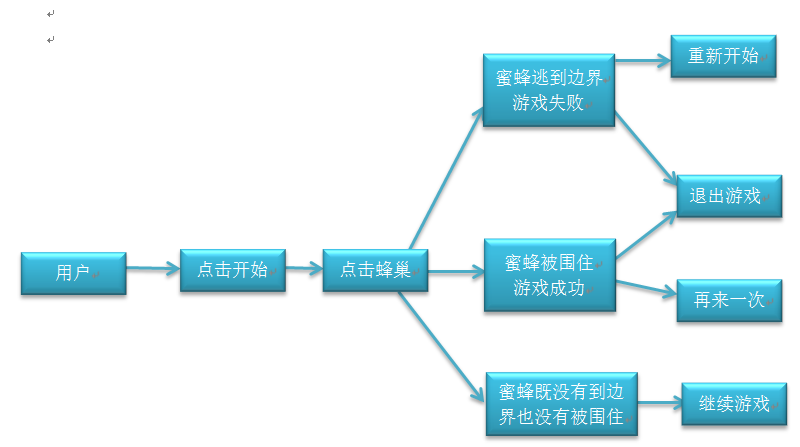
**蜜蜂，不要逃的开发文档**

1. **整体介绍**

蜜蜂不要逃是一款益智类小游戏，通过不断点击蜜蜂周围的蜂巢来围住蜜蜂以防止蜜蜂逃到边界。该游戏玩起来十分简单，很适合用户打发空闲时间，又不会浪费太多的时间玩这款游戏。

1. **游戏流程介绍**



1. **游戏主要实现介绍**

**3.1蜂巢以及蜜蜂设计**

主界面主要是由9\*9的蜂巢以及一只蜜蜂组成。为了使蜂巢能够完全的围住蜜蜂，蜂巢必须行行交错：

for(int i=0;i<9;i++){

NSMutableArray \*array=[NSMutableArray arrayWithCapacity:9];

[self.allButtonArray addObject:array];

for(int j=0;j<9;j++){

UIButton \*btn=[[UIButton alloc]init];

if(i%2==0){

btn.frame=CGRectMake(28\*j+20, 28\*i+170, 28, 28);

}

else{

btn.frame=CGRectMake(28\*j+34, 28\*i+170, 28, 28);

}

游戏开始的时候蜜蜂应该在蜂巢的正中央，也就是第5行第5列的那个点。但是由于蜜蜂高度大约两行的高度，因为蜜蜂初始放置的位置是第4行第5列的那个点：

self.beefImageView=[[UIImageView alloc]initWithFrame:CGRectMake(28\*4+20, 28\*3+170, 30, 56)];

为了增加游戏的可玩性，游戏开始的时候一部分蜂巢会变成墙方便围住蜜蜂。本游戏设计游戏开始时会随机产生10-49个墙并且位置随机：

-(void) createNestWall{

int wallNumber=arc4random()%40+10;

int num=0;

while (num<wallNumber) {

int row=arc4random()%9;

int col=arc4random()%9;

if (row!=4&&col!=4&&map[row][col]==0) {

num++;

map[row][col]=1;

UIImage \*image=[UIImage imageNamed:@"yellow.png"];

