UNIVERSITY OF TEESSIDE

Advanced Game Software

Platform Tuned Game Demo

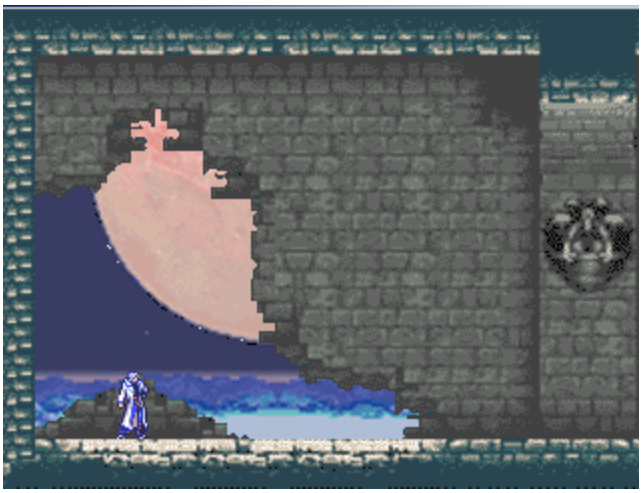
Paul Demeulenaere – G7105735

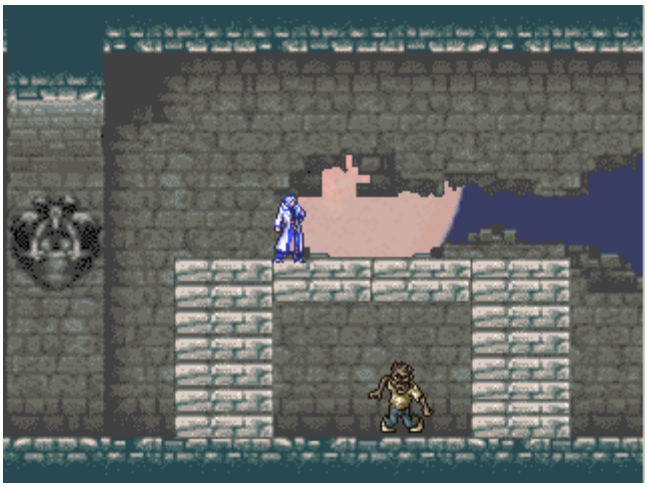
12/05/2008

Chapter 1 - Introduction

This project aim to develop a demonstration game on a dedicated computed hardware, the GP2X. The GP2X is a console of video game, there is seven generation of this hardware. The system of this console is based on Linux and this console support the SDL (Simple DirectMedia layer) with 2D acceleration.

The SDL is written in C but works in C++ natively; the development of this game has been done in C++ because this language ease the object oriented programming. The game is functional but hasn't really any interest (it isn't possible to die) but it is the organisation and programming techniques which are interesting.





Chapter 2 - User Guide

1. Installation

On windows:

First, it is necessary to copy the folder named "*Executable*" on a writable hard disk (some temporary files have to be created, so the application will not work directly from the CD). Then, execute "*Game.exe*" which is located in the folder "*BinWin*".

On Linux environment of GP2X:

Copy the folders "BinGp2x" and "Resources" in "mnt/nand". Then, in the GP2X browser execute "Game" which is located in "BinGp2x".

2. Controls

Action	Key on windows	Key on GP2X
Run left	Left arrow	Left joystick
Run right	Right array	Right joystick
Jump	Key A	Button A
Start	Pause |Attn	Start
Select	Backspace	Select
Quit	Start then Select	L,R and Start

3. Bugs

Due to a leak of time, there is still some problem with desallocation which could crash the GP2X when the application is exited.

The sound could be in mono instead of stereo but it is play two times faster in mono.

Chapter 3 - Demo creation

This section going to explain how the application is designed, it will be especially focus on interesting aspects.

1. Tools

a. Assert

The assertion is a useful tool in development. It permits to easily identify problems or errors and this aspect is totally ignored in release compilation mode. It is based on a simple macro:

```
#ifdef _DEBUG
#   define ASSERT(BOOLEAN) if (!(BOOLEAN)) { fprintf(stderr," assert on line %d and file %s ",__LINE__,__FILE__); exit(0); }
#else
      inline void DoNothing(bool) {}
#   define ASSERT(BOOLEAN) DoNothing(BOOLEAN)
#endif
```

We used this last by the call of “ASSERT(condition)”, when the condition is false, a message is displayed to the standard error output indicating where the program has been exited.

b. File Loading

All information about the game is stored in text files, these files can be commented simply beginning a line by the char ‘#’. To simplify the development and keep all function of a std::ifstream, a “CFileConfig” class derivate from ifstream has been implemented. The way of working is very simple: when we create a CFileConfig instance with a text file, all the file is read and copy without comment in another file, the same file with the extension “.temp”, then, the original file is close and the new is open. So it is possible to use common function like “getLine”. The temporary file will be destroyed with the instance of this class.

It would be possible to avoid the creation of another file and only save data in memory but it is this solution which has been chosen.

c. String manager

The main goal of the string manager is to avoid several allocation of std::string for each display of text. The string manager allocate a fixed number of strings (char *) at a maximal size. It is possible to call a sprintf function without allocate any char keeping the possibility of writing thanks to flag (like “%9ld”). The string manager is a useful tool with the CFileConfig class; indeed, it is possible to get a next line of a file in a preallocated string.

It necessary to be careful when this String Manager is used, a char * return can be changed by another call, so, if we need to keep a value, it is important to save this last in another string.

d. Memory Manager

Like the assert macro, the memory manager is only functional in debug mode. The memory manager permit to identify values which has been allocated and hasn’t been desallocated, in other words, the memory manager shows leak in memory. It use an overloaded of new and delete operator. All allocation in memory are saved in a std::map, when a value is delete, information about allocation are delete from the map and, at the end of the program, if there is still values in the map, there is leak in memory. All of this information is stored in a text file.

