INFO7 - SAÉ

L1 informatique

Année 2024 – 2025

Présentation

Ce projet consiste à réaliser un jeu vidéo 2D mélangeant “snake” et “tetris”. Pour cela nous allons devoir gérer l'affichage graphique ainsi que les évènements claviers. En fonction du système utilisé Window / Mac / Linux sur la machine hôte, nous devons utiliser telle ou telle librairie pour accéder à ces fonctionnalités. Afin d'avoir un code le plus portable possible nous allons utiliser la libraire SDL (Simple DirectMedia Layer).
Sur la page wikipedia dédiée [1], nous pouvons lire que c'est une librairie libre, multiplateforme, permettant la création d'applications multimédias en deux dimensions. Elle permet entre autres :

- l'affichage vidéo dans des fenêtres systèmes ou en plein écran;
- la manipulation des évènements matériels et logiciels;
- la manipulation de l'audio numérique;
- la manipulation des périphériques d'entrée communs comme le clavier et la souris;
- l'utilisation de timers pour mesurer le temps.

De plus, il existe des modules complémentaires pour ajouter d'autres fonctionnalités :

- `SDL_ttf` permettant l'utilisation de polices de caractères;
- `SDL_image` prenant en charge différents formats d'images (BMP, PNG, JPEG, ...);

1 Organisation du projet

Le projet va se dérouler dans le cadre d'une Situation d'Apprentissage et dÉvaluation (SAÉ). Pour cela vous aurez des travaux à rendre et des évaluations tout au long du projet.

Durant le projet nous allons vous évaluer, vous devez donc être présent;e à toutes les séances.
Si vous êtes absent.e à une séance vous aurez alors -1 à la note finale de contrôle continu, et et si lors de cette séance une évaluation était prévue vous aurez en plus un 0 à cette évaluation.

1.1 Calendrier

Le projet va se diviser en 9 séances de 3h en salle machine de la façon suivante:

1. Séances 1 et 2 : affichages basiques avec la libriarie SDL ; gestion des évènements claviers / souris et boucle évènementielle.
2. Séances 3 à 5 : codage d'un jeu simple : le ”casse-brique”.
3. Séances 6 à 9 : codage du jeu final plus complet : le ”snaketris”.

[1] `https://fr.wikipedia.org/wiki/Simple_DirectMedia_Layer`

1.2 Consignes

- Un espace Moodle "info7" a été créé, vous y déposerez les fichiers demandés au fur et à mesure des séances.
- Les séances 1 et 2 se feront seules (en monome) donc chacun devra rendre son travail.
- Vous devrez travailler en binôme à partir de la séance 3 (sauf condition exceptionnelle validée par l'enseignant), lors des évaluations orales l'enseignant choisira qui du binôme répond aux questions. Un seul code sera fourni et les 2 membres devront pouvoir expliquer la démarche : si la solution de l'un s'impose alors l'autre doit pouvoir répondre, ce qui signifie que vous devez avoir communiqué.
- Les rendus des séances 3 à 9 se feront **par binome avec le nom que nous vous aurons indiqué lors de la séance 2**, et sous forme d'un seul fichier archive.
- Pour les séances 6 et 7, un rendu sur Moodle dans "Rendu_Snaketris_Bases". Vous serez interrogés sur le contenu de ces 2 séances au cours de la séance 8.
- Le jeu "casse-brique" de la séance 5 est à rendre au plus tard le 18 avril 2025 à 8h.
- Le jeu "snaketris" de la séance 9 est à rendre au plus tard le 9 mai 2025 à 8h.

Remarques de politesse :

- tout code rendu doit se compiler sans faute, une faute de compilation divisera la note par 2 ;
- le code doit être documenté : avant chaque fonction 1 ou 2 lignes doivent décrire la fonction et présenter les noms des variables utilisées notamment dans les paramètres ;
- les commentaires dans les fonctions seront mis avec parcimonie, uniquement lors de code un peu plus compliqué nécessitant une explication.

Critères d'évaluation :

- La qualité du code : correction, concision, précision, duplication du code, etc.
- Qualité d'écriture du code : espacement, choix des noms de variables, pertinence des commentaires, etc.
- Qualité des algorithmes : place mémoire utilisée, complexité (nombre de cases testées, nombre de comparaison, etc.)
- Capacité d'explication du code et écriture de tests incrémentaux

Évaluation :

- note finale :
  - Session1 = 1/2 examen + 1/2 C.C.
  - Session2 = Sup(examen2, (1/2 examen2 + 1/2 C.C.), Session1)

La section **??** détaille l'architecture du code et les fonctions nécessaires et les sections suivantes présentent le travail à faire et à rendre.

2 Environnement du projet

Vous allez écrire en C++ (sans passage de variables par référence dans les fonctions, sans classes, sans array/vector, ...) deux petits jeux vidéos.

Le travail va se découper en plusieurs phases et le code va se découper en plusieurs grandes parties avec une partie par fichier, et les fonctions se rapportant à une même partie seront regroupées dans un même fichier. Vous serez guidé.e.s au fur et à mesure des séances.

2.1 Modularité du programme

Dans ce projet vous allez utiliser la modularité des programmes, soit la possibilité de découper votre programme en plusieurs modules (c'est-à-dire plusieurs fichiers). Pour cela, vous allez déclarer les types de vos variables et de vos fonctions dans des fichiers d'*en-tête* qui auront comme extension .hpp. Ceux-ci seront ensuite indiqués par une instruction `include` dans les fichiers .cpp où les types et les fonctions seront utilisés.

Dans ce projet, tous les types des données et toutes les fonctions seront d'abord déclarées dans des fichiers dont l'extension sera `.hpp` et ensuite définies dans des fichiers de même nom où seule l'extension sera modifiée en `.cpp`. Cette décomposition va permettre de répartir les fonctions dans différents fichiers en fonction de leur thématique et de ne recompiler que le(s) fichier(s) modifié(s) depuis la dernière compilation, c'est ce qu'on appelle la compilation séparée.

2.1.1 Les fichiers d'en-tête

Les fichiers d'*en-tête* sont des fichiers dont l'extension sera `.hpp` et qui contiennent les en-têtes des fonctions qui seront utilisées ultérieurement dans des fichiers de code `.cpp` . Pour appeler les fonctions définies dans un fichier `.hpp` il suffira d'inclure ce fichier en en-tête du fichier code dans lequel les fonctions seront utilisées. C'est exactement ce que vous faites quand vous incluez une bibliothèque `#include <cmath>` et que vous utilisez la fonction `sqrt`.

En général on définit les entêtes des fonctions dans un fichier d'extension `.hpp` et les fonctions dans un fichier de même nom avec l'extension `.cpp`.

Exemple

Code du fichier calcul.hpp

```
    // calcul.hpp : déclaration des fonctions
    // Carre d'un nombre
    float carre(float);
3

    // x puissance y
    double puissance(int, int);
```

Code du fichier calcul.cpp

```
    // calcul.cpp : définition des fonctions
    // Carre d'un nombre
    float carre(float x) {return x*x;}
    // x puissance y
```

```
double puissance(int x, int y){
    double res = 1;
    for (int i=1; i<=y ; i++)
        res = res*x;
    return res;
}
```

Dans le code du fichier main.cpp

```
// main.cpp : programme principal
#include "calcul.hpp" //inclusion du fichier contenant les entêtes des fonctions,
                      //remarquez les guillemets "" à la place des <>
```

Parfois il arrive que les définitions se fassent directement dans le fichier `.hpp` mais le code risque d'être moins lisible, et surtout oblige à recompiler tout le code à chaque modification.

2.1.2 Compilation séparée : protection d'inclusion multiples d'un même fichier

La programmation modulaire implique automatiquement une compilation séparée. Lorsque des fonctions sont déclarées (ou définies) dans différents fichiers d'en-tête, il arrive souvent que l'inclusion d'un même fichier d'en-tête soit faite plusieurs fois car utilisée dans plusieurs fichiers. Pour éviter cette redéfinition des fonctions, des directives permettent de faire de la compilation conditionnelle.

Il existe plusieurs directives permettant la compilation conditionnelle (`#ifdef`, `#ifndef`, `#if`) afin de compiler les lignes comprises entre une de ces directives et la directive de fin du bloc `#endif` uniquement si la condition est remplie. Pour éviter l'inclusion multiple d'un même fichier d'en-tête, nous allons définir au début de ces fichiers une variable dite **variable de compilation** à l'aide de la directive `#define` et c'est seulement si cette variable `nom_variable`n'est pas encore connue lors la compilation (`#ifndef nom_variable`) que les fonctions seront compilées. Par convention la variable de compilation porte le nom du fichier d'en-tête et est écrite en majuscule (exemple : la variable de compilation du fichier "toto.hpp" sera `TOTO_HPP_`)
Exemple avec le fichier précédent :

```
//calcul.hpp : déclaration des fonctions
#ifndef CALCUL_HPP  //si la variable de compilation n'est pas
                     //connue (définie)
#define CALCUL_HPP  //on la définit

float carre(float x);
double puissance(int x, int y);

#endif  //fin des fonctions à compiler si la variable
        //de compilation n'est pas définie
```

2.1.3 Compilation séparée : ligne de commande

Pour compiler un programme simple (un seul fichier source), on utilise la commande suivante :

```
$ g++ nom_prog.cpp -o nom_prog
```

On demande au compilateur g++ de compiler le fichier source `nom_prog.cpp` et de créer le fichier exécutable `nom_prog` .
Rappel : si on ne précise pas `-o nom_prog` , le compilateur créera un exécutable nommé `a.out`.

