Puzzle Game 输入系统

1. 预期

抽象游戏按键，以枚举值代表单个游戏棋步（Single\_Move），而不是由游戏逻辑完成从按键代码（Keycode）到具体游戏棋步的映射。

统一游戏输入接口，游戏逻辑无需关心具体输入设备（按键、鼠标、虚拟手柄等等）。

完成移动端虚拟游戏手柄的UI逻辑，包括操作键、十字键和摇杆。

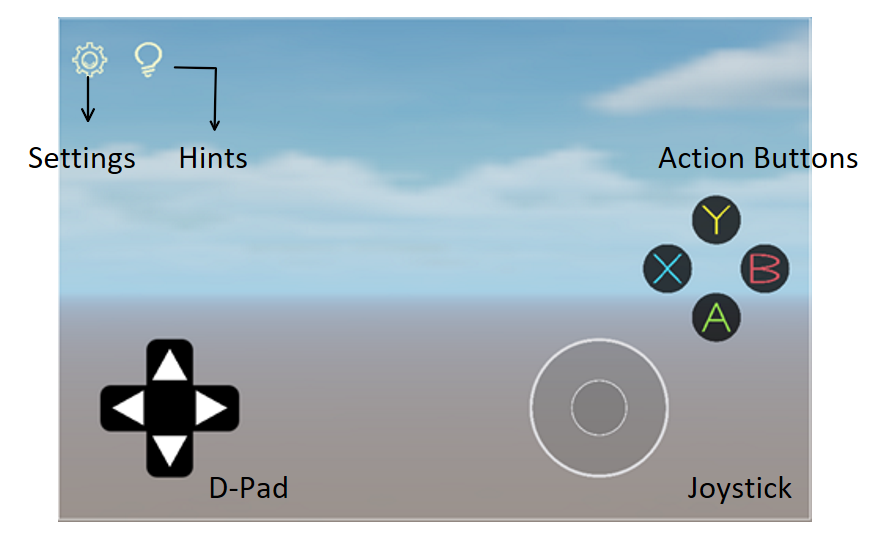
持续按压按键时，能持续创建游戏棋步，保证玩家的输入都得到正确的处理。

实现可自定义化的按键映射配置，保存到玩家本地内存中，能够被重复读取使用。

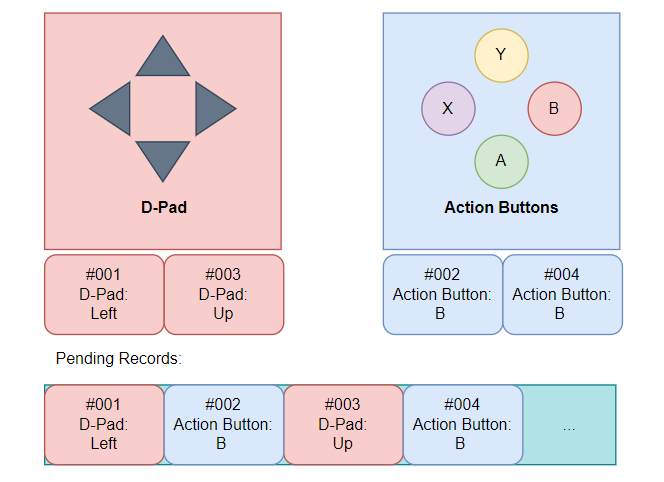
作弊检测，能够检测单个游戏帧内的异常输入情况。

1. 问题

（1）需要按功能区分按键，以提示按钮和设置按钮为例，部分按键响应与游戏逻辑无关，无需暂存到游戏主循环处理，而可以即时处理。



（2）由于游戏主循环是一个100ms一次的定时调用，在此逻辑帧内的所有用户输入需要被暂存统一处理。



用户操作步（Controller\_Proc\_Move）在一个逻辑帧内只会被处理一次，但存在用户在一个逻辑帧内连续两次按压的情况——如同时按压D-Pad Left和D-Pad Up。如果只展示了一步角色移动，会给用户造成输入与画面不匹配的感受，游戏输入系统需要尽可能的让玩家的所有输入都被正确的处理。

