**北京外国语大学网络教育学院**

**计算机专业毕业设计（论文）**

**策略游戏地图系统的设计与实现**

姓 名： 阮复伸

学 号： 181713600007

专 业： 计算机科学与技术

指导老师： 王洋

论文提交日期：

**北京外国语大学网络教育学院**

**学士学位论文诚信声明**

本人郑重声明： 所呈交的学士学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。论文所涉及的项目为本人亲自负责或者参与实施的项目。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品成果。本人完全了解本声明的法律结果由本人承担。

学士学位论文作者签名:

日期： 年 月 日

**目 录**

[**目 录** I](#_Toc40968913)

[**摘 要** I](#_Toc40968914)

[**Abstract** II](#_Toc40968915)

[**插 图 索 引** III](#_Toc40968916)

[**附 表 索 引** IV](#_Toc40968917)

[第1章 绪 论 5](#_Toc40968918)

[1.1 项目背景 5](#_Toc40968919)

[1.2 项目意义 5](#_Toc40968920)

[1.3 预期目标 6](#_Toc40968921)

[第2章 创建基础的正六边形地图系统 7](#_Toc40968922)

[2.1.1 HexMap的形状 7](#_Toc40968923)

[2.1.2 创建地图网格 10](#_Toc40968924)

[2.1.3 Dsadsada 10](#_Toc40968925)

[2.1.4 Dsadsadsa 10](#_Toc40968926)

[2.1.5 Dsadsadsa 11](#_Toc40968927)

[2.1.6 Dsadsadsa 11](#_Toc40968928)

[2.1.7 Dsadsadsa 11](#_Toc40968929)

[2.1.8 Dsadsadsa 11](#_Toc40968930)

[2.1.9 Dsadsadsa 11](#_Toc40968931)

[2.1.10 Dsadsadsa 11](#_Toc40968932)

[2.2.1 Sdsadsa 11](#_Toc40968933)

[2.2.2 dsadsa 11](#_Toc40968934)

[2.1 C# Jobs System 13](#_Toc40968935)

[第3章 Unity3D ECS架构设计目的和原理 15](#_Toc40968936)

[第4章 U独立游戏架构设计、项目编码 16](#_Toc40968937)

[4.1游戏的基本功能模块划分 16](#_Toc40968938)

[4.2部分游戏内核心代码 17](#_Toc40968939)

[第5章 性能测试 22](#_Toc40968940)

[**结** **论** 26](#_Toc40968941)

[**参 考 文 献** 27](#_Toc40968942)

[**致** **谢** 28](#_Toc40968943)

**摘 要**

地图系统作为游戏中的一个重要组成部分，在许多RPG游戏、策略游戏、回合制游戏中是不可或缺的。地图系统随着硬件的不断更新换代，从最初的2D平面类型，如FC恶魔城、SFC塞尔达传说、GBA圣剑传说，由像素矩阵绘制而成的地图系统，经历了伪3D俯视角类型，如魔力宝贝、石器时代，由多种单一贴图所构成的地图系统，到现在非常流行的3D地图系统，如奇迹时代3、文明6、无尽传奇、英雄无敌，由材质、贴图、模型构成的地图系统。

因此，地图系统的性能、存储容量、美观程度、灵活程度是一个游戏是否成功的重要因素。随着移动设备的不断普及，对游戏中地图系统的要求也在不断的提高。本文通过分析现有主流游戏的地图系统，结合了移动端快速生成、美术资源占用小、生成灵活、可数据化存储等要求，设计并开发了一个简单的可程序化生成与控制的地图系统。

本文主要工作如下

1. 查阅相关资料，收集相关类型的游戏，使用Unity3D、Cocos2D、Unreal Engine引擎导入其地图美术资源，分析其地图系统的组成和实现方式。
2. 综合比较同类游戏的地图系统，分析各自的优缺点。
3. 根据分析结果创建基本的地图系统架构，并进行编码。
4. 反复试验，修改代码，得到最优结果。

关键词： Unity3D；游戏引擎；地图系统；架构设计

**Abstract**

Along with the wide application of mobile devices, an increasing variety of applications and games are developed. The growing demand for the graphics, the sound effect and the particle effect from gamers highlight the importance of the architecture design of a game. A game with improper architecture design may have dropped frames, freeze interfaces and application crash, which gravely undermine the user experience. The possibly sudden emerge of massive particle effects or models on the screens of mobile devices requires the game architects to take all scenarios into consideration. Unity ECS is designed to address this challenge. Different from the traditional Unity designs, ECS employs the C# job system, multiple threads and synchronous processing pattern and therefore performs better to render massive particles and polygons.

The contents of this paper include

1. Learn about the concept and implementation of the Unity3D Job System + ECS architecture

2. Compile the Game Design Document

3. Implement the Unity Job System + ECS design in accordance with the Game Design Document

4. Code the game and arrange the art resources with the LWRP rendering mode

5. Test the game

**Key Words：Unity3D;Game engine;Mobile device;** **Architecture design**

**插 图 索 引**

[图2.0‑1-1 正方形网格 7](#_Toc40986013)

[图2.1-0‑2 正六边形外接圆和内切圆 8](#_Toc40986014)

[图2.1-0‑3 两种排列方式 9](#_Toc40986015)

**附 表 索 引**

表3.1 基础游戏设计文档 13

表5.1 三种设计模式的性能对比 38

# 第1章 绪 论

* 1. 项目背景

Unity3D是由Unity Technologies开发的一个让玩家轻松创建诸如三维视频游戏、建筑可视化、实时三维动画等类型互动内容的多平台的综合型游戏开发工具，是一个全面整合的专业游戏引擎。Unity类似于Director,Blender game engine, Virtools 或 Torque Game Builder等利用交互的图型化开发环境为首要方式的软件。其编辑器可运行在Windows、Linux、Mac OS X下，可发布游戏至Windows、Mac、Wii、iPhone、WebGL、Android等多种类型的平台上。

Unity3D在2018年引入了可编程渲染管线(Scriptable Render Pipeline)的概念，SRP主要分为两个方向，针对于PC、PS4、XBOX等高性能设备的High Definition Render Pipeline(HDRP)，和针对于移动设备、网页平台的Light Weight Render Pipeline(LWRP)。SRP的引入，使得开发者可以使用C#代码来控制渲染过程中，各个物体的渲染顺序和效果，让程序和渲染更加灵活、效率更高。

