Rubik's Cam Project PHY573a

Parakian Alexis, Skrzypczak Nathan

17 décembre 2013

Résumé

Rubik's cam est un projet développé au cours de l'Enseignement d'Approfondissement : "Conception expérimentale micro et nanoélectronique" partie FPGA. Le but étant comme son nom l'indique, d'afficher un rubik's cube à l'écran en 3 dimensions et de le résoudre à l'aide de mouvements effectués devant la caméra. Ce projet ambitieux nécessite alors la réalisation de deux parties bien distinctes qui nous permirent de se départager le travail. Nathan Skrzypczak prit en charge la réalisation d'un moteur 3D et Alexis Parakian, celle de la détection de mouvements via la caméra.

Table des matières

1 Présentation **2**

2 Architecture **3**
2.1 Contexte et Entrées Sorties 3
2.1.1 Sortie VGA 3
2.1.2 Entrées : Camera CMOS 4
2.1.3 Mémoires 5
2.1.4 Horloge 6
2.2 Moteur d'affichage 7
2.3 Détection de mouvement 9

3 Connexion des blocs **13**

4 Difficultés **15**
4.1 Bus SRAM 15
4.2 Développement d'un "Pong" de test 16

5 Possibilités d'amélioration **18**
5.1 Mémoires 18
5.2 Affichage 18
5.3 Acquisition 18
5.4 Reconnaissance des formes 18

Chapitre 1

Présentation

Présentation Scientifique a faire

Chapitre 2

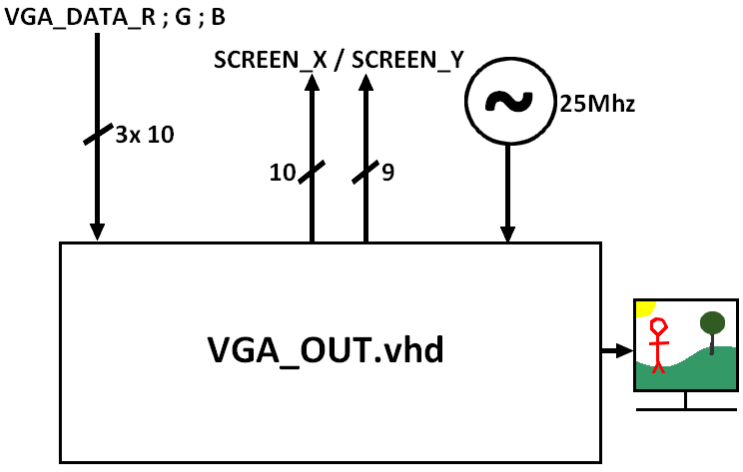
Architecture

2.1 Contexte et Entrées Sorties

2.1.1 Sortie VGA

La première partie du projet à être réalisée aura été la sortie VGA. Elle l'a été sous la forme d'un composant pour plus de facilité d'utilisation. Il est contenu dans le fichier VGA_OUT.vhd

Le point à observer en particulier est la fréquence d'horloge, en effet celle-ci conditionne la conception du reste du circuit. On cherche à cadencer le rafraichissement à 60Hz. Une trame comprends 795 colonnes et 525 lignes (pour 640x480 utiles) soit 417900 points sur l'écran. Obtenir 60Hz par trame revient alors à **cadencer les points à 25.074Mhz**. On choisira ainsi une **fréquence utile de 25Mhz** pour rester en phase avec le reste du circuit, soit 59.82Hz par frame (on commet une erreur de 0.3% qui est admise dans la norme VGA.



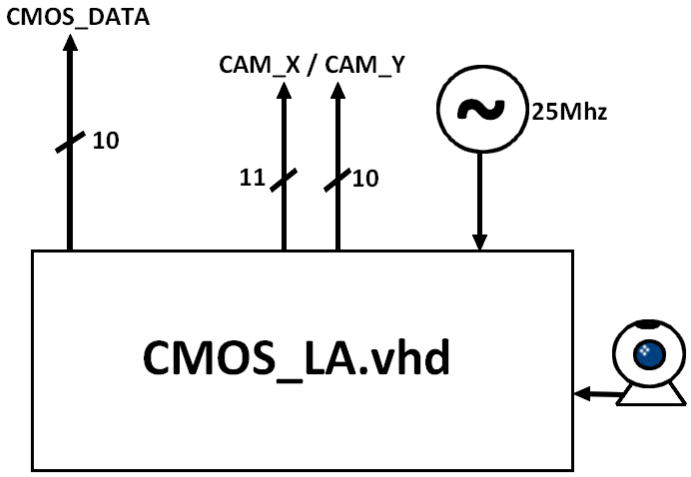
– Ce composant est cadencé sur une horloge à 25Mhz (fréquence d'affichage d'un pixel)

