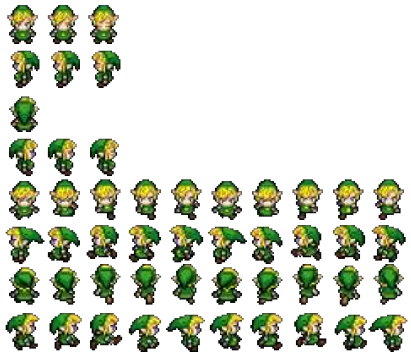
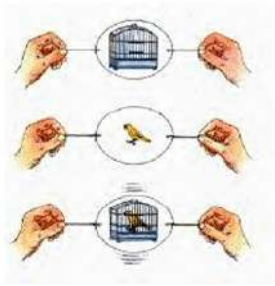
Création d'un RPG

1. Sprites

Dans le domaine des jeux vidéo, un ***sprite*** (***lutin*** en anglais) est un élément graphique animé (personnages, objets, …). Le ***sprite sheet*** représente l'ensemble des mouvements du sprite sous la forme d'une matrice d'images.

Exemple d'un sprite sheet :



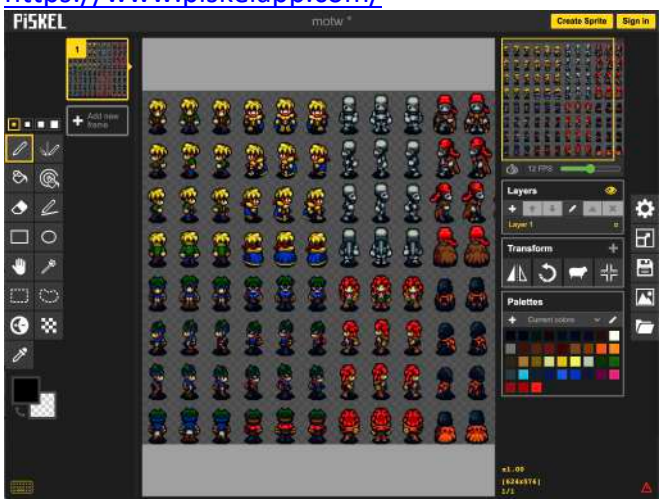


Pour réaliser l'animation d'un sprite, il faut faire défiler les images à l'écran avec une vitesse supérieure à la persistance rétinienne. La persistance rétinienne est le phénomène qui nous permet d'observer un défilement d'images fixes sans voir les coupures entre celles-ci (l'œil permet de voir à peu près une image toutes les 1/25ème de secondes).

L'animation consiste donc à gérer le défilement des images.

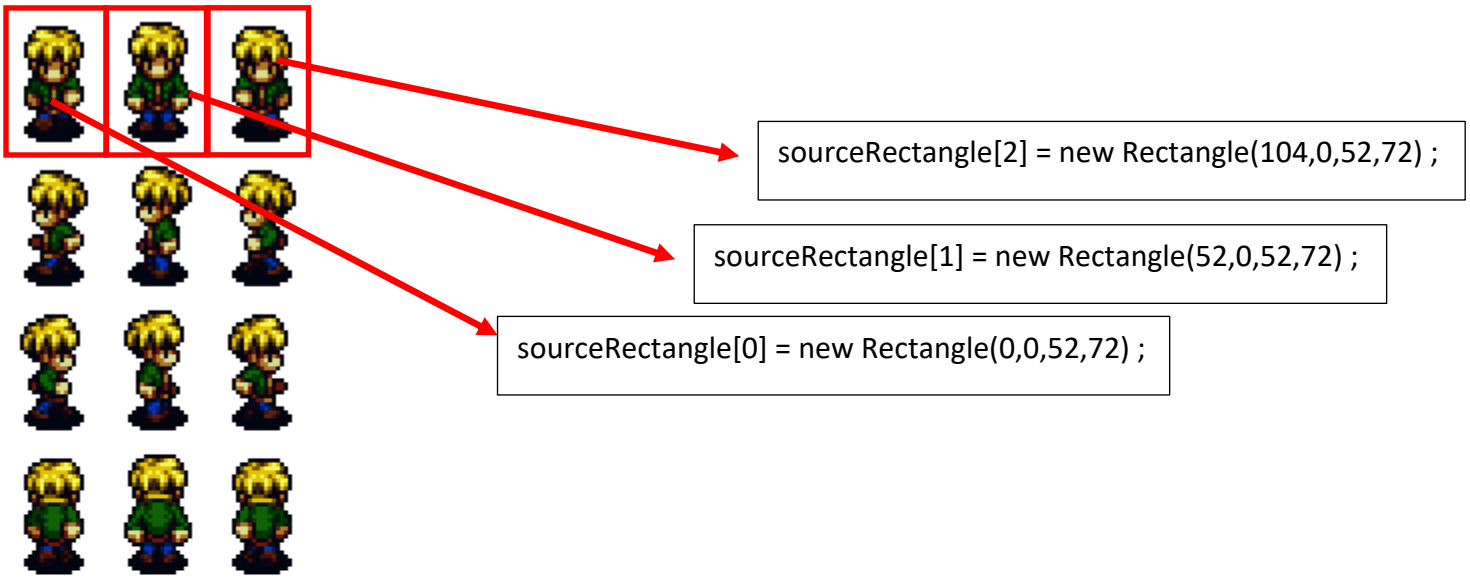
En matière de sprites, de nombreuses ressources sont disponibles sur le web. Mais attention aux droits d'auteur, vous devez bien identifier les licences avant de pouvoir utiliser les sprites mis à disposition.

