Assets ZIP Manager with PhysFS

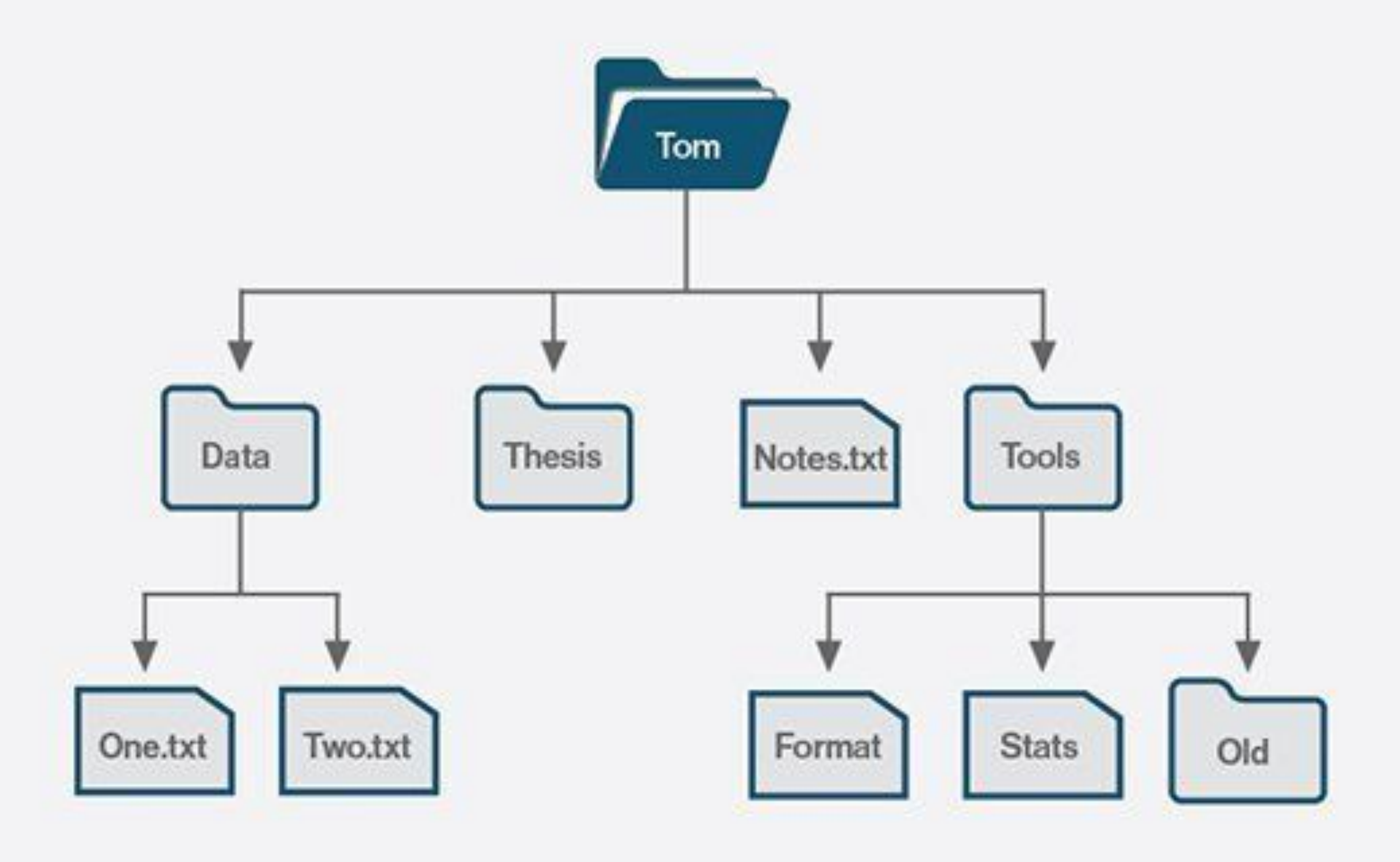
By Yeray Tarifa

Index

1. Introduction
    - 1.1 Filesystems
    - 1.2 Types of Filesystems
    - 1.3 Filesystems API
2. Market Study
    - 2.1 Operating Systems
    - 2.2 Filesystems APIs in Games
3. Selected Approach
    - 3.1 PhysFS
    - 3.2 Building PhysFS
    - 3.3 Implementation
4. Exercises (TODOs)
5. Possible Improvements

1.1 Introduction: Filesystems

- Structure and Logic norms to manage groups of data (files)
- How data is stored and read
- Information in disks, drives or RAM usable for programs
- We will see two types of filesystems but there are many more for different applications
- Structure, logic, properties, security…
- Depending on the device, platform or application


Tom
Data
Thesis
Notes.txt
Tools
One.txt
Two.txt
Format
Stats
Old

1.2 Introduction: Types of Filesystems

- Physical Filesystem
    - Responsible for physical operations of the storage device
    - Manages the computing processes to read and write chunks of data
    - Communicates directly with device drivers or the specific channel used by the device

1.2 Introduction: Types of Filesystems

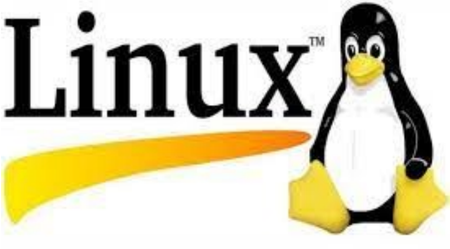
- Virtual Filesystem
  - Works with virtual files requested by the system
  - Manages access to the content of a file and its metadata
  - Still responsible to use enough storage, be efficient and reliable
  - Are not essential for all applications but the physical filesystems are needed

1.3 Filesystem API

- Application Program Interface that is between the user application and the logical filesystem
- Manages file operations that may be required (READ, WRITE, OPEN, CLOSE etc.)
- Used by language specific libraries and user programs to transfer and position data, and manage directories
- Is the medium layer between raw files in system and what the users sees as a result
- Brings security to avoid unwanted accesses

1.3 Filesystem APIs & Operating Systems

- OS are in charge of providing abstraction to access transparently for a proper functioning
- Microsoft Windows uses APIs for:
  - NTFS (proprietary filesystem)
  - FAT (File Allocation Table)
- Linux uses APIs for:
  - ReiserFS (by Hans Reiser)
  - Btrfs (by Oracle Corporation)
  - Among many others



2.1 Market Study: Operating Systems

- Linux: implements a kernel-level API, the lowest-level instructions from the OS
  - Provide interfaces to develop the filesystems
  - Where the filesystem code and logic are
  - MS-DOS operating system developed by Microsoft that used this type of API
- Windows NT: the common Microsoft Windows OS
  - Uses a driver-based API: filesystem code and logic are totally external
  - Allows Microsoft to keep their kernel closed to the public and modify the Windows filesystem online

2.2 Market Study: Filesystem APIs in Games

- Loading of assets stored in disk
    - Sprites
    - Music
    - SFX
    - Fonts
- **Hardcoded paths breaks portability to other platforms**

2.2 Market Study: Filesystem APIs in Games

- The solution:

To have a virtual filesystem that mounts our assets archive

- Gets rid of any working directory issue and paths like: C:\MyGame\MyWritingDirectory
- Removes OS differences in directories: trades the dependencies the virtual filesystem

2.2 Market Study: Filesystem APIs in Games

- Other benefits
  - Run-time efficiency due to less security checks and processes in comparison to the OS
  - Handling duplicated alias for the same file allows to:
    - Introduce patches to our game
    - Mods
    - Drawback: avoid if it is a huge file because of long updating times for the user

2.2 Market Study: Filesystem APIs in Games

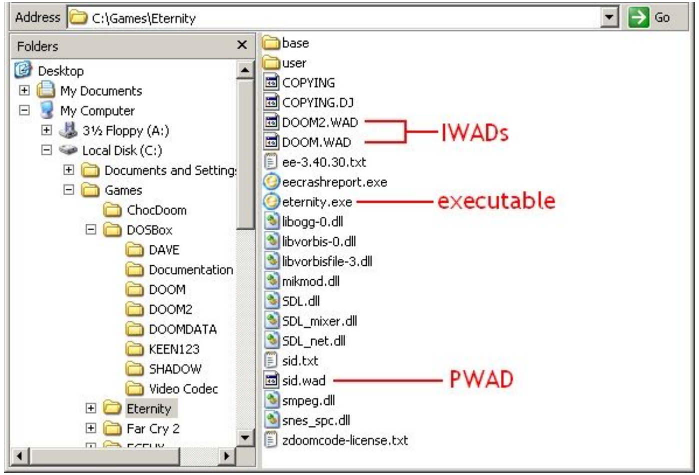
DOOM (1993) by ID Software

- From Wolfenstein 3D (1992) WAD files: “Where’s All the Data” that Doom fully implemented.

