目录

hso!mania 下落式音乐游戏设计

一、题目简介

电脑游戏种类十分繁杂，音乐游戏（音游）就是其中一类，目前在电脑平台的音乐游戏大致可以分为几种类型，常见的为键盘操作配合音乐节奏来接住下落的音符，本次课程设计所选题为自选题（小游戏），游戏类型为经典的 4Key 下落式音乐游戏。

hso!mania 是我给这款游戏起的名字，作为一款游戏，基本需求是能够流畅运行游玩，其他基本需求大致如下：

a) 基本 UI 完善，操作简洁明了符合人类习惯

b) 能够从文件读取游戏所需要的数据（谱面和歌曲）

c) 能够计算并评价游玩结果

d) 尽量符合良好的审美习惯

要实现一个典型下落式音乐游戏，其基本游戏系统如下：

游戏整体均与音乐相关，以音乐为主题，因此各个界面均配有背景音乐；游戏游玩时的核心是根据音乐节奏来敲打指定的按键，4Key 下落式音乐游戏需要操作的键位一共有 4 个，分别对应四条不同的轨道，在游玩中音符会随着音乐节奏从轨道上落下，当音符落到判定线时只需要按下对应轨道的按键就可以成功演奏音符并获得加分。每一次按下对应按键的时机不同都会带来不同的判定，常见的音乐游戏一般都设置了 4 种以上的判定，分别用来对应按键时机与准确时机的相差情况，相差的时间越短，判定越高，得分以及总体达成率就越高；另外，在某个判定以上的判定会增加演奏连击数，连击数也是评判游玩结果的一个重要指标之一。

二、系统概述

hso!mania 音乐游戏（下简称“本游戏”）的游戏系统和主要功能大致如下：

1. 直接对每帧的图形进行绘制

在保证帧速率的情况下，每帧调用绘图函数对屏幕图像进行绘制，可以在大多数情况下对静态图形显示、动画显示得到极佳的控制，同时为了解除对底层绘图程序的耦合，在本系统中引入了一个 CanvasHelper 来封装基本绘图操作，后续进行底层绘图驱动更换时可以保证对外接口不变从而不必对整个程序进行修改；此外编写了一次的 CanvasHelper 也可以直接复制源代码或者导出为动态链

接库给其他需要绘图的程序使用。

2. 对绘图区域进行封装以实现不同窗口大小下绘图的一致性

在封装好的 CanvasHelper 中，我设计了一个基于比例的绘图坐标系统，在初始化时设置绘图坐标总大小，就可以在后续的所有绘图流程中使用转换后的坐标系统而不是基于像素值的坐标系统，由此可以保证所绘制图形的比例不变，从而保证在不同窗口大小情况下都能正常工作。

3. 使用有限状态机来控制游戏流程

对于相对较为复杂的且存在一些较为明显的阶段（状态）区分的系统设计，使用有限状态机（FSM）可以大大降低设计的复杂度并提高可维护性。 本游戏在不同页面下的显示内容、交互逻辑都有明显的不同，因此在游戏总架构上维护一个状态机，每个状态由不同的状态类响应用户的交互并负责绘图。

4. 游戏谱面格式按照目前玩家数众多的一款音游谱面设计

谱面，也就是描述下落的音符和音乐节奏如何对应的文件；为了保证游戏性（玩家游戏体验）以及节省程序测试时间，分清课程设计重点工作，决定直接使用成熟音游（osu!）的谱面格式，不另外设计谱面；由于 osu!本身有几个不同的游戏模式且支持 4~7 Key 下落，因此本游戏扫描谱面文件时，会首先判断谱面是否是 4Key 下落式的谱面格式。

使用兼容现有音游的谱面格式这样做的好处是，可以直接使用现有的数据不用在自行设计谱面上花费诸多时间；另外由于市面上的音游谱面都是专业作谱师花费心血对着歌曲节拍并且参考诸多音游机制设计的，因此谱面质量（音符排列的合理性）也能够得到保证。

5. 对歌曲资源和谱面解析均进行封装

由于需要游玩不同的歌曲，所以对歌曲资源以及其相关的谱面序列进行封装可以实现更为复杂的逻辑，而对谱面序列进行封装后，还能够更为方便的设置下落速度效果、动画效果、判定难易度等等。

