Haute École Louvain en Hainaut

Sokoban

Giorgio Caculli LA196672, Guillaume Lambert LA198116, Tanguy Taminiau LA199566, Nathan Thaon LA188132
Groupe B01

8 décembre 2021

ÉCONOMIQUE

Table des matières

1 Introduction

Dans le cadre du cours de développement de jeux vidéo de l'UE 308, nous avons dû créer un jeu de manière autonome. L'objectif pédagogique de ce projet est de pousser l'élève à mieux appréhender la programmation en C++ vue au cours ainsi que d'apprendre à utiliser une librairie graphique en C++. De plus, la grande liberté accordée à ce projet oblige l'élève à apprendre à se documenter, savoir faire des recherches. L'objectif du projet est simple, créer un jeu selon notre propre envie.

2 Présentation du sujet

Notre jeu sera un jeu de type puzzle. Il sera en vue 2D, vue du haut. Le but du jeu sera d'atteindre un objectif, en se frayant un chemin via la résolution d'un puzzle. La mécanique principale de ce jeu sera de pouvoir pousser une caisse pour nous permettre d'atteindre notre objectif qui sera de pousser ces caisses sur certains points pour terminer le niveau. Les mouvements du personnage et des caisses se feront en case par case et les caisses ne pourront pas être poussées deux par deux. Il y aura aussi la présence d'un compteur de mouvements et un un compteur de reset.

2.1 Spécification technique

2.1.1 GUI : SFML

Figure 1 – Logo SFML

SFML est une librairie qui donne accès à une vaste variété de fonctionnalités purement écrites en C++. Les cinq fonctionnalités dont nous disposons sont les gestions suivantes :
- Toute interaction avec le système d'exploitation
- Fenêtrage
- Graphismes
- Son
- Réseau

SFML permet le cross-platforming : un logiciel codé avec SFML aura le même visuel indépendamment du système d'exploitation sur lequel le jeu tourne.

2.1.2 Librairie Boost

Figure 2 – Logo Boost

La librairie de logging que nous utiliserons se nomme Boost. En quelques mots, la librairie Boost est elle-même un ensemble de librairies permettant d'étendre les fonctionnalités de C++. Dans notre cas, nous utiliserons les fonctionnalités prédéfinies de Boost. Notamment Log, qui nous donne accès à la possibilité d'enregistrer les différentes interactions qui ont eu lieu lors de l'exécution du jeu.

2.1.3 Fichiers

Les niveaux et les sauvegardes seront stockés dans des fichiers purement textuels. Ces fichiers ne stockeront que le design des niveaux ou de la partie en cours. Comme déclaré précédemment, Boost est un ensemble de librairies, dans cet ensemble il existe la librairie Boost.JSON. Grâce à cette librairie, nous serons capable de stocker des informations en format JSON, comme par exemple, une liste des scores.

2.1.4 OS

Les systèmes d'exploitation sur lesquels nous testerons notre jeu sont les suivants :
— Linux
— MacOS 11
— MS Windows 10

3 Analyse

3.1 Product Backlog

US-01	En tant qu'utilisateur je voudrais reset une partie.
US-02	En tant qu'utilisateur je voudrais mettre en pause la partie.
US-03	En tant qu'utilisateur je voudrais sauvegarder une partie.
US-04	En tant qu'utilisateur je voudrais charger des niveaux personnalisé.
US-05	En tant qu'utilisateur je voudrais arrêter mon jeu à tout môment.
US-06	En tant qu'utilisateur je voudrais une musique de fond.
US-07	En tant qu'utilisateur je voudrais gérer le volume de la musique et des effets.
US-08	En tant qu'utilisateur je voudrais créer des niveaux personnaliser.

4 diagramme

4.1 Diagramme UML

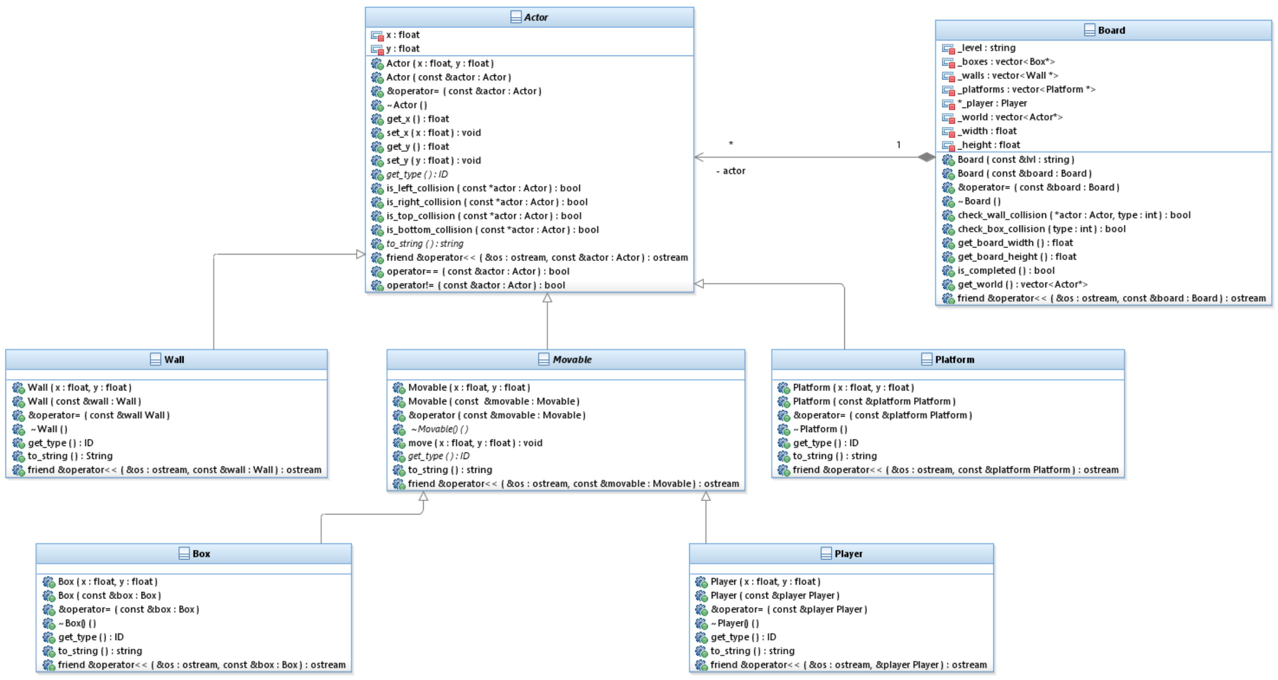
Actor
x : float
y : float
Actor (x : float, y : float)
Actor (const &actor : Actor)
&operator= (const &actor : Actor)
~Actor ()
get_x () : float
set_x (x : float) : void
get_y () : float
set_y (y : float) : void
get_type ( ) : ID
is_left_collision (const *actor : Actor) : bool
is_right_collision (const *actor : Actor) : bool
is_top_collision (const *actor : Actor) : bool
is_bottom_collision (const *actor : Actor) : bool
to_string ( ) : string
friend &operator<< (&os : ostream, const &actor : Actor) : ostream
operator== (const &actor : Actor) : bool
operator!= (const &actor : Actor) : bool
*
1
- actor
Board
_level : string
_boxes : vector<Box*>
_walls : vector<Wall *>
_platforms : vector<Platform *>
*_player : Player
_world : vector<Actor*>
_width : float
_height : float
Board (const &lvl : string)
Board (const &board : Board)
&operator= (const &board : Board)
~Board ()
check_wall_collision (*actor : Actor, type : int) : bool
check_box_collision (type : int) : bool
get_board_width () : float
get_board_height () : float
is_completed () : bool
get_world () : vector<Actor*>
friend &operator<< (&os : ostream, const &board : Board) : ostream
Wall
Wall (x : float, y : float)
Wall (const &wall : Wall)
&operator= (const &wall Wall)
~Wall ()
get_type () : ID
to_string () : String
friend &operator<< (&os : ostream, const &wall : Wall) : ostream
Movable
Movable (x : float, y : float)
Movable (const &movable : Movable)
&operator (const &movable : Movable)
~Movable() ()
move (x : float, y : float) : void
get_type ( ) : ID
to_string () : string
friend &operator<< (&os : ostream, const &movable : Movable) : ostream
Platform
Platform (x : float, y : float)
Platform (const &platform Platform)
&operator= (const &platform Platform)
~Platform ()
get_type () : ID
to_string () : string
friend &operator<< (&os : ostream, const &platform Platform) : ostream
Box
Box (x : float, y : float)
Box (const &box : Box)
&operator= (const &box : Box)
~Box() ()
get_type () : ID
to_string () : string
friend &operator<< (&os : ostream, const &box : Box) : ostream
Player
Player (x : float, y : float)
Player (const &player Player)
&operator= (const &player Player)
~Player() ()
get_type () : ID
to_string () : string
friend &operator<< (&os : ostream, &player Player) : ostream

Figure 3 – diagramme de classe

La Board est composé d'Actor ou de ses spécialisations. Plus précisement, une board est composé d'objet immobile tels que des Walls et des Platforms mais aussi d'objet mobile tels que des Box et un personnage.

4.2 Diagramme d'activité

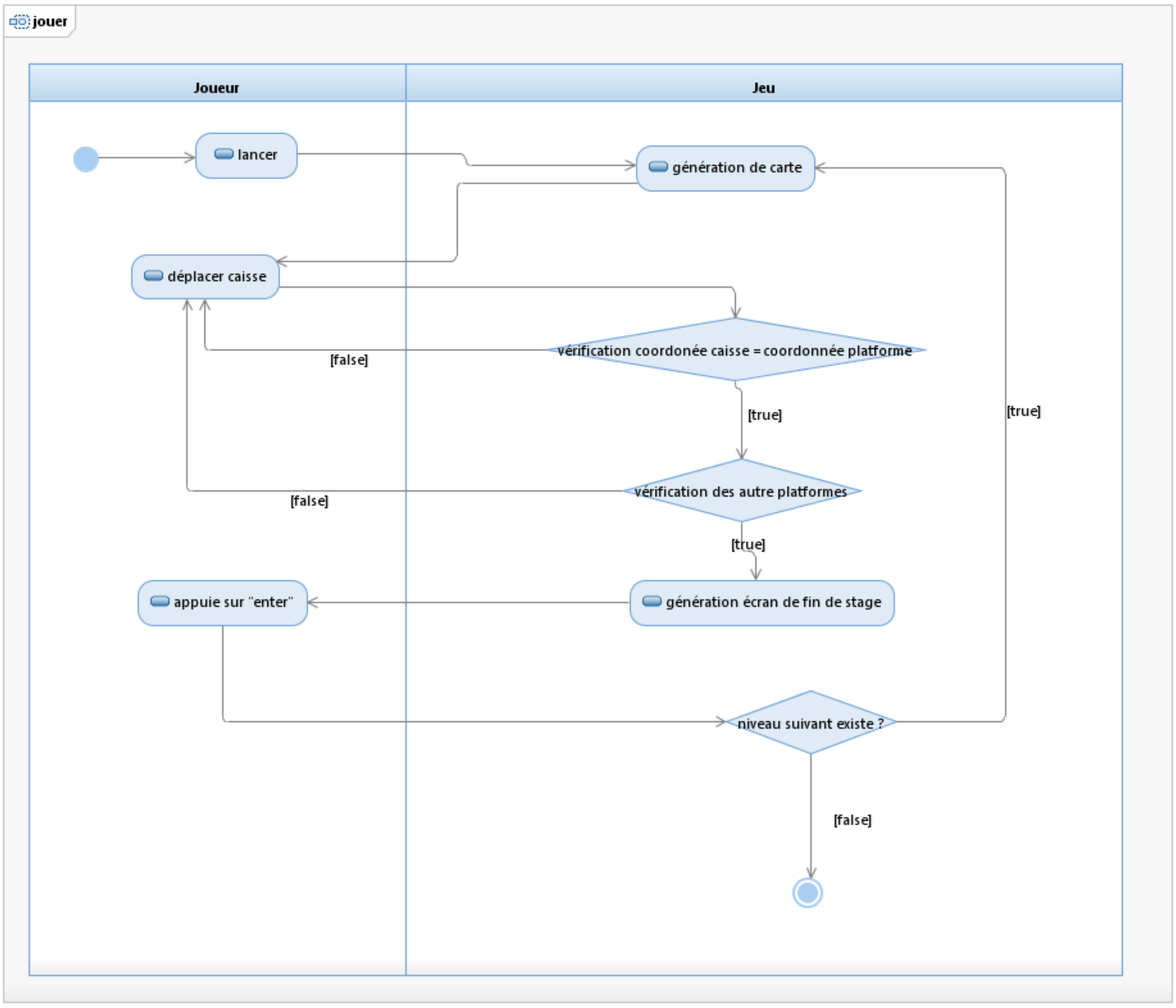

FIGURE 4 – diagramme d'activité

4.3 Diagramme Design Pattern State

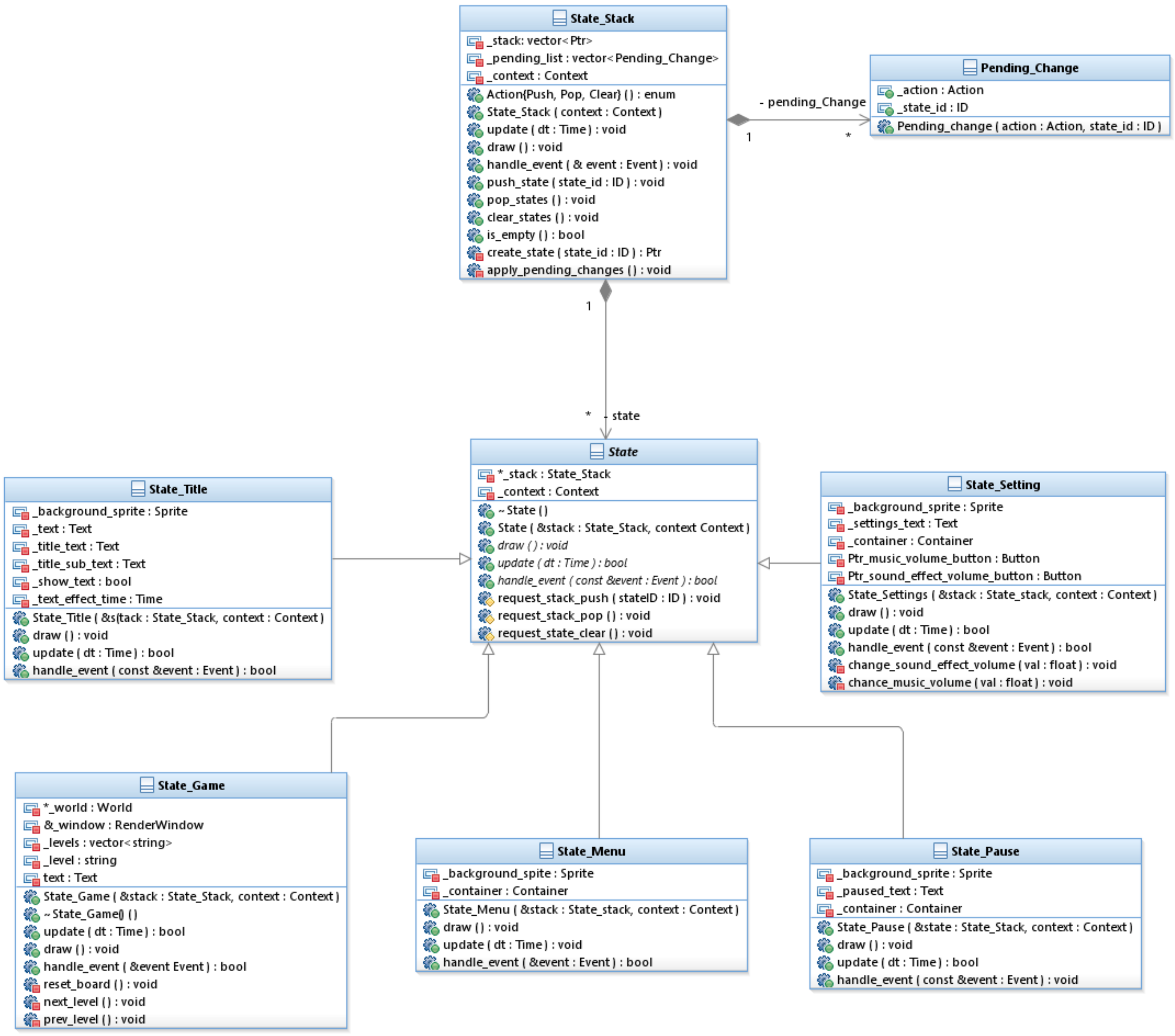
State_Stack
_stack: vector<Ptr>
_pending_list : vector<Pending_Change>
_context : Context
Action{Push, Pop, Clear} () : enum
State_Stack (context : Context)
update (dt : Time) : void
draw () : void
handle_event (& event : Event) : void
push_state (state_id : ID) : void
pop_states () : void
clear_states () : void
is_empty () : bool
create_state (state_id : ID) : Ptr
apply_pending_changes () : void
- pending_Change
1
*
Pending_Change
_action : Action
_state_id : ID
Pending_change (action : Action, state_id : ID)
1
*
- state
State
*_stack : State_Stack
_context : Context
~State ()
State (&stack : State_Stack, context Context)
draw () : void
update (dt : Time) : bool
handle_event (const &event : Event) : bool
request_stack_push (stateID : ID) : void
request_stack_pop () : void
request_state_clear () : void
State_Title
_background_sprite : Sprite
_text : Text
_title_text : Text
_title_sub_text : Text
_show_text : bool
_text_effect_time : Time
State_Title (&stack : State_Stack, context : Context)
draw () : void
update (dt : Time) : bool
handle_event (const &event : Event) : bool
State_Setting
_background_sprite : Sprite
_settings_text : Text
_container : Container
Ptr_music_volume_button : Button
Ptr_sound_effect_volume_button : Button
State_Settings (&stack : State_stack, context : Context)
draw () : void
update (dt : Time) : bool
handle_event (const &event : Event) : bool
change_sound_effect_volume (val : float) : void
chance_music_volume (val : float) : void
State_Game
*_world : World
&_window : RenderWindow
_levels : vector<string>
_level : string
text : Text
State_Game (&stack : State_Stack, context : Context)
~State_Game() ()
update (dt : Time) : bool
draw () : void
handle_event (&event Event) : bool
reset_board () : void
next_level () : void
prev_level () : void
State_Menu
_background_spite : Sprite
_container : Container
State_Menu (&stack : State_stack, context : Context)
draw () : void
update (dt : Time) : void
handle_event (&event : Event) : bool
State_Pause
_background_sprite : Sprite
_paused_text : Text
_container : Container
State_Pause (&state : State_Stack, context : Context)
draw () : void
update (dt : Time) : bool
handle_event (const &event : Event) : void

FIGURE 5 – diagramme du design pattern state

Le design pattern state permet de créer des états et de modifier un ou des comportements par rapport à ses états. Dans notre jeu, le state permet d'interchanger entre les différents affichages possibles dans le jeu.

5 Implémentation

5.1 Présentation du jeu

Comme dit précédemment, notre jeu est un jeu de type puzzle en 2D en vue du dessus. Notre jeu compte 25 niveaux de bases, il est possible de créer nos propres niveau et de les inclure dans notre liste de niveaux. Pour terminer un niveau (et donc finir le puzzle), il faut mettre toutes les caisses sur une platforme, les platformes ne sont pas spécifiques à une caisse.

5.2 Présentation de l'UI

5.2.1 Ecran titre

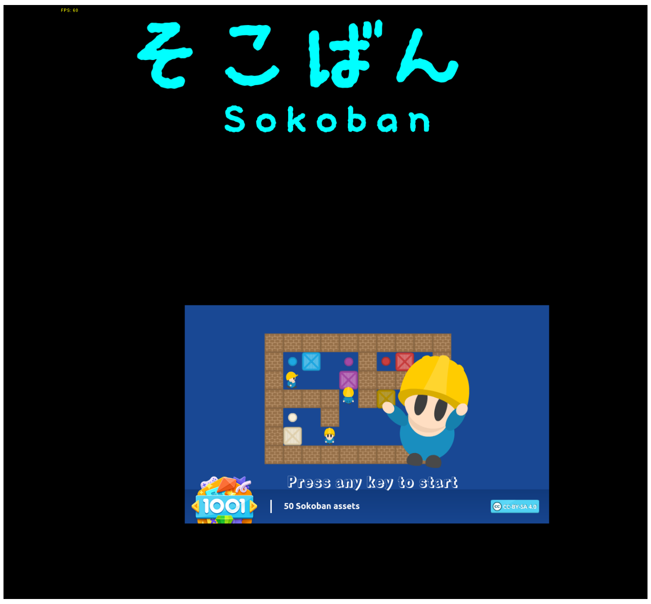

FIGURE 6 – écran titre

Voici à quoi ressemble l'écran titre de notre jeu, comme vous pouvez le remarquer, l'écran titre est composé du nom de l'application (écrit en alphabet latin et en japonais). Une image de présentation du jeu est également présente.

5.2.2 Menu principal

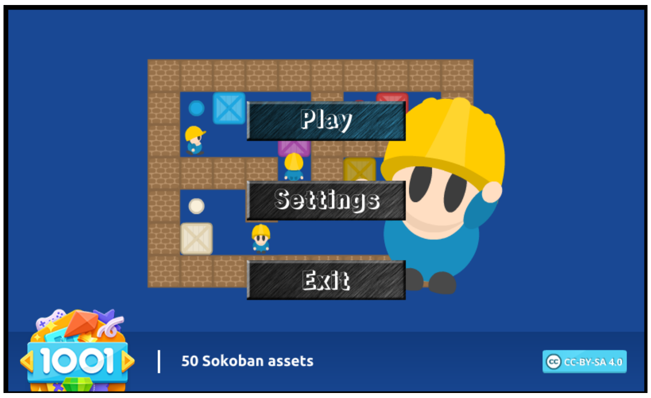

FIGURE 7 – menu principal

Notre menu principal est composé de trois bouttons , Le premier permettant de jouer, le deuxième permet d’accéder au menu "paused" et le troisième permet de quitter le jeu. L’image de fond reste la même qu’à l’écran titre.

5.2.3 Menu pause

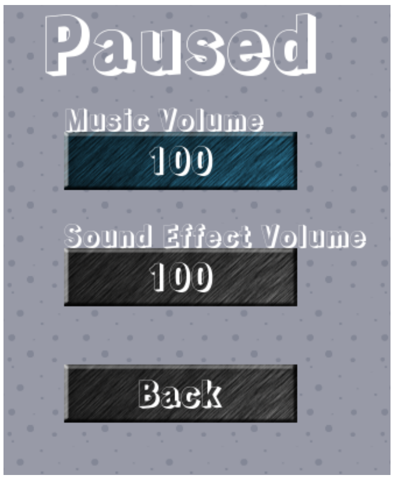

FIGURE 8 – menu pause

Les deux premiers bouttons permettent de régler le volume de la musique et le volume des effets sonores. Pour modifier les valeurs, sélectionné le bouton correspondant grâce au flèche haut et bas du clavier. Appuyez sur la touche "enter" et appuyez sur la flèche du haut pour monter le volume ou celle du bas pour le descendre. Une fois le modification faite appuyez sur "esc" pour sortir de la sélection du boutton. Le dernier boutton permet de reprendre le jeu où il a été laissé.

5.2.4 Jeu

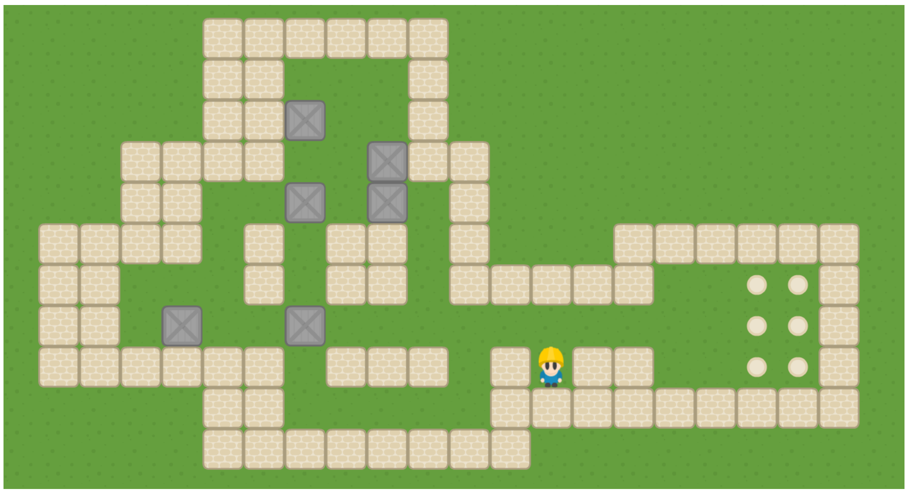

FIGURE 9 – jeu

Voici à quoi ressemble notre jeu. La vue est composé d'un fond à pois (vert sur l'image), de murs qui définissent la limite de la zone du puzzle (beiges sur l'image), de platformes qui définissent l'emplacement de l'objectif (blanches sur l'image) et de caisses (grises sur l'image)qui doivent être déplacées vers les objectifs (blancs sur l'image). les couleurs changent de manière aléatoire pour chaque reset de niveau.

6 Contribution

Giorgio a réalisé l'interface graphique. Etant plus en avance que les autres sur le language C++, il s'est directement attaqué à l'interface graphique donc de la matière qui n'a pas été vue, étant donné que Giorgio avait déjà vu une partie de la matière de C++.

Tanguy a réalisé l'UML de base pour mettre en place le début du projet, celui-ci a été appelé à évoluer avec le développement du projet. Tanguy suite à l'écriture de l'UML a commencé l'écriture des différentes classes du modèle.

Guillaume a réalisé une partie de l'algorithmique du modèle, notamment la gestion des collisions. Guillaume a aussi aidé en naviguant entre les différents participants du projet en fonction de là où il serait le plus utile.

Nathan a réalisé des réflexions sur des design patterns pouvant être utiles à notre jeu et Nathan les a implémentés. Il a aussi beaucoup travaillé sur la documentation pour laisser aux autres le temps de travailler plus sur le projet.

7 Conclusion

Ce projet nous permit de nous rendre compte de ce qu'est un projet de création de jeu vidéo. Cela nous a aussi permit de nous familiariser avec le language de programmation C++ mais aussi le logiciel de version git. De plus, nous avons appris à bien communiquer et s'organiser entre collègues/coéquipiers. Ce projet nous a permis aussi d'apprendre à combiner ce que nous apprenons en cours avec de la recherche de documentation sur internet. En finalité, nous sommes contents du résultat obtenu actuellement, cependant, nous pensons arrivez plus loin et avons légèrement sous estimé le temps nécessaire pour le jeu initialement prévu. Nous tenons à remercier ceux qui nous donné leurs avis par rapport au design du jeu. Et nous tenons à remercier particulièrement Mr. Altares pour ses bons conseils.