2.2 Market Study: Filesystem APIs in Games

DOOM (1993) by ID Software

- Became the standard way to pack assets
- Patches
- Mods
- Performance Increase (Just a few binary files)

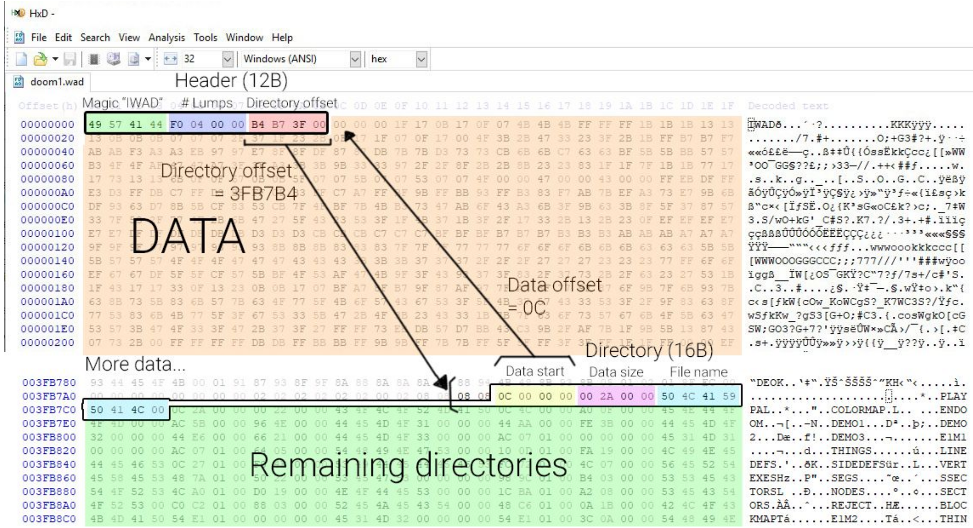


Address C:\Games\Eternity

2.2 Market Study: Filesystem APIs in Games

DOOM (1993) by ID Software

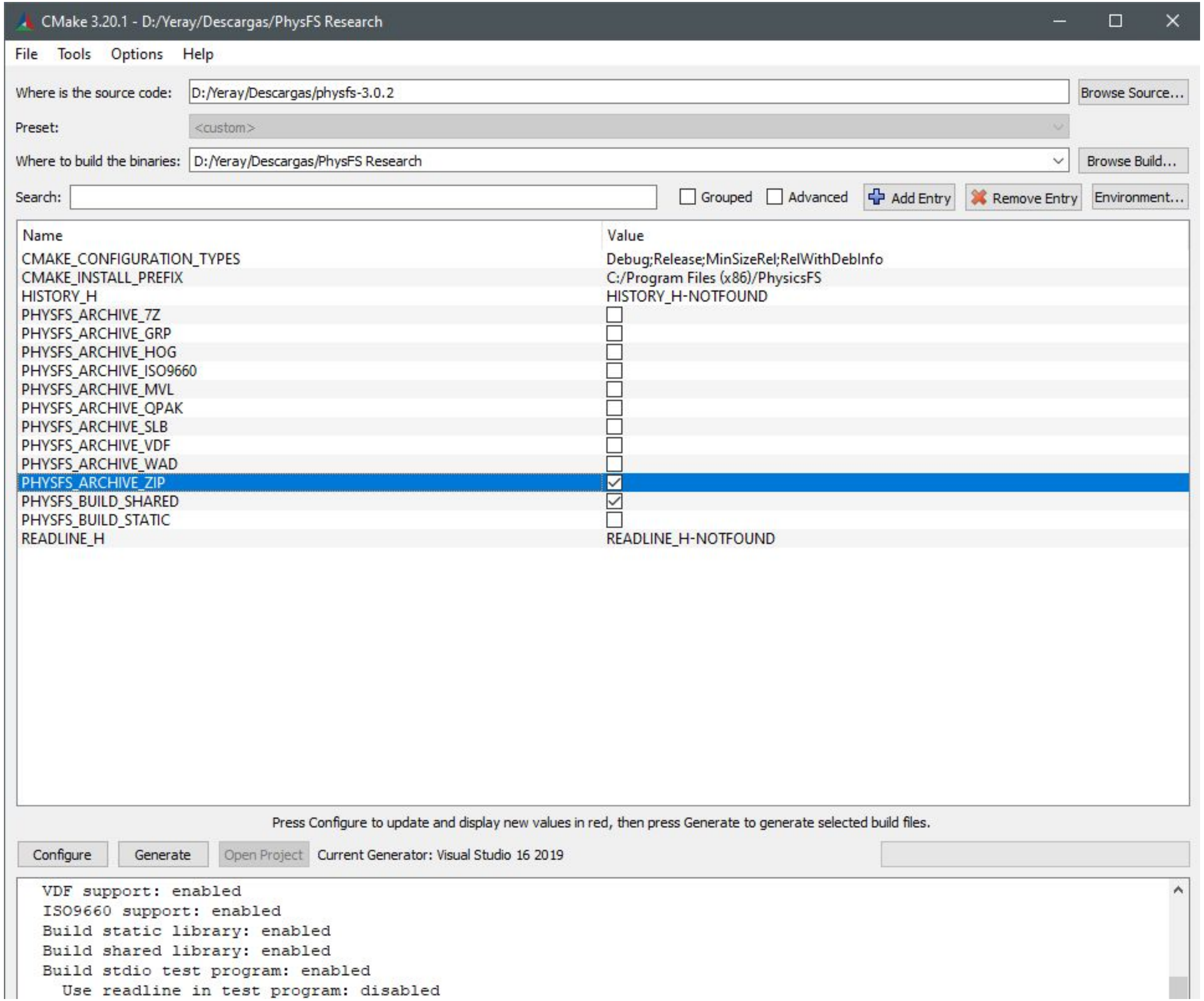
- Readable with an Hex Editor
- Header containing pointers to a series of directories at the end of the file
- Those directories point to where the data starts in the file



3.1 Selected Approach: PhysFS

- PhysicsFS or PhysFS is an API and a library that provides abstract access to archives
- By Ryan C. Gordon (Icculus.org)
- Inspired by Quake's 3 file subsystem (where he worked)
- Is a Transparent Hierarchical File System that allows to access ZIP files
- We will use a search path specified by us into the Assets archive

3.2 Selected Approach: Building PhysFS



Where is the source code: D:/Yeray/Descargas/physfs-3.0.2

Preset: <custom>

Where to build the binaries: D:/Yeray/Descargas/PhysFS Research

Search:

