论文初稿

封面

- 题目：基于Unity3D的跨平台回合制游戏App应用设计与开发

- 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：

目录

摘要

随着移动设备的不断普及，移动设备中的各类应用与游戏也变得丰富多彩。然而玩家对游戏的画面、音效、粒子特效等要求也不断增高，越发凸显出一个游戏的架构设计的好坏。当一个游戏的架构设计不合理时，会发生掉帧、卡顿、程序崩溃等各种问题，严重影响了玩家在游戏中的体验。尤其是在移动设备平台中，如果在屏幕中突然出现大量粒子效果或大量的模型，将对游戏的架构设计有很高的要求。而Unity ECS则正是为了解决这个问题诞生的，ECS相较于传统的Unity设计思路而言，使用了C# Job system、多线程、协同处理等多种方式。在渲染大量粒子、多边形等方面更加有优势。

本文主要工作如下

1. 参照官方文档，了解Unity3D Job System + ECS架构设计的概念及实现方式。

2. 编写独立游戏的Game Design Document。

3. 根据Game Design Document，实现Unity Job System + ECS 设计模式

4. 编写游戏代码，使用LWRP渲染模式组织美术资源。

5. 测试实机运行效率。

关键词： Unity3D；游戏引擎；移动设备；架构设计

Abstract

插 图 索 引

附 表 索 引

第一章 绪论

1.1 研究背景及意义

研究背景：

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：

1. 程序结构更加清晰，代码易于维护。

2. 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。

3. 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。

研究意义

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。

研究现状

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，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，目前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。

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

第二章 Unity3D Job System + ECS架构设计目的和原理

1. Unity Job System 与 ECS简介

传统的Unity设计设计框架中，MonoBehavior和各种Component在其中扮演了重要的角色。游戏中每一个Object，无论是树木、房屋、士兵、车辆、灯光或者音效，都是由它自身的类定义的。而有些类可能具有很多相似的特性，因此这些相似的类可以从更抽象的类继承公共行为。例如坦克和卡车这两个类是非常相似的，他们都具有行驶、停止、转向和发出声音等特性。这些公共行为对很多类型的车辆都适用，所以可以从更抽象的车辆类中继承。这样做的好处是非常直观，开发人员可以直接从某个非常抽象的类中实例化出一个他们期望的实体，并且同时还可以给实体添加更多的特性。

面向对象游戏的设计需要这个类层次结构的详细设计，每个具体的游戏内对象都继承自一个越来越抽象的类树，如果这个层次结构在实现之前就已经计划好了，那么很可能构建一个利用OOP继承的大型复杂游戏。

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：

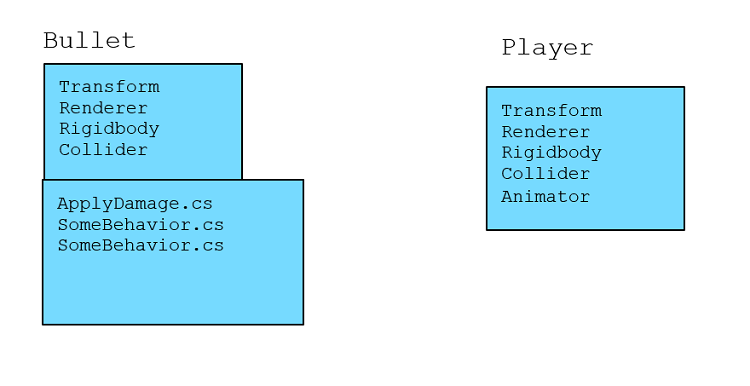
1 创建一个GameObject。

2 在GameObject上添加各类组件，如MeshRender、Collider、Rigidbody等

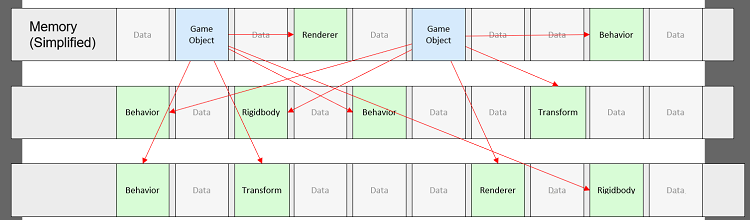
3 创建 MonoBehaviour 脚本并将其添加到对象中，以便在运行时控制和更改这些组件的状态属性

