实验课 4：人工生命游戏实验

4.1 实验目的

学习人工生命理论，基于智能Agent感知-推理-行为机制，实验经典人工生命游戏。本次实验为基础性实验，要求通过LSL脚本，操纵智能Agent，实现生命游戏实验。

4.2 人工生命理论

人工生命(AL:Artificial life)是通过人工模拟生命系统,来研究生命的领域。世界上首先提出“人工生命”概念的人，是美国洛斯•阿莫斯国家实验室的克里斯•兰顿博士。他在 1987 年时指出：生命的特征在于具有自我繁殖，进化等功能。地球上的生物只不过是生命的一种形式，只有用人工的方法，用计算机的方法或其他智能机械制造出具有生命特征的行为并加以研究，才能揭示生命全貌。

人工生命的研究现状与人工智能早期历史可以说是并行的。40 年代末，50 年代初，冯.诺伊曼提出了机器自增长的可能性理论。以计算机为工具，迎来了信息科学的发展。1956 年达特默斯的夏季讨论会上麦卡锡提出人工智能的术语。正式形成人工智能学科的研究。人工生命许多早期的工作也源出于人工智能。60 年代，罗森勃拉特(Rosenblatt)研究感知机，斯塔勒(Stahl)建立了一些细胞活动的模型，他把图灵机用作“算法酶”，将生化表示成字符串。60 年代后期，林登麦伊尔(Lindenmayer)提出了生长发展中的细胞交互作用的数学模型，现在称为 L-系统。这些相当简单的模型，可以明显地显示复杂的发展历史，支持细胞间的通信和差异。 70 年代以来，科拉德(Conrad)和他的同事研究人工仿生系统中的自适应、进化和群体动力学，提出了不断完善的“人工世界”模型。 后来侧重研究系统突发性的个体适应性。乔姆斯基(Chomsky)的形式语言理论应用在程序设计语言的规范说明和开发编译程序。细胞自动机应用于图象处理。科伟(Conway)提出生命的细胞自动机对策论。

人工生命是形成新的信息处理体系强大的推动力，并成为研究生物的一个特

别有用的工具。人工生命的研究可能将信息科学和生命科学结合起来，形成生命信息科学。在 21 世纪初人工生命的研究将会蓬勃发展，并取得突破性进展。

人工生命研究的科学问题如下：

- 生命自组织和自复制：研究天体生物学、宇宙生物学、自催化系统、分子自装配系统、分子信息处理等
- 发育和变异：研究多细胞发育、基因调节网络、自然和人工的形态形成理论。目前人们采用细胞自动机、L-系统等进行研究。细胞自动机是一种对结构递归应用简单规则组的例子。在细胞自动机中，被改变的结构是整个有限自动机格阵。典型的形态形成理论是 1968 年 Lindenmayer 提出的 L-系统。L-系统由一组符号串的重写规则组成，它与乔姆斯基(Chomsky)形式语法有密切关系。
- 系统复杂性：对生命从系统角度来看它的行为，首先在物理上可以定义为非线性、非平衡的开放系统。生命体是混沌和有序的复合。非线性是复杂性的根源，这不仅表现在事物形态结构的无规分布上，也表现在事物发展过程中的近乎随机变化上。然而，通过混沌理论，我们却可以洞察到这些复杂现象背后的简单性。非线性把表象的复杂性与本质的简单性联系起来。
- 进化和适应动力学：研究进化的模式和方式、人工仿生学、进化博弈、分子进化、免疫系统进化、学习等。在自然界，通过物种选择实现进化。遗传算法和进化计算是目前极为活跃的研究领域。
- 智能主体：智能主体是主体是具有自治性、智能性、反应性、预动性和社会性的计算实体。研究理性主体的形式化模型、通信方式、协作策略；研究涌现集体行为、通信和协作的群体智能进化、社会语言系统。
- 自主系统：研究具有自我管理能力的系统。自我管理具体体现在以下四个方面：自我配置：系统必须能够随着环境的改变自动地，动态地进行系统的配置；自我优化：系统不断的监视各个部分的运行状况，对性能进行优化；自我恢复：系统必须能够发现问题或潜在的问题，然后找到替代的方式或重新调整系统使系统正常运行；自我保护：系统必须能够察觉、识别和使自己免受各种各样的攻击，维护系统的安全性和完整性。

