Introduction à la librairie SFML
Projet programmation 2

Vincent Lafeychine

1er février 2024

Ce document vous propose une préparation de votre projet sur GitLab, ainsi qu'une introduction à la librairie graphique SFML accompagnée du *binding* SFML pour Scala, nommé Scala Native SFML.

Avant de continuer, vous devez avoir un compte GitLab auprès du Crans et y être connecté. Si cela n'est toujours pas fait, veuillez vous inscrire sur le site intranet du Crans puis envoyer un mail à `pigeonmoelleux@crans.org` pour demander l'accès au GitLab tout en précisant votre pseudo.

Table des matières

0 Installation des dépendances

Avant de commencer, veuillez vous assurer que vous avez toutes les dépendances nécessaires pour faire le projet. Choisissez la sous-section qui correspond à votre cas.

Il est à noter que les ordinateurs de la salle 1S53 sont les environnements de référence : Votre projet devra pouvoir être lancé sur ces ordinateurs et c'est sur ces ordinateurs que vos jeux seront testés.

0.1 Script d'installation pour les ordinateurs de la salle 1S53

Afin de vous permettre de trouver vos fichiers sur n'importe quel ordinateur de la salle informatique, les ordinateurs se connectent sur un disque commun accessible en réseau, en utilisant *Network File System*. Cependant, la mise en réseau de vos répertoires influe sur les performances des accès lecture/écriture.

Ainsi, il est conseillé de lancer le script ci-dessous, ce qui vous permet de contourner ces restrictions. Aucune action n'est à effectuer par la suite, vous pourrez utiliser `sbt` comme avant mais en vous assurant que vous lancez qu'une seule instance d'`sbt` à la fois !

▶ *Copiez ce script, en vérifiant que les caractères ont été copiés correctement, puis exécutez-le.*

Le script ci-dessous n'est *pas* idempotant, veillez à ne le lancer qu'une seule fois :

```
#!/usr/bin/env bash

mkdir -p ~/.local/bin

echo 'PATH=~/.local/bin:${PATH}' >> ~/.bashrc
echo ". ~/.bashrc" >> ~/.bash_profile

cat <<'EOF' >| ~/.local/bin/sbt
#!/usr/bin/env bash

DIR=/dev/shm/$(whoami)

function clean() {
    rm -rf ${DIR}
    rm target
}

trap 'clean; exit 1' EXIT HUP INT QUIT PIPE TERM

clean

mkdir -p ${DIR}/target
ln -s ${DIR} target

/usr/bin/sbt \
    -Dsbt.global.base=${DIR}/sbt \
    -Dsbt.boot.directory=${DIR}/boot \
    -Dsbt.coursier.home=${DIR}/coursier \
    -Dsbt.ivy.home=${DIR}/ivy $@
EOF

chmod +x ~/.local/bin/sbt

echo "Install finished. Please close your terminal for the changes to take effect."
```

Attention, le dossier `target/` sera supprimé dès que vous quitterez `sbt`.

0.2 Windows

Le projet n'est malheureusement *pas* compatible avec votre environnement nativement.

Cependant, vous pouvez utiliser *Windows Subsystem for Linux* (*WSL*) afin d'avoir un système virtuel GNU/Linux, supporté officiellement par *Microsoft*.

Vous devez cependant avoir un système récent de *Microsft Windows*, ainsi mettez à jour votre système, puis suivez ce tutoriel en laissant la distribution proposée (Ubuntu), puis redémarrez votre ordinateur.

Pour vérifier que l'instance Ubuntu affiche correctement les fenêtres graphiques, lancez `xclock` :

```
$ sudo apt update && sudo apt install x11-apps
$ xclock    # Si aucune fenêtre s'affiche. Veuillez refaire les étapes d'installation de `WSL'.
```

Maintenant que *WSL* est installé, regardez maintenant la section d'installation pour GNU/Linux.

0.3 MacOS

Le projet n'est malheureusement *pas* compatible avec votre environnement nativement.

Veuillez utiliser l'une des solutions suivantes :

- les ordinateurs de la salle 1S53 via un tunnel `SSH` avec le support `X11`-Forwarding (`XQuartz`) ;
- les ordinateurs de la salle 1S53 directement, ou ;
- sur votre système, installez une machine virtuelle GNU/Linux.