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。

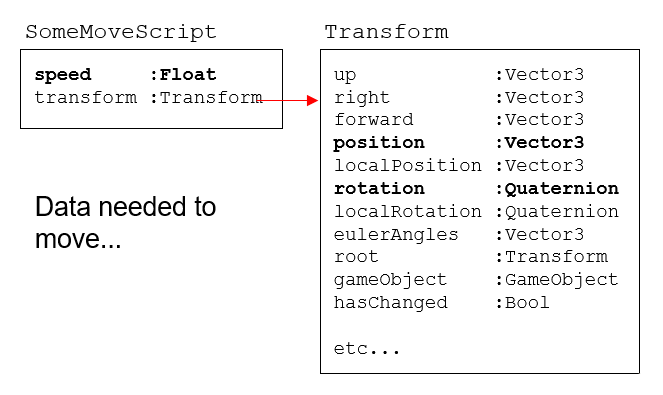
例如下图所示的 GameObject 和 Components 示例中，GameObject 依赖于 Transform、Renderer、Rigidbody 和 Collider 引用，在这些脚本中引用的对象分散在堆内存中。



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下图所示



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。比如我们继承的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，如图所示：



在上图中，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。

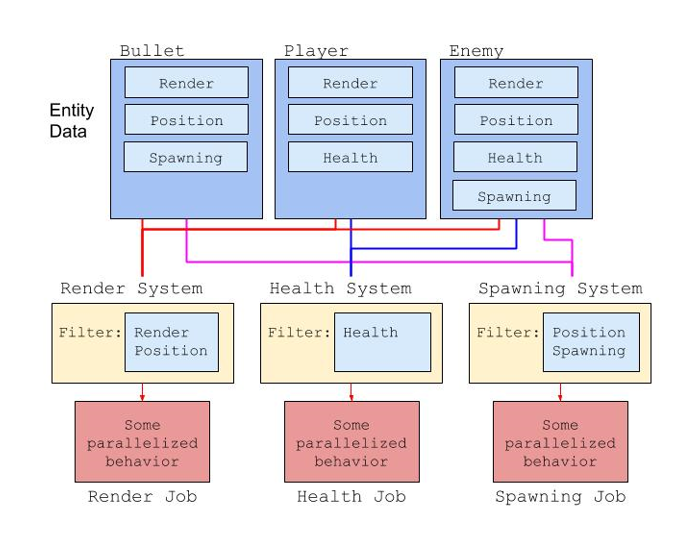
组件(Component)是功能模块，也可以称它为属性(attribute)。Component是Entity具有的属性，比如位置、移动速度、Collider和外观。Entity只不过是Component的容器，它是各部分的总和。重要的是，Entity不清楚它包含哪些部分，这意味着所有Entity都可以被游戏的其他部分以同样的方式对待。

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。

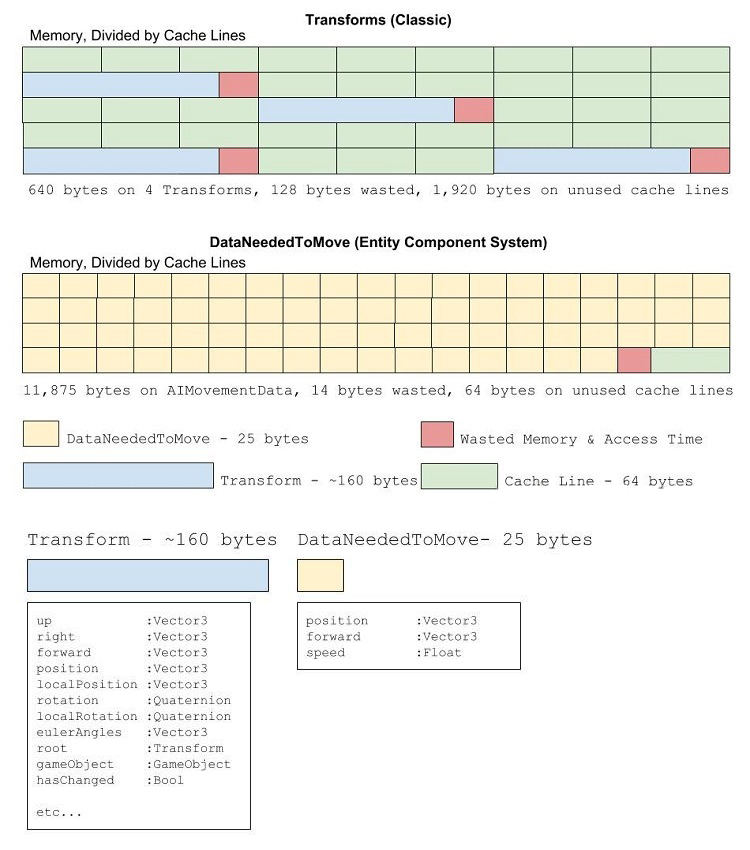
这样设计的优点是Component是通用的，它们只是执行者，不管Component的父Entity是什么，Component都以相同的方式执行。因此，坦克物体的Mesh Renderer会像汽车物体的Mesh Renderer那样渲染自己，唯一的区别就是分配给每个Component的网格形状。因此，通过将不同的可重用Component插入到空Entity中，可以很容易地制造出不同类型的Entity。