同样，Unity3D在2018年还引入了全新的ECS开发理念。传统的GameObject-MonoBehaviour机制适合无基础的新手快速入门，但是随着项目的复杂度不断增加，这种传统的设计模式会让代码变得难以阅读、维护和优化。而Unity3D新的ECS(Entity-Component-System)开发理念，因Blizzard开发的射击游戏Over Watch被开发者所熟知。ECS开发理念主要关注如何组建并处理游戏中的数据和行为。

使用了LWRP与ECS流程制作的Unity3D移动端游戏，相较于传统MonoBehavior 与PBR相结合制作游戏的流程，有以下优势：

程序结构更加清晰，代码易于维护。

在较低性能的移动设备中运行时，也可以启用更多的动态光照，保证画面的渲染效果。

LWRP以更少的性能消耗实现相同甚至更好的粒子特效、环境光照、Post Processing。

* 1. 项目意义

目前，各大应用市场中，使用Unity3D ECS框架与LWRP渲染管线的三维项目非常少，在多数移动端游戏中，还是以传统的BPR渲染方式为主。Post Processing大多依赖第三方组件，可靠性与稳定性较差。

但是，Unity3D ECS框架中所依赖的主要类库，均为Beta版本，有很多的不确定性，在各类主流移动端设备上未进行充分测试。并且在项目编码过程中，Unity3D官方可能会更新ECS框架类库，导致项目部分功能需要重写。

LWRP与ECS相结合的项目流程目前多应用在大型多人在线MMORPG或者3A单机游戏中，在中小型独立游戏中却很少有此类流程的应用。而且LWRP与传统BPR项目在Substance Designer贴图生成、材质制作、后期效果表现等环节内也有很大的不同。本论文尝试使用LWRP、ECS等Unity3D的新特性，以一款简单的独立游戏为载体，实现在移动平台(iPhone6s至iPhone11 Pro Max或同代Android设备)中，运行时在保证游戏整体的运行帧率条件下，实现更好的画面渲染效果、更多动态光照。

* 1. 预期目标

通过一个简单的独立游戏项目，了解并熟悉以下流程：

1. Unity3D中，使用Pro Builder创建简单的游戏原型，并在项目中使用。
2. 基于Unity Job System和ECS的游戏框架设计、编码
3. 在移动设备中测试游戏性能

# 第2章 创建基础的正六边形地图系统

* 1. HexMap的形状

在目前移动端与PC端的回合制策略游戏中，主要使用两种底图系统，分别是：正方形网格地图系统和正六边形网格地图系统。其中，日式RPG游戏大多使用正方形网格地图。目前，越来越多的游戏已经逐渐从正方形转变到了正六边形。

为什么使用六边形的网格地图？在传统游戏中。正方形网格相比其他形状，更容易绘制和定位。但是正方形网格也有明显的缺点，图2.1-1为一组正方形网格：

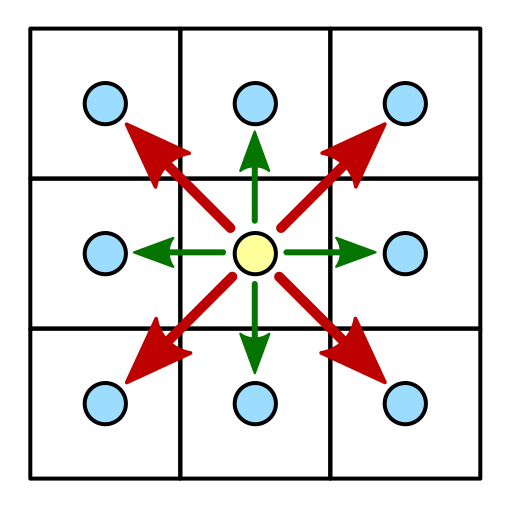


图2.0‑1-1 正方形网格

中间这个正方形网格中，一共有8个与其相邻的网格。与中间正方形网格边相邻的网格，称为水平和垂直相邻，与中间正方形网格对角相邻的网格，称为对角相邻。

从上图2.1中可以看出，如果每个正方形网格的边长为1，那么，水平和垂直相邻的正方形网格中点之间的距离也为1.但是，对角相邻的正方形网格中点之间的距离为倍的边长。

这种形式的相邻网格，造成了距离的差异化。在游戏中，如果一个人物在正方形网格中进行移动，这种距离的差异化将会更加明显。为了避免这种差异化，不同的游戏使用了不同的方法，也有不同的有点和缺点。其中一种方法，就是使用正六边形网格代替传统的正方形网格。图2.1-2为一组正六边形网格：

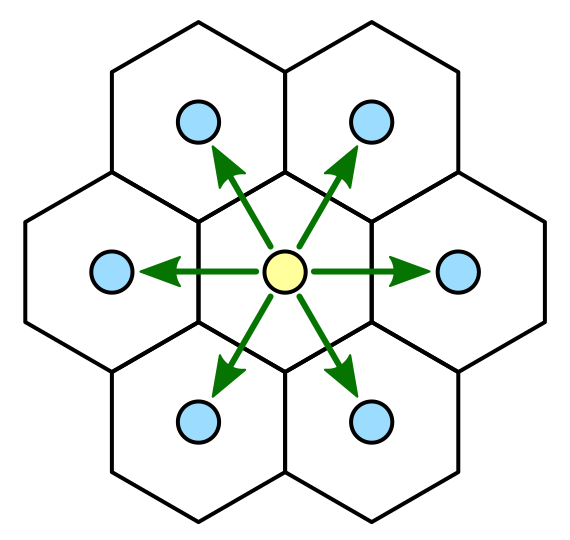


图2.1-2 正六边形网格

与正方形网格相比，在正六边形网格中，每个地图块之间中点的距离都是相等的，这样就避免了距离的差异化。当然，使用正六边形网格也会有其他的问题，不过相对于正方形地图网格的距离差异化更容易解决。

在开始设计正六边形网格地图之前，必须先确定每个正六边形地图块的边长。如图2.1-3所示。如果一个正六边形地图块的边长为10，那么，这个正六边形地图块的外接圆半径也是10。每个正六边形网格还有一个内切圆，根据勾股定理可以得出，其内切圆的半径为倍的边长，也就是。

