ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

UNIVERSITÉ DU QUÉBEC

rapport technique

PRÉSENTÉ À L’ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

DANS LE CADRE DU PROJET DE FIN D’ÉTUDES

<titre DU DOCUMENT>

par

<nom de famille, Prénom>

<code permanent>

PPRÉSENTÉ À

<PrénoM ET nom de famille DU PROFESSEUR)>

montréal, le <date>

**REMERCIEMENTS**

Thaieasyelec - <http://www.thaieasyelec.net/>

Merci à toute l’équipe de Thaieasyelec qui n’ont pas hésité à m’envoyer une nouvelle plateforme de développement par poste express de Thaïlande lorsque la mienne s’est avérée défectueuse. Sans eux, le prototype comme présenté dans ce rapport n’aurait pas vue le jour.

Louis Lynch

Merci à Louis, étudiant en génie logiciel, pour son aide lors de la modélisation et la réalisation du moteur de jeux. Sans lui, le moteur ne serait assurément pas aussi flexible et facile d’utilisation qu’il l’est maintenant.

**<titre du rapport TECHNIQUE>**

<NOM, Prénom>

**RÉSUMÉ**

<Texte interligne simple>

**TABLE DES MATIÈRES**

Page

[INTRODUCTION 2](#_Toc185065390)

[CHAPITRE 1 <titre> 2](#_Toc185065391)

[1.1 <Titre> 2](#_Toc185065392)

[CHAPITRE 2 <titre> 2](#_Toc185065393)

[CHAPITRE 3 <Titre> 2](#_Toc185065394)

[3.1 <Titre> 2](#_Toc185065395)

[CHAPITRE 4 <titre> 2](#_Toc185065396)

[4.1 <Titre> 2](#_Toc185065397)

[CHAPITRE 5 <titre> 2](#_Toc185065398)

[5.1 <Titre> 2](#_Toc185065399)

[CHAPITRE 6 <titre> 2](#_Toc185065400)

[6.1 <Titre> 2](#_Toc185065401)

[CHAPITRE 7 <titre> 2](#_Toc185065402)

[7.1 <Titre> 2](#_Toc185065403)

[CHAPITRE 8 <titre> 2](#_Toc185065404)

[8.1 <Titre> 2](#_Toc185065405)

[CHAPITRE 9 <titre> 2](#_Toc185065406)

[9.1 <Titre> 2](#_Toc185065407)

[CHAPITRE 10 <titre> 2](#_Toc185065408)

[10.1 <Titre> 2](#_Toc185065409)

[CONCLUSION 2](#_Toc185065410)

[RECOMMANDATIONS <S’il y a lieu> 2](#_Toc185065411)

[ANNEXE I <titre> 2](#_Toc185065412)

[ANNEXE II <titre> 2](#_Toc185065413)

[ANNEXE III <titre> 2](#_Toc185065414)

[ANNEXE IV <titre> 2](#_Toc185065415)

[ANNEXE V <titre> 2](#_Toc185065416)

[ANNEXE VI <titre> 2](#_Toc185065417)

[ANNEXE VII <titre> 2](#_Toc185065418)

[ANNEXE VIII <titre> 2](#_Toc185065419)

[ANNEXE IX <titre> 2](#_Toc185065420)

[ANNEXE X <titre> 2](#_Toc185065421)

[APPENDICES <S’il y a lieu> 2](#_Toc185065422)

[LISTE DE RÉFÉRENCES 2](#_Toc185065423)

[BIBLIOGRAPHIE 2](#_Toc185065424)

**LISTE DES TABLEAUX**

Page

[Tableau 1.1 Titre](#_Toc150853242) x

**LISTE DES FIGURES**

Page

[Figure 1.1 Test](#_Toc150853834) x

**LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES**

<Texte interligne simple>

**LISTE DES SYMBOLES ET UNITÉS DE MESURE**

<Texte interligne simple>

INTRODUCTION

<Texte interligne 1 1/2>

# Analyse de la problèmatique

## Définition du besoin

Si vous voulez développer un jeu vidéo, quelques possibilités s’offrent à vous. Vous pouvez bien sûr le faire sur ordinateur, mais il est souvent plus intéressant de jouer dans le confort de son salon. Dans ce cas, vous pouvez vous tourner vers les dernières consoles de jeux qui offrent souvent la possibilité de faire son propre jeu. Cependant dans le dernier cas, vous êtes limité aux fonctionnalités que le fabricant veut bien vous donner.

Il vous reste donc les consoles ouvertes. Ces consoles sont faites dans le but de fournir une plateforme de base pour le développement de jeux vidéo tout en permettant à l’utilisateur de modifier tout le logiciel s’exécutant sur celle-ci. De plus, dans la plupart des cas, ce genre de plateforme offre les plans du matériel afin de permette à quiconque ayant les habiletés nécessaires, de modifier la plateforme. Bref, tous les pouvoirs sont donnés à l’utilisateur. Ceci est très intéressant, mais les choix disponibles présentement comportent plusieurs lacunes.

Premièrement, les solutions existantes n’utilisent pas de contrôleur graphique matériel. Ceci limite grandement les performances, car il est très difficile pour le logiciel de faire le rafraichissement d’un écran. Comme un grand pourcentage du temps du processeur est utilisé pour la gestion de l’écran, il ne reste que peu de temps pour la gestion du jeu lui-même qui est une tâche assez difficile en soi.

Encore du coté du matériel, les solutions existantes on souvent trop peu de mémoire disponible ce qui rend impossible l’utilisation d’image de haute qualité visuelle à l’intérieur du jeu.

Une autre lacune des consoles ouvertes est le logiciel fourni. En effet, celui-ci est souvent très optimisé dû au manque de performance du microcontrôleur ce qui le rend souvent difficile à comprendre. De plus, la plupart de temps, aucun moteur de jeux n’est disponible ce qui rend la programmation de jeux assez ardus. Pour finir, le jeu et le noyau sont souvent indissociables.

Du côté de l’interface utilisateur, les consoles ouvertes actuelles utilisent souvent de simples contrôleurs désuets ce qui limite l’envergure des jeux développés.

## Définition du système

Vue de haut niveau, le système répondant aux besoins d’écrit plus haut ressemble à ceci.

Figure - Représentation du système

Nous avons donc une entré utilisateur et deux sorties soit l’audio et la vidéo.

Bien que sous cette forme le système semble simple, plusieurs choix s’offre à nous pour la réalisation d’une tel machine. Nous aurons donc un choix à faire et le cheminement est exposé au chapitre suivant.

## Risques et opportunités

# Conceptualisation de la solution

## Contraintes

Comme nous l’avons vue au chapitre précédent, nous avons besoin d’un système ne comportant que quelque entré/sortie. Ceci laisse une grande marge de manœuvre quant au matériel et logiciel utilisé pour la réalisation. Nous devons donc préciser quelques contraintes afin de réduire les possibilités. Suite à une longue réflexion, nous en sommes arrivés à la liste de contraintes suivante.

Table - Lise de contraintes

|  |
| --- |
| Utilisation de manette de jeux récente |
| Capacité plus grande ou égale au Super Nintendo |
| Utilisation d’un moteur de jeux 2D pour les jeux de type « side scroller » |
| Coût de production plus petit ou égale à 50$ |

La contrainte la plus importante est le coût de réalisation du prototype matériel. Ce dernier ne devrait, idéalement, pas dépasser 50$. Il faudra donc être minutieux dans le choix des composants utilisés, sans trop influencer la qualité du produit final. Une courbe d’apprentissage élevée est à prévoir pour le choix des composants tel le type de mémoire externe et le microcontrôleur idéal à utiliser pour notre projet. Aussi, la réalisation du circuit-imprimé à l’aide du logiciel Altium, la conception efficace d’un circuit imprimé afin de minimiser le bruit qui pourrait être causé par les signaux à haute fréquences et l’évaluation des différents types de signaux vidéos disponibles à la sortie de la console sont des tâches importantes qui demandent beaucoup de temps. Il faudra donc se limiter à ce qu’on peut faire dans le temps alloué.

Cependant, du côté logiciel nous serons limités à la puissance de calcul du microprocesseur choisi ce qui ne sera sûrement pas très élevé. De plus, comme nous voulons produire du code facilement réutilisable, nous nous limiterons aux langages de programmation orientés objet. Ce langage devra aussi être assez populaire afin de simplifier l’apprentissage.

## Calculs préliminaires

Suite aux barèmes établies dans la section précédente, nous devons faire quelque calcule pour avoir une meilleur idée de la taille des composent requis.

Pour ce faire, nous devons dès maintenant préciser ce que nous voulons dire exactement par capacité supérieur ou égale au Super Nintendo. Premièrement, cette console possède les caractéristiques suivantes.

Table - Caractéristique du Super Nintendo

|  |  |
| --- | --- |
| Caractéristiques | Valeurs |
| Architecture | 16 bits |
| Fréquence | 3.58 Mhz (valeur effective max) |
| Résolution | 256x224 (plus utilisé) |
| Couleurs | 16 bits |
| Image par seconde | 10 (valeur moyenne) |
| Espace pour le jeu | 6 MB (max) |
| Nombre de couches d’affichage | 2-3 |
| Son | 16 bits, 32 kHz |

À partir de ces valeurs, nous avons fait une série de choix qui seraient des valeurs acceptables pour notre console.

Table - Caractéristique de la console du projet

|  |  |
| --- | --- |
| Caractéristiques | Valeurs |
| Architecture | 32 bits |
| Résolution | 480x272 |
| Couleurs | 24 bits |
| Image par seconde | 10 |
| Nombre de couches d’affichage | 2 |
| Son | 16 bits, 44.1 kHz (Valeur de départ) |

Nous faisons ces choix pour plusieurs raisons. Premièrement, les architectures 32 bits sont très populaires de nos jours. Deuxièmement, les images de 24 bits par pixel sont un format très répandu et 10 images par seconde, en moyenne, est un taux de rafraichissement encore utilisé aujourd’hui et assez facile à atteindre. Nous nous limiterons à deux couches d’affichage, car la plupart des jeux n’utilisent que 2 couches. Pour finir, la résolution choisie est un bon compromis entre la haute résolution et la résolution du Super Nintendo. C’est cette résolution qui est normalement utilisée pour les écrans 4.3 pouces, très populaire chez les consoles portables.

Avec ces informations, nous pouvons estimer la puissance de calcul du processeur dont nous aurons besoin.



Le processeur devra donc être capable de faire en moyenne 2.6 millions de transferts par seconde pour l’affichage. Ceci peut être assez contraignant, mais il sera toujours possible de faire des choix matériels afin d’alléger la tâche comme avec l’utilisation de canaux DMA.

D’autre calcules sont nécessaires notamment au niveau de la taille de la mémoire requise. La mémoire contiendra le code du noyau ainsi que celui du moteur de jeux et du jeu. Elle contiendra aussi les images et le son. Compte tenue que le code ne devrait pas occuper plus de 200 KB, nous pouvons considérer l’espace utilisé par le code négligeable par rapport aux images et aux sons.

À ce stade nous ne savons pas quel type de fichier nous allons utiliser mais prenons le pire cas en considérant des formats non compressés.