There were some problems with the memory manager because the instance of the singleton was deleting before destructor call. Finally, this problem has been resolved but this tool isn’t really used but it is now functional.

2. Resource Manager

The resource manager only load surface data from bmp file and give the reference of this surface. The interesting functionality of the resource manager is that is not possible to load several times the same BMP file. Like with the memory manager, there is a map which save address of bmp already load. If a BMP isn’t yet loaded, the resource manager creates a new surface and pushes it into a vector, else, if this BMP already exist in memory, the index of this reference in the vector is simply returned.

The resource manager has been designed for several other type of resource like sounds, it will be easy to quickly add this functionality but it is still a leak in this project.

3. Visualisation

The visualisation component is common; the main originality is in the storage of animation directly in visualisation instead of the managing of these animations by the world component. The sprites are separated in two types. First, there are the sprites animated. To load an animated sprite, we don't use directly the bitmap file; a FileConfig (cfg) is structured to give the main information like the colour of the transparency or size of sprit, number of animation and of course, the bitmap which contain all these sprites. The animation of a particular sprite is stored in one file; it isn't a list of files.

Figure 1: Sample of a bitmap animated file

The sprites which haven't animation are optimized thanks to a simpler class to display these lasts. About the optimisation, the display of surfaces on the screen is very costly. A bitmap no indexed in 8bit by pixel is very slow compared to indexed colour in 4bits, it is the best optimisation realised on this application, SDL is really faster with indexed colours.

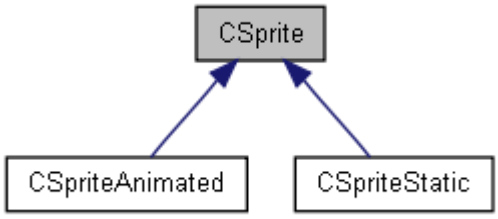


Figure 2: Derivation of the sprites

4. Message Manager

The event system is common in computer science. For example, we can see this technology in some API, like QT with its system of event and slot, a signal can be connected to a function and send by classes to another. This system is based on Meta information with script executed before the compilation; it is very interesting but also very complex.

In this project, all classes communicate mainly thanks to a message manager. It is a singleton which has to be optimized because used a lot of time. The main advantage of this kind of execution is the modularity and the ease to add some component. To explain the working of this message manager, a simple sample will be used.

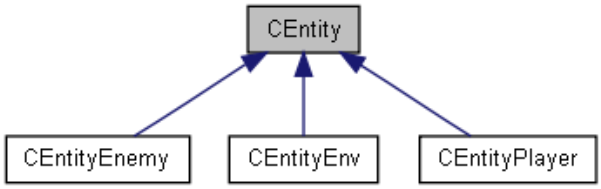
The button A is press and the player jump. The first message is sent when the button is pressed by the controller class, the controller reader catch this message and understand that the entities of the player want to jump. The physic interprets this wishing and accepts it or not, if it is possible, the physic sends another message to indicate that the player move. This last message is read by the world which displaces the player but this message could also be catch by a sound effect manager to play a particular file. It is interesting to see that the message “want move” could be also send by an artificial intelligence component.

The main difficulty of this message system is the allocation; we have to avoid all new or constructor and destructor call on each sending of message. A message should be deleted when every potential reader had read it. In fact, we used a preallocated vector of messages with a Boolean which represent if a message is used or not. There is different data linked for each type of message, to avoid construction and delete, the biggest data structure is allocated and its emplacement in memory is used to store all other type of data.

This message manager is very useful, the black box became really independent and this technique had permitted to isolate a physic and an artificial intelligence component from the world even if we have to access of some minimal information of entities. We could easily imagine that this type of organisation will permit a multithreaded application but there isn’t really an interest on the GP2X.

5. World

Excluding the message manager, the world is more classical. There is an entities class with some child to represent the player, the enemy or the environment.



The levels are divided in scene, for each scene, there are some entities. The tests of collision of reaction of the artificial intelligence are only done between entities of a particular scene. It is a kind of space partitioning. This information is all stored in text file, the game has some levels, for each levels, we have a constant background and a list of scene, and for each scene, there are several entities.

```
#Sample of cfg files for a scene#
#decor file
../Resources/Level/0/Back2.cfg

#Number of environment entity
5
#for each element of decor, position x, position y, size x and size z
0
215
320
50

100
130
25
200

250
130
25
200

120
130
100
20

0
0
320
20

#Number of enemies, positions, files
1
220
176
../Resources/Enemy/Zombie.cfg
```

Chapter 4 - Conclusions

1. Project

The game itself isn't really interesting. The main quality of this project is the source code beyond the game. This application is fully designed, even if some utilities aren't really judicious, it was an interesting research.

The main advantages of the development on a dedicated games platform is the optimisation possible, we know precisely the hardware and we can optimize an application for this specific hardware without thinking of potential incompatibility or optimisation which became bottlenecks on different hardware.

2. Personal

My knowledge in computer game programming has been really improved with this project, especially because it was my first attempt in game programming and I hope it isn't my last. I have learnt to structure game application thinking about a black box development to ease modification and making a reusable code.

If I have to do this kind of project again, I will think more about the game itself than the tools that I can develop, the game development is really an interesting domain which involve a lot of development skills. I think I will reuse some part of my source for my future works, the memory manager created could be a very interesting tool for the development. Even if this project revolutionary, I am satisfied of my work and I hope I will continue to increase my knowledge in this domain.