR(const D3DCOLORVALUE &);
    D3DXCOLOR(float r, float g, float b, float a);

    // casting
    operator DWORD() const;
    operator float *();
    operator const float*() const;
    operator D3DCOLORVALUE *();
    operator const D3DCOLORVALUE *() const;
    operator D3DCOLORVALUE &();
    operator const D3DCOLORVALUE &() const;

    // assignment operators
    D3DXCOLOR &operator +=(const D3DXCOLOR &rhs);
    D3DXCOLOR &operator -=(const D3DXCOLOR &rhs);
    D3DXCOLOR &operator *=(float rhs);
    D3DXCOLOR &operator /=(float rhs);

    // unary operators
    D3DXCOLOR operator +() const;
    D3DXCOLOR operator -() const;

    // binary operators
    D3DXCOLOR operator +(const D3DXCOLOR &value) const;
    D3DXCOLOR operator -(const D3DXCOLOR &value) const;
    D3DXCOLOR operator *(float value) const;
    D3DXCOLOR operator /(float value) const;

    friend D3DXCOLOR operator *(float, const D3DXCOLOR &rhs);

    BOOL operator ==(const D3DXCOLOR &) const;
    BOOL operator !=(const D3DXCOLOR &) const;

    float r, g, b, a;
```

Assignment Operator	Meaning
`operator +=`	$T \leftarrow T + C$
`operator -=`	$T \leftarrow T - C$
`operator *=`	$T \leftarrow fT$
`operator /=`	$T \leftarrow \frac{1}{f}T$

Unary Operator	Meaning
`operator +`	$+T$
`operator -`	$-T$

Binary Operator	Meaning
`operator +`	$T + C$
`operator -`	$T - C$
`operator *`	fT
`operator /`	$\frac{1}{f}T$

Comparison Operator	Meaning
`operator ==`	$T = C$
`operator !=`	$T \neq C$

Table 16.1: `D3DXCOLOR` class members. T represents the color value of `*this`, C represents a color passed to a method, f represents a scalar value passed to a method. Subscripts indicate individual color channels within the color structure.

```
} D3DXCOLOR, *LPD3DXCOLOR;
```

D3DX provides several global functions for operating on `D3DXCOLOR` instances. The functions are summarized in table 16.2. Most of the functions correspond to arithmetic expressions that can be written directly. The functions are provided so that C programs to manipulate `D3DXCOLOR` objects as structures. When writing C++ programs, prefer the native expression over the function call style of manipulating `D3DXCOLOR`s.

```
D3DXCOLOR *D3DXColorAdd(D3DXCOLOR *result,
                const D3DXCOLOR *c1,
                const D3DXCOLOR *c2);
D3DXCOLOR *D3DXColorSubtract(D3DXCOLOR *result,
                const D3DXCOLOR *c1,
                const D3DXCOLOR *c2);
D3DXCOLOR *D3DXColorScale(D3DXCOLOR *result,
                const D3DXCOLOR *c,
                float s);
D3DXCOLOR *D3DXColorModulate(D3DXCOLOR *result,
                const D3DXCOLOR *c1,
                const D3DXCOLOR *c2);
```

Function	Meaning
`D3DXColorAdd`	$C_1 + C_2$
`D3DXColorSubtract`	$C_1 - C_2$
`D3DXColorScale`	sC
`D3DXColorModulate`	$\langle C_{1r}C_{2r},\ C_{1g}C_{2g},\ C_{1b}C_{2b},\ C_{1a}C_{2a}\rangle$
`D3DXColorLerp`	$sC_1 + (1-s)C_2$
`D3DXColorNegative`	$\langle 1,\ 1,\ 1,\ 1\rangle - C$
`D3DXColorAdjustSaturation`	$sC + (1-s)G,\ G = \langle f,\ f,\ f,\ f\rangle,$ $f = 0.2125C_r + 0.7154C_g + 0.0721C_b$
`D3DXColorAdjustContrast`	$sC + (1-s)G,\ G = \langle \frac{1}{2},\ \frac{1}{2},\ \frac{1}{2},\ \frac{1}{2}\rangle$

Table 16.2: `D3DXCOLOR` global functions. C_1 and C_2 represent colors passed to a function, s represents a scalar value passed to a function. The subscripts r, g, b, and a indicate the red, green, blue and alpha color channels, respectively.

```
D3DXCOLOR *D3DXColorLerp(D3DXCOLOR *result,
                const D3DXCOLOR *c1,
                const D3DXCOLOR *c2,
                float s);
```

The functions `D3DXColorNegative`, `D3DXColorAdjustSaturation` and `D3DXColorAdjustContrast` operate only on the red, green, and blue channels of the resulting color. The alpha channel is copied directly from the source color.

```
D3DXCOLOR *D3DXColorNegative(D3DXCOLOR *result,
                const D3DXCOLOR *c);
D3DXCOLOR *D3DXColorAdjustSaturation(D3DXCOLOR *result,
                const D3DXCOLOR *c,
                float s);
D3DXCOLOR *D3DXColorAdjustContrast(D3DXCOLOR *result,
                const D3DXCOLOR *c,
                float c);
```

16.3 Vectors

D3DX provides a vector data type in the `D3DXVECTOR2`, `D3DXVECTOR3` and `D3DXVECTOR4` classes. The three-dimensional vector class derives directly from `D3DVECTOR`, providing automatic conversion to that type. Common vector arithmetic operations are provided directly as class members and are summarized in table 16.3. Some common operations must be performed using functions, as not all vector operators are represented with C++ operator methods.

Each vector type has a constructor from an appropriately sized array of `float`s, or an appropriate number of `float` arguments. Conversion operators are provided to allow the vector to be used in place of an array of floats. Bounds checking on these `float` array conversions are not provided, so use them with caution.

Assignment Operator	Meaning
`operator +=`	$\vec{t} \leftarrow \vec{t} + \vec{v}$
`operator -=`	$\vec{t} \leftarrow \vec{t} - \vec{v}$
`operator *=`	$\vec{t} \leftarrow f\vec{t}$
`operator /=`	$\vec{t} \leftarrow \frac{1}{f}\vec{t}$

Unary Operator	Meaning
`operator +`	$+\vec{t}$
`operator -`	$-\vec{t}$

Binary Operator	Meaning
`operator +`	$\vec{t} + \vec{v}$
`operator -`	$\vec{t} - \vec{v}$
`operator *`	$f\vec{t}$
`operator /`	$\frac{1}{f}\vec{t}$

Comparison Operator	Meaning
`operator ==`	$\vec{t} = \vec{v}$
`operator !=`	$\vec{t} \neq \vec{v}$

Table 16.3: D3DX vector class members. $\vec{t}$ represents the vector stored in `*this`, $\vec{v}$ represents a vector passed into the method and f represents a scalar value passed into the method.