Pour les images nous voulons 24 bits couleur, une résolution de 480x272 et deux couches d’affichage. Prenons tout de même 32 bits par couleur car il est plus facile de faire 1 transfert de 32 bits que 3 transferts de 8 bits ou 1 transfert de 16 bits et un de 8 bits.



Bien sûr ceci n’est qu’une approximation et en réalité nous utiliserons probablement plus. Pour plus de sureté, mettons nous une marge de 50%. Ceci nous donne 2 Méga-octets utilisé pour les images. Notez que nous n’avons pas calculé la taille de l’entête du fichier ici mais ceci est négligeable.

Pour rester dans l’affichage, nous aurons aussi besoins de mémoire pour le contrôleur LCD. Le calcules est pratiquement le même que le précédant. Cependant ici ce qui nous intéresse c’est le nombre de tampons d’affichage et non le nombre de couche. L’utilisation de 2 tampons est standard et fonctionnera très bien pour notre application.



Ici cette utilisation est fixe donc nous n’avons pas besoin de marge de sécurité. Tant que la résolution reste 480x272, la mémoire utilisé par le contrôleur LCD est exactement celle calculé.

Du coté de l’audio maintenant, considérons le standard Wave 16 bits par échantillons, 44.1 kilo-échantillons par secondes. Le Super Nintendo qui est notre console de référence utilisait 32 kHz et ceci sera probablement assez pour nos besoins mais prenons tout de même le pire cas. Commençons par traiter un fichier de 3 minutes qui serait parfait pour une musique d’arrière plan.



Comme nous le voyons, la musique non compressé occupe rapidement beaucoup d’espace. De plus, nous aurons aussi des effets sonores mais ceux-ci serait probablement d’une duré de quelque milliseconde à une seconde et donc beaucoup plus petit. Comme nous pouvons utiliser un type de fichier sonore de moins longue duré en augmentant la fréquence de répétition et/ou diminuer la qualité, nous avons tout de même beaucoup de façon de réduire la taille utilisé par l’audio. Nous pouvons donc considérer le 15 Mo comme le maximum sans utiliser de marge de sécurité.

À la lumière de ces calcules, la mémoire devrait être d’une taille de 17 Mo

## Définition des barèmes menant à la prise de décision

En partant des contraintes et des calcules préliminaires, nous pouvons construire une liste de barème nous permettant de faire le choix des composants de notre système. Commençons par les simples déductions.

Nous voulons utiliser une manette de jeux récente. Les trois choix que nous avons sont la manette de Xbox360, de Wii et de Playstation 3. La première pourrais se connecter sur notre plateforme par un adapteur propriétaire de Microsoft USB. Les deux autres quant à elles peuvent se connecter par un adapteur Bluetooth encore une fois USB. Notre entré devra donc vraisemblablement être de type USB.

La partie la plus compliqué est probablement la puissance de calcul requise. Les éléments pouvant diriger notre choix est la nombre de transfert par seconde que le processeur aura à faire pour l’affichage ainsi que le traitement requis pour le moteur de jeux. Rappelons que pour l’affichage le processeur doit traiter transfert par seconds en moyenne comme calculé à la section « Calculs préliminaires ». Pour pouvoir offrir le support du moteur de jeux il est difficile de prévoir la puissance minimal. Pour faire le choix nous devrons donc nous basé principalement sur l’expérience du développeur de l’équipe. Selon lui, il serait possible de faire le travail sur un microcontrôleur d’à peu près 100 Mhz sans cache ni module d’opération point flottant matériel. Cependant, un module de division matériel serait préférable. Nous avons aussi comme référence une autre console du même type fonctionnant sur un processeur de 28 MHz. Celle-ci offre une résolution de 240x224 avec 8 bits par couleur. Elle n’utilise pas de contrôleur LCD mais elle n’utilise pas non de moteur de jeux. Par contre, en considérant l’utilisation d’un contrôleur LCD il est censé de croire que nous serons en mesure d’utiliser un moteur de jeux et offrir une résolution de 480x272 en utilisant une fréquence d’horloge un peu plus grande que celle de cette autre console. Ceci vient appuyer les propos du développeur du projet.

Pour la sortie vidéo le mieux serait d’avoir un contrôleur LCD pouvant nous offrir la résolution voulu en 24 bits couleurs. Cette sortie pourra ensuite être convertie au besoin en S-Vidéo, VGA ou composite.

La sortie tant qu’à elle sera assurément faite par un convertisseur numérique à analogique spécialisé pour l’audio ou non. Peut importe le type, celui-ci sera capable de supporter facilement des fréquences d’échantillons bien supérieure aux limite de l’oreille humaine donc ce sera au développeur logiciel de décider. Afin de laisser une marge de manœuvre, prenons un échantillon de 10 bits comme la limite minimal acceptable. Nous verrons par la suite ce que nous décidons.

Il faut aussi prévoir un support pour le moteur de jeux et pour les jeux. Un support standard et facile à utiliser est la carte mémoire de type SD. Celle-ci peut être utilisée à grande vitesse avec un contrôleur MMC/SD ou à relativement basse vitesse par SPI. Comme la plupart des microcontrôleurs possèdent un port SPI, ceci ne devrait pas trop limiter nos recherches. De plus, ces cartes peuvent facilement être exploité jusqu’à des tailles de 4 Go ce qui est amplement pour notre utilisation. Passé 4 Go il est possible que l’utilisation se complique dû au système de fichier.

Pour cette mémoire externe, nous avons le choix entre le type DRAM et SRAM. La mémoire de type DRAM est souvent moins dispendieuse mais est volatile et doit donc être rafraîchie fréquemment, ce qui complexifie son utilisation. Par contre, les circuits récents incluent souvent le rafraîchissement automatique. Une mémoire de type SRAM est plus coûteuse mais possède une bande passante plus élevée étant donnée l’absence de condensateur, contrairement à la DRAM. La SRAM est aussi de plus grande dimension que la DRAM pour une même taille mémoire, étant donné le plus grand nombre de transistors utilisés. Ici, le coût étant un facteur important et étant donné la rapidité relativement élevée, pour notre projet, des mémoires DRAM synchrone (SDRAM), nous nous pencherons donc intuitivement vers le choix d’une mémoire de ce dernier type. N’ayant pas de connaissances poussées dans le domaine, et le temps limité alloué au projet, nous n’utiliserons que les principales caractéristiques, dont nous jugeons pour le moment nécessaire, afin de faire un choix le plus éclairé possible. Les trois principales caractéristiques à tenir en compte seront la bande passante, la taille et le temps de latence, qui est défini comme le temps d’accès à la mémoire en lecture et mesuré en nombre de coup d’horloge. Comme vu dans les calculs préliminaires, nous aurons besoin d’une mémoire nous aurions besoin d’à peu près 17 Mo de mémoire RAM avec un BUS de 32bits. Comme la taille se compte en puissance de 2, prenons 16 Mo comme espace minimal. Nous préconiserons un temps de latence de 2 coups d’horloges afin d’optimiser le temps de chargement des données.

Pour finir, il serait souhaitable que notre microcontrôleur offre des canaux DMA pour permettre des transferts en mémoire sans l’intervention du processeur. Ceci pourrait nous donner un filet de sécurité si le microprocesseur manque un peu de puissance pour le rafraichissement de l’image.

Voici donc un tableau récapitulatif de nos barèmes.

Table - Barèmes des choix matériel

|  |  |
| --- | --- |
| Caractéristiques | Valeurs |
| Entré utilisateur | USB |
| Fréquence du microcontrolleur | ~100 MHz |
| Instruction de division matérielle | Souhaité |
| Sortie vidéo | 480x272, 24 bits couleur |
| Sortie audio | Minimalement 10 bits |
| Unité de stockage | Carte SD |
| Mémoire | Minimalement 16 Mo |
| Autre | Canaux DMA |

## Évaluation des solutions possibles

Premièrement, du côté du choix du microcontrôleur, nous avons plusieurs options disponibles. Ci-dessous, une liste de quelques microcontrôleurs qui ont été analysé suivant le barème fixé.

|  |  |
| --- | --- |
| **Composante** | **Caractéristiques principales** |
| LPC2478 |          Vieux cœur ARM7 |
|          Pas de division hardware |
|          72 Mhz |
|          Contrôleur LCD |
|          Contrôleur SDRAM |
|          Contrôleur USB host |
|          Boitier TQFP pouvant être soudé facilement |
| STM32F103 |          Nouveau cœur Cortex-M3 |
|          Division hardware |
|          72 Mhz |
|          15$ |
|          Pas d’USB host. Besoin d’un contrôleur USB externe (10$) |
|          Contrôleur SRAM seulement |
|          Pas de contrôleur LCD. Possibilité d’en ajouter un hardware externe sur le bus de mémoire externe. |
|          Nous possédons déjà du code fonctionnel |
|          Boitier TQFP pouvant être soudé facilement |
| LPC1768 |          Nouveau cœur Cortex-M3 |
|          100 Mhz |
|          10$ |
|          USB host |
|          Pas de mémoire externe |
|          Pas de contrôleur LCD |
|          Boitier TQFP pouvant être soudé facilement |
| AT91SAM9260 |          Vieux cœur ARM9¸ |
|          ~200 MHz |
|          15$ |
|          Possède de la cache |
|          Pas de contrôleur LCD |
|          Boitier PQFP pouvant être soudé facilement |
| OMAP3550 |          Microcontrôleur très performant, L1 et L2 cache, grand pipeline, division et opération point flottant matériel. |
|          600 MHz |
|          50$ |
|          Co-processeur graphique |
|          Affichage HDMI 720p |
|          Le beagleboard contient déjà la sortie audio/vidéo donc possiblement pas de développement matériel. |
|          Pas de code bas niveau ainsi que de fichier de définition pour les registres interne fournis par la compagnie |
|          Boitier BGA ne pouvant pas être soudé facilement |

Deuxièmement, les choix disponibles pour la mémoire externe sont plutôt nombreux et complexes. Ci-dessous, une brève liste de quelques modèles que nous avons cru bon sélectionner.

|  |  |
| --- | --- |
| **Composant** | **Paramètres** |
| W9812G6IH | Latence: 2 @ 100MHz |
| Taille: 2MB x 4 banks x 16 bits bus wide |
| Fréquence: 200MHz max |
| Rafraîchissement: 4000cycle/64ms = 16us/cycle |
| Tension: 3.3V |
| Prix: 3,74 CAD |
| 54 TSOP |
| IS42S16800E | Latence: 2 @ 100MHz |
| Taille: 2MB x 4 banks x 16 bits bus wide |
| Fréquence: 200MHz max |
| Rafraîchissement: 4096cycle/64ms = 16us/cycle |
| Tension: 3.3V |
| Prix: 5,76 CAD |
| 54TSOP |
| MT48LCM16A2 | Latence: 2 @ 100MHz |
| Taille: 2MB x 4 banks x 16 bits bus wide |
| Fréquence: 167MHz max |
| Rafraîchissement: 4096cycle/64ms = 16us/cycle |
| Tension : 3,3V |
| Prix: 4,62 CAD |
| 54 TSOP |

À la suite d’une brève analyse, excepté une légère différence de prix, on voit rapidement que les paramètres dont nous tenons compte nous donne peu de différences entre les modèles et que nous auront besoin de connaissances plus poussées afin de faire un choix éclairé sur ce type de composant.