Pour créer vos propres sprites, il existe des logiciels générateurs de sprites en ligne :
https://www.piskelapp.com/



Animation du sprite

Pour animer notre sprite il suffit d'extraire chaque sprite du spritesheet à l'aide d'un rectangle de sélection.



Ce spritesheet est une matrice d'images (3x4=12 images).
Une image fait 52px de large par 72px de haut.

Si on veut animer notre personnage il suffit donc d'appeler le bon rectangle dans la méthode Draw :

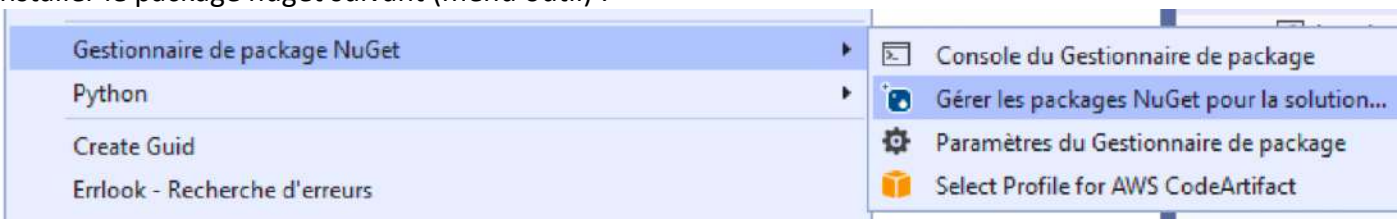
```
_spriteBatch.Draw(player,new Vector2(100,100),sourceRectangle[currentAnimationIndex], Color.White);
```

CurrentAnimationIndex est un index en boucle pour assurer l'animation.
Pour aller plus loin dans la théorie : https://www.industrian.net/tutorials/using-sprite-sheets/

Tout ceci explique le principe de l'animation des sprites. Mais il existe un gestionnaire complet des animations de sprites dans Monogame Extended :
https://www.monogameextended.net/docs/features/animations/animations/

Mise en œuvre :

Faites un nouveau projet MonoGame OpenGL.
Installer le package nuget suivant (menu outil) :



Ce dernier package permet de gérer directement l'animation d'un spritesheet à l'aide d'un fichier au format JSON.
On utilise alors 2 classes :
La classe SpriteSheet pour charger le fichier JSON
La classe AnimatedSprite pour lancer les animations

Exemple de JSON du spritesheet :

```
{
  "textureAtlas": {
    "texture": "motw.png",
    "regionWidth": 52,
    "regionHeight": 72
  },
  "cycles": {
    "idle": {
      "frames": [1],
      "isLooping": false,
      "frameDuration": 0.1
    },
    "walkSouth": {
      "frames": [0,1,2,1],
      "isLooping": false,
      "frameDuration": 0.1
    },
    "walkWest": {
      "frames": [12,13,14,13],
      "isLooping": false,
      "frameDuration": 0.1
    },
    "walkEast": {
      "frames": [24,25,26,25],
      "isLooping": false,
      "frameDuration": 0.1
    },
    "walkNorth": {
      "frames": [36,37,38,37],
      "isLooping": false,
      "frameDuration": 0.1
    }
  }
}
```

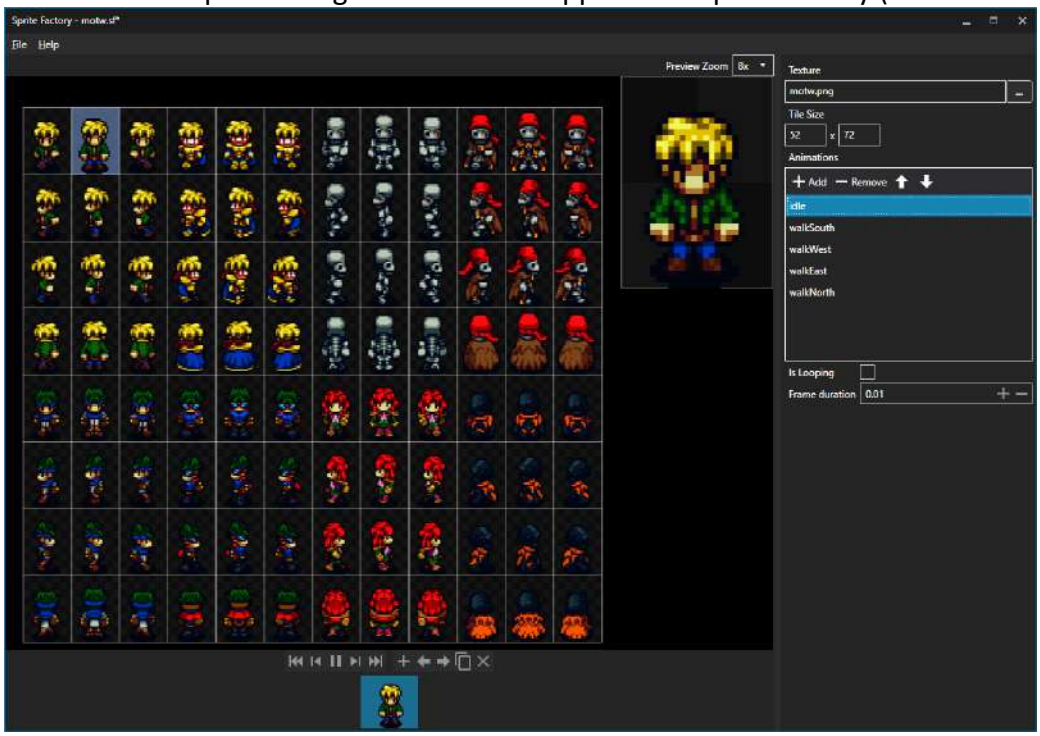
Source du spritesheet et largeur hauteur du sprite

