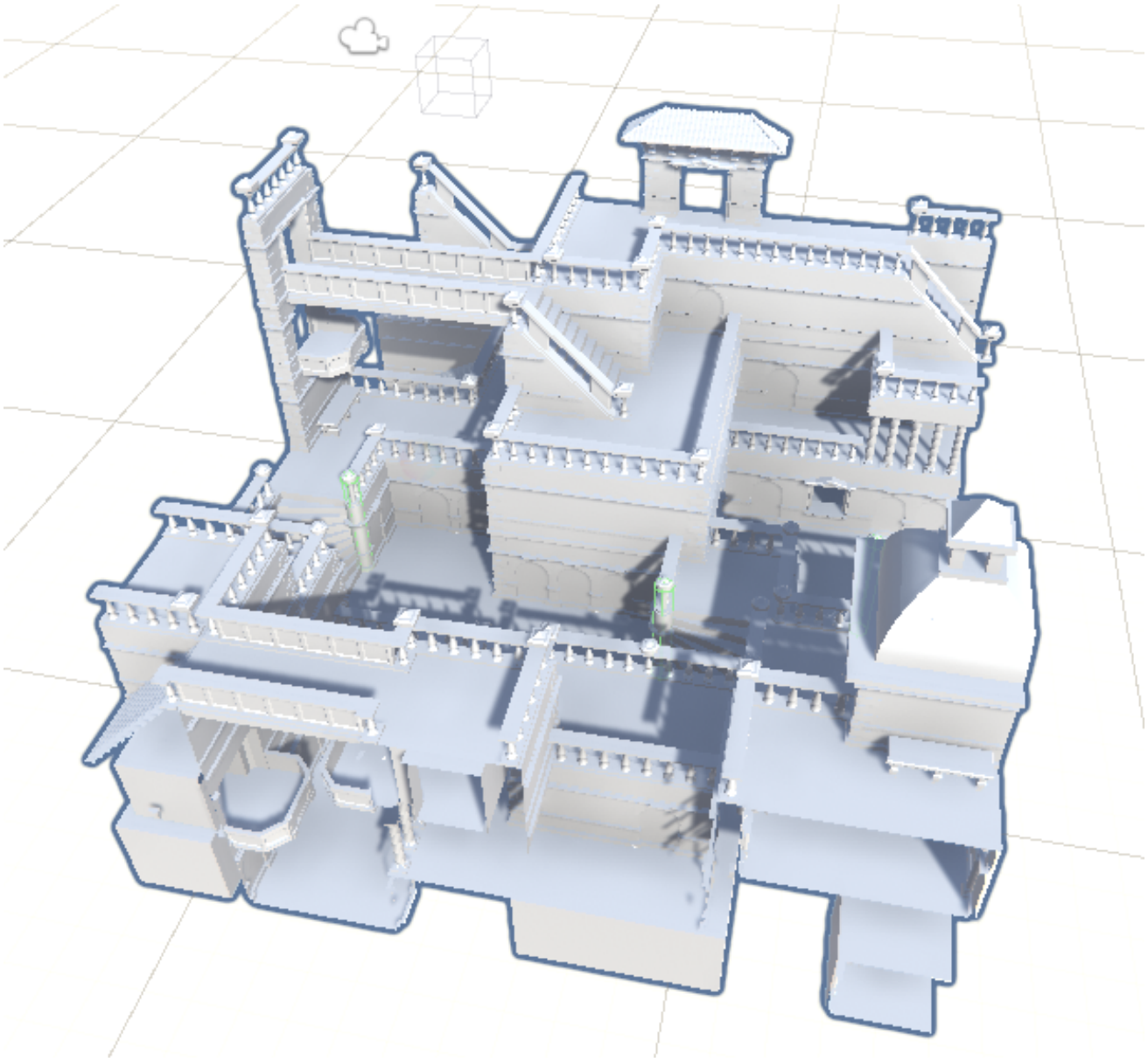
基于《波函数坍缩算法》的无限城市程序化随机生成



1.简介

之前在Github上看到《无限城市的随机程序化生成算法》,效果很惊艳,所以花了一些时间对这个项目源码进行了学习,在此备忘和分享学习过程中的一些心得体会.

在这个项目中,提到的算法叫做**《波函数坍缩》** .光是听到这个名字,就想:"这啥啊?波函数?坍缩?啥跟啥啊?",然后经过查阅,发现是跟量子力学有关的,"量子力学我没学过啊",觉得量子力学会不会太过难以理解,一度差点弃坑...在看到一些这个算法的应用示例以后,终于下定决心想把这个算法原理弄明白,结果发现并不是想象中的那么难懂.希望读者们也不要像我开始一样.因为"量子力学我不会啊","波函数坍缩是啥玩意啊"等等原因就放弃了,其实这个算法很简单的.

1.1 波函数坍缩是什么?

OK,那我们先来简单介绍一下什么是"波函数坍缩"(由于本人对量子力学没有经过专业的学习,所以仅仅是表达个人理解,不对的地方,请不要深究细节).

《双缝干涉》实验

在量子力学中,有一个很著名的实验,叫做《双缝干涉》.实验的过程可以看看这个视频 世界十大经典物理实验之首——电子双缝干涉实验.实验的结果就**当没有观测时,光子呈现的是波的性质.当观测时,光子呈现的是粒子的性质.(这是人类在科学实验中正式遭遇灵异的著名试验.)**

薛定谔的猫

著名的奥地利物理学家(Nue Mao Kuang Ren)薛定谔提出了一个思想实验,实验的内容可以看这里「薛定谔的猫」是指什么？ .实验的结论就是**当匣子里的猫被观测时,猫的状态要么死要么活,只能在两者之间二选一,而没有观测的话,那么猫的状态既不死也不活(这个状态不符合实际,但是量子力学确实是这么定义它的)** .

