**Компьютерная академия «ШАГ»**

**Одесский филиал**

**Кафедра разработки программного обеспечения**

**КУРСОВОЙ ПРОЕКТ**

**“Ниндзя”**

**Студента группы ЕКО-15-П1  
Пырхавко М. А.**

**Научный руководитель:**

**Соколова М. О.**

**(Одесса) 2017**

Аннотация

Программа представляет собой компьютерную игру в жанре платформер.

Это игра, в которой игрок прыгает с платформа на платформу попутно уклоняясь от противников в попытке пройти уровень.

Победителем в игре становится игрок, добравшийся до флага.

Содержание

Оглавление

[Введение 2](#_Toc487665734)

[Техническое задание 2](#_Toc487665735)

[Использованные технологии 2](#_Toc487665736)

[Разработка структуры системы 2](#_Toc487665737)

[Разработка алгоритмов 2](#_Toc487665738)

[Разработка интерфейса 2](#_Toc487665739)

[Руководство пользователя 2](#_Toc487665740)

[Выводы 2](#_Toc487665741)

[Список использованной литературы 2](#_Toc487665742)

[Приложения 2](#_Toc487665743)

# Введение

Задачей курсовой работы было создания игрового приложения. В качестве игрового приложения мною была написана игра “Ниндзя” в жанре платформер. Для себя в данной курсовой я поставил цель применить на практике физический движок, и все навыки, полученные в компьютерной академии.

Платформер — это вид игр, где вы прыгаете по платформам собираете предметы, побеждаете монстров и проходите уровень за уровнем на пути к своей главной цели. Культовые представители жанра — Mario, Sonic, Donkey Kong.

Игра “Ниндзя” выполнены в жанре платфомер. Основной чертой игрового процесса является прыгание по платформам, собирание монет и уклонение от противников, всё это необходимо для завершения уровня. Предметы собираются простым прикосновением персонажа и для применения не требуют специальных действий со стороны игрока.

Игра характеризуются не реалистичностью и рисованной мультяшной графикой. Героем игры является мифический ниндзя перед которым стоит цель дойти до флага и завершить уровень.

Противники перемещаются по круговой дистанции и совершают повторяющиеся действия. В игре существует два вида противников это Демон и пила. Соприкосновение с противником отнимает ‘жизнь’. Если у героя остались ‘жизни’ герой переместится на ближайшую точку возрождения, если нет, то игра проиграна.

# Техническое задание

1. Настроить взаимодействие графической библиотеки и физического движка.
2. Базовый игровой объект.  
   Объект который представляет собой игровую сущность. Через него происходит настройка игрового объекта. Внутри налаживает взаимодействие физического движка и графической библиотеки.
3. Создать класс анимации.  
   Класс, который занимается загрузкой спрайта, разбиением спрайта на части и воспроизведением анимации.
4. Построить графический интерфейс.  
   Интерфейс, с помощью которого пользователь может запустить уровень, посмотреть правила игры и выйти из игры.
5. Настроить управление персонажем.  
   Управление персонажем ведется при помощи Клавиатуры.   
   Клавиши [W, D, SPACE] – используются для управления персонажем.  
   Для того чтобы открыть контекстное меню необходимо нажать клавишу escape.
6. Создать производные игровые объекты.  
   Объекты, которые конкретизируют базовые игровой объект. Являются производными от Базового игрового объекта.
7. Создать игровой мир.  
   Класс, который загружает и хранит игровые объекты. Управляет ими и обновляет их состояние.
8. Создать игровое Окно.  
   Класс, через который происходит управление камерой, отображение графики и воспроизведение звука.
9. Разобраться со статистикой героя.  
   Сбор и отображение количества жизней и монет героя, таймер.
10. Инициализировать игровой мир.  
    Создания экземпляров игровых объектов и наполнение игрового уровня.

# Использованные технологии

Для написания игры использовался объектно-ориентированный подход на базе языка С++.

Игра была разработана в интегрированной среде разработки visual studio community 2015. Для отображения графики использовалась графическая библиотека

SFML (Simple and Fast Multimedia Library), для расчёта физики объектов использовался физический движок BOX2D. Спрайты были созданы в графическом редакторе Photoshop.

Visual Studio 2015 — это интегрированная среда разработки с широкими возможностями для создания потрясающих приложений для Windows, Android и iOS, а также современных веб-приложений и облачных служб.

Box2D необходим для симуляции механики твёрдых тел с учётом ограничений. Движок может симулировать физические тела, составленные из выпуклых многоугольников, окружностей и линий. Тела могут быть связаны ограничителями в кинематические пары и подвергаться действию разных физических сил, таких как гравитация, трение и удар. Тела также могут подвергаться действию внутренних сил, таких как упругость.

SFML (Simple and Fast Multimedia Library — простая и быстрая мультимедиа библиотека) — свободная кроссплатформенная мультимедиа библиотека. Написана на [C++](https://ru.wikipedia.org/wiki/C%2B%2B), но доступна также для  [Python](https://ru.wikipedia.org/wiki/Python), [C](https://ru.wikipedia.org/wiki/C_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), [D](https://ru.wikipedia.org/wiki/D_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)),[Java](https://ru.wikipedia.org/wiki/Java),  [Ruby](https://ru.wikipedia.org/wiki/Ruby), [OCaml](https://ru.wikipedia.org/wiki/OCaml), [.Net](https://ru.wikipedia.org/wiki/.NET_Framework) и [Go](https://ru.wikipedia.org/wiki/Go). SFML предоставляет простой интерфейс для различных компонентов компьютера, чтобы облегчить развитие игры и мультимедийные приложения. Он состоит из пяти модулей system, window, graphics, audio и network.

# Разработка структуры системы

Класс GameWorld – класс, в котором инкапсулированы все игровые объекты в виде списка указателей на класс BaseGameObject. Также инкапсулирован класс Player и класс Window. В методе Launch происходит весь игровой процесс.

Класс BaseGameObject – класс базовый абстрактный класс игрового мира. Обладает телом, набором анимации и именем. Все остальные объекты игрового мира являются его потомками. Содержит в себе виртуальный метод update, в котором происходит отображения объекта и работа с коллизиями, который может быть переопределён в классах наследниках.

Классы Circle Box and Polygon класс наследники класс BaseGameObject. Эти классы определяют тело игрового объекта. Круг, Квадрат, Многоугольник.

Класс Player ялвяется производным от класса Box. Имеет своё переопределённый метод update в котором описана реакция игрока на столкновения с другими игровыми объектами, а также поведение игрока на пользовательский ввод.

Класс GameMenu занимается формированием меню, метод GetInput() рисует меню и ждёт пользовательского ввода. Возвращает индекс выбранного элемента меню.

Класс PictureFrames хранит в себе границы кадров. Метод GetFrame() возвращает границы кадра который нужно отрисовать.

Класс AnimatinFrames хранит в себе ассоциативный контейнер классов PictureFrames. Инкапсулирует в себе спрайт анимации игрового объекта.

Можно задать активную анимацию, метод draw обратиться к контейнеру кадров, получит границу кадра. Применит границу кадра к спрайту и отрисует нужный кадр из анимации в нужном месте.

Классы Platform and CirclePlatform от наследованы от класса Box and Circle соответственно. Обладают всеми свойствами классов родителей, а также инкапсулируют вектор задающий направление и скорость передвижения и время, после которого вектор перемещения будет перевёрнут в обратную сторону.

Класс Coin отнаследованный от класса Circle и класс Heart от наследованный от класса Box, не несут в себе ничего нового. Однако они в зависимости от параметров могут не сталкиваться с другими объектами.  
Схема UML находится в приложения

# Разработка алгоритмов

В основе Игры лежит несколько ключевых алгоритмов.

1. Алгоритм анимации - который делит картинки на отдельные спрайты и занимается подстановкой нужного кадра для отображения. За него отвечают классы
   1. Animation Frames – содержит информацию о конкретном кадре в рамках спрайта.
   2. Animation. Содержит коллекцию Animation Frames и по запросу подставляет нужный фрейм.
2. Алгоритм взаимодействия физического движка и графической библиотеки.  
   занимается переводом координат. Высчитывает, путём обращения к Физическому движку, где должна находится текстура и камера на игровом окне. За него отвечают:
   1. BaseGameObject.
   2. Window.
   3. Coord и производные от него классы.

# Разработка интерфейса

Игра обладает простым и понятным интерфейсом.

Основное меню:

1. Играть – запуск уровня
2. Правила – вывод правил на экран
3. Выход – завершение работы игры



Игровое меню:

1. Продолжить - вернуться в игру
2. Выход – покинуть уровень.



# Руководство пользователя

Для Запуска Игры выберите пункт “Play” в меню и нажмите Enter.

Игрок должен избегать врагов. Игрок может собирать монеты, а также использовать передвигаемые коробки чтобы допрыгнуть до не досягаемых вещей.

Управление

**Space – прыжок A – бежать влево D – бежать вправо**







Игровые Объекты

Маленькая платформа Коробка



Жизнь Монета



Враги

Демон Пила





# Выводы

В ходе написания курсовой работы была изучена графическая библиотека sfml и основой физического движка Box2d. Были улучшены навыки написания ООП программ.

Основой сложностью было совместить физический движок box2d с графической библиотекой sfml, что бы две различные библиотеки работали как одно целое. В будущем игра может быть легко развита, так как структура игры построена таким образом, что добавления новых возможностей не требует особого труда. В планах имеется добавления новых уровней в игру с новыми локациями, новыми более тяжёлыми врагами и новыми игровыми возможностями.

Разработка 2d довольно не сложный и увлекательный процесс, если ты изучил нужные технологии!

