C++ Builder 中高精度定时器管理类设计及其应用①

李晓京, 张利利, 胡文东, 文治洪, 马 进

(第四军医大学 航空航天医学教育部重点实验室, 西安 710032)

摘 要: 介绍一种在 BCB 开发环境基于多线程的高精度定时器类的设计实现. 由 VCL 组件库中的 TThread 派生类结合 Windows 消息机制设计了实测误差<0.5ms 的定时器类, 该类具有单实例-多定时器管理、线程工作状态可控、回调方式可控等优点, 宜用性及可定制性均优于普遍采用的系统定时器及多媒体定时器, 可在工业控制、多媒体应用设计、实时信号采集与处理等领域得到应用.

关键词: C++ Builder IDE; Windows; 定时器; 多线程; 线程优先级; Windows 消息机制; 定时精度; 实时性

Design and Application of High Resolution Timer Managing Class in C++Builder IDE

LI Xiao-Jing, ZHANG Li-Li, HU Wen-Dong, WEN Zhi-Hong, MA Jin

(Key Laboratory of Aerospace Medicine of Ministry of Education, Fourth Military Medical University, Xi'an 710032, China)

Abstract: This paper present a realization of hi-resolution timer class, based on multithread technique, in the BCB IDE. Utilizing of the class, derived from TThread class of VCL and integrated with Windows message mechanism, achieved a general timing-error < 0.5ms. The class is provided of advantages such as multiple timers - single object managing, work state of timer-thread adjusting, callback mode controlling, as well as more convenient and more customizable than traditional windows System Timer and Multimedia Timer. Thus it could be applied in projects with relatively time-critical requirement, e.g. industrial control, multimedia app development and realtime signal sampling & processing.

Key words: C++ Builder IDE; Windows; timer; multithread; thread priority; message mechanism; timing precision; real-time

相当多的应用对系统实时性有较高的要求, 需要软件定时/计时达到 ms 甚至更高的精度. 但 Windows 系统的多任务分时特性决定了单纯依靠软件来得到高精度的定时事件是较为困难的[1]. 在这一前提下, 软件开发者们提出了多种方法以获得相对高精确度的定时/计时性能, 这些方法主要包括系统定时器、多媒体定时器、主线程/多线程系统 API 查询等.

综合各种文献及实际测试结果, 系统定时器是精度最低的模式[2,3], 在 win2000 以上的操作系统中大致可得到精度在 20ms 左右的定时事件; 多媒体定时器则可以提供精度在 1ms 左右的定时事件[4], 但定时回调函数定义较为繁琐且非线程安全, 函数体内只能尽可能简单地调用少数几个 API 函数, 往往仍需借助 Windows 消息机制完成复杂任务处理, 也没有提供对定时器线程工况进行调整的接口, 使用起来并不方便; API 查询方式则是在主线程或子线程内设置条件循环, 不断调用 API 函数查询系统时间并判断达到定时条件时, 执行定时任务. 根据应用精度需求, 该方式有两类可供采用的 API 函数, 一类是 GetTickcount 函数, 该函数返回 PC 开机以来经过的 DWORD 型毫秒数, 但受各种因素影响较, 其精度误差也时常>10ms, 与系统定时器精度相近; 另一类包括 QueryPerformance Counter、QueryPerformanceFrequency 两个函数[5], 前者获得 CPU 上电以来的振荡次数, 后者获得 CPU 频率, 根据这两个数值, 可以获得精确到 ns 的计时精度. 有研究应用该方式编制简单的主线程延时程序用于产生

① 基金项目:国家自然科学基金青年项目(81202178/H2602);全军医药卫生重大项目(AWS12J003)
收稿时间:2013-01-18;收到修改稿时间:2013-03-08

2000Hz 信号输出[6], 由于程序体本身简单无额外系统开销, 具有精度较高的优点, 但这种方法并不适用于多任务程序. 也有文献将该方法与多线程技术结合在具体应用中实现了精度几十微秒的实时功能, 但未提出代码重用和多定时器管理的方法[7]. 本研究的目的, 即是设计一种精度可以达到甚至超过多媒体定时器、线程可控、回调函数线程安全、符合代码重用理念的高精度定时器管理类(High-Resolution Timer Managing Class, HRTMC).

