Documentation des tests des bibliothèques

Auteur : David ODIN - Forma3Dev

Version : 0.6

Date : ? ? novembre 2010

Diffusion : AVT – ED3L – Forma3Dev

Version	Date	Auteur	Commentaires
0.1	août 2010	David Odin	Première version.
0.2	10 septembre 2010	David Odin	– Les *threads* sont maintenant dans un test à part – Ajout de la documentation du test des séparateurs – Ajout de la documentation du test des fenêtres
0.3	4 octobre 2010	David Odin	– Documentation des tests des listes doublement chaînées – Suppression de la mention « non implémentée » pour la partie clavier USB de la bibliothèque Input
0.4	8 octobre 2010	David Odin	– Mise à jour pour refléter l'implémentation du support des touches de l'interface intégrée.
0.5	11 octobre 2010	David Odin	– Ajout du test pour les messages d'erreur.
0.6	? ? novembre 2010	David Odin	– Ajout du test de visibilité. – Mise à jour des captures d'écran pour la bibliothèque Widget.

DRAFT

Sommaire

1 Introduction **4**
1.1 Notes 4

2 Bibliothèque Base **4**
2.1 Système de *log* 4
2.2 Listes chaînées 5
2.3 Listes doublement chaînées 5
2.4 Files 5
2.5 Notes 6
2.6 Gestion des *threads* 6

3 Bibliothèque Graphic **6**
3.1 Primitives graphiques 6
3.2 Affichage de texte 7
3.3 Affichage d'images 8
3.4 Utilisation de l'opacité 9
3.5 Utilisation du *double buffering* 10
3.6 Notes 11

4 Bibliothèque Image **11**
4.1 Images opaques 11
4.2 Images semi-transparentes 11
4.3 Notes 13

5 Bibliothèque Input **13**
5.1 Notes 13

6 Bibliothèque Engine **14**

7 Bibliothèque Video **14**
7.1 Affichage simple de l'image de la caméra 14
7.2 Encodage vidéo 14
7.3 Décodage vidéo 14
7.4 Conversion YUV 422 vers RGB 565 14
7.5 Notes 14

8 Bibliothèque Widget **15**
8.1 label 15
8.2 Alignement des labels dans les boites horizontales 15
8.3 Boites verticales 16
8.4 Boutons 16
8.5 Cases à cocher 16
8.6 Boutons radio 17
8.7 Entrées de texte 17
8.8 Séparateurs 18
8.9 Barres de progression 19
8.10 Labels messages 19
8.11 Fenêtres 19
8.12 Messages d'erreur 20
8.13 Visibilité des widgets 20

9 Tableau de validation des tests **22**

DRAFT

Table des figures

1 Dégradés de couleurs. 6
2 Rectangles. 7
3 Rectangles à bords épais. 7
4 Rectangles remplis. 7
5 Lignes. 7
6 Triangles. 7
7 Chaînes de caractères. 7
8 Texte de différentes tailles, aliasé. 8
9 Texte de différentes tailles, anti-aliasé. 8
10 Images 1. 9
11 Images 2. 9
12 Rectangles dans un damier. 9
13 Rectangles à bords épais dans un damier. 9
14 Rectangles pleins dans un damier. 9
15 Lignes dans un damier. 10
16 Triangles dans un damier. 10
17 Images 1 dans un damier. 10
18 Images 2 dans un damier. 10
19 Triangle dansant. 10
20 Chargement d'une image PNG. 11
21 Chargement d'une image JPEG. 11
22 Chargement d'une image TIFF. 11
23 Changement de taille d'une image. 11
24 Mario avec un contour transparent. 12
25 Affichage d'une image masquée par une autre 1/5. 12
26 Affichage d'une image masquée par une autre 2/5. 12
27 Affichage d'une image masquée par une autre 3/5. 12
28 Affichage d'une image masquée par une autre 4/5. 12
29 Affichage d'une image masquée par une autre 5/5. 12
30 Affichage d'une partie d'une image 1/3 13
31 Affichage d'une partie d'une image 2/3 13
32 Affichage d'une partie d'une image 3/3 13
33 Exécution du programme de test de la bibliothèque Input alors que l'utilisateur vient d'utiliser l'entrée "capture" 13
34 Affichage du programme de test de conversion YUV vers RGB 14
35 Affichage simple de labels 15
36 Multiples labels avec différents alignements. 15
37 Test d'empilement vertical de labels avec différents alignements. 16
38 Test des boutons. 16
39 Test des cases à cocher. 17
40 Test des boutons radio. 17
41 Test des entrées de texte. 18
42 Test des séparateurs. 18
43 Test des barres de progressions. 19
44 Test des labels messages. 19
45 Test des fenêtres. 20
46 Test des messages d'erreur 20
47 Test de visibilité 1. 21
48 Test de visibilité 2. 21
49 Test de visibilité 3. 21
50 Test de visibilité 4. 21

DRAFT

1 Introduction

Ce document a pour but de lister et de décrire les différents programmes de tests des bibliothèques développées par Forma3Dev pour le programme principal de l'AVTBox. Chaque bibliothèque propose en effet un certain nombre de tests afin de vérifier le fonctionnement correct de chacune des fonctionnalités qu'elle propose.

Contrairement à ce qui est parfois fait, ces tests unitaires ne sont pas répartis chacun dans un programme. Au contraire, on a un (ou quelques) programme qui exécute tous les tests d'une bibliothèque à la suite. Cela rend l'écriture des tests plus simples ainsi que leur validation.

Ce document détaille chacun de ces tests et indique quel en est le fonctionnement normal.

1.1 Notes

– Ce document est en cours d'écriture. Certaines bibliothèques n'étant pas terminées, les programmes qui les testent ne peuvent pas être actuellement écrits ni décrits.

2 Bibliothèque Base

Les fonctions de la bibliothèque **Base** sont testées dans le programme base_test.

2.1 Système de *log*

Le programme base_test commence par tester le système de log, en affichant des messages des divers niveaux supportés directement à l'écran. Ces messages comportent tous la criticité entre crochet, suivie du contexte du message (chaque bibliothèque définit un contexte, mais il est possible d'en imposer un pour des messages particuliers dans un programme). Puis viennent le nom du fichier source contenant l'appel qui a généré le message, ainsi que le numéro de ligne dans le fichier et le nom de la fonction contenant l'appel, ce qui permet de rapidement connaître l'origine d'un problème. Enfin, le corps du message est ajouté.