Les 5 cycles d'animations et les numéros de sprites associés dans le spritesheet (frames). FrameDuration correspond à la vitesse d'animation

Les 12 premiers éléments du spritesheet

<u>Pour votre culture personnelle :</u>

Le fichier JSON peut être généré à l'aide l'application SpriteFactory (windows uniquement) :



Pour l'installer :

git clone https://github.com/craftworkgames/SpriteFactory.git

cd SpriteFactory

dotnet run --project SpriteFactory

<u>Application :</u>

Récupérer le fichier motw.png et motw.sf sur le réseau

Ajouter la référence au spritesheet (motw.png) au projet (mgcb-editor).

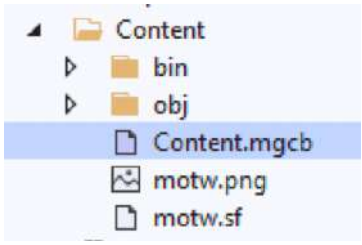
Votre fichier Content.mgcb doit contenu seulement :

```
#----------------------------- References -----------------------------#


#----------------------------- Content -----------------------------#

#begin motw.png
/importer:TextureImporter
/processor:TextureProcessor
/processorParam:ColorKeyColor=255,0,255,255
/processorParam:ColorKeyEnabled=True
/processorParam:GenerateMipmaps=False
/processorParam:PremultiplyAlpha=True
/processorParam:ResizeToPowerOfTwo=False
/processorParam:MakeSquare=False
/processorParam:TextureFormat=Color
/build:motw.png

```

Copier le fichier JSON (motw.sf) dans le répertoire Content (**attention procéder seulement à la copie**)
Ajouter dans la configuration du projet :

```
<ItemGroup>
    <None Update="Content\motw.sf">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
 </ItemGroup>
```

Vous devez avoir seulement les fichiers suivants :



Recharger le projet.
Nous allons créer un petit personnage animé en utilisant ce spritesheet.

Créer les champs privés suivants :

```
private Vector2 _persoPosition;
private AnimatedSprite _perso;
```

Initialiser ces champs privés :

```
protected override void LoadContent()
 {
       _spriteBatch = new SpriteBatch(GraphicsDevice);

       // spritesheet
       SpriteSheet spriteSheet = Content.Load<SpriteSheet>("motw.sf", new JsonContentLoader());
      _perso = new AnimatedSprite(spriteSheet);
 }
```

Résoudre le problème en important les bons packages :

```
using MonoGame.Extended.Sprites;
using MonoGame.Extended.Content;
using MonoGame.Extended.Serialization;
```

Ajouter l'affichage du sprite dans votre fenêtre :

```
protected override void Draw(GameTime gameTime)
{
	GraphicsDevice.Clear(Color.CornflowerBlue);

	// TODO: Add your drawing code here
	_spriteBatch.Begin();
	_spriteBatch.Draw(_perso, _persoPosition);
	_spriteBatch.End();
	base.Draw(gameTime);
}
```

Et vérifier le bon fonctionnement :

Dans la méthode Update nous allons procéder au déplacement du personnage et lancer son animation.
Définir une vitesse de déplacement (_vitessePerso) et l'initialiser à 100.

