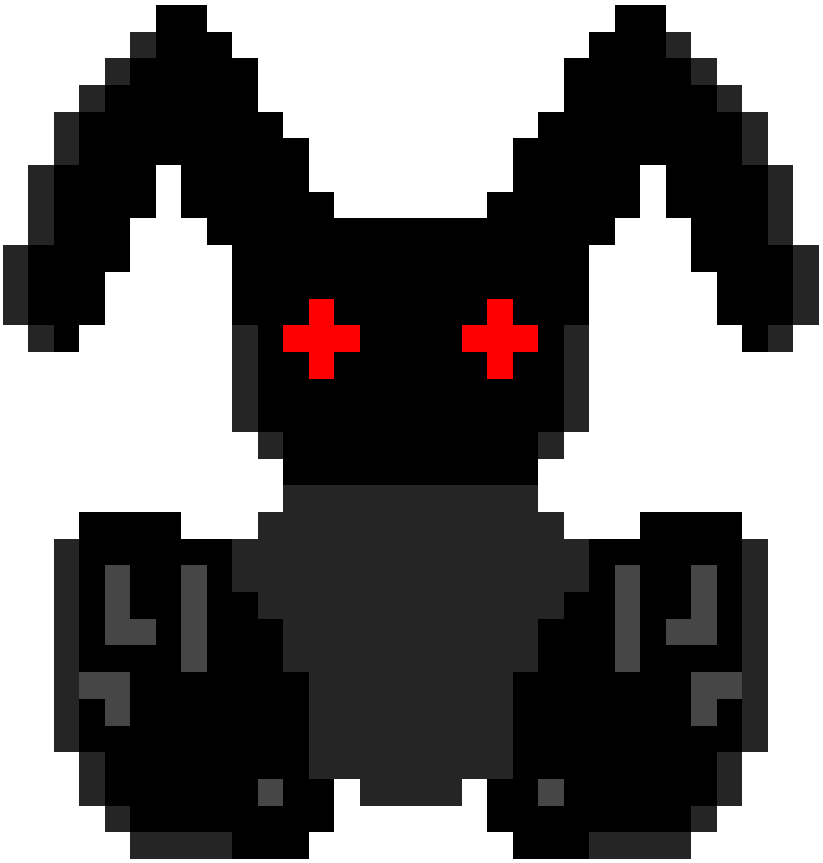


LAPINS NOIRS

MULTIMEDIA

- La Caverne aux Lapins Noirs -

Bienvenue.

Vous allez apprendre à utiliser la LibLapin pour quatre choses : dessiner à l'écran, jouer du son et réagir aux événements.

01 – 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.

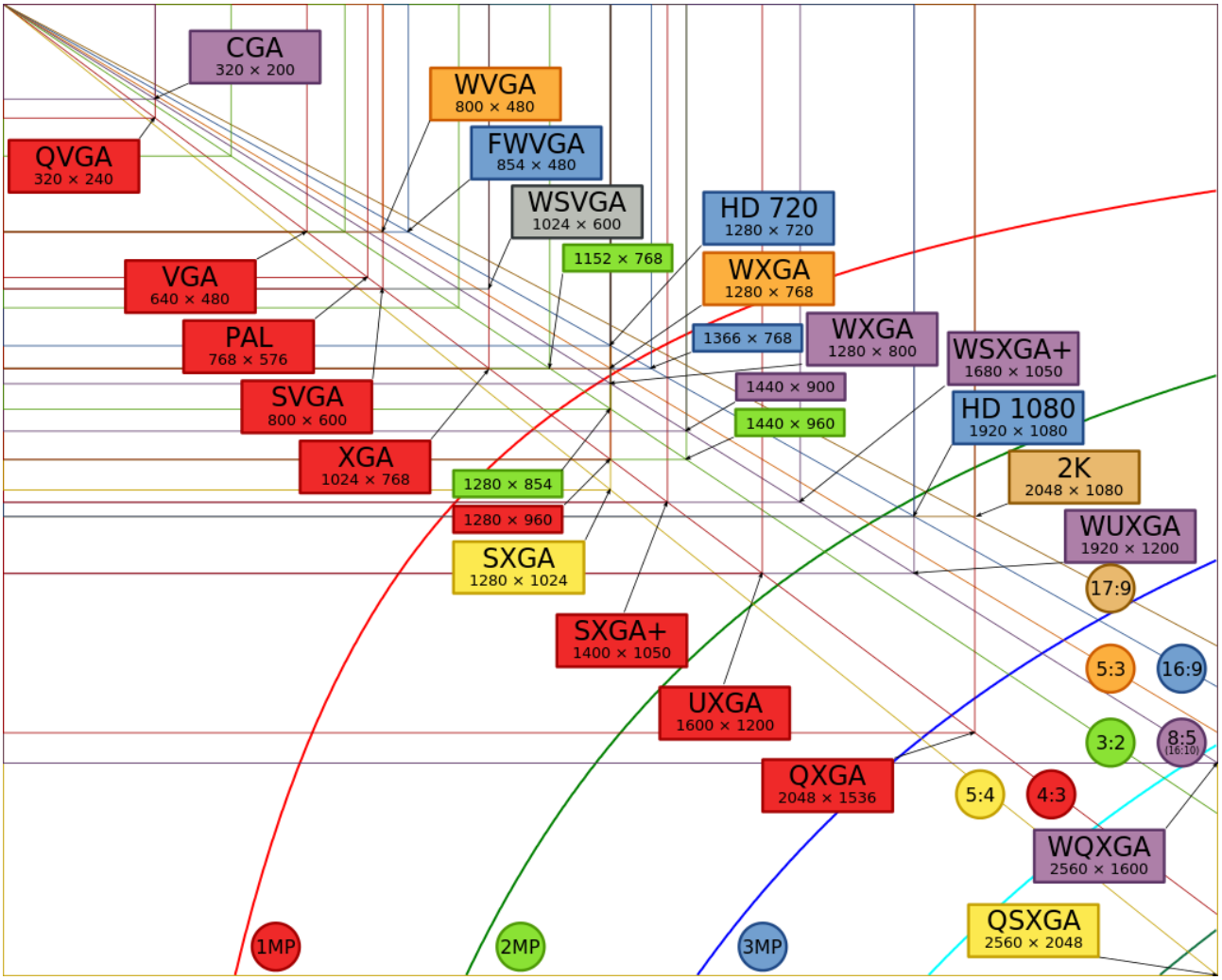
Un **écran** est une surface sur laquelle on projette une image. Cette image est rafraîchie plusieurs fois par seconde, permettant ainsi de modifier ce que l'on y voit.

