1. 前期调查

先看了一下俄罗斯方块的设定



发现Tetris99有10列，20行，会显示next的五个方块，会预测方块的下落位置。这些设定确实不错，准备直接照搬。

1. 问题分析
2. 基本界面绘制
3. 方块定义和移动
4. 碰撞处理
5. 运行起来
6. 获取键盘输入
7. 消行处理
8. 瞬间下落
9. 细节优化
10. Next方块预览
11. 加上BGM
12. 加上音效
13. 分数计算
14. 落点预测
15. 程序设计
16. 基本界面绘制

想法是用一个二维数组保存界面上每一个格子的状态，然后每一帧调用一次绘制函数进行页面的绘制

紧接着就遇到第一个问题，shell只有一维数组，这个问题很好解决，直接用一个函数实现一维坐标和二维坐标的一一映射即可

界面内部有10列，20行，再加上边框，分数显示，next显示等其它元素，暂时先开一个30\*30的数组。

遇到了shell的第一个坑，$?只能返回0~255的值，因此无法表示30\*30的数组，所以需要自己维护一个全局变量来表示函数的返回值

如果一个格子为空，值为0，如果是边框，值为1，接着用echo -ne来完成彩色输出

下列代码完成了基本边框的绘制

#ret这个变量用来表示函数的返回值

ret=0

#map用来维护界面的信息

#-1代表边框

#俄罗斯方块主界面的行列位置

colbegin=10

colend=20

rowbegin=3

rowend=23

#将二维坐标转换为一维编号，x为行号，y为列号

function Decode() {

local x=$1

local y=$2

ret=$(((x\*30)+(y)))

}

#将map[x][y]赋值为v

function Assign() {

local x=$1

local y=$2

local v=$3

Decode $x $y

map[$ret]=$v

}

function Draw() {

for((i = 0; i < 30; ++i)) do

for(( j = 0; j < 30; ++j)) do

Decode $i $j

case "${map[$ret]}" in

"0")

echo -ne "\e[1;33;40m \e[0m";;

"-1")

echo -ne "\e[1;33;44m \e[0m";;

esac

done

echo ""

