ЗМІСТ

ВСТУП

У даній курсовій роботі я спробував свої сили у реалізації достатньо складного з точки зору алгоритму, use-case взаємодії, і навіть складності обчислень програми, що являє собою фундамент гри. Я не хочу казати що те, що міститься на диску, є моєю грою, тому что я ні у якому разі не визнаю, що це є кінцевим результатом моєї роботи. Це є лише гарною основою для гри, яка містить більшу частину основних функції та реалізовує ігрову механіку. У програмі продемонстровані програмні рішення для реалізації динамічного руху персонажів по глобальній карті, руху камери, динамічне створення / знищення об'єктів на карті; Поза увагою не слід залишати складну систему графічних ефектів, які реалізовані при, наприклад, русі первонажів, стрільбі, вибухах, прискоренні герою. У грі я реалізував типовий 2D shooter – ігровий інтерфейс, що являє собою представлення динамічної камери виду top-down з статичними елементами контролю (наприклад, стрічка здоров'я головного героя, міні-карта і т.д.) Одною з найбільш складних речей, які були реалізовані мною у цьому проекті є створення так званих ігрових івентів — певних подій, що теоритично можуть відбутися у певний момент часу при виконанні користувачем певних умов. Прикладом таких івентів може бути, наприклад, поява повідомлення з підказкою у певний момент гри, кори герой наближається до сильного ворога. Або івент несподіваної появи чотирьох супротивників на початку гри прямо біля героя, саме коли він дуже близько до них підлітає. Більш того, я створив зручний програмний інтерфейс для створення і додавання таких івентів у дизайн ігрового рівня. Про івенти я розповім дуже детально трохи пізніше.

Мною було заплановано те, що гра буде містити декілька повністю реалізованих у плані сюжету і геймдизайну рівнів, які будуть створювати разом деяку сюжетку кампанію. Звичайно ж, я не встиг це зробити. На данному етапі реалізації ми маємо лише один неповний рівень, який навіть

не містить умову проходження. Але у майбутньому їх буде багато і вони будуть гарно зроблені.

Під кінець вступу хочу додати, що у цій записці я намагатимусь якнайкраще описати хід моїх думок при знаходженні первих вирішень проблем, яких було дуже немало. Всього я працюю над цією роботою близько двох місяців. За цей час дуже багато всього я виправляв і змінював. Майже кожен день я знаходив певні кращі рішення для старих проблем і переписував код, таким чином вдосконалюючи програму все більше і більше. Але тут є ще багато всього, що можна покращити. І я впевнений, що мій код все одно я далеко не кращим, оскільки досвіду у розробці програм такого маштабу я не маю зовсім.

Вся розробка веляся на моїй улюбленій мові програмування C++ з використанням мого улюбленного (і єдиного, яким я володію) фреймворка Qt, C++, як на мою думку, є одним з найскладніших мов програмування, але з найширшими можливостями і найкращим потенціалом у плані оптимізації. Більш того, gamedev та C++ є майже синонімами. При виконанні цієї курсової роботи я знайшов і виправив таку кількість багів, яку ще я ніколи не бачив. І, звичайно ж, я набув гарного досвіду проектування та програмування на мові C++. Але мені є ще дуже багато чого дізнатися, оскільки цю мову треба вчити і вчити роками.

1. Розробка програми

1.1 Скруктуризація програми

1.1.1 Реалізація переходів між станами програми

Перед початком програмування важливим етапом було визначити структуру, яку матиме програма. Під структурою я маю на увазі такі елементи архітектури, як шаблони проектування і перний мініатюрний ігровий движок. Необхідний функціонал “мініатюрного ігрового движка” у моїй програмі виконує стандартний класс фреймворка *Qt - QGraphicsScene* , що містить повний інструментарій для розміщення, руху і фіксації колізій об'єктів на моїй глобальній карті. Тепер про паттерни. Головною проблемою, яка вимагала використання паттернів, була проблема динамічної зміни стану вікна гри піл час її взаємодії з користувачем. З самого початку я планував, що гра буде містити як мінімум 4 наступні стани:

- Стан гри (GameplayState)
- Стан налаштувань (SettingsState)
- Стан з допоміжною інформацією (AboutState)
- Стан меню, що дозволяє переходити до інших станів (MainMenuState)

У мене на час структуризації вже був невеликий досвід у користуванні паттерном State, і першою думкою була думка скористатись саме цим паттерном. У перший час я не знав, як саме його реалізвати. Але у ближчий час я знайшов непогане рішення. Я знайшов класс Qt, який є найбильш правильним рішенням для реалізації різних станів віджета: QstackedWidget. Отже, моя проблема була вирішена простим прийомом — наслідуванням від цього самого QstackedWidget. Тобто, таким чином і був створений головний клас моєї програми — GameWindow, що являє собою управляючий елемент-контейнер для всіх станів ігрового вікна.

(header-файл з імплементацією класу на наступній сторінці)

```
#ifndef GAMEWINDOW_H
#define GAMEWINDOW_H

#include <QStackedWidget>
#include "state.h"
#include "global_enumerations.h"

class QGraphicsView;
class QWidget;
class QLabel;
class QGridLayout;
class CustomScene;
class QVBoxLayout;
class BGMusicPlayer;
class GameplayState;
class SettingsState;
class MediaCache;

class GameWindow : public QStackedWidget
{
    Q_OBJECT
public:
    GameWindow();
    ~GameWindow();
    void setState(State::ID);
    QCursor getStandardCursor() const;
    QPixmap *getPictureFromCache(Pictures::Type);

private:
    /*Widnow states*/
    State           *menu_state;
    State           *settings_state;
    State           *gameplay_state;
    State           *about_state;

    QCursor standard_cursor;

    BGMusicPlayer   *music_player;
    MediaCache      *m_cache;
    int             wwidth;
    int             wheight;
};

#endif // GAMEWINDOW_H
```

Лістинг 1 — хедер-файл управляючого класу GameWindow

У прайват-даних цього класу поки що звернемо увагу лише на 4 покажчика на State. Це і є наші об'єкти станів. Тепер трохи більше саме про цей клас State, оскільки це буде важливо для пояснення механізму зміни станів.

Для реалізації станів я створив абстрактний клас State. Подивимось на його лістинг:

```
#ifndef STATE_H
#define STATE_H

#include <QObject>
#include <QPixmap>
#include <QCursor>

class GameWindow;
class QGridLayout;
class QGraphicsView;

class State : public QObject
{
    Q_OBJECT
public:
    enum ID{MainMenu = 0, Settings, Gameplay, About, NewGameplay};

    State() {}
    virtual ~State() {}
    virtual QWidget *getStateWidget() const = 0;
};

#endif // STATE_H
```

Лістинг 2 — абстрактний клас State

Одже, пояснюю: по перше, цей клас є об'єктом QObject. Це потрібно для того, щоб класи-нащадки могли використовувати механізм сигналів і слотів Qt. Далі, ми бачимо перелік ID. Він створений для зручної реалізації “імен” різних ігрових станів і можливості передачі цих “імен” аргументами функцій. Це буде продемонстровано пізніше. Подивимось на інтерфейс класу:

Пустий конструктор, оскільки нам нема що ініціалізувати. Віртуальний деструктор для того, щоб через покажчик на State викликався деструктор класа-нащадка. І найголовніше — віртуальна функція getStateWidget(), яка повертає покажчик на QWidget. Для чого потрібна ця функція? Поясню механізм роботи QStakedWidget : спокачту у нього виконується вставка

віджетів, а потім поточний стан обирається методом setCurrentIndex(int). Тобто план дій такий:

- Наслідуванням від State створюємо класи для всіх станів.
- Кожен класс-нащадок повинен створити свій об'єкт QWidget і реалізувати його так, як йому потрібно.
- Отримуємо покажчики на віджети станів, вставляємо їх у стек віджетів при ініціалізації об'єкта GameWindow.
- Більш того, за нами залишається можливість динамічно створювати, вставляти і обирати віджети у нашому стеку віджетів. Саме такий підхід я використав для реалізації опції New Game : з стеку видаляється віджет GameplatState, повністю видаляється об'єкт GamplayState, створюється новий, заново отримуємо його віджет і заново виконуємо його вставку у стек.

Перехід від одного стану до іншого відбувається через фунцію GameWindow::setState(State::ID). Для того, щоб один стан міг ініціювати перехід до іншого, він повинен зберігати покажчик на GameWindow. Він передається кожному з об'єктів при створенні у конструктор, разом з розмірами екрану. Виглядає це так:

```
/*All states memory allocation*/
    menu_state =      new MainMenuState(this, wwidth, wheight);
    settings_state =  new SettingsState(this, wwidth, wheight);
    gameplay_state =  new GameplayState(this, wwidth, wheight);
    about_state =     new AboutState   (this, wwidth, wheight);
```

Лістинг 3 — створення об'єктів-станів

Приклад переходу з одного стану до іншого:

```
void MainMenuState::slotButtonAbout()
{
    player->stop();
    player->setMedia(QUrl::fromLocalFile(SELECT_SOUND));
    player->play();

    connect(timer_before_change, SIGNAL(timeout()), SLOT(slotAboutState()));
    timer_before_change->start(CHANGE_T_MS);
}
```

```
void MainMenuState::slotAboutState()
{
    timer_before_change->stop();
    disconnect(timer_before_change, SIGNAL(timeout()), this,
                                    SLOT(slotAboutState()));
    player->stop();

    game_window->setState(State::About);
}
```

Лістинг 4 — перехід з стану MainMenuState до стану AboutState

На лістингу показаний типовий перехід з одного стану у інший. Всі переходи реалізовані саме таким чином: тобто, спокатку, при натисканні певної кнопки викликається слот slotButtonAbout(). Після цього вимикається музика стану (про музику трохи пізніше), далі у плеєр завантажується мелодія тапу (звук, який супроводжує натискання будь-якої клавіші) і плеєр програє цю мелодію. Після цього я з'єдную сигнал таймеру для переходу до іншого стану, який має назву *timer_before_change,* зі слотом кінцевого переходу до стану About, і запускаю таймер. Отже, через час затримки після натискання клавіші, який задається макросом CHANGE_T_MS, відбудеться виклик слота переходу, який вимкне звук , від'єднає сигнал таймера *timer_before_change* від цього слота і викличе функцію для переходу до іншого стану з індексом цього стану. Момент з від'єднанням сигналу і слота дуже важливий, тому что я витратив багато часу на непорозуміння, викликане тим, что я цього не робив. Фреймворк Qt при подвійному з'єднанні сигналів і слотів викликатиме слот 2 рази, тобто, якщо користувач декілька разів виходив у меню, то переходи до станів будуть відбуватися дивним чином.