1 多线程及TThread类

每个正在系统上运行的程序都是一个进程, 每个进程包含一或多个可在程序中独立执行的指令集合——线程, 它们负责在单个程序中执行多任务, 以提升 CPU 使用效率及用户响应能力. 随着多核 CPU 的普及, 如果不使用多线程技术, 甚至不可能充分发挥多核 CPU 的性能. 在 Windows 系统中, 线程的 CPU 占用率由其优先级决定, 共有 32 个分级, 级别越高占用的 CPU 时间片越长[8]. 事实上, 多媒体定时器就是由一个拥有较高优先级的线程触发的.

TThread 类是 Borland C++ Builder(BCB)IDE 封装的线程 VCL 基类, 其提供了线程对象的生成、运行、暂停、结束、优先级控制等一系列成员函数和属性, 利用该类可方便快捷的生成多线程应用程序[9].

2 Windows消息处理机制

众所周知, 消息是指 Windows 发出的一个通知, 告诉应用程序某个事情发生了, 需要应用程序进行相应的响应处理. 几乎所有的 Windows 窗体应用都是建立在消息处理机制上的. 在多线程应用中, 消息机制的使用相当普遍. 线程向窗体发送消息一般下两个函数完成: PostMessage 和 SendMessage. Windows 对于这两个函数的处理方式是不同的, PostMessage 将消息发送到消息队列中就返回, 由系统按 FIFO 的原则分发、处理消息. 而 SendMessage 则直接调用消息响应函数, 处理过程结束才返回[10]. 可见, 对于实时性要求高的消息应用, 后者是最佳选择.

3 HRTMC的设计

综上所述, 在 BCB 环境中, 自定义 HRTMC 基于多线程和 Windows 消息机制实现, 主要包含三部分: 作为定时器句柄并包含定时器相关设置的的 st_TimerRec 结构、TTimerThread 线程类与 THRTimer 定时器管理类.

3.1 st_TimerRec 结构

该结构实例指针将作为定时器句柄返回给用户, 包含由用户设定的定时器消息响应窗体句柄、是否强制立即执行回调函数、自定义消息值、定时周期等参数以及用于内部管理的定时触发计数、启动时刻、历经时长、对应线程指针等数据, 用户可根据需要决定是否解封该结构.

定义如下:

```
struct st_TimerRec{
  HWND hWnd; //定时器消息目的窗体
  bool bForceCallback; //是否强制立即执行定时回调函数
  DWORD dwUerMsg; //用户自定义定时消息值
  DWORD dwInterval; //定时间隔(单位: ms)
  DWORD dwCount;  //用于定时事件触发次数计数
  double dblTotalElapsed; //自定时器启动以来经过的时间(ms)
  LONGLONG llBegin; //计时器启动初始时刻(CPU周期数)
  TThread* CorrespondThread; //对应独立线程对象指针, 非独立置空
};
```

3.2 TTimerThread 类

● TTimerThread 由 TThread 类派生, 作为定时器对象的计时线程, 由 THRTimer 类实例维护所有线程实例. 该类设计基于以下考虑:

● 可以主管理线程、独立线程两种模式运行, 主管理线程可负责多个轻量定时器任务触发条件的计算, 独立线程则仅对单个复杂定时任务负责;

● 为减轻 CPU 负荷, 当定时任务周期较长(>2ms), 允许线程在等待期间进入休眠状态;

● 可由用户决定线程优先级水平;

● 可由用户决定定时回调事件执行方式(Send Message 或 PostMessage).