– Il sort deux signaux entiers donnant la position du pixel courant : SCREEN_X [0..639] et SCREEN_Y [0..479] sur 10 et 9 bits respectivement.
– la couleur du pixel courant est donnée en entrée sur trois bus de 10 bits : VGA_DATA_R ; VGA_DATA_G ; VGA_DATA_B

2.1.2 Entrées : Camera CMOS

Concernant l'entrée par la camera, nous nous sommes inspirés d'un exemple de la bibliothèque, que nous avons quasiment entièrement réécrit. Il s'agit du programme de dialogue avec le CMOS, paramétrage de la caméra et formatage des données.

Un point particulièrement important dans l'utilisation de ce périphérique connecté au GPIO de la carte DE2 est **la gestion des horloges**. En effet, la caméra prends en entrée une horloge cadencée à 25Mhz, et retourne les données lues sur les pixels CMOS à cette même fréquence. Cependant, ce processus a une certaine durée d'exécution, surtout à cause de la présence de la nappe, et du câblage interne, qui ajoute un retard moyen de 10ns (soit $\frac{1}{4}$ de periode à 25Mhz). L'horloge générale utilisée pour le reste du circuit devra donc tenir compte de ce retard de 10ns de manière à assurer la synchronisation des données. (Voir la section Horloge pour plus de précisions)



– Ce composant est cadencé sur une horloge à 25Mhz (acquisition des pixels)
– Il sort deux signaux entiers donnant la position du pixel courant : CAM_X $[0..(640*2-1)]$ et CAM_Y $[0..(480*2-1)]$ sur 11 et 10 bits respectivement.
– la couleur du pixel courant est donnée par l'organisation du capteur

G	R	...
B	G	...
⋮	⋮	⋱

2.1.3 Mémoires

Le choix des mémoires aura posé de nombreuses difficultés, dans la mesure où les composants présents fixent de nombreuses contraintes, et que le matériel disponible (pour les mémoires) possède également les siennes. On souhaite donc afficher sur la sortie VGA une figure 3D projetée dans le plan de l'écran. Les éléments qui seront donc sujet à une forte occupation mémoire, et qui ne peuvent pas se résoudre par l'utilisation de bascules sur la carte seront donc le stockage des points de cette figure et de l'affichage de l'écran si celui-ci doit être stocké dans un buffer.

1. **Figure 3D**
   - Pour la représentation en 3D de la figure, la solution la plus simple est de la décomposer en triangles (dans la mesure ou ils sont nécessairement plans). Ces triangles (ensembles de 3 points, donc 9 coordonnées de l'espace) seront tout stockés indépendamment les uns des autres par souci de simplicité. Il est possible d'optimiser grandement le stockage en utilisant des recouvrements (dans la mesure ou ils partagent souvent des sommets), mais ceci génère de la complexité dans le calcul et la gestion de cette mémoire.
   - La quantité de mémoire à prévoir pour afficher un Rubik's cube et donc en première approximation **de 108 triangles soit 972 coordonnées**.
   - Il faut également considérer que nous manipulons des points qui seront probablement sujet à des translations et des rotations. Il nous faut donc garantir une bonne précision pour éviter la divergence des données par approximations successives. Une précision de 32bits avec virgule fixe à 16bits semble raisonnable (Avec des coordonnées allant de -32768 à 32767 et une précision de 10^{-4}, on peut exprimer des points à 'l'infini visuel' et se prémunir des erreurs d'approximation grossières) Cette précision reste ajustable en réduisant les coordonnées accessibles.
   - Avec ces observations, on atteint **32Kbits de données à stocker**, or le Cyclone II possède 100 puces M4K accessibles en lecture / écriture. On peut donc réaliser ce stockage directement sur la carte avec 8 telles puces.

   Les triangles seront donc stockés sur une mémoire ***double port*** *de* ***32bits de données*** *et de* ***10bits de bus d'adresses****.*

   Les coordonnées seront stockées de cette manière pour un triangle (en partant de l'adresse 0x000) :

...	x_1	y_1	z_1	x_2	y_2	z_2	x_3	y_3	z_3	...

