DESARROLLO DE COMPONENTES PARA MOTOR DE VIDEOJUEGOS

Javier Moya Nájera

Desarrollo de Aplicaciones Multiplataforma

Memoria del proyecto de DAM

IES Abastos, curso 2012-13. Grupo 7U. 18 de Junio de 2013

Tutora individual: Cristina Ausina Víctor

ÍNDICE

1. INTRODUCCIÓN Y JUSTIFICACIÓN

1.1. Contextualización.

La empresa para la cual se ha desarrollado este proyecto es **Arcatium Studios S.L.**, cuyo nombre comercial es **Ninja Fever** (http://www.ninjafever.com). Se trata de una empresa pequeña, que consta de tres programadores (los tres socios fundadores de la empresa) y dos grafistas, y cuya actividad principal es el desarrollo de videojuegos, tanto propios como para terceras personas, para diversas plataformas (PC, iOS, consolas...).

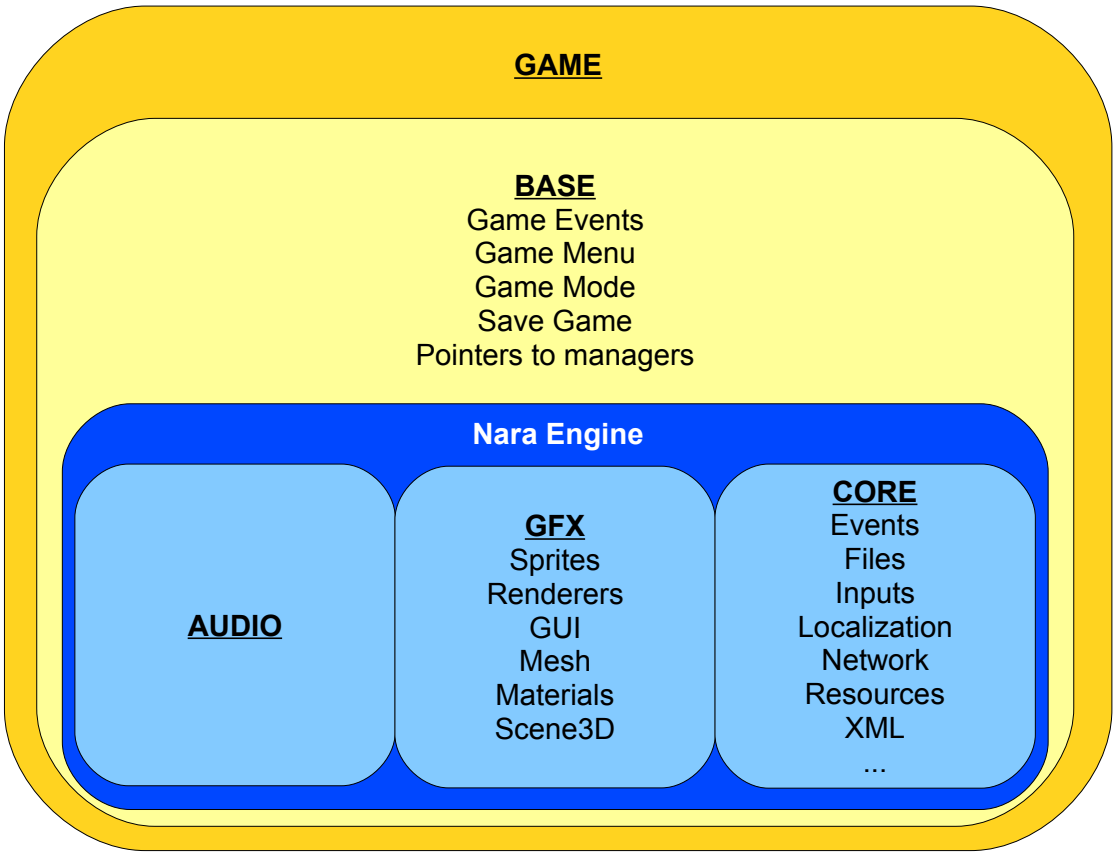
La empresa utiliza un motor de videojuegos llamado **Nara Engine**, desarrollado por ellos mismos, y que contiene todas las herramientas y características necesarias para el desarrollo de su actividad empresarial. Utiliza el lenguaje de programación C++, y tiene soporte para crear videojuegos para plataformas tan dispares como PC, iOS, Android, PlayStation Portable o Wii, todo ello bajo el entorno de desarrollo integrado (IDE) *Visual Studio*. Algunos ejemplos de juegos creados con este motor son: ***Carnivalz*** para iOS, o ***Dual Zone*** para Xbox live, cuyas descripciones e imágenes pueden verse en su página web.

La **estructura básica de *Nara Engine*** es la siguiente:

- ***Audio***: Como su nombre indica, es la parte del motor que se encarga de todo lo relacionado con el audio: música, efectos de sonido... Utiliza hilos para que la reproducción del audio no se vea interferida por el procesamiento de otras partes del motor, y cuenta con un *manager* específico para cada una de las plataformas soportadas, para adaptarse a las características propias de cada una de ellas.
- ***Gfx***: Es la parte del motor que se encarga de todo el procesamiento de gráficos. Así, cuenta con *managers* para sprites, renderizadores (OpenGL, Direct3D...), elementos de la GUI (*Graphical User Interface,* Interfaz Gráfica de Usuario), mallas, texturas y escenas 3D, cada uno de ellos con su versión correspondiente para cada plataforma.
- ***Core***: Es la parte del motor que se encarga de todo aquello de lo que no se encargan los dos módulos anteriores; esto es: eventos, archivos, *inputs*, localización, redes, recursos, manejo de archivos XML...

Sobre el núcleo formado por estos tres módulos, el motor tiene definida una capa, ***Base***, con las cosas básicas que se utilizan para la creación de videojuegos: eventos de juego, menú de juego, modo de juego, *savegame*, y una estructura que contiene una lista de punteros a los principales *managers* de *Nara Engine*.

Finalmente, **sobre esta capa se define finalmente el juego en sí**, creando clases que hereden directamente de las clases de *Base* para que todas conserven el mismo funcionamiento básico, y ampliándolas para adaptarlas a las necesidades del juego.



1.2. Identificación, justificación y objetivos del proyecto.

En el momento del inicio del proyecto y de las prácticas, la empresa estaba portando uno de sus últimos juegos, ***Arson & Plunder*** (http://www.arsonandplunder.com), desde la plataforma iOS a PC. Se trata de una aventura 2D arcade / beat'em up basada en los géneros míticos de los años 90, donde controlas a un orco o a un elfo mientras te deshaces de las hordas de enemigos que intentan acabar contigo, todo ello reforzado por una hilarante historia llena de humor y sarcasmo.

En su versión para iOS, todos los recursos utilizados por el juego (imágenes, música, scripts...) quedan empaquetados por el propio sistema utilizado por la plataforma, de manera que el usuario no puede acceder directamente a ellos. En cambio, en su versión para PC, todos estos archivos quedaban a la vista de los usuarios, siendo fácilmente editables. Por esta razón, se hacía necesario realizar por una parte un empaquetador que cogiese todos los recursos del juego y los empaquetase en un único archivo para evitar su modificación por parte de los futuros usuarios y, por otra parte, crear e integrar en el motor de desarrollo un desempaquetador capaz de obtener los datos empaquetados para su uso.

Por otra parte, la empresa también estaba trabajando en la traducción al idoma chino de otro videojuego diseñado por otra empresa, pero programado por ellos, ***Push Cars 2*** (http://www.push-cars.com). Se trata de la segunda parte de un juego para iOS y Android en el que el jugador tiene que utilizar la lógica y el ingenio para superar los niveles propuestos. El funcionamiento básico del juego es hacer que los coches ecológicos salgan de la pantalla, que representa la ciudad, sin chocar con nadie, mientras que los coches contaminantes deben chocar y permanecer en las calles de la ciudad.

En este proyecto tenían la problemática de que su motor, *Nara Engine*, no aceptaba los caracteres con codificación UTF-8 necesarios para soportar dicho idioma, por lo que se hacía necesaria una modificación del motor para trabajar con textos, independientemente de si éstos utilizaban una codificación ASCII, UTF-8 o incluso ambos.

Así pues, tras ver estas problemáticas, se me encomienda el trabajo de realizar los cambios y modificaciones pertinentes en el motor de videojuegos *Nara Engine* para solventar los problemas arriba descritos, pasando a constituir este encargo el motivo de mi proyecto.

2. PLANIFICACIÓN Y DISEÑO DEL PROYECTO

2.1. Empaquetador y desempaquetador de archivos.

Para la problemática de *Arson & Plunder,* lo primero que se hizo fue un **análisis exhaustivo del funcionamiento actual del motor**, para entender cómo se habían estado realizando hasta ahora las cargas de los archivos. De este análisis se dedujo, por una parte, la estructura de los recursos de los proyectos, de la cual hablaremos a continuación, y por otra parte que, hasta entonces, se accedía a los archivos usando una clase del motor llamada *CFile*, que se encargaba de abrir el archivo en su formato correspondiente (entrada, salida, binaria...). En este sentido **cabe destacar** que en cada acceso se debía calcular previamente la ruta completa del archivo a acceder.

La estructura de los recursos de los proyectos consiste básicamente en una carpeta llamada *Data* en cuyo interior podemos encontrar dos carpetas al menos: una llamada *Common*, que contiene en su interior los recursos comunes a todas las plataformas, y otra con el nombre de la plataforma de destino (por ejemplo, *Win* para Windows), que contiene los recursos propios de la plataforma de destino.

Una vez realizado este análisis, se definieron los siguientes **pasos a seguir para desarrollar una solución**:

1. Creación del empaquetador como un programa independiente, de manera que pueda utilizarse mediante scripts y programas externos.

2. Creación de una clase desempaquetadora independiente del motor, que será la encargada de buscar y recuperar los datos de los archivos empaquetados por el empaquetador.

3. Integración de la clase desempaquetadora en el motor.

4. Realización de pruebas sobre proyectos pequeños.

5. Realización de pruebas sobre proyectos de grandes dimensiones, especialmente en *Arson & Plunder*, y detección y corrección de errores.

Finalmente, para llevar a cabo este desarrollo, **se tuvieron presentes las siguientes premisas**, consensuadas con los socios de la empresa:

- El **empaquetador** aceptaría, como mínimo, dos parámetros de entrada: uno que indicase la carpeta de la cual se deberían tomar los archivos para empaquetar (-i <carpeta de entrada>), y otro que indicase dónde se crearía el paquete (-o <carpeta de salida>). Además, el empaquetador podría indicar un tercer parámetro opcional, que sería la ruta a un documento de texto en cuyo interior se indicase una lista de archivos que no debían incluirse en el paquete (-e <lista de excluidos>). Por último, si se especificaba una lista de excluidos, se podía indicar un último parámetro (-r) para que hiciese una copia de los archivos excluidos en la carpeta de salida, ya que estos archivos excluidos serían en su mayoría archivos de configuración necesarios por el juego y, de este modo, se podía juntar en una sola carpeta todos los archivos necesarios para el correcto funcionamiento del juego sin necesidad de copiar archivos no empaquetados manualmente.

  Así pues, la sintaxis aceptada por el empaquetador sería la siguiente:

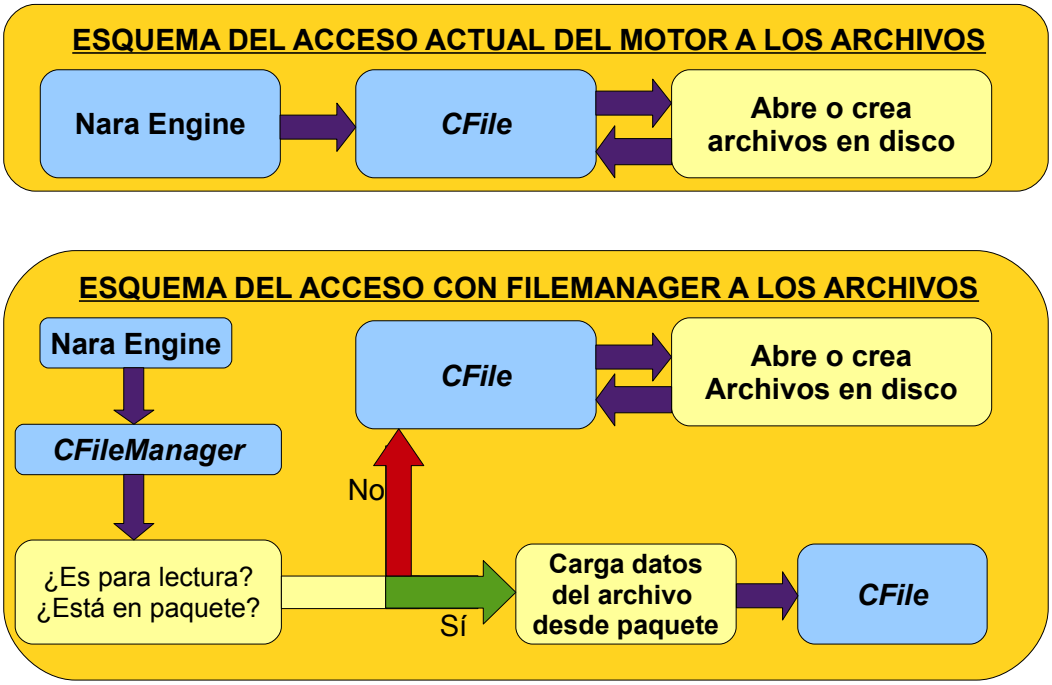
  Empaquetador.exe -i <carpeta de entrada> -o <carpeta de salida> [-e <archivo con la lista de archivos y carpetas excluidas> [-r]]

- El empaquetador debería guardar en la cabecera del paquete, los siguientes datos de cada archivo: tamaño, posición de inicio, longitud del nombre de archivo, nombre de archivo, y hash calculado a partir del nombre de archivo. Además, estos datos deberían guardarse ordenados por hash para facilitar las búsquedas desde el desempaquetador.

- El **desempaquetador** debería tener, al menos, una función para obtener un archivo (*GetFile*) y una función para comprobar la existencia de un archivo (*FileExists*), y ambas funciones deberían ser válidas tanto para archivos contenidos en el paquete como para archivos existentes en el disco duro, comprobando primero si existen en el apartado *Plataforma* del paquete, luego en el apartado *Plataforma* del disco duro, luego en el apartado *Common* del paquete, y finalmente en el apartado *Common* del disco duro.

- Las búsquedas de los archivos empaquetados se realizarán por comparación de hash, y se utilizará la técnica de 'Divide y vencerás'.

- Una vez integrado el desempaquetador en el motor, éste debería encargarse de manejar absolutamente todos los accesos a ficheros, deprecando el uso directo de la clase que utiliza el motor actualmente para acceder a los archivos (*CFile*) en pos del uso de la clase del desempaquetador. Por esta razón, se decide que el nombre de la clase desempaquetadora sea ***CFileManager***.



2.2. Soporte para UTF-8.

En el caso de ***Push Cars 2***, además de realizar un análisis de la parte del motor correspondiente a la carga de textos, se realizó un estudio previo del funcionamiento de la codificación UTF-8, que resumiremos a continuación.

UTF-8 (*8 bit Unicode Transformation Format*) es un formato de codificación de caracteres Unicode que utiliza símbolos de longitud variable, con un máximo de 4 bytes por carácter. Además, incluye la implementación del código ASCII, por lo que es completamente compatible con el mismo.

Para su decodificación, basta con tener presentes las siguientes pautas:

- Si el primer bit es cero, se trata de un carácter ASCII de un solo byte y no requiere de ningún tratamiento. Por ejemplo: **0**0100100 se corresponde con el carácter '$'.

- Si el primer bit es uno, se trata de un carácter UTF-8. En este caso, el primer byte contendrá tantos unos como bytes tenga dicho carácter, y estos unos irán seguidos de un cero que actuará como marcador de final de cabecera, de manera que el resto de bits se corresponden con el cuerpo del carácter (por ejemplo, el primer byte de un carácter de 3 bytes sería **1110**+cuerpo). El resto de bytes tendrán la estructura **10**+cuerpo, para seguir indicando en su cabecera que se trata de un carácter UTF-8.

Así pues, para decodificar un carácter UTF-8, lo primero que debemos hacer es saber de cuántos bytes consta el carácter, e ir eliminando las cabeceras de los bytes correspondientes. Por ejemplo, para el carácter '¢', el código binario de sus bytes es: **110**00010 **10**100010. Si aplicamos las pautas expuestas, el 110 de su primer byte nos indica que es un carácter de dos bytes, y tras eliminar las correspondientes cabeceras tenemos que el código real del carácter es **00000**000 10100010 (los ceros marcados a la izquierda se han añadido para completar el byte).

Una vez estudiado el funcionamiento de la codificación UTF-8, se definieron los siguientes **pasos para proceder a su implementación en el motor**:

1. Creación de unas funciones básicas para el manejo de cadenas UTF-8, con compatibilidad para cadenas ASCII.

2. Modificación del sistema de carga de texto para que recurra a las funciones creadas en vez de recurrir a las funciones estándar.

3. Realización de pruebas sobre proyectos de la empresa, y detección y corrección de errores.

Finalmente, para llevar a cabo este desarrollo, **se tuvieron presentes las siguientes premisas**, consensuadas con los socios de la empresa:

- Las **funciones básicas de manejo de cadenas UTF-8** serían:

  - *unsigned int GetStringSizeUTF8 (const char* pszString)*: dado que los caracteres UTF-8 ocupan más de un byte, las cadenas estándar no sirven para saber la longitud real de una cadena con caracteres UTF-8, ya que éstas devuelven la cantidad de bytes de una cadena. Así, se hace imprescindible tener una función para saber la longitud real de una cadena UTF-8.

