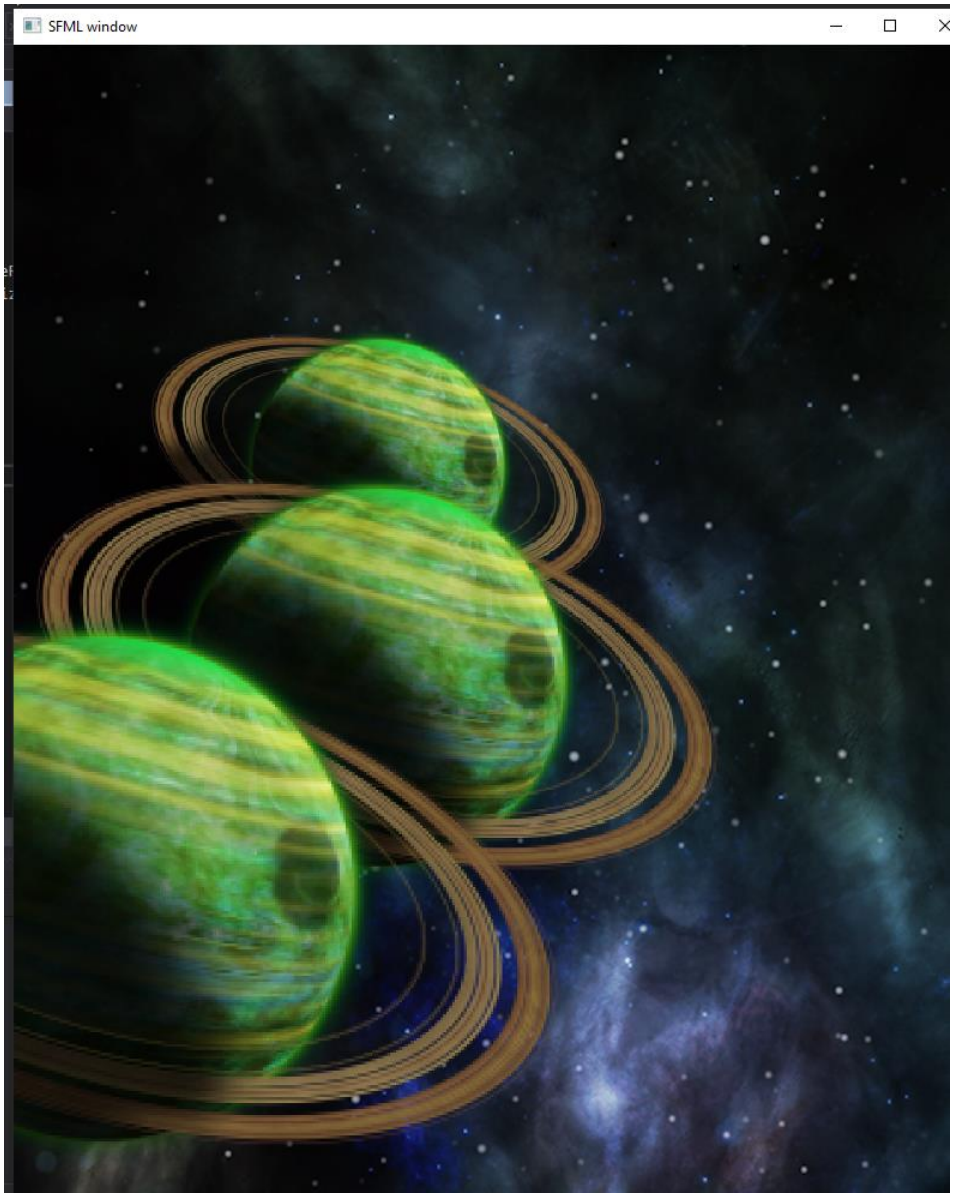
PROJET ASTEROID BELT

Tuto infinite background, parallax objects et système de layers



Introduction

Ce tuto va vous servir à créer une image de fond qui défile à l'infini (ou tout du moins qui en donne l'impression – c'est la magie du cinéma !).