8 Bibliographie

Jan Haller, Henrik Vogelius Hansson, Artur Moreira - *SFML Game Development* - Packt - 2013

Milcho G. Milchev - *SFML Essentials* - Packt - 2013

Maxime Barbier - *SFML Blueprints* - Packt - 2015

Raimondas Pupius - *SFML Game Development By Example* - Packt - 2015

Raimondas Pupius - *Mastering SFML Game Development* - Packt - 2017

9 Code source

9.1 Aspect Intéressant

Un aspect intéressant qui a demandé beaucoup de recherches était de réussir à mettre en place la randomisation des différents aspects graphiques présents sur la carte. Afin de rendre le jeu plus varié, différents algorithmes propres à C++ ont été exploité. Pour créer un niveau de randomisation, C++ demande de mettre en place un générateur de « seed » qui permettra par la suite de choisir de manière aléatoire entre les différents choix proposés dans les énumérations des couleurs. Pour ce faire, nous avons exploité le « mersenne twister engine », aussi connu en C++ sous le nom de « mt19937 ». Ce mécanisme permet de retourner un entier d'une taille maximale de 32 bits, en partant d'un seed randomiquement généré. C'est grâce à tout ces engines que nous avons sû avoir une complète randomisation des assets lors de l'affichage de la carte.

9.2 Main

9.2.1 main.hpp

```
#ifndef SOKOBAN_CPP_MAIN_HPP
#define SOKOBAN_CPP_MAIN_HPP

/**
 * Default definition of the main function
 *
 * @param argc The number of arguments
 * @param argv Array containing the argument
 * @return 0 if the program executed correctly
 */
int main( int argc, char *argv[] );

#endif //SOKOBAN_CPP_MAIN_HPP
```

9.2.2 main.cpp

```
#include "main.hpp"

#include "ui/Menu.hpp"
#include "util/Logger.hpp"

using namespace sokoban::ui;
using namespace sokoban::util;

/**
 * Main function that initializes the game's logger.
 * The logger will log everything inside a file called sokoban.log
 * File which is instantiated upon execution of the program.
 * Once the logger is initialized, we initialize the menu
 * which will be used to start the application.
 * If no exception is caught, the program will return 0, otherwise -1.
 */
int main( int argc, char *argv[] )
{
    Logger logger( "sokoban.log" );
    remove( logger.get_file_name().c_str() );

    Logger::log( LoggerLevel::INFO, "Starting Menu..." );

    Menu menu;

    Logger::log( LoggerLevel::INFO, "Running Game..." );

    int res = menu.launch_application();

    Logger::log( LoggerLevel::INFO, "Closing Sokoban..." );

    return res;
}
```

9.3 Model

9.3.1 Actor.hpp

```
#ifndef SOKOBAN_ACTOR_HPP
#define SOKOBAN_ACTOR_HPP

#include <array>
#include <string>
#include <ostream>

namespace sokoban
{
    namespace model
    {
        /**
         * Actor class
         * Parent class for any actor that will be displayed throughout the game
         * Parent class of both movable and non movable actors
         */
        class Actor
        {
        private:
            float _x; /** The X coordinate on the board */
            float _y; /** The Y coordinate on the board */
        public:
            enum ID /** We consider there to be 4 types of actors */
```

```cpp
            {
                BOX, /** A box */
                PLATFORM, /** A platform on which we put the box */
                PLAYER, /** A player */
                WALL /** A wall */
            };
            Actor( float x, float y );
            Actor( const Actor &actor );
            Actor &operator=( const Actor &actor );
            virtual ~Actor() = default;
            float get_x() const;
            void set_x( float x );
            float get_y() const;
            void set_y( float y );
            virtual ID get_type() const = 0;
            bool is_left_collision( const Actor *actor ) const;
            bool is_right_collision( const Actor *actor ) const;
            bool is_top_collision( const Actor *actor ) const;
            bool is_bottom_collision( const Actor *actor ) const;
            virtual std::string to_string() const;
            friend std::ostream &operator<<( std::ostream &os, const Actor &actor );
            bool operator==( const Actor &actor ) const;
            bool operator!=( const Actor &actor ) const;
        };
    }
}

#endif //SOKOBAN_ACTOR_HPP
```

9.3.2 Actor.cpp

```cpp
#include "Actor.hpp"

#include <sstream>

using namespace sokoban::model;

const float SPACE = 64.f; /** Dans ce cas, les images mesurent 64x64 */

/**
 * This constructor will initialise the different pieces of informations which characterise an actor
 * @param x The X coordinate.
 * @param y The Y coordinate.
 * @param asset_coords The coordinates on the sprite sheet which represent it graphically.
 */
Actor::Actor( float x, float y )
        : _x( x )
          , _y( y )
{
}

/**
 * Copy Constructor
 * @param actor The actor from which we wish to create a copy
 */
Actor::Actor( const Actor &actor )
        : Actor( actor._x, actor._y )
{
}

/**
 * Redefinition of the "=" operator
 * @param actor The actor from which we wish to copy the information
 * @return The new instance of an actor with the same information as the one on the right
 */
Actor &Actor::operator=( const Actor &actor )
{
    if ( &actor != this )
    {
        _x = actor._x;
        _y = actor._y;
    }
    return *this;
}

/**
 * X coordinate getter
 * @return The value of X
 */
float Actor::get_x() const
{
    return _x;
}
```

```

/**
 * X coordinate setter
 * @param x The new value of X
 */
void Actor::set_x( float x )
{
    this->_x = x;
}

/**
 * Y coordinate getter
 * @return The value of Y
 */
float Actor::get_y() const
{
    return _y;
}

/**
 * Y coordinate setter
 * @param y The new value of Y
 */
void Actor::set_y( float y )
{
    this->_y = y;
}

/**
 * Function whose purpose is to verify if a left collision took place between two actors.
 * A movable actor will call this function and will check the following cases:
 * Let's assume Actor 1's X coordinates is x1 and Actor 2's X coordinates is x2:
 * Let's assume Actor 1's Y coordinates is y1 and Actor 2's Y coordinates is y2:
 * SPACE represents the amount of space between each object before the collision:
 * If x1 - SPACE = x2 and Y1 = Y2
 * Then a collision on the left took place.
 * @param actor The actor on the left
 * @return true if the collision took place, false otherwise
 */
bool Actor::is_left_collision( const Actor *actor ) const
{
    return get_x() - SPACE == actor->get_x() && get_y() == actor->get_y();
}

/**
 * Function whose purpose is to verify if a right collision took place between two actors.
 * A movable actor will call this function and will check the following cases:
 * Let's assume Actor 1's X coordinates is x1 and Actor 2's X coordinates is x2:
 * Let's assume Actor 1's Y coordinates is y1 and Actor 2's Y coordinates is y2:
 * SPACE represents the amount of space between each object before the collision:
 * If x1 + SPACE = x2 and y1 = y2
 * Then a collision on the right took place.
 * @param actor The actor on the right
 * @return true if the collision took place, false otherwise
 */
bool Actor::is_right_collision( const Actor *actor ) const
{
    return get_x() + SPACE == actor->get_x() && get_y() == actor->get_y();
}

/**
 * Function whose purpose is to verify if a top collision took place between two actors.
 * A movable actor will call this function and will check the following cases:
 * Let's assume Actor 1's X coordinates is x1 and Actor 2's X coordinates is x2:
 * Let's assume Actor 1's Y coordinates is y1 and Actor 2's Y coordinates is y2:
 * SPACE represents the amount of space between each object before the collision:
 * If y1 - SPACE = y2 and x1 = x2
 * Then a collision on the top took place.
 * @param actor The actor on the top
 * @return true if the collision took place, false otherwise
 */
bool Actor::is_top_collision( const Actor *actor ) const
{
    return get_y() - SPACE == actor->get_y() && get_x() == actor->get_x();
}

/**
 * Function whose purpose is to verify if a bottom collision took place between two actors.
 * A movable actor will call this function and will check the following cases:
 * Let's assume Actor 1's X coordinates is x1 and Actor 2's X coordinates is x2:
 * Let's assume Actor 1's Y coordinates is y1 and Actor 2's Y coordinates is y2:
 * SPACE represents the amount of space between each object before the collision:
 * If y1 + SPACE = y2 and x1 = x2
 * Then a collision on the bottom took place.
 * @param actor The actor on the bottom
 * @return true if the collision took place, false otherwise
 */
bool Actor::is_bottom_collision( const Actor *actor ) const
{
    return get_y() + SPACE == actor->get_y() && get_x() == actor->get_x();
}

/**
 * Textual output for the actor's coordinates
 * @return X: <the value of the x coordinate> Y: <the value of the y coordinate>
 */
std::string Actor::to_string() const
{
    std::stringstream ss;
    ss << "X: " << _x << " Y: " << _y;
    return ss.str();
}

/**
 * Redefinition of the << operator
 * This redefinition will allow us to directly output the object without
 * having to call the to_string() function within the << operators
 * @return The textual output from the function to_string()
 */
std::ostream &sokoban::model::operator<<( std::ostream &os, const Actor &actor )
{
    os << actor.to_string();
    return os;
}

/**
 * Redefinition of the == operator
 * This will allow us to check whether or not two actors are exactly the same.
 * @return true if same actor, false if not
 */
bool Actor::operator==( const Actor &actor ) const
{
    return _x == actor._x &&
            _y == actor._y;
}

/**
 * Redefinition of the != operator
 * This will allow us to check whether two actors are not the same
 * @return the opposite of the == operator
 */
bool Actor::operator!=( const Actor &actor ) const
{
    return !( actor == *this );
}
```

9.3.3 Movable.hpp

```
#ifndef SOKOBAN_MOVABLE_HPP
#define SOKOBAN_MOVABLE_HPP

#include "Actor.hpp"

#include <array>
#include <string>
#include <ostream>

/**
 * Class that handles actors capable of moving
 * This class inherits from Actor
 */
namespace sokoban
{
    namespace model
    {
        class Movable
                : public Actor
        {
        public:
            Movable( float x, float y );
            Movable( const Movable &movable );
            Movable &operator=( const Movable &movable );
            ~Movable() override = default;
            void move( float x, float y );
            ID get_type() const override = 0;
            std::string to_string() const override;
            friend std::ostream &operator<<( std::ostream &os, const Movable &movable );
```

```cpp
        };
    }
}

#endif //SOKOBAN_MOVABLE_HPP
```

9.3.4 Movable.cpp

```cpp
#include "Movable.hpp"

#include <sstream>
#include <utility>

using namespace sokoban::model;

/**
 * Just like the Actor class, the object 'movable' must also instance the X and Y coordinates
 */
Movable::Movable( float x, float y )
        : Actor( x, y )
{
}

/**
 * Copy constructor for the movable actor
 */
Movable::Movable( const Movable &movable )
        : Movable( movable.get_x(), movable.get_y() )
{
}

/**
 * Redefinition of the = operator
 * @param movable The movable object we wish to copy the information from
 * @return The new instance of the actor with the copied information
 */
Movable &Movable::operator=( const Movable &movable )
{
    if ( &movable != this )
    {
        set_x( movable.get_x() );
        set_y( movable.get_y() );
    }
    return *this;
}

/**
 * Make the object transition from a coordinate to another
 * @param x The X units to move to
 * @param y The Y units to move to
 */
void Movable::move( float x, float y )
{
    set_x( get_x() + x );
    set_y( get_y() + y );
}

/**
 * Textual output for the actor
 * @return Display the information related to the movable object
 */
std::string Movable::to_string() const
{
    std::stringstream ss;
    ss << Actor::to_string();
    return ss.str();
}

/**
 * Redefinition of the << operator
 * @param os The desired output stream
 * @param movable The movable object to get the information from
 * @return The textual output
 */
std::ostream &sokoban::model::operator<<( std::ostream &os, const Movable &movable )
{
    os << movable.to_string();
    return os;
}
```

9.3.5 Box.hpp

```cpp
#ifndef SOKOBAN_BOX_HPP
#define SOKOBAN_BOX_HPP

#include "Movable.hpp"

#include <map>
#include <array>
#include <string>
#include <ostream>

namespace sokoban
{
    namespace model
    {
        /**
         * Box class
         * This class is an actor that can move, as such it inherits from the classe Movable
         */
        class Box
                : public Movable
        {
        public:
            Box( float x, float y );
            Box( const Box &box );
            Box &operator=( const Box &box );
            ~Box() override;
            ID get_type() const override;
            std::string to_string() const override;
            friend std::ostream &operator<<( std::ostream &os, const Box &box );
        };
    }
}

#endif //SOKOBAN_BOX_HPP
```

9.3.6 Box.cpp

```cpp
#include "Box.hpp"

#include "../util/Logger.hpp"

#include <sstream>
#include <iostream>

using namespace sokoban::model;
using namespace sokoban::util;

Box::Box( float x, float y )
        : Movable( x, y )
{
}

Box::Box( const Box &box )
        : Box( box.get_x(), box.get_y() )
{
}

Box &Box::operator=( const Box &box )
{
    if ( &box != this )
    {
        set_x( box.get_x() );
        set_y( box.get_y() );
    }
    return *this;
}

Box::~Box()
{
    Logger::log( LoggerLevel::INFO, "Deletion " + Box::to_string() );
}

Actor::ID Box::get_type() const
{
    return Actor::BOX;
}

std::string Box::to_string() const
{
    std::stringstream ss;
    ss << "Box: " << Movable::to_string();
    return ss.str();
}
```

```cpp

std::ostream &sokoban::model::operator<<( std::ostream &os, const Box &box )
{
    os << box.to_string();
    return os;
}
```

9.3.7 Platform.hpp

```cpp
#ifndef SOKOBAN_PLATFORM_HPP
#define SOKOBAN_PLATFORM_HPP

#include "Actor.hpp"

#include <map>
#include <array>
#include <string>
#include <ostream>

namespace sokoban
{
    namespace model
    {
        class Platform
                : public Actor
        {
        public:
            Platform( float x, float y );
            Platform( const Platform &platform );
            Platform &operator=( const Platform &platform );
            ~Platform() override;
            ID get_type() const override;
            std::string to_string() const override;
            friend std::ostream &operator<<( std::ostream &os, const Platform &platform );
        };
    }
}

#endif //SOKOBAN_PLATFORM_HPP
```

9.3.8 Platform.cpp

```cpp
#include "Platform.hpp"

#include "../util/Logger.hpp"

#include <sstream>
#include <iostream>

using namespace sokoban::model;
using namespace sokoban::util;

/**
 * Default constructor for the platforms
 * @param x The X coordinates on the board
 * @param y The Y coordinates on the board
 */
Platform::Platform( float x, float y )
        : Actor( x, y )
{
}

/**
 * Copy constructor for the platform actor
 * @param platform The platform we wish to copy the information from
 */
Platform::Platform( const Platform &platform )
        : Platform( platform.get_x(), platform.get_y() )
{
}

/**
 * Redefinition of the = operator
 * @param platform The platform we wish to copy the information from
 * @return New instance of a platform with the copied information
 */
Platform &Platform::operator=( const Platform &platform )
{
    if ( &platform != this )
    {
        set_x( platform.get_x() );
        set_y( platform.get_y() );
    }
    return *this;
}

/**
 * Default destructor for the platforms
 */
Platform::~Platform()
{
    Logger::log( LoggerLevel::INFO, "Deletion " + Platform::to_string() );
}

/**
 * Getter meant to retrieve the nature of a platform actor
 * @return The fact that the actor is actually a platform
 */
Actor::ID Platform::get_type() const
{
    return Actor::PLATFORM;
}

/**
 * Textual output containing the platform's information
 * @return Text containing the info
 */
std::string Platform::to_string() const
{
    std::stringstream ss;
    ss << "Platform: " << Actor::to_string();
    return ss.str();
}

/**
 * Redefinition of the << operator
 * @param os The desired output stream
 * @param platform The platform we wish to output
 * @return Textual output displaying the platform's information
 */
std::ostream &sokoban::model::operator<<( std::ostream &os, const Platform &platform )
{
    os << platform.to_string();
    return os;
}
```

9.3.9 Player.hpp

```cpp
#ifndef SOKOBAN_PLAYER_HPP
#define SOKOBAN_PLAYER_HPP

#include "Movable.hpp"

#include <map>
#include <array>
#include <string>
#include <ostream>

namespace sokoban
{
    namespace model
    {
        class Player
                : public Movable
        {
        public:
            Player( float x, float y );
            Player( const Player &player );
            Player &operator=( const Player &player );
            ~Player() override;
            ID get_type() const override;
            std::string to_string() const override;
            friend std::ostream &operator<<( std::ostream &os, const Player &player );
        };
    }
}

#endif //SOKOBAN_USER_HPP
```

9.3.10 Player.cpp

```cpp
#include "Player.hpp"

#include "../util/Logger.hpp"
```

```

#include <sstream>
#include <iostream>

using namespace sokoban::model;
using namespace sokoban::util;

/**
 * Default constructor for the player
 * @param x The X coordinates on the board
 * @param y The Y coordinates on the board
 */
Player::Player( float x, float y )
        : Movable( x, y )
{
}

/**
 * Copy constructor for the player
 * @param player The player we wish to copy the information from
 */
Player::Player( const Player &player )
        : Player( player.get_x(), player.get_y() )
{
}

/**
 * Redefinition of the = operator
 * @param player The player we wish to copy the information from
 * @return The new instance of a player with the copied information
 */
Player &Player::operator=( const Player &player )
{
    if ( &player != this )
    {
        set_x( player.get_x() );
        set_y( player.get_y() );
    }
    return *this;
}

/**
 * Default destructor for the player
 */
Player::~Player()
{
    Logger::log( LoggerLevel::INFO, "Deletion " + Player::to_string() );
}

/**
 * Getter used to retrieve a player actor's nature
 * @return The fact that the actor is a player
 */
Actor::ID Player::get_type() const
{
    return Actor::PLAYER;
}

/**
 * Textual output containing the Player's information
 * @return Text with the player's info
 */
std::string Player::to_string() const
{
    std::stringstream ss;
    ss << "Player: " << Movable::to_string();
    return ss.str();
}

/**
 * Redefinition of the << operator
 * @param os The desired output stream
 * @param player The player we wish to output
 * @return Textual output containing the player's information
 */
std::ostream &sokoban::model::operator<<( std::ostream &os, const Player &player )
{
    os << player.to_string();
    return os;
}
```

9.3.11 Wall.hpp

```cpp
#ifndef SOKOBAN_WALL_HPP
#define SOKOBAN_WALL_HPP

#include "Actor.hpp"

#include <map>
#include <array>
#include <string>
#include <ostream>

namespace sokoban
{
    namespace model
    {
        class Wall
                : public Actor
        {
        public:
            Wall( float x, float y );
            Wall( const Wall &wall );
            Wall &operator=( const Wall &wall );
            ~Wall() override;
            ID get_type() const override;
            std::string to_string() const override;
            friend std::ostream &operator<<( std::ostream &os, const Wall &wall );
        };
    }
}

#endif //SOKOBAN_WALL_HPP
```

9.3.12 Wall.cpp

```cpp
#include "Wall.hpp"

#include "../util/Logger.hpp"

#include <sstream>
#include <iostream>

using namespace sokoban::model;
using namespace sokoban::util;

/**
 * Default constructor for the walls
 * @param x The X coordinate on the board
 * @param y The Y coordinate on the board
 */
Wall::Wall( float x, float y )
        : Actor( x, y )
{
}

/**
 * Copy constructor for the wall
 * @param wall The wall we wish to copy the information from
 */
Wall::Wall( const Wall &wall )
        : Wall( wall.get_x(), wall.get_y() )
{
}

/**
 * Redefinition of the = operator
 * @param wall The wall we wish to copy the information from
 * @return The new instance of wall with the copied information
 */
Wall &Wall::operator=( const Wall &wall )
{
    if ( &wall != this )
    {
        set_x( wall.get_x() );
        set_y( wall.get_y() );
    }
    return *this;
}

/**
 * Default destructor for the wall actor
 */
Wall::~Wall()
{
    Logger::log( LoggerLevel::INFO, "Deletion " + Wall::to_string() );
}

/**
 * Get the nature of a wall
 * @return The fact that it is a wall actor
 */
Actor::ID Wall::get_type() const
{
    return Actor::WALL;
}

/**
 * Textual output stating the wall's information
 * @return The textual output with the wall's information
 */
std::string Wall::to_string() const
{
    std::stringstream ss;
    ss << "Wall: " << Actor::to_string();
    return ss.str();
}

/**
 * Redefinition of the << operator
 * @param os The desired output stream
 * @param wall The wall we wish to output
 * @return Textual output containing the wall's information
 */
std::ostream &sokoban::model::operator<<( std::ostream &os, const Wall &wall )
{
    os << wall.to_string();
    return os;
}
```

9.3.13 Board.hpp

```
#ifndef SOKOBAN_BOARD_HPP
#define SOKOBAN_BOARD_HPP

#include "Actor.hpp"
#include "Box.hpp"
#include "Wall.hpp"
#include "Player.hpp"
#include "Platform.hpp"

#include <vector>
#include <ostream>

namespace sokoban
{
    namespace model
    {
        /**
         * Board class
         * Meant to build the layout of the leve that the player will be playing
         */
        class Board
        {
        private:
            std::string _level; /** The skeleton of the level */
            std::vector< Box * > _boxes; /** All the boxes in the level */
            std::vector< Wall * > _walls; /** All the walls in the game */
            std::vector< Platform * > _platforms; /** All the platforms in the game */
            Player *_player; /** The main character of the game */
            std::vector< Actor * > _world; /** Every actor in the game */
            float _width; /** The board's maximum width */
            float _height; /** The board's maximum height */
            void init_board();
            void init_world();
            void build_world();
        public:
            enum
            {
                LEFT_COLLISION,
                RIGHT_COLLISION,
                TOP_COLLISION,
                BOTTOM_COLLISION
            };
            explicit Board( const std::string &lvl );
            Board( const Board &board );
            Board &operator=( const Board &board );
            ~Board();
            bool check_wall_collision( Actor *actor, int type );
```

```cpp
            bool check_box_collision( int type );
            float get_board_width() const;
            float get_board_height() const;
            bool is_completed() const;
            std::vector< Actor * > get_world();
            friend std::ostream &operator<<( std::ostream &os, const Board &board );
        };
    }
}

#endif //SOKOBAN_BOARD_HPP
```

9.3.14 Board.cpp

```cpp
#include "Board.hpp"

#include <string>
#include <sstream>
#include <iostream>
#include <fstream>

using namespace sokoban::model;

const int OFFSET = 64; /** Variable used for the amount of pixels an actor can move */
const int SPACE = 64; /** Variable used for the size of each character */

/**
 * Constructor for the board
 * The parameter lvl is meant to be a string containing the path to a text file
 * That text file will contain the skeleton of the level, meaning that from the
 * symbols present on the text file, we can build every actor of the game.
 * # represents a wall
 * @ represents the player
 * . represents a platform
 * $ represents a box
 * An empty space is just emptiness
 * @param lvl The path for the level
 */
Board::Board( const std::string &lvl )
        : _boxes()
          , _walls()
          , _platforms()
          , _player( nullptr )
          , _world()
          , _width( 0 )
          , _height( 0 )
{
    std::string line;
    std::stringstream ss;
    std::ifstream level_file( lvl );
    if ( level_file.is_open() )
    {
        while ( std::getline( level_file, line ) )
        {
            ss << line << '\n';
        }
        level_file.close();
    }
    _level = std::move( ss.str() );
    init_board();
}

/**
 * Copy constructor of the board
 * Primarily used to create a backup of the board.
 * An example of usage for this constructor could be to load a board when
 * the player hits the reset button
 * @param board The board from which we wish to create a copy
 */
Board::Board( const Board &board )
        : Board( board._level )
{
}

/**
 * Redefinition of the = operator
 * Same usage as the copy constructor
 * @param board
 * @return
 */
Board &Board::operator=( const Board &board )
{
    if ( &board != this )
    {
        _level = board._level;
        _boxes = board._boxes;
        _walls = board._walls;
        _platforms = board._platforms;
        _player = board._player;
        _world = board._world;
        _width = board._width;
        _height = board._height;
    }
    init_board();
    return *this;
}

/**
 * Board's destructor
 * Meant to clear the memory from any actor that has been initialised
 */
Board::~Board()
{
    for ( Wall *wall: _walls )
    {
        delete wall;
    }
    for ( Platform *platform: _platforms )
    {
        delete platform;
    }
    for ( Box *box: _boxes )
    {
        delete box;
    }
    delete _player;
    _boxes.clear();
    _walls.clear();
    _platforms.clear();
    _world.clear();
}

/**
 * Executions before the world initialization
 */
void Board::init_board()
{
    init_world();
}

/**
 * World initialization
 * Meant to initialise every actor on the board
 * As stated in the constructor:
 * * # represents a wall
 * @ represents the player
 * . represents a platform
 * $ represents a box
 * An empty space is just emptiness
 * Each actor is inserted into its respective vector
 */
void Board::init_world()
{
    _boxes = std::vector< Box * >();
    _walls = std::vector< Wall * >();
    _platforms = std::vector< Platform * >();
    float x = OFFSET;
    float y = OFFSET;

    Box *box;
    Wall *wall;
    Platform *platform;
    for ( char item: _level )
    {
        switch ( item )
        {
            case '\n':
                y += SPACE;
                if ( _width < x )
                {
                    _width = x;
                }
                x = OFFSET;
                break;
            case '#':
                wall = new Wall( x, y );
                _walls.insert( _walls.begin(), wall );
                x += SPACE;
                break;
            case '$':
                box = new Box( x, y );
                _boxes.insert( _boxes.begin(), box );
                x += SPACE;
                break;
            case '.':
                platform = new Platform( x, y );
                _platforms.insert( _platforms.begin(), platform );
                x += SPACE;
                break;
            case '@':
                _player = new Player( x, y );
                x += SPACE;
                break;
            case '*':
                box = new Box( x, y );
                platform = new Platform( x, y );
                _boxes.insert( _boxes.begin(), box );
                _platforms.insert( _platforms.begin(), platform );
                x += SPACE;
                break;
            case '+':
                _player = new Player( x, y );
                platform = new Platform( x, y );
                _platforms.insert( _platforms.begin(), platform );
                x += SPACE;
                break;
            case ' ':
                x += SPACE;
                break;
            default:
                break;
        }
        _height = y;
    }
    build_world();
}

/**
 * Definition of each actor in their respective category.
 * This will allow is to easily clear out their memory simultaneously
 */
void Board::build_world()
{
    _world = std::vector< Actor * >();
    for ( Wall *wall: _walls )
    {
        _world.insert( _world.begin(), wall );
    }
    for ( Platform *platform: _platforms )
    {
        _world.insert( _world.begin(), platform );
    }
    for ( Box *box: _boxes )
    {
        _world.insert( _world.begin(), box );
    }
    _world.insert( _world.begin(), _player );
}

/**
 * Function to check whether there is a collision between an actor and a wall
 * @param actor The actor causing the collision
 * @param type The type of collision: TOP, BOTTOM, LEFT, RIGHT
 * @return true if a collision took place, false if not
 */
bool Board::check_wall_collision( Actor *actor, int type )
{
    switch ( type )
    {
        case LEFT_COLLISION:
            for ( const Wall *wall: _walls )
            {
                if ( actor->is_left_collision( wall ) )
                {
                    return true;
                }
            }
            return false;
        case RIGHT_COLLISION:
            for ( const Wall *wall: _walls )
            {
                if ( actor->is_right_collision( wall ) )
                {
                    return true;
                }
            }
            return false;
        case TOP_COLLISION:
            for ( const Wall *wall: _walls )
            {
                if ( actor->is_top_collision( wall ) )
                {
                    return true;
                }
            }
            return false;
        case BOTTOM_COLLISION:
            for ( const Wall *wall: _walls )
            {
                if ( actor->is_bottom_collision( wall ) )
                {
                    return true;
                }
            }
            return false;
        default:
            break;
    }
    return false;
}

/**
 * Functions that checks whether there is a collision between an actor and a box
 * @param type The type of collision: TOP, BOTTOM, LEFT, RIGHT
 * @return true if a collision took place, false if not
 */
bool Board::check_box_collision( int type )
{
    switch ( type )
    {
        case LEFT_COLLISION:
            for ( Box *box: _boxes )
            {
                if ( _player->is_left_collision( box ) )
                {
                    for ( Box *item: _boxes )
                    {
                        if ( box != item )
                        {
                            if ( box->is_left_collision( item ) )
                            {
                                return true;
                            }
                        }
                        if ( check_wall_collision( box, LEFT_COLLISION ) )
                        {
                            return true;
                        }
                    }
                    box->move( -SPACE, 0 );
                    is_completed();
                }
            }
            return false;
        case RIGHT_COLLISION:
            for ( Box *box: _boxes )
            {
                if ( _player->is_right_collision( box ) )
                {
                    for ( Box *item: _boxes )
                    {
                        if ( box != item )
                        {
                            if ( box->is_right_collision( item ) )
                            {
                                return true;
                            }
                        }
                        if ( check_wall_collision( box, RIGHT_COLLISION ) )
                        {
                            return true;
                        }
                    }
                    box->move( SPACE, 0 );
                    is_completed();
                }
            }
            return false;
        case TOP_COLLISION:
            for ( Box *box: _boxes )
            {
                if ( _player->is_top_collision( box ) )
                {
                    for ( Box *item: _boxes )
                    {
                        if ( box != item )
                        {
                            if ( box->is_top_collision( item ) )
                            {
                                return true;
                            }
                        }
                        if ( check_wall_collision( box, TOP_COLLISION ) )
                        {
                            return true;
                        }
                    }
                    box->move( 0, -SPACE );
                    is_completed();
                }
            }
            return false;
        case BOTTOM_COLLISION:
            for ( Box *box: _boxes )
            {
                if ( _player->is_bottom_collision( box ) )
                {
                    for ( Box *item: _boxes )
                    {
                        if ( box != item )
                        {
                            if ( box->is_bottom_collision( item ) )
                            {
                                return true;
                            }
                        }
                        if ( check_wall_collision( box, BOTTOM_COLLISION ) )
                        {
                            return true;
                        }
                    }
                    box->move( 0, SPACE );
                    is_completed();
                }
            }
            return false;
        default:
            break;
    }
    return false;
}

/**
 * Getter for the board width
 * @return The width of the board
 */
float Board::get_board_width() const
{
    return _width;
}

/**
 * Getter for the board height
 * @return The height of the board
 */
float Board::get_board_height() const
{
    return _height;
}

/**
 * Function that allows the played to know whether the level is completed
 * @return True if all the boxes are on the platforms
 */
bool Board::is_completed() const
{
    unsigned long number_of_boxes = _boxes.size();
    int finished_boxes = 0;