2. **Ecran**
   - Ayant vu ceci, il est nécessaire de parcourir l'ensemble des M4K occupés et de traiter les points présents pour obtenir l'image à afficher à l'écran. La fréquence d'affichage des pixels étant de 25Mhz et comme il est déraisonnable d'espérer faire fonctionner la mémoire interne à plus de 100Mhz, il est impossible de compter sur un traitement temps réel : **Nécessairement les différents pixels devront être stockés en mémoire**
   - Reste donc à définir une telle mémoire pour un écran de 640x480 pixels. La mémoire la plus simple d'utilisation offrant tout de même une capacité raisonnable est la SRAM. En effet, pour rentrer dans les 100 x

M4K de la mémoire on-chip avec une profondeur de couleur de 10bits, il faudrait se limiter à 160x120...

- La SRAM permet de stocker 256 kilomots de 16 bits ce qui est largement suffisant pour une profondeur de 10bits (niveaux de gris ou couleurs réduites) sur 640x480 pixels. Cette mémoire peut également être accédée jusqu'à 100Mhz, ce qui permet une réactivité suffisante pour la sortie VGA.
- Pour ce qui est de la répartition des données, cette dernière est un peu subtile dans la mesure ou les plages d'adresses et de données ne sont pas paramétrables de la même façon que sur le carte. On se limite dans une première approximation à une plage de pixel de 320x240 pour simplification, mais la modification de la formule permettrait d'étendre à la résolution supérieure. L'adresse se calcule de la manière suivante :

$$Adresse_{18bits} = \frac{x}{2} + 320.\frac{y}{2}$$

- On observe rapidement que le passage à 640 x 480 n'est pas particulièrement complexe, du moment que le codage des couleurs est modifié : Étant donnés les 16bits de largeur de données pour la SDRAM, on choisit un codage sur 8 bits comme suit :

R	R	R	G	G	B	B	B

Une lettre représente ici un bit - on choisit de ne stocker que les bits les plus significatifs de chaque couleur.

L'adresse du pixel (x, y) se calcule alors par

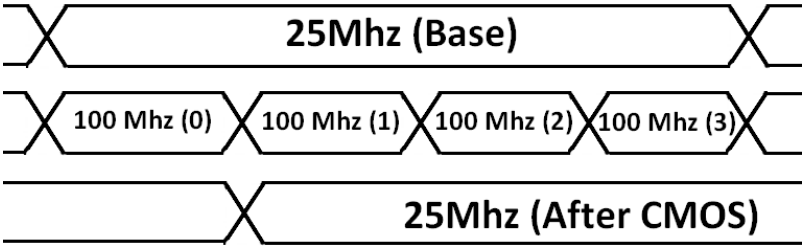
$$Adresse_{18bits} = \frac{y}{2}_{8bits} @ x_{10bits}$$

Avec "@" l'opérateur concaténation
On y trouve les données des pixels (x, y) et $(x, y + 1)$ concaténées sur la donnée de 16bits, ce qui permet de stocker l'intégralité de l'écran dans la SRAM.

2.1.4 Horloge

Étant donné les éléments en jeu dans ce circuit, **l'horloge principale sera donc cadencé à 100Mhz** (fréquence maximale de fonctionnement de la SRAM), de manière à permettre à cette dernière une lecture et une écriture dans une étape de calcul à 25Mhz (Une écriture dure en effet 3 cycles de 10ns et une lecture un cycle de 10ns).

La caméra induit un décalage de 10ns, on cadence donc cette dernière sur le la base 25Mhz alors que l'ensemble du circuit suit une cadence avec un cycle de 100Mhz de retard (donc 10ns)



Ces horloges seront donc générées par un PLL du circuit :
- CLOCK_50 : 50 Mhz base du FPGA
- CLOCK_25 : 25 Mhz PLL
- CLOCK_25d : 25 Mhz décalé de 10ns PLL
- CLOCK_50d : 50 Mhz décalé de 10ns PLL

Le choix a beaucoup oscillé entre 100Mhz et 50Mhz pour le choix de l'horloge principale cadencent les accès SDRAM. Mais le graphe ci-dessus n'en reste pas tout autant valable.

2.2 Moteur d'affichage

C'est ce moteur d'affichage qui impose par son temps de réaction (qui ne peut pas être cadencé pour répondre en temps réel (25Mhz) à la demande des pixels) l'utilisation d'une mémoire pour l'écran et pour ses donnée personnelles.

