Keeping Things in (Z)Order

Itay Duvdevani

August 23, 2012

Abstract

The MoMinis Game Engine renderer and touch handler requires that game objects will be sorted in a back-to-front (or front-to-back) order when textures with translucency and touch-events are in use. To keep this management overhead to a minimum when a large number of objects is to be managed, we developed a method for keeping objects in the correct Z-order at constant-time complexity that is inspired by the Linux Kernel's $O(1)$ scheduler

Contents

1 Why Keep Things in Z-Order?

It should come as no surprise that some of the tasks the MoMinis game engine is responsible for, beside driving the game's compiled logic, is to draw pretty things to the screen, and respond to touch input.

For that, objects need to be arranged in a back-to-front order, to decide which object receives touch input, and which gets drawn first.

We'll examine *why* we need it for these two things separately.

1.1 The Touch Handler

Our touch handler is a simple adapter between the platform and the game logic running on the engine. The adapter expects screen-level touch events from the OS, and dispatches game-level events - so when I click at a certain location on the screen, the object at that location receives a logical touch-down-event instead of just "The user clicked at: (x, y)".

We provide this layer of abstraction to save game developers the need to handle low-level touch-events and decide which object should respond. It is very important that our development platform be kept suitable for entry-level developers, and be easy to use with as little hassle as possible.

The case where game objects are stacked on one another and a touch event is received at a location that's within multiple objects can be difficult to handle at the game developer's level. That's why we decided that the touch event in this case is dispatched to the top-most object that's visible under the the finger, and that's it.

The easiest way to do it is to scan all the objects in a front-to-back order and find the first that can handle the event.

1.2 The Render Queue

The MoMinis Game Engine is powered by OpenGL ES on both Android and iPhone. For compatibility reasons, we are limited to the OpenGL ES 1.1 API.

At compile time, our compiler will generate the game's assets from the asset set provided by the game developer, and will arrange all drawn objects in a set of atlas textures as best as it can. Since hi-res games can have big assets and asset generation is done automatically, not by hand, we cannot avoid "scene fragmentation" - the situation where different parts of the same scene lies in different textures.

As an internal optimization, just before submitting textures and geometry to OpenGL, we group objects by texture, to minimize texture swapping. However, due to the way the fixed render pipeline works in OpenGL ES 1.1, we must submit geometry in a back-to-front order when translucent textures are involved - otherwise, Z-culling and backbuffer blending interfere, and objects that should be partially visible behind a translucent object will not be, giving a strange feeling to the scene.

2 The Challenge

The demands from our cross-platform development environment and the games it produces are challenging:

- Inexperienced, non-programmer developers should be able to create nice games easily
- Games should have good performance

- Games should be pretty
- Final game size should be small
- Games should be portable
- Developer should be bothered with porting issues as little as possible (preferably none at all)

These demands conflict most of the time, and makes the game engine development a challenge.

Game's logic can get quite complex pretty quickly when you're making a fully-fledged game. You have all these power ups and bonuses and special animations and sounds that change your basic game mechanics temporarily, and in some games you have to dynamically generate the level as the user advances (Ninja Chicken, Jelly Jump). As our platform developers aren't always seasoned programmers, or simply doesn't know the in-and-outs of the game engine's implementation, game logic is at times written in a sub-optimal fashion. In addition, common practices used by our game developers may worsen the situation, usually with no real alternative due to lack of certain features in the platform.

This means that sometimes a game developer will create and destroy many auxiliary objects in a short time during game-play, making instance management a challenge. When we first encountered performance issues with object creation, we were able to identify the Z-order management section as a bottleneck.

2.1 The Problem

The entity that manages the game holds all existing objects in an array sorted by Z-order. Every time an object was being created, destroyed, or it's Z-order changed - it was removed from the array (moving all the elements after it one index back) and re-inserted at the correct location after a binary search (again, moving all the objects after it one place forward).

Since scanning the array for the object is an $O(n)$ operation, removing and inserting an element is again an $O(n)$ operation, and finding the new spot is a $O(\log n)$ operation - all the basic actions on an object were proportional to the number of objects existing at that moment.

Needless to say, as games got more complex and required more objects, this became a problem.

2.1.1 Runtime Complexity Analysis

create an object $O(\log n)$ binary-search for the correct place, then inserting, possible shifting the entire array: $O(n)$

destroy an object Find the sprite in the array and remove it, possibly shifting the entire array: $O(n)$

changing object's Z-order Remove, then add: $O(n)$

Not so good.

3 Initial Optimization Experiments

We needed to fix it, and we needed to fix it fast. At the time, this problem was one of several that blocked the release of a ready-to-ship game.

