STL is slow

<EDIT: I've updated the text and image to reflect an improved test code>

Hi everyone,

We all know the STL containers are extremely convenient but their convenience comes at a price: they perform a lot of safety and management tests internally which of course means less performance. Is the convenience worth the performance cost? I wanted to quantify this more carefully...

In our engine, we've used a few basic containers: Maps, lists, priority queues. If these, the lists are the most common (All 'able) so I figured I'd start with this one. (Side note: the priority queue gets quite a workout with the time manager, so it's next on to do)

So let's look at list performance: Reading up on STL containers, we learn that they manage their own pool of 'nodes': when you remove an item form the list, the now empty container is stored for later reuse on the next insert. In my case: the widgets themselves inherit the link nodes and so serve as their own container.

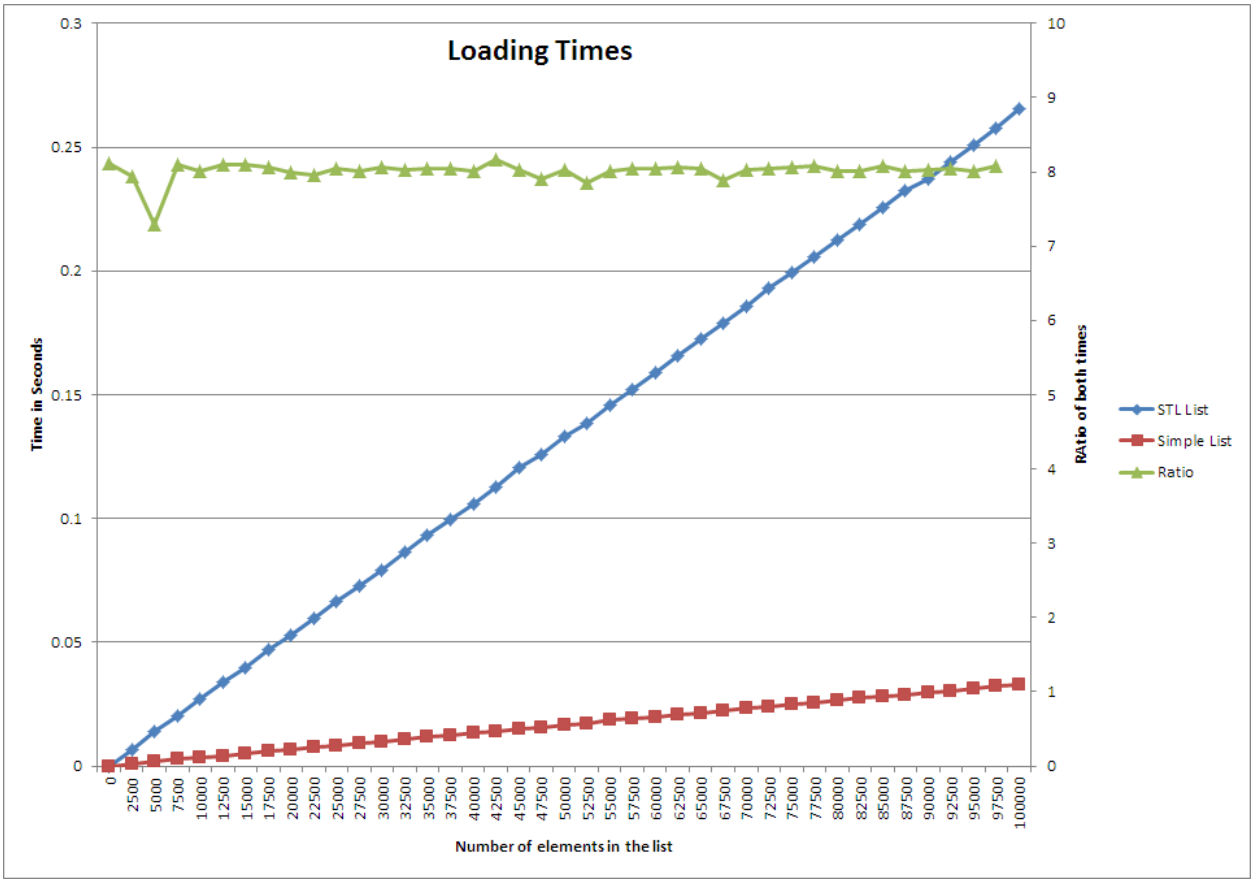
As far as the engine's list usage, there are two scenario I'm most concerned about:
1. The cost of adding new objects in the scene
2. Normal looping operation with no inserts or delete: just the cost of a normal looping

Obviously the processing time will vary quite a bit depending on the game objects. However, I'm really only interested in the raw cost of these operations on the lists themselves. Therefore, my methodology is this:

- Our lists are almost always list of pointers so Instead of gameobjects pointers, I'll use pointer to a 'widget' class containing a single integer as data
- To 'load' the list with n elements, I'll simply create new widgets with data = 1, 2, 3, … , n
- When looping through a list of n elements, I'll simply change each integer (adding one to data). I need to do something to make sure the compiler doesn't optimize out the loop altogether, but I don't want the something to be expensive