三、系统设计

本小节对整个系统的设计由上而下地进行分析，从程序执行流程为切入点对游戏进行大体介绍，较为详细的技术细节将在“关键技术”中进行介绍。

1. 游戏系统中主要的类与对象

整个游戏系统中主要的类与对象的设计如下图所示（大部分类的成员已隐藏）：

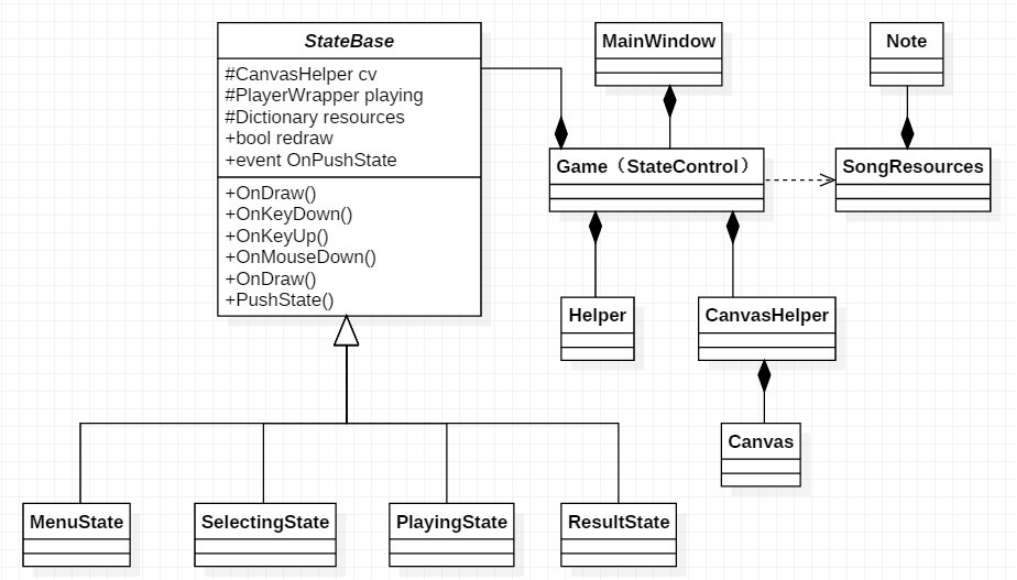

图 3.1 系统中主要的类与对象

其中 Game 类为游戏主要流程与控制逻辑实现的部分，也是有限状态机实现的位置，本系统中状态机的部分定义了一个抽象类 StateBase 作为基类，然后派生出四个派生类来具体实现四种子状态；而静态类 Helper 则提供一些常用的静态方法例如加载图片资源、加载歌曲资源、创建一个指定颜色和透明度的画刷等等；CanvasHelper 类通过内部维护其应该绘制的控件 Canvas，封装所有的绘图方法，对外提供友好的、便捷的绘图接口。

此外，为了描述游戏所游玩的对象（歌曲极其对应的谱面），还额外定义了 SongResources 类负责游戏资源的载入和谱面格式的解析，解析完成的谱面将会生成 Note 序列，Note 类中则包含了每一个下落音符相关的信息（包括对应时值以及判定状态等）。

2. 程序启动与主要事件循环

主窗口上只有一个 Canvas 控件负责作为图形载体，在主窗口启动后，首先会创建游戏运行需要的各个基本对象，包括 Game 类对象、用于搭建状态机的字典、各个子状态对象等，在各个对象创建完成后程序将一些必要的事件绑定到实际接受事件的控件上，随后读取指定文件夹下的图形资源文件、歌曲资源文件；在这些初始化工作都完成之后启动绘图循环来启动整个游戏。

上面的流程可以用流程图描述为下图：

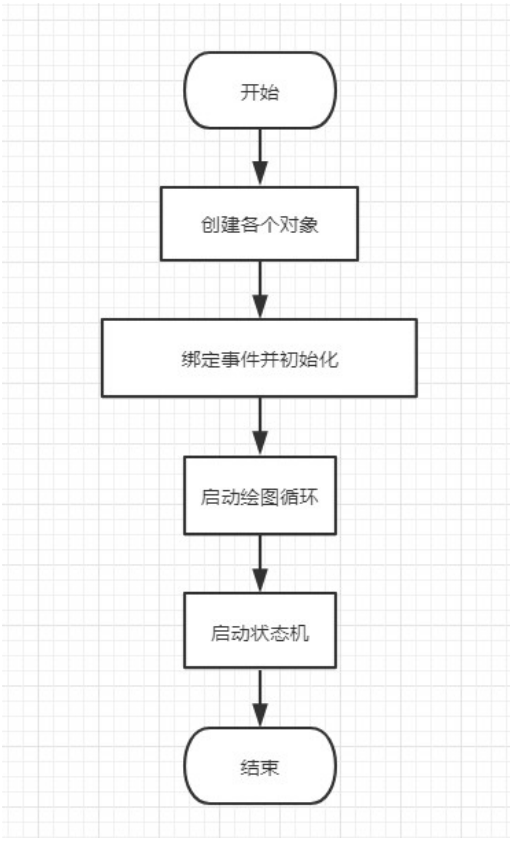

图 3.2 程序启动流程

3. 有限状态机系统

由于在本游戏下状态机所需要的复杂逻辑不多，故状态机管理逻辑直接集成在 Game 类下，不再独立实现单独的完整的状态机管理系统，关于有限状态机的具体实现细节我将会在“关键技术”一节详细介绍。

而如果要实现一个较为通用的完整的有限状态机控制系统，则需要进行进一步的接口设计来保证低耦合，因此为了节约事件以及减少系统的复杂度，在本游戏系统中直接使用字典来管理子状态实例，使用事件来处理状态切换逻辑。

