高级程序语言设计

实验报告

南开大学 人工智能学院
姓名：彭东阳
学号：2010920
班级：智能
2023年5月11日

目录

高级程序语言设计大作业实验报告

一个基于 Qt6.5 框架开发的音乐游戏

1 项目选题背景

音游是一种伴随着交互技术发展而产生的游戏，通过按照音乐节奏点击或操作游戏屏幕上的音符，来达到得分或通关的目的。常见的音游类型包括节奏游戏、舞蹈游戏、打击乐游戏、音乐盒等，其中节奏游戏最为普及。音游的历史可以追溯到 20 世纪 80 年代，当时的游戏机上出现了一些音乐节奏相关的游戏，比如《跳舞革命》等，这些游戏受到了很多玩家的喜爱。随着计算机技术的发展，音游逐渐从游戏机转移到了 PC 平台，最早的 PC 音游是 1998 年的《$GuitarFreaks$》。此后还有 $Beatmaniaiidx$，$osu!$ 等游戏风靡一时。这些游戏不仅在音乐和视觉效果上达到了非常高的水准，还开创了全新的游戏模式，让玩家体验到更加丰富的音乐游戏乐趣。

对于玩家而言，音游不仅可以提高玩家的反应能力和手眼协调能力，还可以增强玩家对音乐的感知和欣赏能力。

目前市面上的音游产品非常丰富。在街机方面，以世嘉，万代南梦宫等为首的日本电子游戏开发商开创了诸如 $maimai$，$Ongeki$，太鼓达人等游戏，受到非常多玩家的追捧；而移动端方面随着手机和平板电脑的渲染机能提升，也涌现了一批玩法新颖并且素质优秀的游戏如 $Cytus$，$Arcaea$，$Project:Sekai$ 等。

音乐游戏的开发对人机交互性能有着非常高的要求。一是交互的及时性；二是交互的可玩性。而前者又是所有音游的基础，可以说，把握了前者就是把握了音游开发的一半。本项目也正是从开发高性能音游的角度出发，尝试完成一个具有高及时性的游戏功能挑战。

在一般的游戏开发中，我们都倾向于使用高渲染机能的游戏和图形开发引擎例如 $Unity$ 等。然而这些开发引擎上手有着一定的难度，许多第一次接触游戏开发引擎的开发者可能需要花费一到两个月的时间来学习才能熟悉并理解其中的主要运行机制。考虑到这个问题，本项目决定从一个使用非常广泛的桌面应用软件开发框架 Qt 6.5 出发，从头实现一个音游完整的游戏逻辑。

Qt 6.5 是截止这个项目完成的时间点最新的一代 Qt 框架，性能稳定，并且自 Qt 6 之后，Qt 对许多经典控件进行了非常大的优化。

2 开发环境与框架

编译工具组：$MingW-w64(Minimalist\ GNU\ on\ Windows,\ version\ 10.0)$

编译工具：$CMake\ version\ 3.24.2$
构建工具：$ninja.exe$
C++ 编译器：$c++.exe$
C 编译器：$gcc.exe$

IDE：$CLion\ 2022.3.3$
开发框架：$Qt\ 6.5.0$

3 课题要点

1、面向对象
2、多线程编程
3、对象池技术
4、Qt 信号与槽函数机制

4 项目实现流程

本项目的基本实现了音乐游戏的基本环节，包括游玩，参数设置以及铺面编辑三个主要部分。遵循一个模块使用一个 $QWidget$ 类实现的规则，项目整体的结构构建如下

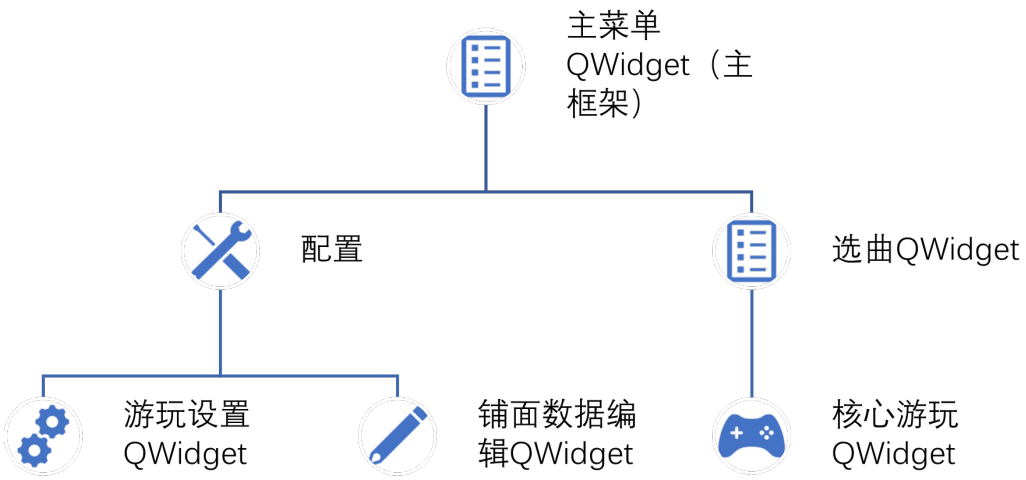

图 1: 项目整体框架图

4.1 主菜单 $QWidget$ 的基本实现思路

主菜单 $QWidget$ 是一个组织起其他所有功能 $QWidget$ 的一个框架 $QWidget$。它的基本组成如下：

```
//file: rg_mainmenu.h
class rg_mainmenu : public QWidget {
Q_OBJECT

public:
    explicit rg_mainmenu(QWidget *parent = nullptr);
    void hide_everything();
    void show_everything();
    void setup_myui();
    void anim_fadein();
    void anim_fadeout();
    ~rg_mainmenu() override;

private:
    Ui::rg_mainmenu *ui;
    rg *core_rg;
    options *core_opt;
    edit *core_edit;
    class select *select_menu;
    QPalette palette;
    QLabel *trans_anim;
    QPropertyAnimation *fadein;
    QPropertyAnimation *fadeout;
    QSequentialAnimationGroup *anim_group;
private slots:
    void pushbtn_slot();//play button
    void pushbtn_slot2();//options button
    void pushbtn_slot3();//back button
    void pushbtn_slot4();//edit button
    void play_songs(QString);
    void edit(QString);
signals:
    void reset_rg();
    };
```

这里我们可以看到私有成员中有许多自定义类的成员，包括core_rg，core_opt，core_edit，select_menu，

这些成员分别对应了框架图中的核心游玩 $QWidget$，游玩设置 $QWidget$，铺面数据编辑 $QWidget$ 以及选取 $QWidget$。公有成员中主要是一些设置外观的函数，包括该界面下按键的行为如`void hide_everything()`，界面整体 ui 的初始化`void setup_myui()`，过渡动画的行为例如`void anim_fadein()`等。而控件实例的定义都在 ui 文件中。这里我们同样给出 ui 头文件。

```
//file:ui_rg_mainmenu.h
class Ui_rg_mainmenu
{
public:
    QStackedWidget *stackedWidget;
    QPushButton *pushButton;
    QPushButton *pushButton_2;
    QPushButton *pushButton_3;
    QPushButton *pushButton_4;

    void setupUi(QWidget *rg_mainmenu)
    {
        if (rg_mainmenu->objectName().isEmpty())
            rg_mainmenu->setObjectName("rg_mainmenu");
        rg_mainmenu->resize(1359, 640);
        stackedWidget = new QStackedWidget(rg_mainmenu);
        stackedWidget->setObjectName("stackedWidget");
        stackedWidget->setGeometry(QRect(0, 0, 1360, 640));
        pushButton = new QPushButton(rg_mainmenu);
        pushButton->setObjectName("pushButton");
        pushButton->setGeometry(QRect(608, 400, 143, 23));
        pushButton_2 = new QPushButton(rg_mainmenu);
        pushButton_2->setObjectName("pushButton_2");
        pushButton_2->setGeometry(QRect(608, 450, 143, 24));
        pushButton_3 = new QPushButton(rg_mainmenu);
        pushButton_3->setObjectName("pushButton_3");
        pushButton_3->setGeometry(QRect(0, 0, 75, 24));
        pushButton_4 = new QPushButton(rg_mainmenu);
        pushButton_4->setObjectName("pushButton_4");
        pushButton_4->setGeometry(QRect(610, 500, 143, 24));

        retranslateUi(rg_mainmenu);

        stackedWidget->setCurrentIndex(-1);
        QMetaObject::connectSlotsByName(rg_mainmenu);
    } // setupUi
```

```
37      void retranslateUi(QWidget *rg_mainmenu)
38      {
39          rg_mainmenu->setWindowTitle(QCoreApplication::translate("rg_mainmenu",
            ↪  "rg_mainmenu", nullptr));
40          pushButton->setText(QCoreApplication::translate("rg_mainmenu", "Play",
            ↪  nullptr));
41          pushButton_2->setText(QCoreApplication::translate("rg_mainmenu", "options",
            ↪  nullptr));
42          pushButton_3->setText(QCoreApplication::translate("rg_mainmenu", "Back",
            ↪  nullptr));
43          pushButton_4->setText(QCoreApplication::translate("rg_mainmenu", "edit",
            ↪  nullptr));
44      } // retranslateUi
45  };
```