Le début de l'affichage du programme base_test ressemble donc à ceci :

```
--------------------------------------------------------------------------
Creating a terminal based logger, logging everything from the "debug" level
--------------------------------------------------------------------------
[DEBUG] "Test (debug)" base_test.c:24 (test_logger_terminal) - This is just some debugging informations
[INFO] "Test (debug)" base_test.c:25 (test_logger_terminal) - Some informations
[WARNING] "Test (debug)" base_test.c:26 (test_logger_terminal) - This isn't really an error, merely a
    warning
[ERROR] "Test (debug)" base_test.c:27 (test_logger_terminal) - This is just a test for error, please
    ignore
[CRITICAL] "Test (debug)" base_test.c:29 (test_logger_terminal) - This kind of error should abort the
    program (not in this test, though!)
[USER] "Main (debug)" base_test.c:30 (test_logger_terminal) - An error-reported error: 1
[DEBUG] "Memory" base_logger.c:77 (base_terminate) - Max memory used: 85
[DEBUG] "Memory" base_logger.c:78 (base_terminate) - Allocated memory still in use: 0
[DEBUG] "Memory" base_logger.c:79 (base_terminate) - Total number of allocations: 8
```

Puis, la fonctionnalité permettant de n'afficher que les messages à partir d'une certaine criticité est testée, on a alors l'affichage suivant :

```
----------------------------------------------------------------------
Creating a terminal based logger, logging everything from "error" level
----------------------------------------------------------------------
[ERROR] "Test (debug)" base_test.c:42 (test_logger_terminal) - This is just a test for error, please
    ignore
[CRITICAL] "Test (debug)" base_test.c:44 (test_logger_terminal) - This kind of error should abort the
    program (not in this test, though!)
[USER] "Main (debug)" base_test.c:45 (test_logger_terminal) - An error-reported error: 1
```

Le système de *log* de la bibliothèque **Base** permet également d'enregistrer les messages dans un fichier, c'est ce que le programe base_base teste ensuite. Étant donné que les messages sont inscrits dans des fichiers et non plus dans la console, l'affichage se réduit à ceci :

```
--------------------------------------------------------------------------
Creating a file based logger, logging everything from "debug" level
--------------------------------------------------------------------------

--------------------------------------------------------------------------
Creating a file based logger, logging everything from "error" level
--------------------------------------------------------------------------
```

Cependant, deux fichiers sont créés : log−file−debug et log−file−error contenant les messages précédemment affichés sur la console. Si ces fichiers existaient préalablement, les messages sont simplement ajoutés à la fin. Les couleurs des criticités ne sont pas enregistrées dans les fichiers de *log*.

2.2 Listes chaînées

La bibliothèque **Base** propose également des implémentations de types abstraits de données comme les listes chainées et les files (*queue* en anglais).

Le programme base_test se poursuit donc par le test des fonctions manipulant les listes chaînées (ajout d'éléments en début de liste, ajout d'élément en fin de liste, suppression d'éléments, etc.). Lorsque ce test réussi, on obtient l'affichage suivant :

```
[INFO] "List" base_test.c:115 (test_slist) - Testing lists
[INFO] "List" base_test.c:121 (test_slist) - The following should be: 1->2->3->4
1->2->3->4
[INFO] "List" base_test.c:124 (test_slist) - The following should be: 2->3->4
2->3->4
[INFO] List: The following should be: 2->3
2->3
[INFO] List: The following should be: Empty list
Empty list
```

2.3 Listes doublement chaînées

La bibliothèque **Base** propose également des listes doublement chaînées (ce qui permet de les parcourir dans les deux sens). Celles-ci sont testées de la même manière que les précédentes, cependant, pour vérifier qu'il est possible de les parcourir aussi bien dans un sens que dans l'autre, l'affichage de leur contenu est fait deux fois : une fois dans le sens direct et une fois dans le sens inverse. Le programme base_test se poursuit donc par l'affichage suivant :

```
[INFO] "List" base_test.c:162 (test_dlist) - Testing double lists
[INFO] "List" base_test.c:168 (test_dlist) - The following should be: 1->2->3->4
Direct: 1->2->3->4
Reverse: 4->3->2->1
[INFO] "List" base_test.c:171 (test_dlist) - The following should be: 2->3->4
Direct: 2->3->4
Reverse: 4->3->2
[INFO] "List" base_test.c:174 (test_dlist) - The following should be: 2->3
Direct: 2->3
Reverse: 3->2
[INFO] "List" base_test.c:177 (test_dlist) - The following should be: Empty list
Empty list
```

2.4 Files

De même que pour les listes, les files sont testées en utilisant les fonctions d'ajouts et de suppression dans différents contextes. Lorsque le test des files réussi, on doit avoir l'affichage suivant :

```
[INFO] Queue: Testing queues
[INFO] Queue: The following should be: 3
3
[INFO] Queue: The following should be: 2->3
2->3
[INFO] Queue: Popped value: 3
[INFO] Queue: The following should be: 2
2
```

```
[INFO] Queue: The following should be: 1->2
1->2
[INFO] Queue: Popped value: 2
[INFO] Queue: The following should be: 1
1
[INFO] Queue: Popped value: 1
[INFO] Queue: The following should be: Empty list
Empty list
```

2.5 Notes

Comme la bibliothèque **Base** a été lancée avec le niveau de rapport `"DEBUG"`, un rapport sur l'utilisation de la mémoire est affichée à la fin de l'exécution :

```
[DEBUG] "Memory" base_logger.c:77 (base_terminate) - Max memory used: 133
[DEBUG] "Memory" base_logger.c:78 (base_terminate) - Allocated memory still in use: 0
[DEBUG] "Memory" base_logger.c:79 (base_terminate) - Total number of allocations: 34
```

2.6 Gestion des *threads*

La bibliothèque **Base** permet également de gérer les *threads* de manière simple. Cette fonctionnalité est testée par threads_test qui commence par créer deux *threads*. Le premier affiche les nombres de 0 à 4999 en haut de la console et le second fait la même chose au centre de la console. Les deux *threads* s'exécutent en même temps. Comme ces deux *threads* accèdent à la même ressource en même temps (la console), cet accès est protégé par un *mutex*.

3 Bibliothèque Graphic

La bibliothèque **Graphic** gère tout l'affichage sur le moniteur, et surtout propose un certain nombre de primitives graphiques.

Le programme graphic_test permet de vérifier que tout cela fonctionne correctement.

3.1 Primitives graphiques

Il commence par afficher quatre dégradés de couleurs : rouge, vert, bleu et blanc. Cela permet de vérifier que tout va bien au niveau des différentes composantes, et que le contraste et la luminosité du moniteur ont des valeurs acceptables (voir figure 1.