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.

02 – 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**.

03 – 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.

04 – clear_pixelarray

Mettre dans le fichier : 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 ? 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** !

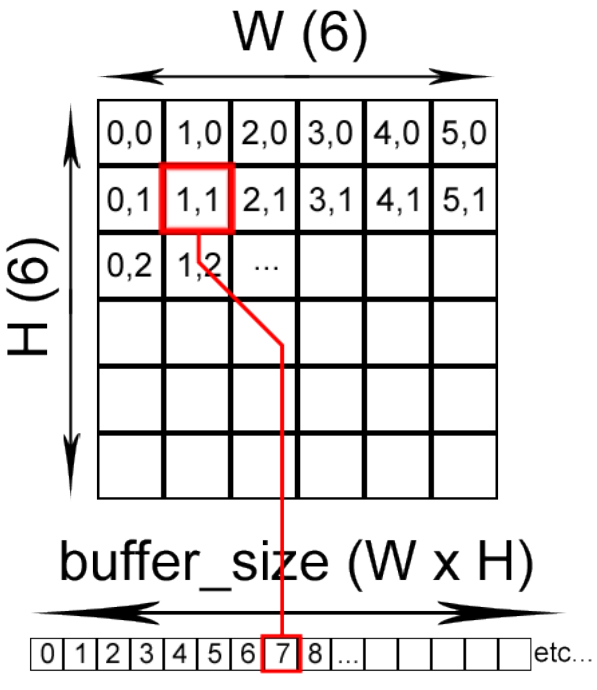
05 – set_pixel

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 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** !

06 – set_circle, set_spiral

Mettre dans le fichier : set_circle.c, set_spiral.c

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

```
void std_set_circle(t_bunny_pixelarray *picture,
                    int                x,
                    int                y,
                    int                radius
                    unsigned int       color);
```

Cette fonction affiche un cercle de rayon **radius** dont le centre est situé à (**x, y**).

Pour réaliser cette tache, vous avez le droit d'utiliser les fonctions **cos** et **sin**, si vous souhaitez dessiner le cercle à partir des règles de trigonométrie. Vous pouvez également vous en passer en dessinant simplement les points situés à une distance **radius** du centre du cercle : vous êtes libre de choisir.

Réalisez ensuite :

```
void std_set_spiral(t_bunny_pixelarray *from,
                    int                x,
                    int                y,
                    int                radius,
                    int                round);
```

Cette fonction affiche une spirale. Chaque tour provoque l'éloignement depuis le point **x, y** de **radius** pixels. Le nombre de tour est indiqué dans **round.**

07 – Événements

Mettre dans le fichier : stars.h, stars.c

Vous allez programmer un logiciel affichant des étoiles allant de la droite de l'écran vers la gauche à des vitesses différentes. Nous allons donc avoir besoin de trois fonctions : une d'initialisation, une pour dessiner les étoiles et une pour les faire avancer.

Pour commencer, définissez ce qu'est une étoile. Nous utiliserons la structure suivante :

```
typedef struct      s_bunny_star
{
    int             y;
    float           x;
    float           x_speed;
}                   t_bunny_star;
```

Cette structure définie la position (**x**, **y**) de l'étoile ainsi que sa vitesse sur l'axe x : **x_speed**. Remarquez l'utilisation de **float :** la vitesse peut en effet être inférieure à 1 sans être égale à 0. Pour représenter cette vitesse, il faut donc utiliser un nombre à virgule. De la même façon, nous utiliserons pour **x** un nombre à virgule, car cette position étant augmenté de **x_speed**, cet attribut a besoin de représenter des valeurs très petit. **En effet, un nombre entier augmenté de moins de 1 ne change pas de valeur...**

Programmez la fonction suivante :

```
void      std_init_stars(t_bunny_pixelarray  *pix,
                         t_bunny_star        *stars,
                         unsigned int        nbr_stars);
```

Cette fonction initialise toutes les étoiles à des positions aléatoires à l'aide de la fonction rand. Elle calcule également une vitesse aléatoire comprise entre 0 et -5. **stars** est un tableau d'étoile et **nbr_stars** sa longueur.