```
typedef struct D3DXVECTOR2
{
public:
    D3DXVECTOR2() {};
    D3DXVECTOR2(const float  *);
    D3DXVECTOR2(float x, float y);

    // casting
    operator float *();
    operator const float *() const;

    // assignment operators
    D3DXVECTOR2 &operator += (const D3DXVECTOR2 &);
    D3DXVECTOR2 &operator -= (const D3DXVECTOR2 &);
    D3DXVECTOR2 &operator *= (float);
    D3DXVECTOR2 &operator /= (float);

    // unary operators
    D3DXVECTOR2 operator + () const;
    D3DXVECTOR2 operator - () const;

    // binary operators
    D3DXVECTOR2 operator + (const D3DXVECTOR2 &) const;
    D3DXVECTOR2 operator - (const D3DXVECTOR2 &) const;
    D3DXVECTOR2 operator * (float) const;
    D3DXVECTOR2 operator / (float) const;

    friend D3DXVECTOR2 operator * (float, const D3DXVECTOR2 &);

    BOOL operator == (const D3DXVECTOR2 &) const;
    BOOL operator != (const D3DXVECTOR2 &) const;

public:
    float x, y;
} D3DXVECTOR2, *LPD3DXVECTOR2;

typedef struct D3DXVECTOR3 : public D3DVECTOR
{
public:
    D3DXVECTOR3() {};
    D3DXVECTOR3(const float *);
    D3DXVECTOR3(const D3DVECTOR &);
    D3DXVECTOR3(float x, float y, float z);

    // casting
    operator float *();
```

```
    operator const float *() const;

    // assignment operators
    D3DXVECTOR3 &operator += (const D3DXVECTOR3 &);
    D3DXVECTOR3 &operator -= (const D3DXVECTOR3 &);
    D3DXVECTOR3 &operator *= (float);
    D3DXVECTOR3 &operator /= (float);

    // unary operators
    D3DXVECTOR3 operator + () const;
    D3DXVECTOR3 operator - () const;

    // binary operators
    D3DXVECTOR3 operator + (const D3DXVECTOR3 &) const;
    D3DXVECTOR3 operator - (const D3DXVECTOR3 &) const;
    D3DXVECTOR3 operator * (float) const;
    D3DXVECTOR3 operator / (float) const;

    friend D3DXVECTOR3 operator * (float, const struct D3DXVECTOR3 &);

    BOOL operator == (const D3DXVECTOR3 &) const;
    BOOL operator != (const D3DXVECTOR3 &) const;

} D3DXVECTOR3, *LPD3DXVECTOR3;

typedef struct D3DXVECTOR4
{
public:
    D3DXVECTOR4() {};
    D3DXVECTOR4(const float*);
    D3DXVECTOR4(float x, float y, float z, float w);

    // casting
    operator float *();
    operator const float *() const;

    // assignment operators
    D3DXVECTOR4 &operator += (const D3DXVECTOR4 &);
    D3DXVECTOR4 &operator -= (const D3DXVECTOR4 &);
    D3DXVECTOR4 &operator *= (float);
    D3DXVECTOR4 &operator /= (float);

    // unary operators
    D3DXVECTOR4 operator + () const;
    D3DXVECTOR4 operator - () const;
```

	Vector Type		
Operation	2D	3D	4D
Addition	Yes	Yes	Yes
Subtraction	Yes	Yes	Yes
Scalar Product	Yes	Yes	Yes
Dot Product	Yes	Yes	Yes
Cross Product	Yes	Yes	Yes
Normalize	Yes	Yes	Yes
Length	Yes	Yes	Yes
Length Squared	Yes	Yes	Yes
Minimize	Yes	Yes	Yes
Maximize	Yes	Yes	Yes
Linear Interpolation	Yes	Yes	Yes
Hermite Interpolation	Yes	Yes	Yes
Catmull-Rom Interpolation	Yes	Yes	Yes
Barycentric Interpolation	Yes	Yes	Yes
Transform	Yes	Yes	Yes
Transform Coordinate	Yes	Yes	Yes
Transform Normal	Yes	Yes	Yes
Project		Yes	
Unproject		Yes	

Table 16.4: D3DX vector class function summary.

```
    // binary operators
    D3DXVECTOR4 operator + (const D3DXVECTOR4 &) const;
    D3DXVECTOR4 operator - (const D3DXVECTOR4 &) const;
    D3DXVECTOR4 operator * (float) const;
    D3DXVECTOR4 operator / (float) const;

    friend D3DXVECTOR4 operator * (float, const D3DXVECTOR4 &);

    BOOL operator == (const D3DXVECTOR4 &) const;
    BOOL operator != (const D3DXVECTOR4 &) const;

public:
    float x, y, z, w;
} D3DXVECTOR4, *LPD3DXVECTOR4;
```

Global functions provide additional operations on the vector type. The operations available for each vector type are summarized in table 16.4. The meaning of the operations is summarized in table 16.5. The basic operations of addition, subtraction, scalar product, dot product, length, length squared, minimize and maximize are straightforward.

The cross product operation is slightly different depending on the vector class. For 2D vectors, the function `D3DXVec2CCW` returns the magnitude of the

Operation	Meaning
Addition	$\vec{v_1} + \vec{v_2}$
Subtraction	$\vec{v_1} - \vec{v_2}$
Scalar Product	$f\vec{v}$
Dot Product	$\vec{v_1} \cdot \vec{v_2}$
Cross Product	$\vec{v_1} \otimes \vec{v_2}$
Normalize	$\frac{1}{\|\vec{v}\|}\vec{v}$
Length	$\|\vec{v}\|$
Length Squared	$\|\vec{v}\|^2$
Minimize	$\langle \min(v_{1x}, v_{2x}), \min(v_{1y}, v_{2y}), \min(v_{1z}, v_{2z})\rangle$
Maximize	$\langle \max(v_{1x}, v_{2x}), \max(v_{1y}, v_{2y}), \max(v_{1z}, v_{2z})\rangle$

Table 16.5: Summary of arithmetic operations on vectors provided by D3DX. The semantics are shown for `D3DXVECTOR3`, and are similar for other vector classes.

z component resulting from the cross product of two vectors in the xy plane. For 3D vectors, the function `D3DXVec3Cross` returns the cross product of two vectors. For 4D vectors, the function `D3DXVec4Cross` returns the cross produce of three vectors.

```
D3DXVECTOR2 *D3DXVec2Add(D3DXVECTOR2 *result,
                const D3DXVECTOR2 *v1,
                const D3DXVECTOR2 *v2);
D3DXVECTOR2 *D3DXVec2Subtract(D3DXVECTOR2 *result,
                const D3DXVECTOR2 *v1,
                const D3DXVECTOR2 *v2);
D3DXVECTOR2 *D3DXVec2Scale(D3DXVECTOR2 *result,
                const D3DXVECTOR2 *v,
                float f);
float D3DXVec2Dot(const D3DXVECTOR2 *v1,
                const D3DXVECTOR2 *v2);
float D3DXVec2CCW(const D3DXVECTOR2 *v1,
                const D3DXVECTOR2 *v2);
D3DXVECTOR2 *D3DXVec2Normalize(D3DXVECTOR2 *result,
                const D3DXVECTOR2 *v);
float D3DXVec2Length(const D3DXVECTOR2 *v);
float D3DXVec2LengthSq(const D3DXVECTOR2 *v);
D3DXVECTOR2 *D3DXVec2Minimize(D3DXVECTOR2 *result,
                const D3DXVECTOR2 *v1,
                const D3DXVECTOR2 *v2);
D3DXVECTOR2 *D3DXVec2Maximize(D3DXVECTOR2 *result,
                const D3DXVECTOR2 *v1,
                const D3DXVECTOR2 *v2);

