Lost In Space

DRAFT 2

Michael S. Engber
Apple Computer - Newton ToolBox Group

This article was (will be) published in the May 1994 issue of PIE Developers magazine. For information about PIE Developers, contact Creative Digital Systems at CDS.SEM@APPLELINK.APPLE.COM or 415.621.4252.

Introduction

This article discusses different data structures for storing data in your package. The data structures are compared with respect to both space and speed. The purpose of this article is not to give the definitive best way to store your data. It is to show you some possibilities and give you a starting point for your own experiments. I assume the reader has some experience using NTK to write Newton applications.

Data In Your Package

Data your application gathers from the user must be stored in soups. That pretty much determines the data structure you need to use for dynamic data.

This article primarily concerns itself with data stored in packages. This means the data is read-only since packages are stored in protected memory. There are a variety of reasons you might store data in your package (as opposed to using soups). Some examples are listed below.

- The data might be static. For example, a table of state names and their abbreviations.
- Although the user might enter data dynamically, there might be a large initial set of data you need to include in your package.
- Your application might have its data split up in separate package. For example, you might have a travel guide application that keeps the data for each state in its own package. This allows the user to load only the data of interest.

The Raw Data

The data that forms the basis for this article is defined in the file `"Data.f"`. It's defined as an array of 300 frames as shown below. For the purposes of our tests we use `slot2` as the key for indexing, although `slot1` could work as well.

```
gData := [
{slot1: 000, slot2: "foo 000", slot3: 42, slot4: true},
{slot1: 001, slot2: "foo 001", slot3: 42, slot4: true},
{slot1: 002, slot2: "foo 002", slot3: 42, slot4: true},
É
{slot1: 297, slot2: "foo 297", slot3: 242, slot4: true},
{slot1: 298, slot2: "foo 298", slot3: 242, slot4: true},
{slot1: 299, slot2: "foo 299", slot3: 242, slot4: true},
];
```

The projects discussed in this article load `"Data.f"` and then use the `gData` array to create the actual data structures of interest.

`"Data.f"` should be easily adapted to represent the data your application.

The Code

This article is based on code in the "LostInSpace" folder. You should obtain a copy of this code so you can run your own experiments. The rest of this section gives a quick overview of the code.

SpaceWars

This project builds the same package using one of several data structures. The difference in package size between these builds is used to compare the data structures for size.

TimeWars

This project builds a package containing all the data structures. The application has various buttons you can use to run the benchmarks. The results are output using Print() so you need to run them with the NTK Inspector connected.

StorePart

This folder contains a package with a store part. When you load it, a new store is made available on Newton. This store contains a soup with the data set used in my experiments.

Data.f

This file defines the basic data set used in our experiments.

DataUtils.f

This file defines routines for creating various data structures.

SearchUtils.f

This file defines routines for searching the various data structures using binary search.

The Data Structures

Array of Frames - Naive

This data structure uses `gData` with no further processing. It is included for purposes of illustrating a pitfall of NewtonScript programming, not as a serious candidate for representing your data.

Associated with each frame is another data structure called a map. The map is used locate the value of a slot given the slot's name. Frames with common structures can share maps, saving space. However, the way `gData` is defined in `"Data.f"` does not use any map sharing. Even though the frames have identical structure, they each have their own map.

In general, each frame constructor in your code has its own map. It would be possible for the compiler to detect frames with identical maps and cause them to share maps. However, this optimization is not currently performed by NTK.

Array of Frames - Map Sharing

This data structure uses the function MakeDataFrame to generate an array, just like `gData`, except that all the frames share a common map. The easiest way to ensure your frame share maps is to make sure they're all generated by the same frame constructor. Usually this is done by using a function to create all your frames. For example:

```
func MakeDataFrame(s1,s2,s3,s4) {slot1: s1, slot2: s2, slot3: s3, slot4:
s4};
```

You may recall the standard functions `SetBounds` and `RelBounds` which return bounds frames. You might have thought them pointless since it's almost as easy to write:

```
local bounds := {left: 0, top: 0, right: 100, bottom: 100};
```

as to write:

```
local bounds := SetBounds(0,0,100,100);
```

Now you know that using `SetBounds` creates bounds frames that map share – a good reason to use this function. You should write similar functions for your own types of frames. In fact the frames returned by the run-time version of `SetBounds` use a shared map that resides in ROM.