  - *unsigned int GetUTF8Position (const char* pszString, unsigned int uiPosition)*: esta función devuelve la verdadera posición de *uiPosition* dentro de una cadena UTF-8.

  - *unsigned int GetCodeFromUTF8Char (unsigned int uiChar)*: en esta función, *uiChar* es la composición de los bytes de un carácter UTF-8 sin decodificar. Por tanto, lo que hace es decodificar el carácter UTF-8 y devolver su código correspondiente.

  - *unsigned int GetCodeFromUTF8String (const char* pszString, unsigned int uiPosition)*: esta función devolverá el código del carácter de *pszString* indicado por *uiPosition* ya decodificado, y hará uso de todas las funciones anteriores.

- Se **sustituirían en el motor todas las funciones estándar** por las creadas para la ocasión, sin que ello conlleve una variación del funcionamiento del motor.

2.3. Normativa aplicable al proyecto.

La única normativa aplicable al proyecto fue la aplicación de la nomenclatura utilizada por la empresa para la denominación de las variables y clases, que podemos resumir de la siguiente manera:

- En todos los casos se utilizará la nomenclatura ***UpperCamelCase***.
- El nombre de las **clases** irá precedido por 'C' (**C**Clase MiClase), y sus variables miembro irán precedidas por 'm_' (**m_**VariableDeLaClase).
- El nombre de las **estructuras** irá precedido por 'T' (**T**Estructura MiEstructura).
- El nombre de los **enumerados** por 'E', y las **variables de tipo enumerado** irán precedidas por 'e' (**E**Enumerador **e**MiEnumerador).
- El nombre de las **variables globales** irá precedido por 'g_' (**g_**MiVariable).
- El **nombre de las variables, en general**, irán precedidos por un identificador de su tipo de dato, siempre que el tipo sea uno de los siguientes:
  - Puntero: p (void* **p**Variable).
  - Booleano: b (bool **b**Variable).
  - Entero: i (int **i**Variable).
  - Carácter: c (char **c**Variable).
  - Cadena terminada en cero: sz (string **sz**Variable).
  - Flotante: f (float **f**Variable).
  - Flotante de doble precisión: d (double **d**Variable).
  - Array: a (int **ai**Variable[10]).

3. DESARROLLO DEL PROYECTO

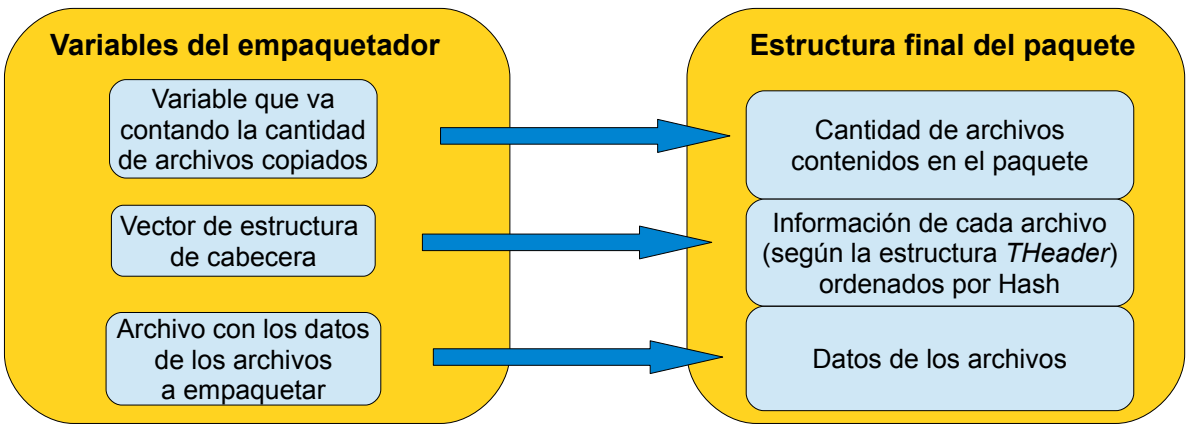
3.1. Empaquetador y desempaquetador de archivos.

3.1.1. Creación del empaquetador como un programa independiente.

Para la creación del empaquetador, lo primero que se hizo fue crear una **estructura que contendría los datos de cada archivo dentro de la cabecera del paquete**, y que se compartiría con la clase desempaquetadora. Así, partiendo de las premisas dadas por la empresa, la estructura quedó:

```
typedef struct
{
        int             m_iInit;        // Punto de inicio dentro del fichero
        int             m_iSize;        // Tamaño del fichero
        unsigned int    m_uiHash;       // Hash
        int             m_iNameLen;     // Longitud de nombre de archivo
        char *          m_pszName;      // Nombre de archivo
}
THeader;
```

Una vez creada la estructura, se pasó a implementar el **funcionamiento básico del empaquetador**, que consiste en recorrer todas las carpetas y subcarpetas del directorio especificado como 'de entrada' (-i) e ir obteniendo los datos de los archivos para ir guardándolos uno detrás de otro en un archivo de salida temporal por un lado, y por otro lado, ir guardando la información de la cabecera en un vector. Así, una vez recorridos y copiados todos los archivos, se crea el archivo 'paquete' (-o) concatenando la cantidad de archivos empaquetados con la información del vector 'cabecera' (ordenado previamente por Hash) y con la información del archivo de datos.



Para la realización de este funcionamiento básico se creó la **clase *CPacker*** con las siguientes **funciones**:

- **CreateFolder (carpeta a crear)**: Esta función crea una carpeta en el disco duro, si ésta no existía con anterioridad.

- **Init (carpeta de entrada, carpeta de salida):** esta función se encarga de inicializar la clase empaquetadora para que pueda realizar su cometido. Recibe como parámetros de entrada la carpeta de entrada y la carpeta de salida recibidas por el propio programa en su línea de argumentos, es decir, la carpeta que contiene los datos a empaquetar y la carpeta donde se desea crear el paquete. Sigue el siguiente pseudocódigo básico:

  Copia la ruta de carpeta de entrada y salida a variables de la clase.
  CreateFolder *(carpeta de salida)*
  Inicializa el vector de cabeceras (THeader).
  Abre archivo de salida (datos).

- **Copy (Archivo de entrada, archivo de salida)**: Esta función recibe como parámetros dos punteros a archivo: el primero al archivo que se desea copiar, y el segundo al archivo de datos. Su función es volcar todos los datos del archivo de entrada en el archivo de salida y devolver el tamaño del archivo copiado, siguiendo para ello este pseudocódigo:

  tamanyo = tamaño del archivo de entrada
  Si tamanyo no es -1
  Crea buffer [tamanyo]
  Copia los datos de archivo de entrada a buffer
  Escribe buffer en archivo de salida
  Elimina buffer creado
  Fin Si
  Devuelve tamanyo

- **CopyFiles (carpeta de entrada)**: esta función se encarga de recorrer una carpeta y sus subcarpetas y, en caso de encontrar un archivo, copiarlo usando la función anterior y guardar los datos necesarios en el vector de cabeceras. Se trata de una función recursiva cuya carpeta de entrada inicial es la carpeta de entrada indicada en los parámetros del programa, y sigue el siguiente pseudocódigo básico:

Handle = EncuentraPrimerArchivo (carpeta de entrada)
Hacer
Si Handle es una carpeta
***CopyFiles** (Handle)*
Si no
Archivo de entrada = Abre (Handle)
*tamanyo = **Copy** (Archivo de entrada, Archivo de salida datos)*
Cierra archivo de entrada
Crea y rellena cabecera (THeader), y añade al vector
CantidadArchivosCopiados ++
Fin Si
Mientras exista EncuentraSiguienteArchivo (Handle)

- **Start ()**: esta función es la que ejecuta la acción de empaquetado, y sigue el siguiente pseudocódigo básico:

***CopyFiles** (carpeta de entrada)*
Cierra archivo de salida (datos)
Abre archivo de salida del paquete
Guarda CantidadArchivosCopiados en paquete
Ordena por Hash y guarda vector de cabeceras en paquete
Copia archivo de datos en paquete
Cierra archivo de paquete
Borra archivo de datos
Muestra en pantalla el resultado del proceso

Cuando el empaquetador ya tuvo implementado el sistema básico, se pasó a añadirle la **opción de excluir archivos y carpetas** (-e) contenidos dentro de la carpeta de entrada. Para ello, se añadieron una serie de vectores, según el tipo de exclusión:

- Vector de **nombres de carpeta**: se comprueba sólo el nombre de la carpeta.
- Vector de **rutas de carpeta**: se comprueba la ruta completa, no sólo el nombre.
- Vector de **nombres de archivo**: se comprueba sólo el nombre del archivo.
- Vector de **rutas de archivo**: se comprueba la ruta completa, no sólo el nombre.
- Vector de **parte de nombre de archivo**: se comprueba el inicio del nombre.
- Vector de **ruta y parte de nombre de archivo**: se comprueba con ruta completa.
- Vector de **extensiones de archivo**: se comprueba la extensión del archivo.

También se añadieron y modificaron las siguientes **funciones**:

- **LoadExcludedFile (archivo de exclusión)**: Esta función se encarga de recorrer el archivo de exclusión y, según el delimitador indicado en el archivo, coloca las rutas y archivos en su correspondiente vector.

Abre archivo de exclusión
Mientras archivo de exclusión no llegue al fin (EOF)
Obtén línea del archivo
Si línea == "#delimitador_de_tipo_de_exclusión_X"
Modo = tipo_de_exclusión_X
[se comprueban los N delimitadores de forma anidada]
Si no y Si linea no está vacía
Si Modo == tipo_de_exclusión_X
Añade línea al vector de tipo_de_exclusión_X
[se comprueban los N tipos de exclusión de forma anidada]
Fin Si
Fin Si
Fin Mientras
Cierra archivo de exclusión

- **Init (… , archivo de exclusión)**: Se añade la ruta al archivo de exclusión, y se añade al inicio la inicialización de los vectores mediante la función anteriormente descrita.

LoadExcludedFile *(archivo de exclusión)*

- **IsExcluded (nombre, es carpeta)**: Esta función busca en los vectores de carpetas o de archivos el nombre indicado, y devuelve true o false según esté excluido o no. El orden seguido para su búsqueda es el siguiente:

Si es carpeta
Busca en vector de nombre de carpeta con ruta completa
Busca en vector nombre de carpeta
Si no
Busca en vector de extensión
Busca en vector de nombre de archivo con ruta completa
Busca en vector de nombre de archivo
Busca en vector de parte de nombre de archivo con ruta completa
Busca en vector de parte de nombre de archivo

- **CopyFiles (…)**: Se añade una comprobación del nombre de archivo y del nombre de carpeta para ver si se debe copiar o no.

```
…
Si Handle es una carpeta
        Si !IsExcluded (Handle, true)
                CopyFiles (Handle)
        Fin Si
Si no
        Si !IsExcluded (Handle, false)
                Copia archivo
        Fin Si
Fin Si
...
```

Por último, se pasó a añadir la opción de **recreación de los archivos excluidos en la carpeta de salida** (-r). Esto requirió de los siguientes cambios y añadidos:

- **CreateFolderTree (ruta)**: Esta función crea todo el árbol de carpetas indicado en ruta. Para ello, va analizando ruta y va creando un vector con las rutas completas de cada carpeta, para luego crear dichas carpetas:

```
Para i = primer ' \' en ruta, i < tamaño(ruta), i++)
        Si ruta[i] == \
                cadena = ruta(0, i)
                vector.Añade (cadena)
        Fin si
Fin Para
Para iterador = vector.Inicio(), iterador < vector.Fin(), iterador ++
        CreateFolder (iterador)
Fin Para
```

- **RecreateFile (archivo con ruta)**: Esta función copia un archivo a la carpeta de salida, creando si es necesario su árbol de carpetas.

```
Si ruta no existe en carpeta de salida
        CreateFolderTree (ruta)
Fin Si
Copia archivo a ruta de carpeta de salida
```

- **RecreateFolder (carpeta)**: Esta función copia todos los archivos de ruta a la carpeta de salida, creando si es necesario sus respectivos árboles de carpetas. Utiliza recursividad para recorrer el interior de las diversas carpetas.

```
Handle = EncuentraPrimerArchivo (carpeta)
Hacer
    Si Handle es una carpeta
        RecreateFolder (Handle)
    Si no
        RecreateFile (Handle)
    Fin Si
Mientras exista EncuentraSiguienteArchivo (Handle)
```

- **Init (… , hay que recrear)**: Se añade a su definición un booleano para indicar si hay que recrear los archivos excluidos, y se copia su valor a una variable miembro.

- **CopyFiles (…)**: Se añade a la comprobación de exclusión la opción de recrear el archivo o la carpeta.

```
…
Si Handle es una carpeta
    Si !IsExcluded (Handle, true)
        CopyFiles (Handle)
    Si no y Si hay que recrear
        RecreateFolder (Handle)
    Fin Si
Si no
    Si !IsExcluded (Handle, false)
        Copia archivo
    Si no y Si hay que recrear
        RecreateFile (Handle)
    Fin Si
Fin Si
...
```

Con esto, el empaquetador quedó listo para su funcionamiento mediante scripts y programas externos.

3.1.2. Creación de una clase desempaquetadora independiente del motor.

Una vez hecho el empaquetador, se creó en el mismo proyecto una nueva clase llamada **CUnpacker**, cuya función principal sería la de ser capaz de encontrar dentro del paquete los archivos empaquetados y recuperar su información, tanto de la cabecera (*THeader*) como de la sección de datos.

La cabecera debía cargarse completamente en memoria para agilizar el proceso de recuperación de archivos, así que se recurrió al uso de un vector de cabeceras (*THeader*), y se crearon las siguientes funciones:

- **LoadHeader ()**: Esta función recorre la cabecera del paquete y va añadiendo la información de cada archivo en un vector miembro de la clase.

  Para i=0, i<ArchivosEmpaquetados, i++
  Leemos información de cabecera de archivo
  Guardamos en el vector
  Fin Para

- **Init (ruta al paquete)**: Esta función recibe la ruta donde se encuentra el paquete, y se encarga de preparar la clase para su funcionamiento, abriendo el fichero del paquete y cargando la cabecera en memoria.

  Abre el archivo paquete
  ArchivosEmpaquetados = Lee cantidad de archivos
  ***LoadHeader** ()*

Con estas dos funciones, la clase ya estaba lista para recuperar datos, para lo cual se establecieron cuatro funciones: dos funciones de búsqueda, y dos funciones de recuperación de información.

- **Search (inicio, fin, hash a buscar)**: Esta función recursiva utiliza el algoritmo de 'divide y vencerás' para buscar un archivo por hash en el vector de cabeceras (recordemos que el vector está ordenado por hash y que este proceso se realiza durante el proceso de creación del paquete). Si encuentra el archivo dentro del vector, devuelve la posición en la que se ha encontrado y, si no, devuelve -1 para indicar que no se ha encontrado.

Devuelve = -1
Posición = (inicio + fin) / 2
Si vectorCabeceras[Posición].Hash == Hash a buscar
    Devuelve = Posición
Si no y Si (fin – inicio == 1) o (inicio – fin == 1)
    Si vectorCabeceras[inicio].Hash == Hash a buscar
        Devuelve = inicio
    Si no y Si vectorCabeceras[fin].Hash == Hash a buscar
        Devuelve = fin
    Fin Si
Si no y Si inicio != fin
    Si Hash a buscar < vectorCabeceras[posición].Hash
        Devuelve = Search (inicio, posición – 1, hash a buscar)
    Si no y Si Hash a buscar > vectorCabeceras[posición].Hash
        Devuelve = Search (posición + 1, fin, hash a buscar)
    Fin Si
Fin Si
return Devuelve

- **FileExists (nombre del archivo)**: Esta función recibe un nombre de archivo y debe comprobar que dicho archivo se encuentre en el paquete, devolviendo true o false según sea conveniente.

    Hash = Calcula hash de nombre de archivo
    *Si **Search** (0, vectorCabeceras.Tamaño - 1, Hash) > -1*
        return true
    return false

- **GetInfo (nombre del archivo)**: Esta función recibe un nombre de archivo y debe devolver la información de cabecera de dicho archivo.

    Hash = Calcula hash de nombre de archivo
    *Posición = **Search** (0, vectorCabeceras.Tamaño - 1, Hash)*
    Si Posición > -1
        Copia información de vectorCabeceras[posición] a un THeader
    Fin Si
    return THeader

- **GetFile (nombre del archivo)**: Esta función recibe un nombre de archivo y debe devolver un puntero a los datos de dicho archivo.

  THeader info = ***GetInfo*** *(nombre del archivo)*
  Reservamos memoria de 'info.m_iSize' tamaño
  Movemos el puntero del archivo del paquete hasta el inicio de los datos
  Copiamos los datos del archivo en el Buffer
  Devolvemos el puntero al buffer creado

Con ésto, el funcionamiento básico de búsqueda y recuperación de archivos desde paquete quedó finalizado.