0.4 GNU/Linux

Si vous avez votre propre ordinateur sous GNU/Linux, il vous faudra installer certains paquets. Les noms donnés ci-dessous sont ceux pour Debian (et de ses dérivés, comme Ubuntu) :

- `git lfs` (fvoir Section 3.3) : `git-lfs` puis lancez la commande `git lfs install` ; (repology)
- Une *Java Virtual Machine* (`jdk` ou `jre`) en version 11+ : `openjdk-17-jre-headless` ; (repology)
- `sbt` en version 1.0.0+ : Suivez ce guide pour les distributions basées sur Debian ; (repology)
- La toolchain LLVM en version 14+ : `clang`, `libclang-rt` et `lld`, et ; (repology)
- La librairie graphique SFML en version 2.5.1 *spécifiquement* : `libsfml-dev`. (repology)

Pour être sûr que tout soit bien installé, vous pouvez copier l'exemple suivant dans le fichier `sfml.cpp` :

```cpp
#include <SFML/Graphics.hpp>

int main() {
    sf::RenderWindow window(sf::VideoMode(800, 600), "SFML window");
    window.setFramerateLimit(30);

    for (int index = 0; window.isOpen() && index <= 30; index++) {
        window.display();
    }
    window.close();
}
```

Compilez le programme en utilisant la ligne ci-dessous. Vous devriez obtenir un exécutable nommé `sfml`, qui ouvre une fenêtre noire pendant une seconde avant de se refermer automatiquement.

```
$ clang++ sfml.cpp -fsanitize=leak -lsfml-graphics -lsfml-system -lsfml-window -fuse-ld=lld -o sfml
```

1 Votre projet sur GitLab

Durant votre projet, vous allez devoir travailler en groupe et donc avoir un moyen d'échanger entre vous vos modifications. Pour cela, vous allez utiliser `git` qui permet de faire des points de sauvegarde de votre projet afin de pouvoir le partager et facilement contribuer.

Même si cette séance n'a pas pour objectif de vous apprendre `git`, elle vous permettra de préparer votre projet sur GitLab, un serveur pour déposer et héberger des points de sauvegardes.

1.1 Création de votre projet sur GitLab

Même si vous êtes plusieurs dans un même groupe, chacun d'entre vous doit être capable de maîtriser la plateforme. Ainsi, même si un seul compte GitLab aura le projet final, il est conseillé que vous fassiez tous ce tutoriel, quitte à ce qu'il y ait des projets doublons pour l'instant.

▶ *Connectez-vous à votre compte, puis allez sur la page de création d'un nouveau projet vierge.*

Vous allez devoir choisir le nom de votre projet. Pas de panique, vous pourrez le changer plus tard :

▶ *Ajoutez un nom de projet, décochez l'initialisation de projet avec un* ***`README`****, puis créez votre projet.*

Vous êtes maintenant sur la page GitLab de votre projet, vide pour l'instant. Suivez chaque instruction sur cette page afin de correctement configurer votre projet.

Dans un premier temps, vous allez devoir travailler à plusieurs sur ce projet :

▶ *Ajoutez chaque membre de votre équipe en leur assignant le rôle de mainteneur.*

Maintenant, identifiez vos futurs points de sauvegardes grâce à votre nom (ou pseudonyme) et votre adresse mail. Des valeurs suivant votre profil vous sont proposés, mais vous êtes libre de les choisir. Cependant, il est préférable que ces valeurs ne changent pas dans le temps. Ainsi, il est pertinent de renseigner « Prénom Nom » ainsi qu'une adresse mail permanente :

▶ *Configurez* ***`git`*** *en renseignant les clés* ***`user.name`*** *et* ***`user.email`*** *de manière globale (*‘***`--global`***’).*

1.2 Synchronisation locale de votre projet

Pour pouvoir contribuer au projet sur GitLab, il va être nécessaire de récupérer une version locale du projet, et cela grâce à `git` et sa commande `clone`. Cette commande va dupliquer le contenu de votre projet sur GitLab dans un dossier ayant pour nom le *slug* de votre projet.

