问题求解（二）项目-第二阶段报告

211240088 徐沐杰

完成情况

基本架构

注：本项目未使用提供的基于 **GVF** 的游戏开发框架，但仍然使用了 **GVF** 的静态显示功能。

游戏的整体逻辑分为两大主体，客户端 和 服务端 。其中，服务端负责计算实时游戏数据，客户端负责接受用户输入以及把游戏数据可视化的渲染出来。游戏的基本时序单位是游戏刻 `TICK`，每一刻，服务端都让游戏局面往前推进一步并将运算结 果发送给客户端。

相比很多常见的服务端-客户端通信方案，本项目采用的方案是发送整个地图的数据而非地图变化的差分 / 其他玩家移动的数据包，整个游戏逻辑完全运行在服务器上，用比较大的传输代价避免了可能出现的多客户端同步问题带来的麻烦。

客户端

客户端的主组件是继承自 `QMainWindow` 的 `MainWindow` 类。

- 获取用户输入

  主窗口本身分别通过菜单栏的 `QAction` 发出的信号和键盘事件来接受用户输入，其中 `QAction` 的信号主要被用作控制游戏流程，包括存档、重启、暂停、退出等，和角色控制有关的键盘事件则被处理后转发服务器。

- 管理游戏资源

  作为主组件，将游戏资源置于其管理之下是非常自然的，它管理的资源包括本地服务器（或远程服务器代理）线程 `localServerWorker`，`GVF` 的对象，玩家信息 `PlayerInfo` 和设置 `UserConfig` 等。

- 显示游戏界面

  主窗口的中心组件是 `QGraphicsView` 的一个派生类（主要是重载了其键盘事件让它忽略方向键事件），它负责作为游戏处理显示的主要容器 `QGraphicsScene` 的宿主，同时还可以便捷地通过 `setScene()` 实现场景的快速切换，如从开始菜单切换到游戏界面等。

- 状态栏

  状态栏分为永久信息栏和临时信息栏，前者用于显示 fps，服务器和客户端当前的状态等，后者则用于显示一些组件的提示。

- 激励接口

  客户端留下了一个用于接收服务器信号的槽函数 `advance(data)`，供监听服务器的线程/本地服务器调用（后面会提到）。

客户端实现的主要功能列下：

- 转换存档：将一阶段的二进制存档转化为二阶段的 json 存档格式（json 是本阶段对象式数据存储和传送的主要格式）
- 重载 `QGraphicsObject` 类实现了带有 `clicked()` 槽的 `GraphicButtons` 类，用于生成场景内的按钮。

此处提到的类见 `gitems/graphicbuttons.h`，`playerinfo.h`，`userconfig.h` 和 `mainwindow.h`。

服务端

客户端需要有一个线程定时的调用槽函数 `advance(data)`，这不仅是对客户端的激励信号，同时也为客户端提供了一次更新所必须的信息，至于这个线程提供的数据到底是来自其本身还是仅仅只是在转发（监听）一个远程服务器的信息，客户端并不在乎。本阶段暂且只实现了一个 `LocalServer` 类（`serverthread.h`），含有该类实例的线程会生成一个计时器，每个游戏刻调用一次 `run()` 函数，它包含了必要的运行游戏和发送内容的操作。

服务器留有如下流程控制接口（都通过信号槽连接到客户端）：

- `start()` 和 `close()` 可以开/关服务器。
- `suspend()` 可以使服务器在暂停和运行状态中切换（断开/连接计时器）。
- `changePlayerBehavior()` 和 `releaseBomb()` 接受角色控制输入。
- `generatePlayer()` 用于按客户端的玩家信息生成新角色（一般是开始游戏时）
- 存读档等小功能。

服务器主要透过其内部存储的 `coreData (GameMainMap *)` 在每次 `run()` 时的变化运行游戏，并在每刻处理结束时将其打包为 json 对象发送给客户端，激励客户端显示该刻的内容。