Розглянемо останній етап переходу до іншого стану:

```
void GameWindow::setState(State::ID id)
{
    switch (id) {
    case State::MainMenu:
        music_player->playMenuMusic();
        this->setCurrentIndex(State::MainMenu);
        break;
```

```
        case State::Gameplay:
            music_player->playGameplayMusic();
            this->setCurrentIndex(State::Gameplay);
            dynamic_cast <GameplayState *>(gameplay_state)
            ->getView()->setFocus();
            break;

        case State::NewGameplay:
            this->removeWidget(gameplay_state->getStateWidget());
            delete gameplay_state;
            gameplay_state = new GameplayState(this, wwidth, wheight);
            this->insertWidget(2, gameplay_state->getStateWidget());

            music_player->playGameplayMusic();
            this->setCurrentIndex(State::Gameplay);
            dynamic_cast <GameplayState *>(gameplay_state)
            ->getView()->setFocus();
            break;

        case State::Settings:
            this->setCurrentIndex(State::Settings);
            break;

        case State::About:
            music_player->playMenuMusic();
            this->setCurrentIndex(State::About);
            break;
        }
}
```

Лістинг 5 — реалізація функції setState

На мою думку не треба пояснювати, як це працює. Тут все просто, отже, без коментарів. Є лише пара моментів, на які треба звернути увагу. По перше, music_player – це об'єкт класа BGMusicPlayer. Так виглядає його інтерфейс:

```
#ifndef BGMUSICPLAYER_H
#define BGMUSICPLAYER_H

#include <QMediaPlayer>
#include <QObject>

/*Class provides background music, that is independent from
 * GameWindow states*/

class BGMusicPlayer : public QObject
{
    Q_OBJECT
public:
    BGMusicPlayer();
    ~BGMusicPlayer();

    /*Play different kinds of music*/
    void playMenuMusic    ();
    void playGameplayMusic();
```

```
    /*Check is any kind of music is playing*/
    bool playingMenuMusic    () const;
    bool playingGameplayMusic() const;

    /*Check volume*/
    int volume() const;

public slots:
    void setVolume(int);

private:
    QMediaPlayer  *player;
    QMediaContent *menu_music;
    QMediaContent *gameplay_music;

    bool is_playing_menu_music     {false};
    bool is_playing_gameplay_music {false};
};

#endif // BGMUSICPLAYER_H
```

Лістинг 6 — інтерфейс класа BGMusicPlayer

Даний клас використовує композицію для зберігання об'єкту QMediaPlayer і реалізує програвання музики на рівні вищої ієрархії. Тобто це є фоновий плеєр, який працює на вищому рівні від станів. Всі інші звуки реалізуються плеєрами нижчих рівнів, тобто вони є об'єктами класів-станів або класів ще нижчих за ієрархією. Поки що даний плеєр може програвати лише музику меню та музику ігрового процесу. Музика у цих станах відрізняється. При переході зі стану меню до будь-якого стану, окрім стану гри, музика не зупиняється і продовжує грати на фоні попри будь-які інші звукові або графічні ефекти переходу.

Отже, ми розглянули повний шлях від натискання клавіші до виконання переходу до іншого стану програми.

1.1.2 Реалізація окремих станів програми

У цьому пункті я розповім про те, як я реалізував чотири основні стани програми. Почнемо у такому самому порядку, у якому з програмою працює звичайний користувач — з стану головного меню. Закінчимо

найбільш, я би навіть сказав, надзвичайно об'ємною реалізацією стану ігрового процесу.

1.1.2.1 Стан меню (MainMenuState)

Рисунок 1 — скріншот головного меню

Без зайвих слів, розгляд кожного стану починатимемо з лістингу його хедер-файлу:

```
#ifndef MAINMENUSTATE_H
#define MAINMENUSTATE_H

#include "state.h"
#include <QSettings>

class QPushButton;
class QVBoxLayout;
class QPixmap;
class QGraphicsView;
class QMediaPlayer;
class QTimer;
class QWidget;
class MenuScene;
class QGraphicsSimpleTextItem;
class QLabel;
class CustomTextItem;

class MainMenuState : public State
```

```
{
    Q_OBJECT
public:
    MainMenuState(GameWindow *, const int ww, const int wh);
    ~MainMenuState();
    QWidget *getStateWidget() const override;

public slots:
    void slotChangeWallpaper(int);

private /*objects*/:
    QSettings settings;

    QGridLayout          *lout;
    MenuScene            *pmenu_scene;
    QGraphicsView        *pgraphics_view;
    GameWindow           *game_window;
    QMediaPlayer         *player;
    /*Buttons:*/
    QPushButton          *btn_play;
    QPushButton          *btn_settings;
    QPushButton          *btn_about;
    QPushButton          *btn_quit;
    QPushButton          *btn_new_game;

    QTimer               *timer_before_change;
    QTimer               *timer_current_time;
    QWidget              *state_widget;
    CustomTextItem       *text_time;
    const int            wwidth;
    const int            wheight;

private /*functions*/:
    void readSettings();

private slots:
    void slotButtonPlay();
    void slotGameplayState();
    void slotSettingsState();
    void slotButtonSettings();
    void slotButtonQuit();
    void slotButtonAbout();
    void slotButtonNewGame();
    void slotAboutState();
    void slotNewGame();

    void slotUpdateTime();

    void slotQuitFromApp();
};

#endif // MAINMENUSTATE_H
```

Лістинг 7 — хедер-файл класу MainMenuState

Розглянемо клас більш детально, але не занадто (деякі речі я буду опускати.)

По-перше, у ньому перевизначена віртуальна функція getStateWidget(). Це ми побачимо у кожному стані. У даних цього класу ми можемо побачити цей самий

```
QWidget *state_widget;
```

Об'єкт стану сам ініціалізує об'єкт свого віджету, кастомізує його і тільки може повернути покажчик на нього. Кастомізація полягає у, можливо, зміні деяких налаштувань віджета, таких як розмір (до речі, розміри вікна передаються у контруктор кожного зі станів,, і кожен зі станів задає свому віджету потрібні розміри.), або те, що використовується найчастіше: натягування лейаутів на віджет.

Далі у даних класу ми бачино такі цікаві речі, як графічна сцена і представлення. Для чого вони побрібні, спитаєте ви мене? Насправді, якщо я б реалізовував простий мінімалістичний дизайн, це ускладнення не знадобилось би. Але я вирішив використовувати графічну сцену для представлення всіх віджетів для того, щоб мати можливість доповнювати дизайн незвичайними растровими графічними ефектами і, можливо, для певного полегшення розміщування об'єктів на віджеті (без сцени доводилося б гратися к лейаутами, а зі сценою гратися з координатами.) Через це я мав перні проблеми з розміщенням віджетів на сцені (сцена на віджеті з віджетами), які , насправді, не були проблемами, оскільки у Qt є можливість вбудовувати віджети у сцену за допомогою методу addWidget() і класу QGraphicsProxyWidget. Сцена у даному випадку представлена класом MenuScene.

```
#ifndef MENUSCENE_H
#define MENUSCENE_H

#include <QGraphicsScene>

class QTimer;

class MenuScene : public QGraphicsScene
{
    Q_OBJECT
public:
    explicit MenuScene(const int w, const int h, QObject *parent = 0);
```

```
    ~MenuScene();
    void setBackgroundImage(const int);

private:
    void drawBackground(QPainter *painter, const QRectF &rect) override;
    QPixmap *bg;
    QPixmap *logo;
    const int WIDTH;
    const int HEIGHT;

    QTimer *opacity_timer;
    qreal opacity {100};

private slots:
    void slotChangeOpacity();
};

#endif // MENUSCENE_H
```

Лістинг 8 — хедер-файл класу MenuScene

Основною причиною для створення свого класу для сцени меню стала потреба перемалювати фон віджета. До речі, створення будь-якої анімації на фоні — також причина використати графічну сцену, так як динамічно змінювати фон звичайного віджета способу немає. У цьому випадку ми перевизначаємо функцію drawBackground() для графічної сцени для того, щоб малювати фон меню. На фоні я також малюю логотип гри, і за допомогою таймера і опції *QPainter::setOpacity()* я створюю анімацію блимання логотипу.

Повернемося до розгляду реалізації стану меню. Клас містить покажчик на *GameWindow* * для того, щоб зберігати покажчик на єдиний об'єкт цього класу, який передається у конструктор. Нагадаю, що зберігаємо ми цей покажчик для того, щоб у майбутньому мати можливість виконати перехід до іншого стану за допомогою функції *GameWindow::setState(State::ID).*

Клас містить покажчик на об'єкт типу *QMediaPlayer*. Цей плеєр використовується для відтворення звуку тапання на кнопки меню. Такі звуки відтворюються поверх фонової музики, що створюється *BGMusicPlayer,* розглянутим раніше.

