1. 页面切换:

思路:

为了避免界面互相调用(死递归)导致的栈内存浪费, 可以使用控制器+flag变量来控制跳转

flag变量: 标识接下来要跳转的界面
controller函数: 判断flag的值 来调用对应界面
View函数: 要跳转其它界面时 设置flag的值 并结束当前界面

补充: easyx不需要每次切换界面都initgraph, 在程序最开始时创建一个窗口即可 后面操作都可以直接在这个窗口上进行

参考代码:

```
#pragma warning(disable: 4996)
#include <stdio.h>
#include <stdlib.h>

#define MY_MENU 0
#define MY_SETTING 1
#define MY_GAME 2
#define MY_LOSE 3
#define MY_WIN 4
#define MY_OVER -1

int flag = MY_MENU;
void menuView();
void settingView();
void gameView();
void loseView();
void winView();
void controller();

int main() {
    controller();
    return 0;
}

void controller() {
    while (1) {
        switch (flag) {
            case MY_MENU:
                menuView();
                break;
            case MY_SETTING:
                settingView();
                break;
            case MY_GAME:
                gameView();
                break;
            case MY_LOSE:
```

```
                loseView();
                break;
            case MY_WIN:
                winView();
                break;
            case MY_OVER:
                exit(0);
        }
    }
}

void menuView() {
    printf("菜单界面: 输入1进入游戏, 输入2进入设置, 输入3结束程序");
    int choose;
    scanf("%d", &choose);
    switch (choose) {
    case 1:
        flag = MY_GAME;
        break;
    case 2:
        flag = MY_SETTING;
        break;
    case 3:
        flag = MY_OVER;
        break;
    default:
        flag = MY_MENU;
    }
    return;
}

void settingView() {

    printf("设置界面: 输入任意数字回到菜单");
    int choose;
    scanf("%d", &choose);
    flag = MY_MENU;
    return;
}

void gameView() {
    printf("游戏界面: 输入1回到菜单, 输入2游戏胜利, 输入3游戏失败");
    int choose;
    scanf("%d", &choose);
    switch (choose) {
    case 1:
        flag = MY_MENU;
        break;
    case 2:
        flag = MY_WIN;
        break;
    case 3:
        flag = MY_LOSE;
        break;
    default:
```

```
        flag = MY_MENU;
    }
    return;
}

void loseView() {
    printf("游戏失败: 输入1回到菜单, 输入2重新开始游戏");
    int choose;
    scanf("%d", &choose);
    switch (choose) {
    case 1:
        flag = MY_MENU;
        break;
    case 2:
        flag = MY_GAME;
        break;
    default:
        flag = MY_MENU;
    }
    return;
}

void winView() {
    printf("游戏胜利: 输入1回到菜单, 输入2重新开始游戏");
    int choose;
    scanf("%d", &choose);
    switch (choose) {
    case 1:
        flag = MY_MENU;
        break;
    case 2:
        flag = MY_GAME;
        break;
    default:
        flag = MY_MENU;
    }
    return;
}
```

2. 结构划分

思路:

将游戏的界面打印，接收输入，逻辑判断及修改划分开 细致一点的可以将判断和修改划分

思路参考:

```
游戏界面(){
    初始化游戏数据
    while(1){
        展示游戏界面();
```

```
        if(用户是否有输入){
            switch(输入值){
                case 操作1: 操作1对应修改();
                case 操作2: 操作2对应修改();
            }
        }

        if(游戏有需要自动执行的代码){ //比如贪吃蛇的移动, 俄罗斯方块的下落
            执行结果 = 自动执行代码1();
            执行结果 = 自动执行代码2();
        }

        switch(执行结果){
            case 结果1: 结果对应操作();
            case 结果2: 结果对应操作();
        }
    }
}
```

贪吃蛇/马里奥思路举例:

```
贪吃蛇游戏界面(){
    初始化地图, 蛇数据();
    while(1){
        打印地图, 打印蛇();
        if(用户是否有输入){
            input = 接收输入;
            switch(input){
                case 方向修改: 修改方向函数(input);
                case 暂停游戏: 暂停界面();
            }
        }

        //每次循环都要执行的代码
        执行结果 = 移动一格的整体逻辑();

        switch(执行结果){
            case 撞墙: 游戏结束();
            case 吃到食物: 播放吃到食物音效();
            case 正常移动: 无事发生;
        }

    }

}


马里奥游戏界面(){
    初始化地图,玩家数据();
    while(1){
        打印地图, 玩家();
        if(用户是否有输入){
            input = 接收输入;
            switch(input){
                case 左右移动: 修改玩家移动的加速度(input);
```

```
                case 跳跃: 修改玩家跳跃的加速度以及玩家状态();
                case 暂停游戏: 暂停界面();
            }
        }

        //每次循环都要执行的代码
        怪物移动的数据修改();
        玩家移动的数据修改(); //跳跃状态, 左右移动等等
        执行结果 = 碰撞判断及数据修改(); //玩家碰到怪物, 跳跃碰到墙壁等等

        switch(执行结果){
            case 撞怪物: 游戏结束();
            case 撞墙: 播放撞墙音效();
            case 无事发生: 无事发生;
        }

    }

}
```