done

}

function Init() {

#初始化为0

for(( i = 0 ; i <900;++i)) do

map[i]=0

done

#先将上底和下底弄成边框

for(( i=colbegin ; i<colend ; ++i)) do

Assign $((rowbegin-1)) $i -1

Assign $((rowend)) $i -1

done

#再把左右两边弄成边框

for(( j = rowbegin;j<rowend;++j))do

Assign $j $((colbegin-1)) -1

Assign $j $colend -1

done

#再补充上四个角

Assign $((rowbegin-1)) $((colbegin-1)) -1

Assign $((rowbegin-1)) $((colend)) -1

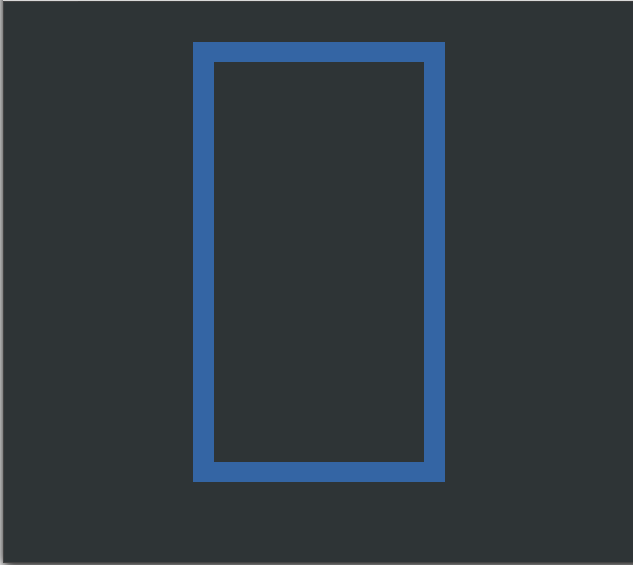
Assign $((rowend)) $((colbegin-1)) -1

Assign $((rowend)) $((colend)) -1

Draw

}

Init



1. 方块定义和移动

方块的移动可以分为平移和旋转

平移很显然，旋转相对平移就难了一点

因为方块的长宽最大为4，所以我们可以直接考虑4\*4的矩阵

以T型方块为例

可以用矩阵表示为

1110

0100

0000

0000

其逆时针旋转90°变为

1000

1100

1000

0000

我们发现这个变换过程实际就是行变为列，列变为行。

那么我们只需要定义7种方块的初始状态在4\*4矩阵下的表示，然后再写一个矩阵的变换函数就完事了

方块定义如下

O={1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0}

I={1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0}

S={0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0}

Z={1,1,0,0,0,1,1,0,0,0,0,0,0,0,0,0}

L={1,0,0,0,1,0,0,0,1,1,0,0,0,0,0,0}

J={0,1,0,0,0,1,0,0,1,1,0,0,0,0,0,0}

T={1,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0}

然后每一个方块的坐标以它的4\*4矩阵的左上角位置为坐标

下列为相关代码

#把正在操作的方块画进去

function Add() {

local x=$1

local y=$2

for((i = x ; i < x + 4 ; ++i)) do

for((j = y; j < y + 4; ++j)) do

Trans $((i - x)) $((j - y))

# echo "ret = $ret"

temp=${now[$ret]}

# echo "temp = $temp"

Decode $i $j

# echo "i = $i, j = $j, ret = $ret"

map[$ret]=$temp

done

done

pre=(${now[\*]})

prex=$x

prey=$y

}

#把正在操作的方块上一个时刻在的位置擦掉

function Del() {

local x=$1

local y=$2

for((i = x ; i < x + 4 ; ++i)) do

for((j = y; j < y + 4; ++j)) do

Trans $((i - x)) $((j - y))

temp=${now[$ret]}

if ((temp>=1))

then

Decode $i $j

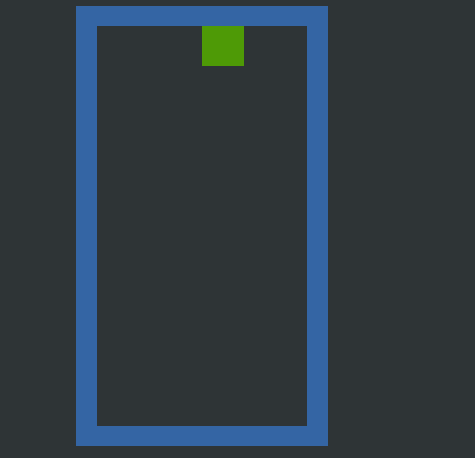
map[$ret]=0

fi

done

done

}



然后就是方块的旋转操作

因为方块的旋转就是矩阵的转置后上下翻转，考虑在矩阵的转置对一维编号的影响

旋转前:

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 16

旋转后:

4 8 12 16

3 7 11 15

2 6 10 14

1 5 9 13

所以如果用New表示新矩阵，Now表示旧矩阵

则New[4-y][x] = Now[x][y]

这里发现一个问题，就是3\*3的方块，如果用4\*4的矩阵进行旋转，位置会出现变动，因此必须长条单独用4\*4，其他的都用3\*3（偷懒大失败）

重新定义方块

O=(1 1 0 1 1 0 0 0 0)

I=(2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0)

S=(0 3 3 3 3 0 0 0 0)

Z=(4 4 0 0 4 4 0 0 0)

L=(5 0 0 5 0 0 5 5 0)

J=(0 6 0 0 6 0 6 6 0)

T=(7 7 7 0 7 0 0 0 0)

对于3\*3的矩阵的旋转就是转置后上下翻转，对于4\*4的旋转就直接转置(因为长条本身只有两种状态，如果转置后翻转反而会出现位置变动)