4. 绘图逻辑以及事件响应

在初始化完成后，由主窗口创建一个线程，该线程基于当前时间的毫秒数来保证绘图循环间隔(帧率)稳定，在每次的绘图循环执行时，主窗口通过调用 Game 类预先定义的 OnUpdate 方法通知其进行绘图更新，而 Game 类则通过内部状态机决定将该事件分发给哪一个子状态处理。

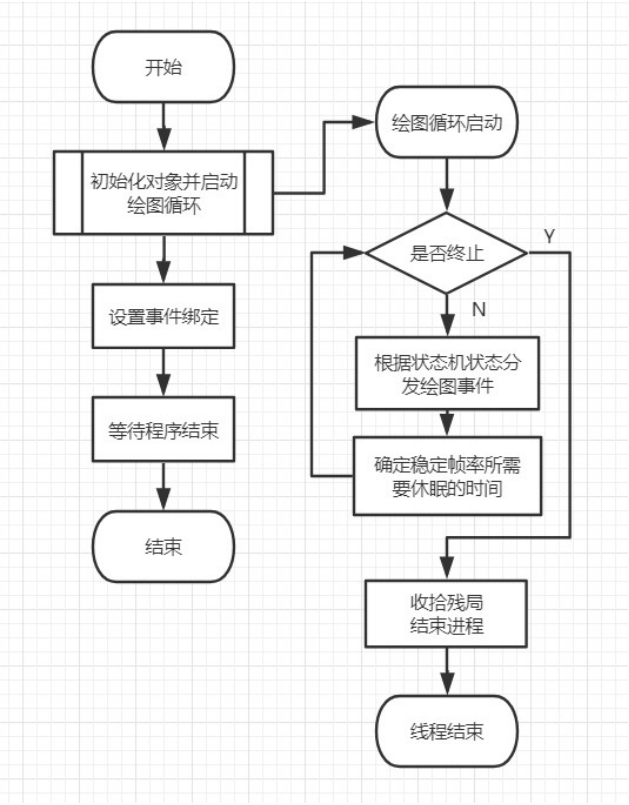

图 3.3 绘图循环处理逻辑

事件响应的逻辑大体与绘图循环类似，在 Game 类创建时需要指定其负责绘图的控件，并绑定该控件的基本鼠标事件、键盘事件到事件处理函数上，而在事件处理函数被触发时，Game 类会通过内部维护的状态机来决定将事件如何分发给子状态处理。

5. 资源字典的维护

在 Game 类维护一个资源字典以获取各类所需要的资源，创建 Game 类的实例后，执行初始化操作时将所需的图形资源、歌曲资源读入到一个 Dictionary<string, object>字典中，在创建状态机的各个子状态时传递这个资源字典，于是在各个子状态内访问资源时可以通过资源的键来取得相应的数据。

6. 游戏判定及分数计算

由于各类音乐游戏的分数、达成率计算方法以及判定准确度要求都有所不同，因此我自己根据多年玩音乐游戏的经验，通过数学方法计算出了一套分数评价机制，在试玩后感觉与现有音游相差不大。

本游戏内音符判定准度在-200ms 以上不做处理，在音符达到+200ms 以上时无论用户是否按下对应按键都自动做 miss 处理，在 200ms 的区间内，分别设 20ms,50ms,100ms 为关键点，将区间划分为四个区域对应四种判定精准度；而在计算分数和达成率时按以下公式计算：

$$\begin{cases} N = N_t + 2N_h \\ S_g = \dfrac{1000000}{N}, A_{pg} = \dfrac{100}{N} \\ S = \left(1.1N_{pg} + N_g + 0.85N_{gd} + 0.7N_b\right) \times S_g \\ A = \left(N_{pg} + 0.97N_g + 0.8N_{gd} + 0.7N_b\right) \times A_{pg} \end{cases}$$

其中 N 为总音符数，$\mathrm{N_t}$ 为单点音符数，$\mathrm{N_h}$ 为长押音符数，pg，g，gd，b 下标分别为四种判定的音符数量。

四、关键技术

1. 有限状态机（FSM）

游戏系统核心部件为有限状态机的简单实现，最初实现时并没有将各个子状态抽象为独立的类，而是在 Game 类中对 CurrentState 进行判断，使用 switch-case 来决定调用的函数，使用这样的做法起初结构还算比较清晰，随着代码量增加，函数数量显著增多，在文件中定位指定的函数显得十分麻烦，于是后来我对整个状态机系统的实现进行了重构，改为了目前的这种设计。