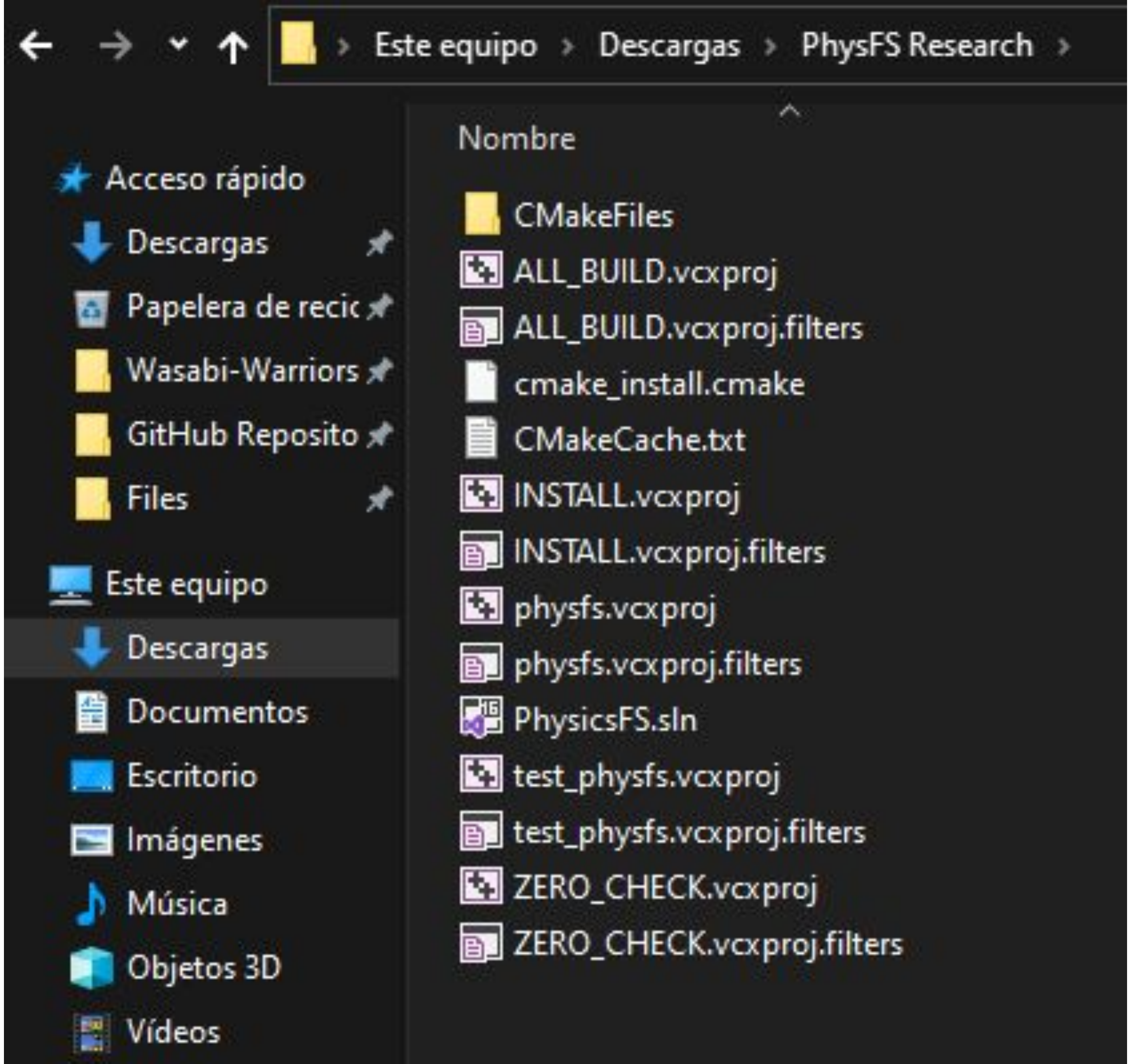
☐ Grouped ☐ Advanced

Name	Value
CMAKE_CONFIGURATION_TYPES	Debug;Release;MinSizeRel;RelWithDebInfo
CMAKE_INSTALL_PREFIX	C:/Program Files (x86)/PhysicsFS
HISTORY_H	HISTORY_H-NOTFOUND
PHYSFS_ARCHIVE_7Z	☐
PHYSFS_ARCHIVE_GRP	☐
PHYSFS_ARCHIVE_HOG	☐
PHYSFS_ARCHIVE_ISO9660	☐
PHYSFS_ARCHIVE_MVL	☐
PHYSFS_ARCHIVE_QPAK	☐
PHYSFS_ARCHIVE_SLB	☐
PHYSFS_ARCHIVE_VDF	☐
PHYSFS_ARCHIVE_WAD	☐
PHYSFS_ARCHIVE_ZIP	☑
PHYSFS_BUILD_SHARED	☑
PHYSFS_BUILD_STATIC	☐
READLINE_H	READLINE_H-NOTFOUND

Press Configure to update and display new values in red, then press Generate to generate selected build files.

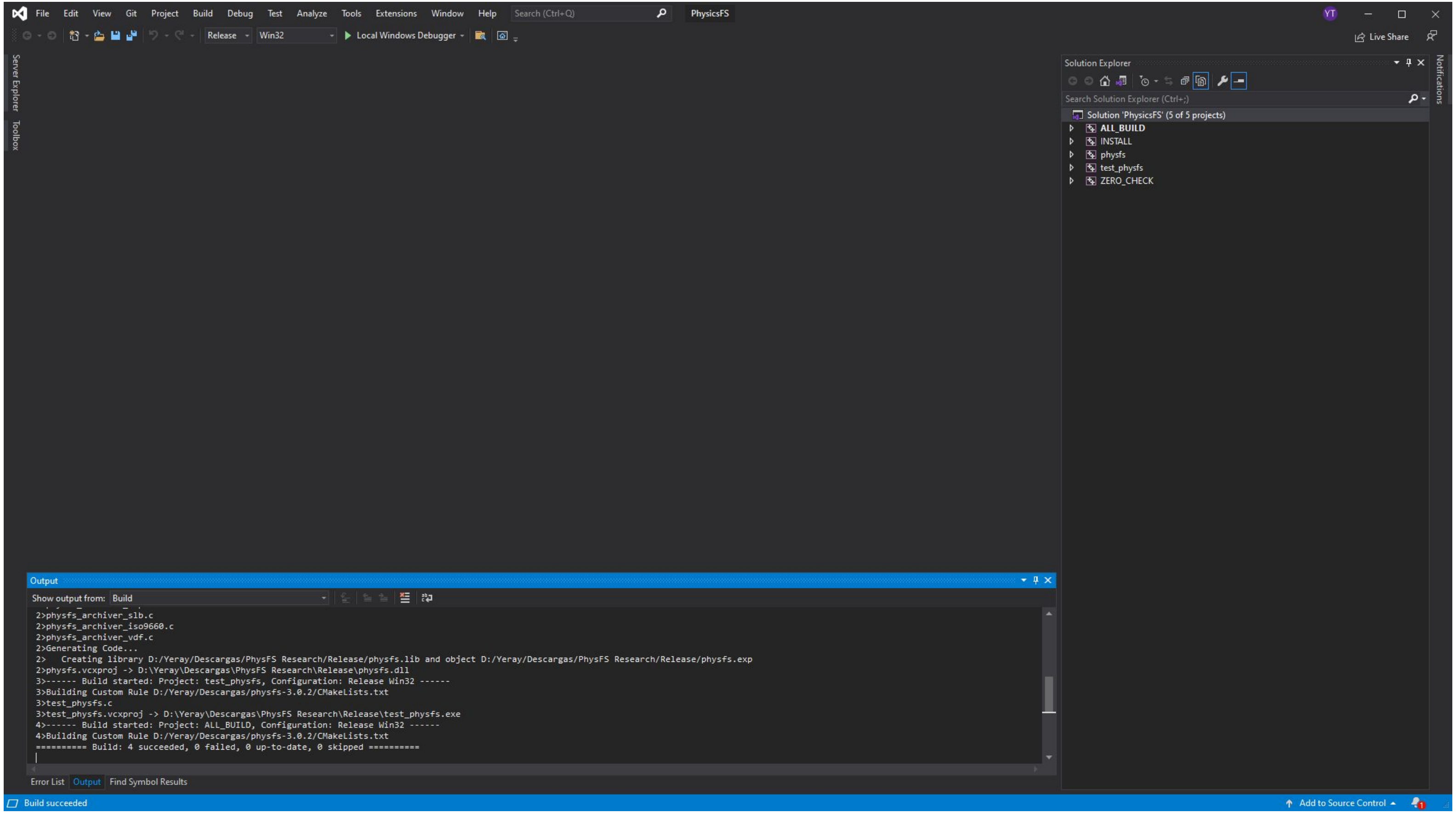
Current Generator: Visual Studio 16 2019

```
 VDF support: enabled
 ISO9660 support: enabled
 Build static library: enabled
 Build shared library: enabled
 Build stdio test program: enabled
   Use readline in test program: disabled
```



Nombre

CMakeFiles
ALL_BUILD.vcxproj
ALL_BUILD.vcxproj.filters
cmake_install.cmake
CMakeCache.txt
INSTALL.vcxproj
INSTALL.vcxproj.filters
physfs.vcxproj
physfs.vcxproj.filters
PhysicsFS.sln
test_physfs.vcxproj
test_physfs.vcxproj.filters
ZERO_CHECK.vcxproj
ZERO_CHECK.vcxproj.filters



