**Packaging and streaming assets to decrease the memory footprint in video games**

Jarryd Hall, Taher Odeh

**ABSTRACT**

Memory related issues such as memory management, limited memory and memory related application crashes, etc. often affect video games. Private game development companies have created the previous solutions to these kinds of issues and there are no open source solutions available for developers to utilize. This paper presents a constructive research approach to solving these memory issues. Operating systems were studied in order to properly understand these memory related issues and determine a viable solution. The solution needed to be implemented in such a way that multiple platforms could be targeted, although for the scope of this paper the focus was on iOS and Mac OSX. The solution

**General Terms**

video games, memory management, virtual memory, cross-platform, game performance, optimization

**Keywords**

virtual memory, mmap, Packaging tool, API, compression, caching, RAM, HDD, memory enhancement, file-system

**1. INTRODUCTION**

Almost all video games are bound by memory constraints. The system running the game can only allocate a certain amount of usable memory for the game’s process and this is usually less than advanced games need. The system’s kernel running the game will allocate memory for the game on RAM as well as on the HDD, which is where the virtual memory is located, and often the assets and game resources are very large, consuming a lot of the available memory.  
  
These memory constraints effects game developers and designers working within the development process of a project.

There are techniques used to solve this memory issue, but they are privately owned and with little documentation surrounding how the memory problems are solved. This study set out to solve the issue and implement an open source solution so that indie game developers who are not aided by proprietary techniques can create better games. Implementing a solution for this problem and licensing it as open source will allow anyone to be able to increase game performance as well as improve the overall game design and development.  
  
The problem above was pointed out during an initial interview with Johan Knutzen, the founder of Senri and Phobic-Games, mobile application and game development companies based in Gothenburg, Sweden, which are strongly involved in this research paper and it’s requirements and solutions.

The solution that was developed included a number of elements. The packaging tool compresses all the assets for a game into a single pak file. The pak file format is a file format that contains archived data, which is either compressed or decompressed. The API allows developers to use the generated pak file, as well as handles the memory optimization.

In order to test the developed solution, a sample game was run numerous times with and without the solution present. The game was profiled and monitored for memory and performance variables and the data was compared.

The question researched in this paper was:

* How can the memory footprint of video games be decreased?

**Constructive Research Method**

[What is a constructive research method]

The method chosen to be followed in this research is the Constructive Research Method, [ref1] describes this method as a way to turn existing knowledge into novelty or innovation by implementing a solution, to an existing problem, whether it is theoretical or practical, as long as it involves the usage of preexisting knowledge and thinking to produce artifact design solutions, for example, plans, diagrams, charts or software implementation; it may be considered constructive.

In addition to the description above, Kasanen et al. (1991) summarizes this method as a solution oriented method where innovation step-by-step solutions are taken in account, followed by testing of the solution and using the data within the testing phrase for analysis purposes.

[ref3] provides 3 category examples of knowledge gaps to be filled using the constructive research method: feasibility, where a solution to a common problem has not been done yet; Novelty, where a unique and new solution is provided to an already solved problem; or an improvement, where the goal of the research focuses on a preexisting solution and aims to produce better results than the ones available.

[why constructive]

The method mentioned has been found the most suitable for this research, basing on the fact that the latter aims to cover a feasible constructivism in terms of finance and freedom by having the solution free of charge and also freely licensed. Additionally, [ref4] argues that the constructive research method befits the computer science and the IT related problems in a usual manner.

The constructive research methodology allows for qualitative and quantitative approaches to be mixed or used as needed according to the kind of needed to be gathered during the development process of the solution.

Steps ….

“According to Kasanen et al. (1993), Lukka (2000), Labro and Tuomela (2003), there are seven crucial steps in the constructive research approach:

* (1)  to find a practically relevant problem, which also has research potential;
* (2)  to examine the potential for long-term research co-operation with the target organisation;
* (3)  to obtain a general and comprehensive understanding of the topic;
* (4)  to innovate and construct a theoretically grounded solution idea;
* (5)  to implement the solution and test whether it works in practice;
* (6)  to examine the scope of the solution’s applicability; and
* (7)  to show the theoretical connections and the research contribution of the solution. “

NOTE: This needs to be referenced properly from the original source.

