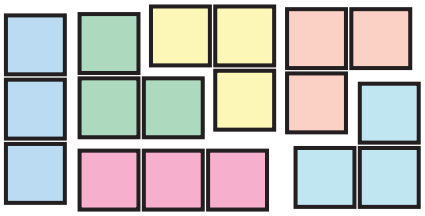
T'es triste

TP BDSP 13/14

Introduction

Dans ce TP, vous allez devoir réaliser un tétris 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.

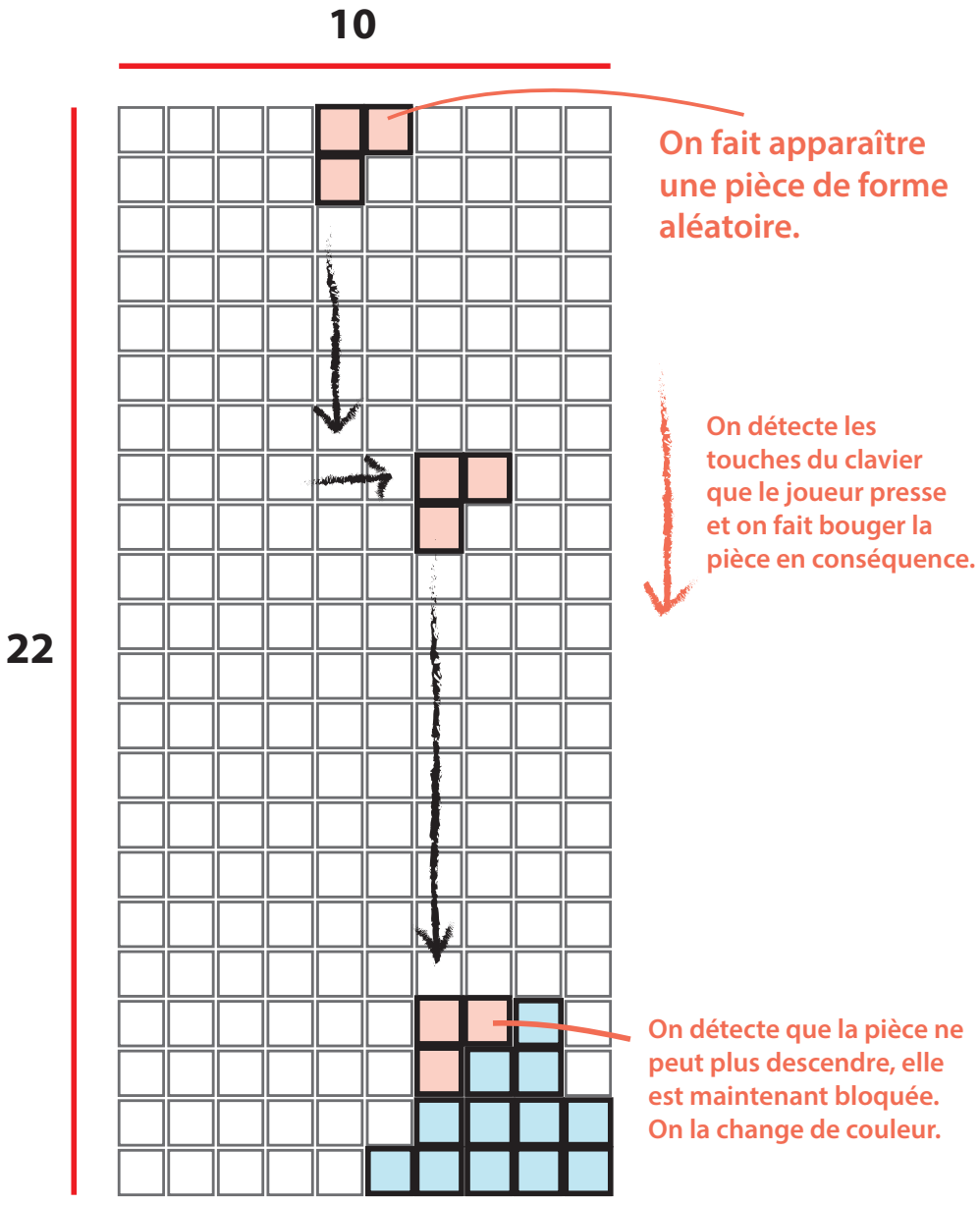
Attention :
Les étapes **2** et **3** sont faites par une seule personne. Choisissez la. Les autres regardent et l'aident.