Troisièmement, nous avons sélectionné 3 choix possibles de sortie vidéo : S-Video, Composite-combiné et VGA. Le signal de sortie du microcontrôleur sera un signal numérique sur 24 bits. Il nous faudra nécessairement un circuit d’adaptation pour pouvoir générer un des trois types de signal désiré.

La principale différence entre les signaux S-Vidéo et Composite est que le signal S-Vidéo transporte l’image sur deux conducteurs, un pour la luminance (niveau de gris) et un pour la chrominance couleur. Le signal Composite-combiné (Connecteur jaune) transporte ces deux signaux encodés sur un même câble. Pour ces deux types de signaux, il nous faut donc, à partir du signal numérique 24 bit, produire un ou deux signaux encodés analogiquement.

Pour sa part, le type VGA est composé de 5 signaux transportés sur 5 câbles différents. Un signal de synchronisation horizontal, un signal de synchronisation vertical TTL et 3 signaux analogique de couleurs rouge, vert et bleu (RGB). Pour générer un signal VGA, nous aurons d’abord besoin que le microcontrôleur génère les deux signaux de synchronisation en sortie. Par la suite, l’utilisation d’un simple convertisseur numérique-analogique pourra être utilisé afin de transformer le signal numérique sortant du microcontrôleur.

Finalement, le choix de la sortie audio se fera entre, une sortie analogique provenant du microcontrôleur si elle est disponible ou encore en utilisant une sortie I2S pour communiquer avec un convertisseur numérique analogique externe. La disponibilité d’un de ces deux périphériques influera donc sur le choix du microcontrôleur à utiliser.

## Prise de décision en fonction des barèmes établis

Après avoir établis une liste de choix possible en fonction des barèmes établis, vient maintenant le temps de faire la sélection des composantes nécessaire à la réalisation du projet.

Le microcontrôleur choisi pour la console est le LPC2478 de la compagnie NXP. Nous avons décidé de choisir celui-ci car c’est le seul, dans notre liste, qui contienne toutes les caractéristiques désirées excepté la division matérielle. Son contrôleur LCD ainsi que son accès à des canaux DMA nous permettra d’optimiser la génération du signal vidéo. Son contrôleur d’hôte USB sera utilisé pour communiquer avec un périphérique de commande (manette) et son contrôleur de mémoire externe pouvant gérer un BUS allant jusqu’à 32 bits de données est idéal pour un module de mémoire externe SDRAM. De plus, il contient un convertisseur numérique/analogique 10 bits et un périphérique I2S qui pourront être utilisés pour générer notre signal audio en sortie.

Le choix de la SDRAM n’a pu être établi pour l’instant. Ce que nous avons comme information est que nous aurons besoin de deux circuit, de 16 bits chacun afin d’obtenir notre BUS de communication de 32 bits. Les périodes de rafraîchissement sont généralement de 4000 cycles pour 64ms, soit 16us par cycle de rafraîchissement. Le microcontrôleur utilisé pourra fonctionner à 72MHz, ce qui permettra, pour les 3 modèles de mémoire sélectionnés, de facilement respecter une latence de 2 coups d’horloges et de générer le signal de rafraîchissement de la SDRAM aux 16us. Précisons encore que, étant donné le temps limité alloué au projet, nous avons laissé cette décision de côté afin de se concentrer sur les autres parties du projet. Pour faire un choix éclairé, il serait convenable de faire appel à une personne plus compétente dans le domaine.

Ensuite, le type de sortie vidéo utilisé sera le VGA. Nous avons choisi celui-ci car on le retrouve autant sur les écrans de télévision récents que sur tous les écrans d’ordinateur. L’utilisation d’un standard répandu permettra de faciliter la mise en marché future du produit.

De plus, c’est le signal le plus facile à généré à partir d’un signal numérique. Le contrôleur LCD du LPC2478 comprend aussi la génération des signaux de synchronisation verticale et horizontale en sortie nécessaire à la communication VGA. Il nous reste donc à trouver un moyen de convertir le signal numérique, normalement transmis au LCD, en signal analogique à l’aide d’un convertisseur. Le choix et la conception de ce convertisseur seront décrits plus en détails dans le chapitre 3 « architecture matérielle ».

Dernièrement, étant donné le microcontrôleur choisi, nous avons toujours le choix entre deux périphériques pour générer la sortie audio. Le DAC interne du LPC2478 ne contient que 10 bits de données, ce qui nous donne un signal de plus faible qualité que le Super Nintendo dont le signal audio est généré sur 16 bits à 32 kHz. Ici, l’utilisation du périphérique I2S et d’un circuit imprimé externe nous permettrait d’atteindre les qualités de la console fétiche. Le prix d’un tel DAC est de moins de 3,00 CAD si on se fie au modèle MAX9850 de la compagnie MAXIM. Une fois encore, le choix et la conception de ce périphérique seront analysé plus en détails dans le chapitre 3.

Prenons maintenant le temps de nous arrêter afin de faire une liste récapitulative des composants utilisés dans la conception matérielle. Nous aurons ainsi une idée du prix de fabrication de notre prototype.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Item** | **Numéro de pièce** | **Fournisseur** | **Prix** | **Option** | **Commentaires** |
| LPC2478 | LPC2478FBD208 | Avnet | $12.43 |  | SOT459 package |
| SDRAM (16Mbits) | N/D | N/A | $8.00 |  | 32Mbits, CAS=2 |
| USB A Connector |  |  | $0.60 |  | SMT |
| Power connector | PJ-002A | Digikey | $0.38 |  | 2.5mm ??? |
| SD card socket |  | N/A | $2.00 |  |  |
| Crystal 16MHz |  |  |  |  | CSM-7X footprint, 11.4x4.8mm |
| DB9 |  |  |  | x |  |
| MAX232 | 595-MAX232DWR | mouser | $0.88 | x | SOIC-16pins, 3.9mm width, 5V |
| Standoff |  | futurlec? | $2.00 |  |  |
| Plexiglass? |  |  | N/D |  |  |
| PCB | 2.50/sq. inch | BatchPCB | $10.00 |  |  |
| VGA DAC | ADV7125KSTZ50 | Avnet | $5.88 |  | Ça ou R2R |
| Audio DAC | MAX9850 | Digikey | $2.79 |  | Ça ou DAC interne au LPC2478 |
| Regulateur 5V | 863-LM2576D2T-005G | mouser | $2.62 |  | Switching regulator 5V 3A (D2PAK) |
| Regulator 3.3V | 863-NCP1117DTAG | mouser | $0.42 |  | Régulateur 3.3V 1A, TO-252 package |
| Protection USB | LM3525M-L-ND | Digikey | $2.32 |  | Single port |
| Diode schottky | MBRA340T3G (2x) | mouser | $0.80 |  |  |
| VGA D-Sub 15 | A35116-ND | Digikey | $1.76 |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | **Total** | $52.88 |  |  |

Dans cette liste, les condensateurs, les résistances et autre composants passifs et quelques connecteurs tels une prise d’insertion pour une carte SD, qui seront fort probablement utilisés ultérieurement, ne sont pas inclues. On peut quand même voir que nous dépassons déjà le prix de fabrication fixé de 50$. Par contre, une sélection plus rigoureuse des composants et des fournisseurs ainsi qu’un prix réduit pour l’achat de plusieurs composants du même modèle devraient nous tenir aux alentours d’une valeur de 50$ incluant les composants non mentionnés dans la liste.

Notons finalement que la conception des circuits d’alimentation, de communication RS232, de communication avec une carte mémoire SD et de protection du canal USB n’a pas été faite durant le projet et est strictement basée sur le schéma de la plaquette de développement BLUESCREEN SUN7 de la compagnie ThaiEasyElec.

# OPTIMISATION DES CHOIX MATÉRIELS AUDIO ET VIDÉO

## Conception d’un adaptateur pour périphériques audio et vidéo

Durant le projet, une décision par rapport au choix du produit final a été prise. Cette décision vient du fait que la compagnie ThaiEasyElec nous a fournit et livré gratuitement une plaquette de développement BLUESCREEN SUN7 lorsque l’ancienne s’est avérée inutilisable. Afin de respecter une certaine éthique de travail, nous avons décidé de ne pas produire, dans un but commercial, une console basé en grande partie sur leur conception, et qui serait vendu à un coût moindre. En remplacement, nous ferons la conception d’un adaptateur vidéo VGA et audio pouvant se fixer sur la plaquette de développement BLUESCREEN. De plus, ce changement de direction nous donnera plus de chance d’avoir un produit final de bonne qualité, étant donné nos connaissances peu développées en conception de circuit imprimé et en mémoire externe.

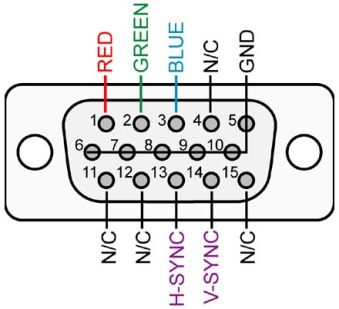
La conception de cet adaptateur nous permettra donc d’avancer dans le projet, tout en nous permettant de prendre de l’expérience et de faire une meilleure conception vis-à-vis du côté périphérique audio et vidéo de la console que nous voulions concevoir à l’origine. En résumé, il sera toujours possible de réutiliser le travail fait dans la conception de cet adaptateur dans un projet de console de jeux.

Reste maintenant à reprendre les choix fait préalablement afin de continuer notre travail. Il nous faut donc établir un choix entre un DAC de type R2R ou un DAC sur circuit intégré pour la sortie VGA et faire un choix entre la sortie analogue du microcontrôleur ou encore utiliser son périphérique I2S afin de communiquer avec un circuit intégré externe pour notre sortie audio. Il est à noter que le BLUESCREEN utilise aussi un LPC2478 comme microcontrôleur.

## Fonctionnement du signal vidéo VGA

Afin de faire des tests sur le choix des composants, il est nécessaire de bien connaître le fonctionnement d’une communication de type VGA. Premièrement, VGA est un acronyme en anglais pour « Video Graphic Array ». Son mode de fonctionnement est tiré du principe des vieux écrans cathodiques qui utilisent, pour résumer, un faisceau d’électrons pour balayer un écran revêtu d’une couche électroluminescente afin de créer les différentes teintes et couleurs à la surface de l’écran.

Le signal VGA contient donc un signal de synchronisation vertical, un signal de synchronisation horizontal et 3 signaux analogiques représentant l’intensité des couleurs rouge, bleu et vert de chaque pixel de l’écran. Les différents niveaux d’amplitude de ces 3 seules couleurs par pixel permettent de recréer toute la gamme des couleurs vue par l’œil humain.



