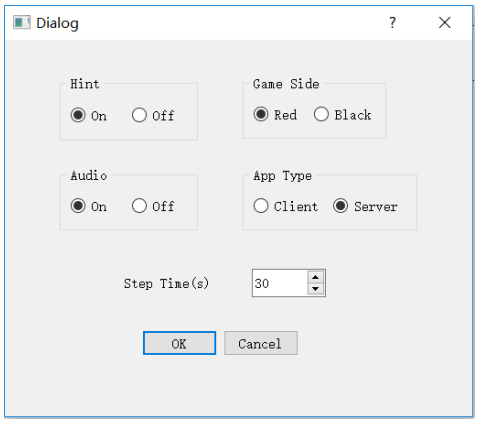
Chinese Chess

Designer: 计 73 李家昊

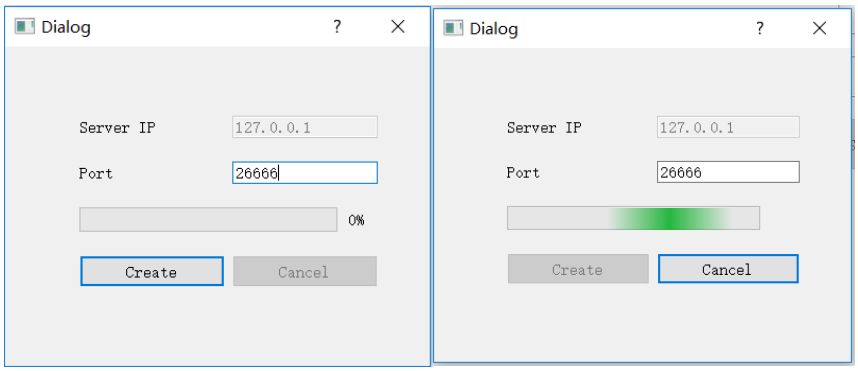
功能实现

网络连接

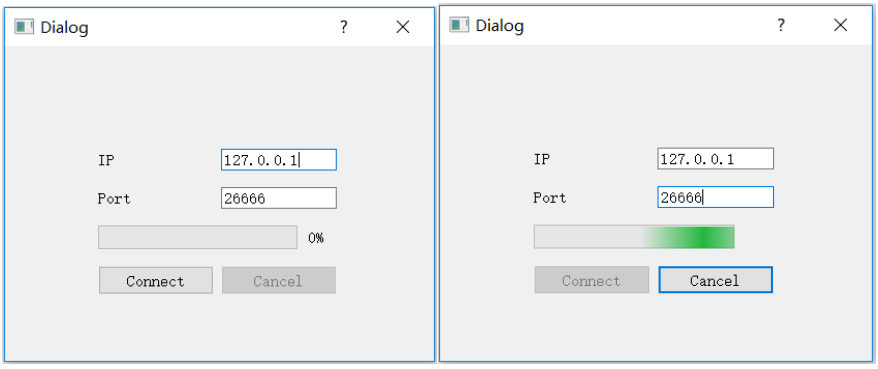
当建立新游戏时，会弹出游戏设置对话框，用户可选择红/黑阵营、主机/客户端，是否显示提示，是否开启声音，并设置步时。



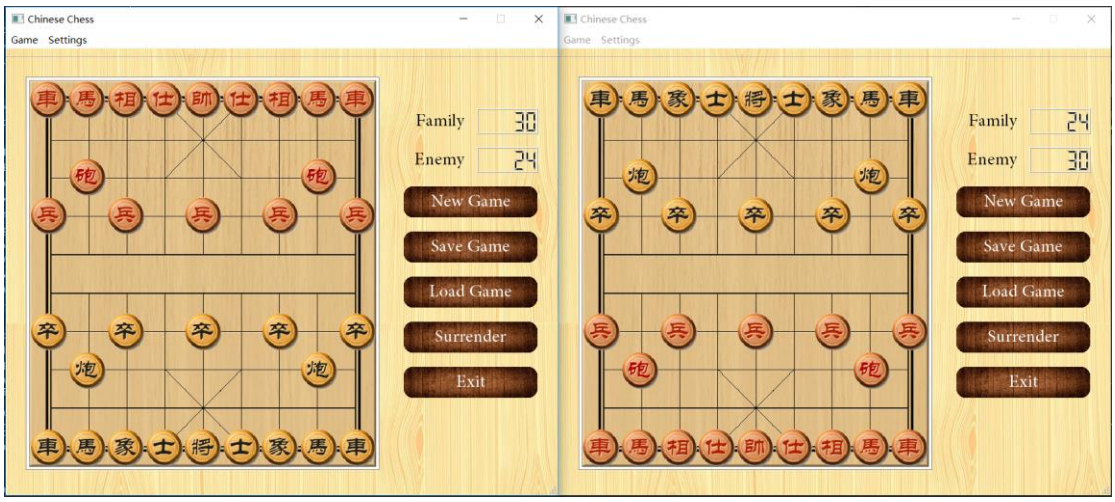
当用户点击 ok 时，服务端会显示开启监听的对话框，当开始监听时，会等待客户端连接，等待过程中，可随时点击 Cancel 按钮取消监听。