D3DXVECTOR3 *D3DXVec3Add(D3DXVECTOR3 *result,
```

```
                const D3DXVECTOR3 *v1,
                const D3DXVECTOR3 *v2);
D3DXVECTOR3 *D3DXVec3Subtract(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v1,
                const D3DXVECTOR3 *v2);
D3DXVECTOR3 *D3DXVec3Scale(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v,
                float f);
float D3DXVec3Dot(const D3DXVECTOR3 *v1,
                const D3DXVECTOR3 *v2);
D3DXVECTOR3 *D3DXVec3Cross(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v1,
                const D3DXVECTOR3 *v2);
D3DXVECTOR3 *D3DXVec3Normalize(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v);
float D3DXVec3Length(const D3DXVECTOR3 *v);
float D3DXVec3LengthSq(const D3DXVECTOR3 *v);
D3DXVECTOR3 *D3DXVec3Minimize(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v1,
                const D3DXVECTOR3 *v2);
D3DXVECTOR3 *D3DXVec3Maximize(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v1,
                const D3DXVECTOR3 *v2);

D3DXVECTOR4 *D3DXVec4Add(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v1,
                const D3DXVECTOR4 *v2);
D3DXVECTOR4 *D3DXVec4Subtract(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v1,
                const D3DXVECTOR4 *v2);
D3DXVECTOR4 *D3DXVec4Normalize(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v);
D3DXVECTOR4 *D3DXVec4Scale(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v,
                float f);
float D3DXVec4Dot(const D3DXVECTOR4 *v1,
                const D3DXVECTOR4 *v2);
D3DXVECTOR4 *D3DXVec4Cross(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v1,
                const D3DXVECTOR4 *v2,
                const D3DXVECTOR4 *v3);
float D3DXVec4Length(const D3DXVECTOR4 *v);
float D3DXVec4LengthSq(const D3DXVECTOR4 *v);
D3DXVECTOR4 *D3DXVec4Minimize(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v1,
                const D3DXVECTOR4 *v2);
```

```
D3DXVECTOR4 *D3DXVec4Maximize(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v1,
                const D3DXVECTOR4 *v2);
```

Four forms of interpolation are provided for the vector classes. Linear interpolation is provided by the `Lerp` functions. Linear interpolation between two vectors computes the resulting vector $\vec{v}$ from the two input vectors $\vec{v_1}$, $\vec{v_2}$ and an interpolating factor f. When the interpolating factor f is zero, the result is $\vec{v_1}$ and when f is one, the result is $\vec{v_2}$. The result is given by the formula:

$$\vec{v} = \vec{v_1} + f(\vec{v_2} - \vec{v_1})$$

```
D3DXVECTOR2 *D3DXVec2Lerp(D3DXVECTOR2 *result,
                const D3DXVECTOR2 *v1,
                const D3DXVECTOR2 *v2,
                float f);
D3DXVECTOR3 *D3DXVec3Lerp(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v1,
                const D3DXVECTOR3 *v2,
                float f);
D3DXVECTOR4 *D3DXVec4Lerp(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v1,
                const D3DXVECTOR4 *v2,
                float f);
```

The `Hermite` functions provide Hermite spline interpolation between two vector, tangent pairs $\vec{v_1}, \vec{t_1}$ and $\vec{v_2}, \vec{t_2}$ and an interpolating factor f. When the interpolating factor f is zero, the result is $\vec{v_1}$ and when f is one, the result is $\vec{v_2}$. The result is given by the formula:

$$\vec{v} = a_0\vec{v_1} + a_1\vec{t_1} + a_2\vec{v_2} + a_3\vec{t_2}$$

where the coefficients a_0, a_1, a_2, and a_3 are given as:

$$\begin{aligned} a_0 &= 2f^3 - 3f^2 + 1 \\ a_1 &= f^3 - 2f^2 + f \\ a_2 &= -2f^3 + 3f^2 \\ a_3 &= f^3 - f^2 \end{aligned}$$

```
D3DXVECTOR2 *D3DXVec2Hermite(D3DXVECTOR2 *result,
                const D3DXVECTOR2 *v1,
                const D3DXVECTOR2 *t1,
                const D3DXVECTOR2 *v2,
                const D3DXVECTOR2 *t2,
                float f);
D3DXVECTOR3 *D3DXVec3Hermite(D3DXVECTOR3 *result,
```

```
                const D3DXVECTOR3 *v1,
                const D3DXVECTOR3 *t1,
                const D3DXVECTOR3 *v2,
                const D3DXVECTOR3 *t2,
                float f);
D3DXVECTOR4 *D3DXVec4Hermite(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v1,
                const D3DXVECTOR4 *t1,
                const D3DXVECTOR4 *v2,
                const D3DXVECTOR4 *t2,
                float f);
```

The `CatmullRom` functions provide Catmull-Rom spline interpolation between the two vectors $\vec{v_1}$ and $\vec{v_2}$ by an interpolating factor f. The additional vectors $\vec{v_0}$ and $\vec{v_3}$ are used as control points on either side of $\vec{v_1}$ and $\vec{v_2}$ to control the shape of the interpolation between $\vec{v_1}$ and $\vec{v_2}$. When the interpolating factor f is zero, the result is $\vec{v_1}$ and when the interpolating factor f is one, the result is $\vec{v_2}$. The result is given by the formula:

$$\vec{v} = \frac{1}{2}\left(a_0\vec{v_0} + a_1\vec{v_1} + a_2\vec{v_2} + a_3\vec{v_3}\right)$$

where the coefficients a_0, a_1, a_2 and a_3 are given as:

$$\begin{aligned} a_0 &= -f^3 + 2f^2 - f \\ a_1 &= 3f^3 - 5f^2 + 2 \\ a_2 &= -3f^3 + 4f^2 + f \\ a_3 &= f^3 - f^2 \end{aligned}$$

```
D3DXVECTOR2 *D3DXVec2CatmullRom(D3DXVECTOR2 *result,
                const D3DXVECTOR2 *v0,
                const D3DXVECTOR2 *v1,
                const D3DXVECTOR2 *v2,
                const D3DXVECTOR2 *v3,
                float f);
D3DXVECTOR3 *D3DXVec3CatmullRom(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v0,
                const D3DXVECTOR3 *v1,
                const D3DXVECTOR3 *v2,
                const D3DXVECTOR3 *v3,
                float f);
D3DXVECTOR4 *D3DXVec4CatmullRom(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v0,
                const D3DXVECTOR4 *v1,
                const D3DXVECTOR4 *v2,
```

```
                const D3DXVECTOR4 *v3,
                float f);
```

The `Barycentric` functions provide interpolation between three vectors $\vec{v_1}$, $\vec{v_2}$, and $\vec{v_3}$ using two barycentric coordinates f and g. When the barycentric coordinates f, g are both zero, the result is $\vec{v_1}$. When the f coordinate is one and g is zero, the result is $\vec{v_2}$. When the g coordinate is one and f is zero, the result is $\vec{v_3}$. The result is given by the formula:

$$\vec{v} = \vec{v_1} + f(\vec{v_2} - \vec{v_1}) + g(\vec{v_3} - \vec{v_1})$$