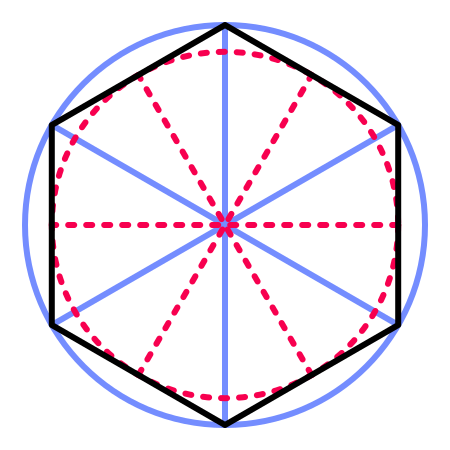


图2.1-0‑2 正六边形外接圆和内切圆

接下来，创建一个新的Unity3D项目，在该项目中创建HexMetrics.cs脚本。将正六边形地图块的边长和内切圆半径长度保存至一个静态类中，方便之后的模块引用，代码如下

1. **using** UnityEngine;
3. **public** **static** **class** HexMetrics
4. {
5. **public** **const** **float** outerRadius = 10f;
7. **public** **const** **float** innerRadius = outerRadius \* 0.866025404f;
8. }

确定了整列变形地图块的外接圆和内切圆半径后，接下来需要确定正六边形网格中，每个正六边形地图块的摆放姿态。在正六边形网格中，每个正六边形地图块会有两种摆放姿态，顶点朝上或者边朝上。如图2.1-4所示：

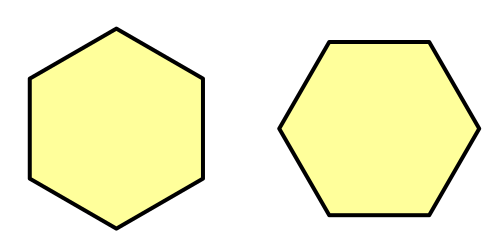


图2.1-0‑3 两种排列方式

为了方便计算与排列，这里选择顶点朝上的排列方式，并将整个正六边形网格放置在unity3D场景中的XZ平面上。从正六边形地图块正上方一个顶点的位置开始，顺时针排列一个其相邻的六个地图块。以下代码为记录每个正六边形地图块其六个顶点位置：

1. **public** **static** Vector3[] corners =
2. {
3. **new** Vector3(0f, 0f, outerRadius),
4. **new** Vector3(innerRadius, 0f, 0.5f \* outerRadius),
5. **new** Vector3(innerRadius, 0f, -0.5f \* outerRadius),
6. **new** Vector3(0f, 0f, -outerRadius),
7. **new** Vector3(-innerRadius, 0f, -0.5f \* outerRadius),
8. **new** Vector3(-innerRadius, 0f, 0.5f \* outerRadius)
9. };
   1. 创建地图网格

要创建一个正六边形网格，需要先创建单个的正六边形地图块。再让各个正六边形地图块按照一定的规则排列，组成正六边形网格。在Unity3D中，可以将一个正六边形地图块制作为预置组件，即Prefab，方便程序引用和统一修改。这里，将每个正六边形地图块称为HexCell，并创建HexCell.cs基础脚本，代码如下：

1. **using** UnityEngine;
3. **public** **class** HexCell : MonoBehaviour
4. {
5. }

创建正六边形地图块Prefab非常简单，首先，在场景中创建一个Plane，并挂载HexCell.cs脚本，然后将其转换成Prefab，结果如下图：

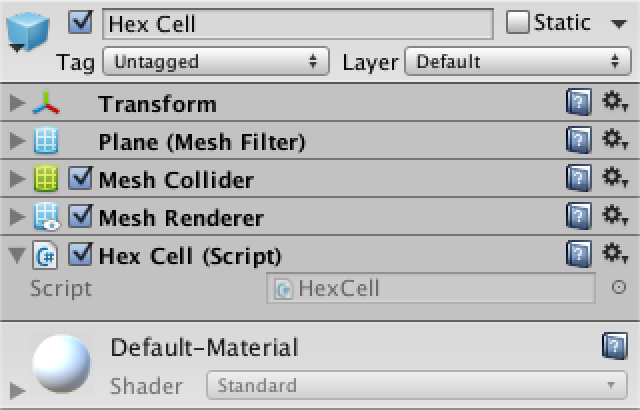


图2.2- 1

接下来，要创建一个容器，用来实例化正六边形地图块，并将每个地图块有序的排列起来。 将其命名为HexGrid。代码下：

1. **using** UnityEngine;
3. **public** **class** HexGrid : MonoBehaviour
4. {
5. **public** **int** width = 6;
6. **public** **int** height = 6;
8. **public** HexCell cellPrefab;
9. }

由于HexCell使用了Plane网格，Plane网格默认为 10x10 单位长度，所以在实例化HexCell时，需要加上边长的偏移量。实例化HexCell的代码如下：

1. **private** HexCell[] cells;
3. **void** Awake()
4. {
5. cells = **new** HexCell[height \* width];
7. **for** (**int** z = 0, i = 0; z < height; z++)
8. {
9. **for** (**int** x = 0; x < width; x++)
10. {
11. CreateCell(x, z, i++);
12. }
13. }
14. }
16. **void** CreateCell(**int** x, **int** z, **int** i)
17. {
18. Vector3 position;
19. position.x = x \* 10f;
20. position.y = 0f;
21. position.z = z \* 10f;
23. HexCell cell = Instantiate<HexCell>(cellPrefab);
24. cells[i] = cell;
26. cell.transform.SetParent(transform, **false**);
27. cell.transform.localPosition = position;
28. }

代码编辑完成后，在Unity3D内点击Play按钮，运行效果如图2.2-

* + 1. Dsadsada

在目前移动端与PC端的回合制策略游戏中，主要使用两种底图系统，分别是：正方形网格地图系统和正六边形网

* + 1. Dsadsadsa

在目前移动端与PC端的回合制策略游戏中，主要使用两种底图系统，分别是：正方形网格地图系统和正六边形网

* + 1. Dsadsadsa
    2. Dsadsadsa
    3. Dsadsadsa
    4. Dsadsadsa
    5. Dsadsadsa
    6. Dsadsadsa
    7. Sdsadsa
    8. dsadsa