▶ *Recopiez la commande commençant par* ***`git clone`*** *et déplacez-vous dans le dossier créé :*

```
$ git clone https://gitlab.crans.org/<votre-pseudonyme>/<slug-de-votre-projet>.git
$ cd <slug-de-votre-projet>
```

▶ *Listez l'ensemble des fichiers de ce dossier avec* ***`ls`*** *et le flag* ***`a`*** *:*

```
$ ls -a
.  ..  .git
```

Lorsque vous avez effectué la commande `git clone`, un dossier nommé `.git` a été créé et permet à `git` de fonctionner. Veillez à ne *pas* supprimer ce répertoire ou vous serez contraint d'effectuer un `git clone` de nouveau.

1.3 Téléchargement du template du projet

Pour ce cours, un template vous est fourni avec l'ensemble des outils que vous allez utiliser lors de votre projet. Chaque fichier présent sera expliqué dans la section 3.

▶ *Pour récupérer ce template, cherchez le projet* ***`template-prog-2`*** *de l'utilisateur* ***`v-lafeychine`****.*

Pour pouvoir télécharger le code source d'un projet, il suffit de se rendre sur la page d'un projet et de cliquer sur le bouton « Code » en bleu pour afficher la liste des choix disponibles.

▶ *Depuis le dossier de votre projet, téléchargez le code source avec* ***`git clone`*** *en utilisant le protocole* ***`HTTPS`*** *(l'adresse devrait commencer par* ***`https://gitlab.crans.org`****).*

Vérifions que tout se passe correctement dans le dossier de votre projet en listant son contenu :

```
$ ls -a
.  ..  .git  template-prog-2
```

Vous avez maintenant le template dans un dossier de votre projet. La façon dont nous procédons pour télécharger le template n'est pas recommandée et nous demande maintenant de déplacer des dossiers. Cependant, il est essentiel de manipuler la ligne de commande de `git`, notamment `clone` aujourd'hui.

Finissons notre travail et installons correctement le template :

```
$ rm -rf template-prog-2/.git    # Suppression du dossier `.git' du dossier `template-prog-2'
$ cp -rT template-prog-2 .       # Copie du dossier `template-prog-2' (sans `.git', car supprimé)
$ rm -rf template-prog-2         # Suppression du dossier `template-prog-2'
```

Vérifions encore que tout se passe correctement dans le dossier de votre projet en listant son contenu :

```
$ ls -a
.  ..  build.sbt  .git  .gitattributes  .gitignore  .gitlab-ci.yml  project  .scalafmt.conf  src
```

Tout est maintenant installé ! Il ne reste plus qu'à utiliser le template :

▶ *Ouvrez* ***`sbt`*** *et lancez le projet avec* ***`run`****, puis vérifiez que vous avez bien une fenêtre qui s'ouvre avec le logo de la* ***`SFML`*** *et le texte « Hello SFML » qui s'affichent. Si ce n'est pas le cas, manifestez-vous !*

Dans le fichier `build.sbt`, pensez à changer le nom de votre projet et sa version après chaque rendu :

```
3  name := "MyProject"
4  version := "0.1.0"
```

1.4 Synchronisation distante de votre projet

Maintenant que votre dossier local est prêt, vous pouvez envoyer votre projet sur GitLab. Pour cela, veuillez écrire ces commandes, elles vous seront expliquées lors de la prochaine séance :

```
$ git add .
$ git commit -m "Initial commit"
$ git push
```

Si vous retournez sur la page GitLab de votre projet, les fichiers de votre dossier ont été publiés.

▶ *Après la séance sur* ***`git`****, n'hésitez pas à explorer l'interface GitLab de votre projet.*

2 La librairie graphique SFML

SFML est une bibliothèque graphique écrite en C++, vous proposant de créer de jeux à travers une couche d'abstraction simple, mais assez puissante pour créer des chefs-d'œuvre comme MarbleMarcher.

Afin de pouvoir utiliser la SFML en Scala, il va falloir utiliser un *binding* qui permet de faire le lien entre les deux langages. Cependant, Scala repose sur la *Java Virtual Machine* (*JVM*) pour s'exécuter, ce qui empêche l'utilisation de tel *binding* pour communiquer avec d'autres langages.

Ainsi, vous allez utiliser Scala Native afin de s'affranchir de la *JVM* en compilant le code Scala en un exécutable natif, tel que le font d'autres langages comme C, Haskell, OCaml ou Rust.

