

# Project Snake 设计文档

© 计96 2019011346 聂鼎宜 (@Delphynium)

## 一 框架设计

设计一个简单的图形小游戏，笔者认为，只需要解决三个问题，一切便迎刃而解。它们是：**控制、演算、渲染**。

控制，意味着玩家可以在与图形界面交互的过程中，影响游戏的运行。游戏主动提供给玩家的选项应该足够丰富和完备，与此同时，游戏还应该虔诚地监听玩家的输入操作，并及时给予反应。这种反应，即是高频率的演算，推演、决定游戏的进程。而此过程的最终目的，是为了得到一系列有意义的、可渲染的数据，即生成图形的“种子”。当我们以特定的方法将数据转化为图形并打印到视窗中，一条完整的、足以被称为“游戏”的交互链便诞生了。

C++ 面向对象的特性为我们实现这条交互链提供了绝佳环境。在 Qt 庞大的类库中，所有类都是图形，因而是“可渲染的”；得天独厚的信号-槽机制进一步使其“可控”。因此，我们可以尽管放开手，用类的语言去抽象我们预期中的游戏为一个个组件，最后在组建中建立演算逻辑，程序便真的成为游戏。下面，让我们开始做贪吃蛇吧。

在常规的 Qt 应用程序中，MainWindow 对象是一切对象的父亲，他因而应具有控制程序的最高权限。所以，**我们将游戏的控制权全权交给它**，这包括控制游戏开始、暂停、退出、S&L 等一系列动作的控件，以及监听玩家在贪吃蛇游戏中唯一需要做的事——按动键盘的监听权。这就是笔者在 MainWindow 对象中设置了成员变量 `QAction *` 数组及 `QPushButton *` 数组，并重写了它的 `keyPressEvent(QKeyEvent *)` 函数的原因：我们希望一切与游戏控制直接相关的对象、方法都成为它的成员。**但有一个例外。**

贪吃蛇作为一个图形小游戏，其灵魂毫无疑问在于游戏小蛇吃果子的游戏场景中。这里所谓的场景，其实可以有多个姓名——视窗、显示屏、渲染设备……无论如何，**它是一个可以抽象的实物**。因此，笔者编写了继承自 `QWidget`（自然也就继承了 `QPaintDevice`）的自定义类 `Scene`，并为 MainWindow 对象实例化一个它的成员 `scene`。它的主要职责，有且仅有输出图形。但这不代表 `scene` 没有副职：当玩家在地图中自定义墙体时，被鼠标直接交互的对象，理应就是 `scene`。因而，**我们把鼠标输入的监听权交给 scene**，而不是 MainWindow 对象，并重写 `Scene` 的 `mousePressEvent(QMouseEvent *)` 函数。在贪吃蛇游戏中，蛇身、墙体和果实的本质都是图形，因而他们都是 `scene` 的一部分。所以，在笔者的设计中，存储蛇、墙、水果的容器都是 `Scene` 的成员。标示蛇的一系列状态的变量自然也由 `scene` 来维护。更自然地，`Scene` 的 `paintEvent(QPaintEvent *)` 函数被重写了。

一个简单的框架示意：

```
MainWindow {Ui {menuBar
mainToolBar
QGroupBox {QPushButton
...
}
...
}
Scene{snake
wall
fruit
...
}
...
}
```

## 二 数据结构

说完了控制，我们来说演算。

演算的直接参与者，或者说，基础，就是数据结构。贪吃蛇游戏中，占用演算资源的主要是蛇的移动、与墙体和水果的碰撞及其判定。因此，设计存储蛇、墙、水果的好的数据结构是十分重要的。

在现实世界中，蛇的爬行是一个连续的动作；但在贪吃蛇的游戏世界中，蛇的爬行是离散的：其本质是**头进尾处**——这便是一个 `queue`，或者说，比 `queue` 更容易操作的 `list`。这里，我们选用 Qt 类库中重新封装的 `stl` 容器 `QList` 来存储蛇的信息，`QList snake` 中的每一个元素是一个 `<int, int>` 型的 `pair`，存储着蛇的一个身体块（`body`）的横纵坐标。

对于墙，笔者并未使用一个 `bool` 型的矩阵来存储某个位置是否有墙的信息。一是因为，贪吃蛇游戏中可通行地块往往是远多于障碍物的，这样做会有大量**冗余**；二是因为，我们认为，既没有蛇，也没有墙，又没有水果的格子是**无意义的**，因此，我们只抽象前面提到的那三类具体的实物，而不是连同无意义的部分一起存储。墙并无进出顺序的特性，因此使用比 `list` 更快的 `vector` 是最佳选择。我们于是使用 `QVector` 来存储每个墙体的坐标。