Load Times:

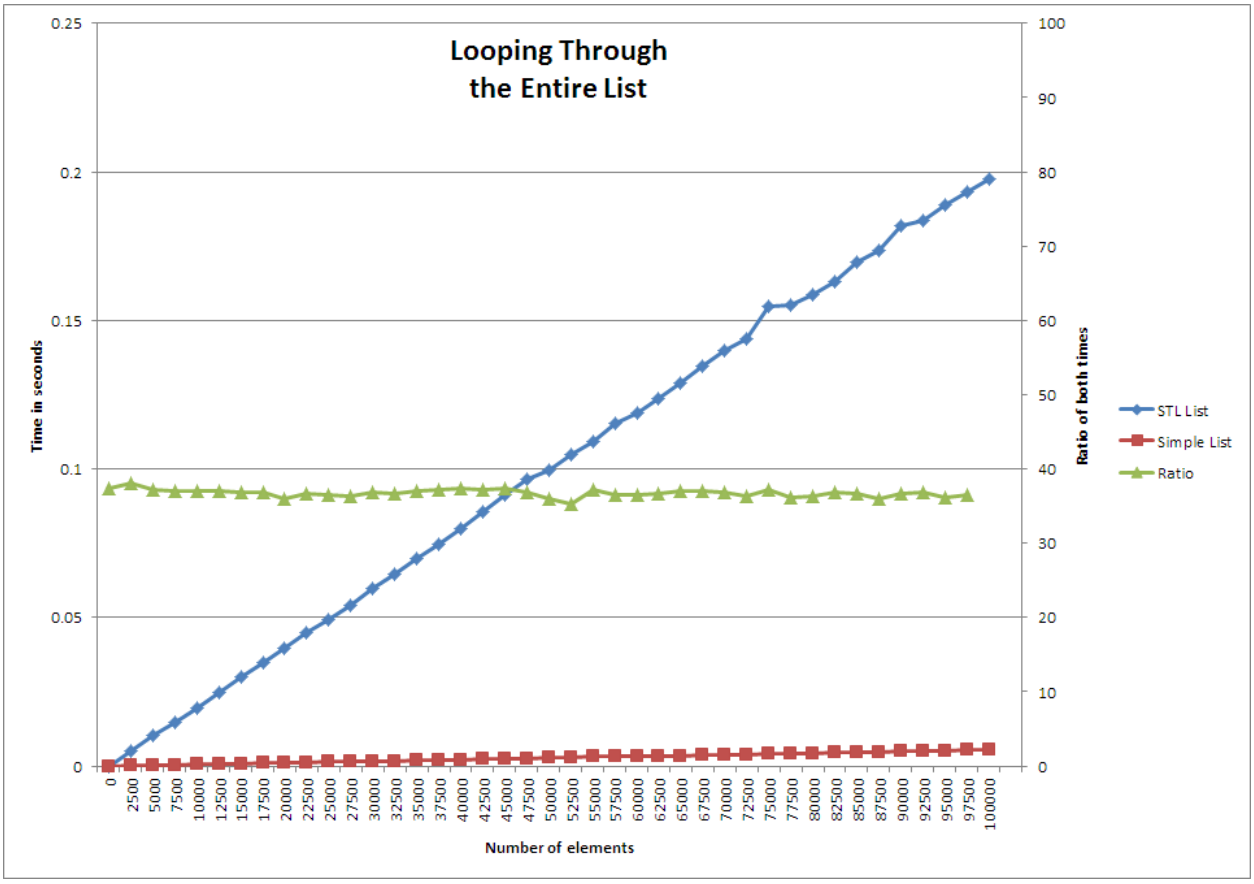
Below is the chart for load times from 1-100,000 elements. Before each load, the lists were emptied (though they kept their pool of empty containers). The blue and red plots are the loading times for both lists measured in seconds on the primary (left vertical) axis and the green plot is their ratio (unit-less, secondary vertical axis on right)



Clearly, STL is significantly slower. It's roughly 8 times slower and pretty constant regardless of the value of n.

Looping Time

For this plot, we pre-load the list with n elements (again, n ranging from 1 through 100,000) and then loop from beginning to end. Only the looping time is tracked. As before: blue and red plots are the times for both lists measured in seconds on the primary (left vertical) axis and the green plot is their ratio (unit-less, secondary vertical axis on right)



This is the most stunning data: STL is consistently 36 times slower across all values of n. Considering that 1 frame = 1/60 second = 0.016666, the STL container would blow all that time just lopping through 8000 objects on the scene (never mind doing all the other engine stuff for that frame)

As a point of comparison: back in week 4 when I presented performance data on the tiered collision system, we were running against a limit around 2500 objects on the scene: the improved collision system wasn't showing any improvement anymore because we were running against 'something else' as a bottleneck Similarly: my particle system (Week 7) caps out around 5000 particles before the fps dips below 30. . It will be interesting to re-run those numbers without STL as our containers.

I definitely need to reconsider my stance on STL for the engine sequence….