3.1.3. Adaptación e integración de la clase desempaquetadora en el motor. Creación de *CFileManager.*

Se podría afirmar que este punto fue el más delicado y costoso de todos. Hasta ahora, se había trabajado independientemente del motor de videojuegos pero, **en este punto, se debía trabajar directamente sobre él, incluyendo nuevas funcionalidades y modificando completamente su comportamiento con respecto al tratamiento de los archivos**. Para llevar a cabo esta tarea, se empezó por crear una copia de la clase *CUnpacker* al motor y cambiándole el nombre por ***CFileManager***, que fue el nombre establecido por la empresa.

Como se ha resaltado en el análisis realizado en el punto dos, en cada acceso a los archivos utilizados por el sistema se debe calcular previamente la ruta completa a dicho archivo, y dado que estas rutas podrían cambiar en el futuro, **se empezó por añadir a *CFileManager* un vector de *strings* que contendría las rutas de cada tipo de recurso**, creando también un enumerador para facilitar su uso:

```
enum EResourceType  {
        NE_RES_<TIPO 1>,
        ...
        NE_RES_<TIPO N>,
        NE_RES_MAX,
        NE_RES_FULLPATH
};
```

En el enumerador existen dos **tipos de recurso especiales**, que son:

- **NE_RES_MAX**: Indica el número máximo de recursos existentes, y se utiliza para inicializar el vector de rutas, pues no se trata de un vector estándar, sino uno propio de la empresa, y éste necesita del número de elementos máximo de que dispondrá.

- **NE_RES_FULLPATH**: Indica que la ruta para el recurso especificado ya es la ruta completa del mismo.

Así mismo, se realizaron las siguientes **modificaciones en las funciones:**

- **Init()**: se añadió la inicialización de dicho vector con NE_RES_MAX elementos, añadiéndole en el mismo orden descrito en el enumerador las rutas por defecto para cada tipo de recurso.

- **SetDir (ruta, tipo de recurso)**: Esta función sustituye la ruta establecida para el tipo de recurso indicado por aquella indicada en 'ruta'.

- **GetDir (tipo de recurso)**: Esta función devuelve la ruta establecida para el tipo de recurso indicado.

En el motor, hasta entonces, se tenía una serie de **funciones globales referentes a la adquisición de rutas completas para los recursos** por lo que, tras la modificación hecha, se hizo necesario deprecarlas e implementarlas y adaptarlas en *CFileManager* para que hiciesen uso de las rutas establecidas en la clase. Estas funciones fueron:

- **GetCommonFullPath (nombre de archivo, tipo de recurso)**: Esta función devuelve la ruta completa para el archivo especificado, dentro del directorio *Common*.

  *Si tipo de recurso es **NE_RES_FULLPATH***

  ruta = nombre de archivo

  Si no

  *ruta = "Common/" + **GetDir** (tipo de recurso) + nombre de archivo*

  Devuelve ruta

- **GetPlatformFullPath (nombre de archivo, tipo de recurso)**: Esta función devuelve la ruta completa dentro del directorio de la plataforma. Su uso es idéntico a *GetCommonFullPath* pero sustituyendo *"Common/"* por *GetPlatformPath()*, una función propia de la empresa que devuelve el nombre de la plataforma.

- **FileExists (…, tipo de recurso, *lugar)**: A esta función se le añade el tipo de recurso a buscar, y se define como orden de búsqueda *Plataforma → Common.*

  También se le añade otro parámetro (**lugar*), un puntero a un entero para que, en caso de no ser NULL, se le asigne un valor que indique dónde se ha encontrado el archivo, si en *Common* o en plataforma. Para ello, también se crea un nuevo enumerador (véase página siguiente):

```
Si tipo de recurso no es NE_RES_FULLPATH
    ruta = GetPlatformFullPath (nombre de archivo, tipo de recurso)
    hash = calcula hash para ruta
    Si Search (0, vectorCabeceras.Tamaño, hash) > -1
        encontado = true
        Si lugar no es NULL
            *lugar = NE_STORAGE_PACKED_PLATFORM
    Fin Si
    Si no encontrado
        ruta = GetCommonFullPath (nombre archivo, tipo de recurso)
        hash = calcula hash para ruta
        Si Search (0, vectorCabeceras.Tamaño, hash) > -1
            encontrado = true
            Si lugar no es NULL
                *lugar = NE_STORAGE_PACKED_COMMON
        Fin Si
    Fin Si
Si no
    hash = calcula hash para nombre de archivo
    Si Search (0, vectorCabeceras.Tamaño, hash) > -1
        encontado = true
        Si lugar no es NULL y nombre contiene “Common/”
            *lugar = NE_STORAGE_PACKED_COMMON
        Si no y Si lugar no es NULL
            *lugar = NE_STORAGE_PACKED_PLATFORM
        Fin Si
    Fin Si
Fin Si
return encontrado
```

```
enum EFileStorageType  {
        NE_STORAGE_PACKED_PLATFORM,
        NE_STORAGE_PACKED_COMMON,
        NE_STORAGE_PACKED
};
```

- **GetResourceFullPath (nombre de archivo, tipo de recurso)**: Esta función devuelve la ruta completa del archivo, detectando si está presente en la carpeta *Common* o en la carpeta de la plataforma.

```
Lugar = -1
Si FileExists (nombre de archivo, tipo de recurso, &Lugar)
        Si Lugar es NE_STORAGE_PACKED_COMMON
                ruta = GetCommonFullPath (nombre archivo, tipo recurso)
        Si no
                ruta = GetPlatformFullPath (nombre archivo, tipo recurso)
        Fin Si
Fin Si
Devolvemos ruta
```

Así mismo, también hubo otra función global que *CFileManager* tuvo que asimilar:

- **HardDriveFileExists (nombre de archivo)**: Esta función comprueba si un archivo existe en el disco duro o no.

```
existe = no
Abre archivo (nombre de archivo)
Si archivo se ha abierto
        existe = sí
        Cierra archivo
Fin Si
Devuelve existe
```

La adaptación de esta función hizo a *CFileManager* responsable no sólo de los archivos del paquete, sino también **de los archivos de disco**, y ello supuso los siguientes cambios en la estructura de funciones ya implementadas:

- **FileExists ()**: Hubo que añadir al enumerador creado opciones para indicar que el archivo encontrado estaba en el disco duro, y se definió como orden de búsqueda Paquete-Plataforma, Disco-Plataforma, Paquete-Common, Disco-Common:

Si tipo de recurso no es ***NE_RES_FULLPATH***
ruta = ***GetPlatformFullPath*** *(nombre de archivo, tipo de recurso)*
Comprueba Paquete-Plataforma
Si no encontrado
encontrado = ***HardDriveFileExists*** *(ruta)*
Si encontrado y lugar no es NULL
lugar =* ***NE_STORAGE_HDRIVE_PLATFORM*
Fin Si
Si no encontrado
ruta = ***GetCommonFullPath*** *(nombre archivo, tipo de recurso)*
Comprueba Paquete-Common
Fin Si
Si no encontrado
encontrado = ***HardDriveFileExists*** *(ruta)*
Si encontrado y lugar no es NULL
lugar =* ***NE_STORAGE_HDRIVE_COMMON*
Fin Si
Si no
Comprueba Paquete
Si no encontrado
encontrado = ***HardDriveFileExists*** *(nombre de archivo)*
Si encontrado
Si lugar no es NULL y nombre contiene "Common/"
lugar =* ***NE_STORAGE_PACKED_COMMON*
Si no y Si lugar no es NULL
lugar =* ***NE_STORAGE_PACKED_PLATFORM*
Fin Si
Fin Si
Fin Si
Fin Si
return encontrado

```
enum EFileStorageType  {
        …
        NE_STORAGE_HDRIVE_PLATFORM,
        NE_STORAGE_HDRIVE_COMMON,
        NE_STORAGE_HDRIVE
};
```

- **GetResourceFullPath ()**: Se añade comprobación de localización en disco.

```
Lugar = -1
Si FileExists (nombre de archivo, tipo de recurso, &Lugar)
        Si Lugar es NE_STORAGE_PACKED_COMMON
        o  Lugar es NE_STORAGE_HDRIVE_COMMON
                ruta = GetCommonFullPath (nombre archivo, tipo recurso)
        Si no
                ruta = GetPlatformFullPath (nombre archivo, tipo recurso)
        Fin Si
Fin Si
Devolvemos ruta
```

Con el sistema de rutas ya establecido y las funciones globales ya deprecadas, se pasó a realizar la **adaptación de *CFileManager* al uso de las clases propias del motor**. Para ello, se debía modificar la función principal, *GetFile*, para que, en vez de devolver un puntero a los datos del archivo cargado en memoria, hiciese uso de la clase *CFile* ya establecida en el motor y, de este modo, el funcionamiento del motor no se viese alterado cuando se aplicase el uso de *CFileManager*.

Hasta entonces, ***CFile*** se inicializaba pasando como parámetros el nombre del archivo que se quería abrir desde el disco duro, cómo se debía abrir el archivo (lectura, escritura, binario...) y si éste debía cargarse en memoria o no, pero dicha inicialización no servía para archivos contenidos en el paquete. Por ello, **fue necesario crear una sobrecarga de la función inicializadora** (*Init*) para poder pasarle directamente un puntero a los datos de un archivo recuperado desde el paquete y su tamaño, y que lo tratase como un archivo ya leído desde el disco duro y cargado en memoria.

// Prototipo de la función inicializadora original
bool Init (const char pszFile, EFileType eFileType, bool bLoadInMemory);*

// Prototipo de la función inicializadora sobrecargada
bool Init (const char pszFile, char* pszData, unsigned long dwSize);*

En *CFileManager*, para su adaptación al funcionamiento actual del motor, **la función *GetFile* establecida hasta ahora pasó a llamarse *GetData*,** y la **nueva definición de *GetFile*** quedó de la siguiente manera:

- **GetFile (nombre archivo, tipo recurso, acceso archivo, carga en memoria)**: ahora recibe como parámetros el nombre de archivo que se desea obtener, el tipo de recurso que es, cómo se desea acceder al archivo (binario, lectura, escritura...) y si debe cargarse en memoria. Su funcionamiento consiste en comprobar si un archivo existe en el paquete o en el disco, e inicializar una variable de tipo *CFile* utilizando la sobrecarga de la función inicializadora correspondiente al lugar en el que se encuentre el archivo. Como resultado, devolverá un puntero a una variable *CFile*:

  CFile file = nueva variable CFile*
  Lugar = -1
  *Si tipo de recurso es **NE_RES_FULLPATH***
  ruta = nombre archivo
  Si no
  *ruta = **GetResourceFullPath** (nombre archivo, tipo recurso)*
  Fin Si
  *Si **FileExists** (ruta, NE_RES_FULLPATH, &Lugar)*
  *Si acceso archivo no es escritura y Lugar < **NE_STORAGE_PACKED***
  *file->**Init** (ruta, **GetData**(ruta), **GetInfo**(ruta).m_iSize)*
  Fin Si
  Si file no se ha inicializado
  *file->**Init** (ruta, acceso archivo, carga en memoria)*
  Fin Si
  devuelve file

Con la adaptación al uso de *CFile*, **la clase *CFileManager* quedó ya preparada para su verdadera integración en el motor.**

Dicha integración comenzó por la situación, creación e inicialización del propio *FileManager*. Respecto a la **creación e inicialización**, ésta se colocó dentro de la función *Init* de la clase principal del motor, *CNaraEngine*, al inicio de la misma, dado que el *FileManager* sería utilizado por el resto de managers y clases del motor.

Respecto a la **situación**, el motor cuenta con una estructura que recoge punteros a la práctica totalidad de managers utilizados (audio, gráficos...), por lo que se añadió a dicha estructura un puntero para el *FileManager* aunque, dada la necesidad del *FileManager* a lo largo de la práctica totalidad de las clases del motor, esta no sería la única ubicación del mismo.

Como consecuencia de esta necesidad, se hizo imprescindible el **paso en cascada del puntero del *FileManager* para que éste se expandiera a lo largo de todas las clases del motor** para ser utilizado en los lugares en que fuese menester. Por ello, fue necesario editar los inicializadores de la práctica totalidad de clases del motor para poder realizar el paso del *FileManager* de hijos a padres y que éste se expandiese y llegase a todas las clases en las que era necesario. En este proceso se modificaron un total de 67 clases, lo cual supone aproximadamente un 98% de las clases que conforman el motor.

En el momento en que se observó el gran cambio que ésto suponía para el motor, y viendo que en casi el 80% de los casos el paso del puntero al *FileManager* únicamente se realizaba para poder pasárselo a la clase padre sin que la clase hijo hiciese uso de él, **se decidió evitar este paso en cascada y utilizar en su lugar una única variable de ámbito global**, *g_pFileManager*, por lo que se deshicieron los cambios realizados con ayuda del repositorio presente en la empresa, y se definió dicha variable global sin modificar la localización de su creación e inicialización.

Cuando el ámbito del *FileManager* fue accesible desde todas aquellas clases en las que era necesario, llegó el momento de su **utilización**. Como ya se ha dicho anteriormente, hasta ahora el acceso a ficheros se realizaba directamente mediante la clase *CFile*, con un cálculo anterior de la ruta completa del archivo. Por ejemplo, para acceder a una textura:

```
const char* pszPath = new char[256];
strcpy (pszPath, “Textures/”);
strcat (pszPath, pszFilename);
CFile* pFile = new CFile();
pFile->Init (pszPath, NE_FILE_READ_BINARY, false);
delete[] pszPath;
```

Ahora, todo este proceso se simplificaba directamente a:

CFile pFile = g_pFileManager->GetFile (pszFilename, NE_RES_TEXTURE);*

pues se toma el valor *NE_FILE_READ_BINARY* como el valor predefinido para el tipo de acceso, y *false* como valor predefinido para su carga en memoria, por lo que no es necesario indicarse.

Así pues, la adaptación del motor al uso de *FileManager* consistió en localizar todos los accesos a archivo y sustituir el acceso directo con *CFile* por el uso de *FileManager*.

Una vez sustituidos todos los accesos a archivo para su uso mediante FileManager, el concluyó la integración de *CFileManager* en el motor y llegó el momento de su prueba en proyectos de diversas envergaduras.

3.1.4. Realización de pruebas sobre pequeños proyectos.

Para esta prueba se utilizó un proyecto básico de ejemplo que tiene la empresa, llamado *TestProject2D*.

Lo primero que se hizo fue empaquetar los recursos utilizados utilizando el empaquetador creado, exceptuando archivos de configuración y borrando posteriormente los archivos empaquetados del disco para asegurar que el sistema recurriese directamente a los archivos del paquete. Después, se sustituyeron todos los accesos a archivos tal y como se hizo en el motor para que recurriesen a *FileManager* para obtenerlos, y, finalmente, se pasó a ejecutar el proyecto colocando puntos de interrupción en todas aquellas partes en las que se utilizase *FileManager* para asegurar que las rutas y los archivos obtenidos eran realmente los que se querían obtener.

Durante la realización de esta prueba no hubo ningún problema, por lo que se pasó a la aplicación del *FileManager* en proyectos de gran envergadura.

3.1.5. Realización de pruebas sobre proyectos de grandes dimensiones, y detección y corrección de errores.

Al probar sobre un proyecto de grandes dimensiones, como *Arson & Plunder*, **el primer problema vino por parte del empaquetador**.

Hasta entonces, las pruebas realizadas con el empaquetador colocaban el programa al mismo nivel que la carpeta que se debía empaquetar, pero para la aplicación sobre el juego se hacía necesaria la separación de ambos elementos. Esto sacó a la luz un error que hacía que las rutas de los archivos no se calculasen correctamente dentro del empaquetador, de manera que se guardaban de forma incorrecta en la cabecera del paquete y el juego era incapaz de encontrar los archivos.

Hasta ese momento, y dada la colocación del programa al mismo nivel que la carpeta a empaquetar, las rutas eran del tipo: "***CarpetaAEmpaquetar**Common\Archivo.ext*" (se ha resaltado en negrita lo que se le indicaba al empaquetador como parámetro de carpeta de entrada). El empaquetador, en su ejecución, eliminaba la ruta de la carpeta a empaquetar dejando sólo la ruta de las carpetas contenidas en su interior, y para ello cortaba de la ruta todos los caracteres que encontraba hasta el primer carácter '\' (incluido), así que mientras el empaquetador estuviese al mismo nivel que la carpeta a empaquetar no había problema, pero, con el cambio de ubicación, las rutas pasaron a ser más largas, del tipo: "***..\..\CarpetaAEmpaquetar**Common\Archivo.ext*", de manera que la ruta final en vez de ser "*Common\Archivo.ext"*, era "*..\CarpetaAEmpaquetar\Common\Archivo.ext*", y el FileManager era incapaz de encontrar el archivo.

Para solucionar este problema simplemente se cambió el delimitador de la ruta, pasando de ser "el primer ' \' que encuentres" a ser "la primera aparición de la carpeta de entrada". Con este simple cambio, el problema del empaquetador quedó solucionado.

El siguiente problema que surgió fue que, en *Arson & Plunder*, **el manager del SaveData**, aquel que se encarga de leer y escribir los datos guardados del juego y de la configuración, **se inicializaba antes que la clase principal del motor**, *CNaraEngine*, para leer la resolución de pantalla definida por el usuario y así poder crear la ventana de juego en base a esta lectura, pero, como ya se ha explicado en el apartado 3.1.3, el *FileManager* se crea e inicializa junto con *CNaraEngine*, por lo que el manager del SaveData no podía acceder a ningún fichero al no estar inicializado el *FileManager*.

