International Journal of Mechanical, Industrial and Aerospace Sciences
ISSN: 2517-9950
Vol:2, No:3, 2008

The Application of Specialized Memory Manager in Interactive CAD Systems

Wei Song, and Lian-he Yang

***Abstract*—**Interactive CAD systems have to allocate and deallocate memory frequently. Frequent memory allocation and deallocation can play a significant role in degrading application performance. An application may use memory in a very specific way and pay a performance penalty for functionality it does not need. We could counter that by developing specialized memory managers.

***Keywords*—**Interactive CAD systems, Specialized Memory Manager.

I. Introduction

INTERACTIVE CAD systems (Fig. 1) have to deal with lots of graphic objects. Every graphic object is made up of basic graphic element (Fig. 2), and every graphic element is determined by different points (Fig. 3). The class diagram of UML is shown as Fig.4. Therefore, interactive CAD systems have to allocate and deallocate memory frequently [1]–[3].

Frequent memory allocation and deallocation can play a significant role in degrading application performance. The performance degradation stems from the fact that the default memory manager is, by nature, general purpose. An application may use memory in a very specific way and pay a performance penalty for functionality it does not need. We could counter that by developing specialized memory managers.

The default memory manager is, by design, a general-purpose one. This is what we get when we call the global *new*() and *delete*(). The implementation of these two functions cannot make any simplifying assumptions. They manage memory in the process context, and since a process may spawn multiple threads, *new*() and *delete*() must be able to operate in a multithreaded environment. In addition, the size of memory requests may vary from one request to the next. This flexibility trades off with speed. The more you have to compute, the more cycles it is going to consume [4].

Manuscript received September 10, 2006. This work described in this paper was supported by the Nature Science Foundation of Tianjin under grant 033602611.

Wei Song is with School of Computer Technology and Automation of Tianjin Polytechnic University, Tianjin, China (corresponding author, e-mail: songvxvei@gmail.com).

Lian-he Yang is with School of Computer Technology and Automation of Tianjin Polytechnic University, Tianjin, China (e-mail: yanglh@tjpu.edu.cn).

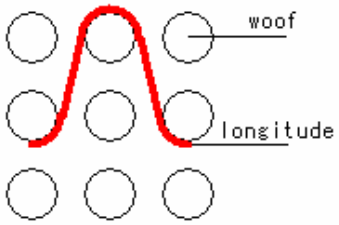

Fig. 1 Interactive CAD system

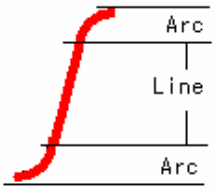

Fig. 2 Composition of graph

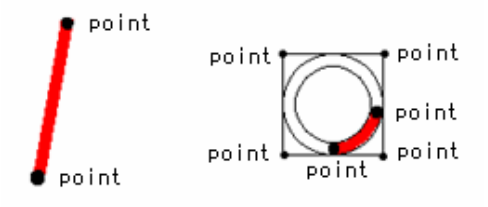

Fig. 3 Composition of element

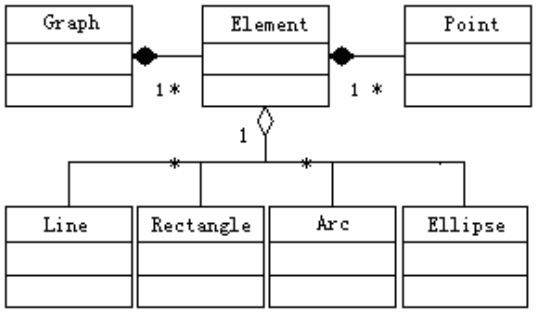

Fig. 4 Class diagram of UML

It is often the case that client code does not need the full power of the global *new*() and *delete*(). It may be that client code only (or mostly) needs memory chunks of specific size. It may be that client code operates in a single-threaded environment where the concurrency protection provided by the

default *new*() and *delete*() is not really necessary. If this is the case, utilizing the full power of those functions is a waste of CPU cycles. We can gain significant efficiency by tailoring a memory allocation scheme to better match our specific requirements.

II. Principle of the Default Memory Manager

When we create a *new*-expression, two things occur: First, storage is allocated using the operator *new*(), and then the constructor is called. In a *delete*-expression, the destructor is called, and then storage is deallocated using the operator *delete*() [5].

