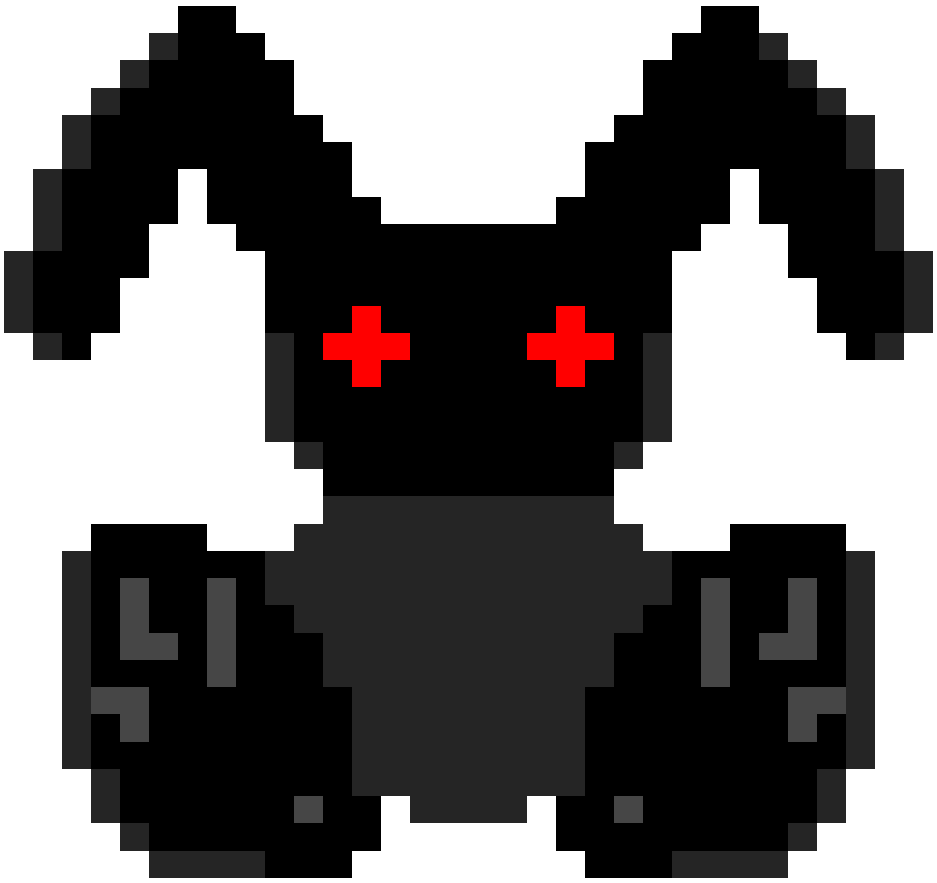


LAPINS NOIRS

Journée 9

Programmation graphique
Programmation audio

- La Caverne aux Lapins Noirs -

Dans cette journée, vous allez réaliser vos premiers programmes graphiques et sonores à l'aide de la LibLapin, une bibliothèque multimédia d'apprentissage

INDEX

01 – Avant-propos

Votre travail doit être rendu via le dossier **~/hyperespace/j8/** dans votre espace personnel.

Si vous faites erreur et que le dossier que vous utilisez pour votre rendu est différent, vous ne serez pas évalué faute d'avoir pu trouver votre travail.

Ce travail est a effectuer seul. Vous pouvez bien sur échanger avec vos camarades, néanmoins vous devez être l'auteur de votre travail. Utiliser le code d'un autre, c'est **tricher**. Et tricher annule **toutes** les médailles que vous avez reçu sur l'activité. La vérification de la triche est réalisée de la même manière que la correction : de manière **automatique**. Prenez garde si vous pensez pouvoir passer au travers.

Votre rendu doit respecter **strictement** l'ensemble des règles suivantes :

- Il ne doit contenir **aucun** fichier objet. (*.o)
- Il ne doit contenir **aucun** fichier tampon. (*~, #*#)
- Il ne doit pas contenir votre production finale (programme ou bibliothèque)

La présence d'un fichier interdit mettra immédiatement fin à votre évaluation.

Votre programme doit respecter les Tables de la Norme dans leur intégralité. Vous êtes invité à les observer depuis **l'Infosphère**. Elles sont disponibles comme ressource de cette activité.