Pour compiler un programme composé de 2 fichiers sources (`fonc.cpp` et `test_fonc.cpp`), il va falloir d'abord compiler chacun des fichiers séparément, compilation qui va renvoyer un fichier objet pour chaque fichier compilé. Ces fichiers objets vont être ensuite liés avec les fichiers précompilés d'une ou plusieurs bibliothèques grâce à l'éditeur de lien, afin de créer le fichier exécutable. La compilation va donc suivre les étapes suivantes :

```
$ g++ -c fonc.cpp #crée le fichier fonc.o
$ g++ -c test_fonc.cpp #crée le fichier test_fonc.o
$ g++ -o test_fonc fonc.o test_fonc.o #crée le fichier exécutable test_fonc
                                      #à partir des fichiers objets précédents
```

2.1.4 Compilation séparée : Makefile

Pour éviter de compiler chaque fois les différents fichiers modifiés et ensuite de créer l'exécutable, et surtout pour éviter d'oublier de compiler un fichier anciennement modifié, il est d'usage de créer un fichier nommé **Makefile** qui va permettre de rassembler toutes les lignes de commande. Une fois ce fichier créé, la seule commande à exécuter sera alors `make` qui permettra alors de créer l'exécutable dont le nom aura été donné dans le fichier Makefile. Il est possible de spécifier quelle règle du Makefile exécuter (par exemple, `make test_fonc`) , sinon c'est la première règle du fichier qui est choisie.

Chaque règle Makefile se compose d'un nom, d'une liste optionnelle d'ingrédients, et d'une liste de commandes à effectuer. Le nom de la règle est généralement le nom du fichier généré par la règle, et les ingrédients correspondent soient à des noms de fichiers, soit à d'autres règles. Les commandes d'une règle ne sont exécutées que si et seulement si le fichier à générer n'existe pas ou qu'un des ingrédient est plus récent que celui-ci. Ainsi, seul le code des fichiers modifiés depuis la dernière compilation sont recompilés.

La création du fichier Makefile demande de bien comprendre la compilation. Aussi dans la SAÉ nous vous donnerons une Makefile simple que vous enrichirez au fur et à mesure dans les 5 premières séances, et que vous adapterez aux séances suivantes.
Exemple de Makefile pour l'exemple précédent:

```
default: test_fonc # regle par defaut
fonc.o: fonc.cpp # regle de creation du fichier fonc.o
    g++ -c fonc.cpp
test_fonc.o: test_fonc.cpp
    g++ -c test_fonc.cpp
test_fonc: fonc.o test_fonc.o  # regle de l'executable en fonction des .o
    g++ -o test_fonc fonc.o test_fonc.o
```

On pourra utiliser la commande `make test_fonc` qui mettra en oeuvre les différentes règles suivant les fichiers modifiés depuis la dernière compilation.

Séance 1

Le but de cette séance est de créer un environnement de développement utilisant la librairie SDL et de faire nos premiers affichages graphiques.

1 Description

Pour cette séance de découverte nous allons vous guider pas à pas.

1.1 Compilation avec Make

1. Créer un répertoires `src`.
2. Créer le fichier `src/main.cpp` suivant :

```
#include <iostream>

using namespace std;

int main(int argc, char** argv) {
    cout << "Vive le C !" << endl;
}
```

3. Créer le fichier `Makefile` suivant :

```
EXE = test
CPP = g++
CFLAGS = -Wall -O3

ALL = $(EXE)

$(EXE): src/main.cpp
	$(CPP) $(CFLAGS) $^ -o $@
```

4. Compiler votre code avec la commande `make` et exécuter le avec `./test`

1.2 Première fenêtre SDL

Dans le but d'avoir un code propre et organisé, nous allons créer une structure `Window` contenant les données nécessaires à la création et à l'utilisation d'une fenêtre SDL. Afin d'avoir un code organisé et facilement modifiable, nous allons utiliser deux nouveaux fichiers : `src/window.hpp` et `src/window.cpp`.

1.2.1 Déclaration de `Window`

Créer le fichier `src/window.hpp` contenant

```
#ifndef WINDOW_HPP
#define WINDOW_HPP

#include <iostream>
#include <SDL2/SDL.h>

using namespace std;

struct Window {
  SDL_Window* sdl_window;
  SDL_Renderer* sdl_renderer;
  int width;
  int height;
};

void init_window(Window* window, int width, int height, string title);

void close_window(Window* window);

#endif
```

- La commande `#include <SDL2/SDL.h>` sert à charger l'entête de la libraire SDL.
- La variable `sdl_window` pointera vers un enregistrement de type `SDL_Window` contenant les données de la fenêtre SDL.
- La variable `sdl_renderer` pointera vers un enregistrement de type `SDL_Renderer` permettant d'effectuer les différents rendus graphiques.
- Les variables `width` et `height` contiendront la largeur et la hauteur de la fenêtre créée.
- La fonction `init_window(Window* window, int width, int height, string title)` initialisera la fenêtre `window` passer en paramètre. Elle sera de largeur `width`, de hauteur `height`. Chaque fenêtre portant un titre, on précise celui de la notre gràce à l'argument `title`.
- La fonction `close_window(Window* window)` permettra de fermer la fenêtre à la fin du programme.

1.2.2 Implémentation de `Window`

Créer le fichier `src/window.cpp` contenant le code des fonctions `init_window` et `close_window` :

```
#include "window.hpp"

void init_window(Window* window, int width, int height, string title) {
  if (SDL_Init(SDL_INIT_VIDEO) < 0) {
```

```
    cerr << "Could not initialize SDL2: error " << SDL_GetError() << endl;
    SDL_Quit();
  }

  window -> width = width;
  window -> height = height;

  int ret = SDL_CreateWindowAndRenderer(width, height, 0, &window -> sdl_window, &window -> sdl_renderer);
  if (ret < 0) {
    cerr << " Could not create SDL window: error " << SDL_GetError() << endl;
    SDL_Quit();
  }

  SDL_SetWindowTitle(window -> sdl_window, title.c_str());
}

void close_window(Window* window) {
  SDL_DestroyWindow(window -> sdl_window);
  SDL_DestroyRenderer(window -> sdl_renderer);
  SDL_Quit();
}
```

- La commande `SDL_Init(SDL_INIT_VIDEO)` initialise la librairie SDL. C'est la toute première commande SDL qui doit être exécutée. Elle retourne une valeur négative si une erreur c'est produite.
- La commande `SDL_GetError()` retourne la dernière erreur rencontrée par la SDL.
- La commande `SDL_Quit()` demande la libération des ressources utilisées par la SDL. Aucune autre commande SDL ne doit être exécutée après celle-ci.
- Le nom des autres commandes est suffisament clair pour que vous compreniez leur fonctionnement.

1.2.3 Modification de la fonction `main`

Pour utiliser notre fenêtre SDL, nous devons modifier le fichier `src/main.cpp`

```
#include <iostream>
#include "window.hpp"

using namespace std;

int main(int argc, char** argv) {
    Window window;
    init_window(&window, 800, 600, "Ma première fenêtre SDL");
    SDL_Delay(2000); // Wait for 2 seconds
    close_window(&window);
}
```

1.2.4 Modification du fichier de compilation

Nous devons préciser au compilateur comment trouver le fichier `SDL2/SDL.h` contenant la déclaration des structures et fonctions SDL. De plus il a besoin d'accéder au code pré-compila de la librairie pour pouvoir compiler notre programme. Pour cela nous allons modifier le fichier Makefile :

```
EXE     = test
CPP     = g++
SDL_INC = `sdl2-config --cflags`
SDL_LIB = `sdl2-config --libs`
CFLAGS  = -O3 -Wall $(SDL_INC)
LIBS    = $(SDL_LIB)

ALL = $(EXE)

obj/%.o: src/%.cpp src/%.hpp
    $(CPP) $(CFLAGS) -c $< -o $@

$(EXE): obj/window.o src/main.cpp
    $(CPP) $(CFLAGS) $^ -o $@ $(LIBS)

clean:
    -$(RM) obj/*.o $(EXE)
```

- `SDL_INC` specifie où se trouve les fichiers d'entêtes de la librairie SDL (les .hpp)
- `SDL_LIB` specifie où se trouve le code pre-compilé de la librairie SDL
- La ligne `obj/%.o: src/%.cpp src/%.hpp` spécifie comment construire un fichier .obj à partir du .hpp et .cpp correspondant. Le fichier `obj/toto.obj` sera ainsi obtenu à partir des fichier `src/toto.hpp` et `src/toto.cpp`. Un tel fichier objet contient du code partiellement compilé.
- Dans la ligne `$(EXE): obj/window.o src/main.cpp` nous précisons que nous avons besoin du fichier de code partiellement compilé `obj/window.o` et du fichier `src/main.cpp` pour créer notre executable `test`
- La dernière ligne permet de supprimer tous les fichiers objets à l'aide de la commande `make clean`.

Attention l'apostrophe utilisé pour les lignes `SDL_INC` et `SDL_LIB` est obtenu avec la touche `7` du clavier.

1.3 Premiers dessins

1.3.1 Couleurs

La SDL possède une structure `SDL_Color` permettant de stocker une couleur au format rgba :

```
struct SDL_Color {
   Uint8 r;
```

```
    Uint8 g;
    Uint8 b;
    Uint8 a;
};
```

Le type `Uint8` répresente un entier non signé codé sur 8 bits et donc à valeurs entre 0 et 255. Les variables `r`, `g` et `b` sont les composantes rouge, verte et bleue de la couleur. La variable `a` est utilisée pour la transparence (c'est le canal alpha).

