TD-TP Allegro 3 – Acteurs multiples, fonds, sprites, animations

OBJECTIFS A ATTEINDRE :

1. **Modéliser des "acteurs" en définissant une structure** regroupant les caractéristiques individuelles
2. **Gérer simultanément un ensemble d'acteurs** : soit en nombre fixe soit en nombre variable
3. **Faire une animation se déroulant sur une image de fond**, éventuellement ajouter **un avant plan**
4. Savoir créer et charger des images avec **couleur spéciale transparence** (rouge 255 vert 0 bleu 255)
5. Utiliser ces images avec transparence avec les **fonctions draw_sprite** ou **masked_blit**
6. **Dérouler une séquence d'animation** en enchaînant des images, synchroniser avec un déplacement

Remarques :

- **Consultez les exemples de programmes** montrés en cours, disponibles sur le site.
- Pour **améliorer votre productivité** et faire plus rapidement le tour des concepts et techniques à connaître, **utilisez ces exemples de code comme des bases de travail** : **repérez les parties pertinentes** qui correspondent à vos objectifs, **copiez/collez et modifiez selon vos besoins**.
- Même si l'objectif final est de comprendre toutes les lignes de code des programmes, vous pouvez dans un 1er temps utiliser tels quels les aspects les plus techniques ou complexes, quitte à revenir dessus plus tard pour les approfondir, les réécrire à votre manière...

Les catégories d'exercices à mettre en place sont :

GESTION DE MULTIPLES « ACTEURS »

On appellera **''acteur''** un élément d'un jeu ou d'une animation qui doit réapparaître à chaque actualisation de l'affichage en évoluant dans son apparence et/ou sa position à chaque fois. Cette évolution dépend de règles fixant son comportement plus ou moins complexe. Un exemple (très simple) d'acteur est un carré qui rebondit sur les bords de l'écran.

Le but des exercices suivants est d'intégrer dans un même programme la gestion de multiples acteurs. Un ordinateur ne peut exécuter qu'une seule instruction à la fois, il ne peut donc gérer **simultanément** les acteurs, il faudra donc les gérer les uns après les autres en répétant : gérer un peu acteur 0, gérer un peu acteur 1, gérer un peu acteur 2, ... gérer un peu acteur n-1

On aura généralement la chronologie suivante dans la **boucle d'animation** (ou boucle de jeu) :

Tant que (condition pour continuer)
    Effacer buffer (ou blit du fond sur buffer si image de fond)
    bouger un peu acteur 0 bouger un peu acteur 1 bouger un peu acteur 2 ... bouger un peu acteur n-1
    afficher acteur 0 afficher acteur 1 afficher acteur 2 ... afficher acteur n-1 (sur le buffer) Actualiser écran (blit du buffer sur l'écran) + petite pause pour temporiser (vitesse globale)

Enfin quand on parle d'acteurs multiples il s'agit de n acteurs avec n possiblement grand : **il est donc exclu de gérer n acteurs en copiant/collant n fois du code (il faut des boucles)**. Selon l'objectif du programme, on souhaite pouvoir changer n ,sinon en cours de jeu, au moins facilement en éditant une constante NACTEUR et recompilant.

Exercice 1 : Casse-pipes *(Voir exemples 1_0 1_1 2_0 du cours 3)*

***Même exercice que l'exercice 5 du TP 2** de la semaine dernière. Si vous pensez avoir déjà réalisé cet exercice de manière satisfaisante, faites valider par votre chargé de TP et passez à la suite.*

Des objets, apparaissant un par un, circulent dans l'écran et vous ne pouvez plus travailler. Heureusement le curseur de la souris se transforme en viseur et vous pouvez vous en débarrasser en positionnant le viseur sur la cible et en cliquant.

Faire le programme, étape par étape :

- Utilisez le **double buffer** avec une BITMAP *page de la même taille que l'écran : les envahisseurs se déplacent sur cette page. Chaque envahisseur aura une couleur unique bien distincte.
- Une fois que les objets gênants (ex : disque de couleur unique chacun) se déplacent, la souris tire dessus. Il y a deux solutions pour retrouver l'objet concerné
    - Par la position de la souris au moment du clic et celle de l'objet
    - Par la couleur du pixel à l'endroit du clic avec un getpixel() sur le buffer permet de récupérer la couleur sous le viseur (dans la séquence de la boucle de jeu, ceci doit avoir lieu avant l'affichage

du viseur !) Cette couleur permet de savoir quel envahisseur est touché par la le tir de la souris (chaque objet est identifié par une couleur unique).

Quand un objet est touché, il réagit en fonction de ce que vous souhaitez :

- il explose
- il se met en colère
- il part de l'écran
- il disparaît
- il se métamorphose en autre chose
- etc.

- Pour avoir un **curseur de souris spécial** :
  - Faites install_mouse() mais pas show_mouse() : le curseur standard reste invisible.
  - A la place du pointeur du souris mettez un cible que vous aurez dessiner vous-même et que vous afficherez directement sur le buffer (page) aux positions mouse_x mouse_y avant chaque appel à blit(page, screen …).

