

L0 直接运行在硬件上的小游戏(amgame)

游戏截图



游戏内容

游戏会随机*打乱GRID×GRID（在include/game.h中定义）个方块。其中，颜色为渐变色*。
玩家通过上下左右键移动，按下空格选中方块。选中两个方块后会自动交换。玩家要使界面内的颜色重新恢复原本的渐变色。

注1：随机指的是游戏文件中调用了srand(uptime())，基于rand函数的特性及uptime自身的不确定性，可以近似认为游戏达到了随机

注2：渐变色指，在初始的图中，同一行/列的方格，RGB值为三个等差数列

文件结构

```
src
├── draw.c
├── font.c
├── game.c
├── logic.c
└── shape.c
```

我的源代码分为5个文件。

game.c为游戏主循环，只包含主要的函数调用。

shape.c提供绘制基本图形的函数，如画叉、圆、箭头、打钩、光标及填充指定方格等。

font.c提供在图形界面上打印字符的函数，包含ASCII编码下所有字符的点阵，一个打印单个字符的函数（都是静态），打印字符串的函数（非静态）

draw.c提供较复杂的绘图接口，包括初始化屏幕和在光标选中格打印光标或提示（由print_flag决定）

logic.c提供复杂逻辑的判定，包括游戏初始化、交换两个方格、计算颜色渐变值、获得下一个输入按键、对输入按键进行处理。

设计亮点

- 考虑到如果在主循环中进行判定while(uptime()< next_frame)，会导致按键的延迟，因此，我没有在主循环中使用时间相关的判定。只在少数需要体现出时间特性的部分单独加入了判断（如光标闪烁频率、长按按键的反馈）。这样可以做到，玩家按下按键后，第一时间*处理该按键信息，且长按时表现出若干毫秒处理一次的特性。
- 考虑到native和qemu的分辨率不同（同时还要考虑兼容其它分辨率），include/game.h中SIDE被定义成16会导致在不同分辨率下的效果差很多。所以我经过计算和尝试，将其改成w/40。使代码能够适应大多数分辨率。同理，draw_str中提供的size参数，我也使用了SIDE以实现自动适应屏幕的效果。但由于整型数精度问题，文字的适应效果有限。
- 考虑到游戏难度较大（作者自己都玩不过），我增加了Hint功能，按下h即可显示提示
- 为了节约空间，没有使用canvas数组，因此在打印时，为了表现出多层的效果，在进行判定时代码量稍多一点。
- 由于裸机本身不提供透明通道，alpha参数被我用来记录其它数值（该方格是否为固定方格）

L1 kalloc

实现原理简述

先将从pm_start到pm_end的堆区分为若干个8 KB大小的页面。

其中，使用区间树进行管理，管理方式如下，

每次必定分配一个叶节点的页面或者某个节点的**所有**子节点对应的页面

也即，不会出现跨页分配的情况，如

2048和2049都是1024的子节点，2050和2051都是1025的子节点，则当需要16 KB（两页）时，要么将2048和2049分配出去，要么将2050和2051分配出去（视页面具体使用情况），而不会出现2049和2050

被分配出去的情况

区间树声明为

```
uint16_t pages[1<<12]
```

其中，`pages[idx]&(1<<shift)`表示idx或者其子节点能否在上述规则下，给出 $8\text{KB} * (1<<\text{shift})$ 的空间

每次使用或归还一页时，则通过enable和disable进行处理

通过简单计算即可知上述两个函数的复杂度都是 $O(\lg n)$ ，n为页面数量

kalloc需要4 KB及以上的空间时，则调用big_size_alloc函数，此时需要上锁，然后从公共的区间树中取下适当数量的page，用header的size记录大小，并返回

kalloc需要4 KB以下的空间时，则在每个处理器自己的free list上寻找一个可用的空间

具体规则为

先将需要的空间大小向上取到16的倍数（减少日后合并的碎片数量）

从free list的头开始往后搜寻，找到第一个空间充裕的块（first bit）。如果该块除了提供所需空间之外，还足够提供头部的空间，则在尾部产生一个header，header的size记录本次分配的内存大小，便于kfree释放。如果该块不足以为新的header提供空间，则在维护好链表的指向关系后，直接返回当前header的space

header声明如下

```
struct header{
    struct header *next;
    uintptr_t size;
    struct{}space;//space doesn't take any storage
    //directly return &space
}static free_list[4]={};
```

其中，next指向下一个free list的节点，size用于标记目前区域的大小。

在申请空间时，先将需要的空间向上变为16的倍数，如果该header对应的区域大小大于所需空间，则从header的尾部划出所需空间加sizeof(header)大小的空间，并将尾部转化为一个header，将size置为分配的实际大小以便free函数解析，然后返回尾部的space的地址

设计亮点：

在header的定义中，包含一个大小为0的space成员，通过取该成员的地址，可以直接得到分配的空间的起始地址（由于标准中说struct的成员顺序一定按照声明顺序，故在内存上排列方式一定如此），且不会浪费空间

每个处理器的free list的第一项是一个Sentinel，用于简化判断逻辑，由于每次free时会考虑该部分是否能和前后的free space互相merge，所以，每个处理器提供一个sentinel，由于分配的时间不同，Sentinel和free list里的free space必然不相连，Sentinel一定不会和要free的space互相merge，极大的简化了判断逻辑

测试代码:

(免责声明:本人与其余同学互相交流/分享了测试函数,因而可能在查重时会出现重合,可以通过git记录的时间证明本人并非抄袭)

```
void test(){
    void *space[100];
    int i;
    for(i=0;i<100;++i){
        space[i]=pmm->alloc(rand()%((1<<10)-1));
    }
    for(i=0;i<1000;++i){
        int temp=rand()%10;
        pmm->free(space[temp]);
        space[temp]=pmm->alloc(rand()%((1<<10)-1));
    }
    for(i=0;i<100;++i){
        pmm->free(space[i]);
    }
}
```

该段代码中有一个大小为100的指针数组space,初始化后,space的每个元素指向一个kalloc返回的空间。

然后,每次随机选一个元素,将其空间free,并重新申请一段新的大小随机的空间。

经过若干次free、alloc循环后,再将space里的每个元素free,然后打印free list,如果实现无误,free list应该只包含若干个很大的块,不应该存在碎片或者free list中互相重叠