客户端则会显示连接对话框，当点击连接时，客户端会不断向服务端发起请求，直到连接成功为止，等待连接过程中，可随时点击 Cancel 按钮取消连接。



当连接成功时，自动跳转到游戏界面，游戏立刻开始。



将军音效

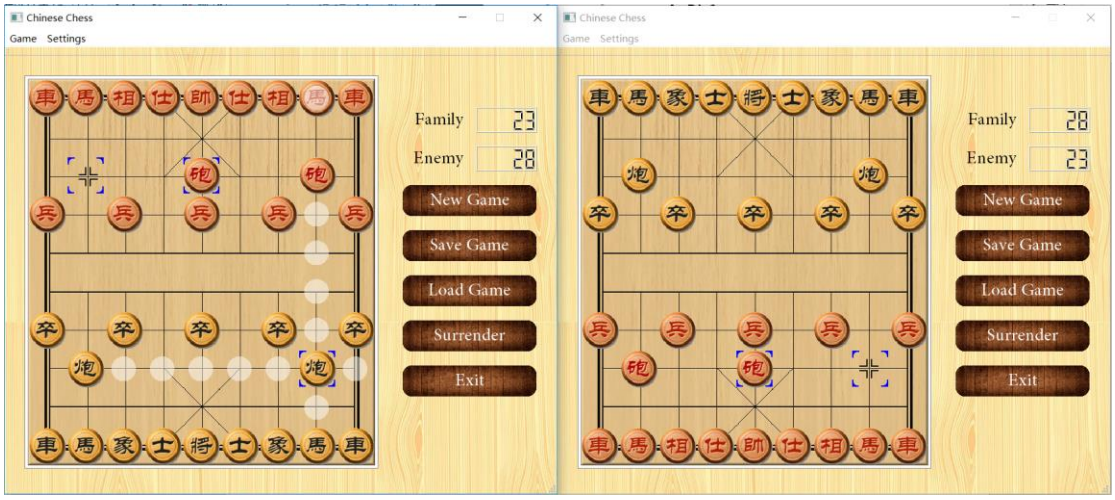
当己方走棋将对方军时，或对方走棋将己方军时，（包括对将的情况），均会发出将军音效