旋转部分代码如下:

#将二维坐标转换为一维编号，x为行号，y为列号

#这是针对n\*n的

function Trans() {

local x=$1

local y=$2

local len=$3

ret=$(((x\*len)+(y)))

}

function InvTrans() {

local id=$1

local len=$2

tempx=$(( (id - 1) / 4 ))

tempy=$(( (id - tempx \* 4) ))

}

#将当前方块逆时针旋转90°

function Rotate() {

pre=(${now[\*]})

local len=${#now[\*]}

# echo "len=$len"

if ((len == 16))

then

len=4

# echo "233"

for((i = 0; i < 4; ++i)) do

for(( j = 0; j < 4; ++j )) do

Trans $i $j $len

local temp1=$ret

Trans $j $i $len

local temp2=$ret

now[$temp1]=${pre[$temp2]}

done

done

fi

if ((len == 9))

then

len=3

for((i = 0; i < 3; ++i)) do

for(( j = 0; j < 3; ++j )) do

Trans $((2-i)) $j $len

local temp1=$ret

Trans $j $i $len

local temp2=$ret

now[$temp1]=${pre[$temp2]}

done

done

fi

}

关于方块的平移，没啥好说的，维护矩阵左上角坐标即可

function Down() {

prex=$nowx

prey=$nowy

((nowx++))

}

function Left(){

prex=$nowx

prey=$nowy

((nowy--))

}

function Right(){

prex=$nowx

prey=$nowy

((nowy++))

}

1. 碰撞处理

想法是移动时或者旋转时先计算出方块下一帧所在的位置，然后看是否与已用的方块有重叠，如果有重叠的话，就说明会发生碰撞，让旋转失败，或者触发方块落地响应

#判断移动后是否会发生碰撞

function Check() {

crash=0

local len=${#now[\*]}

if ((len==16))

then

len=4

fi

if ((len==9))

then

len=3

fi

for((i = 0; i < len; ++i)) do

for(( j = 0; j < len; ++j )) do

Trans $i $j $len

local temp1=$ret

Decode $((i+nowx)) $((j+nowy))

local temp2=$ret

if ((${now[$temp1]} != 0 && ${map[$temp2]} != 0)) #如果成立就说明发生了碰撞

then

crash=1

return

fi

done

done

}

然后在旋转和平移函数最后加上相应的Check流程。

function Down() {

prex=$nowx

prey=$nowy

((nowx++))

Check

if ((crash==1))

then

#发生碰撞,方块落地

nowx=$prex

nowy=$prey

prex=$tprex

prey=$tprey

FallToGround

fi

}

function Left(){

prex=$nowx

prey=$nowy

((nowy--))

if ((crash==1))

then

#发生碰撞,平移失败

nowx=$prex

nowy=$prey

prex=$tprex

prey=$tprey

fi

}

function Right(){

prex=$nowx

prey=$nowy

((nowy++))

if ((crash==1))

then

#发生碰撞,平移失败

nowx=$prex

nowy=$prey

prex=$tprex

prey=$tprey

fi

}

function Move() {

local type=$1

case "type" in

"0")

return;;

"1")

Down;;

"2")

Left;;

"3")

Right;;

esac

}

function Rotate() {

pre=(${now[\*]})

local len=${#now[\*]}

# echo "len=$len"

if ((len == 16))

then

len=4

# echo "233"

for((i = 0; i < 4; ++i)) do

for(( j = 0; j < 4; ++j )) do

Trans $i $j $len

local temp1=$ret

Trans $j $i $len

local temp2=$ret

now[$temp1]=${pre[$temp2]}

done

done

fi

if ((len == 9))

then

len=3

for((i = 0; i < 3; ++i)) do

for(( j = 0; j < 3; ++j )) do

Trans $((2-i)) $j $len

local temp1=$ret

Trans $j $i $len

local temp2=$ret

now[$temp1]=${pre[$temp2]}

done

done

fi

Check

if ((crash==1))

then

now=(${pre[\*]}) #发生碰撞，此次旋转无效

pre=(${tpre[\*]})

fi

}

如果方块发生碰撞且落地，就会触发落地事件

function FallToGround() {

Add nowx nowy

GetNextBlock

CreateNewBlock $nextBLock

}

它会先让当前的方块固定在map里，然后获取新的方块，把新的方块放在屏幕最上方，开始新一轮的方块下落

function GetNextBlock(){

return $(($RANDOM %7 + 1))

