iOS Performance Best Practices

Performance Prescriptive Guidance, Requirements, and Notes from the Ayars Animation Development Teams

© 2010 Ayars Animation Inc.

Contents

Introduction 3
The Dispose Pattern 3
NIB Usage 7
NULL Pointers 8
Threading and iOS Timers 8
Pointer Assignments and Retain Count 9
Debugging Retain Count Issues 11

Introduction

This document was created as a reference guide for developers at Ayars Animation Inc. For any application to provide a good user experience on the iOS platform, it is important to consider performance. This document does not dive into the fine details of the iOS platform, rather it is prescriptive guidance document that has been bored out of experiencing the right and wrong approaches to achieve a well performing application. That being said, this is a working document that will continually be updated to provide our development team with the best advice for making high quality iOS applications.

The Dispose Pattern

For highest performance it is a good idea for an application to have immediate control to free memory. This principle applies to any platform, even those where garbage collection is available. The idea is to free memory at the time that it is no longer needed. Although this can cause memory thrashing in some situations, in general it is the preferred approach. There is a pattern called the Dispose Pattern that assists in ensuring that memory is freed as soon as possible. As well, this pattern protects against many of the problems associated with memory management. Such things as circular references, null pointers, and duplicate de-referencing. The implementation is fairly simple. For class structures that may be allocated, create a *dispose* method in that structure. All class level memory to be freed should be de-allocated in that method. The dispose method should be called within the *dealloc* method and explicitly by the class that instanced the structure - when the particular object is no longer needed. But won't the dispose method be called twice? Implicitly by the structure's own *dealloc* method and explicitly by the instancing class? Yes. And that is no problem at all. This will ensure that if, for some reason, the instancing class does not call dispose, it, at least, will be called if *dealloc* is called. But doesn't this cause memory to be freed twice? No. According the pattern the dispose method should be managed in such a way that germane code of the method can only be called once. This is accomplished by using a simple boolean flag. Here's a good example of a very simple *dispose* method.

```
-(void)dispose
{
        if(disposed == FALSE)
        {
                disposed = TRUE;
                //release or free objects here
        }
}
```

Our development team ensures that all inherited base classes implement a *dispose* method.

The following is a discussion note regarding circular references. What is a circular reference? If you have ever had a scenario where you expect *dealloc* to be called and you are sure that you have released all objects, and yet the *dealloc* method on your one of your instanced objects is not being called, then you may be experiencing a circular reference. Here's an example.

```
@interface MyObject : NSObject
{
        AnotherObject* anotherObject;
}
@property (nonatomic, retain) AnotherObject* anotherObject;
@end
```

```
@interface AnotherObject : NSObject
{
        MyObject* myObject;
}
@property (nonatomic, retain) MyObject* myObject;
@end

MyObject* myObject = [[MyObject alloc] init];
AnotherObject* anotherObject = [[AnotherObject alloc] init];

myObject.anotherObject = anotherObject;
anotherObject.myObject = myObject;
```

In this example both instanced objects hold references to each other - a circular reference. Since both object properties, the myObject property and anotherObject property, were created using the retain attribute, the retain counts on both objects were incremented when the properties were assigned their respective pointers.

Now, if you have followed the examples up to this point, you may see the problem with the following code. We will release our objects in our *dealloc* method on each of the objects.

```
@implementation MyObject
-(void)dealloc
{
        [anotherObject release];
        [super dealloc];
}
@end

@implementation AnotherObject
-(void)dealloc
{
        [myObject release];
        [super dealloc];
}
@end
```

Because the objects reference each other, neither *dealloc* will be called since each keeps one another alive by its reference. A very typical example of this problem is when using delegates. So, here's a more typical example.

```
@interface MyObject : NSObject
{
        UtilityObject* utility;
}

@property (nonatomic, retain) UtilityObject* utility;

-(void)useUtility;
-(void)onComplete;

@end


@implementation MyObject

@synchronize utility;

-(void)dealloc
{
        [utility release];
        [super dealloc];
}
```

```
-(void)useUtility
{
	utility.delegate = self;
	utility.completionSelector = @selector(onComplete);
	[utility doSomething];
}

-(void)onComplete
{
	//react to completion
}

@end

@interface UtilityObject : NSObject
{
	NSObject* delegate;
	SEL completionSelector;
}

@property (nonatomic, retain) NSObject* delegate;
@property (nonatomic) SEL completionSelector;

-(void)doSomething;
-(void)notifyDelegate;

@end

@implementation UtilityObject

@synchronize delegate;
@synchronize completionSelector;

-(void)dealloc
{
	[delegate release];
	[super release];
}

-(void)doSomething
{
	//execute something, then notify completion selector
	if(delegate != NULL && completionSelector != NULL)
	{
		[delegate performSelector:completionSelector];
	}
}

@end
```

