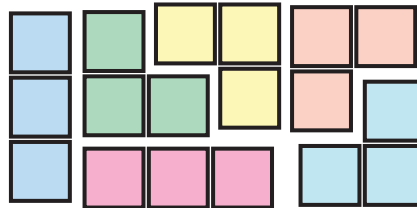


Tés triste

TP BDSP 13/14

Introduction

Dans ce TP, vous allez devoir réaliser un téttris très simplifié : les pièces sont de taille 3 (et non 4) et elles ne peuvent pas tourner.



Ce TP est fait pour être réalisé en équipe (2-4 membres) mais vous pouvez très bien le faire tout seul, si vous en êtes capable (même si c'est moins marrant).

But du TP : apprendre à utiliser **Git** à plusieurs et apprendre les bases d'**XNA**.

Table des matières

1 Création du dépôt Git et du projet XNA

Si vous êtes bloqué à une étape, les assistants sont là pour vous aider. Alors n'hésitez pas à leur demander.

1. Commencez par tous vous inscrire sur <https://bitbucket.org> ou <http://github.com> (mais tous les membres au même endroit sinon ça va pas le faire). Ces deux sites permettent à des développeurs d'héberger leurs dépôts Git, et ce gratuitement.

Attention : GitHub ne permet pas de faire de dépôt privé de base ; il faut un compte étudiant. Préférez BitBucket si vous n'êtes pas quelqu'un d'opensource.

Attention :

Les étapes 2 et 3 sont faites par une seule personne. Choisissez la. Les autres regardent et l'aident.

2. L'heureux élu se rend donc sur le site que vous avez choisi et crée un nouveau dépôt (en général y'a un gros bouton pour ça, vous allez y arriver). Editez les réglages de votre dépôt pour ajouter vos coéquipiers et donnez leur au moins le droit d'écriture, voire les droits admins si vous êtes de bonne humeur.

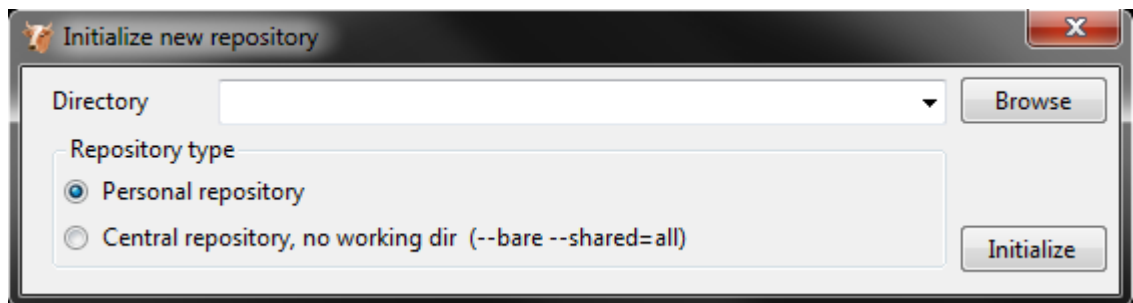
3. Cette même personne crée un nouveau projet XNA sur VS.

% **Git Extension in Visual Studio 2010 :**

1. Commencez par installer le plugin Git Extension pour Visual Studio 2010 si celui-ci n'est pas déjà installé. Vous devriez avoir ceci dans votre barre de menus :