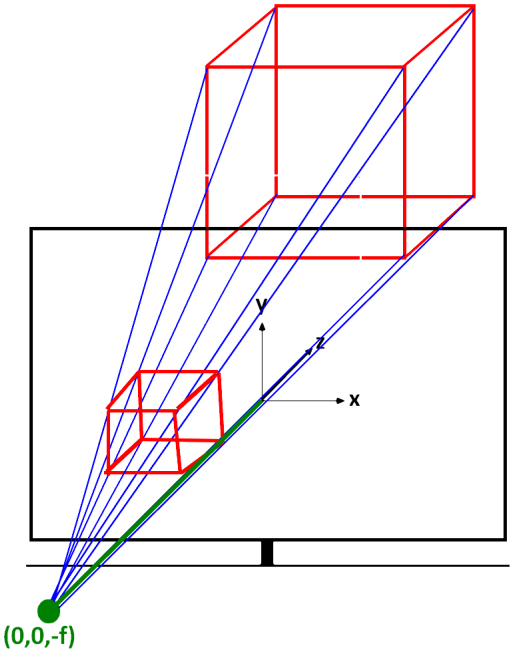
Son fonctionnement est donc fortement dépendant de toutes ces structures de données. Il est cadencé à 25Mhz (décalé de 10ns pour suivre la Caméra) et procède par plusieurs étapes :

1. Les trois coordonnées constituant un point de l'espace sont dépilées et stockées dans un registre à décalage de 3x 32bits.
2. Une fois ce registre plein, la projection sur l'écran est calculée par la formule suivante :

$$X = \frac{L}{2} + \left(x - \frac{L}{2}\right)\frac{f}{z+f}$$

$$Y = \frac{H}{2} + \left(y - \frac{L}{2}\right)\frac{f}{z+f}$$

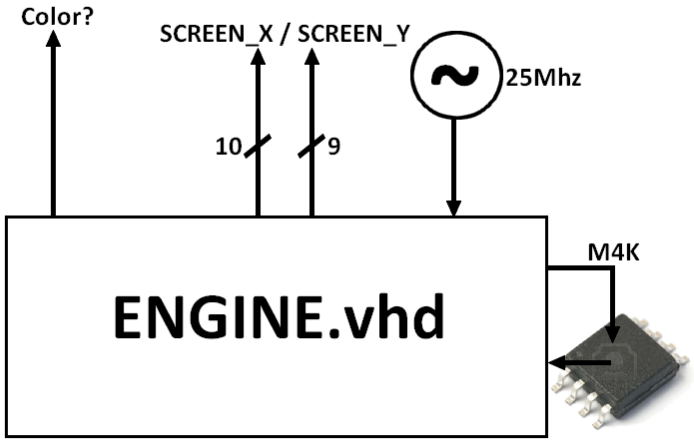
Où (X, Y) sont les coordonnées projetées sur l'écran (de hauteur H, largeur L) ; (x, y, z) les coordonnées 3D présentes dans le registre à décalage et f la focale de projection. On considère l'origine des espaces comme étant située au centre de l'écran et orientée vers l'avant comme sur la figure suivante.



Projection d'ou l'on tire la formule précédente.

3. La plage mémoire associée à ces données est d'une profondeur de 32bits pour les coordonnées 3D, la virgule fixe étant définie après 16bits. Le calcul est réduit à des coordonnées entières sur 10 et 9bits (640 et 480), ce qui correspond à une troncature entière.
4. Ces coordonnées (X, Y) sont ensuite à nouveau inscrites dans un registre à décalage de manière à pouvoir rassembler trois couples de coordonnées successifs afin de constituer un triangle.
5. Une fois ce triangle constitué au sein du registre à décalage, on l'inscrit dans la SDRAM - dans un premier temps en n'imprimant que les sommets - ce qui se révèle finalement le plus simple, avec la perspective de pouvoir afficher les triangles sous forme de fil de fer, puis dans un deuxième temps de les remplir (ce qui requiert alors de traiter l'occlusion d'un triangle par un autre).

Ceci donne un composant chargé des calculs de la forme ci-dessous, avec une sortie 'couleurs' non encore définie, car dans des perspectives de test, ce composant ne sort que des points, colorés en blanc, pour ne pas surcharger la mémoire M4K et réduire la complexité du tout.



2.3 Détection de mouvement

Le programme de détection de mouvement est assez simpliste comparé à ce qui existe déjà mais l'idée principale était de faire quelque chose d'assez léger qui puisse tourner en temps réel et sans trop utiliser de mémoire (La SRAM et les M4K étant déjà mises à rude épreuves par le moteur 3D).

Nous nous sommes principalement inspirés de l'algorithme de Viola et Jones de détection d'objet dans une image numérique. Cette méthode est l'une des plus connues et plus utilisées dans la détection de visages et de personnes.