在 *ui* 头文件中,最重要的莫过于`QStackedWidget *stackedWidget`这个成员,这是一个存储 *QWidget* 的一个容器类，而在 *ui* 中，我们设置其大小恰好为整个窗口的大小，故而程序整体的可视位置都是这个`stackedWidget`，所有不同 *QWidget* 的切换，其实都是这个 *QWidget* 容器的显示内容切换。最终不同 *QWidget* 的显示效果可以参见下图

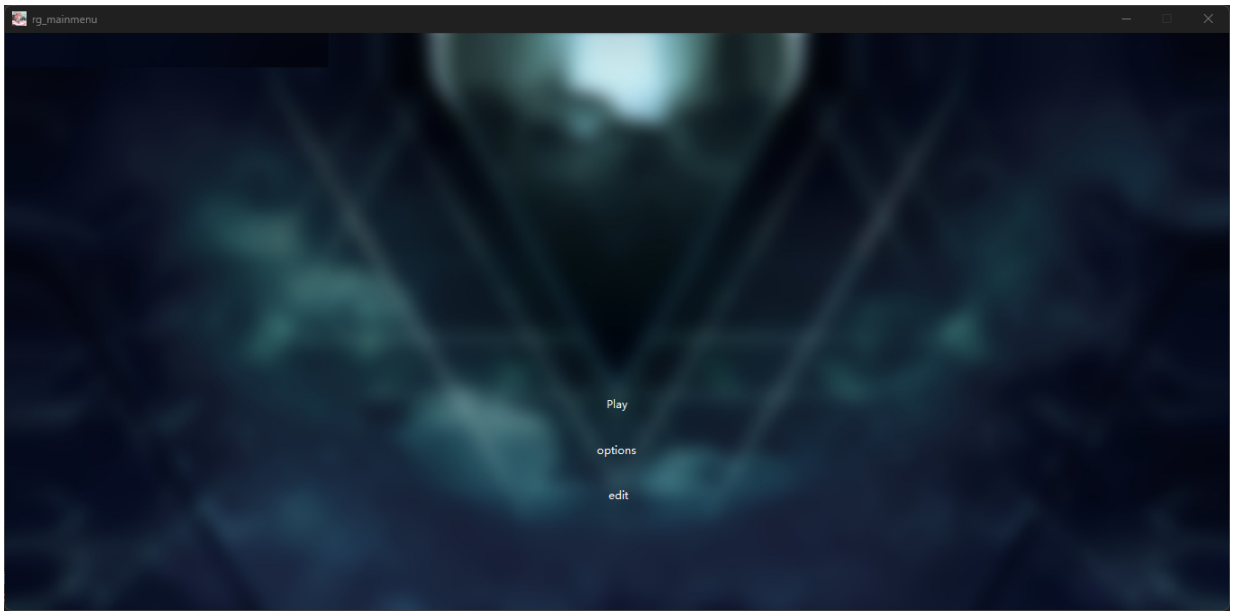

图 2: 主菜单的显示效果

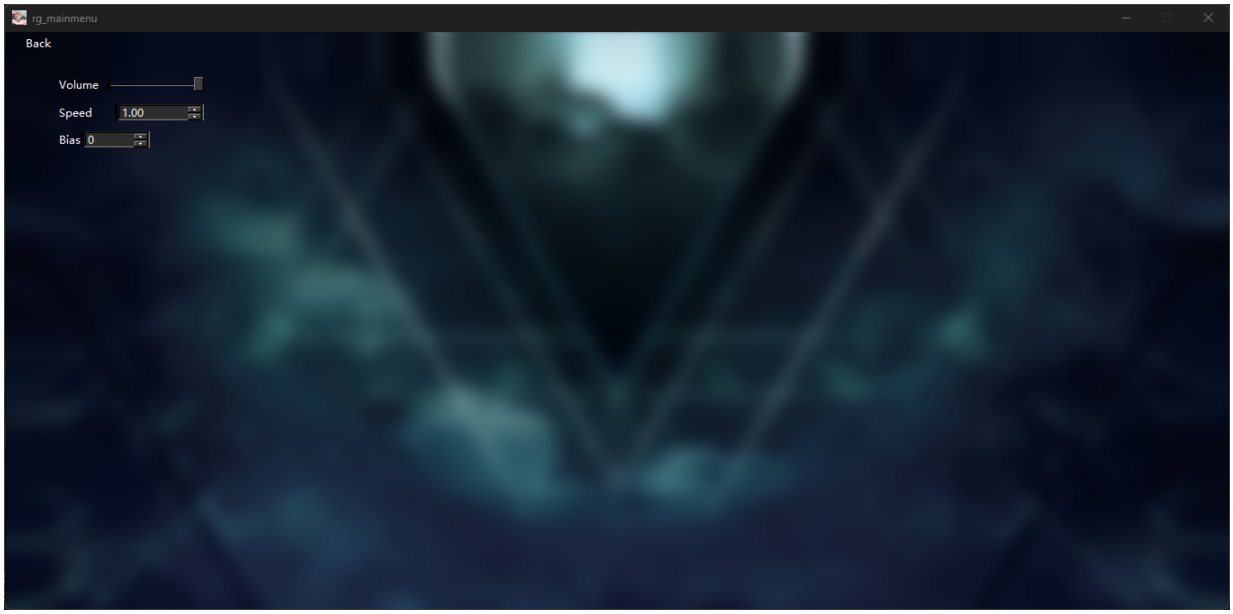

图 3: 设置界面的显示效果

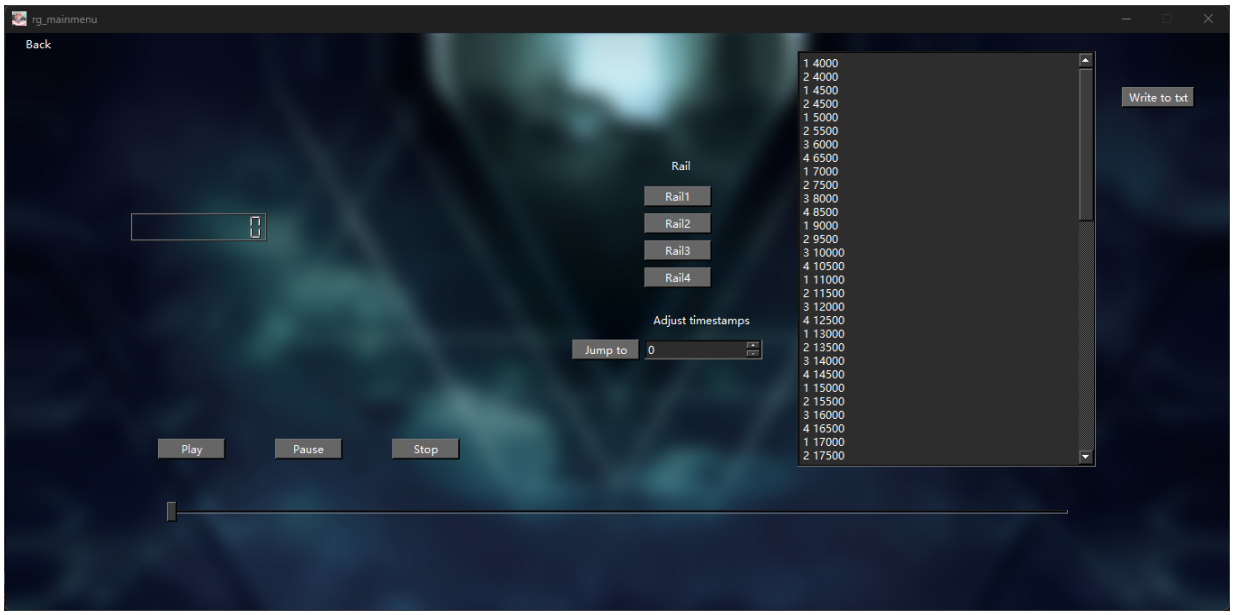

图 4: 铺面编辑界面的显示效果

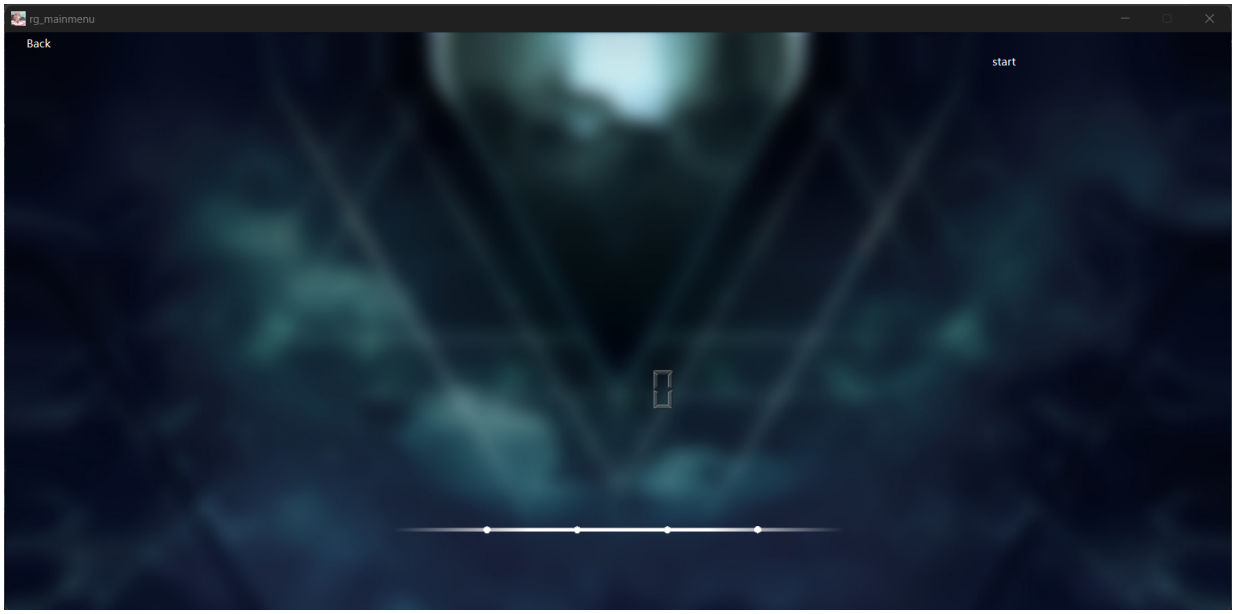

图 5: 游玩界面的显示效果

总而言之，rg_mainmenu这个 $QWidget$ 的核心作用就是将其他各个不同的 $QWidget$ 组织起来，并且在玩家选择不同的 $QWidget$ 功能时切换不同的 $QWidget$ 显示。rg_mainmenu也是我们主函数中调用的程序，是主线程。

4.2 选曲 $QWidget$ 的实现

选曲 $QWidget$ 的功能在于能够读取指定文件下的游戏资源文件，包括游戏音乐资源，铺面资源以及其他信息，然后将其全部显示出来。这里我们首先给出代码进行分析：

```
//file: select.h
struct artist_bpm{
    QString artist;
    int bpm;
};

class select : public QWidget {
Q_OBJECT

public:
    rg* bind_rg;
    explicit select(QWidget *parent = nullptr);
    QListWidget songlist_widget;
    void add_songs(QString name);
    void display(QListWidgetItem* item);