02 – Fonctions autorisées

La bibliothèque logicielle venant avec le C est vaste et disponible. La LibLapin, que vous utilisez dans vos projets multimédia, est également vaste… Cependant nous avons fait le choix de vous interdire leurs utilisation intégrales, afin de vous amener progressivement à reprogrammer vous même ses fonctionnalités les plus utiles.

L'utilisation d'une fonction interdite est assimilée à de la triche. La triche provoque l'arrêt de l'évaluation et la perte des médailles.

Vous n'avez le droit d'utiliser aucune fonction issue de la LibC ou de la LibLapin à l'exception de celles que nous vous autoriserons explicitement.

Pouvoir utiliser une fonction ne signifie pas nécessairement que celle-ci soit utile à votre cas.

Pour cette activité, issu de la LibC, vous n'avez le droit qu'à la liste suivante :

- open
- close
- read
- srand
- cos
- atan2
- write
- alloca
- atexit
- rand
- sin
- sqrt

Remarquez bien l'absence de **malloc** et de **free**. En effet ils sont **interdits** ! Issu de la LibLapin, vous avez le droit aux fonctions suivantes :

- bunny_start
- bunny_stop
- **bunny_malloc**
- **bunny_free**
- bunny_new_pixelarray
- bunny_delete_clipable
- bunny_blit
- bunny_display
- bunny_set_*_function
- bunny_set_*_response
- bunny_loop
- bunny_create_effect
- bunny_compute_effect
- bunny_sound_play
- bunny_sound_stop
- bunny_delete_sound
- bunny_open_configuration
- bunny_delete_configuration
- bunny_configuration_getf
- bunny_configuration_setf
- bunny_configuration_*
- bunny_set_memory_check
- bunny_release
- bunny_usleep

La suite sur la page d'après.

Pour utiliser **bunny_malloc**, vous pouvez soit programme directement avec, soit en utilisant le modèle de projet qui vous a été transmis, mettre **1** dans la variable de Makefile **BMALLOC**, qui transformera **malloc** en **bunny_malloc**.

Vous appellerez **bunny_set_memory_check** au début de votre fonction **main** de sorte à provoquer une vérification de vos allocations à la fermeture du programme.

bunny_malloc, par défaut, limitera votre consommation de RAM à 20Mo.

Pour information :
Une image en 1920*1080 fait environ 8Mo.

Une musique en 44kHz de 1 minute en stéréo fait environ 10Mo.

Vous devrez donc disposer d'une discipline de fer avec vos allocations... et probablement trouver des compromis.

L'utilisation de **bunny_malloc** parfois **cachera** des erreurs dans votre programme, et parfois en **révélera** : son principe d'allocation était différent de **malloc**, il sera parfois plus « fort » ou plus « faible ». Ne vous mentez pas à vous en disant « l'utilisation de **bunny_malloc** fait planter mon programme », ce n'est pas **bunny_malloc**, c'est *vous*.

N'hésitez pas à l'activer, à le désactiver (Et lorsqu'il est désactivé, à utiliser valgrind). Et n'oubliez pas que désormais, votre demande de RAM a de véritables chances d'échouer. Car exploiter aussi peu de RAM, cela va très vite…

Dans votre rendu, il ne devra pas y avoir la moindre trace de malloc.

03 – Méthode de construction

Il peut vous être demandé d'écrire des programmes ou des fonctions.

Dans le cas des programmes, il vous sera toujours demandé de fournir un **dossier** pour l'exercice le requérant. Un **Makefile** vous sera également demandé. Le **nom du programme** de sortie vous sera précisé à chaque fois. Un Makefile incorrect, un mauvais nom de programme, et votre correction n'aura pas lieu…

Dans le cadre des fonctions, il vous ai demandé de fournir le fichier dans votre **dossier de bibliothèque personnelle**, de sorte à ce que vous puissiez utiliser toutes les fonctions que vous avez déjà réalisé jusqu'ici. Pour rappel, le dossier de votre bibliothèque doit être placé à la racine de votre espace personnel et s'appeler **libstd/.**

N'oubliez pas d'entretenir avec soin votre dossier **libstd/** de sorte à ce qu'il soit toujours propre, respecte la norme et soit en état de compiler… sans quoi elle fera obstacle à la correction.