介绍完上面两个著名实验以后,肯定有人提出疑问了,那么它们跟"波函数塌陷"有什么关系呢?关系大着呢.上面两个实验都有表现了一个物质在量子力学中,可能存在多种状态(光子:波,粒子;猫:死,活,不死不活.)而当物质被观测时,它的状态就被确定了.这个"状态被确定了"的过程,我们就称之为"波函数坍缩".(所以专业术语就是如此,用一个看起来很高大上的名字来形容一个现象或者性质,就算"波函数坍缩"这五个字换成"XXXXX"也不影响理解)

1.2 熵是什么?

介绍完"波函数坍缩"以后,再来谈谈什么是"熵(shang)"吧,熵是热力学的表示物质状态的参量,它的物理意义表示物质的混乱程度,熵越大,说明物质越混乱,熵越小,说明物质越稳定,例如水加热变成雾,这个过程熵值增加(熵增),水降温变成冰,这个过程熵值减少(熵减),把这一概念引入到量子力学中,可以表示当一个物质从叠加态坍缩到某一个状态时,熵减,反之则熵增.

以薛定谔的猫的实验为例,可以这么理解,当没有观察猫的时候,猫处于不死不活叠加态,此时猫死的概率是50%,活的概率也是50%,熵值最大,当打开匣子去观察猫的时候,猫的状态就从不死不活坍缩成死或活的状态,坍缩完以后熵值最小.既然熵是个度量值,那么就有求熵公式,针对事件发生的概率而言,标准求熵公式(也叫信息熵)为:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log p(x)$$

其中p(x)表示事件x发生的概率.H(X)就是熵值.

到此为止.量子力学的部分简单介绍完毕,其实这些目前看不懂不明白也没关系,不会影响了解这个随机算法.只是这个算法借用了这些概念和术语而已.

2.原理概述

以日常生活中,**以去看电影为例(下面会经常引用到这个例子),假设一家电影院的出售的电影票是没有座位号的,入座由一位入座引导员安排,设总座位个数(即影院容纳人数)为n,从入座引导员的角度来说,如果客人没有要求,那么任何一个人都可以坐到他想坐的座位上,也就是说,每个人能坐在任一座位的概率都是1/n,而现在我们要加入一些规则了,比如你跟你的朋友一起去,你就需要跟引导员提要求,说你们必须挨着座位坐,那么引导员给你选好了一个座位以后,就会再把你的朋友安排在旁边.从座位的角度来说,你一旦入座,当前你的座位人选概率塌陷成只有你,而**

旁边的座位人选是你朋友的概率被大大提高.所以类似这样的从混乱变成确定的现象,用一个术语来表示就叫做波函数塌陷.

所以**算法的核心原理,就是动态使每个座位的候选对象的范围变得越来越小,直到最后所有的座位都能够选取到合适的对象.** (比如每个座位都入座了一个观众(对应坍缩的对象)).而如何动态使候选对象范围变小呢?就是通过**约束规则,传播和回溯.**

2.1 约束规则

在初始情况下(宇宙一片混沌?)(理论上此时熵值最大),每个座位的可选对象集合都一样,如果没有规则,那随机选取以后的结果,也将会杂乱无章,约束规则的存在,就类似上面提到的”跟引导员提要求,我的朋友要坐在我旁边”,各种各样的规则,使得当一个座位被确定(坍缩)了以后,会根据规则影响到其他的座位的可选对象集合.

2.2 传播

当一个座位的人选被确定了(坍缩)了以后,通过规则加上传播,就可以对其他的座位的可选对象集合进行处理.(比如,按上述场景,给引导员一个”我的右手边只能坐妹子”的规则,那么一旦我入座,我右手边的座位可选集合就被处理成只包含妹子了),

在2D世界中,一个物体有4个边(前后左右),而在3D世界中,一个物体有6个面(前后左右上下).每个面对应的一个邻居,所以波函数坍缩的算法传播思路,就可以根据已坍缩位置的邻居进行传播.把规则套进去,对每个邻居的可选集合进行处理.

2.3 回溯

有的时候,当我们的算法在传播过程中,出了错(比如:我给引导员提的规则是”我的右手只能坐我朋友”,而我朋友给引导员提的规则是”我不要坐第一排”,这样就有可能出现这样的一个情况,一旦引导员把我的座位安排在第一排,我一入座,算法传播以后,我右边的座位可选集合被处理成只能坐我朋友,又因为我朋友”不坐第一排”的规则,最终导致了我右边的座位没有一个人能坐,而我右边没人又跟”我的右手只能坐我朋友”这个规则相悖....这样容易造成冲突矛盾,资源浪费),我们就希望能够把出错的方案排除,然后再回退.重新选一遍.这就是回溯的意义.