在目前移动端与PC端的回合制策略游戏中，主要使用两种底图系统，分别是：正方形网格地图系统和正六边形网

然而，类层次结构越深，它就越脆弱。如果在实现开始后需求发生变化，反映代码中的这些变化通常需要向那些抽象类中添加或者删除功能。这些更改将继续影响所有子类，不管这些子类是否需要更改。结果是混乱的代码添加被推到树的根上，以便为更多的子类提供更改。

在Unity中，创建一个游戏中的实例是如下流程：

1. 创建一个GameObject。
2. 在GameObject上添加各类组件，如MeshRender、Collider、Rigidbody等
3. 创建 MonoBehaviour 脚本并将其添加到对象中，以便在运行时控制和更改这些组件的状态属性。

以上的步骤就是Unity中创建一个实例的基本流程。但是这种做法有它自己的缺点和性能问题。比如数据和逻辑是紧密耦合的，这意味着代码重用的频率较低，因为逻辑与特定数据相关联，无法单独分离出来。

如图1.1所示的 GameObject 和 Components 示例中，GameObject 依赖于 Transform、Renderer、Rigidbody 和 Collider 引用，在这些脚本中引用的对象分散在堆内存中。

在Unity中，传统的GameObject-MonoBehaviour设计思路中，游戏对象、其行为及其组件之间的内存引用如图2.2所示

Unity 通过这种方式可以在非常短的时间内完成原型构建并运行，这个也是Unity的特色。可以让开发者快速上手，但它对于性能来说并不太理想。每个引用类型都包含可能不需要访问的许多额外数据，这些未使用的成员也占用了处理器缓存中的空间。比如继承的Mono就是一个典型的案例，如果只需要使用现有组件的极少功能接口函数或者变量，则可以将其余部分视为浪费空间，如图2.3所示：

在图2.3中，粗体表示实际用于移动操作的成员，其余的就是浪费空间，若要移动 GameObject，脚本需要从 Transform 组件访问位置和旋转数据成员。当硬件从内存中获取数据时，缓存行中会填充许多可能无用的数据，如果只是为所有应该移动的GameObjects 设置一个只有位置和旋转成员的阵列，这将能够在很短的时间内执行，而且，类似这样的无用数据，会随着继承的层级深度和范围不断扩大，呈爆炸式的增长，最后导致严重影响游戏性能，导致掉帧、卡顿和崩溃等情况。

那么，该如何去掉无用的数据？ECS就是为解决此问题而设计的。

与传统的Unity面向对象设计（Object Oriented Design）相对而言的，是面向数据设计（Data Oriented Design）。而实体组件系统（Entity Component System系统，即ECS）正是面向数据设计的体现。新的ECS设计模式，可以消除低效的对象引用。

在ECS的架构中, 只对具体的实例感兴趣，例如坦克和汽车，而不是抽象的“车辆”，游戏中的任何游戏对象层次结构都应该是几乎完全平坦的。而不是从抽象的父类继承。

实体(Entity)代表游戏中一个具体的实例，比如坦克、汽车或士兵。然而，Entity没有特定于实体的逻辑。实际上，一个Entity几乎没有任何逻辑，而且仅仅是一个ID。

组件(Component)是功能模块，也可以称它为属性(attribute)。Component是Entity具有的属性，比如位置、移动速度、Collider和外观。Entity只不过是Component的容器，它是各部分的总和。重要的是，Entity不清楚它包含哪些部分，这意味着所有Entity都可以被游戏的其他部分以同样的方式对待。

因为Component只对自己负责，而不管它们属于哪个Entity。例如，Mesh Renderer组件包含渲染模型网格的功能。模型网格被分配给Component，Component被分配给实体，所以Entity不需要知道模型是什么样子。Entity所需要做的就是对它的每一个Component，每一帧调用一个通用的更新函数，每个Component都会做自己的事情。例如，Mesh Renderer就可以把自己绘制到显示器上。

这样设计的优点是Component是通用的，它们只是执行者，不管Component的父Entity是什么，Component都以相同的方式执行。因此，坦克物体的Mesh Renderer会像汽车物体的Mesh Renderer那样渲染自己，唯一的区别就是分配给每个Component的网格形状。因此，通过将不同的可重用Component插入到空Entity中，可以很容易地制造出不同类型的Entity。

这样，对于在开发期间的修改和之后的维护，都是非常灵活的。对Entity的更改通常涉及隔离地更改一个或两个Component，而不更改任何不相关的Component或污染其他Entity。新功能可以通过独立添加新Component来添加。

下图来此Unity官方ECS的说明文档，在下图中Bullet 实体没有附加Transform 或 Rigidbody 组件，Bullet 实体只是显式运行更新所需的原始数据，借助这个新系统，可以将逻辑与各个对象类型完全分离。

ECS具有很大的优势：它不仅可以提高缓存效率，缩短访问时间；它还支持在需要使用这种数据对齐方式的现代 CPU 中采用自动矢量化/SIMD技术，极大地提高了游戏的性能。2.5图为缓存中的碎片和继承Mono系统生成的空间浪费，两种设计模式的对比：

* 1. C# Jobs System

开发者在使用多线程代码时，线程争抢资源就可能会发生。如果开发者没有想到这个问题，可能会导致潜在的严重错误。除此之外，上下文切换的成本很高，因此学习如何平衡工作负载以尽可能高效地运行是很困难的。新的 Unity C# Jobs System可以解决所有这些难题，结合上文中的案例，如图2.6所示：

图2.6 C# Job System系统原理

图2.6展示了简单的子弹运动系统，大多数游戏开发者都会为 GameObject 编写一个管理器，如 Bullet Manager ，这些管理器会管理一个 GameObjects 列表，并每帧更新场景中所有子弹活动的位置。这非常符合使用 C# Jobs System的条件，由于子弹运动可以单独处理，因此非常适合并行化，借助 C# Jobs System，可以轻松地将此功能独立出来，并行运行不同的数据块，作为开发者，只需要专注于游戏逻辑代码即可。

ECS和 C# Jobs System的结合可以提供更强大的功能，由于实体组件系统以高效、紧凑的方式设置数据，因此Jobs System可以拆分数据阵列，以便可以高效地并行操作。