2. L'heureux élu se rend donc sur le site que vous avez choisi et créé un nouveau dépôt (en général y'a un gros bouton pour ça, vous allez y arriver).

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

% **TODO** : Corwin doit rajouter des étapes pour l'extension Git de VS À la fin, ils sont tous sensés avoir le nouveau projet sur leur VS.
Expliquer comment utiliser l'extension pour commit, push etc.

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 téléchargez, rendez vous ici : `http://too.gy/bdsp/tp.zip`.

GameBackground.png : image de fond du jeu, avec le plateau. On affichera les autres élements (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 mettres 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>("MonFicher");
```

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).

TexturesManager.cs
```
// ATTENTION au mot clef 'static'
public static class TexturesManager
{
    public static Texture2D GameBackground,
                            GameOver,
                            Piece,
                            PieceBlocked;

    public static void LoadContent(ContentManager content)
    {
        // TODO: Charger les 4 textures
    }
}
```

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

Game1.cs
```
protected override void LoadContent()
{
    IsMouseVisible = true;
    spriteBatch = new SpriteBatch(GraphicsDevice);

    TexturesManager.LoadContent(Content);
}
```

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

```
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 :

Plate.cs
```
public class Plate
{
    public byte[,] Cells { get; set; }
}
```

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

```
Plate p = new Plate();

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 :

```
for (int x = 0; x < 10; x++)
{
    for (int y = 0; y < 22; x++)
    {
        // Valeur de la case =  p.Cells[x,y]
    }
}
```

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

Plate.cs

```
public class Plate
{
    public void Draw(SpriteBatch spriteBatch)
    {
        // On boucle sur chaque case
        // Si la case est un bloc on affiche sa texture (bloc bloqué ou en mouvement)
        // Sinon on fait rien
    }
}
```

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

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

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

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

5.2.3 Fonctions à implémenter

Plate.cs
```cs
public class Plate
{
    public bool IsLineFull(int y)
    {
        // Renvoie vrai si la ligne numéro y est pleine (byte 1 ou 2)
    }

    public void EmptyLine(int y)
    {
        // Vide la ligne y et fait tomber tous les blocs
        // au dessus de cette ligne d'une case
    }
}
```

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) :

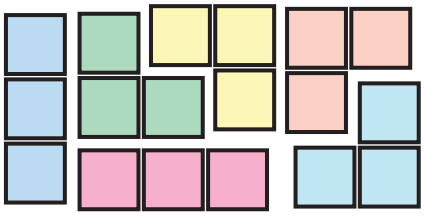
Piece.cs
```cs
public class Piece
{
    public Cell[] Cells { get; set; }

    public struct Cell
    {
        public byte X { get; set; }
        public byte Y { get; set; }

        public Cell(byte x, byte y)
        {
            X = x;
            Y = y;
        }
    }
}
```

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.

```
Random random = new Random();
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
```
public class Piece
{
    private Plate _plate;

    public Piece(Plate plate)
    {
        _plate = plate;
    }
}
```

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
```
public class Piece
{
    // Met les bytes des cases de la pièce à 0 sur le plateau
    public void Clear(){}

    // Met les bytes des cases de la pièce à 2 sur le plateau
    public void Print(){}
}
```

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

Elles seront appelées dans les fonctions :

Piece.cs
```
public class Piece
{
    ...

    public void MoveDown(){}

    public void MoveLeft(){}

    public void MoveRight(){}

    ...
}
```

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 tétris la pièce descend constamment).