You may note that we could have released the UtilityObject instance in the *onComplete* method. And that is true and for some cases that will be quite doable. However, it is more likely that the UtilityObject must be persisted for the life of the instancing class, i.e., it may be used more than once. After all we did name it 'UtilityObject'. In such a case, we expect that the instancing class of the 'MyObject' class will release the MyObject instance and allow the *dealloc* method to be called; therefore releasing the UtilityObject instance. However, the UtilityObject instance holds a reference to the MyObject instance. Therefore, even after the instancing class of MyObject releases its reference to the MyObject instance, the UtilityObject instance still keeps the MyObject instance's *dealloc* from being called since it still holds a reference. Our solution is to ensure that when the instancing class of MyObject instance is called, it also calls the *dispose* method on the MyObject class - thereby releasing the UtilityObject instance and setting the chain of events that will ultimately free both objects.

Here's the proper example.

```
@interface MyObject : NSObject
{
        bool disposed;
        UtilityObject* utility;
}

@property (nonatomic, retain) UtilityObject* utility;

-(void)dispose;
-(void)useUtility
-(void)onComplete;
-(void)releaseUtilityObject;

@end


@implementation MyObject

@synchronize utility;

-(void)dealloc
{
        [self dispose];
        [super dealloc];
}

-(void)dispose
{
        if(disposed == FALSE)
        {
                disposed = TRUE;
                [self releaseUtilityObject];
        }
}

-(void)releaseUtilityObject
{
        if(utility != NULL)
        {
                [utility dispose];
                [utility release];
                utility = NULL;
        }
}

-(void)useUtility
{
        utility.delegate = self;
        utility.completionSelector = @selector(onComplete);
        [utility doSomething];
}

-(void)onComplete
{
        //react to completion
}

@end

@interface UtilityObject : NSObject
{
        bool disposed;
        NSObject* delegate;
        SEL completionSelector;
}

@property (nonatomic, retain) NSObject* delegate;
@property (nonatomic) SEL completionSelector;

-(void)dispose;
-(void)doSomething;
```

```
-(void)notifyDelegate;
-(void)releaseMyObject;

@end

@implementation UtilityObject

@synchronize delegate;
@synchronize completionSelector;

-(void)dealloc
{
        [self dispose];
        [super release];
}

-(void)dispose
{
        if(disposed == FALSE)
        {
                disposed = TRUE;
                [self releaseMyObject];
        }
}

-(void)releaseMyObject
{
        if(myObject != NULL)
        {
                [myObject dispose];
                [myObject release];
                myObject = NULL;
        }
}

-(void)doSomething
{
        //execute something, then notify completion selector
        if(delegate != NULL && completionSelector != NULL)
        {
                [delegate performSelector:completionSelector];
        }
}

@end
```

NIB Usage

NIB files preload all objects in the NIB file. Meaning that if the NIB holds 100 UIView objects and each of those UIView objects holds two UIImageView objects, then when loaded, the allocated memory will hold 100 UIView instances and 200 UIImageView instances. From a memory management standpoint this may not be sustainable. Therefore, it is a requirement for our development team to ensure the following.

- Each NIB only contains only one parent UIView.
- Only one covering view may be loaded at one time.
  This means that if our main window is completely covered by a view, then only that view and its children may be loaded into memory. Any other view must be completely de-allocated.
- Lazy load images that are not required for viewing when the NIB is initially loaded.