Votre compilation devra toujours comporter les options **-W**, **-Wall** et **-Werror**.

Dans le cadre de la programmation multimédia, le système de correction établira toujours la variable d'environnement **BMALLOC** à 1. Si vous utilisez le modèle de projet, cela provoquera l'utilisation de **bunny_malloc** dans votre bibliothèque personnelle comme dans votre projet rendu.

04 – Un peu d'histoire

Autrefois, les ordinateurs disposaient de témoins lumineux, allumés ou éteints, comme sortie d'information simple. Pour les cas plus complexe, ils disposaient d'une imprimante. Pendant très longtemps, les **« pupitres »** comportaient un clavier face à une collection de lampes et d'interrupteurs à coté d'une imprimante.

Depuis un peu moins de quarante ans, nos ordinateurs sont équipés **d'écrans**.

Cela n'a pas remplacé les témoins lumineux qui restent présents pour des taches primitives : état de la machine, accès disque, ni l'imprimante, qui reste présente également. L'écran a par contre pris leur place en tant qu'élément de communication machine vers homme principal.

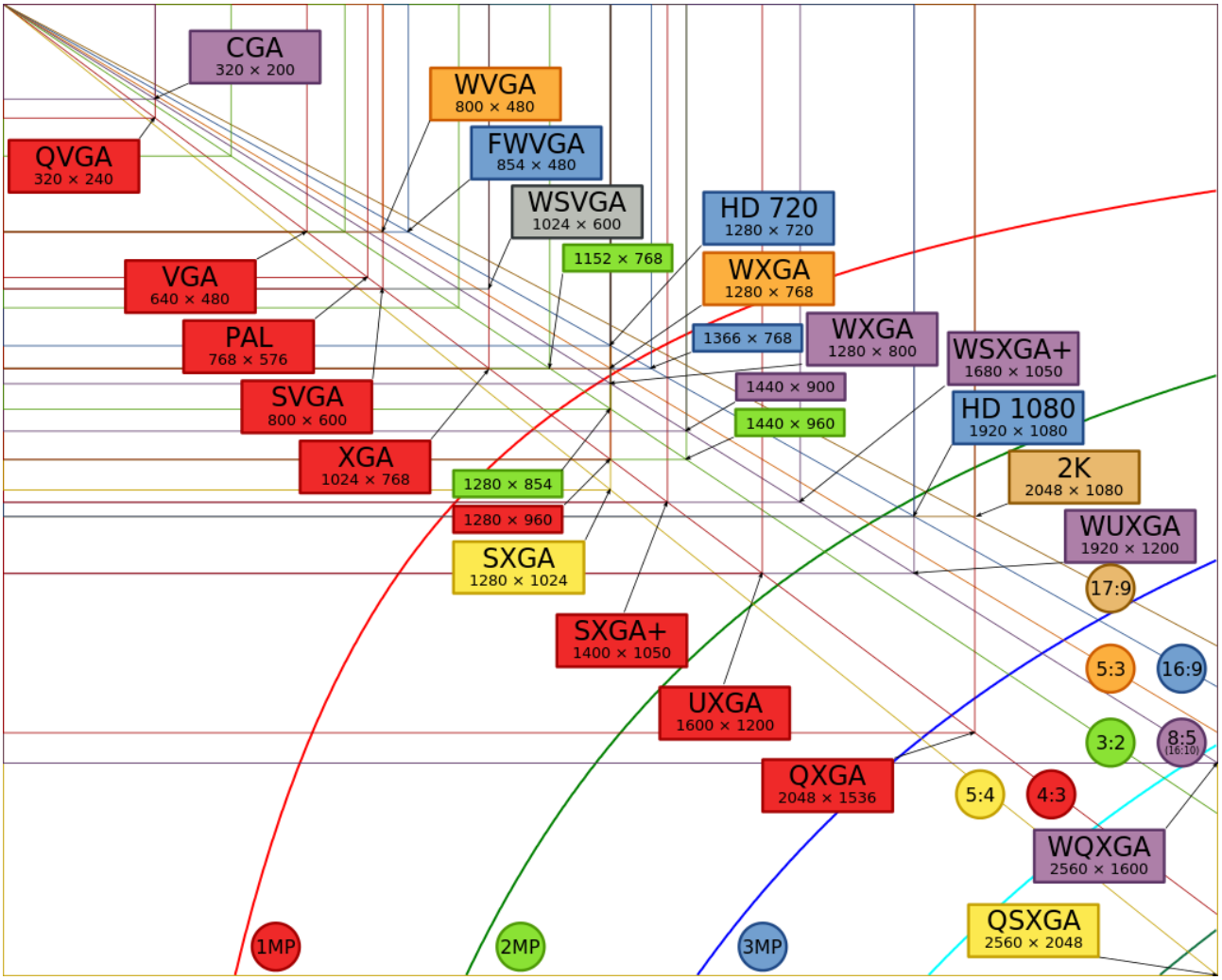
Un **écran** est une surface sur laquelle on projette une image. Cette image est rafraîchie plusieurs fois par seconde, permettant ainsi de modifier ce que l'on y voit.