# 第3章 Unity3D ECS架构设计目的和原理

传在程序开发中，每一种文档在不同阶段都有不用的作用，最后这些文档都会汇总至同一份文档中——游戏设计文档(Game Design Document)，或者简写为GDD。

GDD并没有严格的格式或长度要求，但是要保证其可以精确第描述出游戏。一般情况下，一份合格的GDD，要具有以下信息：

* 游戏名称
* 游戏系统
* 目标玩家年龄（ESRB分级）
* 游戏概要、可玩性
* 游戏玩法的独特性
* 与众不同的卖点

此独立游戏的基本GDD内容如表3.1:

|  |  |
| --- | --- |
| 游戏设计文档 | |
| 游戏名称 | 僵尸大战 |
| 游戏系统 | iOS、PC |
| 目标玩家年龄（ESRB分级） | 10岁以上 【E 10+】 |
| 游戏概要、可玩性 | 玩家控制多个士兵，对战成群而来的僵尸大军 |
| 游戏玩法的独特性 | 简单易上手，可单独操作每个士兵的站位，自由度高。 |
| 与众不同的卖点 | 屏幕中可以容纳更多的敌人和玩家控制角色，游戏在移动设备中运行流畅，模型及贴图细节高。与多数回合制、半回合制游戏不同，本游戏更突出实时性和压迫感，相比其他类型更加有代入感。 |

表3.1 基础游戏设计文档

# 第4章 U独立游戏架构设计、项目编码

4.1游戏的基本功能模块划分

此游戏使用Unity Job System + ECS + Burst Compiler的设计模式，旨在屏幕中出现大量网格、粒子效果的情况下，不改变光照条件、不降低模型细节、不减少粒子发射量的同时最大幅度的提高游戏运行帧数。

Unity Job System + ECS的设计模式中，数据相对于传统Unity设计架构更加扁平化，游戏物体、数据与操作逻辑完全分离，通过多种不同的组合方式，来实现多个游戏物体的形态和功能。核心设计思路是组合优于继承，是去掉每一个物体上不必要的数据。改善Unity传统设计模式中继承耦合过重及多继承的问题。

根据Unity Job System + ECS官方推荐的设计架构，图4.1为本游戏的流程图：

图4.1 游戏流程图

根据游戏流程图，可以将游戏的功能模块分为以下几个部分：

* 生成系统（Spawning System）负责生成场景中各类的entity，包括敌人、玩家控制角色，武器特效、死亡特效等。
* 地图模块系统（Quadrant System）由于设计模式由传统的OOD变为了DOD，地图寻路部分参照了部分A star Path Finding的思路，将一整快地图拆分为多个小块地图组成的阵列，每个阵列内的寻路在当前区块内单独计算。这样即保证了寻路的准确性，又充分发挥出了Unity Job System并行计算的特性。
* 瞄准系统（Targeting System）包括敌人寻找随机玩家人物，玩家人物瞄准最近的敌人。在每个Entity中单独计算自身的目标。
* 寻路系统（Path Finding System）与地图模块系统相结合，当玩家人物或敌人进入一个地图区块时，会遍历当前区块中是否有目标，如果存在目标，则会使用A star Path Finding的算法遍历地图内的可移动路径，向目标前进。如果当前区块内没有目标，则会在周围8格的范围内寻找最近的目标。
* 队列系统（Queued System）与生成系统相结合，当一个Entity被生成时，会自动判断自己的类型，然后放入相应的对象池。
* 游戏人物动画系统（Animation）传统的Unity动画系统，可以通过2D sprite来生成人物动画序列，同事可以改变很小一部分贴图，就可以生成新的人物动画。

4.2部分游戏内核心代码

初始化玩家人物：

将各类Component组合设置为预置体，然后再根据预置体实例化玩家人物，其中Marine属性为该Entity的标签，用来在地图模块系统中判断当前地图模块中存在哪些单位。

初始化敌人：

敌人的各类属性设置与玩家人物的属性设置基本相同，区别在于动画组件和骨骼骨骼组件引用了不同的资源。

游戏难度控制：

生成僵尸时需要控制其生成频率，如果在短时间没内生大量僵尸，不仅会造成设备卡顿，还会在玩家没进入游戏状态之前造成游戏失败，降低了游戏体验。

敌人移动控制：

僵尸的移动控制，该部分除了要计算移动距离，还会控制僵尸的动画组件的状态。

动画组件中，会引用已经设置完成的2D sprite，缓存成为不同状态的序列帧，根据状态来切换显示。

地图模块系统：

该系统为游戏中较为重要的一个部分，为了充分利用Unity Job System + ECS + Burst的优势，区别与传统的存储游戏内物体实例进行位置判断的模式，将整块大地图切割为若干个小地图，每一个小地图区块均为一个独立的Entity，保存在一个哈希表中。这样，每一个小地图就可以充分使用Unity Job System的优势来进行独立计算，可以非常快速的获得当前地图区块中的敌人和玩家人物的数量及位置信息。

1. }

玩家人物瞄准系统：

与地图模块系统配合使用，判断当前玩家人物所在地图区块中是否存在敌人，如果存在则会触发攻击状态，如果没有则会寻找当前区块周边8个区块范围。

1. **public** **class** MarineTargetZombieSystem : JobComponentSystem
2. {
3. **private** **struct** Job : IJobForEachWithEntity<Translation, Skeleton\_PlayAnim, HasTarget, MarineShoot>
4. {
5. **public** **float** time;
6. **public** **float** deltaTime;
7. **public** NativeQueue<MarineShotZombieAction>.Concurrent queuedActions;
9. **public** **void** Execute(Entity entity, **int** index, **ref** Translation translation, **ref** Skeleton\_PlayAnim skeletonPlayAnim, **ref** HasTarget hasTarget, **ref** MarineShoot marineShoot)
10. {
11. float3 targetDir = math.normalize(hasTarget.targetPosition - translation.Value);
12. **if** (time >= marineShoot.nextShootTimer)
13. {
14. // Shoot
15. marineShoot.nextShootTimer = time + marineShoot.nextShootTimerMax;
16. skeletonPlayAnim.PlayAnimForced(ECS\_UnitAnimType.TypeEnum.dMarine\_Attack, targetDir, Skeleton\_Anim\_OnComplete.Create(ECS\_UnitAnimType.TypeEnum.dMarine\_Aim, targetDir));
17. queuedActions.Enqueue(**new** MarineShotZombieAction
18. {
19. marineEntity = entity,
20. zombieEntity = hasTarget.targetEntity,
21. damageAmount = 10,
22. });
23. }
24. **else**
25. {
26. //skeletonPlayAnim.PlayAnim(ECS\_UnitAnimType.TypeEnum.dMarine\_Aim, targetDir, default);
27. }
28. }
29. }
31. **protected** **override** JobHandle OnUpdate(JobHandle inputDeps)
32. {
33. Job job = **new** Job
34. {
35. time = Time.time,
36. deltaTime = Time.deltaTime,
37. queuedActions = TestECS.queuedActions.ToConcurrent()
38. };
39. **return** job.Schedule(**this**, inputDeps);
40. }
41. }