NULL Pointers

Objective-C is very gracious and, in some cases, handles NULL pointers so that the application continues to run smoothly. However, NULL pointers can cause serious problems as well as create hard to find bugs. What is a NULL pointer? Here's a very obvious coding example.

```
NSObject* myObject = NULL;
[myObject doSomething];
```

If you executed this code on the iOS platform nothing would happen. If you executed similar code using C or C++, you would crash the application. This was an obvious example. The more likely scenario is when you create a class member that is never initialized and is later released in, say, the *dealloc* method. Therefore, it is a good practice to test for a null pointer condition prior to using the pointer.

```
if(myObject != NULL)
{
        [myObject doSomething];
}
```

In some scenarios it just does not make sense to check - it can be highly redundant and bloat the code. For our development team, the only requirement for this practice is to implement it in 'release' type methods - as can be observed in the Dispose Pattern example provided. For instance, if there is an object that may be freed more that once during the life of the parenting object, then we create a 'release' method that must be called by the dispose method and by any other method that seeks to release the object during the life of the parenting object. Here's an example of a 'release' type method.

```
-(void)releaseMyObject
{
        if(myObject != NULL)
        {
                [myObject dispose];
                [myObject release];
                myObject = NULL;
        }
}
```

Threading and iOS Timers

Using multiple threads has the power to make an application scalable up to the number of processors in the device. So, typically you want to use threading to achieve the highest performance. However, in our experience on the iOS platform, there is little advantage to using multiple threads. Threading is scalable, but it also comes at a steep development cost. Not necessarily in the time to develop the initial multi-threaded application, but the number of bugs that *will* be introduced and the ability to correct them. Debugging multiple-threaded applications raises the development curve significantly. So, if you can avoid such development, you will have avoided a number of extra development hours. But threading not only provides the ability to scale an application, it also provides the ability to conduct asynchronous operations. This is where iOS timers comes into play. iOS timers run on the main thread, yet can behave asynchronously. It may even be possible (no real research has been exercised by our team) that iOS does or will transparently use multiple threads under the covers.

The following represents prescriptive guidance for iOS timers.

- When creating a timer, always create it at the class level.

This provides the ability to retain the timer and have full control over when it is released.

- Always retain the timer instance.
- Always create a 'release' method for the timer and ensure that method is called in the initiating method, the timer delegate method, and in the *dispose* method.
  There are several variations of timer use that can slightly change this guidance. However, the general principle remains. That is that the control over the timer life should be well managed.
- Always invalidate, release, and set the timer to NULL in the 'release' method.

Here's an example of the guidance.

```
-(void)dealloc
{
        [self dispose];
        [super dealloc];
}

-(void)dispose
{
        if(disposed == FALSE)
        {
                disposed = TRUE;
                [self releaseMyTimer];
        }
}

-(void)releaseMyTimer
{
        if(myTimer != NULL)
        {
                [myTimer invalidate];
                [myTimer release];
                myTimer = NULL;
        }
}

-(void)useMyTimer
{
        [self releaseMyTimer];

        myTimer = [[NSTimer scheduledTimerWithTimeInterval:timestep
                        target:self
                        selector:@selector(mySelector)
                        userInfo:nil
                        repeats:TRUE] retain];
}

-(void)mySelector
{
        if(stopTimer == FALSE)
        {
                //do some work
        }
        else
        {
                [self releaseMyTimer];
        }
}
```

Pointer Assignments and Retain Count

It is tempting for a seasoned C or C++ developer to pass a pointer from class to class without incrementing the retain count. On the iOS platform this is completely legal and in many cases will not result in any issues. However, the applications that our organization develops rely heavily on asynchronous operations for both audio and animated material. As a result, our teams are required to always increment the retain count when passing an object pointer. Here's an example that demonstrates our reasoning.

- An instance of the NSArray class is created in Object-A.
- The pointer of the NSArray is passed to Object-B for processing.
- Object-B uses the NSArray in an asynchronous UIView animation.
- While the asynchronous animation is executing Object-A is dereferenced and releases its reference to NSArray.
- Object-B attempts to access the NSArray and encounters an error since the NSArray memory has been released. Object-A released its ownership of the NSArray and in turn brought the reference count to zero - causing de-allocation of the object's memory. Object-B never obtained ownership of the NSArray since it never incremented the reference count on NSArray.