Le *binding* Scala Native SFML comporte également une documentation de son API et une série de tutoriels, transposée depuis la documentation officielle C++, utilisant la syntaxe Scala :

- Les tutoriels : `https://lafeychine.codeberg.page/scala-native-sfml/docs/index.html`
- L'API : `https://lafeychine.codeberg.page/scala-native-sfml/index.html`

Il est à préciser que le projet du *binding* est encore en développement et que certaines fonctionnalités ne sont pas encore portées. Si vous rencontrez des problèmes, vous pouvez ouvrir une issue sur le dépôt ou me contacter directement sur Discord (`lafeychine`) : Je réponds généralement en moins de 24 heures.

2.1 Découverte de la librairie SFML : Initialisation

Comme vu lors de la précédente séance, vos fichiers Scala doivent être placés le dossier `src/main/scala`. Le template installé comporte un fichier `Main.scala` que l'on va analyser dans cette section.

Dans un premier temps, nous allons avoir besoin de récupérer des fonctions présentes dans les modules `graphics` et `window` de la SFML, importons les maintenant avec le mot-clé `import` :

```
3  import sfml.graphics.*
4  import sfml.window.*
```

Définissons maintenant la fonction `main` avec l'annotation `@main`, notre fonction principale :

```
8  @main def main =
```

Créons une fenêtre avec le constructeur `RenderWindow` qui prends la taille avec `VideoMode` et un titre :

```
11 val window = use(RenderWindow(VideoMode(800, 600), "Hello SFML"))
12 window.framerateLimit = 30
```

▶ *Le constructeur `RenderWindow` est polymorphe : Il existe différentes signatures pour le même nom. En changeant les paramètres de `RenderWindow`, créez une fenêtre qui ne peut être redimensionnée.*

Chargeons maintenant notre première image dans une `Texture` avec la méthode `loadFromFile`, en définissant au préalable le répertoire des ressources, relatif depuis le dossier où `sbt` est exécuté :

```
6  val RESOURCES_DIR = "src/main/resources/"
```

```
15 val texture = use(Texture())
16 if !(texture.loadFromFile(RESOURCES_DIR + "SFML.png")) then System.exit(1)
```

Similairement, chargeons une police d'écriture dans une `Font` avec la méthode `loadFromFile` :

```
21 val font = use(Font())
22 if !(font.loadFromFile(RESOURCES_DIR + "FiraSans.ttf")) then System.exit(1)
```

Une fois une texture chargée, il est possible d'en faire un `Sprite` qui est un élément `Drawable` :

```
18 val sprite = Sprite(texture)
```

▶ *La classe `Sprite` hérite également de `Transformable` et essayez d'afficher le logo à l'envers. Comprenez-vous ce que vous obtenez ? Lisez l'explication sur la composante `origin` dans le tutoriel sur les transformations.*

Similairement, une fois une police d'écriture chargée, il est possible d'en faire un `Text` :

```
24 val text = use(Text("Hello SFML", font, 50))
```

▶ *Tout comme `Sprite`, `Text` est `Drawable` : Quelles sont les autres classes `Drawable` ? Pour obtenir cette information, allez sur la page de `Drawable` et cherchez 'Graph' dans la catégorie des attributs.*

2.2 Découverte de la librairie SFML : Boucle de jeu

Maintenant que tout est prêt, nous pouvons nous intéresser à la boucle principale. Il est important de correctement construire les étapes de cette boucle afin de faciliter le développement. Suivre ces étapes n'est pas obligatoire, mais permet de séparer la logique de votre application.

Par exemple, il paraît contre-intuitif que la fonction `nextMove` qui calcule le prochain déplacement d'un ennemi requiert l'accès à la classe `RenderWindow`. Il faudrait mieux que cette fonction `nextMove` modifie un attribut public `position` de la classe `Enemy`, qui sera finalement récupérée par la fonction s'occupant de l'affichage qui, elle, aura accès à `RenderWindow`.

Ainsi, les prochaines sections effectuerons la séparation suivante :

- Gestion des événements : Récupération des événements extérieurs du joueur (clavier, souris...) ;
- Gestion de la logique de jeu : Itération du jeu, influencé par les interactions du joueur, et ;
- Gestion de l'affichage : Affichage des éléments suivant leur position dans le jeu.