Indications complémentaires :

Ne partez sur l'idée d'un nombre variable d'acteurs que si vous vous sentez à l'aise car dans un premier temps il est plus simple de gérer un nombre fixe d'acteurs : exemples 1_0 et 1_1 du cours 3.
Pour donner l'illusion qu'un acteur disparaisse et un autre apparaisse, deux solutions sont proposées :

- un **acteur est détruit et un autre apparait en réinitialisant les données d'une structure** (positions/taille/forme/couleur) : au moment ou un acteur disparaît, un autre apparaît. En fait c'est le même acteur, mais qui a changé de costume et qui s'est téléporté instantanément !
- Ajouter un **champ "actif"** (simulation d'un booléen) à la structure de l'acteur : **quand l'acteur est créé le champ est à 1 quand il disparait ou meurt, le champ est à 0**: les acteurs avec **"actif" à 0 ne sont plus affichés**, **ni gérés** (tests au moment de l'affichage et au moment des traitements). Cette approche est comparable à l'utilisation d'un pointeur NULL pour signaler une "case" disponible (exemple 2_0) en un peu moins sophistiqué.

Chaque acteur est référencé par un indice dans le tableau de pointeurs sur structures acteurs. Il peut être intéressant d'écrire une **fonction *trouveActeur()*** qui prend en paramètre une couleur (ce qu'on obtient à partir d'un getpixel) et qui retourne l'indice de l'acteur qui a cette couleur si un des acteurs a effectivement cette couleur, ou -1 dans le cas où aucun acteur ne correspond.

Cette fonction *trouveActeur()* peut être utilisée pour savoir quel acteur est touché lors d'un tir à la souris :

```
if (condition pour tirer){
      cible = trouveActeur( getpixel(page, mouse_x, mouse_y) );
      if (cible != -1)
            //Il arrive quelque chose à  mesActeurs[cible]
}
```

Pour déterminer une couleur unique pour chaque acteur le plus simple est de tirer aléatoirement sa couleur. Pour être sûr qu'elle est effectivement unique on peut aussi utiliser *trouveActeur()* :

```
do {
      // couleur au hasard
      couleur = makecol(du hasard,du hasard,du hasard);

      //si déjà utilisée, retirer l'acteur dont la couleur est connue
      //et déjà utilisée par un autre acteur
      while ( trouveActeur(couleur)!=-1 );
```

Une carabine ne peut pas tirer en "continu" à chaque passage dans la boucle de jeu. Pour **simuler un temps minimum de rechargement** le plus simple est d'utiliser une variable tmprecharge initialisée avec la valeur du délai pour recharger (en nombre de tours de boucle de jeu) et une variable cptrecharge incrémentée à chaque passage dans la boucle de jeu. Dès que cptrecharge>=tmprecharge on peut tirer (par exemple en cliquant). Au moment où on tire (que le tir touche un acteur ou pas) cptrecharge est réinitialisée à 0.

Pour que le joueur puisse voir qu'il a tiré, on peut dans ce cas dessiner un graphique supplémentaire, par exemple un rond rouge, avant d'afficher le viseur. L'aspect de cet impact pourra dépendre du succès du tir (touché ou pas). Comme on ne le dessine qu'une fois au moment du tir et que ce dessin se fait sur le buffer (page) qui est effacé à chaque tour de jeu, l'impact n'apparaitra que très brièvement et il n'y aura pas besoin de l'effacer spécifiquement.

FOND, TRANSPARENCES, AVANTS-PLANS

Exercice 2 : Ecran de veille avec effets sur images *(Voir exemple 3_0 du cours 3)*

Imaginer et réaliser un écran de veille à partir d'une ou plusieurs images bougeant sur un fond et derrière un avant-plan.

Tester différentes fonctions draw_sprite de la bibliothèque pour obtenir un résultat original, effets de miroirs, déplacements en zigzag (fonction sinus ?), jouer avec un stretch_blit du fond sur le buffer … Pour plus de choix consultez la documentation allegro à la rubrique Blitting and sprites.

Attention aux unités spéciales d'Allegro pour les **angles des rotations** (paramètres de type fixed angle) :

- Angle en radians : f**`tofix(angle*128.0/M_PI)`**
- Angle en degrés : f**`tofix(angle*128.0/180.0)`**

Attention aux unités particulières d'Allegro pour les **changements d'échelles** (paramètres de type fixed scale) :

- utilisez **`ftofix(echelle)`** : echelle est une valeur float : 0.5 pour la moitié de la taille, 2.0 pour la taille double

Exercice 3 : Destruction de l'Etoile de la Mort *(Voir exemples 2_1 3_0 du cours 3)*

Il s'agit encore de tirer à la souris sur une cible mais cette fois-ci la cible est unique, grosse, se déplace de manière irrégulière (au hasard 1 fois sur 20 elle change au hasard de vitesse) et nous voulons donner l'impression que **la cible se détruit progressivement précisément aux endroits où les tirs touchent**.