Para solventar este problema se creó e inicializó el puntero al *FileManager* antes de inicializar el manager del SaveData, y en la inicialización dentro de *CNaraEngine*, se definió una cláusula para que sólo se inicializase si éste no había sido cread aún:

```
Si g_pFileManager es NULL
        g_pFileManager = Crea FileManager
        Inicializa g_pFileManager
Fin Si
```

De este modo, se podía inicializar sin problemas el *FileManager* antes de inicializar *CNaraEngine* y poder acceder a los ficheros del SaveData para poder leer la resolución de pantalla establecida por el usuario.

El último problema se encontró al aplicar el uso de *FileManager* en otro gran proyecto en su versión para Android, *Push Cars 2*. En este caso el problema vino por cómo se había definido la función *HardDriveFileExists*. En su versión original era:

```
encontrado = no
#Si estamos en Android
        Abre archivo usando AssetManager
        Si se ha abierto
                encontrado = sí
                cierra archivo
        Fin Si
#Si no
        Abre archivo de disco
        Si se ha abierto
                encontrado = sí
                cierra archivo
        Fin Si
#Fin Si
Devuelve encontrado
```

Como se puede ver, en el caso de Android las comprobaciones en disco se hacían directamente sobre el *AssetManager*, que se encarga de recuperar recursos empaquetados en el propio programa, por lo que en ningún momento se llega a comprobar si el archivo se encuentra en el disco o tarjeta de memoria del teléfono. Por ello, cuando se intentaba acceder a un archivo de disco, éste nunca llegaba a encontrarlo.

Para resolver este problema, simplemente se cambiaron las cláusulas del preprocesador para que la comprobación en disco se realizase en todos los casos, siempre que no se hubiese encontrado ya el archivo, quedando:

```
encontrado = no
#Si estamos en Android
        Abre archivo usando AssetManager
        Si se ha abierto
                encontrado = sí
                cierra archivo
        Fin Si
#Fin Si
Si no encontrado
        Abre archivo de disco
        Si se ha abierto
                encontrado = sí
                cierra archivo
        Fin Si
Fin Si
Devuelve encontrado
```

3.2. Soporte para UTF-8.

3.2.1. Creación de unas funciones básicas para el manejo de cadenas UTF-8, con compatibilidad para cadenas ASCII.

Para la creación de estas funciones, partimos de la declaración ya explicada en el apartado anterior, que fue dada por la propia empresa.

En el caso de *GetStringSizeUTF8*, el proceso para obtener el tamaño real de una cadena UTF-8 consiste en tomar el primer carácter, contar la cantidad de bits a uno iniciales y saltar tantos caracteres como unos se hayan contado,con un mínimo de uno, ya que en caracteres ASCII no hay unos en la cabecera del carácter. En pseudocódigo, el funcionamiento básico de esta función sería:

```
Para 'i = 0' hasta 'fin de la cadena', i++
        byte = pszString[i]
        Para 'j=0' hasta '8', j++
                Si byte[j] == 1
                        Salta ++
                Si no
                        Si Salta == 0
                                Salta = 1
                        Fin Si
                Fin Si
        Fin Para
        i += Salta – 1
        LongitudDeCadenaUTF8 ++
Fin Para
Devuelve LongitudDeCadenaUTF8
```

En el caso de *GetUTF8Position*, el proceso es idéntico a la función anterior, pero añadiendo una comprobación para saber si *LongitudDeCadenaUTF8* se corresponde con la posición que se está buscando (*uiPosition*):

```
Para 'i = 0' hasta 'fin de la cadena', i++
        Si encontrado == false
                byte = cadena[i]
                Para 'j=0' hasta '8', j++
                        Si LongitudDeCadenaUTF8 == uiPosition
                                encontrado = true
                                PosicionReal = i
                        Fin Si
                        Si encontrado == false
                                …
                        Fin Si
                Fin Para
                i += Salta – 1
                LongitudDeCadenaUTF8 ++
        Fin Si
Fin Para
Devuelve PosicionReal
```

En el caso de *GetCodeFromUTF8Char*, el algoritmo básico es similar, aunque primero comprobamos si se trata de un carácter ASCII o no (el carácter ASCII no requiere de ningún procesamiento),y en caso de no ser ASCII, se van saltando los unos de la cabecera y el cero que le precede en cada uno de los bytes, y va copiando el resto de bits para formar el código.

Finalmente, en el caso de *GetCodeFromUTF8String*, haciendo uso de las funciones anteriores, nos aseguramos de que el *uiPosition* indicado no sobrepase la longitud de la cadena (*GetStringSizeUTF8*), nos colocamos en la posición exacta de la cadena donde empieza el carácter del cuál queremos saber el código (*GetUTF8Position*), concatenamos los bytes que forman ese carácter en un *unsigned int*, y pasamos esta concatenación a *GetCodeFromUTF8Char* para obtener el código real del carácter, sea UTF8 o ASCII.

3.2.2. Modificación del sistema de carga de texto para que recurra a las funciones creadas en vez de recurrir a funciones estándar.

La modificación consistió en recorrer la clase *CFont*, encargada del procesamiento de los caracteres para su búsqueda en el archivo de fuente y su posterior impresión en la pantalla, en busca de fragmentos de código donde se utilizasen funciones estándar de tratamiento de cadenas y caracteres, con el objetivo de sustituir éstas por las funciones creadas.

Principalmente, los cambios se realizaron en bucles donde se recorría una cadena de texto desde su posición inicial a su posición final (cambio de la función de longitud de cadena estándar -*strlen(pszString)*- a propia -*GetStringSizeUTF8(pszString)*-), y donde se iba analizando el código de cada carácter de la misma (cambio de la forma en que se obtenía el carácter, pasando de una manera directa -*pzStrnig[position]*- a una indirecta -*GetCodeFromUTF8String (pszString, position)*-).

3.2.3. Realización de pruebas sobre proyectos de la empresa, y detección y corrección de errores.

Una vez realizadas las modificaciones sobre el motor de la empresa, se realizaron pruebas de rendimiento sobre algunos proyectos de gran envergadura, como *Arson & Plunder.* Estas pruebas consistieron principalmente en aplicar los cambios realizados y comprobar mediante la colocación de puntos de interrupción y ejecución paso a paso que, efectivamente, los códigos de los caracteres se calculan correctamente, las longitudes de las cadenas se corresponden al número de caracteres y no al número de bytes, y se muestran los caracteres correspondientes.

Durante esta fase de pruebas, y tras cambiar la codificación de los archivos de texto de ANSI a UTF-8, se comprobó que las funciones creadas calculaban correctamente el código de los caracteres, tanto los ASCII como los UTF-8, así como las longitudes de las cadenas, aún con mezcla de caracteres. Por tanto, la visualización de los caracteres no sufría ningún cambio respecto del uso con funciones estándar, pero sí **se apreció un notable descenso de la velocidad de carga de los archivos de texto**, debido al procesamiento de los mismos.

Para solucionar este problema, se revisó la estructura de las funciones creadas, y se determinó que en *GetCodeFromUTF8Char*, donde hasta entonces se hacía uso de un array de booleanos para representar los bits de cada carácter y los bits del *unsigned int* final, se debía trabajar directamente a nivel de bits sobre el *unsigned int* final, de manera que se ahorrase tiempo de procesamiento. Además, también se compactaron la cantidad de iteraciones sobre los bits del *unsigned int*, fusionando varios bucles en uno solo y simplificando el código, y se extrajeron cálculos estáticos del interior de los bucles, para que únicamente se calculasen una vez y no en cada iteración. En *GetCodeFromUTF8String*, para la concatenación de los bytes en un *unsigned int* también se pasó a realizar directamente a nivel de bits.

Tras la aplicación de estas mejoras se apreció una **importante mejoría de los tiempos de carga,** llegando a tiempos similares a los que había antes de la realización de los cambios. De este modo, el sistema quedó ya preparado para la aplicación de la traducción al idioma chino del *Push Cars 2*, como se puede observar en las siguientes capturas de pantalla.

A la izquierda, el uso de las funciones descritas usando caracteres ASCII. Abajo, la aplicación de las funciones para mostrar caracteres UTF-8.

4. EVALUACIÓN Y CONCLUSIONES FINALES

A lo largo de este documento se ha detallado la creación e implementación de varias funcionalidades sobre el motor de videojuegos utilizado por la empresa: *Nara Engine*.

Actualmente, **el *FileManager* y el empaquetador** asociado al mismo se han establecido ya como parte de todos los proyectos de la empresa sin más problemas que los detallados en el apartado anterior. Esto ha supuesto una gran mejora para el motor, tanto por la simplicidad con la que ahora se accede a los archivos, como por la posibilidad de poder evitar que los usuarios finales modifiquen los recursos del juego a placer.

Respecto a las **funciones creadas para el manejo de cadenas UTF-8**, éstas se han aplicado sin problema alguno a todos los proyectos sin que por ello varíe un ápice sus funcionalidades, y ha supuesto una gran mejora para el motor y para la empresa, puesto que de este modo puede ampliar su mercado hacia países en los que los caracteres utilizados puedan codificarse mediante UTF-8, como son los países orientales.

Ambas mejoras han cumplido con los objetivos marcados, solucionando los problemas con los que contaba la empresa de manera práctica y eficiente, mejorando el comportamiento del motor y ampliando sus características, por lo que **se da por finalizado el proyecto con una visión positiva de los cambios realizados**, tanto por parte de la empresa como personalmente.

5. REFERENCIAS

C++ (Reference):

http://www.cplusplus.com/reference/

UTF-8 (Wikipedia):

http://en.wikipedia.org/wiki/UTF-8

Ninja Fever:

http://www.ninjafever.com

Arson & Plunder:

http://www.arsonandplunder.com

Push Cars 2:

http://www.push-cars.com

ANEXO A

Empaquetador

PACKER.H

```
#ifndef _PACKER_H_
#define _PACKER_H_

#include <fstream>
#include <sstream>
#include <iomanip>
#include <algorithm>
#include <iostream>
#include <vector>
#include <cstring>
#include <windows.h>
#include "CoreUtils.h"
#include "SizedVector.h"
#include "FileManager.h"

#define DATA_FILE     "data.tmp"

enum EErrors
{
        ERROR_NONE = 0,
        ERROR_INIT,
        ERROR_FILE,
        ERROR_FOLDER,
        ERROR_MEMORY
};

enum EExcludedType
{
        EX_FOLDER,
        EX_FOLDER_WITH_PATH,
        EX_FILENAME,
        EX_FILE_WITH_PATH,
        EX_EXTENSION,
        EX_PART_FILENAME,
        EX_PART_FILE_WITH_PATH
};

class CPacker
{
public:
        Cpacker         () : m_bInit (false) {}
        ~Cpacker        (){ CPacker::End(); }

        bool            Init            (const char *pszInputFolder, const char *pszOutputFolder,
                                         const char *pszExcludedFileName, bool bRecreate);
        bool            IsOk            ()              const   { return m_bInit; }
        int             GetLastsError   ()              const   { return m_iError; }
        virtual void    End             ();
        bool            Start           ();

private:
        bool            m_bInit;
        int             m_iError;
        const char      *m_pszOutputFolder;             // Nombre de la carpeta de salida
        const char      *m_pszInputFolder;              // Nombre de la carpeta a empaquetar

        int             m_iHeaderFiles;                 // Numero de archivos empaquetados
        int             m_iHeaderSize;                  // Tamanyo del header
        int             m_iDataSize;                    // Tamanyo de datos
        bool            m_bRecreate;                    // Recrear estructura de excluidos?
        int             m_iCounter;                     // Contador de archivos empaquetados
```

```
        std::ifstream    m_ifsIn;                    // Archivo de lectura
        std::ofstream    m_ofsData;                  // Archivo con los datos empaquetados

        std::vector<std::string>        m_msgError;         // Log de errores
        std::vector<std::string>        m_msgDiscard;       // Log de archivos descartados

        std::vector<std::string>        m_excludedFolder;                   // Exclusiones
        std::vector<std::string>        m_excludedFolderWithPath;
        std::vector<std::string>        m_excludedFilename;
        std::vector<std::string>        m_excludedFileWithPath;
        std::vector<std::string>        m_excludedPartFilename;
        std::vector<std::string>        m_excludedPartFileWithPath;
        std::vector<std::string>        m_excludedExtension;

        ne::core::SizedVector<ne::core::THeader>    m_Header;           // Datos del header


        int     m_iSize;            // Contadores, posicionadores y variables de copia de archivos
        int     m_iOldSize;
        int     m_iPos;
        int     m_iOldPos;


        bool        LoadExcludedFile        (const char *pszFileName);
        bool        CopyFiles               (const char *pszDir);
        int         Copy                    (std::ifstream &ifsIn, std::ofstream &ofsOut);
        bool        IsExcluded              (const char *pszPath, bool bIsFolder);
        bool        HasSpaces               (const char *pszPath);
        std::string GetCompletePath         (const char* pszFileName);
        std::string ReplaceChar             (std::string str, char cChar1, char cChar2);
        void        PrintErrors             ();
        bool        FolderContainsFile      (const char *pszDir, const char *pszFileName);
        int         CountFilesToPack        (const char *pszDir);
        bool        ValidateHeaderHash      ();
        bool        RecreateFolder          (const char* pszDir);
        bool        RecreateFile            (const char* pszFile);
        bool        CreateFolder            (const char* pszDir);
        bool        CreateFolderTree        (const char* pszDir);
        bool        DeleteFolderFiles       (const char* pszDir);
        bool        DeleteFolderTree        (const char* pszDir);
};

#endif
```