Avec cette séparation, nous obtenons le pseudo-code suivant :

```
val engine = Engine(...)
val game = Game(...)
val window = RenderWindow(...)

while window.isOpen() do                    // Tant que la fenêtre n'est pas fermée:
    engine.readEvents(window.pollEvent())   //     Gestion des événements, sans utiliser `game'
    engine.tickUpdate(game)                 //     Gestion de la logique, sans utiliser `window'
    engine.render(game, window)             //     Gestion de l'affichage
```

Dans cet exemple, une séparation à été faite entre le « jeu » (`Game`), définissant la logique, du « moteur de jeu » (`Engine`), s'occupant de faire fonctionner la logique en proposant des fonctions utilitaires (comme ce que propose Godot par exemple).

Vous êtes cependant libre de choisir un autre découpage ou même d'autres interactions avec les objets : Il peut être justifiable, par exemple, que la classe `Game` soit un attribut de la classe `Engine`.

2.2.1 Boucle principale de jeu

Commençons avec la méthode `isOpen`, où votre logique de jeu doit impérativement se faire dans le corps de cette boucle, la SFML nécessitant l'appel de cette fonction périodiquement pour fonctionner :

```
27 while window.isOpen() do
28     ...
```

La méthode `isOpen` sur une fenêtre correctement initialisée ne renverra `false` uniquement après la fermeture explicite de la fenêtre via la méthode `close`.

2.2.2 Gestion des événements

Traitons maintenant les `Event` qui peuvent intervenir sur la fenêtre de jeu avec la méthode `pollEvent`. Dans cet exemple, nous traitons uniquement les événements de fermeture et de redimensionnement :

```
29 for event <- window.pollEvent() do
30     event match
31         case Event.Closed()               => window.close()
32         case Event.Resized(width, height) => window.view = View((0, 0, width, height))
33         case _                            => ()
```

▶ *Supprimez la ligne 31 sur la fermeture de fenêtre et essayer de fermer la fenêtre : Qu'observez-vous ? Appuyer sur `Ctrl + C` dans `sbt` si besoin. Relisez la section précédente si nécessaire.*

▶ *Supprimez la ligne 32 sur le redimensionnement de fenêtre et observez le changement lorsque vous redimensionnez la fenêtre. Lisez ensuite la partie sur le redimensionnement du tutoriel sur les vues.*

Le redimensionnement est une logique complexe à gérer : Il est tout à fait raisonnable, voir même conseillé, de n'offrir que quelques choix de résolution (comme `800x600`, `1024x768`, `1280x720`...) et de laisser la SFML élargir le contenu de la fenêtre si besoin.

2.2.3 Gestion de la logique de jeu

Dans notre exemple très simpliste, il n'y a aucune interaction avec le joueur et donc aucun code équivalent. Cependant, c'est dans cette section que votre la magie de votre jeu va opérer !

2.2.4 Gestion de l'affichage

Comme le redimensionnement est pris en compte, actualisations les positions pour centrer nos éléments :

```
36 sprite.position = (window.size.x / 2 - sprite.globalBounds.width / 2, 100f)
37 text.position = (window.size.x / 2 - text.globalBounds.width / 2, 400f)
```

D'ailleurs, nous voyons ici un potentiel problème de découpage : Nous changeons la position des éléments de notre jeu en dehors de la partie « Gestion de la logique de jeu ».

Dans le cas de cet exemple, la justification est que ces deux éléments font partie de l'affichage tête haute de notre jeu (*HUD*), qui est dépendante de la taille de la vue (vu que le *HUD* se place généralement en limite de vue).

Ainsi, vous pouvez percevoir la complexité entre la position d'un élément dans un jeu et sa position sur une fenêtre, le premier étant géré par la logique de votre jeu et le second par la logique d'affichage, dépendant du premier. N'hésitez pas à séparer ces deux notions dès le début de votre le développement.