3. 界面闪屏的解决方案

界面闪屏是因为不必要的屏幕刷新导致的
如以下代码

```
#include<stdio.h>
#include<conio.h>
#include<windows.h>
void gotoXY(int x, int y)
{
    COORD c;
    c.X = x - 1;
    c.Y = y - 1;
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), c);
}

void clear(int x, int y, int w, int h)
{
    for (int i = 0; i < h; i++) {
        gotoXY(x, y + i);
        for (int j = 0; j < w; j++) putchar(' ');
    }
}
int main()
{
    int x = 0, y = 0;
    while (1) {
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (i == y && j == x)printf(".");
                else printf("o");
```

```
            }
            printf("\n");
        }
        if (_kbhit()) {
            switch (_getch()) {
            case 'w': y--; break;
            case 's': y++; break;
            case 'a': x--; break;
            case 'd': x++; break;
            }

        }
        clear(1, 1, 10, 10);
        gotoXY(1, 1);
    }
    return 0;
}
```

比较常见的解决方案有
覆盖打印
仅清空指定块
不操作时暂停
使用缓冲区

1. 覆盖打印:

下次打印前 不清空界面 而是直接在原有界面上直接覆盖打印
比如 去掉样例中clear(1, 1, 10, 10);这行代码

2. 仅清空指定块:

如贪吃蛇的移动 推箱子的移动
通常只需要打印有修改的部分界面 而不是所有界面重新打印

样例修改:

```
int main()
{
    int x = 0, y = 0;

    int oldX = x, oldY = y;
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            printf("o");
        }
        printf("\n");
    }
    while (1) {
        if (oldX != x || oldY != y) {
            gotoXY(oldX + 1, oldY + 1);
            printf("o");
            gotoXY(x+1, y+1);
            printf(".");
        }
```

```
        oldX = x, oldY = y;
        if (_kbhit()) {
            switch (_getch()) {
            case 'w': y--; break;
            case 's': y++; break;
            case 'a': x--; break;
            case 'd': x++; break;
            }

        }
    }
    return 0;
}
```

3. 不操作时暂停:

仅限于类似推箱子, 2048这种不操作时程序不动的情况
在用户操作后才重新打印界面 而不是一直死循环打印

样例修改:

```
int main()
{
    int x = 0, y = 0;
    while (1) {

        if (_kbhit()) {
            switch (_getch()) {
            case 'w': y--; break;
            case 's': y++; break;
            case 'a': x--; break;
            case 'd': x++; break;
            }

            clear(1, 1, 10, 10);
            gotoXY(1, 1);
            for (int i = 0; i < 9; i++) {
                for (int j = 0; j < 9; j++) {
                    if (i == y && j == x)printf(".");
                    else printf("o");
                }
                printf("\n");
            }

        }
        else {
            Sleep(100);
        }
    }
    return 0;
}
```

4. 使用缓冲区:

easyx中 将需要绘制的内容放到缓冲区 在某个时间点一起绘制出来

参考代码(取消这三个注释 就不会出现闪烁了)

```
#include <graphics.h>

int main()
{
    initgraph(640,480);
    // BeginBatchDraw();

    setlinecolor(WHITE);
    setfillcolor(RED);

    for(int i=50; i<600; i++)
    {
        cleardevice();
        circle(i, 100, 40);
        floodfill(i, 100, WHITE);
        // FlushBatchDraw();
        Sleep(10);
    }

    // EndBatchDraw();
    closegraph();
}
```

4. 帧数/速度控制

思路:

通过clock函数获取当前时间 然后减去上一次的时间 如果超过了某个阈值 则进行对应操作(比如贪吃蛇移动, 俄罗斯方块下落等等)

样例代码:

```
#include<stdio.h>
#include<conio.h>
#include<time.h>
#include<windows.h>

int FPS(int f) {
    static long long oldTime = 0;
    long long nowTime = clock();
    if (nowTime - oldTime >= f) {
        oldTime = nowTime;
        return 1;
    }
    return 0;

}
```

```c
int main()
{
    while (1) {
        // 每秒大约3帧
        if (FPS(333)) {
            printf("1");
        }
        else {
            Sleep(10);
        }
    }
}
```