En utilisant le DeltaTime de l'Update, déclarer le nombre de pixels de déplacement du personnage :

```
float deltaSeconds = (float)gameTime.ElapsedGameTime.TotalSeconds;
float walkSpeed = deltaSeconds * _vitessePerso;
```

Définir une chaine de caractère contenant le cycle d'animation par défaut du sprite animé (ici position de repos) :

```
string animation = "idle";
```

Intercepter les évènements claviers pour calculer la nouvelle position du personnage et modifier le cycle d'animation associé :

```
KeyboardState keyboardState = Keyboard.GetState();
if (keyboardState.IsKeyDown(Keys.Left))
{
	animation = "walkWest";
	_persoPosition.X -= walkSpeed;
}
```

Déclencher l'animation dans l'Update :

```
_perso.Play(animation);
_perso.Update(deltaSeconds);
```

Compléter pour les autres directions et vérifiez le bon fonctionnement.
Compléter votre code pour gérer les collisions avec les bords de la fenêtre (vous pouvez récupérer la largeur et la hauteur à l'aide de `_perso.TextureRegion.Height` et `_perso.TextureRegion.Width`)

2. Tile Mapping

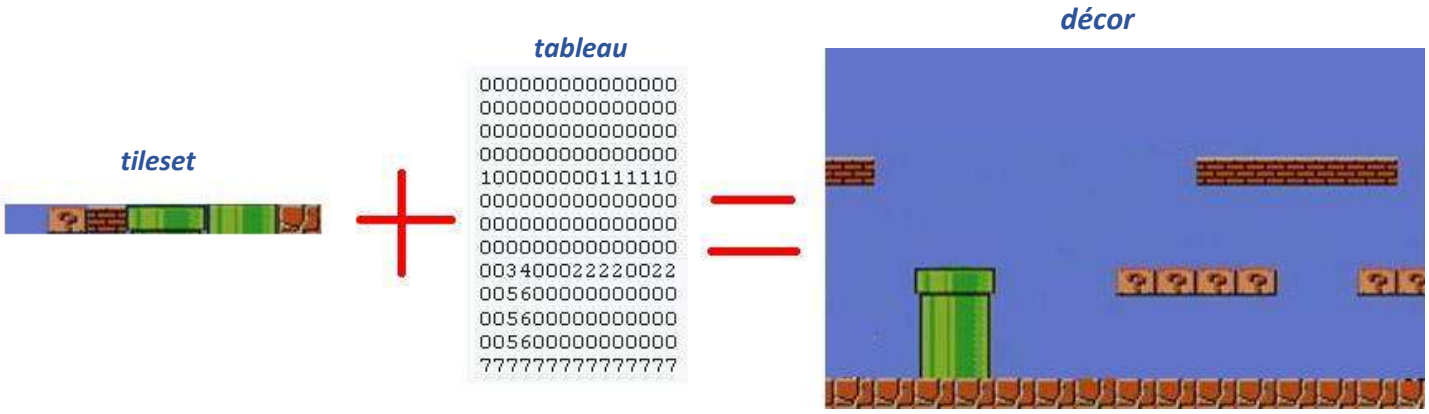
Le tile Mapping est une technique de création de carte utilisée depuis la création des premiers jeux vidéo.
Afin d'éviter de gaspiller une grande quantité de mémoire avec l'utilisation de grandes images pour réaliser le décor d'un jeu, le tile mapping va reconstruire le décor à partir de motifs nommés ***tiles***.
Le concept de Tile Mapping est de placer côte à côte des "***tiles***" (des tuiles en anglais) dans une fenêtre de taille prédéfinie. L'écran est alors divisé en une grille dont la taille de chaque case correspond à celle d'une tuile.

Sur cette image ci-dessus, Mario, l'ennemi, l'étoile, et les nuages, sont des ***sprites*** que l'on insère sur le décor composé de tuiles. Il y a les tuiles uniques, comme les briques et les points d'interrogation, et les tuiles composées, comme le pot de fleur.
Le décor est donc composé de 8 tuiles différentes :

- 1 tuile ciel
- 1 tuile sol
- 1 tuile point d'interrogation
- 1 tuile brique
- 4 tuiles du pot de fleurs

Cet ensemble forme notre ***tileset*** : 

La dimension du décor est ici de 13*15 cases soit 195 cases. Nous pouvons donc imaginer un tableau de 195 cases où la valeur 0 correspond à la tuile ciel, la valeur 1 correspond à la tuile brique et ainsi de suite.



Pour faire ce décor, nous avons donc besoin d'un tileset et du tableau.

Si l'on considère que nous avons 256 tuiles différentes (1 octet de 8 bits soit 2^8 = 256), nous pouvons compter un octet par case soit 195 octets.

Considérons, un niveau entier de 300 cases de long sur 20 cases de haut :

Cela nous donne 300 * 20 = 6000 octets.

Sans cette méthode de tile mapping, nous aurions utilisé une grande image d'une dizaine de MégaOctets.

3. Tileset

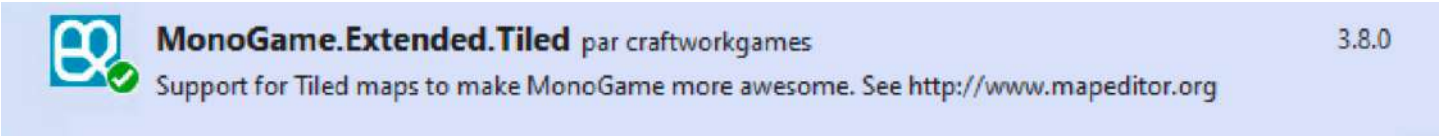
Un ***tileset*** est une image qui contient un ensemble de tuiles pour créer le décor de votre jeu.