Далі ми маємо 4 кнопки *QPushButton*. Для кнопок не довелось створювати власний клас, оскільки стандартний клас містить всі необхідні засоби для кастомізації кнопок. Для кастомізації кнопок я використовував власний дизайн іконки, динамічний дизайн QSS (аналог CSS) для user-взаємодії, маску для того, щоб “обрізати” форму та, звичайно ж, змінював розмір так, як меня потрібно. Оскільки всі кнопки (або майже всі) у цій грі повинні мати однаковий дизайн, я зробив шаблонну функцію, яка приймає покажчик на екземпляр кнопки (*QPushButton* *) та *const char* * строку, що містить URL-адресу на зображення іконки. Функція має наступний вигляд і винесена у хедер-файл *templates.h:*

```
static void setUpButton(QPushButton *pbtn, const char *icon_name)
{
    //sets up clear button to required form with icon
    static QRegion reg(QPolygon() << QPoint(0,10) << QPoint(10, 0) <<
                                     QPoint(240, 0) << QPoint(240, 42.12) <<
                                     QPoint(230, 52.12) << QPoint(0, 52.12));
    pbtn->setIcon(QIcon(QPixmap(icon_name)));
    pbtn->setIconSize(QSize(240, 52.12));
    pbtn->setFixedSize(240, 52.12);
    pbtn->setMask(reg);
    pbtn->setStyleSheet("QPushButton:hover {"
                                "border: 1px;"
                                "background: lightblue;"
                        "}");}
```

Лістинг 9 — реалізація функції setUpButton() у файлі templates.h

QSS у цьому випадку дає ефект певного об'єму кнопки при наведенні на неї курсору. Приклад:

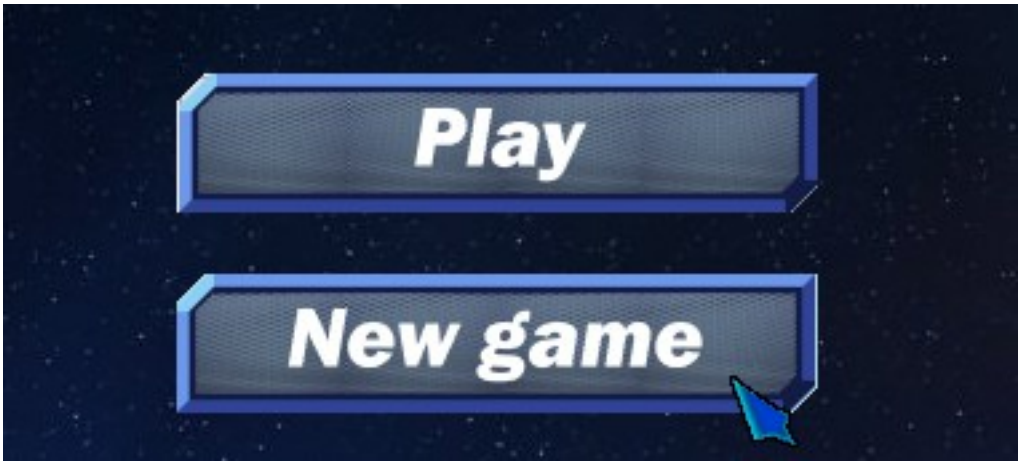

Рисунок 2 — ефект об'ємності як результат використання QSS

Нагадаю, що дизайн кнопок я робив, використовуючи свої вміння роботи з графічними програмами *Adobe Photoshop, Adobe Illustrator*, які до мого

курсу не входять. Але, як на мене, не слід залишати поза увагою те, що кнопочки я зробив досить гарні.

Далі ми маємо два таймера. Причину створення *timer_before_change* я вже пояснив. Тепер перейдемо до другого таймера — *timer_current_time*. Він потрібен для того, щоб оновлювати час. Як ви можете побачики з рисунку 1 або з роботи моєї програми — у верхньому правому куті я реалізував віджет відображення часу і дати, який також має ефект блимання. Час оновлюється кожну секунду, але секунди не показує. Для того, щоб фіксувати час і дату, а також отримувати її у форматі локалізованої строки, я використав клас QDateTime. Розглянемо клас *CustomTextItem*, який реалізовує цей самий віджет (так, назва не дуже вдала. Насправді треба було назвати його якось накшталт C*ustomDateTimeItem.*)

```
#ifndef CUSTOMTEXTITEM_H
#define CUSTOMTEXTITEM_H

#include <QGraphicsSimpleTextItem>
#include <QFont>

class QTimer;

class CustomTextItem : public QGraphicsSimpleTextItem
{
public:
    CustomTextItem();
    ~CustomTextItem();

private:
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
QWidget *widget) override;
    QFont font;
    int opacity {0};
    bool flag {false};
    inline void changeOpacity();
};

#endif // CUSTOMTEXTITEM_H
```

Лістинг 10 — хедер-файл класа CustomTextItem

Отже, бачимо, що цей клас наслідується від *QGraphicsSimpleTextItem* для того, щоб можна було розміщувати його на графічній сцені. Далі все по класиці: переписуємо функцію *paint()* так, як нам треба.

Зверну увагу лише на використання класа QDateTime:

```
painter->drawText(0, 0,
QDateTime::currentDateTime().time().toString().left(5));
    font.setPointSize(15);
    painter->setFont(font);
    painter->drawText(70, 30,
QDateTime::currentDateTime().date().toString("ddd.dd"));
```

Лістинг 11 — частина функції paint()

Як бачимо, кожен раз при малюванні об'єкта ми малюємо форматований текст, який отримуємо за допомогою цього класа.

У цьому класі нам залишилось лише звернути увагу на функцію readSettings(). У цій програмі для зберігання налаштувань користувача я використовую клас QSettings, який надає зручний інтерфейс програмісту для збереження і зчитування файлів користувача. Ці файли зберігаються у реєстрі даних програми. По зчитування налаштувань ми поговоримо трохи пізніше, коли я буду розповідати про клас SettingsState, а у цьому випадку я лише скажу, що ця функція використовується для того, щоб завантажити та намалювати ту картинку фону, яку обрав користувач.

1.1.2.2 Реалізація стану налаштувань

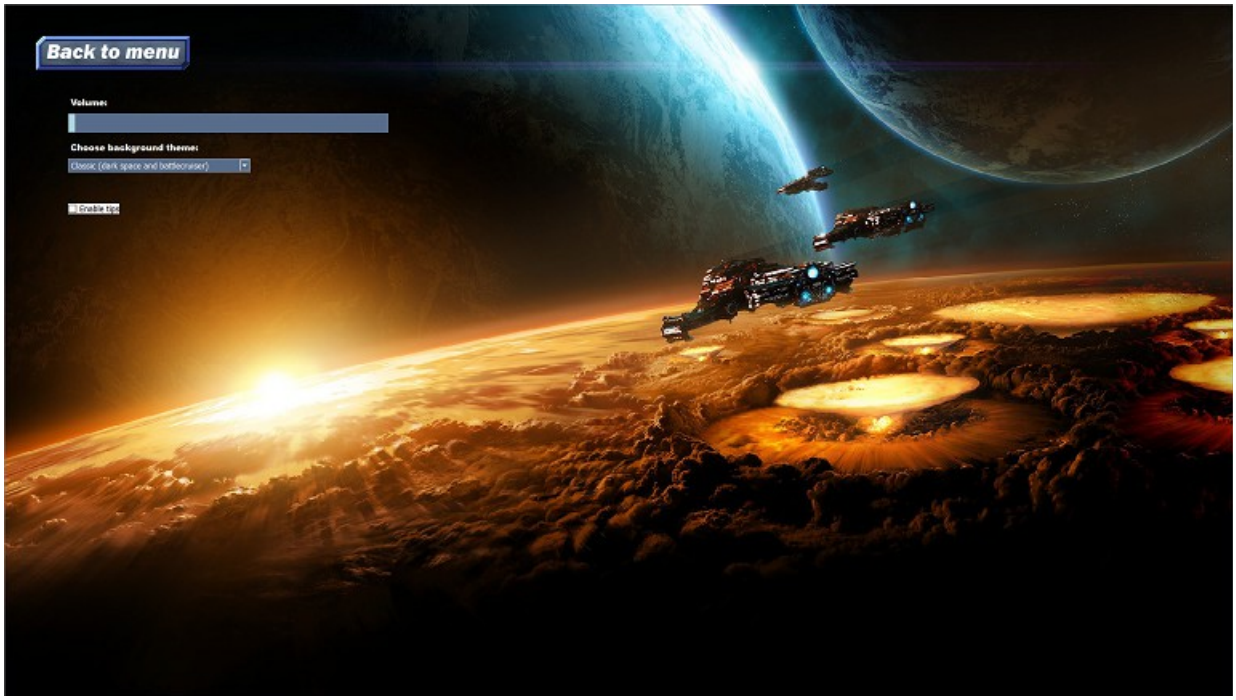

Рисунок 3 — налаштування

Сказати чесно, хотілося б, щоб налаштувань було більше. Але на даний момент я не придумав нічого більше, окрім цих двох (третій віджет

нічого не робить) віджетів для налаштувань — слайдера для звуку та комбо-бокс для вибору фонової картинки.

Почнемо, як завжди, з лістингу:

```
#ifndef SETTINGSSTATE_H
#define SETTINGSSTATE_H

#include "state.h"
#include <QSettings>

class SettingsScene;
class QVBoxLayout;
class QSlider;
class QGraphicsView;
class QLabel;
class QPushButton;
class QMediaPlayer;
class QTimer;
class QComboBox;
class QCheckBox;

class SettingsState : public State
{
    Q_OBJECT
public:
    SettingsState(GameWindow *, const int, const int);
    ~SettingsState();
    QWidget *getStateWidget() const override;
    int Volume() const;

signals:
    void signalChangeVolume(int);
    void signalChangeWallpaper(int);

private /*objects*/:
    /*Settings:*/
    QSettings settings;

    GameWindow *game_window;
    QGridLayout *lout;
    /*Settings setters:*/
    QSlider *volume_slider;
    QComboBox *wp_combo_box;
    QCheckBox *ntf_check_box;

    QPixmap *bg;
    SettingsScene *pstgs_scene;
    QGraphicsView *pgraphics_view;
    QTimer *timer_before_change;
    QLabel *lbl_volume;
    QPushButton *btn_to_menu;
    QPushButton *btn_apply_wp_change;
    QMediaPlayer *player;
    QWidget *state_widget;
    const int wwidth;
    const int wheight;
```

```
    unsigned short _last_background_index;

private /*functions*/:
    void readSettings();
    void writeSettings();

private slots:
    void slotChangeVolume(int);
    void slotBtnToMenuClicked();
    void slotMenuState();
    void slotWpChangeApplied(int);
};

#endif // SETTINGSSTATE_H
```

Лістинг 12 — хедер-файл класу SettingsState

У цьому класі увагу слід звернути лише на механізм збереження/зчитування користувацьких налаштувань. Як я казав раніше, для цього я використав клас QSettings. Функції readSettings(), whiteSettings() виконують функції зчитування та запису налаштувань під час створення і знищення об'єкта стану. Тобто функція для зчитування викликається у конструкторі класа, а для запису — у деструкторі. QSettings надає нам можливість у зручному вигляді працювати з будь-яким форматом збережених даних. Для прикладу приведемо функцію зчитування:

```
void SettingsState::readSettings()
{
    settings.beginGroup("/Settings");

    volume_slider->setValue(settings.value("/volume", 100).toInt());
    wp_combo_box->setCurrentIndex(settings.value("/bg_index", 0).toInt());

    settings.endGroup();
}
```

Лістинг 13 — функція readSettings()

Тобто, я зчитую дані, які зберігаються у реєстрі програми за адресою *Settings/Volume*, і перетворюю їх з типу *QVariant* до типа *int*.