Elle consiste au parcours de l'ensemble d'une image en calculant un certain nombre de caractéristiques dans des zones rectangulaires qui se chevauchent. Pour calculer rapidement et efficacement ces caractéristiques sur une image, les auteurs proposent également une nouvelle méthode, qu'ils appellent "image intégrale. C'est une représentation sous la forme d'une image de même taille que l'image d'origine, qui en chacun de ses points contient la somme des pixels situés au-dessus de lui et à sa gauche.

Les caractéristiques sont calculées à toutes les positions et à toutes les échelles dans une fenêtre de détection de petite taille, typiquement de 24 x 24 pixels. En phase de détection, l'ensemble de l'image est parcouru en déplaçant la fenêtre de détection d'un certain pas dans le sens horizontal et vertical. Les changements d'échelle se font en modifiant successivement la taille de la fenêtre de détection. Viola et Jones utilisent un facteur multiplicatif de 1,25, jusqu'à ce que la fenêtre couvre la totalité de l'image. Les caractéristiques étant ensuite définies par machine learning sur le type d'image considéré (visage, personnes, etc.).

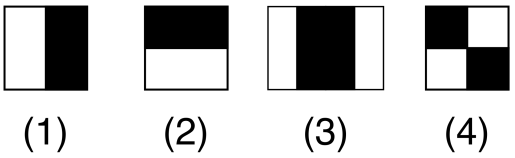
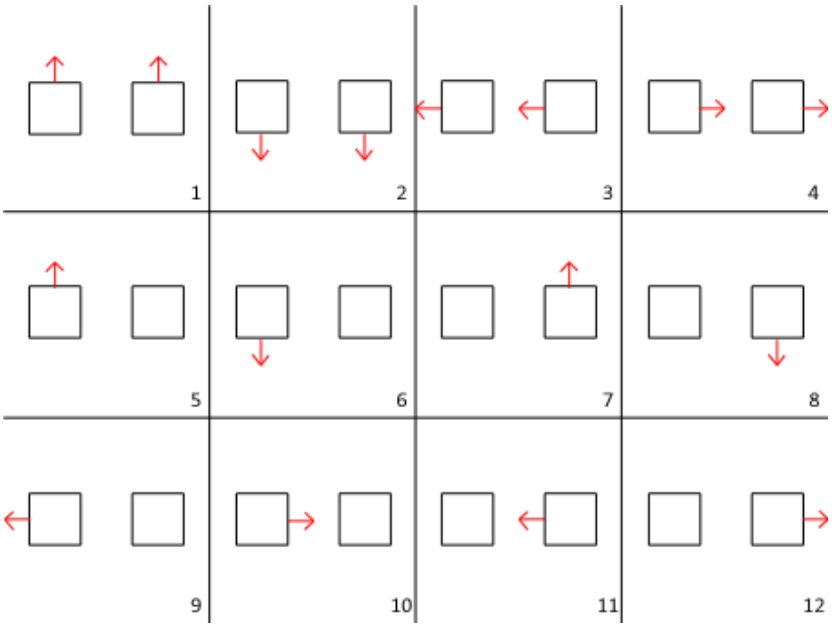


FIGURE 2.1 – Exemples de caractéristiques calculées par différences des sommes de pixels sur deux ou plusieurs zones adjacentes

Dans notre cas, étant donné que nous ne voulions pas stocker l'intégralité de l'image (sa taille est de 1280x960x10 = 12 Mbits, il n'est donc pas concevable de stocker une image complète en utilisant simplement les bascules) et que les caractéristiques cherchées étaient moins complexes que pour la détection de visage, nous avons agi différemment : Tout d'abord nous avons simplifié le problème en se disant que l'objet à trouver serait deux mains sur un fond noir. Donc la caractéristique à trouver est une simple caractéristique de contraste.

Ensuite, le concept de parcours d'une image par une fenêtre de taille croissante devait être revu car cela impliquait de sauvegarder en mémoire une image, de la traiter, puis de faire la même chose pour l'image suivante et par différence des deux de définir un mouvement. Mais comme nous ne voulions pas utiliser la mémoire et tout faire en temps réel, nous avons donc défini un quadrillage fixe de l'image qui contenait dans chaque fenêtre, la somme des pixels présents à l'intérieur. L'avantage étant qu'à la lecture d'un pixel, étant donné que la position était connue, il suffisait d'incrémenter les fenêtres correspondantes.