[framework]

Qualitative methods were used to gather information that was needed to construct the solution. Open discussions were had with a stakeholder on an as-needed basis. These discussions were more suited than interviews as no time was wasted on establishing the focus of them. Discussing ideas and possible solutions to the current tasks at hand, allowed for a more streamlined iterative development process. These discussions also saved valuable development time as analysis was not done on interview or survey gathered data. The discussions dynamically led to focused, implementable solutions for the task at hand.

The process used can be shown with the following diagram:

ADD DIAGRAM HERE>

**Bundle**

The solution developed was planned from the start to be an open source project (add reference to github here), releasing the product under a Bsd License or similar (add reference to Bsd License here). The reason for this decision was the fact that memory management is something every game developer has to deal with and offering an open source solution could not only help other developers but possibly gain interest of the community and evolve the project. The focus of Bundle was on the iOS and Mac OSX platforms, although the code itself is C, which allows multiplatform support.

The first milestone in the construction of such a solution was to understand how the current hardware and software operating systems could allow for such a solution. As the issue is usually experienced with manually allocated memory in RAM, researching how RAM works on a low level with an operating system was necessary. Virtual Memory resides on disk and acts similar to a swap file.(add swap file reference here). A MMU(add MMU reference here) is used by the CPU to translate virtual memory addresses.

As Apple’s iOS and OSX platforms where the focus of this research, the memory management was researched on Apple’s self- published documentation.

The kernel manages RAM and Virtual Memory segments. When an application launches the kernel allocates a block of memory in RAM, e.g. around 40MB on the iPhone 4 and also assigns a virtual memory data segment, e.g. around 700MB on the iPhone 4. (add profiling diagrams? Or some reference). The RAM and Virtual Memory work together by means of paging data segments or pages, in and out, between each other. When a data segment is needed that resides in virtual memory, the kernel pages the page that holds that data into RAM. (add virtual memory reference to apple’s site here) . Data is paged into RAM from virtual memory until the calculated max RAM size is reached. Once this occurs the kernel will page out data that is not currently needed, and replace it with the page from virtual memory that holds the currently needed data.

Virtual memory became the focus at this point to be the basis of the solution. Data allocated to virtual memory rather than strictly allocated to RAM using objective C’s alloc method would allow larger segments of data to be allocated due to the size of the virtual memories data block, e.g. around 700MB on iPhone 4, compared to the iPhone 4’s 40MB RAM.

A POSIX function called mmap (reference mmap here) is a low-level virtual memory mapping technique that allows one to allocate a file on disk into virtual memory addresses.

As mmap could map a file to virtual memory for use within a running process, creating a file for a game’s assets became the focus at this point.

**Tool and Pak File Creation**

The pak file format is an archive file format that is not standardized on its contents. Pak files are usually used for games, where game assets are archived into a single file. (add reference to pak file here) (possibly add info about other games that use this as well as battlefield that uses it for every level, having multiple pak files)

A tool was then developed to archive these game assets into a single pak file. This was created as a command line tool that takes two arguments, the source folder and the output file. The tool takes the source folder, traverses the entire directory to count the files, ignoring DS\_Store files that are present on Apple’s platforms(reference DS\_Store). Once the file count is obtained, the value is used to calculate the variable header length.

Add header information here.

Each asset within the source folder, e.g. sound files, textures, images etc. is compressed using zlib(reference zlib here) and indexed within the pak file, with its index information stored in the header segment. (reference file format creation link)

The tool outputs a pak file at the user specified destination path. Once this pak file is created it is ready to be used within the application.

**The API**

The base API was developed in C allowing for future evolution of the product, which is especially needed for an open source project where many people might have ideas to evolve the product.

For the focus of this research paper, the operating systems being Apple’s iOS and Mac OS X, wrapper methods were needed to support Apple’s Objective-C language, which forms the basis of the Cocoa and Cocoa Touch frameworks.(add reference here to objc and cocoa/touch) Objective-C is a superset of C, allowing it to integrate seamlessly.