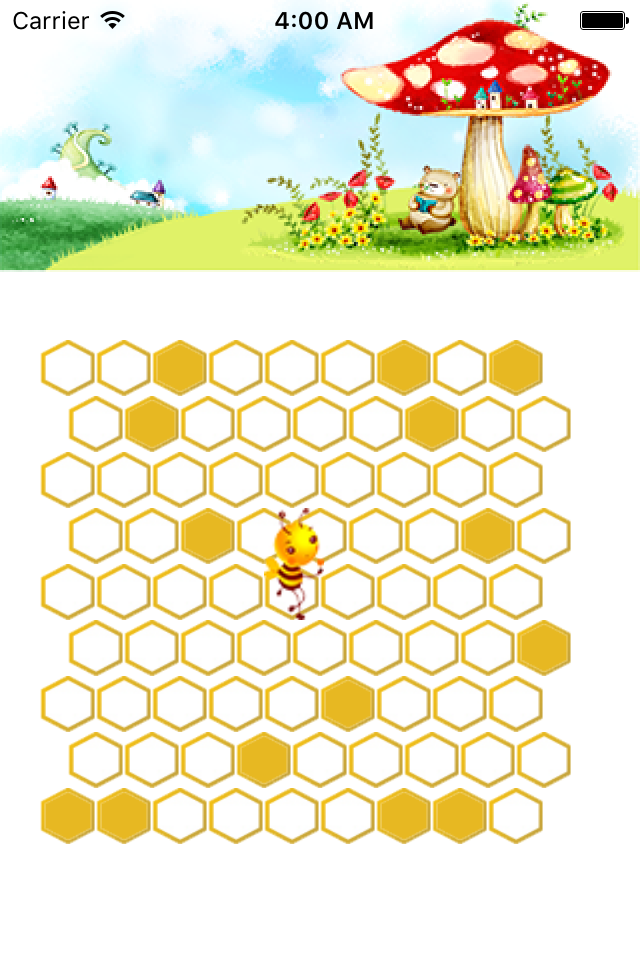
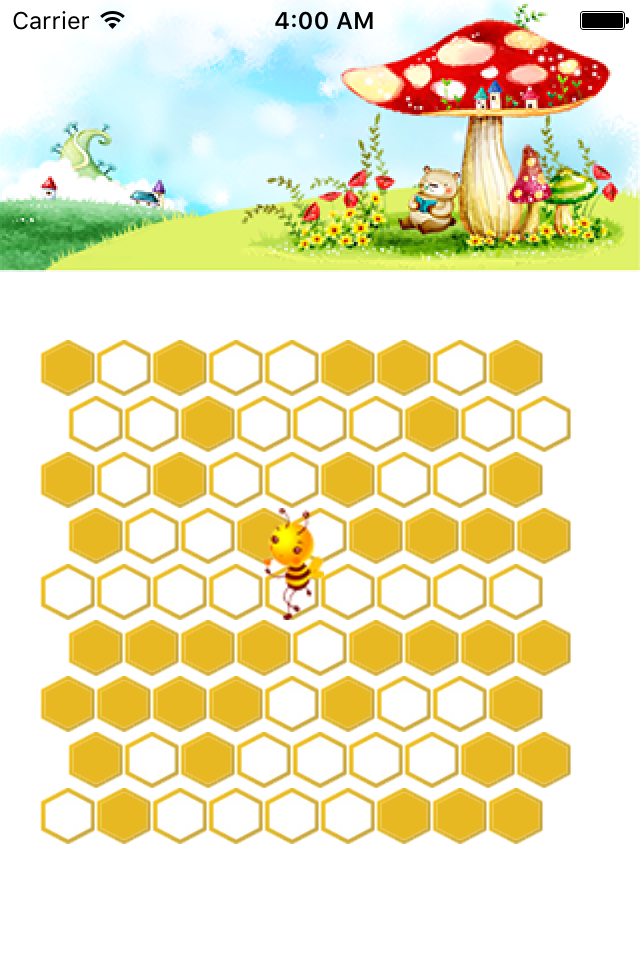
[self.allButtonArray[row][col] setImage:image forState:UIControlStateNormal];

}

}

}

效果图如下：



**3.2蜜蜂前行的算法设计**

当用户点击某个蜂巢时，蜜蜂便会前进一步。但是蜜蜂前行并不是随机的，为了逃避用户的堵截，蜜蜂会选择周围六个点中最优的那个点前行。使用两个变量分别存储每个点距离边界的最短距离path以及邻接非墙点的个数cost。当蜜蜂并没有陷入由墙堵成的圆圈中时，蜜蜂会选择周围六个点中距离边界最短且没有变成墙的蜂巢；但是当蜜蜂陷入由墙堵成的圆圈中时，蜜蜂周围的六个点的path都会变成100，即没有到达边界的通路，这个时候蜜蜂便会选择邻接非墙点数最大的非墙临界点。

* + 1. **存储邻接点**

为了找出邻接点中最优点，首先要实现一个数组将邻接的六个点存储起来。

-(NSMutableArray\*) getAllConnectNest{

NSMutableArray\* array=[[NSMutableArray alloc]initWithCapacity:6];

[array insertObject:[self getLeft] atIndex:0];

[array insertObject:[self getRight] atIndex:1];

[array insertObject:[self getLeftUp] atIndex:2];

[array insertObject:[self getRightUp] atIndex:3];

[array insertObject:[self getLeftDown] atIndex:4];

[array insertObject:[self getRightDown] atIndex:5];

return array;

}

左边的邻接点：

非第一列的点全都有左边的邻接点，只要当前点的列数减一便是左边的邻接点的坐标。

if(self.col>0){

newp=allNest[self.row][self.col-1];

}

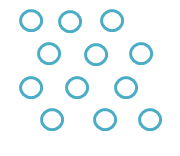
右边的邻接点：

非第九列的点全都有右边的邻接点，只要当前点的列数加一便是右边的邻接点的坐标。

if(self.col<8){

newp=allNest[self.row][self.col+1];

}



左上的邻接点：

如上图所示，除了第一行的奇数行中，第一列的点都没有左上的邻接点，其他点的左上的邻接点的列数是当前点的列数减一，行数是当前点的行数减一；偶数行中，所有的点都有左上的邻接点，当前点的行数减一便是左上邻接点的坐标。

if (self.row>0) {

if(self.row%2==0){

if(self.col==0){

newp=NULL;