首先通过定义抽象基类 StateBase 来描述状态所具有的共同特性，不使用接口的原因是大部分函数我都准备提供默认实现，这样在定义子状态时就不用写太多重复代码。

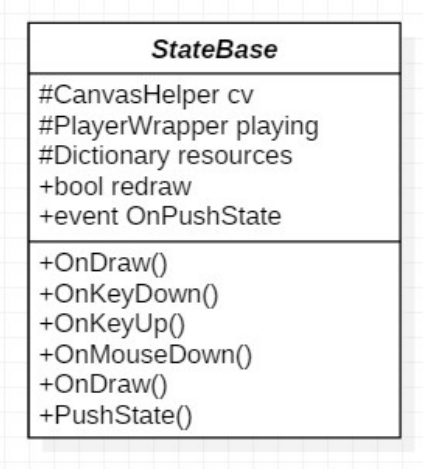

图 4.1 状态基类的主要成员

抽象状态基类中定义了状态绘图所应该操作的 CanvasHelper、当前正在播放的背景音乐包装器 PlayerWrapper、加载完成的资源字典以及重新绘制的 flag，此外定义公开的事件处理函数以供调度器调用。

另外，由于有从状态内部提出变更状态请求的需求，而状态本身不应该拥有调度状态的权限以及接口，因此通过在基类上定义事件，在子类调用 PushState 方法来变更状态时，触发事件给外部的状态管理器处理。

在各个具体状态类的实现时，如果其需要处理某一类事件，则通过虚函数重

写相应的事件处理函数即可，一些子状态所特有的操作、状态则通过其私有成员来管理。

2. 封装相对坐标系统绘图方法

为了绘图方便，符合直觉，在这个系统中我编写了一个 CanvasHelper 类来封装底层的绘图逻辑。通过在 CanvasHelper 声明的接口上设置绘图区域的坐标范围，之后在其声明一些常用的绘图方法如画线、画矩形等方法上传递坐标时，就可以直接使用坐标来指定位置，这个功能实现的核心就是在内部进行坐标到实际像素位置转化的计算；另外在使用 CanvasHelper 的过程中，有遇到需要改变填充颜色、边缘颜色等需求，于是在后期我对其接口进行了一些改动使得可以从外部传递所使用的 Brush，而不是使用默认值颜色。

CanvasHelper 的内部绘图逻辑是直接使用 Canvas 提供的功能来做的。经过一定的上网检索之后，发现在 wpf 中除了向 Canvas 控件添加 Children 以外似乎没有别的简便的画图方法，虽然查询到了可以通过继承 Canvas 类并重写 OnRender 方法来从底层控制渲染过程，但由于个人感觉 OnRender 提供的接口甚至不如直接使用 GDI/DirectX 绘图方便，而后者也因为暂时不想封装而放弃了这些做法，最终选择直接在每帧需要绘制的时候将 Canvas 类的 Children 全部清除重新绘制。

3. PlayerWrapper 包装器类

游戏几乎全程都有播放背景音乐，在各个状态之间切换时背景音乐一般也需要根据不同需求切换、停止、继续播放等，因此为了方便，我在状态基类中定义一个 PlayerWrapper 包装器类来引用当前正在播放的音乐。

之所以不直接使用 MediaPlayer 是因为使用 MediaPlayer 时，对象本身的地址不能被改变，否则会导致其他地方的引用全部失效（意思是不再引用同一个对象），因此在切换音乐时必须通过 Open 方法来重新读取歌曲数据，这样显然比较麻烦而且较为重复，因此为了使所有的状态中引用的 MediaPlayer 都具有相同的值（哪怕时在其被置为 null 或者重新赋值后），引入 PlayerWrapper 包装器将其包装起来，这样只需要使所有的状态所引用的包装器是同一个即可。

4. 动画实现方法

使用上述提到的 CanvasHelper 类解决了绘图坐标的问题后，决定采用最常用的游戏动画实现方案，即每帧重新绘制图形，不过由于 Canvas 控件内部有自行维护状态，故在 wpf 下使用 Canvas 来每帧清空 Children 并重新绘制执行效率会相对偏低，但实际测试和编写后发现并没有明显的卡顿和掉帧现象，所以底层的绘图驱动也就暂时不做修改了。

一般较为简单的动画在其相应的状态类中定义动画阶段变量，通过读取动画阶段变量就可以知道每一帧该如何绘制并更新阶段变量，这样每帧的画面连起来就成为了动画。

5. 谱面解析过程

在之前的章节中提到，本游戏的谱面设计完全采用于 osu!mania 相同的格式，幸运的是 osu!提供了维基[1]并详细描述了谱面的构成格式，但我不准备完全实现其谱面描述的所有内容（因为过于复杂和庞大），简化后的谱面格式大致可以描述如下：