}

因为后续才会写展示Next方块的函数，所以先用随机数代替，后续就是取出队列头部元素了

function CreateNewBlock(){

case $1 in

"1")

now=(${O[\*]});;

"2")

now=(${I[\*]});;

"3")

now=(${S[\*]});;

"4")

now=(${Z[\*]});;

"5")

now=(${L[\*]});;

"6")

now=(${J[\*]});;

"7")

now=(${T[\*]});;

esac

nowx=3

nowy=15

prex=3

prey=15

tprex=3

tprey=15

}

根据下一个方块的编号，来生成新的方块，然后重置它的坐标

1. 每一帧之中的各种操作基本上都写完了，接下来就是让它们组合起来，自动运行

function Run() {

Init

GetNextBlock

CreateNewBlock $nextBLock

while [ 1 -eq 1 ]

do

for ((i = 0; i < 20; i++))

do

sleep 0.02

done

Del $tprex $tprey

Move 1

Add $nowx $nowy

Draw

tprex=$nowx

tprey=$nowy

done

}

写了一个函数，让方块自然下落，然后发现每次绘制时由于echo30\*30\*3个字符，终端的闪烁很明显，发现需要优化Draw函数

首先把空白背景

#输出三个空格

echo -ne " ";;

改成

#光标右移三格

echo -ne "\033[3C";;

虽然闪烁的现象有所减轻，还是肉眼可见

发现每次屏幕上变化的就只有当前操作的方块，绘制操作还可以进一步优化

可以在Add和Del函数中在更改map的时候直接修改屏幕上绘制的内容

tj=$((j\*3+1))

ti=$((i+1))

echo -ne "\033[${ti};${tj}H"

直接让光标出现在需要修改的位置，然后echo，这样每次移动就只需要修改几个字符而已，相比原来的每次2700个字符，优化了非常多。

这样输出的话，屏幕的闪烁已经肉眼不可见了

1. 到目前为止，这个俄罗斯方块里的方块还只能自然下落，下一步就是读取键盘输入，让方块可以根据玩家的要求进行移动

for ((i = 0; i < 20; i++))

do

read -t 0.02 -n 1 -s key for ((i = 0; i < 20; i++))

do

read -t 0.02 -n 1 -s key

case "$key" in

"A")

Rotate;;

"B")

Move 1;;

"C")

Move 3;;

"D")

Move 2;;

esac

done

case "$key" in

"A")

Rotate;;

"B")

Move 1;;

"C")

Move 3;;

"D")

Move 2;;

esac

done

就像这样，使用read -t 0.02 -n 1-s

表示等待0.02秒，读入一个字符，取消回显

本来想着有可能要开多线程，但是直接这样单线程循环，测试了一下，肉眼看不出差别

于是就懒得开多线程了

1. 消行处理

增加一个Clean 函数，判断每一行是否被填满，如果被填满那就擦掉那一行，然后map里的行依次向下平移

由于只有在方块落地时有可能触发消行，直接把Clean函数加在落地函数里就行了

#判断一行是否填满

function CheckRow() {

row=$1

local i=0

for((i = colbegin; i < colend; ++i)) do

Decode $row $i

if((map[ret]<=0))