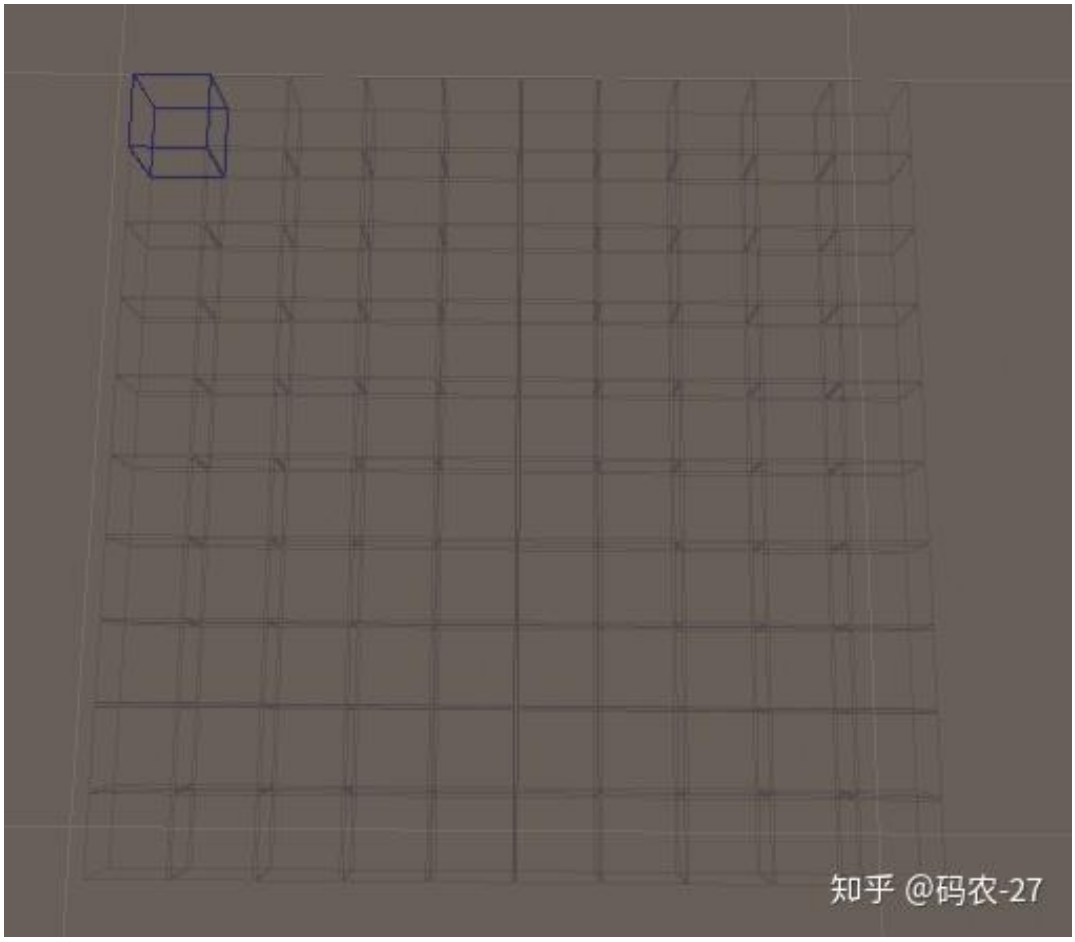
3.源码解析

了解上述基本概念和原理以后,结合源代码来看看是如何实现的吧.

3.1 Slot(插槽?座位?)

`Slot`这个类对象**对应了上述例子中的座位** .这个类有个很重要的成员对象,就是`Modules`,它代表了**当前Slot所有可选模块的集合** .

```
//表示一个可以放置模块的插槽
public class Slot
{
    public Vector3Int Position;      //位置
    public ModuleSet Modules;  //可选的模块集合
    public short[][] ModuleHealth;  //6个方向,可选模块对应的Health,Health为0时,表示不可选取
    private AbstractMap map;          //地图的引用
    public Module Module;         //最终选取的模块
    public bool Collapsed;     //是否已坍缩
    //其他....略
}
```