1.3.2 Ajout de fonctionnalités

Modifier la structure `window` dans le fichier `src/window.hpp` en y ajoutant :

- une variable `background` de type `SDL_Color`, qui sera utilisée pour stocker la couleur de fond;
- une variable `foreground` de type `SDL_Color`, qui sera utilisée pour stocker la couleur de dessin;
- les déclarations des fonctions suivantes :
  - `void set_color(SDL_Color* dst, int r, int g, int b, int a)`, qui sera utilisée pour définir une couleur au format SDL;
  - `void set_color(SDL_Color* dst, SDL_Color* src)`, qui sera utilisée pour affecter une couleur;
  - `void clear_window(Window* window)`, qui effacera la fenêtre `window` avec sa couleur de fond `background`;
  - `void draw_fill_rectangle(Window* window, int x, int y, int w, int h);` qui dessinera un rectangle plein sur la fenêtre `window`;
  - `void refresh_window(Window* window);` qui mettra à jour la fenêtre.

Vous implémenterez les quatre fonctions précédentes dans le fichier `src/window.cpp`. Le fonctionnement de la fonction `set_color` étant sufffisamment clair nous ne la détaillons pas davantage.

Voici le code pour les fonctions `clear_window` et `refresh_window` :

```
void clear_window(Window* window) {
  // Select the background color
  SDL_SetRenderDrawColor(window -> sdl_renderer,
                         window -> background.r,
                         window -> background.g,
                         window -> background.b,
                         window -> background.a);
  // We fill the window
  SDL_RenderClear(window -> sdl_renderer);
}

void refresh_window(Window* window){
  SDL_RenderPresent(window -> sdl_renderer);
}
```

1.3.3 Rectangle

La librairie SDL possède une structure `SDL_Rect` permettant de représenter des rectangles à coordonnées entières et dirigé horizontalement ou verticalament :

```
struct SDL_Rect {
   int x, y;
   int w, h;
};
```

Les variables `x` et `y` donnent les coordonnées du coin supérieur gauche du rectangle. Les variables `w` et `h` précisent alors la largeur et la hauteur du rectangle. **Attention**, en informatique l'orgine du repére pour le dessin se strouve dans le coin supérieur gauche de l'écran.

Voici le code de la fonction `draw_fill_rectangle` :

```
void draw_fill_rectangle(Window* window, int x, int y, int w, int h) {
  SDL_SetRenderDrawColor(window -> sdl_renderer,
                         window -> foreground.r,
                         window -> foreground.g,
                         window -> foreground.b,
                         window -> foreground.a);
  SDL_Rect rectangle;
  rectangle.x = x;
  rectangle.y = y;
  rectangle.w = w;
  rectangle.h = h;
  SDL_RenderFillRect(window -> sdl_renderer, &rectangle);
}
```

1.3.4 Mise en place

Modifier votre fichier `src/main.cpp` afin d'obtenir :

```
int main(int argc, char** argv) {
    Window window;
    init_window(&window, 800, 600, "Mon premier rectangle en SDL");
    set_color(&window.background, 255, 255, 255, 255); // Blanc
    set_color(&window.foreground, 0, 0, 255, 255); // Bleu
    clear_window(&window);
    draw_fill_rectangle(&window, 300, 200, 250, 100);
    refresh_window(&window);
    SDL_Delay(5000); // Wait for 5 seconds
    close_window(&window);
}
```

Testez votre programme. Vous pouvez modifier les différents paramètres du dessin précisés dans la fonction `main`.

1.4 Deux dessins

Pour la fin de cette séance, vous allez travailler uniquement dans le fichier `src/main.cpp`

1.4.1 Un échiquier

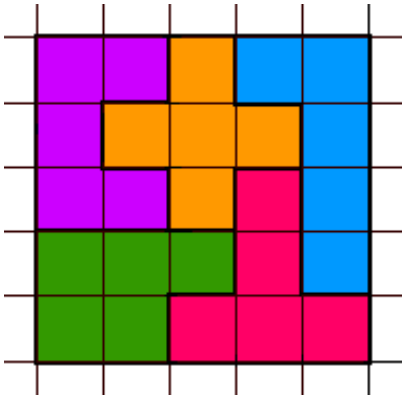
Ecrire une fonction `echiquier` permettant de dessiner un échiquier sur toute la fenêtre. Votre fonction prendra en paramètres :

- le nombre de lignes de l'échiquier
- le nombre de colonnes de l'échiquier
- deux couleurs de type `SDL_Color`

Les cases de votre échiquer ne seront pas forcément carré. Les cases seront alternativement coloriée avec les deux couleurs passées en paramètre.

1.4.2 Un dessin en pentominos

Ecrire une fonction `void dessine_pentominos` permettant de reproduire le dessin suivant:



2 A rendre

Pour cette première séance vous rendrez une archive contenant :

- les répertoires `src` et `obj`
- les fichiers : `Makefile`, `src/window.hpp` et `src/window.cpp` obtenus à l'issue du 1.3.4
- le fichier : `src/main.cpp` contenant le code des fonctions `echiquier` et `pentomino`
- lors de son exécution, votre programme devra :
  - faire le dessin d'un échiquier à 8 lignes et 10 colonnes avec des cases bleues et rouges

– faire une pause de 4 secondes
– faire le dessin en pentominos
– faire une pause de 4 secondes
– se fermer

Séance 2

Le but de cette séance est l'ajout d'une boucle événementielle au code de la séance précédente.

1 Description

Nous vous conseillons de repartir d'une **copie** du code obtenu lors de la séance précédente. Commencer par supprimer l'implémentation et les appels aux fonctions `echiquier` et `dessin_pentomino`.

1.1 Fermeture du programme à la demande

Dans votre fonction `main` du fichier `src/main.cpp` apporter les modifications suivantes :

- Ajouter une variable `quit` de type `bool` et l'initialiser à `false`.
- Ajouter une variable `event` de type `SDL_Event` non initialisée.
- Faire un boucle `while` qui tourne tant que la variable `quit` n'est pas arrivée à `true`
- Dans le pas de boucle, détecter si l'action de *demande de fermeture de fenêtre* à été déclenchée à l'aide de la commande :

```
while (SDL_PollEvent(&event) != 0){
  if(event.type == SDL_QUIT){
    quit = true;
  }
}
```

- Attendre 100 millisecondes avant de recommencer un pas de boucle.
- Tester votre programme.

Pour le moment nous ne gérons que l'évènement du type `SDL_QUIT`. Il existe d'autres types d'évènements :

- `SDL_KEYDOWN`, pour l'appui d'une touche du clavier,
- `SDL_KEYUP`, pour le relâchement d'une touche du clavier,
- `SDL_MOUSEMOTION`, pour le déplacement de la souris,
- `SDL_MOUSEBUTTONDOWN`, pour l'appui d'un bouton de la souris,
- `SDL_MOUSEBUTTONUP`, pour le relâchement d'un bouton de la souris,
- `SDL_MOUSEWHEEL`, pour l'action de la molette de la souris.

1.2 Gestion du clavier

Nous allons nous intéresser aux évènements claviers. Modifier la boucle

```
while (SDL_PollEvent(&event) != 0)
```

de sorte qu'elle détecte l'appui d'une touche sur le clavier. Une fois l'appui détecté, nous devons déterminer quelle touche a été enfoncée, pour cela nous utilisons la commande

```
SDL_KeyboardEvent key_event = event.key;
```

qui récupére l'évènement *clavier* à partir de l'évènement `event`. L'identification de la touche enfoncée est alors obtenue grâce à `key_event.keysym.sym` A l'aide de la commande `switch` nous pouvons alors détecter de quelle touche du clavier il s'agit. Il est néanmoins nécessaire de connaître le nom associé à chaque touche du clavier. La liste des *key codes* est disponible ici :

`https://github.com/libsdl-org/SDL/blob/SDL2/include/SDL_keycode.h`

Voici une petite liste de code :

- `SDLK_a` pour la touche `A`;
- `SDLK_b` pour la touche `B`;
- ...
- `SDLK_z` pour la touche `Z`;
- `SDLK_SPACE` pour la touche `espace`;
- `SDLK_UP` pour la touche *flèche du haut*
- `SDLK_DOWN` pour la touche *flèche du bas*
- `SDLK_LEFT` pour la touche *flèche de gauche*
- `SDLK_RIGHT` pour la touche *flèche de droite*

Modifier la fonction `main` pour que votre programme possède les fonctionalités suivantes :

- Arrêt du programme après appui de la touche `Q`
- Lors de la pression d'une touche du pavé directionnel, affichage (avec un `cout`) de la direction de la flèche

1.3 Gestion de la souris

Comme pour le cas de la gestion clavier, nous pouvons récupérer les informations d'un évènement *souris* après que celui-ci ait lieu. Pour se faire on utilise la commande

```
SDL_MouseMotionEvent motion_event = event.motion;
```

Les coordonnées de la souris lorsque l'évènement s'est déclenché sont alors obtenues à l'aide de `motion_event.x` et `motion_event.y`. Si un bouton de la souris a été enfoncé ou relâché, on récupère l'information concernant l'évènement grâce à la commande :

```
SDL_MouseButtonEvent button_event = event.button;
```

Les coordonnées de la souris lors de l'évènement *bouton souris* sont obtenues comme précédement à l'aide de `button_event.x` et `button_event.y`. L'identifiant du bouton ayant déclenché l'évènement est obtenu à l'aide de `button_event.button`.