Exemple de tileset :

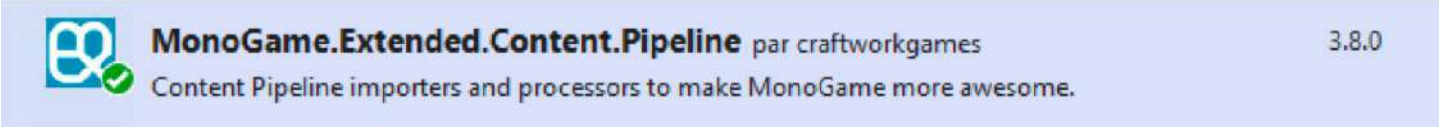
On trouve facilement des tilesets sur internet. Il faut cependant penser à vérifier les licences pour pourvoir les utiliser dans un jeu. Sinon vous ne pourrez pas diffuser votre projet.
Le site OpenGameArt.org propose de nombreuses ressources libres de droit (tilesets, sprites, effets sonores, …).

Il existe un package nuget qui permet de gérer les TileMap.

Installer le package nuget MonoGame.Extended.Tiled :



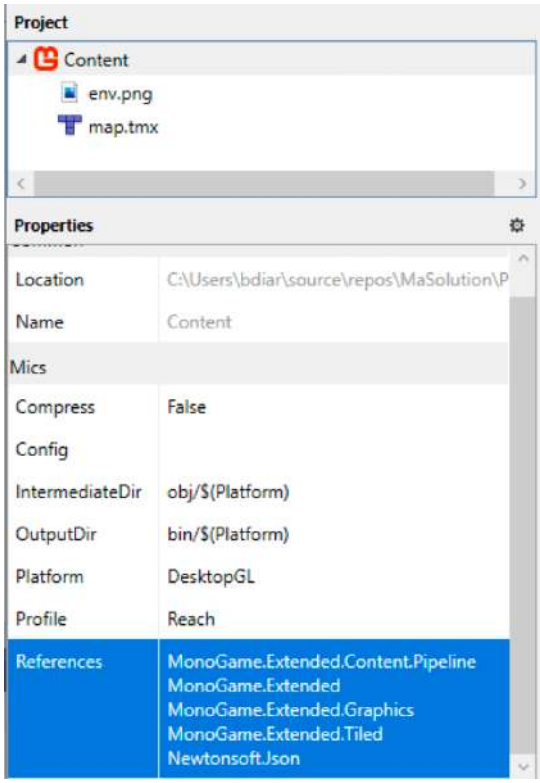
Installer le package nuget MonoGame.Extended.Content.Pipeline



Vous pouvez aussi utiliser la console nuget :

dotnet add package MonoGame.Extended.Tiled
dotnet add package MonoGame.Extended.Content.Pipeline

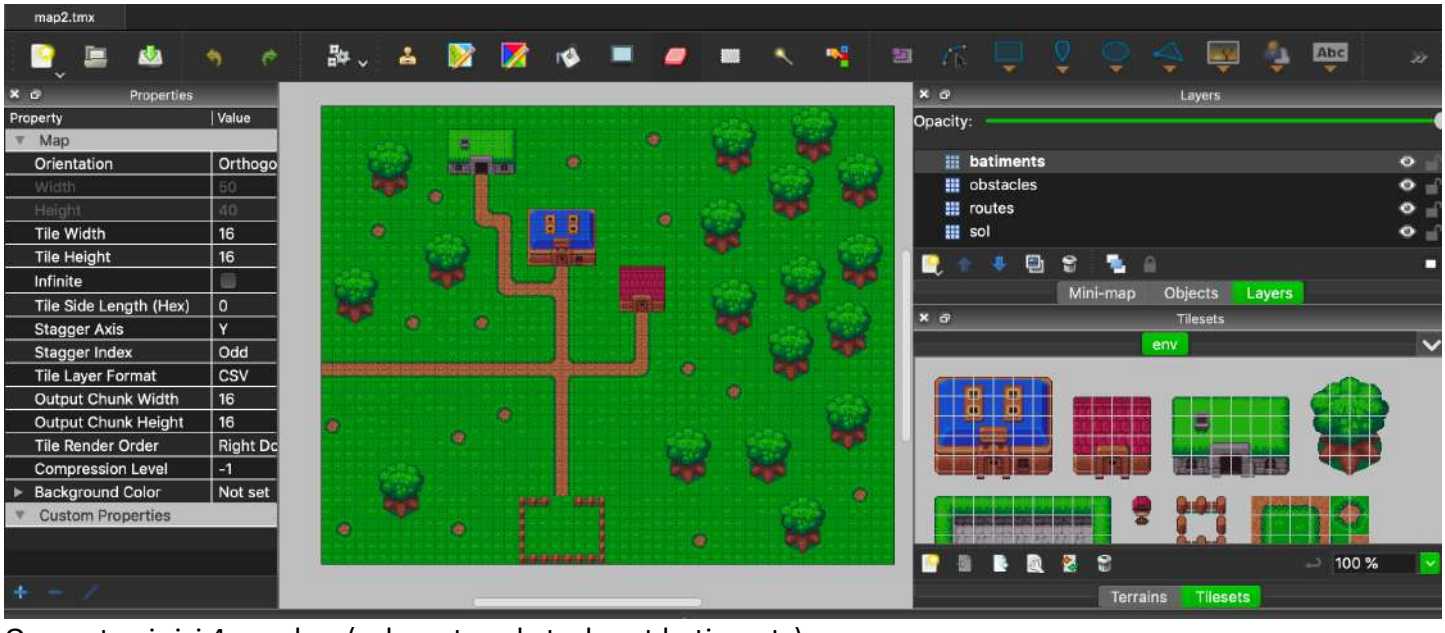
Configurer manuellement le fichier Content.mgcb pour faire une référence aux dll utiles :