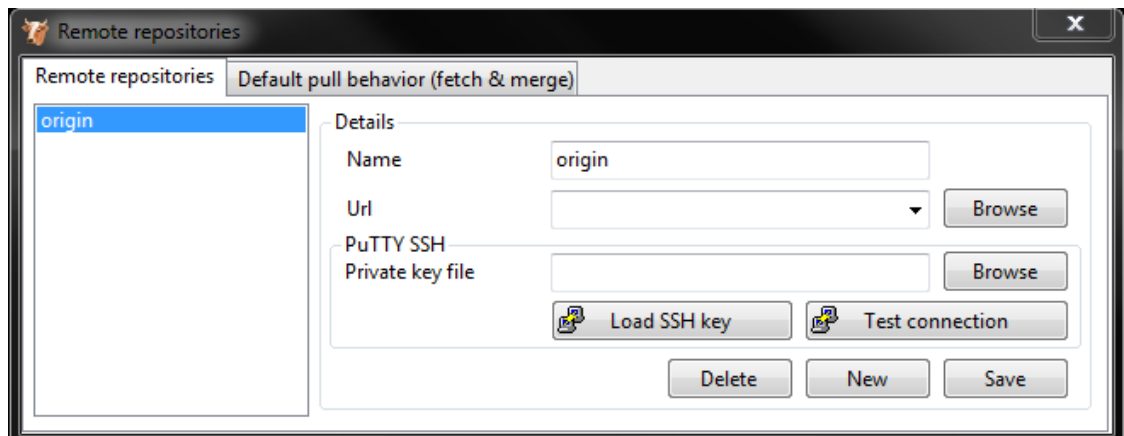
Fichier Edition Affichage Git

2. Ensuite créez un dépôt local vide. Pour cela, allez dans **Git >> Initialize new repository**. Une fenêtre telle que celle-ci devrait s'afficher :



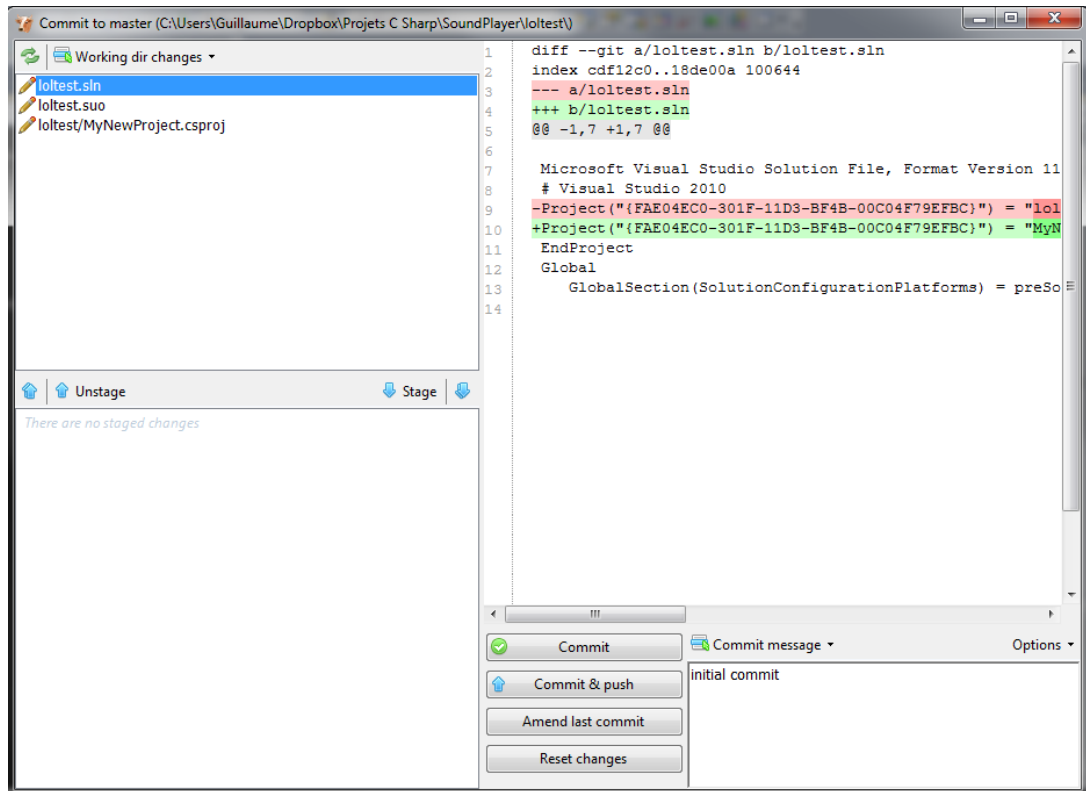
Indiquez le chemin vers le dossier de votre projet et cliquez sur **Initialize**.