此处提到的类见 `serverthread.h`。

游戏核心

游戏的核心运行逻辑主要依靠 `coreData` 存储的实体类及其派生类来实现。

实体类

实体系列的类派生关系如下：

- `Entity`
- `ItemEntity`：`Entity`
- `MoveableEntity`：`Entity`
- `BWEntity`：`Entity`
- `WaveEntity`：`BWEntity`
- `BombEntity`：`BWEntity, MoveableEntity`
- `Player`：`MoveableEntity`

`Entity` 类拥有实体最基本的属性和方法，包括：

- `pos` 在地图中的位置。

- `entityType` 实体类型，由一个枚举类定义。
- `update(coreData)` 更新地图中本实体的信息。
- `die(coreData)` 从地图中擦去本实体。
- `wear(coreData)` 进行一刻更新，包括移动，减少炸弹时间，无敌时间等。

从这里可以看到，我们使用实体操作都是将实体从地图中抽离出来，然后令实体按照自己的属性（最重要的就是其位置）对地图进行操作来实现实体更新的，下面我们就会知道这样操作的原因。

伪多态

其实本人基本并没有把多态玩的太明白，为避免难以理解的问题发生，以及照顾对于 json 对象的兼容性，本项目采用 `QVariantMap`（aka `QMap <QString, QVariant>`）进行动态类型对象的存储，这便是所谓伪多态了（这类似 JavaScript），它的好处在于不需要摆弄复杂的指针而进行更高层次的抽象（全变成容器），坏处是性能损耗很大（然而我们在这个项目里并不很关心性能）。

为引入伪多态，我们为 `Entity` 建立虚函数 `Variant()` 生成一个对应的 `QVariantMap`，注意 `QVariant` 可以是 `QVariantMap`，`QList <QVariant>` 和基本数据类型，因此其表示对象的能力是通用的，据此，其它的类也可以递归的被封装成这种通用对象，并能很容易的被转化为 `QJsonObject`（说的就是 `coreData`）。

实体容器

有了上面的伪多态，我们很容易地把所有实体都视作同一类型存在一个容器里，由于地图的结构，我们很容易想到开辟一个二维数组（实际上我采用一维映射二维）来安放所有这些实体，鉴于一个点可能有多个实体，我们对于每个点建立一个 `MapNode` 类实例来管理一个点内的实体（实际上就是一个 `QVariant`数组），而 `coreData` 就是一个 `MapNode` 数组。由此我们就把实体容器构建好了，并且游戏的全部信息都在这个容器之内，只要将其转化为 `QJsonObject` 转发给客户端就可。

因此我们也能够理解了以上的实体操作逻辑：每次运行时遍历实体容器，我们都是将遍历到的伪多态实体转化为普通实体，然后再令这一普通实体对它的来源地图直接进行修改——其中也包括它在容器里自己本身的「影子」。

关于实体类对游戏逻辑的具体实现，内容过于琐碎复杂，这里仅作概要介绍表明实现进度。

- 通用实体类：拥有无敌时间，现阶段主要用在道具生成，炸弹炸开软墙后，内部的道具会获得一段无敌时间，保护自己不被炸弹炸掉；拥有血量，软墙，道具，玩家的血量都很低，会被直接炸死，其它的则很高（如硬墙，炸弹，光波），永远不会被炸死。
- 可移动类：这个类没有实例，但被玩家和炸弹继承，继承这个类的实体可以移动，拥有统一的移动逻辑（目前由于实际原因还未被统一），包括滑动和被推动。
- 定时触发类：这个类也没有实例，但被炸弹和光束继承，它们具有一个计时器，归零时发生特定的行为（对于光束是消失，对于炸弹是消失并爆炸）
- 玩家类：可以拾获其位置或移动向的位置的道具，获得增益，并有特定的朝向属性用于切换不同朝向之贴图。
- 炸弹类、光束类：炸弹类在时间归零时释放光束，在未归零时客户端会根据其时间选择合适的贴图。