Implémentez donc les fonctions suivantes :

Piece.cs
```
public class Piece
{
    ...

    // Vérifie qu'une pièce peut descendre d'une case
    public bool CanGoDown();

    // Passe les bytes des cases de la pièce à 1 sur le plateau de jeu
    public void Block();

    ...
}
```

5.4 Boucle de jeu principale

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

Game1.cs

```cs
public class Game1
{
    ...


    // On va mettre notre plateau de jeu ici :)
    private Plate _plate;

    // Ainsi que notre pièce actuellement en mouvement
    private Piece _currentPiece;

    // Une énumération des états possibles que peut prendre notre jeu
    // Si vous avez du mal avec les enum, mettez un booléen IsRunning
    enum GameState { Running, GameOver }

    private GameState _gameState;

    // La fonction Update() de notre classe principale tourne en boucle un
    // nombre fini de fois par seconde. On veut que notre pièce descende de
    // manière constante.
    // On va avoir besoin de compter le nombre de fois qu'on passe sur la
    // fonction Update(), pour bouger la pièce automatiquement vers le bas
    // à partir d'un certain nombre de fois. (Si on le fait à chaque Update()
    // ça le fera plusieurs fois par seconde et c'est beaucoup trop. On veut
    // contrôler ça)
    private int _timer;

    ...
}
```

5.4.1 Initialisation du jeu

Game1.cs

```cs
public class Game1
{
    ...

    protected override void Initialize()
    {
        // TODO: Créer un nouveau plateau de jeu (Plate) et le mettre dans _plate

        // TODO: Créer une nouvelle pièce

        // Passe l'état du jeu en 'Running'
        _gameState = GameState.Running;

        // Le code qui suit change la taille de la fenêtre :
        graphics.PreferredBackBufferWidth = 400;
        graphics.PreferredBackBufferHeight = 580;
        graphics.ApplyChanges();

        // TODO: Récupérer l'état courant du clavier dans _pKs et _cKs

        base.Initialize();
    }

    ...
}
```

5.4.2 Détection du GameOver

Implémenter la fonction qui détecte si la partie est finie.

Game1.cs

```cs
public class Game1
{
    ...

    private bool IsGameOver()
    {
        // TODO: vérifier les bytes de la map pour savoir si la partie est
        //       terminée
    }

    ...
}
```

5.4.3 Update()

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

Game1.cs

```
public class Game1
{
    ...

    protected override void Update(GameTime gameTime)
    {
        // TODO: vérifier que l'état de la partie n'est pas à GameOver
        //       Si c'est le cas, retourner (return;)

        // Vérifier que le joueur n'appuie pas sur 'flèche gauche' ou
        // 'flèche droite'. Si c'est le cas, on bouge la pièce en conséquence.

        // TODO: Incrémenter le _timer

        // TODO: Si (_timer == 20) et que la pièce peut descendre vers le bas,
        //          on la descend.
        //       Sinon, on bloque la pièce, on vérifie qu'il n'y a pas de ligne
        //          pleine. Si c'est le cas on les explose (récursivement, il peut
        //          y avoir plusieurs lignes pleines en même temps (max 3)
        //       Et on vérifie que la partie n'est pas terminée. Si c'est le cas
        //          on passe _gameState à GameState.GameOver
    }

    ...
}
```

5.4.4 Draw()

Game1.cs

```
public class Game1
{
    ...

    protected override void Draw(GameTime gameTime)
    {
        spriteBatch.Begin(); // Rajoutez cette ligne

        // TODO: Afficher la texture GameBackground

        // TODO: Afficher le plateau de jeu

        // TODO: Si l'état du jeu est GameOver, on affiche la texture
        //       GameOver par dessus le tout

        spriteBatch.End(); // Rajoutez cette ligne

        base.Draw(gameTime);
    }

    ...
}
```

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. :)