PACKER.CPP

```
#include "Packer.h"


// Indica que hash es mayor
int SortHeaders (const void* tHead1, const void* tHead2)
{
	int iRes = 0;
	ne::core::THeader* ptH1 = (ne::core::THeader*)tHead1;
	ne::core::THeader* ptH2 = (ne::core::THeader*)tHead2;

	if (ptH1->m_uiHash < ptH2->m_uiHash)
		iRes = -1;
	else if (ptH1->m_uiHash > ptH2->m_uiHash)
		iRes = 1;

	return iRes;
}



//////////////////////////////////////////////////////
///  Inicializacion
//////////////////////////////////////////////////////
bool CPacker::Init (const char *pszInputFolder, const char *pszOutputFolder, const char
*pszExcludedFileName, bool bRecreate)
{
	bool bOk = true;

	End();

	m_pszInputFolder = pszInputFolder;
	m_pszOutputFolder = pszOutputFolder;
	m_bRecreate = bRecreate;
	m_iError = ERROR_NONE;

	// Anyade por defecto el archivo de salida y su temporal a la lista de archivos excluidos
	m_excludedFilename.push_back(RESOURCES_FILE);
	m_excludedFilename.push_back(DATA_FILE);

	// Cargamos en memoria la lista de archivos y directorios excluidos (Si es necesario)
	if ( strcmp(pszExcludedFileName, "") )
		bOk = LoadExcludedFile(pszExcludedFileName);

	// Creamos carpeta de salida
	if (bOk)
	{
		if (bRecreate)
		{
			DeleteFolderFiles(m_pszOutputFolder);
			DeleteFolderTree(m_pszOutputFolder);
		}

		bOk = CreateFolder(m_pszOutputFolder);

		if (!bOk)
		{
			m_msgError.push_back("   Error creating output folder.");
			m_iError = ERROR_FOLDER;
		}
	}
```

```
	// Inicializamos sized vector
	if (bOk)
	{
		bOk = m_Header.Init( CountFilesToPack(m_pszInputFolder), true);

		if (!bOk)
		{
			m_msgError.push_back("   Error creating Sized Vector.");
			m_iError = ERROR_MEMORY;
		}
	}

	// Abrimos archivo de salida
	if (bOk)
	{
		m_ofsData.open (GetCompletePath(DATA_FILE), std::ios::binary);
		bOk = m_ofsData.is_open();

		if (!bOk)
		{
			m_msgError.push_back("   Error opening data temp file for writing.");
			m_iError = ERROR_INIT;
		}
	}

	if (bOk)
	{
		m_iSize = 0;
		m_iOldSize = 0;
		m_iPos = 0;
		m_iOldPos = 0;
		m_iHeaderFiles = 0;
		m_iHeaderSize = 0;
		m_iDataSize = 0;
	}

	m_bInit = true;
	if ( !bOk )
	{
		PrintErrors();
		CPacker::End();
	}

	return bOk;
}
```

```
//////////////////////////////////////////////////
/// Fin
//////////////////////////////////////////////////
void CPacker :: End ()
{
	if ( IsOk() )
	{
		if (m_ofsData.is_open())
			m_ofsData.close();
		if (m_ifsIn.is_open())
			m_ifsIn.close();

		m_excludedFolder.clear();
		m_excludedFolderWithPath.clear();
		m_excludedFilename.clear();
		m_excludedFileWithPath.clear();
		m_excludedPartFilename.clear();
		m_excludedPartFileWithPath.clear();
		m_excludedExtension.clear();

		m_msgError.clear();
		m_msgDiscard.clear();

		for (int i=0; i<m_Header.GetNumItems(); i++)
			SAFE_DELETE_ARRAY(m_Header[i].m_pszName);
		m_Header.Clear();

		m_bInit = false;
	}
}
```

```
//////////////////////////////////////////////////
/// Empaqueta
//////////////////////////////////////////////////
bool CPacker :: Start ()
{
	bool bOk = false;

	if (m_bInit)
	{
		// Copiamos todos los archivos y generamos un índice de ellos (list)
		std::cout << std::endl << "PACKING FILES..." << std::endl;
		m_iCounter = 0;
		bOk = CopyFiles (m_pszInputFolder);
	}


	if (bOk)
	{
		// Cerramos temporales
		m_ofsData.close();

		// Ordenamos sized vector
		std::cout << std::endl << "SORTING FILES..." << std::endl;
		m_Header.Sort(SortHeaders);

		// Comprobamos que no haya hash repetidos
		bOk = ValidateHeaderHash ();
	}

	if (bOk)
	{
		std::cout << std::endl << "CREATING PACKAGE..." << std::endl;

		// Abrimos archivo de salida definitivo
		m_ofsData.open ( GetCompletePath(RESOURCES_FILE), std::ios::binary);
		bOk = m_ofsData.is_open();

		if (!bOk)
		{
			m_msgError.push_back("   Error opening output package file for writing.");
			m_iError = ERROR_INIT;
		}
	}

	if (bOk)
	{
		// guardamos cantidad de archivos empaquetados
		m_ofsData.write (reinterpret_cast<const char *> (&m_iHeaderFiles), sizeof(int));
		m_iHeaderSize = sizeof(int);

		// Guardamos Header
		for (int i=0; i<m_Header.GetNumItems(); i++)
		{
		  m_ofsData.write (reinterpret_cast<const char *>(&m_Header[i].m_iInit), sizeof(int));
		  m_ofsData.write (reinterpret_cast<const char *>(&m_Header[i].m_iSize), sizeof(int));
		  m_ofsData.write (reinterpret_cast<const char *>(&m_Header[i].m_uiHash), sizeof(unsigned int));
		  m_ofsData.write (reinterpret_cast<const char *>(&m_Header[i].m_iNameLen), sizeof(int));
		 m_ofsData.write (reinterpret_cast<const char *>(m_Header[i].m_pszName), m_Header[i].m_iNameLen);

		  m_iHeaderSize += sizeof(int)*3 + sizeof(unsigned int) + m_Header[i].m_iNameLen;
		}
```

```
		bOk = (m_iHeaderSize > sizeof(int)) ? true : false;
		if (!bOk)
		{
			m_msgError.push_back("   Error creating header.");
			m_iError = ERROR_FILE;
		}
	}

	if (bOk)
	{
		// Abrimos data para copiarlo
		m_ifsIn.open (GetCompletePath(DATA_FILE), std::ios::binary);
		bOk = m_ifsIn.is_open();

		if (!bOk)
		{
			m_msgError.push_back("   Error opening data temp file for reading.");
			m_iError = ERROR_INIT;
		}
	}


	if (bOk)
	{
		// copiamos data y cerramos
		m_iDataSize = Copy (m_ifsIn, m_ofsData);
		bOk = (m_iDataSize > 0) ? true : false;;
		m_ifsIn.close();

		if (!bOk)
		{
			m_msgError.push_back("   Error copying data temp file.");
			m_iError = ERROR_FILE;
		}
	}


	if (bOk)
	{
		// Cerramos archivo definitivo y borramos temporales
		m_ofsData.close();
		remove ( GetCompletePath(DATA_FILE).c_str() );
	}

	// Resumen final del proceso
	if (m_bInit)
	{
		std::cout << std::endl << "-------------------------------------------------" << std::endl << std::endl;

		if (!bOk)
		{
			PrintErrors();
			End();
			remove ( GetCompletePath(DATA_FILE).c_str() );
			remove ( GetCompletePath(RESOURCES_FILE).c_str() );

			if (m_bRecreate)
			{
				DeleteFolderFiles(m_pszOutputFolder);
				DeleteFolderTree(m_pszOutputFolder);
			}
		}
```

```
                else
                {
                    std::cout << std::endl << "PACKAGE CREATED CORRECTLY!" << std::endl << std::endl;

                    std::cout << "Package info:" << std::endl;
                    std::cout << "   Files Packed:\t" << m_iHeaderFiles << " files." << std::endl;
                    std::cout << "   Header Size: \t";
                    if (m_iHeaderSize > 1024)
                        std::cout << std::fixed << std::showpoint << std::setprecision(2) << (float)
                        (m_iHeaderSize/1024.f) << " KB (" << m_iHeaderSize << " B)" << std::endl;
                    else
                        std::cout << m_iHeaderSize << " B" << std::endl;

                    std::cout << "   Data Size:   \t";
                    if (m_iDataSize > (1024^2) )
                        std::cout << std::fixed << std::showpoint << std::setprecision(2) << (float)
                        (m_iDataSize/1024.f/1024.f) << " MB (" << m_iDataSize << " B)" << std::endl;
                    else if (m_iDataSize > 1024)
                        std::cout << std::fixed << std::showpoint << std::setprecision(2) << (float)
                        (m_iDataSize/1024.f) << " KB (" << m_iDataSize << " B)" << std::endl;
                    else
                        std::cout << m_iDataSize << " B" << std::endl;

                    std::cout << "   Total Size:  \t";
                    if (m_iDataSize > (1024^2) )
                        std::cout << std::fixed << std::showpoint << std::setprecision(2) << (float)
                        ((m_iHeaderSize+m_iDataSize)/1024.f/1024.f) << " MB (" <<
                        (m_iHeaderSize+m_iDataSize) << " B)" << std::endl;
                    else if (m_iDataSize > 1024)
                        std::cout << std::fixed << std::showpoint << std::setprecision(2) << (float)
                        ((m_iHeaderSize+m_iDataSize)/1024.f) << " KB (" << (m_iHeaderSize+m_iDataSize)
                        << " B)" << std::endl;
                    else
                        std::cout << (m_iHeaderSize+m_iDataSize) << " B" << std::endl;
                }
        }

        return bOk;
}


////////////////////////////////////////////////////////
/// Recorre carpeta y subcarpetas, y copia los archivos al paquete
////////////////////////////////////////////////////////
bool CPacker :: CopyFiles (const char *pszDir)
{
        WIN32_FIND_DATA fdFile;
        HANDLE hFind = NULL;
        char szPath[2048];
        bool bOk = true;
        ne::core::THeader tHeader;

        // Listamos todos los archivos, sea cual sea su tipo
        sprintf_s (szPath, "%s\\*.*", pszDir);

        // Comprobamos si es un directorio accesible
        if((hFind = FindFirstFile(szPath, &fdFile)) == INVALID_HANDLE_VALUE)
        {
                m_msgError.push_back( std::string(pszDir) + std::string(": Path not found!") );
                m_iError = ERROR_FOLDER;
                bOk = false;
        }
```

```
if (bOk)
{
        do
        {
            // Evita los predefinidos "." y ".."
            if (strcmp(fdFile.cFileName, ".") != 0 && strcmp(fdFile.cFileName, "..") != 0)
            {
                // Contruimos el path completo
                sprintf_s (szPath, "%s\\%s", pszDir, fdFile.cFileName);

                // Si es un directorio, recorrelo con recursividad
                if(fdFile.dwFileAttributes &FILE_ATTRIBUTE_DIRECTORY )
                {
                    if (HasSpaces (szPath))        // Si tiene espacios
                    {
                        m_msgError.push_back( std::string(szPath) + std::string(": Foldername has spaces.") );
                        m_iError = ERROR_FOLDER;
                        bOk = false;
                    }
                    else if (!IsExcluded (szPath, true) ) // Si no esta excluido
                    {
                        std::cout << std::endl << szPath << std::endl;
                        bOk = CopyFiles (szPath);
                    }
                    else if (m_bRecreate)          // Si hay que recrear carpeta
                    {
                        std::cout << std::endl << "To DataPacker: " << szPath << std::endl;
                        bOk = RecreateFolder (szPath);
                    }
                }

                // Y si es un archivo, guardalo
                else
                {
                    if (!IsExcluded (szPath, false))
                    {
                        m_iCounter++;

                        // Nombre de archivo con ruta
                        std::string filename = szPath;
                        int iPos = filename.find(m_pszInputFolder) + strlen(m_pszInputFolder) + 1;
                        // Elimina nombre de carpeta contenedora
                        filename = filename.substr(iPos);
                        // Cambia los \ por / para evitar errores en el unpacker
                        filename = ReplaceChar(filename, '\\', '/');
                        std::cout << "   " << filename << std::endl;

                        // Abrimos archivo de entrada
                        m_ifsIn.open(szPath, std::ios::binary);
                        bOk = m_ifsIn.is_open();

                        if (bOk)
                        {
                                // Copiamos en data y cerramos
                                m_iSize = Copy (m_ifsIn, m_ofsData);
                                tHeader.m_iSize = m_iSize;

                                // Calculamos posicion
                                m_iPos = m_iOldPos + m_iOldSize;
                                tHeader.m_iInit = m_iPos;
```

```
                                        // Anyadimos nombre
                                        tHeader.m_iNameLen = filename.length() + 1;
                                        tHeader.m_pszName = new char[tHeader.m_iNameLen];
                                       strcpy_s (tHeader.m_pszName, tHeader.m_iNameLen,  filename.c_str());

                                        // Calculamos hash
                                        tHeader.m_uiHash = ne::core::neHash(tHeader.m_pszName);

                                        // Guardamos en memoria
                                        m_Header.AddItem(&tHeader);
                                        m_iHeaderFiles++;

                                        // Reseteamos estructura
                                        tHeader.m_iInit = 0;
                                        tHeader.m_iSize = 0;
                                        tHeader.m_iNameLen = 0;
                                        tHeader.m_pszName = NULL;
                                        tHeader.m_uiHash = 0;

                                        // Preparamos datos para siguiente archivo
                                        m_iOldPos = m_iPos;
                                        m_iPos = 0;
                                        m_iOldSize = m_iSize;
                                        m_iSize = 0;

                                        // Cerramos archivo
                                        m_ifsIn.close();
                                }
                                else
                                {
                                        m_msgError.push_back( std::string(szPath) + std::string(": Can't open this file!") );
                                        m_iError = ERROR_FILE;
                                }
                        }

                        // Si esta excluido y esta activa la recreacion
                        else if (m_bRecreate)
                        {
                                bOk = RecreateFile (szPath);
                        }
                }
        }
    }
    while(FindNextFile(hFind, &fdFile) && bOk); // Buscamos el siguiente archivo
}

// Cerramos lector y salimos
FindClose(hFind);

return bOk;
}
```

```
//////////////////////////////////////////////////////
/// Copia una carpeta entera en DataPackage
//////////////////////////////////////////////////////
bool CPacker :: RecreateFolder (const char* pszDir)
{
	WIN32_FIND_DATA fdFile;
	HANDLE hFind = NULL;
	char szPath[2048];
	bool bOk = true;

	// Listamos todos los archivos, sea cual sea su tipo
	sprintf_s (szPath, "%s\\*.*", pszDir);

	// Comprobamos si es un directorio accesible
	 if((hFind = FindFirstFile(szPath, &fdFile)) == INVALID_HANDLE_VALUE)
	{
		m_msgError.push_back( std::string(pszDir) + std::string(": Path not found!") );
		m_iError = ERROR_FOLDER;
		 bOk = false;
	}

	if (bOk)
	{
		do
		{
			// Evita los predefinidos "." y ".."
			if (strcmp(fdFile.cFileName, ".") != 0 && strcmp(fdFile.cFileName, "..") != 0)
			{
				// Contruimos el path completo
				sprintf_s (szPath, "%s\\%s", pszDir, fdFile.cFileName);

				// Si es un directorio, recorrelo con recursividad
				if(fdFile.dwFileAttributes &FILE_ATTRIBUTE_DIRECTORY )
				{
				   if (HasSpaces (szPath))		// Si tiene espacios
				   {
					m_msgError.push_back( std::string(szPath)
						+ std::string(": Foldername has spaces.") );
					m_iError = ERROR_FOLDER;
					bOk = false;
				   }
				   else // Todo bien
				   {
					bOk = RecreateFolder (szPath);
				   }
				}

				// Y si es un archivo, guardalo
				else
				   bOk = RecreateFile(szPath);
			}
		}
		while(FindNextFile(hFind, &fdFile) && bOk); // Buscamos el siguiente archivo
	}

	// Cerramos lector y salimos
	FindClose(hFind);

	return bOk;
}
```

```
//////////////////////////////////////////////////////
/// Copia un archivo en DataPackage
//////////////////////////////////////////////////////
bool CPacker :: RecreateFile (const char* pszFile)
{
	std::ifstream ifsIn;
	std::ofstream ofsOut;
	bool bOk = true;

	// Comprueba que no sea el archivo .dat ni su temporal
	std::string szPackerFile = "";
	std::string szFolder = pszFile;
	std::string szFilename = szFolder.substr(szFolder.find_last_of('\\') + 1);

	if (strcmp(szFilename.c_str(), RESOURCES_FILE) && strcmp(szFilename.c_str(), DATA_FILE) )
	{

		// Saca el path principal del archivo de destino
		int iPos = szFolder.find_first_of(m_pszInputFolder);
		if (iPos == std::string::npos)
		{
			bOk = false;
			std::string szMsg = std::string("   Error creating dest filename for recreate.");
			m_iError = ERROR_FOLDER;
			m_msgError.push_back( szMsg );
		}

		// Comprueba si existe la carpeta de destino y, si no existe, la crea con su arbol
		if (bOk)
		{
			szFolder = szFolder.substr(iPos + strlen(m_pszInputFolder));
			if (szFolder[0] != '\\')
			{
				szFolder.insert(0, "\\");
			}
			szPackerFile = m_pszOutputFolder + szFolder;

			szFolder = szPackerFile.substr(0, szPackerFile.find_last_of('\\')+1);
			bOk = CreateFolderTree (szFolder.c_str());
		}

		// Abrimos archivo de entrada
		if (bOk)
		{
			ifsIn.open(pszFile, std::ios::binary);
			bOk = ifsIn.is_open();

			if (!bOk)
			{
				std::string szMsg = std::string("   Error opening input file '") +
					std::string(pszFile) + std::string("' for recreate.");
				m_iError = ERROR_FILE;
				m_msgError.push_back( szMsg );
			}
		}

		// Abrimos archivo de salida
		if (bOk)
		{
			ofsOut.open(szPackerFile, std::ios::binary);
			bOk = ofsOut.is_open();
```

```
                        if (!bOk)
                        {
                                std::string szMsg = std::string("   Error opening output file '") +
                                        std::string(szPackerFile) + std::string("' for recreate.");
                                m_iError = ERROR_FILE;
                                m_msgError.push_back( szMsg );
                        }
                }

                if (bOk)
                {
                        // Copiamos en data y cerramos
                        bOk = (Copy (ifsIn, ofsOut) > 0) ? true : false;

                        if (!bOk)
                        {
                                std::string szMsg = std::string("   Error copying file '") +
                                        std::string(pszFile) + std::string("' for recreate.");
                                m_iError = ERROR_FILE;
                                m_msgError.push_back( szMsg );
                        }

                        ifsIn.close();
                        ofsOut.close();
                }
        }

        return bOk;
}


//////////////////////////////////////////////////////
/// Crea un arbol de carpetas
//////////////////////////////////////////////////////
bool CPacker :: CreateFolderTree (const char* pszDir)
{
        bool bOk = true;
        std::string szFolder = pszDir;
        std::vector<std::string> pszFolders;

        // Nos aseguramos de que termine en '\'
        if (szFolder[szFolder.length()-1] != '\\')
                szFolder += "\\";

        // Sacamos carpetas
        for (unsigned int i=szFolder.find_first_of('\\')+1; i<szFolder.length(); i++)
        {
                if (szFolder[i] == '\\')
                {
                        std::string a = szFolder.substr(0, i);
                        pszFolders.push_back(szFolder.substr(0, i));
                }
        }

        // Creamos carpetas
        for (std::vector<std::string>::const_iterator i = pszFolders.begin(); i<pszFolders.end(); i++)
                bOk = bOk & CreateFolder( i->c_str() );

        return bOk;
}
```

```
//////////////////////////////////////////////////////
/// Crea una carpeta
//////////////////////////////////////////////////////
bool CPacker :: CreateFolder (const char* pszDir)
{
	bool bReturn = true;

	if (!CreateDirectory(pszDir, NULL) && ERROR_ALREADY_EXISTS != GetLastError())
	{
	   bReturn = false;
	   m_iError = ERROR_FOLDER;
	   std::string szMsg = std::string("   Error creating folder '") + std::string(pszDir) + std::string("'.");
	   m_msgError.push_back( szMsg );
	}
	return bReturn;
}

//////////////////////////////////////////////////////
/// Elimina archivos de una carpeta
//////////////////////////////////////////////////////
bool CPacker :: DeleteFolderFiles (const char* pszDir)
{
	WIN32_FIND_DATA fdFile;
	HANDLE hFind = NULL;
	char szPath[2048];
	bool bOk = true;

	// Listamos todos los archivos, sea cual sea su tipo
	sprintf_s (szPath, "%s\\*.*", pszDir);

	// Comprobamos si es un directorio accesible
	if((hFind = FindFirstFile(szPath, &fdFile)) == INVALID_HANDLE_VALUE)
	{
		bOk = false;
	}
	if (bOk)
	{
		do
		{
			// Evita los predefinidos "." y ".."
			if (strcmp(fdFile.cFileName, ".") != 0 && strcmp(fdFile.cFileName, "..") != 0)
			{
				// Contruimos el path completo
				sprintf_s (szPath, "%s\\%s", pszDir, fdFile.cFileName);

				// Si es un directorio, recorrelo con recursividad
				if(fdFile.dwFileAttributes &FILE_ATTRIBUTE_DIRECTORY )
					bOk = DeleteFolderFiles (szPath);

				// Y si es un archivo, elimina
				else
					remove(szPath);
			}
		}
		while(FindNextFile(hFind, &fdFile) && bOk); // Buscamos el siguiente archivo
	}

	// Cerramos lector y salimos
	FindClose(hFind);

	return bOk;
}
```

```
////////////////////////////////////////////////////
/// Elimina carpetas de una carpeta
////////////////////////////////////////////////////
bool CPacker :: DeleteFolderTree (const char* pszDir)
{
	WIN32_FIND_DATA fdFile;
	HANDLE hFind = NULL;
	char szPath[2048];
	bool bOk = true;

	// Listamos todos los archivos, sea cual sea su tipo
	sprintf_s (szPath, "%s\\*.*", pszDir);

	// Comprobamos si es un directorio accesible
	 if((hFind = FindFirstFile(szPath, &fdFile)) == INVALID_HANDLE_VALUE)
	{
		bOk = false;
	}

	if (bOk)
	{
		do
		{
			// Evita los predefinidos "." y ".."
			if (strcmp(fdFile.cFileName, ".") != 0 && strcmp(fdFile.cFileName, "..") != 0)
			{
				// Contruimos el path completo
				sprintf_s (szPath, "%s\\%s", pszDir, fdFile.cFileName);

				// Si es un directorio, recorrelo con recursividad
				if(fdFile.dwFileAttributes &FILE_ATTRIBUTE_DIRECTORY )
				{
					bOk = DeleteFolderTree (szPath);
					RemoveDirectory(szPath);
				}
			}
		}
		while(FindNextFile(hFind, &fdFile) && bOk); // Buscamos el siguiente archivo
	}

	// Cerramos lector y salimos
	FindClose(hFind);

	return bOk;
}



////////////////////////////////////////////////////
/// Sustituye un caracter en un string
////////////////////////////////////////////////////
std::string CPacker :: ReplaceChar (std::string str, char cChar1, char cChar2) {
	for (unsigned int i = 0; i < str.length(); ++i) {
		if (str[i] == cChar1)
			str[i] = cChar2;
		}

	return str;
}
```

```
//////////////////////////////////////////////////////
/// Copia archivo de entrada en archivo de salida
//////////////////////////////////////////////////////
int CPacker :: Copy (std::ifstream &ifsIn, std::ofstream &ofsOut)
{

	// Obtenemos tamanyo
	ifsIn.seekg (0,std::ios::end);
	int iSize = (int)ifsIn.tellg();

	if (iSize != -1)
	{
		// Volvemos al inicio del archivo
		ifsIn.seekg (0, std::ios::beg);

		// Obtiene datos
		char *pszBuffer = new char[iSize];
		ifsIn.read (pszBuffer, iSize);

		// Guarda datos en data
		ofsOut.write(pszBuffer, iSize);

		// Libera memoria y cierra archivo
		delete[] pszBuffer;
	}

	// Devuelve el tamanyo del archivo copiado
	return iSize;
}

//////////////////////////////////////////////////////
/// Comprueba si un directorio esta excluido
//////////////////////////////////////////////////////
bool CPacker :: IsExcluded (const char *pszPath, bool bIsFolder)
{
	bool bExcluded = false;
	int iPos = 0;
	std::string szPath = pszPath;

	// CARPETAS
	if (bIsFolder)
	{
		// Path completo
		for (std::vector<std::string>::const_iterator i = m_excludedFolderWithPath.begin();
		i<m_excludedFolderWithPath.end() && !bExcluded; i++)
		{
			iPos = szPath.find((*i));
			if (iPos >= 0)
				bExcluded = true;
		}

		// Nombre de carpeta
		if (!bExcluded)
		{
			std::string szFolderName = szPath.substr(szPath.find_last_of('\\') + 1);
			for (std::vector<std::string>::const_iterator i = m_excludedFolder.begin();
			i<m_excludedFolder.end() && !bExcluded; i++)
			{
				if (!strcmp((*i).c_str(), szFolderName.c_str()) )
					bExcluded = true;
			}
		}
	}
```

```
	// ARCHIVOS
	else
	{
		std::string szFilename = szPath.substr(szPath.find_last_of('\\') + 1);
		std::string szExtension = szFilename.substr(szFilename.find_last_of('.') + 1);

		// Extension
		if (!bExcluded)
		{
			for (std::vector<std::string>::const_iterator i = m_excludedExtension.begin();
			i<m_excludedExtension.end() && !bExcluded; i++)
			{
				if (!strcmp((*i).c_str(), szExtension.c_str()))		bExcluded = true;
			}
		}

		// Archivo con ruta
		if (!bExcluded)
		{
			for (std::vector<std::string>::const_iterator i = m_excludedFileWithPath.begin();
			i<m_excludedFileWithPath.end() && !bExcluded; i++)
			{
				iPos = szPath.find((*i));
				if (iPos >= 0)		bExcluded = true;
			}
		}

		// Archivo sin ruta
		if (!bExcluded)
		{
			for (std::vector<std::string>::const_iterator i = m_excludedFilename.begin();
			i<m_excludedFilename.end() && !bExcluded; i++)
			{
				if (!strcmp((*i).c_str(), szFilename.c_str()) )		bExcluded = true;
			}
		}

		// Parte de nombre de archivo con ruta
		if (!bExcluded)
		{
			for (std::vector<std::string>::const_iterator i = m_excludedPartFileWithPath.begin();
			i<m_excludedPartFileWithPath.end() && !bExcluded; i++)
			{
				iPos = szPath.find((*i));
				if (iPos >= 0)		bExcluded = true;
			}
		}

		// Parte de nombre de archivo sin ruta
		if (!bExcluded)
		{
			for (std::vector<std::string>::const_iterator i = m_excludedPartFilename.begin();
			i<m_excludedPartFilename.end() && !bExcluded; i++)
			{
				iPos = szPath.find((*i));
				if ( iPos >= 0)		bExcluded = true;
			}
		}
	}

	if (bExcluded)
		m_msgDiscard.push_back(pszPath);

	return bExcluded;
}
```

```
//////////////////////////////////////////////////////
/// Obtiene path completo
//////////////////////////////////////////////////////
std::string CPacker :: GetCompletePath (const char* pszFileName)
{
	std::string szPath = "";
	szPath.append(m_pszOutputFolder);
	szPath.append("\\");
	szPath.append(pszFileName);

	return szPath;
}



//////////////////////////////////////////////////////
/// Comprueba si hay espacios
//////////////////////////////////////////////////////
bool CPacker :: HasSpaces (const char *pszPath)
{
	bool bSpaces = false;

	for (unsigned int i = 0; i<strlen(pszPath) && !bSpaces; i++)
	{
		if (pszPath[i] == ' ')
			bSpaces = true;
	}

	return bSpaces;
}


//////////////////////////////////////////////////////
/// Muestra los errores encontrados
//////////////////////////////////////////////////////
void CPacker :: PrintErrors ()
{
	std::cout << std::endl << "CRITICAL ERRORS FOUND DURING THE PROCESS!" << std::endl;

	std::cout << std::endl <<  "Errors found:" << std::endl;
	for (std::vector<std::string>::const_iterator i = m_msgError.begin(); i < m_msgError.end(); i++)
		std::cout << "    " << *i << std::endl;
}
```

```
//////////////////////////////////////////////////////
/// Carga archivo con archivos y carpetas excluidas
//////////////////////////////////////////////////////
bool CPacker :: LoadExcludedFile (const char *pszFileName)
{
	bool bOk = true;

	// Abre el archivo
	if (bOk)
	{
		m_ifsIn.open (pszFileName, std::ios::binary);
		bOk = m_ifsIn.is_open();

		if (!bOk)
		{
			m_msgError.push_back("   Can't open <list of excluded F/D> file.\n   " +
				std::string(pszFileName));
			m_iError = ERROR_FILE;
		}
	}

	// Carga el archivo en memoria
	if (bOk)
	{
		int iMode = -1;
		char szLine[256] = "";

		while (!m_ifsIn.eof())
		{
			// Obtenemos linea
			memset(szLine, 0, 256);
			m_ifsIn.getline(szLine, 256);

			// Corregimos / por '\' y eliminamos los saltos de linea o retornos de carro para
			// evitar errores en comprobaciones
			for (unsigned int i = 0; i<strlen(szLine); i++)
			{
				if (szLine[i] == '/')
					szLine[i] = '\\';
				else if (szLine[i] == '\r' || szLine[i] == '\n')
					szLine[i] = '\0';
			}


			// Elegimos modo
			if (!strcmp (szLine, "#FOLDERS_NAME"))
				iMode = EX_FOLDER;
			else if (!strcmp (szLine, "#FOLDERS_WITH_PATH"))
				iMode = EX_FOLDER_WITH_PATH;
			else if (!strcmp (szLine, "#FILES_NAME"))
				iMode = EX_FILENAME;
			else if (!strcmp (szLine, "#FILES_WITH_PATH"))
				iMode = EX_FILE_WITH_PATH;
			else if (!strcmp (szLine, "#PART_OF_FILE_NAME"))
				iMode = EX_PART_FILENAME;
			else if (!strcmp (szLine, "#PART_OF_FILE_NAME_WITH_PATH"))
				iMode = EX_PART_FILE_WITH_PATH;
			else if (!strcmp (szLine, "#EXTENSIONS"))
				iMode = EX_EXTENSION;
```

```
                        // Guardamos
                        else if (strcmp (szLine, ""))
                        {
                                if (iMode == EX_FOLDER)
                                        m_excludedFolder.push_back(szLine);
                                else if (iMode == EX_FOLDER_WITH_PATH)
                                        m_excludedFolderWithPath.push_back(szLine);
                                else if (iMode == EX_FILENAME)
                                        m_excludedFilename.push_back(szLine);
                                else if (iMode == EX_FILE_WITH_PATH)
                                        m_excludedFileWithPath.push_back(szLine);
                                else if (iMode == EX_PART_FILENAME)
                                        m_excludedPartFilename.push_back(szLine);
                                else if (iMode == EX_PART_FILE_WITH_PATH)
                                        m_excludedPartFileWithPath.push_back(szLine);
                                else if (iMode == EX_EXTENSION)
                                        m_excludedExtension.push_back(szLine);
                        }
                }

                m_ifsIn.close();
        }

        return bOk;
}

//////////////////////////////////////////////////////
/// Comprueba si hay hashes repetidos en header
//////////////////////////////////////////////////////
bool CPacker :: ValidateHeaderHash ()
{
        bool bReturn = true;

        for (int i=0; i < m_Header.GetNumItems(); i++)
        {
                for (int j=i; j< m_Header.GetNumItems(); j++)
                {
                        if (i != j && m_Header[i].m_uiHash == m_Header[j].m_uiHash)
                        {
                                char msg[256] = "";
                                sprintf_s (msg, "ERROR: Equal hashes in files %d and %d!", i, j);
                                m_msgError.push_back (msg);
                                memset (msg, 0, 256);

                                sprintf_s (msg, "   File %d: %s (%u).", i, m_Header[i].m_pszName,
                                        m_Header[i].m_uiHash);
                                m_msgError.push_back (msg);
                                memset (msg, 0, 256);

                                sprintf_s (msg, "   File %d: %s (%u).", j, m_Header[j].m_pszName,
                                        m_Header[j].m_uiHash);
                                m_msgError.push_back (msg);

                                m_iError = ERROR_MEMORY;

                                bReturn = false;
                        }
                }
        }

        return bReturn;
}
```

```
//////////////////////////////////////////////////////
/// Comprueba si un archivo esta dentro de una carpeta o sus subcarpetas
//////////////////////////////////////////////////////
bool CPacker :: FolderContainsFile (const char *pszDir, const char *pszFileName)
{
	bool bFound = false;
	bool bOk = true;
	WIN32_FIND_DATA fdFile;
	HANDLE hFind = NULL;
	char szPath[2048];

	// Listamos todos los archivos, sea cual sea su tipo
	sprintf_s (szPath, "%s\\*.*", pszDir);

	// Comprobamos si es un directorio accesible
	if((hFind = FindFirstFile(szPath, &fdFile)) == INVALID_HANDLE_VALUE)
	{
		bOk = false;
	}

	if (bOk)
	{
		do
		{
			// Evita los predefinidos "." y ".."
			if (strcmp(fdFile.cFileName, ".") != 0 && strcmp(fdFile.cFileName, "..") != 0)
			{
				// Contruimos el path completo
				sprintf_s (szPath, "%s\\%s", pszDir, fdFile.cFileName);

				// Si es un directorio, recorrelo con recursividad
				if(fdFile.dwFileAttributes &FILE_ATTRIBUTE_DIRECTORY )
				{
					bFound = FolderContainsFile (szPath, pszFileName);
				}

				// Y si es un archivo, comprueba
				else
				{
					// Sacamos nombre de archivo
					std::string filename = szPath;
					filename = filename.substr(filename.find_last_of("\\")+1);

					if ( !strcmp (pszFileName, filename.c_str()) )
						bFound = true;
				}
			}
		}
		while(FindNextFile(hFind, &fdFile) && !bFound); // Buscamos el siguiente archivo
	}

	// Cerramos lector y salimos
	FindClose(hFind);

	return bFound;
}
```

```
//////////////////////////////////////////////////////
/// Cuenta los archivos que se copiaran
//////////////////////////////////////////////////////
int CPacker :: CountFilesToPack (const char *pszDir)
{
	WIN32_FIND_DATA fdFile;
	HANDLE hFind = NULL;
	char szPath[2048];
	bool bOk = true;

	int iCount = 0;

	// Listamos todos los archivos, sea cual sea su tipo
	sprintf_s (szPath, "%s\\*.*", pszDir);

	// Comprobamos si es un directorio accesible
	 if((hFind = FindFirstFile(szPath, &fdFile)) == INVALID_HANDLE_VALUE)
		bOk = false;

	if (bOk)
	{
		do
		{
			// Evita los predefinidos "." y ".."
			if (strcmp(fdFile.cFileName, ".") != 0 && strcmp(fdFile.cFileName, "..") != 0)
			{
				// Contruimos el path completo
				sprintf_s (szPath, "%s\\%s", pszDir, fdFile.cFileName);

				// Si es un directorio, recorrelo con recursividad
				// (Si no esta excluido y no tiene espacios)
				if(fdFile.dwFileAttributes &FILE_ATTRIBUTE_DIRECTORY )
				{
					if (!HasSpaces(szPath) && !IsExcluded(szPath, true))
						iCount += CountFilesToPack (szPath);
				}

				// si es un archivo
				else
				{
					// Si no esta excluido, suma
					if (!IsExcluded(szPath, false) )
						iCount++;
				}
			}
		}
		while(FindNextFile(hFind, &fdFile) && bOk); // Buscamos el siguiente archivo
	}

	// Cerramos lector y salimos
	FindClose(hFind);

	return iCount;
}
```