then

return 0

fi

done

return 1

}

#删掉一行并平移上面的

function Wash() {

local i=$1

local j=0

for((;i>rowbegin;--i)) do

for((j = colbegin; j < colend; ++j)) do

Decode $i $j

local temp1=$ret

Decode $((i-1)) $j

local temp2=$ret

map[$temp1]=${map[$temp2]}

#((map[temp1]=map[temp2]))

done

done

for((j = colbegin; j < colend; ++j)) do

Decode $i $j

local temp1=$ret

map[$temp1]=0

done

Draw

}

#消行操作

function Clean() {

local i=0

for(( i = rowbegin; i < rowend; ++i)) do

# echo "i=$i"

CheckRow $i

if(($? == 1))

then

# echo "fuck i = $i"

Wash $i

((--i))

fi

done

}

function FallToGround() {

Add nowx nowy

Clean

GetNextBlock

CreateNewBlock $nextBLock

}

1. 瞬间下落

实机玩了几局Tetris之后，发现有一个重要的操作我没写出来，那就是按空格键让方块瞬间下落，要增加一个这个操作

在代码上魔改了一波发现，我在bash里输入空格键，它居然什么也不给我读入。。

所以只能换成按Z键瞬间下落了

if [[ $key == "Z" || $key == "z" ]]

then

#echo "fuck"

AllDown

fi

AllDown函数就是Down函数套一个while循环直到碰撞为止

1. 细节优化

到目前位置，俄罗斯方块的主体已经完成了，然而我玩了一下现代俄罗斯方块(如puyo puyo Tetris)之后，发现写出了的程序不支持T-spin,不支持左旋，不支持Hold导致手感不佳。

（T-spin指通过旋转弹墙使T方块落到正常操作进不去的地方）

现在就得要对Rotate函数和FallToGround函数进行修改，并添加Hold相关函数使得其支持T-spin等操作

先添加左旋功能:

首先我惊喜地发现shell读取不了ctrl键和shift键。。。

这样的话我就得再次魔改键位，把z设置为Hold,x设置为瞬间下落，c设置为左旋

再次惊喜地发现由于我开始写的时候没玩过现代的俄罗斯方块，所以我的上键的默认旋转就是左旋。。。那么我把↑和Z互换一下，再单独写个右旋就好了。。。

右旋很简单，就是把左旋的代码其中的

now[$temp1]=${pre[$temp2]}

的temp1和temp2交换一下就行了，逆时针旋转，交换一下很自然地就变成了顺时针旋转

然后添加Hold功能：

function Hold() {

if ((change>0))

then

return

fi

change=1

#如果手里现在没有方块，就直接把now的方块丢到inHand里，然后now读取下一个方块

local len=${#inHand[\*]}

#echo "x=${x}, y=${y}"

#echo "length=$len"

if ((len<9))

then

# echo "fuck"

inHand=0

inHand=(${now[\*]})

AddInHand $inhandx $inhandy

Del $nowx $nowy

GetNextBlock

CreateNewBlock $nextBLock

return

fi

#如果手里有方块，那就交换inHand的方块和now的方块

#echo "有方块"

DelInHand $inhandx $inhandy

#echo "${inHand[\*]}">>fuck

local tempblock=(${inHand[\*]})

inHand=(${now[\*]})

AddInHand $inhandx $inhandy

Del $nowx $nowy

now=(${tempblock[\*]})