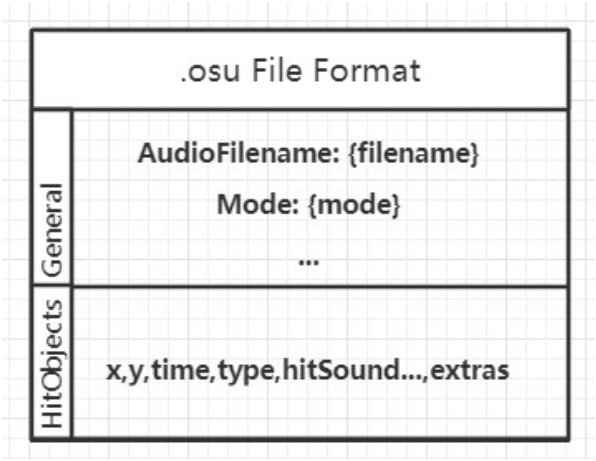

图 4.2 谱面格式（简化版本）

其文件格式分节方式类似 ini，在 General 节中描述了歌曲相关的一些元信息，而 HitObjects 节中则包含了所有音符的相关信息，由于其格式较为简单，解析谱面由 SongResources 类在打开文件夹扫描有效文件后一并进行；而由于只有在游玩时具体的音符数据才是必要的，因此该部分的读取和解析延后到游玩前处理读入处理。

6. 游玩判定

在看到 osu!的谱面设计之后不难发现其使用的时值均是绝对值时间，因此在本游戏的游玩判定时也使用绝对值时间的方法来判定。为了减小按键到判定完成的延迟，我在按键事件被复杂的逻辑处理分析前就将按键事件触发的时间点记录下来，之后再进行进一步的判断和检查判定，相信使用这样的做法可以减少数毫秒的延迟。

为了简化判定逻辑以及修改方便，判定相关的方法均封装在 Note 类中实现，判定的结果也可以直接修改对应 Note 的相关变量以供游玩后的统计。

7. 不同线程的异常捕捉

在游戏执行时，由业务代码产生的就有至少 2 个线程，而为了设计方便，我所设计的类、Helper 函数等也会在一定情况下抛出异常以供统一处理，而在不同的线程下执行的代码抛出异常时只能由其对应的线程所捕捉；此外，部分 C#

内建类型如 MediaPlayer、ImageBitmap 类等在读取文件时均是异步读取，在其发生失败时将不能通过业务代码中的 try-catch 块捕捉。

于是部分异步产生的异常我参考了 MediaPlayer 的实现方式，通过声明一个事件来处理抛出的异常，以此来将异常通过事件传递出来并委托给相应的异常处理程序。

五、效果及存在问题

1. 游戏执行效果

游戏按流程执行大致效果如下：

执行游戏后展示欢迎界面，显示游戏 LOGO 以及提示文字，其中提示开始的文字将不断闪烁。

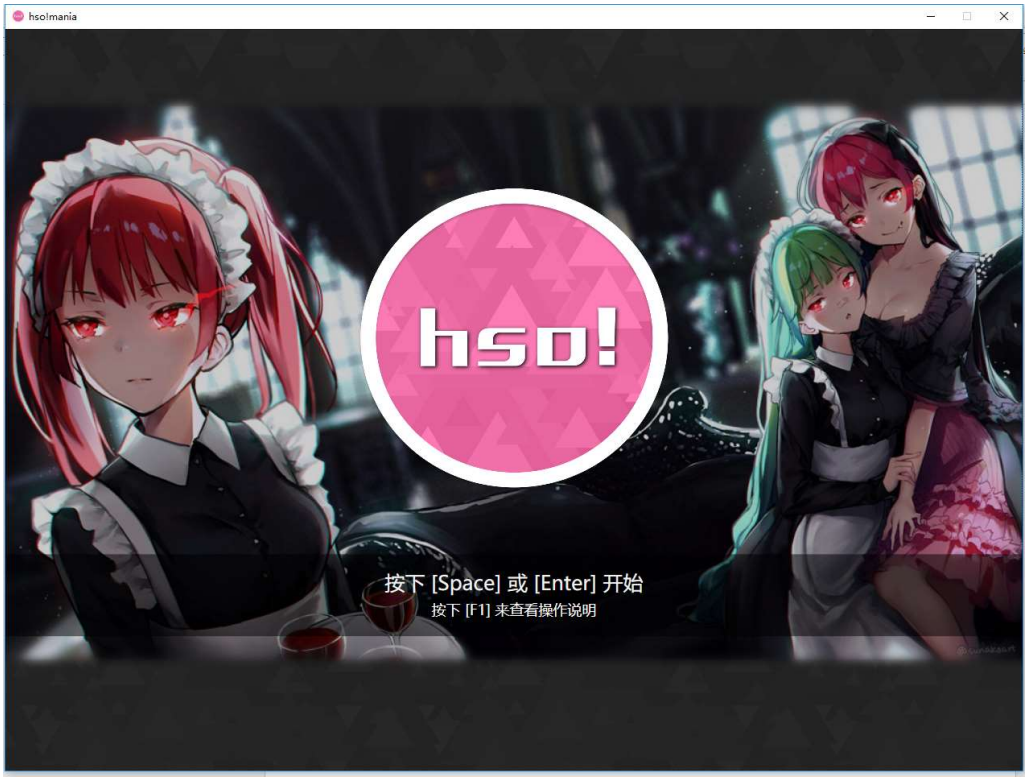

图 5.1 欢迎界面

按下 F1 会展示游戏操作说明，按下空格或回车后进入“选择歌曲”页面，在此页面会展示所有保存在 Songs 文件夹下所被支持的歌曲文件以及其对应的谱面列表，在这个页面可以通过键盘左右切换歌曲，上下切换难度。