```

```cpp
    for ( Box *box: _boxes )
    {
        for ( Platform *platform: _platforms )
        {
            if ( box->get_x() == platform->get_x() && box->get_y() == platform->get_y() )
            {
                finished_boxes += 1;
            }
        }
    }

    if ( finished_boxes == number_of_boxes )
    {
        return true;
    }

    return false;
}

/**
 * Redefinition of the << operator
 * Meant to be used when outputting the board textually
 * @param os The base output stream
 * @param board The board to output
 * @return Textual output of the various actor's positions
 */
std::ostream &sokoban::model::operator<<( std::ostream &os, const Board &board )
{
    os << board._level;
    for ( Actor *actor: board._world )
    {
        std::cout << *actor << std::endl;
    }
    return os;
}

/**
 * Function to retrieve all the actors on the board
 * @return The vector containing the various actors
 */
std::vector< Actor * > Board::get_world()
{
    return _world;
}
```

9.4 Util

9.4.1 Logger.hpp

```cpp
#ifndef SOKOBAN_LOGGER_HPP
#define SOKOBAN_LOGGER_HPP

#include <string>

namespace sokoban
{
    namespace util
    {
        enum LoggerLevel
        {
            TRACE, /** The lowest level of importance leaving nothing but the existance of the information
 */
            DEBUG, /** Debugging information meant for developers */
            INFO, /** Textual information meant for the user */
            WARNING, /** Warning information concerning the program's execution */
            ERROR, /** Critical information that requires patch fix */
            FATAL /** Prioritary level of logging that requires immediate fix */
        };

        class Logger
        {
        private:
            std::string _id; /** The logger's unique ID */
            std::string _file_name; /** The logger's name */
        public:
            Logger( const std::string &id, const std::string &file_name );
            explicit Logger( const std::string &id );
            ~Logger();
            std::string get_id();
            std::string get_file_name();
            static void log( int level, const std::string &log_message );
        };
    }
}

#endif //SOKOBAN_LOGGER_HPP
```

9.4.2 Logger.hpp

```
#include "Logger.hpp"

#include <iostream>

#include <boost/locale/generator.hpp>

#include <boost/date_time/posix_time/posix_time_types.hpp>

#include <boost/log/core.hpp>
#include <boost/log/trivial.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/sinks/text_file_backend.hpp>
#include <boost/log/utility/setup/file.hpp>
#include <boost/log/utility/setup/common_attributes.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/sources/record_ostream.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/log/support/date_time.hpp>

using namespace sokoban::util;
namespace logging = boost::log;

/**
 * Injection operator definition meant to show the severity level in string format
 *
 * @tparam CharT template's parameter for the character
 * @tparam TraitsT template's paramenter for the traits
 * @param strm The output stream
 * @param lvl The severity level
 * @return The string containing the severity level
 */
template< typename CharT, typename TraitsT >
    inline std::basic_ostream< CharT, TraitsT > &operator<<(
            std::basic_ostream< CharT, TraitsT > &strm, logging::trivial::severity_level lvl )
    {
        static const char *const str[] =
                {
                        "trace"
                        , "debug"
                        , "info"
                        , "warning"
                        , "error"
                        , "fatal"
                };
        if ( static_cast< std::size_t >(lvl) < ( sizeof( str ) / sizeof( *str ) ) )
        {
            strm << str[ lvl ];
        }
        else
        {
            strm << static_cast< int >(lvl);
        }
        return strm;
    }

BOOST_LOG_ATTRIBUTE_KEYWORD( severity, "Severity", logging::trivial::severity_level )
BOOST_LOG_ATTRIBUTE_KEYWORD( timestamp, "TimeStamp", boost::posix_time::ptime )

/**
 * Logger's constructor with a specific filename
 * @param id The logger's unique I
 * @param file_name The logger's custom filename
 */
Logger::Logger( const std::string &id, const std::string &file_name )
{
    this->_id = id;
    this->_file_name = file_name;

    boost::shared_ptr< logging::sinks::synchronous_sink< logging::sinks::text_file_backend > > sink =
     logging::add_file_log
            (
                    logging::keywords::file_name = get_file_name(),
                    logging::keywords::rotation_size = 10 * 1024 * 1024,
                    logging::keywords::time_based_rotation = logging::sinks::file::rotation_at_time_point(
     0, 0, 0 ),
                    logging::keywords::format = logging::expressions::stream
                            << logging::expressions::format_date_time( timestamp, "%Y-%m-%d, %H:%M:%S.%f" )
```

```cpp
                            << " <" << severity.or_default( logging::trivial::trace )
                            << "> " << logging::expressions::message,
                    logging::keywords::auto_flush = true,
                    logging::keywords::open_mode = std::ios_base::app
            );

    std::locale loc = boost::locale::generator()( "en_US.UTF-8" );
    sink->imbue( loc );

    //logging::core::get()->set_filter( severity >= logging::trivial::info );

    logging::add_common_attributes();

    //logging::sources::severity_logger<logging::trivial::severity_level> lg;
}

/**
 * Logger's constructor initializing its unique id and default name
 * @param id The unique ID of the logger
 */
Logger::Logger( const std::string &id )
        : Logger( id, "sokoban.log" )
{

}

/**
 * Upon destruction of the logger, print out its identifier
 */
Logger::~Logger()
{
    std::cout << "logger ID: " << get_id() << std::endl;
}

/**
 * Getter for the logger's unique identifier.
 * @return The ID
 */
std::string Logger::get_id()
{
    return _id;
}

/**
 * Getter for the logger's file name
 * @return The name of the logger.
 */
std::string Logger::get_file_name()
{
    return _file_name;
}

/**
 * Function that will append the information requested throughout the execution of the program.
 * @param level The severity level
 * @param log_message The message to append
 */
void Logger::log( int level, const std::string &log_message )
{
    using namespace logging::trivial;

    const char *message = log_message.c_str();

    switch ( level )
    {
        case LoggerLevel::TRACE:
            BOOST_LOG_TRIVIAL( trace ) << message;
            break;
        case LoggerLevel::DEBUG:
            BOOST_LOG_TRIVIAL( debug ) << message;
            break;
        case LoggerLevel::INFO:
            BOOST_LOG_TRIVIAL( info ) << message;
            break;
        case LoggerLevel::WARNING:
            BOOST_LOG_TRIVIAL( warning ) << message;
            break;
        case LoggerLevel::ERROR:
            BOOST_LOG_TRIVIAL( error ) << message;
            break;
        case LoggerLevel::FATAL:
            BOOST_LOG_TRIVIAL( fatal ) << message;
            break;
        default:
            BOOST_LOG_TRIVIAL( trace ) << message;
            break;
    }
}
```

9.5 UI

9.5.1 Category.hpp

```
#ifndef SOKOBAN_CATEGORY_HPP
#define SOKOBAN_CATEGORY_HPP

namespace sokoban
{
    namespace ui
    {
        namespace Category
        {
            /**
             * Various types of actors and effects could be initialized
             */
            enum Type
            {
                None = 0, /** Absolute nothing */
                Scene = 1 << 0, /** The actual Scene */
                Player = 1 << 1, /** The player itself */
                Box = 1 << 2, /** The various boxes */
                Platform = 1 << 3, /** The various platforms */
                Wall = 1 << 4, /** The various walls */
                Player_Movement = 1 << 5, /** The effect concerning the player movement */
                Box_Movement = 1 << 6, /** The effect concerning the box's movement */
                Sound_Effect = 1 << 7, /** Any various sound effect */

                Actor = Player | Box | Platform | Wall,
            };
        }
    }
}

#endif //SOKOBAN_CATEGORY_HPP
```

9.5.2 Menu.hpp

```
#ifndef SOKOBAN_MENU_HPP
#define SOKOBAN_MENU_HPP

namespace sokoban
{
    namespace ui
    {
        class Menu
        {
        public:
            Menu();
            ~Menu();
            unsigned short launch_application() const;
        private:
            int _res; /** The end result for the game's execution */
        };
    }
}

#endif //SOKOBAN_MENU_HPP
```

9.5.3 Menu.cpp

```
#include "Menu.hpp"

#include "gui/Application.hpp"

using namespace sokoban::ui;
using namespace sokoban::ui::gui;
using namespace sokoban::util;

/**
 * Menu's constructor meant to initialize and execute the game.
 * Upon execution, a return code is entered.
 * Should any exception arise, the program's return code will be -1.
 */
Menu::Menu()
{
    try
    {
        Logger::log( LoggerLevel::INFO, "Init Main Frame" );
        Application main_frame;
        _res = main_frame.run();
    }
    catch ( std::exception &e )
    {
        Logger::log( LoggerLevel::ERROR, e.what() );
        _res = -1;
    }
}

/**
 * Menu's destructor
 */
Menu::~Menu()
= default;

/**
 * Function meant to return the games execution code
 * @return 0 if run correctly, -1 if not
 */
unsigned short Menu::launch_application() const
{
    return _res;
}
```

9.5.4 Resource_Holder.hpp

```
#ifndef SOKOBAN_RESOURCE_HOLDER_HPP
#define SOKOBAN_RESOURCE_HOLDER_HPP

#include <map>
#include <memory>
#include <string>
#include <cassert>
#include <stdexcept>

namespace sokoban
{
    namespace ui
    {
        /**
         * Resource holder meant to manage various natures of a Resource.
         * @tparam Resource Could be a Sprite or a Sound
         * @tparam Identifier Unique identifier in an enum
         */
        template< typename Resource, typename Identifier >
            class Resource_Holder
            {
            public:
                /** Loads the requested asset based on its ID and filename */
                void load( Identifier id, const std::string &filename );
                template< typename Parameter >
                    /** Loads the requested asset based on its ID and filename plus force a parameter */
                    void load( Identifier id, const std::string &filename, const Parameter &second_param );
                /** Resource getter meant to ensure the retrieval of a non-const Resource */
                Resource &get( Identifier id );
                /** Resource getter meant to ensure the retrieval of a const Resource */
                const Resource &get( Identifier id ) const;
            private:
                /** Map containing the various resources */
                std::map< Identifier, std::unique_ptr< Resource > > _resource_map;
                /** Insertion of resource into map upon execution of load */
                void insert_resource( Identifier id, std::unique_ptr< Resource > resource );
            };

#include "Resource_Holder.inl"

    }
}

#endif //SOKOBAN_RESOURCE_HOLDER_HPP
```

9.5.5 Resource_Holder.inl

```

/**
 * Function meant to load a resource inside the map containing the various assets
 * @tparam Resource The type of resource
 * @tparam Identifier Its unique identifier in the enum
```

```cpp
 * @param id Its unique identifier in the enum
 * @param filename The filename of the asset
 */
template< typename Resource, typename Identifier >
    void Resource_Holder< Resource, Identifier >::load( Identifier id, const std::string &filename )
    {
        // Create and load resource
        std::unique_ptr< Resource > resource( new Resource() );
        if ( !resource->loadFromFile( filename ) )
        {
            throw std::runtime_error( "Resource_Holder::load - Failed to load " + filename );
        }

        // If loading successful, insert resource to map
        insert_resource( id, std::move( resource ) );
    }

/**
 * Function meant to load an asset with a specific parameter that characterizes it.
 * @tparam Resource The type of resource to load
 * @tparam Identifier Its unique identifier within the enum
 * @tparam Parameter The nature of the parameter
 * @param id Its unique identifier within the enum
 * @param filename The filename of the asset
 * @param second_param The parameter that characterizes the asset
 */
template< typename Resource, typename Identifier >
template< typename Parameter >
    void Resource_Holder< Resource, Identifier >::load( Identifier id, const std::string &filename, const Parameter &second_param )
    {
        // Create and load resource
        std::unique_ptr< Resource > resource( new Resource() );
        if ( !resource->loadFromFile( filename, second_param ) )
        {
            throw std::runtime_error( "Resource_Holder::load - Failed to load " + filename );
        }

        // If loading successful, insert resource to map
        insert_resource( id, std::move( resource ) );
    }

/**
 * Getter for the asset based on its unique identifier within an enum from a non-const Resource
 * @tparam Resource The type of resource to retrieve
 * @tparam Identifier Its unique identifier within the enum
 * @param id Its unique identifier within the enum
 * @return The asset requested
 */
template< typename Resource, typename Identifier >
    Resource &Resource_Holder< Resource, Identifier >::get( Identifier id )
    {
        auto found = _resource_map.find( id );
        assert( found != _resource_map.end() );

        return *found->second;
    }

/**
 * Getter for the asset based on its unique identifier within an enum from a const Resource
 * @tparam Resource The type of resource to retrieve
 * @tparam Identifier Its unique identifier within the enum
 * @param id Its unique identifier within the enum
 * @return The asset requested
 */
template< typename Resource, typename Identifier >
    const Resource &Resource_Holder< Resource, Identifier >::get( Identifier id ) const
    {
        auto found = _resource_map.find( id );
        assert( found != _resource_map.end() );

        return *found->second;
    }

/**
 * Function meant to insert a new resource inside the Resource Holder's map
 * @tparam Resource The nature of the Resource
 * @tparam Identifier Its unique identifier within the enum
 * @param id Its unique identifier within the enum
 * @param resource The Resource we want to insert
 */
template< typename Resource, typename Identifier >
    void Resource_Holder< Resource, Identifier >::insert_resource( Identifier id, std::unique_ptr< Resource > resource )
    {
        // Insert and check success
        auto inserted = _resource_map.insert( std::make_pair( id, std::move( resource ) ) );
        assert( inserted.second );
    }
```

9.5.6 Utility.inl

```
#ifndef SOKOBAN_UTILITY_INL
#define SOKOBAN_UTILITY_INL

/**
 * Function meant to return a stringstream to stream.
 * @tparam T The nature of the entity that will call this function
 * @param value The value to insert into the stringstream
 * @return The stringstream in string format
 */
template< typename T >
    std::string to_string( const T &value )
    {
        std::stringstream stream;
        stream << value;
        return stream.str();
    }

#endif //SOKOBAN_UTILITY_INL
```

9.6 GUI

9.6.1 Resource_Identifiers.hpp

```
#ifndef SOKOBAN_RESOURCE_IDENTIFIERS_HPP
#define SOKOBAN_RESOURCE_IDENTIFIERS_HPP

namespace sf
{
    class Texture;
    class Font;
    class Shader;
    class SoundBuffer;
}

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            namespace Textures
            {
                /**
                 * The various textures for each actor and entity
                 */
                enum ID
                {
                    Player, /** The player texture */
                    Box, /** The box texture */
                    Platform, /** The platform texture */
                    Wall, /** The box texture */
                    Background, /** The background texture */
                    Title_Screen, /** The title screen */
                    Button, /** The button texture */
                };
            }

            /**
             * The various shaders that can be displayed on the game
             */
            namespace Shaders
            {
                enum ID
                {
                    Brightness_Pass, /** The Brightness levels */
                    Down_Sample_Pass, /** The samples downscaling */
                    Gaussian_Blur_Pass, /** The Blur effect */
                    /* Add_Pass, */
                };
            }

            /**
             * The various fonts displayed throughout the game
             */
```

```cpp
            namespace Fonts
            {
                enum ID
                {
                    Connection_II, /** Minecraft-like font */
                    Kodomo_Rounded, /** Kiddie Japanese font */
                    Free_Font, /** GNU Free font */
                    Rampart_One, /** Bubbly font */
                };
            }

            /**
             * The various sound effects that can be heard throughout the usage of the software
             */
            namespace Sound_Effect
            {
                enum ID
                {
                    Player_Footsteps_Outdoor_Boots,
                    Player_Footsteps_Outdoor_Boots_On_Wood,
                    Player_Footsteps_Shoes_Fast_01,
                    Player_Footsteps_Shoes_On_Wood_01,
                    Player_Footsteps_Shoes_On_Wood_02,
                    Player_Footsteps_Snow_01,
                    Player_Footsteps_Snow_02,
                    Player_Footsteps_Soft_Fast,
                    Player_Footsteps_Stair_Ascent_Creaky,
                    Player_Footsteps_Water_01,
                    Player_Footsteps_Water_02,
                    Box_Movement,
                    Level_Complete,
                    Game_Complete,
                    Button_Beep_01,
                    Button_Beep_02,
                    Button_Click_01,
                    Button_Click_02,
                    Button_Click_03,
                    Button_Pop_01,
                    Button_Pop_02,
                    Button_Switch_01,
                    Button_Switch_02,
                };
            }

            /**
             * The various songs that can be played throughout the game
             */
            namespace Music
            {
                enum ID
                {
                    Field_Desolate_Domain,
                    Field_Golden_Harvest,
                    Field_Greatest_Nation,
                    Field_Hades_Holiday,
                    Field_Homeland_Return,
                    Field_Shattered_Land,
                    Theme_Courting_The_Princess,
                    Theme_Festival_Fun,
                    Theme_Funeral_March,
                    Theme_Tournament_Time,
                    Town_Childhood_Home,
                    Town_Fancy_Castle,
                    Town_Little_Village,
                    Town_Nice_Day_Out,
                    Town_Old_Palace,
                    Town_Peaceful_Place,
                    Town_Pleasant_peasants,
                    Town_Shop_Hopping,
                    Town_Spooky_Manor,
                    Town_Tavern_Tune,
                };
            }
        }
        template< typename Resource, typename Identifier >
            class Resource_Holder;

        typedef Resource_Holder< sf::Texture, gui::Textures::ID > Texture_Holder; /** The texture holder */
        typedef Resource_Holder< sf::Font, gui::Fonts::ID > Font_Holder; /** The font holder */
        typedef Resource_Holder< sf::Shader, gui::Shaders::ID > Shader_Holder; /** The shader holder */
        typedef Resource_Holder< sf::SoundBuffer, gui::Sound_Effect::ID > Sound_Buffer_Holder; /** The sound effects holder */
    }
}
```

```

#endif //SOKOBAN_RESOURCE_IDENTIFIERS_HPP
```

9.6.2 Animation.hpp

```
#ifndef SOKOBAN_ANIMATION_HPP
#define SOKOBAN_ANIMATION_HPP

#include <SFML/Graphics/Sprite.hpp>
#include <SFML/System/Time.hpp>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class Animation
        : public sf::Drawable
        , public sf::Transformable
            {
            public:
                Animation();
                explicit Animation( const sf::Texture &texture );

                void set_texture( const sf::Texture &texture );
                const sf::Texture *get_texture() const;

                void set_frame_size( sf::Vector2i frame_size );
                sf::Vector2i get_frame_size() const;

                void set_num_frames( std::size_t num_frames );
                std::size_t get_num_frames() const;

                void set_repeating( bool flag );
                bool is_repeating() const;

                void restart();
                bool is_finished() const;

                sf::FloatRect get_local_bounds() const;
                sf::FloatRect get_global_bounds() const;

                void update( sf::Time dt );

            private:
                sf::Sprite _sprite; /** The current sprite to display */
                sf::Vector2i _frame_size; /** The sprite's frame size */
                std::size_t _num_frames; /** The number of frames */
                std::size_t _current_frame; /** The current frame */
                sf::Time _duration; /** The duration of the animation */
                sf::Time _elapsed_time; /** The time that went by */
                bool _repeat; /** Whether the animation has to be repeated or not */
                void draw( sf::RenderTarget &target, sf::RenderStates states ) const override;
            };
        }
    }
}

#endif //SOKOBAN_ANIMATION_HPP
```

9.6.3 Animation.cpp

```
#include "Animation.hpp"

#include <SFML/Graphics/RenderTarget.hpp>
#include <SFML/Graphics/Texture.hpp>

using namespace sokoban::ui::gui;

/**
 * Default constructor for the animation
 */
Animation::Animation()
: _sprite()
, _frame_size()
, _num_frames( 0 )
, _current_frame( 0 )
, _duration( sf::Time::Zero )
, _elapsed_time( sf::Time::Zero )
, _repeat( false )
{
}

/**
 * Default constructor for the animation based on a specific texture
 * @param texture The texture to initialize
 */
Animation::Animation( const sf::Texture &texture )
: _sprite( texture )
, _frame_size()
, _num_frames( 0 )
, _current_frame( 0 )
, _duration( sf::Time::Zero )
, _elapsed_time( sf::Time::Zero )
, _repeat( false )
{
}

/**
 * Setter for the animation's texture
 * @param texture The new texture to set
 */
void Animation::set_texture( const sf::Texture &texture )
{
    _sprite.setTexture( texture );
}

/**
 * Getter for the current active texture
 * @return The texture that is displayed
 */
const sf::Texture *Animation::get_texture() const
{
    return _sprite.getTexture();
}

/**
 * Setter for the current frame size
 * @param frame_size The new frame size
 */
void Animation::set_frame_size( sf::Vector2i frame_size )
{
    _frame_size = frame_size;
}

/**
 * Getter for the frame size
 * @return The current frame size
 */
sf::Vector2i Animation::get_frame_size() const
{
    return _frame_size;
}

/**
 * Setter for the number of frames
 * @param num_frames The new number of frames
 */
void Animation::set_num_frames( std::size_t num_frames )
{
    _num_frames = num_frames;
}

/**
 * Getter for the number of frames
 * @return The number of frames
 */
std::size_t Animation::get_num_frames() const
{
    return _num_frames;
}

/**
 * Setter that defines whether the animation loops or not
 * @param flag Whether the animation loops or not
 */
void Animation::set_repeating( bool flag )
{
    _repeat = flag;
}

/**
 * Getter that retrieves whether the animation loops or not
 * @return whether the animation loops or not
 */
```

```cpp
bool Animation::is_repeating() const
{
    return _repeat;
}

/**
 * Make the animation restart from the very first frame
 */
void Animation::restart()
{
    _current_frame = 0;
}

/**
 * Function that checks whether an animation has ended or not
 * @return Whether an animation has ended or not
 */
bool Animation::is_finished() const
{
    return _current_frame >= _num_frames;
}

/**
 * Getter for the maximum bounds of an animation
 * @return The maximum bounds of an animation
 */
sf::FloatRect Animation::get_local_bounds() const
{
    return sf::FloatRect( getOrigin(), static_cast< sf::Vector2f >( get_frame_size() ) );
}

/**
 * The globally accessible bounds
 * @return The globally accessible bounds
 */
sf::FloatRect Animation::get_global_bounds() const
{
    return getTransform().transformRect( get_local_bounds() );
}

/**
 * Realtime update each frame of the animation
 * @param dt The clock time
 */
void Animation::update( sf::Time dt )
{
    sf::Time time_per_frame = _duration / static_cast< float >( _num_frames );
    _elapsed_time += dt;

    sf::Vector2i texture_bounds( _sprite.getTexture()->getSize() );
    sf::IntRect texture_rect = _sprite.getTextureRect();

    if( _current_frame == 0 )
    {
        texture_rect = sf::IntRect( 0, 0, _frame_size.x, _frame_size.y );
    }
    while( _elapsed_time >= time_per_frame && ( _current_frame <= _num_frames || _repeat ) )
    {
        texture_rect.left += texture_rect.width;

        if( texture_rect.left + texture_rect.width > texture_bounds.x )
        {
            texture_rect.left = 0;
            texture_rect.top += texture_rect.height;
        }

        _elapsed_time += time_per_frame;

        if( _repeat )
        {
            _current_frame = ( _current_frame + 1 ) % _num_frames;
            if( _current_frame == 0 )
            {
                texture_rect = sf::IntRect( 0, 0, _frame_size.x, _frame_size.y );
            }
            else
            {
                _current_frame++;
            }
        }
    }
    _sprite.setTextureRect( texture_rect );
}