Pour plus d'information concernant les évènements gérés par la SDL il peut être utile de consulter

`https://github.com/libsdl-org/SDL/blob/SDL2/include/SDL_events.h`

Modifier votre fonction main pour que lorsque vous cliquez sur un bouton de la souris vous affichiez en console :

- l'index du bouton qui vient d'être appuyé
- son statut (appuyé ou relâché à l'aide de `.state` et de `SDL_PRESSED` et `SDL_RELEASED`
- les coordonnées du pointeur de la souris

1.4 Affichage d'une image

1.4.1 Modification du fichier `Makefile`

Commençons par charger la librairie `SDL_image` lors de la compilation de notre programme. Modifier la ligne `SDL_LIB` du fichier `Makefile` :

```
SDL_LIB = `sdl2-config --libs` -lSDL2_image
```

1.4.2 Modification du fichier `window.hpp`

Dans le fichier `window.hpp` :

- Ajouter la ligne `#include <SDL2/SDL_image.h>` en début de fichier.
- Ajouter la ligne `SDL_Texture* load_image(Windows* window, string image);`
- Ajouter la ligne `void draw_texture(Window* window, SDL_Texture* texture, int x, int y, int w, int h);`

La commande `load_image` permettra de créer une texture à partir d'un fichier contenant une image au format PNG. La commande `draw_texture` permet d'afficher une texture à l'écran. Pour cela nous précisons les coordonnées du coin supérieur gauche ainsi que la largeur et la hauteur utilisées lors de l'affichage.

1.4.3 Modification du fichier `window.cpp`

Dans le fichier `window.cpp` ajouter le code suivant:

```
SDL_Texture* load_image(Window* window, string str) {
  SDL_Surface* surface = IMG_Load(str.c_str());
  if (surface == nullptr) {
    cerr << "Erreur SDL_Image : " << IMG_GetError();
    return nullptr;
  }
  SDL_Texture* texture = SDL_CreateTextureFromSurface(window -> sdl_renderer, surface);
  SDL_FreeSurface(surface);
  if (texture == nullptr) {
    cerr << "Erreur lors de la creation d'une textture à partir d'une surface" <<endl;
    return nullptr;
  }
  return texture;
}

void draw_texture(Window* window, SDL_Texture* texture, int x, int y, int w, int h) {
  SDL_Rect box;
  box.x = 10;
  box.y = 10;
  box.w = 400;
  box.h = 400;
  SDL_RenderCopy(window->sdl_renderer, texture, NULL, &box);
}
```

1.4.4 Modification du fichier `main.cpp`

Télécharger l'image `tux.png` depuis `Moodle` et modifier votre fichier `main.cpp` pour qu'il

- charge votre image dans une texture à l'aide de la commande `load_image`;
- affiche la texture correspondant à l'aide de la commande `draw_texture`.

Attention le chemin d'accès à votre image doit être valide depuis l'emplacement de l'exécutable.

Comme vous pouvez le voir (en ouvrant l'image avec `gimp` par exemple) l'image `tux.png` contient des zones transparentes qui sont bien rendues dans votre fenêtre `SDL`. C'est à cela que sert la composante alpha dans les couleurs de la `SDL`.

1.5 Affichage de texte

Curieusement il est assez compliqué d'afficher un texte dans une fenêtre `SDL`. En effet, supposons que vous souhaitiez afficher *Bonjour*. Après avoir chargé une police de caractères, il vous faudra créer une texture à partir de votre chaîne de caractères, puis afficher votre texture à l'écran comme lors de la section précédente.

Commencer par modifier le fichier `Makefile` pour qu'il contienne la ligne :

```
SDL_LIB = `sdl2-config --libs` -lSDL2_image -lSDL2_ttf
```

Aller ensuite sur `moodle` pour télécharger la police `VeraMono` contenu dans le fichier `VeraMono.ttf`. Vous enregistrez ce fichier dans le répertoire contenant l'éxecutable de votre programme.

1.5.1 Modification du fichier `window.hpp`

Modifier le fichier `src/window.hpp` :

- Ajouter la ligne `#include <SDL2/SDL_ttf.h>`
- Ajouter la variable `TTF_Font* sdl_font;` à la structure `Window`;
- Ajouter la ligne `void draw_text(Window* window, string text, int x, int y);`

1.5.2 Modification du fichier `window.cpp`

Modifier le fichier `src/window.cpp` :

- A la fin de la fonction `init_window`, ajouter les lignes

```
// Initialise SDL_TTF for TTF font rendering
if (TTF_Init() == -1){
  cout << "Could not initialize SDL_TTF: error " << TTF_GetError() << endl;
  SDL_Quit();
}
// Spécifie la police
window -> sdl_font = TTF_OpenFont("VeraMono.ttf", 20);
if (window -> sdl_font == NULL) {
  cout << "Could not load font: error " << TTF_GetError() << endl;
  SDL_Quit();
}
```

- Ajouter le code

```
void draw_text(Window* window, string str, int x, int y) {
  SDL_Surface* surface = TTF_RenderText_Shaded(window -> sdl_font, str.c_str(),
 window -> foreground, window -> background);

  // If an error occured
  if (surface == NULL) {
     cout << "Could not init the SDL surface for TTF: " << TTF_GetError() << endl;
     exit(0);
  }

  // Retrieve width and height of the surface
  int h = surface -> h;
  int w = surface -> w;
```

```
  // Make a SDL Texture from the Surface
  SDL_Texture* texture = SDL_CreateTextureFromSurface(window -> sdl_renderer, surface);

  // Free the SDL surface
  SDL_FreeSurface(surface);

  // Dispaly an error if we cannot make the texture
  if (texture == nullptr){
    cout << "Could not init the SDL Texture for TTF: " << SDL_GetError() << endl;
    SDL_Quit();
  }

  draw_texture(window, texture, x, y, w, h);

  SDL_DestroyTexture(texture);
}
```

1.5.3 Modification du fichier `main.cpp`

Apporter les modifications suivantes au fichier `src/main.cpp` :

- Définir la couleur `backround` de `window`;
- Définir la couleur `foreground` de `window`;
- Ajouter la ligne `draw_text(&window, "Bonjour le monde !", 10, 10);`

1.6 A rendre

Vous allez réaliser un petit programme affichant une balle se déplaçant et rebondissant sur les bords de la fenêtre.

Concernant la balle vous aurez besoin de:

- un entier `x` pour son abscisse;
- un entier `y` pour son ordonnée;
- un entier `dx` valant -1 ou 1 pour le déplacement horizontal de la balle
- un entier `dy` valant -1 ou 1 pour le déplacement vertical de la balle

Au lancement de votre programme, la balle sera à une position prédéfinie à l'avance et `dx` et `dy` vaudront tous les deux 1. La balle sera dessinée sur un carré de 20x20 pixels à l'aide de l'image `balle.png` disponible sur `moodle`.

Une variable entière `speed` comprise entre 1 et 10 donnera une information sur la vitesse de déplacement de la balle. La boucle évènementielle lira la demande de fermeture du programme / l'appui d'une touche

au clavier / le clic d'un bouton de la souris. Une pause de 10-`speed` ms sera effectuée afin de faire un nouveau pas de boucle.

La variable `speed` sera augmentée de 1 lors de l'appui de la flèche du haut et diminuée de 1 lors de l'appui de la flèche du bas. Vous quitterez le programme lors de l'appui de la touche `q`.

Lorsque l'utilisateur cliquera dans la fenêtre avec la souris, la balle viendra se positionner sous le curseur.

Séance 3

Durant cette séance, ainsi que les séances 4 et 5 vous allez développer un jeu de type "casse-brique" simple. Dans ce jeu, le joueur contrôle le déplacement horizontal d'une raquette située en bas de l'écran à l'aide des flèches `gauche` et `droite`. Une balle se déplace à l'écran et ne doit pas atteindre la zone en dessous de la raquette. Différents types de bloc sont placés à l'écran. Certains sont incassables et d'autres sont cassables. Voici une copie d'écran d'un tel casse-brique.

La balle est le carré blanc, la raquette est de couleur cyan. Les briques incassables, de couleur grise, sont sur les bords gauche, haut et droite de l'écran. Les briques vertes et rouges sont toutes cassables. Pour mener ce jeu à bien il faut :

- Représenter le monde contenant les blocs et dans lequel la balle et la raquette se déplaceront;
- Permettre le déplacement de la raquette;
- Implémenter les différentes règles de rebond et de casse pour la balle;
- Animer tout ceci grâce à une boucle évènementielle unique.

Cette première séance sur le jeu casse-brique sera essentiellement consacrée à la représentation et au chargement du monde.

1 Ce qu'il faudra rendre

Les 3 séances vont être évaluées en une seule fois à la fin de la séance 5 et le dépôt attendu contiendra une archive contenant :

- 3 répertoires :
  - "S3" qui contiendra le code de fin de séance 3 : structure et affichage en console du monde à charger;
  - "S4" qui contiendra le code de l'affichage du jeu dans une fenêtre SDL et du déplacement de la raquette et d'un déplacement simple de la balle;
  - "S5" qui contiendra le jeu complet avec le déplacement de la balle prenant en compte les différents rebonds et l'impact de la balle sur les briques, lse bords, la raquette.
- un fichier:
  - reprenant par fichier .hpp, pour chaque fonction une ligne de description et ses dépendances aux autres fonctions;
  - Un manuel expliquant quelles sont les fonctions à appeler pour créer une fenêtre, dessiner le jeu, mettre à jour le jeu, ...

2 Représentation du monde

Nous allons commencer par concevoir une structure `World` permettant de représenter le monde de briques dans lequel évolueront notre balle et notre raquette. L'implémentation de la structure `World` et des fonctions associées se fera dans les fichiers `src/world.hpp` et `src/world.cpp`.

Commencez par créer un fichier `src/world.hpp` contenant

```
#ifndef WORLD_HPP
#define WORLD_HPP

#include <iostream>

usign namespace std;

#endif
```

ainsi qu'un fichier `src/world.cpp` contenant

```
#include "world.hpp"
```

Afin que vos fichiers `src/world.hpp` et `src/world.cpp` soient intégrés à votre programme, il est nécessaire de modifier le fichier `Makefile` en remplacant la ligne commencant par `$(EXE)` par

```
$(EXE): obj/window.o obj/world.o src/main.cpp
```

Finalement vous ajouterez la ligne

```
#include "world.hpp"
```

au fichier `src/main.cpp`.

Vous êtes maintenant prêt à concevoir et implémenter la structure `World`.

2.1 Différents types de bloc

Notre monde sera constitué d'une grille contenant des blocs de différentes natures. Pour représenter ces différents blocs nous allons utiliser une énumération.

2.1.1 Enumération

Une énumération en `C` (mot clé `enum`) est un type qui permet de modéliser une variable décrivant une catégorie. Ainsi, une énumération définit un ensemble fini de valeurs entières. Chaque valeur entière est identifiée par un nom qui fait sens pour l'utilisateur. Par exemple, la catégorie Fruits peut avoir comme valeur Pomme, Poire, etc. L'énumération se définit à l'aide du mot clé `enum` et porte un nom.
Déclaration du type :

```
enum nom_enum {
    val1, val2, val3
};
```

Utilisation du type comme une variable classique :

```
nom_enum var;
var = val1;
```

Par exemple, pour décrire la catégorie des fruits :

```
enum Fruit {
    Pomme, Poire, Cerise
};

Fruit dessert;    // déclaration d'une variable de type Fruit
dessert = Pomme;  // Affectation d'une valeur
```

2.1.2 Cas des blocs

Voici les différents types de bloc dont nous aurons besoin.

- `Empty` pour les blocs vides (en noir sur la capture d'écran)
- `Border` pour les blocs incassables (en gris sur la capture d'écran)
- `Type1` pour les blocs cassables de type 1 (en rouge sur la capture d'écran)
- `Type2` pour les blocs cassables de type 2 (en vert sur la capture d'écran)
- `Lose` pour les blocs de la ligne du bas en dessous de la raquette (aussi en noir sur la capture d'écran).

Dans le fichier `src/world.hpp` créér l'énumération `Block` contenant les catégories précédentes.

2.2 Structure `World`

Le monde sera représenté par une grille contenant des blocs. La largeur et la hauteur de cette grille ne seront pas connues à la compilation. Dans le fichier `src/world.hpp` créer une structure `World` avec les données suivantes :

- `width` de type `int` pour la largeur de la grille;
- `height` de type `int` pour la hauteur de la grille;
- `grid` de type pointeur vers `Block` pour les blocs constituant la grille.

Dans les fichiers `src/world.hpp` et `src/world.cpp` définir et implémenter les fonctions:

- `void init_world(World* world, int w, int h)` qui initialise la structure pointée par `world`. Les paramètres `h` et `w` précisant la hauteur et la largeur de la grille respectivement. La grille sera un tableau de `Block` de taille `h` $\times$ `w` alloué dynamiquement. Par défaut, les blocs constituant la grille seront initialisés à `Empty`;
- `void free_world(World* world)` qui détruit le tableau `grid` alloué dynamiquement.

2.3 Accès aux blocs

Notre monde est constitué d'une grille 2D de blocs mais la variable `grid` de notre structure `World` est un tableau 1D. Supposons que notre grille soit de largeur w et de hauteur h. Les coordonnées (x, y) d'un bloc de la grille vérifie alors $x \in \{0, \ldots, w-1\}$ et $y \in \{0, \ldots, h-1\}$. L'indice i du tableau `grid` du bloc de coordonnées (x, y) vérifiera alors

$$i = y \times w + x$$

Créer les fonctions suivantes:

- `Block read(World* world, int x, int y)` qui retourne le bloc aux coordonnées (`x`, `y`). Vous afficherez un message d'erreur si les coordonnées sont incorrectes et vous retournerez alors la valeur `Empty`;
- `void write(World* world, int x, int y, Block b)` qui écrit la valeur `b` dans le bloc de coordonnées (`x`, `y`). Vous afficherez un message d'erreur et ne ferez aucune modification si les coordonnées sont incorrectes.

2.4 Affichage texte du monde

Créer une fonction `void display(World* world)` permettant d'afficher en mode texte le monde pointé par `world`. Nous vous proposons de représenter chaque type de bloc par un caractère en suivant le tableau suivant :
La cellule de la grille de coordonnées $(0, 0)$ sera affichée dans le coin supérieur gauche de la fenêtre.

À ce stade il est important de tester toutes les fonctions que vous avez créées afin de vous assurez que tout fonctionne correctement.

Type de bloc	Caractère
Empty	'.'
Border	'#'
Lose	'$'
Type1	'1'
Type2	'2'

Table 1: Correspondance type de bloc et caractère

2.5 Chargement du monde à partir d'un fichier

Le monde du casse-brique sera initialisé à partir d'un fichier texte constitué de la manière suivante:

- la première ligne précise la largeur de la grille;
- la deuxième ligne précise la hauteur de la grille;
- les autres lignes décrivent la grille en respectant le codage donné à la table 1.

Le fichier `world.dat` disponible sur Moodle est un exemple d'un tel fichier. C'est celui qui a été utilisé pour initialiser le monde de la capture d'écran.
Les deux sections suivantes présentent comment lire et écrire dans un fichier texte en `C`.

2.5.1 Lecture d'un fichier

La classe `fstream` permet de lire ou écrire un fichier à partir d'un flux. Plus particulièrement, la classe `ifstream` permet de gérer des opérations d'entrée sur des fichiers :

1. Pour ouvrir le fichier de données dont le nom est contenu dans la variable `nom_fic` :

   ```
   string nom_fic = "toto";
   ifstream fic(nom_fic);
   ```

   Dans ce cas, le fichier **toto** est ouvert par un flux nommé **fic** de la même façon que le flux d'entrée standard "cin" que vous connaissez déjà. La récupération des données contenues dans le fichier est aussi simple que lire les données saisies au clavier.

2. Pour lire les données contenues dans un fichier, en fonction du type de données récupérées le code devra être adapté mais ressemblera aux quelques lignes suivantes lorsque le fichier contient des entiers :

   ```
   if (fic) {
     int tab[100]; // lecture des données de type entier
                   // enregistrées dans un tableau
     int i = 0;
     while (!fic.eof() && i < 100) { // tant que la fin
                           // du fichier n'est pas lue et
                           // que le tableau n'est pas plein
   ```

```
      fic >> tab[i]; // lire chaque donnée comme un entier
                     // à enregistrer dans tab[i]
      i++;
  }
}
else {
  cout << "ERREUR : impossible d'ouvrir le fichier" << endl;
}
```

3. Pour fermer le fichier lorsque la lecture a été effectuée : `fic.close();`

2.5.2 Écriture dans un fichier

De la même manière que la lecture d'un fichier s'effectue à l'aide d'un flux d'entrée, l'écriture (d'un fichier texte) s'effectue à l'aide d'un flux de sortie en utilisant la classe ofstream qui gère les opérations de sortie sur des fichiers :

1. Pour ouvrir un fichier de données :

   ```
   string nom_fic = "toto";
   ofstream fic(nom_fic);
   ```

   Dans ce cas, le fichier **toto** est ouvert par un flux nommé **fic** de la même façon que le flux de sortie standard "cout". L'écriture des données dans le fichier s'effectue aussi simplement que l'affichage à l'écran.

2. Le code suivant illustre l'écriture de données (ici des entiers) dans un fichier texte :

   ```
   if (fic) {
     int i = 0;
     while (i < 100) { // enregistrement des valeurs d'un tableau tab
                       // de 100 éléments dans le fichier
         fic << tab[i] << " "; // écriture de l'élément i
                               // syntaxe similaire à l'affichage
         i++;
     }
   }
   else {
     cout << "ERREUR : impossible d'ouvrir le fichier en sortie" << endl;
   }
   ```

3. Pour fermer un fichier lorsque l'écriture est terminée : `fic.close();`

Les exemples précédents décrivent deux opérations de base, mais il existe de nombreuses opérations permettant de lire à partir d'une position, de lire un caractère à la fois, etc. et de même pour les opérations d'écriture.

2.5.3 Retour à la structure `World`

Ecrire une fonction `void init_world_from_file(World* word, string filename)` initialisant le monde pointé par `world` à partir du fichier dont le nom est donné par le paramètre `filename`.

Séance 4

Suite de la conception d'un *casse-brique* simple. Vous allez commencer par créer les fichiers `src/game.hpp`, `src/game.cpp`, modifier le fichier `src/main.cpp` ainsi que le fichier `Makefile` comme lors de la séance précédente. Dans un premier temps, le fichier `src/game.hpp` contiendra

```
#ifndef GAME_HPP
#define GAME_HPP

#include "world.hpp"
#include "window.hpp"

#endif
```

1 Evaluation orale

Durant cette séance, l'enseignant interrogera chaque binôme sur le résultat de chacun sur les séances 1 et 2 et sur le code réutilisable entre la séance 1-2 et le jeu du casse-briques.

2 Structure `Game`

Les données de notre jeu seront stockées dans une structure `Game` qui sera définie dans les fichiers `src/game.hpp` et `src/game.cpp`.

2.1 Déclaration de la structure

Créer une structure `Game` composée :

- d'un pointeur `world` de type `World*`;
- d'une couleur SDL pour chaque type de bloc. Vous pouvez, par exemple, associer la couleur noir au bloc `Empty` à l'aide de la commande

```
SDL_Color empty_color = {0, 0, 0, 255};
```

Créer la fonction `init_game(Game* game, string filename)` permettant d'initialiser la donnée `world` du jeu pointé par `game`. Vous utiliserez la fonction `init_world(World*, string filename)`.

2.2 Affichage du jeu

Créer une fonction `void display_game(Window* window, Game* game)` qui affiche le monde pointé par `game->world` dans la fenêtre pointée par `window`.

Testez votre fonction en affichant le jeu dont le monde est initialisé à l'aide du fichier `world.dat`.

Au fur et à mesure du développement du jeu, cette fonction sera modifiée pour inclure l'affichage d'autres éléments comme :

- la raquette;
- la balle;
- le score.

2.3 Raquette

Nous allons maintenant ajouter une raquette à notre jeu. La raquette sera caractérisée par les coordonnées de son centre et sa demi largeur. Ajouter les données suivantes à la structure `Game` :

- `int racket_x` pour la coordonnée x de la cellule de la grille contenant le centre de la raquette;
- `int racket_y` pour la coordonnée y de la cellule de la grille contenant le centre de la raquette;
- `int racket_half_width` pout la demi-largeur de la raquette.

Par exemple si `racket_half_width` vaut 3, la raquette occupera $2 \times 3 + 1$ cellules de la grille: la cellule du centre plus 3 cellules à droite et 3 cellules à gauche.

2.3.1 Initialisation de la raquette

Modifier la fonction `init_game` pour initialiser les donnée de la raquette de sorte que

- le centre de la raquette est au milieu de la deuxième ligne la plus basse;
- la demi largeur de la raquette soit 3.

2.3.2 Affichage de la raquette

Pour afficher la raquette il vous faut :

- Ajouter une donnée `racket_color` dans la structure `Game`;
- Modifier la fonction `display_game` en conséquence.

Pensez à faire des tests.

2.4 Balle

Nous allons maintenant ajouter la balle à notre jeu :

- ajouter deux entiers `ball_x` et `ball_y` pour les coordonnées de la cellule de la grille contenant la balle;
- ajouter deux entiers `ball_dx` et `ball_dy` pour les coordonnées du vecteur de déplacement de la balle;
- modifier la fonction `init_game` pour qu'elle initialise ces valeurs.

Modifier la fonction `init_game` pour initialiser la balle :

- au debut du jeu la balle se trouve sur la case juste au dessus du centre de la raquette;
- le vecteur de déplacement de la balle a pour coordonnées $(0, -1)$ correspondant à un déplacement vertical.

Modifier la fonction `display_game` pour afficher la balle. Pensez à définir la couleur de la balle dans la structure `Game`.

3 Boucle évenementielle

Maintenant que le monde, la raquette et la balle sont mis en place, nous pouvons commencer à animer notre jeu.

3.1 Gestion du clavier

Le contrôle du jeu se fera exclusivement au clavier :

- la touche `Q` quittera le jeu;
- la touche `R` fera un reset du jeu;
- les flèches gauche et droite permettront de contrôler la raquette;
- la touche `Espace` permettra de lancer / mettre en pause le jeu.

Nous rappelons que lors des différents tests, vous utiliserez le monde contenu dans le fichier `world.dat`.

Modifier les fichiers `game.hpp` et `game.cpp` en ajoutant une fonction `bool keyboard_event(Game* game)` lisant les évènements clavier et affichant (à l'aide d'un cout) l'action associée. Votre fonction retournera `true` si la touche `Q` a été enfoncée et `false` sinon.

Dans le fichier `main.cpp` créer une boucle évenementielle

1. lisant la demande la fermeture de la fenêtre;
2. appelant la fonction `keyboard_event` sur votre jeu;
3. affichant le jeu à l'aide de la commande `display_game`;
4. faisant une pause de 10ms avant le prochain pas de boucle.

Pensez à tester votre programme.

3.2 Déplacement de la raquette

Dans les fichiers `src/game.hpp` et `src/game.cpp`, écrire une fonction

```
move_racket(Game* game, int d)
```

qui déplace la raquette d'un bloc vers la droite si `d` vaut `1` et d'un bloc vers la gauche si `d` vaut `-1`.

Avant de faire le déplacement de la raquette, vous vérifierez que celui-ci est possible. Si le mouvement ne peut pas être effectué, vous ne déplacez pas la raquette.

3.3 Mouvement simple de la balle

Créer une fonction `move_ball(Game* game)` qui modifie les coordonnées de la balle en fonction de son vecteur de déplacement. Nous modifierons cette fonction plus tard pour que la balle rebondisse, casse des blocs, etc.

Dans la structure `Game` ajouter une variable entière `ball_speed` précisant la vitesse de la balle. La valeur de cette variable sera comprise 0 et 100.

Modifier la boucle évenementielle pour que la balle soit déplacée tous les $100 -$ `game.ball_speed` pas de boucle.

3.4 Statut du jeu

Votre jeu pourra être dans 5 statuts différents :

- `Begin` où la balle n'est pas encore lancée. Quand le jeu est dans ce statut, un déplacement de la raquette provoque le déplacement de la balle en conséquence pour que celle-ci reste sur la cellule de la grille juste au dessus du centre de la raquette;
- `Play` où la balle a été lancée, son mouvement est alors indépendant de celui de la raquette. Le mouvement de la balle sera alors effectué par la fonction `move_ball` que l'on définira ultérieurement;
- `Pause` où aucun mouvement de la raquette ni de la balle sont effectués;
- `Win` et `GameOver` lorsque la partie est finie. Aucun mouvement de la balle ou de la raquette ne sont effectués.

Créer un type `enum` Statut composé de ces cinq catégories de statuts. Vous ajouterez une variable `statut` de type `Statut` dans votre structure `Game`.

Le passage d'un statut à l'autre se fera à l'aide de la touche `Espace`.

Ecrire une fonction `change_statut` qui

- si `statut` vaut `Begin`, le modifie en `Play`
- si `statut` vaut `Play`, le modifie en `Pause`
- si `statut` vaut `Pause`, le modifie en `Play`

- si `statut` vaut `Win` ou `GameOver`, le modifie en `Begin`.

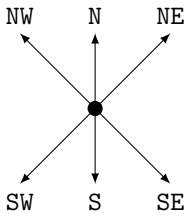
Modifier les fonctions `keyboard_event` et `move_racket` pour que votre jeu ait le comportement demandé en fonction de la variable `statut`.

Séance 5

Le but de cette séance est de finaliser le jeu du *casse-brique*. Certaines fonctionnalités bonus sont proposées à la fin de la séance.

1 Mouvement de la balle

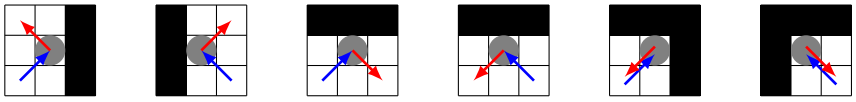
La balle se déplacera suivant l'une des directions suivantes :



Lorsque le passage du statut du jeu de `Begin` à `Play` le mouvement de la balle suit la direction `N`.

1.1 Rebond sur un bloc de type `Bord`

Les rêgles de rebond de le balle sur un bloc de type `Border` sont données par



La flèche bleue représente la direction actuelle de la balle tandis que la flèche rouge représente la prochaine direction. La principale différence entre ces différentes rêgles de rebond réside sur l'existence d'un bloc de type `Border` au dessus de la balle ou non.

1.2 Rebond sur la raquette

Pour le rebond sur la raquette nous proposons les rêgles suivantes :

- si la balle frappe le milieu de la raquette, elle repart dans la direction `N`;
- si la balle touche le bloc le plus à droite de la raquette, elle repart en dirction `NE`;
- si la balle touche le bloc le plus à gauche de la raquette, elle repart en dirction `NW`;
- si la balle touche un autre bloc de la raquette:
  - elle part en direction `N` si sa direction actuelle est `S`;
  - elle part en direction `NW` si sa direction actuelle est `SW`;
  - elle part en direction `NE` si sa direction actuelle est `SE`.

1.3 Destruction de bloc

Lorsque la balle touche un bloc de type `Type1` ou `Type2`, elle le détruit. Nous vous laissons imaginer les différentes rêgles de rebonds de la balle lorsqu'elle rencontre ce type de bloc. Vous pouvez vous inspirer du cas des blocs de type `Bord`.

2 Finalisation

2.1 Fin de partie

La partie se termine lorque le joueur a détruit tous les blocs ou que la balle est arrivée sur un bloc de type `Lose`. Modifier votre programme pour inclure ce comportement.

2.2 Score

Modifier la structure `Game` en ajoutant une variable `score` de type `int`. A chaque fois que la balle détruira un bloc, le score sera augmenté de 1 point. Apporter les modifications nécessaires pour inclure ce comportement.

Modifier la fonction `display_game` pour afficher le score tout en haut de la fenêtre de jeu.

3 Améliorations possibles

Si vous avez un jeu complètement fonctionnel et que vous souhaitez l'améliorer, nous vous proposons les améliorations suivantes :

- Changer l'affichage à base de rectangles de couleurs pour un affichage utilisant des sprites;
- Les blocs de type `Type2` nécessitent deux coups pour être détruits;
- Ajouter une sauvegarde / chargement de la partie en cours.

Séance 6

Dans les 4 séances à venir, vous allez concevoir le jeu qu'on nommera *Snaketris* combinant le jeu du *Snake* auquel vous rajouterez des règles s'apparentant au jeu du *Tétris*. Pour l'élaboration de ce jeu vous serez un peu moins guidés que pour le jeu du *casse-brique* quant aux fonctions à écrire, aux fichiers à créer, mais vous serez tenus de suivre rigoureusement le cadre imposé.

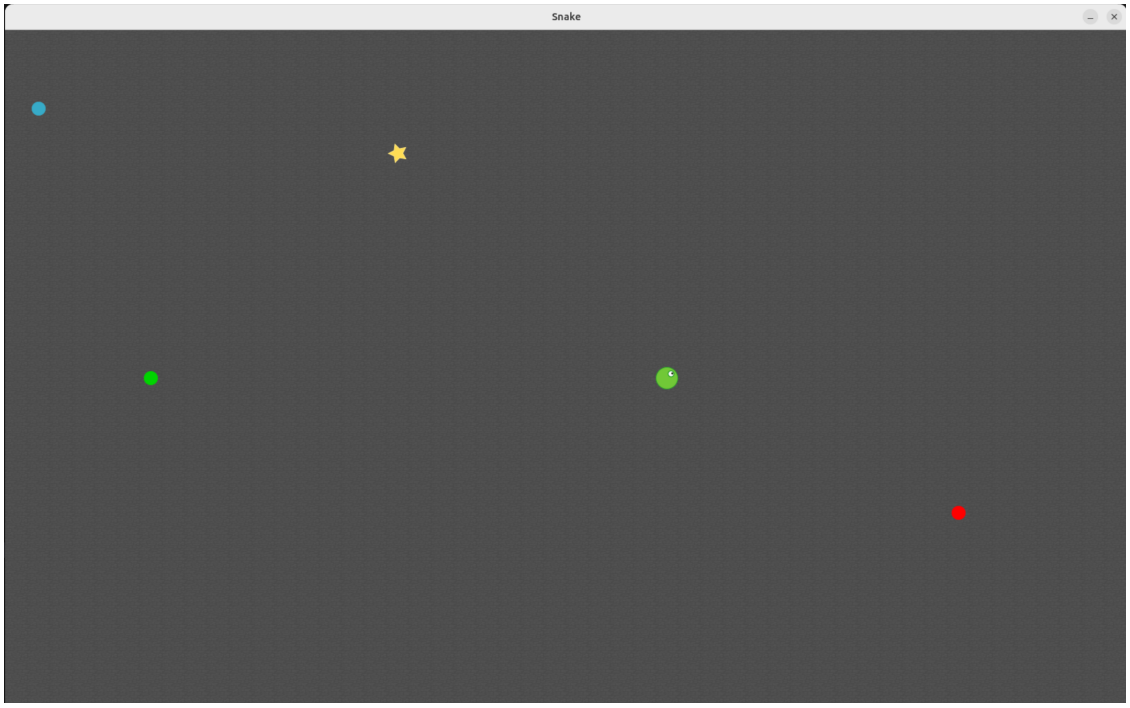


Figure 1: Monde au départ, Snaketris constitué d'une simple tête

1 Consignes

- Rassembler les structures et les fonctions d'un même thème dans un fichier *theme.hpp* avec le code des fonctions dans *theme.cpp*.
- Rassembler les objets de type *enum* dans un fichier *enum.hpp*.
- Respecter les consignes de codage (exemple : le snake représenté par une liste doublement chaînée).
- Mettre avant chaque fonction, une explication de la fonction en commentaire.

- Rajouter des commentaires dans les fonctions soit pour "aérer" le code (exemple : direction Nord, puis plus loin direction Sud, ...), soit pour expliquer une partie moins évidente. Ne pas expliquer chaque instruction.
- Rédiger **au fur et à mesure** un glossaire des fichiers créés et des fonctions qu'ils contiennent, en indiquant les interactions entre les fonctions (exemple : draw_game() appelle draw_world() et draw_snake()).

2 Jeu du Snaketris

Ce jeu consiste à déplacer un serpent dans un monde contenant de la nourriture et de lui faire manger le plus de nourriture possible. Les règles de base sont :

- le serpent se déplace tout seul dans une direction donnée au départ ;
- on peut le faire changer de direction par l'appui d'une touche du clavier, il ne peut pas faire demi-tour. Si le serpent avance vers l'Est (flèche droite) alors on peut le faire changer de direction pour qu'il se déplace vers le Nord (flèche haut) ou vers le Sud (flèche bas), mais il ne pourra pas se déplacer vers l'Ouest (flèche gauche) ;
- quand le serpent passe sur une cellule qui contient de la nourriture alors il grossit d'un anneau de ce type ;
- il pourra exister différents types de nourriture que vous pourrez rajouter au fil du jeu, au départ il en existera 4 :
  - nourriture rouge : il grossit d'un anneau rouge,
  - nourriture verte : il grossit d'un anneau vert,
  - nourriture bleue : il grossit d'un anneau bleu,
  - nourriture étoile : il se réduit à la façon du tétris. On choisira une règle très simple au départ qui consiste à supprimer dans une liste de 3 anneaux consécutifs contenant la même nourriture l'anneau du milieu ;
- toute la nourriture n'est pas placée au départ, on affiche 1 ou 2 nourritures et quand le serpent en mange une on en fait réapparaître une autre aléatoirement ;
- si le serpent touche un bord du monde ou qu'il se touche lui-même alors le jeu est terminé ;
- si le serpent possède 3 anneaux consécutifs de même couleur, alors l'anneau du milieu sera supprimé.

Le but est donc a minima de créer le monde, de visualiser les déplacements et les transformations du snake dans ce monde à l'aide des règles de base. Vous pourrez ensuite rajouter à l'envi des règles, des types de nourriture, des types de cellules, ...

Pour créer votre jeu vous suivrez le même processus que dans le *casse-brique*, pour la création du jeu, du monde, ...

3 Création du jeu

Dans un premier temps et comme pour le *casse-brique* la structure du jeu comprendra :

- un pointeur vers la fenêtre d'affichage ;
- un pointeur vers une variable représentant le monde du snake ;
- un pointeur vers une variable représentant le snake.

3.1 Fenêtre d'affichage

Pour cette structure vous reprendrez ce que vous avez déjà écrit dans les fichiers *window.hpp* et *window.cpp*.

3.2 Création du monde

Pour la structure *world* vous reprendrez le modèle du *world.hpp* du casse-brique, en adaptant le type des éléments de la grille. Ici les éléments décrivent de la nourriture, vous utiliserez une énumération pour les décrire. Il faudra prévoir des fonctions qui permettent d'initialiser le monde, d'y ajouter de la nourriture.

3.3 Affichage du monde

Avant de continuer et créer le snake, écrire une fonction *draw_world()* qui affiche un monde avec les 4 nourritures de base. Vous pourrez représenter les cellules contenant de la nourriture par des rectangles de la couleur de la nourriture, prendre la couleur jaune pour l'étoile.

4 Création du snake

La structure du snake va contenir:

- une tête avec son emplacement ;
- une direction de déplacement ;
- un pointeur vers le premier anneau entré qui correspondra à la queue du snake;
- un pointeur vers le dernier anneau entré qui correspondra à l'anneau situé juste derrière la tête.

Au départ le snake n'aura pas d'anneaux, sera situé au centre du monde, et aura une direction donnée.

4.1 Structure d'un anneau

La structure d'un anneau contiendra :

- les coordonnées de l'anneau dans le monde ;
- la valeur de la nourriture qui a permis sa création ;
- un pointeur vers l'anneau suivant ;
- un pointeur vers l'anneau précédent, afin de mettre à jour plus rapidement la chaîne.

4.2 Affichage du snake et déplacement

Avant de gérer le *corps* du snake composé des anneaux créés petit à petit, vous ferez un premier affichage du snake composé uniquement de la tête, il se déplacera suivant la direction initiale et changera de direction par l'appui des touches *flèches* du clavier.

Dans cet affichage vous commencerez donc la gestion des évènements.

Séance 7

Cette séance va être consacrée à la gestion du snake.

1 Création du corps

Chaque fois que la tête du snake passe sur une cellule contenant de la nourriture, un anneau va s'ajouter :

- au départ, le snake est constitué uniquement de la tête, donc lors de l'ajout d'un anneau, le pointeur vers le premier anneau aura la même valeur que le pointeur vers le dernier qui sera l'anneau rajouté ;
- lorsqu'il existe déjà un anneau, alors l'ajout d'un anneau se fera en tête : le pointeur vers le premier anneau pointera sur ce nouvel anneau, et le pointeur vers le dernier ne sera pas modifié.

Attention à initialiser l'anneau en fonction de l'emplacement de la tête, l'anneau prendra la place de la tête.

2 Déplacement des anneaux

Quand le serpent avance, la tête avance dans la direction donnée, le premier anneau prend la place de la tête, le 2e anneau prend la place du 1er, ... jusqu'au dernier qui prend la place de l'avant-dernier. Écrire une fonction permettant de déplacer tous les anneaux.

Appeler cette fonction dans la fonction écrite dans la séance 6 qui permet de déplacer le snake.

3 Gestion évènementielle

Compléter la boucle évènementielle afin de prendre en compte le passage de la tête sur une cellule du monde contenant de la nourriture.

Rajouter une fonction qui permet d'ajouter une nourriture dans une cellule du monde lorsque la tête du snake est passée sur une cellule contenant une nourriture.

4 Travail à rendre

Quand vous aurez écrit le code permettant d'afficher le monde, et de déplacer et faire grossir le snake dans ce monde, faites une archive de votre code et du rapport explicatif des fichiers et des fonctions et déposez-la dans le répertoire sur Moodle "Rendu_Snaketris_Bases".

Séance 8

Cette séance va être consacrée à l'ajout de textures pour un affichage plus agréable. Pour chaque type de cellules du monde, et pour chaque état de la tête du snake, vous pourrez "plaquer" une texture.

Vous trouverez des images dans le répertoire "img_snaketris" de Moodle, mais vous pourrez dessiner d'autres images à l'envi. Les images proposées sont décrites dans la section 3.