```

```
    std::unordered_map<QString,artist_bpm> songinfo_hashmap;
    bool is_edit=false;
    void set_rg(rg*);
    ~select() override;

private:
    Ui::select *ui;

signals:
    void start_song(QString);
    void edit_song(QString);
};
```

从头文件中可以看到选曲 *QWidget* 中的一些重要成员包括存放曲目信息的`songinfo_hashmap`，与之绑定的`bind_rg`以及显示的控件`songlist_widget`。

`songinfo_hashmap`使用的基本数据结构是一个哈希表，键是`QString`类型，用于存放曲目的名字，值是`artist_bpm`自定义结构体类型。使用哈希表而不是简单的队列的好处在于哈希表有着更高效的查询能力，并且由于哈希冲突的存在，最终的程序不会出现因为误操作导致的重名游戏资源出现。

在点击 *play* 按键之后，正常的游玩之前，选曲 *QWidget* 就会出现，让用户选择自己想要游玩的曲子进行游玩，这时游玩 *QWidget* 就会加载对应选择的曲子并且进行一些预操作。这里为了便于后期更好调用核心游玩类 *QWidget* 的一些成员函数，我们直接添加了一个`rg`类型的指针成员`bind_rg`来直接实现对单例的游玩 *QWidget* 的控制。

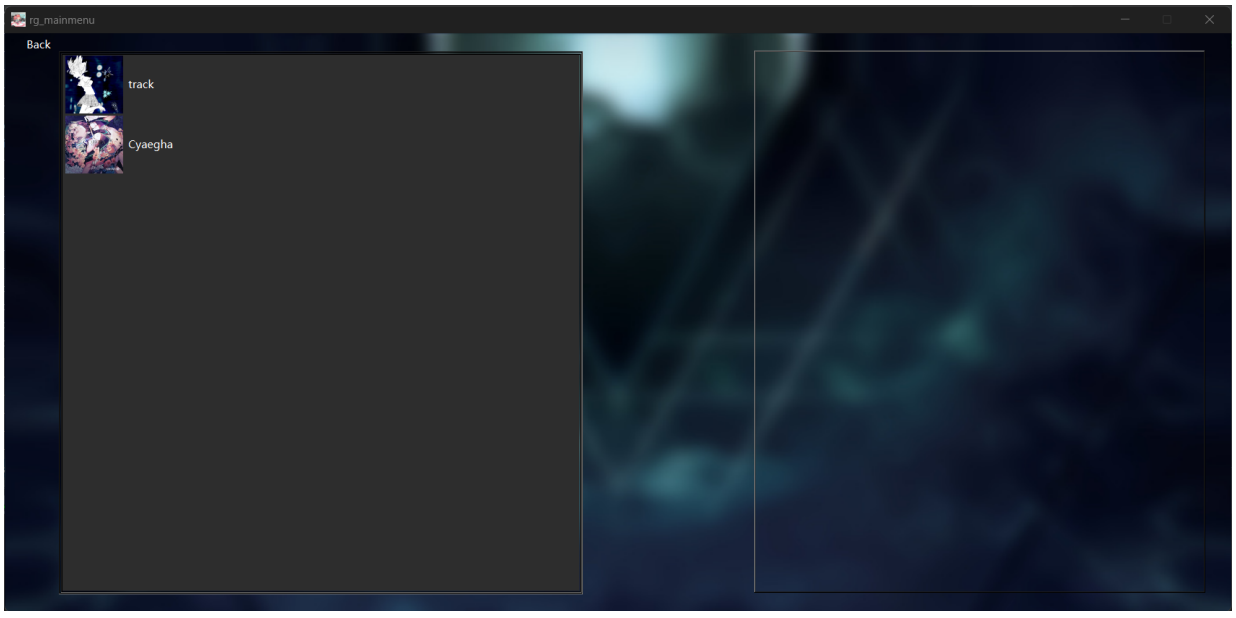

图 6: 选取界面的显示效果

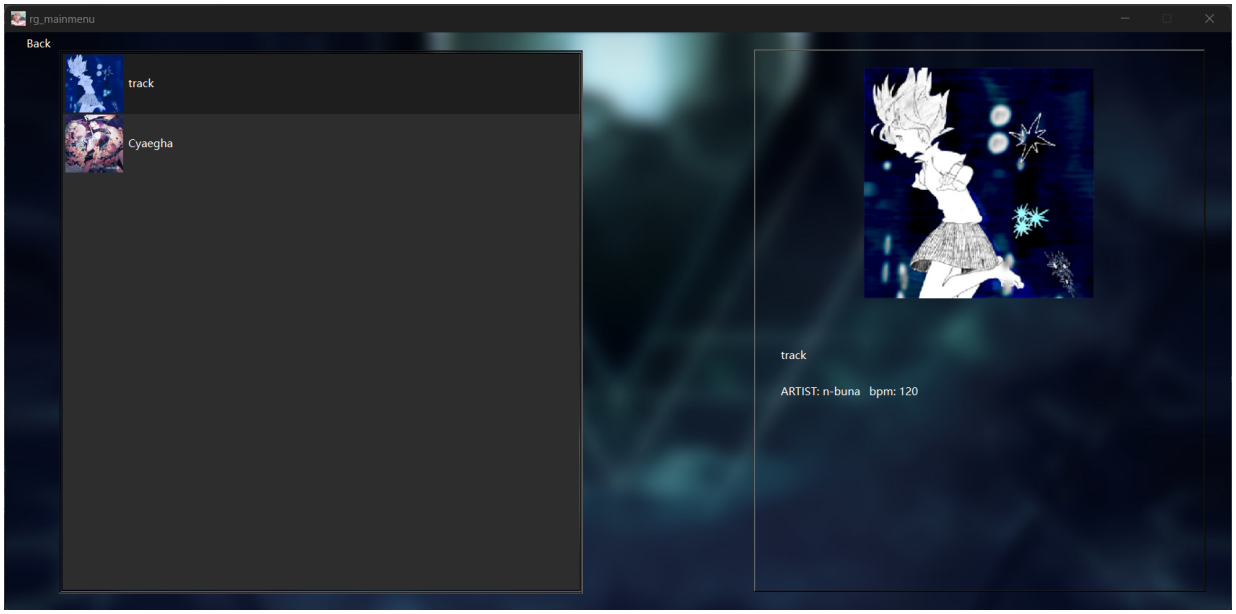

图 7: 选取界面单击对应曲目的显示效果

与之形成对比的是，当我们在进行铺面编辑时，也需要选择一个曲子进行铺面编辑。设计的时候便复用了这一部分的 $QWidget$，不过铺面编辑之前并没有很多的预处理，故而我们并没有像游玩一样添加一个对应edit类型的指针成员，而是直接使用信号与槽函数通信的方式。在上面的源码中，signals信号成员就对此有所体现。