Le signal de synchronisation horizontal est utilisé, dans le cas d’un écran cathodique, afin de donner le signal au contrôleur du faisceau d’électron afin de le ramener au début de la ligne suivante. Le faisceau balayant l’écran de gauche à droite et de haut en bas. Lorsqu’on veut rafraîchir l’écran, c'est-à-dire ramener le faisceau d’électrons complètement en haut à gauche, on utilise le signal de synchronisation verticale. Pour les nouveaux écrans n’utilisant plus de faisceau d’électrons, les mêmes signaux sont quand réutilisés par le contrôleur interne afin de rafraîchir chaque pixel de l’image dans le même ordre de balayage. Ces signaux de synchronisation sont de niveau TTL.

La génération des signaux de synchronisation doit donc être faite avec une certaine précision afin d’éviter un scintillement et un décalage entre chaque ligne de l’image. Il est aussi important de retenir que les signaux de couleurs doivent être remis à 0 lorsque vient le temps de recommencer une nouvelle ligne. Ceci peut se traduire par le fait qu’il fallait éteindre le faisceau d’électrons, sur les écrans cathodiques, durant la traverse en diagonale vers la ligne suivante ou pour revenir en haut à gauche de l’écran à la fin d’une image.

t3

t2

Video

t1

t5

t4

HSync

Le signal de synchronisation horizontal (HSync) est actif bas, ce qui veut dire qu’un changement de ligne survient lorsque celui-ci est mis à zéro. Lorsque le signal vidéo est à 0, les 3 signaux analogiques de couleurs doivent être mis à 0. Les temps haut et bas pour le signal vidéo et pour les signal de synchronisation variera en fonction de la résolution de l’image à l’écran. Pour de petites résolutions, on peut aussi augmenter la valeur t1 tout en rafraîchissant moins souvent les valeurs des pixels, ce qui nous donnera l’impression d’une image étirée. Le taux de rafraîchissement des pixels est aussi différent en fonction de la résolution de l’image affichée.

Pour sa part, le signal de synchronisation vertical, VSync, survient lorsqu’il est temps de rafraîchir l’image à l’écran. Il est lui aussi actif bas et il changera aussi en fonction de la résolution désirée. Il est par contre à une fréquence beaucoup plus basse que HSync et doit rester bas pendant quelques cycles. Pour une résolution de 640x480, cette durée est de deux cycles de HSync. Le signal vidéo est aussi mis à 0 lorsque VSync est bas.

HSync

t9

t8

Video

t7

t6

VSync

L’information sur le VGA est peu répandue, et lorsqu’on en trouve, celle-ci n’est souvent pas très claire et certaines sources d’information se contrarient. Néanmoins, nous avons quand même dressé une liste des différents temps de cycles trouvés et qui semble bien fonctionner pour une résolution de 640x480 pixels à une fréquence de rafraîchissement d’image de 60Hz. Cela permet de se donner une idée de l’ordre de grandeur de ces différents signaux.

|  |  |  |
| --- | --- | --- |
|  | **Nom commun** | **Durée** |
| t1 | Active video time | 25.17us |
| t2 | Front porch | 0.94us |
| t3 | Back porch | 1.89us |
| t4 | Scanline time | 28.00us |
| t5 | Hor. Sync pulse | 3.77us |
|  |  |  |
| t6 | Total frame time | 16.62ms |
| t7 | Vert. Sync pulse | 0.06ms |
| t8 | Front porch | 0.35ms |
| t9 | Back porch | 1.02ms |

En plus des informations sur la temporisation des signaux il faut respecter la valeur de tension analogique maximale des signaux de couleurs. Les signaux analogiques doivent varier entre 0 et 0.7 V. Aussi, il faudra tenir compte de la résistance de charge typique d’une valeur de 75Ω pour chacun de ces signaux analogiques.

## Analyse d’un circuit R-2R pour la sortie vidéo

Afin d’obtenir un signal vidéo analogique, nous avons testé un circuit de type R-2R afin d’analyser ses fonctionnalités et la quantité de composants nécessaires à sa réalisation, étant donné que nous désirons toujours minimiser le coût de fabrication. Rappelons d’abord un peu le fonctionnement de ce type de montage. Un circuit R-2R est un circuit comprenant des résistances montées en échelle. Le circuit contient autant d’entrée que le nombre de bits contenu dans le signal numérique à convertir.



La tension de sortie dépendra du nombre binaire en entrée, Bin, et de la tension haute des bits, Vref.



Afin d’obtenir une variation le plus linéaire possible, et ainsi conserver l’information sur la couleur la plus juste qu’il soit, il est nécessaire d’utiliser des résistances d’exactement la même valeur. En pratique, cela s’avère impossible. On peut donc songer à utiliser des résistances ayant la plus petite marge d’erreur possible quant à leurs valeurs. Il est aussi important d’utiliser des résistances 2R ayant exactement le double de la valeur des résistances R.

Tout ça peut sembler bien compliquer à mettre en pratique. De plus, utiliser des résistances ayant une faible tolérance de variation de leur valeur typique ferait nécessairement augmenter le coût de fabrication. Par contre on peut supposer qu’en utilisant des résistances provenant du même lot de fabrication et ayant une tolérance plus élevée, on aura plus de chance de trouver des résistances identiques. On voudra aussi utiliser deux résistances en série pour les valeurs de 2R afin d’avoir exactement les valeurs doubles de R.

Une autre solution possible serait d’utiliser des réseaux de résistances pré-assemblés. Ces circuits sont peu dispendieux et sont disponible en format SO-M. Ils n’ont par contre pas été étudié durant ce projet.

La valeur des résistances utilisée pourrait être de l’ordre de 1Kohm pour R. On ne veut pas choisir une valeur trop basse pour ne pas demander trop de courant des sorties du microcontrôleur, et on ne veut pas une valeur trop grande pour éviter que des petites fluctuations de courant dû à du bruit possible sur le signal, ne se transmette par une plus grande variation de tension de sortie.

## Ajustement de la tension de sortie pour les signaux analogiques

En utilisant un circuit R-2R pour convertir notre signal numérique, nous somme limité à une variation de tension en sortie du circuit de 0 à 3.3V. Rappelons-nous que le standard VGA demande un signal analogique d’entrée variant entre 0 et 0.7V. Il nous faudra donc utiliser un circuit d’adaptation qui servira de suiveur, en ayant une impédance d’entrée élevée et de convertisseur de tension avec une entrée variant de 0 à 3.3V pour une sortie le plus linéaire possible variant de 0 à 0.7V.

Au début du projet, nous pensions utiliser un amplificateur opérationnel pour exécuter cette tâche. Par contre, on s’est vite rendu compte que ces derniers ne feraient pas le travail demandé. Les amplificateurs opérationnels standard, s’ils sont alimenté, dans notre cas, entre 0 et 3.3V donneront un signal de sortie linéaire à partir d’une certaine tension de sortie. Une solution plus simple, est d’utiliser un transistor bipolaire NPN en mode linéaire.

Évaluons d’abord la fréquence de variation du signal numérique en entrée.



Notre circuit devra donc être capable de répondre à une variation de plus de 1,3MHz. Supposons pour l’instant qu’on utilise le transistor BBT5088 pour des fins de calculs. Il faut donc calculer les valeurs des résistances à insérer autour de celui-ci.

Pour qu’un transistor opère en régime linéaire, il doit pouvoir consommer, du collecteur, un courant supérieur ou égal au courant consommé par la base multiplié par son gain hfe. En supposant une tension d’alimentation de 3.3V au collecteur et une variation entre 0 et 3.3V comme signal d’entrée à la base, et en supposant que nous voulons un courant au collecteur de l’ordre de 10mA.





Pour une tension d’entrée de 3.3V, on désire une tension de 0.7V aux bornes de RL. Le courant maximal en sortie sera de :



Selon la fiche technique du transistor utilisé, nous avons une tension Vbe=0.8V. Toujours pour une tension d’entrée de 3.3V, nous aurions à la base une tension de 0.7V + 0.8V = 1.5V.

Pour rester en mode d’opération linéaire, on veut la relation suivante :





On doit donc avoir Rb plus grand que 63kΩ.

Maintenant la valeur réelle de Rb afin d’obtenir une tension de 0.7V en sortie pour une entrée de 3.3V :



La valeur calculée pourrait être atteinte à l’aide d’un potentiomètre.

## Analyse d’un DAC vidéo sur circuit intégré

Une deuxième option en ce qui concerne la conversion du signal numérique en un signal analogique pour la sortie VGA est l’utilisation d’un DAC sur circuit intégré. Le modèle choisi pour l’étude est l’ADV7125. Voici une liste de ses caractéristiques :

* Contient 3 DAC 8 bits
* Entrée compatible TTL
* Alimentation simple de 5V ou 3.3V
* Nécessite une horloge de minimum 50MHz
* Peut recevoir un signal d’extinction vidéo (blank)
* Conçu pour les signaux vidéos

En utilisant ce circuit, il faudrait donc générer une horloge à partir du microcontrôleur. Un avantage est l’utilisation d’un signal « blank » afin d’éteindre les sortie vidéo juste avant un changement de ligne à l’envoie du signal de synchronisation horizontal. Le fait de dédier cette manipulation au circuit intégré, évite d’avoir à gérer l’extinction et la réalimentation des 3 signaux de couleurs à l’intérieur du microcontrôleur. L’avantage principal est d’éviter les décalages en bordure gauche de l’écran lors d’une réalimentation successive et non synchronisé des 3 signaux de couleurs.

D’autres avantages à utiliser un tel circuit sont sa facilité de mise en place lors de la fabrication, le peu d’espace requis, et la diminution du risque d’avoir du bruit sur notre signal.

## Analyse d’un signal I2S pour la sortie audio

# architecture logiciel

## Mise en perspective

Afin de bien comprendre le travail à faire, voici une représentation sous forme de couche de la section logicielle.

Figure - Présentation logiciel sous forme de couche

Pour assurer un bon fonctionnement de la console de jeux, nous devrons minimalement fournir 4 modules de bas niveau sur lesquels pourra reposer le moteur de jeux. Le module USB sera dédié à l’entré utilisateur. Dans notre cas, c’est ce pilote qui assurera l’interface avec la manette de Xbox360. Ensuite le module du système de fichier inclus aussi l’interface avec la carte mémoire SD. Bien que lors de l’implémentation ceci sera constitué de plusieurs petits modules, nous le représentons ici comme un seul bloque pour plus de simplicité. Il y aura aussi un module graphique pour la communication avec le contrôleur LCD. Pour finir, un mixeur sonore est utilisé afin de pouvoir jouer le son d’arrière plan et les sons d’avant plan en même temps.

## Vue d’ensemble de la couche d’abstraction

Comme mentionné à la section précédente, le moteur de jeux trouve encrage sur une couche logiciel permettant de le rendre indépendant du matériel sur lequel il repose. Nous appelons cette couche, la couche d’abstraction matérielle.

Nous avons utilisé cette façon de faire pour plusieurs raison, une de ces raisons est la possibilité d’utiliser le moteur de jeux sur plusieurs plateformes, plus précisément sur PC. En effet, comme l’objectif de ce projet est de permettre la réalisation de jeux facilement et rapidement, il serait souhaitable que le développement de jeux puisse prendre part sur PC pour accélérer le développement. À la page suivante, vous trouverez une représentation sous forme de diagramme de classes de cette couche d’abstraction.