The wrapper function was only needed to pass the needed data’s pointer, the pointer to the data in virtual memory, to the object needing it within an objective-C runtime environment.

The Objective-C NSData class has a method for passing a pointer to an address as well as the length of bytes to process, with the alternative option of freeing the data when done or not. (add reference here to the NSData class)

+ (id)dataWithBytesNoCopy:(void \*)bytes length:(NSUInteger)length freeWhenDone:(BOOL)freeWhenDone

This allows an object to be created using the virtual memory pointer and size of bytes of the segment, without the needed to allocate memory in RAM for the object. This perfectly integrated with the developed base C API that allowed retrieval of a file’s index data within the pak file, using the filename itself. (add a possible function call and small explanation of the process)

This feature would make it as simple as possible for developers to use the API as they can use their filenames as usual without needed an understanding of how the file’s data is passed to the called NSData method.

Usability was always an important attribute of the solution as it is an open source project and will hopefully be used my many developers.

The use of the API was minimized to 3 important functions:

* int bundle\_start(char \*pakFile, struct mappedData \*mData)
* offset\_p bundle\_getIndexDataFor(char \*fileName)
* int bundle\_stop(struct mappedData \*mData)

The first starts Bundle by hashing and memory mapping a given pak file. The second function is the one that is always used directly or from within a wrapper method to retrieve the needed data pointer and size, that, if called from a wrapper can be passed to the native languages method. The third function stops Bundle, by destroying the hashmap and unmapping the file from virtual memory. Allowing such ease of use of the Bundle system developers do not need to lose valuable development time, as another important attribute of Bundle is to improve the development process in various ways. Developers can integrate Bundle quickly into existing games.

Add information here about the crossroads we came to with the different options regarding compression and decompression, or will that come in the discussion?

Once the tool was packaging assets correctly into a pak file and the API could correctly place the pak file’s header info into a hashmap and memory map the file to virtual memory, the focus was on reading this data and making it available to the caller. An issue arose regarding what to do with the needed compressed data. The issue was that if a number of compressed files are within the pak file, and mapped to virtual memory, when using this data for objects, the data should be used directly from virtual memory, rather than copying the data temporarily to the stack, whether it was decompressed or not. A number of options were considered.

1. Package the game assets into the pak file without compression.
2. Package the compressed assets into the pak file, and decompress the entire file to virtual memory, re indexing an internal structure.
3. Package the compressed assets into the pak file, and decompress data as needed.
4. Package the compressed assets into the pak file, and decompress and memory map individual files on demand.

All of the above had side effects for the solution such as loading times, decreased memory size, memory fragmentation which would lead to no memory being available of a large enough size for a needed block, even if the total free memory was more than enough. The main issue with the mentioned options was the fact that they all forced a temporary duplicate of the virtual memory mapped data to be placed somewhere else in memory, either the stack or the heap. Keeping with the focus of reducing strictly allocated RAM usage to a minimum, the stack would be the option. If a game had n objects all with their source data memory mapped in virtual memory and a duplicate of the decompressed data in memory, it would not be a solution to a problem as much as it is reading from a file and using the same amount or memory.

This led to a solution where filetypes needed to be defined as to how they are used. Defining which file types needed to be decompressed or not during runtime would allow the most dynamic and efficient way to handle the data for use by objects.

The game assets would be packaged as compressed files into the pak file and when the bundle\_start function is called, the assets filetypes would need to be analyzed and depending on the type, either decompressed into virtual memory or left compressed.

The reason for this is that certain functions or methods that use this data work more efficiently with compressed data; others work more efficiently with uncompressed data.

Give some examples of the file types?

**References:**

[ref1 = **Constructivist Research and Info-Computational Knowledge Generation** Gordana DODIG CRNKOVIC]

[ref2 = Kasanen et al. (1991)]

[ref3] Anna-Liisa Lindholm

[ref4]= Information systems research methodologies and models(Albertas Caplinskas, Olegas Vasilecas)