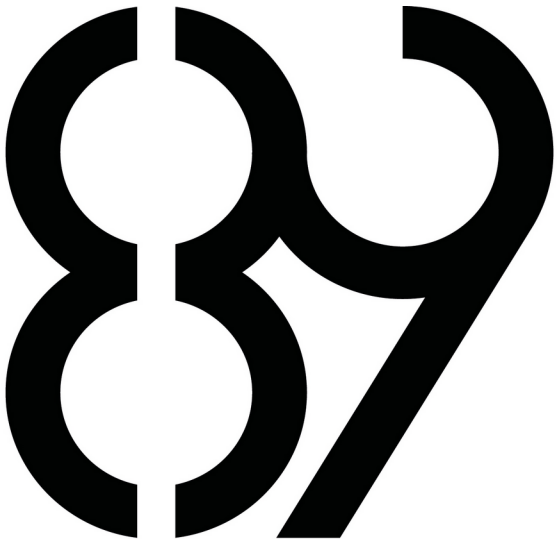


Journée 09

Programmation graphique

- Le Laboratoire aux Lapins Noirs -
lapinsnoirs@ecole-89.com

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

Nom de code : !pac1j09
Clôture du ramassage : 22/12/2019 23:59

Médailles accessibles :

Définition des médailles à venir

INDEX

01 – Détails administratifs

Votre travail doit être envoyé via l'interface de ramassage du **TechnoCentre** :

http://technocentre.ecole89.com/ramassage

Le numéro de code présent sur ce sujet vous est propre : vous devrez le renseigner en rendant votre travail. En cas d'erreur, votre travail ne sera pas associée à l'activité et votre travail ne sera pas ramassé.

Pour cette activité, vous rendrez votre travail sous la forme d'une archive au format tar.gz. Cette archive devra contenir l'ensemble de votre travail tel que demandé dans la section 4.

Pour créer cette archive .tar.gz, il vous suffit d'utilliser la commande suivante :

```
$> tar cvfz mon_fichier.tar.gz fichier1 fichier2 fichier3
```

Le nom « mon_fichier.tar.gz » étant à remplacé par le nom que vous souhaitez donner votre fichier archive, et « fichier1 », « fichier2 », « fichier3 » par les fichiers ou dossiers que vous souhaitez mettre dans cette archive. Vous pouvez vérifier le contenu de votre archive à l'aide de la commande « **`tar -t mon_archive.tar.gz`** ».

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.

Médailles accessibles :

Réussir à rendre :
Vous avez réussi à envoyer votre travail au système de correction.

02 – Propreté de votre rendu

Votre rendu, c'est à dire le contenu de l'archive ou du dépôt que vous entrez sur l'interface du TechnoCentre, doit respecter **strictement** l'ensemble des règles suivantes :

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

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

Médailles accessibles :

Rendu propre
Votre rendu respecte les règles de propretés imposées.

03 – Règlement quant à la rédaction du code C

En général, le code source de votre programme doit impérativement respecter un ensemble de règles de mise en page définie par les Table de la Norme.

Pour cette activité, nous vous libérons de cette contrainte qui vous sera néanmoins très bientôt imposée pour l'ensemble de vos programmes écrits en C.

04 – Construction partielle de votre rendu

Le programme de correction va construire une sous-partie déterminée de votre rendu afin d'effectuer des tests dessus. En voici les paramètres :

- Les fichiers qui seront compilés sont ceux qui auront l'extension *.c.
- Seuls les fichiers dans le(s) dossier(s) ./ seront compilés.
- Les fichiers seront compilés avec **-W -Wall -Werror**.

- Tous les fichiers seront compilés **ensemble**, cela signifie donc que chaque fonction doit être unique, et que vous pouvez utiliser les fonctions des autres exercices dans chaque exercice.

L'ensemble du code que vous rendez doit pouvoir être compilé.
En cas d'échec de la compilation, vous ne serez pas évalué.

Médailles accessibles :

Construction partielle
Les éléments requis de votre projet se construisent séparement.

05 – Fonctions interdites

Vous n'avez le droit à aucune autre fonction que celle précisée dans la liste ci-dessous :

Fonctions systèmes

- open
- close
- write
- read
- ioctl

Fonctions de la LibLapin

- bunny_start
- bunny_stop
- bunny_new_pixelarray
- bunny_delete_clipable
- bunny_blit

- bunny_set_loop_main_function
- bunny_set_display_function
- bunny_set_*_response
- bunny_loop

- bunny_malloc
- bunny_free

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.

06 – 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.