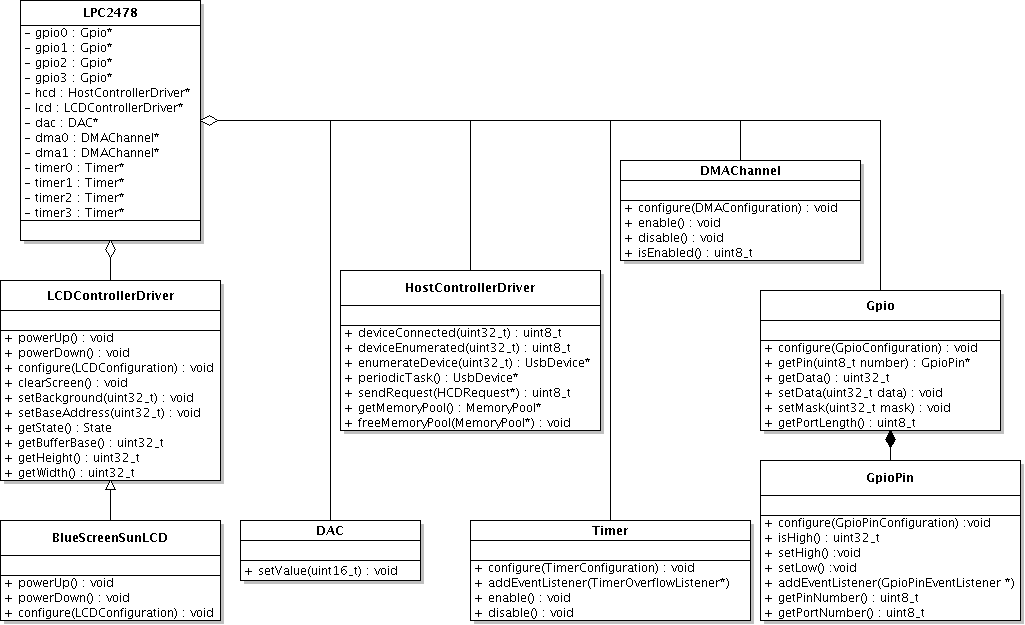


Figure - Diagramme de classes de la couche d'abstraction

Comme vous le constatez sur le diagramme, chacun des périphériques utilisés par le moteur de jeux est représenté par une classe correspondante. Le reste du code ne font donc jamais appel au périphérique directement. À la place, il demande au module logiciel de le faire. Vous remarquez aussi une classe nommé LPC2478 qui contient une référence à chacun des périphérique du microcontrôleur. En effet, il ne peut y avoir qu’une seul instance de la classe Timer pour le timer 0, une seule pour le timer 1, idem pour tous les périphériques. Il faut donc une classe pour répertoriez ces instance et c’est ce que fait LPC2478.

Voici un exemple pour mettre tout ça en perspective.

Gpio \*gpio1 = LPC2478::getGpio1();  
  
// Set default port behavior  
GpioConfiguration portConfig(32, Gpio::INPUT);  
gpio1->configure(portConfig);  
  
// Configure blinking led  
GpioPinConfiguration ledPinConfig;  
ledPinConfig.pin = Gpio::OUTPUT;  
gpio1->getPin(12)->configure(ledPinConfig);  
  
GpioPin \*led = gpio1->getPin(12);  
  
// Blink led  
while(1) {  
 led->setHigh(); // On  
 for(uint32\_t i=0; i<100000; i++);  
 led->setLow(); // Off  
 for(uint32\_t i=0; i<100000; i++);  
}

Ce cas est le classique où nous voulons faire allumer et éteindre une LED. Pour ce faire, nous demandons à la classe LPC2478 de nous transmettre un pointeur à l’instance de la classe faisant la gestion du GPIO1. Une fois que nous avons cette instance, il nous est possible de spécifier une configuration pour les broches ce port. Nous choisissons ici de les mettre toutes en sortie mais nous aurions pus configurer chacune des broches individuellement. Nous spécifions aussi que le port a une taille de 32 broches. Ensuite il est possible de demander une pointer vers l’instance d’une des broches de ce port et c’est ce que nous faisons dans la seconde section de l’exemple. Nous configurons cette broche en sortie. Ensuite, nous pouvons utiliser instance pour allumer ou éteindre la LED en appelant la fonction setHigh() et setLow() de le classe.

Remarquez qu’en aucun cas nous avons configuré de registre. Cette tâche est assurée par la classe correspondant au périphérique. C’est ce genre de fonctionnement que nous allons avoir dans la totalité du code.

# réalisation matériel

## <Titre>

<Texte>

# Réalisation logiciel

## Système de fichier

Rappelons que le but de ce projet est ultimement de faire une console de jeux vidéo. Pour pouvoir enregistrer le jeu il faudra une unité de stockage. Celle que nous avons choisie est la carte SD. Pour enregistrer le jeu sur cette carte plusieurs système de fichier s’offre à nous. Cependant, un système de fichier se démarque par sa popularité pour les unités de stockage amovible soit le Fat32.

Faire une bonne implémentation de Fat32 est assez compliqué surtout dans un espace restreint comme dans un environnement embarqué. C’est pourquoi nous avons décidé d’utiliser FatFs pour faire le travail. Cette implémentation est très légère autant au niveau de la taille du code que de la mémoire utilisé. De plus, elle peut être portée sur plusieurs plateformes très facilement. Tout ce qu’il faut faire c’est implémenter quelques fonctions qui permette à FatFs d’accéder à l’unité de stockage peut importe sa nature. Pour bien vous mettre en contexte, voici une image venant directement du site officiel de FatFs démontrant ou se situe le module.

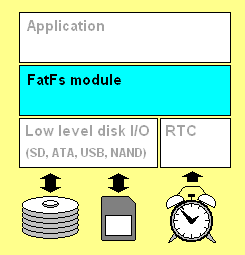


Figure – FatFs

Par contre, cette application ne règle pas tous nos problèmes. En effet même si nous sommes en mesure d’enregistrer et de charger des fichiers sur notre unité de stockage, nous n’avons aucun support pour aucun type de fichier. Dans les sections qui suivent, nous allons regarder comment nous avons ajouté le support pour les fichiers Bitmap pour les images et Wave pour les sons.

### Fichier Bitmap

Comme ce format de fichier n’est pas compressé, il est assez facile d’en ajouter le support. C’est d’ailleurs pourquoi nous avons choisi d’utiliser ce format. De plus, c’est un type de fichier assez commun qui est le standard de plusieurs petits programmes d’édition d’images comme Microsoft Paint.

Premièrement il est important de comprendre que peut importe le type de fichier, le principe est le même. Le fichier est composé d’une entête d’un certain format suivit de l’information du fichier. Lorsque nous ajoutons le support à un certain type de fichier il faut être capable de reconnaitre cet entête et de se servir de l’information pour extraire les données du fichier. Dans notre cas, nous avons fait une classe Bitmap prenant le chemin vers un fichier du même type en paramètre du constructeur. Cette classe extrait l’information de l’entête et organise cette information en mémoire afin qu’elle soit facilement accessible par le reste de l’application. L’entête d’un fichier Bitmap est divisé en 3 parties et se présente comme suit.

Table - Entète de fichier BMP

|  |  |  |
| --- | --- | --- |
| Magic | | |
| Offset | Taille | Description |
| 0000h | 2 octets | Contient : ‘BM’ |
| Header | | |
| Offset | Taille | Description |
| 0002h | 4 octets | Taille du fichier en octets |
| 0006h | 2 octets | Réservé |
| 0008h | 2 octets | Réservé |
| 000Ah | 4 octets | Offset de l’information de l’image |
| InfoHeader |  |  |
| Offset | Taille | Description |
| 000Eh | 4 octets | Taille de l’entête (40 octets) |
| 0012h | 4 octets | Largeur de l’image |
| 0016h | 4 octets | Hauteur de l’image |
| 001Ah | 2 octets | Nombre de palette de couleur (1) |
| 001Ch | 2 octets | Nombre de bit par pixel |
| 001Eh | 4 octets | Type de compression |
| 0022h | 4 octets | Taille de l’image en octets |
| 0026h | 4 octets | Pixels par mètre X |
| 002Ah | 4 octets | Pixels par mètre Y |
| 002Eh | 4 octets | Nombre de couleur |
| 0032h | 4 octets | Nombre de couleur importante |

Comme mentionné précédemment, l’information du fichier Bitmap n’est pas compressée. Nous pouvons donc lire le contenu et s’en servir directement pour afficher l’image à l’écran. Il est à noter que les lignes de l’image sont enregistrer en sens inverse dans le ficher. La première ligne est donc en bas du fichier et la dernière ligne en haut. De plus, chaque ligne est remplie de façon à ce que le nombre d’octet sur chaque ligne soit un multiple de 4. Ceci à été pris en compte lors de l’implémentation

### Fichier Wave

Tout comme le fichier Bitmap, le fichier Wave contient un entête faisant une description de l’information. L’entête est la suivante.

|  |  |  |
| --- | --- | --- |
| RIFFHeader | | |
| Offset | Taille | Description |
| 0000h | 4 octets | Contient : ‘RIFF’ |
| 0004h | 4 octets | Taille du fichier moins la taille de cette entête (8) |
| WAVEHeader | | |
| Offset | Taille | Description |
| 0008h | 4 octets | Contient : ‘WAVE’ |
| 000Ch | 4 octets | Contient : ‘fmt ’ |
| 0010h | 4 octets | Taille du reste de l’entête (0x10) |
| 0014h | 2 octets | Format (PCM = 0x01) |
| 0016h | 2 octets | Nombre de canaux (mono = 0x01, stéréo = 0x02) |
| 0018h | 4 octets | Taux d’échantillon |
| 001Ch | 4 octets | Octet par seconde |
| 0020h | 2 octets | Alignement des bloques |
| 0022h | 2 octets | Bit par échantillons |
| DataHeader | | |
| Offset | Taille | Description |
| 0024h | 4 octets | Contient : ‘data’ |
| 0028h | 4 octets | Nombre d’octets d’information |

Comme le fichier Wave n’est pas non plus compressé, l’information peut être directement utilisée pour générer le son. Cependant, il est important de noter que lorsque nous utilisons 8 bits par échantillons les valeurs sont enregistré sous forme non signé. Par contre, avec 16 bits par échantillons les valeurs sont enregistrées sous forme signé.

## Affichage

Comme mentionné précédemment, la classe LCDControllerDriver de la couche d’abstraction s’occupe de l’initialisation du contrôleur LCD. Une fois celui-ci initialisé, il est très facile d’afficher des images à l’écran. En effet, le contrôleur LCD utilise un canal DMA qui lui est réservé pour transférer l’information se trouvant dans la mémoire vidéo jusque sur l’écran. L’emplacement de la mémoire vidéo est spécifier par l’utilisateur et peu être modifier lors de l’exécution. Il est donc possible d’utiliser une technique appelé « double buffering ». Nous reviendrons sur cette technique un peu plus loin dans cette section. Pour en revenir à la mémoire vidéo, celle-ci n’est qu’un très grand tableau qui possède une case de 32 bits par pixels. L’affichage d’un fichier Bitmap peut donc se faire d’une seule opération de copie.