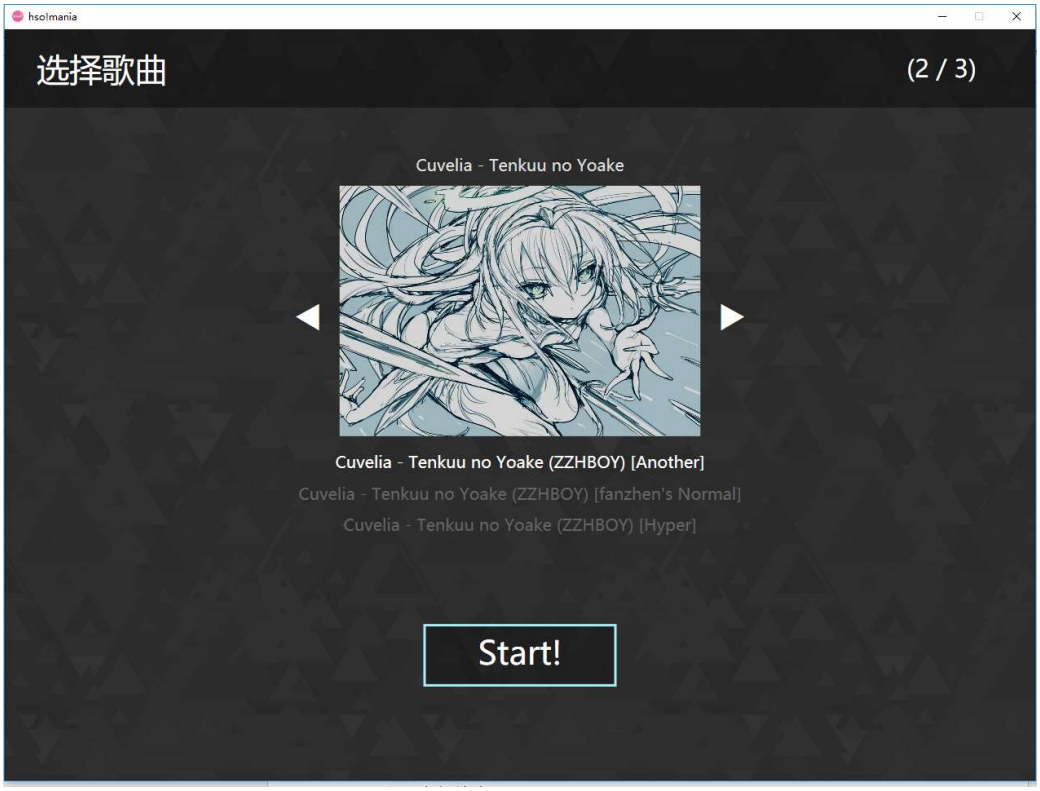

图 5.2 选择歌曲界面

按下开始后在游玩中的效果如下图所示，轨道会在按下相应按键时有光效反馈，另外顶部显示进度条以及当前连击数，在画面的右上角有分数以及达成率显示。

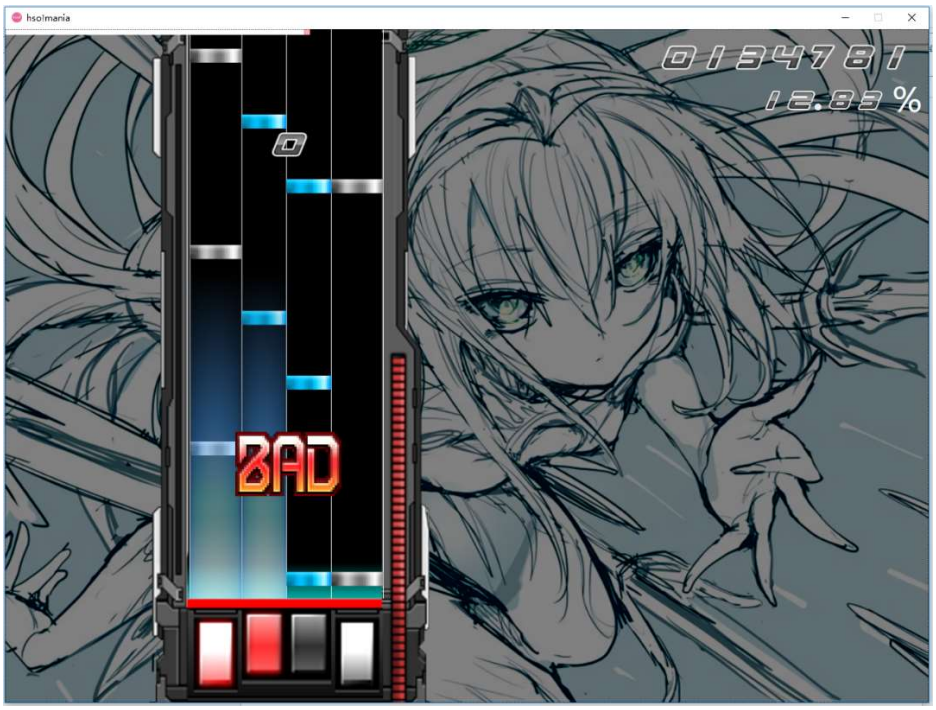

图 5.3 游玩界面

游玩结束后进入评分界面，在评分界面显示用户各个判定的音符数量以及分

数、达成率，并在画面右侧有相应的等级评价。

图 5.4 结果判定界面

2. 存在问题和解决方法

a) 底层使用的绘图驱动为 wpf 的 Canvas，在一定程度上影响程序绘图效率；更好的做法是直接使用 DirectX 等其他更为底层接口绘图，绕过 Canvas 控件提供的实时更新，以此减少计算量。

b) 加载文件时为同步加载，在歌曲数量多时耗时会明显增加，改为异步加载会更合适但复杂度也会相应提升。

c) 在缺少资源文件时仅仅捕捉异常并退出程序，程序内部有许多素材是非必需就能运行的，因此对于一些非必需的资源文件如果允许其丢失的话鲁棒性会更强。

d) 没有实现谱面变速逻辑、歌曲前导缓冲逻辑，在 osu! 原游戏以及其谱面信息中还存有许多数据，有影响游戏性的数据也有更详细的描述信息，而本游戏在搭建过程中暂时忽略了那部分数据。进一步丰富游戏功能时可以考虑将其描述的一些功能加入。

e) 虽然实现了部分鼠标点击交互逻辑，但没有实现更精细的鼠标交互和其他动画。

f) 部分地方的面向对象设计不符合最佳实践。为了编写方便，有部分类之间产生了耦合，理应可以进一步考虑解耦和封装。

六、心得体会

通过这次的课程设计，我对 C# 中的多线程、事件机制有了更深入的了解，同时对 C# 中的图形驱动引擎也有了粗略的印象，虽然其提供的一些图形类、动画类可以在窗体应用中提供良好的动画支持；对 C#中的 Brush 机制和多媒体相关知识也有了进一步的理解。

在编写游戏的过程中，我也通过谱面设计对现有的一些音乐游戏的设计有了一个大概的印象，也感受到了从零开始实现绘图、动画、交互的不易。

此外，在这次游戏的编写中，一些图片资源和音乐资源我也有使用 Photoshop 和 Adobe Audition 来编辑，游戏的一些细节方面我也有从做一个产品的角度来思考该如何交互、如何表现更加合适。

总之此次课程设计可以作为一个比较完整的项目经验给我带来不少提升。

七、附录

1. 主窗口管理绘图线程的逻辑

```
private void Canvas_Loaded(object sender, EventArgs e) {
    canvas.Focus();
    cv = new CanvasHelper(canvas);
    game = new Game(cv);

    try {
        game.Initialize();
    } catch (Exception ex) {
        MessageBox.Show("在初始化时发生错误！\n\n详细信息：\n" + ex.Message, "错误",
MessageBoxButton.OK, MessageBoxImage.Error);
        Close();
        return;
    }

    loop = new Thread(new ThreadStart(Loop));
    looping = true;
    loop.Start();
}

bool looping = false;
private void Loop() {
    try {
        while (looping) {
            // 同步调用