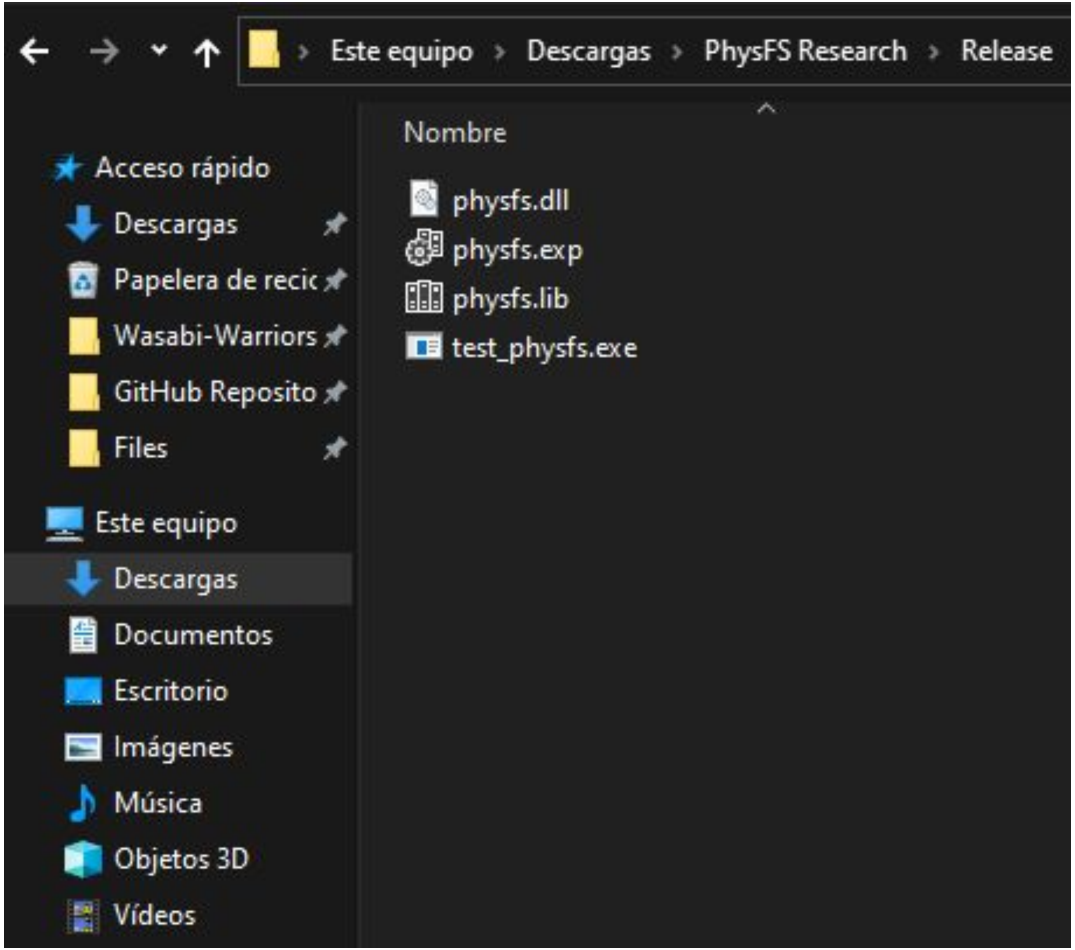
Release

Win32

Show output from: Build

```
2>physfs_archiver_slb.c
2>physfs_archiver_iso9660.c
2>physfs_archiver_vdf.c
2>Generating Code...
2>   Creating library D:/Yeray/Descargas/PhysFS Research/Release/physfs.lib and object D:/Yeray/Descargas/PhysFS Research/Release/physfs.exp
2>physfs.vcxproj -> D:\Yeray\Descargas\PhysFS Research\Release\physfs.dll
3>------ Build started: Project: test_physfs, Configuration: Release Win32 ------
3>Building Custom Rule D:/Yeray/Descargas/physfs-3.0.2/CMakeLists.txt
3>test_physfs.c
3>test_physfs.vcxproj -> D:\Yeray\Descargas\PhysFS Research\Release\test_physfs.exe
4>------ Build started: Project: ALL_BUILD, Configuration: Release Win32 ------
4>Building Custom Rule D:/Yeray/Descargas/physfs-3.0.2/CMakeLists.txt
========== Build: 4 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Build succeeded

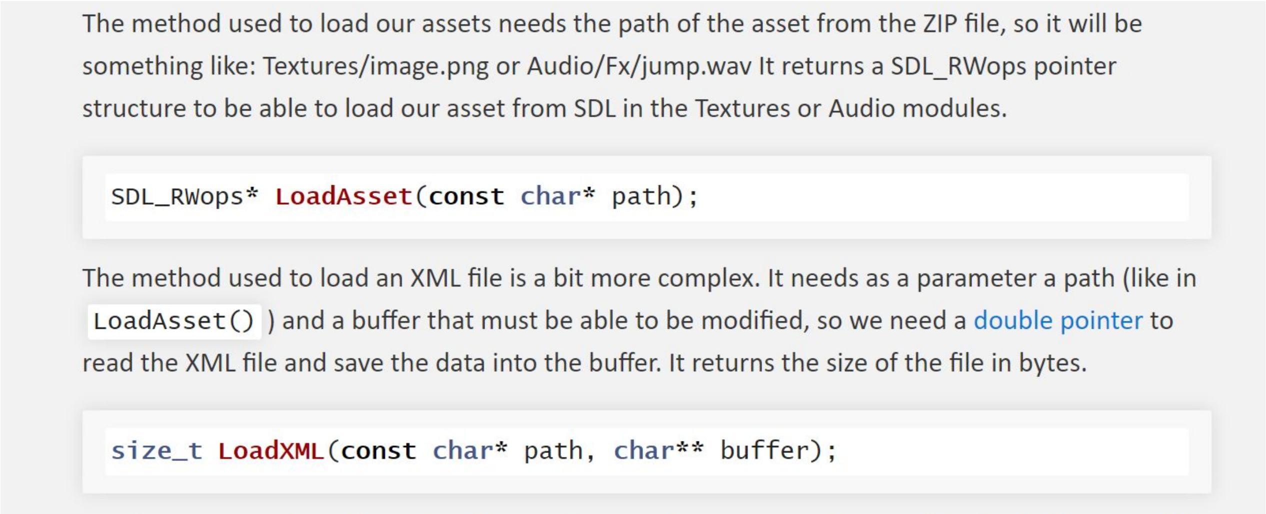


Nombre

physfs.dll
physfs.exp
physfs.lib
test_physfs.exe

3.3 Selected Approach: Implementation

- AssetsManager Module



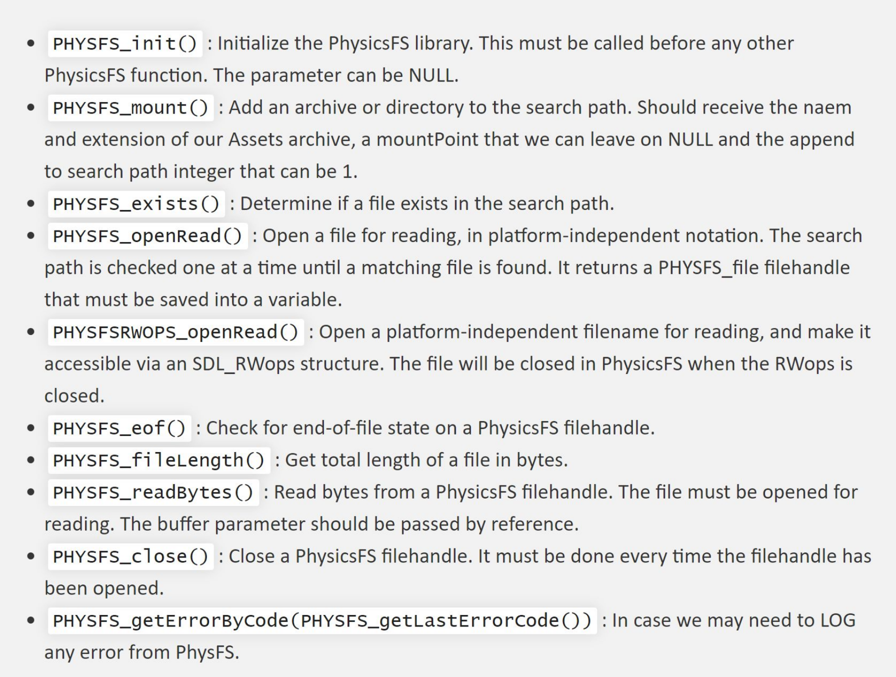
The method used to load our assets needs the path of the asset from the ZIP file, so it will be something like: Textures/image.png or Audio/Fx/jump.wav It returns a SDL_RWops pointer structure to be able to load our asset from SDL in the Textures or Audio modules.