At first we took a conservative approach, trying to make as little changes as possible so close to a release. We concluded that there's no reason for the collection to be kept in order at all times - just when rendering and touch-handling. Our first attempt was to defer sorting the array to these two occasions.

This should have addressed the object creation problem, reducing it from $O(n)$ to $O(1)$, as we intended to simply append the new object to the end of the list. This would also cut down by half Z-order changes, as we were no longer adding the object to the collection as we used to.

At first we tried good-ol'-quicksort for the just-in-time sort. Unfortunately, the game engine's contract with the developer is that objects at the same Z-order are sub-ordered by the order their Z-value changed - meaning our sorting algorithm should be stable, and quicksort isn't. (It took us a good while to figure that *that* was what was causing the weird phenomenons we were seeing with the game's graphics)

Somewhat humbled by our reckless arrogance, thinking this was such an easy problem to fix, we looked for a good sorting algorithm that is both stable and efficient. Tree-sort was the next thing we tried. Though the sort was now stable, it was also very slow. We failed at identifying that we were using Tree-sort and a mostly-sorted collection, which is Tree-sort's Achilles Heel - in that case you end up with a degenerate tree and efficiency is no more.

Reading a little bit on the web we encountered all sorts of sorting algorithms and variations on existing algorithms that could or could not do, but we didn't have the time to start learning and understanding each one and decide if it was appropriate for our needs.

Back to quicksort then. This time we applied a bias to the Z-order, giving a monotonic index for each object that got it's Z-order changed. This wasn't perfect, but it was reasonable for the time we had.

Eventually, this optimization didn't make it to the release - we were able to improve various parts of the game enough to make it releasable, by avoiding creation of multiple objects in a single logical iteration, thus not triggering the Z-order problem in the first place.

Now we had some time to find our silver-bullet.

4 $O(1)$

Back when I was running Red Hat Linux 8 with kernel 2.4.18 (or something), I remember reading that the under-development Linux 2.6 had an all-new $O(1)$ scheduler[1]. The news about this scheduler were that it could schedule many tasks in a constant time, no matter how loaded was the system. This scheduler held up till 2.6.23, when it was replaced by the $\log n$ Completely Fair Scheduler.

What made this an $O(1)$ scheduler was that the author, Ingo Molnar, decided not to store priority information in the tasks themselves and sort them in a collection according to these priorities. Instead, he chose to store the task's priority information in the collection it was contained in - he created a different

[1] http://www.ibm.com/developerworks/linux/library/l-scheduler/

queue for each priority level, allowing him direct access to the head of each queue. Since the number of priority levels was predetermined, scanning all priority queues is done in constant time.

Since tasks behave almost erratically, much like objects in our games, but still must be kept in order for scheduling, we tried a new approach, based on the Linux solution.

4.1 The Ideal Case

We'll first describe our solution in the ideal case, where we are free to set the external constraints. Lets begin by defining an object's Z-order to be a natural, bounded number (say, between 0 and 100), where lower Z-order means an object is at the back.

The idea is to have a "bucket" for every legal Z-order, holding all objects that have that Z-order by order of insertion. We'll implement a "bucket" with a double-ended linked-list, which allows us an $O(1)$ insertion of a new object to its end. Traversing the entire structure is trivial - all we have to do is scan the objects in the first bucket, and move to the next until there are no more buckets.

However, we still have to confront the removal of objects from a bucket (either when it is being destroyed, or moved to another bucket as its Z-order changes). To avoid having to scan the entire collection to find the location of the object's link in the bucket's list, we'll use an auxiliary field in the object itself - we're going to cache the current linked-list link holding the object in the bucket *inside the sprite*, allowing us $O(1)$ access to it.

We're creating a back-reference from an object to its location in the bucket it is currently in.

From here things look trivial. Since selecting a bucket is matter of array lookup by index, adding to the end of a linked list is $O(1)$ and removing a link from the linked-list is also $O(1)$ (given the link object itself), we can achieve an always-sorted collection with $O(1)$ runtime for all the actions we need.

Let's examine how it's implemented[2]. First, A game object will have to be able to report its Z-order, and hold the auxiliary link:

```
interface ZSortable {
    public int getZOrder();
    public Unlinkable getCurrentLink();
    public void setCurrentLink(Unlinkable currentLink);
}
```

For the sake of abstraction, we're not holding the actual linked-list link in the object, but we hold it through an interface that allows us to remove the link from the list:

```
interface Unlinkable {
    public void unlink();
}
```

[2] I've included only interface listings for things that are trivial to implement. For full code listing, checkout: https://github.com/mominis/zorder and the `SimpleZCollection` class

Next step is to implement the list that will hold a bucket's contents. Remember that it has to expose the link that holds an object so we could cache it:

```
interface ZLinkedList extends Iterable<ZSortable> {
    public Unlinkable append(ZSortable object);
    public Iterator<ZSortable> iterator();
}
```

Now all we have to do is to create a bucket for each legal Z-order:

```
ZLinkedList[] buckets = new ZLinkedList[MAX_Z_LEVEL + 1];
for (int i = 0; i <= MAX_Z_LEVEL ; ++i) {
        buckets[i] = new ZLinkedListImpl();
}
```

And by that we've written our basic data structure. Lets examine each action separately, and see how we make it an $O(1)$ operation.