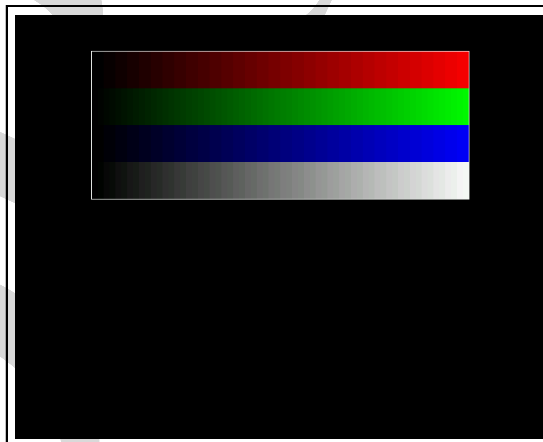

FIGURE 1 – Dégradés de couleurs.

.

DRAFT

Puis, après une brève pause, 50000 rectangles sont affichées avec des valeurs aléatoires de taille, de position et de couleur. Viennent ensuite 50000 rectangles à bords épais, toujours avec des propriétés aléatoires. Le programme se poursuit par un affichage de 5000 rectangles remplis, puis de 50000 lignes, puis de 50000 triangles, tous aux propriétés aléatoires.

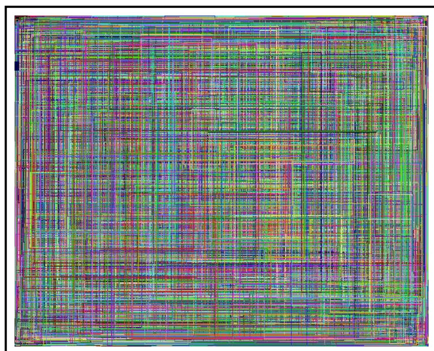

Figure 2 – Rectangles.

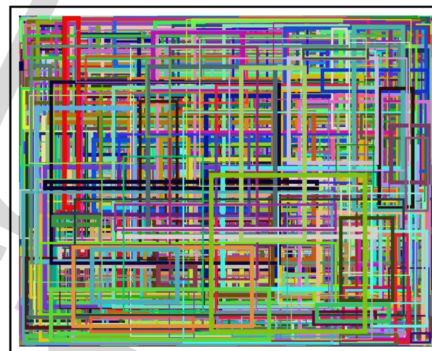

Figure 3 – Rectangles à bords épais.

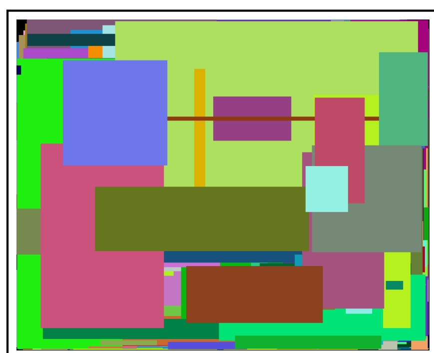

Figure 4 – Rectangles remplis.

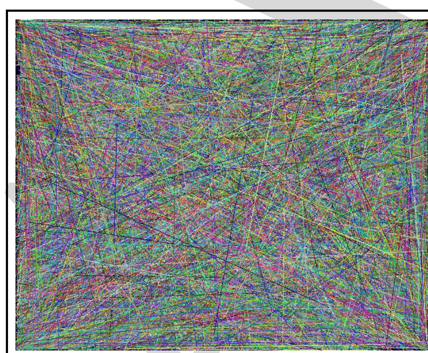

Figure 5 – Lignes.

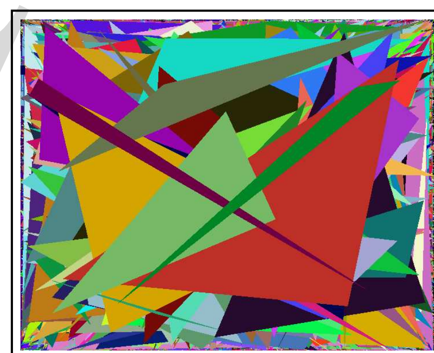

Figure 6 – Triangles.

3.2 Affichage de texte

Le programme graphic_test se poursuit par un affichage de texte. La police utilisée lors de ces tests est *Helvetica*. Les tests d'affichage de texte commencent par afficher la chaîne `"Dispoflex"` avec des tailles, des positions et des couleurs aléatoires 500 fois.

Figure 7 – Chaînes de caractères.

DRAFT

Puis une chaîne contenant 30 chiffres est affichée avec différentes tailles afin de vérifier la qualité de rendu de l'affichage de texte pour chaque taille. L'encombrement en pixels de cette chaîne pour chaque taille est affiché dans la console.

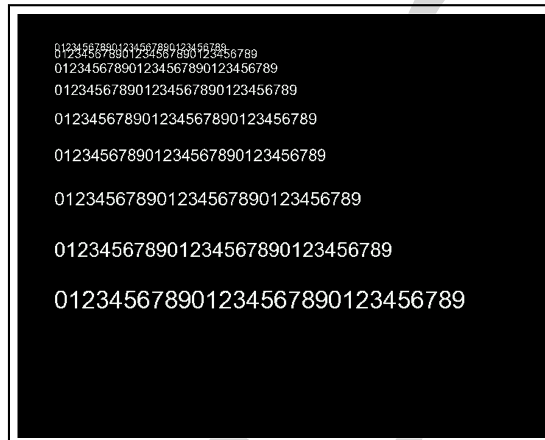

Figure 8 – Texte de différentes tailles, aliasé.

Finalement, cette même chaine est affichée à nouveau dans toutes ces tailles, mais en utilisant des fonctions mettant en œuvre l'antialiasing afin d'avoir un rendu beaucoup plus doux. Ces fonctions ont des limitations au niveau du choix des couleurs.

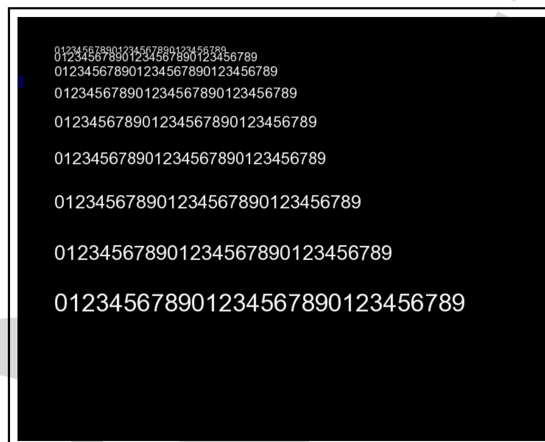

Figure 9 – Texte de différentes tailles, anti-aliasé.