队列系统：

游戏场景内生成的各类人物，都会自动加入队列协同的对象池中，对象池会单独计算每个Entity的生命值，根据生命值得出该人物是否已经死亡，并赋予其相应的状态。

1. **public** **class** QueuedActionSystem : ComponentSystem
2. {
3. **protected** **override** **void** OnUpdate()
4. {
5. EntityQuery entityQuery = GetEntityQuery(**typeof**(Zombie));
6. **int** entityCount = entityQuery.CalculateLength();
8. MarineShotZombieAction marineShotZombieAction;
9. **while** (TestECS.queuedActions.TryDequeue(**out** marineShotZombieAction))
10. {
11. **if** (EntityManager.Exists(marineShotZombieAction.marineEntity) && EntityManager.Exists(marineShotZombieAction.zombieEntity))
12. {
13. float3 marinePosition = EntityManager.GetComponentData<Translation>(marineShotZombieAction.marineEntity).Value;
14. float3 zombiePosition = EntityManager.GetComponentData<Translation>(marineShotZombieAction.zombieEntity).Value;
15. float3 marineToZombieDir = math.normalize(zombiePosition - marinePosition);
17. **bool** bonusEffects = (entityCount < 400 || UnityEngine.Random.Range(0, 100) < 60);
18. **if** (bonusEffects)
19. {
20. WeaponTracer.Create(marinePosition + marineToZombieDir \* 10f, (Vector3)zombiePosition + UtilsClass.GetRandomDir() \* UnityEngine.Random.Range(0, 20f));
21. Shoot\_Flash.AddFlash(marinePosition + marineToZombieDir \* 14f);
22. Blood\_Handler.SpawnBlood(2, zombiePosition, marineToZombieDir);
23. UtilsClass.ShakeCamera(TestECS.GetCameraShakeIntensity(), .05f);
24. }
26. Health zombieHealth = EntityManager.GetComponentData<Health>(marineShotZombieAction.zombieEntity);
27. zombieHealth.health -= marineShotZombieAction.damageAmount;
28. **if** (zombieHealth.health < 0)
29. {
30. // Zombie dead!
31. FlyingBody.TryCreate(GameAssets.i.pfEnemyFlyingBody, zombiePosition, marineToZombieDir);
32. EntityManager.DestroyEntity(marineShotZombieAction.zombieEntity);
33. EntityManager.RemoveComponent<HasTarget>(marineShotZombieAction.marineEntity);
34. }
35. **else**
36. {
37. // Zombie still has health
38. EntityManager.SetComponentData(marineShotZombieAction.zombieEntity, zombieHealth);
39. }
40. }
41. **else**
42. {
43. **if** (EntityManager.Exists(marineShotZombieAction.marineEntity) && !EntityManager.Exists(marineShotZombieAction.zombieEntity))
44. {
45. // Marine exists but zombie is dead
46. EntityManager.RemoveComponent<HasTarget>(marineShotZombieAction.marineEntity);
47. }
48. }
49. }
50. }
52. **protected** **override** **void** OnDestroy()
53. {
54. TestECS.queuedActions.Dispose();
55. }
56. }

# 第5章 性能测试

在游戏发布至移动端后，可以连接Unity Profiler查看当前APP在移动端的运行情况。这里摘取游戏中常用的游戏帧数、CUP使用率、合批数及多线程状态作为判断依据。