# Список использованной литературы

* <http://www.sfml-dev.org/tutorials/2.3/>
* <https://www.iforce2d.net/b2dtut/>
* <http://box2d.org/manual.pdf>

# Приложения

Приложение A: Исходный код

Animation.h

#pragma once

#include "AnimationFrames.h"

#include <map>

#include <string>

#include <iostream>

#include <memory>

#include "GameWindow.h"

class AnimationPack

{

public:

AnimationPack(std::string const & image\_path, WORD frame\_width, WORD frame\_height);

AnimationPack(std::shared\_ptr<sf::Texture> & texture, WORD frame\_width, WORD frame\_height);

AnimationPack(std::shared\_ptr<sf::Texture> & texture);

AnimationPack(std::string const & image\_path);

bool ReloadSprite(std::string const & image\_path);

void ReloadSprite(sf::Texture const & texture);

void Draw(Game::Window & window, Game::Position pos, float angle, bool flip = false);

Game::Size TextureSize();

WORD GetFrameWidth() const;

WORD GetFrameHeight() const;

bool SetActive(std::string name);

bool AddFrameSequence(std::string name, WORD pos\_x, WORD pos\_y, WORD count,

float speed, SpritesDir sprites\_dir, WORD frame\_width, WORD frame\_height);

bool AddFrame(std::string name, WORD pos\_x, WORD pos\_y, WORD frame\_width, WORD frame\_height);

bool AddFrameSequence(std::string name, WORD pos\_x, WORD pos\_y, WORD count,

float speed, SpritesDir sprites\_dir);

bool AddFrame(std::string name, WORD pos\_x, WORD pos\_y);

void SetLoop(std::string name, bool loop);

bool GetLoop(std::string name);

bool Replay(std::string);

bool Replay();

protected:

std::map<std::string, PictureFrames> \_frames;

WORD \_frame\_width, \_frame\_height;

float32 \_sprite\_scale;

std::string \_active;

std::shared\_ptr<sf::Texture> \_texture;

sf::Sprite \_sprite;

};

class CircleAnimationPack

:public AnimationPack

{

public:

CircleAnimationPack(std::string const & image\_path, WORD circle\_radius);

CircleAnimationPack(std::shared\_ptr<sf::Texture> & texture, WORD cirlce\_radius);

WORD GetTextureRadius() const;

};

Animation.cpp

#include "Animation.h"

#include "GameWindow.h"

#include <iostream>

AnimationPack::AnimationPack(std::string const & file\_path, WORD frame\_width, WORD frame\_height)

{

try

{

\_texture = std::make\_shared<sf::Texture>();

}

catch (std::bad\_alloc bad\_alloc)

{

std::clog << bad\_alloc.what();

return;

}

if (\_texture->loadFromFile(file\_path) == false)

{

std::clog << "can't load texture " << file\_path;

return;

}

\_frame\_width = frame\_width;

\_frame\_height = frame\_height;

\_sprite.setTexture(\*\_texture);

}

AnimationPack::AnimationPack(std::shared\_ptr<sf::Texture> & texture, WORD frame\_width, WORD frame\_height)

{

\_texture = texture;

\_frame\_width = frame\_width;

\_frame\_height = frame\_height;

\_sprite.setTexture(\*\_texture);

}

AnimationPack::AnimationPack(std::shared\_ptr<sf::Texture>& texture)

{

\_texture = texture;

\_frame\_width = \_texture->getSize().x;

\_frame\_height = \_texture->getSize().y;

\_sprite.setTexture(\*\_texture);

}

AnimationPack::AnimationPack(std::string const & image\_path)

{

try

{

\_texture = std::make\_shared<sf::Texture>();

}

catch (std::bad\_alloc bad\_alloc)

{

std::clog << bad\_alloc.what();

return;

}

if (\_texture->loadFromFile(image\_path) == false)

{

std::clog << "can't load texture " << image\_path;

return;

}

\_frame\_width = \_texture->getSize().x;

\_frame\_height = \_texture->getSize().y;

\_sprite.setTexture(\*\_texture);

}

bool AnimationPack::ReloadSprite(std::string const & image\_path)

{

sf::Texture texture;

if (texture.loadFromFile(image\_path) == false)

{

std::clog << "can't load texture " << image\_path;

return false;

}

\_sprite.setTexture(texture);

return true;

}

void AnimationPack::ReloadSprite(sf::Texture const & texture)

{

\_sprite.setTexture(texture);

}

void AnimationPack::Draw(Game::Window & window, Game::Position pos, float angle, bool flip)

{

sf::Vector2f origin;

if (\_active.length() == 0)

{

\_sprite.setTextureRect(sf::IntRect(0, 0, \_frame\_width, \_frame\_height));

\_sprite.setOrigin(\_frame\_width / 2, \_frame\_height / 2);

}

else if (\_frames[\_active].GetFramesCount() == 0)

{

std::cout << "can't draw empty frame";

return;

}

else {

PictureFrames frame = \_frames[\_active];

\_frames[\_active].SetGetFlip() = flip;

\_sprite.setTextureRect(\_frames[\_active].GetFrame());

\_sprite.setOrigin(\_frame\_width / 2, \_frame\_height / 2);

}

\_sprite.getTextureRect();

\_sprite.setRotation(angle);

\_sprite.setPosition(pos.GetPixelPos().x, pos.GetPixelPos().y);

window.Draw(\_sprite);

}

Game::Size AnimationPack::TextureSize()

{

return Game::Size();

}

bool AnimationPack::SetActive(std::string name)

{

if (\_frames.count(name) == 0)

{

std::clog << "no frames with such name: " << name;

return false;

}

\_active = name;

return true;

}

bool AnimationPack::AddFrameSequence(std::string name, WORD pos\_x, WORD pos\_y, WORD count, float speed, SpritesDir sprites\_dir, WORD frame\_width, WORD frame\_height)

{

if (\_active.length() == 0)

\_active = name;

switch (sprites\_dir)

{

case SpritesDir::DOWN:

if (pos\_x + frame\_width > \_texture->getSize().x) {

std::clog << "frame is beyond limits (width) " << name;

return false;

}

else if (pos\_y + frame\_height \* count > \_texture->getSize().y) {

std::clog << "frame is beyond limits (height) " << name;

return false;

}

break;

case SpritesDir::RIGHT :

if (pos\_y + frame\_height > \_texture->getSize().y) {

std::clog << "frame is beyond limits (height) " << name;

return false;

}

else if (pos\_x + frame\_width \* count > \_texture->getSize().x) {

std::clog << "frame is beyond limits (width) " << name;

return false;

}

break;

case SpritesDir::UP:

if (pos\_x + frame\_width > \_texture->getSize().x) {

std::clog << "frame is beyond limits (width) " << name;

return false;

}

else if (pos\_y - frame\_height \* count < 0) {

std::clog << "frame is beyond limits (height) " << name;

return false;

}

break;

case SpritesDir::LEFT:

if (pos\_y + frame\_height > \_texture->getSize().y) {

std::clog << "frame is beyond limits (height)" << name;

return false;

}

if (pos\_x - frame\_width \* count < 0) {

std::clog << "frame is beyond limits (width)" << name;

}

break;

}

\_frames[name] = PictureFrames(pos\_x, pos\_y, frame\_width, frame\_height, count, sprites\_dir, speed);

return true;

}

bool AnimationPack::AddFrame(std::string name, WORD pos\_x, WORD pos\_y, WORD frame\_width, WORD frame\_height)

{

if (\_active.length() == 0)

\_active = name;

if (pos\_x + frame\_width > \_texture->getSize().x) {

std::clog << "frame is beyond limits (width)";

return false;

}

if (pos\_y + frame\_height > \_texture->getSize().y) {

std::clog << "frame is beyond limits (height)";

return false;

}

\_frames[name].AddFrame(pos\_x, pos\_y, frame\_width, frame\_height);

return true;

}

bool AnimationPack::AddFrameSequence(std::string name, WORD pos\_x, WORD pos\_y, WORD count, float speed, SpritesDir sprites\_dir)

{

return AddFrameSequence(name,pos\_x, pos\_y, count, speed, sprites\_dir, \_frame\_width, \_frame\_height);

}

bool AnimationPack::AddFrame(std::string name, WORD pos\_x, WORD pos\_y)

{

return AddFrame(name, pos\_x, pos\_y, \_frame\_width, \_frame\_height);

}

void AnimationPack::SetLoop(std::string name, bool loop)

{

\_frames[name].SetGetLoop() = loop;

}

bool AnimationPack::GetLoop(std::string name)

{

return \_frames[name].SetGetLoop();

}

bool AnimationPack::Replay(std::string name)

{

if (\_frames.count(name) == 0)

{

std::clog << "no frames with such name: " << name;

\_frames[name].FromStart();

return false;

}

\_frames[name].FromStart();

return true;

}

bool AnimationPack::Replay()

{

if (\_frames.count(\_active) == 0)

{

std::clog << "empty animation";

return false;

}

\_frames[\_active].FromStart();

return true;

}

WORD AnimationPack::GetFrameWidth() const

{

return \_frame\_width;

}

WORD AnimationPack::GetFrameHeight() const

{

return \_frame\_height;

}

CircleAnimationPack::CircleAnimationPack(std::string const & image\_path, WORD circle\_radius)

: AnimationPack(image\_path, circle\_radius \* 2, circle\_radius \* 2)

{

}

CircleAnimationPack::CircleAnimationPack(std::shared\_ptr<sf::Texture> & texture, WORD cirlce\_radius)

: AnimationPack(texture, cirlce\_radius \* 2, cirlce\_radius \* 2)

{

}

WORD CircleAnimationPack::GetTextureRadius() const

{

return \_frame\_height / 2;

}

AnimationFrames.h

#pragma once

#include <SFML\Graphics.hpp>

#include <vector>

#include "Settings.h"

#include "Coord.h"

enum class SpritesDir

{

DOWN, RIGHT, UP, LEFT

};

class PictureFrames // storage of frame borders

{

public:

PictureFrames(WORD pos\_x, WORD pos\_y, WORD width, WORD height, WORD count, SpritesDir dir, float speed = 0.05);

PictureFrames();

~PictureFrames();

void AddFrame(float pos\_x, float pos\_y, WORD width, WORD height);

void AddFramesSequence(WORD pos\_x, WORD pos\_y, WORD width, WORD height, WORD count, SpritesDir dir);

sf::IntRect GetFrame(); // return the border of frame to draw

WORD GetFramesCount();

float GetSpeed();

void Flip();

void SetSpeed(float speed);

void FromStart();

bool & SetGetLoop();

bool & SetGetFlip();

private:

std::vector<sf::IntRect> \_frames, \_flip\_frames;

WORD \_frames\_count;

float \_current\_frame;

bool \_flip;

float \_speed;

bool \_loop;

};

AnimationFrames.cpp

#include "AnimationFrames.h"

#include <iostream>

PictureFrames::PictureFrames(WORD pos\_x, WORD pos\_y, WORD width, WORD height, WORD count, SpritesDir dir, float speed)

{

\_speed = speed;

\_frames\_count = 0;

AddFramesSequence(pos\_x, pos\_y, width, height, count, dir);

\_loop = true;

\_flip = false;

\_current\_frame = 0;

}

PictureFrames::PictureFrames()

{

\_loop = true;

\_speed = 0.05f;

\_current\_frame = 0;

\_frames\_count = 0;

}

PictureFrames::~PictureFrames()