nowx=3

nowy=15

prex=3

prey=15

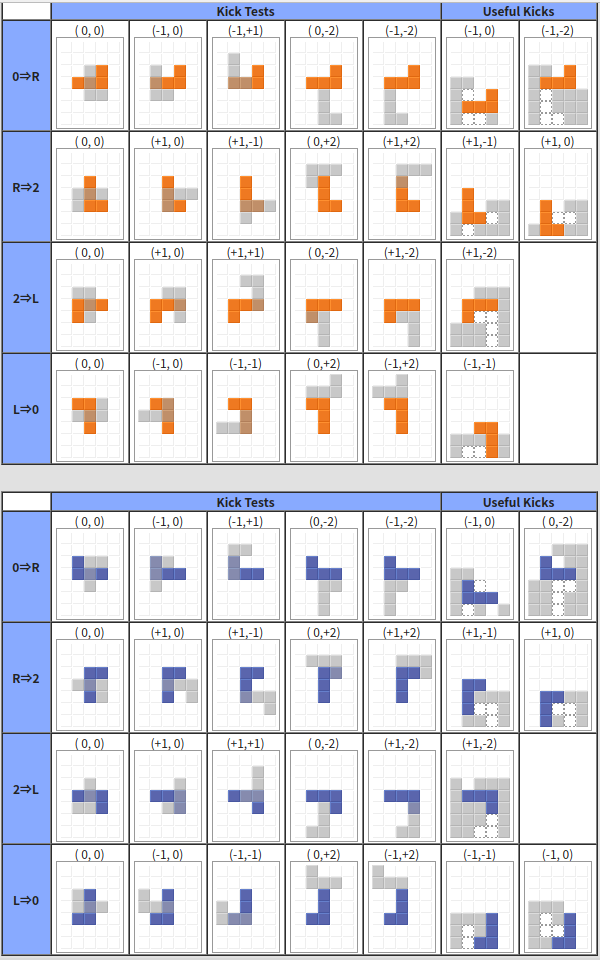
tprex=3

tprey=15

Add $nowx $nowy

}

最后修改旋转函数，使得其能支持T-spin:





在Tetris wiki上找到了SRS的旋转规则，大致就是根据当前方块的朝向，来依次对几个邻近坐标进行检查，看能否插进去（于是我现在得要维护方块的朝向，然后把这个检查表敲进去）

写了一段时间后，这个表太多了吧，用shell写太难受了，这个feature我不要了（guna!x

所以到最后为止，它还是不支持T-spin...

1. Next方块预览

现代俄罗斯方块以每7个方块一组，每一组称为1个bag，1个bag中的方块总是包含所有的7种方块，只是排列顺序随机而已。

所以我们需要用两个个队列来维护next方块

第一个队列是显式队列，长度为6，表示下六个方块分别是什么

第二个队列是隐式队列，长度为0~7，表示下下个bag里的方块种类

读取新方块时,第一个队列队首元素出队成为now,第二个队列队首元素出队然后入到第一个队列的队尾。

当第二个队列空掉的时候，新获得一个bag的方块种类，塞到第二个队列里

（shell里显然没有自带的队列容器，所以我们对于第二个个队列需要记录头指针和尾指针的位置来模拟出队和入队操作，为啥第一个队列不用？因为它的长度不变）

关于生成bag，如果用现代一点的语言的话，可以用random\_shuffle类似的函数实现乱序排列，然而shell连二维数组都不支持，怎么可能会有这种函数

所以我们得要自己实现乱序排列函数（这玩意叫洗牌算法）

功能很强大，但是原理很简单，

将最后一个数和前面任意 n个数（包括自己）中的一个数进行交换，然后倒数第二个数和前面任意 n-1 个数（包括自己）中的一个数进行交换，以此类推

这样就能使每种排列出现的概率相等，证明就略过了

#生成第二个队列

function GenerateSecondQueue(){

head=0

#先把应该有的元素塞进去

secondQueue=(1 2 3 4 5 6 7)

#执行洗牌算法

local i=6

for(( i = 6; i >= 0; --i )){

local target=$(($RANDOM %7))