1.1.2.3 Реалізація стану додаткової інформації

Рисунок 4 — додаткова інформація

У цьому стані абсолютно немає нічого цікавого. Все дуже просто: я палюю текст на фоні. І все. Промальовка фона працює так само, як і на інших станах: через наслідування графічної сцени.

```
#ifndef ABOUTSTATE_H
#define ABOUTSTATE_H

#include "state.h"

class GameWindow;
class AboutScene;
class QGraphicsView;
class QPushButton;
class QMediaPlayer;
class QTimer;
class QGridLayout;
class QLabel;

class AboutState : public State
{
    Q_OBJECT
public:
    AboutState(GameWindow *, const int, const int);
    ~AboutState();
    QWidget *getStateWidget() const override;

private:
    /*scene and view of this state*/
    AboutScene    *pabout_scene;
```

```
    QGraphicsView *pgraphics_view;

    /*GameWindow pointer to emit change state signal*/
    GameWindow    *game_window;

    /*Buttons*/
    QPushButton   *btn_menu;

    /*Player for any exactly this state's sounds or music*/
    QMediaPlayer  *player;

    /*Timers*/
    QTimer        *timer_before_change;

    /*State widget layout*/
    QGridLayout   *lout;

    /*State widget*/
    QWidget       *state_widget;

    QLabel        *lbl_text_about;

    /*Sizes*/
    const int     wwidth;
    const int     wheight;

private slots:
    void slotBtnMenuClicked();
    void slotMenuState();
};

#endif // ABOUTSTATE_H
```

Лістинг 14 — хедер-файл класа AboutState

1.1.2.4 Реалізація стану ігрового процеса

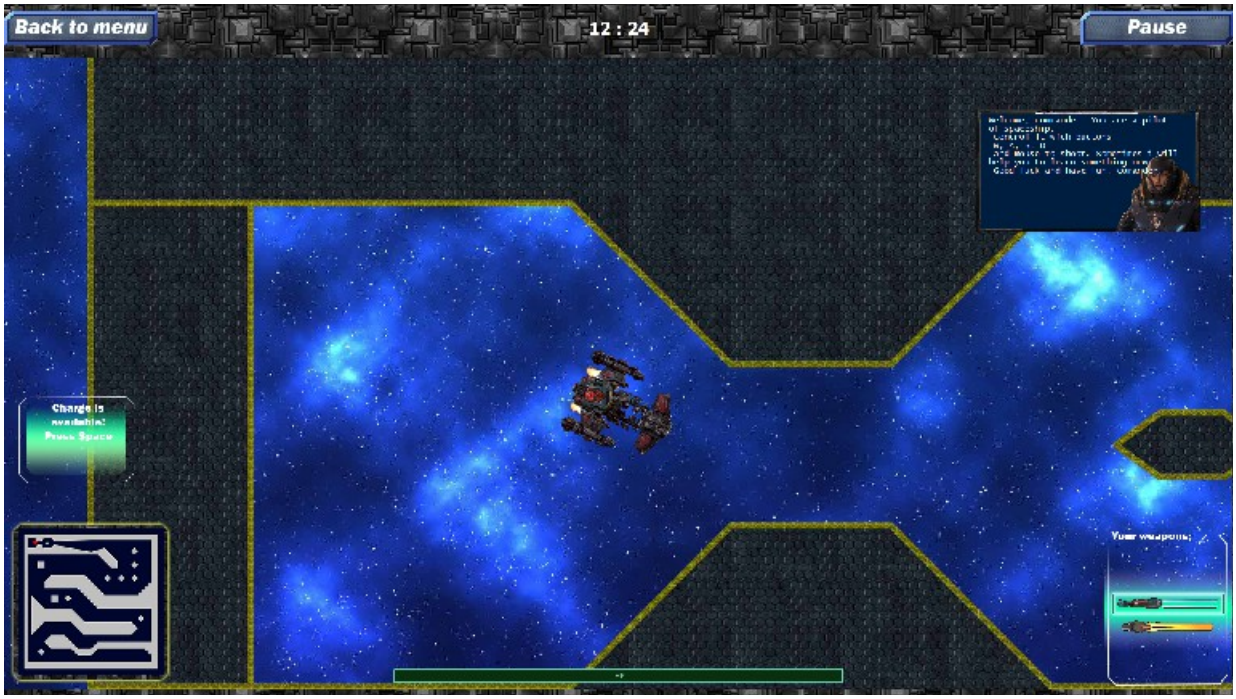

Рисунок 5 — початок гри

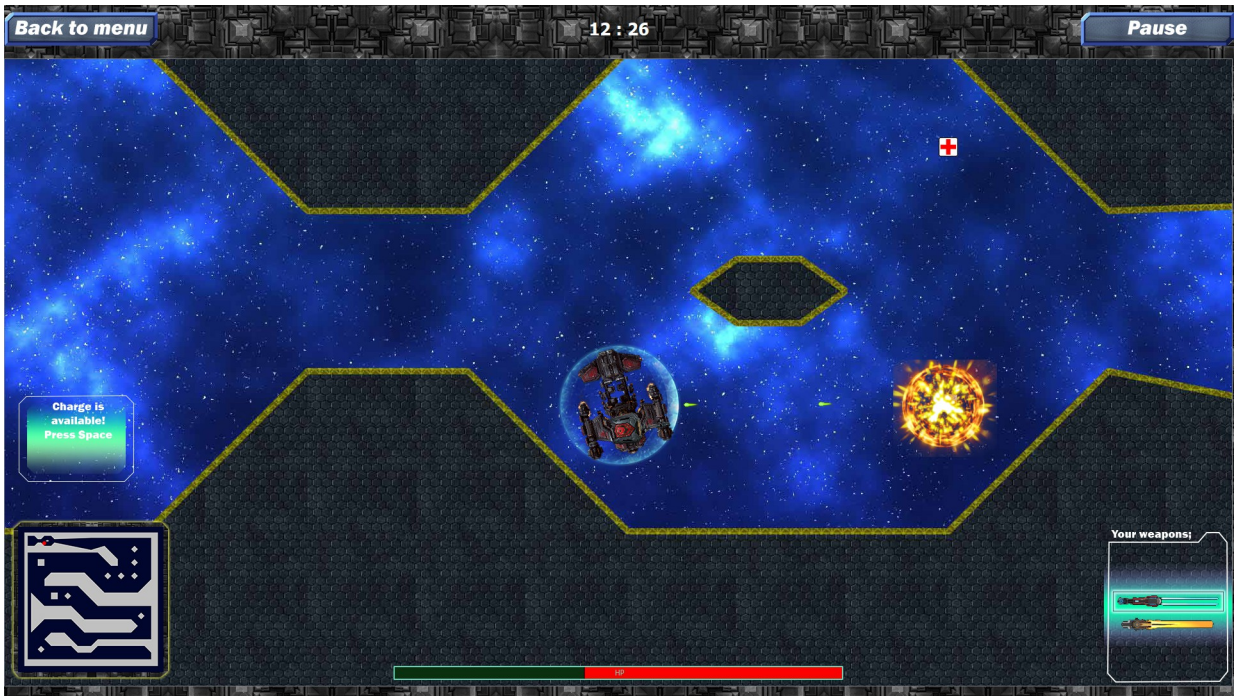

Рисунок 6 — знищення перших супротивників

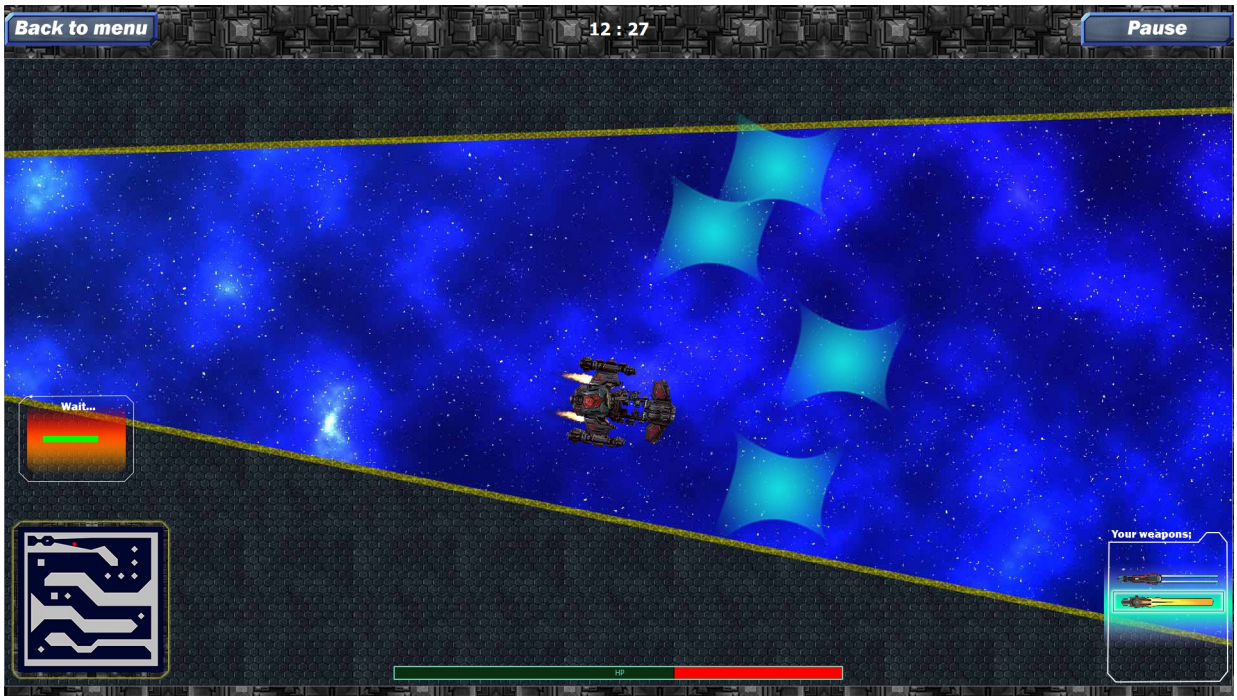

Рисунок 7 — спрацювання ігрової події з'явлення чотирьох супротивників і анімація їх з'явлення