When we write code like this,

string *ps = new string("Memory Management");

The *new* we are using is the *new* operator. This operator is built into the language and, like *sizeof*, we can't change its meaning: it always does the same thing. What it does is twofold. First, it allocates enough memory to hold an object of the type requested. In the example above, it allocates enough memory to hold a string object. Second, it calls a constructor to initialize an object in the memory that was allocated. The *new* operator always does those two things; we can't change its behavior in any way.

The *new* operator calls a function to perform the requisite memory allocation, and we can rewrite or overload that function to change its behavior. The name of the function the *new* operator calls to allocate memory is operator *new*.

The operator *new* function is usually declared like this:

void * operator new(size_t size);

The return type is void *, because this function returns a pointer to raw, uninitialized memory. The size_t parameter specifies how much memory to allocate.

Only responsibility of operator *new* is to allocate memory. It knows nothing about constructors. All operators *new* understand is memory allocation. It is the job of the *new* operator to take the raw memory that operator *new* returns and transform it into an object. When our compilers see a statement like:

string *ps = new string("Memory Management");

Our compilers do three things. First, they call operator *new* to get raw memory for a string object. Second, they call a constructor to initialize the object in the memory. Third, they make ps point to the new object.

III. Principle of Specialized Memory Manager

As for *new* operator, we can't change its behavior---calling a constructor in any way, but we can change how the memory for an object is allocated by rewriting or overloading operator *new* to change its behavior [6].

When we overload operator *new* and operator *delete*, it's important to remember that we're changing only the way raw storage is allocated. The compiler will simply call our *new* instead of the default version to allocate storage, and then call the constructor for that storage. So, although the compiler allocates storage and calls the constructor when it sees *new*, all we can change when we overload *new* is the storage allocation portion. (*delete* has a similar limitation.)

Overloading *new* and *delete* is like overloading any other operator. However, we have a choice of overloading the global allocator or using a different allocator for a particular class.

This is the drastic approach, when the global versions of *new* and *delete* are unsatisfactory for the whole system. If we overload the global versions, we make the defaults completely inaccessible---we can't even call them from inside our redefinitions.

Although you don't have to explicitly say *static*, when you overload *new* and *delete* for a class, you're creating *static* member functions. As before, the syntax is the same as overloading any other operator. When the compiler sees you use *new* to create an object of your class, it chooses the member operator *new* over the global version. However, the global versions of *new* and *delete* are used for all other types of objects (unless they have their own *new* and *delete*).

The local operator *new* has the same syntax as the global one. All it does is search through the allocation map looking for a false value, then sets that location to true to indicate it's been allocated and returns the address of the corresponding memory block.

IV. An Example of Specialized Memory Manager

We define Point class. To avoid frequent hits to the default manager, the Point class will maintain a static linked list of preallocated Point objects, which will serve as the free list of available objects (Fig. 5). When we need a new Point object, we will get one from the free list. When we are done with an object, we will return it to the free list for future allocations [4].

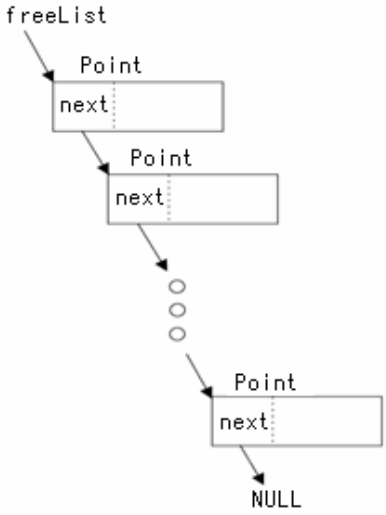

Fig. 5 A free list of point object

The free list is declared as a static member of the Point class. The static free list is manipulated by the Point *new*() and *delete*() operators. These operators overload the global ones.

```
class Point
{
public:
   Point(int a = 0, int b = 1, Point * c = 0 ):m(a), n(b),
         next(c){ }
   inline void * operator new(size_t size);
   inline void operator delete(void * doomed, size_t size);
   static void newMemList(){ expandMemoryList();}
   static void deleteMemList();
private:
   static Point * memoryList;
   static void expandMemoryList();
   enum{ EXPANSION_SIZE = 50 };
   Point * next;
   int m;
   int n;
};
```

Operator *new*() allocates a new Point object from the free list. If the free list is empty, it will get expanded. We pick off the head of the free list and return it after adjusting the free list pointer.

```
inline void * Point::operator new(size_t size)
{
   if(memoryList == 0)
      expandMemoryList();
   Point * head = memoryList;
   memoryList = head->next;
   return head;
}
```