An alternative way to create frames with a shared map is to make them from a Clone of a common frame. Consider the code below.

```
constant kDummy := '{s1: nil, s2: nil};
É
local frame1 := Clone(kDummy );
frame1.s1 := 42;
frame1.s2 := true;

local frame2 := Clone(kDummy );
frame2.s1 := 43;
```

Both `frame1` and `frame2` will share a common map. I point out this second technique only for completness. Writing functions to build your frames is a better approach. It makes for better legible and maintainabiliy. In addition, using a frame constructor to create your frame is generally faster than calling Clone followed by multiple assignment statments.

Array of Arrays

This data structure uses an arrays rather than frames to represent the elements of `gData`. For example, the following element of `gData`:

```
{slot1: 002, slot2: "foo 002", slot3: 42, slot4: true}
```

is represented as:

```
[002, "foo 002", 42, true]
```

Using frames makes for a more legible data structure. We want to measure the price we pay for this legibility.

Frame of Frames

This data structure replaces the `gData` array with a frame containing a slot for each element of `gData`. The `slot2` values are used as the slot names. For example, consider the following portion of `gData`.

```
[
É
{slot1: 297, slot2: "foo 297", slot3: 242, slot4: true},
{slot1: 298, slot2: "foo 298", slot3: 242, slot4: true},
É
]
```

The corresponding part of the frame-of-frames data structure is:

```
{
É
|foo 297|: {slot1: 297, slot3: 242, slot4: true},
|foo 298|: {slot1: 298, slot3: 242, slot4: true},
É
}
```

Using a frame instead of as an array lets us utilize of NewtonScript's slot lookup operation to retrieve our data. To retrieve the data for the `"foo 297"` entry we can use code like:

```
frameOfFrames.(Intern("foo 297"))
```

Notice that we use the Intern function to turn the string key value into a symbol. Slot names are symbols, not strings. It is important to note that characters in symbols (and therefore, slot names) are restricted to 7-bit ASCII values. This means if you wish your slot names to contain special characters you will need to transform them to 7-bit ASCII in some way. It should not be difficult to come up with a suitable encoding scheme.

Binary Objects

This data structure encodes the entire `gData` array as a single binary object. Each element of `gData` takes up 15 bytes of the object as follows:

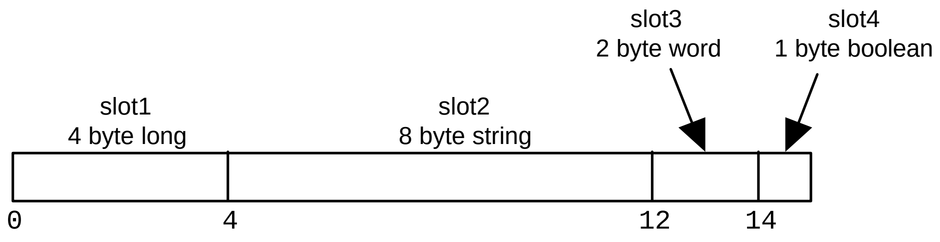

Figure 1 - Layout of Binary Object Data Structure

The binary object is accessed using the various NewtonScript StuffXXX and ExtractXXX functions.

An alternative way to use binary data from NTK is to import resources using GetNamedResource. GetNamedResource returns a binary object made from the specified resource. If you already have a Macintosh application (or ResEdit TMPL) that will generate your data as resources you can use the resources directly if that's more convenient.

Soups

One of my experiments creates a soup containing an entry for each element of `gData`. Creating a soup in the user store does not qualify as method for including data in your package. However, it is interesting to see how soups compare to the other data structures.

Store Parts

Store parts allow you to include a store as part of your package. The store part I made contains a soup with an entry for each element of `gData`. Soups in store parts do qualify as a method for including data in your package. However, NTK does not currently support the creation of store parts. You will not be able to perform this part of the experiment with your own data set. The store part was created with experimental tools.

