# 4.3 计时和动画

要正确实现动画效果，我们就必须记录时间，尤其是要精确测量动画帧之间的时间间隔。当帧速率高时，帧之间的时间间隔就会很短；所以，我们需要一个高精确度计时器。

## 4.3.1 性能计时器

我们使用性能计时器（或性能计数器）来实现精确的时间测量。为了使用用于查询性能计时器的Win32函数，我们必须在代码中添加包含语句“#include<windows.h>”。

性能计时器采用的时间单位称为计数（count）。我们使用**QueryPerformanceCounter**函数来获取以计数测量的当前时间值：

\_\_int64 currTime;

QueryPerformanceCounter((LARGE\_INTEGER\*)&currTime);

注意，该函数通过它的参数返回当前时间值，该参数是一个64位整数。

我们使用**QueryPerformanceFrequency**函数来获取性能计时器的频率（每秒的计数次数）：

\_\_int64 countsPerSec;

QueryPerformanceFrequency((LARGE\_INTEGER\*)&countsPerSec);

而每次计数的时间长度等于频率的倒数（这个值很小，它只是百分之几秒或者千分之几秒）：

mSecondsPerCount = 1.0 / (double)countsPerSec;

这样，要把一个时间读数**valueInCounts**转换为秒，我们只需要将它乘以转换因子**mSecondsPerCount**：

valueInSecs = valueInCounts \* mSecondsPerCount;

由**QueryPerformanceCounter**函数返回的值本身不是非常有用。我们使用**QueryPerformanceCounter**函数的主要目的是为了获取两次调用之间的时间差——在执行一段代码之前记下当前时间，在该段代码结束之后再获取一次当前时间，然后计算两者之间的差值。也就是，我们总是查看两个时间戳之间的相对差，而不是由性能计数器返回的实际值。下面的代码更好地说明了这一概念：

\_\_int64 A = 0;

QueryPerformanceCounter((LARGE\_INTEGER\*)&A);

/\* Do work \*/

\_\_int64 B = 0;

QueryPerformanceCounter((LARGE\_INTEGER\*)&B);

这样我们就可以知道执行这段代码所要花费的计数时间为（B−A），或者以秒表示的时间为（B−A）\*mSecondsPerCount。

注意：MSDN指出当使用**QueryPerformanceCounter**函数时，有以下注意事项：“在多处理器计算机中，任何一个处理器单独调用该函数都不会出现问题。但是，由于基础输入/输出系统（BIOS）或硬件抽象层（HAL）存在技术瓶颈，所以你在不同的处理器上调用该函数会得到不同的结果”。你可以使用**SetThreadAffinityMask**函数让主应用程序线程只运行在一个处理器上，不在处理器之间进行切换。

## 4.3.2 游戏计时器类

在下面的两节中，我们将讨论**GameTimer**类的实现。

class GameTimer

{

public:

 GameTimer();

 float TotalTime()const; // 单位为秒

 float DeltaTime()const; // 单位为秒

 void Reset(); // 消息循环前调用

 void Start(); // 取消暂停时调用

 void Stop(); // 暂停时调用

 void Tick(); // 每帧调用

private:

 double mSecondsPerCount;

 double mDeltaTime;

 \_\_int64 mBaseTime;

 \_\_int64 mPausedTime;

 \_\_int64 mStopTime;

 \_\_int64 mPrevTime;

 \_\_int64 mCurrTime;

 bool mStopped;

};

需要特别注意的是，构造函数查询了性能计数器的频率。其他成员函数将在随后的两节中讨论。

GameTimer::GameTimer()

: mSecondsPerCount(0.0), mDeltaTime(-1.0), mBaseTime(0),

 mPausedTime(0), mPrevTime(0), mCurrTime(0), mStopped(false)

{

 \_\_int64 countsPerSec;

 QueryPerformanceFrequency((LARGE\_INTEGER\*)&countsPerSec);

 mSecondsPerCount = 1.0 / (double)countsPerSec;