- 机器人和人工脑：研究生物感悟的机器人、自治和自适应机器人、进化机器人、人工脑。

4.3 经典生命游戏实验

4.3.1 生命游戏理论背景

生命游戏（Game of Life），又称生命棋，是英国数学家 John Horton Conway 在 1970 年发明的细胞自动机[17]。它最初于 1970 年 10 月在《Scientific American》中的数学游戏专栏出现。

生命游戏其实是一个自主进行的游戏，玩家之需要规定游戏规则和初始状态。它由一个二维矩阵世界组成，这个世界中，每个方格居住着一个活着的或死了的细胞（cell），细胞也是整个游戏中最小且唯一的单元。每个细胞有且只有两种状态，活着或者死亡，而每个细胞在下一时刻的状态都取决于当前与它相邻的八个细胞的状态。如果相邻方格中活着的细胞数量过多，这个细胞会因为资源匮乏而在下一个时刻死去；相反，如果周围活细胞过少，这个细胞会因太孤单而死去。在实际操作中，玩家可以设定游戏规则中周围存活细胞的数量来控制整个世界中细胞的存活。如果这个数目设定过高，世界中的大部分细胞会因为找不到太多的活的邻居而死去，直到整个世界都没有生命；如果这个数目设定过低，世界中又会被生命充满而没有什么变化。

在一般情况下，游戏的规则就是：

- 当一个活细胞的周围有少于 2 个活细胞时，它会在下一时刻死去；
- 当一个活细胞的周围有多余 3 个活细胞时，它会在下一时刻死去；
- 当一个活细胞的周围有且仅有 2 或 3 个活细胞时，它会在下一时刻存活；
- 当一个死细胞的周围又且仅有 3 个活细胞时，它会在下一时刻变为活细胞；

在游戏的进行中，细胞的初始状态往往是杂乱无序的，但是经过一代代的存活后，往往会产生出各种有序的结构。这些结构有些具有很好的稳定性，一旦形成变不再发生任何变化，有些具有对称性，并且以一定的周期重复出现。