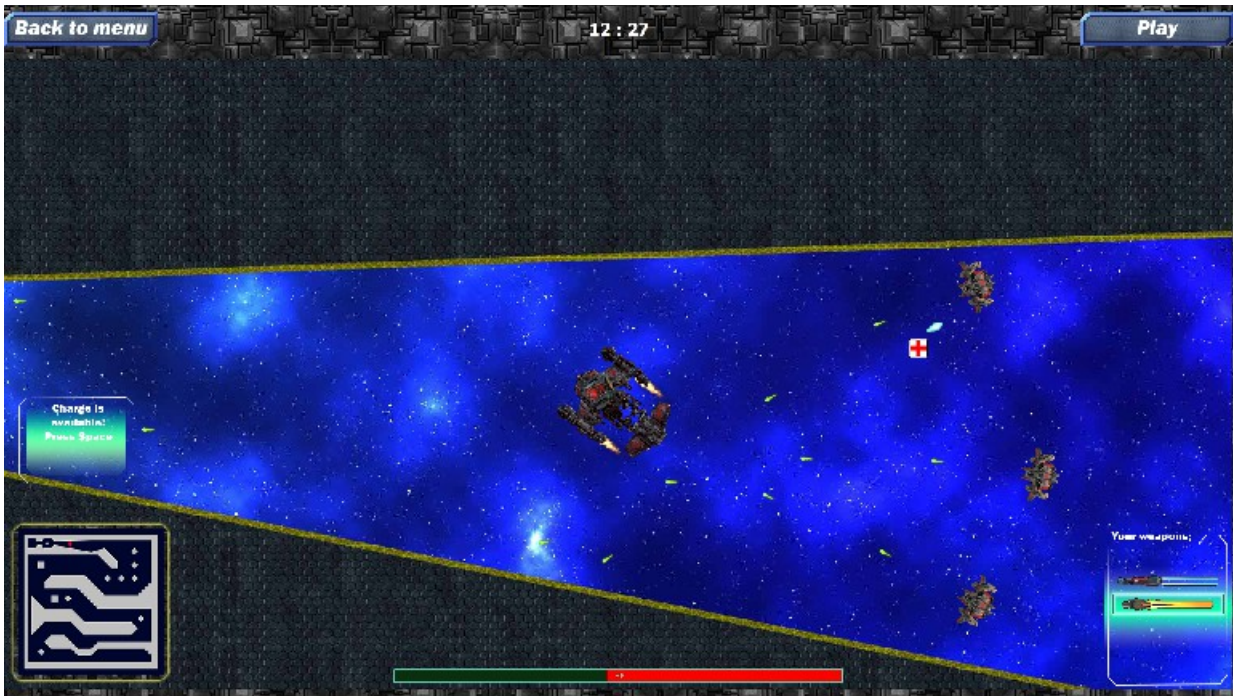

Рисунок 8 — ігровий процес. Перестрілка з супротивниками

Рисунок 9 — вікно виходу до головного меню

Про цей клас ми поговоримо набагато більше, ніж про інші. Саме на цей клас пішло більшість часу. Це є логічним, оскільки основна складність у розробці гри полягає у програванні ігрового процесу та геймдизайні.

Але все по порядку. Спочатку розглянемо хедер-файл.

```
#ifndef GAMEPLAYSTATE_H
#define GAMEPLAYSTATE_H

#include "state.h"

class QuitWindow;
class GameScene;
class QGraphicsView;
class Hero;
class State;
class QPushButton;
class QVBoxLayout;
class QMediaPlayer;
class QTimer;
class QGraphicsProxyWidget;
class QGraphicsScene;
class GameView;
class QDateTime;
class QLabel;
class HpLine;
class GameEvent;

class GameplayState : public State
{
    Q_OBJECT
public:
```

```
    GameplayState(GameWindow *, const int, const int);
    ~GameplayState();
    QWidget *getStateWidget() const override;
    QGraphicsView *getView() const;

private /*objects*/:
    /*Main view elements: Scene, View and it's layout*/
    GameScene            *pgraphics_scene;
    GameView             *pgraphics_view;
    QVBoxLayout          *lout;
    Hero                 *phero;

    /*We store Game Window ptr to change it's state when required*/
    GameWindow           *game_window;

    /*Buttons*/
    QPushButton          *btn_to_menu;
    QPushButton          *btn_pause;
    QGraphicsProxyWidget *proxy_btn_ok;
    QGraphicsProxyWidget *proxy_btn_no;

    /*Music features*/
    QMediaPlayer         *player;               /*Media player for button
clicked sound*/
    QTimer               *timer_before_change;  /*Timer after some button
has clicked*/
    QTimer               *change_time_timer;

    /*Graphics*/
    QuitWindow           *quit_window;

    /*Time*/
    QLabel               *lbl_time;

    /*Main state widget*/
    QWidget              *state_widget;

    /*Game events*/
    QList <GameEvent *>  events;

    /*Widget size constants*/
    const int            wwidth;
    const int            wheight;

    /*Pause flag*/
    bool paused {false};

private /*functions*/:
    void quitHandler();
    void executeEvents();
    void makeEvents();

private slots:
    void slotButtonToMenuClicked();
    void slotMenuState();
    void slotQuitConfirmed();
    void slotDontQuit();
    void slotButtonPauseClicked();
    void slotUpdateTime();
```

```
};

#endif // GAMEPLAYSTATE_H
```

Лістинг 15 — хедер-файл класа GameplayState

У цьому стані слід звернути увагу на клас GameScene. На відміну від інших станів, сцена цього є набагато складнішим класом. Розглянемо її лістинг:

```
#ifndef GAMESCENE_H
#define GAMESCENE_H

#include <QGraphicsScene>
#include <QSet>
#include <QList>
#include <QVector>

class QPixmap;
class Decoration;
class Hero;
class HpLine;
class MiniMap;
class Weapon;
class Charger;
class QGraphicsView;
class InfoWindow;

class GameScene : public QGraphicsScene
{
    Q_OBJECT
public:
    explicit GameScene(const int, const int, const QPixmap *, QObject * =
0);
    ~GameScene();

    void readDecorations(const QString);              //read decoration
polygons from file
    void setHero(Hero *);
    void setView(QGraphicsView *);
    void addInfoWindow(InfoWindow *);
    void popInfoWindow();

signals:
    void signalTargetCoordinate();
    void signalShot             (bool);
    void signalButtons          (QSet <Qt::Key> &);

private slots:
    void slotUpdateViewport();                                    //slot
which controls program FPS.

private /*functions*/:
    void mouseMoveEvent    (QGraphicsSceneMouseEvent *) override;     //Mouse
tracking;
```

```
    void keyPressEvent     (QKeyEvent *)                override;     //Key
tracking;
    void keyReleaseEvent   (QKeyEvent *)                override;     //Key
release control;
    void mousePressEvent   (QGraphicsSceneMouseEvent *) override;
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *) override;
    void drawBackground    (QPainter *, const QRectF &) override;
//function that draws BG;

private /*objects*/:
    Hero *phero;
    QList <Decoration *>    dec_list;             //vector which contains all
Decorations;
    QSet <Qt::Key>          pr_keys;              //set which contains all
legal pressed and not released keys;
    QVector <InfoWindow *> info_windows_v;
    const QPixmap           *bg_image;
    QBrush                  *footer_brush;
    QPolygon                footer_polygon;
    QTimer                  *fps_timer;
    QPointF                 target;

    /*Custom elements:*/
    HpLine                  *hp_line;
    MiniMap                 *mini_map;
    Weapon                  *weapon;
    Charger                 *charger;
    InfoWindow              *info_window {nullptr};

    qreal                   hero_angle;
    QGraphicsView           *view;
    const int               wwidth;
    const int               wheight;
    const int               STEP {3};
};

#endif // GAMESCENE_H
```

Лістинг 16 — хедер-файл класа GameScene

Цей клас є ключовим у нашому стані гри. Оскільки саме він є оболонкою для зберігання всіх об'єктів на глобальній карті та рух камери. Він містить функцію *readDecorations()*, яка зчитує з файлика *décor.txt* у кореневій папці всі декорації для розміщення на глобальній карті. На даний момент всі декорації представляють собою полігони, яки мають однакову текстуру. У майбутньому я планую зробити різні типи декорацій, у тому числі динамічні. Функції *setHero()* та *setView()* потрібні для того, щоб клас стану міг передати даному класу ці покажчики. Покажчик на героя потрібен для того, щоб сцена мала можливість відслідковувати його дії, а покажчик на представлення потрібний для того, щоб сцена мала змогу фіксувати

положення камери на глобальній карті. До речі, цікава річ: функція *sceneRect()* класа *QGraphicsScene* та однойменна функція класа *QGraphicsView* повертають різні значення. Перша повертає прямокутник глобальної карти, у той час коли друга — прямокутник камери на цій карті.

Функції *addInfoWindow()* та *popInfoWindow()* реалізовані для взаємодії сцени з івентами інформаційних повідомлень. Про роботу цих івентів поговоримо пізніше, а поки що поясню, що роблять ці функції:

Перша функція додає у вектор інформаційних вікон *info_windows_v* новий покажчик на таке вікно, таким чином це повідомлення буде виведене на екран.