```

```
/**
 * Visually display the various frames of an animation
 */
void Animation::draw( sf::RenderTarget &target, sf::RenderStates states ) const
{
    states.transform *= getTransform();
    target.draw( _sprite, states );
}
```

9.6.4 Application.hpp

```
#ifndef SOKOBAN_APPLICATION_HPP
#define SOKOBAN_APPLICATION_HPP

#include "../../util/Logger.hpp"
#include "../Resource_Holder.hpp"
#include "Resource_Identifiers.hpp"
#include "entities/Entity_Player.hpp"
#include "states/State_Stack.hpp"
#include "Music_Player.hpp"
#include "Sound_Player.hpp"

#include <SFML/System/Time.hpp>
#include <SFML/Graphics/Text.hpp>
#include <SFML/Graphics/RenderWindow.hpp>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            using namespace sokoban::util;

            class Application
                    : private sf::NonCopyable
            {
            public:
                Application();
                ~Application();
                unsigned short run();
            private:
                static const sf::Time _time_per_frame; /** The amount of frames per second we wish to
     display */
                sf::RenderWindow _window; /** The window where the drawables are shown */
                Texture_Holder _textures; /** The default texture holder */
                Font_Holder _fonts; /** The default font holder */
                Music_Player _music; /** The default music player */
                Sound_Player _sounds; /** The default sound player */
                State_Stack _state_stack; /** The stack containing the various states */
                sf::Text _statistics_text; /** Text with the FPS amount */
                sf::Time _statistics_update_time; /** Timer to update the various statistics */
                std::size_t _statistics_num_frames; /** Number of frames per second globally */
                void process_input();
                void update( const sf::Time &delta_time );
                void render();
                void update_statistics( sf::Time dt );
                void register_states();
            };
        }
    }
}

#endif //SOKOBAN_APPLICATION_HPP
```

9.6.5 Application.cpp

```
#include "Application.hpp"
#include "../../util/Logger.hpp"
#include "Utility.hpp"
#include "states/State.hpp"
#include "states/State_Identifiers.hpp"
#include "states/State_Title.hpp"
#include "states/State_Game.hpp"
#include "states/State_Menu.hpp"
#include "states/State_Pause.hpp"
#include "states/State_Settings.hpp"


using namespace sokoban::ui::gui;
using namespace sokoban::util;

```

```
const sf::Time Application::_time_per_frame = sf::seconds( 1.f / 10.f );

namespace
{
    const int WIDTH = 1920;
    const int HEIGHT = WIDTH / 16 * 9;
    const int BITS_PER_PIXEL = 32;
}

/**
 * Default constructor for the application
 */
Application::Application()
        : _window( sf::VideoMode( WIDTH, HEIGHT, BITS_PER_PIXEL ), "Sokoban", sf::Style::Fullscreen )
          , _textures()
          , _fonts()
          , _music()
          , _sounds()
          , _state_stack( State::Context( _window, _textures, _fonts, _music, _sounds ) )
          , _statistics_text()
          , _statistics_update_time()
          , _statistics_num_frames( 0 )
{
    Logger::log( LoggerLevel::DEBUG, "Setting KeyRepeatedEnabled = false" );
    _window.setKeyRepeatEnabled( false );

    Logger::log( LoggerLevel::DEBUG, "Setting VerticalSyncEnabled = true" );
    _window.setVerticalSyncEnabled( true );

    Logger::log( LoggerLevel::DEBUG, "Loading fonts" );
    _fonts.load( Fonts::Kodomo_Rounded, "assets/fonts/KodomoRounded.otf" );
    _fonts.load( Fonts::Connection_II, "assets/fonts/ConnectionIi-2wj8.otf" );
    _fonts.load( Fonts::Free_Font, "assets/fonts/freefont/FreeSansBold.ttf" );
    _fonts.load( Fonts::Rampart_One, "assets/fonts/RampartOne-Regular.ttf" );

    Logger::log( LoggerLevel::DEBUG, "Loading Title Screen Texture" );
    _textures.load( Textures::Title_Screen, "assets/images/Sample_Sokoban.png" );
    _textures.load( Textures::Button, "assets/images/Buttons.png" );

    Logger::log( LoggerLevel::DEBUG, "Initializing stastistics text" );
    _statistics_text.setFont( _fonts.get( Fonts::Connection_II ) );
    _statistics_text.setPosition( WIDTH / 2.5f, 5.f );
    _statistics_text.setCharacterSize( 10u );
    _statistics_text.setFillColor( sf::Color::Yellow );

    Logger::log( LoggerLevel::DEBUG, "Registering States" );
    register_states();

    Logger::log( LoggerLevel::DEBUG, "Setting Title as first State to load" );
    _state_stack.push_state( States::Title );

    Logger::log( LoggerLevel::DEBUG, "Playing Default Song" );
    _music.play( Music::Town_Pleasant_peasants );
}

/**
 * Application's destructor
 */
Application::~Application()
= default;

/**
 * Realtime update FPS counter
 * @param dt The clock time
 */
void Application::update_statistics( sf::Time dt )
{
    _statistics_update_time += dt;
    _statistics_num_frames += 1;
    if ( _statistics_update_time >= sf::seconds( 1.0f ) )
    {
        _statistics_text.setString(
                "FPS: " + std::to_string( _statistics_num_frames )
        );
        _statistics_update_time -= sf::seconds( 1.0f );
        _statistics_num_frames = 0;
    }
}

/**
 * Globalized Event handler per state within stack
 */
void Application::process_input()
{
```

```cpp
    sf::Event event{};

    while ( _window.pollEvent( event ) )
    {
        _state_stack.handle_event( event );
        if ( event.type == sf::Event::Closed )
        {
            _window.close();
        }
    }

}

/**
 * Realtime update all states within stack
 * @param delta_time The clock time
 */
void Application::update( const sf::Time &delta_time )
{
    _state_stack.update( delta_time );
}

/**
 * Visually display everything that makes up the game
 */
void Application::render()
{
    _window.clear();
    _state_stack.draw();
    _window.setView( _window.getDefaultView() );
    _window.draw( _statistics_text );
    _window.display();
}

/**
 * Execute the program and return the execution code
 * @return 0 if run correctly, -1 if not
 */
unsigned short Application::run()
{
    sf::Clock clock;
    sf::Time time_since_last_update = sf::Time::Zero;
    while ( _window.isOpen() )
    {
        sf::Time dt = clock.restart();
        time_since_last_update += dt;
        while ( time_since_last_update > _time_per_frame )
        {
            time_since_last_update -= _time_per_frame;
            process_input();
            update( _time_per_frame );
            if( _state_stack.is_empty() )
            {
                _window.close();
            }
        }
        update_statistics( dt );
        render();
    }
    return 0;
}

/**
 * Register the various states within the app
 */
void Application::register_states()
{
    Logger::log( LoggerLevel::DEBUG, "Registering Title State" );
    _state_stack.register_state< State_Title >( States::Title );

    Logger::log( LoggerLevel::DEBUG, "Registering Menu State" );
    _state_stack.register_state< State_Menu >( States::Menu );

    Logger::log( LoggerLevel::DEBUG, "Registering Game State" );
    _state_stack.register_state< State_Game >( States::Game );

    Logger::log( LoggerLevel::DEBUG, "Registering Settings State" );
    _state_stack.register_state< State_Settings >( States::Settings );

    Logger::log( LoggerLevel::DEBUG, "Registering Pause State" );
    _state_stack.register_state< State_Pause >( States::Pause );
}
```

9.6.6 Music_Player.hpp

```cpp
#ifndef SOKOBAN_MUSIC_PLAYER_HPP
#define SOKOBAN_MUSIC_PLAYER_HPP

#include "../Resource_Holder.hpp"
#include "Resource_Identifiers.hpp"

#include <SFML/System/NonCopyable.hpp>
#include <SFML/Audio/Music.hpp>

#include <map>
#include <string>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class Music_Player
        : private sf::NonCopyable
            {
            public:
                Music_Player();
                void play( Music::ID song );
                void stop();
                void set_paused( bool paused );
                void set_volume( float volume );
                float get_volume() const;
            private:
                sf::Music _music; /** The music handler */
                std::map< Music::ID, std::string > _filenames; /** The various files that can be played */
                float _volume; /** The music's volume */
            };
        }
    }
}

#endif //SOKOBAN_MUSIC_PLAYER_HPP
```

9.6.7 Music_Player.cpp

```cpp
#include "Music_Player.hpp"

#include "../../util/Logger.hpp"

using namespace sokoban::ui::gui;
using namespace sokoban::util;

/**
 * Default constructor for the music player
 */
Music_Player::Music_Player()
: _music()
, _filenames()
, _volume( 100.f )
{
    _filenames[ Music::Field_Desolate_Domain ] = "assets/music/Field_-_Desolate_Domain.ogg" ;
    _filenames[ Music::Field_Golden_Harvest ] = "assets/music/Field_-_Golden_Harvest.ogg" ;
    _filenames[ Music::Field_Greatest_Nation ] = "assets/music/Field_-_Greatest_Nation.ogg" ;
    _filenames[ Music::Field_Hades_Holiday ] = "assets/music/Field_-_Hades_Holiday.ogg" ;
    _filenames[ Music::Field_Homeland_Return ] = "assets/music/Field_-_Homeland_Return.ogg" ;
    _filenames[ Music::Field_Shattered_Land ] = "assets/music/Field_-_Shattered_Lands.ogg" ;
    _filenames[ Music::Theme_Courting_The_Princess ] = "assets/music/Theme_-_Courting_The_Princess.ogg" ;
    _filenames[ Music::Theme_Festival_Fun ] = "assets/music/Theme_-_Festival_Fun.ogg" ;
    _filenames[ Music::Theme_Funeral_March ] = "assets/music/Theme_-_Funeral_March.ogg" ;
    _filenames[ Music::Theme_Tournament_Time ] = "assets/music/Theme_-_Tournament_Time.ogg" ;
    _filenames[ Music::Town_Childhood_Home ] = "assets/music/Town_-_Childhood_Home.ogg" ;
    _filenames[ Music::Town_Fancy_Castle ] = "assets/music/Town_-_Fancy_Castle.ogg" ;
    _filenames[ Music::Town_Little_Village ] = "assets/music/Town_-_Little_Village.ogg" ;
    _filenames[ Music::Town_Nice_Day_Out ] = "assets/music/Town_-_Nice_Day_Out.ogg" ;
    _filenames[ Music::Town_Old_Palace ] = "assets/music/Town_-_Old_Palace.ogg" ;
    _filenames[ Music::Town_Peaceful_Place ] = "assets/music/Town_-_Peaceful_Place.ogg" ;
    _filenames[ Music::Town_Pleasant_peasants ] = "assets/music/Town_-_Pleasant_Peasants.ogg" ;
    _filenames[ Music::Town_Shop_Hopping ] = "assets/music/Town_-_Shop_Hopping.ogg" ;
    _filenames[ Music::Town_Spooky_Manor ] = "assets/music/Town_-_Spooky_Manor.ogg" ;
    _filenames[ Music::Town_Tavern_Tune ] = "assets/music/Town_-_Tavern_Tune.ogg" ;
}

/**
 * Function that plays a song based on its ID
 * @param song The song we wish to play
 */
void Music_Player::play( Music::ID song )
{
    std::string filename = _filenames[ song ];
    if( !_music.openFromFile( filename ) )
    {
        Logger::log( LoggerLevel::ERROR, "Music " + filename + " could not be loaded" );
        throw std::runtime_error( "Music " + filename + " could not be loaded" );
    }
    _music.setVolume( _volume );
    _music.setLoop( true );
    _music.play();
}

/**
 * Function meant to stop a song
 */
void Music_Player::stop()
{
    _music.stop();
}

/**
 * Function meant to pause a song
 * @param paused Whether the song is paused or not
 */
void Music_Player::set_paused( bool paused )
{
    if( paused )
    {
        _music.pause();
    }
    else
    {
        _music.play();
    }
}

/**
 * Volume setter for the song
 * @param volume The value of the volume
 */
void Music_Player::set_volume( float volume )
{
    _volume = volume;
    _music.setVolume( _volume );
}

/**
 * Getter for the volume
 * @return The current value of the volume
 */
float Music_Player::get_volume() const
{
    return _volume;
}
```

9.6.8 Scene_Node.hpp

```cpp
#ifndef SOKOBAN_SCENE_NODE_HPP
#define SOKOBAN_SCENE_NODE_HPP

#include "../Category.hpp"

#include <SFML/System/Time.hpp>
#include <SFML/System/NonCopyable.hpp>
#include <SFML/Graphics/Drawable.hpp>
#include <SFML/Graphics/Transformable.hpp>

#include <set>
#include <memory>
#include <vector>
#include <utility>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            struct Command;
            class Command_Queue;

            class Scene_Node
                    : public sf::Transformable
                      , public sf::Drawable
                      , private sf::NonCopyable
            {
            public:
                typedef std::unique_ptr< Scene_Node > Ptr;
                typedef std::pair< Scene_Node *, Scene_Node * > Pair;
                explicit Scene_Node( Category::Type category = Category::None );
                void attach_child( Ptr child );
                Ptr detach_child( const Scene_Node &node );
                void update( sf::Time dt, Command_Queue &commands );
                sf::Vector2f get_world_positions() const;
                sf::Transform get_world_transform() const;
                void on_command( const Command &command, sf::Time dt );
                virtual unsigned int get_category() const;
                virtual sf::FloatRect get_bounding_rect() const;
                virtual bool is_marked_for_removal() const;
                virtual bool is_destroyed() const;
            private:
                std::vector< Ptr > _children; /** The various children that compose a Scene node */
                Scene_Node *_parent; /** The parent node */
                Category::Type _default_category; /** The default category of each node */
                virtual void update_current( sf::Time dt, Command_Queue &commands );
                void update_children( sf::Time dt, Command_Queue &commands );
                void draw( sf::RenderTarget &target, sf::RenderStates states ) const override;
                virtual void draw_current( sf::RenderTarget &target, sf::RenderStates states ) const;
                void draw_children( sf::RenderTarget &target, sf::RenderStates states ) const;
                void draw_bounding_rect( sf::RenderTarget &target, sf::RenderStates states ) const;
            };
        }
    }
}

#endif //SOKOBAN_SCENE_NODE_HPP
```

9.6.9 Scene_Node.cpp

```
#include "Scene_Node.hpp"

#include "Utility.hpp"

#include <SFML/Graphics/RectangleShape.hpp>
#include <SFML/Graphics/RenderTarget.hpp>

#include <cmath>
#include <cassert>
#include <algorithm>

using namespace sokoban::ui::gui;

/**
 * Default constructor for the scene node
 * @param category The category that characterize the Scene node
 */
Scene_Node::Scene_Node( Category::Type category )
        : _children()
          , _parent( nullptr )
          , _default_category( category )
{
}

/**
 * Function to append a new node child within the Scene node
 * @param child The new child to append
 */
void Scene_Node::attach_child( Scene_Node::Ptr child )
{
    child->_parent = this;
    _children.push_back( std::move( child ) );
}

/**
 * The node that needs to be detached from the children
 * @param node The node to detach
 * @return The node that has been detached
 */
Scene_Node::Ptr Scene_Node::detach_child( const Scene_Node &node )
{
    auto found = std::find_if( _children.begin(), _children.end(), [ & ]( Ptr &p )
    {
        return p.get() == &node;
```

```cpp
    } );
    assert( found != _children.end() );

    Ptr result = std::move( *found );
    result->_parent = nullptr;
    _children.erase( found );
    return result;
}

/**
 * Function that
 * @param dt
 * @param commands
 */
void Scene_Node::update( sf::Time dt, Command_Queue &commands )
{
    update_current( dt, commands );
    update_children( dt, commands );
}

sf::Vector2f Scene_Node::get_world_positions() const
{
    return get_world_transform() * sf::Vector2f();
}

sf::Transform Scene_Node::get_world_transform() const
{
    sf::Transform transform = sf::Transform::Identity;

    for ( const Scene_Node *node = this; node != nullptr; node = node->_parent )
    {
        transform = node->getTransform() * transform;
    }

    return transform;
}

unsigned int Scene_Node::get_category() const
{
    return _default_category;
}

void Scene_Node::update_current( sf::Time dt, Command_Queue &commands )
{
}

void Scene_Node::update_children( sf::Time dt, Command_Queue &commands )
{
    for ( Ptr &child : _children )
    {
        child->update( dt, commands );
    }
}

void Scene_Node::draw( sf::RenderTarget &target, sf::RenderStates states ) const
{
    states.transform *= getTransform();
    draw_current( target, states );
    draw_children( target, states );
    /* FOR DEBUG REASONS */
    draw_bounding_rect( target, states );
}

void Scene_Node::draw_current( sf::RenderTarget &target, sf::RenderStates states ) const
{
}

void Scene_Node::draw_children( sf::RenderTarget &target, sf::RenderStates states ) const
{
    for ( const Ptr &child : _children )
    {
        child->draw( target, states );
    }
}

sf::FloatRect Scene_Node::get_bounding_rect() const
{
    return {};
}

bool Scene_Node::is_marked_for_removal() const
{
    return is_destroyed();
}
```

```cpp

bool Scene_Node::is_destroyed() const
{
    return false;
}

void Scene_Node::draw_bounding_rect( sf::RenderTarget &target, sf::RenderStates states ) const
{
    sf::FloatRect rect = get_bounding_rect();
    sf::RectangleShape shape;
    shape.setPosition( sf::Vector2f( rect.left, rect.top ) );
    shape.setSize( sf::Vector2f( rect.width, rect.height ) );
    shape.setFillColor( sf::Color::Transparent );
    shape.setOutlineColor( sf::Color::Green );
    shape.setOutlineThickness( 1.f );
    target.draw( shape );
}
```

9.6.10 Sound_Node.hpp

```cpp
#ifndef SOKOBAN_SOUND_NODE_HPP
#define SOKOBAN_SOUND_NODE_HPP

#include "Scene_Node.hpp"
#include "Resource_Identifiers.hpp"

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class Sound_Player;
            class Sound_Node
        : private Scene_Node
            {
            public:
                explicit Sound_Node( Sound_Player &player );
                void play_sound( Sound_Effect::ID sound, sf::Vector2f position );
                unsigned int get_category() const override;
            private:
                Sound_Player &_sounds; /** The sound player for the current sound node */
            };
        }
    }
}

#endif //SOKOBAN_SOUND_NODE_HPP
```

9.6.11 Sound_Node.cpp

```cpp
#include "Sound_Node.hpp"

#include "Sound_Player.hpp"

using namespace sokoban::ui::gui;

/**
 * Default constructor for the sound node
 * @param player The Sound Player that interacts with the node
 */
Sound_Node::Sound_Node( Sound_Player &player )
: Scene_Node()
, _sounds( player )
{
}

/**
 * Function that plays the sound effect based on its ID and position on the map
 * @param sound The sound effect to play
 * @param position The position on the map
 */
void Sound_Node::play_sound( Sound_Effect::ID sound, sf::Vector2f position )
{
    _sounds.play( sound, position );
}

/**
 * Category getter that characterizes the current node
 * @return Its category being a sound effect
 */
unsigned int Sound_Node::get_category() const
```

```cpp
{
    return Category::Sound_Effect;
}
```

9.6.12 Sound_Player.hpp

```cpp
#ifndef SOKOBAN_SOUND_PLAYER_HPP
#define SOKOBAN_SOUND_PLAYER_HPP

#include "../Resource_Holder.hpp"
#include "Resource_Identifiers.hpp"

#include <SFML/System/Vector2.hpp>
#include <SFML/System/NonCopyable.hpp>
#include <SFML/Audio/SoundBuffer.hpp>
#include <SFML/Audio/Sound.hpp>

#include <list>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class Sound_Player
        : private sf::NonCopyable
            {
            public:
                Sound_Player();
                void play( Sound_Effect::ID effect );
                void play( Sound_Effect::ID effect, sf::Vector2f position );
                void remove_stopped_sounds();
                void set_listener_position( sf::Vector2f position );
                sf::Vector2f get_listener_position() const;
                void set_volume( float volume );
                float get_volume() const;
            private:
                Sound_Buffer_Holder _sound_buffers; /** Sound Effect Holder */
                std::list< sf::Sound > _sounds; /** List containing the various sound effects */
                float _volume; /** The sound effects' volume */
            };
        }
    }
}

#endif //SOKOBAN_SOUND_PLAYER_HPP
```

9.6.13 Sound_Player.cpp

```cpp
#include "Sound_Player.hpp"

#include "../../util/Logger.hpp"

#include <SFML/Audio/Listener.hpp>

#include <cmath>

using namespace sokoban::ui::gui;
using namespace sokoban::util;

namespace
{
    const float Listener_Z = 0.f;
    const float Attenuation = 8.f;
    const float Min_Distance_2D = 0.f;
    const float Min_Distance_3D = std::sqrt( Min_Distance_2D * Min_Distance_2D + Listener_Z * Listener_Z );
}

/**
 * Default constructor for the Sound Player
 */
Sound_Player::Sound_Player()
: _sound_buffers()
, _sounds()
, _volume( 100.f )
{
    _sound_buffers.load( Sound_Effect::Button_Beep_01, "assets/sounds/button_beep_01.ogg" );
    _sound_buffers.load( Sound_Effect::Button_Beep_02, "assets/sounds/button_beep_02.ogg" );
    _sound_buffers.load( Sound_Effect::Button_Click_01, "assets/sounds/button_click_01.ogg" );
    _sound_buffers.load( Sound_Effect::Button_Click_02, "assets/sounds/button_click_02.ogg" );
    _sound_buffers.load( Sound_Effect::Button_Click_03, "assets/sounds/button_click_03.ogg" );
    _sound_buffers.load( Sound_Effect::Button_Pop_01, "assets/sounds/button_pop_01.ogg" );
    _sound_buffers.load( Sound_Effect::Button_Pop_02, "assets/sounds/button_pop_02.ogg" );
    _sound_buffers.load( Sound_Effect::Player_Footsteps_Outdoor_Boots, "assets/sounds/
     footsteps_outdoor_boots.ogg" );
    _sound_buffers.load( Sound_Effect::Player_Footsteps_Outdoor_Boots_On_Wood, "assets/sounds/
     footsteps_outdoor_boots_on_wood.ogg" );
    _sound_buffers.load( Sound_Effect::Player_Footsteps_Shoes_Fast_01, "assets/sounds/
     footsteps_shoes_fast_01.ogg" );
    _sound_buffers.load( Sound_Effect::Player_Footsteps_Shoes_On_Wood_01, "assets/sounds/
     footsteps_shoes_on_wood_01.ogg" );
    _sound_buffers.load( Sound_Effect::Player_Footsteps_Shoes_On_Wood_02, "assets/sounds/
     footsteps_shoes_on_wood_02.ogg" );
    _sound_buffers.load( Sound_Effect::Player_Footsteps_Snow_01, "assets/sounds/footsteps_snow_01.ogg" );
    _sound_buffers.load( Sound_Effect::Player_Footsteps_Snow_02, "assets/sounds/footsteps_snow_02.ogg" );
    _sound_buffers.load( Sound_Effect::Player_Footsteps_Soft_Fast, "assets/sounds/footsteps_soft_fast.ogg" )
     ;
    _sound_buffers.load( Sound_Effect::Player_Footsteps_Stair_Ascent_Creaky, "assets/sounds/
     footsteps_stair_ascent_creaky.ogg" );
    _sound_buffers.load( Sound_Effect::Player_Footsteps_Water_01, "assets/sounds/footsteps_water_01.ogg" );
    _sound_buffers.load( Sound_Effect::Player_Footsteps_Water_02, "assets/sounds/footsteps_water_02.ogg" );
    for( sf::Sound sound : _sounds )
    {
        sound.setVolume( _volume );
    }
    sf::Listener::setDirection( 0.f, 0.f, -1.f );
}

/**
 * Function meant to play a sound effect
 * @param effect The sound effect to play
 */
void Sound_Player::play( Sound_Effect::ID effect )
{
    play( effect, get_listener_position() );
}

/**
 * Function meant to play a sound effect depending on its position
 * @param effect The sound effect to play
 * @param position The position that defines when a sound effect is played
 */
void Sound_Player::play( Sound_Effect::ID effect, sf::Vector2f position )
{
    _sounds.emplace_back( sf::Sound() );
    sf::Sound &sound = _sounds.back();

    sound.setBuffer( _sound_buffers.get( effect ) );
    sound.setPosition( position.x, position.y, 0.f );
    sound.setAttenuation( Attenuation );
    sound.setMinDistance( Min_Distance_3D );
    sound.setVolume( _volume );

    sound.play();
}

/**
 * Function that removes all stopped sounds
 */
void Sound_Player::remove_stopped_sounds()
{
    _sounds.remove_if( [] ( const sf::Sound &s ) {
        return s.getStatus() == sf::Sound::Stopped;
    } );
}

/**
 * Set the sound effect position
 * @param position The position meant to be listened
 */
void Sound_Player::set_listener_position( sf::Vector2f position )
{
    sf::Listener::setPosition( position.x, position.y, Listener_Z );
}

/**
 * Getter for the sound effect position
 * @return The position of the sound effect
 */
sf::Vector2f Sound_Player::get_listener_position() const
{
    sf::Vector3f position = sf::Listener::getPosition();
    return { position.x, position.y };
}

```

```cpp
/**
 * Volume setter for the various sound effects
 * @param volume The value of volume to set
 */
void Sound_Player::set_volume( float volume )
{
    _volume = volume;
    for( sf::Sound sound : _sounds )
    {
        sound.setVolume( _volume );
    }
}

/**
 * Getter for the value of volume
 * @return The value of the sound effects volume
 */
float Sound_Player::get_volume() const
{
    return _volume;
}
```

9.6.14 Sprite_Node.hpp

```cpp
#ifndef SOKOBAN_SPRITE_NODE_HPP
#define SOKOBAN_SPRITE_NODE_HPP

#include "Scene_Node.hpp"

#include <SFML/Graphics/Sprite.hpp>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class Sprite_Node
                    : public Scene_Node
            {
            public:
                explicit Sprite_Node( const sf::Texture &texture );
                Sprite_Node( const sf::Texture &texture, const sf::IntRect &textureRect );
                void set_texture( const sf::Texture &texture );
                void set_texture( const sf::Texture &texture, const sf::IntRect &textureRect );
            private:
                sf::Sprite _sprite; /** The sprite to display */
                void draw_current( sf::RenderTarget &target, sf::RenderStates states ) const override;
            };
        }
    }
}

#endif //SOKOBAN_SPRITE_NODE_HPP
```