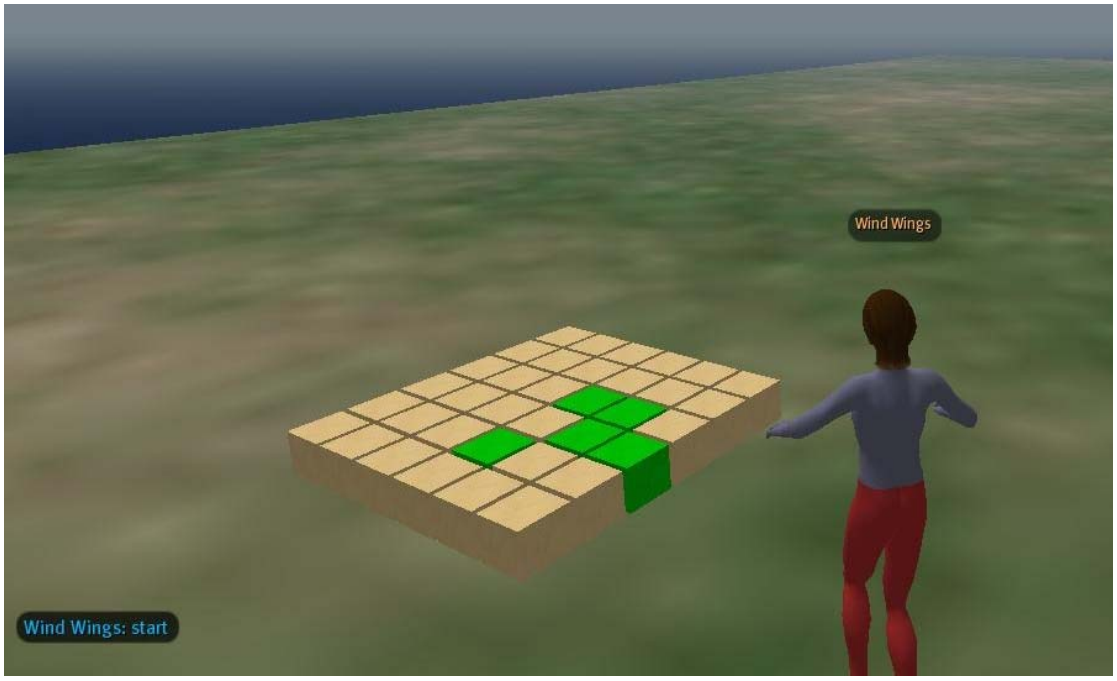
4.3.2 生命游戏实验步骤

1. 细胞建模

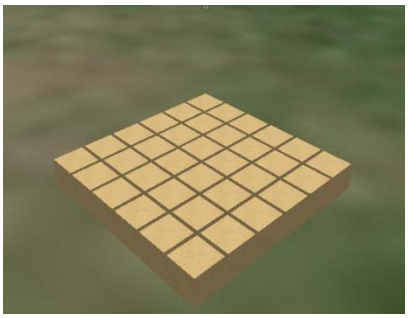
生命游戏是一种最简单的多 Agent 系统。每一个细胞就是一个独立的 Agent，它可以主动探知周围的局部环境，并根据环境做出相应的改变。同时，自身的状态和变化也在影响着周围的其它 Agent。对于整个 Agent 系统来说，每一细胞的存活变化是相对独立的，但是整个系统可以呈现出从无序到有序的变化。

2. 游戏场景设计

在虚拟世界仿真平台中，用户（avatar）可以自由的与虚拟世界中的物体进行操作和交互的特性。用户可以通过触摸（touch）来改变细胞的初始状态，还可以通过信息通道向虚拟世界中的所有细胞体发送命令，控制生命游戏的开始，进行和终止，如下图所示：



对细胞进行建模，所具有的属性如下表：

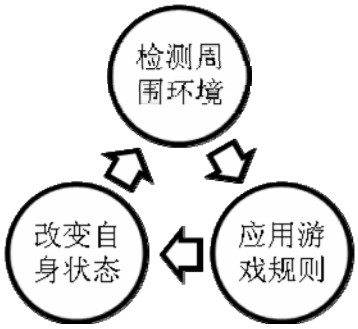


属性	变量
位置	vector position<x,y,z>
颜色（用来标记生或死）	vector color<R,G,B>
感知方向	float senseArc

感知半径	float senseRange

3. 细胞脚本状态设置

细胞个体的状态图如下：



生命游戏的简单状态图

当生命游戏开始后，每一细胞会首先判断自身的状态，然后进入对应的状态中。接下来探测周围八个细胞的状态，应用游戏规则对自身的下一个状态作出判断，并做出相应的变化。

算法的伪代码如下：

```
default
{
    self detect()
    if (live)
    {
        state LIVE;
    }
    else
    {
        state DEATH;
    }
}
```

```
state LIVE
{
    sensor()
    if (change)
    {
        change();
        state DEATH;
    }
}
```

```
state DEATH
{
    sensor()
    if (change)
    {
        change();
        state LIVE;
    }
}
```

4.3.3 生命游戏详细规则参考 pdf

Conway's Game of Life in Second Life - Technical Details

Joel Dearden

Centre for Advanced Spatial Analysis, University College London

Overview

This document is a collection of rough notes about how CASA's version of the Game of Life in Second Life works.