Друга функція каже сцені про те, що треба видалити перше повідомлення у цьому векторі. Цю функцію викликає саме інформаційне вікно у той момент, коли збирається зникнути. Я реалізував саме такий механізм для максимального розширення можливостей касомізації повідомлень. Тобто, сцена працює з ними лише на рівні показав — видалив, за все інше, наприклад, за свій вигляд, за час свого існування, інформаційні повідомлення відповідають самі. До речі, поки що всі інформаційні повідомлення існують однаковий час, і спрощеня видалення через pop без явного вказання, яке повідомлення треба видалити, працює. Але у майбутньому функцію *popInfoWindow()* треба буде переробити на вигляд *deleteInfoWindow(const InfoWindow *).*

Тепер поговоримо трохи про сигнали. У класі ігрової сцени ми маємо їх всього 3. Їх функції достатньо прості.

- *signalTargetCoordinate()* сповіщає героя сцену про те, що користувач перемістив курсор і треба оновити його позицію, а саме у випадку героя це потрібно для того, щоб почати повертати башту зі зброєю до нового напрямку курсора.

- *signalShot()* передається також до героя і сповіщає його про те, що користувач дав сигнал до стрільби. Чого саме такий механізм? Тому що герой сам по собі не є віджетом і він не може відлідковувати дії користувача. А сцена є віджетом, і тому вона може відслідковувати натискання клавіш і рух курсора. У дакону випадку ми маємо переписану функцію *mousePressedEvent()*, яка і викликає сигнал початку стрільби, і функцію *mouseReleasedEvent()* яка за допомогою того ж самого сигналу сповіщає героя про те, що треба припинити стрільбу.
- *signalButtons()* сповіщає героя про те, що множина натиснених у даний момент клавіш змінилася. Викликається з переписаних функцій *keyPressedEvent(), keyReleasedEvent().*

Далі поговоримо про *slotUpdateViewport()*. У цього слота дуже проста і безцінно корисна функція. Він оновлює зображення сцени. Тобто, цей слот просто викликає функцію *QGraphicsScene::update()*. За допомогою цього слота можна контролювати FPS ігрової сцени. Я багато досліджував залежність продуктивності процесора від цієї частоти і дійшов до висновку, що накраще буде оновлювати її раз у 17-18 мс. Такий параметр дає майже непомітний ефект візуальної дискретизації. Якщо оновлювати частіше, тобто 16 і менше мс, тоді процесорна складність обчислень дуже різно йде вгору. Так, наприклад, при 17 мс мій процесор завантажений на ~10%, а при 15 вже на ~30%.

Навіщо взагалі мені потрібно оновлювати сцену у цьому слоті? Так, дійсно, по замовченню update() викликається кожен раз, коли викликається paintEvent() віджета або коли будь-який елемент сцени змінює будь-які свої графічні параметри. Саме такий підхід я і використовував на початку роботи, але в результати дійшов висновку, що це дуже неоптимально. Коли кількість об'єктів на сцені зростає, тим більш, якщо вони швидко динамічно

змінюються (наприклад, кулі, які швидко летять) то дуже сильно зростає навантаження на процесор. Тому я заборонив автоматичне оновлення сцени

```
pgraphics_view->setViewportUpdateMode(QGraphicsView::NoViewportUpdate);
```

і викликаю його викоючно у цьому слоті. Зверніть увагу, що в інших станах такий підхід не використовується через непотрібність такої оптимізації.

Ми майже повністю розглянули основні моменти у роботі ігрової сцени. Слід звернути увагу на такі дані цього класу, як *HpLine, Charger, MiniMap, Weapon*. Ці класи є статичними об'єктами на сцені, тобно їх положення фіксоване при русі камери. Можливо, у майбутньому я перенесу їх до класу *QGraphicsView*, оскільки з точки зору правильної програмної архітектури там їх буде доречніше.Всі ці 4 класи зв'язані з героєм через механізм сигналів і слотів. Їх реалізація достатньо проста і на ній ми зупинятись не будемо.

Далі перейдемо до розгляду класа *GameView*. Але , насправді, абсолютно нічого цікавого тут немає. По суті, на даний момент розробки цей клас не потрібен і він абсолютно нічим не відрізняється від звичайного *QGraphicsView*. Початково я створював його для того, щоб малювати динамічний курсор на представленні. Але потім я відмовився від цієї ідеї.

Далі, ми маємо об'єкт QVBoxLayout. Він є головним лейаутом на нашій сцені. Він відповідає за розміщення верхньої панелі з двома кнопками і годинником, і розміщення графічного представлення.

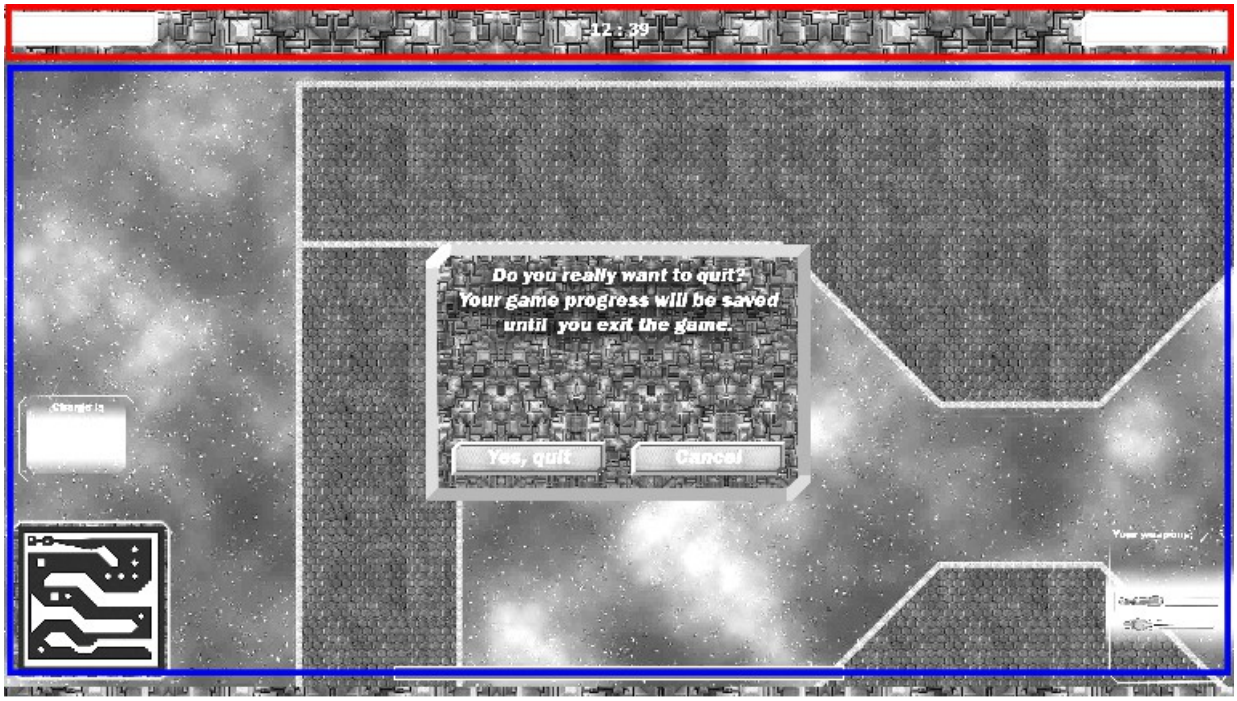

Рисунок 10 — лейаути ігрового віджета

На рисунку 11 показана схема розміщення лейаутів на головному віджеті. Хедер віджета — дві кнопки та лейбл, який відображає поточний час. Далі йде віджет графічного представлення. І нижня частина — невеликий незайнятий простір.

Більш поглиблено розглядати цей клас я не бачу сенсу, оскільки реалізація всього іншого є досить примітивною, а реалізацію таких об'єктів, як герой, супротивники та ігрові івенти я розгляну пізніше у спеціальних пунктах.

1.1.3 Герой і супротивники

Цікавими у плані реалізації є класи героя і супротивників. Але для їх детального розгляду треба для початку коротко пояснити мою абстракцію цих класів на рівні ігрової сцени:

- Клас *GameplayItem* реалізовує нерухомий клас, який не може отримати шкоди від куль.
- Клас *GameplayMovableItem* є абстрактним нащадком попереднього класа, і реалізовує інтерфейс класа, що має можливість переміщуватись і отримувати шкоду від куль.
- Для кожного класа-нащадка цих класів визначається тип об'єкта в залежності від особливостей взаємодії з іншими.

Все досить просто. Будь-який об'єкт, який будет розміщуватися на сцені, або повинен бути гарантовано видаленим до переключення до іншого стану або натискання кнопки паузи, або повинен бути пронаслідуваним від цього класу. Ці класи дають зручний програмний інтерфейс для взаємодії об'єктів на ігровій сцені.

Розглянемо хедер-файл героя:

```
#ifndef HERO_H
#define HERO_H

#include <QObject>
#include <QSet>
#include "gameplaymovableitem.h"

class QTimer;
class Gun;
class HeroThrust;
class QMediaPlayer;
class Shield;

class Hero : public QObject, public GameplayMovableItem
{
    Q_OBJECT
public:
    explicit Hero(QObject *parent = 0);
    ~Hero();

    void stopTime() override;
    void startTime() override;
    void getDamage(const int) override;
    int type() const override;
```

```
    void changeWeapon();
    void charge();
    void activateShield();

signals:
    void moveBackground(qreal);
    void signalHpChanged(int);
    void signalCharged();

public slots:
    void slotTarget();
    void slotShot  (bool);

private slots:
    void slotHeroTimer();
    void slotButtons  (QSet <Qt::Key> &);
    void slotChargeEnded();
    void slotCharged();
    void slotShieldExpired();

private /*functions*/:
    QRectF boundingRect() const override;
    void paint         (QPainter *, const QStyleOptionGraphicsItem *,
QWidget *) override;
    QPainterPath shape () const override;
    QVariant itemChange(GraphicsItemChange, const QVariant &)
override;

    void moveSystem(const QPointF &);
    void updateThrustsPos();

private /*objects*/:
    QMediaPlayer    *player;
    QPixmap         *hero_pic;
    QTimer          *ptimer;
    QTimer          *charge_timer;
    QPointF         target;
    Gun             *pgun;
    Shield          *shield {nullptr};
    QSet <Qt::Key> keys;
    HeroThrust      *left_th;
    HeroThrust      *right_th;
    const int       STEP {4};
    int hp          {1000};
    int step {3};
    int default_step {3};
    int charged_step {8};
    bool can_charge {true};
    bool can_activate_shield {true};
    bool has_shield {false};
};

#endif // HERO_H
```

Лістинг 17 — хедер-файл hero.h

Поки що не слід звертати уваги на переписані віртуальні *public*-функції цього класу. Для чого вони потрібні і як вони працюють я поясню пізніше, коли буду розглядати абстракцію.