local temp=${secondQueue[$target]}

secondQueue[$target]=${secondQueue[$i]}

secondQueue[$i]=$temp

}

}

洗牌算法如上

然后队列间的元素流动

function UpdateFirstQueue(){

local i=0

for(( i = 0; i < 5; ++i)) {

firstQueue[$i]=${firstQueue[$((i+1))]}

}

firstQueue[5]=${secondQueue[$head]}

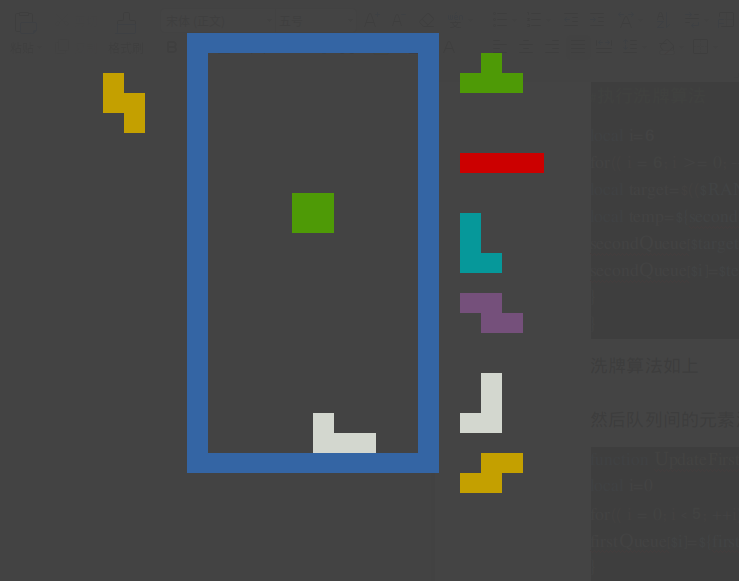
((head++))

if((head==7))

then

GenerateSecondQueue

fi



写到这个程度，感觉已经有模有样了呢

1. 加上BGM

打块时没有BGM总觉得少了点什么，那我们就来给它加上吧233

首先

sudo apt-get install sox

sudo apt-get install sox libsox-fmt-all

安装可以在命令行中播放音乐的软件

然后在程序中开多一个线程专门用来播放音乐

好像，直接加一行

play -q Game1.mp3 &

就完事了？？

想了想好像不太行，如果音乐放完了就很尴尬，所以得要让BGM的播放成为一个死循环，然后主线程退出时通知子线程退出

{ while ((1==1))

do

play -q Game1.mp3

done } &

bgmPid=$!

trap "kill $bgmPid;exit" SIGINT SIGTERM SIGQUIT SIGHUP SIGSTOP SIGTSTP

这样就好，多了几行而已

最后的&表示告诉shell我要并行执行这一句话

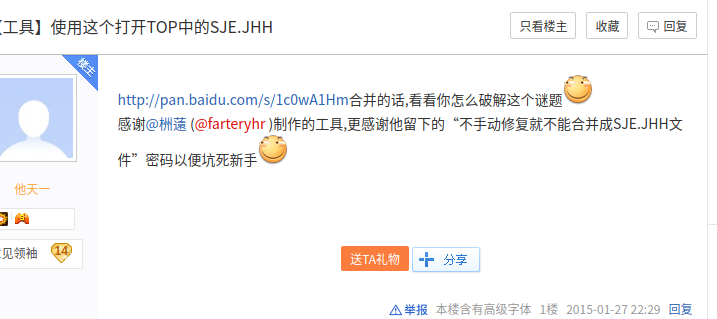
(多线程真的是，写挂一次炸一次系统。。。)

（既然这么容易，那我岂不是可以加上音效(兴奋x)）

1. 加上音效

搜索俄罗斯方块音效，每个都不满意，就想到把已有的游戏解包，直接用解包出来的音效

找了半天，终于找到俄罗斯方块的解包工具了，解包之后得到音效



想法是在移动，移动失败，旋转，旋转失败，hold，hold失败，消除n行，连续消除n行时都有对应的音效

移动，移动失败，旋转，旋转失败，hold，hold失败这些音效都是直接在相应的函数里面加几行就行，

然后消除n行和连续消除则需要在消行的时候进行判断，得额外维护上一次落地有没有消除，上一次消除是消除的几行, 如下

function Clean() {

local i=0

#记录该次消行行数

local tot=0

for(( i = rowbegin; i < rowend; ++i)) do

# echo "i=$i"

CheckRow $i

if(($? == 1))

then

# echo "fuck i = $i"

((++tot))