3. Allez dans les options de **Git Extension** et configurez votre nom et votre adresse email.
4. Maintenant configurez le dépôt distant : Allez dans **Git >> Manage remote repositories**, et indiquez le nom du dépôt ainsi que son url dans la fenêtre suivante :



Cliquez sur **Save** puis fermez la fenêtre.

5. Vous allez maintenant effectuer votre premier commit. Pour cela, allez dans **Git >> Commit**. Vous devriez obtenir une fenêtre similaire à celle-ci :

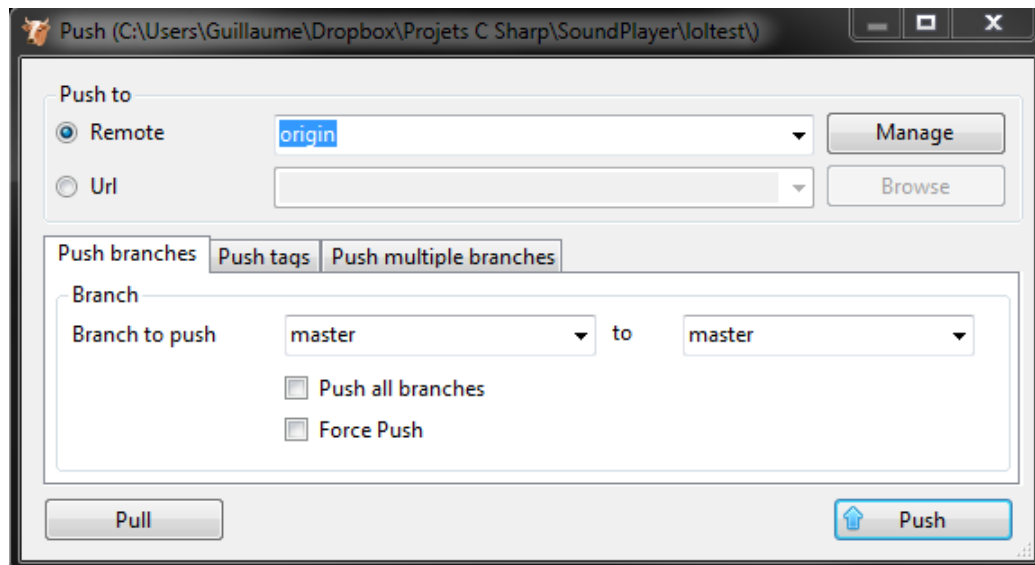


Choisissez les fichiers que vous voulez commit en les sélectionnant et en cliquant sur **Stage**. Entrez un message de commit, ce message sera affiché avec le commit dans votre repo, il vous permet d'indiquer quels sont les changements apportés à votre projet depuis le dernier commit.

Comme il s'agit ici du premier commit de votre projet, vous pouvez mettre ce que vous voulez en message.

Cette fenêtre comporte aussi une information très utile : les différences apportées à vos fichiers depuis votre dernier commit.

