Design of Zip File System

The basic idea

The inspiration of ZFS comes from a famous id Software game, namely Quake III that uses a virtual file system using zip files. In fact ZFS is compatible with Q3's pk3 files. ZFS also tries to be compatible with the standard C++ I/O library interface, this means that all functions that takes "std::istream &" as a parameter will still work with ZFS own stream object; this makes the integration of ZFS into an existing C++ project very easy (given that this project does not use FILE * or other non standard C++ I/O methods).

ZFS packages are basically zip files. To be more precise it uses the 32 bits zip file format and supports two compressions methods: stored and deflated. This means it's not 100% compliant with the latest zip specifications (i.e. no zip-64 support), but it does support the zip files made by tools such as WinZip, WinRAR or even Windows XP.

The idea of supporting the 'storage' compression method comes from the fact that some files don't compress very well and that the runtime overhead added by the decompression can be a handicap. Already-compressed files (such as mpg, mp3, avi, divx, png, etc) should be simply stored in a package and shouldn't be compressed. Tools such as WinRAR allow conditional compression, which makes the task easier.

How ZFS manages the directories/files

ZFS is a combination of "istream" C++ objects and central file system manager. When you create a zip file system object, you specify the base directory of the virtual file system, the extension used by the packages and you also tell it whether or not you want it to become the default zip file system.

To open a file via a zip file system, you simply use a ZFS "istream" object (much alike "ifstream") and you simply give it the name of the file you want to open. You can also specify a particular zip file system if you don't want to use the default one.

The corresponding zip file system will then look in its base directory to find the specified file. (Note If you specified a directory relative to the process "current directory" then modifying the current directory will also affect any zip file system with relative base directory. If you don't want this behavior, you should then always give the full path to the base directory.)

When looking for a file in the given base directory, it may happen that more than one file have the same path. In that case, the zip file system will open the file with the highest priority. In practice, three rules can sum up this mechanism:

1. If a corresponding file can be found outside the zip packages, it will be the one that's opened. This gives a way to override the files contained in the packages.
2. If there is no corresponding file that can be found outside the packages but that one or more corresponding files are found inside the packages, then the latest encountered file in these packages will be opened.
3. Packages are opened in alphabetic order using the ASCII codes. When a package is opened, all the files it contains override the any previous files with the same paths.

This gives a smooth way to handle the packages and to provide future updates to the data. For example, Point 1 allows adding new files or modifying existing one without creating a new package, which can be useful when developing or modifying the application. Point 2 and 3 allows to provide an update mechanism, it means you can update the files contained in the packages or even add new files simply by providing a new package file without modifying old ones; it's much alike Quake 3 "Pak*n*.pk3" system.

For example, we could have the following packages: !Pak00.zip, !Pak01.zip, !Pak15.zip, Supermod.zip and Data.zip. It can be easily seen how "!Pack15.zip" overrides "!Pak00.zip" and "!Pak01.zip", and that "Supermod.zip" overrides all the other packages. (Note This also means that there can be collisions between some packages. Currently ZFS lets this problem up to the application and to however makes the packages; some discipline may be required when creating a third-party

package, like putting the new files in a separate directory inside the package-)

Compatibility with C++ I/O standard library

ZFS can be seen as an extension to the standard C++ I/O library. In fact, "izfstream" is an "istream" object, so everything that works with "istream" will work with "izfstream". This was made to keep a maximum compatibility with any code using the standard C++ library for I/O. Anyone who has used "istream" or "ifstream" will quickly understand what I mean (see Figure 1).

```
void DoSomething(std::istream & File)
{
    // Output the file via cout (note: rdbuf() method is a std C++ method, not zfs specific)
    std::cout << File.rdbuf() << std::endl;
}



int main(int argc, char * argv[])
{
    using namespace std;
    using zipfilesystem::filesystem;
    using zipfilesystem::izfstream;

    // Create and initialize the Zip File System (basepath, file_extension, makedefault)
    // and output the its status via cout
    filesystem FileSystem("D:/Games/Quake III Arena/baseq3", "pk3", true);
    cout << FileSystem << endl;

    // Try to open a zipped file (Careful! The openmode is always 'ios::in | ios::binary'.)
    izfstream File("productid.txt");

    if (! File)
        cout << "ERROR: Cannot open file!" << endl;

    // Call some function expecting an istream object
    DoSomething(File);

    // The End.
    cout << "\nPress any key to continue." << endl;
    getch();

    return 0;
}
```