3.3 Affichage d'images

La bibliothèque **Graphic** ne dépendant pas de la bibliothèque **Image**, le test d'affichage d'images de la bibliothèque **Graphic** se résume à afficher une image composée d'un simple dégradé 500 fois à diverses positions dans l'écran. Immédiatement après, le même test est réalisé en utilisant un format interne d'image permettant un affichage nettement plus rapide.

DRAFT

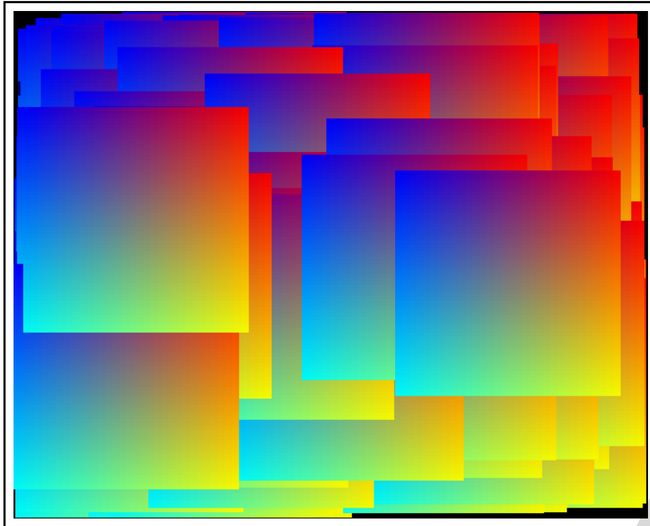

FIGURE 10 – Images 1.

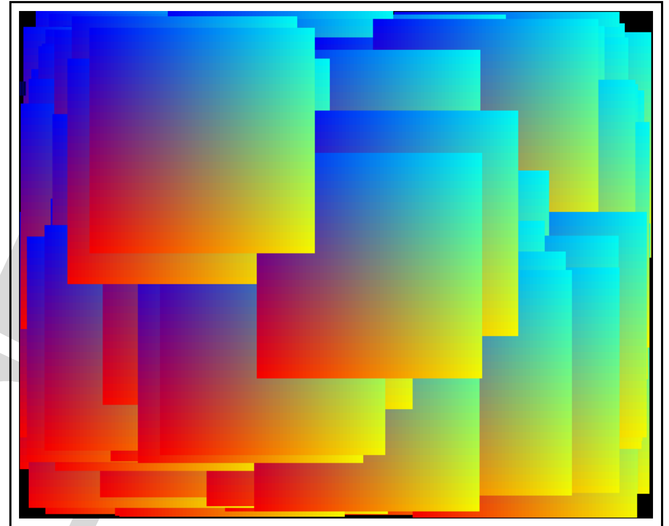

FIGURE 11 – Images 2.

3.4 Utilisation de l'opacité

Tous les tests précédents ont été réalisés en utilisant des fonctions qui mettent à jour à la fois le calque de couleur (OSD) et le calque d'opacité du framebuffer DaVinci. Les tests suivants sont les mêmes que précédemment à ceci près qu'ils utilisent des fonctions ne mettant pas à jour le calque d'opacité (ces fonctions sont donc plus rapides que les précédentes). Afin de ne pas laisser le calque d'opacité complètement transparent (rien ne serait alors visible lors des tests...) un damier est dessiné dans ce calque, laissant apparaître le résultat des tests dans les cases "blanches" du damier (les cases "noires" restant vides).

Les tests réalisés dans cette partie sont ceux concernant : les rectangles, les rectangles à bords épais, les rectangles pleins, les lignes, les triangles et les images.

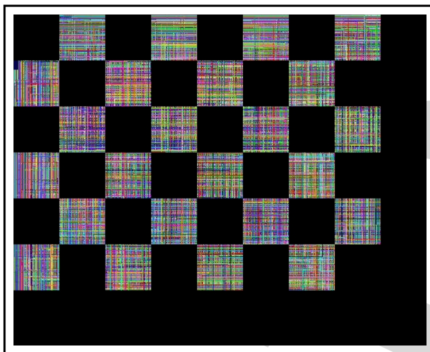

FIGURE 12 – Rectangles dans un damier.

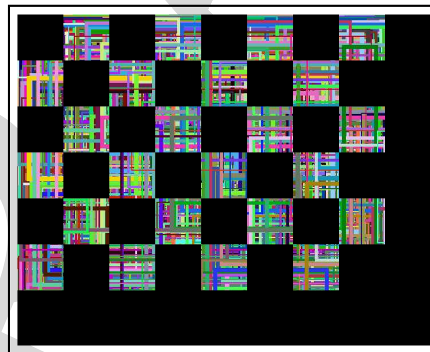

FIGURE 13 – Rectangles à bords épais dans un damier.

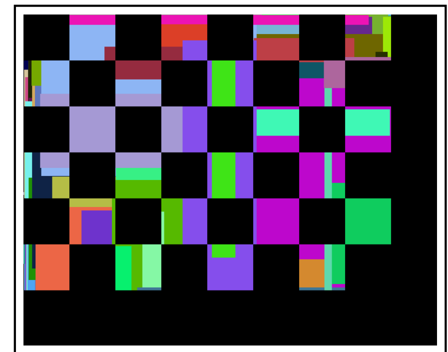

FIGURE 14 – Rectangles pleins dans un damier.

DRAFT

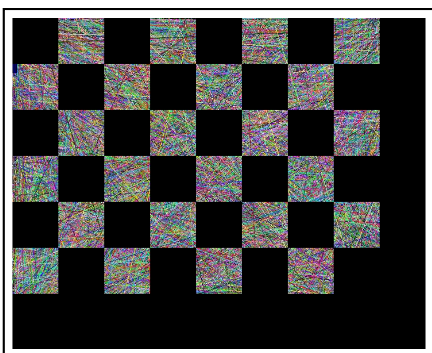

Figure 15 – Lignes dans un damier.

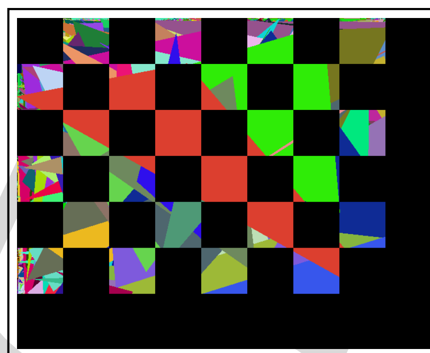

Figure 16 – Triangles dans un damier.