超时判负

单步走子超时一方无条件判负。

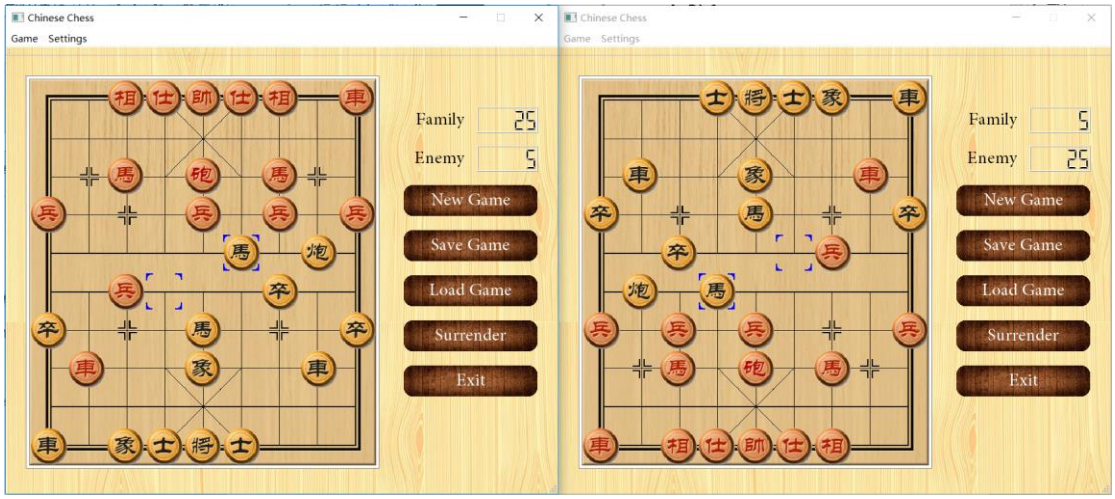
走子提示

当用户选中某只棋子时，棋子开始闪烁并带上边框，所有能够到达的位置在界面中用白色的圆圈标明。当用户落子时，原位置和目标位置都会带上蓝色边框，方便己方和对方查看。



保存和加载残局

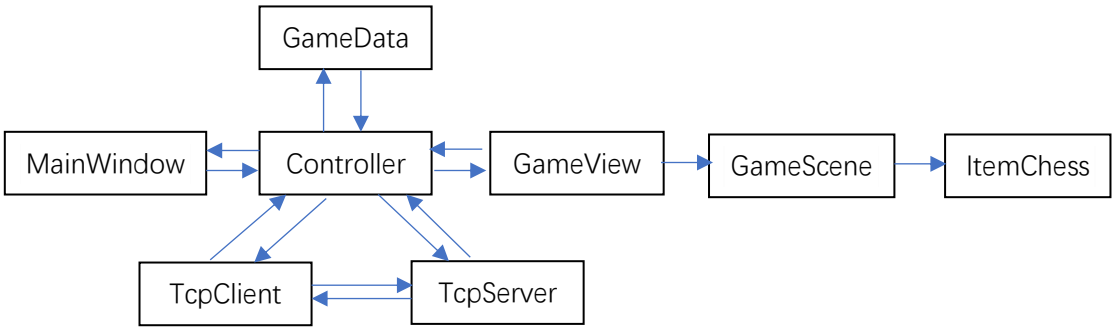
严格按照助教给定的格式保存和加载残局，如下图的残局保存为下面的.txt 文件，再次导入时，则从保存的位置继续游戏。




```
red
1 <4,0>
2 <3,0> <5,0>
2 <0,2> <8,2>
2 <1,0> <7,0>
2 <0,0> <8,0>
2 <4,6> <7,2>
4 <6,5> <2,4> <0,3> <8,3>
black
1 <4,8>
2 <3,9> <5,9>
2 <2,9> <8,7>
2 <7,9> <0,7>
2 <0,9> <8,9>
2 <2,7> <4,3>
3 <0,6> <6,6> <8,6>
```

程序逻辑

整个程序分为显示模块、控制算法模块和网络模块，以 MainWindow 为入口，以 MainWindow 和 GameView 为显示部分，用以显示用户界面，以 Controller 为中心控制，完成核心的运算和判断。以 TcpServer 和 TcpClient 为网络模块，负责网络连接和收发数据。
游戏数据的定义为 GameData 类，其中包含了地图信息、先手阵营、上一步棋的原位置和目标位置等。在 Controller 中定义了一个 GameData 对象，存储了游戏的所有数据。
游戏设置则定义在 GameConfig 类中，包含了己方阵营设置、主机/客户端设置、音效设置、提示设置、步时设置。GameConfig 定义了一个全局对象，更改设置后能够立即刷新。



显示模块

使用了 Qt 图形视图框架 QGraphics。
ItemChess 多继承 QObject 和 QGraphicsItem，完成单个棋子在各种状态下的显示操作，如用户选中时加边框并闪烁、需要提示时加白色圆圈等。
GameScene 继承 QGraphicsScene，完成棋盘背景的绘制和 ItemChess 的管理。
GameView 继承 QGraphicsView，负责处理鼠标事件，将用户的落子请求传回 Controller，同时接受 Controller 发来的最新游戏数据，并通过 GameScene 更新界面上的 ItemChess。

控制和算法模块

Controller 类中集成了控制部分和游戏的核心算法，具体包括落子的合法性判断、是否将军、输赢判断、倒计时控制、数据坐标系和视图坐标系之间的坐标变换、获取选中棋子的移动范围、走子操作、投降操作、播放音乐、对 GameView、TcpServer、TcpClient 的控制。
Controller 含有一个 GameData 私有对象，存储了游戏的所有数据，Controller 通过访问和更新 GameData 数据，完成整个游戏流程。

网络模块

网络编程框架

只使用了 QTcpSocket 和 QTcpServer 两个类作为 TCP 通信的框架。

自定义数据传输协议

数据包分为数据头和数据正文两部分，数据头包括消息总长度和消息类型。

消息总长度为 qint64 类型，是数据头和数据正文的总长度(包括这个 qint64 的长度)
消息类型有两种：棋子移动信息、游戏结束信息

```
enum MsgType { MOVE, GAMEOVER };
```