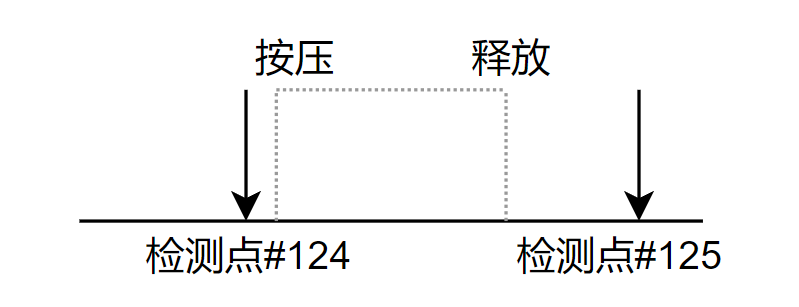
参考：<https://learn.microsoft.com/zh-cn/gaming/gdk/_content/gc/input/overviews/input-fundamentals>

（3）自定义化配置不仅是键盘按键到游戏按键的映射，还包括虚拟手柄的D-Pad和Joystick等UI组件的复用——即可用于角色调整朝向，也可用于角色技能释放。

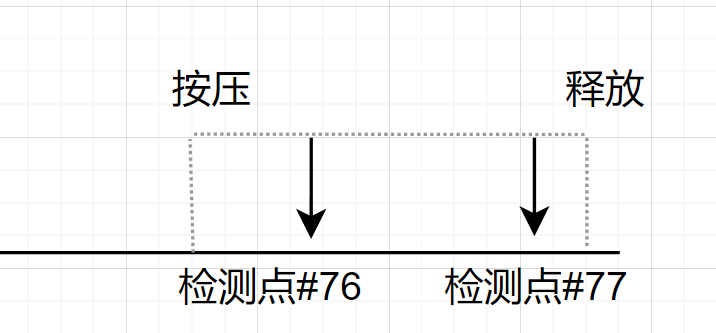
（4）采样频率

由于持续按压按键时，会持续生成游戏棋步，但是由于逻辑帧的更新频率是确定的，就存在由于采样频率造成的错误。

（a）采样频率不足，可能导致用户输入被忽略：



（b）采样频率过高，可能在用户按压较长时产生意料之外的多余棋步：



1. 实现

（1）虚拟手柄的UI实现

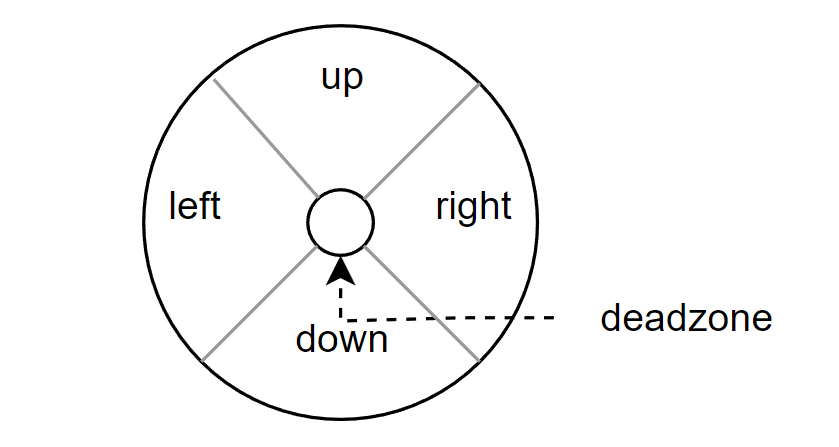
Cocos Creator引擎的输入事件系统在触摸事件方面提供了触摸开始（Touch\_Start）、触摸移动（Touch\_Move）、触摸结束（Touch\_End）、触摸取消（Touch\_Cancel）四个事件，在编写虚拟手柄的UI逻辑时都有使用。

其中触摸取消事件代表用户在保持触摸移动的状态下，触摸点移出了触摸事件注册对象在屏幕空间内的尺寸范围，许多常见的错误都来自没有对触摸取消事件进行处理——例如操作杆UI，在触摸点出界时没有归位，导致