类定义如下:

```
class TTimerThread : public TThread
{
private:
```

```
  TList* lstHandle;  //保存来自 THRTimer 的定时器句柄列表
  st_TimerRec* pstOwnedRec;//线线程所对应的定时器句柄
  LONGLONG llCPUFrq;  //保存 CPU 频率值
  LONGLONG llSleepThreshold; //用于休眠控制的阈值
  void Initial(TList* lst,
               st_TimerRec* ptimerrec,
               bool isloresolution);  //线程初始化函数
protected:
  void __fastcall Execute();  //线程主函数
public:
  bool bIsLoResolution;//初始化参数，决定线程是否允许休眠
  //构造函数，初始化为主管理线程，维护多个定时器
  __fastcall TTimerThread(bool CreateSuspended,
                          TList* lst,
                          bool isloreslotion=true);
  //构造函数，初始化为独立线程，维护单一定时器
  __fastcall TTimerThread(bool CreateSuspended,
                          st_TimerRec* ptimerrec,
                          bool isloreslotion=true);
};
```

该类 Initial 函数为主初始化函数，构造函数均调用其完成私有数据的初始化，较为简单. 主运行函数 Execute 伪码如下:

```
void __fastcall TTimerThread::Execute()
{
  LONGLONG llNowCPUCount;
  bool bCouldSleep=true;
  LONGLONG llIdleMS=0,llTemp;
  while(!Terminated){
    QueryPerformanceCounter((LARGE_INTEGER*)&
                            llNowCPUCount);// 获取当前时刻
    if(isIndependentThread()){ //若是独立线程
      //若定时器触发时间到
      if(IsTrigerTime()){
        AddCount(); //触发次数递增
        UpdateNextTirgerTime();//更新下次触发时间条件
        CalcElapsedTime();  //计算定时器启动以来 ms 数
        if(isForceCallback())  //强制立执行即回调函数
          SendMessage(…);
        else PostMessage(…); //非强制回调
      }
      //判断是否满足可休眠条件
      bCouldSleep=CheckSleepCondition();
      llIdleMS =CalcSleepTime();
    }
    else { //若是主管理线程
      if(lstHandle==NULL)Terminate();  //不存在定时器句柄列表
      if(lstHandle->Count==0)Suspend();  //句柄列表为空，暂停
      if(csFlag!=NULL)csFlag->Acquire();
      llIdleMS=200; //休眠初始值 200ms
      for(int i=0;i<lstHandle->Count;i++){  //逐一检查定时器列表
        //若满足触发条件
        if(isTrigerTimer (lstHandle->Items[i])){
        AddCount(lstHandle->Items[i]); //触发次数递
        //更新下次触发时间条件
        UpdateNextTirgerTime(lstHandle->Items[i]);
        //计算定时器启动以来 ms 数
        CalcElapsedTime(lstHandle->Items[i]);
        //更新定时器启动历经 ms 数
        if(isForceCallback(lstHandle->Items[i])) //强制回调
          SendMessage(…);
        else PostMessage(…); //发送队列消息
        }
        //判断是否满足休眠条件
        bCouldSleep=CheckSleepCondition
(lstHandle->Items[i]);
        llIdleMS =CalcSleepTime(lstHandle->Items[i]);
      }
      if(csFlag!=NULL)csFlag->Release();
    }
    if(bCouldSleep)Sleep(llIdleMS);
  }
}
```

其中 TCriticalSection* csFlag 为全局变量,用于对定时器句柄列表的互斥访问以维护线程安全, 该变量由管理类 THRTimer 实例创建和销毁. 定时结构指针被转换为句柄类型, 作为 WParam 消息参数传递给用户窗体用于判断定时消息来源.