```
D3DXVECTOR2 *D3DXVec2BaryCentric(D3DXVECTOR2 *result,
                const D3DXVECTOR2 *v1,
                const D3DXVECTOR2 *v2,
                const D3DXVECTOR2 *v3,
                float f,
                float g);
D3DXVECTOR3 *D3DXVec3BaryCentric(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v1,
                const D3DXVECTOR3 *v2,
                const D3DXVECTOR3 *v3,
                float f,
                float g);
D3DXVECTOR4 *D3DXVec4BaryCentric(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v1,
                const D3DXVECTOR4 *v2,
                const D3DXVECTOR4 *v3,
                float f,
                float g);
```

The transformation functions allow a vector to be treated as a position and transformed by a matrix. The `Transform` functions perform a vector, matrix product. Two-dimensional vectors are expanded to $\langle x, y, 0, 1\rangle$ and three-dimensional vectors are expanded to $\langle x, y, z, 1\rangle$ before being multiplied by the 4×4 matrix. In all cases, the resulting vector is a four-dimensional homogeneous point. The `TransformCoord` functions perform the same vector, matrix product for two- and three-dimensional vectors, but project the resulting homogeneous point back into cartesian space. The `TransformNormal` functions operate similarly to the `TransformCoord` functions, but treat the incoming vector as a normal vector. The result is that scaling and rotation transformations are applied, but not translations. If the matrix passed to the `TransformNormal` functions represents a non-affine transformation, then the transpose of the inverse of the matrix should be passed to the function.

```
D3DXVECTOR4 *D3DXVec2Transform(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v,
                const D3DXMATRIX *m);
```

```
D3DXVECTOR2 *D3DXVec2TransformCoord(D3DXVECTOR2 *result,
                const D3DXVECTOR2 *v,
                const D3DXMATRIX *m);
D3DXVECTOR2 *D3DXVec2TransformNormal(D3DXVECTOR2 *result,
                const D3DXVECTOR2 *v,
                const D3DXMATRIX *m);

D3DXVECTOR4 *D3DXVec3Transform(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v,
                const D3DXMATRIX *m);
D3DXVECTOR3 *D3DXVec3TransformCoord(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v,
                const D3DXMATRIX *m);
D3DXVECTOR3 *D3DXVec3TransformNormal(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v,
                const D3DXMATRIX *m);

D3DXVECTOR4 *D3DXVec4Transform(D3DXVECTOR4 *result,
                const D3DXVECTOR4 *v,
                const D3DXMATRIX *m);
```

For three-dimensional vectors, D3DX provides functions to project them through the transformation pipeline given the viewport and the world, view, and projection matrices. The input vector is a position in model coordinates and the output vector is a position in three-dimensional screen space. The unproject function performs the reverse transformation.

```
D3DXVECTOR3 *D3DXVec3Project(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v,
                const D3DVIEWPORT9 *viewport,
                const D3DXMATRIX *projection,
                const D3DXMATRIX *view,
                const D3DXMATRIX *world);
D3DXVECTOR3 *D3DXVec3Unproject(D3DXVECTOR3 *result,
                const D3DXVECTOR3 *v,
                const D3DVIEWPORT9 *viewport,
                const D3DXMATRIX *projection,
                const D3DXMATRIX *view,
                const D3DXMATRIX *world);
```

16.4 Matrices

The `D3DXMATRIX` class provides a 4×4 homogeneous matrix that derives directly from `D3DMATRIX`. The class provides constructors that can build a matrix from an array of floats, a `D3DMATRIX`, or an explicit list of initializers. A function call operator is provided that allows array elements to be accessed with subscripts.

Assignment Operator	Meaning
`operator +=`	$\mathbf{T} \leftarrow \mathbf{T} + \mathbf{M}$
`operator -=`	$\mathbf{T} \leftarrow \mathbf{T} - \mathbf{M}$
`operator *=`	$\mathbf{T} \leftarrow f\mathbf{T}$
`operator *=`	$\mathbf{T} \leftarrow \mathbf{TM}$
`operator /=`	$\mathbf{T} \leftarrow \frac{1}{f}\mathbf{T}$

Unary Operator	Meaning
`operator +`	$+\mathbf{T}$
`operator -`	$-\mathbf{T}$

Binary Operator	Meaning
`operator +`	$\mathbf{T} + \mathbf{M}$
`operator -`	$\mathbf{T} - \mathbf{M}$
`operator *`	$f\mathbf{T}$
`operator *`	$\mathbf{TM}$
`operator /`	$\frac{1}{f}\mathbf{T}$

Comparison Operator	Meaning
`operator ==`	$\mathbf{T} = \mathbf{M}$
`operator !=`	$\mathbf{T} \neq \mathbf{M}$

Table 16.6: `D3DXMATRIX` class members. $\mathbf{T}$ represents the matrix value of `*this`, $\mathbf{M}$ represents a matrix passed to a method, f represents a scalar value passed to a method.

For example, if `m` is declared as a `D3DXMATRIX`, then `m(0,1)` is identical to `m._12`. Casting operators are provided that expose the matrix as an array of `float`s. As with vectors and colors, no bounds checking is performed on these arrays of `float`s, so they should be used with care. The usual matrix arithmetic operators are provided and their meaning is summarized in table 16.6. The comparison operators provided perform a bitwise comparison. As mentioned in chapter 1, exact comparisons of floating-point values can lead to problems and it is better to compare floating-point values within an ϵ tolerance. Matrices are often computed by composition and the resulting matrix multiplication will introduce small floating-point errors, making bitwise comparison less than useful most of the time.

```
typedef struct D3DXMATRIX : public D3DMATRIX
{
public:
    D3DXMATRIX() {};
    D3DXMATRIX(const float *);
    D3DXMATRIX(const D3DMATRIX &);
    D3DXMATRIX(float _11, float _12, float _13, float _14,
               float _21, float _22, float _23, float _24,
```

```
                float _31, float _32, float _33, float _34,
                float _41, float _42, float _43, float _44);

    // access grants
    float &operator () (UINT Row, UINT Col);
    float  operator () (UINT Row, UINT Col) const;

    // casting operators
    operator float *();
    operator const float *() const;

    // assignment operators
    D3DXMATRIX &operator *= (const D3DXMATRIX &);
    D3DXMATRIX &operator += (const D3DXMATRIX &);
    D3DXMATRIX &operator -= (const D3DXMATRIX &);
    D3DXMATRIX &operator *= (float);
    D3DXMATRIX &operator /= (float);

    // unary operators
    D3DXMATRIX operator + () const;
    D3DXMATRIX operator - () const;

    // binary operators
    D3DXMATRIX operator * (const D3DXMATRIX &) const;
    D3DXMATRIX operator + (const D3DXMATRIX &) const;
    D3DXMATRIX operator - (const D3DXMATRIX &) const;
    D3DXMATRIX operator * (float) const;
    D3DXMATRIX operator / (float) const;

    friend D3DXMATRIX operator * (float, const D3DXMATRIX &);

    BOOL operator == (const D3DXMATRIX &) const;
    BOOL operator != (const D3DXMATRIX &) const;

} D3DXMATRIX, *LPD3DXMATRIX;
```

Matrix operations can be accelerated with the Intel SSE or AMD 3DNow! instruction set extensions, but only when the matrices are aligned on 16 byte boundaries. D3DX provides a `D3DXMATRIXA16` class that derives from `D3DXMATRIX` and forces alignment to a 16 byte boundary. This class works with Visual C++ 7 and Visual C++ 6 with the processor pack installed. Unfortunately, there is no way to detect the installation of the processor pack from inside an include file, so D3DX enables the 16 byte alignment only in Visual C++ 7. For other compilers, the `D3DXMATRIXA16` class is identical to the `D3DXMATRIX` class.

16.4.1 Arithmetic Functions

Most of the global matrix functions construct particular types of matrices from the supplied parameters. However, some functions perform mathematical operations on matrices that are not supported by the matrix member functions. The `D3DXMatrixIsIdentity` function returns `TRUE` if a matrix is exactly equal to the identity matrix. This function does not use ϵ comparisons, so care should be taken to use it only on suitable matrices. The transpose of a matrix is computed by `D3DXMatrixTranspose`.