关于选曲 $QWidget$ 的文件读取与加载部分，我们也给出部分实现的代码如下：

```
//file: select.cpp
select::select(QWidget *parent) :
    QWidget(parent), ui(new Ui::select) {
ui->setupUi(this);

songlist_widget.setIconSize(QSize(64,64));
ui->scrollArea->setWidget(&songlist_widget);
ui->scrollArea->show();
//emit signals to execute some actions in rg or edit when double-click a item
connect(&songlist_widget,&QListWidget::itemDoubleClicked,[=](QListWidgetItem
    *item){if(!is_edit)emit start_song(item->text());else{is_edit=false;emit
    edit_song(item->text());}});
connect(&songlist_widget,&QListWidget::itemClicked,[=](QListWidgetItem *item){
    display(item);});

std::fstream file;
file.open("../songs/songinfo.txt");//songinfo.txt is a file including all the
    available songs. program will read names from songinfo.txt
```

```
//and load songs via names. all source file are renamed in a regular form.
while(true)
{
    char name[50],artist[50];
    int bpm;
    file>>name>>artist>>bpm;
    if(strcmp(name,"eof")==0)break;
    else
    {
        artist_bpm ab;
        ab.artist=QString(artist);
        ab.bpm=bpm;
        songinfo_hashmap.insert(std::pair<QString,artist_bpm>(name,ab));
        add_songs(QString(name));
    }
}
file.close();
}

void select::add_songs(QString song) {
QString base_path="../songs/";

QPixmap bg;
bg.load(base_path+song+QString("_bg.png"));

QListWidgetItem* aitem;
aitem=new QListWidgetItem;
aitem->setIcon(QIcon(bg));
aitem->setText(song);

songlist_widget.addItem(aitem);
}
```

为了方便，我们规定将一个曲目的音频文件命名为songname.mp3，对应的铺面文件为songname.txt，封面图片为songname_bg.jpg，这样只需要程序从songinfo.txt读出可用曲名，就可以根据曲名直接读取对应的资源文件。然后将这些曲目加入到 $QListWidget$ 显示控件中，得到最终的显示效果。

上文提到，选曲 $QWidget$ 被游玩 $QWidget$ 和编辑 $QWidget$ 公用，也就是说选曲 $QWidget$ 在接收到双击曲目进入功能信号的时候，要能够辨别什么时候进入编辑 $QWidget$，什么时候进入游玩 $QWidget$，这里我们通过一个简单的bool类型变量实现功能的选择。成员 is_edit 为 $true$ 时，双击信号会触发edit_song()信

号，进入编辑功能，反之则触发start_song() 信号，进入游玩功能。关于变量is_edit的管理，我们会在后续讲解编辑 $QWidget$ 时提及，这个值默认是 $false$。

在本节的最后，我们给出这部分界面结构设定的代码。

```
//file: ui_select.h
class Ui_select
{
public:
    QScrollArea *scrollArea;
    QWidget *scrollAreaWidgetContents;
    QFrame *frame;
    QLabel *bglabel;
    QLabel *songnamelabel;
    QLabel *songinfolabel;

    void setupUi(QWidget *select)
    {
        if (select->objectName().isEmpty())
            select->setObjectName("select");
        select->resize(1360, 640);
        scrollArea = new QScrollArea(select);
        scrollArea->setObjectName("scrollArea");
        scrollArea->setGeometry(QRect(60, 20, 581, 601));
        scrollArea->setWidgetResizable(true);
        scrollAreaWidgetContents = new QWidget();
        scrollAreaWidgetContents->setObjectName("scrollAreaWidgetContents");
        scrollAreaWidgetContents->setGeometry(QRect(0, 0, 579, 599));
        scrollArea->setWidget(scrollAreaWidgetContents);
        frame = new QFrame(select);
        frame->setObjectName("frame");
        frame->setGeometry(QRect(830, 19, 500, 600));
        frame->setFrameShape(QFrame::StyledPanel);
        frame->setFrameShadow(QFrame::Raised);
        bglabel = new QLabel(frame);
        bglabel->setObjectName("bglabel");
        bglabel->setGeometry(QRect(122, 20, 255, 255));
        songnamelabel = new QLabel(frame);
        songnamelabel->setObjectName("songnamelabel");
        songnamelabel->setGeometry(QRect(30, 330, 450, 16));
        songinfolabel = new QLabel(frame);
        songinfolabel->setObjectName("songinfolabel");
```

```
        songinfolabel->setGeometry(QRect(30, 370, 450, 16));

        retranslateUi(select);

        QMetaObject::connectSlotsByName(select);
    } // setupUi

    void retranslateUi(QWidget *select)
    {
        select->setWindowTitle(QCoreApplication::translate("select", "select",
        ↪  nullptr));
        bglabel->setText(QString());
        songnamelabel->setText(QString());
        songinfolabel->setText(QString());
    } // retranslateUi

    };
```

从 *ui* 文件不难看出，界面显示中的选择列表使用的是 *QListWidget* 控件，而信息的显示选择的是 *QLabel* 控件。

4.3 核心游玩 *QWidget* 的实现

这一部分是整个游戏的核心部分，是游戏的主体功能实现部分。我们首先给出游戏框架类rg部分的头文件代码：

```
//file: rg.h
class rg : public QWidget {
    Q_OBJECT

public:
    explicit rg(QWidget *parent = nullptr);
    void begin();
    note_generator* gen;
    note *v_note;
    ~rg() override;
    QLabel line;
    void kill(int);
    music_thread *mt;
```

```
    QAudioOutput out;
    QMovie *sprite1;
    QMovie *sprite2;
    QMovie *sprite3;
    QMovie *sprite4;
    QMovie *far1;
    QMovie *far2;
    QMovie *far3;
    QMovie *far4;
    void set_song(QString);
    void reset_lcdCombo();
private:
        Ui::rg *ui;
        void keyPressEvent(QKeyEvent *event) override{
            if(event->key()==Qt::Key_R)
                kill(1);
            else if(event->key()==Qt::Key_T)
                kill(2);
            else if(event->key()==Qt::Key_Y)
                kill(3);
            else if(event->key()==Qt::Key_U)
                kill(4);
}


private slots:
    void set_volume(float);
    void pushbtn_slot();
        };
```

这一部分的头文件包含了主要的动画特效（或者称之为 *sprite*），判定线的显示等，这些都是可视化资源文件。除此之外，还有一些游玩控制类成员主要包括

```
    note_generator* gen;
    note *v_note;
    music_thread *mt;
```

其中note_generator* gen是音符生成类，这个类是一个管理音符生成，开始与结束运动以及回收的一个类；note *v_note是一个音符类的虚拟实例，音符类是管理游戏中音符表现，动画等特性的类，在我们游玩过程中，一个音符就是一个这个类的实例，而这里的虚拟实例则是为了设置音符类的一些静态成

员而声明的成员，本身和我们游戏中的音符不是一个东西。而最后的music_thread *mt则是管理音乐的播放，铺面读取与解析以及音符什么时候该生成的一个类，这是一个不同于主线程的一个子线程，后文我们会具体介绍。

另外，我们也在这一部分完成了按键监听事件函数keyPressEvent()，接收玩家的键盘信号，然后调用kill()函数来实现击中音符。关于kill()函数，我们将在介绍完后文note_generator类介绍。