3.3 定时器管理类 THRTimer 设计

该类用于实现对所有定时器句柄及相关线程的维护与管理, 并向用户公开若干函数实现定时器操作及参数设置. 类定义如下:

```
class THRTimer{
private:
  //非独立、独立线程定时器句柄列表
  TList *lstHandle,*lstIndependentHandle;
  void CreateTimerList(); //创建定时器句柄列表函数
  void ClearTimerList(); //清空定时器句柄列表函数
  st_TimerRec* pstTimerRec; //通用定时器句柄指针
  TTimerThread* TimerThread; //主管理线程指针, 构造生成析构销毁
public:
  //增加一个定时器函数
  const HANDLE AddTimer(HWND hwnd, //接受定时消息的窗体句柄
                        DWORD interval,//定时周期(单位: ms)
                        DWORD usermsg,//用户的定时消息值
                        bool forcecallback=false,//是否强制回调
                        bool isindependent=false,//是否独立线程
                        bool isloRes=true); //低精度时允许休眠
  //按句柄删除定时器, 空参数则删除所有定时器
  void RemoveTimer(HANDLE handle=NULL);
  //调整指定定时器句柄线程优先级
  void ChangePriority(HANDLE timerhandle,TThreadPriority priority);
  //获取指定句柄线程优先级
  TThreadPriority GetPriority(HANDLE timerhandle);
  //构造函数, 缺省设置主管理线程优先级为普通
  THRTimer(TThreadPriority priority=tpNormal);
  ~THRTimer();
};
```

以下对三个重要公开函数详细说明.

3.3.1 AddTimer 函数

伪码如下.

```
const HANDLE THRTimer::AddTimer(…)
{
  pstTimerRec=new st_TimerRec;
  InitialHandle(pstTimerRec);//按入口参数初始化句柄结构
  if(!isindependent){  //非独立线程定时器由主管理线程管理
    pstTimerRec->CorrespondThread=NULL;// 无对应独立线程
    if (TimerThread->Suspended)TimerThread->Resume();
    CouldSleep(…);//按入口参数设置主线程是否可休眠
    csFlag->Acquire(); //独占访问定时句柄列表
    lstHandle->Add((PVOID)pstTimerRec); //纳入非独立句柄列表
    csFlag->Release(); //解除占用
  }
  else { //独立线程定时器
    TTimerThread* th=new TimerThread(…); //创建新线程
    pstTimerRec->CorrespondThread=th; //独立线程存入句柄结构
    lstIndependentHandle->Add((PVOID)pstTimerRec); //纳入独立句柄列表
  }
  return (HANDLE)pstTimerRec; //向用户返回定时器句柄
};
```

该函数用于用户申请定时器句柄, 申请后立刻启动. 函数共有 6 个入口参数, 前三个参数用于向指定窗体按固定周期发送用户自定义消息(参见类声明代码), 后三个参数则用于定时精度控制: forcecallback 决定回调函数执行方式, isindependent 决定是否独立线程定时器, isloRes 决定线程是否可休眠, 三者均有缺省值. 一般而言, 强制回调、独立线程、无休眠定时器具有最优实时性.

3.3.2 RemoveTimer 函数

伪码如下:

```
void THRTimer::RemoveTimer(HANDLE handle){
  int i;
  if(handle!=NULL){ //句柄参数不为空,
    if(IsIndependent(handle)){//独立线程
      //检索独立线程列表获取索引值
      i=GetHandleIndexFromIndependentList(handle);
      if(i>=0){
          EndThreadHandle(handle); //结束线程
          RemoveFormIndependentList(handle);
      }
      return;
    }
    //非独立线程定时器句柄, 从列表中删除
    csFlag->Acquire();
    //检索非独立线程列表获取索引值
    i=GetHandleIndexFromList(handle);
    if(i>=0)lstHandle->Delete(i);
    RemoveFormIndependentList(handle);
    csFlag->Release();
  }
  else ClearTimerList(); //空句柄或无参数, 默认清除所有定时器
};
```

该函数用于注销定时器句柄, 有一个可缺省定时器句柄参数, 缺省状况下默认注销所有定时器.

3.3.3 ChangePriority 函数

```
void THRTimer::ChangePriority(HANDLE timerhandle,
                              TThreadPriority priority)
{
  pstTimerRec=(st_TimerRec*)timerhandle;
  if(pstTimerRec->CorrespondThread!=NULL){// 调 整独立线程优先级
    pstTimerRec->CorrespondThread->Priority=priority;
  }
  else TimerThread->Priority=priority;  //调整主管理线程优先级
}
```

ChangePriority 函数用于动态调整定时器线程优先级. 线程优先级是多任务系统中影响定时器精度的关键因素, 因此定义单独的函数以便用户根据应用需求动态调整.