La fameuse technique du « double buffering » est en réalité très simple. Tout ce qu’on fait c’est utiliser deux mémoires vidéo identique pour le rafraichissement de l’écran LCD. L’idée c’est d’afficher une des mémoires tandis que nous faisons nos modifications sur l’autre. Ensuite, nous inversons les rôles. Ceci empêche de rafraichir directement l’information affichée à l’écran et de rendre celle-ci instable. C’est une technique très utilisé dans le monde du jeu vidéo et elle peut être facilement implémentée avec notre contrôleur LCD. Il suffit de mettre à jour le pointeur vers la première case de la mémoire vidéo lors que nous voulons changer de tampon.

### Utilisation de l’écran comme console

Dans le cas d’affichage des éléments du jeu, l’information est écrite directement dans la mémoire vidéo pour maximiser les performances. Cependant dans la phase de développement il pourrait être intéressant d’afficher de l’information à l’écran sous forme de texte et ce sans trop d’effort. Nous avons donc créé quelques classes qui s’occupent de ce travail. Commençons par regarder le diagramme de classe.

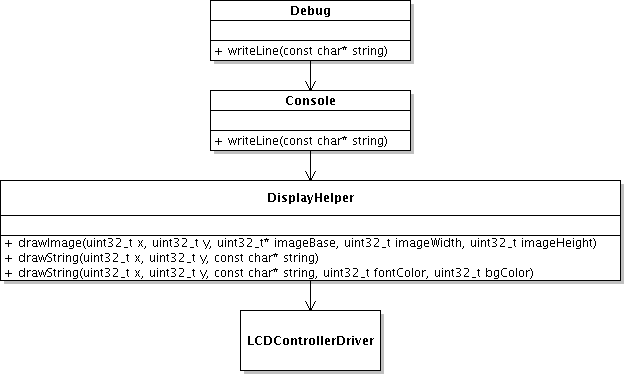


Figure - Classes d'aide à l'affichage

La classe qui fait le plus grand travail est la classe DisplayHelper. Cette classe offre la possibilité d’afficher une image ou du texte peut importe l’endroit à l’écran. Une fois cette tâche réglé, il serait intéressant de pouvoir utiliser l’écran comme une console. Par la, nous voulons dire afficher des lignes de textes les unes en dessous des autres et faire défiler le texte quand nous avons atteint la bas de l’écran. Cette tâche n’est pas triviale car l’écran n’est pas normalement divisé en ligne de texte. Souvenons nous que la classe DisplayHelper affiche le texte peu importe l’endroit dans l’écran. Le but ici est de ne pas afficher de texte un par-dessus l’autre. Il faut aussi garder un historique des lignes de texte pour faire défiler le texte. C’est ce que fait la classe Console. Cette classe offre une fonction prenant une ligne de texte et s’occupe de la faire afficher à l’écran. Il ne reste qu’un problème et c’est qu’il ne doit exister qu’une instance de la classe Console par écran physique sinon il y aura superposition. Pour régler ce problème nous utilisons la classe Debug qui offre une fonction statique pouvant être appelé de n’importe où dans le code. Cette classe s’occupe de n’utiliser qu’une seul instance de la classe Console.

## Son

Dans la section sur le système de fichier, nous avons vue comment était enregistré un fichier WAVE. Nous avons vue que l’entête nous fournis toute l’information dont nous avons besoins et que le fichier n’est pas compressé. Il reste cependant plusieurs problèmes à résoudre. Non seulement nous voulons jouer des sons mais en plus nous voulons les faire jouer de façon asynchrone. De plus, nous voulons possiblement faire jouer plus d’un son en même temps. C’est ce que fait la classe d’aide AudioHelper. Encore une fois, regardons le diagramme de classe.

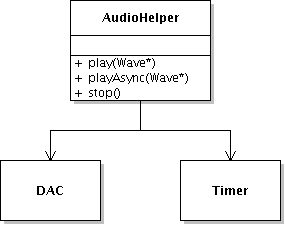


Figure - Classe d'aide au son

Comme nous le voyons cette section est plus simple que la précédente. La classe offre principalement une fonction de génération de son synchrone et asynchrone. Les deux types de génération utilisent la classe DAC comme sortie. Par contre, la version asynchrone utilise aussi les interruptions d’un timer. La version asynchrone offre aussi la possibilité de jouer plusieurs sons en même temps. C’est ce que nous appelons mixeur sonore. L’implémentation est faire en faisant une simple addition des signaux sonores. Cette façon de faire oblige une diminution du volume de chacun des sons. Nous ne pouvons donc jouer qu’un nombre prédéterminé de sons maximum en même temps. Pour le moment la limite est de 2 ce qui permet de jouer un son d’arrière plan et un d’avant plan. Par contre cette limite peut être facilement modifiée. Cependant, pour des raisons d’optimisation, le nombre de son doit être une puissance de 2. Ceci est dû au faite que notre microcontrôleur n’a pas de module de division matériel. Nous devons donc remplacer un maximum de division par des décalages binaires.

## USB

La section USB est une des grosses sections du rapport. En bref, nous avons implémenté une pile USB basé sur le contrôleur matériel standard OHCI. OHCI est une spécification développé par une association de Compaq, Microsoft et National Semoconductor en 1999. Cette norme spécifie comment le contrôleur matériel doit être implémenté ainsi que la première couche logicielle interagissant avec le contrôleur. Comme l’implémentation d’une pile USB requiert beaucoup de code, nous n’allons pas entrer dans les détails pour ne pas alourdir le rapport. Si vous voulez plus de détails vous pouvez toujours consulter le code à l’adresse suivante : <http://code.google.com/p/armconsole/>

Vue de haut niveau, la section USB ressemble à ceci :

Figure - Représentation de la norme OHCI

La partie de nous avons implémenté et qui s’occupe de tous les service USB est la classe HostControllerDriver. Il est à noter que nous avons aussi fait cette section en utilisant une programmation orienté objet en C++. Ceci vient du même coup prouver qu’il est possible de faire des pilotes de bas niveau en programmation objet sans perte d’efficacité.

Lorsqu’un périphérique USB est connecté, le controlleur matériel déclanche une interruption de type « Root hub port status changed ». Le pilote doit donc faire une requête au pont racine pour lui demandé quel est ce changement et sur quel port. Nous verrons ensuite que le changement est un périphérique connecté et nous pourrons démarrer la séquence d’énumération. Cette séquence est définit dans le standard USB 2.0 et son explication sort du cadre de ce rapport. Ce qui est important de comprendre par contre c’est que chaque périphérique USB contient des bloque d’information sous un format standard définit encore une fois par la spécification USB. Ces bloque d’information sont appelés « descriptor ». Lors de l’enumeration nous allons faire plusiseurs requêtes au périphérique afin de connaitre cette information et la sauvegarder dans un objet UsbDevice. Cette objet est générique peut importe le type de périphérique connecté. Durant l’énumération nous allons aussi assigner une adresse unique au périphérique et préparer le contrôleur OHCI matériel à communiquer avec le périphérique. Voici une image pour mettre tout ceci en perspective.

Figure - Énumération USB

### Limitations

Bien que notre implémentation couvre une assez grande partie de la spécification OHCI, il y a tout de même quelque limitation. Premièrement, nous ne pouvons supporter qu’un seul périphérique USB par port racine. Nous ne supportons donc pas les pont USB. Au passage, notez que les pont USB sont appelé pont mais sont probablement mieux représenté par le terme commutateur ou du moins pont intelligent. En effet, ces appareils ne font pas seulement diviser le conducteur comme un pont Ethernet. Vue du pilote, ce sont des périphérique USB avec quelques commandes spéciales pour entre autre connaitre l’état de ses ports. C’est pourquoi nous n’en faisons pas le support.

Nous ne supportons pas non plus les périphériques hautes vitesse (480 Mbps). Ceci est dû à une limitation matérielle.

Pour finir, nous ne supportons pas les transferts de type « isochronous ». Ces transferts sont de type spécial et sont utilisé pour la transmission d’information temps réel qui ne requiert pas de correction d’erreur comme pour la transmission vidéo. Par contre les transmissions de type « Bulk » sont supporté malgré qu’elles ne sont pas utilisé dans le projet.

### Manette de Xbox360

Une fois que l’application détient un objet de type UsbDevice, elle ne peut toujours pas communiquer avec celui-ci. En effet bien qu’à ce stade le type de transmission est connu, il reste à connaitre quel information on doit envoyer et recevoir pour faire fonctionner le périphérique. Par exemple, bien que la manette de Xbox et une souris utilise le même mode de communication, nous nous imaginons bien que les informations échangé ne son pas les même étant donné que ces périphérique sont très différent. C’est pourquoi nous avons besoins d’une autre pilote de plus haut niveau pour prendre en charge le périphérique.

Afin de connaitre quel pilote de haut niveau utilsé, nous pouvons demander à l’objet UsbDevice de quel type est le périphérique qu’il représente en regardant les champs « idVendor » et « idProduct » du device descriptor. Notez que cette façon de faire est standard et est utilisé sur les ordinateur conventionnel depuis les premières implémentations de USB. Voici un exemple de code pour bien mettre en perspective ce dont nous venons d’exposer.

UsbDevice\* device;

XboxControllerDriver\* controller;

while(1) {

device = hcd->periodicTask();

if(device != 0) {

// Xbox receiver

if(device->getDeviceDescriptor()->idVendor == 0x045e

&& device->getDeviceDescriptor()->idProduct == 0x0719) {

MemoryPool\* memoryPool = hcd->getMemoryPool();

if(memoryPool != 0) {

controller = new XboxControllerDriver(device, memoryPool);

break;

}

else {

Debug::writeLine("Not enough USB memory");

}

}

}

LPC2478::delay(100000);

}

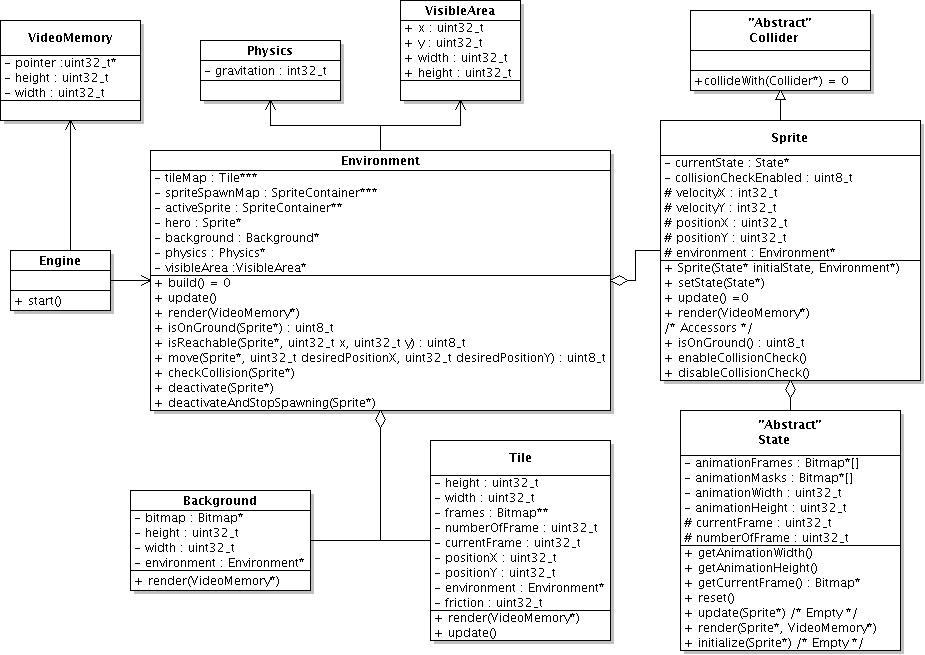
Debug::writeLine("Xbox controller ready to use");

La tâche periodique c’est celle qui va procéder à l’énumération au besoin et retourner une objet UsbDevice si tel est le cas. Ensuite si l’objet représente un récepteur de manette de Xbox, nous allons créer un objet XboxControllerDriver qui est le pilote haut niveau pour la manette de Xbox. Nous pourrons ensuite faire des appel à la manette par ce pilote. Nous pourrons entre autres demander l’état des boutons, changer l’état des LED ou encore faire vibrer la manette.