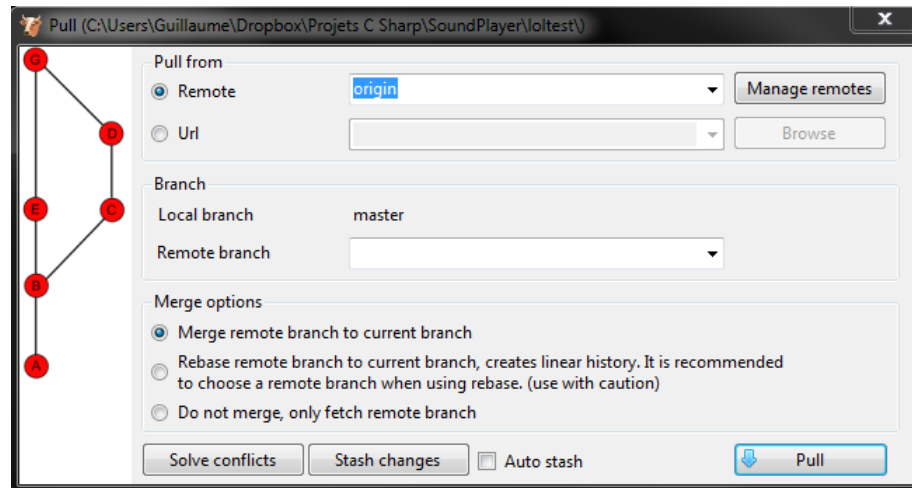
6. Les commits c'est bien gentil mais on veut pouvoir partager son code. Pour ça il faut **push**. Cela va envoyer les changements de votre projet sur le dépôt distant afin que vos coéquipiers puissent les récupérer. Pour faire un **push** il existe deux méthodes :
- Tout d'abord, au lieu de faire juste un **commit**, vous auriez pu faire directement un **commit and push** pour envoyer les changements appliqués à votre dépôt local au dépôt distant d'un seul coup.
 - Ou alors vous pouvez aller dans **Git >> push** et obtenir une fenêtre comme celle-ci :



La deuxième méthode a l'avantage de vous permettre d'envoyer d'un seul coup le résultat de plusieurs commits d'affilée sur le dépôt distant.

Choisissez le dépôt et la branche sur lequel **push** puis appuyez sur **Push**, entrez votre password **Bitbucket** ou **GitHub** selon le site sur lequel vous avez créé un compte, puis attendez. Ça ne devrait pas être très long.