Poursuivons en commençant par effacer le contenu de la fenêtre avec la méthode `clear` :

```
40  window.clear()
```

▶ *Supprimez cette ligne, et faites déplacer un objet à chaque tour de boucle : Qu'observez-vous ?*

▶ *Un fond noir est assez triste, mettez un fond de la couleur de votre choix.*

Demandons maintenant à la librairie d'afficher chaque objet sur la fenêtre avec la méthode `draw` :

```
43  window.draw(sprite)
```

```
46  window.draw(text)
```

Et finalement d'effectuer l'affichage de la fenêtre sur l'écran de l'utilisateur avec la méthode `display` :

```
49  window.display()
```

Comme mentionné dans le tutoriel sur l'affichage d'éléments, le cycle `clear`, `draw` et `display` est la *seule* manière correcte d'effectuer un rendu : N'essayez pas d'éviter un `clear` par exemple.

De plus, n'ayez pas peur d'effectuer ces cycles, même avec des milliers de *sprites*. En effet, vos cartes graphiques sont optimisées pour cela. Vos cartes graphiques sont plus sensibles au nombre de textures chargées par exemple, ce qui explique l'utilisation répandue de *spritesheets* dans les jeux.

2.3 Découverte de la librairie SFML : Gestion des ressources

La bibliothèque SFML manipule des ressources qui doivent être libérées au bon moment pour limiter l'utilisation de la mémoire de votre carte graphique.

En C++, les objets sont libérés *dès* qu'ils ne sont plus utilisés (*reference counting*).
En Scala, les objets sont libérés *que* lorsqu'il n'y a plus de mémoire disponible (*garbage collector*).

Comme les ressources sont majoritairement graphiques, la mémoire utilisée n'est pas celle du processeur mais celle de votre carte graphique, ce que le *garbage collector* ne peut pas détecter.

Pour contourner cette limitation, le *binding* hérite actuellement chaque ressource devant être libérée avec le trait `Resource`, permettant d'utiliser l'un des paradigmes suivants :

- Appeler explicitement la méthode `close` lorsque la ressource n'est plus utilisée, comme en `C` avec la fonction `free`.
- Utiliser la *closure* de `Using` libérant automatiquement les ressources englobée par la fonction reçue en paramètre de la *closure*, ici nommée `use`, lorsque la *closure* est terminée.

C'est cette deuxième méthode qui a été utilisée dans le template :

```
1   import scala.util.Using
```

```
9   Using.Manager { use =>
```

```
11      val window = use(RenderWindow(VideoMode(800, 600), "Hello SFML"))
```

```
50  }.get    // `.get' permet de récupérer les effets de bord de la closure: Ne l'oubliez pas!
```

Si vous créez des ressources à chaque tour de boucle, faites attention à la portée de la *closure* de `Using`.

Afin de détecter les oublis de libération mémoire, le mode de développement (voir dans la section suivante) compile votre projet en utilisant `LeakSanitizer`, qui analysera l'état de la mémoire en fin d'exécution de votre programme.

▶ *Supprimez la fonction englobante* `use` *de la ligne 21 concernant la classe* `Font`*. Lancez l'application puis fermez la fenêtre : Analysez la sortie obtenue par votre programme et retrouvez d'où vient l'erreur.*

▶ *Supprimez* `Using` *et appelez explicitement la méthode* `close` *dès qu'une ressource n'est plus utilisée.*

2.4 Mettre à jour les dépendances du projet

Les prochains mois vont voir arriver de nombreuses mises à jour améliorant l'écosystème, notamment du côté du *binding*. Nous vous tiendrons informés de l'arrivée de ces mises à jour.

Les versions sont marquées dans le fichier `project/build.properties`, il suffit de modifier ces valeurs puis de relancer complètement `sbt` pour appliquer les changements.

2.5 Liste des commandes disponibles dans `sbt`

Lors du développement, vous allez vous apercevoir que la compilation est assez lente. Cela est dû à la dernière étape, *l'éditions de liens*, qui génère l'exécutable final.

Ainsi, vous avez les différents niveaux de compilation suivants :

- `compile` : Compile les fichiers Scala dans leur forme intermédiaire ;
- `nativeLink` : Rassemble les fichiers en forme intermédiaire pour former l'exécutable final, et ;
- `run` : Exécute l'exécutable et lance ainsi votre application graphique.

`sbt` se charge de vérifier que chaque étape est effectuée avant de passer à la suivante : La commande `run` va d'abord exécuter la commande `nativeLink`, qui va ensuite exécuter `compile`.

Si vous voulez seulement vérifier que votre projet ne comporte pas d'erreur de compilation, vous pouvez uniquement utiliser la commande `compile` et ainsi éviter de passer sur l'étape lente `nativeLink`.