In order to prevent these types of problems, it is a requirement for our teams to create properties that automatically increment the retain counts when passing objects (pointers to those objects). As a result of this requirement, objects should not be passed through methods. They should be explicitly assigned - ensuring that the retain count is properly incremented.

There are four primary steps involved creating an object property. These steps should always be followed when creating a class level object that may be shared between classes.

- Define the pointer.
  ```
  MyObject* myPointerToMyObject;
  ```

- Define the property with the retain attribute.
  ```
  @property(nonatomic, retain) MyObject* myPointerToMyObject;
  ```

- Synchronize the property.
  ```
  @synchronize myPointerToMyObject;
  ```

- Create a release method for the property call that method in the parenting class' *dispose* method.
  ```
  -(void)releaseMyPointerToMyObject
  {
          if(myPointerToMyObject != NULL)
          {
                  [myPointerToMyObject dispose]; //if the method exists
                  [myPointerToMyObject release];
                  myPointerToMyObject = NULL;
          }
  }
  ```

If these steps are followed, then you can ensure that objects will be persisted as long as they are necessary and will help to avoid many unexpected and hard-to-fix bugs.

Debugging Retain Count Issues

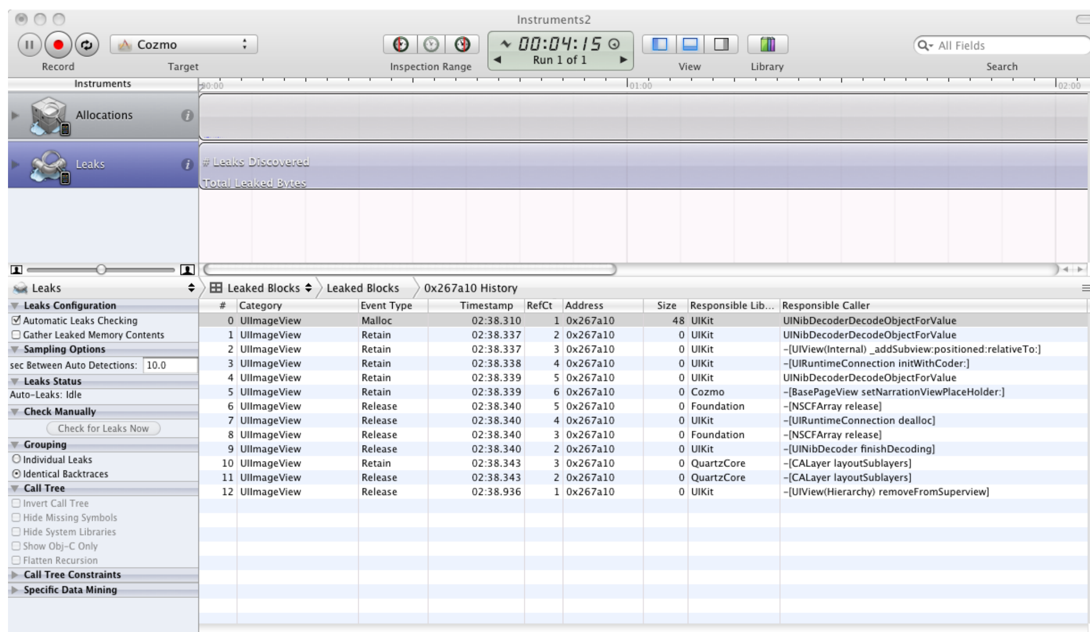
There are two methods that we use for handling retain count problems such as referencing bad memory and memory leaks.

- In development code for debugging the problem, place NSLog statements that outputs the retain count for suspect objects.

```
NSLog(@"Retain Count: %d on Object %x", myObject.retainCount, myObject);
```

- The other method is to use the Leaks tool or the Allocation tool; both of which are integrated with XCode. The Leaks tool attempts to report only leaking objects while the Allocation tool can report allocations for all objects (living or not). You can drill down on a suspect object and observe the log of its retain count history (the RefCt column).

The following is a screenshot of the Leaks application in action. Using the Event Type column and the RecCt column, you can trace the history of the 'Retain Count'. The Responsible Library column and the Responsible Caller column provide strong clues about where one might have missed a necessary call to de-reference the object.



©2010 Ayars Animation Inc.