MAIN.CPP

```
#include <iostream>
#include "Packer.h"

bool IsValid(int argc, char *argv[]);
void Normalize (char *path);


int main(int argc, char *argv[])
{
	CPacker packer;
	char szInput[256] = "";
	char szOutput[256] = "";
	char szExcluded[256] = "";
	bool bRecreate = false;
	bool bOk = true;

	// Obtenemos datos de entrada
	if ( IsValid(argc, argv) )
	{
		for (int i=1; i<argc; i++)
		{
			if (!strcmp(argv[i], "-i")) // Especifica ruta de entrada
			{
				i++;
				Normalize(argv[i]);
				strcpy_s(szInput, argv[i]);
			}
			else if (!strcmp(argv[i], "-o")) // Especifica ruta de salida
			{
				i++;
				Normalize(argv[i]);
				strcpy_s(szOutput, argv[i]);
			}
			else if (!strcmp(argv[i], "-e")) // Especifica archivo con archivos a exlucir
			{
				i++;
				Normalize(argv[i]);
				strcpy_s(szExcluded, argv[i]);
			}
			else if (!strcmp(argv[i], "-r")) // Especifica si deben replicarse los excluidos
			{
				bRecreate = true;
			}
		}
	}
	else
	{
		std::cout << "USAGE: Packer.exe" << std::endl << "   -i <directory to pack>"
		<< std::endl << "   -o <package output directory>" << std::endl
		<< "   [-e <list of excluded files/directories file> [-r] ]" << std::endl;
		std::cout << std::endl << "* -r: Recreate excluded files into -o folder." << std::endl
		<< "      CAUTION! All data of -o folder will be DELETED!" << std::endl;
		bOk = false;
	}
```

```
	if (bOk)
		bOk = packer.Init(szInput, szOutput, szExcluded, bRecreate);

	if (bOk)
		bOk = packer.Start();

	int iError = packer.GetLastsError();

	if (bOk)
		packer.End();

	std::cout << std::endl;
	if (iError)
	{
		system("PAUSE");
	}

	return iError;

}

// Comprueba si los argumentos recibidos son validos
bool IsValid(int argc, char *argv[])
{
	bool bOk = true;

	if (argc < 5 || argc > 8)
		bOk = false;

	else
	{
		bool bIn = false;
		bool bOut = false;
		bool bExclude = false;
		bool bRecreate = false;

		for (int i=1; i<argc; i++)
		{
			if (!strcmp(argv[i], "-i"))
				bIn = true;
			else if (!strcmp(argv[i], "-o"))
				bOut = true;
			else if (argc > 5 && !strcmp(argv[i], "-e"))
				bExclude = true;
			else if (argc > 7 && !strcmp(argv[i], "-r"))
				bRecreate = true;
		}
		bOk = bIn & bOut;
		if (argc > 5)
			bOk = bOk & bExclude;
		if (argc > 7)
			bOk = bOk & bRecreate;
	}

	return bOk;
}
```

```
// Normaliza las rutas
void Normalize (char *path)
{
	for (unsigned int i = 0; i< strlen(path); i++)
	{
		if (path[i] == '/')
			path[i] = '\\';
		else if (path[i] == '\r' || path[i] == '\n')
			path[i] = '\0';
	}
}
```