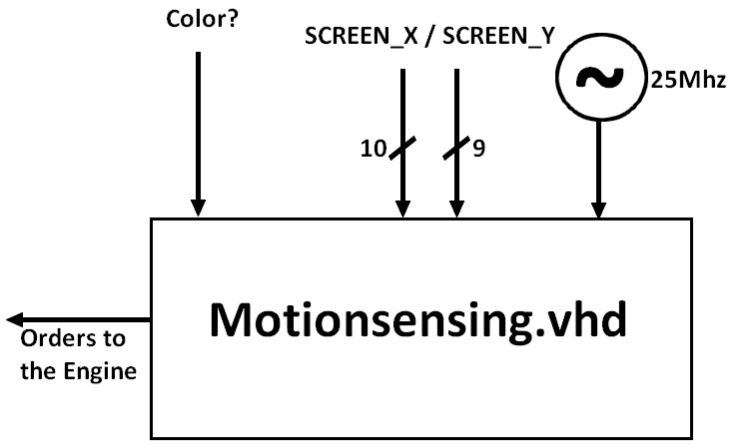
Le quadrillage fut définit par une taille de fenêtre fixe de 160x160 pixels et un pas de 80 pixels, ce qui donnait 35 fenêtre à analyser sur l'image. Ce quadrillage était largement suffisant au vu des commandes à envoyer au moteur 3D : rotation du rubik's cube et rotation d'une couronne du rubik's cube soit un total de 12 commandes possibles.



1. rotation du rubik's cube vers le haut
2. rotation du rubik's cube vers le bas
3. rotation du rubik's cub vers la gauche
4. rotation du rubik's cube vers la droite
5. rotation de la couronne de gauche vers le haut
6. rotation de la couronne de gauche vers le bas
7. rotation de la couronne de droite vers les haut
8. rotation de la couronne de droite vers le bas
9. rotation de la couronne du bas vers la gauche
10. rotation de la couronne du bas vers la droite
11. rotation de la couronne du haut vers la gauche
12. rotation de la couronne du haut vers la droite

Une fois l'image entièrement parcourue, les mains sont repérées comme étant dans les deux carrés les plus lumineux, soit les deux plus grandes valeurs du tableau de 7x5 en excluant les fenêtres pouvant se recouper car une main peut recouvrir deux carrés. Enfin le déplacement est réalisé en comparant avec l'image précédente. Les déplacements s'incrémentent à chaque nouvelle image et le déplacement moyen est envoyé sous la forme d'une commande au fichier principal.

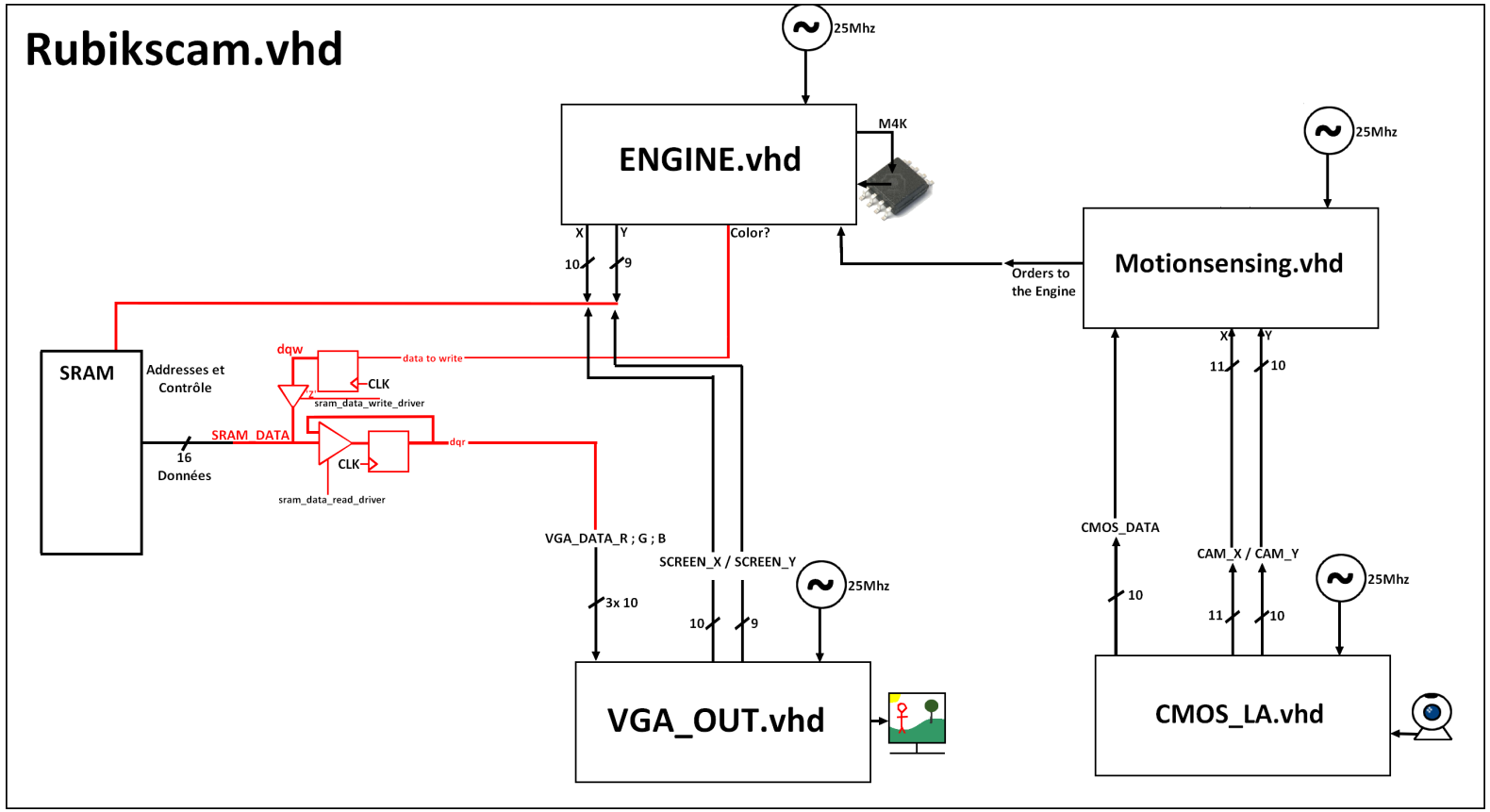
Finalement le bloc général s'organise de la manière suivante :