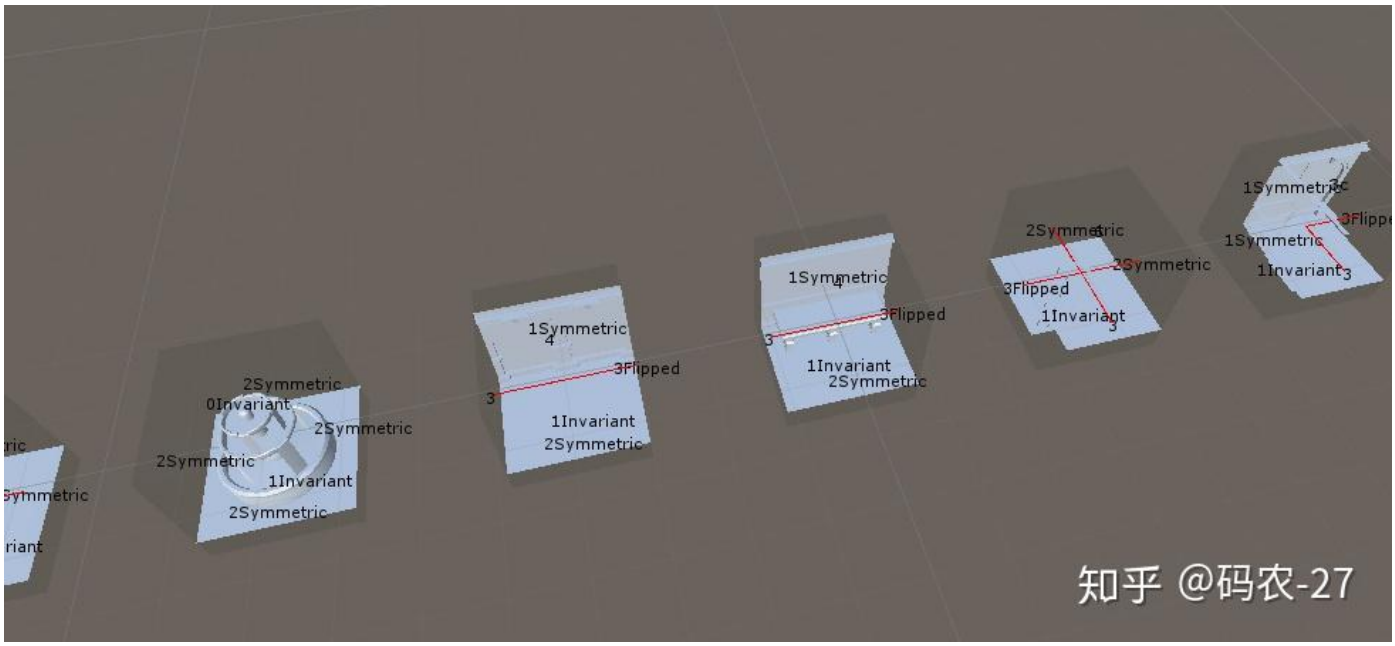
每个格子对应一个Slot

3.2 ModuleProtoType, Module, ModuleSet (模块相关)

`ModuleProtoType`即**模块原型 ,它包含了6个面的信息(前后左右上下),每个面都对应了一个`ConnectorID`,只有相同的`ConnectorID`才可以相互连接(这是最基础的约束规则)** ,面的信息分成HorizontalFaceDetails(水平面,前后左右)和VerticalFaceDetails(垂直面,上下),作者通过一个比较Mesh顶点的辅助代码,来自动批量的生成FaceDetails信息.

```
// 通过计算Mesh的六个面的数据,得到6个方向的ConnectorId,相同的ConnectorId才可以连接,通过Flipped / Rotation来决定要如何连接
public class ModulePrototype : MonoBehaviour
{
    [System.Serializable]
    public abstract class FaceDetails
    {
        public bool Walkable;    //此面是否允许行走
        public int Connector;    //ConnectorID,相同的ConnectorID才可以相连
        public Fingerprint Fingerprint;  //指纹,通过mesh顶点计算得到vector4数组,用以批量生成ConnectorID
        public ModulePrototype[] ExcludedNeighbours;        //想要排除在外的邻近模块原型
        //其他....略
```

```
    }
    public float Probability = 1.0f;     //被随机选取的概率,求熵时也需要用到
    public bool Spawn = true;            //是否可用
    public FaceDetails[] Faces;
        //6个朝向
}
```



一部分模块原型,数字代表ConnectorId

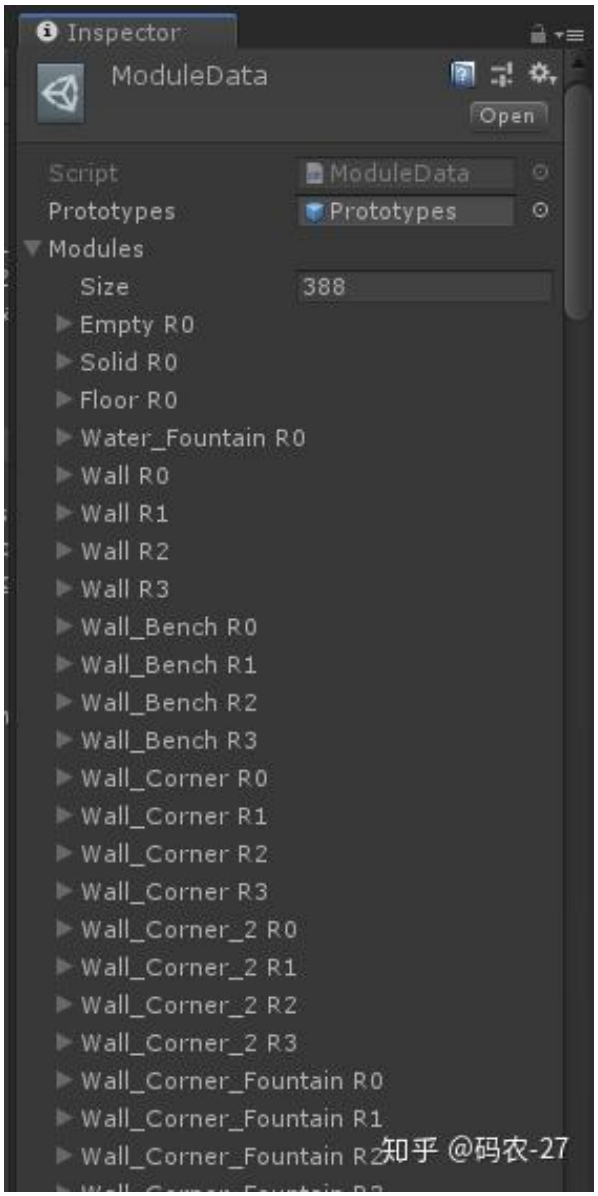
`Module`即**模块,对应了上述例子中的看电影的观众** ,在算法做规则判断时候用来的模块,它是由`ModuleProtoType`(模块原型)绕着Y轴旋转以后得到的.比如:一个`ModuleProtoType`,分别绕着Y轴旋转0,90,180,270度,就得到4个Module,这样可以使得模块原型的资源得到复用.

```
public class Module
```

```
{
      public string Name;         //名称
      public ModulePrototype Prototype;  //模块原
型
      public GameObject Prefab;       //GameObject
      public int Rotation;       //绕Y轴旋转度数,0-0
1-90 2-180 3-270
      public ModuleSet[] PossibleNeighbors;
            //6个方向邻居的可选模块集合
      public Module[][] PossibleNeighborsArray;
            //6个方向邻居的可选模块数组
      [HideInInspector]    public int Index;
//索引,与ModuleSet位数组相对应
      public float PLogP;    //概率p*log(p),求熵时
需要用到

      //构造方法
      public Module(GameObject prefab, int
rotation, int index)
      {
          this.Rotation = rotation;
          this.Index = index;
          this.Prefab = prefab;
          this.Prototype =
this.Prefab.GetComponent<ModulePrototype>();
          this.Name =
this.Prototype.gameObject.name + " R" + rotation;
```

```
            this.PLogP = this.Prototype.Probability
* Mathf.Log(this.Prototype.Probability);
        }
}
```



