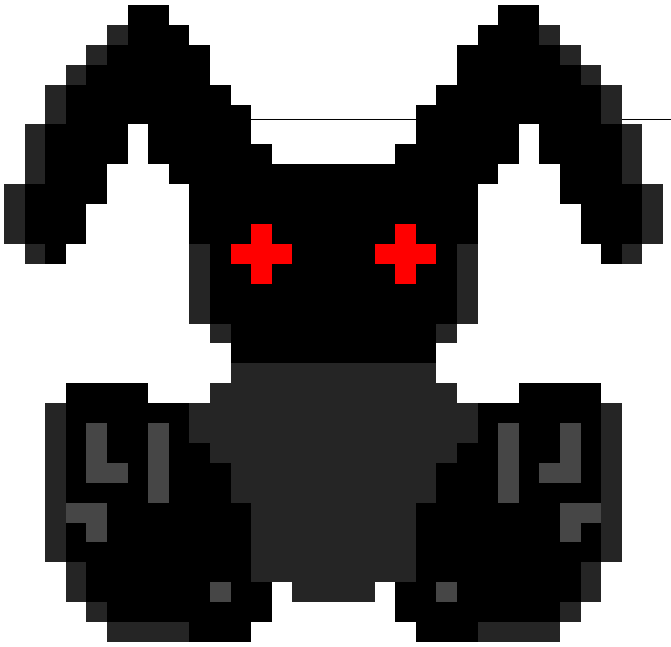


LAPINS NOIRS

- La Caverne Aux Lapins Noirs -

SURFACE

Ce mini-projet consiste à réaliser un logiciel d'affichage de surface en isométrique ou parallèle.

- La Caverne aux Lapins Noirs -

Auteur : Jason Brillante

INDEX

01 – Avant-propos

Votre travail doit être rendu via le dossier **~/projets/surface/** 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 – Pré-requis

Programmez la fonction suivante, si vous ne l'avez pas déjà faite, comme pré-requis au projet :

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

Cette fonction dessine un pixel à la position **pos** dans **px** et de couleur **color**. Vous pouvez, si vous savez le faire, gérer la transparence.

Vous pouvez, de plus également réaliser celle-ci :

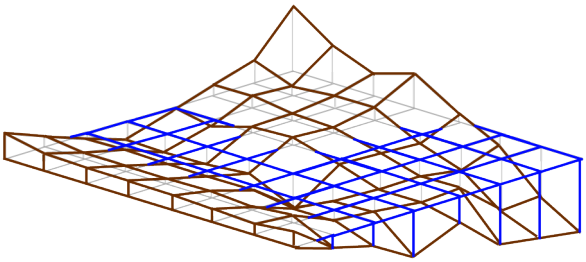
```
void std_set_line(t_bunny_pixelarray    *px,
                  t_bunny_position      *pos,
                  unsigned int          *color)
```

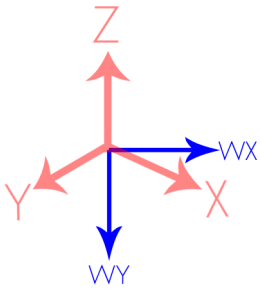
Cette fonction trace une ligne depuis la position **pos[0]** jusqu'à la position **pos[1]** dans **px.** La couleur de la ligne sera soit **color[0]** - soit un dégradé allant de **color[0]** jusqu'à **color[1]** si vous savez le faire. De même, la gestion de la transparence est un plus.

La fonction de dessin de ligne n'est pas requise pour commencer le projet : vous pouvez tout à fait commencer en plaçant de simples points. Évidemment, le rendu final nécessite une fonction de tracé de ligne pour débloquer toutes les médailles accessibles.

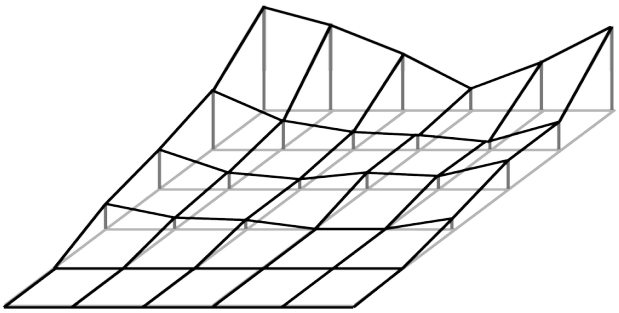
05 – Isométrique ou parallèle ?