3.4 HRTMC 主要数据类型之间的关系

如图 1 所示, 管理类 THRTimer 实例使用两个句柄列表负责所有定时器句柄管理, 直接管理唯一的非独立定时器线程, 并通过独立线程句柄列表间接管理独立定时器线程(可多个); 各定时器句柄保存客户应用申请时传入的窗体句柄供定时线程向其发送消息, 独立定时器句柄还保存对应线程实例指针以便管理类进行维护; 非独立定时线程实例保存初始化时由管理类传入的非独立句柄列表指针, 负责向其中各句柄对应客户窗体发送定时消息; 独立定时线程保存对应定时句柄, 并向相应客户窗体发送定时消息.

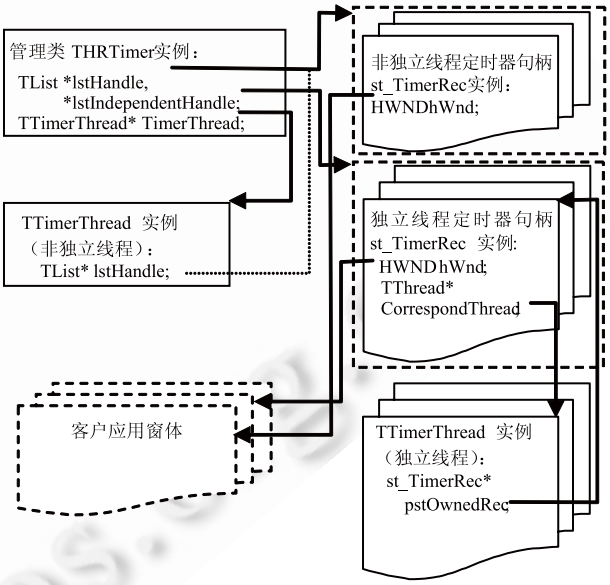

图 1 HRTMC 主要数据类型之间的关系

4 HRTMC应用

BCB 中, THRTimer 类的应用极为简便, 一般性的过程如图 2 所示.

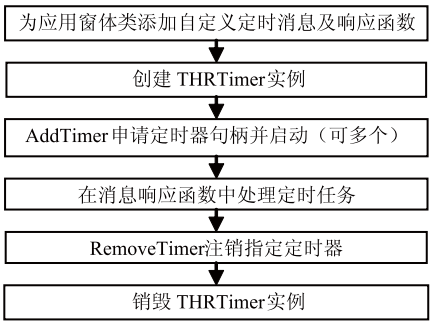

图 2 HRTMC 应用流程

尽管提供了较为丰富的定时器控制参数, 但大多数情况下, 默认设置启动定时器即可. 对于只申请一个定时器的应用, 甚至连句柄管理都不需要, 如下示例:

```
//FormCreate 函数中
THRTimer hrTimer=new THRTimer();
……
// 申请定时器, 可不保留返回句柄, MSG_HRTIMER 为宏定义消息值
hrTimer->AddTimer(Handle,20,MSG_HRTIMER);
……
//注销定时器, 贯穿窗体生命周期的定时器这一步也可省略
hrTimer->RemoveTimer();
……
//FormClose 函数中销毁管理类, 析构函数自动释放所有资源
delete hrTimer;
```

5 精度测试比较分析与讨论

上述 HRTMC 在 BCB 6.0 IDE 中编译通过, 在 WinXP 下运行正常. 为了检验 HRTMC 的精度, 在相同系统环境条件下, 分别用多种定时方法以 20ms 周期采样 200 次, 将实测计时值与理想计时值的差值进行统计分析, 结果如表 1.