L'image **pix** est passé en paramètre de manière à ce que les valeurs valides pour les positions soient connues. Toutes les étoiles doivent être dans l'image.

Programmez la fonction suivante :

```
void std_draw_stars(t_bunny_pixelarray  *pix,
                    t_bunny_star        *stars,
                    unsigned int        nbr_stars);
```

Cette fonction dessine les étoiles dans l'image. Elle ne dessine rien d'autre !

Programmez la fonction suivante :

```
void std_move_stars(t_bunny_pixelarray  *pix,
                    t_bunny_star        *stars,
                    unsigned int        nbr_stars);
```

Cette fonction ajoute à la position en **x** de chaque étoile leur vitesse **x_speed**.
Si une étoile sort de la fenêtre, sa position en **x** est réinitialisé à la largeur de l'image - 1.

Bon, tout cela est très bien, mais comment réunir toutes ces fonctions dans un programme ? Découvrons ensemble ce qu'est un fonctionnement dit **asynchrone**.

Dans notre contexte, il s'agit d'actions ne se déroulant pas parce qu'appelée dans l'ordre dans une fonction mais du fait de l'écoulement du temps (et donc, encore une fois, non du programme). Le rafraîchissement de l'écran et la captation des événements utilisateurs (clavier, souris) donnant cependant l'impression que tout est synchrone : que tout se passe en même temps.

Comment faire ? La **LibLapin** dispose d'un ensemble de fonctions permettant de fonctionner de manière asynchrone. La fonction **bunny_loop** est la pierre angulaire de son système : elle nécessite la fenêtre, une fréquence de fonctionnement ainsi qu'un paramètre libre.

Cette fréquence de fonctionnement est le nombre d'appel moyen garanti à la fonction passée en paramètre à **bunny_set_loop_main_function**. On appellera cette fonction la **boucle principale**.

La fonction **bunny_set_display_function** sert à définir une fonction qui servira au dessin. On l'appelle **fonction d'affichage**. Cette fonction est appelée après la boucle principale et ne l'est que si celle-ci a été appelée. A quoi cette fonction sert-elle ? En cas de ralentissement, plusieurs appels à la boucle principale peuvent être faits. Dans ces cas là, un unique appel à la fonction d'affichage est réalisé afin d'économiser le plus possible de temps machine et ne pas afficher des données qui seront immédiatement écrasé.

Il existe énormément de fonctions servant à répondre à des événements utilisateurs ou machine : une touche pressée sur le clavier, un mouvement de la souris, un bouton d'une manette de jeu… une commande réseau, un déclencheur temporel, etc.

Ces fonctions ont toutes le même format de nom : bunny_set_*_response, où * est a remplacer par le type d'événement. Voici quelques exemples, **bunny_set_key_response, bunny_set_move_response et bunny_set_click_response.**

Écrivez une fonction **main** qui établira la fonction de boucle principale, la fonction d'affichage ainsi qu'une fonction de réponse au clavier qui quittera le programme si l'on appui sur la touche d'échappement. La fonction d'affiche s'occupera de noircir l'image avant d'appeler la fonction dessinant les étoiles.

Vous devriez voir un champ d'étoile défilant de droite à gauche afin une légère impression de profondeur s'afficher.

08 – Poser un pixel avec transparence

Réalisez les fonctions suivante comme étape intermédiaire :

```
double          std_ratio(int           value,
                          int           ref);
```

La fonction **std_ratio** calcule un coefficient de **value** sur **ref**. C'est à dire que si **value** vaut **ref**, la fonction renvoi 1. Si **value** vaut la moitié de **ref**, la fonction renvoi 0.5, etc.

```
unsigned char   std_color_ratio(unsigned char top,
                                unsigned char bottom,
                                unsigned char transparency);
```

La fonction **std_color_ratio** calcule le mélange entre **top** et **bottom** d'après **transparency**. Le maximum d'une composante de couleur étant 255 et le minimum 0. Si **transparency** vaut 0, alors 0 % de top est utilisé et 100 % de **bottom** est utilisé pour le calcul de la valeur finale de la composante, mélange des deux. Si **transparency** vaut 255, alors 100 % de top est utilisé et 0 % de **bottom** est utilisé. Si **transparency** vaut 128, alors les deux sont utilisés à hauteur de 50 %.

```
unsigned int    std_get_pixel(const t_bunny_pixelarray *px,
                              t_bunny_position         pos);
```

La fonction **std_get_pixel** renvoi la couleur présente à la position **pos** dans **px.**

Vous allez maintenant modifier votre fonction **std_set_pixel** de manière à gérer la composante de transparence : à la place de simplement gérer l'ajout d'une couleur à une position donnée, vous allez **d'abord calculer cette couleur d'après la transparence, la couleur à poser et la couleur qui se trouvait la avant.** N'hésitez pas à utiliser l'union **t_bunny_color**.

Voici un programme de test :

```
void            sweet_noise(t_bunny_pixelarray      *px)
{
        t_bunny_position pos;
        t_bunny_color clr;
        int i;

        clr.full = BLACK;
        clr.argb[ALPHA_CMP] = 64;
        // Votre clear_pixelarray DOIT utiliser le set_pixel
        // qui gère la transparence
        std_clear_pixelarray(px, clr.full);
        i = 0;
        while (i < 200)
        {
                pos.x = rand() % px->clipable.buffer.width;
                pos.y = rand() % px->clipable.buffer.height;
                std_set_pixel(px, pos, WHITE);
                i = i + 1;
        }
}
```

09 – Lancer de rayons

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.*

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.


```
#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);
}
```

Vous allez maintenant vous échauffer avec un programme plutôt sympathique. Vous allez réaliser un programme qui :

- Rempli de noir votre écran. 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 ».