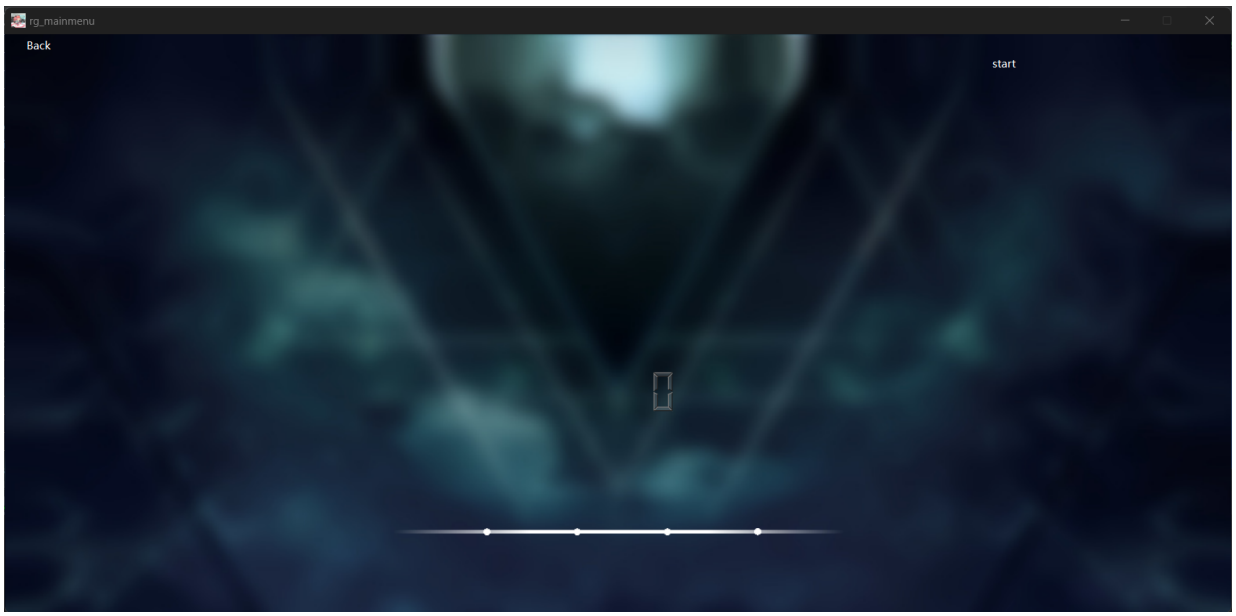

图 8: 游玩界面的显示效果

4.3.1 note 类及 note_generator 类介绍

接下来为了便于理解，我们首先介绍音符类note。头文件见下：

```
//file: note_generator.h
class note:public QObject
{
Q_OBJECT
private:
    static QPixmap* src_skin;
    QPropertyAnimation anim;
    note_generator* gen=nullptr;
    int rail=0;
    static int duration;
    QElapsedTimer timer;
public:
    QLabel *entity;
    bool ready_to_be_hit=false;
```

```
15      bool is_hit=false;
16      bool is_perfect=false;
17      double posx=0,posy=0,velocity=5;
18      bool recycled=false;
19      void setSkin();
20      void setAnim();
21      void start_anim();
22      note(note_generator*,QWidget*,QObject* parent=nullptr);
23      void show();
24      void setRail(int);
25      void setpos(double x,double y);
26      void recycle();
27      void stop();
28      void setDuration(int d);
29  public slots:
30
31  signals:
32      void me(int r);
33  };
```

首先需要指出的是，note类是一个继承自QObject的一个子类，QObject 本身是一个抽象类，没有显示的能力。要想让音符能够在平面中显示出来并且运动，必须为它设置一个显示的控件，这里我们选择了QLabel *entity作为音符可视化的实体基础。通过给entity添加材质文件以及对应的动画，就实现了我们游玩过程中音符自上而下运动的效果。

图 9: 游玩时的效果

关于音符的运动实现方式，最初的设计方式是通过为每一个音符分配一个线程，使用线程来控制每一个时间点音符的位置（也就是entity的位置）来实现，然后多个音符的多个线程用线程池技术实现动态管理以及回收，这样子的好处就是每个音符完全独立，可以有更多的不同的运动表现效果，但是实际程序运行过程中，可能会出现同时有 7 到 8 个的线程实例并发运行，一方面这样并发切换线程的执行会导致时延的出现，音符会不合拍；另一方面，小型的个人计算机运行起来非常吃力，后来放弃了这个方案。选择了使用动画更为简单，性能更加高效的QPropertyAnimation来实现音符的运动效果。

此外，音符类的成员中还有若干布尔变量，用于判定音符是否为完美击中或者未击中。这些变量的管理与计时器成员QElapsedTimer timer有关，当音符启动之后，计时器会根据当前已经运动的时间，决定如果当前时刻被击中应该判定为完美判定/模糊判定/未击中判定三种状态。这一部分在后文给出的代码中有相关体现。

游戏过程中，note_generator是一个单例。为了更方便控制音符，每一个音符也绑定了这个单例。关于音符的初始化和判定，我们给出相关的代码如下：

```
//file: note_generator.cpp
note::note(note_generator* g,QWidget* fa_pt,QObject* parent):QObject(parent)
{
    connect(this, &note::me,g, &note_generator::expire_out_list);
    entity=new QLabel(fa_pt);
    setAnim();
    entity->move(0,0);
    gen=g;
}


void note::recycle() {//when note is hit, call this function to recycle
    anim.stop();
    timer.invalidate();
    gen->note_pool->out_list();
    ready_to_be_hit=false;
    is_hit=false;
    //entity->move(0,-20);
    entity->hide();
    //QMutexLocker lock(&_stack::lock);
    gen->note_stack.push(this);
}


void note::setAnim() {//setup the animation of note
    if(anim.state()==QAbstractAnimation::Running)
        anim.stop();
```

```
    anim.setPropertyName("pos");
    anim.setTargetObject(entity);
    anim.setStartValue(entity->pos());
    anim.setEndValue(QPoint(posx,posy+600));
    anim.setDuration(duration);//base duration is 1500, speed is 0.4px/ms
}

void note::start_anim() {//when note is ready, call this function to start moving.
    int base=base_prepare_time;
    if(!recycled){
        setAnim();
        recycled=true;
    }
    else{
        anim.setStartValue(entity->pos());
        anim.setEndValue(QPoint(posx,posy+600));

    }
    if(timer.isValid())
        timer.start();
    else timer.restart();//base prepare time is 1375ms
    QTimer::singleShot(base-75,[=](){ready_to_be_hit=true;});
    QTimer::singleShot(base-50,[=](){is_perfect=true;});
    QTimer::singleShot(base+50,[=](){is_perfect=false;});
    QTimer::singleShot(base+75,[=](){emit me(rail);});
    anim.start();
}
```

QTimer::singleShot是一个一次性的计时器，通过这个计时器我们在不同的时间点更新音符当前被击中的判定状态。这个计时器在音符开始运动的时候被启动。关于回收函数，主要的操作是将音符的运动动画以及其他参数重置，然后放回对象池。

接下来引入note_generator类，我们依然首先给出头文件的代码进行分析：

```
class note_generator:public QObject
{
Q_OBJECT
private:
    QWidget *parent;
    _list note_pool[5];//0 is not used
    _stack note_stack;
```

```
    QTimer clock;
public:
    note_generator(){

    };
    void generate_note(int railSeq);
    void read();
    void set_Parent(QWidget* w){parent=w;};
    bool is_empty(int r) { return note_pool[r].isEmpty();};
    note* get_first(int r){return note_pool[r].getFirst();};
    friend note;
    void clear();
public slots:
    void expire_out_list(int r);
    void generate_by_music(int r);
signals:
    void combo_break();
};
```

相比之下，note_generator类的定义比note要简洁很多。在此之前，我们先简要介绍对象池技术

对象池技术是一种常用于游戏中的资源管理技术。对于某种重复性非常高，运行过程中使用量非常大的元素，我们通常都使用这种方式来管理资源。以本项目的音符实例为例。在游戏过程中，一首曲目一般前后会涉及到 400 个音符实例，但是实际上如果我们前后一共实例化 400 个音符对象，对于计算机的渲染是一个非常大的负担，而且不必要的反复生成和销毁会对游戏性能带来影响；为了改善这个问题，我们在每次音符对象使用完之后（也就是被击中之后），并不马上销毁，而是放入到一个对象池中回收；每次需要生成的时候，直接从对象池中取出一个实例而不用调用构造函数，这样就节省了反复构造和销毁的时间与资源；在从对象池取实例的时候，如果对象池为空，那么就调用构造函数构造一个新的放入对象池。