```
SDL_RWops* LoadAsset(const char* path);
```

The method used to load an XML file is a bit more complex. It needs as a parameter a path (like in `LoadAsset()`) and a buffer that must be able to be modified, so we need a double pointer to read the XML file and save the data into the buffer. It returns the size of the file in bytes.

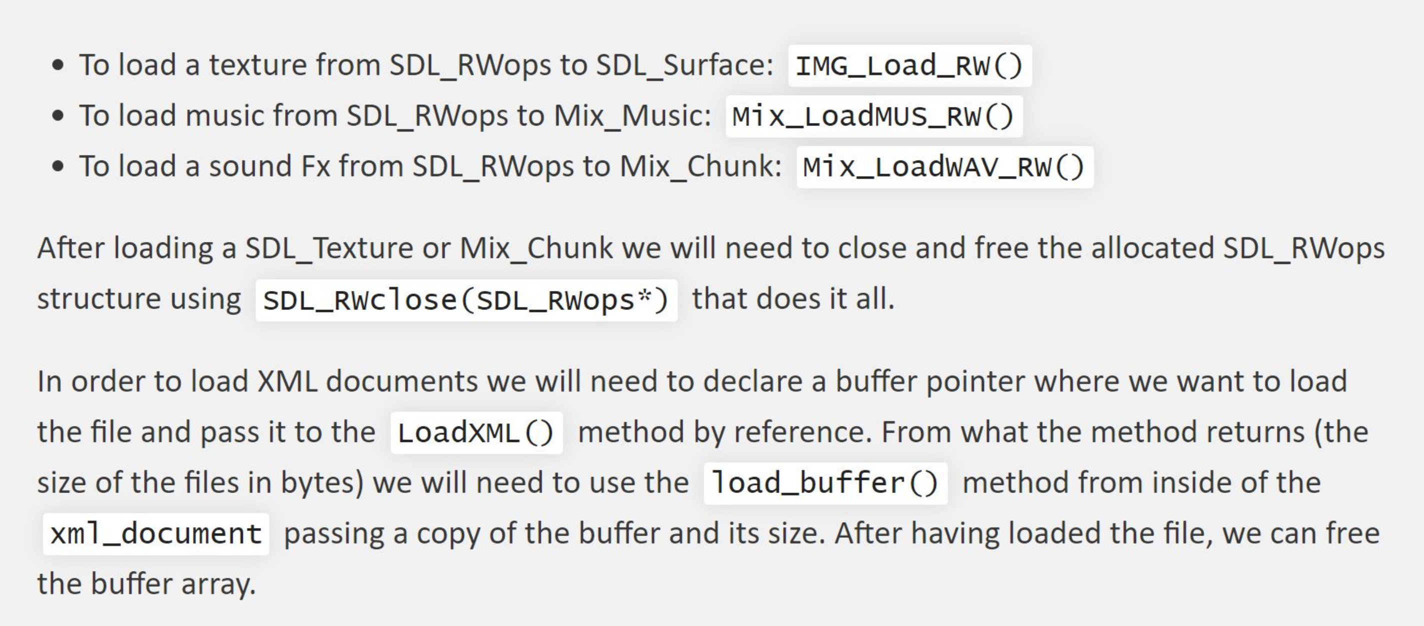
```
size_t LoadXML(const char* path, char** buffer);
```

3.3 Selected Approach: Implementation



- `PHYSFS_init()` : Initialize the PhysicsFS library. This must be called before any other PhysicsFS function. The parameter can be NULL.
- `PHYSFS_mount()` : Add an archive or directory to the search path. Should receive the naem and extension of our Assets archive, a mountPoint that we can leave on NULL and the append to search path integer that can be 1.
- `PHYSFS_exists()` : Determine if a file exists in the search path.
- `PHYSFS_openRead()` : Open a file for reading, in platform-independent notation. The search path is checked one at a time until a matching file is found. It returns a PHYSFS_file filehandle that must be saved into a variable.
- `PHYSFSRWOPS_openRead()` : Open a platform-independent filename for reading, and make it accessible via an SDL_RWops structure. The file will be closed in PhysicsFS when the RWops is closed.
- `PHYSFS_eof()` : Check for end-of-file state on a PhysicsFS filehandle.
- `PHYSFS_fileLength()` : Get total length of a file in bytes.
- `PHYSFS_readBytes()` : Read bytes from a PhysicsFS filehandle. The file must be opened for reading. The buffer parameter should be passed by reference.