L'image elle-même est un très grand ensemble de points lumineux. Nous appelons ces points des **pixels**. Un écran ayant une largeur et une hauteur, l'image dispose elle aussi d'une largeur et d'une hauteur.

Ces pixels peuvent-être plus ou moins petit. Plus ils sont petit, plus il est difficile des les discerner et plus pour une même taille d'écran la largeur et la hauteur de cette image seront grands.

On appelle la taille d'une image sa **résolution**.

Plusieurs résolutions d'écrans ont existé durant l'histoire des ordinateurs.



Au départ, les fabricants d'ordinateurs personnels, durant les années 70 principalement, ne fournissaient pas la résolution réelle de leurs machine en pixel mais leur taille en **caractère**, du fait qu'il était **impossible** d'accéder directement au pixel à cause du manque de mémoire.

Un Commodore PET de 1977 permettait d'afficher 25 lignes de 40 caractères. Si l'on considère que chaque caractère peut faire au minimum 5 pixels sur 5, on obtient une résolution de 200*125 pixels. Notez que cet écran est **monochrome** et que cette résolution au pixel près n'est pas utilisable par manque de mémoire vive. Accéder à chaque pixel, même si chacun ne contient qu'un seul bit de couleur amène la consommation de mémoire vidéo à 3125 octets, alors qu'un accès par caractère n'en consomme que 1000.

Le fait de ramener un ensemble de pixels (ici, des blocs de 5*5 pixels comprenant un caractère) s'appelle un système de **tuile**. Il est fréquent aussi de ramener des couleurs complexes à un numéro plus court, dans ce cas on parle de **palette de couleur**.

Pour votre culture, l'architecture de la Game Boy tourne autour de ce procédé. De nombreux jeux vidéo utilisent aussi ce principe : Zelda par exemple.

En comparaison, un Atari ST de 1985 dispose de trois modes de fonctionnement : base résolution, avec une résolution d'écran de 320*200 pouvant employer une palette de 16 couleurs (4bits) choisis parmi 4096 **« couleurs vraies »**, moyenne résolution avec une résolution de 640*200 et seulement 4 couleurs (2bits), ou 640*400 en monochrome (1bit).

Couleurs vraies signifie que la couleur est construite en ajustant ses composantes de couleur rouge, verte et bleue et non en puisant dans une palette avec un nombre arbitraire.

Un PC au milieu des années 90 atteignait en général 640*480 (VGA) en 256 couleurs fixes, 800*600 (SVGA) en 16 bits couleurs vraies pour les configurations plus sportives, jusqu'à 1024*768 (XGA) en 24 bits début 2000, qui fut le standard plusieurs années.

Aujourd'hui, un moniteur classique atteint 1920*1080 (HD 1080) en 24 bits et nous commençons à tendre vers du « 4K », soit 3840*2160. Il est intéressant de noter que cette évolution est susceptible d'être ralentie hors de l'audiovisuel par le fait que les pixels sont de moins en moins dissociables, et que de ce fait, multiplier leur nombre sans agrandir la taille de nos écrans est moins intéressant qu'autrefois : il n'y a pas autant de gain esthétique qu'auparavant, les éléments à câbler matériellement sont plus nombreux, logiciellement, la surface à traiter est plus large et la consommation électrique globale augmente de ce fait.