```
BOOL D3DXMatrixIsIdentity(const D3DXMATRIX *m);
D3DXMATRIX *D3DXMatrixTranspose(D3DXMATRIX *result,
                const D3DXMATRIX *m);
```

The determinant of a matrix $\mathbf{M}$, given by the formula

$$\begin{aligned} |\mathbf{M}| = {} & (m_{11}m_{22} - m_{12}m_{21})(m_{33}m_{44} - m_{34}m_{43}) + \\ & (m_{13}m_{21} - m_{11}m_{23})(m_{32}m_{44} - m_{34}m_{42}) + \\ & (m_{11}m_{24} - m_{14}m_{21})(m_{32}m_{43} - m_{33}m_{42}) + \\ & (m_{12}m_{23} - m_{13}m_{22})(m_{31}m_{44} - m_{34}m_{41}) + \\ & (m_{14}m_{22} - m_{12}m_{24})(m_{31}m_{43} - m_{33}m_{41}) + \\ & (m_{13}m_{24} - m_{14}m_{23})(m_{31}m_{42} - m_{32}m_{41}) \end{aligned}$$

is computed by `D3DXMatrixDeterminant`. The determinant is also computed while finding the inverse of a matrix and both are computed by the `D3DXMatrixInverse` function. If the matrix is not invertible, the function returns `NULL`. If the determinant of the matrix is not needed, the `determinant` parameter can be `NULL`.

```
float D3DXMatrixfDeterminant(const D3DXMATRIX *m);
D3DXMATRIX *D3DXMatrixInverse(D3DXMATRIX *result,
                float *determinant,
                const D3DXMATRIX *m);
```

Matrix products are computed with the `D3DXMatrixMultiply` and `D3DXMatrixMultiplyTranspose` functions. The latter performs a transpose on the matrix product before storing it in the result parameter. This is useful for computing matrix products for use with shader constant registers.

```
D3DXMATRIX *D3DXMatrixMultiply(D3DXMATRIX *result,
                const D3DXMATRIX *m1,
                const D3DXMATRIX *m2);
D3DXMATRIX *D3DXMatrixMultiplyTranspose(D3DXMATRIX *result,
                const D3DXMATRIX *m1,
                const D3DXMATRIX *m2);
```

16.4.2 Identity, Translation, and Scaling Matrices

The remainder of the matrix functions construct matrices of various types. An identity matrix is constructed with `D3DXMatrixIdentity`. A translation matrix is constructed with `D3DXMatrixTranslation` and a scaling matrix is constructed with `D3DXMatrixScaling`.

```
D3DXMATRIX *D3DXMatrixIdentity(D3DXMATRIX *result);
D3DXMATRIX *D3DXMatrixTranslation(D3DXMATRIX *result,
                float x,
                float y,
                float z);
D3DXMATRIX *D3DXMatrixScaling(D3DXMATRIX *result,
                float sx,
                float sy,
                float sz);
```

16.4.3 Rotation Matrices

There are several functions for computing rotation matrices. Rotations about one of the principal axes can be computed with the functions `D3DXMatrixRotationX`, `D3DXMatrixRotationY`, and `D3DXMatrixRotationZ`. A rotation about an arbitrary axis is computed with `D3DXMatrixRotationAxis`. A combined rotation about all three principal axes is computed with the `D3DXMatrixRotationYawPitchRoll` function. The resulting matrix is equivalent to the composite transformation $R_x(\text{yaw})R_y(\text{pitch})R_z(\text{roll})$. An orientation represented by a quaternion can be used to build a rotation matrix with `D3DXMatrixRotationQuaternion`. The resulting matrix is equivalent to a matrix transforming the x axis into the orientation specified by the quaternion.

```
D3DXMATRIX *D3DXMatrixRotationX(D3DXMATRIX *result,
                float angle);
D3DXMATRIX *D3DXMatrixRotationY(D3DXMATRIX *result,
                float angle);
D3DXMATRIX *D3DXMatrixRotationZ(D3DXMATRIX *result,
                float angle);
D3DXMATRIX *D3DXMatrixRotationAxis(D3DXMATRIX *result,
                const D3DXVECTOR3 *axis,
                float angle);
D3DXMATRIX *D3DXMatrixRotationYawPitchRoll(D3DXMATRIX *result,
                float yaw,
                float pitch,
                float roll);
D3DXMATRIX *D3DXMatrixRotationQuaternion(D3DXMATRIX *result,
                const D3DXQUATERNION *q);
```

16.4.4 Composite Transformation Matrices

Complex composite transformations involving translation, rotation and scaling can be built from the `D3DXMatrixAffineTransformation` and `D3DXMatrix-Transformation` functions. The former computes the composite matrix

$$T(-C)R(\vec{q})S(s)T(C)T(O)$$

where C is the center of rotation, $\vec{q}$ is the quaternion specifying the new orientation, s is the scale factor, and O is the translation offset. The latter computes the composite matrix

$$T(-C_s)R(\vec{q_s})S(s_x, s_y, s_z)T(C_s)T(-C_r)R(\vec{q_r})T(C_r)T(O)$$

where C_s is the center of scaling, $\vec{q_s}$ is the scaling orientation quaternion, s_x, s_y, and s_z are the scaling factors, C_r is the center of rotation, $\vec{q_r}$ is the rotation orientation quaternion, and O is the translation offset.

```
D3DXMATRIX *D3DXMatrixAffineTransformation(D3DXMATRIX *result,
                float scale,
                const D3DXVECTOR3 *center_of_rotation,
                const D3DXQUATERNION *rotation,
                const D3DXVECTOR3 *translation);
D3DXMATRIX *D3DXMatrixTransformation(D3DXMATRIX *result,
                const D3DXVECTOR3 *center_of_scaling,
                const D3DXQUATERNION *scaling_rotation,
                const D3DXVECTOR3 *scaling,
                const D3DXVECTOR3 *center_of_rotation,
                const D3DXQUATERNION *rotation,
                const D3DXVECTOR3 *translation);
```

16.4.5 Viewing Matrices

The functions `D3DXMatrixLookAtLH` and `D3DXMatrixLookAtRH` compute suitable view transformation matrices. The `eye` parameter gives the position of the camera in world space which becomes the origin of camera space. The `at` parameter gives the world space point where the camera is looking to define the gaze direction which will become the $+z$ axis in camera space. The `up` parameter gives the orientation of the camera around the gaze direction and will define the $+y$ axis in camera space. The `LH` version computes a proper view matrix for a left-handed coordinate system and the `RH` version computes a proper view matrix for a right-handed coordinate system.