Pour les curieux, nous devons demander une portions de la mémoire USB au HostControllerDriver car les transmissions entre la mémoire du microcontrolleur et le périphérique ce fait par transfert DMA. Un canal est réservé spécifiquement pour le contrôleur USB. Par contre, ce canal ne peut faire de transfert qu’à partir d’une région de la mémoire très spécifique appelé mémoire USB. Cette mémoire est de 16 kB. Dans notre implémentation c’est le HCD qui gère cette mémoire.

## Moteur de jeux

Passons maintenant à la section de plus haut niveau de la console soit l’implémentation du moteur de jeux. Nous allons voir au cours des différentes sections comment à été conçu ce moteur. Rappelons que l’objectif est de créer un moteur pour les jeux de type « side scroller » afin de simplifier la création de nouveau jeux. Ceci est fait en encapsulant les problèmes commun aux jeux et permettre l’utilisation de ces modules par une interface simple. Commençons par le diagramme de classe.



Comme vous le voyez, c’est la classe environnement qui est au cœur du moteur de jeux. C’est lui contient l’architecture du niveau et qui est capable de savoir si un sprite est sur la « terre » ou non. C’est aussi lui qui peut regarder si deux sprite entre en collision. Les plus vif d’esprit aurons reconnu le patron de conception « état » qui est utilisé pour la gestion du sprite. Ce sont tous des concepte sur lesquels nous allons revenir dans les prochaines sections.

De façon générale, lorsque le moteur de jeux démarre, il crée un nouveau niveau. Un niveau est une instance d’une classe héritant de la classe Environment et qui implémente la méthode abstraite build(). C’est à ce moment que l’environnement du jeu est créé. Lors de l’exécution de la méthode build(), Tous les élément du niveau sont instancié et passé à la super classe Environment. De la physique à la surface visible du jeu. C’est aussi là que les tiles et les sprites sont instancié. De plus, c’est là que va être décidé si l’Environment devrait regarder pour les collisions pour un sprite donné. Pour finir, c’est à la fin de cette fonction que la musique de fond est démarrée.

### Affichage des tiles

Si vous n’avez pas remarqué, ce moteur de jeux est de type gestion de tile. Ceci implique que tous les éléments dans lequel pourrait évoluer notre héro on une taille unitaire fixe. Ceci permet une réutilisation maximal des textures et est donc plus léger pour la mémoire ce qui est crucial dans notre cas. Voici à quoi ressemble le niveau avec la division en tiles :

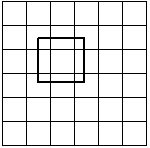


Figure - Division du niveau en tiles

Le carré plus foncé représente la partie visible du niveau à l’écran. En ayant les coordonné de cette partie visible vous voyez qu’il n’est pas très difficile de connaitre quel tuile devront être affichée. Afin de facilité la compréhension, voici un exemple concret sur la console. Avec les deux tiles suivante :



Nous sommes en mesure de créer l’environnement suivant :

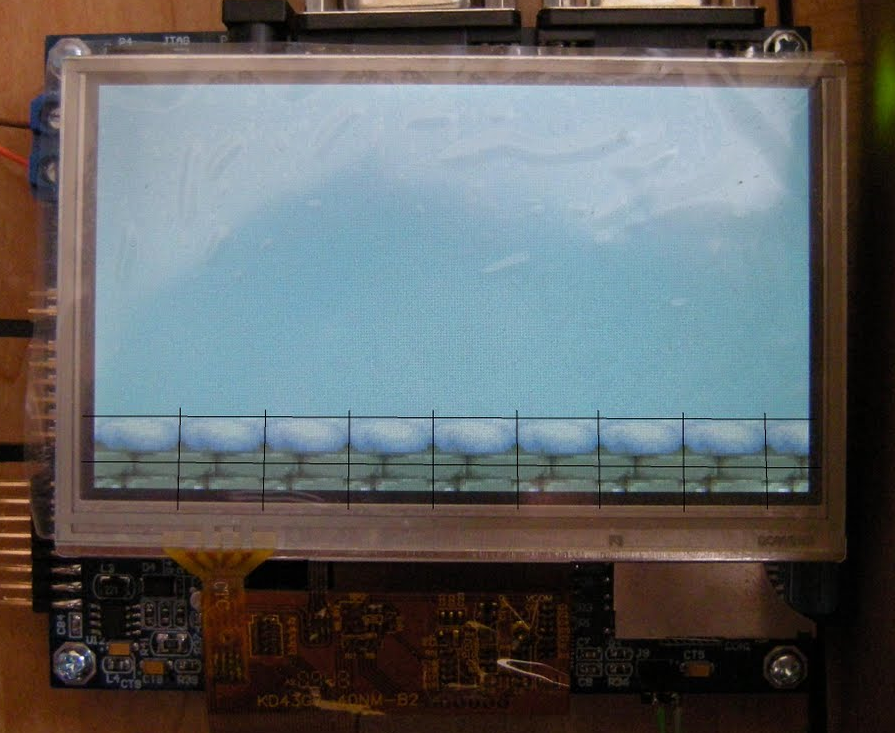


Figure - Exemple concret d'une division en tiles

Pour faire l’affichage proprement dit, lorsque la fonction render() est appelé sur l’Environment, celui-ci calcule quel tuile sont visible selon la classe VisibleArea. Il appel ensuite rendre à sont tour sur les tiles visé et leur indique à quel position elle devrait être affiché à l’écran encore selon VisibleArea. VisibleArea peut être vue comme la caméra du jeu.

### Affichage de l’image de fond

L’affichage de l’image de fond est en fait très simple. Il suffit de faire une mosaïque avec l’image selon la position de la caméra du jeu. Lorsque nous utilisons cette façon de faire, nous voyons qu’en pratique la processeur de 72 MHz n’est pas en mesure de rafraichir l’écran avec une assez grande efficacité. En effet pour les tiles nous ne rafraichissons que les endroits ou il y a des tiles ce qui réduit l’ampleur du rafraichissement. Par contre pour l’image de fond, il faut rafraichir tout l’écran à chaque fois.

Nous avons essayé plusieurs façons d’optimiser le processus et les résultats les plus concluants est un hybride entre une optimisation du code et l’utilisation de canaux DMA. Les canaux DMA permettent de faire des copie en mémoire d’une partie continue de mémoire d’une taille quelconque. Un problème se pose dans notre cas car comme nous ne voulons qu’afficher une portion de l’image de fond, la section de la mémoire n’est pas continue. Par contre, elle est continue pour chaque ligne de l’image. L’astuce est donc assez simple. À chaque rafraichissement, nous calculons la portion de l’image de fond à rafraichir à l’aide du VisibleArea. Ensuite, nous faisons une boucle pour toutes les lignes de l’image qui doit être affiché et nous créons une tâche DMA pour chacune d’elle. Pour finir, nous créons des liens entre chacune de ces tâches et nous activons le canal DMA. Comme ces canaux sont indépendants du processeur, ils sont très rapides. De plus, ils peuvent faire des copies de plusieurs octets en mémoire sans ré-arbitration du BUS. Ceci augment encore plus l’efficacité. Les résultats sont très satisfaisants. Une fois optimisé de cette façon, La partie la plus lente du rafraichissement n’est plus l’image de fond mais les tiles qui ne couvrent même pas toute l’écran.

### Gestion de la gravité

Bien que ce soit la classe Environment qui contient la gravité courante du niveau, celle-ci est gérer par le sprite. Selon son état, il va prendre la décision si est ou non affecté par la gravité. Par exemple, un sprtite volant pourrait ne pas être soumis à la gravité bien qu’il soit dans les air. Lorsque le sprite veut connaitre la gravité, il demande à l’environement dans lequel il évolue. Il est donc possible de changer la gravité n’importe quand pendant le jeu. Ceci pourrait permettre des cas où le héro passe à un endroit qui fait changer la gravité de sens ou encore augmenter ou diminuer celle-ci. La formule utilisée est la simple formule de physique élémentaire.

### Affichages des sprites

Contrairement aux tiles, les sprites peuvent potentiellement se retrouver n’importe où à l’écran. Il faut donc que chacun des sprite connaissent sa position. Bien que le héro et les ennemis soit de même type, leur affichage est gérer un peu différemment et nous allons donc les traiter en deux sections distinctes.

#### Héro

Une des facteurs les plus importants c’est que le héro est toujours présent à l’écran. S’il ne s’y trouve pas, le jeu est terminé. Nous pouvons donc appeler sa fonction d’affichage aveuglément à chaque fois. Une fois appelé, l’affichage est fait comme les autres sprites en fonction de l’objet VisibleArea. Voici un exemple d’affichage du héro. Vous constaté qu’il est entouré d’un carré blanc car à ce stade nous n’avons pas encore de gestion de transparence.



Figure - Affichage du héro

#### Ennemi

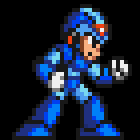
Pour les ennemis, la tâche est un peu plus complexe. Premièrement, les ennemi peuvent être actif ou innactif. En effet, nous voulons que ceux-ci apparaissent dans le jeu a un endroit prédéterminé à chaque fois. Nous avons donc un tableau nous permettant de répertorier les ennemis dans le jeu. Lorsque la caméra découvre une nouvelle partie de l’écran, certains ennemis vont potentiellement apparaitre et donc devenir actif. Une fois actif, ces sprites vont se comporté librement dans l’espace visible jusqu’à ce qu’ils sortent de cette espace. Ils vont donc redevenir inactif.

Pour l’affichage nous allons aller voir dans notre liste de sprite actif et non le tableau comprenant tous les sprites du niveau. De plus nous allons nous assuré que ces sprites actif sont toujours visible et dans le cas échéant, nous allons les désactiver.

#### Gestion de la transparence

La gestion de la transparence est un point assez intéressant. Bien qu’au niveau de la programmation du moteur de jeux ceci ne fût qu’un détail, visuellement c’est très important. Comme le format d’affichage choisi est le Bitmap 24 bits par couleur, nous n’avons pas de moyen d’afficher la transparence. Nous n’avons donc pas de « couleur » transparente. Une des techniques utilisé pour contourner le problème est d’utiliser une couleur et de la désigné comme couleur transparente. Ceci implique par contre que nous en pouvons plus utiliser cette couleur par la suite. Cette solution ne nous semblait pas très attirante et ce n’est donc pas ce que nous avons utilisé pour le moteur de jeux.