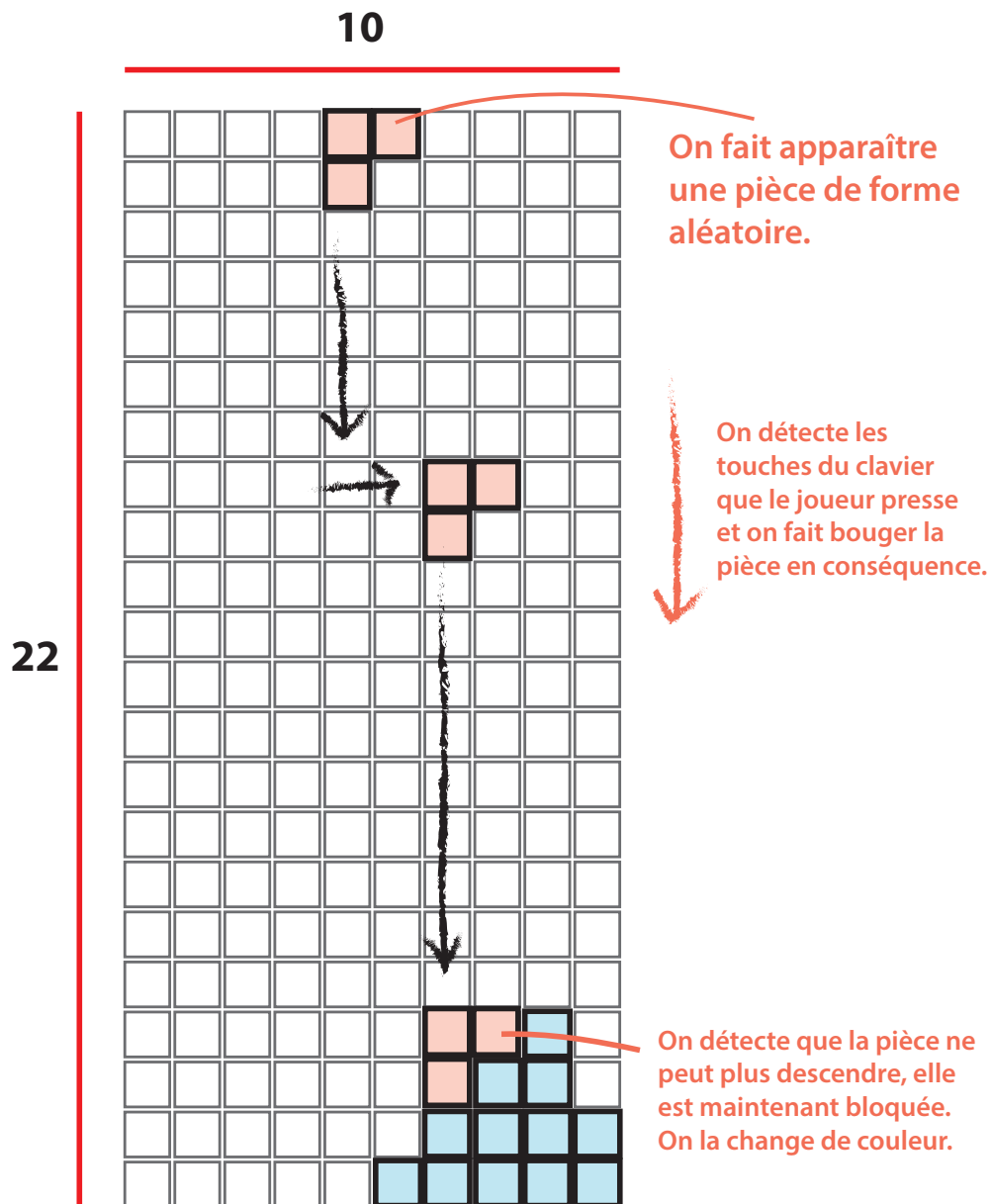
7. Maintenant que votre code se trouve sur le dépôt distant, vos coéquipiers peuvent cloner ce dépôt depuis votre service de hosting préféré. Pour ça aussi il devrait y avoir un gros bouton prévu à cet effet.
8. Cloner un dépôt pour récupérer les modifications d'un de vos camarade c'est quand même pas très pratique, vous en conviendrez, pour éviter de devoir à chaque fois cloner le dépôt, git a une fonctionnalité très pratique : **pull**. Cela sert justement à charger les 'mises à jour' du dépôt distant afin de ne seulement télécharger ce qui a été modifié en lieu et place du dépôt entier.
- Pour cela allez dans **Git >> pull** pour obtenir la fenêtre suivante :



Sélectionnez le dépôt ainsi que la branche à **pull** puis cliquez sur **Pull**. Entrez votre password si demandé puis patientez.

Et vous voilà fin prêts pour travailler en toute quiétude !

2 Principe du jeu



Remarque : ce processus tourne en boucle puisque quand une pièce devient bloquée, on en fait apparaître une nouvelle qui subira le même sort que la précédente.

3 Liste des fonctionnalités à implémenter

1. TexturesManager : charger les textures à dessiner
2. Plateau de jeu (map sur laquelle les pièces se déplaceront)
3. Pièce (composée de 3 cases)
 - (a) Génération d'une pièce aléatoire
 - (b) Déplacement
4. Boucle de jeu principale
 - (a) Initialisation du jeu (dans la fonction du même nom)
 - (b) Détection du GameOver
 - (c) Update() : Détection de l'input du joueur (touches clavier)
 - (d) Draw()

Pour vous aider, la partie 5 vous explique comment implémenter chaque fonctionnalité. Vous n'êtes pas obligés de vous en servir. Si vous voulez un peu plus de challenge vous pouvez proposer votre propre implémentation. :) Vous êtes libres.

4 Les images du jeu sont fournies !

Pour les télécharger, rendez vous ici : <http://too.gy/bdsp/tp.zip>.

GameBackground.png : image de fond du jeu, avec le plateau. On affichera les autres éléments (comme les pièces) par dessus.

GameOver.png : image à moitié transparente. On l'affichera par dessus toutes les autres quand la partie sera terminée.