The notes were made as part of the development process and details were inevitably changed during implementation and bug fixing so there may be some differences between this document and the real code.

Agent Rules

From Wikipedia (http://en.wikipedia.org/wiki/Conway's_Game_of_Life):

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. *Any live cell with fewer than two live neighbours dies, as if by loneliness.*
2. *Any live cell with more than three live neighbours dies, as if by overcrowding.*
3. *Any live cell with two or three live neighbours lives, unchanged, to the next generation.*
4. *Any dead cell with exactly three live neighbours comes to life.*

The initial pattern constitutes the 'seed' of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed — births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations.

Block Creation and State Display

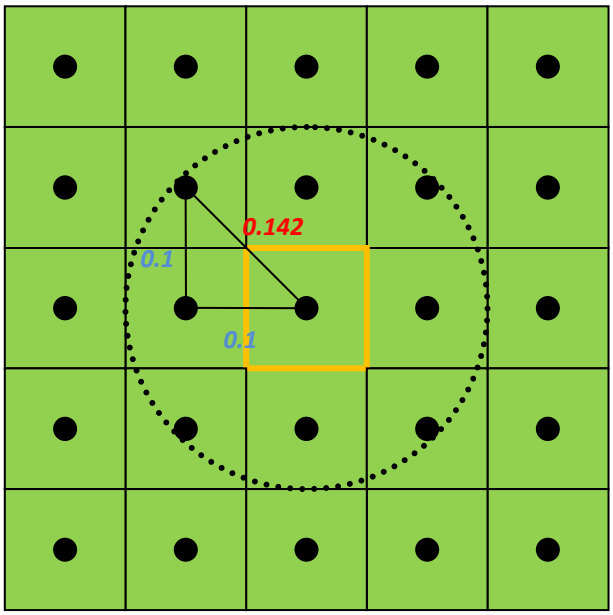
We manually create all the grid blocks initially and change their opacity (via script) to 1 for ALIVE and 0 for DEAD. This is faster than creating, deleting or moving them.

Calculation

Each agent is responsible for calculating its state. We don't use any centralised control except to synchronise the blocks.

Agent Vision

Each agent needs to be able to see the state of the eight agents adjacent to it. We implement this using the llSensor command in LSL. The diagram below illustrates the sensor range required.



Assuming each block is 0.1m square, the sensor range needs to be:

$0.1^2 + 0.1^2 = c^2$

c = 0.142 (rounded up)

Synchronising Block Agents

We use a controller to synchronise the blocks in the grid. Obviously they all need to think and act at the same rate otherwise the grid will not behave as intended.

The sequence for the game of life is:

1. *Block start think – Sync point A*
2. *Block stop think – Sync point B*
3. *Block act*
4. *Repeat*

Sync point A is required to prevent any blocks from thinking too soon, i.e. before other blocks have acted, and so sending the whole grid out of sync.

Sync point B is required to prevent any blocks from acting too soon, i.e. before others have finished thinking and again sending the whole grid out of sync.

How do we synchronise 100 block scripts quickly to perform an action at the same time?

All blocks register with the controller initially so it has a saved list of block keys.

All blocks also save the controller key.

For SyncA:

Blocks check llKey2Name(controllerKey) to see if it is called "think"

The controller only sets its name to "think" once all blocks have set their name to "<alive/dead state>:wt" which indicates that they are waiting to think.

When blocks see controller name as "think" they do their thinking and then when finished set their name to "<state>:wa" to indicate that they are waiting to act.

For SyncB:

Blocks check llKey2Name(controllerKey) to see if it is called "act"

The controller only sets its name to "act" once all blocks have set their name to "<state>:wa" which indicates that they are waiting to act.

When blocks see controller name as "act" they do their acting and then when finished set their name to "<state>:wt" to indicate that they are waiting to think.

Repeat...