表 1 多种定时方法的性能比较(采样 200 次)

定时类型	均值（ms）	标准差（ms）	误差 >1ms 次数	误差 >0.1ms 次数	CPU 占用（%）
HRTMC1	0.057	0.059	0	8	25
HRTMC2	0.069	0.164	3	7	8
多媒体定时	-0.317	0.29	0	157	8
系统定时器	1132.2	651.1	200	200	2
GetTickcount	4.2	4.519	136	144	-

注: HRTMC1 为最优精度设置: 强制回调、独立线程、无休眠、最高优先级; HRTMC2 为缺省设置: 非强制回调、非独立线程、可休眠、普通优先级; GetTickcount 函数为主窗体线程死循环查询. CPU 为 4 核 3.0GHz.

由表 1 可见, HRTMC 最优设置下精度可控制在 0.2ms 以内, 相应的 CPU 占用达到 25%, 几乎完全占用了一个内核的处理能力; 而缺省设置下, 计时精度也可大致控制在 0.5ms 以内. 相较于多媒体定时器, 缺省设置精度高但稳定性稍差(3 次误差值>1ms), 而最优设置则无论精度和稳定性均高于多媒体定时器; 此外, 多媒体定时器多数触发早于理想时刻(误差均值为负). GetTickcount 查询精确性与前三种方法明显不在一个数量级上; 系统定时器则不但误差大, 且有累积误差(200 周期时误差>2s), 性能最差.

应用 HRTMC 时要注意几点:

① 定时任务若在一个周期内无法完成, 势必导致系统响应不及, 定时精度也就无从谈起. 应充分发挥多核处理器的并行处理能力, 或降低复杂任务执行频率;

② 非强制回调任务由主窗体线程按消息顺序依次执行, 因而是线程安全的, 定时精度稍低. 而强制回调定时任务精度高, 但必须考虑线程安全;

③ 非独立定时器由同一线程管理, 极端情况下, 多个定时器按 FIFO 原则执行/触发, 时延将随其排序增长. 尽量避免同时申请多个周期重叠的定时器句柄, 或以独立线程模式申请(会增加系统资源开销)以避免此类情况发生;

④ 归根结底, 定时精度的提高是以牺牲 CPU 效率换取的. 因而对高实时性应用而言, 提供宽松的系统环境, 严格控制定时器使用数量都是必要的.

6 结论

综上所述, HRTMC 提供了较为丰富的控制参数和函数, 可由用户综合考虑需求和系统环境定制定时器, 能够实现优于多媒体定时器的定时和计时功能, 最高精度<0.2ms 左右, 进一步还可将其封装为 ActiveX 控件在其他开发环境中应用[11], 为高实时性需求的应用程序开发提供了快捷、简便的方法.

参考文献

1 Timmerman M, Monfret JC. Windows NT as a real-time OS? Real-Time Magazine Issue 1997/2.April 1997.

2 徐立萍,张健.Windows 2000 平台下精确定时研究.微型电脑应用,2005,(4).

3 覃曾锋,林景栋,韩兴连.高精度定时器和多线程技术在音乐灯光实时控制中的应用.自动化技术与应用,2008,27(10): 50–53.

4 张志明,孙广清,王磊.Windows 下高精度软件定时器的研究与实现.微型机与应用,2003,1:55–57.

(下转第 201 页)

立锚点的相应 VRML 代码如下:

```
Anchor
{    url "geoInformation.aspx?id=1"
description   "the first information"
parameter["target=_blank"]
}
```

其中组节点 Anchor 引起一个 url 链接, 即名为 geoInformation 的 asp 文件, 返回 ID 的值, 以决定要查询数据库的哪一条记录; description 是用来描述链接的信息, parameter 参数用来向浏览器提供链接时的附加信息, target=_blank 表示以新窗口显示.

利用 ASP 内置对象 Request 的 QuerryString 方法, 通过 VRML 的超链接返回 ID 的值, 本文使用的是 access 数据库, 主要包括钻孔数据表和地层数据表.