{

}

void PictureFrames::AddFrame(float pos\_x, float pos\_y, WORD width, WORD height)

{

\_frames.push\_back(sf::IntRect(pos\_x, pos\_y, width, height));

\_frames.push\_back(sf::IntRect(pos\_x + width, pos\_y, -static\_cast<int>(width), height));

\_frames\_count++;

}

void PictureFrames::AddFramesSequence(WORD pos\_x, WORD pos\_y, WORD width, WORD height, WORD count, SpritesDir dir)

{

\_frames\_count += count;

switch (dir)

{

case SpritesDir::DOWN:

for (WORD i = 0; i < \_frames\_count; i++)

{

\_frames.push\_back(sf::IntRect(pos\_x, pos\_y + height \* i, width, height));

\_flip\_frames.push\_back(sf::IntRect(pos\_x + width, pos\_y + height \* i, -static\_cast<int>(width), height));

}

break;

case SpritesDir::RIGHT:

for (WORD i = 0; i < \_frames\_count; i++)

{

\_frames.push\_back(sf::IntRect(pos\_x + width \* i, pos\_y, width, height));

\_flip\_frames.push\_back(sf::IntRect(pos\_x + width \* i + width, pos\_y, -static\_cast<int>(width), height));

}

break;

case SpritesDir::UP:

for (WORD i = 0; i < \_frames\_count; i++)

{

\_frames.push\_back(sf::IntRect(pos\_x, pos\_y - height \* i, width, height));

\_flip\_frames.push\_back(sf::IntRect(pos\_x + width, pos\_y - height \* i, -static\_cast<int>(width), height));

}

break;

case SpritesDir::LEFT:

for (WORD i = 0; i < \_frames\_count; i++)

{

\_frames.push\_back(sf::IntRect(pos\_x - width \* i, pos\_y, width, height));

\_flip\_frames.push\_back(sf::IntRect(pos\_x - width \* i + width, pos\_y, -static\_cast<int>(width), height));

}

break;

}

}

sf::IntRect PictureFrames::GetFrame()

{

if (\_frames\_count == 0) {

return sf::IntRect(0, 0, 0, 0);

std::clog << "no frames";

}

int temp\_frame = \_current\_frame;

\_current\_frame += \_speed;

if (\_current\_frame > \_frames\_count - 1 && \_loop == false)

\_current\_frame = \_frames\_count - 1;

else if (\_current\_frame > \_frames\_count - 1)

\_current\_frame -= \_frames\_count;

return \_flip == false ? \_frames[static\_cast<WORD> (temp\_frame)] : \_flip\_frames[static\_cast<WORD> (temp\_frame)];

}

WORD PictureFrames::GetFramesCount()

{

return \_frames\_count;

}

float PictureFrames::GetSpeed()

{

return \_speed;

}

void PictureFrames::Flip()

{

\_flip == false ? \_flip = true : \_flip = false;

}

void PictureFrames::SetSpeed(float speed)

{

\_speed = speed;

}

void PictureFrames::FromStart()

{

\_current\_frame = 0;

}

bool & PictureFrames::SetGetLoop()

{

return \_loop;

}

bool & PictureFrames::SetGetFlip()

{

//std::cin.get();

return \_flip;

}

BaseGameObject.h

#pragma once

#include "Exceptions.h"

#include "Coord.h"

#include "Animation.h"

#include "GameWindow.h"

struct GameObjectParameters

{

GameObjectParameters(float pos\_px\_x\_, float pos\_px\_y\_, float density\_, float friction\_, float restitution\_, Game::BodyType body\_type);

Game::Position initial\_pos;

float dencity;

float friction;

float restitution;

Game::BodyType type;

};

class BaseGameObject // base object

{

public:

BaseGameObject(AnimationPack const & animation\_pack);

BaseGameObject(Game::Window & window, std::string name, AnimationPack const & animation, GameObjectParameters const & BoxParameteres, b2Shape const & shape);

BaseGameObject(BaseGameObject const && game\_obj);

BaseGameObject(BaseGameObject const & game\_obj) = delete;

BaseGameObject & operator=(BaseGameObject const & base\_game\_object) = delete;

bool Initialized();

virtual bool update(Game::Window & window); // update Game object

Game::Position GetPosition() const;

void move(Game::Position const &, float angle = 0);

const std::string & GetName();

virtual ~BaseGameObject() = 0;

protected:

//data

b2Body \* \_body;

AnimationPack \_animation\_pack;

std::string \_name;

//method(s)

void Initialize(Game::Window & window, std::string const & name, GameObjectParameters const & parameters, b2Shape const & shape, float angle = 0);

void draw(Game::Window & window, bool flip = false);

};

BaseGameObject.cpp

#include "BaseGameObject.h"

BaseGameObject::BaseGameObject(AnimationPack const & animation\_pack)

: \_animation\_pack(animation\_pack)

{

\_body = nullptr;

}

BaseGameObject::BaseGameObject(Game::Window & \_window,

std::string name,

AnimationPack const & animation,

GameObjectParameters const & BoxParameteres,

b2Shape const & shape)

: BaseGameObject(animation)

{

Initialize(

\_window,

name,

BoxParameteres,

shape

);

}

BaseGameObject::BaseGameObject(BaseGameObject const && game\_obj)

: \_animation\_pack(game\_obj.\_animation\_pack)

{

\_name = game\_obj.\_name;

\_body = game\_obj.\_body;

const\_cast<BaseGameObject &> (game\_obj).\_body = nullptr;

\_body->SetUserData(this);

}

bool BaseGameObject::Initialized()

{

return \_body ? true : false;

}

bool BaseGameObject::update(Game::Window & window)

{

draw(window);

return true;

}

Game::Position BaseGameObject::GetPosition() const

{

return Game::Position(

Game::CoordType::Physics,

\_body->GetPosition().x,

\_body->GetPosition().y

);

}

void BaseGameObject::move(Game::Position const & position, float angle)

{

\_body->SetTransform(

position.GetPhysicsPos(),

Game::Position::DegreesToRadians(angle)

);

}

const std::string & BaseGameObject::GetName()

{

return \_name;

}

BaseGameObject::~BaseGameObject()

{

if (\_body != nullptr)

\_body->GetWorld()->DestroyBody(\_body);

\_body = nullptr;

}

void BaseGameObject::Initialize(Game::Window & window,

std::string const & name,

GameObjectParameters const & parameters,

b2Shape const & shape,

float angle) // initialize Game object

{

if (\_body)

this->~BaseGameObject();

b2BodyDef body\_def;

body\_def.position.x =

parameters.initial\_pos.GetPhysicsPos().x;

body\_def.position.y =

parameters.initial\_pos.GetPhysicsPos().y;

switch (parameters.type) // Set Body Type

{

case Game::BodyType::Static: // object doesn't react on collisions and can't move

body\_def.type = b2\_staticBody;

break;

case Game::BodyType::Dynamic: // obect react on collisions and can move

body\_def.type = b2\_dynamicBody;

break;

case Game::BodyType::Kinematic :

body\_def.type = b2\_kinematicBody; // object doesn't react on collisions and can move

break;

}

\_name = name;

body\_def.angle = Game::Position::DegreesToRadians(angle);

body\_def.fixedRotation = false;

b2FixtureDef fixture\_def;

fixture\_def.density = parameters.dencity;

fixture\_def.friction = parameters.friction;

fixture\_def.restitution = parameters.restitution;

fixture\_def.shape = &shape; // setting the shape of fixture

\_body = window.GetWorld().CreateBody(&body\_def); // adding body to physics world

\_body->CreateFixture(&fixture\_def); // ading fixture to body

\_body->SetUserData(this);

}

void BaseGameObject::draw(Game::Window & window, bool flip)

{

Game::Position \* position = new Game::Position(

Game::CoordType::Physics,

\_body->GetPosition().x,

\_body->GetPosition().y

);

float angle = Game::Position::RadiansToDegress(\_body->GetAngle()) / 2;

\_animation\_pack.Draw(

window, \*position,

angle, flip

);

delete position;

}

GameObjectParameters::GameObjectParameters(float pos\_px\_x\_,

float pos\_px\_y\_,

float dencity\_,

float friction\_,

float restitution\_,

Game::BodyType body\_type)

{

initial\_pos.Set(Game::CoordType::Pixel, pos\_px\_x\_, pos\_px\_y\_);

dencity = dencity\_;

friction = friction\_;

restitution = restitution\_;

type = body\_type;

}

Box.h

#pragma once

#include "BaseGameObject.h"

class Box :

public BaseGameObject

{

public:

Box(Game::Window & window, std::string name, GameObjectParameters const & parameters, AnimationPack const & animations);

};

Box.cpp

#include "Box.h"

Box::Box(Game::Window & window, std::string name, GameObjectParameters const & parameters, AnimationPack const & animations)

: BaseGameObject(animations)

{

b2PolygonShape shape;

shape.SetAsBox((static\_cast<float32> (animations.GetFrameWidth()) / 2) \* GameParameters::scaling, (static\_cast<float32> (animations.GetFrameHeight()) / 2) \* GameParameters::scaling);

Initialize(window, name, parameters, shape);

}

Circle.h

#pragma once

#include "BaseGameObject.h"

class Circle

: public BaseGameObject

{

public:

Circle(Game::Window & window, std::string name, GameObjectParameters const & parameters, CircleAnimationPack const & animations);

};

Circle.cpp

#include "Circle.h"

Circle::Circle(Game::Window & world, std::string name, GameObjectParameters const & parameters, CircleAnimationPack const & animations)

: BaseGameObject(animations)

{

b2CircleShape shape;

shape.m\_radius = static\_cast<float32> (animations.GetTextureRadius()) \* GameParameters::scaling;

Initialize(world, name, parameters, shape);

}

CircleEnemy.h

#pragma once

#include "CirclePlatform.h"

class CircleEnemy :

public CirclePlatform

{

public:

CircleEnemy(Game::Window & window, Game::Position const & pos, CircleAnimationPack const & animations, float x\_velocity, float y\_velocity, float loop\_time, bool sensor, int linearVelocity = 0);

};

CircleEnemy.cpp

#include "CircleEnemy.h"

CircleEnemy::CircleEnemy(Game::Window & window, Game::Position const & pos, CircleAnimationPack const & animations, float x\_velocity, float y\_velocity, float loop\_time, bool sensor, int linearVelocity)

:CirclePlatform(window, pos, animations, x\_velocity, y\_velocity, loop\_time)

{

\_name = "enemy";

\_body->SetBullet(true);

\_body->SetAngularVelocity(linearVelocity);