Si vous êtes parvenu jusqu'ici en ayant réussi tous les exercices,
vous êtes parés pour le mini-projet RUNNER!

10 – set_line

Mettre dans le fichier : set_circle.c, set_spiral.c

Programmez la fonction suivante :

```
void std_set_line(t_bunny_pixelarray    *picture,
                  t_bunny_position      *position,
                  unsigned int          *color);
```

Cette fonction trace une ligne depuis **position[0]** jusqu'à **position[1]**. La couleur de cette ligne sera **color[0].** Un couleur se situera également dans **color[1]** mais nous allons l'ignorer pour l'instant. Le dessin a lieu dans **picture**.

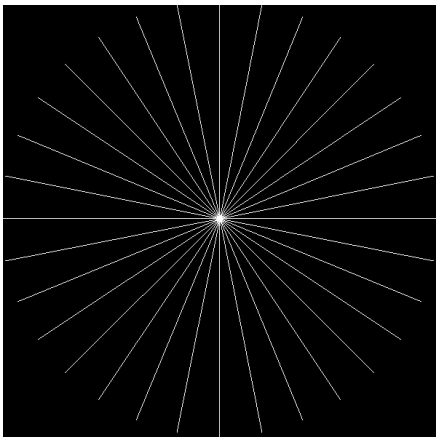
Comment faire ?

Vous allez partir de **position[0].x** jusqu'à **position[1].x**, ajoutant à **position[0].y** le coefficient directeur pour chaque pixel.

Ci-dessous, un morceau de programme de test afin de vous aider.

```
void                            linedisc(t_bunny_pixelarray         *px)
{
        unsigned int            color[2];
        t_bunny_position        pos[2];
        double                  i;

        i = 0;
        color[0] = WHITE;
        color[1] = WHITE;
        pos[0].x = px->clipable.buffer.width / 2;
        pos[0].y = px->clipable.buffer.height / 2;
        while (i < 2 * M_PI)
        {
                pos[1].x = px->clipable.buffer.width * (0.5 + cos(i) / 2);
                pos[1].y = px->clipable.buffer.height * (0.5 + sin(i) / 2);
                std_set_line(px, &pos[0], &color[0]);
                i += M_PI / 16;
        }
}
```



11 – Tourbillon

Mettre dans le fichier : whirlpool.c

Écrivez la fonction suivante :

```
void std_whirlpool(t_bunny_pixelarray      *picture,
                   t_bunny_position        *pos,
                   double                  prog,
                   int                     depth);
```

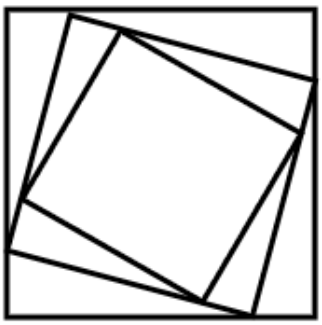
La fonction **etd_whirlpool** trace dans **picture** un quadrilatère blanc dans constitué des lignes suivantes:

- **pos**[0] à **pos**[1]
- **pos**[1] à **pos**[2]
- **pos**[2] à **pos**[3]
- **pos**[3] à **pos**[0]

Ensuite, si **depth** n'est pas nul, alors la fonction recommence le dessin d'un quadrilatère, mais cette fois dans le premier, avec ses coordonnées décalés de **prog**. Ce décalage est une valeur situé entre 0 et 1 et s'effectue en considérant les coordonnées de chaque ligne comme étant des rails sur lesquels les coordonnées du prochain quadrilatère vont se situer.

Le paramètre **depth** indique le niveau de récursion maximal. Si **depth** vaut 0, alors rien n'est dessiné. Le paramètre

Le rendu devrait être de ce type là, avec **depth** valant 3, **prog** valant 0,2 et les coordonnées originales du quadrilatère étant celle de l'image :



12 – set_triangle, set_polygon

Mettre dans le fichier : set_triangle.c, set_polygon.c

Programmez la fonction suivante :

```
void std_set_triangle(t_bunny_pixelarray          *pix,
                      t_bunny_position            *pos,
                      unsigned int                *col);
```

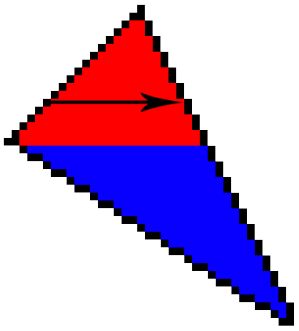
Cette fonction dessine à l'intérieur de **pix** un triangle dont les coins sont aux positions **pos[0]**, **pos[1]** et **pos[2]**. Le triangle sera coloré progressivement, en fonction de la proximité de ces angles des couleurs **col[0]**, **col[1]** et **col[2]**.

Comment faire ? Les points d'un triangle peuvent être trié : il y a un point plus haut que les deux autres, un plus bas que les deux autres et le dernier entre les deux autres.

Considérant cet état de fait, alors il est possible de découper n'importe quel triangle en deux triangles dont l'un des cotés est horizontal.