Header

```
qint64 length
MsgType msgType
```

Body

```
● MOVE
int srcRow          // source row index
int srcCol          // source column index
int dstRow          // destination row index
int dstCol          // destination column index
● GAMEOVER
GameSide winner     // RED, BLACK or NEUTRAL. specifying the winner side.
```

客户端

自定义 TcpClient 类，继承 QTcpSocket，对底层的读写进行了进一步的封装。

封装 readyRead()信号

将 readyRead()信号与槽函数 slotReadyRead()相连，在其中用循环将所有数据读出，确保数据完整性，将完整数据转换成 QByteArray，并触发信号 dataReceived(QByteArray, qint64)，将数据分发出去。

等待连接功能

实现函数 keepConnectingToHost()，通过 connectToHost()尝试连接到主机，若连接成功，则返回，若连接失败，则启动 timer，然后返回，3 秒后自动重连，直到连接成功，连接成功时检测到 connected 信号，则关闭 Timer，不再继续请求连接。如此可避免 waitForConnected()造成的线程阻塞。

取消等待连接功能

实现函数 cancelConnectingToHost()，关闭 timer，停止自动重连。

服务端

自定义类 TcpServer，继承 QTcpServer，对底层的读写进行了进一步的封装。

维护用户连接链表

```
    QList<QTcpSocket*> tcpClientList;
```

当有新的连接到来时，链表中增加一个指针，有连接断开时，链表中移除相应的指针。

创建新的连接

当有新连接到来时，会触发函数。

```
virtual void incomingConnection(qintptr handle) override;
```

因此，TcpServer 复写了此函数，当有新的连接到来时，在 TcpServer 内创建一个 QTcpSocket 对象，并把这个对象的 socketDescriptor 设置为 handle，以区分每一个 QTcpSocket 对象。将新创建的对象指针添加进 tcpClientList 内。

断开连接

当客户端断开连接时，会触发 QTcpSocket::disconnected()信号，服务器接收到这个信号时，在用户连接链表 tcpClientList 中根据查询 SocketDescriptor 相同的 QTcpSocket 对象指针，将其删除，并在列表中移除。

工作流程

连接过程

若服务端先开启监听，则客户端调用一次 connectToHost()即可完成连接。
若客户端首先尝试连接，此时若连接失败，timer 启动，间隔 3s 再次尝试 connectToHost()，直到连接成功，timer()停止为止。

数据传输与校验

当双方建立连接之后，一端向另一端发送数据传输协议规定的内容，发送端一次发送完成，接收端先接收消息头，读出消息总长度，并判断消息的实际长度是否与之相等，如果不相等，则弹出提示，说明数据传输错误。若相等，则可根据协议正常解析消息，并更新游戏数据。

设计感想

棋盘翻转与数据一致

考虑到用户的使用体验，在本次象棋界面设计中，必须保证己方阵营在界面下方，对方阵营在界面上方，这样一来，就要根据不同的阵营显示出不同的界面。在开发过程中，我尝试过两种解决方案。
解决方案 1：双方均以视图坐标系建立棋盘，此种方法的优势是，双方显示的逻辑、走棋事件的处理逻辑均是完全相同的，不需要对红黑阵营分别处理。但是在网络传输、棋盘同步、保存和加载残局上就非常麻烦，每次发送前和接收后都需要先计算翻转坐标，再对其进行进一步处理，保存和加载残局时同样需要处理。
解决方案 2：选取一个统一的坐标系，双方在这个坐标系下建立相同的棋盘，这种方法的优势是保证了数据的一致性，双方存储的数据都是一模一样的，因此在网络传输时不需要进行坐标变换，更加方便。但是在显示上则需进行处理，将统一坐标系变换成不同阵营的视图坐标系，显示不同的视图，处理用户点击事件时，也同样需要将视图坐标系转换为统一坐标系。
本次设计最后采用的是第二种方案，完成了视图坐标与统一坐标之间的转换，保证了数据的

一致性。

Server 端的处理

Server 与 Client 建立连接的方式有许多种，示例代码中通过连接 newConnection()信号，在槽函数中调用 nextPendingConnection()，建立连接，这种方法稍麻烦，且若不加判断的使用 nextPendingConnection()，很有可能会返回一个空指针，导致程序崩溃。
本次设计采用了另一种方式，就是重写函数：

```
virtual void incomingConnection(qintptr handle)
```

当有新连接到来时，此函数会被调用，只需要在函数内部进行相关处理即可。此种方法并不需要任何信号槽的连接，也不会出现空指针，还可以直接获取 SocketDescriptor，比前面一种方法更加安全快捷。