\_body->GetFixtureList()->SetSensor(sensor);

}

CirclePlatform.h

#pragma once

#include "Circle.h"

class CirclePlatform :

public Circle

{

public:

CirclePlatform(Game::Window & window, Game::Position const & pos, CircleAnimationPack const & animations, float x\_velocity, float y\_velocity, float loop\_time);

bool virtual update(Game::Window & windoow);

private:

sf::Time \_timer;

b2Vec2 \_velocity;

float \_time\_to\_change\_dir;

};

CirclePlatform.cpp

#include "CirclePlatform.h"

CirclePlatform::CirclePlatform(Game::Window & window,

Game::Position const & pos,

CircleAnimationPack const & animations,

float x\_velocity,

float y\_velocity,

float loop\_time)

:Circle(window,

"platform",

GameObjectParameters(

pos.GetPixelPos().x,

pos.GetPixelPos().y,

1.f, 0.09f, 0.01f,

Game::BodyType::Kinematic

),

animations)

{

\_timer = window.GetTime();

\_time\_to\_change\_dir = loop\_time;

\_velocity = b2Vec2(x\_velocity, y\_velocity);

\_body->SetLinearVelocity(\_velocity);

\_velocity.x \*= -1;

\_velocity.y \*= -1;

}

bool CirclePlatform::update(Game::Window & window)

{

if (window.GetTime().asSeconds() - \_timer.asSeconds() > \_time\_to\_change\_dir)

{

\_timer = window.GetTime();

\_body->SetLinearVelocity(b2Vec2(0, 0));

\_body->SetLinearVelocity(\_velocity);

\_velocity.x \*= -1;

\_velocity.y \*= -1;

}

draw(window, false);

return true;

}

Coin.h

#pragma once

#include "Circle.h"

class Coin :

public Circle

{

public:

Coin(Game::Window & window, Game::Position const & pos, CircleAnimationPack const & coin\_animations, bool static\_ = true);

virtual bool update(Game::Window & window);

private:

bool \_destruct;

};

Coin.cpp

#include "Coin.h"

Coin::Coin(Game::Window & window, Game::Position const & pos, CircleAnimationPack const & coin\_animations, bool static\_)

:Circle(window, "coin", GameObjectParameters(pos.GetPixelPos().x, pos.GetPixelPos().y, 1.f, 0.09f, 0.1f, static\_ ? Game::BodyType::Static : Game::BodyType::Dynamic), coin\_animations)

{

if (static\_)

\_body->GetFixtureList()->SetSensor(true);

\_destruct = false;

}

bool Coin::update(Game::Window & window)

{

if (\_destruct == true)

return false;

for (b2ContactEdge \* contact = \_body->GetContactList(); contact != nullptr; contact = contact->next)

{

std::string object\_a = static\_cast<BaseGameObject \*>(contact->contact->GetFixtureA()->GetBody()->GetUserData())->GetName();

std::string object\_b = static\_cast<BaseGameObject \*>(contact->contact->GetFixtureB()->GetBody()->GetUserData())->GetName();

if (object\_a == "player") {

\_destruct = true;

}

}

draw(window, false);

return true;

}

Coord.h

#pragma once

#include <SFML\Graphics.hpp>

#include <Box2D\Box2D.h>

#include "Settings.h"

namespace Game

{

enum class CoordType

{

Pixel, Physics

};

enum class BodyType

{

Static, Dynamic, Kinematic

};

class Vec

{

public:

Vec();

Vec(CoordType coord\_type, float x\_, float y\_);

Vec(sf::Vector2f vector);

Vec(b2Vec2 vector);

void Set(CoordType pos\_type, float x\_, float y\_);

sf::Vector2f GetPixelPos() const;

b2Vec2 GetPhysicsPos() const;

protected:

float \_x;

float \_y;

};

class Position

:public Vec

{

public:

Position();

Position(CoordType coord\_type, float x\_, float y\_);

Position(sf::Vector2f vector);

Position(b2Vec2 vector);

public:

static float DegreesToRadians(float degrees);

static float RadiansToDegress(float radians);

};

class Size

: public Vec

{

public:

Size(CoordType coord\_type, float x\_, float y\_);

float GetX();

float GetY();

Size();

Size(float x\_, float y\_);

};

}

Coord.cpp

#include "Coord.h"

Game::Vec::Vec()

{

\_x = \_y = 0;

}

Game::Vec::Vec(CoordType pos\_type, float x\_, float y\_)

{

if (pos\_type == CoordType::Pixel)

{

\_x = x\_;

\_y = y\_;

}

else

{

\_x = x\_ / GameParameters::scaling;

\_y = y\_ / GameParameters::scaling;

}

}

Game::Vec::Vec(sf::Vector2f vector)

{

\_x = vector.x;

\_y = vector.y;

}

Game::Vec::Vec(b2Vec2 vector)

{

\_x = vector.x / GameParameters::scaling;

\_y = vector.y / GameParameters::scaling;

}

void Game::Vec::Set(CoordType pos\_type, float x\_, float y\_)

{

if (pos\_type == CoordType::Physics) {

\_x = x\_ / GameParameters::scaling;

\_y = y\_ / GameParameters::scaling;

}

else {

\_x = x\_;

\_y = y\_;

}

}

Game::Position::Position()

:Vec()

{

}

Game::Position::Position(CoordType coord\_type, float x\_, float y\_)

:Vec(coord\_type, x\_, y\_)

{

}

Game::Position::Position(sf::Vector2f vector)

:Vec(vector)

{

}

Game::Position::Position(b2Vec2 vector)

:Vec(vector)

{

}

float Game::Position::DegreesToRadians(float degrees)

{

return degrees \* (b2\_pi / 180);

}

float Game::Position::RadiansToDegress(float radians)

{

return radians \* (180 / b2\_pi);

}

sf::Vector2f Game::Vec::GetPixelPos() const

{

return sf::Vector2f(\_x, \_y);

}

b2Vec2 Game::Vec::GetPhysicsPos() const

{

return b2Vec2(\_x \* GameParameters::scaling, \_y \* GameParameters::scaling);

}

Game::Size::Size(CoordType coord\_type, float x\_, float y\_)

:Vec(coord\_type, x\_, y\_)

{

}

float Game::Size::GetX()

{

return \_x;

}

float Game::Size::GetY()

{

return \_y;

}

Game::Size::Size()

: Vec()

{

}

Game::Size::Size(float x\_, float y\_)

: Vec(CoordType::Pixel, x\_, y\_)

{

}

EndGameObject.h

#pragma once

#include "Box.h"

class EndGameObject

: public Box

{

public:

EndGameObject(Game::Window & window, Game::Position const & pos, AnimationPack const & coin\_animations);

};

EndGameObject.cpp

#include "EndGameObject.h"

EndGameObject::EndGameObject(Game::Window & window, Game::Position const & pos, AnimationPack const & coin\_animations)

:Box(window, "end", GameObjectParameters(pos.GetPixelPos().x, pos.GetPixelPos().y, 0, 0, 0, Game::BodyType::Static), coin\_animations)

{

\_body->GetFixtureList()->SetSensor(true);

}

EnemyObject.h

#pragma once

#include "Platform.h"

class EnemyObject :

public Platform

{

public:

EnemyObject(Game::Window & window, Game::Position const & pos, AnimationPack const & animations, float x\_velocity, float y\_velocity, float loop\_time, bool sensor);

};

EnemyObject.cpp

#include "EnemyObject.h"

EnemyObject::EnemyObject(Game::Window & window, Game::Position const & pos, AnimationPack const & animations, float x\_velocity, float y\_velocity, float loop\_time, bool sensor)

:Platform(window, pos, animations, x\_velocity, y\_velocity, loop\_time)

{

\_name = "enemy";

\_body->SetBullet(true);

\_body->GetFixtureList()->SetSensor(sensor);

}

Event.h

#pragma once

#include<SFML\Graphics.hpp>

sf::Event ev;

namespace Game

{

enum EventType

{

KeyPressed, Resiezed, RightClick, LeftClick, WheelMoved, Exit

};

class Event

{

public:

Event(EventType ev);

private:

EventType \_type;

};

}

Event.cpp

#include "Event.h"

namespace Game

{

Event::Event(EventType ev)

{

\_type = ev;

}

}

GameObjects.h

#pragma once

#include "Circle.h"

#include "Box.h"

#include "Polygon.h"

#include "Platform.h"

#include "Coin.h"

#include "Heart.h"

#include "EnemyObject.h"

#include "GamePoint.h"

#include "CirclePlatform.h"

#include "CircleEnemy.h"

GamePoint.h

#pragma once

#include "Box.h"

class GamePoint :

public Box

{

public:

GamePoint(Game::Window & window, Game::Position const & position, AnimationPack const & animation);

};

GamePoint.cpp

#include "GamePoint.h"

GamePoint::GamePoint(Game::Window & window, Game::Position const & pos, AnimationPack const & animations)

:Box(window, "check\_point", GameObjectParameters(pos.GetPhysicsPos().x, pos.GetPhysicsPos().y, 0.f, 0.f, 0.f, Game::BodyType::Static), animations)

{

\_body->GetFixtureList()->SetSensor(true);

}

GameWindow.h

#pragma once

#include "Coord.h"

#include <SFML\Audio.hpp>

namespace Game

{

enum class KEYBOARD\_ACTION

{

Up, Down, Right, Left, Jump, Inaction, Enter, Exit

};

enum class MOUSE\_ACTION

{

RightButton, LeftButton, WheelUp, WheelDown, MiddleButton, Inaction

};

class Window

{

public:

Window(WORD width, WORD height, std::string title, float32 x\_gravity = 0, float32 y\_gravity = 9.8);

Window(WORD width, WORD height, std::string title, WORD frame\_rate\_limit, float32 x\_gravity = 0, float32 y\_gravity = 9.8);

Game::Position GetPostion();

Game::Size GetSize();

void Draw(sf::Sprite const & sprite);

b2World & GetWorld();

KEYBOARD\_ACTION GetKeyboardAction();

Position GetMousePosition();

MOUSE\_ACTION GetMouseAction();

void MoveCamera(Game::Position position);

void MoveCamera(float pos\_px\_x, float pos\_px\_y);

void Clear();

bool PlaySound(std::string soundName);

void Close();

void SetFrameRateLimit(WORD limit);

void Display();

void DrawBackground();

void Step();

void UpdateView(Game::Size);

void SetGravity(float32 x\_gravity, float32 y\_gravity);

sf::Time GetTime();

void SetBackGround(std::string path);

void PrintString(sf::Text const & text);

void GameEnd();

bool IsOpen();

