Double Ended Stack allocator

-Clément JACOB-

I) Presentation:

In game programming, it's often usefull to allocate new memory dynamically, with *malloc()* or *new* operator. But sadly, these two methods suffers from problems. When you're asking for memory dynamically, the operating system kernel takes the control. For example, on Windows platform, the user space execution environment switch to the kernel execution environment, to create the free space you're asking. This operation is costly in term of speed of execution. The switch from user mode to kernel mode is long. So, there are tips to decrease this time. That's what this paper is talking about. It presents one method to reduce the cost created by *malloc()* or the *new* operator.

II) Overview:

The method presented here is the double-ended stack. But before talking about a double-ended one, let's talk about a normal stack, or a simple-ended stack. This method is often used in game development to reduce the dynamic memory allocation cost we've seen before. The principle of that kind of allocators is really simple:

1. You first create a big buffer block of free space with *malloc()* or *new* operator.
2. When an object asks for memory, you only pass it the pointer to the top of the stack, only if there is still enough place!
3. Then, you increment the address of the top of the stack by the size requested by the object.
4. Repeat 2,3...

After you've allocated many objects, you can free its, but only in the reverse order you've allocated its. To free something, you only had to say that the top of the stack is the latest top of the stack address minus the top object size you're freeing. Figure 1 depicts how it works.

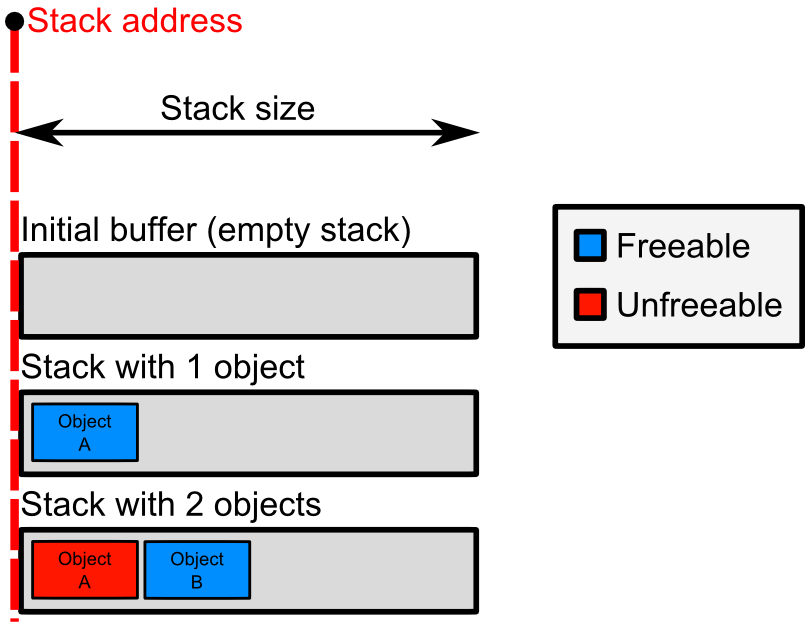

Figure 1 : Single-ended stack allocation example

This kind of allocation system is useful in games that use a lot of objects in a level. After the level has ended, you can delete all your objects by freeing its in the reverse order you've allocated its.

Now that you've learned how stuff works for a single-ended stack, we can see how it works for the double-ended stack. The main disadvantage of a stack is that you can't delete any object you want while you don't delete its top object, but this top object can not be deleted while don't delete its top object, etc... You can easily see the problem with that example:

The first level of your game allocate many small objects, such as sprites, bitmap, fonts, matrices, vectors. After that, you allocate a handle to file of your filesystem. But the fact is that you need to pass it trough the levels. Problem: you can not free all the small objects you've allocated during the level because of the file handle you need to keep alive trough the levels. So your application is growing in RAM, and unfortunately, it will stop working properly when the stack will be full!

One solution is to allocate in the "right" way. In this example, you should allocate the file handle first, after that, you'll be able to obtain and free memory for your small objects.

Sometimes, it's not possible. You need to have two stacks, or a double-ended stack! In that way, you'll be able to keep one stack for your small objects, that have a small lifetime, a level time, whilst another stack keeps persistent objects, such as a configuration file handler that is always asked for informations through levels.

III) Double-Ended stack theory: