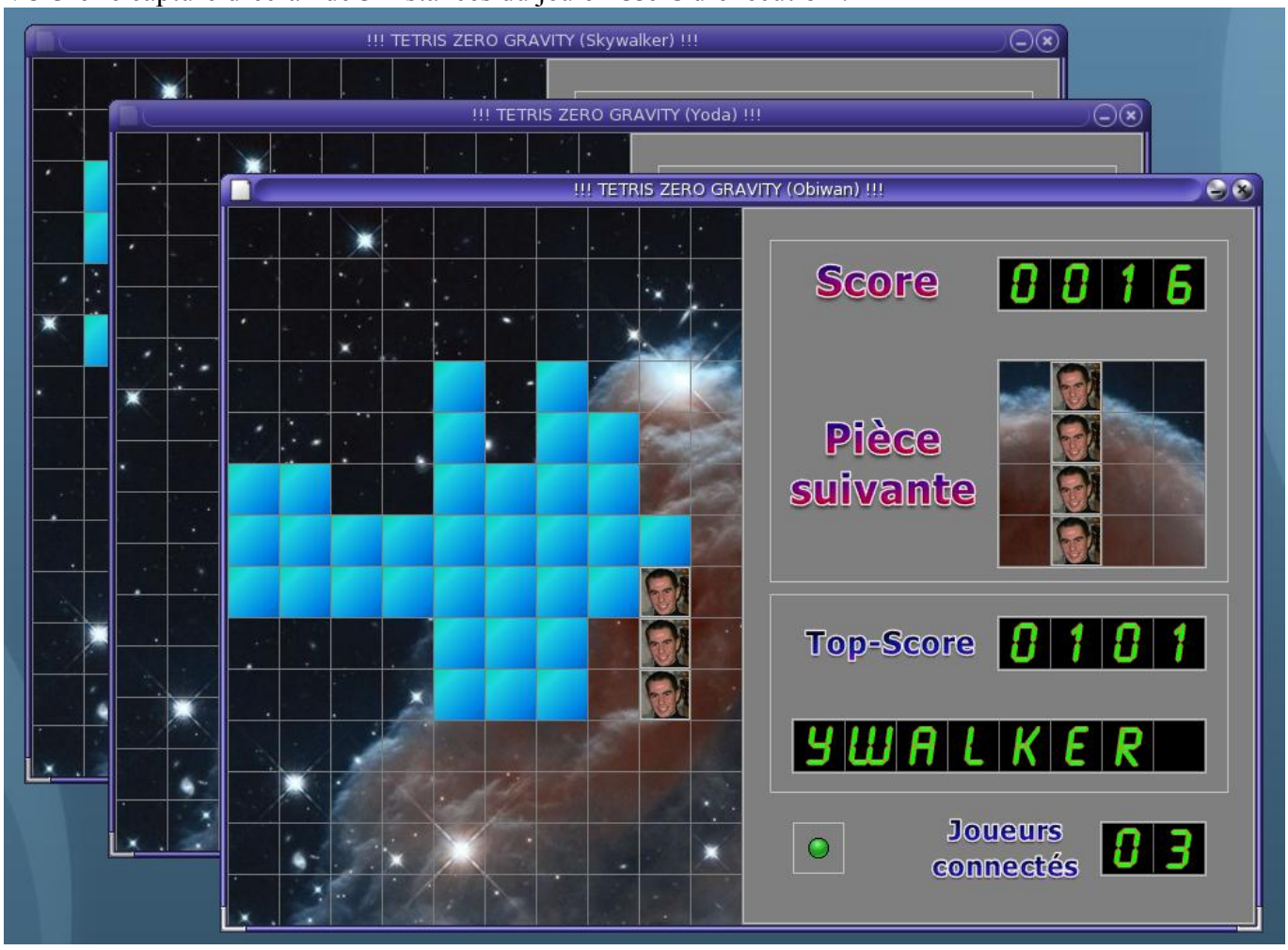
Laboratoire de Threads - Enoncé du dossier final
Année académique 2013-2014
Jeu de « Tetris Zero Gravity »

Il s'agit de créer un jeu du type « Tetris-Like » pour un joueur. Le principe fondamental reste identique à celui d'un Tetris normal. Il s'agit de poser des pièces de différentes formes sur une grille de jeu afin de former des lignes complètes, mais également dans ce cas-ci des colonnes complètes. Les pièces ne tombent pas du haut de l'écran (d'où le terme « Zero Gravity ») et ne peuvent pas subir de rotation mais peuvent être placées n'importe où sur la grille de jeu à la manière d'un puzzle, et cela à l'aide de la souris. Dès qu'une colonne ou une ligne est remplie, celle-ci disparaît et décale le reste des pièces vers le centre de la grille. La partie se termine quand le joueur est incapable de trouver un endroit libre sur la grille de jeu lui permettant de déposer la pièce suivante.
Une particularité supplémentaire est que l'application pourra se connecter à une petite application serveur (fournie) qui permettra d'afficher en direct le nombre de joueurs connectés, ainsi que le Top score actuel.

L'application devra être exécutée à partir d'un terminal console mais est gérée par une fenêtre graphique. C'est dans cette fenêtre que devra apparaître la grille de jeu, le score, la pièce suivante, le Top score, le détenteur actuel du Top score, et le nombre de joueurs actuellement connectés au serveur. Toutes les actions que pourra réaliser le joueur seront gérées par la souris.

Voici une capture d'écran de 3 instances du jeu en cours d'exécution :



Le Top score actuel est de 101 et est détenu par le joueur dont le pseudo est « Skywalker ». La zone contenant le nom du détenteur du Top score défile de la droite vers la gauche en permanence. La « led » verte indique que le joueur peut cliquer actuellement, mais nous y reviendrons...

Notez dès à présent que plusieurs éléments sont fournis dans le répertoire **/export/home/public/wagner/EnonceThread2014**, dont notamment

- **GrilleSDL** : librairie graphique qui permet de gérer une grille dans la fenêtre graphique à la manière d'un simple tableau à 2 dimensions. Elle permet de dessiner, dans une case déterminée de la grille, différents « sprites » (obtenus à partir d'images bitmap fournies)
- **images :** répertoire contenant toutes les images bitmap nécessaires à l'application : image de fond, têtes de vos professeurs préférés ☺, lettres, chiffres, …
- **Ressources :** Module permettant de charger les ressources graphiques de l'application, de définir un certain nombre de macros associées aux sprites propres à l'application et des fonctions permettant d'afficher des lettres et des chiffres dans la fenêtre graphique.

De plus, vous trouverez le fichier Tetris.c qui contient déjà les bases de votre application (dessin de la grille de jeu) et dans lequel vous verrez des exemples d'utilisation de la librairie GrilleSDL et du module Ressources. Vous ne devez donc en aucune façon programmer la moindre fonction qui a un lien avec la fenêtre graphique. Vous ne devrez accéder à la fenêtre de jeu que via les fonctions de la librairie GrilleSDL et du module Ressources. Vous devez donc vous concentrer uniquement sur la programmation des threads !

Les choses étant dites, venons-en aux détails de l'application… La grille de jeu sera représentée par un tableau à 2 dimensions (variable **tab**) défini en global, et comportant 14 lignes et 20 colonnes. Seules les 10 premières colonnes (celles de gauche) seront utilisées pour la partie jouable. Les 10 colonnes de droite serviront à l'affichage du score, de la pièce suivante, etc… Ce tableau contient des entiers dont le code (les macros sont déjà définies dans Tetris.c et Ressources.h) correspond à

- 0 : VIDE
- 300000 à 300006 : WAGNER, MERCENIER, …, codes des sprites correspondant aux têtes des professeurs
- 400000 : BRIQUE, code qui correspond à une case « bleue » c'est-à-dire correspondant à une brique/case d'une pièce précédente qui a été mise en place.

Chaque pièce de Tetris (L,T,J,…) est composée de 4 briques/cases (valeur par défaut pour chaque pièce). Les pièces devront être insérées dans la grille de jeu, brique par brique, une brique à chaque clic gauche de la souris.
Donc, le joueur utilisera la souris pour positionner les pièces dans la grille de jeu :

- Un clic gauche fera apparaître, sur une case vide, la tête du professeur correspondant à la pièce suivante à positionner sur la grille. A chaque clic gauche, une seule brique est insérée dans la grille de jeu. Une fois que le joueur aura inséré quatre (nombre de cases par pièce par défaut) cases dans la grille de jeu et que celles-ci correspondront à la pièce à insérer, les 4 cases « tête de prof » seront remplacées par des briques bleues précisant que la pièce a été correctement déposée et le joueur pourra alors s'attaquer à la pièce suivante. Sinon, les 4 « têtes de prof » disparaissent.
- Un clic droit fera disparaître toutes les cases actuellement insérées pour la pièce en cours, c'est-à-dire les cases « tête de prof » et non les briques bleues qui, elles, sont définitivement mises en place.

L'appui sur la croix de la fenêtre graphique permettra de fermer proprement l'application à n'importe quel moment.

Notez dès à présent que pour dessiner un sprite dans la grille de jeu, vous devrez utiliser les macros définies dans Ressources.h, ainsi que la fonction **DessineSprite** de GrilleSDL.h. Par exemple, pour dessiner la tête de Wagner à la ligne 2 et colonne 3 de la grille, vous devez faire appel à DessineSprite(2,3,WAGNER). En ce qui concerne les caractères (char) et les chiffres (int), vous devrez faire appel aux fonctions **DessineChiffre** et **DessineLettre** de Ressources.h.

Afin de réaliser cette application, il vous est demandé de suivre les étapes suivantes dans l'ordre et de **respecter les contraintes d'implémentation citées**, **même si elles ne vous paraissent pas les plus appropriées**. **Le schéma global de l'application est fourni à la dernière page de cet énoncé.**

Etape 1 : Création du threadDefileMessage et d'un premier mutex

Dans un premier temps, le thread principal va lancer le **threadDefileMessage** dont la tâche est de faire défiler en boucle un message dans la zone réservée au détenteur du Top score. Pour cela, vous allez déclarer les **variables globales** suivantes :

```
char* message;  // pointeur vers le message à faire défiler
int tailleMessage; // longueur du message
int indiceCourant;  // indice du premier caractère à afficher dans la zone graphique
```

La zone graphique réservée au détenteur du Top score ne contient que 8 cases (ligne 10, colonnes 11 à 18), il est donc impossible d'y afficher un message plus long, d'où la raison du défilement. La variable **indiceCourant** indique simplement l'indice du premier caractère à afficher. Par exemple, si **message** contient « Bienvenue dans Tetris Zero Gravity » et que indiceCourant vaut 5, la chaine affichée dans la zone graphique est « evenue d ». Le **threadDefileMessage** doit donc tourner en boucle et incrémenter indiceCourant toutes les 0,4 secondes sans oublier de le remettre à zéro quand il a atteint la fin du message.

On vous demande également de définir une fonction

```
void setMessage(const char *texte);
```

qui modifiera la variable message et la réallouera en fonction de texte. Remarquez que n'importe quel autre thread (voir plus loin) pourra exécuter cette fonction. Vous devez donc protéger l'accès aux variables globales message, tailleMessage et indiceCourant par un **mutexMessage**. Après avoir lancé le threadDefileMessage, le thread principal initialisera le message à « Bienvenue dans Tetris Zero Gravity ».

Etape 2 : Création du threadPiece

Dans un second temps, le thread principal va lancer le **threadPiece** dont le rôle est de générer la pièce suivante à placer dans la grille de jeu, puis ensuite d'attendre que le joueur ait inséré suffisamment de cases avant de vérifier la correspondance entre les cases insérées par le joueur et la pièce à insérer. Avant de donner les détails de ce que fait ce thread, regardons comment représenter une pièce.

Une pièce est composée de plusieurs cases (4 par défaut) et une case est représentée par la structure

```
typedef struct
{
  int ligne;
  int colonne;
} CASE;
```

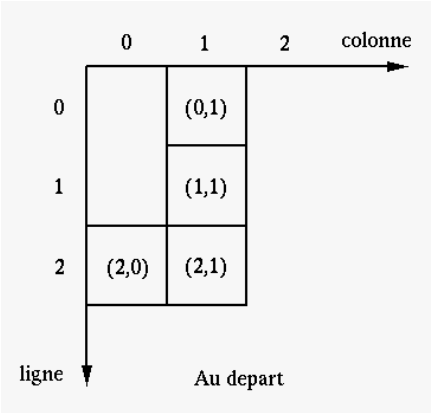
Une pièce est représentée par la structure

```
typedef struct
{
  CASE cases[NB_CASES];
  int  nbCases;
  int  professeur;
} PIECE;
```

où nbCases est le nombre de cases composant chaque pièce (par défaut 4 pour toutes les pièces). La variable professeur contient une des macros WAGNER, MERCENIER, … précisant quel professeur est associé à cette pièce. Les 7 pièces possibles vous sont fournies dans le vecteur

```
PIECE pieces[7] = { 0,0,0,1,1,0,1,1,4,WAGNER,       // carre
                    0,0,1,0,2,0,2,1,4,MERCENIER,    // L
                    0,1,1,1,2,0,2,1,4,VILVENS,      // J
                    0,0,0,1,1,1,1,2,4,DEFOOZ,       // Z
                    0,1,0,2,1,0,1,1,4,GERARD,       // S
                    0,0,0,1,0,2,1,1,4,CHARLET,      // T
                    0,0,0,1,0,2,0,3,4,MADANI };     // I
```

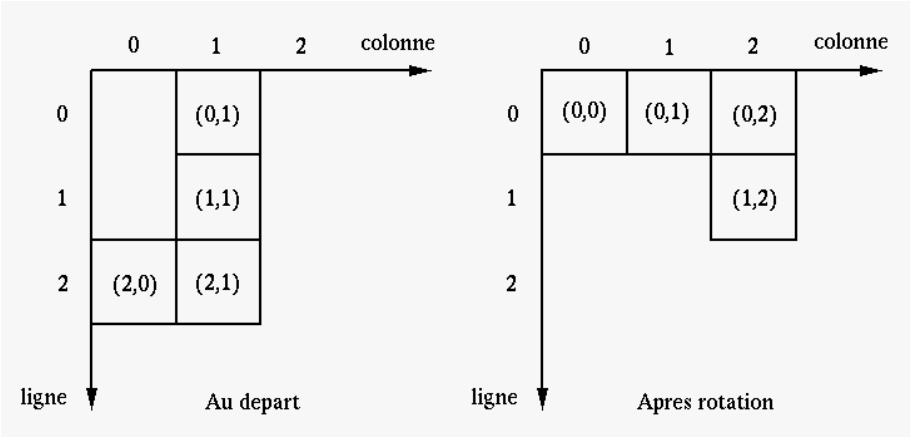
Prenons l'exemple de la pièce « J ». Ses cases sont (0,1), (1,1), (2,0), (2,1). Graphiquement, cela correspond à ceci :



Remarquez qu'une pièce ne peut jamais avoir de cases avec des coordonnées (ligne,colonne) négatives et que les cases de la pièce sont triées d'abord sur la ligne puis sur la colonne. Pour simplifier l'algorithme de vérification, il devra toujours en être ainsi. Avant de proposer une nouvelle pièce au joueur, le **threadPiece** fera une ou plusieurs rotations de la pièce de base. Pour cela vous devez utiliser les formules de rotation suivantes d'une case (L,C) autour du point (0,0) :

L (nouveau) = -C (ancienne)
C (nouveau) = L (ancienne)

Comme exemple, reprenons la pièce « J ». Après application de ces formules, ses cases sont (-1,0), (-1,1), (0,2), (-1,2). Des coordonnées sont négatives, il va falloir translater la pièce pour les rendre positive. Pour cela, on calcule la ligne minimum et la colonne minimum de chaque case : Lmin = -1 et Cmin = 0. Il suffit donc retirer Lmin et Cmin aux lignes et colonnes de chaque case. Les cases deviennent alors (0,0), (0,1), (1,2), (0,2). Finalement, afin de respecter la contrainte de tri des cases citées plus haut, on trie les cases selon la ligne puis la colonne, ce qui donne : (0,0), (0,1), (0,2), (1,2). Graphiquement, cela correspond à :



Il vous est donc fortement conseillé de concevoir une petite fonction

```
void RotationPiece(PIECE * pPiece);
```

qui réalise cette opération de rotation sur une pièce passée en paramètre.

Nous pouvons à présent revenir sur le détail de ce que fait le **threadPiece**. Une fois démarré, celui-ci

1. choisit une pièce au hasard parmi celles fournies dans le vecteur pieces et la stocke dans une variable globale **pieceEnCours** (qui représente la pièce suivante à insérer),
2. fait subir à **pieceEnCours** un nombre aléatoire de rotations (0,1,2 ou 3) avant de la dessiner dans la zone graphique qui lui est réservée (lignes 3 à 6, colonnes 15 à 18),
3. se met en attente que le joueur ait inséré suffisamment de cases (voir plus loin),
4. vérifie la correspondance entre les cases insérées par le joueur et la **pieceEnCours** et agira en conséquence (voir étape 3). Selon le cas, il remonte au point 1 (correspondance OK) ou au point 3 (correspondance KO).

Afin de gérer les cases insérées par le joueur, nous allons utiliser les variables globales suivantes :

```
CASE casesInserees[NB_CASES];  // cases insérées par le joueur
int  nbCasesInserees;  // nombre de cases actuellement insérées par le joueur.
```

Donc, pour le point 3, le **threadPiece** va se mettre en attente (utilisation de **pthread_cond_wait**) de la réalisation de la condition (à l'aide d'un mutex **mutexCasesInserees** et d'une variable de condition **condCasesInserees**) suivante :

« Tant que (nbCasesInserees < pieceEnCours.nbCases), j'attends… »

Reste maintenant à permettre au joueur d'insérer des cases… Nous y venons.

Etape 3 : Création du threadEvent

Le thread principal va lancer le **threadEvent** dont le rôle est de gérer les événements provenant de la souris. Pour cela, le threadEvent va se mettre en attente d'un événement provenant de la fenêtre graphique. Pour récupérer un de ces événements, le threadEvent utilise la fonction **ReadEvent()** de la librairie GrilleSDL. Ces événements sont du type « souris », « clavier » ou « croix de la fenêtre ». Les événements du type « clavier » devront être ignorés. Dans le cas « croix de fenêtre », le threadEvent fermera la fenêtre graphique et terminera proprement le processus.

Le **threadEvent** attend donc deux types d'événements provenant de la souris :

- Un clic gauche (event.type == CLIC_GAUCHE) : dans ce cas, le threadEvent récupère la ligne L et la colonne C de la cases cliquée dans la grille de jeu. Si tab[L][C] est égal à VIDE, la case est libre, et le threadEvent insère la valeur pieceEnCours.professeur dans la case correspondante du tableau tab et dessine la tête du professeur correspondant dans la grille de jeu (utilisation de **DessineSprite**). De plus, il insère la case cliquée dans la variable globale **casesInserees** et incrémente **nbCasesInserees** de 1. Il réveille alors le threadPiece (utilisation de **pthread_cond_signal**). Dans le cas où tab[L][C] n'est pas vide et est donc occupé par une brique bleue ou une tête de professeur, rien ne se passe.

- Un clic droit (event.type == CLIC_DROIT) : dans ce cas, le threadEvent efface les cases dernièrement insérées par le joueur de la grille de jeu (utilisation de **EffaceCarre()** de GrilleSDL), remet les cases correspondantes de tab à VIDE et remet **nbCasesInserees** à 0.

Retour sur le threadPiece :

Revenons à présent sur le réveil du threadPiece, qui correspond au fait que le nombre de cases insérées par le joueur est égal à pieceEnCours.nbCases. Il peut à présent vérifier la correspondance entre les cases insérées par le joueur et les cases de la pièce en cours. Le plus simple pour cela est de

- trier les cases insérées selon le même critère qu'utilisé pour la pieceEnCours,
- calculer la ligne minimum Lmin et la colonne minimum Cmin de toutes les cases insérées, ce qui permet de translater toutes les cases insérées à « l'origine (0,0) » en soustrayant Lmin des lignes et Cmin des colonnes de toutes les cases insérées.
- La vérification de la correspondance entre les cases insérées et les cases de la pièce en cours devient alors évidente. Par exemple, supposons que le joueur ait cliqué sur les cases (8,4), (8,6), (9,6) et (8,5). Après tri, ces cases deviennent (8,4), (8,5), (8,6) et (9,6). Le calcul des minima fournissent Lmin=8 et Cmin=4. Après soustraction de Lmin et Cmin, nous obtenons (0,0), (0,1), (0,2) et (1,2). Il suffit alors de comparer case par case avec la **pieceEnCours**. On observe ici qu'il y a correspondance entre les cases insérées et la pieceEnCours (à condition que celle-ci corresponde à la pièce « J » ayant subi une rotation, voir plus haut).

Après vérification,

- S'il y a correspondance, le threadPiece remplace dans tab les têtes des professeurs par la macro BRIQUE, signifiant que la pieceEnCours a bien été mise en place. De plus, dans la fenêtre graphique, il remplace les têtes de professeur par des cases bleues (macro BRIQUE). Ensuite, il remet **nbCasesInserees** à zéro et remonte dans sa boucle pour générer une nouvelle pièce à insérer.
- S'il n'y a pas correspondance, le threadPiece remplace dans tab les têtes de professeurs par VIDE. De plus, dans la fenêtre graphique, il efface les têtes de professeur (utilisation de **EffaceCarre()**) et remet **nbCasesInserees** à zéro. Il se remet alors en attente sur la variable de condition **condCasesInserees** (la pieceEnCours n'ayant pas été mise en place, elle reste donc la même).

Jusqu'à présent, le joueur peut positionner les pièces dans la grille de jeu mais les lignes et les colonnes complètes ne disparaissent pas et il n'y a toujours pas de score. On y vient…

Etape 4 : Création du threadScore

A partir du thread principal, lancer le **threadScore** dont le rôle est d'afficher le **score** en haut à droite dans la zone graphique correspondante.

Une fois lancé, le threadScore entrera dans une boucle dans laquelle il se mettra tout d'abord sur l'attente (via un **mutexScore** et une variable de condition **condScore**) sur la réalisation de l'événement suivant

« Tant que (!MAJScore), j'attends… »

En d'autres mots, cela signifie, que tant qu'il n'y a pas de mise à jour de la **variable globale score**, le threadScore attend. **MAJScore** est une **variable globale** booléenne (donc char pour cette application) reflétant que le score a été mise à jour par un autre thread. Les 2 variables globales **score** et **MAJScore** sont donc protégées par le même **mutexScore**.

Une fois réveillé, le threadScore doit simplement afficher la valeur de **score** (variable de type int) dans les cases (1,18), (1,19), (1,20) et (1,21) de la fenêtre graphique. Ensuite, il doit remettre **MAJScore** à 0 avant de remonter dans sa boucle.

La question à présent est de savoir qui va réveiller le threadScore. Donc…

Retour sur le threadPiece :

A chaque fois que le threadPiece détecte une correspondance entre les cases insérées par le joueur et la **pieceEnCours**, celle-ci est donc mise en place définitivement dans la grille de jeu. Le threadPiece doit alors incrémenter le **score** de 1 et réveiller le threadScore (**pthread_cond_signal**).

On peut donc à présent, poser les pièces sur la grille de jeu, ainsi que gagner des points au score, mais les lignes et les colonnes complètes ne disparaissent toujours pas. On y vient…

Etape 5 : Création des threadCases

A partir du thread principal, lancer les **threadCases** dont le rôle est d'analyser/vérifier si la ligne et la colonne sur laquelle il se trouve est complète, c'est-à-dire remplie de brique. Il faut donc autant de threadCases qu'il y a de cases dans la grille de jeu, c'est-à-dire 14 x 10 = 140. Les identifiants de chaque threadCase seront stockés dans un tableau global de pthread_t :

```
pthread_t tabThreadCase[14][10];
```

Chaque **threadCase** réalisant le même travail mais sur une case qui lui est bien spécifique, chaque threadCase disposera d'une **variable spécifique du type CASE**. Dès lors, dans sa boucle (en fait une double boucle sur L=0…13 et C=0…9) de création des threadCases, le **thread principal**

- alloue dynamiquement une structure CASE qu'il initialisera avec ligne=L et colonne=C,
- crée le threadCase d'identifiant tabThreadCase[L][C] en lui passant en paramètre la structure qu'il vient d'allouer.

Une fois démarré, chaque **threadCase**

- met la variable CASE reçue en paramètre dans sa zone spécifique (utilisation de **pthread_setspecific**),
- entre dans une boucle dans laquelle il attend simplement la réception du signal SIGUSR1 (utilisation de **pause()**) lui demandant d'analyser sa ligne et sa colonne. Le signal SIGUSR1 lui sera envoyé par le threadPiece (voir plus loin).

Plusieurs threadCases devront faire leurs analyses simultanément. Les résultats de leurs analyses seront stockés dans les **variables globales** suivantes

```
int lignesCompletes[4];
int nbLignesCompletes;
int colonnesCompletes[4];
int nbColonnesCompletes;
int nbAnalyses;
```

L'accès mutuellement exclusif à ces variables globales sera assuré par le **mutexAnalyse**.

A la réception de SIGUSR1, un **threadCase** entre dans un **handlerSIGUSR1** dans lequel :

- il récupère sa variable spécifique (utilisation de **pthread_getspecific**), et donc la ligne L et la colonne C de la case à laquelle il est associé,
- analyse la ligne L dans le tableau tab. Si elle est complète, c'est-à-dire remplie de BRIQUE, il insère L dans le vecteur **lignesCompletes** et incrémente **nbLignesCompletes** de 1, mais cela **à condition que L ne soit pas déjà présent dans le vecteur lignesCompletes**. En effet, un autre threadCase aurait déjà pu traiter cette ligne. En cas de ligne complète, les briques « entrent en fusion ☺ », afin de montrer qu'elles vont bientôt disparaître. Pour cela, dans la fenêtre graphique, le threadCase remplace le sprite BRIQUE par le sprite FUSION pour chaque case de la ligne complète.
- analyse la colonne C dans le tableau tab. Si elle est complète, c'est-à-dire remplie de BRIQUE, il insère C dans le vecteur **colonnesCompletes** et incrémente **nbColonnesCompletes** de 1, mais cela **à condition que C ne soit pas déjà présent dans le vecteur colonnesCompletes**. En effet, un autre threadCase aurait déjà pu traiter cette colonne. En cas de colonne complète, les briques « entrent en fusion ☺ », afin de montrer qu'elles vont bientôt disparaître. Pour cela, dans la fenêtre graphique, le threadCase remplace le sprite BRIQUE par le sprite FUSION pour chaque case de la colonne complète.
- incrémente **nbAnalyses** de 1, signifiant qu'il a terminé son analyse. Il réveillera alors (utilisation de **pthread_cond_signal**) le **threadGravite** (voir étape 6).

Vous remarquez donc que les threadCases ne font aucune modification du tableau tab, ce n'est pas leur rôle de supprimer les colonnes et les lignes complètes (qui sont juste « entrées en fusion ☺ »), ainsi que de décaler les lignes et les colonnes restantes, ce sera le rôle du threadGravite (voir étape 6).

Retour sur le threadPiece :

Dès que le threadPiece détecte une correspondance entre les pièces insérées par le joueur et la pieceEnCours, on sait qu'il incrémente le score 1. En plus, à partir de maintenant, il enverra le signal SIGUSR1 à chaque threadCase correspondant aux cases de la pièce qui vient d'être correctement insérée (utilisation de **pthread_kill**, leur pthread_t sont connus dans le tableau **tabThreadCases**), afin de les réveiller pour qu'ils analysent les lignes et les colonnes sur lesquelles la pièce a été insérée.

Etape 6 : Création du threadGravite

A partir du thread principal, lancer le **threadGravite** dont le rôle est de supprimer les lignes et les colonnes en fusion (complètes) et de décaler le reste des briques vers le « centre de gravité », c'est-à-dire le centre de la grille de jeu. Dès lors, une fois lancé, il entre dans une boucle dans laquelle il commence par attendre (via le **mutexAnalyse** et une variable de condition **condAnalyse**) sur la réalisation de l'événement suivant

« Tant que (nbAnalyses < pieceEnCours.nbCases), j'attends… »

En d'autres mots, cela signifie, qu'il attend que tous les threadCases concernés (il y en a autant qu'il y a de cases dans la **pieceEnCours**).

Une fois son attente terminée, le **threadGravite**

- regarde s'il y a des lignes et/ou des colonnes en fusion (complètes). Si il n'y a aucune ligne ni aucune colonne complète, il remet simplement **nbAnalyses** à 0, et remonte dans sa boucle pour se remettre en attente sur sa variable de condition.

- Dans le cas où il y a au moins une ligne et/ou une colonne en fusion, il commence par attendre 2 secondes (utilisation de **nanosleep**) afin de donner au joueur le temps d'observer les lignes et colonnes en fusion avant qu'elles ne disparaissent. Il peut ensuite récupérer les résultats d'analyse des threadCases et commencer son travail de suppression et de décalage vers le centre de gravité. Pour cela, **4 cas** se présentent :
    - Une colonne complète d'indice C=0,1,2,3 ou 4 provoque le décalage de toutes les colonnes d'indice inférieur à C vers la droite, ce qui a pour effet d'écraser la colonne complète. La colonne d'indice 0 est alors remplie de VIDE. Vous devez donc faire les décalages dans le tableau tab et mettre à jour correctement l'affichage dans la fenêtre graphique. Pour éviter l'oubli de suppression de colonnes complètes (dans le cas où il y a plusieurs colonnes complètes en même temps), vous devez commencer par la suppression des colonnes d'indice **minimum**.
    - Une colonne complète d'indice C=5,6,7,8 ou 9 provoque le décalage de toutes les colonnes d'indice supérieur à C vers la gauche, ce qui a pour effet d'écraser la colonne complète. La colonne d'indice 9 est alors remplie de VIDE. Vous devez donc faire les décalages dans le tableau tab et mettre à jour correctement l'affichage dans la fenêtre graphique. Pour éviter l'oubli de suppression de colonnes complètes (dans le cas où il y a plusieurs colonnes complètes en même temps), vous devez commencer par la suppression des colonnes d'indice **maximum**.
    - Une ligne complète d'indice L=0,1,2,3,4,5 ou 6 provoque le décalage de toutes les lignes d'indice inférieur à L vers le bas, ce qui a pour effet d'écraser la ligne complète. La ligne d'indice 0 est alors remplie de VIDE. Vous devez donc faire les décalages dans le tableau tab et mettre à jour correctement l'affichage dans la fenêtre graphique. Pour éviter l'oubli de suppression de lignes complètes (dans le cas où il y a plusieurs lignes complètes en même temps), vous devez commencer par la suppression des lignes d'indice **minimum**.
    - Une ligne complète d'indice L=7,8,9,10,11,12 ou 13 provoque le décalage de toutes les lignes d'indice supérieur à L vers le haut, ce qui a pour effet d'écraser la ligne complète. La ligne d'indice 13 est alors remplie de VIDE. Vous devez donc faire les décalages dans le tableau tab et mettre à jour correctement l'affichage dans la fenêtre graphique. Pour éviter l'oubli de suppression de lignes complètes (dans le cas où il y a plusieurs lignes complètes en même temps), vous devez commencer par la suppression des lignes d'indice **maximum**.
    - Dans tous les cas, afin de donner un « effet graphique dynamique » lors de la suppression de ligne/colonne, le **threadGravite** attendra 0,5 seconde (utilisation de **nanosleep**) après la suppression d'une ligne/colonne. Une fois son travail de suppression/décalage terminé, le threadGravite remet **nbAnalyses** à 0 et se remet en attente sur sa variable de condition.

Enfin, pour chaque ligne/colonne complète supprimée, le **threadGravite** incrémentera le **score** de 5 et réveillera le threadScore.

Bon… Les lignes et colonnes complètes disparaissent permettant, en plus, d'augmenter le score du joueur, mais que se passe-t-il lorsque le joueur n'a plus de place pour insérer la nouvelle pieceEnCours dans la grille de jeu ? La partie doit se terminer, nous y venons…

Etape 7 : Création du threadFinPartie

A partir du thread principal, lancer le **threadFinPartie** dont le rôle est de vérifier que la nouvelle **pieceEnCours** générée par le threadPiece peut trouver une place dans la grille de jeu. Pour cela, dès qu'il a démarré, il entre dans une boucle dans laquelle

- il attend le signal SIGUSR2 (utilisation de **pause()**) lui demandant de commencer sa vérification.
- Dès réception de SIGUSR2, il balaie tout le tableau tab et teste la possibilité d'y insérer la **pieceEnCours**, en vérifiant les cases vides correspondantes. Dès qu'il trouve une place possible, il arrête son balayage et se remet en attente sur son **pause()**. Dans le cas où il ne trouve aucune place disponible, le **threadFinPartie** se termine par un **pthread_exit**, ce qui permettra au thread principal de réagir en conséquence (voir étape 9).

La question maintenant est de savoir qui va envoyer le signal SIGUSR2 au threadFinPartie. Dès qu'une pièce a été insérée, il se peut qu'il n'y ait plus de place pour la suivante. Mais il se peut aussi que le threadGravite ait réalisé son travail. Le plus simple est que ce soit le **threadGravite** qui envoie le signal SIGUSR2 au **threadFinPartie**, car quoi qu'il arrive, il est réveillé à chaque fois qu'une pièce a été insérée (que des lignes/colonnes aient été supprimées ou pas).

Retour sur le threadGravite :

Donc, à chaque fois que le **threadGravite** a terminé son travail de suppression/décalage (qu'il ait supprimé des lignes/colonnes ou non), il envoie (utilisation de **pthread_kill**) le signal SIGUSR2 au threadFinPartie.

Etape 8 : Mise en place d'une synchronisation générale du traitement

1) Synchronisation du traitement global :

Vous remarquerez que les threads **threadEvent**, **threadPiece**, **threadCases**, **threadGravite** et **threadFinPartie** travaillent tous à la suite l'un de l'autre (un peu dans le style du modèle pipeline des notes de cours + voir schéma général en fin d'énoncé). Mais le souci ici est que le threadEvent ne peut pas laisser le joueur insérer une nouvelle case tant que toute la chaine de traitement (du threadPiece qui détecte une correspondance jusqu'au threadFinPartie qui teste la possibilité d'insertion de la nouvelle pieceEnCours) n'a pas fini son travail.

Pour cela, on vous demande d'utiliser une **variable globale** booléenne (type char ici) **traitementEnCours** protégée par le **mutexTraitement**, et qui sera mise à 1 pendant tout le traitement (ce qui empêchera le joueur d'insérer une nouvelle case tant que le traitement n'est pas terminé) et mise à 0 lorsque le joueur pourra insérer une nouvelle case. Dès lors,

Retour sur le threadEvent :

Dès que le **threadEvent** se rend compte qu'il a inséré suffisamment de cases (dans **casesInserees**), il met la variable globale **traitementEnCours** à 1. Dès lors, à partir de maintenant, dès que le joueur cliquera sur une case de la grille de jeu, le threadEvent vérifiera la variable traitementEnCours avant de faire quoi que ce soit.

Graphiquement, tant que le traitement n'est pas en cours, et donc que le joueur peut insérer une case, la « led » située à la case (12,11) de la fenêtre graphique est **verte** (macro VOYANT_VERT), signifiant que le joueur peut insérer une nouvelle case. **Tant que le traitement est en cours, la led doit être bleue** (macro VOYANT_BLEU). Si le joueur tente de cliquer sur une case déjà occupée (par une brique ou un professeur) ou même sur une case vide alors que le traitement est en cours, la led doit passer au **rouge**

(macro VOYANT_ROUGE) pendant 0,25 seconde (utilisation de **nanosleep**). De plus, quoiqu'il arrive, si le joueur clique dans la zone droite de la fenêtre graphique, la led passe au **rouge** pendant 0,25 seconde, avant de reprendre sa couleur précédente (**verte** ou **bleue**).