Concernant les images affichées à l'écran et non l'écran lui-même, une nuance doit être apportée à la quantité de bits des couleurs : il y en a **32** et non 24. En effet, les bits supplémentaires sont généralement utilisé pour contenir la transparence du pixel. Cette transparence s'applique lorsqu'une couleur est placée par dessus une autre. **Ainsi, aujourd'hui, nous ne manipulons que des pixels faisant 32 bits, soit 4 octets.**

07 – Votre première fenêtre graphique

Nous allons donc utiliser la **« LibLapin »**.

La **LibLapin** est une bibliothèque multimédia, orienté jeu vidéo dont la structure est conçue pour servir de support à l'apprentissage de la programmation en C et de la programmation graphique.

Sa documentation est disponible sur le bouton « LibLapin » de l'intranet, où à l'adresse suivante. Celle de l'intranet est plus récente.

http://hangedbunnystudio.com/sub/liblapin/

Ce premier exercice ne sera pas évalué.

Commencez par ouvrir une fenêtre graphique. Cela se fait avec la fonction **bunny_start**. Pour faire apparaître l'intérieur de la fenêtre, vous devrez utiliser **bunny_display**. Ensuite, vous attendrez une seconde à l'aide de **bunny_usleep** et enfin vous fermerez la fenêtre avec **bunny_stop**.

08 – number_pixelarray

Mettre dans libstd, src/number_pixelarray.c

Maintenant, nous allons dessiner dans la fenêtre que vous avez ouvert. Pour cela, nous allons demander à votre système d'exploitation un morceau de mémoire graphique : une **image**. Pour cela, vous allez utiliser **bunny_new_pixelarray**.

Créez l'image de la même taille que la fenêtre.

Pour coller votre image sur la fenêtre, vous utiliserez **bunny_blit**. Une difficulté ? En effet, **bunny_blit** nécessite un **t_bunny_buffer** et un **t_bunny_clipable**. Un petit tour dans la documentation de **t_bunny_window** et de **t_bunny_pixelarray** vous apprendra que **t_bunny_window** contient un **t_bunny_buffer** et que **t_bunny_pixelarray** contient un **t_bunny_clipable**…
Tout va bien alors. A vous de trouver comment envoyer ces « sous-parties » à la fonction **bunny_blit…** Un indice ? Comment récupérer leur adresse ?

Une fois que vous aurez réussi à trouver comment faire, nous allons dessiner dans la mémoire vidéo de votre ordinateur. Où se trouve-t-elle ? Elle est dans le champ **pixels** du **t_bunny_pixelarray**…

Pour commencer, posez-vous la question : qu'est ce qu'un pixel ? Quel type de variable permettrait d'accéder aux pixels un par un ? La réponse est dans le chapitre d'histoire du début du sujet.

Une fois que vous aurez répondu à cette question, vous allez numéroter les pixels : vous allez mettre dans le pixel 0 une valeur de 0. Dans le pixel 1 une valeur de 1. Dans le pixel 2 ? Une valeur de 2 ! Etc. Affichez ensuite le résultat. L'opération de numérotage devra être faite dans la fonction suivante :

```
void std_number_pixelarray(t_bunny_pixelarray *picture);
```

Ne rendez que la fonction. Le reste du code, constitué de la création de la fenêtre, du **bunny_blit**, etc. font par de votre « main de test », comme durant le reste des exercices de la période d'apprentissage.

Cette fonction doit fonctionner pour n'importe quel **t_bunny_pixelarray**, quelque soit sa taille ! Le rendu va être plutôt psychédélique.

09 – clear_pixelarray

Mettre dans libstd, src/clear_pixelarray.c

Vous allez maintenant écrire une autre fonction permettant de manipuler des images.

```
void std_clear_pixelarray(t_bunny_pixelarray *picture,
                          unsigned int       color);
```

Cette fonction va appliquer **color** à chaque pixel de **picture**. La **LibLapin** comporte plusieurs **constantes de précompilation** faisant office de couleur. Essayez en quelques unes : **RED**, **GREEN**, **PINK2**, etc.