Game::Position GetCameraLeftTopPosition() const;

bool PollEvent(sf::Event & event);

private:

sf::RenderWindow \_window;

b2World \_world;

float32 \_world\_step;

int \_wheel\_delta;

sf::Clock \_clock;

sf::Texture \_back\_ground\_texture;

sf::Sprite \_back\_ground\_sprite;

std::map<std::string, sf::SoundBuffer> \_soundBufferMap;

sf::Sound \_sound;

};

}

GameWindow.cpp

#include "GameWindow.h"

#include <iostream>

namespace Game

{

Window::Window(WORD width, WORD height, std::string title, float32 x\_gravity, float32 y\_gravity)

:\_world(b2Vec2(x\_gravity, y\_gravity))

{

\_window.create(sf::VideoMode(width, height), title);

\_world\_step = 1.f / 60.f;

sf::Image Icon;

if (Icon.loadFromFile("resources\\ninja.png"))

\_window.setIcon(32, 32, Icon.getPixelsPtr());

sf::SoundBuffer soundBuffer = sf::SoundBuffer();

soundBuffer.loadFromFile("audio\\Coins.wav");

\_soundBufferMap["coin"] = soundBuffer;

soundBuffer.loadFromFile("audio\\1up.wav");

\_soundBufferMap["heart"] = soundBuffer;

soundBuffer.loadFromFile("audio\\jump2.wav");

\_soundBufferMap["jump"] = soundBuffer;

//\_soundBufferMap[0].loadFromFile("audio\\Coins.wav");

}

Window::Window(WORD width, WORD height, std::string title, WORD frame\_rate\_limit, float32 x\_gravity, float32 y\_gravity)

:Window(width, height, title, x\_gravity, y\_gravity)

{

\_window.setFramerateLimit(frame\_rate\_limit);

\_world\_step = 1.f / frame\_rate\_limit;

}

Game::Position Window::GetPostion()

{

return Game::Position(Game::CoordType::Pixel, \_window.getPosition().x, \_window.getPosition().y);

}

Game::Size Window::GetSize()

{

return Game::Size(Game::CoordType::Pixel, \_window.getSize().x, \_window.getSize().y);

}

void Window::Draw(sf::Sprite const & sprite)

{

\_window.draw(sprite);

}

b2World & Window::GetWorld()

{

return \_world;

}

KEYBOARD\_ACTION Window::GetKeyboardAction()

{

if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up) ||

sf::Keyboard::isKeyPressed(sf::Keyboard::W))

return KEYBOARD\_ACTION::Up;

else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space))

return KEYBOARD\_ACTION::Jump;

else if (sf::Keyboard::isKeyPressed(sf::Keyboard::S) ||

sf::Keyboard::isKeyPressed(sf::Keyboard::Down))

return KEYBOARD\_ACTION::Down;

else if (sf::Keyboard::isKeyPressed(sf::Keyboard::D) ||

sf::Keyboard::isKeyPressed(sf::Keyboard::Right))

return KEYBOARD\_ACTION::Right;

else if (sf::Keyboard::isKeyPressed(sf::Keyboard::A) ||

sf::Keyboard::isKeyPressed(sf::Keyboard::Left))

return KEYBOARD\_ACTION::Left;

else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape))

return KEYBOARD\_ACTION::Exit;

else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Return))

return KEYBOARD\_ACTION::Enter;

else

return KEYBOARD\_ACTION::Inaction;

}

Position Window::GetMousePosition()

{

return Game::Position(

Game::CoordType::Pixel,

sf::Mouse::getPosition(\_window).x,

sf::Mouse::getPosition(\_window).y

);

}

MOUSE\_ACTION Window::GetMouseAction()

{

if (\_wheel\_delta != 0)

{

int temp\_value = \_wheel\_delta;

\_wheel\_delta = 0;

return temp\_value < 0 ? MOUSE\_ACTION::WheelDown : MOUSE\_ACTION::WheelUp;

}

if (\_wheel\_delta > 0)

{

\_wheel\_delta = 0;

return MOUSE\_ACTION::WheelUp;

}

else if (\_wheel\_delta < 0)

{

\_wheel\_delta = 0;

return MOUSE\_ACTION::WheelDown;

}

else if (sf::Mouse::isButtonPressed(sf::Mouse::Right))

return MOUSE\_ACTION::RightButton;

else if (sf::Mouse::isButtonPressed(sf::Mouse::Left))

return MOUSE\_ACTION::LeftButton;

else if (sf::Mouse::isButtonPressed(sf::Mouse::Button::Middle))

return MOUSE\_ACTION::MiddleButton;

else

return MOUSE\_ACTION::Inaction;

}

void Window::MoveCamera(Game::Position position)

{

sf::View view(sf::FloatRect(sf::Vector2f(position.GetPixelPos().x, position.GetPixelPos().y), sf::Vector2f(\_window.getSize())));

\_back\_ground\_sprite.setPosition(position.GetPixelPos().x, position.GetPixelPos().y);

\_window.setView(view);

}

void Window::MoveCamera(float pos\_px\_x, float pos\_px\_y)

{

sf::View view(sf::FloatRect(sf::Vector2f(pos\_px\_x, pos\_px\_y), sf::Vector2f(\_window.getSize())));

\_back\_ground\_sprite.setPosition(pos\_px\_x, pos\_px\_y);

\_window.setView(view);

}

bool Window::IsOpen()

{

if (\_window.isOpen() == false)

return false;

sf::Event \_event;

while (\_window.pollEvent(\_event))

{

switch (\_event.type)

{

case sf::Event::Closed:

Close();

exit(0);

case sf::Event::MouseWheelMoved:

this->\_wheel\_delta = \_event.mouseWheel.delta;

break;

default:

break;

}

}

return true;

}

Game::Position Window::GetCameraLeftTopPosition() const

{

return Game::Position(

Game::CoordType::Pixel,

\_window.getView().getCenter().x - (\_window.getSize().x / 2),

\_window.getView().getCenter().y - (\_window.getSize().y / 2)

);

}

bool Window::PollEvent(sf::Event & event)

{

if (!\_window.pollEvent(event))

return false;

switch (event.type)

{

case sf::Event::Resized:

{

UpdateView(

Game::Size(event.size.width, event.size.height)

);

break;

}

default:

break;

}

return true;

}

void Window::Clear()

{

\_window.clear();

}

bool Window::PlaySound(std::string soundName)

{

if (\_soundBufferMap.count(soundName) != 1)

{

return false;

}

\_sound.setBuffer(\_soundBufferMap[soundName]);

\_sound.play();

return true;

}

void Window::Close()

{

\_window.close();

}

void Window::SetFrameRateLimit(WORD limit)

{

\_window.setFramerateLimit(limit);

\_world\_step = 1.f / limit;

}

void Window::Display()

{

\_window.display();

\_window.clear();

DrawBackground();

}

void Window::DrawBackground()

{

float xScale =

GetSize().GetPixelPos().x

/

\_back\_ground\_sprite.getGlobalBounds().width;

float yScale =

GetSize().GetPixelPos().y

/

\_back\_ground\_sprite.getGlobalBounds().height;

\_back\_ground\_sprite.scale(xScale, yScale);

\_window.draw(\_back\_ground\_sprite);

}

void Window::Step()

{

\_world.Step(\_world\_step, 8, 3);

}

void Window::UpdateView(Game::Size newSize)

{

sf::FloatRect rect = sf::FloatRect(

0, 0,

newSize.GetX(),

newSize.GetY()

);

\_window.setView(sf::View(rect));

MoveCamera(0, 0);

}

void Window::SetGravity(float32 x\_gravity, float32 y\_gravity)

{

\_world.SetGravity(b2Vec2(x\_gravity, y\_gravity));

}

sf::Time Window::GetTime()

{

return \_clock.getElapsedTime();

}

void Window::SetBackGround(std::string path)

{

\_back\_ground\_texture.loadFromFile(path);

\_back\_ground\_texture.setSmooth(true);

\_back\_ground\_sprite.setTexture(\_back\_ground\_texture, true);

}

void Window::PrintString(sf::Text const & text)

{

\_window.draw(text);

}

void Window::GameEnd()

{

\_window.clear();

}

}

GameWorld.h

#pragma once

#include <SFML\Graphics.hpp>

#include <Box2D\Box2D.h>

#include <list>

#include <memory>

#include "Settings.h"

#include "GameObjects.h"

#include "GameWindow.h"

#include "Player.h"

class GameWorld

{

public:

GameWorld(); // default constructor sets the World gravity (0, 9.8)

GameWorld(std::string title, float X\_gravity, float Y\_gravity);

void InstructionsScreen();

void Launch();

~GameWorld();

private:

std::list<BaseGameObject \*> \_entities;

void LastScreen();

void GameClosing();

void Initialize();

Game::Window \_window;

Player \_player;

};

#include "GameWorld.h"

#include "Menu.h"

#include "EndGameObject.h"

GameWorld::GameWorld()

:GameWorld("Game", 0.f, 9.8f)

{

}

GameWorld::GameWorld(std::string title, float X\_gravity, float Y\_gravity)

: \_window(GameParameters::WindowWidth, GameParameters::WindowHeight, title, X\_gravity, Y\_gravity),

\_player(\_window, Game::Position(Game::CoordType::Pixel, 100, 100))

{

\_window.SetFrameRateLimit(60);

//\_entities.push\_back(Player(\_window));

}

void GameWorld::InstructionsScreen()

{

sf::String rules =

"Use [WASD] or Arrows to Control Character.\n\Overcome obstacles and get to the flag";

sf::Font font;

font.loadFromFile("fonts\\AdobeFanHeitiStd-Bold.otf");

sf::Text rulesText(rules, font, 30);

sf::Event ev;

while (true)

{

while (\_window.PollEvent(ev))

{

switch (ev.type)

{

case sf::Event::EventType::KeyPressed:

switch (ev.key.code)

{

case sf::Keyboard::Escape:

case sf::Keyboard::Return:

return;

}

break;

case sf::Event::MouseButtonPressed:

switch (ev.mouseButton.button)

{

case sf::Mouse::Left:

return;

}

break;

case sf::Event::Closed:

GameClosing();

break;

default:

break;

}

}

rulesText.setPosition(

\_window.GetSize().GetX() / 2 - rulesText.getGlobalBounds().width / 2,

\_window.GetSize().GetY() / 2 - rulesText.getGlobalBounds().height / 2

);

\_window.PrintString(rulesText);

\_window.Display();

}

}

void GameWorld::Launch()