Wash $i

((--i))

fi

done

#如果有消行的话

if((tot>0))

then

#如果不是连续消行

if((suc==0))

then

case "$tot" in

"1")

play -q skin/sfx/default/sfx\_single.wav &

play -q skin/voice/default/01\_single.wav &;;

"2")

play -q skin/sfx/default/sfx\_double.wav &

play -q skin/voice/default/02\_double.wav &;;

"3")

play -q skin/sfx/default/sfx\_triple.wav &

play -q skin/voice/default/03\_triple.wav &;;

"4")

tet=4

play -q skin/sfx/default/sfx\_tetris.wav &

play -q skin/voice/default/04\_tetris.wav &;;

esac

case "$tot" in

"1")

play -q skin/sfx/default/sfx\_single.wav &

play -q skin/voice/default/01\_single.wav &;;

"2")

play -q skin/sfx/default/sfx\_double.wav &

play -q skin/voice/default/02\_double.wav &;;

"3")

play -q skin/sfx/default/sfx\_triple.wav &

play -q skin/voice/default/03\_triple.wav &;;

"4")

if((tet==4))

then

play -q skin/sfx/default/sfx\_b2b\_tetris.wavwav &

play -q skin/voice/default/05\_b2btetris.wav &

fi

tet=4

play -q skin/sfx/default/sfx\_tetris.wav &

play -q skin/voice/default/04\_tetris.wav &;;

esac

else

case "$suc" in

"1")

play -q skin/voice/default/15\_combo01.wav &;;

"2")

play -q skin/voice/default/15\_combo02.wav &;;

"3")

play -q skin/voice/default/15\_combo03.wav &;;

"4")

play -q skin/voice/default/15\_combo04.wav &;;

"5")

play -q skin/voice/default/15\_combo05.wav &;;

esac

if((suc>5))

then

play -q skin/voice/default/15\_combo05.wav &

fi

if((suc>=1))

then

play -q skin/sfx/default/sfx\_combo${suc}.wav &

fi

if((tot==4))

then

if((tet==4))

then

play -q skin/sfx/default/sfx\_b2b\_tetris.wavwav &

play -q skin/voice/default/05\_b2btetris.wav &

fi

tet=4

play -q skin/sfx/default/sfx\_tetris.wav &

play -q skin/voice/default/04\_tetris.wav &

fi

fi

((suc++))

return

fi

suc=0

}

1. 分数计算

消一行- 0

消两行-1

消三行-2

消四行-4

B2B Tetris-5

Combo

0→+ 0

1→+ 0

2→+1（总计：1）

3→+1（总数：2）

4→+ 2（总计：4）

5→+ 2（总数：6）

6→+ 3（总数：9）

7→+ 3（总数：12）

8→+ 4（总数：16）

9→+ 4（总数：20）

10→+ 4（总数：24）

11→+ 5（总数：29）

12→+ 5（总数：34）

因为在之前处理音效的时候就做过是哪一种消除的逻辑判断了，所以只需要简单增加几行,再单独写一个分数绘制函数即可

#分数显示函数

function scoreUpdate() {

echo -ne "\033[23;16H"

echo -ne " "

echo -ne "\033[24;16H"

echo -ne " "

echo -ne "\033[23;16H"

echo "score:$score"

echo -ne "\033[24;16H"

echo "combo:$suc"

echo $score > score

}

1. 落点预测

玩了几局体验了一下，发现有点费眼睛。。。因为shell控制台画不出格子，也没有落点预测。

所以就想加一个落点预测

落点预测就是维护一个新的虚拟方块，让它在map上的值为-8，然后其余部分都和实体方块差不多。

相当于每一次做一个虚拟方块的alldown把它放在最下面。



效果很不错