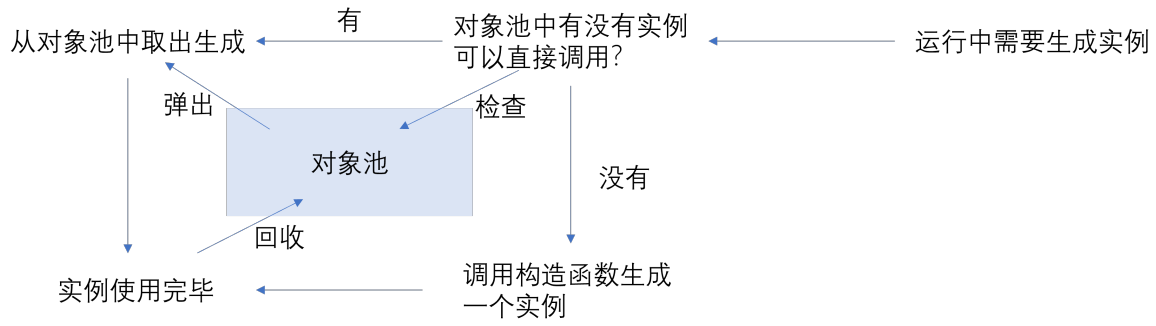

图 10: 对象池的调用流程

对象池的实现基础是一个承接特定元素的容器，由于对象的取出不需要顺序，这里使用队列或者栈都可以实现。利用 Qt 自带的容器类 `QList`即可实现一个存放音符对象的对象池。

```
//file: note_generator.h
class _stack//a simple stack to store QObject
{
public:
    _stack() {}
    static QMutex lock;
    void push(note *obj) {
        stack_.append(QPointer<note>(obj));
    }

    note *pop() {
        if (stack_.isEmpty()) {
            return nullptr;
        }
        note *obj = stack_.last().data();
        stack_.removeLast();
        return obj;
    }

    bool isEmpty() const {
        return stack_.isEmpty();
    }

    int size() const {
        return stack_.size();
    }

    note* at(int seq) const{
        return stack_[seq];
    }
    void clear(){
        stack_.clear();
    }
private:
    QList<QPointer<note>> stack_;
    };
```

这也是上文中`note_generator`类中`note_stack`的类型定义。

另一方面，为了保证不同轨道之间音符的独立性，我们为每个轨道都设置了一个队列，这个队列用于存放该轨道中正在运动的音符对象，采用队列的数据结构是考虑到先进入轨道的音符应该先被击中，也就是 $FIFO$ 的原则。

```
//file: note_generator.h
class _list//a simple list to store note
{
public:
    _list() {}
    static QMutex lock;
    void push_back(note *obj) {
        list_.append(QPointer<note>(obj));
    }

    note *out_list() {
        if (list_.isEmpty()) {
            return nullptr;
        }
        note *obj = list_.first().data();
        list_.removeFirst();
        return obj;
    }
    note* getFirst(){
        return list_.first();
    }

    bool isEmpty() const {
        return list_.isEmpty();
    }

    int size() const {
        return list_.size();
    }
    note* at(int seq) const{
        if(seq<list_.size())
            return list_.at(seq);
        else return nullptr;
    }
    void clear(){
        list_.clear();
    }
```

```
38
39 private:
40     QList<QPointer<note>> list_;
41     };
```

介绍完前面的所有工作后，我们接下来就可以按照上面对象池调用流程图的步骤给出每一步是怎么实现的。首先是音符的生成。note_generator在接收到生成的信号之后会调用自己的音符生成函数：

```
1  //file: note_generator.cpp
2  void note_generator::generate_note(int railSeq) {
3      note* n=nullptr;
4      QMutexLocker lock(&_list::lock);
5      QMutexLocker Lock(&_stack::lock);
6
7      if(note_stack.isEmpty())
8          n=new note(this,parent);
9      else
10         n=note_stack.pop();
11     note_pool[railSeq].push_back(n);
12     n->setRail(railSeq);
13     n->setpos(480+19+100*(railSeq-1),0);
14     n->show();
15     n->start_anim();
16     }
```

由于音符生成和音符消去的操作分别是由两个线程引起（音符生成信号由线程music_thread发射，音符消去kill()是主线程的函数），而对象池是两个线程共享的资源，这里我们需要为对象池加上互斥锁来保证同步访问，避免出现竞争现象。这里可以看到生成函数的第一步是给对象池上锁，保证这个时间点如果有音符生成信号则阻塞等待。然后检查对象池是否为空，如果为空，则通过new来实例化一个新的note；否则直接使对象池弹栈来获取一个回收的实例，进行后续的初始化。

在这样的条件下，每当一个音符被要求在某个轨道生成，这个音符实例就会被加入到对应轨道的队列中；每次用户按下按键之后，首先从队列头里取出一个音符实例，检查其当前状态是否为可被击中，也就是其成员ready_to_be_hit的值是否为 $true$，如果是则将其从队列头移除；否则将其放回头部。这也正是我们前面所提及的kill()函数完成的功能。

```
1  //file: rg.cpp
2  void rg::kill(int r) {
3      note* n=nullptr;
4      if(!gen->is_empty(r))
```

```
        n=gen->get_first(r);
    if(n!=nullptr&&n->ready_to_be_hit)
    {
        n->stop();
        n->entity->hide();
        switch (r) {
            case 1: sprite1->start();break;
            case 2: sprite2->start();break;
            case 3: sprite3->start();break;
            case 4: sprite4->start();break;
            default:break;
        }
        n->is_hit=true;
        ui->lcdCombo->display(ui->lcdCombo->value()+1);

        if(n->is_perfect==false)
        {
            switch(r){
                case 1: far1->start();break;
                case 2: far2->start();break;
                case 3: far3->start();break;
                case 4: far4->start();break;
                default:break;
            }
        }
    }

    }
```

音符被击中之后，按照对象池调用流程，这个音符应该被回收，回收的函数如下：

```
file: note_generator.cpp
void note::recycle() {
    anim.stop();
    timer.invalidate();
    gen->note_pool->out_list();
    ready_to_be_hit=false;
    is_hit=false;
    entity->hide();
    QMutexLocker lock(&_stack::lock);
```

```
    gen->note_stack.push(this);
}
```

这里我们看到回收函数首先将音符停止并隐藏，然后让轨道音符队列的音符从队首出队，重置相关参数，最后上锁压回对象池即可。

至此音符的整个生成和回收的调用流程已经全部介绍完毕。

4.3.2 music_thread 类介绍

前面已经提及，music_thread类是一个线程类，与主线程并发执行。这里我们同样先给出实现代码

```
//file: music.h
class music_player:public QMediaPlayer{
private:
    QString current_song;
public:
    void set_current(QString);
    music_player();
    QList<int> *note_rail_info;
    QList<int> *time_info;
    void read(QString s);
};

class music_thread:public QThread{
    Q_OBJECT
private:
    note_generator *gen;
    int prepare_ms;
    int bias_ms;
    QElapsedTimer *timer;
    QString current;
public:
    music_player *music;
    music_thread(note_generator* g,QWidget* parent,QAudioOutput *output);
    void mstop();
    void reset();
    void run() override{
        prepare_ms=base_prepare_time;
        bias_ms=bias_time;//base_prepare_time and bias_time are global variables
        int rail=0,time=0;
```

```
        //music->read(current);
        music->play();

        if(timer->isValid())
            timer->start();
        else
            timer->restart();

        while(!QThread::isInterruptionRequested()){
            double now=timer->elapsed();
            if(!rail&&!time&&!music->note_rail_info->isEmpty())
            {
                rail=music->note_rail_info->takeFirst();
                time=music->time_info->takeFirst();
                time+=bias_ms;
            }
            else if(music->note_rail_info->isEmpty()) break;
            if(time<=now+prepare_ms+3&&time>=now+prepare_ms-3){
                emit generate(rail);
                rail=0,time=0;
            }
        }
    };

    signals:
    void generate(int rail);
};
```

music_thread类主要有两个功能，一是控制音乐的播放，二是读取铺面数据并控制音符的生成。在代码实现中，我们首先继承了一个QMediaPlayer类定义了一个子类music_player作为我们这个游戏全局的音乐播放器，然后将其作为成员加入到music_thread类中。

为了读取铺面数据，music_thread中添加了两个列表成员：存放轨道信息的note_rail_info和存放时间信息的time_info，这两个队列是按序号对齐的，也就是同一个序号位置的信息是同一个音符的。

将这两个功能集成到一起主要为了能够更方便读取当前音乐播放的时间点，在对应的时间点生成音符；线程类music_thread的run()函数写成了一个死循环函数，每次循环都会检测音乐播放的位置然后判断是否应该生成音符。由于底层逻辑中，音乐的播放是一个新的线程（也就是说这里的music_player播放音乐是一个独立于music_thread和主线程之外的全新线程），没法获取当前播放的时间戳。于是我们直接内置一个QElapsedTimer timer计时器启动音乐的时候同步启动计时器的方式解决时间戳问题。通过函数timer.elapsed()获取当前音乐的播放位置时间点。