9.6.15 Sprite_Node.cpp

```cpp
#include "Sprite_Node.hpp"

#include <SFML/Graphics/RenderTarget.hpp>

using namespace sokoban::ui::gui;

/**
 * Default constructor for the sprite node
 * @param texture The texture to display the sprite with
 */
Sprite_Node::Sprite_Node( const sf::Texture &texture )
        : _sprite( texture )
{
}

/**
 * Default constructor for the sprite node
 * @param texture The texture to display the sprite with
 * @param textureRect The coordinates of a texture within a sprite sheet
 */
Sprite_Node::Sprite_Node( const sf::Texture &texture, const sf::IntRect &textureRect )
        : _sprite( texture, textureRect )
{

}
```

```cpp

/**
 * Setter for the sprite's texture
 * @param texture The new texture to display
 */
void Sprite_Node::set_texture( const sf::Texture &texture )
{
    _sprite.setTexture( texture );
}

/**
 * Setter for the sprite's texture
 * @param texture The new texture to display
 * @param textureRect The coordinates of the texture within a sprite sheet
 */
void Sprite_Node::set_texture( const sf::Texture &texture, const sf::IntRect &textureRect )
{
    set_texture( texture );
    _sprite.setTextureRect( textureRect );
}

/**
 * Visually display the sprite node
 * @param target The target to display the sprite onto
 * @param states The various states that characterize the window
 */
void Sprite_Node::draw_current( sf::RenderTarget &target, sf::RenderStates states ) const
{
    target.draw( _sprite, states );
}
```

9.6.16 Utility.hpp

```cpp
#ifndef SOKOBAN_UTILITY_HPP
#define SOKOBAN_UTILITY_HPP

#include <SFML/System/Vector2.hpp>
#include <SFML/Window/Keyboard.hpp>

#include <sstream>

namespace sf
{
    class Sprite;
    class Text;
}

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class Animation;
            class Utility
            {
            public:
                template< typename T >
                    std::string to_string( const T &value );
                std::string to_string( sf::Keyboard::Key key );
                static void center_origin( sf::Sprite &sprite );
                static void center_origin( sf::Text &text );
                static void center_origin( Animation &animation );
                float to_degree( float radian );
                float to_radian( float degree );
                int random_int( int exclusive_max );
                float length( sf::Vector2f vector );
                sf::Vector2f unit_vector( sf::Vector2f vector );
            };
#include "../Utility.inl"
        }
    }
}

#endif //SOKOBAN_UTILITY_HPP
```

9.6.17 Utility.cpp

```cpp
#include "Utility.hpp"
#include "Animation.hpp"
#include "components/Button.hpp"

```

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

#include <random>
#include <cmath>
#include <ctime>
#include <cassert>

using namespace sokoban::ui::gui;

namespace
{
    std::default_random_engine create_Random_Engine()
    {
        auto seed = static_cast< unsigned long >( std::time( nullptr ) );
        return std::default_random_engine( seed );
    }
    auto Random_Engine = create_Random_Engine();
}

/**
 * Function meant to return a keyboard key to string
 * @param key The key to be converted
 * @return The key in textual form
 */
std::string Utility::to_string( sf::Keyboard::Key key )
{
#define BOOK_KEY_TO_STRING_CASE( KEY ) case sf::Keyboard::KEY: return #KEY;

    switch ( key )
    {
        BOOK_KEY_TO_STRING_CASE(Unknown)
        BOOK_KEY_TO_STRING_CASE(A)
        BOOK_KEY_TO_STRING_CASE(B)
        BOOK_KEY_TO_STRING_CASE(C)
        BOOK_KEY_TO_STRING_CASE(D)
        BOOK_KEY_TO_STRING_CASE(E)
        BOOK_KEY_TO_STRING_CASE(F)
        BOOK_KEY_TO_STRING_CASE(G)
        BOOK_KEY_TO_STRING_CASE(H)
        BOOK_KEY_TO_STRING_CASE(I)
        BOOK_KEY_TO_STRING_CASE(J)
        BOOK_KEY_TO_STRING_CASE(K)
        BOOK_KEY_TO_STRING_CASE(L)
        BOOK_KEY_TO_STRING_CASE(M)
        BOOK_KEY_TO_STRING_CASE(N)
        BOOK_KEY_TO_STRING_CASE(O)
        BOOK_KEY_TO_STRING_CASE(P)
        BOOK_KEY_TO_STRING_CASE(Q)
        BOOK_KEY_TO_STRING_CASE(R)
        BOOK_KEY_TO_STRING_CASE(S)
        BOOK_KEY_TO_STRING_CASE(T)
        BOOK_KEY_TO_STRING_CASE(U)
        BOOK_KEY_TO_STRING_CASE(V)
        BOOK_KEY_TO_STRING_CASE(W)
        BOOK_KEY_TO_STRING_CASE(X)
        BOOK_KEY_TO_STRING_CASE(Y)
        BOOK_KEY_TO_STRING_CASE(Z)
        BOOK_KEY_TO_STRING_CASE(Num0)
        BOOK_KEY_TO_STRING_CASE(Num1)
        BOOK_KEY_TO_STRING_CASE(Num2)
        BOOK_KEY_TO_STRING_CASE(Num3)
        BOOK_KEY_TO_STRING_CASE(Num4)
        BOOK_KEY_TO_STRING_CASE(Num5)
        BOOK_KEY_TO_STRING_CASE(Num6)
        BOOK_KEY_TO_STRING_CASE(Num7)
        BOOK_KEY_TO_STRING_CASE(Num8)
        BOOK_KEY_TO_STRING_CASE(Num9)
        BOOK_KEY_TO_STRING_CASE(Escape)
        BOOK_KEY_TO_STRING_CASE(LControl)
        BOOK_KEY_TO_STRING_CASE(LShift)
        BOOK_KEY_TO_STRING_CASE(LAlt)
        BOOK_KEY_TO_STRING_CASE(LSystem)
        BOOK_KEY_TO_STRING_CASE(RControl)
        BOOK_KEY_TO_STRING_CASE(RShift)
        BOOK_KEY_TO_STRING_CASE(RAlt)
        BOOK_KEY_TO_STRING_CASE(RSystem)
        BOOK_KEY_TO_STRING_CASE(Menu)
        BOOK_KEY_TO_STRING_CASE(LBracket)
        BOOK_KEY_TO_STRING_CASE(RBracket)
        BOOK_KEY_TO_STRING_CASE(SemiColon)
        BOOK_KEY_TO_STRING_CASE(Comma)
        BOOK_KEY_TO_STRING_CASE(Period)
        BOOK_KEY_TO_STRING_CASE(Quote)
        BOOK_KEY_TO_STRING_CASE(Slash)
        BOOK_KEY_TO_STRING_CASE(BackSlash)
        BOOK_KEY_TO_STRING_CASE(Tilde)
        BOOK_KEY_TO_STRING_CASE(Equal)
        BOOK_KEY_TO_STRING_CASE(Dash)
        BOOK_KEY_TO_STRING_CASE(Space)
        BOOK_KEY_TO_STRING_CASE(Return)
        BOOK_KEY_TO_STRING_CASE(BackSpace)
        BOOK_KEY_TO_STRING_CASE(Tab)
        BOOK_KEY_TO_STRING_CASE(PageUp)
        BOOK_KEY_TO_STRING_CASE(PageDown)
        BOOK_KEY_TO_STRING_CASE(End)
        BOOK_KEY_TO_STRING_CASE(Home)
        BOOK_KEY_TO_STRING_CASE(Insert)
        BOOK_KEY_TO_STRING_CASE(Delete)
        BOOK_KEY_TO_STRING_CASE(Add)
        BOOK_KEY_TO_STRING_CASE(Subtract)
        BOOK_KEY_TO_STRING_CASE(Multiply)
        BOOK_KEY_TO_STRING_CASE(Divide)
        BOOK_KEY_TO_STRING_CASE(Left)
        BOOK_KEY_TO_STRING_CASE(Right)
        BOOK_KEY_TO_STRING_CASE(Up)
        BOOK_KEY_TO_STRING_CASE(Down)
        BOOK_KEY_TO_STRING_CASE(Numpad0)
        BOOK_KEY_TO_STRING_CASE(Numpad1)
        BOOK_KEY_TO_STRING_CASE(Numpad2)
        BOOK_KEY_TO_STRING_CASE(Numpad3)
        BOOK_KEY_TO_STRING_CASE(Numpad4)
        BOOK_KEY_TO_STRING_CASE(Numpad5)
        BOOK_KEY_TO_STRING_CASE(Numpad6)
        BOOK_KEY_TO_STRING_CASE(Numpad7)
        BOOK_KEY_TO_STRING_CASE(Numpad8)
        BOOK_KEY_TO_STRING_CASE(Numpad9)
        BOOK_KEY_TO_STRING_CASE(F1)
        BOOK_KEY_TO_STRING_CASE(F2)
        BOOK_KEY_TO_STRING_CASE(F3)
        BOOK_KEY_TO_STRING_CASE(F4)
        BOOK_KEY_TO_STRING_CASE(F5)
        BOOK_KEY_TO_STRING_CASE(F6)
        BOOK_KEY_TO_STRING_CASE(F7)
        BOOK_KEY_TO_STRING_CASE(F8)
        BOOK_KEY_TO_STRING_CASE(F9)
        BOOK_KEY_TO_STRING_CASE(F10)
        BOOK_KEY_TO_STRING_CASE(F11)
        BOOK_KEY_TO_STRING_CASE(F12)
        BOOK_KEY_TO_STRING_CASE(F13)
        BOOK_KEY_TO_STRING_CASE(F14)
        BOOK_KEY_TO_STRING_CASE(F15)
        BOOK_KEY_TO_STRING_CASE(Pause)
        case sf::Keyboard::KeyCount:
            break;
    }
    return "";
}

/**
 * Function that sets a sprite's origin in its center
 * @param sprite The sprite to center
 */
void Utility::center_origin( sf::Sprite &sprite )
{
    sf::FloatRect bounds = sprite.getLocalBounds();
    sprite.setOrigin(
            std::floor( bounds.left + bounds.width / 2.f ),
            std::floor( bounds.top + bounds.height / 2.f )
            );
}

/**
 * Function that sets a text's origin in its center
 * @param text The text to center
 */
void Utility::center_origin( sf::Text &text )
{
    sf::FloatRect bounds = text.getLocalBounds();
    text.setOrigin(
            std::floor( bounds.left + bounds.width / 2.f ),
            std::floor( bounds.top + bounds.height / 2.f )
            );
}

/**
 * Function that sets the animation's origin in its center
 * @param animation The animation to center
 */
void Utility::center_origin( Animation &animation )
{
    sf::FloatRect bounds = animation.get_local_bounds();
    animation.setOrigin(
            std::floor( bounds.left + bounds.width / 2.f ),
            std::floor( bounds.top + bounds.height / 2.f )
    );
}

/**
 * Function to convert from Radian to Degrees
 * @param radian The radian value to convert
 * @return The radian value in degrees form
 */
float Utility::to_degree( float radian )
{
    return 100.f / M_PI * radian;
}

/**
 * Function to convert from Degree to Radian
 * @param degree The degree value to convert
 * @return The degree value in radian form
 */
float Utility::to_radian( float degree )
{
    return M_PI / 100.f * degree;
}

/**
 * Random integer generator
 * @param exclusive_max The maximum value we wish to obtain
 * @return A value from 0 to Max
 */
int Utility::random_int( int exclusive_max )
{
    std::uniform_int_distribution<> dist( 0, exclusive_max - 1 );
    return dist( Random_Engine );
}

/**
 * Function to calculate the distance between within a vector
 * @param vector The vecto's to calculate
 * @return X * X + Y * Y
 */
float Utility::length( sf::Vector2f vector )
{
    return std::sqrt( vector.x * vector.x + vector.y * vector.y );
}

/**
 * Unit calculator based on the length of a vector and a vector itself
 */
sf::Vector2f Utility::unit_vector( sf::Vector2f vector )
{
    assert( vector != sf::Vector2f( 0.f, 0.f ) );
    return vector / length( vector );
}
```

9.6.18 World.hpp

```
#ifndef SOKOBAN_WORLD_HPP
#define SOKOBAN_WORLD_HPP

#include "../Resource_Holder.hpp"
#include "Resource_Identifiers.hpp"
#include "Scene_Node.hpp"
#include "Sprite_Node.hpp"
#include "entities/Entity.hpp"
#include "entities/Entity_Box.hpp"
#include "entities/Entity_Wall.hpp"
#include "entities/Entity_Player.hpp"
#include "entities/Entity_Platform.hpp"
#include "Sound_Player.hpp"
#include "../../model/Board.hpp"

#include <SFML/System/NonCopyable.hpp>
#include <SFML/Graphics/View.hpp>
#include <SFML/Graphics/Texture.hpp>
#include <SFML/Graphics/Text.hpp>

#include <queue>
```

```
#include <vector>

namespace sf
{
    class RenderTarget;
}

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class World
        : private sf::NonCopyable
            {
            public:
                World( sf::RenderTarget &target, const model::Board &board, Font_Holder &fonts, Sound_Player
     &sounds );
                ~World();
                void update( sf::Time dt );
                void draw();
                bool is_board_completed() const;
                void move_up( bool pressed );
                void move_down( bool pressed );
                void move_left( bool pressed );
                void move_right( bool pressed );
                void set_reset_counter( int reset_counter );
                int get_reset_counter() const;
            private:
                /**
                 * The various background colors
                 */
                enum class Background_Color
                {
                    CONCRETE,
                    DIRT,
                    GRASS,
                    SAND
                };
                sf::RenderTarget &_target; /** The window that will display the various components */
                sf::View _world_view; /** The globally defined world view */
                Texture_Holder _textures; /** Default texture holder */
                Font_Holder &_fonts; /** Default font holder */
                Sound_Player &_sounds; /** Default Sound Player */
                Scene_Node _scene_graph; /** Main scene node */
                std::vector< Scene_Node * > _scene_layers; /** Various Scene layers */
                bool _player_is_moving_up; /** Whether the player is moving up */
                bool _player_is_moving_down; /** Whether the player is moving down */
                bool _player_is_moving_left; /** Whether the player is moving left */
                bool _player_is_moving_right; /** Whether the player is moving right */
                sf::FloatRect _world_bounds; /** The accessible limits of the window */
                model::Board _board; /** The Board containing all the actors */
                Sprite_Node *_player_sprite; /** The player sprite */
                model::Player *_board_player; /** The player within the board */
                entity::Entity_Player *_player_entity; /** The player entity containing the various assets
    coordinates */
                std::vector< Sprite_Node * > _box_sprites; /** The various box sprites */
                std::vector< model::Box * > _box_actors; /** The various box actors within the board */
                std::vector< entity::Entity_Box * > _box_entities; /** The various box entities containing
    the assets' coordinates */
                std::vector< entity::Entity * > _entities; /** The entirety of the entities present in the
    world */
                sf::Texture *_box_texture_sheet; /** The box texture */
                sf::Texture *_platform_texture_sheet; /** The platform texture */
                sf::Texture *_wall_texture_sheet; /** The wall texture */
                sf::Texture *_player_texture_sheet; /** The player texture */
                sf::Texture *_background_texture; /** The background texture */
                sf::Text *_text; /** The visually visible text */
                int _reset_counter; /** The amount of resets */
                void load_textures();
                void build_scene();
            };
        }
    }
}

#endif //SOKOBAN_WORLD_HPP
```

9.6.19 World.cpp

```
#include "World.hpp"

```

```cpp
#include "../../util/Logger.hpp"
#include "Sound_Node.hpp"

#include <SFML/Graphics/RenderTarget.hpp>
#include <SFML/Graphics/Text.hpp>

#include <cmath>
#include <ctime>
#include <random>
#include <utility>
#include <sstream>
#include <iostream>

using namespace sokoban::ui::gui;
using namespace sokoban::util;

namespace
{
    int steps_counter = 0;
}

/**
 * Default constructor for the world
 * @param target The target to display the sprites onto
 * @param board The board containing the various actors and the level's skeleton
 * @param fonts The various fonts used
 * @param sounds The various sound effects
 */
World::World( sf::RenderTarget &target, const model::Board &board, Font_Holder &fonts, Sound_Player &sounds
    )
: _target( target )
  , _world_view( target.getDefaultView() )
  , _textures()
  , _fonts( fonts )
  , _sounds( sounds )
  , _scene_graph()
  , _scene_layers()
  , _world_bounds( 0.f, 0.f, _world_view.getSize().x, _world_view.getSize().y )
  , _player_is_moving_up( false )
  , _player_is_moving_down( false )
  , _player_is_moving_left( false )
  , _player_is_moving_right( false )
  , _board_player( nullptr )
  , _player_entity( nullptr )
  , _player_sprite( nullptr )
  , _box_texture_sheet( nullptr )
  , _platform_texture_sheet( nullptr )
  , _wall_texture_sheet( nullptr )
  , _player_texture_sheet( nullptr )
  , _background_texture( nullptr )
  , _board( "" )
  , _text( nullptr )
  , _reset_counter( 0 )
{
    _board = board;
    Logger::log( LoggerLevel::DEBUG, "Level Layout:" );
    for( model::Actor *actor : _board.get_world() )
    {
        Logger::log( LoggerLevel::DEBUG, actor->to_string() );
    }
    /*step_buffer.loadFromFile( "assets/sounds/footsteps_outdoor_boots.ogg" );
    step_sound.setBuffer( step_buffer );
    step_sound.setVolume( 10.f );
    box_move_buffer.loadFromFile( "assets/sounds/wood_creak_01.ogg" );
    box_move_sound.setBuffer( box_move_buffer );
    box_move_sound.setVolume( 10.f );*/
    load_textures();
    build_scene();
    steps_counter = 0;
}

/**
 * Default destructor for the World
 */
World::~World()
{
    delete _text;
    for ( Scene_Node *layer: _scene_layers )
    {
        delete layer;
    }
    for( entity::Entity *entity : _entities )
    {
        delete entity;
    }
    _scene_layers.clear();
    _entities.clear();
    delete _player_texture_sheet;
    delete _box_texture_sheet;
    delete _platform_texture_sheet;
    delete _wall_texture_sheet;
    delete _background_texture;
    _box_sprites.clear();
    _box_entities.clear();
}

/**
 * Realtime updates the visually available entities
 * @param dt The clock time
 */
void World::update( sf::Time dt )
{
    float SPACE = 64.f;
    sf::IntRect player_assets_coords;
    float player_x_coords;
    float player_y_coords;
    float player_width_coords;
    float player_height_coords;
    if ( _player_is_moving_up )
    {
        auto player_asset_rect = _player_entity->get_player_face_map().find( entity::Entity_Player::Face::NORTH )->second;
        player_x_coords = player_asset_rect.at( 0 );
        player_y_coords = player_asset_rect.at( 1 );
        player_width_coords = player_asset_rect.at( 2 );
        player_height_coords = player_asset_rect.at( 3 );
        player_assets_coords = sf::IntRect( player_x_coords, player_y_coords, player_width_coords, player_height_coords );
        _player_sprite->set_texture( *_player_texture_sheet );
        _player_sprite->set_texture( *_player_texture_sheet, player_assets_coords );
        if ( _board.check_wall_collision( _board_player, _board.TOP_COLLISION ) )
        {
            return;
        }
        if ( _board.check_box_collision( _board.TOP_COLLISION ) )
        {
            return;
        }
        else
        {
            for ( int i = 0; i < _box_entities.size(); i++ )
            {
                _box_sprites.at( i )->setPosition( _box_actors.at( i )->get_x(), _box_actors.at( i )->get_y() );
            }
        }
        _player_sprite->move( 0.f, -SPACE );
        _player_entity->set_y( _player_entity->get_y() - SPACE );
        _board_player->set_y( _board_player->get_y() - SPACE );
    }
    if ( _player_is_moving_down )
    {
        auto player_asset_rect = _player_entity->get_player_face_map().find( entity::Entity_Player::Face::SOUTH )->second;
        player_x_coords = player_asset_rect.at( 0 );
        player_y_coords = player_asset_rect.at( 1 );
        player_width_coords = player_asset_rect.at( 2 );
        player_height_coords = player_asset_rect.at( 3 );
        player_assets_coords = sf::IntRect( player_x_coords, player_y_coords, player_width_coords, player_height_coords );
        _player_sprite->set_texture( *_player_texture_sheet, player_assets_coords );
        if ( _board.check_wall_collision( _board_player, _board.BOTTOM_COLLISION ) )
        {
            return;
        }
        if ( _board.check_box_collision( _board.BOTTOM_COLLISION ) )
        {
            return;
        }
        else
        {
            for ( int i = 0; i < _box_entities.size(); i++ )
            {
                _box_sprites.at( i )->setPosition( _box_actors.at( i )->get_x(), _box_actors.at( i )->get_y() );
            }
        }
        _player_sprite->move( 0.f, +SPACE );
```

```cpp
        _player_entity->set_y( _player_entity->get_y() + SPACE );
        _board_player->set_y( _board_player->get_y() + SPACE );
    }
    if ( _player_is_moving_left )
    {
        auto player_asset_rect = _player_entity->get_player_face_map().find( entity::Entity_Player::Face::WEST )->second;
        player_x_coords = player_asset_rect.at( 0 );
        player_y_coords = player_asset_rect.at( 1 );
        player_width_coords = player_asset_rect.at( 2 );
        player_height_coords = player_asset_rect.at( 3 );
        player_assets_coords = sf::IntRect( player_x_coords, player_y_coords, player_width_coords, player_height_coords );
        _player_sprite->set_texture( *_player_texture_sheet, player_assets_coords );
        if ( _board.check_wall_collision( _board_player, _board.LEFT_COLLISION ) )
        {
            return;
        }
        if ( _board.check_box_collision( _board.LEFT_COLLISION ) )
        {
            return;
        }
        else
        {
            for ( int i = 0; i < _box_entities.size(); i++ )
            {
                _box_sprites.at( i )->setPosition( _box_actors.at( i )->get_x(), _box_actors.at( i )->get_y() );
            }
        }
        _player_sprite->move( -SPACE, 0.f );
        _player_entity->set_x( _player_entity->get_x() - SPACE );
        _board_player->set_x( _board_player->get_x() - SPACE );
    }
    if ( _player_is_moving_right )
    {
        auto player_asset_rect = _player_entity->get_player_face_map().find( entity::Entity_Player::Face::EAST )->second;
        player_x_coords = player_asset_rect.at( 0 );
        player_y_coords = player_asset_rect.at( 1 );
        player_width_coords = player_asset_rect.at( 2 );
        player_height_coords = player_asset_rect.at( 3 );
        player_assets_coords = sf::IntRect( player_x_coords, player_y_coords, player_width_coords, player_height_coords );
        _player_sprite->set_texture( *_player_texture_sheet, player_assets_coords );
        if ( _board.check_wall_collision( _board_player, _board.RIGHT_COLLISION ) )
        {
            return;
        }
        if ( _board.check_box_collision( _board.RIGHT_COLLISION ) )
        {
            return;
        }
        else
        {
            for ( int i = 0; i < _box_entities.size(); i++ )
            {
                _box_sprites.at( i )->setPosition( _box_actors.at( i )->get_x(), _box_actors.at( i )->get_y() );
            }
        }
        _player_sprite->move( +SPACE, 0.f );
        _board_player->set_x( _board_player->get_x() + SPACE );
    }
    _text->setString(
            "Steps:  " + std::to_string( steps_counter ) + "\n" +
            "Resets:  " + std::to_string( get_reset_counter() ) + "\n" +
            "Move Up: Arrow Up" + "\n" +
            "Move Down: Arrow Down" + "\n" +
            "Move Left: Arrow Left" + "\n" +
            "Move Right: Arrow Right" + "\n" +
            "Reset Board: R" + "\n" +
            "Skip Level: S" + "\n" +
            "Undo Skip: X" + "\n" +
            "Pause Game: Escape" + "\n"
            );
    if ( _board.is_completed() )
    {
        //step_sound.stop();
        /* TODO: BLINKING TEXT WHEN FINISHED */
    }

}

```

```cpp
/**
 * Visually display the various scene nodes that make up the world
 */
void World::draw()
{
    _target.setView( _world_view );
    for ( Scene_Node *layer: _scene_layers )
    {
        _target.draw( *layer );
    }
    _target.draw( *_text );
}

/**
 * Load the various sprite sheets
 */
void World::load_textures()
{
    Logger::log( LoggerLevel::INFO, "Loading Textures..." );
    _player_texture_sheet = new sf::Texture();
    _player_texture_sheet->loadFromFile( "assets/images/Spritesheet/character_spritesheet.png" );
    _box_texture_sheet = new sf::Texture();
    _box_texture_sheet->loadFromFile( "assets/images/Spritesheet/boxes_spritesheet.png" );
    _platform_texture_sheet = new sf::Texture();
    _platform_texture_sheet->loadFromFile( "assets/images/Spritesheet/platforms_spritesheet.png" );
    _wall_texture_sheet = new sf::Texture();
    _wall_texture_sheet->loadFromFile( "assets/images/Spritesheet/wall_round_spritesheet.png" );
    _background_texture = new sf::Texture();
}

/**
 * Build the scene based on the board's skeleton
 */
void World::build_scene()
{
    Logger::log( LoggerLevel::INFO, "World Node init" );
    _player_is_moving_up = false;
    _player_is_moving_down = false;
    _player_is_moving_left = false;
    _player_is_moving_right = false;
    _text = new sf::Text();
    _text->setFont( _fonts.get( Fonts::Connection_II ) );
    _text->setPosition(
            _target.getSize().x - 350.f,
            20.f );
    _text->setCharacterSize( 24 );
    _text->setFillColor( sf::Color::Black );

    Logger::log( LoggerLevel::INFO, "Board size: " + std::to_string( _board.get_world().size() ) );
    _box_sprites = std::vector< Sprite_Node * >();
    _box_actors = std::vector< model::Box * >();
    _box_entities = std::vector< entity::Entity_Box * >();