En passant, nous verrons comment créer des objets permettant de créer l'illusion de parallaxe, un effet bien connu utilisé depuis au moins le premier Mario Bros. sur Famicom.

Etape 0 – Lancez le projet Sample.

« First make it work, then make it work good, then make it work fast. »

Le projet Sample est un simple projet vide qui contient une boucle d'inputs vide et un dessin qui affiche un cercle rouge avec la SFML.

Assurez-vous qu'il fonctionne. Il va nous servir de « bac à sable » pour tester les nouvelles features que nous souhaitons ajouter à notre jeu. En effet il peut être difficile d'ajouter de nouvelles choses au milieu de l'architecture complexe du jeu final.

Une pratique courante est donc de tester d'abord cette feature à fond, en isolation, dans un projet de test « quick and dirty » afin de vérifier que notre idée fonctionne, et corriger les bugs inhérents si nécessaire avant de tenter de l'intégrer au reste du projet. C'est aussi ce qu'on appelle un POC (pour Proof Of Concept, ou « preuve que ça marche »).

Le projet sample sera notre étape « First make it work ». Une bonne idée serait de garder une copie « vierge » du projet Sample pour pouvoir tester une nouvelle feature plus tard sans être gêné par le code de tests précédents !

Etape 1 – Infinite background

Pour le projet Asteroids, il faut donner l'impression qu'on se déplace dans l'espace à l'infini.

Pour cela nous allons devoir coder une texture de fond qui se déplace vers le bas de l'écran, puis revient tout en haut : un effet de scroll.

La particularité pour faire fonctionner cet effet est que nous avons besoin d'une texture dite « seamless » (en français : « sans bords »), c'est-à-dire une texture dont le bord gauche est la continuité du bord droit et inversement, et idem pour le bord haut/bas.

Ça tombe bien car nous avons dans nos assets une texture « space_background.png » qui est une texture seamless de haut en bas… et nous faisons un shooter vertical.

Personnellement, j'ai mis les dimensions de la fenêtre à 800,980 pour que ça colle à l'image.

Donc : back to the basics, commençons par créer un sprite de cette texture hors de la boucle de jeu :

```
// Load a sprite to display
sf::Texture texture;
if (!texture.loadFromFile("space_background.png"))
{
    std::cout << "Il y a une couille dans le potage.\n";
    return 1;
}

sf::Sprite background(texture);
```

Ensuite nous aurons besoin de faire bouger cette texture à l'écran.

Pour la faire bouger indépendamment du frame rate, nous allons avoir besoin du delta time.

Une façon très simple de l'obtenir en SFML est d'utiliser une sf ::Clock qu'on restart à chaque tour de boucle.

```
// Start the game loop

sf::Clock clock;

static int scroll_per_sec = 50;

while (window.isOpen())
{
    sf::Time time = clock.restart();
    float dt = time.asSeconds();
    std::cout << dt << std::endl;
```

La VSync étant activée, nous devrions avoir au maximum un frame rate de 60 FPS, soit environ 16 ms par frame.

Votre dt en secondes devrait donc être environ égal à 16 ms, soit 0.01666666 … seconde.

Ensuite nous devons faire bouger cette texture en fonction du delta time. Nous allons donc lui définir une « vitesse de scrolling » en pixels par seconde afin de la faire bouger un peu à chaque

tour de boucle. C'était la variable scroll_per_sec de la capture précédente. Ensuite, dans la boucle :

```
sf::Vector2f pos = background.getPosition();
pos.y += scroll_per_sec * dt;
background.setPosition(pos);
window.draw(background);
```

Si vous lancez le projet à ce moment, vous devriez voir que votre texture descend lentement vers le bas…. Vous pouvez varier la valeur de scroll_per_sec pour vérifier que ça fonctionne bien.

C'est un début, mais ce n'est pas très « infini » tout ça !!