}

else{

newp=allNest[self.row-1][self.col-1];

}

}

else{

newp=allNest[self.row-1][self.col];

}

}

右上的邻接点：

如上图所示，除了第一行的奇数行中，所有的点都有右上的邻接点，当前点的行数减一便是右上邻接点的坐标；偶数行中，第九列的点都没有右上的邻接点，其他点的右上的邻接点的列数是当前点的列数加一，行数是当前点的行数减一。

if(self.row>0){

if (self.row%2==0) {

newp=allNest[self.row-1][self.col];

}

else{

if(self.col==8){

newp=NULL;

}

else{

newp=allNest[self.row-1][self.col+1];

}

}

}

左下的邻接点：

如上图所示，除了第九行的奇数行中，第一列的点都没有左下的邻接点，其余点的左下的邻接点的列数是当前点的列数减一，行数是当前点的行数加一；偶数行中，所有的点都有左下的邻接点，当前点的行数加一便是左下邻接点的坐标。

if (self.row<8) {

if(self.row%2==0){

if(self.col==0){

newp=NULL;

}

else{

newp=allNest[self.row+1][self.col-1];

}

}

else{

newp=allNest[self.row+1][self.col];

}

}

右下的邻接点：

如上图所示，除了第九行的奇数行中，所有的点都有右下的邻接点，当前点的行数加一便是右下邻接点的坐标；偶数行中，第九列的点都没有右下的邻接点，其他点的右下的邻接点的列数是当前点的列数加一，行数是当前点的行数加一。

if(self.row<8){

if(self.row%2==0){

newp=allNest[self.row+1][self.col];

}

else{

if(self.col==8){

newp=NULL;

}

else{

newp=allNest[self.row+1][self.col+1];

}

}

}

* + 1. **path和cost的计算**

初始时，所有点的cost和path都是-100，每次蜜蜂移动都要计算一次所有点的cost值和path值。

如果当前点是墙点，该点的cost是100；如果当前点是边界点，蜜蜂到达该点游戏变回结束，因此该点的cost是0；否则，遍历当前点的邻接点数组，cost等于该数组中非墙点的个数：

for (NestCircle \*obj in allConnectNest) {

if (map[obj.row][obj.col]==0) {

number++;

}

}

如果当前点是墙点，该点的path是100；如果当前点是边界点，该点的path是0；否则，当前点的path是周围六个点中path最小值加1：

int min=100;

for(NestCircle \*obj in allConnectNest){

if (obj.path>-100) {

int tmp=obj.path;

if (obj.path<0) {

tmp=-tmp;

}

if (min>tmp) {

min=tmp;

}

}

}

if (min<100) {

self.path=min+1;

}

else{

self.path+=1;//当该点从未计算过时，path为-99;四次计算后，该点的path值最大为-96

}

**3.2.3所有点的cost和path计算**

初始时，所有点的cost和path都为-100。每次蜜蜂移动，都要先将所有点的path值和cost值初始化为-100，再重新计算一次path值和cost值。

所有点的cost值只与相邻点是否是非墙点有关系，因此计算十分简单：

for (int i=0; i<9; i++) {

for (int j=0; j<9; j++) {

[allNest[i][j] calculateCost];

}

}

但是path值和周围的path值有关，并且从每个方向层层计算path值所得到的最终结果都不一样，因此需要从左上角、右上角、左下角、右下角分别依次计算每个点的path值，从而得到每个点最小即正确的path值。

// 左上角

for (int i=0; i<9; i++) {

for (int j=0; j<9; j++) {

[allNest[i][j] calculatePath];

[allNest[j][i] calculatePath];

}

}

// 右上角

for (int i=0; i<9; i++) {

for (int j=0; j<9; j++) {

[allNest[i][8-j] calculatePath];

[allNest[j][8-i] calculatePath];

}

}

// 左下角

for (int i=0; i<9; i++) {

for (int j=0; j<9; j++) {

[allNest[8-i][j] calculatePath];

[allNest[8-j][i] calculatePath];

}

}

// 右下角

for (int i=0; i<9; i++) {

for (int j=0; j<9; j++) {

[allNest[8-i][8-j] calculatePath];

[allNest[8-j][8-i] calculatePath];

}

}

**3.2.4判断蜜蜂是否在由墙堵成的圈中**

获取蜜蜂当前点的邻接点数组，依次遍历所有点的path值，如果path大于等于100，说明该邻接点是墙点，若path大于-100且小于0，说明该点path值被计算过并且是墙内的点。如果所有的邻接点都为墙点或者墙内的点，那么蜜蜂是在由墙堵成的圈中。