Il existe plusieurs types de bibliothèque graphique : les bibliothèques graphiques exploitant le micro-processeur seulement : SDL1, Allegro4… et les bibliothèques graphiques exploitant principalement la carte graphique : SDL2, Allegro5, SFML…

La **LibLapin** est une bibliothèque **mixte**. Elle permet de faire soit l'un soit l'autre, et même de mélanger les deux. Dans la LibLapin, vous trouverez un type : **t_bunny_picture**, représentant les images dont les traitements sont fait par la carte vidéo. A contrario, **t_bunny_pixelarray** représente les images dont le traitement est fait par le micro-processeur : c'est à dire par votre code directement.

La **LibLapin** est une bibliothèque à **trou**. Elle dispose de très nombreuses fonctions mais une large partie d'entre elles ne fonctionnent que sur les **t_bunny_picture**. Pourquoi ? Parce que faire fonctionner celle-ci sur **t_bunny_pixelarray** est à **votre** charge !

Comment faire ? Et bien vous venez d'accomplir un premier travail de complétion avec cette fonction. En effet, il existe une fonction **bunny_clear** qui permet de mettre tous les pixels d'une image d'une seule et unique couleur… et une variable globale **gl_bunny_my_clear**, qui est un pointeur sur fonction, qui vaut par défaut **NULL**. En attribuant l'adresse de votre fonction a **gl_bunny_my_clear**, vous permettez a **bunny_clear** d'opérer également sur **t_bunny_pixelarray** !

Essayez dès à présent ! Assignez votre fonction, et au lieu de l'appeler explicitement, passez maintenant par **bunny_clear** !

Chaque fonctionnalité réalisé *parfaitement* pour t_bunny_pixelarray débloquera une fonctionnalité pour t_bunny_picture disposant de l'accélération graphique.

Pour disposer du droit d'utiliser **bunny_clear** sur **t_bunny_picture**, vous devrez être capable de limiter le remplissage du **pixelarray** à la zone déterminée par les propriétés du **t_bunny_clipable** : **clip_x_position**, **clip_y_position**, **clip_width** et **clip_height**.

10 – noise_pixelarray

Mettre dans libstd, src/noise_pixelarray.c

Vous allez maintenant écrire une autre fonction permettant de manipuler des images.

```
void std_noise_pixelarray(t_bunny_pixelarray *picture);
```

Cette fonction va appliquer une couleur aléatoire à tous les pixels de **picture**. Pour réaliser cette couleur aléatoire, vous partirez du premier pixel et irez jusqu'au dernier en leur appliquant **rand() | BLACK**.

Si vous jouez cette fonction en boucle dans votre main, en affichant a chaque fois le contenu de l'image, vous aurez un premier programme **animé**.

11 – clear_effect

Mettre dans libstd, src/clear_pixelarray.c

Vous allez maintenant écrire une première fonction permettant de jouer du **son** – ou presque. *Pour éviter tout risque d'incident, baissez le volume de votre ordinateur.*

```
void std_clear_effect(t_bunny_effect    *effect);
```

La création d'un son passe par la fonction **bunny_create_effect**. Cette fonction demande le nombre de secondes que durera le son, et le nombre **d'échantillons** par **seconde**. Une qualité commerciale serait **44100**.

Qu'est ce qu'un son ? C'est une **vibration**. Cette vibration est un état **fluctuant**, passant d'un état à l'autre à des fréquences différentes. Plus la **fréquence** est **élevée**, plus le son est **aigu**. Plus **l'oscillation** est **forte**, plus le **volume** est élevé.

On appelle le **temps** d'une **unique oscillation** la **période**. On appelle **l'écart** entre le **niveau** de fluctuation le **plus bas** et le **plus haut l'amplitude**.

Un **échantillon** (sample, en anglais) est l'enregistrement **ponctuel** de cette fluctuation. Une fréquence d'échantillonnage de 44100 indique que 1 seconde, il y a 44100 échantillons. Un échantillon est représenté par un **entier sur 16 bits signé** (un *short int*).

Dans un premier temps, notre son va être très silencieux. En effet, nous allons mettre tous les échantillons à zéro. Pour cela, vous devrez savoir jusqu'où écraser les valeurs du tableau **sample** du **t_bunny_effect**. *Vous disposez de toutes les informations pour le faire dans la structure !*

Pour jouer votre son, vous devrez d'abord l'envoyer à la carte son avec **bunny_compute_effect**, et ensuite le jouer avec **bunny_sound_play**. Si votre programme se termine avant votre son, évidemment, *rien ne sera joué.* Utilisez donc **bunny_usleep** pour savourez votre silence, ou placez une boucle infinie, à votre convenance.