Imaginez, par exemple, que notre cible soit l'étoile de la mort. Il faut deux images de même taille, sur fond couleur transparente :

- Etoile1 : une image de l'étoile de la mort en bon état
- Etoile2 : une image de l'étoile de la mort totalement ravagée en chaque point.

Pour obtenir Etoile2, le plus simple est de partir d'Etoile1 et de l'éditer (logiciel de dessin) : l'outil **smudge** ou le **clone brush** permettent de déstructurer l'image, puis ajouter des taches de roussi (ne passez pas trop de temps, l'important est le code).

Prévoyez aussi une image de fond. Eventuellement trouvez une image pour un avant- plan et pour un viseur, sur fonds transparents.

L'idée est d'avoir la séquence suivante dans la boucle de jeu :

```
Tant que (condition pour continuer)
        blit bitmap du fond sur buffer
        Bouger l'étoile de la mort
        Si tir au but
        Alors
                dessiner un petit disque de couleur invisible makecol(255,0,255)
                à la position correspondante directement sur la bitmap de Etoile1
                attention au changement de repère (coord. écran vers coord. image),
                        faites un schéma, réfléchissez !
                draw_sprite sur buffer de bitmap Etoile2
                draw_sprite sur buffer de bitmap Etoile1  (même position que Etoile2)
                draw_sprite sur buffer de bitmap de viseur (en position mouse_x et  mouse_y)
```

masked_blit de la bitmap avant plan sur le buffer
Actualiser écran (blit du buffer sur l'écran)
Petite pause pour temporiser (vitesse globale)
FinSi

En affichant Etoile1 au même endroit que Etoile2, on cache totalement Etoile2 au début du jeu. A mesure qu'on fait des "trous de transparence" avec la couleur spéciale dans Etoile1, on découvre progressivement la version détruite de l'image de départ qui est en dessous.

Approfondissement (facultatif):
Ecrivez une fonction qui prend en paramètre une bitmap et qui compte le pourcentage de l'image qui est colorée en transparence. En appelant cette fonction avec la bitmap Etoile1 en paramètre, il est possible d'avoir une **estimation quantitative du niveau de destruction atteint**. Au-delà d'un certain seuil, le jeu se termine (message de félicitation et/ou ce qui reste de l'étoile de la mort explose)

SEQUENCE ANIMEE

Exercice 4 : Histoire d'un bonhomme qui marche *(Voir exemple 4_0 du cours 3)*

Pour cet exercice vous pouvez prendre la séquence animée d'un bonhomme dans le ZIP regroupant l'ensemble des fichiers sources du Cours3 : images/bonhomme. Vous pouvez aussi essayer de dessiner grossièrement une séquence d'animation avec un logiciel de dessin (mais dans ce cas restez simple et ne faites pas trop d'images car le dessin animé prend beaucoup de temps à créer).

- récupérer dans un tableau de BITMAP* toutes les images de la séquence d'animation (tester !)
- gérer l'affichage de la succession des images à l'aide d'une variable imgcourante initialisée à 0
- contrôler la rapidité de l'animation et ne changer d'image qu'une fois sur tmpimg à l'aide d'une variable cptimg incrémentée de 1 à chaque tour : si cptimg >= tmpimg alors remettre cptimg à 0 et passer à l'image suivante de l'animation. L'animation tourne en boucle. Quand imgcourante est arrivée à NIMAGE, elle repart à 0.
- gérer simultanément le déplacement de l'animation à l'écran : le bonhomme se dirige vers la droite ou vers la gauche à partir des flèches du clavier

Tout ceci devra se faire en double buffer.

Si vous utilisez le bonhomme, restez sur un simple fond noir car il n'est pas dessiné sur fond transparent.

Exercice 5 : Flip le chat *(Voir exemple 4_0 du cours 3)*

C'est bien connu, un chat retombe toujours sur ses pattes.
Ce programme devra le démontrer en laissant tomber un chat, la tête à l'envers, depuis une abscisse aléatoire du haut de l'écran jusqu'au sol (situé à SCREEN_H/2). Entretemps il se sera retourné pour atterrir comme il convient à un félin. Arrivé au sol il partira en courant jusqu'au bord écran le plus proche. Dès qu'il sort complètement un nouveau chat est lâché du haut de l'écran...
Pour simplifier l'expérience on ne demande pas de décomposer le retournement du chat en séquence animée (le passage de draw_sprite_v_flip à draw_sprite suffira). De même il n'est pas demandé un amorti impeccable de l'atterrissage qui paraîtra sans doute un peu brutal...
Utilisez le décor et les images de la séquence images/cat dans le ZIP regroupant l'ensemble des fichiers sources du Cours3

Tout ceci devra se faire en double buffer.

Approfondissement (facultatif):
Gérer une "pluie" de chats qui tombent aléatoirement du haut de l'écran...