Figure 1: a small example program reading "productid.txt" from Quake III Arena packages

The example above create a zip file system object, sets Quake 3 data directory as the base directory, sets "pk3" as the package extension, and makes this zip file system the default one. Then it outputs the current state of the zip file system via cout. Finally it opens the file "productid.txt", verify it was correctly opened and displays it via cout (see Figure 2).

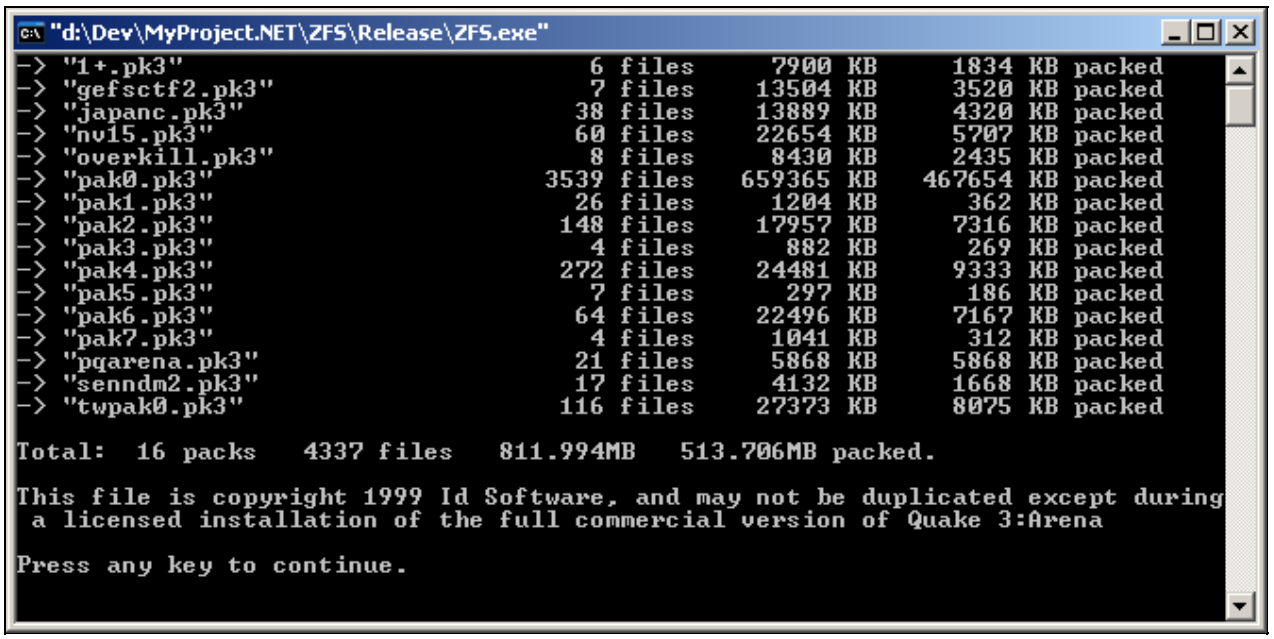

Figure 2: the result of the above example program

Conclusion

For more information on the public methods of zipfilesystem::izfstream and zipfilesystem::filesystem you should look inside "zfsystem.h".
I hope ZFS will help you making your own virtual file system. Now you will have no excuse for providing programs with thousands of tinny files spread in hundreds of directories. ;-)