部分模块.R+数字代表旋转

`ModuleSet`即模块集合,它使用**位数组** 的方式(什么是位数组,请看这)),把预处理好的所有`Module`都封装到集合中,在`Slot`中的ModuleSet就表示了当前`Slot`所有可选的模块了.

```
public class ModuleSet : ICollection<Module>
{
    [SerializeField]   private long[] data;
                //位数组,1个位对应1模块index,0表示不可选,1表示可选
    private float entropy;                  //熵
    private bool entropyOutdated = true;     //熵是否过期(下次需要重新计算)
    public int Count;                 //可选模块的总数量
    //其他...略
}
```

3.3 AbstractMap(地图相关)

`AbstractMap`这个抽象基类,可以与**上述例子的入座引导员做类比**,**它负责生成和管理了所有地图上的`Slot`**,接下来就看看它是如何执行波函数坍缩的核心算法的.

先获取当前所有需要处理的Slot:

```
// 坍缩函数
public void Collapse(Vector3Int size,
IEnumerable<Vector3Int> targets, bool
showProgress = false)
{
    //其他...略
     this.workArea = new HashSet<Slot>
(targets.Select(target =>
this.GetSlot(target)).Where(slot => slot != null
&& !slot.Collapsed));  //把所有需要处理的Slot加到
workArea中,等待处理
    //其他...略
}
```

然后计算每个Slot中可选集合的熵,找到熵值最小的Slot

```
// 坍缩函数
public void Collapse(Vector3Int size,
IEnumerable<Vector3Int> targets, bool
showProgress = false)
{
    //其他...略
     this.workArea = new HashSet<Slot>
(targets.Select(target =>
this.GetSlot(target)).Where(slot => slot != null
&& !slot.Collapsed));
     while (this.workArea.Any())
     {
```

```
            float minEntropy =
float.PositiveInfinity;
            Slot selected = null;
            foreach (var slot in workArea)
//遍历Slot,找出熵值最小的那个
            {
                float entropy =
slot.Modules.Entropy;
                if (entropy < minEntropy)
                {
                    selected = slot;
                    minEntropy = entropy;
                }
            }
            try
            {
                 selected.CollapseRandom();    //对
Slot进行随机坍缩
            }
            catch (CollapseFailedException)      //
捕获出错的异常,为了回溯
            {
                  //回溯,暂略,下文介绍
            }
        //其他...略
}
```

对这个Slot进行坍缩,坍缩就是在`ModuleSet`(可选模块集合)中随机取一个模块(模块的概率越大,越容易被选中)

```
    public void CollapseRandom()
    {
        if (!this.Modules.Any())    //一个可选模块都没有
        {
            throw new CollapseFailedException(this);
        }
        if (this.Collapsed)
        {
            throw new Exception("Slot is already collapsed.");    //此Slot已坍缩
        }
        float max = this.Modules.Select(module => module.Prototype.Probability).Sum();      //所有可能性累加
        float roll = (float)(InfiniteMap.Random.NextDouble() * max);      //获取Roll值
        float p = 0;
        foreach (var candidate in this.Modules)

        {
            p += candidate.Prototype.Probability;
            if (p >= roll)
```

```
            {
                this.Collapse(candidate);
                  //在可选模块中随机选择一个
                return;
            }
        }
        this.Collapse(this.Modules.First());
    //坍缩完毕

     }
```

坍缩完毕以后,就需要对6个方向的邻居的ModuleSet按规则进行处理.即**传播** .

```
     // 坍缩完毕的函数
     public void Collapse(Module module)
     {
         if (this.Collapsed)
         {
             Debug.LogWarning("Trying to collapse
already collapsed slot.");
             return;
         }
         this.map.History.Push(new
HistoryItem(this));       //加入历史记录,便于回溯

         this.Module = module;
```

```
        ModuleSet toRemove = new
ModuleSet(this.Modules);    //复制一份当前可选模块的
集合
        toRemove.Remove(module);         //在集合中
删除当前模块

        this.RemoveModules(toRemove);  //将其他的
模块集合从当前集合中删除

        this.map.NotifySlotCollapsed(this);
    }

    //从当前集合中删除一个模块集合,并传播影响邻居
    public void RemoveModules(ModuleSet
modulesToRemove, bool recursive = true)
    {
        //其他...略
        for (int d = 0; d < 6; d++)     //6个方向
        {
            int inverseDirection = (d + 3) % 6;
  //反方向,也就是指向自己
            var neighborSlot =
this.GetNeighbor(d);
            if (neighborSlot == null ||
neighborSlot.Forgotten)
            {
                continue;
            }
```

```
            foreach (var module in
modulesToRemove)
            {

                for (int i = 0; i <
module.PossibleNeighborsArray[d].Length; i++)
                {
                    var possibleNeighbor =
module.PossibleNeighborsArray[d][i];
                    if
(neighborSlot.ModuleHealth[inverseDirection]
[possibleNeighbor.Index] == 1 &&
neighborSlot.Modules.Contains(possibleNeighbor))
//如果邻居的可选模块是属于待删模块集合中的话,如果Health减
完以后变成0,就从邻居的模块集合中删除掉
                    {

this.map.RemovalQueue[neighborSlot.Position].Add(
possibleNeighbor);     //添加进删除队列
                    }

#if UNITY_EDITOR

                    if
(neighborSlot.ModuleHealth[inverseDirection]
[possibleNeighbor.Index] < 1)
                    {
```

```
                        throw new
System.InvalidOperationException("ModuleHealth
must not be negative. " + this.Position + " d: "
+ d);
                    }
#endif

neighborSlot.ModuleHealth[inverseDirection]
[possibleNeighbor.Index]--;  //将Health-1
                }
            }
        }
         //其他...略
    }
```