Si vous souhaitez nettoyer l'ensemble des fichiers générés, vous pouvez effectuer la commande `clean`.

Votre projet, par défaut, compile dans un mode « rapide » nommé `Debug`, utile pour le développement. Il existe également un mode `Release` qui compile « efficacement » votre projet en effectuant des optimisations, utile pour des présentations mais plus lent à compiler.

La valeur par défaut est définie dans le fichier `build.sbt` :

```
6 buildMode := BuildMode.Debug    // BuildMode.Release
```

Vous pouvez également changer cette valeur de façon temporaire dans `sbt` en utilisant cette syntaxe :

```
set buildMode := BuildMode.Debug    // BuildMode.Release
```

Si vous changer de mode, pensez à utiliser `clean` pour être sûr d'avoir un environnement sain.

D'autres commandes utiles sont disponibles et vous allez les découvrir plus en détail dans la Section 3 :

- `scalafmt` : Formater votre code dans un style unique (voir Section 3.1).
- `scalafmtCheck` : Vérifier si le formatage de votre code est correct.
- `doc` : Génère la documentation de votre projet (voir Section 3.4)

3 Outils d'aide au développement

3.1 Formatage : `scalafmt` (Site web de `scalafmt`)

Lorsque l'on travaille sur un projet avec différents contributeurs, il se peut qu'il y ait plusieurs styles d'écriture : Certains vont préférer des indentations de 2 espaces lorsque d'autres en feront 4 ou même 8, si ce n'est pas des tabulations.

Afin de conserver une cohérence dans le projet et éviter des guerres de modifications, il existe des outils de formatage comme `scalafmt`. Cet outil uniformise le code en suivant un ensemble de règles.

▶ *Configurez votre éditeur afin d'intégrer `scalafmt` pour développer en utilisant le bon style d'écriture.*

3.2 Ignorer les fichiers générés : `.gitignore`

Lorsque vous développez, de nombreux fichiers sont générés, comme les fichiers intermédiaires (représentation intermédiaire, exécutable final, ...), ou les fichiers par votre éditeur (analyse de code, configuration personnelle, ...).

Ces fichiers sont générés par le projet lui-même (dans votre cas, majoritairement par `sbt`) et peuvent être ainsi obtenus de nouveau à tout moment. De plus, certains de ces fichiers sont dans un format binaire propre à votre environnement comme votre processeur (`RISC-V`, `x86-64` ...).

Ainsi, ces fichiers ne sont pas intéressants à garder dans les points de sauvegardes avec `git`, il vaut mieux les ignorer en listant ce que l'on ne souhaite pas conserver dans un fichier nommé `.gitignore`.

Par exemple, le dossier `target` contient des fichiers intermédiaires et est donc listé dans le `.gitignore` :

```
4 target/
```

Le projet github/gitignore liste des fichiers `.gitignore` suivant les langages de programmation. Prenez l'habitude, pour tout nouveau projet, de copier le contenu du fichier `.gitignore` du langage de programmation de votre projet, afin d'éviter d'envoyer des fichiers inutiles.

Le projet github/gitignore liste également des fichiers pour les éditeurs dans le répertoire `Global`.

▶ *Copiez le contenu du fichier `.gitignore` de votre éditeur dans le fichier `$HOME/.config/git/ignore`.*

3.3 Publier des ressources binaires : Git LFS (Site web de Git LFS)

Lorsque vous utiliserez `git`, vous vous rendrez compte qu'il fonctionne en appliquant des différences successives sur votre code afin d'en garder l'historique.

Ce procédé fonctionne très bien sur des fichiers contenant du texte, ce qui constituera une grande partie de vos contributions. Cependant, comme vous développez un jeu, vous aurez besoin également d'autres types de ressources (image, police d'écriture, son, ...), provoquant des lenteurs dans `git`.

Pour éviter cela, nous allons utiliser *Large File Storage* (*Git LFS*), déjà configuré sur votre projet :

```
$ git lfs ls-files
0123456789 * src/main/resources/FiraSans.ttf
abcdef0123 * src/main/resources/SFML.png
```

▶ *Sur GitLab, ouvrez `SFML.png` et trouvez le badge `LFS` vous assurant que Git LFS à été utilisé.*

3.4 Documentation : Scaladoc (Site web de Scaladoc)

Lorsque l'on travaille sur un projet, il est essentiel que chaque participant *documente* les divers éléments de code qu'il produit (classes, méthodes, ...).