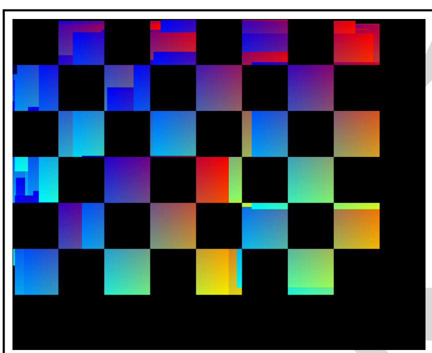

Figure 17 – Images 1 dans un damier.

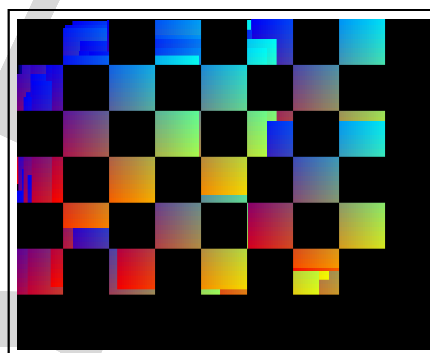

Figure 18 – Images 2 dans un damier.

3.5 Utilisation du *double buffering*

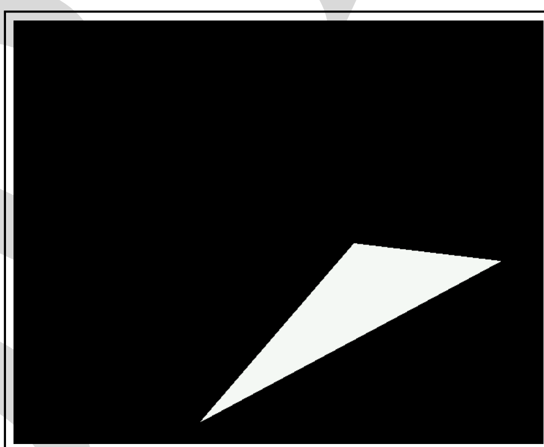

Figure 19 – Triangle dansant.

Le programme graphic_test se termine en utilisant le mode *double buffering* pour afficher un triangle blanc sur fond noir dont les sommets varient, donnant l'impression que le triangle se déforme (voir figure 19). Le *double buffering* permet à cette animation d'être le plus fluide possible.

DRAFT

3.6 Notes

- Pendant tous ces tests, des messages indiquant quel test est en cours sont affichés dans la console.
- L'exécution de graphic_test nécessite la présence du fichier `"Helvetica.ttf"` dans le même répertoire.

4 Bibliothèque Image

Les fonctionnalités de la bibliothèque **Image** sont testées par le programme image_test. Ce programme de test dépend du bon fonctionnement des bibliothèques **Base** et **Graphic**, qui doivent donc être testées préalablement.

4.1 Images opaques

image_test commence par tester les fonctions de chargement des images aux formats *PNG*, *JPEG* et *TIFF* en affichant trois images. Les commentaires éventuellement inclus dans les images *JPEG* et *TIFF* sont affichés dans la console.

FIGURE 20 – Chargement d'une image PNG.

FIGURE 21 – Chargement d'une image JPEG.

FIGURE 22 – Chargement d'une image TIFF.

La série de tests se poursuit par le test de la fonction de changement de taille en chargeant à nouveau l'image `"test_image.tiff"` en pleine taille avant de l'afficher deux fois moins grande.

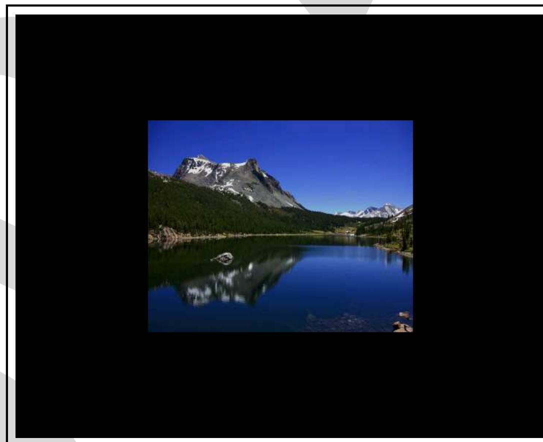

FIGURE 23 – Changement de taille d'une image.

4.2 Images semi-transparentes

Les tests suivants ont rapport avec des images transparentes ou semi-transparentes. Le premier de ces tests affiche une image du héros "Mario" 5000 fois à l'écran en respectant les parties opaques et les parties transparentes de l'image (ce qui donne l'impression que l'image n'est pas rectangulaire).

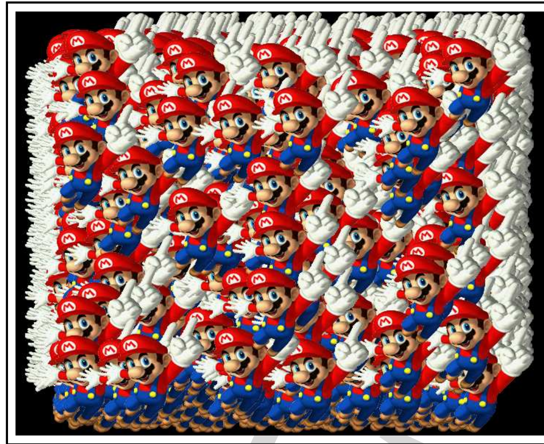

FIGURE 24 – Mario avec un contour transparent.

Le second test utilise deux images : "test_image.tiff" qui va être retaillée à 100x100 pixels et rebondir sur les côtés de l'écran pendant un bon moment et "test_image_mask.png" qui sert de masque. La première image ne sera affichée qu'aux endroit où la seconde est opaque.

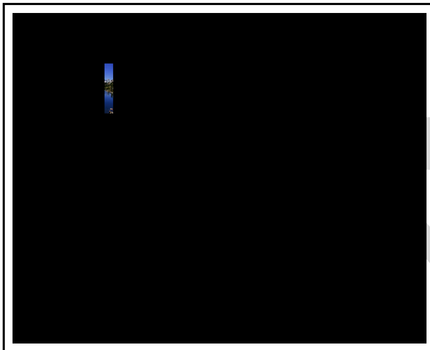

FIGURE 25 – Affichage d'une image masquée par une autre 1/5.

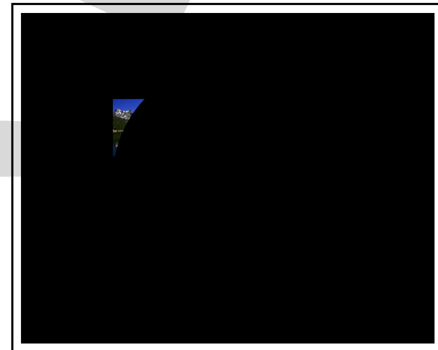

FIGURE 26 – Affichage d'une image masquée par une autre 2/5.