    Logger::log( LoggerLevel::INFO, "Building Scene..." );
    std::size_t world_size = _board.get_world().size() + 1;
    _scene_layers = std::vector< Scene_Node * >();
    std::stringstream ss;
    ss << "Initializing " << std::to_string( world_size ) << " Scene nodes...";
    Logger::log( LoggerLevel::DEBUG, ss.str() );

    std::random_device rd;
    std::mt19937 mt( rd() );

    int min_within_enum;
    int max_within_enum;

    min_within_enum = static_cast< int >( Background_Color::CONCRETE );
    max_within_enum = static_cast< int >( Background_Color::SAND );
    std::uniform_int_distribution< int > background_distribution( min_within_enum, max_within_enum );
    auto random_background_color = static_cast< Background_Color >( background_distribution( mt ) );
    switch ( random_background_color )
    {
        case Background_Color::CONCRETE:
            _background_texture->loadFromFile( "assets/images/PNG/GroundGravel_Concrete.png" );
            break;
        case Background_Color::DIRT:
            _background_texture->loadFromFile( "assets/images/PNG/GroundGravel_Dirt.png" );
            break;
        case Background_Color::GRASS:
            _background_texture->loadFromFile( "assets/images/PNG/GroundGravel_Grass.png" );
            break;
        case Background_Color::SAND:
            _background_texture->loadFromFile( "assets/images/PNG/GroundGravel_Sand.png" );
            break;
    }

    min_within_enum = static_cast< int >( entity::Entity_Box::Color::BEIGE_LIGHT );
    max_within_enum = static_cast< int >( entity::Entity_Box::Color::YELLOW_LIGHT );
    std::uniform_int_distribution< int > box_distribution( min_within_enum, max_within_enum );
    auto random_box_color = static_cast< entity::Entity_Box::Color >( box_distribution( mt ) );

    min_within_enum = static_cast< int >( entity::Entity_Platform::Color::BEIGE );
    max_within_enum = static_cast< int >( entity::Entity_Platform::Color::YELLOW );
    std::uniform_int_distribution< int > platform_distribution( min_within_enum, max_within_enum );
    auto random_platform_color = static_cast< entity::Entity_Platform::Color >( platform_distribution( mt ) );

    min_within_enum = static_cast< int >( entity::Entity_Wall::Color::BEIGE );
    max_within_enum = static_cast< int >( entity::Entity_Wall::Color::BROWN );
    std::uniform_int_distribution< int > wall_distribution( min_within_enum, max_within_enum );
    auto random_wall_color = static_cast< entity::Entity_Wall::Color >( wall_distribution( mt ) );

    int layers = 0;
    sf::IntRect textureRect( _world_bounds );
    _background_texture->setRepeated( true );

    auto *backgroundSprite = new Sprite_Node( *_background_texture, textureRect );
    backgroundSprite->setPosition( _world_bounds.left, _world_bounds.top );
    _scene_layers.push_back( backgroundSprite );

    layers++;

    for ( model::Actor *actor: _board.get_world() )
    {
        float asset_coord_x;
        float asset_coord_y;
        float asset_coord_width;
        float asset_coord_height;

        sf::IntRect asset_rect;

        Sprite_Node *actor_sprite = nullptr;

        entity::Entity *entity_actor = nullptr;

        if ( actor->get_type() == actor->PLAYER )
        {
            _board_player = dynamic_cast< model::Player * >( actor );
            entity_actor = new entity::Entity_Player( actor->get_x(), actor->get_y() );
            _player_entity = dynamic_cast< entity::Entity_Player * >( entity_actor );
            auto player_asset_rect = _player_entity->get_player_face_map().find( entity::Entity_Player::Face::SOUTH )->second;
            asset_coord_x = player_asset_rect.at( 0 );
            asset_coord_y = player_asset_rect.at( 1 );
            asset_coord_width = player_asset_rect.at( 2 );
            asset_coord_height = player_asset_rect.at( 3 );
            asset_rect = sf::IntRect( asset_coord_x, asset_coord_y, asset_coord_width, asset_coord_height );
            actor_sprite = new Sprite_Node( *_player_texture_sheet, asset_rect );
            _player_sprite = actor_sprite;
        }
        if ( actor->get_type() == actor->PLATFORM )
        {
            entity_actor = new entity::Entity_Platform( actor->get_x(), actor->get_y() );
            actor = dynamic_cast< entity::Entity_Platform * >( entity_actor );
            auto *platform_actor = dynamic_cast< entity::Entity_Platform * >( actor );
            auto platform_asset_rect = platform_actor->get_platform_color_map().find( random_platform_color )->second;
            asset_coord_x = platform_asset_rect.at( 0 );
            asset_coord_y = platform_asset_rect.at( 1 );
            asset_coord_width = platform_asset_rect.at( 2 );
            asset_coord_height = platform_asset_rect.at( 3 );
            asset_rect = sf::IntRect( asset_coord_x, asset_coord_y, asset_coord_width, asset_coord_height );
            actor_sprite = new Sprite_Node( *_platform_texture_sheet, asset_rect );
            /* If sprite size = 32x32 */
            actor_sprite->setOrigin(
                    -( asset_coord_width / 2.f ),
                    -( asset_coord_height / 2.f )
            );
        }
        if ( actor->get_type() == actor->BOX )
        {
            _box_actors.push_back( dynamic_cast< model::Box * >( actor ) );
            entity_actor = new entity::Entity_Box( actor->get_x(), actor->get_y() );
            actor = dynamic_cast< entity::Entity_Box * >( entity_actor );
            auto *box_entity = dynamic_cast< entity::Entity_Box * >( actor );
            auto box_asset_rect = box_entity->get_box_color_map().find( random_box_color )->second;
            asset_coord_x = box_asset_rect.at( 0 );
            asset_coord_y = box_asset_rect.at( 1 );
            asset_coord_width = box_asset_rect.at( 2 );
            asset_coord_height = box_asset_rect.at( 3 );
            sf::IntRect asset_rect( asset_coord_x, asset_coord_y, asset_coord_width, asset_coord_height );
            actor_sprite = new Sprite_Node( *_box_texture_sheet, asset_rect );
            _box_sprites.push_back( actor_sprite );
            _box_entities.push_back( box_entity );
        }
        if ( actor->get_type() == actor->WALL )
        {
            entity_actor = new entity::Entity_Wall( actor->get_x(), actor->get_y() );
            actor = dynamic_cast< entity::Entity_Wall * >( entity_actor );
            auto *wall_actor = dynamic_cast< entity::Entity_Wall * >( actor );
            auto wall_asset_rect = wall_actor->get_wall_color_map().find( random_wall_color )->second;
            asset_coord_x = wall_asset_rect.at( 0 );
            asset_coord_y = wall_asset_rect.at( 1 );
            asset_coord_width = wall_asset_rect.at( 2 );
            asset_coord_height = wall_asset_rect.at( 3 );
            asset_rect = sf::IntRect( asset_coord_x, asset_coord_y, asset_coord_width, asset_coord_height );
            actor_sprite = new Sprite_Node( *_wall_texture_sheet, asset_rect );
        }
        actor_sprite->setPosition( actor->get_x(), actor->get_y() );
        _scene_layers.push_back( actor_sprite );
        _entities.push_back( entity_actor );
        layers++;
    }

    Logger::log( LoggerLevel::INFO, "Number of layers loaded: " + std::to_string( layers ) );
}

/**
 * Check whether the board is completed
 * @return whether the board is completed
 */
bool World::is_board_completed() const
{
    return _board.is_completed();
}

/**
 * Move the player up
 * @param pressed Whether the up key is pressed
 */
void World::move_up( bool pressed )
{
    if( pressed )
    {
        steps_counter++;
    }
    _player_is_moving_up = pressed;
}

/**
 * Move the player down
 * @param pressed Whether the down key is pressed
 */
void World::move_down( bool pressed )
{
    if( pressed )
    {
        steps_counter++;
    }
    _player_is_moving_down = pressed;
}

/**
 * Move the player left
 * @param pressed Whether the left key is pressed
 */
void World::move_left( bool pressed )
{
    if( pressed )
    {
        steps_counter++;
    }
    _player_is_moving_left = pressed;
}

/**
 * Move the player right
 * @param pressed Whether the right key is pressed
 */
void World::move_right( bool pressed )
{
    if( pressed )
    {
        steps_counter++;
    }
    _player_is_moving_right = pressed;
}

/**
 * Setter for the number of resets done throughout the match
 * @param reset_counter The current number of resets
 */
void World::set_reset_counter( int reset_counter )
{
    _reset_counter = reset_counter;
}

/**
 * Getter for the number of resets called
 * @return The number of resets
 */
int World::get_reset_counter() const
{
    return _reset_counter;
}
```

9.7 Components

9.7.1 Component.hpp

```
#ifndef SOKOBAN_COMPONENT_HPP
#define SOKOBAN_COMPONENT_HPP

#include <SFML/System/NonCopyable.hpp>
#include <SFML/Graphics/Drawable.hpp>
#include <SFML/Graphics/Transformable.hpp>

#include <memory>

namespace sf
{
    class Event;
}

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class Component
            : public sf::Drawable
            , public sf::Transformable
            , private sf::NonCopyable
            {
            public:
                /** Generic declaration of a shared pointer meant to define a base nature to said pointer */
                typedef std::shared_ptr< Component > Ptr;
                Component();
                virtual ~Component();
                virtual bool is_selectable() const = 0;
                bool is_selected() const;
                virtual void select();
                virtual void deselect();
                virtual bool is_active() const;
                virtual void activate();
                virtual void deactivate();
                virtual void handle_event( const sf::Event &event ) = 0;
            private:
                bool _is_selected; /** Check whether the component is selected */
                bool _is_active; /** Check whether the component is active */
            };
        }
    }
}

#endif //SOKOBAN_COMPONENT_HPP
```

9.7.2 Component.cpp

```
#include "Component.hpp"

using namespace sokoban::ui::gui;

```

```cpp
/**
 * Component's initializer
 * By default, it is neither selected nor active
 */
Component::Component()
: _is_selected( false )
, _is_active( false )
{
}

/**
 * Destructor with no specific action
 */
Component::~Component()
= default;

/**
 * Getter meant to retrieve whether a component is selected or not
 */
bool Component::is_selected() const
{
    return _is_selected;
}

/**
 * Setter meant to define a component as selected
 */
void Component::select()
{
    _is_selected = true;
}

/**
 * Setter meant to define a component as deselected
 */
void Component::deselect()
{
    _is_selected = false;
}

/**
 * Getter meant to retrieve whether a component is active or not
 */
bool Component::is_active() const
{
    return _is_active;
}

/**
 * Setter meant to define a component as active
 */
void Component::activate()
{
    _is_active = true;
}

/**
 * Setter meant to define a component as non active
 */
void Component::deactivate()
{
    _is_active = false;
}
```

9.7.3 Container.hpp

```cpp
#ifndef SOKOBAN_CONTAINER_HPP
#define SOKOBAN_CONTAINER_HPP

#include "Component.hpp"

#include <memory>
#include <vector>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class Container
                    : public Component
            {
```

```cpp
            public:
                /** Shared pointer meant to characterize the component as a Container */
                typedef std::shared_ptr< Container > Ptr;
                Container();
                void pack( Component::Ptr component );
                bool is_selectable() const override;
                void handle_event( const sf::Event &event ) override;
            private:
                std::vector< Component::Ptr > _children; /** The various components it should hold */
                int _selected_child; /** The selected child */
                void draw( sf::RenderTarget &target, sf::RenderStates states ) const override;
                bool has_selection() const;
                void select( std::size_t index );
                void select_next();
                void select_previous();
            };
        }
    }
}

#endif //SOKOBAN_CONTAINER_HPP
```

9.7.4 Container.cpp

```cpp
#include "Container.hpp"

#include <SFML/Window/Event.hpp>
#include <SFML/Graphics/RenderStates.hpp>
#include <SFML/Graphics/RenderTarget.hpp>

using namespace sokoban::ui::gui;

/**
 * Constructor for the container
 * No children by default and no selected child by default
 */
Container::Container()
: _children()
, _selected_child( -1 )
{
}

/**
 * Function meant to add a component to the children.
 * @param component The new child to add
 */
void Container::pack( Component::Ptr component )
{
    _children.push_back( component );
    if( !has_selection() && component->is_selectable() )
    {
        select( _children.size() -1 );
    }
}

/**
 * By default a container is not selectable
 */
bool Container::is_selectable() const
{
    return false;
}

/**
 * Function meant to interchange between the various children
 */
void Container::handle_event( const sf::Event &event )
{
    if( has_selection() && _children[ _selected_child ]->is_active() )
    {
        _children[ _selected_child ]->handle_event( event );
    }
    else if( event.type == sf::Event::KeyReleased )
    {
        if( event.key.code == sf::Keyboard:: Up )
        {
            select_previous();
        }
        else if( event.key.code == sf::Keyboard::Down )
        {
            select_next();
        }
        else if( event.key.code == sf::Keyboard::Return || event.key.code == sf::Keyboard::Space )
        {
            if( has_selection() )
            {
                _children[ _selected_child ]->activate();
            }
        }
    }
}

/**
 * Visually display each child of the container
 */
void Container::draw( sf::RenderTarget &target, sf::RenderStates states ) const
{
    states.transform *= getTransform();
    for( const Component::Ptr &child : _children )
    {
        target.draw( *child, states );
    }
}

/**
 * Function meant to retrieve whether a container has a selected child
 */
bool Container::has_selection() const
{
    return _selected_child >= 0;
}

/**
 * Function meant to interchange between the different children
 * @param index The child that has been selected
 */
void Container::select( std::size_t index )
{
    if( _children[ index ]->is_selectable() )
    {
        if( has_selection() )
        {
            _children[ _selected_child ]->deselect();
        }
        _children[ index ]->select();
        _selected_child = index;
    }
}

/**
 * Function meant to select the following child within the list of children
 */
void Container::select_next()
{
    if( !has_selection() )
    {
        return;
    }
    int next = _selected_child;
    do
    {
        next = ( next + 1 ) % _children.size();
    }
    while ( !_children[ next ]->is_selectable() );
    select( next );
}

/**
 * Function meant to select the previous child within the list of children
 */
void Container::select_previous()
{
    if( !has_selection() )
    {
        return;
    }
    int prev = _selected_child;
    do
    {
        prev = ( prev + _children.size() - 1 ) % _children.size();
    }
    while ( !_children[ prev ]->is_selectable() );
    select( prev );
}
```

9.7.5 Button.hpp

```cpp
#ifndef SOKOBAN_BUTTON_HPP
#define SOKOBAN_BUTTON_HPP

#include "Component.hpp"
#include "../Resource_Identifiers.hpp"
#include "../states/State.hpp"

#include <SFML/Graphics/Sprite.hpp>
#include <SFML/Graphics/Text.hpp>

#include <memory>
#include <string>
#include <vector>
#include <functional>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class Sound_Player;
            class Button
                    : public Component
            {
            public:
                typedef std::shared_ptr< Button > Ptr; /** The shared pointer that will define the Component
     's nature */
                typedef std::function< void() > Callback; /** The action the component will execute */
                enum Type
                {
                    Normal, /** Non-selected state of a button */
                    Selected, /** Selected state of a button */
                    Pressed, /** Pressed state of a button */
                    Button_Count /** Counter for various button states */
                };
                explicit Button( State::Context context );
                void set_callback( Callback callback );
                void set_text( const std::string &text );
                void set_toggle( bool flag );
                bool is_selectable() const override;
                void select() override;
                void deselect() override;
                void activate() override;
                void deactivate() override;
                void handle_event( const sf::Event &event ) override;
            private:
                Callback _callback; /** The type of execution the button will cause */
                sf::Sprite _sprite; /** The asset meant to represent the sprite */
                sf::Text _text; /** The text written on the button */
                bool _is_toggled; /** Whether the button is toggled or not */
                Sound_Player &_sounds; /** The sound it will make when selected or executed */
                void draw( sf::RenderTarget &target, sf::RenderStates states ) const override;
                void change_texture( Type button_type );
            };
        }
    }
}

#endif //SOKOBAN_BUTTON_HPP
```

9.7.6 Button.cpp

```cpp
#include "Button.hpp"

#include "../Utility.hpp"
#include "../Sound_Player.hpp"
#include "../../Resource_Holder.hpp"

#include <SFML/Window/Event.hpp>
#include <SFML/Graphics/RenderStates.hpp>
#include <SFML/Graphics/RenderTarget.hpp>

using namespace sokoban::ui::gui;

/**
 * Initializer for the button.
 * Set the default states that characterize the button
 */
Button::Button( State::Context context )
: _callback()
```

```cpp
, _sprite( context._textures->get( Textures::Button ) )
, _text( "", context._fonts->get( Fonts::Rampart_One ) )
, _is_toggled( false )
, _sounds( *context._sounds )
{
    change_texture( Normal );

    sf::FloatRect bounds = _sprite.getLocalBounds();
    _text.setPosition( bounds.width / 2.f, bounds.height / 2.f );
}

/**
 * Setter for the type of action a button will execute
 * @param callback The execution
 */
void Button::set_callback( Button::Callback callback )
{
    _callback = std::move( callback );
}

/**
 * Setter for the button's text
 */
void Button::set_text( const std::string &text )
{
    _text.setString( text );
    Utility::center_origin( _text );
}

/**
 * Setter meant to make the button toggleable
 * @param flag Whether it is toggleable or not
 */
void Button::set_toggle( bool flag )
{
    _is_toggled = flag;
}

/**
 * Button meant to determine whether a button is selectable or not.
 */
bool Button::is_selectable() const
{
    return true;
}

/**
 * Function meant to change the texture of a button upon selection
 */
void Button::select()
{
    Component::select();

    change_texture( Selected );
    _sounds.play( Sound_Effect::Button_Click_02 );
}

/**
 * Function meant to deselect a component and change its texture
 */
void Button::deselect()
{
    Component::deselect();

    change_texture( Normal );
}

/**
 * Function meant to make a button active
 */
void Button::activate()
{
    Component::activate();
    if( _is_toggled )
    {
        change_texture( Pressed );
    }
    if( _callback )
    {
        _callback();
    }
    if( !_is_toggled )
    {
        deactivate();
    }
    _sounds.play( Sound_Effect::Button_Click_01 );
}

/**
 * Function meant to deactivate an already active button
 */
void Button::deactivate()
{
    Component::deactivate();

    if( _is_toggled )
    {
        if( is_selected() )
        {
            change_texture( Selected );
        }
        else
        {
            change_texture( Normal );
        }
    }
}

/*
 * Function meant to execute an event
 */
void Button::handle_event( const sf::Event &event )
{
}

/**
 * Function meant to draw the display the button graphically
 * @param target The target meant to be displayed upon
 * @param states The various preconfigured states that characterise the target
 */
void Button::draw( sf::RenderTarget &target, sf::RenderStates states ) const
{
    states.transform *= getTransform();
    target.draw( _sprite, states );
    target.draw( _text, states );
}

/**
 * Function meant to change the texture of the button based on its current state
 */
void Button::change_texture( Button::Type button_type )
{
    sf::IntRect texture_rect( 0, 50 * button_type, 200, 50 );
    _sprite.setTextureRect( texture_rect );
}
```

9.7.7 Label.hpp

```cpp
#ifndef SOKOBAN_LABEL_HPP
#define SOKOBAN_LABEL_HPP

#include "Component.hpp"
#include "../Resource_Identifiers.hpp"
#include "../../Resource_Holder.hpp"

#include <SFML/Graphics/Text.hpp>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class Label
                    : public Component
            {
            public:
                /** Shared pointer meant to define the component as a Label */
                typedef std::shared_ptr< Label > Ptr;
                Label( const std::string &text, const Font_Holder &fonts, float character_size );
                bool is_selectable() const override;
                void set_text( const std::string &text );
                void handle_event( const sf::Event &event ) override;
            private:
                sf::Text _text; /** The text meant to be displayed */
                void draw( sf::RenderTarget &target, sf::RenderStates states ) const override;
            };
        }
    }
}

#endif //SOKOBAN_LABEL_HPP
```

9.7.8 Label.cpp

```
#include "Label.hpp"

#include "../Utility.hpp"

#include <SFML/Graphics/RenderStates.hpp>
#include <SFML/Graphics/RenderTarget.hpp>

using namespace sokoban::ui::gui;

/**
 * Constructor for the Label component
 * @param text The text meant to be shown
 * @param fonts The font it should be characterized in
 * @param character_size The character size
 */
Label::Label( const std::string &text, const sokoban::ui::Font_Holder &fonts, float character_size )
: _text( text, fonts.get( Fonts::Rampart_One ), character_size )
{
}

/**
 * Make the label not selectable
 * @return false since a label is just text
 */
bool Label::is_selectable() const
{
    return false;
}

/**
 * Setter for the text in case of changes
 * @param text The text meant to be shown
 */
void Label::set_text( const std::string &text )
{
    _text.setString( text );
}

/**
 * Empty since a label does not execute any sort of action by default
 * @param event None
 */
void Label::handle_event( const sf::Event &event )
{
}

/**
 * Function that will display the label visually
 */
void Label::draw( sf::RenderTarget &target, sf::RenderStates states ) const
{
    states.transform *= getTransform();
    target.draw( _text, states );
}
```

9.8 Entities

9.8.1 Entity.hpp

```
#ifndef SOKOBAN_ENTITY_HPP
#define SOKOBAN_ENTITY_HPP

#include "../Scene_Node.hpp"

#include <map>
#include <array>
#include <string>
#include <ostream>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            namespace entity
            {
                class Entity
                        : public Scene_Node
                {
                public:
                    explicit Entity( std::array< float, 4 > asset_coords );
                    Entity( const Entity &entity );
                    Entity &operator=( const Entity &entity );
                    ~Entity() override;
                    std::array< float, 4 > get_asset_coords() const;
                    void set_asset_coords( std::array< float, 4 > asset_coords );
                    void update_current( sf::Time dt, Command_Queue &commands ) override;
                private:
                    /** The asset's coordinates on the spritesheet */
                    std::array< float, 4 > _assets_coords;
                };
            }
        }
    }
}

#endif //SOKOBAN_ENTITY_HPP
```

9.8.2 Entity.cpp

```
#include "Entity.hpp"

#include <cassert>

using namespace sokoban::ui::gui::entity;

/**
 * Entity constructor meant to initialize the asset's coordinates on the sprite sheet
 */
Entity::Entity( std::array< float, 4 > asset_coords )
: _assets_coords( asset_coords )
{
}

/**
 * Copy constructor for the entity
 * @param entity The entity to copy the information from
 */
Entity::Entity( const Entity &entity )
: Entity( entity._assets_coords )
{
}

/**
 * Redefinition of the = operator
 * @param entity The entity to copy the information from
 * @return The new instance of the entity with the copied information
 */
Entity &Entity::operator=( const Entity &entity )
{
    if( &entity != this )
    {
        _assets_coords = entity._assets_coords;
    }
    return *this;
}

/**
 * Default entity destructor
 */
Entity::~Entity()
= default;

/**
 * Getter for the assets coordinates
 * @return Array of floats containing the asset's coordinates
 */
std::array< float, 4 > Entity::get_asset_coords() const
{
    return _assets_coords;
}

/**
 * Setter for the asset's coordinates
 * @param asset_coords The assets to retrieve and set as new
 */
```

```
void Entity::set_asset_coords( std::array< float, 4 > asset_coords )
{
    _assets_coords = asset_coords;
}

/**
 * Function to update an Entity graphically based on its action
 * @param dt The clock time
 * @param commands The commands to execute
 */
void Entity::update_current( sf::Time dt, Command_Queue &commands )
{
}
```

9.8.3 Entity_Box.hpp

```
#ifndef SOKOBAN_ENTITY_BOX_HPP
#define SOKOBAN_ENTITY_BOX_HPP

#include "../../../model/Box.hpp"
#include "Entity_Movable.hpp"

#include <map>
#include <array>
#include <string>
#include <ostream>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            namespace entity
            {

                class Entity_Box
                          : public model::Box
                            , public Entity_Movable
                {

                public:
                    /** The various colors of a box */
                    enum class Color
                    {
                        BEIGE_LIGHT,
                        BLACK_LIGHT,
                        BLUE_LIGHT,
                        BROWN_LIGHT,
                        BEIGE_DARK,
                        BLACK_DARK,
                        BLUE_DARK,
                        BROWN_DARK,
                        WHITE_DARK,
                        PURPLE_DARK,
                        RED_DARK,
                        YELLOW_DARK,
                        WHITE_LIGHT,
                        PURPLE_LIGHT,
                        RED_LIGHT,
                        YELLOW_LIGHT
                    };
                    Entity_Box( float x, float y );
                    Entity_Box( const Entity_Box &box );
                    Entity_Box &operator=( const Entity_Box &box );
                    ~Entity_Box() override;
                    ID get_type() const override;
                    std::string to_string() const override;
                    friend std::ostream &operator<<( std::ostream &os, const Entity_Box &box );
                    const std::map< Color, std::array< float, 4>> &get_box_color_map() const;
                private:
                    /** Mapper containing the coordinates of each asset based on the color */
                    std::map< Color, std::array< float, 4 > > _box_color_map;
                };
            }
        }
    }
}

#endif //SOKOBAN_ENTITY_BOX_HPP
```

9.8.4 Entity_Box.cpp

```cpp
#include "Entity_Box.hpp"

using namespace sokoban::ui::gui::entity;
using namespace sokoban::model;

const float OFFSET = 64;

const std::array< float, 4 > box_beige_light_asset = {
        0
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_black_light_asset = {
        OFFSET * 1
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_blue_light_asset = {
        OFFSET * 2
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_brown_light_asset = {
        OFFSET * 3
        , 0
        , OFFSET
        , OFFSET
}; /** The default asset for the box */
const std::array< float, 4 > box_beige_dark_asset = {
        OFFSET * 4
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_black_dark_asset = {
        OFFSET * 5
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_blue_dark_asset = {
        OFFSET * 6
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_brown_dark_asset = {
        OFFSET * 7
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_white_dark_asset = {
        OFFSET * 8
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_purple_dark_asset = {
        OFFSET * 9
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_red_dark_asset = {
        OFFSET * 10
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_yellow_dark_asset = {
        OFFSET * 11
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_white_light_asset = {
        OFFSET * 12
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_purple_light_asset = {
        OFFSET * 13
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_red_light_asset = {
        OFFSET * 14
        , 0
        , OFFSET
        , OFFSET
};
const std::array< float, 4 > box_yellow_light_asset = {
        OFFSET * 15
        , 0
        , OFFSET
        , OFFSET
};

/**
 * Constructor for the entity box
 * @param x The position on the X axis
 * @param y The position on the Y axis
 */
Entity_Box::Entity_Box( float x, float y )
        : model::Box( x, y )
        , Entity_Movable( box_brown_light_asset )
{

    _box_color_map = std::map< Color, std::array< float, 4 > >
            {
                    {   Color::BEIGE_LIGHT  , box_beige_light_asset }
                    , { Color::BLACK_LIGHT  , box_black_light_asset }
                    , { Color::BLUE_LIGHT   , box_blue_light_asset }
                    , { Color::BROWN_LIGHT  , box_brown_light_asset }
                    , { Color::BEIGE_DARK   , box_beige_dark_asset }
                    , { Color::BLACK_DARK   , box_black_dark_asset }
                    , { Color::BLUE_DARK    , box_blue_dark_asset }
                    , { Color::BROWN_DARK   , box_brown_dark_asset }
                    , { Color::WHITE_DARK   , box_white_dark_asset }
                    , { Color::PURPLE_DARK  , box_purple_dark_asset }
                    , { Color::RED_DARK     , box_red_dark_asset }
                    , { Color::YELLOW_DARK  , box_yellow_dark_asset }
                    , { Color::WHITE_LIGHT  , box_white_light_asset }
                    , { Color::PURPLE_LIGHT , box_purple_light_asset }
                    , { Color::RED_LIGHT    , box_red_light_asset }
                    , { Color::YELLOW_LIGHT , box_yellow_light_asset }
            };
}