{

GameMenu menu(\_window, 70, { "Play", "Rules" ,"Exit" }, 40);

do

{

\_window.SetBackGround("textures//BG4.jpg");

for (size\_t input = menu.GetInput(\_window); input != 0; input = menu.GetInput(\_window))

{

switch (input)

{

case 1:

InstructionsScreen();

break;

case 2:

case -1:

GameClosing();

}

}

Initialize();

\_window.Step();

while (\_window.IsOpen())

{

for (std::list<BaseGameObject \*>::iterator iterator = \_entities.begin(); iterator != \_entities.end();) // cycle walks through all game\_elements and apply update

{

if ((\*iterator)->update(\_window) == false) // if update fails (object is expire), the object is removed from array

{

delete \*iterator;

std::list<BaseGameObject \*>::iterator iterator\_copy = iterator;

iterator++;

\_entities.erase(iterator\_copy);

continue;

}

iterator++;

}

if (\_player.update(\_window) == false)

break;

\_window.Display();

\_window.Step();

}

for (std::list<BaseGameObject \*>::iterator iterator = \_entities.begin(); iterator != \_entities.end(); iterator++)

delete \*iterator;

\_entities.clear();

\_window.GameEnd();

\_window.MoveCamera(Game::Position(Game::CoordType::Pixel, 0, 0));

\_window.Display();

LastScreen();

}

while (true);

}

void GameWorld::LastScreen()

{

if (\_player.isWin())

\_window.PlaySound("heart");

sf::Event event;

bool on = true;

while (\_window.PollEvent(event)); // clear events stack

while (on)

{

while (\_window.PollEvent(event))

{

switch (event.type)

{

case sf::Event::MouseButtonPressed:

switch (event.mouseButton.button)

{

case sf::Mouse::Left:

on = false;

break;

default:

break;

}

break;

case sf::Event::KeyPressed:

switch (event.key.code)

{

case sf::Keyboard::Return:

case sf::Keyboard::Escape:

on = false;

break;

default:

break;

}

break;

case sf::Event::Closed:

GameClosing();

break;

default:

break;

}

}

\_window.Display();

\_player.LastScreen(\_window);

}

}

void GameWorld::GameClosing()

{

exit(0);

}

GameWorld::~GameWorld()

{

for (std::list<BaseGameObject \*>::iterator iterator = \_entities.begin(); iterator != \_entities.end(); iterator++) // cycle walks through all game\_elements

delete \*iterator;

}

void GameWorld::Initialize()

{

\_player.Reset(

Game::Position(Game::CoordType::Pixel, 30, 300),

\_window.GetTime().asSeconds()

);

\_player.AddSpawnPoints(

{

Game::Position(Game::CoordType::Pixel, 3500, 800),

Game::Position(Game::CoordType::Pixel, 5397, 300)

}

);

\_window.SetBackGround("textures\\BG4.jpg");

AnimationPack box\_animation("textures\\box.png", 128, 128);

AnimationPack platform\_anim("textures\\StoneBlock.png");

AnimationPack heart\_anim("textures\\heart.png", 99, 90);

AnimationPack big\_platform\_anim("textures\\big\_platform.png");

CircleAnimationPack coin\_animation("textures\\coin.png", 62);

CircleAnimationPack saw\_anim("textures\\saw.png", 70);

coin\_animation.AddFrame("inaction", 0, 0);

coin\_animation.AddFrameSequence("inaction", 0, 0, 10, 0.1f, SpritesDir::RIGHT, 122, 122);

CircleAnimationPack enemy\_anim("textures\\enemy.png", 70);

AnimationPack crate("textures\\crate.png");

AnimationPack end\_anim("textures\\flag.png");

//adding Game objects to the list

//boxes

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(30, 500, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(480, 500, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 480, 300), coin\_animation, false));

\_entities.push\_back(new Heart(\_window, Game::Position(Game::CoordType::Pixel, 980, 400), heart\_anim, false));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(980, 500, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(1280, 500, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(1280, 350, 1.0f, 1.f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 1280, 50), coin\_animation, false));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(1450, 500, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(1450, 350, 1.0f, 1.f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(1450, 150, 1.0f, 1.f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 1450, 50), coin\_animation, false));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(1620, 500, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(1620, 350, 1.0f, 1.f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(1620, 150, 1.0f, 1.f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(1620, 0, 1.0f, 1.f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 1620, -50), coin\_animation, false));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(1790, 500, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(1790, 350, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(1790, 150, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(1790, 0, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(1790, -50, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 1790, -150), coin\_animation, false));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(2250, 500, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(2250, 350, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(2250, 150, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(2250, 0, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(2250, -50, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 2250, -150), coin\_animation, false));

\_entities.push\_back(new CircleEnemy(\_window, Game::Position(Game::CoordType::Pixel, 2030, -150), saw\_anim, 0, 8, 1, false, 18));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(2420, 500, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(2420, 350, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(2420, 150, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(2420, 0, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 2420, -50), coin\_animation, false));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(2590, 500, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(2590, 350, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(2590, 150, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 2590, 50), coin\_animation, false));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(2760, 500, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Box(\_window, "crate", GameObjectParameters(2760, 350, 1.0f, 1.0f, 0.1f, Game::BodyType::Dynamic), crate));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 2760, 50), coin\_animation, false));

\_entities.push\_back(new Platform(\_window,Game::Position(Game::CoordType::Pixel, 3150, 350), big\_platform\_anim, 0, 3, 3.5f));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(3500, 1100, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 3500, 1000), coin\_animation, false));

\_entities.push\_back(new CircleEnemy(\_window, Game::Position(Game::CoordType::Pixel, 3694, 600), enemy\_anim, 0, 6, 1.3f, false));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(3900, 1100, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 3900, 1000), coin\_animation, false));

\_entities.push\_back(new CircleEnemy(\_window, Game::Position(Game::CoordType::Pixel, 4094, 600), enemy\_anim, 0, 6, 1.3f, false));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(4300, 1100, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 4300, 1000), coin\_animation, false));

\_entities.push\_back(new CircleEnemy(\_window, Game::Position(Game::CoordType::Pixel, 4494, 600), enemy\_anim, 0, 6, 1.3f, false));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(4700, 1100, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Heart(\_window, Game::Position(Game::CoordType::Pixel, 4700, 1000), heart\_anim, false));

\_entities.push\_back(new Platform(\_window, Game::Position(Game::CoordType::Pixel, 5050, 350), big\_platform\_anim, 0, 3, 3.5f));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 5050, 250), coin\_animation, false));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(5400, 500, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new Platform(\_window, Game::Position(Game::CoordType::Pixel, 5800, 300), platform\_anim, 0.f, 2.f, 5.f));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 5800, 0), coin\_animation, false));

\_entities.push\_back(new CircleEnemy(\_window, Game::Position(Game::CoordType::Pixel, 6009, 300), saw\_anim, 0, 10, 1.25f, false, 18));

\_entities.push\_back(new Platform(\_window, Game::Position(Game::CoordType::Pixel, 6220, 300), platform\_anim, 0.f, 2.f, 5.f));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 6220, 0), coin\_animation, false));

\_entities.push\_back(new CircleEnemy(\_window, Game::Position(Game::CoordType::Pixel, 6429, 300), saw\_anim, 0, 10, 1.25f, false, 18));

\_entities.push\_back(new Platform(\_window, Game::Position(Game::CoordType::Pixel, 6640, 300), platform\_anim, 0.f, 2.f, 5.f));

\_entities.push\_back(new Coin(\_window, Game::Position(Game::CoordType::Pixel, 6640, 0), coin\_animation, false));

\_entities.push\_back(new Box(\_window, "box", GameObjectParameters(7070, 800, 2.0f, 1.f, 0.1f, Game::BodyType::Static), box\_animation));

\_entities.push\_back(new EndGameObject(\_window, Game::Position(Game::CoordType::Pixel, 7135, 635), end\_anim));

Heart.h

#pragma once

#include "Box.h"

class Heart :

public Box

{

public:

Heart(Game::Window & window, Game::Position const & pos, AnimationPack const & coin\_animations, bool static\_ = true);

virtual bool update(Game::Window & window);

private:

bool \_destruct;

};

Heart.cpp

#include "Heart.h"

Heart::Heart(Game::Window & window, Game::Position const & pos, AnimationPack const & heart\_animations, bool static\_)

:Box(window, "heart", GameObjectParameters(pos.GetPixelPos().x, pos.GetPixelPos().y, 1.f, 0.09f, 0.1f, static\_ ? Game::BodyType::Static : Game::BodyType::Dynamic), heart\_animations)

{

if (static\_ == true)

\_body->GetFixtureList()->SetSensor(true);

\_destruct = false;

}

bool Heart::update(Game::Window & window)

{

if (\_destruct == true)

return false;

for (b2ContactEdge \* contact = \_body->GetContactList(); contact != nullptr; contact = contact->next)

{

std::string object\_a = static\_cast<BaseGameObject \*>(contact->contact->GetFixtureA()->GetBody()->GetUserData())->GetName();

std::string object\_b = static\_cast<BaseGameObject \*>(contact->contact->GetFixtureB()->GetBody()->GetUserData())->GetName();

if (object\_a == "player")

\_destruct = true;

}

draw(window, false);

return true;

}

Menu.h

#pragma once

#include "SFML\Graphics.hpp"

#include "Settings.h"

#include <vector>

#include "GameWindow.h"

#include <initializer\_list>

class GameMenu

{

public:

GameMenu(Game::Window & window, WORD yOffset, std::initializer\_list<std::string> titles, WORD font\_size);

void SetActiveColor(sf::Color color);

void SetInactiveColor(sf::Color color);

sf::Color GetInactiveColor();

sf::Color GetActiveColor();

int GetInput(Game::Window & window); // method getsuser input and returns menu element index

private:

void CalculateTitlesPos(Game::Window & window);

sf::Font \_font;

std::vector<sf::Text> \_titles;

int \_active\_element;

sf::Color \_active\_element\_color;

sf::Color \_inactive\_element\_color;

WORD \_fontSize;

WORD \_yOffset;

};

Menu.cpp

#include "Menu.h"

#include <iostream>

GameMenu::GameMenu(Game::Window & window,

WORD yOffset,

std::initializer\_list<std::string> titles,

WORD font\_size)

{

bool result;

const char\* fontPath = "fonts//AdobeFanHeitiStd-Bold.otf";

result = \_font.loadFromFile(fontPath);

if (!result)

{

std::cerr << "failed to load font " << fontPath;

}

\_active\_element = 0;

\_active\_element\_color = sf::Color::Cyan;