12 – noise_effect

Mettre dans libstd, src/noise_effect.c

Vous allez maintenant écrire une première fonction permettant de jouer du **son – pour de** vrai. *A nouveau, attention au volume de votre ordinateur, plus encore si vous portez un casque !*

```
void std_noise_effect(t_bunny_effect    *effect);
```

Cette fonction placera des valeurs aléatoires comprise entre -32000 et 3200 dans les échantillons. Vous utiliserez rand() de manière appropriée pour varier dans cette amplitude. Vous remplirez entièrement l'espace son disponible.

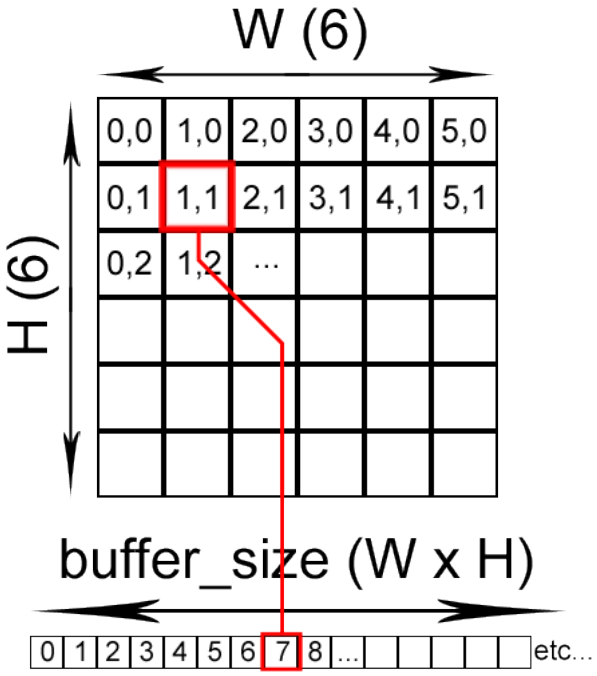
13 – set_pixel

Mettre dans libstd, dans src/set_pixel.c

Vous voila arrivé sur la pierre angulaire de la programmation graphique. Vous allez maintenant écrire une fonction permettant d'attribuer une couleur à un pixel précis. Problème : une image est en deux dimensions et votre tableau n'en a qu'une seule…

```
void std_set_pixel(t_bunny_pixelarray *picture,
                   t_bunny_position    pos,
                   unsigned int        color);
```

Cette fonction va attribuer au pixel situé à la position **(pos.x, pos.y)** la couleur **color**.



Sur la gauche, un schéma représentant une image et le tableau qui la contient. Il est important de noter que sur ordinateur, le coin supérieur gauche est l'origine, contrairement au modèle mathématique ou c'est le coin inférieur gauche !

L'image ici est de 6 pixels de largeur et de 6 pixels de haut. Le premier pixel, en haut à gauche, est aux coordonnées X=0 et Y=0. Ce pixel est en toute logique, l'équivalent de la case 0 du tableau… Sur le même principe, le pixel suivant, en X=1 et Y=0 devrait donc être l'équivalent de la case 1 du tableau, et ainsi de suite…

La difficulté consistera alors, à partir des informations dont vous disposez sur l'image (sa taille) et de la position où vous souhaitez poser votre pixel (X et Y), à trouver la formule pour choisir la bonne case dans le tableau.

De la même manière qu'il existait **gl_bunny_my_clear** pour **bunny_clear,** il existe **gl_bunny_my_set_pixel** pour **bunny_set_pixel** !

14 – set_square

Mettre dans libstd dans src/set_square.c

Maintenant que vous avez votre fonction de pose de pixel… Réalisez la fonction ci-dessous, qui vous sera utile pour réaliser un **bunny_clear** complet – entre autres !.

```
void std_set_square(t_bunny_pixelarray *picture,
                    t_bunny_area        area,
                    unsigned int        color);
```

Cette fonction dessine un carré de couleur **color** dont le coin supérieur gauche est à **(area.x, area.y)** et de taille **(area.w, area.h)**.