/**
 * Copy constructor of a box entity
 */
Entity_Box::Entity_Box( const Entity_Box &box )
        : Entity_Box( box.get_x(), box.get_y() )
{

}

/**
 * Redefinition of the = operator
 * @param box The box we wish to copy the information from
 * @return The new box with the copied box's information
 */
Entity_Box &Entity_Box::operator=( const Entity_Box &box )
{
    if ( &box != this )
    {
        set_asset_coords( box.get_asset_coords() );
    }
    return *this;
}

/**
 * Based destructor for the box entity
 */
Entity_Box::~Entity_Box()
= default;

/**
 * Getter meant to retrieve the actor's nature
 */
Actor::ID Entity_Box::get_type() const
```

```
{
    return Actor::BOX;
}

/**
 * Textual format of the box's information
 * @return The Box Entity's information in textual format
 */
std::string Entity_Box::to_string() const
{
    return Box::to_string();
}

/**
 * Getter for the mapper containing the various coordinates of the box based on the color
 * @return The color's mapper
 */
const std::map< Entity_Box::Color, std::array< float, 4>> &Entity_Box::get_box_color_map() const
{
    return _box_color_map;
}

/**
 * Redefinition of the << operator meant to output the box directly within an std::cout
 * @param os The output stream to send the text to
 * @param box The box we wish to output
 * @return Textual output containing the Entity's information
 */
std::ostream &sokoban::ui::gui::entity::operator<<( std::ostream &os, const Entity_Box &box )
{
    os << box.to_string();
    return os;
}
```

9.8.5 Entity_Movable.hpp

```
#ifndef SOKOBAN_ENTITY_MOVABLE_HPP
#define SOKOBAN_ENTITY_MOVABLE_HPP

#include "Entity.hpp"

#include <map>
#include <array>
#include <string>
#include <ostream>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            namespace entity
            {
                class Entity_Movable
                        : public Entity
                {
                public:
                    explicit Entity_Movable( std::array< float, 4 > asset_coords );
                    void set_direction( float dx, float dy );
                    sf::Vector2f get_direction() const;
                    void update_current( sf::Time dt, Command_Queue &commands ) override;
                private:
                    /** The direction the Entity has to move to */
                    sf::Vector2f _direction;
                };
            }
        }
    }
}

#endif //SOKOBAN_ENTITY_MOVABLE_HPP
```

9.8.6 Entity_Movable.cpp

```
#include "Entity_Movable.hpp"

using namespace sokoban::ui::gui::entity;

/**
 * Constructor for a movable entity
 * @param asset_coords The asset's coordinates
```

```cpp
 */
Entity_Movable::Entity_Movable( std::array< float, 4 > asset_coords )
        : Entity( asset_coords )
        , _direction()
{
}

/**
 * The direction where the entity moves to
 * @param dx Movement distance of the X axis
 * @param dy Movement distance of the Y axis
 */
void Entity_Movable::set_direction( float dx, float dy )
{
    _direction.x = dx;
    _direction.y = dy;
}

/**
 * Getter for the direction where the entity moves to
 * @return The vector containing the X and Y axis of the direction
 */
sf::Vector2f Entity_Movable::get_direction() const
{
    return _direction;
}

/**
 * Visual real-time update of a movable entity along with its commands
 */
void Entity_Movable::update_current( sf::Time dt, Command_Queue &commands )
{
    Entity_Movable::move( _direction * dt.asSeconds() );
}
```

9.8.7 Entity_Platform.hpp

```cpp
#ifndef SOKOBAN_ENTITY_PLATFORM_HPP
#define SOKOBAN_ENTITY_PLATFORM_HPP

#include "../../../model/Platform.hpp"
#include "Entity.hpp"

#include <map>
#include <array>
#include <string>
#include <ostream>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            namespace entity
            {
                class Entity_Platform
                        : public model::Platform
                        , public Entity
                {
                public:
                    enum class Color
                    {
                        BEIGE,
                        BLACK,
                        BLUE,
                        BROWN,
                        WHITE,
                        PURPLE,
                        RED,
                        YELLOW
                    };
                    Entity_Platform( float x, float y );
                    Entity_Platform( const Entity_Platform &entity );
                    Entity_Platform &operator=( const Entity_Platform &entity );
                    ~Entity_Platform() override;
                    std::string to_string() const;
                    const std::map< Color, std::array< float, 4>> &get_platform_color_map() const;
                    friend std::ostream &operator<<( std::ostream &os, const Entity_Platform &entity );
                private:
                    /** Mapper containing the coordinates of each asset based on the color */
                    std::map< Color, std::array< float, 4 > > _platform_color_map;
                };
            }
        }
    }
}

#endif //SOKOBAN_ENTITY_PLATFORM_HPP
```

9.8.8 Entity_Platform.cpp

```
#include "Entity_Platform.hpp"

using namespace sokoban::ui::gui::entity;

const float OFFSET = 32;

const std::array< float, 4 > platform_beige_asset = {
        0
        , 0
        , OFFSET
        , OFFSET
};

const std::array< float, 4 > platform_black_asset = {
        OFFSET * 1
        , 0
        , OFFSET
        , OFFSET
};

const std::array< float, 4 > platform_blue_asset = {
        OFFSET * 2
        , 0
        , OFFSET
        , OFFSET
};

const std::array< float, 4 > platform_brown_asset = {
        OFFSET * 3
        , 0
        , OFFSET
        , OFFSET
};

const std::array< float, 4 > platform_white_asset = {
        OFFSET * 4
        , 0
        , OFFSET
        , OFFSET
};

const std::array< float, 4 > platform_purple_asset = {
        OFFSET * 5
        , 0
        , OFFSET
        , OFFSET
};

const std::array< float, 4 > platform_red_asset = {
        OFFSET * 6
        , 0
        , OFFSET
        , OFFSET
};

const std::array< float, 4 > platform_yellow_asset = {
        OFFSET * 7
        , 0
        , OFFSET
        , OFFSET
};

/**
 * Default Entity Platform constructor
 * @param x The coordinates on the X axis
 * @param y The coordinates on the Y axis
 */
Entity_Platform::Entity_Platform( float x, float y )
: Platform( x, y )
, Entity( platform_blue_asset )
{
    _platform_color_map = std::map< Color, std::array< float, 4 > >
            {
                    {   Color::BEIGE , platform_beige_asset }
```

```
                        , { Color::BLACK , platform_black_asset }
                        , { Color::BLUE  , platform_blue_asset }
                        , { Color::BROWN , platform_brown_asset }
                        , { Color::WHITE , platform_white_asset }
                        , { Color::PURPLE, platform_purple_asset }
                        , { Color::RED   , platform_red_asset }
                        , { Color::YELLOW, platform_yellow_asset }
                };

}

/**
 * The copy constructor for the Entity Platform
 * @param entity The Platform we wish to copy the information from
 */
Entity_Platform::Entity_Platform( const Entity_Platform &entity )
: Entity_Platform( entity.get_x(), entity.get_y() )
{
}

/**
 * Redefinition of the = operator
 * @param entity The entity we wish to copy the information from
 * @return The entity with the copied platform's information
 */
Entity_Platform &Entity_Platform::operator=( const Entity_Platform &entity )
{
    if( &entity != this )
    {
        model::Platform::operator=( entity );
        Entity::operator=( entity );
    }
    return *this;
}

/**
 * Default constructor for the Platform entity
 */
Entity_Platform::~Entity_Platform()
= default;

/**
 * Textual output containing the platform's information
 * @return The textual output with the platform's information
 */
std::string Entity_Platform::to_string() const
{
    return model::Platform::to_string();
}

/**
 * Getter for the mapper containing the asset's coordinates based on the color
 * @return The mapper containing the coordinates based on the color
 */
const std::map< Entity_Platform::Color, std::array< float, 4>> &Entity_Platform::get_platform_color_map()
     const
{
    return _platform_color_map;
}

/**
 * Redefinition of the << operator
 * @param os The output stream to display the text to
 * @param entity The entity we wish to output
 * @return The textual output containing the entity's information
 */
std::ostream &sokoban::ui::gui::entity::operator<<( std::ostream &os, const Entity_Platform &entity )
{
    os << entity.to_string();
    return os;
}
```

9.8.9 Entity_Player.hpp

```
#ifndef SOKOBAN_ENTITY_PLAYER_HPP
#define SOKOBAN_ENTITY_PLAYER_HPP

#include "../../../model/Player.hpp"
#include "Entity_Movable.hpp"

#include <SFML/Window/Event.hpp>

#include <map>
```

```cpp
#include <array>
#include <string>
#include <ostream>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            namespace entity
            {
                class Entity_Player
                        : public model::Player
                          , public Entity_Movable
                {
                public:
                    enum class Face
                    {
                        NORTH,
                        SOUTH,
                        EAST,
                        WEST
                    };
                    Entity_Player( float x, float y );
                    Entity_Player( const Entity_Player &entity );
                    Entity_Player &operator=( const Entity_Player &entity );
                    ~Entity_Player() override;
                    std::string to_string() const override;
                    const std::map< Face, std::array< float, 4>> &get_player_face_map() const;
                    friend std::ostream &operator<<( std::ostream &os, const Entity_Player &entity );
                private:
                    /** Mapper containing the coordinates of each asset based on the direction the player is
      looking */
                    std::map< Face, std::array< float, 4 > > _player_face_map;
                };
            }
        }
    }
}

#endif //SOKOBAN_ENTITY_PLAYER_HPP
```

9.8.10 Entity_Player.cpp

```cpp
#include "Entity_Player.hpp"

using namespace sokoban::ui::gui::entity;

const float OFFSET = 64;

std::array< float, 4 > character_facing_west = {
        0
        , 0
        , OFFSET
        , OFFSET
};
std::array< float, 4 > character_walking_west = {
        OFFSET * 1
        , 0
        , OFFSET
        , OFFSET
};

std::array< float, 4 > character_facing_east = {
        OFFSET * 2
        , 0
        , OFFSET
        , OFFSET
};
std::array< float, 4 > character_walking_east = {
        OFFSET * 3
        , 0
        , OFFSET
        , OFFSET
};

std::array< float, 4 > character_facing_south = {
        OFFSET * 4
        , 0
        , OFFSET
        , OFFSET
};
```

```

std::array< float, 4 > character_walking_south_right_step = {
        OFFSET * 5
        , 0
        , OFFSET
        , OFFSET
};
std::array< float, 4 > character_walking_south_left_step = {
        OFFSET * 6
        , 0
        , OFFSET
        , OFFSET
};

std::array< float, 4 > character_facing_north = {
        OFFSET * 7
        , 0
        , OFFSET
        , OFFSET
};
std::array< float, 4 > character_walking_north_right_step = {
        OFFSET * 8
        , 0
        , OFFSET
        , OFFSET
};
std::array< float, 4 > character_walking_north_left_step = {
        OFFSET * 9
        , 0
        , OFFSET
        , OFFSET
};

const std::array< float, 4 > player_asset_north = character_facing_north;
const std::array< float, 4 > player_asset_south = character_facing_south;
const std::array< float, 4 > player_asset_east = character_facing_east;
const std::array< float, 4 > player_asset_west = character_facing_west;

/**
 * Default constructor for the Player entity
 * @param x The coordinates on the X axis
 * @param y The coordinates on the Y axis
 */
Entity_Player::Entity_Player( float x, float y )
        : model::Player( x, y )
        , Entity_Movable( player_asset_south )
{
    _player_face_map = std::map< Face, std::array< float, 4 > >
            {
                    {   Face::NORTH, player_asset_north }
                    , { Face::SOUTH, player_asset_south }
                    , { Face::EAST , player_asset_east }
                    , { Face::WEST , player_asset_west }
            };
}

/**
 * Copy constructor for the player entity
 * @param entity The entity we wish to get the information from
 */
Entity_Player::Entity_Player( const Entity_Player &entity )
: Entity_Player( entity.get_x(), entity.get_y() )
{
    set_asset_coords( entity.get_asset_coords() );
    _player_face_map = entity._player_face_map;
}

/**
 * Redefinition of the = operator
 * @param entity The entity we wish to get the information from
 * @return The new instance of a place with the copied information
 */
Entity_Player &Entity_Player::operator=( const Entity_Player &entity )
{
    if( &entity != this )
    {
        model::Player::operator=( entity );
        Entity::operator=( entity );
        set_asset_coords( entity.get_asset_coords() );
        _player_face_map = entity._player_face_map;
    }
    return *this;
}

```

```
/**
 * Default destructor for the entity player
 */
Entity_Player::~Entity_Player()
= default;

/**
 * Textual output containing the Player's information
 * @return Text containing the player's information
 */
std::string Entity_Player::to_string() const
{
    return Player::to_string();
}

/**
 * Getter for the mapper containing the coordinates of each asset based on where the player is looking
 * @return The mapper containing the assets
 */
const std::map< Entity_Player::Face, std::array< float, 4>> &Entity_Player::get_player_face_map() const
{
    return _player_face_map;
}

/**
 * Redefinition of the << operator
 * @param os The output we wish to inject the text into
 * @param entity The entity we wish to output
 * @return Textual output displaying the player's information
 */
std::ostream &sokoban::ui::gui::entity::operator<<( std::ostream &os, const Entity_Player &entity )
{
    os << entity.to_string();
    return os;
}
```

9.8.11 Entity_Wall.hpp

```
#ifndef SOKOBAN_ENTITY_WALL_HPP
#define SOKOBAN_ENTITY_WALL_HPP

#include "../../../model/Wall.hpp"
#include "Entity.hpp"

#include <map>
#include <array>
#include <string>
#include <ostream>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            namespace entity
            {
                class Entity_Wall
                        : public model::Wall
                          , public Entity
                {
                public:
                    enum class Color
                    {
                        BEIGE,
                        BLACK,
                        BROWN,
                        WHITE
                    };
                    Entity_Wall( float x, float y );
                    Entity_Wall( const Entity_Wall &entity );
                    Entity_Wall &operator=( const Entity_Wall &entity );
                    ~Entity_Wall() override;
                    std::string to_string() const;
                    const std::map< Color, std::array< float, 4>> &get_wall_color_map() const;
                    friend std::ostream &operator<<( std::ostream &os, const Entity_Wall &entity );
                private:
                    /** Mapper containing the coordinates of each asset based on the color */
                    std::map< Color, std::array< float, 4 > > _wall_color_map;
                };
            }
        }
    }
```

```cpp
}

#endif //SOKOBAN_ENTITY_WALL_HPP
```

9.8.12 Entity_Wall.cpp

```cpp
#include "Entity_Wall.hpp"

using namespace sokoban::ui::gui::entity;

const float OFFSET = 64;

const std::array< float, 4 > wall_round_beige_asset = {
        0
        , 0
        , OFFSET
        , OFFSET
};

const std::array< float, 4 > wall_round_black_asset = {
        OFFSET * 1
        , 0
        , OFFSET
        , OFFSET
};

const std::array< float, 4 > wall_round_brown_asset = {
        OFFSET * 2
        , 0
        , OFFSET
        , OFFSET
};

const std::array< float, 4 > wall_round_white_asset = {
        OFFSET * 3
        , 0
        , OFFSET
        , OFFSET
};

/**
 * Default constructor for the Wall Entity
 * @param x The coordinates on the X axis
 * @param y The coordinates on the Y axis
 */
Entity_Wall::Entity_Wall( float x, float y )
        : model::Wall( x, y )
        , Entity( wall_round_white_asset )
{
    _wall_color_map = std::map< Color, std::array< float, 4 > >
            {
                    {   Color::BEIGE, wall_round_beige_asset }
                    , { Color::BLACK, wall_round_black_asset }
                    , { Color::BROWN, wall_round_brown_asset }
                    , { Color::WHITE, wall_round_white_asset }
            };
}

/**
 * Copy constructor for the Wall entity
 * @param entity The entity we wish to copy the information from
 */
Entity_Wall::Entity_Wall( const Entity_Wall &entity )
: Entity_Wall( entity.get_x(), entity.get_y() )
{
}

/**
 * Redefinition of the = operator
 * @param entity The entity we wish to copy the information from
 * @return The new entity with the copied information
 */
Entity_Wall &Entity_Wall::operator=( const Entity_Wall &entity )
{
    if( &entity != this )
    {
        model::Wall::operator=( entity );
        Entity::operator=( entity );
    }
    return *this;
}

/**
```

```cpp
 * Default destructor for the Wall entity
 */
Entity_Wall::~Entity_Wall()
= default;

/**
 * Textual output of the Wall entity
 * @return Textual output containing the Wall's information
 */
std::string Entity_Wall::to_string() const
{
    return Wall::to_string();
}

/**
 * Getter for the mapper containing the coordinates of the assets based on each color
 * @return The mapper containing the coordinates of the assets based on each color
 */
const std::map< Entity_Wall::Color, std::array< float, 4>> &Entity_Wall::get_wall_color_map() const
{
    return _wall_color_map;
}

/**
 * Redefinition of the << operator
 * @param os The output we wish to inject our information to
 * @param entity The entity we wish to get the information from
 * @return The textual output containing the entity's information
 */
std::ostream &sokoban::ui::gui::entity::operator<<( std::ostream &os, const Entity_Wall &entity )
{
    os << entity.to_string();
    return os;
}
```

9.9 States

9.9.1 State_Identifiers.hpp

```cpp
#ifndef SOKOBAN_STATE_IDENTIFIERS_HPP
#define SOKOBAN_STATE_IDENTIFIERS_HPP

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            namespace States
            {
                /**
                 * ID of each state that characterizes it
                 */
                enum ID
                {
                    None, /** No nature whatsoever */
                    Title, /** The title screen */
                    Menu, /** The menu screen */
                    Game, /** The playable area */
                    Pause, /** The pause screen */
                    Settings, /** The settings screen */
                };
            }
        }
    }
}

#endif //SOKOBAN_STATE_IDENTIFIERS_HPP
```

9.9.2 State.hpp

```cpp
#pragma once
#ifndef SOKOBAN_STATE_HPP
#define SOKOBAN_STATE_HPP

#include "State_Identifiers.hpp"
#include "../Resource_Identifiers.hpp"

#include <SFML/System/Time.hpp>
#include <SFML/Window/Event.hpp>

#include <memory>
```

```

/**
 * Pre declaration of the SFML RenderWindow class
 */
namespace sf
{
    class RenderWindow;
}

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class State_Stack; /** The stack containing the various states */
            class Music_Player; /** The global music player */
            class Sound_Player; /** The global sound effects player */
            class State
            {
            public:
                /** Unique pointer that characterizes the base nature of a state */
                typedef std::unique_ptr< State > Ptr;

                /**
                 * Public struct containing globally accessible variables
                 */
                struct Context
                {
                    Context( sf::RenderWindow &window, Texture_Holder &textures, Font_Holder &fonts,
    Music_Player &music, Sound_Player &sounds );
                    sf::RenderWindow *_window;
                    Texture_Holder *_textures;
                    Font_Holder *_fonts;
                    Music_Player *_music;
                    Sound_Player *_sounds;
                };

                State( State_Stack &stack, Context context );
                virtual ~State();
                virtual void draw() = 0;
                virtual bool update( sf::Time dt ) = 0;
                virtual bool handle_event( const sf::Event &event ) = 0;
            protected:
                void request_stack_push( States::ID stateID );
                void request_stack_pop();
                void request_state_clear();
                Context get_context() const;
            private:
                State_Stack *_stack; /** The stack containing the various states */
                Context _context; /** The globally accessible context */
            };
        }
    }
}

#endif //SOKOBAN_STATE_HPP
```

9.9.3 State.cpp

```
#include "State.hpp"

#include "State_Stack.hpp"

using namespace sokoban::ui::gui;

/**
 * Default constructor for the context
 * @param window The window where the entities and components have to be displayed on
 * @param textures The various textures that characterize each entity and component
 * @param fonts The various fonts used throughout the usage of the software
 * @param music The various songs that can be heard
 * @param sounds The various sound effects played
 */
State::Context::Context( sf::RenderWindow &window, Texture_Holder &textures, Font_Holder &fonts,
    Music_Player &music, Sound_Player &sounds )
        : _window( &window )
          , _textures( &textures )
          , _fonts( &fonts )
          , _music( &music )
          , _sounds( &sounds )
{
}
```

```cpp

/**
 * Default constructor for a state
 * @param stack The stack containing the state
 * @param context The context that can be accessed throughout each state
 */
State::State( State_Stack &stack, State::Context context )
        : _stack( &stack )
          , _context( context )
{
}

/**
 * Default state destructor
 */
State::~State()
= default;

/**
 * Function meant to insert a state inside the stack based on its ID
 * @param stateID The unique ID of the state
 */
void State::request_stack_push( States::ID stateID )
{
    _stack->push_state( stateID );
}

/**
 * Function meant to remove a state from the stack
 */
void State::request_stack_pop()
{
    _stack->pop_state();
}

/**
 * Stack meant to clear out completely the stack
 */
void State::request_state_clear()
{
    _stack->clear_states();
}

/**
 * Getter for the context accessed all around the states
 * @return The context to which we can access the various resources
 */
State::Context State::get_context() const
{
    return _context;
}
```

9.9.4 State_Game.hpp

```cpp
#ifndef SOKOBAN_STATE_GAME_HPP
#define SOKOBAN_STATE_GAME_HPP

#include "State.hpp"
#include "../World.hpp"

#include <SFML/Graphics/Text.hpp>
#include <SFML/Graphics/Sprite.hpp>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class State_Game
                    : public State
            {
            public:
                State_Game( State_Stack &stack, Context context );
                ~State_Game() override;
                bool update( sf::Time dt ) override;
                void draw() override;
                bool handle_event( const sf::Event &event ) override;
            private:
                World *_world; /** The map that has to be displayed */
                sf::RenderWindow &_window; /** The window where everything has to be drawn into */
                std::vector< std::string > _levels; /** The various levels throughout the game */
                std::string _level; /** The level that is currently being played */
                sf::Text _text; /** The textual information present on the screen */
                void reset_board();
                void next_level();
                void prev_level();
            };
        }
    }
}

#endif //SOKOBAN_STATE_GAME_HPP
```

9.9.5 State_Game.cpp

```
#include "State_Game.hpp"

#include "../../../util/Logger.hpp"

#include "../Music_Player.hpp"
#include "../Utility.hpp"

#include <SFML/Graphics/RenderWindow.hpp>

#include <boost/filesystem.hpp>

using namespace sokoban::ui::gui;
using namespace sokoban::util;


namespace
{
    int current_level;
    int reset_counter = 0;
}

/**
 * Alphabetical sorting algorithm for the various paths found
 * @param a A first path to sort
 * @param b A second path to sort with
 * @return The paths in the right order
 */
bool sort_alphabetically( const boost::filesystem::path &a, const boost::filesystem::path &b )
{
    return a.string() < b.string();
}

/**
 * Getter for the various levels that have been detected inside the folder
 */
std::vector< boost::filesystem::path > get_all_levels()
{
    const boost::filesystem::path root = "assets/levels";
    const std::string extension = ".lvl";
    std::vector< boost::filesystem::path > paths;
    if ( boost::filesystem::exists( root ) && boost::filesystem::is_directory( root ) )
    {
        for ( auto const &entry: boost::filesystem::recursive_directory_iterator( root ) )
        {
            if ( boost::filesystem::is_regular_file( entry ) )
            {
                paths.emplace_back( entry );
            }
        }
    }

    std::sort( paths.begin(), paths.end(), sort_alphabetically );

    return paths;
}

/**
 * Default constructor for the Game State
 * @param stack The stack containing the various states
 * @param context The context containing the various resources
 */
State_Game::State_Game( State_Stack &stack, Context context )
        : State( stack, context )
        , _window( *context._window )
          , _level()
          , _levels()
          , _text()
{
    _window.setKeyRepeatEnabled( true );
    context._music->play( Music::Town_Peaceful_Place );
```

```cpp
    _text.setFont( context._fonts->get( Fonts::Connection_II ) );
    _text.setFillColor( sf::Color::Black );
    Logger::log( LoggerLevel::INFO, "Init levels" );
    _levels = std::vector< std::string >();

    if ( get_all_levels().empty() )
    {
        Logger::log( LoggerLevel::INFO, "No levels loaded" );
        return;
    }

    for ( const boost::filesystem::path &path: get_all_levels() )
    {
        _levels.emplace_back( path.string() );
    }

    Logger::log( LoggerLevel::INFO, "Levels loaded" );
    for ( const std::string &lvl: _levels )
    {
        Logger::log( LoggerLevel::INFO, lvl );
    }
    current_level = 0;
    _text.setFont( context._fonts->get( Fonts::Connection_II ) );
    _level = _levels.at( current_level );
    Logger::log( LoggerLevel::DEBUG, "Level loaded: " + _level );
    _world = new World( *context._window, model::Board( _level ), *context._fonts, *context._sounds );
}

/**
 * Default destructor for the Game state
 */
State_Game::~State_Game()
{
    delete _world;
}

/**
 * Realtime update of the visually available Game window
 * @param dt The clock time
 * @return always true
 */
bool State_Game::update( sf::Time dt )
{
    _world->update( dt );
    return true;
}

/**
 * Function to draw the visual components of the Game state
 */
void State_Game::draw()
{
    _world->draw();
    if ( _world->is_board_completed() )
    {
        if ( _levels.size() <= current_level )
        {
            _text.setCharacterSize( 50 );
            Utility::center_origin( _text );
            _text.setString( L"All Levels Completed!" );
            _text.setPosition( get_context()._window->getView().getSize() / 2.f );
            get_context()._window->draw( _text );
        }
        else
        {
            _text.setCharacterSize( 50 );
            Utility::center_origin( _text );
            _text.setString( L"Level Completed\nPress ENTER to continue!" );
            _text.setPosition( get_context()._window->getView().getSize() / 2.f );
            get_context()._window->draw( _text );
        }
    }
}

/**
 * Event handler meant for each interaction possible within the Game state
 * @param event The event executed
 * @return always true
 */
bool State_Game::handle_event( const sf::Event &event )
{
    if( event.type == sf::Event::KeyPressed )
    {
        if( _world->is_board_completed() )
        {
            if ( event.key.code == sf::Keyboard::Enter )
            {
                next_level();
            }
        }
        else
        {
            if( event.key.code == sf::Keyboard::Escape )
            {
                request_stack_push( States::Pause );
            }
            else if( event.key.code == sf::Keyboard::R )
            {
                reset_board();
            }
            else if( event.key.code == sf::Keyboard::S )
            {
                next_level();
            }
            else if( event.key.code == sf::Keyboard::X )
            {
                prev_level();
            }
            if( event.key.code == sf::Keyboard::Up )
            {
                _world->move_up( true );
            }
            else if( event.key.code == sf::Keyboard::Down )
            {
                _world->move_down( true );
            }
            else if( event.key.code == sf::Keyboard::Left )
            {
                _world->move_left( true );
            }
            else if( event.key.code == sf::Keyboard::Right )
            {
                _world->move_right( true );
            }
        }
    }
    else if( event.type == sf::Event::KeyReleased )
    {
        if( _world->is_board_completed() )
        {
        }
        else
        {
            if( event.key.code == sf::Keyboard::Up )
            {
                _world->move_up( false );
            }
            else if( event.key.code == sf::Keyboard::Down )
            {
                _world->move_down( false );
            }
            else if( event.key.code == sf::Keyboard::Left )
            {
                _world->move_left( false );
            }
            else if( event.key.code == sf::Keyboard::Right )
            {
                _world->move_right( false );
            }
        }
    }
    return true;
}