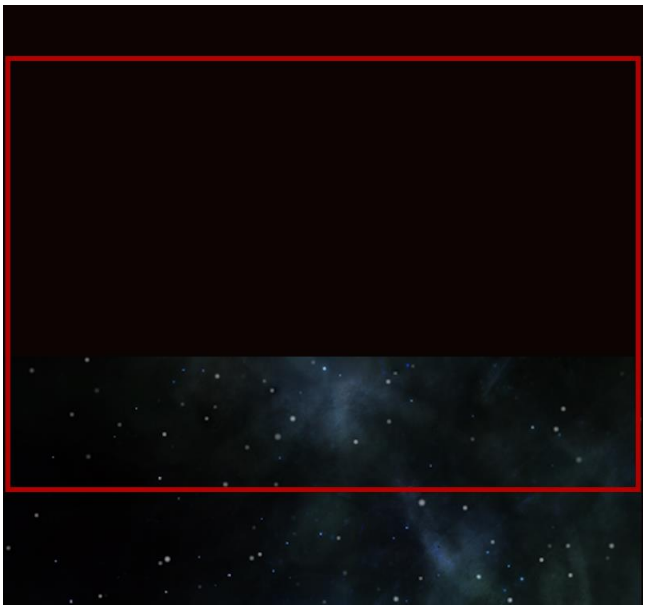
Comment gérer la répétition de la texture à l'infini ?

Il faut réfléchir à la texture comme une feuille de papier qu'on ne ferait que « glisser » le long de notre écran : supposons que notre window soit le cadre rouge :

Frame 1 :

Frame 2 :

Frame 934 :



Du coup, une façon d’arriver à nos fins est de se dire :

« Comme la texture est seamless, il faut que je colle le bas de ma texture au haut de ma texture pour donner l'impression que le haut et le bas se suivent ».

Comment faire ?

Solution 1 : avoir deux sprites

On pourrait tout à fait créer un deuxième sprite pointant sur la même texture.

Nous pouvons obtenir la taille (largeur x hauteur) de notre texture avec :

```
sf::Vector2u size = background.getTexture()->getSize();
```

Avec la largeur dans size.x et la hauteur dans size.y.

Nous pourrions donc créer un deuxième sprite, le placer exactement à la position initiale de notre premier sprite (ici, 0,0) moins <la hauteur de la texture>, et l'avancer et l'afficher exactement comme le premier.

Par exemple si la texture fait 800 par 2400, nous aurions la première texture qui commence en (0,0) et l'autre en (0, -2400)…

Et pour ajouter ce petit côté **infini**, que faudrait-il vérifier ?

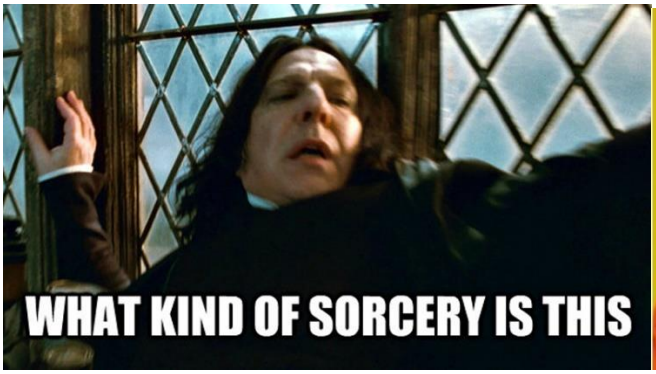
Réponse : le moment où la première texture dépasse le bord bas de l'écran (ou lorsque le y de la deuxième atteint 0). À ce moment-là, il faut « échanger » en mettant la première texture en (0, — <la hauteur de la texture>), c'est-à-dire au-dessus de la deuxième, et permuter comme cela en permanence.

Solution 2 : déplacer notre sprite

Le problème de la solution 1 est qu'elle crée une duplication du sprite, une duplication de la logique de scroll et un code pas très lisible de manière générale.

Ce sont toutes des choses que nous souhaitons éviter.

La bonne nouvelle : il est possible de ne se servir que d'un seul sprite pour l'effet de scroll infini.



Le tricks consiste à se servir de la méthode setTextureRect du sf ::Sprite. Elle permet de délimiter un rectangle dans la texture qui sera la seule « zone » utilisée en cas de draw.

Voyez-vous où on veut en venir ?