Rq : le chemin des dll varie en fonction de l'utilisateur mais par défaut vous les trouverez dans /Utilisateurs/login/.nuget :

```
#------------------------------ References ------------------------------#

/reference:..\..\..\..\..\.nuget\packages\monogame.extended.content.pipeline\3.8.0\tools\MonoGame.Extended.Content.Pipeline.dll
/reference:..\..\..\..\..\.nuget\packages\monogame.extended.content.pipeline\3.8.0\tools\MonoGame.Extended.dll
/reference:..\..\..\..\..\.nuget\packages\monogame.extended.content.pipeline\3.8.0\tools\MonoGame.Extended.Graphics.dll
/reference:..\..\..\..\..\.nuget\packages\monogame.extended.content.pipeline\3.8.0\tools\MonoGame.Extended.Tiled.dll
/reference:..\..\..\..\..\.nuget\packages\monogame.extended.content.pipeline\3.8.0\tools\Newtonsoft.Json.dll
```

On va pouvoir importer un map au format tmx. C'est un format XML pour décrire les différentes couches d'une map. Il existe un logiciel gratuit Tiled qui permet de créer des cartes en vue 2D classique ou 2D isométrique. Il est open source et possède plusieurs formats d'export.

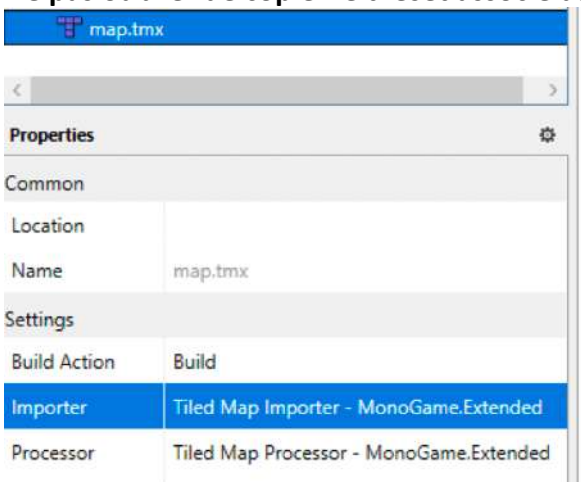


On peut voir ici 4 couches (sol, routes obstacles et batiments).
Ces couches seront utilisées pour la gestion des collisions (voir plus loin).

On va importer map.tmx avec la bonne configuration de build.
Attention le fichier tmx possède la référence au tileset :

```
<image source="tiles_12.png"
```

Ne pas oublier de copier le tileset associé dans le même dossier que le tmx.



Définir les champs suivants :

```
private TiledMap _tiledMap;
private TiledMapRenderer _tiledMapRenderer;
```

avec les using :

```
using MonoGame.Extended.Tiled;
using MonoGame.Extended.Tiled.Renderers;
```

Charger la map :

```
_tiledMap = Content.Load<TiledMap>("map");
_tiledMapRenderer = new TiledMapRenderer(GraphicsDevice, _tiledMap);
```

L'afficher :

```
_tiledMapRenderer.Draw();
```

La mettre à jour :

```
_tiledMapRenderer.Update(gameTime);
```

Ajouter la gestion de la transparence :

```
GraphicsDevice.BlendState = BlendState.AlphaBlend;
```

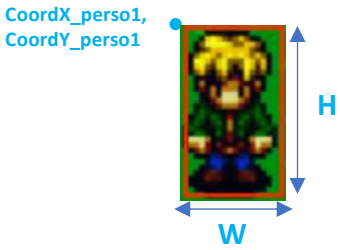
Vérifier l'affichage de la map :

On verra plus tard qu'on peut déplacer cette map en créant une caméra.

4. Collisions

Il existe de nombreux algorithmes pour gérer les collisions, nous allons étudier le plus simple, l'**AABB *Axis Aligned Bounding Box***. Comme son nom l'indique, il s'agit d'un rectangle dont les axes sont alignés avec le repère orthonormé de notre écran.