4.1.1 Object Creation

To add a new object to the collection, all we have to do is to append it to the end of the bucket of its initial Z-order, and cache the link in the object. That will also ensure sub-order of same-Z objects:

```
void add(ZSortable object) {
    // ... check that 'object' isn't null ...
    int zOrder = object.getZOrder();
    // ... assert that: zOrder >= 0 and zOrder <= MAX_Z_LEVEL
    Unlinkable link = buckets[zOrder].add(object);
    object.setCurrentLink(link);
}
```

Since this is a double-ended linked-list, appending to its end is an $O(1)$ action.

4.1.2 Destroying an Object

To remove an object from the Z-collection, all we have to do is unlink it from the bucket's list:

```
void remove(ZSortable object) {
        // ... check that 'object' isn't null ...
        // ... check that 'object' has a valid link ...
        object.getCurrentLink().unlink();
        object.setCurrentLink(null);
}
```

Since unlinking a link from a double-linked-list is an $O(1)$ operation, and we have a direct access to the link, this is also the runtime complexity for removing an object from the collection.

4.1.3 Changing Z-order

Changing the Z-order of an object is as simple as removing it from the previous bucket, and adding it to the new one, just like before:

```
void change(ZSortable object) {
        // ... check that 'object' isn't null ...
        // ... check that 'object' has a valid link ...
        remove(object);
        add(object);
}
```

Note that we're not skipping the operation even if the object was being "changed" to the same Z as its current - this is on purpose as we have to pop the object up to be above all the rest according to the engine's contract with the developer. Since we're only using $O(1)$ operations, this is also an $O(1)$ operation.

4.1.4 Traversing in Order

Traversing in order (either back-to-front or front-to-back) is trivial - we traverse all the buckets in the desired order, and for each bucket we traverse its list in the desired order (for back-to-front we'll iterate normally, and for front-to-back we'll iterate in reverse on both buckets and bucket-lists)

Since advancing in a linked-list is an $O(1)$ operation, traversing the entire collection is $O(n)$.

4.2 The Real World

Though the previous solution will work well in new designs, we couldn't use it as it was in our engine. The reason being, as always, backward compatibility.

The engine works in fixed-point arithmetic, and supports "non-integer" and negative Z-orders. In addition, the range of possible Z-orders spans the entire integer space - way too many for an array (there's more than four billion possible values). Moreover, *every* object created gets the default maximal Z-order *possible* ($2^{31} - 1$), which is more than two billion.

To preserve backward compatibility with old games, but still enjoy the benefits of the new Z-management collection, we've taken a best-effort approach. We allow negative an non-integer Z-levels, as well as very high or low values - but these aren't as optimal as natural bounded Z-orders. We also provide a special-case optimization for the default Z-order.

First thing we did was to replace the buckets array with a linked-list. This will allow us to open new buckets for non-integer Z-levels in between "optimized" buckets, or add new large/small buckets on-demand.

We also allocate at the end of the list the last level, for the maximal default entry. Every item in the buckets list holds a list of sortables, and the Z-order it represents.

Initially, the buckets list is initialized with a bucket for each optimized integer Z-order, plus the maximal level. Then, the links of the buckets list holding the buckets themselves are put in an array (we call it the "quick-access" array). As before, we cache the internal structure the linked-list uses.

This ensures we can always access optimized Z-orders buckets quickly, even when new Z-order buckets are inserted in-between.

When we need to add a new object, we first check if it's Z-order is the default order. If so, it is appended to that bucket, which we hold direct reference to. Otherwise, we'll test whether it's an optimized Z-order. If it is, we can simple append it to the proper bucket after looking it up in the quick-access array.

If this is an unoptimized Z-order, we'll fall back to an $O(n)$ search method that will look for an existing bucket with the new Z-order. We simply traverse all the buckets in order, starting from the nearest optimized one, and try to find a bucket with that Z-order. If we can't find one, we'll insert a new bucket between links in the appropriate location in the buckets list.

Support for negative Z-order is similar.

This allows us to support old games, but give new games that follow the guidelines better performance[3].

4.3 Conclusion

...

[3]You can find the complete implementation of this collection at: https://github.com/mominis/zorder in the `FixedPointZCollection` class