FIGURE 27 – Affichage d'une image masquée par une autre 3/5.

FIGURE 28 – Affichage d'une image masquée par une autre 4/5.

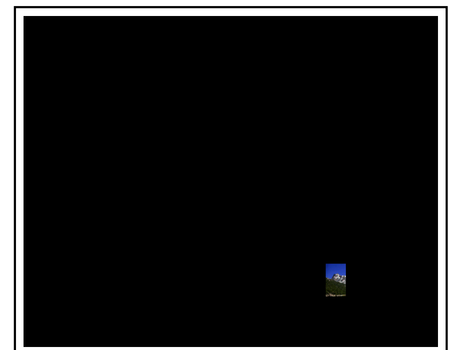

FIGURE 29 – Affichage d'une image masquée par une autre 5/5.

Et le dernier test vérifie le bon fonctionnement de l'affichage d'une partie d'image en tenant compte de l'opacité. Pour cela l'image de *Mario* est affichée colonne par colonne à l'écran.

DRAFT

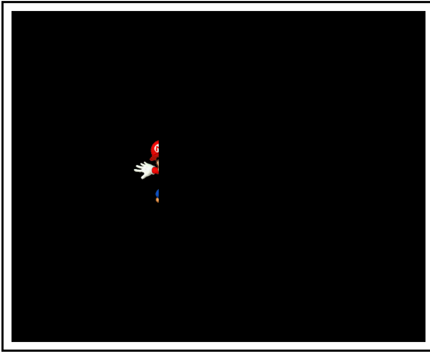

FIGURE 30 – Affichage d'une partie d'une image 1/3

FIGURE 31 – Affichage d'une partie d'une image 2/3

FIGURE 32 – Affichage d'une partie d'une image 3/3

4.3 Notes

- Pendant tous ces tests, des messages sont affichés dans la console.
- L'exécution de image_test nécessite la présence, dans le même répertoire, des fichiers `"splash.png"`, `"test_image.jpeg"`, `"test_image.tiff"`, `"mario.png"` et `"test_image_mask.png"`.

5 Bibliothèque Input

Les fonctions de la bibliothèque **Input** sont testées par le programme input_test.

Lors de son lancement, ce programme affiche un triangle qui se déforme à écran (cet affichage n'est là que pour contrôler que la bibliothèque **Input** s'exécute bien dans son propre *thread*, sans arrêter les affichages).

Le programme attend que l'utilisateur utilise l'une des entrées possibles et l'affiche au milieu de l'écran pendant un certain temps. Les actions `"up"`, `"down"`, `"left"` ou `"right"` sont affichées à leur position respective.

Le programme se termine lorsque l'utilisateur utilise l'entrée `"stop"`.

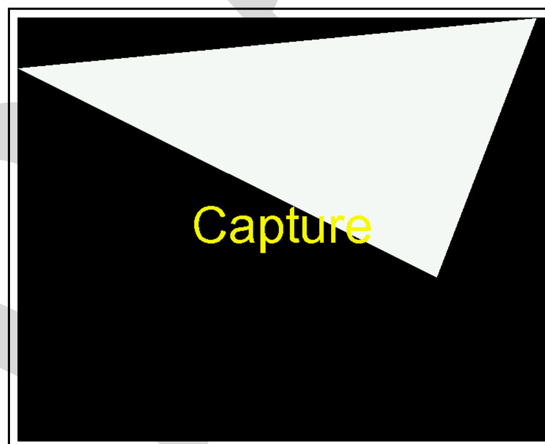

FIGURE 33 – Exécution du programme de test de la bibliothèque Input alors que l'utilisateur vient d'utiliser l'entrée "capture"

5.1 Notes

- Ce programme teste toutes les entrées simultanément les entrées via les trois possibilités : pseudo-console, clavier USB ou interface intégrée.
- Le clavier USB peut être branché en cours de test ou avant le test. Il peut également être débranché puis rebranché.
- L'exécution de input_test nécessite la présence du fichier `"Helvetica.ttf"` dans le même répertoire.

DRAFT

6 Bibliothèque Engine

Cette bibliothèque n'est qu'une pseudo bibliothèque (un réarrangement de bibliothèques fournies par *Texas Instrument* et *Montavista*). Il n'y a donc pas de tests pour cette bibliothèque.

7 Bibliothèque Video

Cette bibliothèque est chargée de gérer les images en provenance de la caméra et de permettre l'enregistrement et la relecture de vidéos au format AVI (utilisant les deux seuls *codecs* supportés par l'AVTBox).

7.1 Affichage simple de l'image de la caméra

Le premier test affiche simplement sur le moniteur l'image prise par la caméra. Le programme correspondant à ce test se nomme passthrough. Il prend fin lorsque l'action "stop" est utilisée.

7.2 Encodage vidéo

Le deuxième test de la bibliothèque **Video** permet d'enregistrer une vidéo depuis la caméra. L'affichage de ce test, nommé encode_test, est le même que celui du test précédent. Ce programme de test prend un paramètre qui doit être soit "mpeg" soit "h264", précisant le *codec* qui devra être utilisé pour encoder la vidéo. Pour terminer ce programme de test, l'utilisateur doit activer l'action "stop". À la fin de l'exécution de ce test, un fichier nommé test.mpeg.avi ou test.h264.avi suivant le cas est créé dans le répertoire courant.

7.3 Décodage vidéo

Le troisième test de la bibliothèque **Video** teste la relecture d'une vidéo enregistrée par le test précédent.

7.4 Conversion YUV 422 vers RGB 565

Afin de pouvoir sauvegarder des images en provenance de la caméra (en YUV 422) vers un affichage en RGB 565 ou un fichier en RGB 24, une conversion est nécessaire. Hors, il n'existe pas de formule parfaite permettant le passage d'un mode à l'autre. Une conversion a été implémentée dans la bibliothèque **Video**. Le programme yuv_rgb_test affiche une vidéo en YUV sur la droite de l'écran et la même vidéo en RGB565 (après transformation) sur la partie gauche de l'écran. Cela permet de vérifier que la formule utilisée est correcte et ne déforme pas les couleurs.