对于水果，虽然常规贪吃蛇游戏场上只能同时存在一个，但出于可扩展性考量，笔者使用了一个 `QList` 来存储水果坐标，以便日后加入 `bonus reward`，或是在双人贪吃蛇模式下允许两个及以上的水果同时存在。

```
1 class Scene: public QWidget{
2     ...
3     QList<std::pair<int, int>> snake;
4     QVector<std::pair<int, int>> wall;
5     QList<std::pair<int, int>> fruit;
6     ...
7 };
```

## 三 运行逻辑





贪吃蛇是一款简单的图形游戏，这体现在它**不是实时渲染**的——“帧”是游戏进行一次演算、渲染的最小时间单位。因此，实现贪吃蛇运行逻辑的基础是实现帧机制。

好在，Qt 的信号-槽机制为该问题提供了非常便利的解。我们在 `scene` 中维护一个 `QTimer` 变量 `frame_timer`，每次启动时，时间间隔都设置为帧时间（`FRAME_TIME`）。再将 `QTimer` 内置的信号 `timeout()`，连接到负责演算、渲染的函数上，就完成了。

下一个问题：一帧的时间里，游戏应该干什么事？对于贪吃蛇游戏，答案是：**蛇的移动和事件的判定及发生**（得分、生长、撞墙等）。

我们在默认蛇一帧移动一格的前提下，移动似乎很好实现，只需要在 `snake` 中维护一个 `heading` 变量，指示蛇当前**应该**移动的方向，并在下一帧时参考这个方向让蛇进行移动即可。（笔者的代码里出于简便考量，将其直接维护在了 `scene` 中。这其实不无道理，移动的方向独立于场景是无法明确指示的。）这时我们遇到了新的问题：贪吃蛇是不可能原地掉头（转向相反方向）的。即，`heading` 不能突变为它的相反方向。解决方法很简单，或者在改变 `heading` 前加入判定机制禁止这一行为，或者直接在 `mousePressEvent` 的监测中无视不合理的输入，即与当前 `heading` 相反的方向键输入。

**然而**，新的问题再一次出现，而且这个问题比较隐蔽，不仔细测试很可能无法发现。由于玩家的输入行为并不是帧机制的一部分，即**玩家的输入是即时的也是随时的**，这就允许了下述情况的发生：

当蛇在向上移动时，在两帧之间快速先后按下 、 或 、，蛇在下一帧时便掉头 180 度撞向自己。这绕过了我们的判定机制。

这就是笔者在 `heading` 变量外，又维护了一个同类的 `facing` 变量的原因。它指示的并不是蛇该往哪个方向移动，而是**蛇头当前帧正在往哪个方向移动**（即蛇头的那个 `body` 和与蛇头相连的那个 `body` 两者确定的方向）。这个变量也在帧机制中，每一帧都需要更新。只要避免了它的突变（反向），就可以彻底根绝这类问题。

对于蛇的生长，笔者维护了一个 `int` 型变量 `grow`，存储蛇还需要生长几格。每次来到新的一帧时，判定它若大于 0，则此次移动不抹去蛇尾，然后 `--grow`。初始时，只在场景中放置蛇头，并将 `grow` 置为初始长度 - 1，这样做还能实现游戏开始时“蛇出洞”的效果。当蛇吃到水果时，将 `grow +=` 所期望的量，蛇的生长机制便做好了。

至于蛇与水果、墙体的碰撞，将简单的坐标运算加入帧机制便可做到，这里就不再赘述了。

## 四 控件一致

一套完整成熟的控件，除了高效性，还应具有**规避违法操作的能力**。比如，在游戏进行时，禁止“开始游戏”操作。这就需要对各控件及时进行 `setEnabled(bool)` 操作。

然而，在 `menuBar` 和 `mainToolBar` 中的控件的本质是 `QAction`，而 `QPushButton` 是独立于 `QAction` 的，所以我们需要实现一个机制，能够做到在游戏状态发生改变时，`QAction` 和 `QPushButton` 两套空间同时作出响应和调整。

为此，笔者在 `scene` 中又维护了一个对 `parent` (`MainWindow` 对象)可见的 `state` 变量，指示游戏状态，它有四种取值：`Ready`、`Playing`、`Paused`、`Ended`。我们希望，每当 `state` 变量被覆盖写入时，调用 `MainWindow` 中的 `updateCtrl()` 方法来更新控件的状态。然而，`Ready`、`Playing`、`Paused` 三个状态对应三个操作（`restart` 或打开程序、`start`、`pasue`），应由负责控制的 `MainWindow` 对象来置 `state` 为这些状态；但 `Ended` 状态只有 `scene` 这个对象内部通过判定才能进入，所以，`state` 的操作权是**分散的**。为此，我们在 `Scene` 中编写信号函数 `gameOver()`，连接到 `MainWindow` 对象中，来托付操作权。

至此，我们已实现全部需求。可喜可贺，可喜可贺。

（完）