在创建UI组件时，有Virtual\_Controller类为其注入input字段用于在触摸事件发生时，更新输入状态。

其中十字键和动作键只注册触摸开始事件，在触摸开始事件发生时，更新对应的按键状态为按压态，更新UI组件，通过缩放提示用户正在产生输入；在触摸结束和触摸取消事件发生时，更新对应的按键状态位释放态，重置UI缩放。

操作杆需要注册触摸移动事件，触摸点位于相同扇区时不做处理，而在触摸点移动到不同扇区时，更新原本位于按压状态的按键为释放态，而更新新扇区对应的按键状态为按压态。



参考：<https://docs.cocos.com/creator/manual/zh/engine/event/event-input.html>

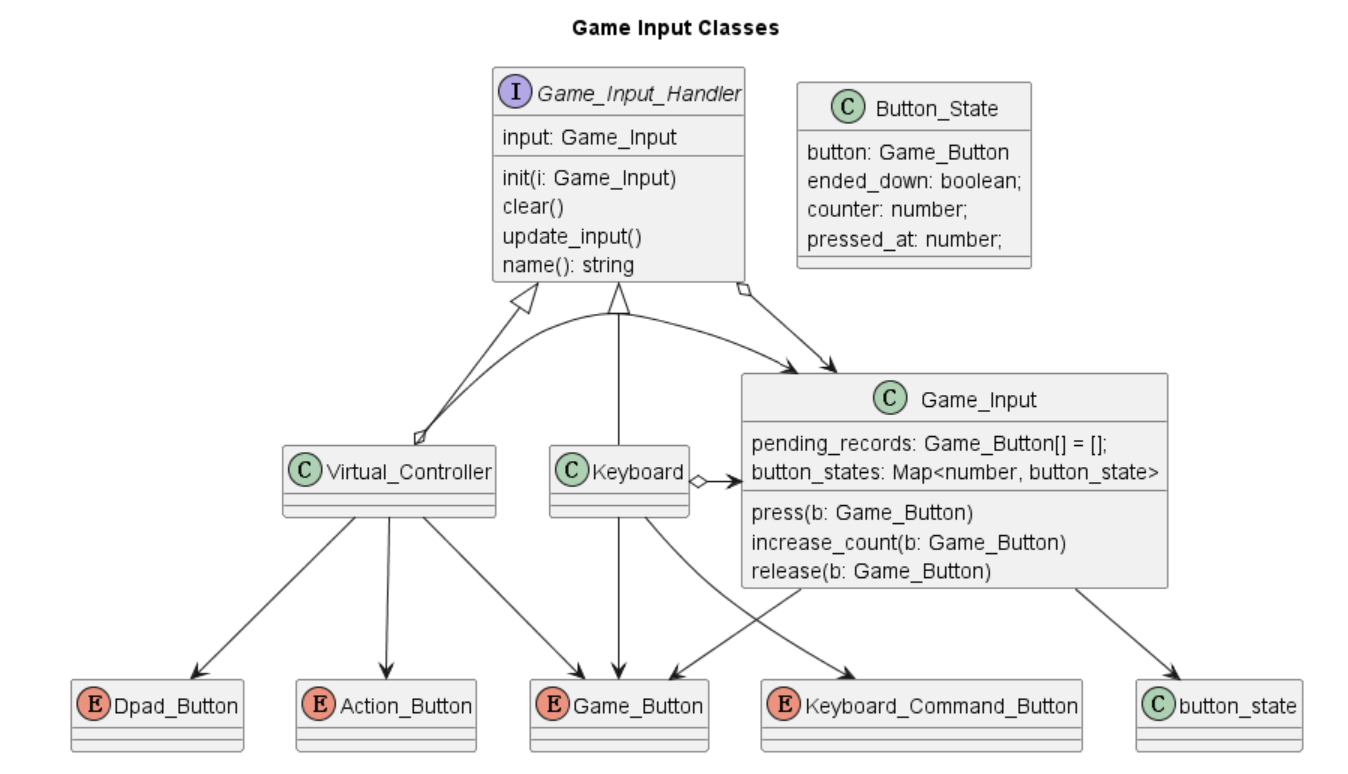
（2）抽象游戏按键

表 1 游戏按键枚举值

|  |  |
| --- | --- |
| NULL | 状态为空 |
| MOVE\_LEFT | 角色向棋盘左侧移动 |
| MOVE\_RIGHT | 角色向棋盘右侧移动 |
| MOVE\_FORWARD | 角色向棋盘近侧移动 |
| MOVE\_BACKWARD | 角色向棋盘远侧移动 |
| FACE\_LEFT | 角色朝向棋盘左侧 |
| FACE\_RIGHT | 角色朝向棋盘右侧 |
| FACE\_FORWARD | 角色朝向棋盘远侧 |
| FACE\_BACKWARD | 角色朝向棋盘近侧 |