Operator *delete*() returns a Point object to the free list by simply adding it to the front of the free list.

```
inline void Point::operator delete(void * doomed, size_t size)
{
   Point * head = static_cast<Point *>(doomed);
   head->next = memoryList;
   memoryList = head;
}
```

When the free list is exhausted, we must allocate more Point objects from the heap.

```
void Point::expandMemoryList()
{
   size_t size = sizeof(Point);
   Point * runner = (Point *)(new char[size]);
   memoryList = runner;
   for(int i = 0; i < EXPANSION_SIZE; i++)
   {
      runner->next = (Point *)(new char[size]);
      runner = runner->next;
   }
   runner->next = 0;
}
```

Our implementation is such that the free list never shrinks; it will grow to a steady-state size and stay there. The free list is deallocated at the end of the program.

```
void Point::deleteMemList()
{
   Point * nextPtr;
   for(nextPtr=memoryList;nextPtr!=0;nextPtr=
   memoryList)
   {
      memoryList = memoryList->next;
      delete[] nextPtr;
   }
}
```

To measure the performance of the specialized Point memory manager and the default Point memory manager we executed the following test:

```
int main()
{  ·····
   Point * array [1000];
   Point::newMemList();
   for(int j = 0; j < 500; j++)
      {
         for(int i = 0; i < 1000; i++)
         {
            array[i] = new Point();
         }
         for(i = 0; i < 1000; i++)
         {
            delete array[i];
         }
      }
   Point::deleteMemList();
   ·····
}
```

V. Experiment Environment and Test Result

A. Experiment Environment

In order to contrast the specialized Point memory manager and the default Point memory manager, we have built two experiment environments.

Experiment environment one:

CPU: Celeron (850MHz); Basic frequency: 100MHZ;
Memory: 256MB; Operating system: Windows XP.

Experiment environment two:

CPU: Pentium 4 (2.8GHz); Basic frequency: 800MHZ;
Memory: 256MB; Operating system: Windows XP.

B. Test Result

Test result as shown in Table I:

TABLE I
EXECUTION TIME AND RADIO

	environment one	environment two
default	931ms	375ms
specialized	160ms	63ms
ratio	5.8	6.0

VI. CONCLUSION

The memory allocation system used by *new* and *delete* is designed for general-purpose use. In special situations, however, it doesn't serve our needs. The most common reason to change the allocator is efficiency: We might be creating and destroying so many objects of a particular class that it has become a speed bottleneck. C++ allows us to overload *new* and *delete* to implement our own storage allocation scheme, so we can handle problems like this.

Since we are in a single-threaded environment, our Point memory management routines do not bother with concurrency issues. We do not protect memory manipulation because we don't have any critical sections to worry about. We also take advantage of the fact that all allocations are fixed size---the size of a Point object. Fixed-size allocations are much simpler; there are a lot less computations to perform (like finding the next available memory chunk big enough to satisfy the request).

Therefore, the application of specialized Point memory manager can gain significant efficiency in interactive CAD systems. In my experiment environment, using specialized Point memory managers is much better than changing a new computer.

REFERENCES

[1] Guo-ming Huang. Develop interactive CAD systems by using Visual C++.net. Beijing: Publishing house of electronics industry, 2003, pp. 236-278.
[2] Lian-he Yang. Computer simulation of elastic properties on any structure 3-D woven composite. Acta materiae compositae sinica, 2000.2.
[3] Lian-he Yang. CAD on elastic properties of 3D woven composites. Journal of Tianjin Polytechnic University, 2005,4.
[4] Dov Bulka, David Mayhew. Efficient C++ Performance Programming Techniques. Beijing: Tsinghua University Press, 2003. pp. 55-65.
[5] Scott Meyers. More Effective C++. Beijing: China Power Press, 2003. pp. 38-39.
[6] Bruce Eckel. Thinking in C++: Volume one: introduction to Standard C++. Beijing: China Machine Press, 2002. pp. 305-324.

Wei Song was born in Hebei Province, P.R. China, in 1980. Currently, he is studying towards master's degree in Computer application technology in Tianjin Polytechnic University in China. His interest is in the fields of CAD and database. Now he is making his graduation topic of interactive CAD system of three-dimensional woven composites.

Lian-he Yang is a professor of School of Computer Technology and Automation of Tianjin Polytechnic University. His research direction is computer simulation and database.