Retour sur le threadPiece :

Après sa vérification de correspondance entre les cases insérées par le joueur et les cases de la pièce en cours, on sait déjà que le **threadPiece** remet la variable nbCasesInserees à 0 **en cas de non correspondance**. Il devra en plus maintenant remettre la variable **traitementEnCours** à 0 et remettre la led au vert. En cas de correspondance, le traitement se poursuit via les threadCases et la led reste bleue.

Retour sur le threadFinPartie :

Une fois sa vérification terminée, le **threadFinPartie** remettra la variable **traitementEnCours** à 0, et remettra la led au vert. Le joueur pourra alors à nouveau insérer des cases dans la grille de jeu.

2) Synchronisation sur la pieceEnCours :

Une dernière synchronisation importante ne doit pas être négligée. Après avoir détecté une correspondance, le **threadPiece** envoie SIGUSR1 aux threadCases correspondants puis génère une nouvelle pièce. Il ne faudrait pas que le **threadFinPartie** fasse son travail de vérification avant que le threadPiece n'ait eu le temps de générer sa nouvelle pièce. Vous devez donc protéger l'accès mutuellement exclusif à **pieceEnCours** à l'aide du **mutexPieceEnCours**.

Etape 9 : Synchronisation du thread principal et fin de partie

Après le lancement de tous l[illegible]reads, le **thread principal** [illegible]ynchronisera sur la fin du **threadFinPartie** (utilisation [illegible]**d_join**). Une fois celui[illegible] le thread principal doit, pour terminer proprement le[illegible]

- tuer le **threadEvent** ([illegible]**d_canc**[illegible] joueur d'insérer des cases inutilement alors que la p[illegible]
- tuer l'ensemble des **thread**[illegible]) qui pourront alors passer par leur fonction « destructeur » [illegible]à **pthread_key_create**) qui leur permettra de libérer (**free**) leur va[illegible]à qu'ils se soient tous terminés avant de passer au point suivant.
- attendre un événement de type [illegible]**dEvent**) avant de fermer la fenêtre graphique.
- tuer le **ThreadDefileMess**[illegible]Celui-ci doit d'abord passer par une fonction de termina[illegible]**éa**[illegible]**thread_cleanup_pop**) dans laquelle il libère la va[illegible]e dynam[illegible]
- terminer le proces[illegible]

Le jeu est à présent complètem[illegible]. Reste maintenant à géré[illegible]ore et le nombre de joueurs connectés.

Etape 10 : Création du threadJoueursConnectes

A partir du thread principal, lancer le **threadJoueursConnectes** dont le rôle est d'afficher en « temps réel » le nombre de joueurs connectés sur le serveur. Le programme Serveur.c vous est fourni. Celui-ci connait en permanence le nombre de joueurs connectés, ainsi que le TopScore, le login (propriétaire du processus Tetris correspondant) et le pseudo du détenteur du TopScore. Les infos du TopScore sont stockées dans un fichier entièrement gérés par le serveur. Vous ne devez en aucun cas modifier le code du serveur. Il est complètement fonctionnel et enverra différents signaux aux processus qui s'y seront connectés. Celui-ci gère ses propres ressources (file de message et mémoire partagée), vous ne devez donc en aucun les gérer vous-même. Pour lancer le serveur, il suffit de lui passer en paramètre une clé (key_t) unique. Pour l'arrêter, il suffira de lui envoyer un SIGINT.

Pour interagir avec le serveur, on vous fournit le module **ClientTetris** (voir ClientTetris.h) qui est également totalement fonctionnel (vous ne devez donc pas le modifier) et qui présente, en ce qui nous concerne actuellement, les fonctions suivantes :

- `int ConnectionServeur(key_t cle,const char* pseudo)` : fonction prenant en paramètre la clé du serveur, ainsi que le pseudo du joueur. La fonction retourne 0 en cas de connexion réussie et une valeur négative en cas d'erreur. Cette fonction n'est absolument pas bloquante. Une connexion non réussie au serveur ne peut pas empêcher le jeu de se lancer. Vous jouez simplement « hors ligne ». Remarquez que la fonction transmet également, de manière implicite, le « login » du joueur, c'est-à-dire le propriétaire du processus Tetris qui appelle cette fonction.
- `int DeconnectionServeur(key_t cle)` : fonction permettant de se déconnecter du serveur dont la clé est passée en paramètre.
- `int GetNbJoueursConnectes(key_t cle)` : fonction permettant d'obtenir le nombre de joueurs connectés au moment de l'appel de la fonction. Un retour positif de la fonction correspond au nombre de joueurs connectés. En cas d'erreur (par exemple, si vous ne vous êtes pas connecté au serveur), le retour est négatif.

Remarquez qu'à partir du moment où vous serez connecté au serveur, celui-ci va régulièrement envoyer les signaux SIGHUP et SIGQUIT à votre processus Tetris. Il est donc impératif d'implémenter correctement un **masque de signal** adéquat dans chaque thread (utilisation de **sigprocmask**).

Venons-en au **threadJoueursConnectes**. Une fois démarré, celui-ci entre dans une boucle dans laquelle

- il se met en attente sur le signal SIGHUP (utilisation de **pause()**). **Seul le threadJoueurConnectes pourra interpréter le signal SIGHUP (vous devez donc masquer correctement les signaux dans les threads)**. Le signal SIGHUP est envoyé par le serveur à chaque fois qu'un joueur se connecte ou se déconnecte.
- Une fois sa pause débloquée, il récupère le nombre de joueurs connectés (utilisation de **GetNbJoueursConnectes**) et affiche cette valeur dans les cases (12,17) et (12,18) de la fenêtre graphique.
- remonte dans sa boucle pour se remettre en attente du signal SIGHUP.