Space Results

The space results were obtained by repeatedly compiling the SpaceWars project using different data structures. There is a constant defined in "Project Data" which can be adjusted to determine which data structure is used.

```
constant kDataStructure := 'ArrayOfFramesCommonMap;
```

The size of the `"SpaceWars.pkg"` file on the Macintosh is the uncompressed size of the package. The compressed sized was determined by using the `UsedSize` message on the store before and after loading the package. The `UsedSize` message returns an approximate value. Don't be alarmed if you find discrepancies in compressed size in your own experiments – even across different downloads of the same package.

The soup experiment does not involve data that is part of the package so there is no entry in the uncompressed column. If you build using one of the array-of-frames options and open the SpaceWars application in the extras drawer you'll see a "Make Soup" button. Pressing this button creates a soup and adds an entry for each element of `gData`. The value in the table for the soup corresponds to the difference in the store size before and after the soup is created.

The store part experiment simply uses a pre-built package containing a store part with a soup in. The values in the table correspond to the uncompressed size of this package and the difference in `UsedSized` before and after loading the package.

Data Structure	Package Size (bytes)	
	uncompressed	compressed
array of frames (naive)	35538	11884
array of frames (map sharing)	25970	8932
array of arrays	25866	8860
frame of frames	22386	9176
binary object	9962	5328
soup	N/A	37260
store part	35012	14900

Table 1 - Results of Space Experiment

We can see that binary-object wins for the smallest size. Some of you might have noticed that we represent our string data in the binary object as plain ASCII rather than unicode. Even so, this can only account for about 2.5K (= 300 entries x 8 bytes/entry) – and even less when compressed. There is no question that encoding your data into a binary object is the most compact representation.

The results also show that map sharing makes a substantial difference in data size. This difference becomes even more pronounced as the number of slots in your frames increases.

Another observation worth making is that using arrays instead of frames to represent the elements of `gData` did not yield much of a space savings – as long as we took advantage of map sharing.

You might also have noticed that the soup in the store part compresses better than the regular soup. This is because store parts are read-only and are compressed just like any other part. Regular soups, on the other hand, are compressed on an entry-by-entry basis using a different compression scheme.

Time Results

The benchmark used for timing data access was retrieving each of the elements of the data set using the `slot2` value as the key.

Below is the basic benchmark code used for all the non-soup data structures (minus some error checking code).

```
func()
  begin
    local iTicks := Ticks();
    local key;
    foreach key in kBenchmarkKeys do
      begin
        <*** data structure dependent line ***>
        //error checking code omitted
      end;
    iTicks := Ticks() - iTicks;
    Print("ArrayOfFramesBenchmark:" && iTicks);
  end
```

The magic lines of code (`<*** data structure dependent line ***>`) for each of the data structures are shown below.

```
local result := call kBSearchArrayOfFramesFn with (kArrayOfFramesData,key);
local result := call kBSearchArrayOfArraysFn with (kArrayOfArraysData,key);
local result := kFrameOfFramesData.(Intern(key));
local result := call kBSearchBinaryObjFn with (kBinObjData,key);
```

The data is kept sorted by `slot2` allowing binary search to be used for the array-of-frames, array-of-arrays, and binary-object data structures. The code for the frame-of-frames binary search is shown below. It is a standard binary search algorithm.

```
func BSearchArrayOfFrames(array,key)
begin
  local lPos := 0;
  local rPos := Length(array) - 1;
  local mPos;

  while lPos <= rPos do
    begin
      mPos := (lPos + rPos) DIV 2;
      mVal := array[mPos].slot2;
      if mVal > key then
        rPos := mPos - 1;
      else
        lPos := mPos + 1;
    end;

  if rPos>=0 AND StrEqual(array[rPos].slot2,key) then rPos;
end;
```

The code for searching the array-of-array is very similar.

```
func BSearchArrayOfArrays(array,key)
begin
  local lPos := 0;
  local rPos := Length(array) - 1;
  local mPos;

  while lPos <= rPos do
    begin
      mPos := (lPos + rPos) DIV 2;
      mVal := array[mPos][1];
      if mVal > key then
        rPos := mPos - 1;
      else
        lPos := mPos + 1;
    end;

  if rPos>=0 AND StrEqual(array[rPos][1],key) then rPos;
end;
```

The code for searching the binary-object is shown below.

```
func BSearchBinaryObj(binObj,key)
begin
  local lPos := 0;
  local rPos := (Length(binObj) DIV kDatumSize) - 1;
  local mPos;

  while lPos <= rPos do
    begin
      mPos := (lPos + rPos) DIV 2;
      mVal := ExtractCString(binObj,mPos*kDatumSize + kSlot2Offset);
      if mVal > key then
        rPos := mPos - 1;
      else
        lPos := mPos + 1;
    end;

  if rPos>=0 AND StrEqual(ExtractCString(binObj,rPos*kDatumSize + kSlot2Offset),key)
then rPos*kDatumSize;
end;
```

The frame-of-frames uses NewtonScript's built-in slot lookup to retrieve values rather than an explicit binary search. The current implementation of NewtonScript looks up slots in large frames using binary search on slot-name hash values.