当游戏中出现相同数量的网格、粒子时，使用传统开发模式的数据情况如图5.1：

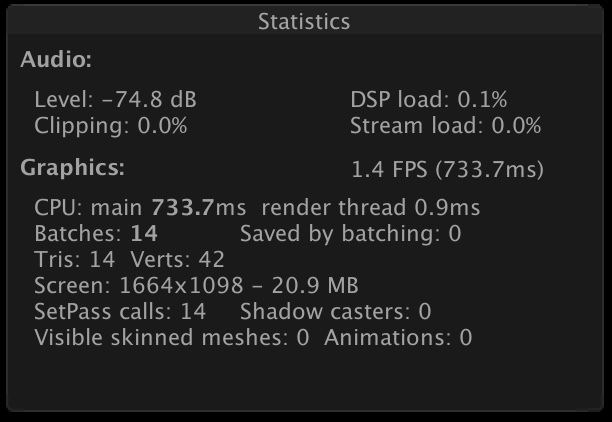


图5.1 传统模式中性能数据

Unity Profiler中多线程使用情况入图5.2：

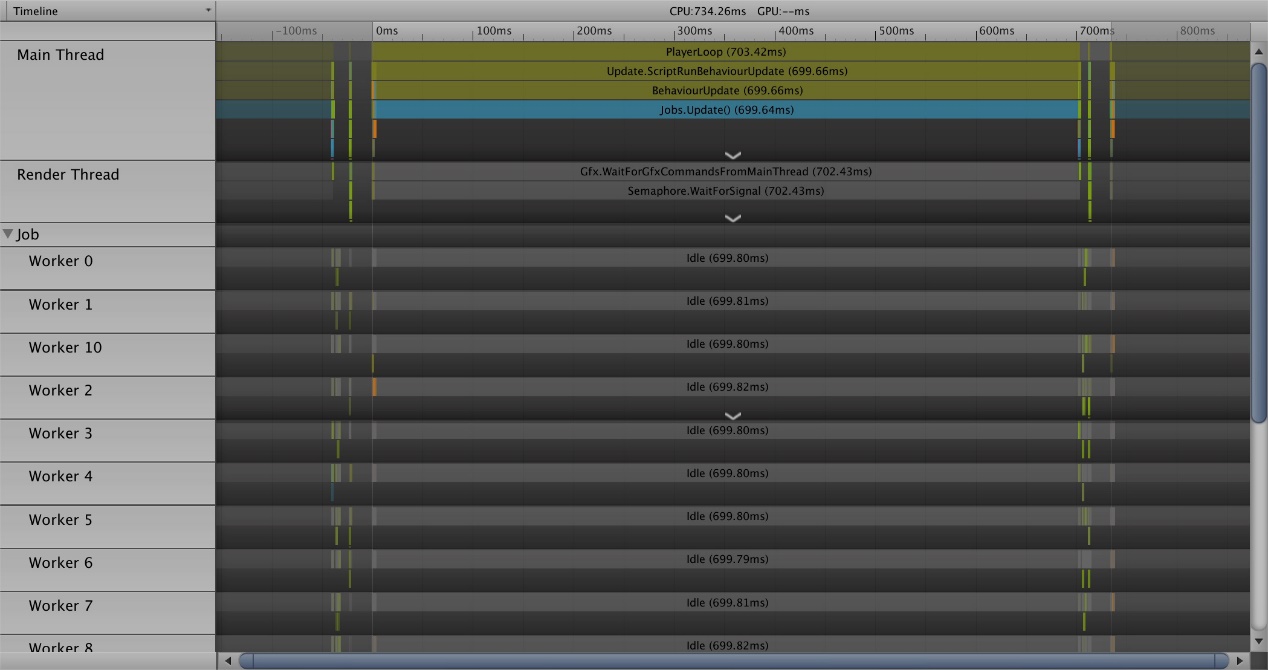


图5.2 传统模式中多线程使用情况

同样场景下，使用了Unity Job System + ECS时的数据如图5.3：

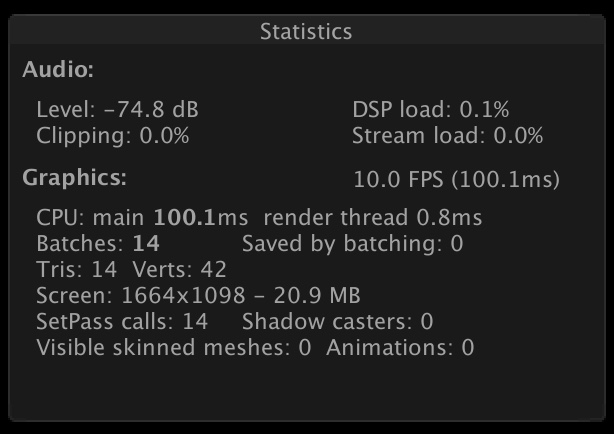


图5.3 Unity Job System + ECS性能数据

Unity Profiler中多线程的使用情况：

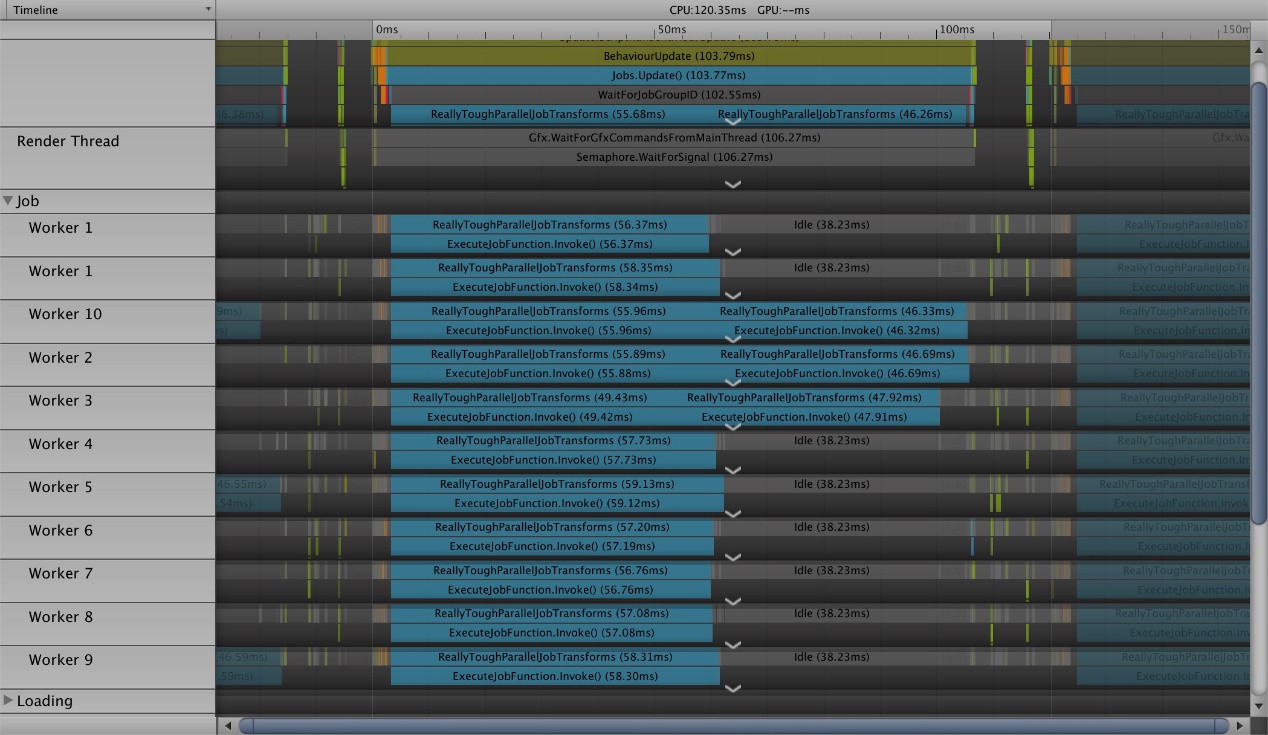


图5.4 Unity Job System + ECS多线程使用情况

同样场景下，使用了Unity Job System + ECS + Burst Compiler时的数据如图5.5

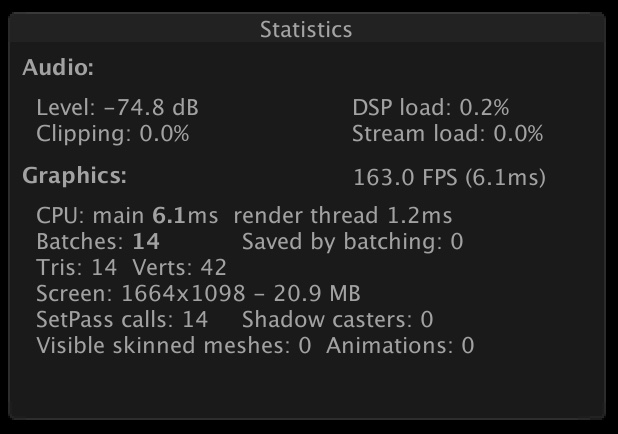


图5.5 Unity Job System + ECS + Burst Compiler性能数据

Unity Profiler中多线程使用情况：

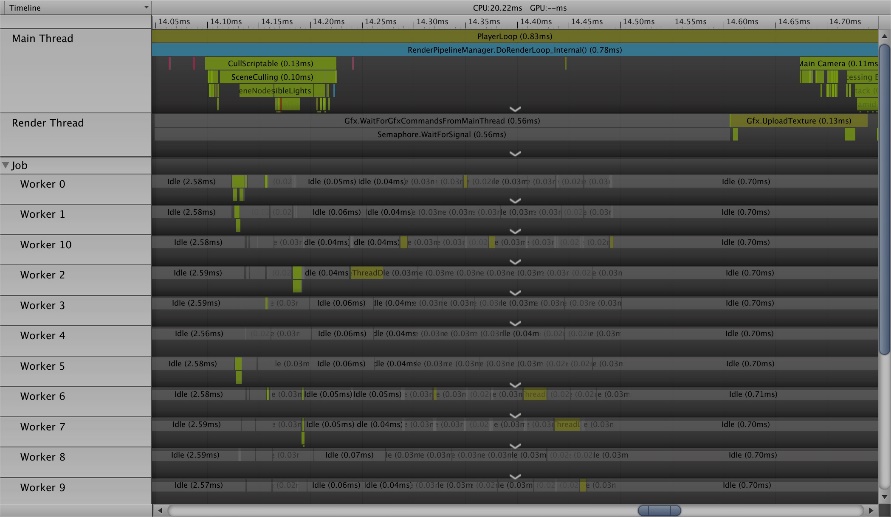


图5.6 Unity Job System + ECS + Burst Compiler多线程使用情况

两种设计模式的对比如下表所示

|  |  |  |  |
| --- | --- | --- | --- |
|  | Unity传统设计模式 | Unity Job System + ECS （Buster Off） | Unity Job System + ECS （Buster On） |
| 运行平台 | iOS | iOS | iOS |
| 屏幕内物体数量 | 300 | 300 | 300 |
| CPU运算耗时（ms） | ~700 | ~100 | ~7 |
| 每秒帧率  （ms/frame） | ~1.5 | ~11 | ~170 |

表5.1 三种设计模式的性能对比

**结** **论**

在传统Unity3D的开发模式中，在有大量网格、粒子特效同屏的情况下，一般的解决办法是，创建一个Manager去管理所有在屏幕中出现的网格、粒子的实例，为每个游戏物体设置多个级别的网格LOD，当屏幕内网格或粒子达到一定数量的时候，采取降低网格精细度或减少粒子发射量的方式来保持游戏帧数。这样做的弊端是：

1. 在不同性能的设备上运行时，网格LOD精细度比较难以控制。
2. 如果针对不同主流机型设置不同的网格LOD，美术人员的工作量非常繁重。
3. 无法发挥多核并行运算的优势。