В цілому герой складається з трьох класів: сам клас *hero,* який реалізовує космічний корабель. Клас *Gun*, який реалізовує башту з гарматою, що може обертатися на 360 градусів. Клас *HeroThrust,* що реалізовує реактивний викид полум'я при русі героя. Також є допомжні класи, які безпосередково взаємодіють з класами “героїчної” групи:
Клас *FlameStream* – анімація героя при стрільбі та клас *Shield* – силове захисне поле героя, яке активується при натисканні клавіші.

Супротивники реалізовані схожим чином, але вони мають спрощені елементи автоматизованого управління. На даний момент вони зроблені дуже примітивники. Це одна з головних проблем моєї ігри на даний момент. Всього я маю 2 типи супротивників: звичайні, реалізовані класом Enemy, та “сильні” супротивники, реалізовані класом *HeavyEnemy*. Розглянемо хедери цих класів:

```
#ifndef ENEMY_H
#define ENEMY_H

#include "gameplaymovableitem.h"
#include <QObject>

class QMediaPlayer;
class QMediaPlaylist;

class Enemy : public QObject, public GameplayMovableItem
{
    Q_OBJECT
public:
    explicit Enemy(QObject * = 0);
    ~Enemy();

    void stopTime() override;
    void startTime() override;
    int type() const override;

    void getDamage(const int) override;
```

```
    static void setHero(QGraphicsItem *);

protected:
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
QWidget *widget) override;
    QRectF boundingRect() const override;
    QVariant itemChange(GraphicsItemChange change, const QVariant &value)
override;
    void expire();
    QPixmap *enemy_pic;
    QPixmap *bullet_pic;
    QTimer *timer;
    QTimer *shooting_timer;
    QMediaPlayer *player;
    QMediaPlaylist *playlist;
    int hp {200};
    int shot_interval;
    static QGraphicsItem *hero;

private slots:
    virtual void slotTimer();
    virtual void slotShoot();
};

#endif // ENEMY_H
```

Лістинг 18 — хедер-файл класа Enemy

Клас супротивника, так само як і клас героя, пронаслідуваний від інтерфейсу *GameplayMovableItem*, оскільки супротивник здатен динамічно змінювати свої координати та отримувати шкоду від інших (тобто, має показник здоров'я.) Від *QObject* клас наслідується для того, щоб мати змогу використовувати механізм сигналів і слотів Qt.

Зверніть увагу на те, що клас Enemy зберігає статичний покажчик на героя. Це зроблено мною у цілях невеликої оптимізації та спрощення вибору пріорітетної цілі для атаки супротивника. Можливо, у майбутньому доведеться це змінити, оскільки цей механізм, коли супротивники знають лише героя і можуть атакувати лише героя, є досить примітивним з точни зору ігрового процесу.

Далі розглянемо клас *HeavyEnemy*. Не те, щоб у ньому було щось особливе, але просто для того, щоб продемонструвати використання потужного механізму наслідування і його зручність.

```
#ifndef HEAVYENEMY_H
#define HEAVYENEMY_H

#include "enemy.h"

class HeavyEnemy : public Enemy
{
public:
    HeavyEnemy();
    ~HeavyEnemy();

private:
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
QWidget *widget) override;
    QRectF boundingRect() const override;

private slots:
    void slotShoot() override;
    void slotTimer() override;
};

#endif // HEAVYENEMY_H
```

Лістинг 19 — хедер-файл класа HeavyEnemy

Цей приклад є непоганим прикладом того, як наслідування може полегшити нам життя. Реалізовуючи клас сильного супротивника, я лише перевизначив основні слоти, функцію *paint()* та фунцію *boundingRect(),* оскільки розмір супротивника змінився. Зверніть увагу на те, що для цього мені довелося зробити дані класу Enemy не *private*, а *protected.*

1.2 Елементи програмної абстракції і створення шаблонів для полегшення написання коду і створення нових елементів програми

1.2.1. Абстрагування елементів стану гри

У цьому пункті детально розглянемо мою реалізацію абстрактних класів (або інтерфейсів) для об'єктів стану ігрового процесу.

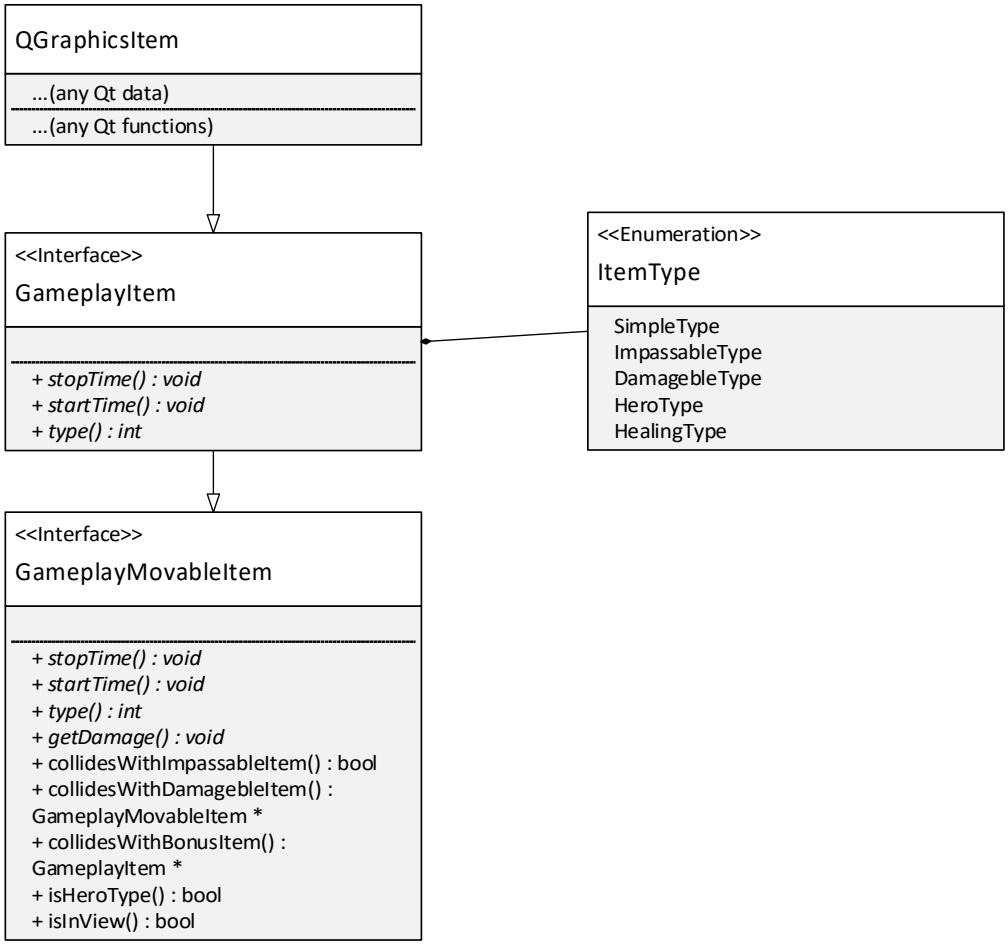

Рисунок 11 — UML-діаграма інтерфейсів групи GamplayItem

Тепер я по порядку поясню, що робить кожна з цих функцій. Почнемо з *GameplayItem*. Він наслідується від *QGraphicsItem* тому що він буде розмішуватися на графічній сцені. Він має всього дві віртуальні функції — *startTime(), stopTime()*. Ці функції потрібні для того, щоб зупинити і запустити знов усю динамічну складову об'єкта, коли користувач ставить гру на паузу та, відповідно, знімає її. Це єдине нормальне рішення для реалізації паузи, яке я знайшов. Тобто, коли користувач тисне на кнопку

паузи, я перебираю ВСІ елементи графічної сцени, динамічно перетворюю їх покажчики на покажчики на *GameplayItem*, та викликаю функцію s*topTime()*. Будь-який об'єкт, що розміщений на сцені на момент паузи повинен бути наслідником цього класу.

GameplayMovableItem є все більш цікавим і складним класом. Він є інтерфейсом будь-якого рухомого об'єкта на сцені.

Розглянемо функцію *type()*. Ця функція повертає індекс типу цього класу. Тип у даному випадку використовується для того, щоб динамічно опрацьовувати взаємодію різних об'єктів на сцені. Наприклад, якщо герой зустрічається з *ImpassableType*, він розуміє, що рухатись далі не можна. Або якщо куля зустрічається з *DamagebleType*, вона розуміє, що можна вибухнути і викликати функцію *getDamage()* у об'єкта.

Розглянемо групу функцій для перевірки колізій.

- *collidesWithImpassableItem()* - повертає *true*, якщо даний об'єкт *GameplayMovableItem* має колізію на сцені з об'єктом типу *ImpassableType*. Інакше повертає *false.*
- *collidesWithDamagebleItem()* - повертає об'єкт типу *GameplayMovableItem* *, якщо даний об'єкт того самого типу має колізію на графічній сцені з іншим об'єктом типу *DamagebleType* або *HeroType*. Інакше, повертає *nullptr*.
- *collidesWithBonusItem()* - повертає об'єкт типу *GameplayItem* *, якщо даний об'єкт має колізію на графічній сцені з об'єктом типу *HealingType.* В іншому випадку повертає *nullptr.*

1.2.2 Абстрагування ігрових івентів

Зараз я розповім про досить цікавий реалізований мною механізм так званих ігрових івентів. Цей механізм може бути використаний для виконання деяких дій під час ігрового процесу при виконанні певних умов.