The soup benchmark code (minus some error checking) is shown below.

```
func(soup)
  begin
    local cursor := Query(soup,'{type: index, indexPath: slot2});
    local iTicks := Ticks();
    local key;
    foreach key in kBenchmarkKeys do
      begin
        local result := cursor:GotoKey(key).slot2;
        //error checking code omitted
      end;
    iTicks := Ticks() - iTicks;
    Print("SoupBenchmark:" && iTicks);
  end
);
```

Table 2 summarizes the results from the benchmarks.

Data Structure	Benchmark Time (ticks)
array of frames	755
array of arrays	725
frame of frames 1st trial subsequent trials	 97 60
binary object	1001
soup	420
store part	343

Table 2 - Results of Time Experiment

We can see that frame-of-frames wins for the fastest lookup. Repeated lookups in the frame-of-frames get faster because Newton caches frame maps and because the the slot name symbols have already been created. `Intern` first checks for an existing symbol to return before creating a new one.

Compression did not make any appreciable difference in the timing results. This is probably because the packages are small enough that they can fit entirely in working memory (where they're paged-in and uncompressed). This is a big hint for those of you running your own experiments. Run your tests on a representative quantity of data – not just representative values.

Discussion of Results

Array of Frames

Clearly, you should be sure to take advantage of map sharing. The difference it can make in your package size can be substantial.

An array-of-frames is a basic general purpose data structure. If speed of access is a concern you should consider the frame-of-frames as an alternative.

Array of Arrays

The array-of-arrays has no real advantage over the array-of-frames. It has the disadvantage of being a less legible data structure. Unless your data is more naturally represented as an array, I don't see any reason to prefer an array-of-array over an array-of-frames.

Frame of Frames

A frame-of-frames is a significantly faster alternative to an array-of-frames. The only disadvantage is that if your key values contain special characters you will have to deal with the complexity of encoding them in 7-bit ASCII – not too big a disadvantage.

Binary Objects

If space is of paramount importance, then you should definitely consider using binary-objects. They are definitely the most complex and least legible data structure we considered in this article. Although, writing a library of setter and getter methods should help with this problem.

Extracting your data from binary-objects entails overhead, especially in the case of strings. This is reflected in the slower lookup times. Remember that extracting strings from a binary-object requires allocating a new object in the NewtonScript heap. If you encode arrays or frames within your binary objects, these too will require creating objects in the NewtonScript heap when they're extracted. Contrast this with the other data structures we've discussed. They allow you to reference their sub-parts directly (but, remember, they're still read-only).

Note that the in ROM versions of some of the ExtractXXX and StuffXXX functions have problems. Especially, StuffPString, which will trash the NewtonScript heap. See the PIE DTS Q&A documents for more information.

Soups

Regular soups are not an option for including data in your package. If you were considering providing data via soups (by writing code to build the soups) you can see that they're not the most space-efficient data structure to use.

On the other hand, soups provide great flexibility. They can be modified. They can have multiple indexes. Indexes can be added and removed dynamically. Data access is faster than any of the other data structures except the frame-of-frames.

Store Parts

Pretty much the same things can be said about store parts as for regular soups. Their primary advantage over soups is they offer better compression, but they're still not as space efficient as the other data structures we considered.

However, until NTK supports the creation of store parts they're not an option.

Conclusions

As you can see, there are a wide variety of trade-offs you can make when deciding how to represent your data. I'm sure the speed-freaks among you already have your sites set on a frames-of-frames and the bit-heads are already laying out binary-objects. However, I advise doing realistic experiments with the actual data data before finally choosing a data structure. I also advise designing your project so it's easy to switch among different data structures at any stage of development.