Retour sur le thread principal :

Votre application Tetris devra prendre en paramètre (argc, argv) la clé du serveur sur lequel elle veut se connecter, ainsi que le pseudo du joueur. De plus, le thread principal

- doit se connecter au serveur, avant de lancer les différents threads,
- doit se déconnecter du serveur avant de terminer le processus par un exit.

Remarquez que si vous ne passez aucun paramètre à votre application, aucune connexion au serveur ne doit être réalisée, et vous jouez alors « hors ligne ».

Etape 11 : Création du threadTopScore

En plus de ce qui a déjà été cité, le module **ClientTetris** dispose des fonctions suivantes :

- `int EnvoiScore(key_t cle,int score)` : fonction permettant d'envoyer son score final au serveur. Le serveur a mémorisé le login et le pseudo du joueur, ainsi que le pid associé à votre processus, donc inutile de lui passer en paramètre. La fonction retourne 1 dans le cas où score a dépassé l'ancien TopScore et est donc devenu le nouveau TopScore, 0 dans le cas où score est inférieur au TopScore actuel et, -1 en cas d'erreur.
- `int GetTopScore(key_t cle,TOPSCORE *pTopScore)` : fonction permettant de récupérer, dans une structure TOPSCORE passée en paramètre, le TopScore actuel, le login et le pseudo du joueur détenteur de l'actuel TopScore (voir ClientTetris.h). La fonction retourne 0 si elle a pu récupérer le TopScore actuel et -1 en cas d'erreur.

A partir du thread principal, lancer le **threadTopScore** dont le rôle est d'afficher « en temps réel » le TopScore dans la zone graphique appropriée et, le login et pseudo dans la zone de défilement gérée par le **threadDefileChaine**. Une fois démarré, le **threadTopScore** entre dans une boucle dans laquelle

- il se met en attente sur le signal SIGQUIT (utilisation de **pause()**). **Seul le threadTopScore pourra interpréter le signal SIGQUIT (vous devez donc masquer correctement les signaux dans les threads)**. Le signal SIGQUIT est envoyé par le serveur à chaque fois qu'un joueur a réussi à battre l'ancien TopScore.
- Une fois sa pause débloquée, il récupère les informations du TopScore dans une structure TOPSCORE locale (utilisation de **GetTopScore**) et affiche la valeur du nouveau TopScore dans les cases (8,15), (8,16), (8,17) et (8,18) de la fenêtre graphique. De plus, il construit une chaine de caractères obtenue par la concaténation du pseudo et du login du joueur détenteur du nouveau TopScore, chaine qu'il transmet au **threadDefileMessage** (utilisation de **setMessage**).
- remonte dans sa boucle pour se remettre en attente du signal SIGQUIT.

Etape 12 : Fin de la partie, envoi du score au serveur et synchronisation finale

Lorsque la partie se termine, c'est-à-dire lorsque le threadFinPartie se termine, le thread principal reprend la main et réalise toutes une série d'opérations avant de terminer le processus (voir étape 9). Il s'agit à présent d'envoyer le score obtenu au serveur afin de voir si le TopScore a été battu ou non. Cette tâche va être réalisée par le **threadScore** avant de se terminer.

Donc, avant d'attendre un événement de type CROIX (voir étape 9), le thread principal va

- tuer le **ThreadScore** (utilisation de **pthread_cancel**). Celui-ci doit d'abord passer par une fonction de terminaison (utilisant de **pthread_cleanup_push** et **pthread_cleanup_pop**) dans laquelle il va envoyer le score final au serveur (utilisation de **EnvoiScore**). Si le TopScore n'est pas battu, le simple message « Game Over » sera transmis au threadDefileMessage (utilisation de **setMessage**). Par contre, si le TopScore est battu, le message « Game Over mais vous avez obtenu le nouveau Top Score » sera transmis au threadDefileMessage.

- attendre que le **threadScore** se termine (utilisation de **pthread_join**).

Remarques : signaux et mutex

La gestion des signaux (armement et masquage) est importante dans ce dossier. N'oubliez pas d'**armer** et de **masquer** correctement tous les **signaux** gérés par l'application ! Pour vous aider, voici un tableau récapitulatif des threads de l'application et des signaux qu'ils peuvent recevoir, les autres devant être masqués :

Threads	Signaux pouvant être reçus
Principal	Aucun
threadDefileChaine	Aucun
threadEvent	Aucun
threadPiece	Aucun
threadCase	**SIGUSR1**
threadGravite	Aucun
threadFinPartie	**SIGUSR2**
threadScore	Aucun
threadJoueursConnectes	**SIGHUP**
threadTopScore	**SIGQUIT**

De la même manière, le tableau suivant rappelle les **mutex** principaux et les variables globales qu'ils protègent :

Mutex	Variables globales
mutexTraitement	traitementEnCours
mutexMessage	message, tailleMessage, indiceCourant
mutexPieceEnCours	pieceEnCours
mutexCasesInserees	casesInserees, nbCasesInserees
mutexAnalyse	nbAnalyses, lignesCompletes, nbLignesCompletes, colonnesCompletes, nbColonnesCompletes
mutexScore	Score, MAJScore

Mais… N'oubliez pas qu'une variable globale utilisée par plusieurs threads doit être protégée par un mutex. Il se peut donc que vous deviez ajouter un ou des mutex non précisé(s) dans l'énoncé !

☺ Remarque ☺

Une instance de Serveur (clé = 4100), propriété de root, sera lancée sur Sunray1. Une fois terminé, vous pourrez connecter votre Tetris dessus. Vous pourrez ainsi confronter vos scores à vos condisciples ☺.

Consignes

Ce dossier doit être **réalisé sur SUN** et par **groupe de 2 étudiants**. Il devra être terminé pour le dernier jour du 3ème quart. Les date et heure précises vous seront fournies ultérieurement.
Votre programme devra obligatoirement être placé dans le répertoire **$(HOME)/Thread2014**, celui-ci sera alors **bloqué (par une procédure automatique) en lecture/écriture à partir de la date et heure qui vous seront fournies !**
Vous défendrez votre dossier oralement et serez évalués **par un des professeurs responsables**. Bon travail !

Schéma général de l'application