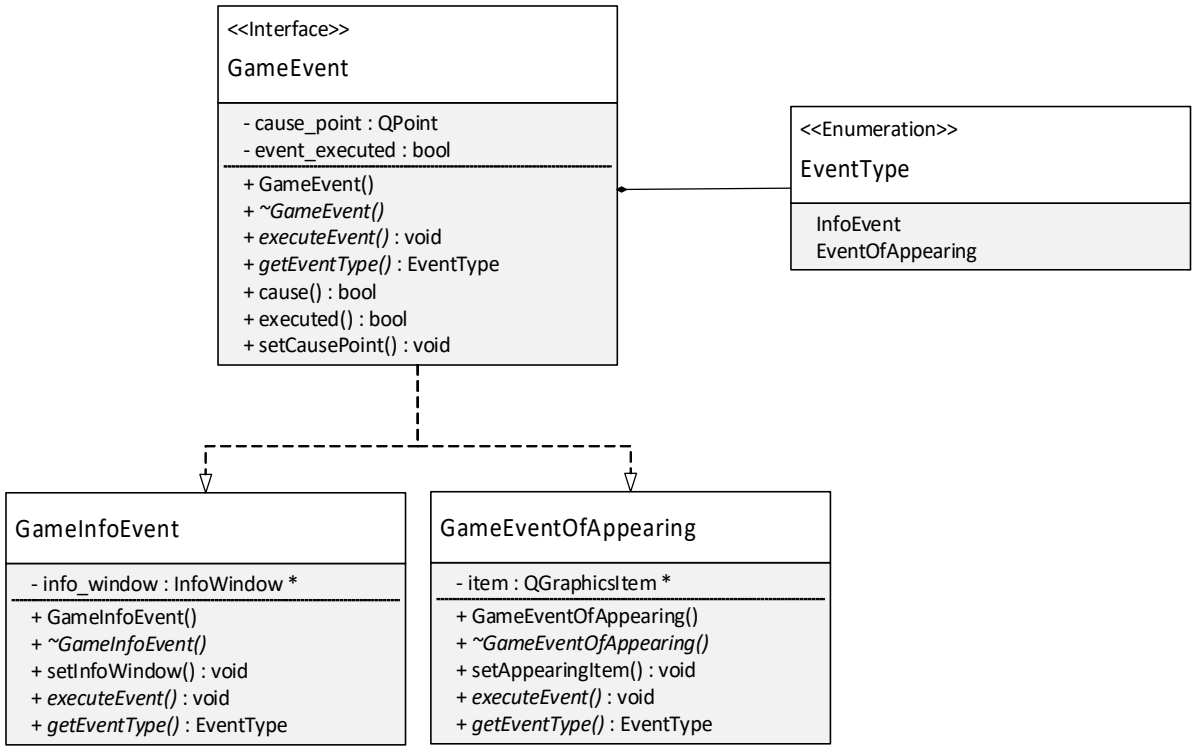

Рисунок 12 — UML-діаграма класів ігрових івентів

Спершу розберемо інтерфейс. Він містить об'єкт типу *QPoint*. На даний момент мої ігрові івенти мають лише один механізм фіксації виконання умови виконання — якщо точка входить до квадрату камери на сцені. У майбутньому я планую додати і інші способи такої фіксації. Наприклад, через таймер, тощо. Отже, кожен івент має точку, яка при потраплянні до квадрату камери призведе до того. Шо функція *cause()* поверне значення true. Той об'єкт, який займається перевіркою умови виконання цього івенту після цього повинен викликати функцію *executeEvent(),* у яку він повинен передати покажчик на графічну сцену. Це зроблено для того, щоб івент мав можливість реалізувати тебе на цій графчній сцені (тобто, додати якісь об'єкти на неї.)

Далі розглянемо клас *GameInfoEvent*. Даний івент розміщує об'єкт класу InfoWindow у квадраті камери при виконанні цього івенту. Я думаю

немає сенсу детально розглядати клас *InfoWinow*. Все, що нам треба про нього знати це те, що він реалізує вікно з підказкою.

Тепер про *GameEventOfAppearing*. Цей клас при виконанні своєї умови розміщує об'єкт item на графічну сцену. Координати об'єкта повинні бути задані заздалегідь.

Тепер розглянемо на прикладі, як користуватися івентами щоб ви зрозуміли, наскільки це полегшує процес програмування сюжету рівня.

```
    GameEventOfAppearing *enemy_1_apr = new GameEventOfAppearing;
    GameEventOfAppearing *enemy_2_apr = new GameEventOfAppearing;

    enemy_1_apr->setCausePoint({2050, 600});
    enemy_2_apr->setCausePoint({2050, 600});

    WarpEffect *enemy_1_warp = new WarpEffect(new Enemy());
    WarpEffect *enemy_2_warp = new WarpEffect(new Enemy());

    enemy_1_warp->setPos(2000, 400);
    enemy_2_warp->setPos(2000, 800);

    enemy_1_apr->setAppearingItem(enemy_1_warp);
    enemy_2_apr->setAppearingItem(enemy_2_warp);

    events.push_back(enemy_1_apr);
    events.push_back(enemy_2_apr);

    GameInfoEvent *first_enemies_event = new GameInfoEvent();
    first_enemies_event->setCausePoint({2055, 600});
    InfoWindow *i2_window = new InfoWindow(350, 200);
    i2_window->setInfoText("It looks like you have just met your first
enemies. Be carefull!\n"
                           "Try to use clever tactic and not to take
damage");
    first_enemies_event->setInfoWindow(i2_window);
    events.push_back(first_enemies_event);
```

Лістинг 20 — створення івентів з'явлення перших двох сутпротивників та івенту інформування про це

Синтаксис є досить простим та інтуїтивно зрозумілим. Думаю,що тут пояснювати майже нічого не треба. Спочатку ми створюємо 2 івенти з'явлення. Далі задаємо їм точки, які при потраплянні до квадрату камери викличуть виконання івента. Потім створюємо об'єкти *WarpEffect* – це клас, який реалізує анімацію з'явлення об'єкта (щось на кшталт анімації варпу протосів з гри *StarCraft* 2). Задаємо позицію, у якій ці об'єкти будуть

розміщені на сцені. І після цього додаємо їх до івенту. Наприкінці ми додаємо івент до списку івентів на виконання.

Після цього створюємо інформаційний івент, який сповіщатиме користувача про те, що з'являються супротивники. Тут все також дуже просто.

Даний спосіб створення, зберігання і відтворення івентів є зручним та гнучким. Він дозволяє додавати івенти як на етапі програмування рівня, так і прямо під час ігрового процесу. У майбутньому я планую додати ще декілька видів івентів і переробити ті, що є.

2. Отримана програма

2.1 Аналіз роботи

В результаті нашої розробки ми отримали програму, яке представляє собою деяку “демонстраційну” версію гри. На даний момент у програмі немає навіть одного дороблено до кінця рівня. У майбутньому, як я і казав раніше, у моєї гри буде повноцінний сюжет і декілька рівнів.

Гра працює досить стабільно. Я подбав про те, щоб користувач ніяк не міг спричинити нестабільну поведінку у програмі. Програма дає середнє значення навантаження на процесор. На прикладі мого процесору можу описати навантаження на нього під час виконання. Мій процесор
Intel Core i7-5500u. Під час стану меню програма навантажує процесор на ~10%. Чому так багато? Насправді, якщо прибрати ефект opacity у QPainter під час промалювання логотипу і дати з часом, то навантаження на процерор буде близько ~1%. Напівпрозорість стала однією з моїх головних проблем оптимізації у цій програмі.

Провівши декілька тестів я прийшов до висновку, що задання прозорості подовжує процес малювання на ЦІЛИХ 7 мс! Ви тільки подумайте, наскільки це багато. Саме через цю причину мені довелось повністю відмовитись від напівпрозорих ефектів у самій грі.

2.2 Недоліки і плани на майбутнє

На даний момент розробки, як я вже і казав, програма має багато недоліків. Серед них я виділю основні і ті, які потребують скорішого косплексного рішення і найбільної кількості роботи:

- Процесорна оптимізація. На даний момент програма не є оптимізованою на задовільному рівні. Всі дії і обчислення виконуються в одному потоці. Дуже багато таймерів, які, працюючи на малих тактах, сповільнюють роботу процесора.
- Неоптимальні медіафайли. Багато картинок, які також призводять до сповільнення роботи. Як мінімум це фонова карта розміром 12000 пікселів на 12000 пікселів. У майбутньому треба буде оптимізувати промальовку фону таким чином, щоб малювалася лише невелика облать, а також завантажувалася лише частина карти.
- Геймдизайн. Це складний і трудоємний процес, на який треба витратити дуже багато часу. В першу чергу пілся процесорної оптимізації я хочу зробити карту гарною. Для цього я створю ще декілька видів декорацій, які будуть динамічно змінюватися. Також я хочу додати певний рух на фоні карти, як на картах у грі StarCraft 2. Після цього я хочу зайнятися ретельною розробкою сюжету.

3. Список використаних джерел

При розробці програми я використовував наступні джерела інформації:

- http://doc.qt.io/ - документація Qt
- https://stackoverflow.com/ - відповіді на деякі питання, що виникали у мене при розробці
- http://en.cppreference.com/w/ - вирішення всіх проблем на рівні мови програмування, які у мене виникали
- http://www.strille.net/tutorials/part1_scrolling.php — невелике пояснення принципу реалізації руху камери по карті
- http://www.freesfx.co.uk/sfx/ - звідси я брав деякі звукові ефекти
- https://evileg.com/ru/ - завдяки цьому сайту я навчився працювати з графікою в Qt, а особливо з графічною сценою, а також на ньому є невеликий приклад гри-шутера, який став мініатюрною основою при розробці моєї гри
- На жаль, першоджерела всіх картинок, які я брав з інтернету, загубилися, оскільки для їх завантаження я використовував виключно інтерфейс Google-картинок.

  Це всі основні використані мною джерела. Багато чого у плані реалізації моментів геймплею, таких, як івенти, я брав у себе з голови. Тому не знаю, чи є більш правильні варіанти, перевірені часом і розробниками.