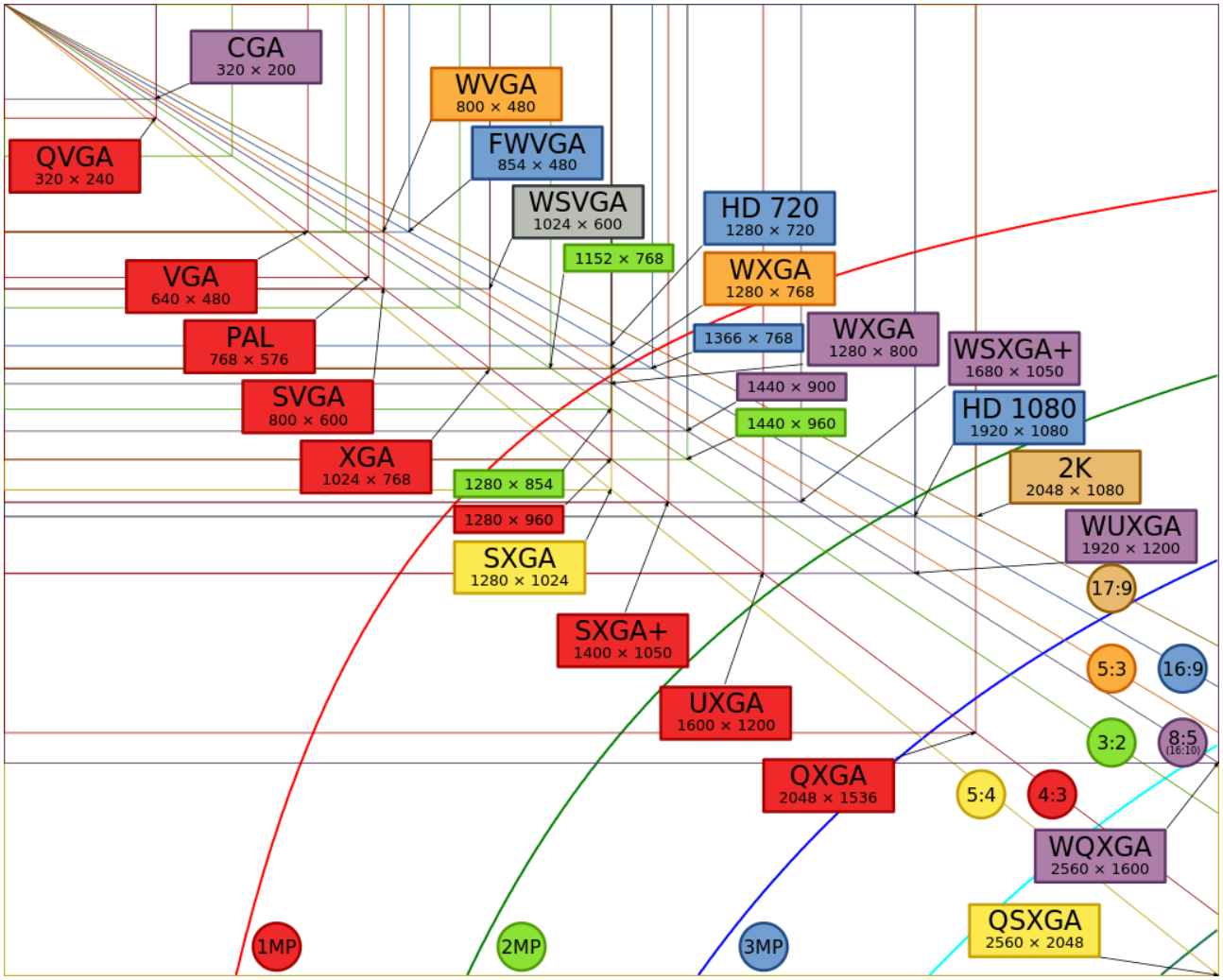
Un **écran** est une surface sur laquelle on projette une image. Cette image est rafraîchie plusieurs fois par seconde, permettant ainsi de modifier ce que l'on y voit.

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 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.

Vous trouverez sa documentation à l'adresse suivante :

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 le fichier : 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** néccessite 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**.

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 ?

Une fois que vous aurez répondu à cette question, vous allez numéroter les pixels : le pixel 0 aura comme valeur 0. Le pixel 1 comme valeur 1, etc. Affichez ensuite le résultat. L'opération de numérotage devra être faite dans la fonction suivante :

```
void 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.

08 – clear_pixelarray

Mettre dans le fichier : clear_pixelarray.c

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

```
void e89_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 ? Parceque 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** !

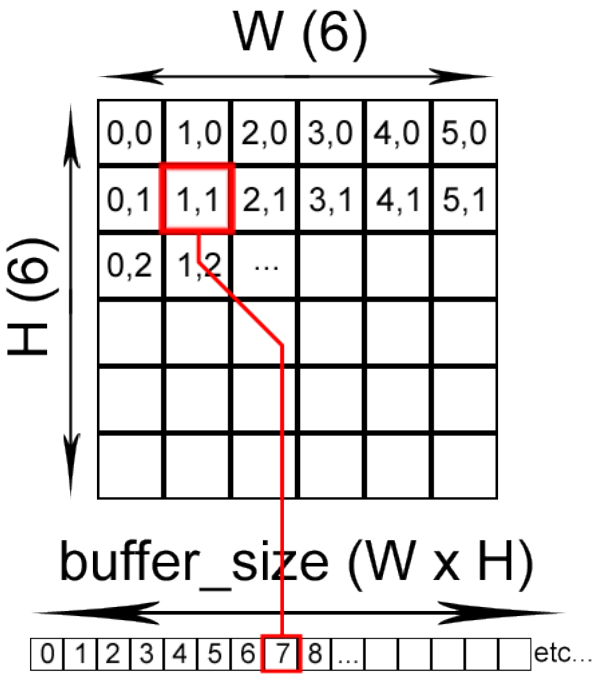
09 – set_pixel

Mettre dans le fichier : 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 e89_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 !

09 – set_square

Mettre dans le fichier : set_square.c

Maintenant que vous avez votre fonction de pose de pixel… Réalisez la fonction ci-dessous :

```
void e89_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).