Si on reprend une image de plus haut…

Considérez en jaune le rectangle que nous devons remplir et ses dimensions, c'est-à-dire le rectangle situé entre le haut de notre texture (qui est en train de scroller vers le bas) et le bord haut de la fenêtre :

Supposons qu'il fasse… 800 de large par 100 de haut. Ce qui veut dire que notre texture a déjà scrollé 100 pixels vers le bas, et que donc background.getPosition().y vaut 100.

Pour « remplir » ce rectangle, nous allons donc devoir y dessiner le rectangle au bas de notre texture, qui possède, comme coordonnées à l'intérieur de la texture (coordonnées : (x,y, width, height)) :

texRect = (0, size.y – 100, size.x, 100 + 1);

(c'est du pseudo-code.)

Bien sûr les « 100 » sont à remplacer par le y de la position actuelle de background, vu que le sprite continue de scroller vers le bas !

Nous aurons donc :

- Un sprite qui descend
- Et un rectangle du bas de la texture que nous allons afficher au-dessus du sprite. Ce rectangle va grandir encore et encore jusqu'à ce qu'il soit temps de faire « remonter » le sprite !
- Ensuite, lorsque le sprite recommencera à descendre, on va « boucher le trou » en affichant la partie de la texture qui est dans le prolongement du haut de la texture que

nous sommes en train de scroller (et comme c'est une texture seamless, c'est un rectangle de texture qui part du bas !)

Autrement dit, avec une habile combinaison de getTextureRect, setTextureRect et setPosition, et deux dessins de notre unique sprite, on est capable d'afficher une texture qui « a l'air » infinie.

Pour résumer sans pour autant vous spoiler complètement la réponse, voici l'algorithme que vous devez implémenter dans la boucle de jeu :

- Récupérer la position du background
- Récupérer la size du background
- Scroller (changer le y de la pos avec le delta time et la speed)
- Si pos.y >= size.y : L'image a complètement scrollé !
    - Il faut remettre pos.y à 0.
- Mettre à jour la position du sprite avec pos
- Afficher une première fois le sprite si pos.y < (la hauteur de la fenêtre).
    - Le « si pos.y < (la hauteur de la fenêtre) » est optionnel mais autant ne pas afficher une image qui est hors de l'écran ….
- Si pos.y > 0 : il y a un « trou » entre le haut de notre sprite et le haut de notre fenêtre !
    - 1) Récuperez le textureRect courant du sprite.
    - 2) Settez un nouveau texture rect du sprite à (0, size.y – pos.y, size.x, pos.y + 1)
    - 3) Mettez le background en (0,0) (temporairement !!)
    - 4) Dessinez de nouveau le background (mais cette fois-ci « coupé » par le textureRect pour remplir le trou)
    - 5) Puis, remettez le textureRect au textureRect que nous avons lu en 1)…
    - 6) et remettez la position du sprite à pos.

Normalement, vous avez fini et votre texture scroll maintenant à l'infini.

Etape 2 – Parallax Objects

Maintenant que nous avons le background infini, profitons-en pour ajouter quelques objets qui apparaîtront dans le fond de votre jeu, sans rapport avec le gameplay, juste pour le décor.

L'effet de parallaxe est une illusion d'optique bien connue que vous avez forcément expérimenté au moins une fois. Grosso modo, elle signifie que lorsque nous sommes en mouvement, les objets plus proches de nous nous paraissent « aller plus vite » que ce qui est au loin. Pourtant, ni les objets près, ni les objets lointains ne se déplacent… C'est nous qui bougeons !

Voir vidéo exemple.

Pour implémenter cet effet dans notre jeu, nous allons devoir programmer un système simple de **layers**, ou « couches ». Séparer nos objets en layers permet de ne pas se tromper d'ordre d'affichage : en effet, la logique voudrait que nous affichions dans l'ordre :

- Le background
- Les objets de décor lointains
- Les objets de décor proches
- Les game objects (player, asteroids, powerups…)
- Le HUD et l'UI (les scores, les vies…) (vous pouvez séparez les deux si vous le souhaitez)

Les objets sujets à l'illusion de parallaxe seront définis par une « proximité » fictive avec le joueur (nous sommes en 2D, mais nous pouvons imaginer qu'il y a une notion de profondeur). En fonction de cette proximité, nous irons modifier :

- Leur scale : si ils sont plus près, ils ont des chances d'apparaître plus gros
- Leur vitesse de défilement à l'écran : plus les objets seront lointains, moins ils iront vite.