**3.2.5蜜蜂前进路径的选择**

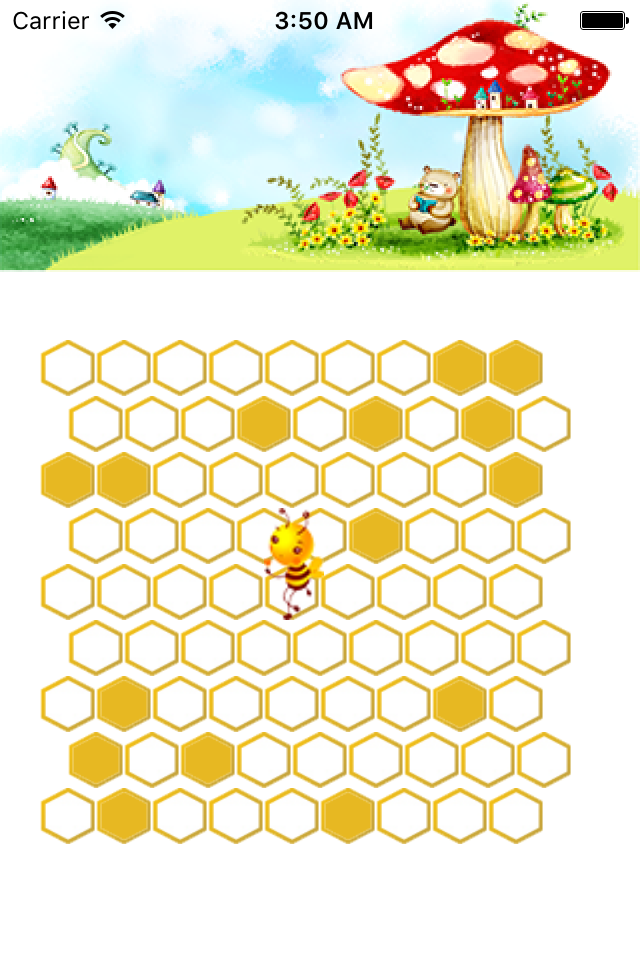
如果蜜蜂在由墙堵成的圈中，选择邻接点中cost值最小的点，蜜蜂移动到该点，如果没有该点，说明蜜蜂邻接点全是墙，则蜜蜂原地不动，用户再点击蜜蜂所在的点，游戏便会胜利；否则，计算相邻点中path值最小的点，蜜蜂移动到该点，游戏继续进行或者失败。

1. **游戏界面展示**

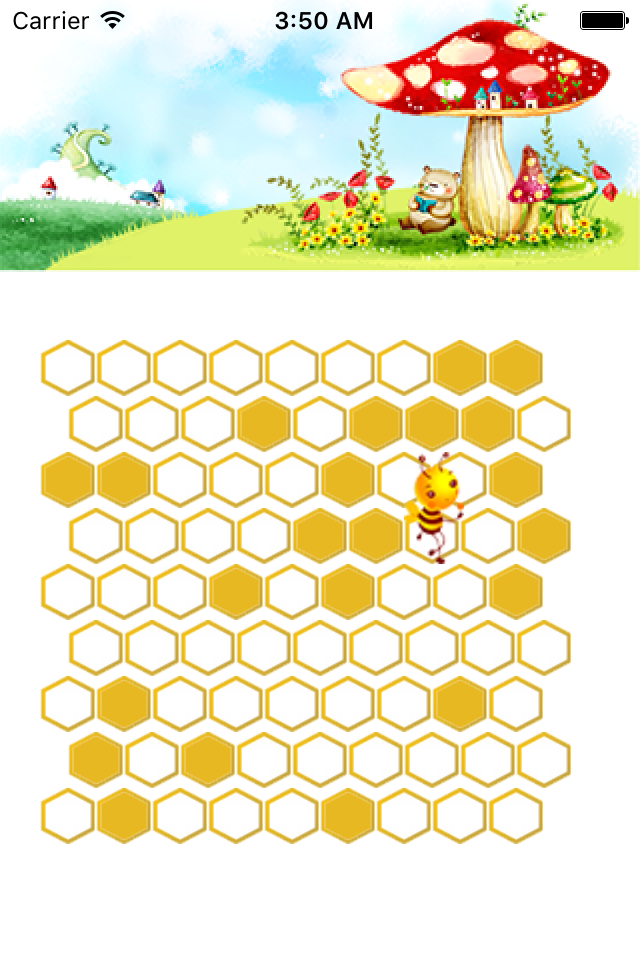
**4.1游戏首页**



**4.2游戏开始界面**



**4.3游戏进行中界面**



**4.4游戏失败或者胜利界面**