| UNDO | 撤销当前棋步 |
| RESET | 重新加载当前关卡 |
| SWITCH\_HERO | 切换当前受控角色 |

（3）抽象接口

游戏逻辑依赖游戏输入类（Game\_Input）而不是具体输入设备，由各个游戏设备的适配器类独立完成抽象游戏按键的更新。



（4）持续行为的实现

游戏按键状态类见上图，counter字段代表当前按键出于按压状态下，被游戏逻辑处理的次数，按键第一次按压时counter赋值为1，同时立即加入暂存记录，以保证每个按键行为至少被成功处理一次，下面的程序是关于检查按压时间是否充分（应再次被处理）：

PROCEDURE pressed\_long\_enough(Game\_Button button);

BEGIN

state ←game\_button\_states.get(button);

UNLESS state.ended\_down RETURN;

duration ← Gameplay\_Timer.calcu\_delta\_time(Gameplay\_Timer.now() – state.pressed\_at);

IF duration >= state.counter \* VALID\_PRESSING\_INTERVAL

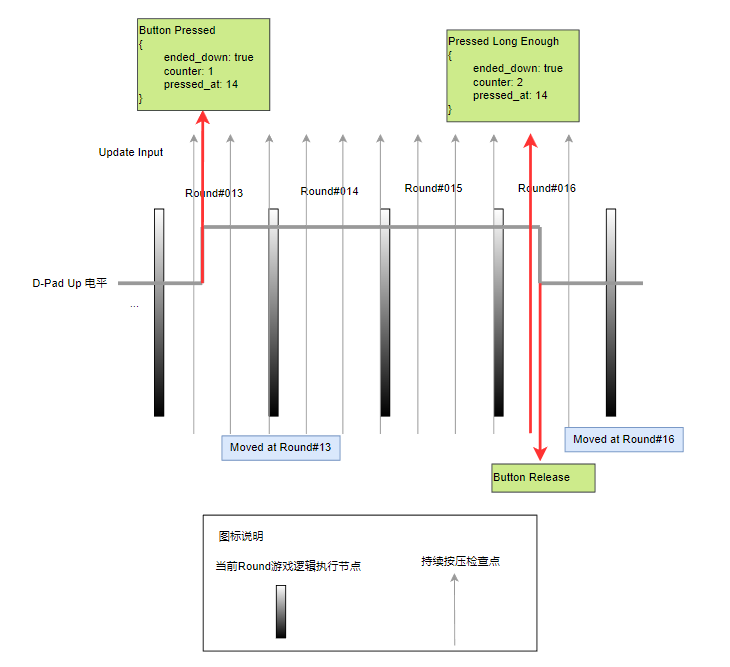
THEN RETURN true;

ELSE

RETURN false;