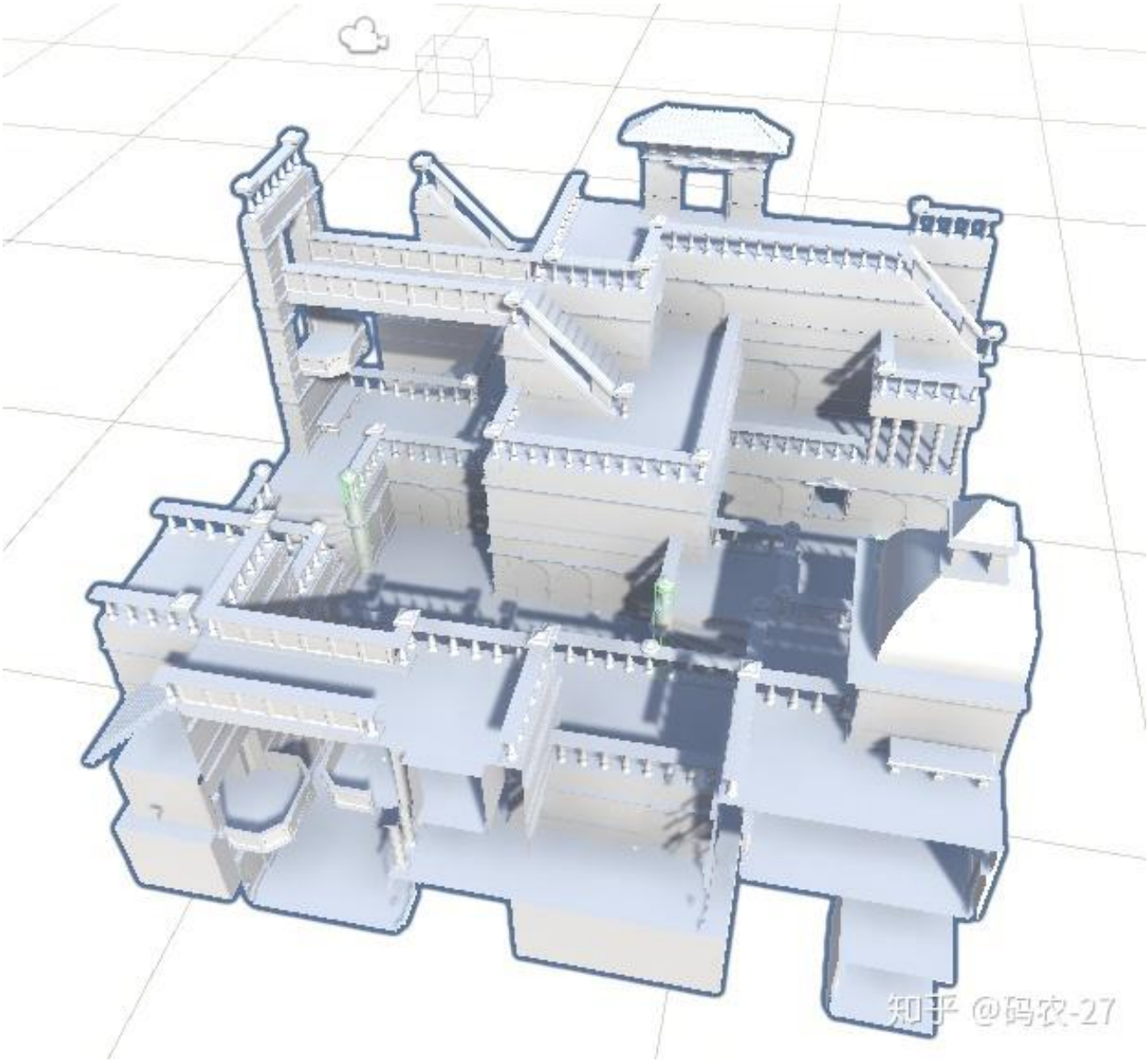
通过`try..catch..`来捕获出错的情况,然后回溯,回溯的方式就是将保存在**History** 中最后的两个元素取出,把删除掉的`ModuleSet`还回去.

```
//坍缩函数
public void Collapse(Vector3Int size,
IEnumerable<Vector3Int> targets, bool
showProgress = false)
{
      //其他...略
      try
      {
```

```
            selected.CollapseRandom();       //尝试随
机坍缩
        }
        catch (CollapseFailedException)         //捕
获异常
        {
            this.RemovalQueue.Clear();
            if (this.History.TotalCount >
this.backtrackBarrier)
            {
                this.backtrackBarrier =
this.History.TotalCount;
                this.backtrackAmount = 2;
            }
            else
            {
                  this.backtrackAmount += 4;
            }
            if (this.backtrackAmount > 0)
            {
                Debug.Log(this.History.Count + "
Backtracking " + this.backtrackAmount + "
steps...");
            }
            this.Undo(this.backtrackAmount);
//回溯backtrackAmount步
        }
        //其他...略
```

```
}
```