ANEXO B

Desempaquetador

UNPACKER.H

```
#ifndef _UNPACKER_H_
#define _UNPACKER_H_

#include <fstream>
#include <vector>
#include <sstream>
#include <algorithm>
#include <iostream>
#include "CoreUtils.h"
#include "FileSystem.h"


// CUIDADO: Estructura compartida con empaquetador
typedef struct
{
	int		m_iInit;		// Punto de inicio dentro del fichero
	int		m_iSize;		// Tamanyo del fichero
	unsigned int	m_uiHash;		// Hash
	int		m_iNameLen;		// Longitud de nombre de archivo
	char *		m_pszName;		// Nombre de archivo
}
THeader;


class CUnpacker
{
public:
	CUnpacker	() : m_bInit (false) {}
	~CUnpacker	(){ CUnpacker::End(); }

	bool		Init		(const char *pszInputFileName);
	bool		IsOk		()				const	{ return m_bInit; }
	virtual void	End		();

	char*		GetFile		(const char* pszFileName);
	THeader		GetInfo		(const char* pszFileName);

	bool		FileExists	(const char* pszFileName);
	int		Search		(unsigned int uiInit, unsigned int uiEnd, const unsigned int &uiHash);

private:
	bool			m_bInit;
	std::ifstream		m_ifsIn;		// Stream de Archivo de datos
	int			m_iFileCount;		// Cantidad de archivos empaquetados
	int			m_iDataBegin;		// Punto de inicio de los archivos dentro del fichero
	std::vector<THeader>	m_header;		// Informacion de los archivos empaquetados

	void		LoadHeader	();
};

#endif
```

UNPACKER.CPP

```
#include "Unpacker.h"


////////////////////////////////////////////////////
///  Inicializacion
////////////////////////////////////////////////////
bool CUnpacker::Init (const char *pszInputFileName)
{
	bool bOk = true;

	End();

	// Abrimos archivos de entrada
	m_ifsIn.open (pszInputFileName, std::ios::binary);
	bOk = m_ifsIn.is_open();

	if (bOk)
	{
		// Punto de inicio de datos dentro del archivo (sumamos int de cantidad de archivos)
		m_iDataBegin = sizeof(int);

		// Leemos cantidad de archivos
		m_iFileCount = 0;
		m_ifsIn.read(reinterpret_cast<char *>(&m_iFileCount), sizeof(int));

		// Cargamos índice en memoria
		LoadHeader ();
	}

	m_bInit = true;
	if ( !bOk )
	{
		NE_ASSERT (bOk && "ERROR: Uninitialized unpacker\n");
		CUnpacker::End();
	}

	return bOk;
}

////////////////////////////////////////////////////
/// Fin
////////////////////////////////////////////////////
void CUnpacker :: End()
{
	if ( IsOk() )
	{
		// Cerramos archivo
		if (m_ifsIn.is_open())
			m_ifsIn.close();

		// Eliminamos punteros de nombres de archivos
		for (std::vector<THeader>::iterator i = m_header.begin(); i < m_header.end(); i++)
			SAFE_DELETE_ARRAY ((*i).m_pszName);

		// Eliminamos resto de datos de archivos
		m_header.clear();

		m_bInit = false;
	}
}
```

```
//////////////////////////////////////////////////////
/// Carga el header en memoria
//////////////////////////////////////////////////////x
void CUnpacker :: LoadHeader ()
{
	THeader tHeader;

	for (int i=0; i<m_iFileCount; i++)
	{
		// Resetea variable
		tHeader.m_iInit = 0;
		tHeader.m_iSize = 0;
		tHeader.m_uiHash = 0;
		tHeader.m_iNameLen = 0;
		tHeader.m_pszName = NULL;

		// Leemos informacion de archivo
		m_ifsIn.read (reinterpret_cast<char *>(&tHeader.m_iInit), sizeof(int));
		m_ifsIn.read (reinterpret_cast<char *>(&tHeader.m_iSize), sizeof(int));
		m_ifsIn.read (reinterpret_cast<char *>(&tHeader.m_uiHash), sizeof(unsigned int));
		m_ifsIn.read (reinterpret_cast<char *>(&tHeader.m_iNameLen), sizeof(int));
		tHeader.m_pszName = new char[tHeader.m_iNameLen];
		m_ifsIn.read (reinterpret_cast<char *>(tHeader.m_pszName), tHeader.m_iNameLen);

		// Sumamos tamanyos para calcular el punto de inicio de los datos
		m_iDataBegin += sizeof(int) * 3 + sizeof(unsigned int) + tHeader.m_iNameLen;

		// Guarda en vector
		m_header.push_back(tHeader);
	}
}


//////////////////////////////////////////////////////
/// Busca y recupera la informacion de un archivo
//////////////////////////////////////////////////////
THeader CUnpacker :: GetInfo (const char* pszFileName)
{
	unsigned int uiHash = ne::core::neHash (pszFileName);
	THeader tReturn;
	tReturn.m_iInit = -1;

	int iPos = Search (0, m_header.size()-1, uiHash);

	if (iPos != -1)
	{
		tReturn.m_iInit = m_header[iPos].m_iInit;
		tReturn.m_iNameLen = m_header[iPos].m_iNameLen;
		tReturn.m_iSize = m_header[iPos].m_iSize;
		tReturn.m_uiHash = m_header[iPos].m_uiHash;
		tReturn.m_pszName = m_header[iPos].m_pszName;
	}

	NE_ASSERT ( iPos > -1 && "Unpacker: Can't found packed file!");

	return tReturn;
}
```

```
//////////////////////////////////////////////////////
/// devuelve puntero a los datos del archivo
//////////////////////////////////////////////////////
char* CUnpacker :: GetFile (const char* pszFileName)
{
	// Obtenemos datos del archivo
	THeader tHeader = GetInfo (pszFileName);

	// Reservamos memoria
	char* pszBuffer = new char[tHeader.m_iSize];

	// Nos colocamos en la posicion inicial del archivo
	m_ifsIn.seekg (m_iDataBegin + tHeader.m_iInit, std::ios::beg);

	// copiamos
	m_ifsIn.read (pszBuffer, tHeader.m_iSize);

	// devolvemos size
	return pszBuffer;
}

//////////////////////////////////////////////////////
/// Indica si un archivo esta en el paquete o no
//////////////////////////////////////////////////////
bool CUnpacker :: FileExists (const char* pszFileName)
{
	bool bFounded = false;
	unsigned int uiHash = ne::core::neHash(pszFileName);

	if (Search (0, m_header.size()-1, uiHash) > -1)
		bFounded = true;

	return bFounded;
}

//////////////////////////////////////////////////////
/// Busqueda por hash mediante 'Divide y venceras'
//////////////////////////////////////////////////////
int CUnpacker :: Search (unsigned int uiInit, unsigned int uiEnd, const unsigned int &uiHash)
{
	int iReturn = -1;
	int iPos = (uiInit + uiEnd) / 2;

	if (m_header[iPos].m_uiHash == uiHash)
		iReturn = iPos;

	else if (uiEnd - uiInit == 1 || uiInit - uiEnd == 1)
	{
		if (m_header[uiInit].m_uiHash == uiHash)
			iReturn = uiInit;
		else if (m_header[uiEnd].m_uiHash == uiHash)
			iReturn = uiEnd;
	}

	else if (uiInit != uiEnd)
	{
		if (uiHash < m_header[iPos].m_uiHash)
			iReturn = Search (uiInit, iPos-1, uiHash);
		else if (uiHash > m_header[iPos].m_uiHash)
			iReturn = Search (iPos+1, uiEnd, uiHash);
	}

	return iReturn;
}
```

ANEXO C

FileManager

FILEMANAGER.H

```
#ifndef _FILEMANAGER_H_
#define _FILEMANAGER_H_

#include <fstream>
#include <vector>
#include <sstream>
#include <algorithm>
#include <iostream>
#include "File.h"
#include "CoreUtils.h"
#include "FileUtils.h"

#define RESOURCES_FILE              "data.dat"

namespace ne
{

namespace core
{

enum EResourceType
{
        NE_RES_TEXTURE,
        NE_RES_SCENE,
        NE_RES_GUI,
        NE_RES_SPRITE,
        NE_RES_FONT,
        NE_RES_AUDIO,
        NE_RES_INPUT,
        NE_RES_MODEL,
        NE_RES_TEXT,
        NE_RES_SEQUENCE,
        NE_RES_PARTICLE,
        NE_RES_ROOT,
        NE_RES_SAVEDATA,
        NE_RES_MAX,
        NE_RES_FULLPATH
};

enum EFileStorageType
{
        NE_STORAGE_PACKED_PLATFORM,
        NE_STORAGE_PACKED_COMMON,
        NE_STORAGE_PACKED,
        NE_STORAGE_HDRIVE_PLATFORM,
        NE_STORAGE_HDRIVE_COMMON,
        NE_STORAGE_HDRIVE
};


// CUIDADO: Estructura compartida con empaquetador
typedef struct
{
        int             m_iInit;                // Punto de inicio dentro del fichero
        int             m_iSize;                // Tamanyo del fichero
        unsigned int    m_uiHash;               // Hash
        int             m_iNameLen;             // Longitud de nombre de archivo
        char *          m_pszName;              // Nombre de archivo
}
THeader;
```

```
class CFileManager
{
public:
	CFileManager		() : m_bInit (false) {}
	virtual ~CFileManager()	{ CFileManager::End(); }

	virtual bool		Init	(const char *pszInputFileName);
	bool			IsOk	()				const	{ return m_bInit; }
	virtual void		End	();

	virtual CFile*		GetFile		(const char* szFile, EResourceType eResType,
						EFileType eFileType = NE_FILE_READ_BINARY,
						bool bLoadInMemory = false);
	virtual void		SetDir		(const char* szDir, EResourceType eResType);
	virtual const char*	GetDir		(EResourceType eResType);
	virtual bool		FileExists	(const char* pszFileName,
						EResourceType eResType = NE_RES_FULLPATH,
						int *piType = NULL);

	virtual const char*	GetResourceFullPath	(const char* pszFileName, EResourceType eResType);
	virtual const char*	GetPlatformFullPath	(const char* pszFileName, EResourceType eResType);
	virtual const char*	GetCommonFullPath	(const char* pszFileName, EResourceType eResType);

private:
	bool		m_bInit;
	bool		m_bFileData;		// Indica si hay paquete
	std::ifstream	m_ifsIn;		// Stream de Archivo de datos
	int		m_iFileCount;		// Cantidad de archivos empaquetados
	int		m_iDataBegin;		// Punto de inicio de los archivos dentro del fichero

	std::vector<THeader>		m_header;	// Informacion de los archivos empaquetados
	SizedVector<std::string>	m_paths;	// Carpetas de los diversos recursos

	void		LoadHeader		();
	char*		GetData			(const char* pszFileName);
	THeader		GetInfo			(const char* pszFileName);
	int		Search			(unsigned int uiInit, unsigned int uiEnd, const unsigned int &uiHash);
	bool		HardDriveFileExists	(const char* pszFileName);
};


}

}

#endif
```