```
D3DXMATRIX *D3DXMatrixLookAtLH(D3DXMATRIX *result,
                const D3DXVECTOR3 *eye,
                const D3DXVECTOR3 *at,
                const D3DXVECTOR3 *up);
D3DXMATRIX *D3DXMatrixLookAtRH(D3DXMATRIX *result,
```

```
                const D3DXVECTOR3 *eye,
                const D3DXVECTOR3 *at,
                const D3DXVECTOR3 *up);
```

16.4.6 Orthographic Projection Matrices

An orthographic projection matrix with a view frustum centered on the origin is computed by the `D3DXMatrixOrthoLH` and `D3DXMatrixOrthoRH` functions. The `width` and `height` parameters give the proportions of the view volume on the x and y axes, respectively. The `z_near` and `z_far` parameters give the location of the near and far planes of the view frustum. The `OffCenter` variations compute an orthographic projection matrix with a view frustum whose x, y bounds are given by the `left`, `right`, `bottom` and `top` parameters. The off center functions are often used to create tiled renderings of a scene. The `LH` versions compute proper orthographic projection matrices for a left-handed coordinate system and the `RH` versions compute proper orthographic projection matrices for a right-handed coordinate system.

```
D3DXMATRIX *D3DXMatrixOrthoLH(D3DXMATRIX *result,
                float width,
                float height,
                float z_near,
                float z_far);
D3DXMATRIX *D3DXMatrixOrthoRH(D3DXMATRIX *result,
                float width,
                float height,
                float z_near,
                float z_far);
D3DXMATRIX *D3DXMatrixOrthoOffCenterLH(D3DXMATRIX *result,
                float left,
                float right,
                float bottom,
                float top,
                float z_near,
                float z_far);
D3DXMATRIX *D3DXMatrixOrthoOffCenterRH(D3DXMATRIX *result,
                float left,
                float right,
                float bottom,
                float top,
                float z_near,
                float z_far);
```

16.4.7 Perspective Projection Matrices

A perspective projection matrix with a view frustum centered on the origin is computed by either the `D3DXMatrixPerspectiveLH` or `D3DXMatrixPerspectiveRH` functions. The extent of the view volume in the near plane is specified by the `width` and `height` parameters. An alternative method of specifying the extent of the view volume in the near plane is to specify a field of view angle in the y direction and an aspect ratio. The functions `D3DXMatrixPerspectiveFovLH` and `D3DXMatrixPerspectiveFovRH` compute perspective projection matrices centered on the origin using the field of view and aspect ratio of the view volume extent on the near plane. The off center versions compute a perspective projection matrix whose view frustum x, y extents in the near plane are given by the `left`, `right`, `bottom` and `top` parameters. The `LH` versions compute proper perspective projection matrices for a left-handed coordinate system and the `RH` versions compute proper perspective projection matrices for a right-handed coordinate system.

```
D3DXMATRIX *D3DXMatrixPerspectiveLH(D3DXMATRIX *result,
                float width,
                float height,
                float z_near,
                float z_far);
D3DXMATRIX *D3DXMatrixPerspectiveRH(D3DXMATRIX *result,
                float width,
                float height,
                float z_near,
                float z_far);
D3DXMATRIX *D3DXMatrixPerspectiveFovLH(D3DXMATRIX *result,
                float fov,
                float aspect_ratio,
                float z_near,
                float z_far);
D3DXMATRIX *D3DXMatrixPerspectiveFovRH(D3DXMATRIX *result,
                float fov,
                float aspect_ratio,
                float z_near,
                float z_far);
D3DXMATRIX *D3DXMatrixPerspectiveOffCenterLH(D3DXMATRIX *result,
                float left,
                float right,
                float bottom,
                float top,
                float z_near,
                float z_far);
D3DXMATRIX *D3DXMatrixPerspectiveOffCenterRH(D3DXMATRIX *result,
                float left,
                float right,
```

```
                float bottom,
                float top,
                float z_near,
                float z_far);
```

16.4.8 Shadow and Reflection Matrices

A matrix that projects geometry along the direction of a light and into a plane is computed by `D3DXMatrixShadow`. The light direction is given by the `light` parameter and the equation of the plane is given by the `plane` parameter. The `D3DXMatrixReflect` function constructs a matrix which reflects geometry about a plane.

```
D3DXMATRIX *D3DXMatrixShadow(D3DXMATRIX *result,
                const D3DXVECTOR4 *light,
                const D3DXPLANE *plane);
D3DXMATRIX *D3DXMatrixReflect(D3DXMATRIX *result,
                const D3DXPLANE *plane);
```

16.4.9 Matrix Helper Classes

While the matrix construction functions are convenient, they complicate the business of building composite transformation matrices because they require that the matrices be constructed by calling functions. It would be more convenient from C++ to be able to construct particular matrices by invoking a constructor so that the resulting matrix object is ready to use in an expression. We can provide matrix helper classes that derive from `D3DXMATRIX` and call the appropriate function in their constructor to build the kind of matrix we need. For example, using only the D3DX functions, our code might look like this:

```
D3DXMATRIX rot_x;
::D3DXMatrixRotationX(&rot_x, m_fWorldRotX);
D3DXMATRIX rot_y;
::D3DXMatrixRotationY(&rot_y, m_fWorldRotY);
D3DXMATRIX world;
::D3DXMatrixMultiply(&world, &rot_x, &rot_y);
THR(device->SetTransform(D3DTS_WORLD, &world));
```

Here we had to create three temporary matrix variables whose purpose was only to live long enough to be passed as an argument to a D3DX function that computed a matrix. If we had matrix construction helper classes for rotations about the x and y axes, we could write this code in a more natural way as follows:

```
THR(device->SetTransform(D3DTS_WORLD,
    &(rt::mat_rot_x(m_fWorldRotX)*rt::mat_rot_y(m_fWorldRotY))));
```

Here we avoided calling `D3DXMatrixMultiply` in favor of the much more natural `operator*` method on `D3DXMATRIX`. The parenthesis around the matrix product are necessary so that we can take the address of the resulting temporary matrix constructed by the compiler. We have the same number of temporary matrices as the function call style code, but the result is more concise and treats matrix objects just like other numeric quantities.

A collection of such matrix construction helper classes is found in the include file `<rt/mat.h>` in the sample code.

16.5 Planes

D3DX provides the `D3DXPLANE` structure to represent three-dimensional planes. A plane can be represented by the equation

$$ax + by + cz + d = 0$$

called the general form of the plane equation. A plane defined by a point on the plane, $P = (x_0, y_0, z_0)$, and a normal to the plane, $\vec{n} = \langle a, b, c \rangle$, is represented by the equation

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0$$

called the point-normal form of the plane equation. The two can be related to one another with $d = -(ax_0 + by_0 + cz_0)$.

The `D3DXPLANE` structure stores the coefficients a, b, c, and d in publicly accessible data members of the same name. Constructors are provided for creating planes from an array of `float`s containing the plane coefficients in the order a, b, c, and d, or from an explicit initializer list giving the coefficients directly. Casting operators are provided that allow a `D3DXPLANE` class to be treated as an array of `float`s. No bounds checking is provided with the casting operators, so care must be taken not to overflow the extent of the array's 4 elements. The unary negation operator returns a plane with the sign of all the coefficients changed. The equality operators compare planes for bitwise equality and do not account for floating-point errors. As discussed in chapter 1, an ϵ based comparison function should be used when comparing structures composed of floating-point values.