Color?
SCREEN_X / SCREEN_Y
25Mhz
10
9
Motionsensing.vhd
Orders to
the Engine

Chapitre 3

Connexion des blocs

Maintenant que les différents modules du projet ont été présentés, leur agencement et leurs connexions réciproques s'effectuent de la manière suivante :


Rubikscam.vhd
25Mhz
ENGINE.vhd
M4K
X
Y
10
9
Color?
Orders to
the Engine
Motionsensing.vhd
25Mhz
X
Y
11
10
SRAM
Addresses et
Contrôle
dqw
data to write
CLK
sram_data_write_driver
SRAM_DATA
16
Données
dqr
CLK
sram_data_read_driver
VGA_DATA_R ; G ; B
3x 10
SCREEN_X / SCREEN_Y
10
9
25Mhz
VGA_OUT.vhd
CMOS_DATA
10
CAM_X / CAM_Y
11
10
25Mhz
CMOS_LA.vhd

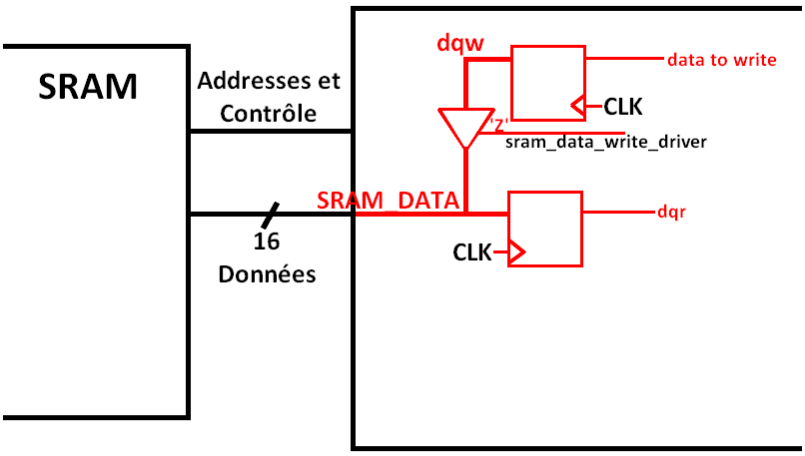
Chapitre 4

Difficultés

4.1 Bus SRAM

Un des problèmes principaux s'étant posé à nous au cours de ce projet est la communication avec les mémoires et leur choix. Le projet était certainement trop ambitieux de ce point de vue, dans la mesure ou l'écriture sur le Bus Entrée/Sortie de la SDRAM n'a été fonctionnel que dans la dernière demi-heure de la dernière séance. Le code sous-jacent étant fondé sur des principes mathématiques relativement simplistes, il est supposé fonctionnel, mais sans la brique de la Mémoire intermédiaire pour le stockage de l'écran, il n'est malheureusement pas possible de le tester.