Dans l'illustration ci-dessous, la partie en rouge est le triangle supérieur et celle en bleu le triangle inférieur. Le point milieu étant la position de la ligne horizontale.

Une fois cette situation obtenue, il suffit pour chaque position de la ligne allant du point milieu au point haut ou au point bas, de rejoindre l'autre bord en bouclant horizontalement.



```
void std_set_polygon(t_bunny_pixelarray       *dest,
                     t_bunny_pixelarray       *tex,
                     t_bunny_position         *pos,
                     t_bunny_position         *tpos);
```

Le triangle sera tracé dans **dest** aux positions décrites dans **pos[0]**, **pos[1]**, **pos[2]**. Les couleurs de ce triangles seront tirées de **tex**, utilisé comme texture, le clip utilisé étant également un triangle décrit par **tpos[0]**, **tpos[1]** et **tpos[2]**.

Il n'est pas demandé d'effet de lissage ni quoi que ce soit, simplement un placage de texture. Les points **pos** comme **tpos** peuvent être dans n'importe quel sens.

13 – Perspective

Mettre dans le fichier : perspective.c

Programmez la fonction suivante, qui projete des coordonnées 3D en coordonnées 2D. Vous placerez le repère au centre, et plus Z est grand, plus l'objet sera loin.

```
t_bunny_position    std_perspective(float    x,
                                    float    y,
                                    float    z);
```

Pour savoir comment projeter en perspective, inspirez vous de la **réalité**. Regardez cette photo, et utilisez la manière dont la profondeur influence la position sur la photo elle-même pour en tirer votre équation mathématique. **Graphez** vos valeurs, la fonction à employer sera évidente si vous la connaissez, sinon vous rechercherez parmi les classiques du lycée.

Vous devrez maintenant adapter vos fonctions de dessin de polygone à l'existence d'une coordonnée Z. Utilisez le type t_bunny_accurate_zposition !

14 – ZBuffer

Mettre dans le fichier : zbuffer.h, zbuffer.c

Ajoutez à votre bibliothèque un type permettant de représenter une position spatiale en 3D avec une bonne précision (flottant ou virgule fixe). Votre type s'appellera **t_bunny_zposition**.

Ajoutez à votre bibliothèque un type contenant un **t_bunny_pixelarray** ainsi qu'un tableau de nombres à haute précision (flottant ou virgule fixe) disposant d'autant de cases qu'il y a de pixel dans le **t_bunny_pixelarray**. Enfin, fournissez à votre bibliothèque des utilitaires de construction et de destruction de votre type. Votre type s'appellera **t_bunny_zbuffer**.

Réalisez la fonction suivante :

```
void    std_set_zpixel(t_bunny_zpixelarray      *piz,
                       t_bunny_zposition        *pos,
                       unsigned int             *col);
```

Cette fonction dessine dans piz le pixel à la position **pos**, ***si et seulement si*** **pos.z** est plus près de l'œil que le pixel qui s'y trouvait déjà. La transparence n'est pas obligatoire, mais c'est évidemment un plus.

Programmez en conséquence les fonctions suivantes : **std_clear_zpixelarray**, **std_set_zline**, **std_set_zpolygon**.

15 – Rotation

Vous allez écrire la fonction suivante :

```
t_bunny_zposition      std_rotation(t_bunny_zposition      target,
                                    t_bunny_zposition      rotation);
```

La structure **t_bunny_zposition** contient trois float, x, y et z.

La fonction **rotate** effectue une rotation d'axe **rotation.x**, **rotation.y**, **rotation.z** des coordonnées situés dans **target** avant de les renvoyer.

Ci-dessous, les matrices de rotation.

$$R_{\mathbf{x}}(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}, \quad R_{\mathbf{y}}(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}, \quad R_{\mathbf{z}}(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Chaque ligne de la matrice représente une somme devant être assignée à) une dimension X, Y ou Z de l'élément à modifier.

Les fonctions **cos** et **sin** sont autorisées, bien entendu.

Si vous êtes parvenu jusqu'ici en ayant réussi tous les exercices,
vous êtes parés pour le projet StarGlider !

16 – Accéder à la mémoire audio

Pour créer un espace dans la carte son, utilisez la fonction **bunny_new_effect** qui prend en paramètre une durée du son en seconde. Cette durée est un flottant, cela signifie qu'il est possible demander un nombre de seconde non rond.

La mémoire renvoyée, de type **t_bunny_effect** contient de nombreux champs dont le plus intéressant est **sample.** Ce champ est similaire au champ **pixels** du **t_bunny_pixelarray** dans le sens où c'est **la donnée qui sera exploitée**. La longueur de l'espace mémoire pointée par **sample** est **duration** multiplié par **sample_per_second**.

Si **duration** n'a certainement aucun secret pour vous : c'est le paramètre passé à **bunny_new_effect**. Le champ **sample_per_second** est par contre probablement inconnu : il s'agit de la fréquence d'échantillonnage, c'est à dire le nombre de niveau que l'onde sonore peut prendre en une seconde. Plus cette valeur est élevée, plus le son est de bonne qualité et des nuances peuvent être entendues.

Par défaut, la fréquence d'échantillonnage est 44100Hz, c'est à dire qu'il faut écrire dans 44100 cases de **sample** pour faire un son d'une seule seconde… Chaque niveau enregistrée l'est dans un entier de 16 bits, c'est à dire un entier dont les valeurs possibles sont comprises entre -32768 et 32767. Plus l'amplitude – la valeur absolue des valeurs formant l'onde sonore – est grande, plus le volume est important.