```
typedef struct D3DXPLANE
{
public:
    D3DXPLANE() {}
    D3DXPLANE(const float*);
    D3DXPLANE(float a, float b, float c, float d);

    // casting
    operator float *();
    operator const float *() const;
```

```
    // unary operators
    D3DXPLANE operator + () const;
    D3DXPLANE operator - () const;

    // binary operators
    BOOL operator == (const D3DXPLANE &) const;
    BOOL operator != (const D3DXPLANE &) const;

    float a, b, c, d;
} D3DXPLANE, *LPD3DXPLANE;
```

The function `D3DXPlaneFromPointNormal` constructs a `D3DXPLANE` from a point on the plane and a normal to the plane. Similarly, the function `D3DXPlaneFromPoints` constructs a `D3DXPLANE` from three non-colinear points. `D3DXPlaneNormalize` normalizes the plane coefficients so that the vector $\langle a, b, c \rangle$ has unit magnitude.

```
D3DXPLANE *D3DXPlaneFromPointNormal(D3DXPLANE *result,
                const D3DXVECTOR3 *point,
                const D3DXVECTOR3 *normal);
D3DXPLANE *D3DXPlaneFromPoints(D3DXPLANE *result,
                const D3DXVECTOR3 *v1,
                const D3DXVECTOR3 *v2,
                const D3DXVECTOR3 *v3);
D3DXPLANE *D3DXPlaneNormalize(D3DXPLANE *result,
                const D3DXPLANE *p);
```

The function `D3DXPlaneDot` computes the dot product between the plane and the vector $\langle x, y, z, w \rangle$ as the value $ax + by + cz + dw$. `D3DXPlaneDotCoord` computes the dot product between the plane and the coordinate (x, y, z) as the value $ax + by + cz + w$. `D3DXPlaneDotNormal` computes the dot product between the plane and the normal $\langle x, y, z \rangle$ as the value $ax + by + cz$.

```
float D3DXPlaneDot(const D3DXPLANE *p,
                const D3DXVECTOR4 *v);
float D3DXPlaneDotCoord(const D3DXPLANE *p,
                const D3DXVECTOR3 *v);
float D3DXPlaneDotNormal(const D3DXPLANE *p,
                const D3DXVECTOR3 *v);
```

The intersection between a line and a plane is computed by the `D3DXPlaneIntersectLine` function. The line is defined by two points on the line given as the `v1` and `v2` arguments. The `result` parameter will contain the point on the plane that intersects the line. When the line is parallel to the plane, the `result` parameter is unchanged and the function returns `NULL`.

```
D3DXVECTOR3 *D3DXPlaneIntersectLine(D3DXVECTOR3 *result,
                const D3DXPLANE *p,
                const D3DXVECTOR3 *v1,
                const D3DXVECTOR3 *v2);
```

A plane can be transformed by a transformation matrix with the `D3DXPlaneTransform` function. The matrix passed to the function should be the inverse of the transpose of the desired transformation matrix so that the plane normal can be properly transformed.

```
D3DXPLANE *D3DXPlaneTransform(D3DXPLANE *result,
                const D3DXPLANE *p,
                const D3DXMATRIX *m);
```

16.6 Quaternions

Quaternions are an extension of complex numbers that are useful in computer graphics for representing orienting transformations (i.e. rotations). Quaternions have the nice property that the product of two quaternions produces a rotation transformation that is the composite of the rotation transformations associated with the individual quaternions. Another nice property of quaternions is that interpolating quaternions provides a much better interpolating orientation transformation than interpolating axis rotation angles or the matrices themselves. This makes quaternions well suited for keyframe animation of orienting transforms.

D3DX provides the `D3DXQUATERNION` class to represent a quaternion as four floating-point values stored in the `x`, `y`, `z`, and `w` public data members. Constructors are provided to create quaternions from an array of floating-point values or from an explicit initializer list. Casting operators are provided to expose a quaternion as an array of 4 `float`s. No range checking is performed on these arrays, so care should be taken not to read or write past the end of the array. The arithmetic operators for `D3DXQUATERNION` are summarized in table 16.7. The comparison operators perform bitwise comparison between the two quaternions. As described in chapter 1, an ϵ based floating-point comparison is preferable in most circumstances.

```
typedef struct D3DXQUATERNION
{
public:
    D3DXQUATERNION() {}
    D3DXQUATERNION(const float *);
    D3DXQUATERNION(float x, float y, float z, float w);

    // casting
    operator float *();
    operator const float *() const;
```

```
    // assignment operators
    D3DXQUATERNION &operator += (const D3DXQUATERNION &);
    D3DXQUATERNION &operator -= (const D3DXQUATERNION &);
    D3DXQUATERNION &operator *= (const D3DXQUATERNION &);
    D3DXQUATERNION &operator *= (float);
    D3DXQUATERNION &operator /= (float);

    // unary operators
    D3DXQUATERNION  operator + () const;
    D3DXQUATERNION  operator - () const;

    // binary operators
    D3DXQUATERNION operator + (const D3DXQUATERNION &) const;
    D3DXQUATERNION operator - (const D3DXQUATERNION &) const;
    D3DXQUATERNION operator * (const D3DXQUATERNION &) const;
    D3DXQUATERNION operator * (float) const;
    D3DXQUATERNION operator / (float) const;

    friend D3DXQUATERNION operator * (float, const D3DXQUATERNION &);

    BOOL operator == (const D3DXQUATERNION &) const;
    BOOL operator != (const D3DXQUATERNION &) const;

    float x, y, z, w;
} D3DXQUATERNION, *LPD3DXQUATERNION;
```

D3DX provides several functions for constructing and converting quaternions. The `D3DXQuaternionIdentity` function creates an identity quaternion. An identity quaternion results in an identity matrix when converted to a rotation matrix. `D3DXQuaternionRotationMatrix` constructs a quaternion from a rotation transformation matrix. `D3DXQuaternionRotationYawPitchRoll` builds a quaternion from the angles of rotation about the three principal coordinate axes. `D3DXQuaternionRotationAxis` builds a quaternion from an axis and an angle of rotation around that axis. The `D3DXQuaternionToAxisAngle` converts a quaternion into an axis and a corresponding angle of rotation around that axis.

```
D3DXQUATERNION *D3DXQuaternionIdentity(D3DXQUATERNION *result);
D3DXQUATERNION *D3DXQuaternionRotationMatrix(
                    D3DXQUATERNION *result,
                    const D3DXMATRIX *m);
D3DXQUATERNION *D3DXQuaternionRotationYawPitchRoll(
                    D3DXQUATERNION *result,
                    float yaw,
                    float pitch,
```

Assignment Operator	Meaning
`operator +=`	$T \leftarrow T + Q$
`operator -=`	$T \leftarrow T - Q$
`operator *=`	$T \leftarrow TQ$
`operator *=`	$T \leftarrow fT$
`operator /=`	$T \leftarrow \frac{1}{f}T$

Unary Operator	Meaning
`operator +`	$+T$
`operator -`	$-T$

Binary Operator	Meaning
`operator +`	$T + Q$
`operator -`	$T - Q$
`operator *`	TQ
`operator *`	fT
`operator /`	$\frac{1}{f}T$

Comparison Operator	Meaning
`operator ==`	$T = Q$
`operator !=`	$T \neq Q$

Table 16.7: `D3DXQUATERNION` class members. T represents the quaternion value of `*this`, Q represents a quaternion passed to a method, f represents a scalar value passed to a method.