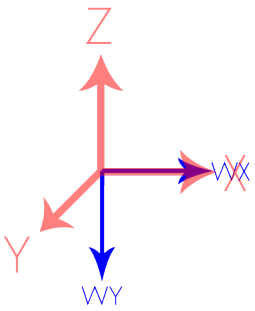
Isométrique





Parallèle

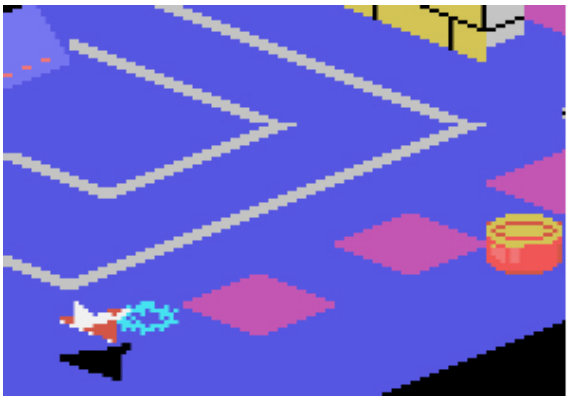


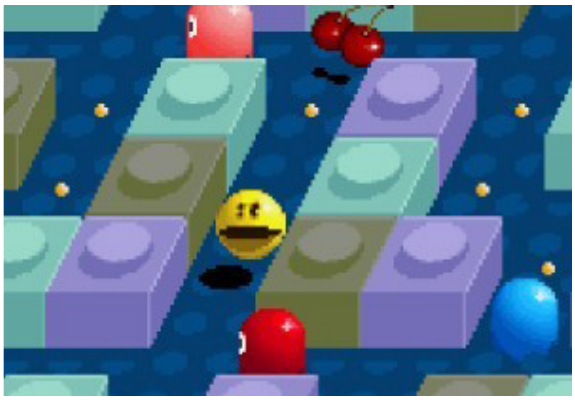


Le projet Surface consiste à réaliser des objets en 3D soit en vue isométrique, soit en vue parallèle à partir d'un fichier contenant une description. Votre programme doit prendre comme unique paramètre un fichier de configuration et afficher son contenu à l'écran. Le choix de la projection isométrique ou parallèle vous appartient.

Les repères bleus que vous pouvez voir ci-dessus représentent le repère de l'écran. Les repères roses sont les repères soit **isométrique** pour celui de gauche, soit **parallèle** pour celui de droite. Pour réussir cet exercice, vous devez parvenir à exprimer WX et WY en fonction de X, Y et Z. Il s'agit pour vous de transformer 3 coordonnées en 2 coordonnées. Cela n'est pas sans rappeler la transformation de 2 coordonnées en 1 unique coordonnée pour la pose de pixel…

Posez vous la question, par exemple, pour X : lorsque l'on va vers X en isométrique par exemple, de combien se déplace-t-on sur le repère WX/WY ?





06 – Surface

Dans un premier temps, vous allez programmer de quoi afficher une surface. Dans le fichier de configuration que vous allez chargez, une surface est décrite de la façon suivante :

```
$> cat surface.dab
{Objects
     [
          Type = "Surface"
          Width = 10
          Height = 10
          {Data
               0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
               0, 1, 1, 1, 1, 1, 1, 1, 1, 0,
               0, 1, 2, 2, 2, 2, 2, 2, 1, 0,
               0, 1, 2, 3, 3, 3, 3, 2, 1, 0,
               0, 1, 2, 3, 4, 4, 3, 2, 1, 0,
               0, 1, 2, 3, 4, 4, 3, 2, 1, 0,
               0, 1, 2, 3, 3, 3, 3, 2, 1, 0,
               0, 1, 2, 2, 2, 2, 2, 2, 1, 0,
               0, 1, 1, 1, 1, 1, 1, 1, 1, 0,
               0, 0, 0, 0, 0, 0, 0, 0, 0, 0
          }
     ]
]
```

Le nœud **Objects** est un tableau. Chaque case du tableau **Objects** a forcément un champ **Type** qui annonce ce que contient la case. Dans le cas présent, l'objet est une surface de taille 10 sur 10. Chaque nombre du tableau **Data** représente un sommet d'une surface à afficher à l'écran et est caractérisé par sa position dans le tableau **(X, Y)** et la valeur contenu, qui joue le rôle de **Z**. Ces **X, Y et Z** doivent être exploiter par une fonction de **projection 3D** que vous allez écrire.

Cette projection sera soit **isométrique**, soit **parallèle**, comme vous le souhaitez. Vous pouvez tout à fait ajouter un champ « Projection » au fichier afin de le déterminer à chaud plutôt qu'en dur dans votre programme.

Un tableau **Colors** peut également être présent, afin de donner à chaque valeur dans **Data** une couleur.

07 – Solides de révolution

Cette partie est un bonus. Vous allez maintenant réaliser vos premiers solides de révolution. Dans un fichier DABSIC, peuvent également être présent l'objet suivant :

```
$> cat sphere.dab
{Objects
     [
          Type = "Sphere"
          X = 10
          Y = 10
          Z = 10
          Radius = 50
     ]
]
```

L'objet **Sphere** a quatre propriétés : une position **X, Y, Z** servant à placer le centre de la sphère et une valeur **Radius** déterminant le rayon de la sphère. Ci-dessous, l'équation paramétrique de la sphère :

$$\begin{cases} x &= r\cos\theta\ \cos\phi \\ y &= r\cos\theta\ \sin\phi \\ z &= r\sin\theta \end{cases} \qquad \left(\frac{-\pi}{2} \leq \theta \leq \frac{\pi}{2} \text{ et } -\pi \leq \phi \leq \pi\right)$$

Comment lire cette équation ? Que les points constituant la sphère sont calculés de la façon suivante :

> Pour chaque valeur de **Théta** allant de -Pi/2 à Pi/2 (donc une boucle)
> Et
> Pour chaque valeur de **Phi** allant de -Pi à Pi (donc une autre boucle, imbriquée)
>
> **X** vaut **Radius** * Cosinus (**Theta**) * Cosinus (**Phi**)
> **Y** vaut **Radius** * Cosinus (**Theta**) * Sinus (**Phi**)
> **Z** vaut **Radius** * Sinus (**Theta**)

Réalisez également les objets **Cone, Cylinder, Tore et Moebius.** Vous trouverez leurs équations sur **Wikipedia**.