FILEMANAGER.CPP

```
#include "FileManager.h"
#include <string>

namespace ne
{

namespace core
{

//////////////////////////////////////////////////////
/// Inicializacion
//////////////////////////////////////////////////////
bool CFileManager::Init (const char *pszInputFileName)
{
	bool bOk = true;

	End();

	// Abrimos paquete
	m_ifsIn.open (pszInputFileName, std::ios::binary);
	bOk = m_ifsIn.is_open();

	if (bOk)
	{
		// Punto de inicio de datos dentro del archivo (sumamos int de cantidad de archivos)
		m_iDataBegin = sizeof(int);

		// Leemos cantidad de archivos
		m_iFileCount = 0;
		m_ifsIn.read(reinterpret_cast<char *>(&m_iFileCount), sizeof(int));

		// Cargamos indice en memoria
		LoadHeader ();

		m_bFileData = true;
	}

	// En caso de no encontrar el archivo .dat
	else
	{
		m_iDataBegin = 0;
		m_iFileCount = 0;
		m_bFileData = false;
		bOk = true;
	}

	// Inicializamos rutas de recursos
	// (CUIDADO! EL ORDEN DEBE COINCIDIR CON EL ENUMERADOR EResourceType)
	m_paths.Init(NE_RES_MAX);
	std::string szPathTextures = "Textures/";
	m_paths.AddItem (&szPathTextures);
	std::string szPathScenes = "Scenes/";
	m_paths.AddItem (&szPathScenes);
	std::string szPathGUI = "GUI/";
	m_paths.AddItem (&szPathGUI);
	std::string szPathSprites = "Sprites/";
	m_paths.AddItem (&szPathSprites);
	std::string szPathFonts = "Fonts/";
	m_paths.AddItem (&szPathFonts);
	std::string szPathAudio = "Audio/";
	m_paths.AddItem (&szPathAudio);
```

```
		std::string szPathVideo = "Video/";
		m_paths.AddItem (&szPathVideo);
		std::string szPathInput = "Input/";
		m_paths.AddItem (&szPathInput);
		std::string szPathModels = "Models/";
		m_paths.AddItem (&szPathModels);
		std::string szPathText = "Text/";
		m_paths.AddItem (&szPathText);
		std::string szPathSequences = "Sequences/";
		m_paths.AddItem (&szPathSequences);
		std::string szPathParticles = "Particles/";
		m_paths.AddItem (&szPathParticles);
		std::string szPathRoot = "";
		m_paths.AddItem (&szPathRoot);				// Root
		std::string szPathSaveGame = "";
		m_paths.AddItem (&szPathSaveGame);			// SaveGame

		m_bInit = true;
		if ( !bOk )
		{
			CFileManager::End();
		}

		return bOk;
}



////////////////////////////////////////////////////
/// Fin
////////////////////////////////////////////////////
void CFileManager :: End()
{
		if ( IsOk() )
		{
			// Cerramos paquete
			if (m_ifsIn.is_open())
				m_ifsIn.close();

			// Eliminamos punteros de nombres de archivos
			for (std::vector<THeader>::iterator i = m_header.begin(); i < m_header.end(); i++)
				SAFE_DELETE_ARRAY ((*i).m_pszName);

			// Eliminamos resto de datos de archivos
			m_header.clear();
			m_paths.Clear();

			m_bInit = false;
		}
}
```

```
//////////////////////////////////////////////////////
/// devuelve CFile*
//////////////////////////////////////////////////////
CFile* CFileManager :: GetFile (const char* szFile, EResourceType eResType, EFileType eFileType,
bool bLoadInMemory)
{
	int iFileType = -1;
	CFile *pFile = new CFile();
	std::string szPath = "";

	// Saca path completo
	if (eResType == NE_RES_FULLPATH)
		szPath = szFile;
	else
		szPath = GetResourceFullPath(szFile, eResType);

	// Comprobamos si existe
	if (FileExists (szPath.c_str(), NE_RES_FULLPATH, &iFileType))
	{
		// Si lo tenemos en el paquete y no es para escritura
		if (eFileType != NE_FILE_WRITE && eFileType != NE_FILE_WRITE_BINARY
		&& iFileType < NE_STORAGE_PACKED )
			pFile->Init(szPath.c_str(), GetData(szPath.c_str()), GetInfo(szPath.c_str()).m_iSize);
	}

	// Si no existe, o se trata de un archivo de disco
	if (!pFile->IsOk())
		pFile->Init(szPath.c_str(), eFileType, bLoadInMemory);

	// Comprobamos inicializacion
	if (!pFile->IsOk())
		SAFE_DELETE(pFile);

	NE_ASSERT_MSG (pFile != NULL, "ERR: couldn't open file %s\n", szPath.c_str());

	return pFile;
}


//////////////////////////////////////////////////////
/// Establece la carpeta donde se encuentra el recurso
//////////////////////////////////////////////////////
void CFileManager :: SetDir (const char* szDir, EResourceType eResType)
{
	m_paths[eResType] = szDir;

	// Comprobacion de / final
	if(szDir[strlen(szDir)-1] != '/' && szDir[strlen(szDir)-1] != '\\' )
		m_paths[eResType] += "\\";
}


//////////////////////////////////////////////////////
/// Devuelve la carpeta establecida para el recurso
//////////////////////////////////////////////////////
const char* CFileManager :: GetDir (EResourceType eResType)
{
	return m_paths[eResType].c_str();
}
```

```
//////////////////////////////////////////////////////
/// Indica si un archivo es accesible
//////////////////////////////////////////////////////
bool CFileManager :: FileExists (const char* pszFileName, EResourceType eResType, int *piType)
{
	bool bFound = false;
	unsigned int uiHash = 0;

	const char* szPath = NULL;

	if (eResType != NE_RES_FULLPATH)
	{
		// Plataforma-Paquete
		szPath = GetPlatformFullPath (pszFileName, eResType);
		uiHash = neHash (szPath);

		if (Search (0, m_header.size()-1, uiHash) > -1)
		{
			bFound = true;
			if (piType != NULL)
				*piType = NE_STORAGE_PACKED_PLATFORM;
		}

		// Plataforma-Disco
		if (!bFound)
		{
			bFound = HardDriveFileExists(szPath);
			if (bFound && piType != NULL)
				*piType = NE_STORAGE_HDRIVE_PLATFORM;
		}

		// Common-Paquete
		if (!bFound)
		{
			SAFE_DELETE_ARRAY(szPath);
			szPath = GetCommonFullPath (pszFileName, eResType);
			uiHash = neHash(szPath);

			if (Search (0, m_header.size()-1, uiHash) > -1)
			{
				bFound = true;
				if (piType != NULL)
					*piType = NE_STORAGE_PACKED_COMMON;
			}
		}

		// Common-Disco
		if (!bFound)
		{
			bFound = HardDriveFileExists(szPath);
			if (bFound && piType != NULL)
				*piType = NE_STORAGE_HDRIVE_COMMON;
		}
	}
```

```
	else
	{
		// Paquete
		std::string szType = pszFileName;
		uiHash = neHash(pszFileName);

		if (Search (0, m_header.size()-1, uiHash) > -1)
		{
			bFound = true;

			int iPosition = szType.find("Common/");
			if (piType != NULL && iPosition >= 0)
				*piType = NE_STORAGE_PACKED_COMMON;
			else if (piType != NULL)
				*piType = NE_STORAGE_PACKED_PLATFORM;
		}

		// Disco
		if (!bFound)
		{
			bFound = HardDriveFileExists(pszFileName);
			if (bFound)
			{
				int iPosition = szType.find("Common/");
				if (piType != NULL && iPosition >= 0)
					*piType = NE_STORAGE_HDRIVE_COMMON;
				else if (piType != NULL)
					*piType = NE_STORAGE_HDRIVE_PLATFORM;
			}
		}
	}

	SAFE_DELETE_ARRAY(szPath);

	return bFound;
}


//////////////////////////////////////////////////////
/// Devuelve la ruta completa del Recurso, detectando si es common o platform
//////////////////////////////////////////////////////
const char* CFileManager :: GetResourceFullPath (const char* pszFileName, EResourceType eResType)
{
	const char* pszFullPath = NULL;
	int iType = -1;

	if (FileExists(pszFileName, eResType, &iType))
	{
		if (iType == NE_STORAGE_PACKED_COMMON
		|| iType == NE_STORAGE_HDRIVE_COMMON)
			pszFullPath = GetCommonFullPath (pszFileName, eResType);
		else
			pszFullPath = GetPlatformFullPath (pszFileName, eResType);
	}

	NE_ASSERT_MSG(pszFullPath != NULL, "ERR: File not found: %s\n", pszFileName);

	return pszFullPath;
}
```

```
//////////////////////////////////////////////////////
/// Carga el header en memoria
//////////////////////////////////////////////////////
void CFileManager :: LoadHeader ()
{
	THeader tHeader;

	for (int i=0; i<m_iFileCount; i++)
	{
		// Resetea variable
		tHeader.m_iInit = 0;
		tHeader.m_iSize = 0;
		tHeader.m_uiHash = 0;
		tHeader.m_iNameLen = 0;
		tHeader.m_pszName = NULL;

		// Leemos informacion de archivo
		m_ifsIn.read (reinterpret_cast<char *>(&tHeader.m_iInit), sizeof(int));
		m_ifsIn.read (reinterpret_cast<char *>(&tHeader.m_iSize), sizeof(int));
		m_ifsIn.read (reinterpret_cast<char *>(&tHeader.m_uiHash), sizeof(unsigned int));
		m_ifsIn.read (reinterpret_cast<char *>(&tHeader.m_iNameLen), sizeof(int));
		tHeader.m_pszName = new char[tHeader.m_iNameLen];
		m_ifsIn.read (reinterpret_cast<char *>(tHeader.m_pszName), tHeader.m_iNameLen);

		// Sumamos tamanyos para calcular el punto de inicio de los datos
		m_iDataBegin += sizeof(int) * 3 + sizeof(unsigned int) + tHeader.m_iNameLen;

		// Guarda en vector
		m_header.push_back(tHeader);
	}
}



//////////////////////////////////////////////////////
/// devuelve puntero a los datos del archivo
//////////////////////////////////////////////////////
char* CFileManager :: GetData (const char* pszFileName)
{
	// Obtenemos datos del archivo
	THeader tHeader = GetInfo (pszFileName);

	// Reservamos memoria
	char* pszBuffer = new char[tHeader.m_iSize];

	// Nos colocamos en la posicion inicial del archivo
	m_ifsIn.seekg (m_iDataBegin + tHeader.m_iInit, std::ios::beg);

	// copiamos
	m_ifsIn.read (pszBuffer, tHeader.m_iSize);

	// devolvemos size
	return pszBuffer;
}
```

```
//////////////////////////////////////////////////////
/// Busca y recupera la informacion de un archivo
//////////////////////////////////////////////////////
THeader CFileManager :: GetInfo (const char* pszFileName)
{
	unsigned int uiHash = neHash(pszFileName);
	THeader tReturn;
	tReturn.m_iInit = -1;

	int iPos = Search (0, m_header.size()-1, uiHash);

	if (iPos != -1)
	{
		tReturn.m_iInit = m_header[iPos].m_iInit;
		tReturn.m_iNameLen = m_header[iPos].m_iNameLen;
		tReturn.m_iSize = m_header[iPos].m_iSize;
		tReturn.m_uiHash = m_header[iPos].m_uiHash;
		tReturn.m_pszName = m_header[iPos].m_pszName;
	}

	NE_ASSERT_MSG(iPos > -1, "ERR: Can't found packed file!: %s\n", pszFileName);

	return tReturn;
}



//////////////////////////////////////////////////////
/// Comprueba si existe un fichero en disco duro
//////////////////////////////////////////////////////
bool CFileManager :: HardDriveFileExists (const char* pszFileName)
{
	bool bExists = false;
#if defined ANDROID
	AAsset* assetFile = AAssetManager_open(g_pAssetManager, pszFileName,
AASSET_MODE_BUFFER);
	if (assetFile)
	{
		AAsset_close(assetFile);
		bExists = true;
	}
#endif

	if (!bExists)
	{
		FILE* fp = fopen(pszFileName, "r");
		bExists = fp != NULL;

		if (bExists)
		{
			fclose(fp);
			fp = NULL;
		}
	}

	return bExists;
}
```

```
//////////////////////////////////////////////////////
/// Devuelve la ruta completa de la plataforma
//////////////////////////////////////////////////////
const char* CFileManager :: GetPlatformFullPath (const char* pszFileName, EResourceType eResType)
{
	char* pszFullPath = new char[256];
	strcpy(pszFullPath, "");

#if defined SN_TARGET_PSP_HW || defined SN_TARGET_PSP_PRX
	strcpy (szFullPath, "host0:");
#endif

	if (eResType == NE_RES_FULLPATH)
		strcat (pszFullPath, pszFileName);
	else
	{
		strcat (pszFullPath, ne::core::GetPlatformPath());
		strcat (pszFullPath, GetDir(eResType));
		strcat (pszFullPath, pszFileName);
	}

	int iSize = strlen(pszFullPath);
	NE_ASSERT(256 >= iSize);

	return pszFullPath;
}



//////////////////////////////////////////////////////
/// Devuelve la ruta completa del Common
//////////////////////////////////////////////////////
const char* CFileManager :: GetCommonFullPath (const char* pszFileName, EResourceType eResType)
{
	char* pszFullPath = new char[256];
	strcpy(pszFullPath, "");

#if defined SN_TARGET_PSP_HW || defined SN_TARGET_PSP_PRX
	strcpy (szFullPath, "host0:");
#endif

	if (eResType == NE_RES_FULLPATH)
		strcat (pszFullPath, pszFileName);
	else
	{
		strcat (pszFullPath, "Common/");
		strcat (pszFullPath, GetDir(eResType));
		strcat (pszFullPath, pszFileName);
	}

	int iSize = strlen(pszFullPath);
	NE_ASSERT(256 >= iSize);

	return pszFullPath;
}
```

```
//////////////////////////////////////////////////////
/// Busqueda por hash mediante 'Divide y venceras'
//////////////////////////////////////////////////////
int CFileManager :: Search (unsigned int uiInit, unsigned int uiEnd, const unsigned int &uiHash)
{
	int iReturn = -1;

	if (m_bFileData)
	{
		int iPos = (uiInit + uiEnd) / 2;

		if (m_header[iPos].m_uiHash == uiHash)
		iReturn = iPos;

		else if (uiEnd - uiInit == 1 || uiInit - uiEnd == 1)
		{
			if (m_header[uiInit].m_uiHash == uiHash)
				iReturn = uiInit;
			else if (m_header[uiEnd].m_uiHash == uiHash)
				iReturn = uiEnd;
		}

		else if (uiInit != uiEnd)
		{
			if (uiHash < m_header[iPos].m_uiHash)
				iReturn = Search (uiInit, iPos-1, uiHash);
			else if (uiHash > m_header[iPos].m_uiHash)
				iReturn = Search (iPos+1, uiEnd, uiHash);
		}
	}

	return iReturn;
}




}

}
```

ANEXO D

Funciones UTF-8

```
// Devuelve el tamaño real de una cadena UTF8
unsigned int GetStringSizeUTF8 ( const char* pszString )
{
    bool bExit = false;
    unsigned int uiLenRet = 0;
    unsigned int uiLen = strlen(pszString);
    unsigned int uiSize = (sizeof(char) * 8) - 1;
    unsigned int i = 0;

    while (i < uiLen)
    {
        int iBytes = 0;
        bExit = false;

        // Cuenta el numero de unos de la cabecera
        for (unsigned int j=0; !bExit && j <= uiSize; ++j)
        {
            unsigned char cChar = pszString[i];
            bool bBit = (cChar >> (uiSize - j)) & 0x1;

            if (bBit)
                iBytes++;
            else
            {
                if (iBytes == 0)
                {
                        iBytes = 1;
                }
                else
                {
                        NE_ASSERT_MSG(iBytes > 1, "ERR: UTF8 error counting number of bytes.\n");
                }
                bExit = true;
            }
        }

        // Suma
        i += iBytes;
        ++uiLenRet;
    }

    return uiLenRet;
}
```

```
// Devuelve la posición real dentro de una cadena UTF8
unsigned int GetUTF8Position (const char* pszString, unsigned int uiPos)
{
    unsigned int uiPosRet = 0;
    bool bExit = false;
    bool bFound = false;
    unsigned int uiLen = strlen(pszString);
    unsigned int uiSize = (sizeof(char) * 8) - 1;

    // Comprueba tamanyo
    if (uiPos >= GetStringSizeUTF8 (pszString))
    {
        bFound = true;
        NE_ASSERT_MSG (false, "ERR: Index out of bounds.\n");
    }

    // Busca
    unsigned int i = 0;
    while (!bFound && i < uiLen)
    {
        // Comprueba si ha encontrado la posicion y, si no, suma
        if (uiPosRet == uiPos)
        {
            uiPosRet = i;
            bFound = true;
        }

        if (!bFound)
        {
            int iBytes = 0;
            bExit = false;

            // Cuenta el numero de unos de la cabecera
            for (unsigned int j=0; !bExit && j <= uiSize; ++j)
            {
                bool bBit = (pszString[i] >> (uiSize - j)) & 0x1;

                if (bBit)
                    iBytes++;
                else
                {
                    if (iBytes == 0)
                        iBytes = 1;
                    else
                        NE_ASSERT_MSG(iBytes > 1, "ERR: UTF8 error counting number of bytes.\n");

                    bExit = true;
                }
            }

            // Suma
            i += iBytes;
            uiPosRet ++;
        }
    }
    // Si no se encuentra, manda codigo de error
    if (!bFound)
        uiPosRet = 0xFFFF;
    NE_ASSERT_MSG (bFound, "ERR: UTF8 position not found.\n");

    return uiPosRet;
}
```

```
// Devuelve el código de un carácter
unsigned int GetCodeFromUTF8Char( unsigned int cChar )
{
    unsigned int uiCode = 0;
    bool bOk = true;

    // Comprueba el primer bit del ultimo Byte para saber si es ASCII o UTF-8
    bool bBit7 = (cChar >> 7) & 0x1;

    if (!bBit7)
    {
        // Es ASCII
        uiCode = cChar;
    }

    else
    {
        // ES UTF-8
        unsigned int uiSize = 0;
        unsigned int uiInitPos = 0xFFFF;
        unsigned int uiBytes = 0;
        unsigned int uiOffset = 0;

        // Calculamos el punto de inicio de los bytes
        uiSize = (sizeof(unsigned int) * 8) - 1;

        for (unsigned int i=0; uiInitPos == 0xFFFF && i<=uiSize; ++i)
        {
            bool bBit = (cChar >> (uiSize - i) ) & 0x1;
            if (bBit)
                uiInitPos = i;
        }

        if (uiInitPos != 0 && uiInitPos != 8  && uiInitPos != 16)
        {
            bOk = false;
            NE_ASSERT_MSG (bOk, "ERR: UTF-8 error at bytes init point.\n");
        }

        if (bOk)
        {
            // Calcula el numero de bytes
            uiBytes = sizeof(unsigned int) - (uiInitPos / 8);

            // Calculamos posicion del primer bit que se escribira (num total bytes que tendra el codigo - 1)
            uiOffset = uiBytes * 8 - (uiBytes + 1 + ( (uiBytes-1) * 2)) - 1;
```

```
                // Comprobamos cabeceras mientras copiamos
                uiSize = sizeof(char) * 8;

                for (unsigned int i = uiBytes; bOk && i>0; --i)
                {
                    unsigned int iPos = i*8-1;
                    for (unsigned int j = 0; bOk && j<uiSize; ++j)
                    {
                        bool bBit = (cChar >> (iPos - j) ) & 0x1;

                        // Es cabecera: comprueba
                        if (j == 0)
                        {
                            bool bBit1 = (cChar >> (iPos - 1) ) & 0x1;

                            // Es el primer byte (debe ser 11...0)
                            if (i == uiBytes)
                            {
                                bool bBit2 = (cChar >> (iPos - 2) ) & 0x1;
                                bool bBit3 = (cChar >> (iPos - 3) ) & 0x1;
                                bool bBit4 = (cChar >> (iPos - 4) ) & 0x1;

                                if (uiBytes == 2 && (!bBit || !bBit1 || bBit2) )
                                    bOk = false;
                                else if (uiBytes == 3 && (!bBit || !bBit1 || !bBit2 || bBit3) )
                                    bOk = false;
                                else if (uiBytes == 4 && (!bBit || !bBit1 || !bBit2 || !bBit3 || bBit4) )
                                    bOk = false;

                                // Saltamos cabecera
                                j += uiBytes;
                            }

                            // No es el primer bit (debe ser 10)
                            else
                            {
                                if (!bBit || bBit1)
                                    bOk = false;

                                // Saltamos cabecera
                                ++j;
                            }
                        }

                        // No es cabecera: setea los bits
                        else
                        {
                            if (bBit)
                                uiCode |= (1u << uiOffset);
                            --uiOffset;
                        }
                    }
                }
                NE_ASSERT_MSG (bOk, "ERR: UTF-8 header error.\n");
            }
        }

        if (!bOk)
            uiCode = 0xFFFF;

        return uiCode;
    }
```

```
// Devuelve el código del carácter de la cadena indicado
unsigned int GetCodeFromUTF8String( const char* pszString, unsigned int iNumChar )
{
    unsigned int uiSize = GetStringSizeUTF8 (pszString);
    unsigned int uiConvertedPos = 0;
    unsigned int uiReturn = 0;
    bool bOk = true;
    bool bUTF8 = false;
    char cChar;

    // Sacamos la posicion real en la cadena UTF8 (se comprueba internamente si iNumChar > uiSize)
    uiConvertedPos = GetUTF8Position (pszString, iNumChar);
    bOk = (uiConvertedPos == 0xFFFF) ? false : true;

    // Comprueba si es utf-8 o no
    if (bOk)
    {
        cChar = pszString[uiConvertedPos];
        bUTF8 = (cChar >> 7) & 0x1;
    }

    // Es ASCII
    if (bOk && !bUTF8)
    {
        uiReturn = cChar;
    }

    // Es UTF8
    else if (bOk)
    {
        bool bExit = false;
        int iBytes = 0;

        // Calcula el numero de Bytes del caracter
        uiSize = (sizeof(char) * 8) - 1;
        for (unsigned int i=0; !bExit && i<=uiSize; ++i)
        {
            bool bBit = (cChar >> (uiSize - i)) & 0x1;

            if (bBit)
            {
                iBytes++;
            }
            else
            {
                if (iBytes == 0)
                {
                    iBytes = 1;
                }
                else
                {
                    NE_ASSERT_MSG(iBytes > 1, "ERR: UTF8 error counting number of bytes.\n");
                }
                bExit = true;
            }
        }
```

```
            // Construye unsigned int con los Bytes del caracter
            unsigned int uiDest = uiConvertedPos + iBytes;
            for (unsigned int i = uiConvertedPos; i < uiDest; ++i)
            {
                cChar = pszString[i];

                for (unsigned int j=0; j<=uiSize ; ++j)
                {
                    bool bBit = (cChar >> (uiSize - j)) & 0x1;

                    if (bBit)
                    {
                        unsigned int uiPos = (iBytes - i + uiConvertedPos) * 8 - j - 1;
                        uiReturn |= (1u << uiPos );
                    }
                }
            }

            // Obtenemos codigo limpio de ese caracter
            uiReturn = GetCodeFromUTF8Char (uiReturn);
        }

        if (!bOk)
            uiReturn = 0xFFFF;

        return uiReturn;
    }
```