}

**注意**：**GameTimer**类的定义和实现部分都保存在了GameTimer.h和GameTimer.cpp文件中，你可以在示例代码的Common目录中找到它们。

## 4.3.3 帧之间的时间间隔

当渲染动画帧时，我们必须知道帧之间的时间间隔，以使我们根据逝去的时间长度来更新游戏中的物体。我们可以采用以下步骤来计算帧之间的时间间隔：设*t*i为第*i*帧时性能计数器返回的时间值，设*t*i-1为前一帧时性能计数器返回的时间值，那么两帧之间的时间差为Δ*t* = *t*i – *t*i-1。对于实时渲染来说，我们至少要达到每秒30帧的频率才能得到比较平滑的动画效果（我们一般可以达到更高的频率）；所以，Δ*t* = *t*i – *t*i-1通常是一个非常小的值。

下面的代码示范了Δ*t*的计算过程：

void GameTimer::Tick()

{

 if( mStopped )

 {

 mDeltaTime = 0.0;

 return;

 }

 \_\_int64 currTime;

 QueryPerformanceCounter((LARGE\_INTEGER\*)&currTime);

 mCurrTime = currTime;

 // 当前帧和上一帧之间的时间差

 mDeltaTime = (mCurrTime - mPrevTime)\*mSecondsPerCount;

 // 为计算下一帧做准备

 mPrevTime = mCurrTime;

 // 确保不为负值。DXSDK中的CDXUTTimer提到：如果处理器进入了节电模式

 // 或切换到另一个处理器，mDeltaTime会变为负值。

 if(mDeltaTime < 0.0)

 {

 mDeltaTime = 0.0;

 }

}

float GameTimer::getDeltaTime() const

{

 return (float)mDeltaTime;

}

函数**Tick**在应用程序消息循环中的调用如下：

int D3DApp::Run()

{

 MSG msg = {0};

 mTimer.Reset();

 while(msg.message != WM\_QUIT)

 {

 // 如果接收到Window消息，则处理这些消息

 if(PeekMessage( &msg, 0, 0, 0, PM\_REMOVE ))

 {

 TranslateMessage( &msg );

 DispatchMessage( &msg );

 }

 // 否则，则运行动画/游戏

 else

 {

 mTimer.Tick();

 if( !mAppPaused )

 {

 CalculateFrameStats();

 UpdateScene(mTimer.DeltaTime());

 DrawScene();

 }

 else

 {

 Sleep(100);

 }

 }

 }

 return (int)msg.wParam;

}

通过这一方式，每帧都会计算出一个Δ*t*并将它传送给**UpdateScene**方法，根据当前帧与前一帧之间的时间间隔来更新场景。下面是**Reset**方法的实现代码：

void GameTimer::Reset()

{

 \_\_int64 currTime;

 QueryPerformanceCounter((LARGE\_INTEGER\*)&currTime);

 mBaseTime = currTime;

 mPrevTime = currTime;

 mStopTime = 0;

 mStopped = false;

}

这里包含一些还未讨论过的变量（请参见4.3.3节）。不过，我们可以看到，当调用**Reset**方法时，**mPrevTime**被初始化为当前时间。这一点非常重要，因为对于动画的第一帧来说，没有前面的那一帧，也就是说没有前面的时间戳。所以个值必须在消息循环开始之前初始化。

## 4.3.4 游戏时间

另一个需要测量的时间是从应用程序开始运行时起经过的时间总量，其中不包括暂停时间；我们将这一时间称为游戏时间（game time）。下面的情景说明了游戏时间的用途。假设玩家有300秒的时间来完成一个关卡。当关卡开始时，我们会获取时间*t*start，它是从应用程序开始运行时起经过的时间总量。当关卡开始后，我们不断地将*t*start与总时间*t*进行比较。如果*t* – *t*start ＞300（如图4.8所示），就说明玩家在关卡中的用时超过了300秒，输掉了这一关。很明显，在一情景中我们不希望计算游戏的暂停时间。



**图4.8：计算从关卡开始时起的时间。注意，我们将应用程序的开始时间作为原点（0），测量相对于这个时间原点的时间值。**

游戏时间的另一个用途是通过时间函数来驱动动画运行。例如，我们希望一个灯光在时间函数的驱动下环绕着场景中的一个圆形轨道运动。灯光位置可由以下参数方程描述：

x = 10 cos*t*

y = 20

z = 10 sin*t*

这里*t*表示时间，随着*t*（时间）的增加，灯光的位置会发生改变，使灯光在平面y = 20上围绕着半径为10的圆形轨道运动。对于这种类型的动画，我们也不希望计算游戏的暂停时间；参见图4.9。