Vous allez commencer simplement en réalisant une **friture**, de la même manière que celle que vous deviez réaliser **graphiquement** : remplissez **sample** de valeurs aléatoires à l'aide de la fonction rand. N'hésitez pas à tenter d'exploiter toute l'amplitude disponible, entre -32768 et 32767, mais *faites attention à mettre le volume à un niveau raisonnable avant de tester*.

```
void std_set_noise(t_bunny_effect       *fx);
```

Pour jouer le son, une fois que vous l'avez écrit, vous devrez d'abord appeler **bunny_compute_effect** qui permet de faire passer le son que vous avez écrit à la carte son. Ensuite, vous pourrez appeler **bunny_sound_play.** Vous remarquerez que **bunny_sound_play** prend un pointeur sur **t_bunny_sound** et non un pointeur sur **t_bunny_effect…** mais ce n'est pas grave, car il y a un **t_bunny_sound** dans **t_bunny_effect** ! Envoyez simplement son adresse et préparez vous à danser avec modération.

17 – Générer une onde simple

Programmez la fonction suivante :

```
void std_set_wave(t_bunny_effect        *fx,
                  int                   start,
                  int                   stop,
                  int                   frequency);
```

Cette fonction remplit depuis la case **start** jusqu'à la case **stop** la mémoire son de **fx** d'une onde sonore allant à la fréquence **frequency**.

Qu'est ce que cette fréquence signifie ? La fréquence d'un son détermine sa note : plus la fréquence est élevée, plus la note sera aiguë. Une fréquence de 100 indique que en **une seconde, il y a 100 ondulations**. L'acuité humaine est généralement comprise entre 20Hz et 20 000Hz, en dessous et au-delà, il est rare de parvenir à entendre le son.

Pour réaliser une ondulation, vous pouvez utiliser la fonction **cos**, il vous suffira d'aller de 0 à 2 Pi sur le temps d'une unique ondulation pour réaliser celle-ci. L'onde sonore sera une sinusoïde et très douce.

Vous pouvez aussi, plus simplement, mais le son sera moins agréable, simplement remplir la première moitié du temps de l'ondulation par la valeur -25 000, par exemple, et la seconde par 25 000. L'onde sonore sera du carré et plutôt sèche.

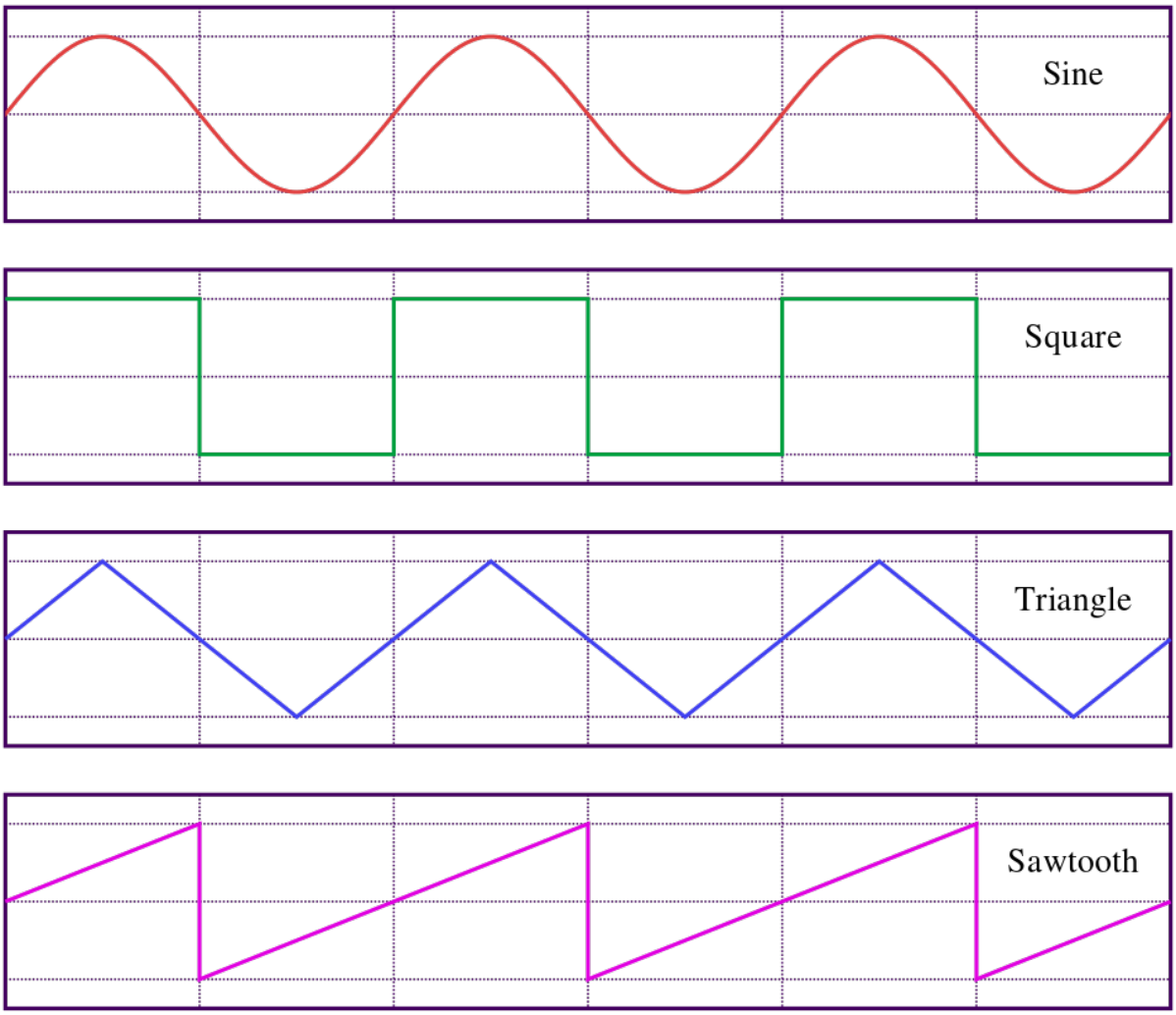
Voici quelques fréquences que vous pouvez essayer :