Piece.jpg : image d'une pièce

PieceBlocked.jpg : image d'une pièce bloquée

5 Comment implémenter tout ça

5.1 TexturesManager : chargement des textures

On va commencer par charger nos textures dans notre jeu. Comme ça ce sera fait et on y aura accès par la suite.

Pour charger des images et les mettre sous forme de textures, XNA se sert d'un objet appelé le 'ContentManager' :

```
1 // Vous remarquerez qu'il n'y a pas l'extension sur le nom du fichier
2 Texture2D MaTexture = content.Load<SpriteFont>("MonFichier");
```

Ce qu'on va faire, c'est créer une sorte de 'service', une classe statique qui va contenir nos textures, à laquelle on aura accès depuis n'importe quelle classe de notre solution (projet).

```

1 // ATTENTION au mot clef 'static'
2 public static class TexturesManager
3 {
4     public static Texture2D GameBackground,
5                             GameOver,
6                             Piece,
7                             PieceBlocked;
8
9     public static void LoadContent(ContentManager content)
10    {
11        // TODO: Charger les 4 textures
12    }
13 }

```

Il faut appeler la fonction LoadContent de notre TexturesManager depuis la classe principale 'Game1', dans sa propre fonction LoadContent :

```

1 protected override void LoadContent()
2 {
3     IsMouseVisible = true;
4     spriteBatch = new SpriteBatch(GraphicsDevice);
5
6     TexturesManager.LoadContent(Content);
7 }

```

On peut maintenant accéder à nos 4 textures depuis n'importe quelle classe de la façon suivante :

```

1 TexturesManager.NomDeLaTexture ;

```

5.2 Plateau de jeu

5.2.1 Principe

Notre plateau de jeu est composé de **10** cases sur la coordonnée x de XNA et de **22** cases sur sa coordonnée y . (voir schéma 'Principe du jeu')

Pour chaque case on a envie de savoir si la case est occupée par soit :

- une case vide
 - un bloc d'une pièce bloquée (une pièce est constituée de 3 blocs)
 - un bloc d'une pièce non-bloquée
- (on différencie pièce bloquée et non-bloquée car on veut les afficher différemment).

Le plus simple c'est de représenter les cases notre plateau par un tableau de bytes :

```

1 public class Plate
2 {
3     public byte[,] Cells { get; set; }
4 }

```

Pour accéder à une case de notre plateau, il suffit alors de faire :


```

1 Plate p = new Plate();
2
3 byte b = p.Cells[0,0]; // récupère la case tout en haut à gauche

```

Pour parcourir l'ensemble de nos cases, il suffit de faire :

```

1 for (int x = 0; x < 10; x++)
2 {
3     for (int y = 0; y < 22; y++)
4     {
5         // Valeur de la case = p.Cells[x,y]
6     }
7 }

```

Si vous avez des problèmes avec toutes ces notions, demandez aux assistants de vous les expliquer. Vous les verrez bientôt en TP, ce sont des choses très basiques.

5.2.2 Affichage

```

1 public class Plate
2 {
3     public void Draw(SpriteBatch spriteBatch)
4     {
5         // On boucle sur chaque case
6         // Si la case est un bloc on affiche sa texture (bloc bloqué ou en mouvement)
7         // Sinon on fait rien
8     }
9 }

```

N.B : la taille de la texture de l'image est de 22x22.

Rappel : pour dessiner une texture avec XNA, c'est :

```

1 spriteBatch.Draw(MaTexture2D, new Vector2(x, y), Color.White);

```

Il est certain que vous ferez attention au décalage de base par rapport au coin en haut à gauche de l'écran :



5.2.3 Fonctions à implémenter

```
1 public class Plate
2 {
3     public bool IsLineFull(int y)
4     {
5         // Renvoie vrai si la ligne numéro y est pleine (byte 1 ou 2)
6     }
7
8     public void EmptyLine(int y)
9     {
10        // Vide la ligne y et fait tomber tous les blocs
11        // au dessus de cette ligne d'une case
12    }
13 }
```

5.3 Pièce

5.3.1 Implémentation

On peut considérer qu'une pièce est un ensemble de 3 blocs. Chaque bloc a des coordonnées qui lui sont propres sur la map, soit un couple (x, y) .