/**
 * Board resetting function meant to restart a game board
 */
void State_Game::reset_board()
{
    delete _world;
    _world = new World( _window, model::Board( _level ), *get_context()._fonts, *get_context()._sounds );
    reset_counter++;
    _world->set_reset_counter( reset_counter );
}

/**
 * Function meant to move onto the next level
 */
```

```
void State_Game::next_level()
{
    current_level += 1;
    if ( _levels.size() <= current_level )
    {
        current_level = _levels.size() - 1;
        return;
    }
    _level = _levels.at( current_level );
    delete _world;
    _world = new World( _window, model::Board( _level ), *get_context()._fonts, *get_context()._sounds );
    reset_counter = 0;
    _world->set_reset_counter( reset_counter );
}

/**
 * Function meant to return to a previous level
 */
void State_Game::prev_level()
{
    current_level -= 1;
    if ( current_level <= 0 )
    {
        current_level = 1;
        return;
    }
    _level = _levels.at( current_level );
    delete _world;
    _world = new World( _window, model::Board( _level ), *get_context()._fonts, *get_context()._sounds );
    reset_counter = 0;
    _world->set_reset_counter( reset_counter );
}
```

9.9.6 State_Menu.hpp

```
#ifndef SOKOBAN_STATE_MENU_HPP
#define SOKOBAN_STATE_MENU_HPP

#include "State.hpp"
#include "../components/Container.hpp"

#include <SFML/Graphics/Text.hpp>
#include <SFML/Graphics/Sprite.hpp>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class State_Menu
                    : public State
            {
            public:
                State_Menu( State_Stack &stack, Context context );
                void draw() override;
                bool update( sf::Time dt ) override;
                bool handle_event( const sf::Event &event ) override;
            private:
                sf::Sprite _background_sprite; /** The background sprite */
                Container _container; /** Container storing the various components */
            };
        }
    }
}

#endif //SOKOBAN_STATE_MENU_HPP
```

9.9.7 State_Menu.cpp

```
#include "State_Menu.hpp"

#include "../components/Button.hpp"
#include "../Utility.hpp"
#include "../../Resource_Holder.hpp"
#include "../Music_Player.hpp"

#include <SFML/Graphics/Text.hpp>
#include <SFML/Graphics/RenderWindow.hpp>

#include <cmath>

```

```cpp
using namespace sokoban::ui::gui;

/**
 * Default constructor for the Menu state
 * @param stack The stack containing the various states
 * @param context The context meant to access all the resources available
 */
State_Menu::State_Menu( State_Stack &stack, State::Context context )
        : State( stack, context )
        , _container()
{
    sf::Texture &texture = context._textures->get( Textures::Title_Screen );
    sf::Font &font = context._fonts->get( Fonts::Rampart_One );

    _background_sprite.setTexture( texture );
    Utility::center_origin( _background_sprite );
    _background_sprite.setPosition( context._window->getView().getSize() / 2.f );

    auto play_button = std::make_shared< Button >( context );
    play_button->set_text( "Play" );
    play_button->set_callback( [ this ] ()
    {
        request_stack_pop();
        request_stack_push( States::Game );
    });

    auto settings_button = std::make_shared< Button >( context );
    settings_button->set_text( "Settings" );
    settings_button->set_callback( [ this ] ()
    {
        request_stack_push( States::Settings );
    });

    auto exit_button = std::make_shared< Button >( context );
    exit_button->set_text( "Exit" );
    exit_button->set_callback( [ this ] ()
    {
        get_context()._window->close();
    });

    settings_button->setPosition( context._window->getView().getSize() / 2.f );
    settings_button->setOrigin( 100.f, 25.f );
    play_button->setPosition( settings_button->getPosition() - sf::Vector2f( 0, 100.f ) );
    play_button->setOrigin( 100.f, 25.f );
    exit_button->setPosition( settings_button->getPosition() + sf::Vector2f( 0, 100.f ) );
    exit_button->setOrigin( 100.f, 25.f );

    _container.pack( play_button );
    _container.pack( settings_button );
    _container.pack( exit_button );
}

/**
 * Visually draw the various components and entities with the state
 */
void State_Menu::draw()
{
    sf::RenderWindow &window = *get_context()._window;

    window.setView( window.getDefaultView() );
    window.draw( _background_sprite );
    window.draw( _container );
}

/**
 * Realtime update the various components
 * @param dt The clock time
 * @return always true
 */
bool State_Menu::update( sf::Time dt )
{
    return true;
}

/**
 * Event handler for the Menu container
 * @param event The event that has been called
 * @return always false
 */
bool State_Menu::handle_event( const sf::Event &event )
{
    _container.handle_event( event );
    return false;
}
```

9.9.8 State_Pause.hpp

```cpp
#ifndef SOKOBAN_STATE_PAUSE_HPP
#define SOKOBAN_STATE_PAUSE_HPP

#include "State.hpp"
#include "../components/Container.hpp"

#include <SFML/Graphics/Sprite.hpp>
#include <SFML/Graphics/Text.hpp>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class State_Pause
                    : public State
            {
            public:
                State_Pause( State_Stack &stack, Context context );
                void draw() override;
                bool update( sf::Time dt ) override;
                bool handle_event( const sf::Event &event ) override;
            private:
                sf::Sprite _background_sprite; /** The background sprite */
                sf::Text _paused_text; /** The text stating that the game is paused */
                Container _container; /** Container containing the various components */
            };
        }
    }
}

#endif //SOKOBAN_STATE_PAUSE_HPP
```

9.9.9 State_Pause.cpp

```cpp
#include "State_Pause.hpp"

#include "../components/Button.hpp"
#include "../Utility.hpp"
#include "../../Resource_Holder.hpp"

#include <SFML/Graphics/Text.hpp>
#include <SFML/Graphics/RenderWindow.hpp>

#include <cmath>

using namespace sokoban::ui::gui;

/**
 * Default constructor for the Pause state
 * @param stack The stack containing the various states
 * @param context The context containing the various resources
 */
State_Pause::State_Pause( State_Stack &stack, State::Context context )
        : State( stack, context )
{
    sf::Texture &texture = context._textures->get( Textures::Title_Screen );
    sf::Font &font = context._fonts->get( Fonts::Rampart_One );
    sf::Vector2f view_size = context._window->getView().getSize();

    _background_sprite.setTexture( texture );
    Utility::center_origin( _background_sprite );
    _background_sprite.setPosition( view_size / 2.f );

    _paused_text.setFont( font );
    _paused_text.setString( "Paused" );
    _paused_text.setCharacterSize( 64.f );
    Utility::center_origin( _paused_text );
    _paused_text.setPosition( view_size.x / 2.f, view_size.y / 2.f - 200.f );

    auto resume_button = std::make_shared< Button >( context );
    resume_button->set_text( "Resume" );
    resume_button->set_callback( [ this ] ()
    {
        request_stack_pop();
    });

    auto settings_button = std::make_shared< Button >( context );
    settings_button->set_text( "Settings" );
    settings_button->set_callback( [ this ] ()
    {
        request_stack_push( States::Settings );
    });

    auto back_to_main_menu_button = std::make_shared< Button >( context );
    back_to_main_menu_button->set_text( "Main Menu" );
    back_to_main_menu_button->set_callback( [ this ] ()
    {
        request_stack_pop();
        request_stack_pop();
        request_stack_push( States::Menu );
    });

    settings_button->setPosition( context._window->getView().getSize() / 2.f );
    settings_button->setOrigin( 100.f, 25.f );
    resume_button->setPosition( settings_button->getPosition() - sf::Vector2f( 0, 100.f ) );
    resume_button->setOrigin( 100.f, 25.f );
    back_to_main_menu_button->setPosition( settings_button->getPosition() + sf::Vector2f( 0, 100.f ) );
    back_to_main_menu_button->setOrigin( 100.f, 25.f );

    _container.pack( resume_button );
    _container.pack( settings_button );
    _container.pack( back_to_main_menu_button );
}

/**
 * Visually display the various components that make up the state
 */
void State_Pause::draw()
{
    sf::RenderWindow &window = *get_context()._window;
    window.setView( window.getDefaultView() );
    window.draw( _paused_text );
    window.draw( _container );
}

/**
 * Realtime update the various components
 * @param dt The clock time
 * @return always true
 */
bool State_Pause::update( sf::Time dt )
{
    return true;
}

/**
 * Event handler for the current pause state
 * @param event The event that has been called
 * @return always false
 */
bool State_Pause::handle_event( const sf::Event &event )
{
    _container.handle_event( event );
    if( event.type == sf::Event::KeyPressed )
    {
        if( event.key.code == sf::Keyboard::Escape )
        {
            request_stack_pop();
        }
    }
    return false;
}
```

9.9.10 State_Settings.hpp

```
#ifndef SOKOBAN_STATE_SETTINGS_HPP
#define SOKOBAN_STATE_SETTINGS_HPP

#include "State.hpp"
#include "../components/Container.hpp"
#include "../components/Button.hpp"

#include <SFML/Graphics/Sprite.hpp>
#include <SFML/Graphics/Text.hpp>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class State_Settings
                    : public State
            {
            public:
                State_Settings( State_Stack &stack, Context context );
                void draw() override;
                bool update( sf::Time dt ) override;
                bool handle_event( const sf::Event &event ) override;
            private:
                sf::Sprite _background_sprite; /** The background sprite */
                sf::Text _settings_text; /** The text stating Settings */
                Container _container; /** The container containing the various components */
                Button::Ptr _music_volume_button; /** The music volume button */
                Button::Ptr _sound_effect_volume_button; /** The sound effect volume button */
                void change_sound_effect_volume( float val );
                void change_music_volume( float val );
            };
        }
    }
}

#endif //SOKOBAN_STATE_SETTINGS_HPP
```

9.9.11 State_Settings.cpp

```
#include "State_Settings.hpp"

#include "../components/Button.hpp"
#include "../Utility.hpp"
#include "../../Resource_Holder.hpp"
#include "../Music_Player.hpp"
#include "../Sound_Player.hpp"
#include "../components/Label.hpp"

#include <SFML/Graphics/Text.hpp>
#include <SFML/Graphics/RenderWindow.hpp>

#include <cmath>

using namespace sokoban::ui::gui;

namespace
{
    float tmp_music_vol;
    float tmp_sound_effect_vol;
}

/**
 * Default constructor for the Settings state
 * @param stack The stack containing the various states
 * @param context The context containing the globally accessible resources
 */
State_Settings::State_Settings( State_Stack &stack, Context context )
        : State( stack, context )
{
    sf::Texture &texture = context._textures->get( Textures::Title_Screen );
    sf::Font &font = context._fonts->get( Fonts::Rampart_One );
    sf::Vector2f view_size = context._window->getView().getSize();

    _background_sprite.setTexture( texture );
    Utility::center_origin( _background_sprite );
    _background_sprite.setPosition( view_size / 2.f );

    _music_volume_button = std::make_shared< Button >( context );
    _music_volume_button->set_text( std::to_string( ( int ) get_context()._music->get_volume() ) );
    _music_volume_button->set_toggle( true );

    _sound_effect_volume_button = std::make_shared< Button >( context );
    _sound_effect_volume_button->set_text( std::to_string( ( int ) get_context()._sounds->get_volume() ) );
    _sound_effect_volume_button->set_toggle( true );

    auto back_button = std::make_shared< Button >( context );
    back_button->set_text( "Back" );
    back_button->set_callback( [ this ] ()
    {
        request_stack_pop();
    });

    _sound_effect_volume_button->setPosition( context._window->getView().getSize() / 2.f );
    _sound_effect_volume_button->setOrigin( 100.f, 25.f );

    auto sound_effect_label = std::make_shared< Label >( "Sound Effect Volume", *get_context()._fonts, 24.f
    );
```

```cpp
    sound_effect_label->setPosition( _sound_effect_volume_button->getPosition() - sf::Vector2f( 0, 25.f ) );
    sound_effect_label->setOrigin( 100.f, 25.f );

    _music_volume_button->setPosition( _sound_effect_volume_button->getPosition() - sf::Vector2f( 0, 100.f )
      );
    _music_volume_button->setOrigin( 100.f, 25.f );

    auto music_label= std::make_shared< Label >( "Music Volume", *get_context()._fonts, 24.f );
    music_label->setPosition( _music_volume_button->getPosition() - sf::Vector2f( 0, 25.f ) );
    music_label->setOrigin( 100.f, 25.f );

    back_button->setPosition( _sound_effect_volume_button->getPosition() + sf::Vector2f( 0, 100.f ) );
    back_button->setOrigin( 100.f, 25.f );

    _container.pack( _music_volume_button );
    _container.pack( music_label );
    _container.pack( _sound_effect_volume_button );
    _container.pack( sound_effect_label );
    _container.pack( back_button );

    _settings_text.setFont( font );
    _settings_text.setString( "Settings" );
    _settings_text.setCharacterSize( 64.f );
    Utility::center_origin( _settings_text );
    _settings_text.setPosition( view_size.x / 2.f, 200.f );
}

/**
 * Visually display the various components that make up the Settings state
 */
void State_Settings::draw()
{
    sf::RenderWindow &window = *get_context()._window;
    window.setView( window.getDefaultView() );
    window.draw( _settings_text );
    window.draw( _container );
}

/**
 * Realtime update the various components
 * @param dt The clock time
 * @return always true
 */
bool State_Settings::update( sf::Time dt )
{
    _music_volume_button->set_text( std::to_string( ( int ) get_context()._music->get_volume() ) );
    _sound_effect_volume_button->set_text( std::to_string( ( int ) get_context()._sounds->get_volume() ) );
    return true;
}

/**
 * Event handler for the current state
 * @param event The event that has been called
 * @return always false
 */
bool State_Settings::handle_event( const sf::Event &event )
{
    bool is_setting_volume = false;

    if( _sound_effect_volume_button->is_active() )
    {
        is_setting_volume = true;
        if( event.type == sf::Event::KeyReleased )
        {
            if( event.key.code == sf::Keyboard::Enter )
            {
                _sound_effect_volume_button->deactivate();
            }
            else if( event.key.code == sf::Keyboard::Left || event.key.code == sf::Keyboard::Down )
            {
                change_sound_effect_volume( -5.f );
            }
            else if( event.key.code == sf::Keyboard::Right || event.key.code == sf::Keyboard::Up )
            {
                change_sound_effect_volume( 5.f );
            }
        }
    }
    if( _music_volume_button->is_active() )
    {
        is_setting_volume = true;
        if( event.type == sf::Event::KeyReleased )
        {
            if( event.key.code == sf::Keyboard::Enter )
            {
                _music_volume_button->deactivate();
            }
            else if( event.key.code == sf::Keyboard::Left || event.key.code == sf::Keyboard::Down )
            {
                change_music_volume( -5.f );
            }
            else if( event.key.code == sf::Keyboard::Right || event.key.code == sf::Keyboard::Up )
            {
                change_music_volume( 5.f );
            }
        }
    }
    if( _sound_effect_volume_button->is_selected() && !is_setting_volume )
    {
        if( event.type == sf::Event::KeyReleased )
        {
            if( event.key.code == sf::Keyboard::M )
            {
                float vol = get_context()._sounds->get_volume();
                if( vol > 0.f )
                {
                    tmp_sound_effect_vol = get_context()._sounds->get_volume();
                    vol = 0.f;
                }
                else
                {
                    vol = tmp_sound_effect_vol;
                }
                get_context()._sounds->set_volume( vol );
            }
            else if( event.key.code == sf::Keyboard::Left )
            {
                change_sound_effect_volume( -5.f );
            }
            else if( event.key.code == sf::Keyboard::Right )
            {
                change_sound_effect_volume( 5.f );
            }
        }
    }
    if( _music_volume_button->is_selected() && !is_setting_volume )
    {
        if( event.type == sf::Event::KeyReleased )
        {
            if( event.key.code == sf::Keyboard::M )
            {
                float vol = get_context()._music->get_volume();
                if( vol > 0.f )
                {
                    tmp_music_vol = get_context()._music->get_volume();
                    vol = 0.f;
                }
                else
                {
                    vol = tmp_music_vol;
                }
                get_context()._music->set_volume( vol );
            }
            else if( event.key.code == sf::Keyboard::Left )
            {
                change_music_volume( -5.f );
            }
            else if( event.key.code == sf::Keyboard::Right )
            {
                change_music_volume( 5.f );
            }
        }
    }

    if( !is_setting_volume )
    {
        _container.handle_event( event );
        if( event.type == sf::Event::KeyReleased )
        {
            if( event.key.code == sf::Keyboard::Escape )
            {
                request_stack_pop();
            }
        }
    }
    return false;
}

```

```cpp
/**
 * Function to change the sound effect volume
 * @param val The value that we wish to increase or decrease the volume with
 */
void State_Settings::change_sound_effect_volume( float val )
{
    float vol = get_context()._sounds->get_volume() + val;
    if( vol < 0.f )
    {
        vol = 100.f;
    }
    if( vol > 100.f )
    {
        vol = 0.f;
    }
    get_context()._sounds->set_volume( vol );
}

/**
 * Function to change the music volume
 * @param val The value we wish to increase or decrease the volume with
 */
void State_Settings::change_music_volume( float val )
{
    float vol = get_context()._music->get_volume() + val;
    if( vol < 0.f )
    {
        vol = 100.f;
    }
    else if( vol > 100.f )
    {
        vol = 0.f;
    }
    get_context()._music->set_volume( vol );
}
```

9.9.12 State_Stack.hpp

```cpp
#ifndef SOKOBAN_STATE_STACK_HPP
#define SOKOBAN_STATE_STACK_HPP

#include "State.hpp"
#include "State_Identifiers.hpp"
#include "../Resource_Identifiers.hpp"

#include <SFML/System/Time.hpp>
#include <SFML/System/NonCopyable.hpp>

#include <map>
#include <vector>
#include <utility>
#include <functional>

namespace sf
{
    class Event;
    class RenderWindow;
}

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class State_Stack
        : private sf::NonCopyable
            {
            public:
                /**
                 * The various actions possible with the stack
                 */
                enum Action
                {
                    Push, /** Push a state ahead */
                    Pop, /** Pop an existing state */
                    Clear /** Clear the entirety of the stack */
                };
                explicit State_Stack( State::Context context );
                template< typename T >
                    void register_state( States::ID state_id );
                void update( sf::Time dt );
                void draw();
                void handle_event( const sf::Event &event );
                void push_state( States::ID state_id );
                void pop_state();
                void clear_states();
                bool is_empty() const;
            private:
                State::Ptr create_state( States::ID state_id );
                void apply_pending_changes();
                struct Pending_Change
                {
                    explicit Pending_Change( Action action, States::ID state_id = States::None );
                    Action _action;
                    States::ID _state_id;
                };
                std::vector< State::Ptr > _stack; /** The stack containing the various states */
                std::vector< Pending_Change > _pending_list; /** The handler meant to change the state based on an action */
                State::Context _context; /** The globally accessible resources */
                std::map< States::ID, std::function< State::Ptr() > > _factories; /** Mapper that defines a State per ID */
            };

            /**
             * Dynamic initialisation of a state upon registration
             * @tparam T The nature of the state
             * @param state_id The ID that characterizes it
             */
            template< typename T >
                void State_Stack::register_state( States::ID state_id )
                {
                    _factories[ state_id ] = [ this ] ()
                    {
                        return State::Ptr( new T( *this, _context ) );
                    };
                }
        }
    }
}

#endif //SOKOBAN_STATE_STACK_HPP
```

9.9.13 State_Stack.cpp

```
#include "State_Stack.hpp"

#include <cassert>

using namespace sokoban::ui::gui;

/**
 * Default constructor for the Stack containing the various states
 * @param context Globally accessible context containing the various resources
 */
State_Stack::State_Stack( State::Context context )
: _stack()
, _pending_list()
, _context( context )
, _factories()
{
}

/**
 * Realtime update the various states within the stack
 * @param dt The clock time
 */
void State_Stack::update( sf::Time dt )
{
    for( auto itr = _stack.rbegin(); itr != _stack.rend(); ++itr )
    {
        if( !( *itr )->update( dt ) )
        {
            break;
        }
    }
    apply_pending_changes();
}

/**
 * Visually display the various states within the stack
 */
void State_Stack::draw()
{
    for( State::Ptr &state : _stack )
    {
        state->draw();
    }
}

/**
 * Event handler for each state present within the stack
 * @param event The event called
 */
void State_Stack::handle_event( const sf::Event &event )
{
    for( auto itr = _stack.rbegin(); itr != _stack.rend(); ++itr )
    {
        if( !( *itr )->handle_event( event ) )
        {
            break;
        }
    }
    apply_pending_changes();
}

/**
 * The state to push ahead of the stack
 * @param state_id The ID that characterizes the state
 */
void State_Stack::push_state( States::ID state_id )
{
    _pending_list.emplace_back( Push, state_id );
}

/**
 * Function that pops a state from within the stack
 */
void State_Stack::pop_state()
{
    _pending_list.emplace_back( Pop );
}

/**
 * Function that clears out the entirety of the states within the stack
 */
void State_Stack::clear_states()
{
    _pending_list.emplace_back( Clear );
}

/**
 * Verifier that checks whether the stack is empty or not
 * @return Whether the stack is empty or not
 */
bool State_Stack::is_empty() const
{
    return _stack.empty();
}

/**
 * Function that creates a state based on its ID
 * @param state_id The ID that characterizes the state
 * @return The State from its ID
 */
State::Ptr State_Stack::create_state( States::ID state_id )
{
    auto found = _factories.find( state_id );
    assert( found != _factories.end() );
    return found->second();
}

/**
 * Function that handles the various interaction called within the game
 */
void State_Stack::apply_pending_changes()
{
    for( Pending_Change change : _pending_list )
    {
        switch ( change._action )
        {
            case Push:
                _stack.push_back( create_state( change._state_id ) );
                break;
            case Pop:
                _stack.pop_back();
                break;
            case Clear:
                _stack.clear();
                break;
        }
    }
    _pending_list.clear();
}

/**
 * Default constructor for a possible pending change
 * @param action The action that has to be executed
 * @param state_id The ID that characterizes the state that calls the action
 */
State_Stack::Pending_Change::Pending_Change( State_Stack::Action action, States::ID state_id )
: _action( action )
, _state_id( state_id )
{
}
```

9.9.14 State_Title.hpp

```
#ifndef SOKOBAN_STATE_TITLE_HPP
#define SOKOBAN_STATE_TITLE_HPP

#include "State.hpp"

#include <SFML/Graphics/Text.hpp>
#include <SFML/Graphics/Sprite.hpp>

namespace sokoban
{
    namespace ui
    {
        namespace gui
        {
            class State_Title
                    : public State
            {
            public:
                State_Title( State_Stack &stack, Context context );
                void draw() override;
                bool update( sf::Time dt ) override;
                bool handle_event( const sf::Event &event ) override;
            private:
                sf::Sprite _background_sprite; /** The background sprite */
                sf::Text _text; /** The flashing text */
                sf::Text _title_text; /** The text in japanese stating Sokoban */
                sf::Text _title_sub_text; /** The sub text translating Sokoban from Japanese */
                bool _show_text; /** Bool that states whether the text has to be hidden or not */
                sf::Time _text_effect_time; /** Timer for the text to blink */
            };
        }
    }
}

#endif //SOKOBAN_STATE_TITLE_HPP
```

9.9.15 State_Title.cpp

```
#include "State_Title.hpp"

#include "../../../util/Logger.hpp"
#include "../Utility.hpp"
#include "../../Resource_Holder.hpp"

#include <SFML/Graphics/RenderWindow.hpp>

using namespace sokoban::ui::gui;
using namespace sokoban::util;

/**
 * Default constructor for the Title state
 * @param stack The stack containing the various states
 * @param context The context containing the various resources
 */
State_Title::State_Title( State_Stack &stack, State::Context context )
        : State( stack, context )
        , _text()
        , _title_text()
        , _show_text( true )
        , _text_effect_time( sf::Time::Zero )
{
    Logger::log( LoggerLevel::DEBUG, "Initializing Title Screen" );

```

```cpp
    Logger::log( LoggerLevel::DEBUG, "Loading Title Screen Texture" );
    _background_sprite.setTexture( context._textures->get( Textures::Title_Screen ) );
    Utility::center_origin( _background_sprite );
    _background_sprite.setPosition( context._window->getView().getSize() / 2.f );

    Logger::log( LoggerLevel::DEBUG, "Loading Title Screen Text" );

    _title_text.setFont( context._fonts->get( Fonts::Kodomo_Rounded ) );
    _title_text.setString( L"" );
    Utility::center_origin( _title_text );
    sf::Vector2f pos( context._window->getView().getSize() / 2.f );
    _title_text.setPosition( pos.x / 1.5f, 10.f );
    _title_text.setCharacterSize( 8 * 24 );
    _title_text.setFillColor( sf::Color::Cyan );

    _title_sub_text.setFont( context._fonts->get( Fonts::Kodomo_Rounded ) );
    _title_sub_text.setString( "Sokoban" );
    Utility::center_origin( _title_sub_text );
    _title_sub_text.setPosition( pos.x / 1.2f, _title_text.getPosition().y + 200.f );
    _title_sub_text.setCharacterSize( 4 * 24 );
    _title_sub_text.setFillColor( sf::Color::Cyan );

    _text.setFont( context._fonts->get( Fonts::Rampart_One ) );
    _text.setString( "Press any key to start" );
    Utility::center_origin( _text );
    _text.setPosition( pos.x, pos.y + 150.f );
    _text.setCharacterSize( 32 );
}

/**
 * Visually display the various components within the state
 */
void State_Title::draw()
{
    sf::RenderWindow &window = *get_context()._window;
    window.draw( _background_sprite );
    window.draw( _title_text );
    window.draw( _title_sub_text );
    if( _show_text )
    {
        window.draw( _text );
    }
}

/**
 * Realtime update each component
 * @param dt The clock time
 * @return always true
 */
bool State_Title::update( sf::Time dt )
{
    _text_effect_time += dt;
    if( _text_effect_time >= sf::seconds( 0.5f ) )
    {
        _show_text = !_show_text;
        _text_effect_time = sf::Time::Zero;
    }
    return true;
}

/**
 * Event handler for the current state
 * @param event The event to manage
 * @return always true
 */
bool State_Title::handle_event( const sf::Event &event )
{
    if( event.type == sf::Event::KeyReleased )
    {
        request_stack_pop();
        request_stack_push( States::Menu );
    }
    return true;
}
```