Do Ré Mi Fa Sol La Si Do : 262, 294, 330, 349, 392, 440, 494, 524

Vous remarquerez que le do plus aigu a le double de fréquence du do grave : pour passer d'un octave à l'autre, il suffit de multiplier par 2 la fréquence d'une note.

18 – Des ondes diverses

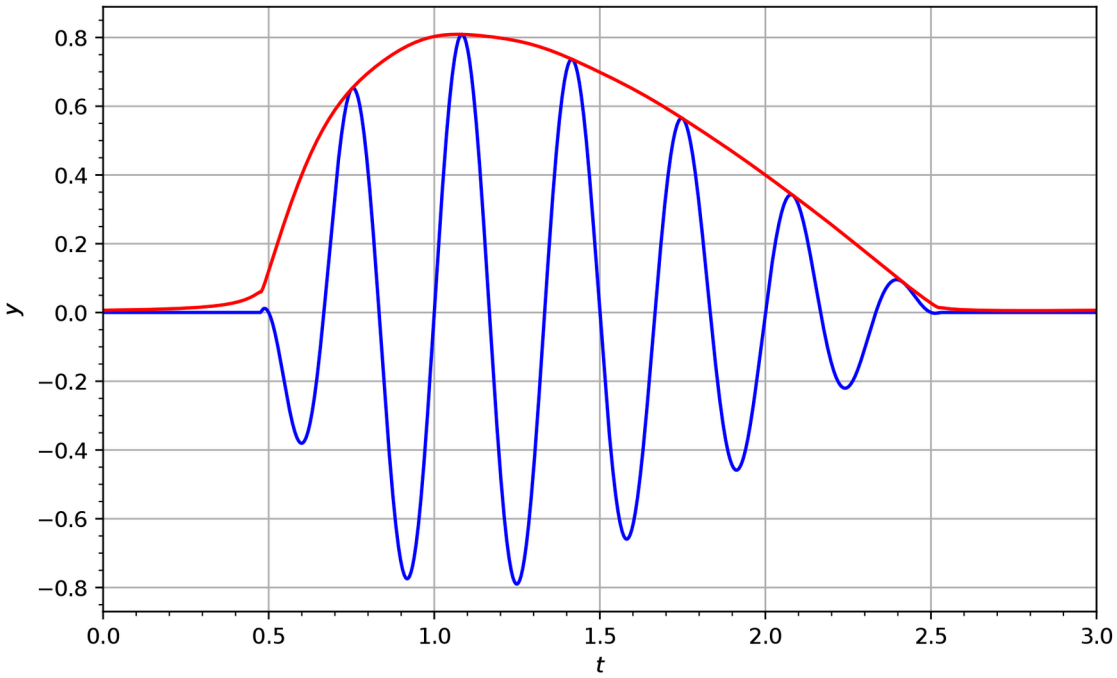
Il est possible de générer des sons très différents simplement en définissant des formes d'ondes diverses. Si vous êtes d'attaque, écrivez une fonction pour chaque ! Voici quelques ondes habituelles :



La sinusoïde est celle générée par **cos** ou **sin.** L'onde carrée est celle qui est générée en établissant deux valeurs stables sur différentes parties de l'onde. L'onde triangulaire définie deux phases : une montante linéaire et une descendante linéaire. L'onde dent de scie définie une unique phase montante faisant toute l'ondulation – une onde seulement descendante est bien sur possible aussi. Il est possible d'effectuer d'autres transformations, nous les verrons dans les pages qui viennent.

19 – L'enveloppe ?

On appelle enveloppe le niveau de volume dans lequel on glisse une onde. Voici un exemple simple : Une onde sinusoïdale dans une enveloppe à forme exotique. *La ligne rouge n'est présente que pour mettre en relief le fait que l'amplitude de l'onde varie en fonction de cette enveloppe.*