Pour plus de clarté, on va déclarer une structure (si vous ne savez pas ce que c'est, considérez que c'est un peu comme les enregistrements en OCaml. Si vous ne voyez toujours pas, considérez que c'est une classe) :

```
1 public class Piece
2 {
3     public Cell[] Cells { get; set; }
4
5     public struct Cell
6     {
7         public byte X { get; set; }
8         public byte Y { get; set; }
9
10        public Cell(byte x, byte y)
11        {
12            X = x;
13            Y = y;
14        }
15    }
16 }
```

5.3.2 Génération aléatoire

On veut qu'à la création de la pièce, lui soit affectée une forme aléatoire. L'une de celles-ci :



Et on sait que cette pièce possède 3 blocs qui ont chacun des coordonnées sur le plateau de jeu. Vous Connaissez les coordonnées du haut du plateau de jeu. À vous de jouer.

Vous devriez randomizer la pièce dans le constructeur, étant donné qu'on veut qu'à chaque création d'une pièce, elle soit aléatoire.

```
1 Random random = new Random();
2 random.Next(6); // renvoie un entier compris entre 0 et 5 (compris)
```

5.3.3 Déplacement

Pour pouvoir se déplacer, la pièce a besoin de savoir sur quel plateau de jeu elle est. On va donc lui rajouter le plateau en attribut. On donnera une valeur à cet attribut au moment de la création de la pièce, dans le constructeur donc :

```
Piece.cs
1 public class Piece
2 {
3     private Plate _plate;
4
5     public Piece(Plate plate)
6     {
7         _plate = plate;
8     }
9 }
```

Pour déplacer la pièce on va :

- d'abord l'enlever du plateau (mettre les bytes de ses cases à 0)
- calculer les nouvelles coordonnées de ses cases
- la remettre sur le plateau (passer les bytes de ses cases à 1)

Implémentez donc :

```
Piece.cs
1 public class Piece
2 {
3     // Met les bytes des cases de la pièce à 0 sur le plateau
4     public void Clear(){}
5
6     // Met les bytes des cases de la pièce à 2 sur le plateau
7     public void Print(){}
8 }
```

Clear() est appelée avant chaque déplacement, Print() juste après.

Elles seront appelées dans les fonctions :

```
Piece.cs
1 public class Piece
2 {
3     ...
4
5     public void MoveDown(){}
6
7     public void MoveLeft(){}
8
9     public void MoveRight(){}
10
11     ...
12 }
```

Ces fonctions déplacent la pièce dans chaque direction (sauf vers le haut) **uniquement** si il est possible pour elle de bouger (il faut vérifier que cette condition est vérifiée avant de Clear() la pièce et de la déplacer).

Lorsqu'une pièce ne peut plus descendre, elle est bloquée (rappelons que dans un téttris la pièce descend constamment).

Implémentez donc les fonctions suivantes :

```
Piece.cs
1 public class Piece
2 {
3     ...
4
5     // Vérifie qu'une pièce peut descendre d'une case
6     public bool CanGoDown();
7
8     // Passe les bytes des cases de la pièce à 1 sur le plateau de jeu
9     public void Block();
10
11     ...
12 }
```

5.4 Boucle de jeu principale

Première chose à faire, ajouter à notre jeu les bons attributs :

```
Game1.cs
1 public class Game1
2 {
3     ...
4
5
6     // On va mettre notre plateau de jeu ici :)
7     private Plate _plate;
8
9     // Ainsi que notre pièce actuellement en mouvement
10    private Piece _currentPiece;
11
12    // Une énumération des états possibles que peut prendre notre jeu
13    // Si vous avez du mal avec les enum, mettez un booléen IsRunning
14    enum GameState { Running, GameOver }
15
16    private GameState _gameState;
17
18    // La fonction Update() de notre classe principale tourne en boucle un
19    // nombre fini de fois par seconde. On veut que notre pièce descende de
20    // manière constante.
21    // On va avoir besoin de compter le nombre de fois qu'on passe sur la
22    // fonction Update(), pour bouger la pièce automatiquement vers le bas
23    // à partir d'un certain nombre de fois. (Si on le fait à chaque Update()
24    // ça le fera plusieurs fois par seconde et c'est beaucoup trop. On veut
25    // contrôler ça)
26    private int _timer;
27
28    ...
29 }
```