            Dispatcher.Invoke(new UpdateDelegate(game.OnUpdate));
            // 用这种方式来保持帧率稳定
            Thread.Sleep(mpf - DateTime.Now.Millisecond % mpf);
        }
    } catch (Exception e) {
        MessageBox.Show("在执行过程中发生错误！\n\n详细信息：\n" + e.Message, "错误",
MessageBoxButton.OK, MessageBoxImage.Error);
        Close();
    }
}
private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e) {
    // 如果使用 Abort 来终止线程的话，还会抛出一个 ThreadAbortException 并且 Close 操作
将会无法执行
    // 所以干脆用一个变量控制让线程主动退出
    looping = false;

}
```

2. Game 类实例的构造及状态机管理的主要逻辑

```
private CanvasHelper cv;
PlayerWraper player = new PlayerWraper();
// 重构状态机
Dictionary<State, States.StateBase> GameStates;
private State CurrentState;
public Game(CanvasHelper _cv) {
    cv = _cv;
    cv.MouseButtonEvent += Canvas_MouseLeftButtonUp;
    cv.KeyDown += CanvasKeyDown;
    cv.KeyUp += CanvasKeyUp;

    GameStates = new Dictionary<State, States.StateBase>(4);
    GameStates[State.Menu] = new States.MenuState(cv, resources, player);
    GameStates[State.Selecting] = new States.SelectingState(cv, resources, player);
    GameStates[State.Playing] = new States.PlayingState(cv, resources, player);
    GameStates[State.Result] = new States.ResultState(cv, resources, player);
    foreach (var s in GameStates) {
        s.Value.OnPushState += OnPushState;
    }
}

void OnPushState(State target, object args = null, bool stopMusic = true) {
    if (stopMusic && player.player != null) {
        player.player.Stop();
        player.player = null;
    }
    CurrentState = target;
    GameStates[CurrentState].OnStateEnter(args);

}
```