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。

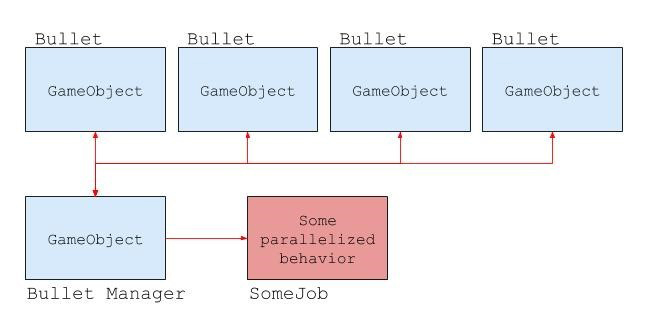


ECS具有很大的优势：它不仅可以提高缓存效率，缩短访问时间；它还支持在需要使用这种数据对齐方式的现代 CPU 中采用自动矢量化/SIMD技术，极大地提高了游戏的性能。下图为缓存中的碎片和继承Mono系统生成的空间浪费，两种设计模式的对比：



2 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，如下图所示：



上图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。

第三章 独立游戏的游戏设计文档编写

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内容如下表

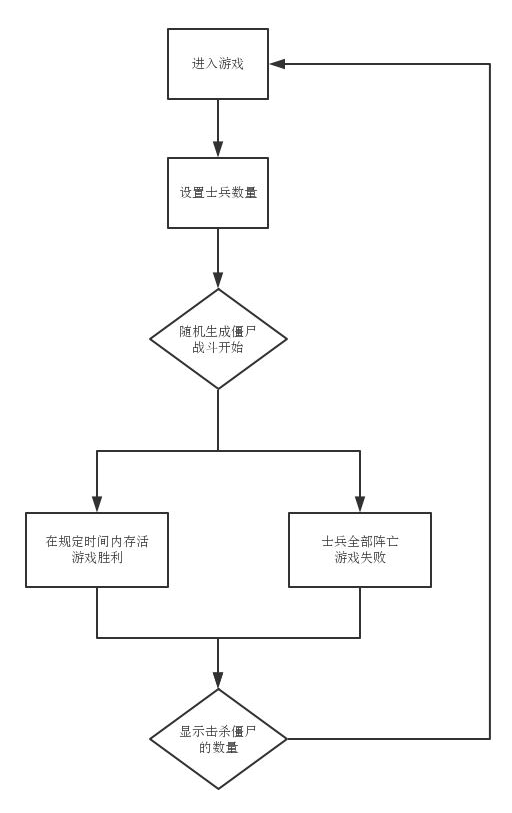
|  |  |
| --- | --- |
| 游戏设计文档 | |
| 游戏名称 | 僵尸大战 |
| 游戏系统 | iOS、PC |
| 目标玩家年龄（ESRB分级） | 10岁以上 【E 10+】 |
| 游戏概要、可玩性 | 玩家控制多个士兵，对战成群而来的僵尸大军 |
| 游戏玩法的独特性 | 简单易上手，可单独操作每个士兵的站位，自由度高。 |
| 与众不同的卖点 | 屏幕中可以容纳更多的敌人和玩家控制角色，游戏在移动设备中运行流畅，模型及贴图细节高。与多数回合制、半回合制游戏不同，本游戏更突出实时性和压迫感，相比其他类型更加有代入感。 |

第四章 独立游戏架构设计、项目编码

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，本游戏的流程图如下：



1. 游戏的基本功能模块划分为：
   1. 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。
   2. 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。
   3. 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。
   4. 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。
   5. 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。
   6. 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。
2. 部分游戏核心代码