5.4.1 Initialisation du jeu

```
Game1.cs
1 public class Game1
2 {
3     ...
4
5     protected override void Initialize()
6     {
7         // TODO: Créer un nouveau plateau de jeu (Plate) et le mettre dans _plate
8
9         // TODO: Créer une nouvelle pièce
10
11        // Passe l'état du jeu en 'Running'
12        _gameState = GameState.Running;
13
14        // Le code qui suit change la taille de la fenêtre :
15        graphics.PreferredBackBufferWidth = 400;
16        graphics.PreferredBackBufferHeight = 580;
17        graphics.ApplyChanges();
18
19        // TODO: Récupérer l'état courant du clavier dans _pKs et _cKs
20
21        base.Initialize();
22    }
23
24    ...
25 }
```

5.4.2 Détection du GameOver

Implémenter la fonction qui détecte si la partie est finie.

```
Game1.cs
1 public class Game1
2 {
3     ...
4
5     private bool IsGameOver()
6     {
7         // TODO: vérifier les bytes de la map pour savoir si la partie est
8         //      terminée
9     }
10
11    ...
12 }
```

5.4.3 Update()

Notre première boucle de jeu. Voilà ce qu'il faut faire dedans :

```

Game1.cs
1 public class Game1
2 {
3     ...
4
5     protected override void Update(GameTime gameTime)
6     {
7         // TODO: vérifier que l'état de la partie n'est pas à GameOver
8         //     Si c'est le cas, retourner (return;)
9
10        // Vérifier que le joueur n'appuie pas sur 'flèche gauche' ou
11        // 'flèche droite'. Si c'est le cas, on bouge la pièce en conséquence.
12
13        // TODO: Incrémenter le _timer
14
15        // TODO: Si (_timer == 20) et que la pièce peut descendre vers le bas,
16        //     on la descend.
17        //     Sinon, on bloque la pièce, on vérifie qu'il n'y a pas de ligne
18        //     pleine. Si c'est le cas on les explose (récursivement, il peut
19        //     y avoir plusieurs lignes pleines en même temps (max 3)
20        //     Et on vérifie que la partie n'est pas terminée. Si c'est le cas
21        //     on passe _gameState à GameState.GameOver
22    }
23
24    ...
25 }

```

5.4.4 Draw()

```

Game1.cs
1 public class Game1
2 {
3     ...
4
5     protected override void Draw(GameTime gameTime)
6     {
7         spriteBatch.Begin(); // Rajoutez cette ligne
8
9         // TODO: Afficher la texture GameBackground
10
11        // TODO: Afficher le plateau de jeu
12
13        // TODO: Si l'état du jeu est GameOver, on affiche la texture
14        //     GameOver par dessus le tout
15
16        spriteBatch.End(); // Rajoutez cette ligne
17
18        base.Draw(gameTime);
19    }
20
21    ...
22 }

```

6 Bonus

Si vous êtes vraiment des malades et que vous avez tout fait, vous pouvez vous amuser à implémenter ces quelques fonctionnalités supplémentaires :

1. Rajout d'un état de jeu 'Pause'
2. Drop (la pièce tombe le plus bas possible et se bloque)
3. Scoring
 - (a) Calcul du score
 - (b) Affichage du score (SpriteFont XNA)
4. Piece holding (le joueur garde une pièce en mémoire)

7 Conclusion

Avec GConfs, GConfiance !

Ceci était un sujet de TP avec des jeux de mots débiles. :)