**图4.9 如果我们在*t*1时暂停，在*t*2时取消暂停，并计算暂停时间，那么当我们取消暂停时，灯光的位置会从p(*t*1) 突然跳到p(*t*2)。**

我们使用以下变量来实现游戏计时：

\_\_int64 mBaseTime;

\_\_int64 mPausedTime;

\_\_int64 mStopTime;

如4.3.3节所述，当调用**Reset**方法时，**mBaseTime**会被初始化为当前时间。我们可以把它视为从应用程序开始运行时起经过的时间总量。在多数情况下，你只会在消息循环开始之前调用一次**Reset**，之后不会再调用个方法，因为**mBaseTime**在应用程序的整个运行周期中保持不变。变量**mPausedTime**用于累计游戏的暂停时间。我们必须累计这一时间，以使我们从总的运行时间中减去暂停时间。当计时器停止时（或者说，当暂停时），**mStopTime**会帮我们记录暂停时间。

**GameTimer**类包含两个重要的方法**Stop**和**Start**，它们分别在应用程序暂停和取消暂停时调用，让GameTimer记录暂停时间。代码中的注释解释了这两个方法的实现思路。

void GameTimer::Stop()

{

 // 如果正处在暂停状态，则略过下面的操作

 if( !mStopped )

 {

 \_\_int64 currTime;

 QueryPerformanceCounter((LARGE\_INTEGER\*)&currTime);

 // 记录暂停的时间，并设置表示暂停状态的标志

 mStopTime = currTime;

 mStopped = true;

 }

}

void GameTimer::Start()

{

 \_\_int64 startTime;

 QueryPerformanceCounter((LARGE\_INTEGER\*)&startTime);

 // 累加暂停与开始之间流逝的时间

 //

 // |<-------d------->|

 // ----\*---------------\*-----------------\*------------> time

 // mBaseTime mStopTime startTime

 // 如果仍处在暂停状态

 if( mStopped )

 {

 // 则累加暂停时间

 mPausedTime += (startTime - mStopTime);

 // 因为我们重新开始计时，因此mPrevTime的值就不正确了，

 // 要将它重置为当前时间

 mPrevTime = startTime;

 // 取消暂停状态

 mStopTime = 0;

 mStopped = false;

 }

}

最后，成员函数**TotalTime**返回了自调用**Reset**之后经过的时间总量，其中不包括暂停时间。它的代码实现如下：

// 返回自调用Reset()方法之后的总时间，不包含暂停时间

float GameTimer::TotalTime()const

{

 // 如果处在暂停状态，则无需包含自暂停开始之后的时间。

 // 此外,如果我们之前已经有过暂停,则mStopTime - mBaseTime会包含暂停时间, 我们不想包含这个暂停时间，

 // 因此还要减去暂停时间：

 //

 // |<--paused time-->|

 // ----\*---------------\*-----------------\*------------\*------------\*------> time

 // mBaseTime mStopTime startTime mStopTime mCurrTime

 if( mStopped )

 {

 return (float)(((mStopTime - mPausedTime)-mBaseTime)\*mSecondsPerCount);

 }

 // mCurrTime - mBaseTime包含暂停时间,而我们不想包含暂停时间，

 // 因此我们从mCurrTime需要减去mPausedTime：

 //

 // (mCurrTime - mPausedTime) - mBaseTime

 //

 // |<--paused time-->|

 // ----\*---------------\*-----------------\*------------\*------> time

 // mBaseTime mStopTime startTime mCurrTime

 else

 {

 return (float)(((mCurrTime-mPausedTime)-mBaseTime)\*mSecondsPerCount);

 }

}

**注意**：我们的演示框架创建了一个**GameTimer**实例用于计算应用程序开始后的总时间和两帧之间的时间；你也可以创建额外的实例作为通用的秒表使用。例如，当点着一个炸弹时，你可以启动一个新的**GameTimer**，当**TotalTime**达到5秒时，你可以引发一个事件让炸弹爆炸。