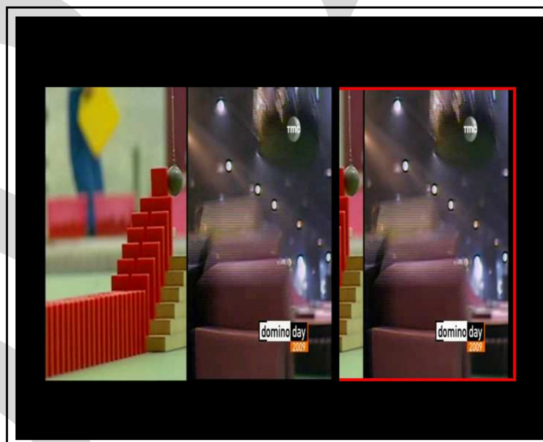

FIGURE 34 – Affichage du programme de test de conversion YUV vers RGB

7.5 Notes

Les corrections vidéos n'étant pas encore implémentées, il n'y a pas encore de programme permettant de tester leur implémentation.

DRAFT

8 Bibliothèque Widget

La bibliothèque **Widget** propose un grand nombre d'objets graphique ayant leur comportement propre. Cette bibliothèque dépend directement des bibliothèques **Base**, **Input** et **Graphic**. Ces trois bibliothèques doivent donc être testée préalablement.

Chaque type d'objet dispose de son propre test. Tous les tests de la bibliothèque **Widget** nécessitent la présence du fichier `"Helvetica.ttf"` dans le même répertoire.

8.1 label

Les *labels* sont les objets les plus simples : ils sont juste des morceaux de textes qui peuvent être affichés à l'écran. Leur comportement est testé par le programme label_test.

FIGURE 35 – Affichage simple de labels

8.2 Alignement des labels dans les boites horizontales

Le test suivant est label_alignment_test. Il permet de vérifier les différentes possibilités d'alignement (gauche, centre, droit, haut, bas, etc.) des labels placés dans des boites horizontales. La taille des labels doit être correctement calculée pour prendre en compte les jambages et les majuscules.

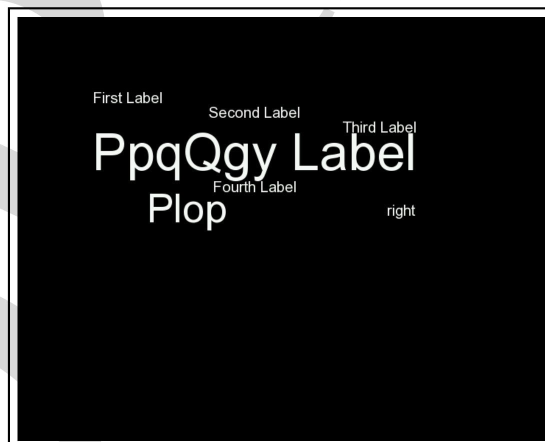

FIGURE 36 – Multiples labels avec différents alignements.

DRAFT

8.3 Boites verticales

Les différents *widgets* supportés par la bibliothèque **Widget** peuvent être inclus non seulement dans des boites horizontales mais aussi dans des boites verticales.

Le programme de test vbox_test permet de vérifier le bon fonctionnement de celles-ci.

FIGURE 37 – Test d'empilement vertical de labels avec différents alignements.

8.4 Boutons

Le *widget* vraiment actif le plus simple est le bouton, dont le rôle est de déclencher des actions lorsqu'il est pressé.

Le programme de test button_test affiche deux boutons l'un au dessus de l'autre. Celui qui a actuellement le focus est affiché en vidéo inversée. Il est possible de passer le focus de l'un à l'autre en utilisant l'action "Down". L'action "OK" quant à elle, invoque un traitement qui est réduit dans cet exemple à afficher "First Data" ou "Second Data" suivant le bouton qui a été utilisé.

FIGURE 38 – Test des boutons.

8.5 Cases à cocher

Les cases à cocher permettent de choisir un état (on ou off). Le programme de test check_button_test affiche trois cases à cocher. Celle qui a actuellement le focus est affiché en vidéo inversée. L'action "Down" permet de changer le focus d'une case à l'autre. L'action "OK" permet de changer l'état de la case, ce qui invoque un traitement affichant un texte qui permet de connaitre le nouvel état de la case qui vient d'être activée.

DRAFT

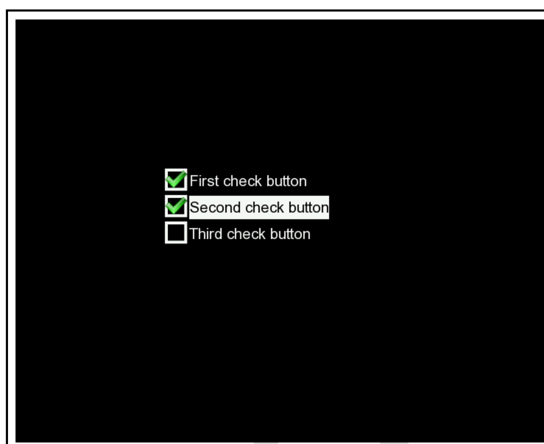

FIGURE 39 – Test des cases à cocher.

8.6 Boutons radio

Les boutons radio fonctionnent à peu près comme les cases à cocher si ce n'est que seul un parmi chaque groupe ne peut être activé à un instant donné. Le programme de test radio_button_test fonctionne sensiblement comme le précédent.

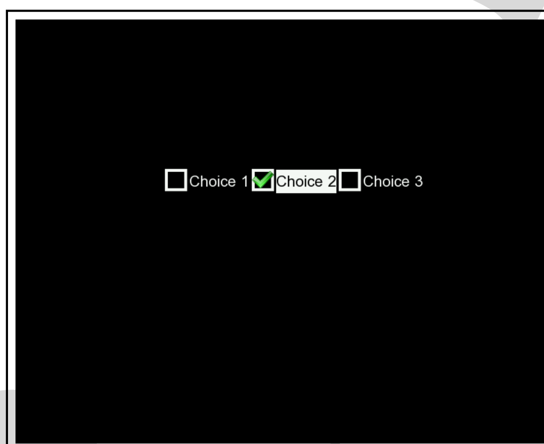

FIGURE 40 – Test des boutons radio.

8.7 Entrées de texte

Les *widgets* d'entrée de texte permettent d'entrer une ligne de texte.
Le programme permettant de les tester est text_entry_test.