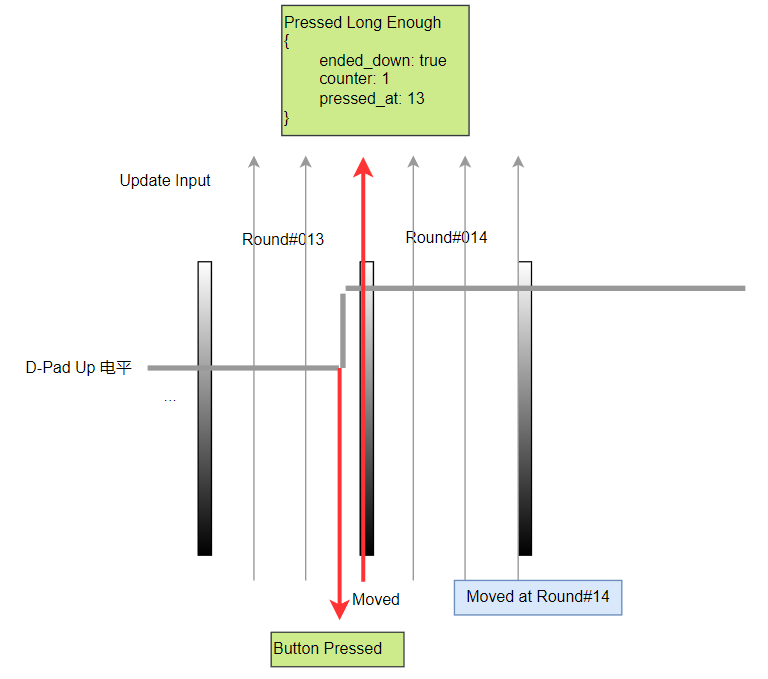
END

定时调用Game\_Input\_Handler的update\_input方法，遍历检查每个游戏按键的按压时间是否充分，若满足要求则加入暂存记录中。

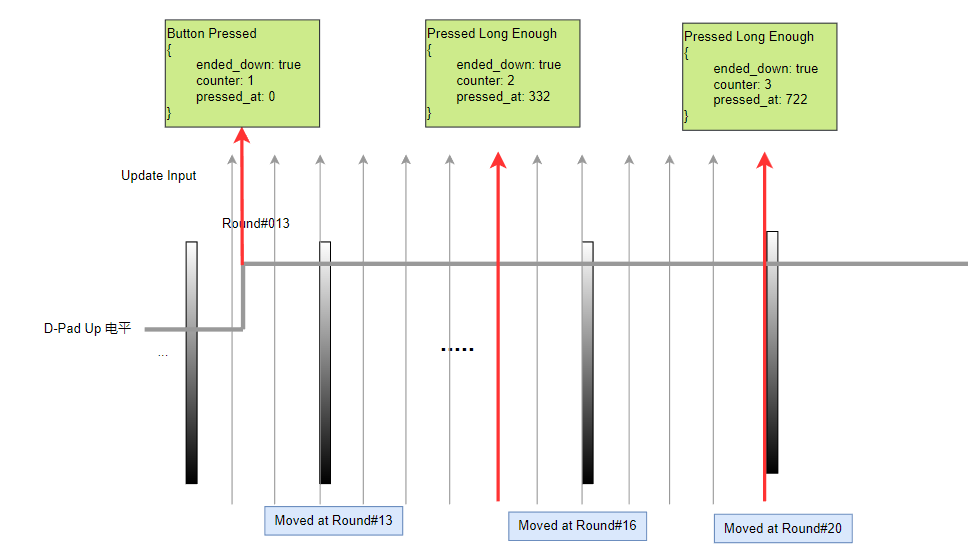


需要注意的是：

1. 与在各个Game\_Input\_Handler的定时调用update\_input中检测到持续按压时间充分而创建的按键记录相比，按键按下时创建的按键记录并不遵循相同的逻辑。按键按下事件若发生在当前逻辑帧的最后一次update\_input调用之后，则此次按键记录将会被推迟到下一逻辑帧执行，如果是角色移动棋步，将会导致下一帧发生的角色移动棋步被忽略。



1. pressed\_long\_enough函数中应使用当前逻辑帧的round索引值来表达gameplay time，而不是使用事件获取的调用来获取当前时间的毫秒表达。由于当前逻辑帧的所有输入会被统一处理，使用精确毫秒时间来固定间隔会导致间隔的游戏帧数反倒不同——如下图第一次移动发生在13帧，第二次移动发生在16帧，第三次移动发生在20帧，尽管该按键事件被记录的间隔是固定的，但由于输入层与逻辑层解耦，持续移动中画面表现反倒会出现卡顿感。



（5）自定义按键配置的实现