```
                    float roll);
D3DXQUATERNION *D3DXQuaternionRotationAxis(
                    D3DXQUATERNION *result,
                    const D3DXVECTOR3 *v,
                    float angle);
void D3DXQuaternionToAxisAngle(const D3DXQUATERNION *q,
                    D3DXVECTOR3 *axis,
                    float *angle);
```

`D3DXQuaternionIsIdentity` returns TRUE when the given quaternion is an identity quaternion. The functions `D3DXQuaternionLength` and `D3DXQuaternionLengthSq` return the length and square of the length of a quaternion, respectively. The length of a quaternion $Q = \langle x, y, z, w \rangle$ is defined by the following formula.

$$|Q| = \sqrt{x^2 + y^2 + z^2 + w^2}$$

`D3DXQuaternionNormalize` normalizes a quaternion so that $|Q| = 1$.

```
BOOL D3DXQuaternionIsIdentity(const D3DXQUATERNION *q);
float D3DXQuaternionLength(const D3DXQUATERNION *q);
float D3DXQuaternionLengthSq(const D3DXQUATERNION *q);
D3DXQUATERNION *D3DXQuaternionNormalize(D3DXQUATERNION *result,
                    const D3DXQUATERNION *q);
```

A dot product between two quaternions $Q_1 = \langle x_1, y_1, z_1, w_1 \rangle$ and $Q_2 = \langle x_2, y_2, z_2, w_2 \rangle$ results in the scalar value

$$Q_1 \cdot Q_2 = x_1x_2 + y_1y_2 + z_1z_2 + w_1w_2$$

and is computed by `D3DXQuaternionDot`. The conjugate of a quaternion $Q = \langle x, y, z, w \rangle$ is given by

$$\overline{Q} = \langle -x, -y, -z, w \rangle$$

and is computed by `D3DXQuaternionConjugate`. The product of two quaternions is computed by `D3DXQuaternionMultiply`. The inverse of a quaternion Q_1 is the quaternion Q_2 such that $Q_1Q_2 = I$, where I is the identity quaternion. The inverse quaternion is computed by `D3DXQuaternionInverse`. The natural logarithm of a quaternion is computed by `D3DXQuaternionLn` and its exponential is computed by `D3DXQuaternionExp`.

```
float D3DXQuaternionDot(const D3DXQUATERNION *q1,
                    const D3DXQUATERNION *q2);
D3DXQUATERNION *D3DXQuaternionConjugate(D3DXQUATERNION *result,
                    const D3DXQUATERNION *q);
D3DXQUATERNION *D3DXQuaternionMultiply(D3DXQUATERNION *result,
                    const D3DXQUATERNION *q1,
                    const D3DXQUATERNION *q2);
D3DXQUATERNION *D3DXQuaternionInverse(D3DXQUATERNION *result,
```

```
                    const D3DXQUATERNION *q);
D3DXQUATERNION *D3DXQuaternionLn(D3DXQUATERNION *result,
                    const D3DXQUATERNION *q);
D3DXQUATERNION *D3DXQuaternionExp(D3DXQUATERNION *result,
                    const D3DXQUATERNION *q);
```

Since quaternions can be thought of as representing an orientation, its often useful to define two quaternions as interpolate between them, generating the sequence of orientations between the two quaternions. This interpolation is best visualized as interpolating a path on the surface of a sphere. The basic operation is spherical linear interpolation, provided by the `D3DXQuaternionSlerp` function. The starting orientation is given by the `q1` parameter, the ending orientation is given by the `q2` parameter and the interpolating parameter is `t`. When `t` is zero, the result is `q1` and when `t` is one, the result is `q2`.

```
D3DXQUATERNION *D3DXQuaternionSlerp(D3DXQUATERNION *result,
                    const D3DXQUATERNION *q1,
                    const D3DXQUATERNION *q2,
                    float t);
```

Spherical cubic interpolation provides smoother interpolation between two quaternions. The technique is similar to Hermite interpolation in that four control points are supplied to smoothly interpolate between the inner two control points. `D3DXQuaternionSquadSetup` takes the four control points as the parameters `q0`, `q1`, `q2` and `q3` and computes the quaternions `a`, `b` and `c` used in the cubic interpolation. `D3DXQuaternionSquad` takes the quaternions computed by the setup function and an interpolation factor `t` to compute the interpolated result. When `t` is zero, the quaternion `q1` passed to the setup function is returned. When `t` is one, the quaternion `q2` passed to the setup function is returned.

```
void D3DXQuaternionSquadSetup(D3DXQUATERNION *a,
                    D3DXQUATERNION *b,
                    D3DXQUATERNION *c,
                    const D3DXQUATERNION *q0,
                    const D3DXQUATERNION *q1,
                    const D3DXQUATERNION *q2,
                    const D3DXQUATERNION *q3);
D3DXQUATERNION *D3DXQuaternionSquad(D3DXQUATERNION *result,
                    const D3DXQUATERNION *q1,
                    const D3DXQUATERNION *a,
                    const D3DXQUATERNION *b,
                    const D3DXQUATERNION *c,
                    float t);
```

For example, the following code from the SkinnedMesh sample uses the spherical cubic interpolation functions to smoothly interpolate through an array of rotation keys. The keys at indices `i1` and `i2` define the current segment that

is used, while the key at index `i0` refers to the end of the previous segment and the key at index `i3` refers to the start of the next segment in the animation.

```
int i0 = i1 - 1;
int i3 = i2 + 1;

if(i0 < 0)
    i0 += m_cRotateKeys;

if(i3 >= (INT) m_cRotateKeys)
    i3 -= m_cRotateKeys;

D3DXQUATERNION qA, qB, qC;
D3DXQuaternionSquadSetup(&qA, &qB, &qC,
    &m_pRotateKeys[i0].quatRotate, &m_pRotateKeys[i1].quatRotate,
    &m_pRotateKeys[i2].quatRotate, &m_pRotateKeys[i3].quatRotate);

D3DXQuaternionSquad(&quat, &m_pRotateKeys[i1].quatRotate,
    &qA, &qB, &qC, fLerpValue);
```

Interpolation between three quaternions can be performed by treating the quaternions as the vertices of a triangle and using barycentric coordinates to compute the interpolated quaternion by the following formula.

$$Q = Q_1 + f(Q_2 - Q_1) + g(Q_3 - Q_1)$$

`D3DXQuaternionBaryCentric` computes the resulting quaternion Q from the three input quaternions and the barycentric coordinates.

```
D3DXQUATERNION *D3DXQuaternionBaryCentric(D3DXQUATERNION *result,
                    const D3DXQUATERNION *q1,
                    const D3DXQUATERNION *q2,
                    const D3DXQUATERNION *q3,
                    float f,
                    float g);
```