\_inactive\_element\_color = sf::Color::White;

for each (auto title in titles)

{

\_titles.push\_back(sf::Text(title, \_font, font\_size));

}

\_yOffset = yOffset;

CalculateTitlesPos(window);

}

void GameMenu::CalculateTitlesPos(Game::Window & window)

{

int i = 0;

Game::Position screenPos = Game::Position(

Game::CoordType::Pixel,

window.GetSize().GetX() / 2,

window.GetSize().GetY() / 3

);

sf::Vector2f cameraPos = window.GetCameraLeftTopPosition().GetPixelPos();

sf::Vector2f textPosition;

\_titles[\_active\_element].setColor(\_active\_element\_color);

for (int i = 0; i < \_titles.size(); i++)

{

if (i == 0)

{

screenPos.Set(

Game::CoordType::Pixel,

screenPos.GetPixelPos().x - (\_titles[0].getGlobalBounds().width / 2),

screenPos.GetPixelPos().y

);

}

textPosition = sf::Vector2f(

cameraPos.x + screenPos.GetPixelPos().x,

cameraPos.y + screenPos.GetPixelPos().y

);

screenPos.Set(

Game::CoordType::Pixel,

screenPos.GetPixelPos().x,

screenPos.GetPixelPos().y + \_yOffset

);

\_titles[i].setPosition(textPosition);

}

}

void GameMenu::SetActiveColor(sf::Color color)

{

\_active\_element\_color = color;

}

void GameMenu::SetInactiveColor(sf::Color color)

{

\_inactive\_element\_color = color;

}

sf::Color GameMenu::GetInactiveColor()

{

return \_inactive\_element\_color;

}

sf::Color GameMenu::GetActiveColor()

{

return \_active\_element\_color;

}

int GameMenu::GetInput(Game::Window & window)

{

static sf::Time last\_press = window.GetTime();

static int last\_active = 0;

sf::Event event;

CalculateTitlesPos(window);

while (window.PollEvent(event)); // clear events stack

while (true)

{

for (std::vector<sf::Text>::iterator iterator = \_titles.begin(); iterator != \_titles.end(); iterator++)

{

window.PrintString(\*iterator);

}

while (window.PollEvent(event))

{

switch (event.type)

{

case sf::Event::KeyPressed:

switch (event.key.code)

{

case sf::Keyboard::Down:

case sf::Keyboard::S:

if (window.GetTime().asSeconds() - last\_press.asSeconds() < 0.1f)

break;

last\_press = window.GetTime();

last\_active = \_active\_element;

\_active\_element++;

if (\_active\_element > \_titles.size() - 1)

\_active\_element = 0;

\_titles[last\_active].setColor(\_inactive\_element\_color);

\_titles[\_active\_element].setColor(\_active\_element\_color);

window.PlaySound("jump");

break;

case sf::Keyboard::Up:

case sf::Keyboard::W:

if (window.GetTime().asSeconds() - last\_press.asSeconds() < 0.1f)

break;

last\_press = window.GetTime();

last\_active = \_active\_element;

\_active\_element--;

if (\_active\_element < 0)

\_active\_element = \_titles.size() - 1;

\_titles[last\_active].setColor(\_inactive\_element\_color);

\_titles[\_active\_element].setColor(\_active\_element\_color);

window.PlaySound("jump");

break;

case sf::Keyboard::Return :

window.Clear();

return \_active\_element;

case sf::Keyboard::Escape:

while (window.PollEvent(event));

return -2;

}

break;

case sf::Event::Closed:

return -1;

case sf::Event::Resized:

{

CalculateTitlesPos(window);

break;

}

case sf::Event::MouseMoved:

for (int i = 0; i < \_titles.size(); i++)

{

sf::Vector2f mousePos(

event.mouseMove.x + window.GetCameraLeftTopPosition().GetPixelPos().x,

event.mouseMove.y + window.GetCameraLeftTopPosition().GetPixelPos().y

);

if (\_titles[i].getGlobalBounds().contains(mousePos))

{

if (last\_active != i)

window.PlaySound("jump");

last\_active = \_active\_element;

\_titles[last\_active].setColor(\_inactive\_element\_color);

\_active\_element = i;

\_titles[i].setColor(\_active\_element\_color);

}

}

break;

case sf::Event::MouseButtonPressed:

if (event.mouseButton.button == sf::Mouse::Left)

{

for (int i = 0; i < \_titles.size(); i++)

{

sf::Vector2f mousePos(

event.mouseButton.x + window.GetCameraLeftTopPosition().GetPixelPos().x,

event.mouseButton.y + window.GetCameraLeftTopPosition().GetPixelPos().y

);

if (\_titles[i].getGlobalBounds().contains(mousePos))

{

return i;

}

}

}

break;

default:

break;

}

}

window.Display();

}

return \_active\_element;

}

Platform.h

#pragma once

#include "Box.h"

#include <iostream>

class Platform

: public Box

{

public:

Platform(Game::Window & window, Game::Position const & pos, AnimationPack const & animations,

float x\_velocity, float y\_velocity, float loop\_time);

bool virtual update(Game::Window & window);

private:

sf::Time \_timer;

b2Vec2 \_velocity;

float \_time\_to\_change\_dir;

};

Platform.cpp

#include "Player.h"

#include "Menu.h"

Player::Player(Game::Window & window, Game::Position const & spawn\_point)

: Box (

window,

"player",

GameObjectParameters(6, 6, 1.0f, 1.f, 0.01f, Game::BodyType::Dynamic),

AnimationPack("textures\\ninja\_base5.png", 77, 146)

)

{

\_body->SetBullet(true);

\_lives = 0;

\_coins = 0;

\_active\_spawn\_point = 0;

\_flags.win = false;

\_gameStartTime = window.GetTime().asSeconds();

\_font.loadFromFile("fonts//AdobeFanHeitiStd-Bold.otf");

\_info\_table\_texure.loadFromFile("textures//info\_table.png");

\_info\_table\_sprite.setTexture(\_info\_table\_texure);

\_spawn\_points.push\_back(spawn\_point);

\_body->SetFixedRotation(true);

\_animation\_pack.AddFrameSequence("stay", 0, 0, 10, 0.1f, SpritesDir::DOWN, 77, 146);

\_animation\_pack.AddFrameSequence("walk", 78, 0, 10, 0.3f, SpritesDir::DOWN, 121, 153);

\_animation\_pack.AddFrameSequence("jump", 199, 0, 10, 0.4f, SpritesDir::DOWN, 121, 161);

\_animation\_pack.SetLoop("jump", false);

b2Vec2 vec = b2Vec2(

\_spawn\_points[\_active\_spawn\_point].GetPhysicsPos().x,

\_spawn\_points[\_active\_spawn\_point].GetPhysicsPos().y

);

\_body->SetTransform(vec, 0);

\_animation\_pack.SetActive("stay");

}

bool Player::CollisionCheck(Game::Window &window)

{

if (\_body->GetPosition().y / GameParameters::scaling > 1600) // check if player fell

{

if (\_lives > 0)

{

\_lives--;

b2Vec2 vec = \_spawn\_points[\_active\_spawn\_point].GetPhysicsPos();

\_body->SetTransform(vec, 0);

\_body->SetLinearVelocity(b2Vec2(0, 0));

\_body->SetAngularVelocity(0);

}

else

{

\_flags.gameLength = window.GetTime().asSeconds() - \_gameStartTime;

return false;

}

}

for (int i = \_active\_spawn\_point; i < \_spawn\_points.size(); i++)

{

if (\_body->GetPosition().x >= \_spawn\_points[i].GetPhysicsPos().x)

\_active\_spawn\_point = i;

}

for (b2ContactEdge \* contact = \_body->GetContactList(); contact != nullptr; contact = contact->next)

{

std::string objectBName = static\_cast<BaseGameObject \*>(contact->contact->GetFixtureB()->GetBody()->GetUserData())->GetName();

b2Body \* objectBBody = contact->contact->GetFixtureB()->GetBody();

if (objectBName == "box" || objectBName == "platform" || objectBName == "crate")

{

/\*

b2WorldManifold manifold;

contact->contact->GetWorldManifold(&manifold);

if (round(manifold.points[0].y \* 100) / 100 != round(manifold.points[1].y \* 100) / 100) break;

\*/

b2PolygonShape\* polygonA = (b2PolygonShape\*)(contact->contact->GetFixtureA()->GetShape());

b2PolygonShape\* polygonB = (b2PolygonShape\*)(contact->contact->GetFixtureB()->GetShape());

if (abs((objectBBody->GetPosition().y + polygonB->GetVertex(1).y) - (\_body->GetPosition().y + polygonA->GetVertex(3).y)) < 0.5)

{

if (\_flags.in\_the\_air == true && contact->contact->IsTouching())

{

\_flags.in\_the\_air = false;

//window.PlaySound("heart");

\_body->SetLinearVelocity(b2Vec2(0, 10));

//\_body->ApplyForceToCenter(b2Vec2(0, -20), true);

}

}

}

if (objectBName == "coin")

{

\_coins++;

window.PlaySound("coin");

}

if (objectBName == "heart")

{

\_lives++;

window.PlaySound("heart");

}

if (objectBName == "enemy" && contact->contact->IsTouching())

{

if (\_lives > 0)

{

\_lives--;

\_body->SetLinearVelocity(b2Vec2(0, 0));

b2Vec2 transform = b2Vec2(

\_spawn\_points[\_active\_spawn\_point].GetPhysicsPos().x,

\_spawn\_points[\_active\_spawn\_point].GetPhysicsPos().y

);

\_body->SetLinearVelocity(b2Vec2(0, 0));

\_body->SetTransform(transform , 0);

}

else

{

\_flags.gameLength = window.GetTime().asSeconds() - \_gameStartTime;

return false;

}

}

if (objectBName == "end" && contact->contact->IsTouching())

{

\_flags.gameLength = window.GetTime().asSeconds() - \_gameStartTime;

\_flags.win = true;

return false;

}

}

if (\_body->GetContactList() == nullptr)

\_flags.in\_the\_air = true;

return true;

}

void Player::UpdateScreenInfo(Game::Window & window)

{

sf::Vector2f info\_pos(

window.GetCameraLeftTopPosition().GetPixelPos().x,

window.GetCameraLeftTopPosition().GetPixelPos().y

);

sf::Vector2f timer\_pos(

info\_pos.x + window.GetSize().GetPixelPos().x,

info\_pos.y + 5);

//sprite show heart

\_info\_table\_sprite.setTextureRect(sf::IntRect(0, 0, 64, 64));

char number\_string[8];