Une AABB est définie par ses coordonnées x,y (en haut à gauche), sa largeur W et sa hauteur H :

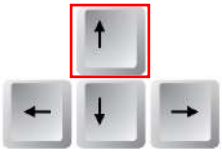


A présent, l'objectif est de localiser l'AABB du personnage par rapport à la grille de tuiles qui composent la carte.
Il faut déterminer si la future position de l'AABB ne sera pas en collision avec un obstacle du décor (couche Obstacle)

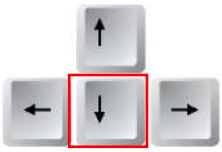
⇨ Si on se déplace vers la droite, il faut tester le coin haut droit le milieu et le coin bas droit de l'AABB :

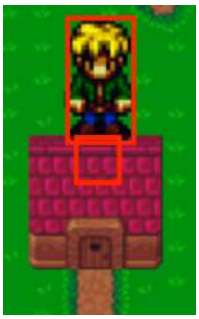
⇨ Si on se déplace vers la gauche, il faut tester le coin haut gauche, le milieu et le coin bas gauche de l'AABB :

⇨ Si on se déplace vers le haut, il faut tester le coin haut gauche et le coin haut droit de l'AABB :



⇨ Si on se déplace vers le bas, il faut tester le coin bas gauche et le coin bas droit de l'AABB :





On va donc définir un objet de type TiledMapTileLayer qui permet de récupérer la couche obstacle :

```
private TiledMapTileLayer mapLayer;

mapLayer = _tiledMap.GetLayer<TiledMapTileLayer>("obstacles");
```

On créé alors une méthode de détection de collision qui teste la couche mapLayer en (x,y)
Elle renvoie Vrai si la tuile en (x,y) n'est pas vide :

```
private bool IsCollision(ushort x, ushort y) {
 // définition de tile qui peut être null (?)
TiledMapTile? tile;
if (mapLayer.TryGetTile(x, y, out tile) == false)
    return false;
if (!tile.Value.IsBlank)
    return true;
return false;
}
```

Il suffit alors de l'utiliser dans la gestion des déplacements du personnage.
Ex pour le déplacement vers le haut :

```
float deltaSeconds = (float)gameTime.ElapsedGameTime.TotalSeconds; // DeltaTime
float walkSpeed = deltaSeconds * _vitessePerso; // Vitesse de déplacement du sprite
KeyboardState keyboardState = Keyboard.GetState();
String animation = "idle";

if (keyboardState.IsKeyDown(Keys.Up))
{
       ushort tx = (ushort)(_persoPosition.X / _tiledMap.TileWidth);
       ushort ty = (ushort)(_persoPosition.Y / _tiledMap.TileHeight - 1); //la tuile au-dessus en y
       animation = "walkNorth";
       if (!IsCollision(tx, ty))
           _persoPosition.Y -= walkSpeed; // _persoPosition vecteur position du sprite
 }
```

Et affichage du sprite animé :
```
_spriteBatch.Draw(_perso, _persoPosition);
```
Vérifiez le bon fonctionnement de la détection de collision sur les éléments du décor.

Détection d'éléments de décors particuliers :
Il est possible de détecter des éléments particuliers du décor en récupérant la valeur de la tuile repérée. Il suffit d'utiliser la propriété :
```
mapLayer.GetTile(x,y).GlobalIdentifier
```
Compléter votre code pour afficher dans la console la détection des portes des maisons.

5. Gestion de scènes

Maintenant qu'on peut détecter les portes des maisons, on va mettre en place le changement de scène pour charger une map correspondant à l'intérieur d'une maison.

Créer un nouveau projet GestionScenes.
Ajouter le package de Monogame.Extended afin d'utiliser les scenes.

Créer un gestionnaire de scènes dans Game1.cs :
```
private readonly ScreenManager _screenManager;
```

Ajouter une propriété dans Game1.cs et modifier le LoadContent() en conséquence :
```
public SpriteBatch SpriteBatch {get; set;}
```

Dans le constructeur de Game1 initialiser ce gestionnaire :
```
_screenManager = new ScreenManager();
Components.Add(_screenManager);
```

Créer une Font de base dans le projet et importer la dans LoadContent

Créer une nouvelle classe nommée MyScreen1.cs héritant de GameScreen :
```
    public class MyScreen1 : GameScreen
    {
        private Game1 _myGame; // pour récupérer le jeu en cours
        private SpriteFont _font;
        public MyScreen1(Game1 game) : base(game) {
            _myGame = game;
         }
        public override void LoadContent()
        {
            _font = Content.Load<SpriteFont>("Font");
            base.LoadContent();
        }
        public override void Update(GameTime gameTime)
        {}
        public override void Draw(GameTime gameTime)
        {
            _myGame.GraphicsDevice.Clear(Color.Red);
            _myGame.SpriteBatch.Begin();
            _myGame.SpriteBatch.DrawString(_font, "Scene 1", new Vector2(350, 200), Color.White);
            _myGame.SpriteBatch.End();
        }
    }
```
Faites de même avec MyScreen2.cs (changer le texte et la couleur d'arrière-plan en rouge dans Draw)

Créer les deux méthodes dans Game1 :

```
private void LoadScreen1()
{
	_screenManager.LoadScreen(new MyScreen1(this), new FadeTransition(GraphicsDevice, Color.Black));
	  }
private void LoadScreen2()
{
	_screenManager.LoadScreen(new MyScreen2(this), new FadeTransition(GraphicsDevice, Color.Black));
	  }
```