La documentation permet à n'importe quel contributeur (y compris l'auteur original) de pouvoir utiliser ces éléments de code comme des « boîtes noires », sans devoir directement lire le code pour savoir comment il fonctionne.

Scaladoc est un système générant de la documentation sous forme de pages HTML à partir de commentaires spécifiques que le programmeur ajoute aux fichiers sources.

Devant chaque élément (classe, méthode, attribut, etc.) que l'on souhaite documenter, on place un bloc de commentaire (débutant par `/**` et terminant par `*/`) contenant :

- un descriptif court (d'au maximum une ligne) ;
- un descriptif plus long si besoin, et ;
- d'autres informations introduites par des étiquettes.

Ces étiquettes permettent d'ajouter une sémantique à la description que l'on fait. Par exemple, `@param` est une étiquette pour décrire un paramètre d'un constructeur ou d'une méthode.

Nous allons prendre une exemple une classe `C` qui contient une variable :

```
/**
 * Short description of the class `C`.
 *
 * Long description of the class `C`.
 * This long description can last on many lines and paragraphs.
 * Remember: Always write your documentation in English.
 *
 * @constructor
 *   Short description of the main constructor of class `C`.
 *
 * @param p
 *   Description of the parameter of this constructor.
 */
class C(p: Int):
    /**
     * Short description of the attribute `a`.
     */
    var a = 0

    /**
     * Short description of the method `m`.
     *
     * @param p
     *   Description of the parameter `p` of the method `m`.
     *
     * @return
     *   Description of the value returned by the method `m`.
     */
    def m(p: Int): Int =
            ...
```

Scaladoc propose de nombreuses autres étiquettes pour ajouter une multitude d'informations de types divers et supporte également le « *markup* » pour formatter le texte.

Pour générer la documentation de votre projet, il suffit de lancer la commande `doc` dans `sbt`, ce qui produira un dossier `doc` dans `target` contenant un site statique HTML dont la racine est `index.html`.

3.5 Vérifier le bon fonctionnement du projet : GitLab CI (Documentation)

Pour assurer que votre projet fonctionne correctement tout au long de son développement, des commandes peuvent être lancés à chaque fois que vous envoyez vos points de sauvegardes sur GitLab. C'est le principe de l'intégration continue (*CI*), vérifiant qu'aucune régression n'est introduite.

Les commandes à exécuter sont décrites dans le fichier `.gitlab-ci.yml`, et effectuent dans cet ordre :

- Une vérification du formatage de votre code avec `scalafmt` :

```
15 format:
16   stage: format
```

```
19   script:
20     - sbt scalafmtCheck
```

- Une passe de compilation avec `compile` vérifiant que votre code compile correctement :

```
22 build:
23   stage: build
24   script:
25     - sbt compile
```

- Une génération de la documentation avec `doc` :

```
27 pages:
28   stage: deploy
```

```
33   script:
34     - sbt doc
```

Cette série d'étapes à réaliser est souvent désigné par le terme *pipeline*, vu que les modifications effectuée à chaque étape est conservée. Par exemple, la commande `doc` a besoin de compiler les fichiers avec `compile`, ce qui a déjà été effectuée par la tâche `build` :

```
10 stages: [ format, build, deploy ]
```

Dans la Section 1.4, vous avez envoyé le premier point de sauvegarde nommé « Initial commit » à GitLab, ce-dernier l'ayant vérifié :

▶ *Trouvez sur l'interface GitLab le résultat du passage de GitLab CI.*

N'oubliez surtout pas ce conseil déjà donné dans ce document :

▶ *Après la séance sur `git`, n'hésitez pas à explorer l'interface GitLab de votre projet.*

Vous voilà fin prêt pour développer votre propre jeu en Scala avec la SFML !

Rendez-vous la semaine prochaine pour découvrir comment utiliser `git`, vous permettant de contribuer à plusieurs sur un même projet.

En attendant, si vous le souhaitez, continuez de parcourir la documentation et les tutoriels sur SFML.