所以，Unity Job System与ECS结合，非常适用于数量庞大的网格在同一屏幕内出现的场景。加之Unity又推出了与Job System相配合的Burst Compiler，可以将Unity3D中的代码，转换为对应平台中高度优化的原生代码，使得程序的整体性能进一步提升。当屏幕内出现同样数量的网格和粒子效果时，新的Unity Job System + ECS + Burst Compiler设计模式，比传统的开发模式中游戏帧数会提升20到50倍。同时，这种设计模式大幅度提升了代码的可维护性和复用性，避免了多层级继承后，如果需求变动，造成的不可预知的风险。

但是，Unity Job System + ECS在追求高性能和并行计算的同时，带来的是更高的电量消耗和更大的发热量。根据现有移动设备的电池续航及用户体验情况，使用传统架构与ECS相结合的方式，是目前移动端游戏比较可行的解决方案之一。

**参 考 文 献**

[1] 陈嘉栋 [M] 《Unity3D 脚本编程 使用C#语言开发跨平台游戏》 北京 电子工业出版社 2019.6

[2] Scott Rogers [M] 《通关 游戏设计之道》 北京 人民邮电出版社 2013.11

[3] Erich Gamma [M] 《Design Patterns》 北京 机械工业出版社 2019.5

[4] 冯乐乐 [M] 《Unity Shader入门精要》 北京 人民邮电出版社 2016.6

[5] 加藤政村 [M] 《Unity游戏设计与实现 南梦宫一线程序员的开发实例》 北京 人民邮电出版社 2015.2

[6] Unity3D官方LWRP文档 [DB/OL] https://docs.unity3d.com/Packages/com.unity.render-pipelines.lightweight@6.9/manual/index.html

[7] Unity3D ECS官方介绍 [DB/OL] https://connect.unity.com/p/part-1-unity-ecs-briefly-about-ecs

[8] Unity官方ECS示例项目Github源码 [DB/OL] https://github.com/Unity-Technologies/EntityComponentSystemSamples

**致** **谢**

紧张、充实而又难忘的大学学习生涯即将结束，在大学四年和撰写论文期间得到了许多人的帮助，使我终身难以忘怀。轻风系不住流云，流云却带走了岁月，打开尘封的记忆，往事如风却又历历在目，大学的学习生活即将结束。在这里我首先要感谢这四年来为我授课的各位老师，真心地说一句：你们辛苦了！感谢王洋老师在我的论文选题、定稿以及中期检查等方面都给了我精心的指导。您提出的宝贵意见使我在论文选题、撰写以及修改的过程中，不再像当初那样茫然无措，而是知道自己论文的不足和修改的方向。您正直、严谨的治学态度对我影响颇深，受益匪浅，无论在今后的学习还是工作当中，我都铭记于心。