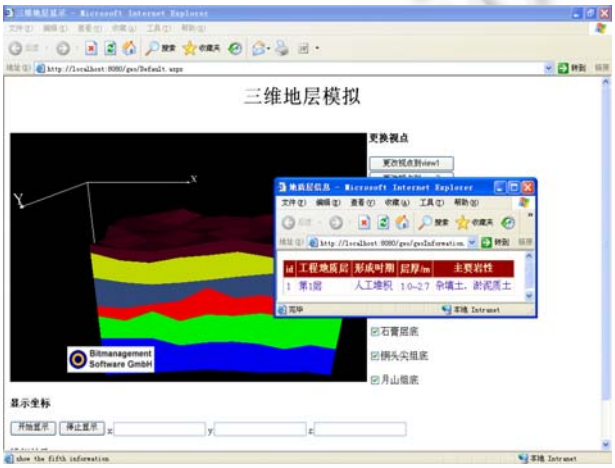

图 4 地质层属性查询

3 结语

本文通过对虚拟现实技术的研究, 实现了 VRML 结合 Javascript 脚本语言对虚拟场景的交互功能, 在 VRML 中引入交互功能可以为虚拟场景提供更出色的用户体验. 如本文中的三维层状地质体模型, 能在场景漫游过程中获取钻孔和地层的属性信息, 可以满足地质工作者对地质体结构、属性等方面的需求. 同时也为 VRML 语言在虚拟地质方面的应用提供了一个很好的开发案例. 但鉴于地质体的复杂性和更多实用功能的设计, 还需随着实际应用的深入而逐步完善.

参考文献

1 吴迪,吴亿维,刘军,等.基于 VRML 及 Javascript 的气垫导轨上交互式三维虚拟实验.物理实验,2012,32(6):17–20.

2 Wang BJ, Shi B, Song Z. A simple approach to 3D geological modelling and visualization.Bull Eng Geol Environ, 2009,68:559–565.

3 芮小平,余志伟,许友志,等.VRML 在三维地质曲面动态显示中的应用.煤田地质与勘探,2001,29(3):5–7.

4 梅晓仁,张瑞新.基于 VRML 的煤矿床三维可视化方法研究.中国矿业大学学报,2004,33(6):665–667.

5 阳化冰.虚拟现实构造语言—VRML 语言.北京:北京航空航天大学出版社,2000.

6 Lemon AM, Jones NL. Building solid models from bore-holes and user defined cross section.Computers and Geosciences, 2003,29(3) :547–555.

7 陈树敏,徐迪威.VRML 网络交互性研究及其在石油管道设计中的应用.现代计算机,2010,(2):131–136.

8 冯桂珍,王大鸣,池建斌,等.基于 VRML 的移动模型造桥机的动态模拟.工程图学学报,2010,(2):139–143.

9 王德新,魏东,黄有群.在 VRML 文件中对数据库信息的访问.沈阳工业大学学报,2002,24(5):417–420.

(上接第 150 页)

5 http://support.microsoft.com/kb/172338/en-us: How To Use QueryPerformance Counter to Time Code.Article ID:172338-Last Review:January 20,2007-Revision:3.3.

6 林砺宗,王启春,张松,王天威.微秒级定时器的制作及其在数字 I/0 卡控制步进电机中的应用.机械与电子,2010,8: 53–55.

7 刘春凤,田延岭.Windows 操作系统下的软件定时器的设计与应用,2004,10(5):38–47.

8 Russinovich M. Inside NT's Interrupt Handling: NT's interrupt management affects real-time system applicability. NT Internals Paper in Windows 2000 Magazine, 1997. http://www.winntmag.com/Articles/Index.cfm?ArticleID=298&SearchString=windows%20nt%20scheduler.

9 Borland c++Builder 6.0 Help: TThread. Borland Softeware Corpration, 2002.

10 Microsoft Visualstudio 2010 文档:Messages and Message Queues.

11 石伟.Delphi 编写高精度定时器 ActiveX 控件.电脑编程技巧与维护,2007.6–8.