À la place, nous avons utilisé un système de masque. Voici à quoi ressemble une image pour un sprite :



Chacune des images est noir à l’endroit où nous voulons la transparence et chaque image est accompagné d’une autre image qui est le masque. Lorsque nous affichons l’image à l’écran, nous commençons par faire un ET logique entre l’image affichée à l’écran. Rappelons qu’un pixel blanc porte la valeur 0xFFFFFF sur 24 bits et un pixel noir 0x000000. Ceci à pour effet de laisser tout les pixels qui seront transparent de la couleur de l’image présentement à l’écran et de mettre les autres pixels noir. Nous prenons ensuite notre image et nous faisons un OU logique entre cette image et l’image à l’écran où nous avons appliqué notre masque. Comme les pixels que nous voulons transparent sont noirs, ils reste de la couleurs présentement à l’écran. Les autres pixels qui font partie de l’image vont être transférés à l’écran car nous avons appliqué un masque et ces pixel sont maintenant noir donc de valeur 0x000000. Cette technique à l’avantage de n’utiliser aucun saut conditionnel et donc ne vide pas le pipeline. Elle est donc très efficace et en pratique nous avons remarqué que le temps d’affichage avec et sans transparence est a toute fin pratique le même.

Figure - Transparence

### Détection de collision

La gestion de collision est une facette très importante du moteur de jeux. En effet, tout ce que l’utilisateur veut c’est d’être avertis lorsqu’il y a collision sur un de ses sprites. Il ne veut pas avoir besoin d’interagir avec les algorithmes de détection. De toute façon ces algorithmes sont bien connus et il n’existe que quelques techniques différentes.

Dans l’optique de rendre le développement de jeux facile, le rôle du moteur de jeux est d’encapsuler cette détection de collision. Voici une description du fonctionnement. Lorsque l’utilisateur crée le niveau, il va activer un attribue sur les sprite dont il desire connaitre les collisions. Ceci a pour but de limiter la tâche de détection du moteur car il n’a pas besoin de regarder pour les collisions de chaque sprite. Dans la plupart des jeux, les ennemis n’interagissent par entre eux et donc les collisions ne sont pas activé sur ceux-ci. Dans ce cas, les collisions sont activées seulement sur le héro. À chaque mise à jour, l’objet Environment va regarder tous les sprites dont l’attribue de détection de collision est activé. Pour chacun de ces sprite il va regardé s’il entre en collision avec n’importe quel autre sprite indépendamment de leur attribue de collision. Ensuite, il va avertir les deux sprites de la collision. Le système de gestion de collision est vue plus un détail dans la section « Gestion de la détection de collision ».

## Démo de jeu

Afin de démontrer les possibilité du moteur de jeu il était nécéssaire de développer un petit démo. Le thème choisi fût le très célèbre Megaman X de Capcom. Voir en détail l’implémentation de ce démo serait sortir du cadre de ca laboratoire. Par contre, nous allons regarder de plus près deux points notables de l’implémentation dans les deux sections suivantes.

### Gestion des états des sprites

Comme nous le savons, les sprites se comportent différemment selon si ils courent, saute vole, glisse, etc. Il faut être capable de gérer ces états et ces changements d’état en temps réel lors du déroulement du jeu. Pour ce faire, une des solutions évidente est l’utilisation d’une machine à état finit. C’est ce que nous avons fait pour notre démo de jeu. Cependant, comme nous utilisons un langage orienté objet, nous avons utilisé plus précisément le patron de conception « état ». Voici ce que ça donne sur un diagramme de classe.

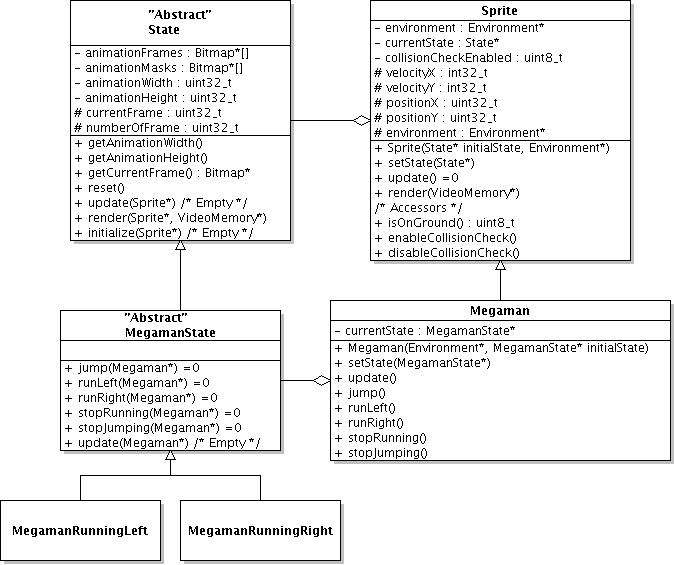


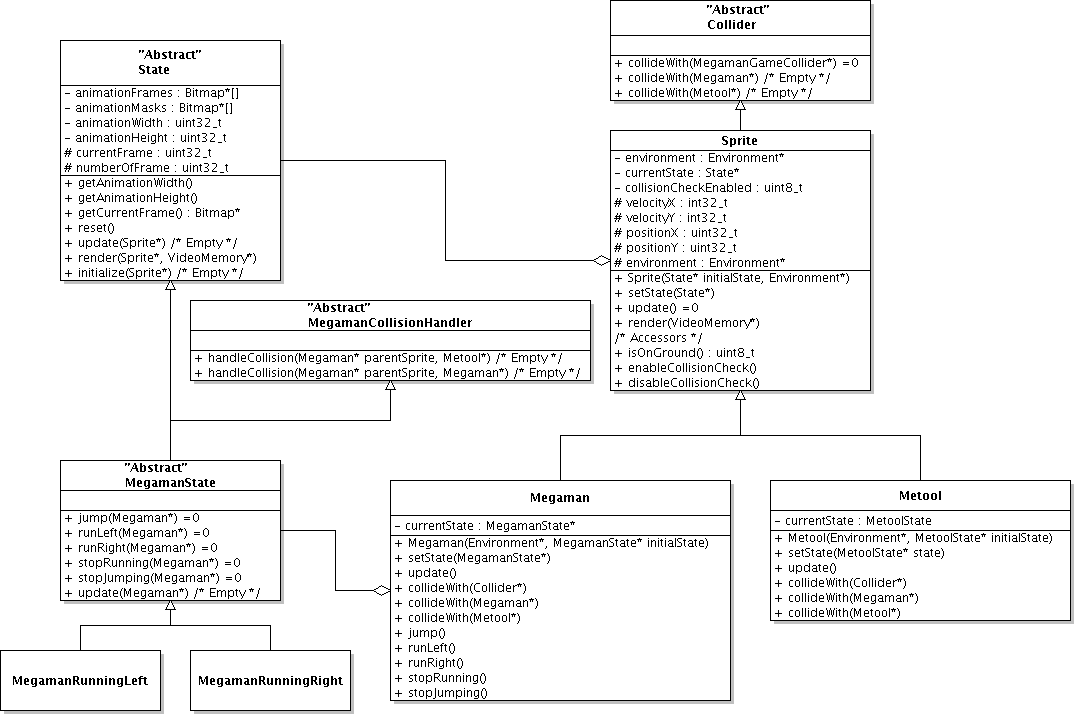
Figure - Gestion des états

Sur ce diagramme nous les classe Sprite et State on chacun leur sous-classe spévifique Megaman et MegamanState. Ensuite nous avons plusieurs sous-classe pour MegamanState qui représente tous les états possibles de ce sprite. Chacun des état est responsable de son comportement lors de l’appelle aux différentes fonction de MegamanState. L’appelle à une de ces fonctions peut aussi provoquer un changement d’état.

Dans cette implémentation, la classe Megaman est ce qu’on appelle le contexte. Cette classe ne fait rien autre que contenir les informations du sprite. Lorsqu’on fait appel a une fonction sur Megaman, celui-ci relais cette appel à la même fonction de son état courant. Cette façon de faire évite les énorme switch/case utilisé pour les machines à état finit dans les code procédurale.

### Gestion de la détection de collision

Nous en avons discuté précédemment, c’est la classe Environment qui s’occupe de la détection de collision. Le problème c’est que la classe Environment ne connait pas le sous-classe de la class Sprite. Pour elle il n’existe que des sprites. Il serait pas contre intéressant lorsque Megaman entre en collision avec un objet ennemi Metool que la fonction Megaman.collideWith(Metool\*) soit appelé. Comme ça nous savoir dès l’entré de la fonction avec quoi nous avons entré en contact. Pour résoudre le problème nous avons utilisé une version légèrement modifié du patron de conception visiteur. Voici le diagramme de classe.



Pour commencer la classe Sprite est un Collider donc une chose qui peut entrer en collision avec quelque chose d’autre. Lorsque la classe Envirionment détecte une collision entre deux sprite elle appelle la fonction collideWith(Collider\*) des deux sprites avec l’autre sprite en paramêtre de la fonction. Comme cette fonction est abstraite, elle est implémentée par la sous classe. Dans notre cas, Megaman ou Metool. Tout ce que ces fonction font c’est faire un appel inverse soit Collider.collideWith(this). Comme nous sommes dans la sous classe, le pointer « this » représente une instance de Megaman ou Metool et non plus cette fois un simple Collider générique. Lorsque cette appel inverse va prendre place, ce n’est donc plus la fonction collideWith(Collider\*) qui sera appelé mais une des deux fonctions collideWith(Megaman\*) ou collideWith(Metool\*). De cette façon, nous savons directement avec qui et avec quoi nous avons entré en contacte seulement selon quel fonction est appelée.

# <titre>

## <Titre>

<Texte>

# <titre>

## <Titre>

<Texte>

# <titre>

## <Titre>

<Texte>

# <titre>

## <Titre>

<Texte>

CONCLUSION

<Texte interligne 1 1/2>

RECOMMANDATIONS <S’il y a lieu>

<Texte interligne 1 1/2>

1. <titre>

<Texte>

1. <titre>

<Texte>

1. <titre>

<Texte>

1. <titre>

<Texte>

1. <titre>

<Texte>

1. <titre>

<Texte>

1. <titre>

<Texte>

1. <titre>

<Texte>

1. <titre>

<Texte>

1. <titre>

<Texte>

APPENDICES <S’il y a lieu>

<Texte>

LISTE DE RÉFÉRENCES

FatFs : <http://elm-chan.org/fsw/ff/00index_e.html>

Entête Bitmap : <http://en.wikipedia.org/wiki/BMP_file_format>

Entête Wave : <http://www.lightlink.com/tjweber/StripWav/Canon.html>

Spécification OHCI : <http://www.compaq.com/productinfo/development/openhci.html>

Spécification USB : <http://www.usb.org/developers/docs/>

Transparence : <http://www.winprog.org/tutorial/transparency.html>

BIBLIOGRAPHIE

<Texte – Style Références bibliographiques déjà activé>