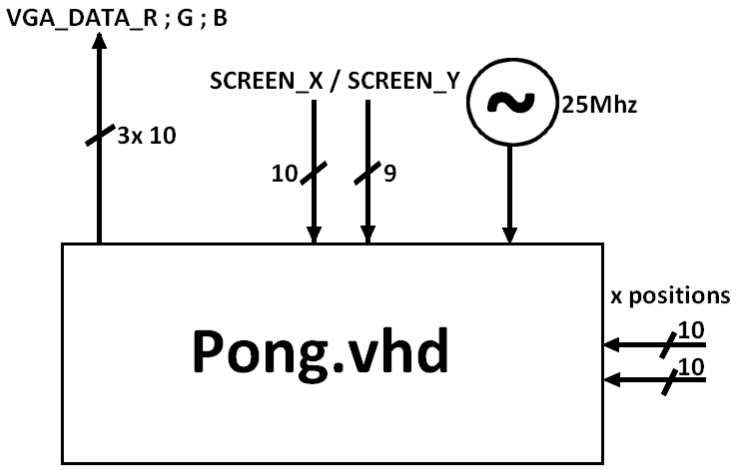
Le blocage du code se réalise au niveau du cadencement des lectures/écritures sur la SRAM. Dans la mesure ou l'écran impose une lecture à 25Mhz, lecture et écriture disposent chacune d'une période temporelle de 20ns pour s'effectuer. On se trouve ici particulièrement proche des valeurs minimales présentes dans la datasheet de la mémoire (de l'ordre de 15ns). De plus la commutation imposée sur le bus entrée/sortie par ce processus est également particulièrement rapide : de l'ordre de la nanoseconde entre la présence des données d'entrée et celles de sortie. C'est ce bus qui pose donc les problèmes principaux, et qui se révèlent physiquement, dans la mesure ou l'écran réplique les données qui sont écrites sur la SDRAM au moment de leur écriture : au lieu de lire le contenu de la SDRAM qui lui est adressé, l'écran se sert dans le contenu qui le moteur 3D est entrain d'inscrire sur le Bus, signe d'un mauvais cadencement de ces opérations ou d'une erreur sur la conception de ce bus.



4.2 Développement d'un "Pong" de test

C'est pour cette raison qu'un exemple rapide de Pong à deux joueurs à été réalisé sous la forme d'un composant annexe prenant la place de ENGINE.vhd, pour permettre d'avoir une présentation fonctionnelle dans le cas d'une incapacité à avoir une lecture / écriture correctement cadencée sur la SRAM.

Ce Pong est situé dans Pong.vhd est suit le même principe que Engine.vhd : on lui entre les signaux d'écran SCREEN_X et SCREEN_Y, et un signal de validité ou non du point courant (un VGA_BLANK en somme). Il retourne alors la couleur du point courant sur 3x10bits (R,G,B). Le tout cadencé à la même fréquence que l'écran.



Ce composant réalise de simple opérations de comparaison pour obtenir la couleur du pixel courant affiché à l'écran, et garde en mémoire dans des registres locaux les informations importantes, telles que la position de la balle, celle des deux joueurs et le nombre de vies restantes. Il reste paramétrable par des entrées de position, permettant au composant d'acquisition d'influencer l'état de l'affichage et du processus en cours.

Nous l'avons rapidement connecté au composant motion_sensing afin de tester la validité de l'algorithme de détection de mouvement. Les effets ne furent pas extrêmement concluants mais la technique adoptée pour la résolution du rubik's cube n'était pas vraiment transposable au contrôle des pads du pong puisqu'ils nécessitent un repérage beaucoup plus fin de la position de la main.

Chapitre 5

Possibilités d'amélioration

5.1 Mémoires

5.2 Affichage

5.3 Acquisition

On pourrait se pencher plus en profondeur sur le paramétrage fin de la caméra, et réaliser des tests voir de l'apprentissage pour que l'acquisition se fasse plus finement. Actuellement, l'exposition est réglée manuellement à l'aide de la bande d'interrupteurs de la carte DE2. On pourrait donc rechercher une valeur idéale pour ce paramètre afin de maximiser les résultats pour l'objectif nous intéressant.

5.4 Reconnaissance des formes

Dans le cas où nous pouvons utiliser la SRAM, il aurait été plus efficace de stocker une frame, puis de réaliser un traitement plus lourd à cette image afin de faire des comparaisons plus justes entre image. Rien que de faire la différence entre deux frames et repérer la zone qui a bougé aurait permis des résultats plus justes.

Chapitre 6

Conclusion