- `PHYSFS_close()` : Close a PhysicsFS filehandle. It must be done every time the filehandle has been opened.
- `PHYSFS_getErrorByCode(PHYSFS_getLastErrorCode())` : In case we may need to LOG any error from PhysFS.

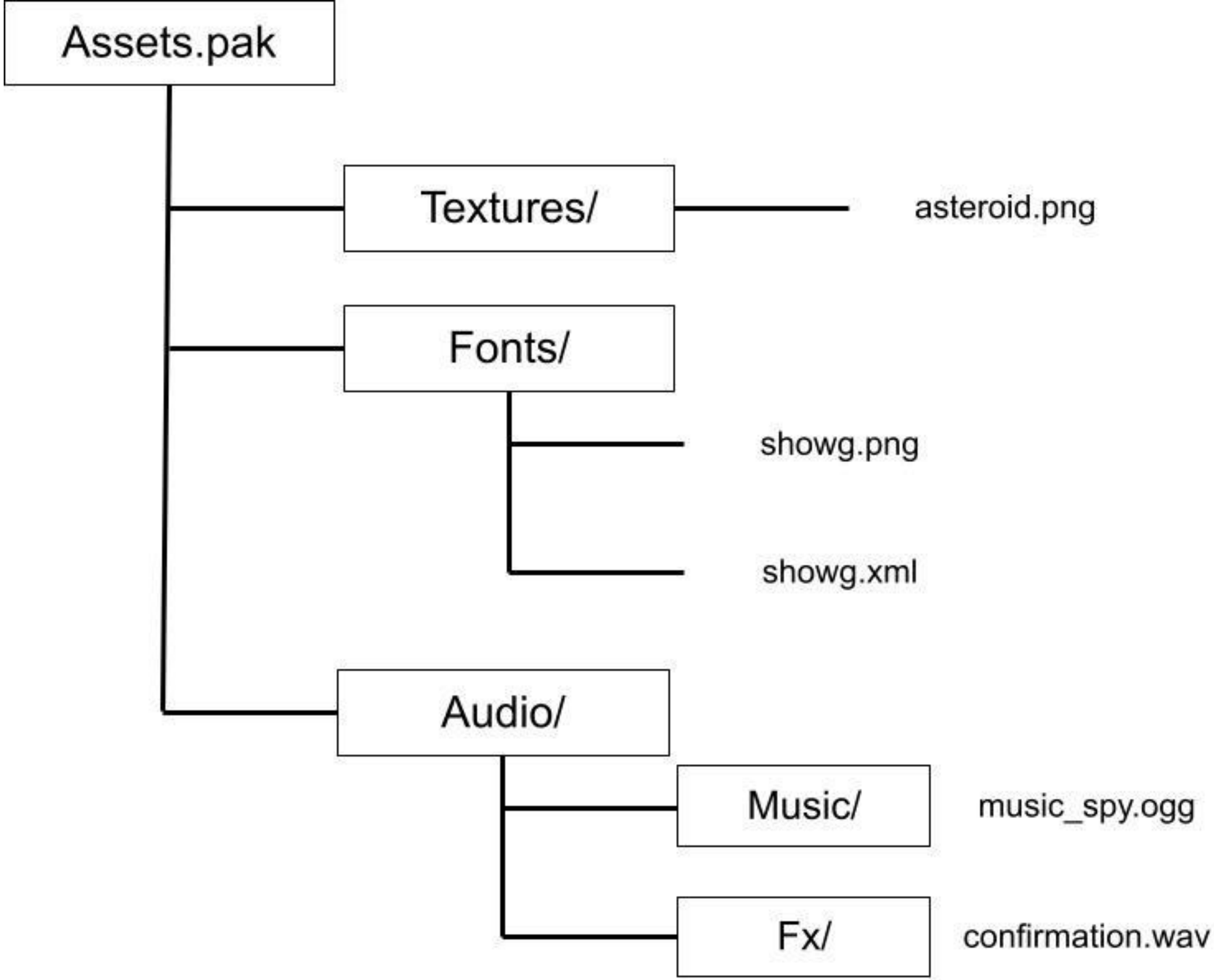
3.3 Selected Approach: Implementation



- To load a texture from SDL_RWops to SDL_Surface: `IMG_Load_RW()`
- To load music from SDL_RWops to Mix_Music: `Mix_LoadMUS_RW()`
- To load a sound Fx from SDL_RWops to Mix_Chunk: `Mix_LoadWAV_RW()`

After loading a SDL_Texture or Mix_Chunk we will need to close and free the allocated SDL_RWops structure using `SDL_RWclose(SDL_RWops*)` that does it all.

In order to load XML documents we will need to declare a buffer pointer where we want to load the file and pass it to the `LoadXML()` method by reference. From what the method returns (the size of the files in bytes) we will need to use the `load_buffer()` method from inside of the `xml_document` passing a copy of the buffer and its size. After having loaded the file, we can free the buffer array.



Assets.pak

Textures/

asteroid.png

Fonts/

showg.png

showg.xml

Audio/

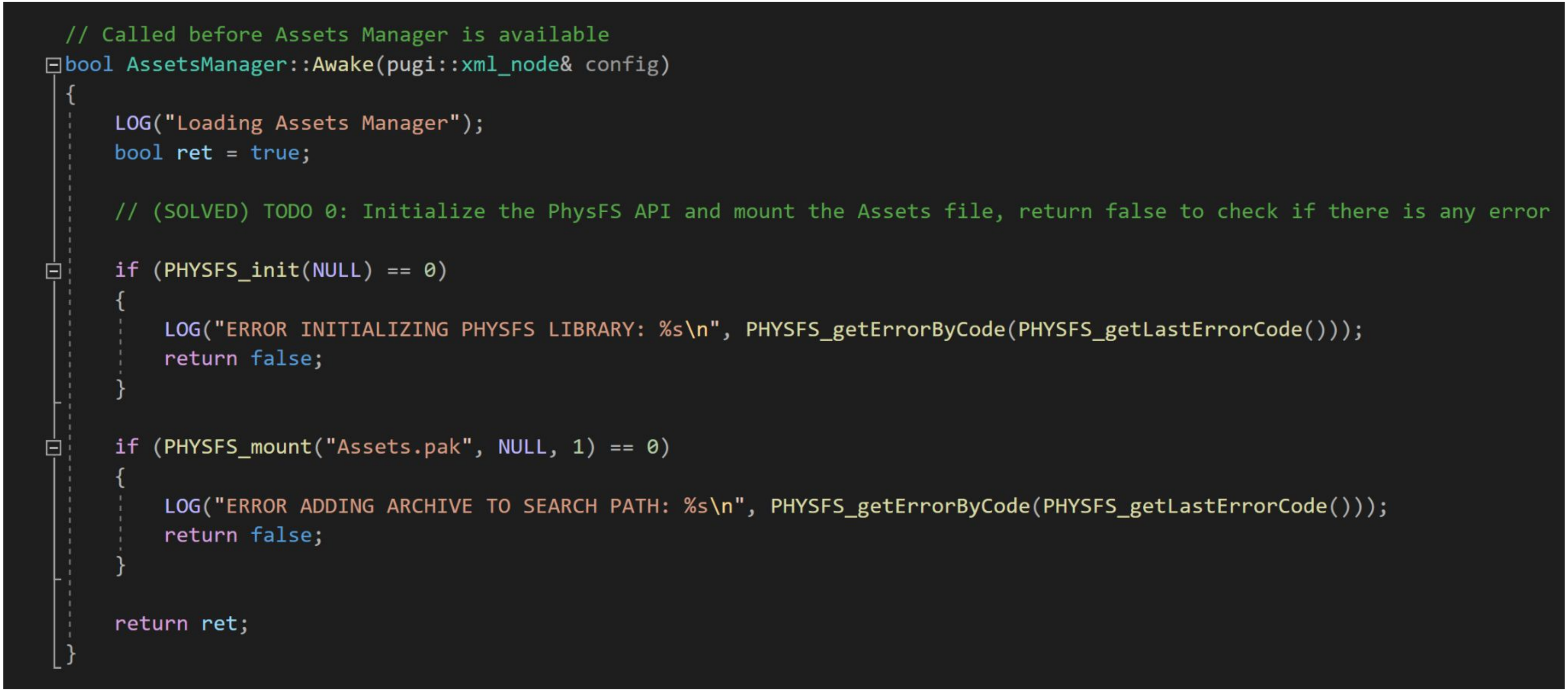
Music/

music_spy.ogg

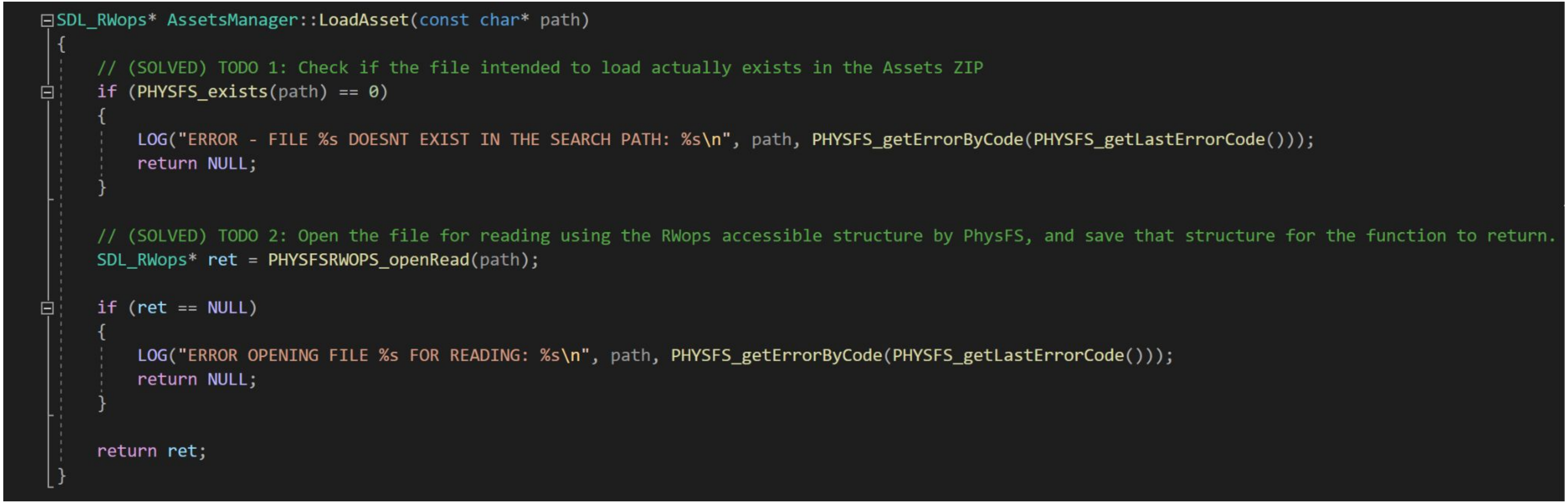
Fx/

confirmation.wav

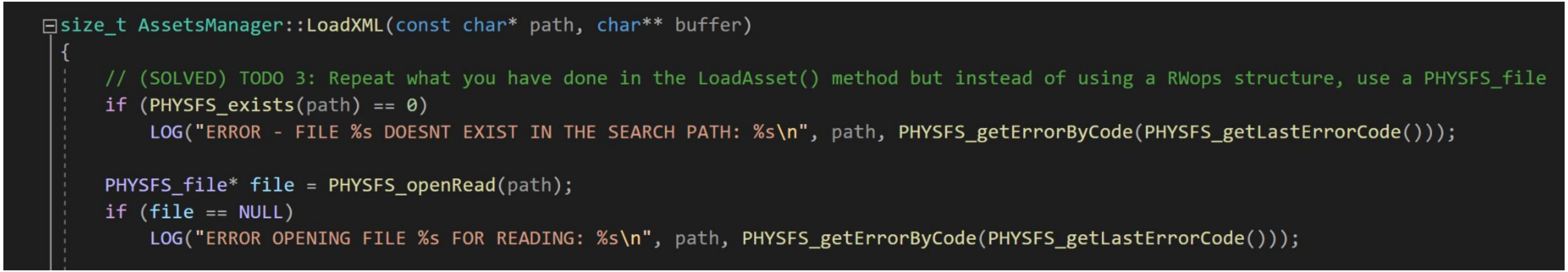
4. Exercises (TODOs)



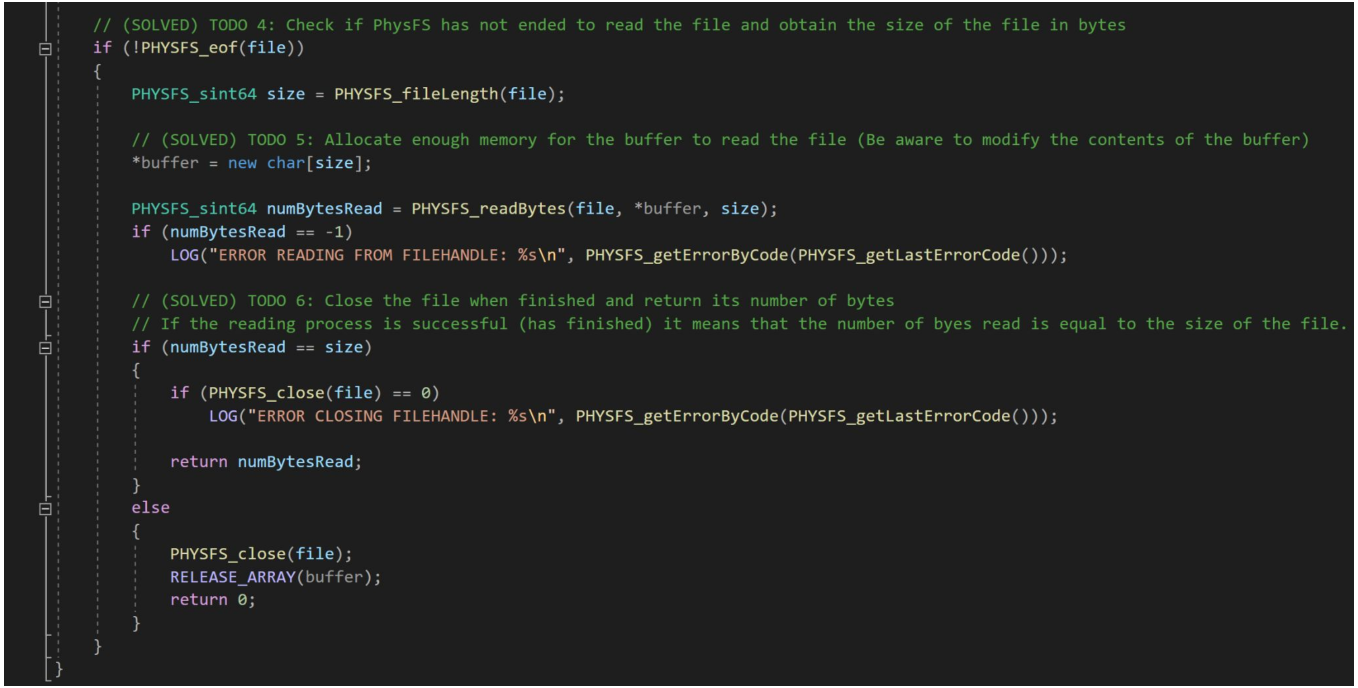
```cpp
// Called before Assets Manager is available
bool AssetsManager::Awake(pugi::xml_node& config)
{
    LOG("Loading Assets Manager");
    bool ret = true;