Pour plus de clarté, créer 2 fichiers *view.hpp* et *view.hpp*, dans lesquels vous déclarerez les textures utilisées pour le monde, et les textures utilisées pour chaque état de la tête du snake et de ses anneaux. Dans ces fichiers vous déplacerez et écrirez toutes les fonctions se rapportant à l'affichage du monde (exemple : les fonctions *draw_world()* et *draw_snake()* seront déplacées dans ce fichier).

1 Affichage du monde - Textures

Dans la fonction *draw_world()* qui affiche un monde avec les 4 nourritures de base, rajouter des textures permettant d'afficher les différents type de nourriture. Pour plus de clarté créer une structure *WorldView* qui va prendre les différentes textures à utiliser en fonction du type de cellules à afficher. Vous rajouterez à la structure du jeu un pointeur vers un objet de ce type.

2 Affichage du snake - Textures

Dans la fonction *draw_snake()* qui affiche le snake, rajouter des textures permettant d'afficher les différents états de la tête, ainsi que le type de nourriture à l'origine de chaque anneau. Pour plus de clarté créer une structure *SnakeView* qui va prendre les différentes textures à utiliser en fonction de l'état de la tête et des anneaux à afficher. Vous rajouterez à la structure du jeu un pointeur vers un objet de ce type.

3 Présentation des images proposées

La liste des images proposées au format png sont :

- *background.png* : MONDE, cellule sans nourriture ;
- *food_*.png* : MONDE, cellule avec nourriture "*" ;
- *food_star.png* : MONDE, cellule avec nourriture étoile ;
- *body_*.png* : SNAKE, anneau construit à partir de nourriture "*" ;
- *head_close_*.png* : SNAKE, tête du snake avec l'oeil qui change de position suivant la direction "*" de déplacement, bouche fermée (pas de nourriture à absorber), voir image 2 ;
- *head_open_*.png* : SNAKE, tête du snake avec l'oeil + bouche ouverte qui change de position suivant la direction "*" de déplacement, voir image 3 ;

Remarque: les images *food_blue.png* et *body_blue.png* n'ont comme différence que la taille du rond coloré, afin de remplir une cellule du monde quand on affiche le snake, et de rendre plus petite la nourriture dans la cellule.

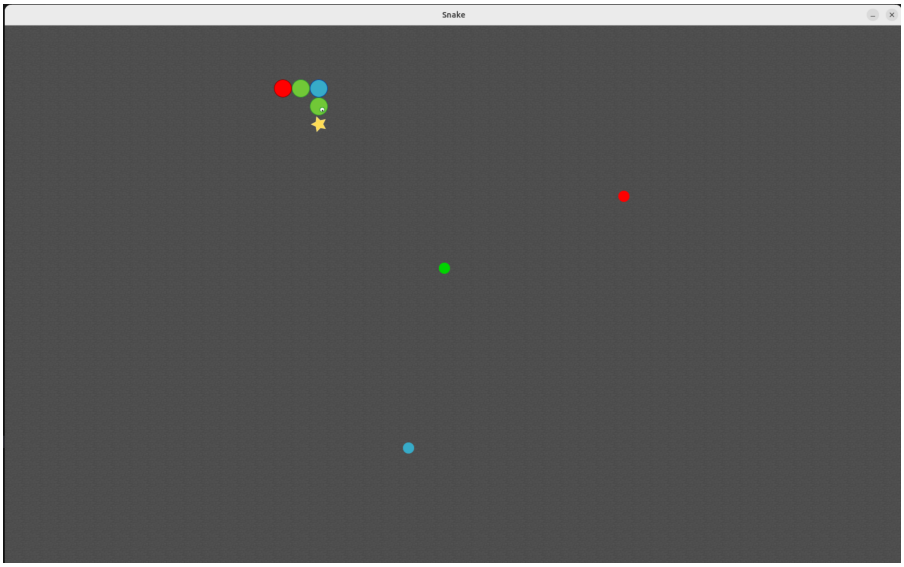


Figure 2: Snaketris : bouche fermée. On remarque que le snake regarde dans la direction de déplacement

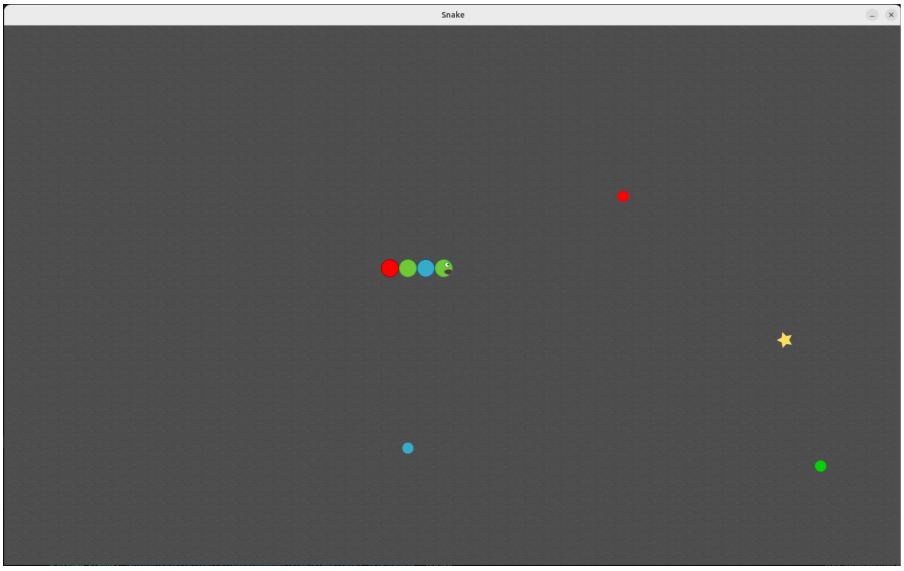


Figure 3: Snaketris : bouche ouverte, est sur une cellule nourriture. On remarque que le snake regarde dans la direction de déplacement

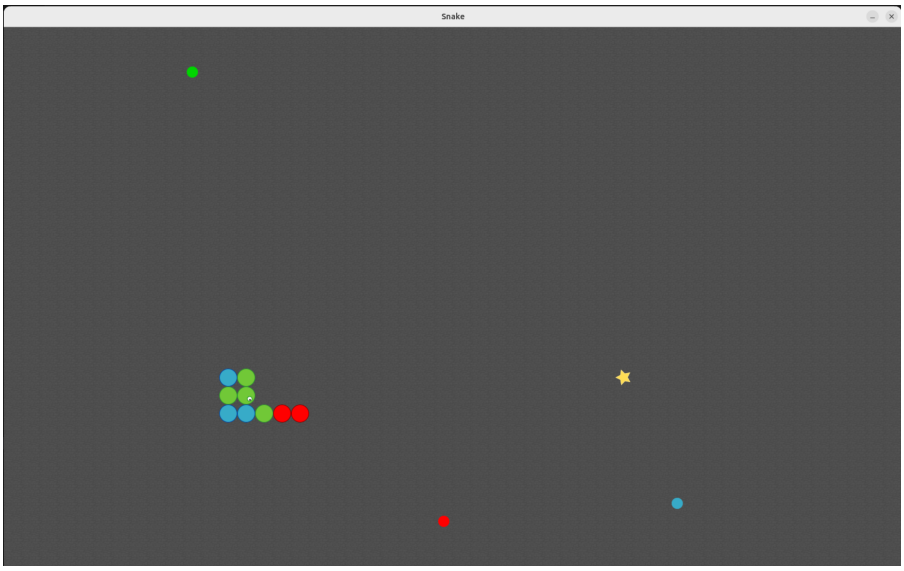


Figure 4: Snaketris : tête touche un anneau qui n'est pas le premier

4 Ajout règles de snake+tétris

Ajouter les 2 règles de base du snake qui consistent à arrêter le jeu :

- si le snake touche un bord du monde ;
- si la tête du snake touche un de ses anneaux qui n'est pas le premier, voir la figure 4

Ajouter ensuite la réponse du snake à l'ingestion d'une étoile. Lorsque la tête du snake passe sur une cellule contenant une étoile, alors si le corps du snake a 1 anneau entouré de 2 anneaux de même type (= ayant ingéré la même nourriture) alors l'anneau est supprimé. Faites attention à supprimer la place mémoire utilisée par cet anneau.

5 Travail à rendre

Quand vous aurez écrit le code de cette séance qui consiste à rajouter les règles du snake et du tétris et à plaquer des textures sur les types de nourriture, le type des anneaux et l'état de la tête, faites une archive de votre code et déposez-la dans le répertoire sur Moodle "Rendu_Snaketris_Textures".

Séance 9

Cette séance va être consacrée à l'ajout à l'envi d'options au monde et à la gestion du snake.

1 Imposés

Vous rajouterez la possibilité de :

1. sauvegarder le jeu en milieu de partie dans un fichier texte suite à l'appui de la touche "S". Le jeu pourra continuer ou non. Le nom du fichier sera passé en paramètre de la ligne d'exécution.
2. rajouter l'affichage d'un score qui pourra augmenter de +1 sur des nourritures de base, -1 chaque fois qu'un anneau est supprimé, +bonus quand une étoile est ingérée ...

2 Libres

Vous pourrez rajouter à l'envi des fonctionnalités. Comme par exemple la possibilité d'avoir des cellules dans le bord qui sont des passages secrets et qui permettent au snake de ne pas mourir mais de passer de l'autre côté de la grille (pas forcément en face).

Attention d'ajouter une seule fonctionnalité à la fois, de la coder, de la tester, de la décrire dans le rapport.

3 Travail à rendre

Quand vous aurez écrit le code qui rajoute de nouvelles fonctionnalités au jeu du snaketris développé dans les séances précédentes, faites une archive de votre code et du fichier texte qui détaille les ajouts proposés et déposez-la dans le répertoire sur Moodle "Rendu_Snaketris_Ajouts".