Créez donc une petite classe ParallaxObject (dans notre « vrai » jeu, elle devra être un peu différente, mais dans notre Sample, on fait du quick and dirty). Nous allons également nous créer quelques constantes utiles pour éviter de manipuler des nombres magiques.

Aussi, créez vous dès maintenant une enum class, qui vous resservira aussi plus tard mais dont nous allons nous servir ici :

```
enum class DrawLayer : char
{
    BACKGROUND = 0,
    LANDSCAPE_VERY_FAR,
    LANDSCAPE_FAR,
    LANDSCAPE_NEAR,
    GAME_OBJECT,
    HUD,
    NONE
};
```

```
class ParallaxObject
{
public:
    ParallaxObject(DrawLayer layer, const sf::Texture& objTex);

    void    DrawOn(sf::RenderWindow& win);

    bool    Update(float dt);

private:
    sf::Sprite  m_sprite;

    float   m_speed = ms_nearObjectSpeed; // in pixels per second
    DrawLayer   m_layer = DrawLayer::LANDSCAPE_NEAR;

    static float    ms_nearObjectScale;
    static float    ms_nearObjectSpeed;

    static float    ms_farObjectScale;
    static float    ms_farObjectSpeed;

    static float    ms_veryFarObjectScale;
    static float    ms_veryFarObjectSpeed;
};

float   ParallaxObject::ms_nearObjectScale = 1.f;
float   ParallaxObject::ms_nearObjectSpeed = 12.f;

float   ParallaxObject::ms_farObjectScale = 0.75f;
float   ParallaxObject::ms_farObjectSpeed = 9.f;

float   ParallaxObject::ms_veryFarObjectScale = 0.5f;
float   ParallaxObject::ms_veryFarObjectSpeed = 6.f;
```

(vous pouvez mettre les static const, au passage)

Puis, dans le constructeur on va juste aller setter la scale et la speed de notre parallax object en fonction du layer qu'on nous a passé. On s'entend que pour un parallax object, seules trois valeurs sont acceptables : LANDSCAPE_{NEAR,FAR,VERY_FAR}. C'est pourquoi j'utilise un assert qui sautera dès qu'on lui passe une autre valeur. On ne sait jamais ! (n'oubliez pas #include <cassert>)

```cpp
ParallaxObject::ParallaxObject(DrawLayer layer, const sf::Texture& objTex) :
    m_layer(layer), m_sprite(objTex)
{
    switch (m_layer)
    {
    case DrawLayer::LANDSCAPE_NEAR:
        m_sprite.setScale(ms_nearObjectScale, ms_nearObjectScale);

        m_speed = ms_nearObjectSpeed;
        break;
    case DrawLayer::LANDSCAPE_FAR:
        m_sprite.setScale(ms_farObjectScale, ms_farObjectScale);

        m_speed = ms_farObjectSpeed;
        break;
    case DrawLayer::LANDSCAPE_VERY_FAR:
        m_sprite.setScale(ms_veryFarObjectScale, ms_veryFarObjectScale);

        m_speed = ms_veryFarObjectSpeed;
        break;
    default:
        assert(false); // wrong value
    }
}

inline void ParallaxObject::DrawOn(sf::RenderWindow & win)
{
    win.draw(m_sprite);
}

inline bool ParallaxObject::Update(float dt)
{
    sf::Vector2f pos = m_sprite.getPosition();
    pos.y += (m_speed * dt);
    m_sprite.setPosition(pos);

    return false;
}
```