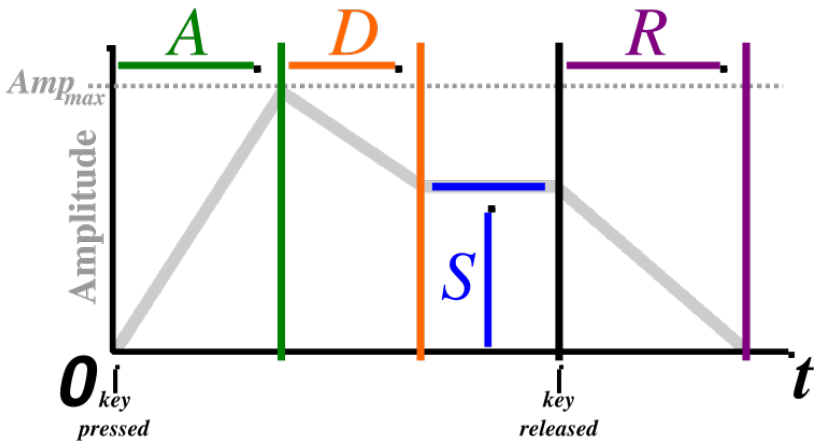
Réalisez la fonction suivante :

```
void std_apply_envelope(t_bunny_effect      *fx,
                        int                 start,
                        int                 stop);
```

Cette fonction applique une enveloppe s'étendant de **start** à **stop**, cette enveloppe aura comme forme une unique demi-période de **sinusoïde** (C'est à dire juste la partie positive). Le volume du son va donc monter doucement puis redescendre doucement. L'onde présente au départ dans le son n'a aucune importance : cela fonctionne avec n'importe laquelle.

20 – ADSR

L'ADSR, ou *Attack, decay, sustain, release*, est une enveloppe standard paramétrable dont l'apparence est la suivante :



Les événements « key_pressed » et « key_released » étant **start** et **stop** dans le cadre de la fonction précédente.

La partie « attack » étant l'évolution du son juste après le lancement du son. Cette évolution s'arrête et on passe au « decay » qui fait évoluer le son jusqu'à un niveau stable, le « sustain » qui est le volume qu'on garde ensuite tant que le son n'est pas arrêté.

Cela signifie que « attack » et « decay » ne sont pas lié à la durée du son mais au simple départ du son. « Sustain » demeure tant que le son n'est pas arrêté. La partie « Release » commence dès que le son est arrêté.

L'enveloppe ADSR est faite pour fonctionner appliquée à une unique note, et non pas à une piste de son complexe.

```
void std_apply_adsr(t_bunny_effect      *fx,
                    int                 start,
                    int                 stop,
                    int                 attack,
                    int                 decay,
                    float               sustain,
                    int                 release);
```

Les paramètres **attack** et **decay** sont des durées. La partie **sustain** est un volume. La partie **release** est également une durée.

21 – Fondu

Écrivez la fonction suivante :

```
void std_sound_fade(t_bunny_effect      *fx,
                    int                 start,
                    int                 stop,
                    bool                out);
```

La fonction **sound_fade** effectue un fondu entrant ou sortant en fonction de si **out** est faux ou vrai. Un fondu est effectué par la multiplication des valeurs d'un son sur la période **start stop** par un coefficient variant de 0 vers 1 ou 1 vers 0.

22 – Élasticité

Écrivez la fonction suivante :

```
void std_sound_stretch(t_bunny_effect       *dest,
                       int                  dstart,
                       int                  dstop
                       t_bunny_effect       *src,
                       int                  sstart,
                       int                  sstop);
```

La fonction **sound_stretch** transfert depuis la période **sstart-sstop** dans **src** le son dans **dest** à la période **dstart-dstop**.

Sans opération spécifique, la hauteur du son va varier (il y aura un pitch).

23 – Sampler

Utilisez **bunny_load_effect** et **bunny_new_effect** pour respectivement charger des sons et créer une musique. Les sons à charger devraient contenir tous la même note, pourquoi pas un unique LA international à 220Hz ?

Programmez les fonctions suivantes :

```
double std_get_note_frequency(const char     *note,
                              int            octave);

void std_write_sample(t_bunny_effect         *dest,
                      t_bunny_effect         *src,
                      int                    start,
                      int                    stop,
                      double                 freq,
                      bool                   loop);
```

La fonction **get_note_frequency** renvoi la fréquence de la note indiqué dans **note** à l'octave **octave.** Vous trouverez les informations concernant les fréquences sur Wikipedia :

https://fr.wikipedia.org/wiki/Note_de_musique#Fr%C3%A9quence_d'une_note

Seul les octaves 1, 2 et 3 sont demandés, mais vous pouvez également fournir tous les autres de -1 à 9. Pour cela, renseignez vous sur la logique présente derrière ces fréquences, remarquez que le LA évolue de manière intéressante d'un octave à l'autre : 55, 110, 220, 440, 880, etc.

https://www.youtube.com/watch?v=cTYvCpLRwao

La fonction **write_sample** écrit dans **dest** l'onde présente dans **src** de **start** à **stop** à la fréquence **freq** sachant que le son dans **src** est à **220Hz**. Si le son est plus court que la distance entre **start** et **stop**, il n'est joué qu'une fois, sauf si **loop** est vrai, dans ce cas, on boucle.

La fonction **write_sample** *ajoute* au son existant et n'écrase **pas**.

Pour rappel :

Qu'est ce que cette fréquence signifie ? La fréquence d'un son détermine sa note : plus la fréquence est élevée, plus la note sera aiguë. Une fréquence de 100 indique que en **une seconde, il y a 100 ondulations**. L'acuité humaine est généralement comprise entre 20Hz et 20 000Hz, en dessous et au-delà, il est rare de parvenir à entendre le son.

Une enveloppe peut-être assignée à n'importe quel son. Ajoutez une enveloppe simple, comme sin, à votre son lorsqu'il est ajouté.