    // (SOLVED) TODO 0: Initialize the PhysFS API and mount the Assets file, return false to check if there is any error

    if (PHYSFS_init(NULL) == 0)
    {
        LOG("ERROR INITIALIZING PHYSFS LIBRARY: %s\n", PHYSFS_getErrorByCode(PHYSFS_getLastErrorCode()));
        return false;
    }

    if (PHYSFS_mount("Assets.pak", NULL, 1) == 0)
    {
        LOG("ERROR ADDING ARCHIVE TO SEARCH PATH: %s\n", PHYSFS_getErrorByCode(PHYSFS_getLastErrorCode()));
        return false;
    }

    return ret;
}
```



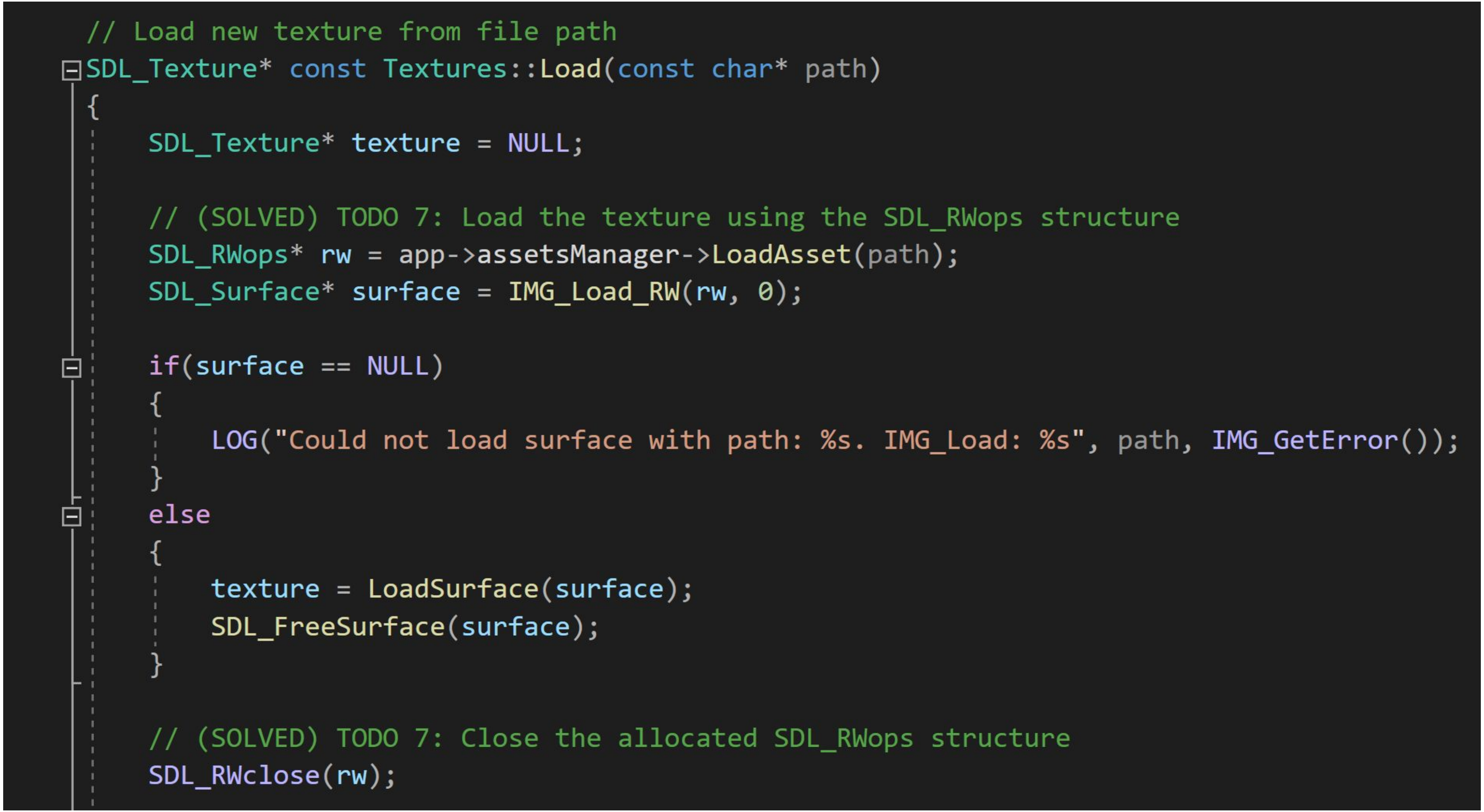
```
SDL_RWops* AssetsManager::LoadAsset(const char* path)
{
    // (SOLVED) TODO 1: Check if the file intended to load actually exists in the Assets ZIP
    if (PHYSFS_exists(path) == 0)
    {
        LOG("ERROR - FILE %s DOESNT EXIST IN THE SEARCH PATH: %s\n", path, PHYSFS_getErrorByCode(PHYSFS_getLastErrorCode()));
        return NULL;
    }