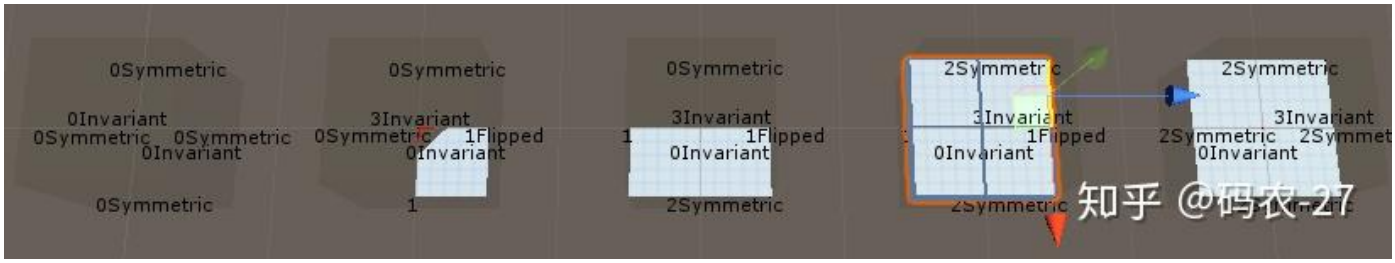
这一套流程下来,波函数坍缩的算法就完成了.增加各式各样的约束规则,可以将模块拼成想要的整体.



原工程生成出来的886的样子

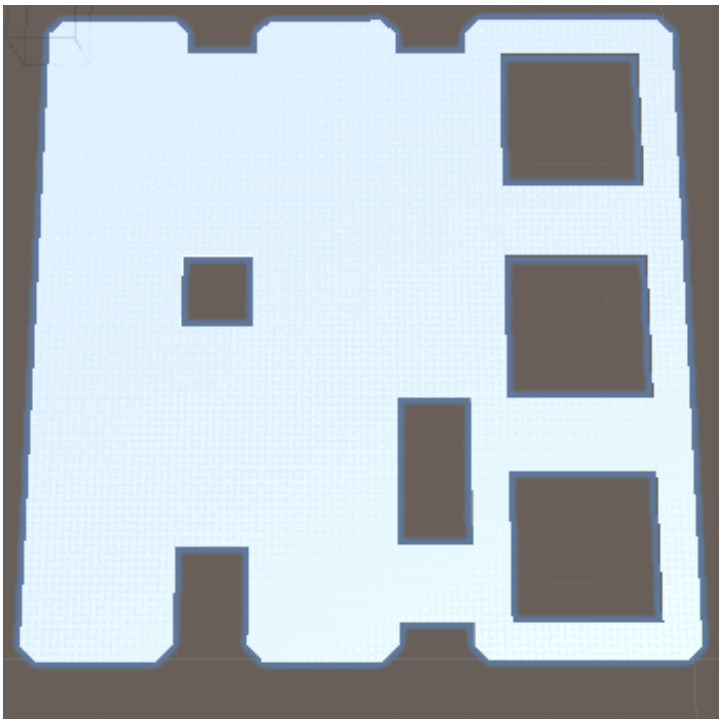
以下是我自己制作一些简单的模块原型,然后套用这个算法制作出来的效果:

仅有五个模块原型:



使用ProBuilder制作的测试用的5个模块原型

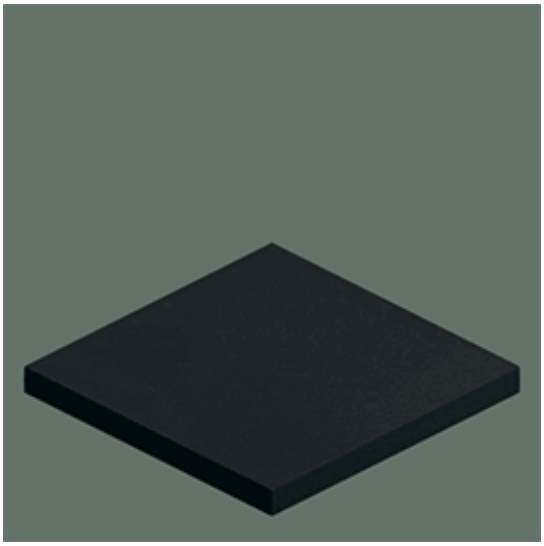
生成结果:



测试模块原型生成的例子

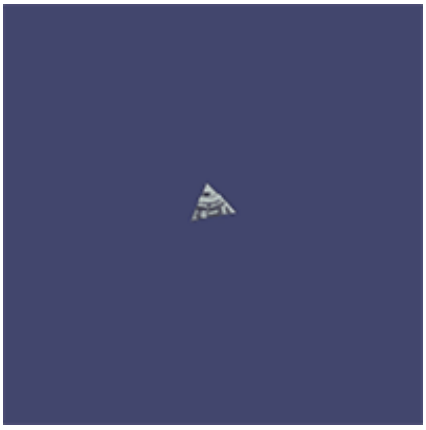
4.应用案例

以下是其他国外开发者,用《波函数坍缩》的算法,制作出来的案例:

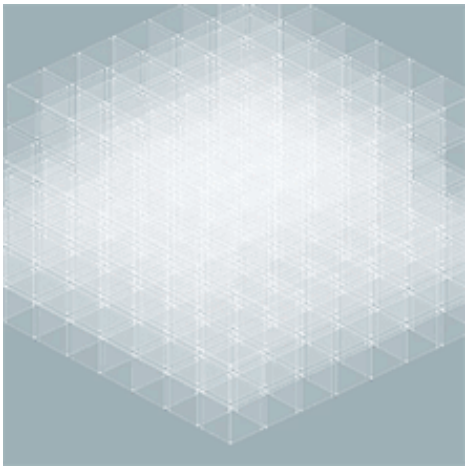


生成城堡

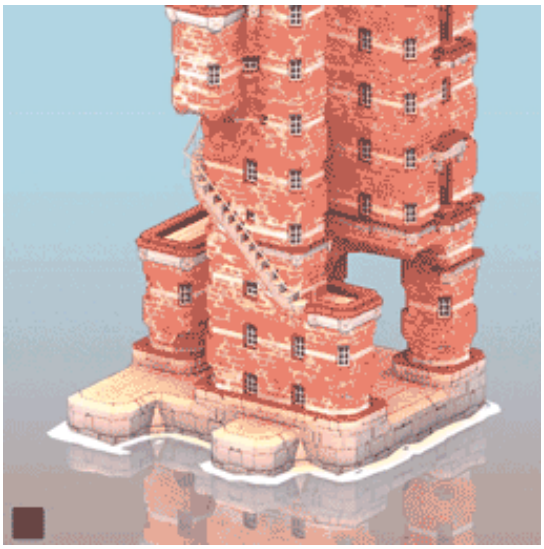
生成管道



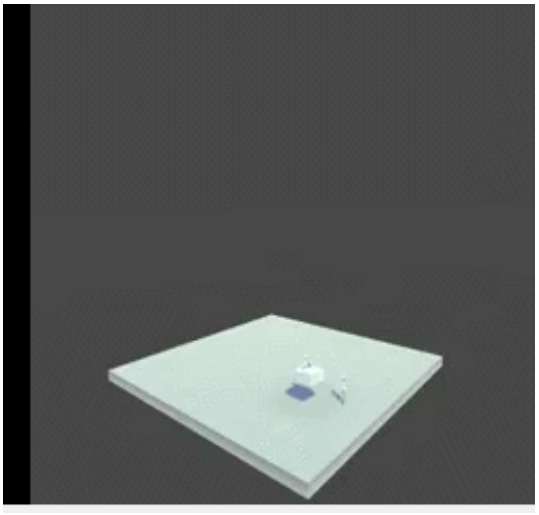
生成地球表面



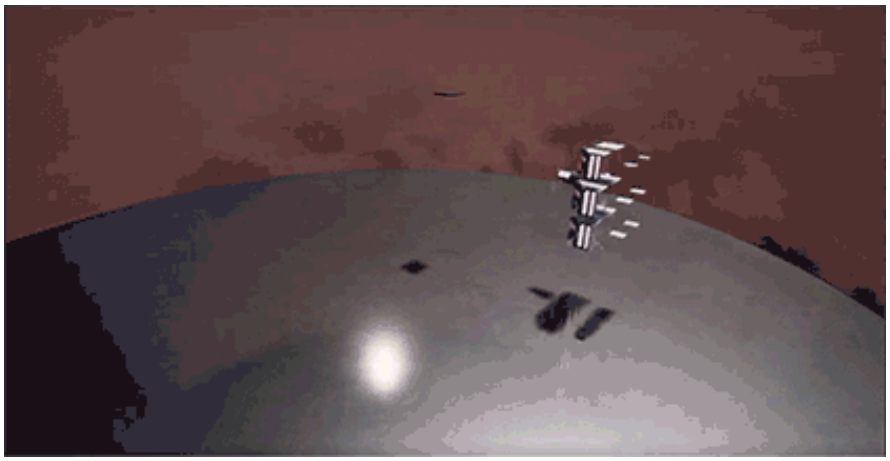
BadNorth生成岛屿的过程



生成楼房



生成风格化模型

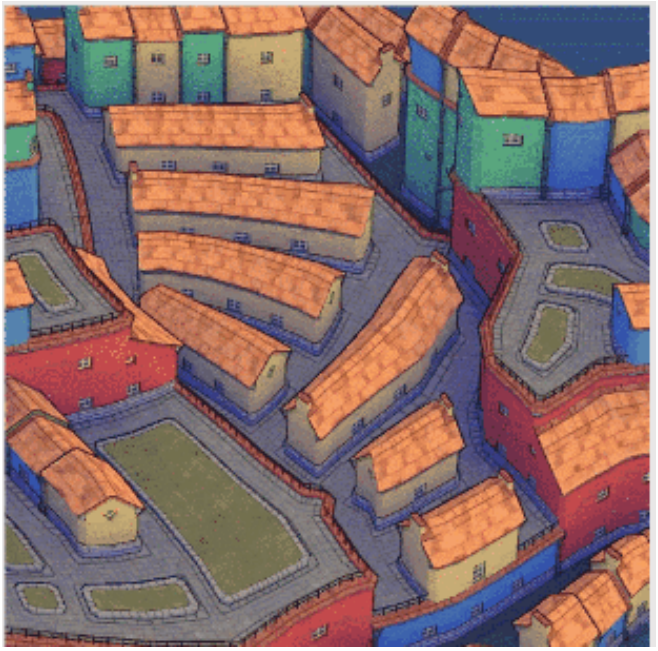


赛博朋克科幻风城市程序化生成

生成类地牢

机器写的诗句

BadNorth岛屿示意图



四边形网格 + 随机生成

5.总结

从以上案例可以看出,通过这个算法,可以做出各式各样的随机地形.以为它是依赖规则,在规则以下做随机坍缩.加以学习和使用,可以做出来很多很惊艳的东西~

5.1 利与弊

> **好处** :
>
> 相对于地图拼接的算法(Warcraft3地形拼接算法),这种算法实现出来的效果更灵活更丰富,更能适应各种需求.
>
> **坏处** :
>
> 1.算法复杂度较高而且不好量化,需要计算机自己试错再回溯...(可以通过多线程来计算)

> 2.暂时没有成熟的制作规范,模块原型的制作难以考虑所有可能的情况.制作进度不好把控.

5.2 模块原型的制作流程?

关于模块原型的制作流程,我个人有个建议可以先使用Unity的建模插件(ProBuilder等)制作简单的原型,然后再让美术同学细化原型.或者美术先制作整体的模型,再通过模型切割工具,把整体切分成多个模块原型.