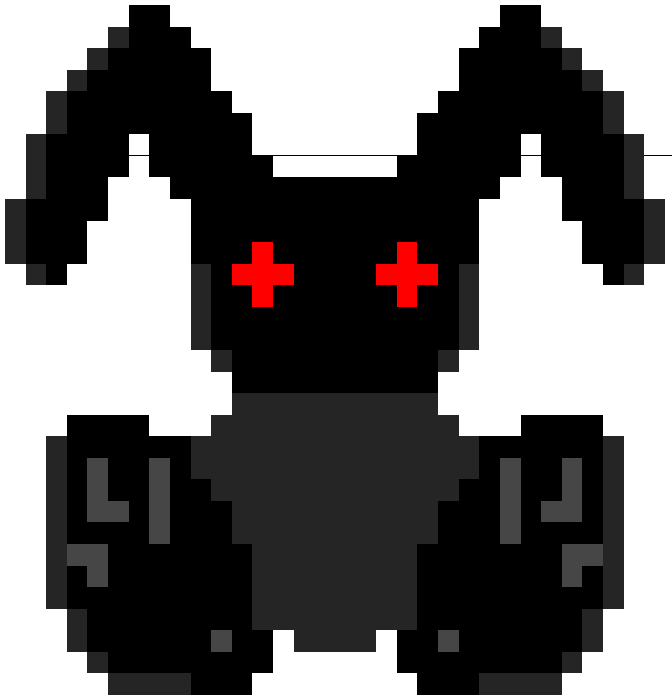

LAPINS NOIRS

- La Caverne Aux Lapins Noirs -

Lancer un rayon

- La Caverne aux Lapins Noirs -

Ce TP est une première étape dans la réalisation de votre premier projet graphique d'envergure, ***Runner****.*

INDEX

01 – Détails administratifs

Votre travail doit être rendu via le dossier **~/tp/lazer/** 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 – Calculer un déplacement

Réalisez les fonctions suivantes :

```
t_bunny_accurate_position     std_walk_to(t_bunny_accurate_position     start,
                                          double                       angle,
                                          double                       len);
t_bunny_position              std_position(t_bunny_accurate_position    pos);
```

La fonction **std_walk_to** calcule la position d'un point qui serait situé à **len** distance de **start** dans la direction **angle.**

La fonction **std_position** effectue un arrondi du **t_bunny_accurate_position pos** en **t_bunny_position**. Cet arrondi est une **troncature**, comme c'est le cas dans n'importe quelle conversion d'un nombre flottant vers un entier en C. *C'est une toute petite fonction.*

04 – Lancer un rayon

Programmez maintenant la fonction suivante :

```
typedef struct          s_map
{
        int             tile_size;
        int             width;
        int             height;
        int             *map;
}                       t_map;

t_bunny_accurate_positon std_send_ray(t_map                     *map,
                                      t_bunny_accurate_position start,
                                      double                    angle);

void                     std_draw_impact(t_map                  *map,
                                         t_bunny_pixelarray     *px,
                                         t_bunny_accurate_position start,
                                         double                    angle);
```

La structure **t_map** fait **width** tuiles de large et **height** tuiles de haut. Une tuile fait **tile_size** pixels de haut et de large. Cela veut dire que chaque case de **map** mesure à l'écran **tile_size** pixels. Le tableau **map** fait **width** * **height** cases de long et est rempli de valeur entière valant soit 0 soit 1. Les 0 sont des cases qu'il est possible de parcourir. Les 1 sont des « murs ».

La fonction **std_send_ray** lance un **rayon** depuis la position **start** avec l'orientation **angle**. Lancer un rayon signifie ici partir de la position **start** et avancer sur **map** dans la direction **angle** jusqu'à heurter une case valant 1 ou jusqu'à sortir de la carte. **std_send_ray** va renvoyer la **position de collision**.

La fonction std_draw_impact fait appel à std_send_ray pour calculer le point de collision, mais en plus, à l'endroit où a lieu la collision, elle place un pixel **rouge**. N'oubliez pas que chaque tuile fait **tile_size** et que donc si la collision à lieu à la position 3 * 2 dans **map**, alors le pixel rouge devrait être positionné vers **3 * tile_size – 2 * tile_size** *environ*.

Vous trouverez sur la page suivante un programme initiant les éléments fondamentaux dont vous aurez besoin pour tester votre fonction.


```c
#include                        <lapin.h>
#include                        <assert.h>
#include                        <math.h>

int                             main(void)
{
        t_bunny_window          *win;
        t_bunny_pixelarray      *px;
        t_bunny_accurate_position pos;
        double                  angle;
        int                     mx[6 * 6] = {
                1, 1, 1, 1, 1, 1,
                1, 0, 0, 0, 0, 1,
                1, 0, 0, 0, 0, 1,
                1, 0, 0, 0, 1, 1,
                1, 1, 0, 1, 1, 1,
                1, 1, 1, 1, 1, 1,
        };
        t_map                   map;

        map.width = 6;
        map.height = 6;
        map.tile_size = 100;
        map.map = &mx[0];
        pos.x = 2.5;
        pos.y = 2.5;
        angle = 0;

        win = bunny_start(
                map.width * map.tile_size,
                map.height * map.tile_size,
                false,
                "LaZeR"
        );
        px = bunny_new_pixelarray(win->buffer.width, win->buffer.height);
        std_clear_pixelarray(px, BLACK);

        // Travaillez ici

        bunny_blit(&win->buffer, &pix->clipable, NULL);
        bunny_display(win);

        bunny_usleep(5e6);
        bunny_delete_clipable(&pix->clipable);
        bunny_stop(win);
        return (0);
}
```

05 – Un joli programme

Votre programme effectuera les tâches suivantes :

- Remplir de noir votre image. Si vous savez faire de la transparence, arrangez vous pour que ce noir soit très transparent.

- Lancer un rayon dans la direction **angle,** mais n'utilisez pas **std_draw_impact**, à la place, récupérez la position de l'impact renvoyée par **std_send_ray** et tracez une ligne entre la position **start** et le point d'impact.

- Augmenter **angle** d'une fraction ridicule de PI.

- Recommencez.

Vous devriez voir apparaître un effet « radar ».

Vous souhaitez aller plus loin ?

Mettez en place la gestion des touches du clavier, et sur une pression des touches fléchées, faites varier la position **start** dans la direction demandée. Arrangez vous pour que la position **start** ne puisse pas aller dans un mur.