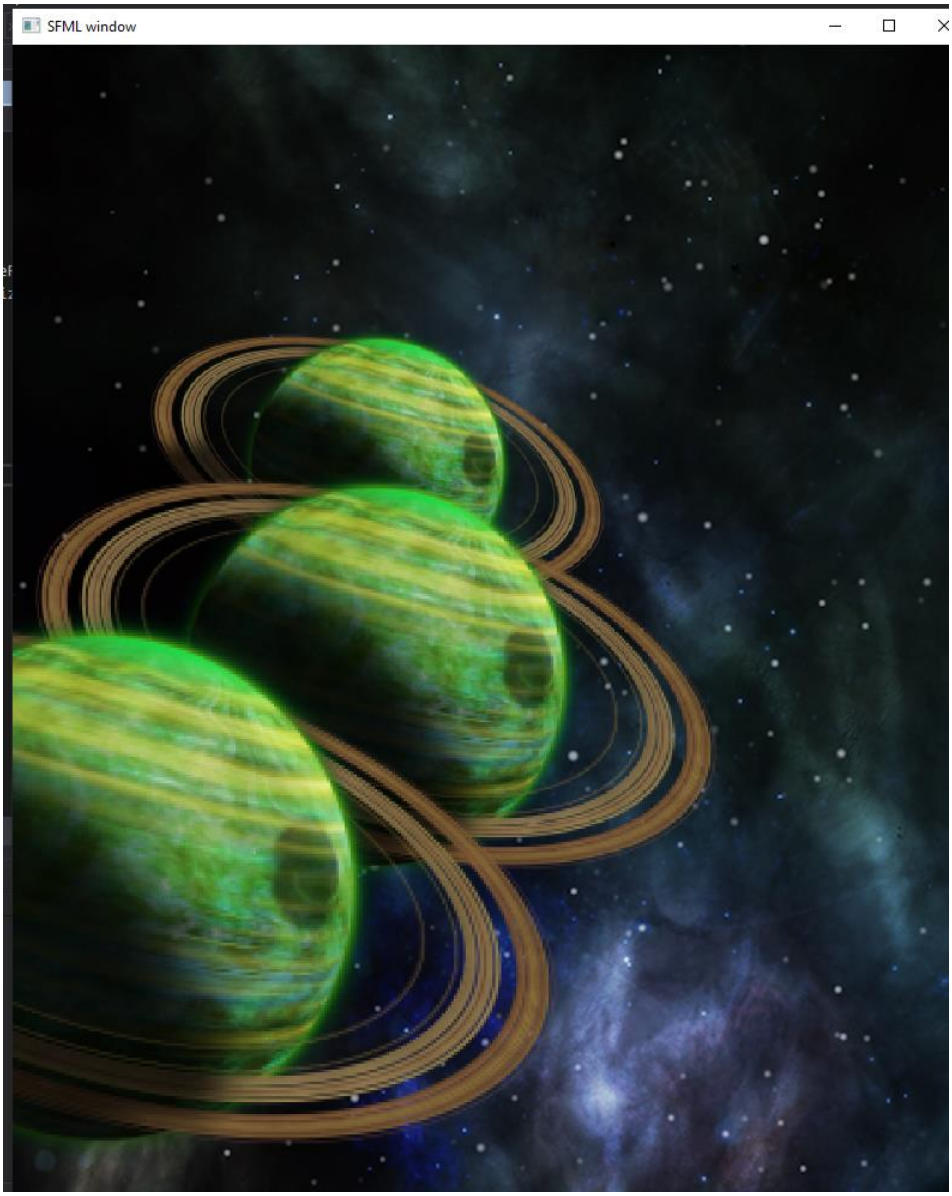
(Vous pouvez aussi utiliser m_sprite.move(0, (m_speed *dt)) pour aller plus vite.)

Essayez maintenant d'afficher 2-3 objets sur des layers LANDSCAPE différents. Donnez-leur des positions négatives en y afin qu'ils donnent l'impression de venir de l'espace au-dessus de l'écran.

Prenez dans le dossier des assets parallax l'image hyperion_pix.png par exemple. Ses dimensions sont 800x522. Vous devrez donc le mettre au moins en -522 en y. Utilisez Update et DrawOn pour afficher vos planètes. Faites attention à dessiner vos planètes APRÈS le background !!