//heart

\_itoa\_s(\_lives, number\_string, 8, 10);

sf::String string;

info\_pos.y += 10;

info\_pos.x += 4;

string += "x ";

string += number\_string;

sf::Text title(string, \_font, 30);

\_info\_table\_sprite.setPosition(info\_pos);

info\_pos.y += 64 / 2;

info\_pos.x += 64;

title.setPosition(info\_pos);

title.setColor(sf::Color::Red);

window.PrintString(title);

window.Draw(\_info\_table\_sprite);

//coins

\_itoa\_s(\_coins, number\_string, 8, 10);

string.clear();

string += "x ";

string += number\_string;

info\_pos.y += 45;

info\_pos.x -= 64;

\_info\_table\_sprite.setPosition(info\_pos);

info\_pos.x += 64;

info\_pos.y += 64 / 2;

//info\_pos.y += 64 / 3;

title.setPosition(info\_pos);

title.setString(string);

title.setColor(sf::Color::Yellow);

\_info\_table\_sprite.setTextureRect(sf::IntRect(64, 0, 64, 64));

window.PrintString(title);

window.Draw(\_info\_table\_sprite);

//time

\_itoa\_s(

(int)window.GetTime().asSeconds() - \_gameStartTime,

number\_string,

8, 10

);

string.clear();

string += number\_string;

string += " sec";

title.setString(string);

timer\_pos.x -= title.getGlobalBounds().width + 10;

title.setPosition(timer\_pos);

title.setColor(sf::Color::White);

window.PrintString(title);

}

void Player::Reset(Game::Position const & spawn\_point, float gameStartTime)

{

if (gameStartTime < 0)

throw std::logic\_error("negative time");

\_lives = 0;

\_coins = 0;

//\_spawn\_points.push\_back(Game::Position(Game::CoordType::Pixel, 1500, 300));

\_body->SetTransform(b2Vec2(0, 0), 0);

\_flags.win = false;

\_active\_spawn\_point = 0;

\_spawn\_points.clear();

\_spawn\_points.push\_back(spawn\_point);

\_body->SetTransform(b2Vec2(spawn\_point.GetPhysicsPos().x, spawn\_point.GetPhysicsPos().y), 0);

\_gameStartTime = gameStartTime;

}

bool Player::isWin()

{

return \_flags.win;

}

void Player::LastScreen(Game::Window & window)

{

sf::Vector2f pos(

window.GetCameraLeftTopPosition().GetPixelPos().x + window.GetSize().GetX() / 2,

window.GetCameraLeftTopPosition().GetPixelPos().y + window.GetSize().GetY() / 4

);

sf::String string;

sf::Text text = sf::Text();

text.setFont(\_font);

text.setCharacterSize(50);

char buff[8];

if (\_flags.win == false)

{

string += "Defeat";

text.setString(string);

text.setStyle(sf::Text::Regular);

text.setColor(sf::Color(sf::Color::Red));

pos.x -= text.getGlobalBounds().width / 2;

text.setPosition(pos);

window.PrintString(text);

}

else

{

string += "Victory";

text.setString(string);

text.setStyle(sf::Text::Regular);

text.setColor(sf::Color(sf::Color::Cyan));

pos.x -= text.getGlobalBounds().width / 2;

text.setPosition(pos);

window.PrintString(text);

}

text.setCharacterSize(35);

text.setStyle(sf::Text::Italic);

\_itoa\_s(\_lives, buff, 8, 10);

string = "Hearts: ";

string += buff;

pos.y += 70;

text.setColor(sf::Color::Red);

text.setPosition(pos);

text.setString(string);

window.PrintString(text);

\_itoa\_s(\_coins, buff, 8, 10);

string = "Coins: ";

string += buff;

pos.y += 40;

text.setColor(sf::Color::Yellow);

text.setPosition(pos);

text.setString(string);

window.PrintString(text);

\_itoa\_s(\_flags.gameLength, buff, 8, 10);

string = "Time: ";

string += buff;

string += " sec";

pos.y += 40;

text.setColor(sf::Color::White);

text.setPosition(pos);

text.setString(string);

window.PrintString(text);

}

void Player::AddSpawnPoints(std::initializer\_list<Game::Position> positions)

{

for (auto iterator = positions.begin(); iterator != positions.end(); iterator++)

{

\_spawn\_points.push\_back(\*iterator);

}

}

bool Player::update(Game::Window & window)

{

if (CollisionCheck(window) == false)

{

b2Vec2 vec = \_spawn\_points[\_active\_spawn\_point].GetPhysicsPos();

\_body->SetTransform(vec, 0);

\_body->SetLinearVelocity(b2Vec2(0, 0));

\_body->SetAngularVelocity(0);

return false;

}

static Game::KEYBOARD\_ACTION last\_action = Game::KEYBOARD\_ACTION::Inaction;

switch (window.GetKeyboardAction())

{

case Game::KEYBOARD\_ACTION::Right:

if (\_flags.in\_the\_air == false)

{

\_body->ApplyForceToCenter(b2Vec2(50.f, 0.f), true);

\_animation\_pack.SetActive("walk");

}

else

{

\_body->ApplyForceToCenter(b2Vec2(5.f, 0.f), true);

}

\_flags.flip = false;

break;

case Game::KEYBOARD\_ACTION::Inaction:

if (last\_action == Game::KEYBOARD\_ACTION::Right || last\_action == Game::KEYBOARD\_ACTION::Left)

{

if (\_flags.in\_the\_air == false)

{

\_body->SetLinearVelocity(b2Vec2(0, 0));

\_animation\_pack.SetActive("stay");

}

}

else if (\_flags.in\_the\_air == false)

\_animation\_pack.SetActive("stay");

else if (last\_action == Game::KEYBOARD\_ACTION::Right || last\_action == Game::KEYBOARD\_ACTION::Left)

\_animation\_pack.SetActive("stay");

break;

case Game::KEYBOARD\_ACTION::Left:

if (\_flags.in\_the\_air == false)

{

\_body->ApplyForceToCenter(b2Vec2(-50.f, 0.f), true);

\_animation\_pack.SetActive("walk");

}

else

{

\_body->ApplyForceToCenter(b2Vec2(-5.f, 0.f), true);

}

\_flags.flip = true;

break;

case Game::KEYBOARD\_ACTION::Up:

case Game::KEYBOARD\_ACTION::Jump:

if (\_flags.in\_the\_air || last\_action == Game::KEYBOARD\_ACTION::Jump || window.GetTime().asSeconds() - \_flags.jump\_time.asSeconds() < 0.5f) {

break;

}

\_animation\_pack.SetActive("jump");

\_animation\_pack.Replay();

window.PlaySound("jump");

//\_body->ApplyForceToCenter(b2Vec2(0.f, -1000.f), true);

\_body->ApplyLinearImpulse(b2Vec2(0.f, -14.f), \_body->GetWorldCenter(), true);

\_flags.jump\_time = window.GetTime();

break;

case Game::KEYBOARD\_ACTION::Exit:

if (\_exitMenuClock.getElapsedTime().asSeconds() < 0.5f) {

break;

}

GameMenu menu { window, 50, {"Continue", "Exit"}, 40 };

window.DrawBackground();

sf::Clock menuLaunchClock;

switch (menu.GetInput(window))

{

case 0:

\_gameStartTime += menuLaunchClock.getElapsedTime().asSeconds();

break;

case 1:

case -1:

return false;

case -2:

break;

}

\_exitMenuClock.restart();

break;

}

last\_action = window.GetKeyboardAction();

/\*switch (window.GetMouseAction())

{

}\*/

Game::Position body\_pos(Game::CoordType::Physics, \_body->GetPosition().x, \_body->GetPosition().y);

//moving the camera

window.MoveCamera(

body\_pos.GetPixelPos().x - (window.GetSize().GetPixelPos().x / 2),

body\_pos.GetPixelPos().y - (window.GetSize().GetPixelPos().y / 2)

);

UpdateScreenInfo(window);

draw(window, \_flags.flip);

return true;

}

Polygon.h

#pragma once

#include "BaseGameObject.h"

class Polygon

: public BaseGameObject

{

public:

//Polygon(b2World & World, std::string name, GameObjectParameters const & parameters, AnimationPack const & animations, std::vector<Game::Position> const & points\_pos); // maximum points count is 8

Polygon(Game::Window & \_world, std::string name, GameObjectParameters const & parameters, AnimationPack const & animations, std::vector<Game::Position> const & points\_pos); // maximum points count is 8

};

Polygon.cpp

#include "Polygon.h"

Polygon::Polygon(Game::Window & world, std::string name, GameObjectParameters const & parameters, AnimationPack const & animations, std::vector<Game::Position> const & points\_pos)

: BaseGameObject(animations)

{

if (points\_pos.size() > b2\_maxPolygonVertices)

throw exc::InvalidGameObjectParameters("more Polygons than allowed");

b2Vec2 \* vertices = new b2Vec2[points\_pos.size()];

for (WORD i = 0; i < points\_pos.size(); i++)

{

if (abs(points\_pos[i].GetPixelPos().x) > animations.GetFrameWidth() / 2 || abs(points\_pos[i].GetPixelPos().y) > animations.GetFrameHeight() / 2)

{

delete[] vertices;

std::string exception = "Invalid polygon point number ";

exception += i + '0';

throw exc::InvalidGameObjectParameters("Invalid polygon point");

}

else

{

vertices[i].x = points\_pos[i].GetPhysicsPos().x;

vertices[i].y = points\_pos[i].GetPhysicsPos().y;

}

}

b2PolygonShape shape;

shape.Set(vertices, points\_pos.size());

Initialize(world, name, parameters, shape);

delete[] vertices;

}

Source.cpp

#include <Box2D\Box2D.h>

#include <SFML\Graphics.hpp>

#include <iostream>

#include "Animation.h"

#include "GameWorld.h"

#include "Menu.h"

#include <Windows.h>

int WINAPI WinMain(\_In\_ HINSTANCE hInstance, \_In\_ HINSTANCE hPrevInstance, \_In\_ LPSTR lpCmdLine, \_In\_ int nCmdShow)

{

GameWorld \*world = new GameWorld("Ninja", 0.f, 9.8f);

world->Launch();

delete world;

return 0;

}

Settings.h

#pragma once

typedef unsigned short WORD;

namespace GameParameters

{

const float scaling = 0.013f;

const WORD WindowWidth = 1024;

const WORD WindowHeight = 768;

enum OBJ\_TYPE

{

SENSOR = 0x01,

PLAYER = 0x02

};

}