剩下的工作只剩下计算减去提前调用时间后的音符生成时间，然后不断比较计时器是否到达规定时间戳即可。必须要指出的时，原则上我们希望音符能够在一个准确的时间点被生成，但是实际上可能由于各种误差的原因，并不是每一个最小间隔的时间我们都可以执行一次时间点检测。比如，在这个项目中，最小的计算时间间隔是毫秒 (ms)，但是实际上，music_thread在不断运行的过程中，并不能做到每一个毫秒都检测一次当前时间是否到达音符的生成（启动）时间；上一次检测计时器读出来是 $100ms$，下一次读出来可能就是 $103ms$，这是不可避免的。

于是在实际处理的时候，我们将“在某一个时间点生成”改为“在某一个时间窗口生成”

```
if(time<=now+prepare_ms+3&&time>=now+prepare_ms-3){
    emit generate(rail);
    rail=0,time=0;
    }
```

这样子就尽可能避免了因为程序运行时间误差导致无法生成音符的问题。

4.4 设置 $QWidget$ 的实现

这一部分是一个参数调整的 $QWidget$ 模块，主要通过信号与槽函数的机制实现。头文件的类定义如下：

```
//file: options.h
class options : public QWidget {
    Q_OBJECT

    public:
        explicit options(QWidget *parent = nullptr);
        void bind(rg *bind);
        void set_volume(float);
        void set_bias(int);
        void set_speed(float);
        ~options() override;

    private:
        rg* bind_rg;
        int volume;
        int bias;
        double speed=1;
        Ui::options *ui;

    signals:
```

```
        void mvolume(float);
        void duration_changed(int);
    };
```

在设置 $QWidget$ 中，我们添加了三个可设置参数，分别是音乐的音量，音符的流速以及音符的偏移。音乐的音量可以直接通过调用 $QAudioOutput$ 输出的相关函数来改变，这里不再赘述，音符的流速是音符的移动速度改变（音符到达判定线的时间点不变），音符的偏移是指音符提前或者延后若干毫秒到达判定线的时间。音符的流速调整是通过改变 $QPropertyAnimation$ 的持续时间来改变的。

我们首先引入两个全局变量

```
//file: music.h
extern int base_prepare_time;
extern int bias_time;//global variables declare in music.h
```

前者是音符从生成到移动至判定线的时间，这里称为准备时间，后者则是音符时间戳的偏移时间。在设置 $QWidget$ 中，有

```
//file: options.h

options::options(QWidget *parent) :
        QWidget(parent), ui(new Ui::options) {
    ui->setupUi(this);
    note *virtual_note=new note(nullptr,nullptr);

    connect(ui->horizontalSlider,&QSlider::sliderMoved,this,[=](int a){ float
    ↪  vol=static_cast<float>(a)/102;set_volume(vol);});
    connect(ui->speedbox,&QDoubleSpinBox::valueChanged,[=](double a){
    ↪  set_speed(a);});
    connect(ui->biasbox,&QSpinBox::valueChanged,[=](int a){ set_bias(a);});
    connect(ui->speedbox,&QDoubleSpinBox::valueChanged,[=](double
    ↪  a){virtual_note->setDuration(1500/a);});

}

void options::set_speed(float t) {
    speed=t;
    base_prepare_time=1375/speed;
}

void options::set_bias(int t) {
```

```
    bias=t;
    bias_time=bias;
}

//file: note_generator.cpp
void note::setDuration(int d) {
    duration=d;
    qDebug()<<duration;
}

```

结合此处的代码块以及上文其他部分的代码，不难得到设置 $QWidget$ 修改游戏参数的机制。这里我们就不重复引用上文的代码赘述了。

4.5 铺面编辑 $QWidget$ 的实现

游戏过程中，音符何时开始生成，从什么位置生成，这些都是由铺面文件决定的。简单期间，在本项目中，我们使用尽可能少的信息来描述一个音符的运动行为。铺面的基本格式为:

```
轨道号 命中时间戳
...
-1
```

-1 表示铺面数据结束。每行对应一个音符。程序在读取数据的时候，通过不断读入行内容解析轨道号和命时间戳，在事件队列 `note_rail_info`和`time_info`中（在介绍`music_thread`中有提及），不断检测播放时间是否到达生成时间窗口，到达则生成。

综上，铺面的制作本身并不具有很高的操作难度，但是如果只能对照 txt 文本文件写铺而没有其他的辅助制铺工具，这个过程是非常困难的。基于此，我们也专门制作了一个制铺的模块 $QWidget$。这一部分的功能主要在于能够在制作铺面的过程中帮助用户准确定位关键时间戳。我们首先给出这部分的头文件以及 ui 文件。

```
//file: edit.h
class edit : public QWidget {
    Q_OBJECT

public:
    explicit edit(QWidget *parent = nullptr);
    QMediaPlayer *player;
    QAudioOutput out;
```

```
    QTimer *Timer;
    QFile *file;
    QString current_song;
    bool en_slider=false;
    void set_song(QString);
    ~edit() override;

private:
    Ui::edit *ui;
    void keyPressEvent(QKeyEvent* event) override;
signals:
    void slider_update(int value);
    void rail(int);
public slots:
    void spin_update();
    void write_txt();
    void write_rail(int);
        };
```

在edit.h中，我们定义了一个音乐播放器以及一些信号和槽函数。而更多的控件在 *ui* 文件中。

```
//file: ui_edit.h
class Ui_edit
{
public:
    QSlider *horizontalSlider;
    QPushButton *play;
    QPushButton *pause;
    QPushButton *stop;
    QLCDNumber *lcdNumber;
    QLabel *railindi;
    QPushButton *rail1;
    QPushButton *rail2;
    QPushButton *rail3;
    QPushButton *rail4;
    QSpinBox *spinBox;
    QLabel *timeindi;
    QPushButton *jump;
    QPushButton *write;
    QTextEdit *textEdit;
```

```
20
21      void setupUi(QWidget *edit){...} // setupUi
22
23      void retranslateUi(QWidget *edit){...};
24      }
25  namespace Ui {
26          class edit: public Ui_edit {};
27      } // namespace Ui
```

为了方便介绍，直接给出显示的效果图

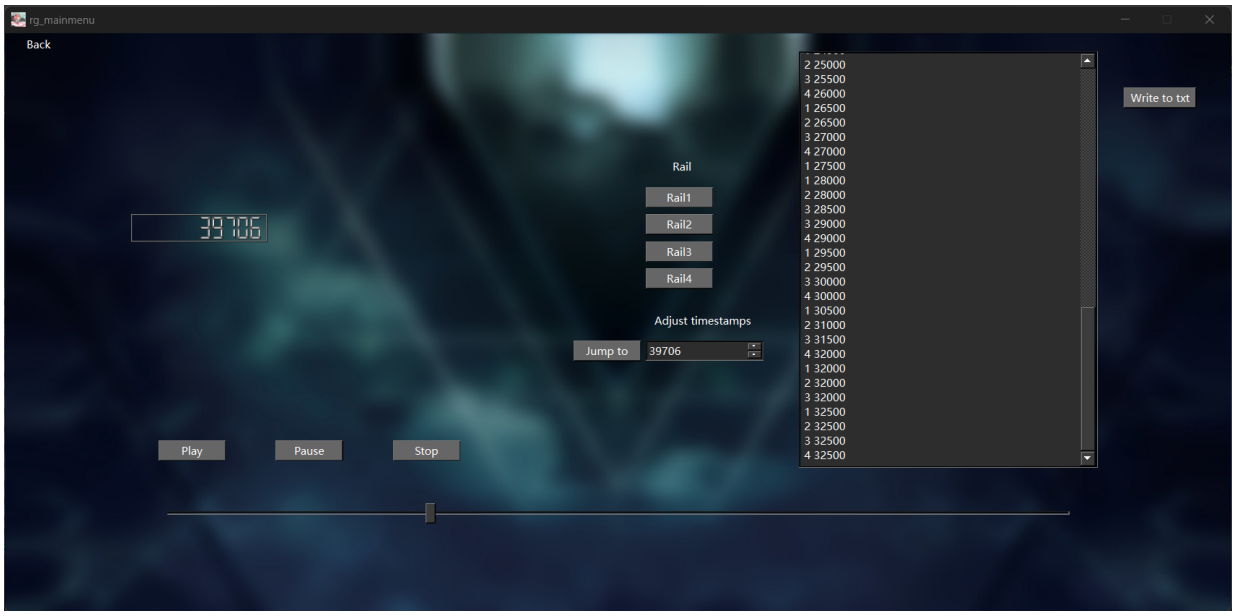

图 11: 编辑时的效果

从图片中，我们可以看到许多不同的显示控件，其中左边的 LCD 数字显示屏的数值为当前播放位置的时间戳，用户可以通过观察当前时间戳位置，快速找到自己想要添加音符的关键时间戳。下半部分是一个进度条以及三个音乐播放控制按键，用户可以通过控制音乐的暂停与播放实现一边听，一边写的制铺过程，下方的进度条会随着播放进度的改变不断移动，也可以手动拖拽进度条。关于拖拽进度条的实现涉及到一些信号与状态变量的关联，这里我们简单展开一下给出对应的实现代码：