Et, pour commencer, veillez à ce que les objets dessinés soient dans le bon ordre, mais pour le moment faites-le à la main. Nous verrons comment ordonnancer le rendu dans un autre tuto…

Normalement, cela devrait fonctionner : Les objets les plus près, donc les plus gros, seront aussi ceux qui vont le plus vite !



Ensuite, vous pourrez utiliser du random pour générer un x aléatoire entre —(largeur sprite / 2) et (largeur de la fenêtre - (largeur sprite/2)), pour les faire parfois sortir un peu de l'écran, dans le constructeur de ParallaxObject.

Pour info, une méthode en C++ pour choisir un nombre float random entre deux valeurs :

#include <random>

```
std::random_device rd; // obtain a random number from hardware
std::mt19937 eng(rd()); // seed the generator
std::uniform_real_distribution<> distr(-(int)m_sprite.getTexture()-
>getSize().x / 2, 980 - (m_sprite.getTexture()->getSize().x / 2));

float randomX = (float)distr(eng);
```

Puis, vous pourrez vous ajouter deux valeurs static float à ParallaxObject :

- ms_minSpawnTimer et ms_maxSpawnTimer (en secondes)

qui vous serviront à générer un nouveau au bout d'un temps random choisi entre ces deux valeurs (qui devront être configurables également).

Dans le cadre de votre vrai jeu, il ne faudra pas non plus oublier de détruire les objets parallax dès que leur y sera supérieur à la hauteur de la fenêtre.

Enfin, rappelez-vous que dans le vrai jeu, vous n'êtes pas censé utiliser ce genre de statiques, mais les valeurs fournies par le fichier de config.

Bravo, vous avez maintenant des objets de parallaxe qui se déplacent en fonction de leur distance et un scrolling background infini.

Final touch : changer la couleur du background

Attention ! Cette étape commence à être plus avancée et il vaut donc mieux revenir la faire quand votre projet est un peu plus avancé (c'est-à-dire : quand le passage d'un « level » à un autre fonctionne).

On aimerait donner au joueur une indication visuelle qu'il a changé de niveau.

Changer la teinte du background est une option.

Pour ce faire, on va utiliser une **interpolation linéaire**, aussi appelée linear interpolation ou **Lerp**.

L'idée de base :

- on détecte un changement de niveau
- on enclenche un chronomètre tournant la durée de la phase de transition vers un autre niveau (normalement lue depuis la config), et une variable float « time_since_level_change » initialisée à 0.
- On n'oublie pas non plus de stocker quelque part la couleur actuelle du background, et on va venir choisir dans un pool de couleurs (au moyen d'un simple index qui va cycler sur la taille du pool, par exemple), la couleur vers laquelle on veut que le background aille.
- pendant que le chronomètre tourne, à chaque tour de boucle : on va incrémenter time_since_level_change de la valeur du delta time (en cappant à la durée du chronomètre). Ensuite, on divise time_since_level_change par la durée totale du chrono ; logiquement, on obtient un ratio qui est **une valeur flottante entre 0 et 1.**

- **C’est de cette valeur dont on va se servir pour l’interpolation des couleurs ;** il vous suffit ensuite de faire un appel à ma fonction LerpColors en passant en paramètre :
- **La couleur d’origine**
- **La couleur destination**
- **Le ratio**
- Pour qu’il vous retourne une nouvelle couleur qui va progressivement tendre vers la couleur destination (et qui deviendra vraiment la couleur destination au bout de la durée du timer). Vous venez de faire une interpolation.
- Arrêtez d’interpoler lorsque vous avez atteint votre couleur destination.

Je vous fournis les fonctions suivantes, à vous de vous en servir correctement :

```
float   Lerp(float from, float to, float ratio)
{
    return (to - from) * ratio + from;
}
sf::Color   LerpColors(const sf::Color& srcColor, const sf::Color& destColor, float ratio)
{
    return sf::Color(
        (sf::Uint8)Lerp(srcColor.r, destColor.r, ratio),
        (sf::Uint8)Lerp(srcColor.g, destColor.g, ratio),
        (sf::Uint8)Lerp(srcColor.b, destColor.b, ratio),
        (sf::Uint8)Lerp(srcColor.a, destColor.a, ratio));
}
```