    // (SOLVED) TODO 2: Open the file for reading using the RWops accessible structure by PhysFS, and save that structure for the function to return.
    SDL_RWops* ret = PHYSFSRWOPS_openRead(path);

    if (ret == NULL)
    {
        LOG("ERROR OPENING FILE %s FOR READING: %s\n", path, PHYSFS_getErrorByCode(PHYSFS_getLastErrorCode()));
        return NULL;
    }

    return ret;
}
```



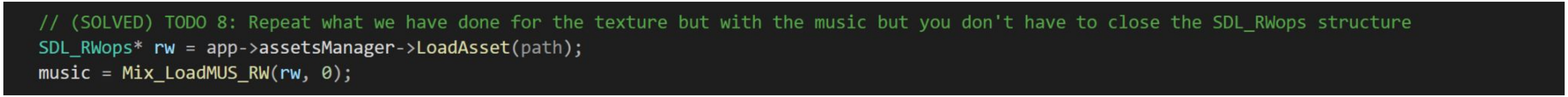
```
size_t AssetsManager::LoadXML(const char* path, char** buffer)
{
    // (SOLVED) TODO 3: Repeat what you have done in the LoadAsset() method but instead of using a RWops structure, use a PHYSFS_file
    if (PHYSFS_exists(path) == 0)
        LOG("ERROR - FILE %s DOESNT EXIST IN THE SEARCH PATH: %s\n", path, PHYSFS_getErrorByCode(PHYSFS_getLastErrorCode()));

    PHYSFS_file* file = PHYSFS_openRead(path);
    if (file == NULL)
        LOG("ERROR OPENING FILE %s FOR READING: %s\n", path, PHYSFS_getErrorByCode(PHYSFS_getLastErrorCode()));
```



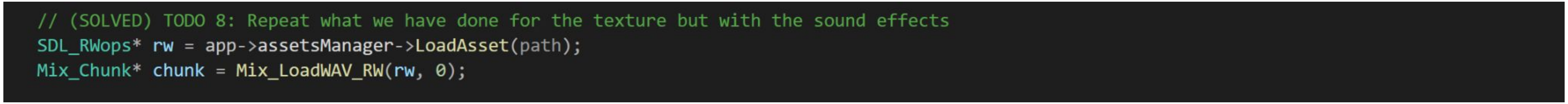
```
    // (SOLVED) TODO 4: Check if PhysFS has not ended to read the file and obtain the size of the file in bytes
    if (!PHYSFS_eof(file))
    {
        PHYSFS_sint64 size = PHYSFS_fileLength(file);

        // (SOLVED) TODO 5: Allocate enough memory for the buffer to read the file (Be aware to modify the contents of the buffer)
        *buffer = new char[size];

        PHYSFS_sint64 numBytesRead = PHYSFS_readBytes(file, *buffer, size);
        if (numBytesRead == -1)
            LOG("ERROR READING FROM FILEHANDLE: %s\n", PHYSFS_getErrorByCode(PHYSFS_getLastErrorCode()));

        // (SOLVED) TODO 6: Close the file when finished and return its number of bytes
        // If the reading process is successful (has finished) it means that the number of byes read is equal to the size of the file.
        if (numBytesRead == size)
        {
            if (PHYSFS_close(file) == 0)
                LOG("ERROR CLOSING FILEHANDLE: %s\n", PHYSFS_getErrorByCode(PHYSFS_getLastErrorCode()));

            return numBytesRead;
        }
        else
        {
            PHYSFS_close(file);
            RELEASE_ARRAY(buffer);
            return 0;
        }
    }
}
```



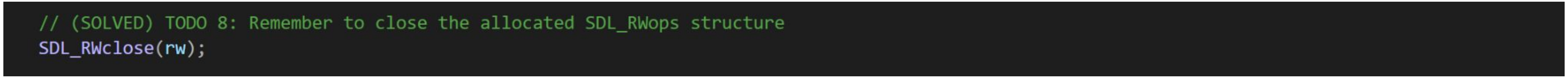
```
// Load new texture from file path
SDL_Texture* const Textures::Load(const char* path)
{
    SDL_Texture* texture = NULL;

    // (SOLVED) TODO 7: Load the texture using the SDL_RWops structure
    SDL_RWops* rw = app->assetsManager->LoadAsset(path);
    SDL_Surface* surface = IMG_Load_RW(rw, 0);

    if(surface == NULL)
    {
        LOG("Could not load surface with path: %s. IMG_Load: %s", path, IMG_GetError());
    }
    else
    {
        texture = LoadSurface(surface);
        SDL_FreeSurface(surface);
    }

    // (SOLVED) TODO 7: Close the allocated SDL_RWops structure
    SDL_RWclose(rw);
```



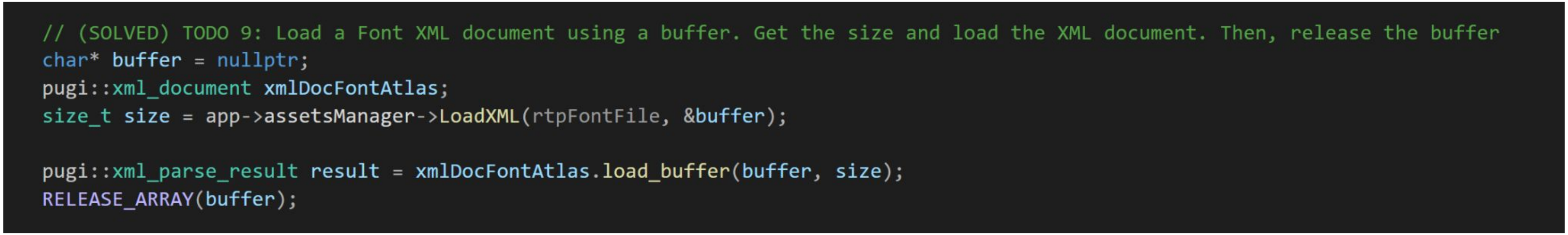
```
// (SOLVED) TODO 8: Repeat what we have done for the texture but with the music but you don't have to close the SDL_RWops structure
SDL_RWops* rw = app->assetsManager->LoadAsset(path);
music = Mix_LoadMUS_RW(rw, 0);
```



```
// (SOLVED) TODO 8: Repeat what we have done for the texture but with the sound effects
SDL_RWops* rw = app->assetsManager->LoadAsset(path);
Mix_Chunk* chunk = Mix_LoadWAV_RW(rw, 0);
```



```
// (SOLVED) TODO 8: Remember to close the allocated SDL_RWops structure
SDL_RWclose(rw);
```



```
// (SOLVED) TODO 9: Load a Font XML document using a buffer. Get the size and load the XML document. Then, release the buffer
char* buffer = nullptr;
pugi::xml_document xmlDocFontAtlas;
size_t size = app->assetsManager->LoadXML(rtpFontFile, &buffer);

pugi::xml_parse_result result = xmlDocFontAtlas.load_buffer(buffer, size);
RELEASE_ARRAY(buffer);
```

5. Possible Improvements

- PhysFS is focused on thread security but increases loading times in bigger games
- Only one operation at the same time by the API
    - Which means that only one thread can be used to load / write

Making Games Start Fast: A Story About Concurrency - Mathieu Ropert - CppCon 2020

5. Possible Improvements

- Encryption using Crypto++ or ZipStorage
- Better way to do it internally and decrypt and compress the data by ourselves

Thanks for your attention!