此处提到的类见 `server/component.h`。

未实现好的功能（部分已预留了接口）

这个部分主要是对未完成必做功能的开脱以及未来实现它们的可行性探讨。助教哥哥手下留情😕

图形按钮

- 未实现帮助和退出等按钮，但实际上和开始菜单通用的类已经被实现，很容易添加上去。

显示和动画

- 未给炸弹安排动画（对应资源文件已经导入，对应逻辑已经实现，但尚未通过测试）。

机器人

- 是否是机器人通过玩家实体信息内的 `isAi` 属性控制，如果一个玩家是机器人，那么 `run()` 将为其指派（固定的） AI 决策类实例进行决策（该类未实现），在这个实现框架下，机器人可以直接通过 `coreData` 的二维数组方便的访问全局信息并以此作为基础进行决策。

地图

- 大小未按照要求，但由于已经能够从第一阶段导入地图，且第一阶段实现了地图编辑器，修改地图是很简单的。

计分板

- 尚未实现，但角色分数已经有内部实现，可视化计分板计划通过 `QGraphicsObject` 的重载实现。

核心逻辑

- 炸弹和玩家的移动逻辑还没有统合好，以致规则有点奇怪，不过未来能够很容易的通过它们共同的基类统合。
- 一些道具参数没有调整得太好，不过通过修改 `server/component.h` 里的常量可以很容易控制。

自建框架为未来实现提供的接口

这个部分解释了使用自建框架的一些原因。

远程服务器

前面说过本地服务器线程，事实上，如果将本地服务器线程的游戏核心部分替换为监听并转发远程服务器发送的游戏内容（事实上就是存档了），客户端部分不需修改就可以直接实现网络联机功能，配合局域网网络发现就可以积极的进行局域网联机（不过不打算实现这种网络发现功能，而是要输入 ip 联机）。

远程服务器将有两个版本，一个是集成在客户端内的版本，一个是独立版本，两者都将有独立的设置用于修改游戏参数。

存读档

藉由服务器功能，这个功能很容易实现，因为服务器发送的就是游戏存档，只需要写入磁盘即可，这个功能已经实现，但读档恢复时需要重新绑定客户端输入和服务器端玩家，这个暂时没有实现，目前的想法是令客户端和服务端记住「房间号」并写入存档，服务端广播房间号即可令客户端恢复远程和本地玩家的对应关系。

设置和高级选项

项目预留了可修改的 `PlayerInfo` 和 `UserConfig`，主要用于修改设置，计划支持的设置有:

- 修改本地玩家输入键值，名称（`PlayerInfo`）。
- 修改本地服务器和客户端偏好设置 `UserConfig`，如是否自动读档等。

这些都将通过重载 `QDialog` 实现，参照实现地图转换器的对话框方法。

心得体会

- 返回引用的函数能链式调用就链式调用，要复制的话要加 auto &。
- 返回值优化（RVO）。此前为避免函数返回容器时发生复制容器的情况（主要是 `Variant()`），一度采用了建在堆上返回指针来避免出现这种情况的方案。然而这完全是多此一举。且不说 qt 容器本身就有所谓隐式数据共享来防止这种数据在栈间对象的复制， `Variant()` 的返回容器由于返回值优化本身就会被构建在目标栈上，担忧的情况完全不可能发生。
- 多线程编程时要考虑线程安全，以及注意线程被阻塞的情况。在多线程编程中要注意尽量让各个线程只访问自己的数据（不要把 `coreData` 直接暴露给主线程），而把线程间通信和数据共享的工作交给 qt 自有的信号槽来完成，它的机制已经是线程安全的了，无需我们去关心。同时主线程在管理服务器线程的时候其本身会被阻塞，没有办法即时运行后者触发的槽函数，直到管理流程结束为止，这一点要求我们在管理子线程的过程中需要子线程对主线程槽函数的回调时，不要阻塞主线程，否则就不要使用回调。