DRAFT

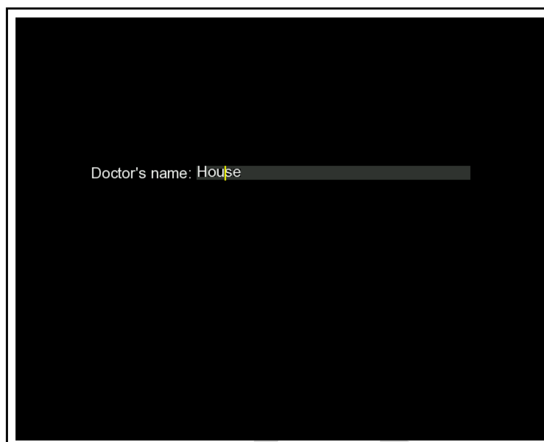

FIGURE 41 – Test des entrées de texte.

8.8 Séparateurs

Les séparateurs sont des *widgets* décoratifs affichant une ligne épaisse soit horizontale soit verticale. Ils permettent de séparer visuellement des groupes de *widgets*.

Le programme de test separator_test permet de vérifier le comportement des séparateurs en affichant quelques labels séparés par un séparateur vertical et un séparateur horizontal.

FIGURE 42 – Test des séparateurs.

DRAFT

8.9 Barres de progression

Figure 43 – Test des barres de progressions.

8.10 Labels messages

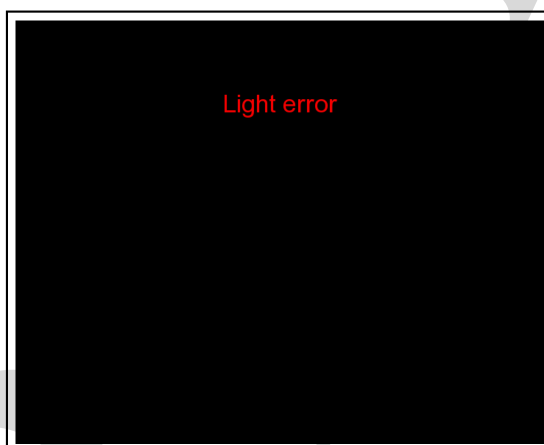

Figure 44 – Test des labels messages.

8.11 Fenêtres

La bibliothèque **Widget** propose également des fenêtres de haut niveau. Leur rôle est de contenir d'autres *widgets* en adaptant sa taille tout en restant centré sur l'écran. Ce *widget* permet donc de créer des boites de dialogue.

Ces fenêtres sont testées par le programme de test window_test.

DRAFT

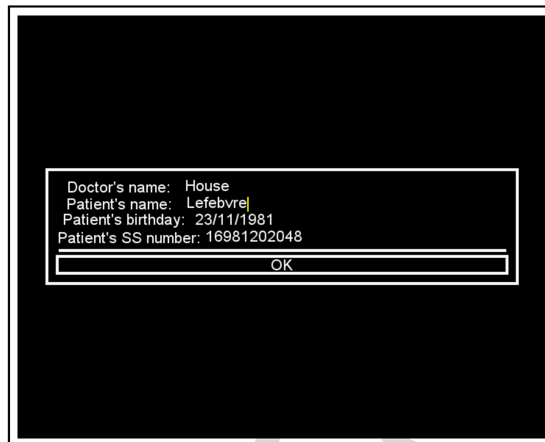

FIGURE 45 – Test des fenêtres.

8.12 Messages d'erreur

Les messages d'erreurs graves doivent être affichés dans des fenêtres prenant tout l'écran, avec un fond rouge et des écritures en noir. Aussi, la bibliothèque **Widget** propose des fenêtres de ce type-là : il s'agit des *WidgetErrorMsg*, autrement dit des messages d'erreur.

Ceci est testé par le programme de test error_msg_test.

FIGURE 46 – Test des messages d'erreur

8.13 Visibilité des widgets

Afin de répondre à certains besoin, les *widgets* doivent quelques fois être cachés momentanément pour être remontré par la suite.

Le programme de test visible_test vérifie que cela fonctionne bien en affichant quatre *labels* et une série de boutons permettant de changer la visibilité de ces labels et des boites qui les contiennent (deux boites horizontales et une verticale).

La figure 47 montre l'affichage de ce test au lancement, avec les quatre *labels* visibles. La figure 48 montre l'affichage de ce test après avoir utilisé le bouton « 2 » qui permet de cacher ou de montrer le *label* « Top Right ». La figure 49 montre l'affichage de ce test après avoir utilisé les boutons « 2 » et « 3 ». Les *labels* « Top Right » et « Bottom Left » sont donc cachés. La figure 50 montre l'affichage de ce test après avoir utilisé le bouton « 6 », qui cache la boite horizontale inférieure.

DRAFT

Figure 47 – Test de visibilité 1.

Figure 48 – Test de visibilité 2.

Figure 49 – Test de visibilité 3.

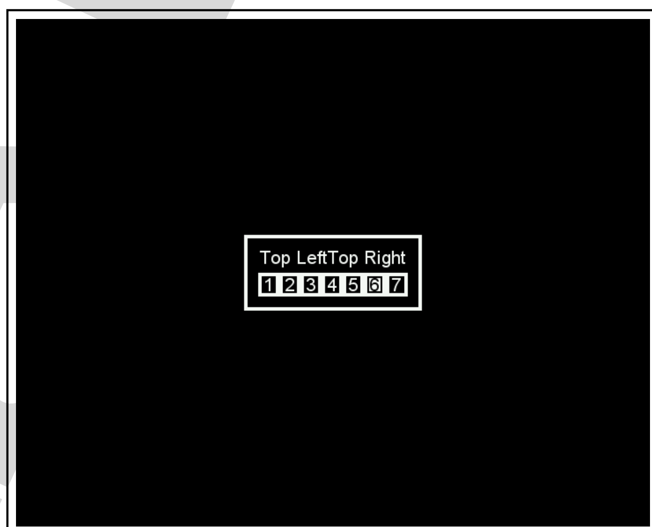

Figure 50 – Test de visibilité 4.

DRAFT

9 Tableau de validation des tests

Test	Validation	Remarque
Base		
Système de Log		
Listes chaînées		
Files		
Gestion des threads		
Graphic		
Primitives graphiques		
Affichage de texte		
Affichage d'images		
Utilisation de l'opacité		
Utilisation du double buffering		
Image		
Images opaques		
Images semi-transparentes		
Input		
Console		
Interface intégrée		
Clavier USB		
Video		
Affichage simple		
Encodage vidéo		
Décodage vidéo		
Conversion YUV / RGB		
Corrections vidéo	non-implémenté	
Widget		
Labels		
Alignement horizontal		
Empilement vertical		
Boutons		
Cases à cocher		
Boutons radio		
Entrées de texte		
Barres de progression		
Labels messages		
Fenêtres		
Message d'erreur		
Visibilité des Widgets		

DRAFT