3. CanvasHelper 的坐标转换逻辑

```
public CanvasHelper(Canvas _cv) {
    cv = _cv;
    if (double.IsNaN(cv.Height) && cv.ActualHeight == 0) {
        throw new Exception("Error spawning CanvasHelper, Canvas should be rendered
first!");
    }
    Height = cv.ActualHeight;
    Width = cv.ActualWidth;
    cv.PreviewMouseLeftButtonDown += MouseEvent;
    cv.PreviewKeyDown += Canvas_KeyDown;
```

```
    cv.PreviewKeyUp += Canvas_KeyUp; ;
}

private void Canvas_KeyUp(object sender, KeyEventArgs e) {
    KeyUp(sender, e);
}

private void Canvas_KeyDown(object sender, KeyEventArgs e) {
    KeyDown(sender, e);
}

private void MouseEvent(object sender, MouseButtonEventArgs e) {
    Point p = e.GetPosition(cv);
    MouseButtonEvent(sender, new PointEventArg() {
        point = new Point(w(p.X, true), h(p.Y, true))
    });
}

public CanvasHelper SetRange(double width, double height) {
    Width = width;
    Height = height;
    return this;
}
protected double w(double x, bool reverse = false) {
    if (reverse)
        return x / cv.ActualWidth * Width;
    return x / Width * cv.ActualWidth;
}

protected double h(double y, bool reverse = false) {
    if (reverse)
        return y / cv.ActualHeight * Height;
    return y / Height * cv.ActualHeight;

}
```

4. 其中一个子状态的部分代码

```
class MenuState :  StateBase
{
    public MenuState(CanvasHelper _cv, Dictionary<string, object> res,
Game.PlayerWraper playing) : base(_cv, res, playing) {
    }

    // 第一个 State 在Enter的时候
```

```
    public override void OnStateEnter(object args) {
        base.OnStateEnter(args);
        playing.player = (MediaPlayer)resources["wav.startup"];
        playing.player.Play();
    }

    double op = 1;
    double dop = -0.04;
    bool showInstruction = false;
    public override void OnDraw() {

        cv.Clear();
        if (redraw) {
            // 绘制菜单，redraw 的时候处理淡入属性
            cv.cv.Opacity = 0;
            redraw = false;
        }
        // 处理淡入
        if (cv.cv.Opacity < 1) {
            cv.cv.Opacity += 0.1;
        }

        if (showInstruction) {
            cv.Image(0, 0, 640, 480, (Brush)resources["img.instruction"]);
        } else {
            ((Brush)resources["img.bg"]).Opacity = 0.8;
            cv.Image(0, 0, 640, 480, (Brush)resources["img.bg"]);
            cv.Image(220, 100, 200, 200, (Brush)resources["img.start"]);
        }

        // 闪烁文字
        if (!showInstruction) {
            cv.Rectangle(0, 340, 640, 53, 0, null, Helper.ColorBrush("#000", 0.5));
            cv.Text(200, 350, 25, "按下 [Space] 或 [Enter] 开始",
Helper.ColorBrush("#fff", op), 240);
            cv.Text(200, 370, 18, "按下 [F1] 来查看操作说明", null, 240);
            if (op < 0 || op > 1) {
                dop *= -1;
            }
            op = op + dop;
        }

        // 循环播放
        if (playing.player != null && playing.player.NaturalDuration.HasTimeSpan &&
```

```
playing.player.Position.TotalSeconds >=
playing.player.NaturalDuration.TimeSpan.TotalSeconds - 5) {
            playing.player.Position = TimeSpan.Zero;
        }
    }

    public override void OnMouseDown(object sender, CanvasHelper.PointEventArg e) {
        if (showInstruction) {
            showInstruction = false;
            cv.cv.Opacity = 0.3;
            return;
        }
        PushState(State.Selecting);
    }

    public override void OnKeyDown(object sender, KeyEventArgs e) {
        if (e.Key == Key.Enter || e.Key == Key.Space) {
            if (showInstruction) {
                showInstruction = false;
                cv.cv.Opacity = 0.3;
                return;
            }
            PushState(State.Selecting);
        } else if (e.Key == Key.F1) {
            showInstruction = true;
            cv.cv.Opacity = 0.3;
        } else if (e.Key == Key.Escape && showInstruction) {
            showInstruction = false;
            cv.cv.Opacity = 0.3;
        }
    }

}
```

八、参考文献

[1] osu! File Formats, https://osu.ppy.sh/help/wiki/osu!_File_Formats/Osu_(file_format)#global-structure, 2019/2/20-2019/2/23

[2] .NET API Browser, https://docs.microsoft.com/en-us/dotnet/api/index?view=netframework-4.7.2, 2019/2/20-2019/2/24