进度条的本质是一个 $QSlider$。

```
1  class Ui_edit
2  {
3  public:
4      QSlider *horizontalSlider;
5      ...
```

滑条的变化长度被设置为了和音乐的长度一致，滑块在最左边，音乐播放的位置就是 $0ms$，滑块在最右边，音乐播放的位置就是最后一毫秒。要想实现滑块随音乐播放移动，可以设置一个定时器，定时发射信号触发更新函数移动滑块

```
//file: edit.cpp
connect(Timer, &QTimer::timeout, [=]() {
↪  ui->horizontalSlider->setValue(player->position());
↪  ui->lcdNumber->display(int(player->position()));});
connect(ui->jump,&QPushButton::clicked,this,&edit::spin_update);
Timer->start(1);
```

但是滑块的移动如果仅由音乐播放控制，那么我们就没法通过拖拽实现调整播放进度。为了实现双向的控制，即滑块随音乐移动，又可以通过拖拽控制音乐播放进度，我们可以通过检测滑块是否被按下来决定这时候的控制权归谁。

```
//file: edit.cpp
ui->horizontalSlider->setRange(0,player->duration());
connect(ui->horizontalSlider,&QSlider::sliderPressed,[=](){Timer->stop();});
connect(ui->horizontalSlider,&QSlider::sliderReleased,[=](){emit
↪  slider_update(ui->horizontalSlider->value());Timer->start();});
connect(this,&edit::slider_update,[=](int value){player->setPosition(value);});
```

这一段函数实现的功能是当滑块被按下时，设置位置更新计时器停止计时，这时滑块因为不被更新故不会随音乐移动，用户就可以任意拖动；当我们松开滑块时，发射一个滑块更新信号，这个信号会导致音乐播放位置更新。完成信号发射更新信号让音乐更新之后，再重新启动计时器，实时检测音乐播放的位置，不断发射信号来更新滑块的位置。

回到控件的介绍中，界面的右边是一个快捷 txt 编辑器。文本框中的内容是直接读取的对应铺面 txt 文件内的数据到内存后显示的内容，实时的内容改变并不会引起源文件的改变（因为源文件只是文件流将内容读进程序之后就关闭了）。文本框中的内容可以直接增删，也可以通过文本框左边的四个按键来快速输入。这四个输入按键的功能就是将一个特定轨道的数据快捷输入到文件中，时间戳是当前 LCD 显示屏的时间戳。比如，现在的时间戳是 39706，那么点击Rail1，文本框的末尾会被快捷添加一个新的行

```
...(previous content)
1 39706
```

当然，一般来说我们制作铺面需要合拍，这需要通过计算当前曲目的 bpm 来得到精确的时间戳位置。为了实现准确定位时间戳，我们还添加了一个跳转入口，也就是四个输入按键的下方。跳转入口本质是一个 $QSpinBox$，可以手动输入一个特定的值实现精确跳转，然后通过快速输入按键向文本框中添加新的行。

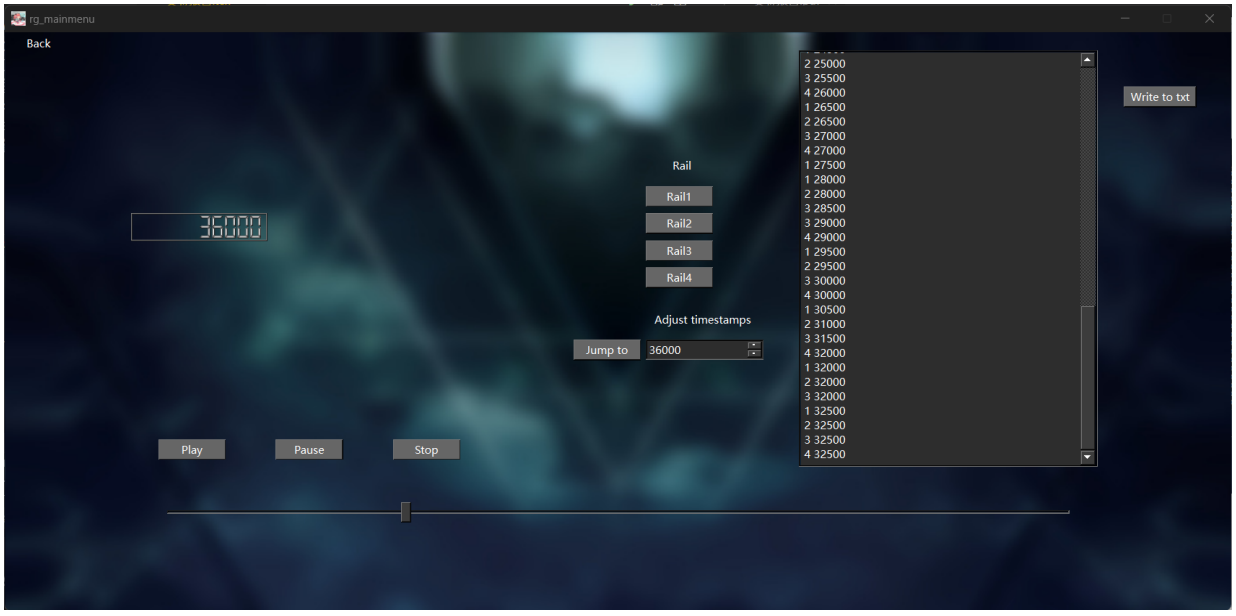

图 12: 通过跳转入口准确跳转到 36000 的位置

当用户完成所有编辑之后，点击`Write to txt`，程序会重新打开对应的铺面文本文件将文件流送入。这时如果我们返回到主菜单选择游玩就可以玩到更新后的铺面了。

5 项目总结

在本次项目的开发中，我们成功使用 Qt 6.5 开发出了一款完整的音乐游戏。这个游戏具有游玩，调整参数以及铺面编辑的功能。其中，在开发核心玩法 $QWidget$ 的过程中，我们使用了多线程技术和对象池技术。

对象池技术是一种管理高重复性实例的方法，它的基本使用要点是：当对象使用结束后并不立即销毁而是重置回收；每当需要实例化的时候优先从对象池取出回收的对象；如果对象池为空，则调用构造函数实例化一个新的对象。这种技术在音乐游戏（音符实例化），射击游戏（弹药实例化）等开发中有非常多的应用。

多线程是一种使用非常广泛的程序开发技术。它的支撑基础是操作系统的多线程管理功能，通过为不同的线程分配以计算机时钟周期为基本单位的时间片，实现程序的并发运行。在本项目中，音符的生成和音符的消去就是一对异步的过程，它们分别位于两个不同的线程中，以保证程序对这两个任务都能有及时的相应。在使用多线程开发的时候必须注意共享资源的操作。如果位于不同线程的不同函数需要调用同一个共享资源，从内存安全的角度考虑必须要使用互斥锁，即一个线程访问资源的过程中，另一个线程如果同时要访问则必须阻塞等待，也就是同一个资源在同一个时间只能被同一个线程访问。基于这种考虑，本项目中的音符对象池由于同时是消去线程和生成线程的共享资源，故而被加上互斥锁以保证访问互斥，尽管阻塞访问会导致一定延迟，但仍然是必要的。

Qt 框架中最具特色的机制就是信号与槽函数的机制，通过将不同位置的信号与槽函数连接，可以实

现程序通信的效果。本项目中几乎所有控件都有使用信号与槽函数通信的机制。值得注意的是，由于连接函数`connect()`可以在任何位置，如果随意书写可能会导致后续开发中难以寻找相匹配的信号/槽函数对，应该尽可能将连接函数写在固定的位置如写在信号定义所在类或者槽函数所在类中。除了一般的槽函数，`connect()`函数也接收匿名函数（*lambda* 函数）作为槽函数，这样写可以简化一部分代码。

完整的项目文件参见 *Github* 仓库链接：https://github.com/zheguabaoshuma/qtrg