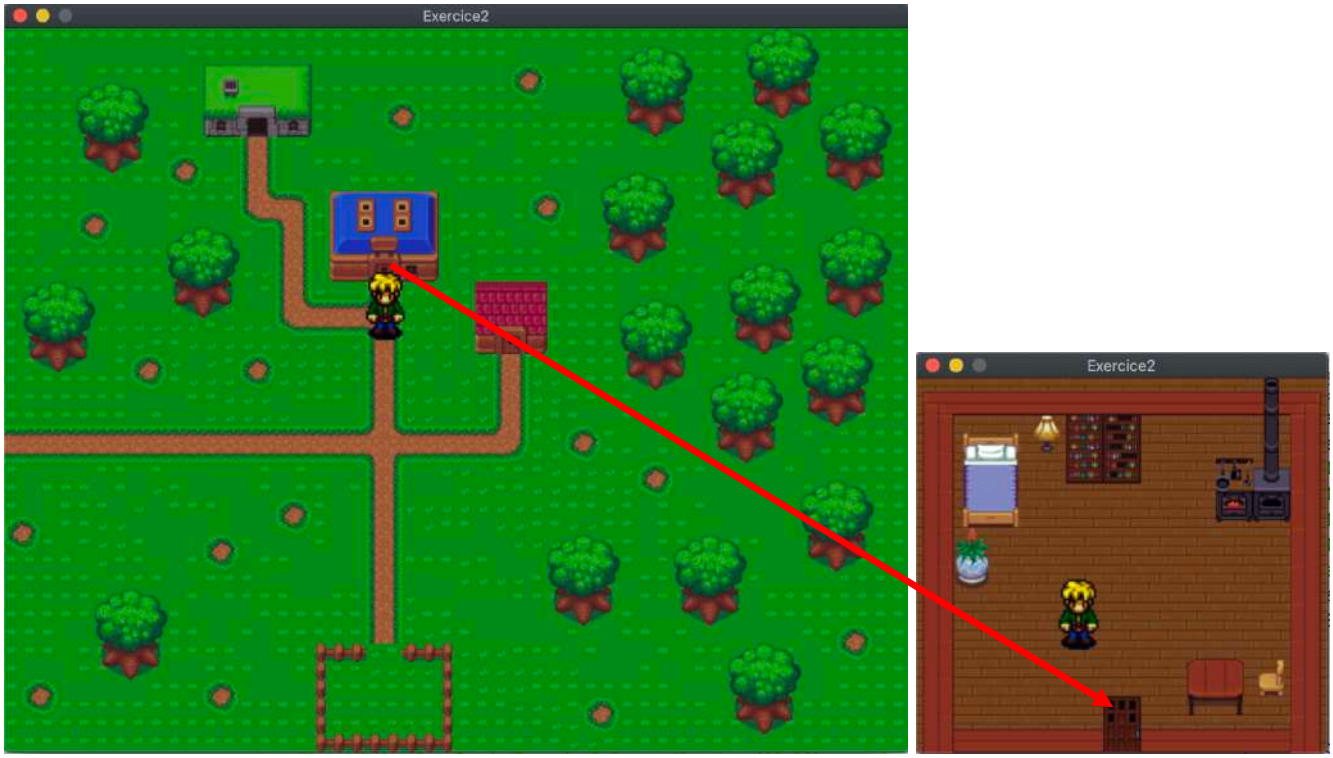
Dans la méthode Update faire appel à ces deux méthodes :

```
KeyboardState keyboardState = Keyboard.GetState();
if (keyboardState.IsKeyDown(Keys.Left))
{
	LoadScreen1();
}
else if (keyboardState.IsKeyDown(Keys.Right))
{
	LoadScreen2();
}
```

Tester les changements de scène avec les flèches gauche et droite.

Application :

Récupérer la map maison_2 et affectez chaque gestion de map dans les scènes associées :
La map principale pour la scène 1 et la map maison_2 pour la seconde maison.



Essayez de factoriser le code **afin de généraliser pour les deux autres maisons (créer un seule classe MyScreen) Évidement il manque les interactions avec d'autres éléments (ennemis, objets à trouver …). A vous d'imaginer la suite !**

6. Publier son jeu

Il s'agit maintenant de publier son jeu sur son OS (soit en ligne de commande soit depuis VisualStudio) :

```
dotnet publish –r win–x64 –c Release
```

Prolongements : Plateforme et Gestion de la caméra

Il est alors simple de créer un jeu de plateforme. La gestion des déplacements et des collisions est juste un peu différente. Il faut notamment gérer la gravité.
Récupérez les ressources des décors pour coder un petit jeu de plateforme statique :

Le personnage est ici en mouvement mais c'est beaucoup plus judicieux de déplacer une caméra et laisser le personnage fixe.
Inspirez-vous du code de la documentation pour améliorer votre code :
https://www.monogameextended.net/docs/features/cameras/cameras/