初始化玩家人物：

1. **private** **void** SpawnMarine(float3 spawnPosition)
2. {
3. EntityArchetype entityArchetype = entityManager.CreateArchetype(
4. **typeof**(Marine),
5. **typeof**(Skeleton\_Data),
6. **typeof**(Skeleton\_Material),
7. **typeof**(Skeleton\_PlayAnim),
8. **typeof**(QuadrantEntity),
9. **typeof**(FindTargetData),
10. **typeof**(Health),
11. **typeof**(MarineShoot),
12. **typeof**(MoveTo),
13. **typeof**(EntityAnims),
14. **typeof**(Translation)
15. );
17. Entity entity = entityManager.CreateEntity(entityArchetype);
19. entityManager.SetComponentData(entity, **new** Translation { Value = spawnPosition });
20. entityManager.SetComponentData(entity, **new** Skeleton\_Data { frameRate = 1f });
21. entityManager.SetComponentData(entity, **new** Skeleton\_Material { materialTypeEnum = Skeleton\_Material.TypeEnum.Marine });
22. entityManager.SetComponentData(entity, **new** Skeleton\_PlayAnim { ecsUnitAnimTypeEnum = ECS\_UnitAnimType.TypeEnum.dBareHands\_Idle, animDir = UnitAnim.AnimDir.Down });
23. entityManager.SetComponentData(entity, **new** MarineShoot { nextShootTimerMax = .1f });
24. entityManager.SetComponentData(entity, **new** Health { health = 50 });
25. entityManager.SetComponentData(entity, **new** MoveTo { move = **false**, position = spawnPosition, moveSpeed = 40f });
26. entityManager.SetComponentData(entity, **new** EntityAnims { idleAnimType = ECS\_UnitAnimType.TypeEnum.dMarine\_Idle, walkAnimType = ECS\_UnitAnimType.TypeEnum.dMarine\_Walk });
27. entityManager.SetComponentData(entity, **new** QuadrantEntity { typeEnum = QuadrantEntity.TypeEnum.Marine });
28. entityManager.SetComponentData(entity, **new** FindTargetData { targetRange = 100f });
30. ECS\_Animation.PlayAnimForced(entity, ECS\_UnitAnimType.TypeEnum.dBareHands\_Idle, **new** Vector3(0, -1), **default**);
31. }

初始化敌人

1. **private** **void** SpawnZombie(**bool** spawnZombieTop)
2. {
3. EntityArchetype zombieArchetype = entityManager.CreateArchetype(
4. **typeof**(Zombie),
5. **typeof**(Skeleton\_Data),
6. **typeof**(Skeleton\_Material),
7. **typeof**(Skeleton\_PlayAnim),
8. **typeof**(QuadrantEntity),
9. **typeof**(FindTargetData),
10. **typeof**(Health),
11. **typeof**(ZombieAttack),
12. **typeof**(MoveTo),
13. **typeof**(EntityAnims),
14. **typeof**(Translation)
15. );
17. Entity entity = entityManager.CreateEntity(zombieArchetype);
19. float3 spawnPosition;
20. **if** (spawnZombieTop)
21. {
22. spawnPosition = **new** float3(UnityEngine.Random.Range(-100f, 100f), 400f, 0f);
23. }
24. **else**
25. {
26. spawnPosition = UtilsClass.GetRandomDir() \* 400f;
27. }
29. entityManager.SetComponentData(entity, **new** Translation { Value = spawnPosition });
30. entityManager.SetComponentData(entity, **new** Skeleton\_Data { frameRate = 1f });
31. entityManager.SetComponentData(entity, **new** Skeleton\_Material { materialTypeEnum = Skeleton\_Material.TypeEnum.Zombie });
32. entityManager.SetComponentData(entity, **new** Skeleton\_PlayAnim { ecsUnitAnimTypeEnum = ECS\_UnitAnimType.TypeEnum.dBareHands\_Idle, animDir = UnitAnim.AnimDir.Down });
33. entityManager.SetComponentData(entity, **new** ZombieAttack { nextAttackTimerMax = .1f });
34. entityManager.SetComponentData(entity, **new** Health { health = 30 });
36. entityManager.SetComponentData(entity, **new** MoveTo
37. {
38. move = **true**,
39. position = spawnZombieTop ? spawnPosition + **new** float3(0, -700f, 0) : (float3)Vector3.zero,
40. moveSpeed = 15f
41. });
43. **bool** useZombieAims = UnityEngine.Random.Range(0, 100) < 50;
44. entityManager.SetComponentData(entity, **new** EntityAnims
45. {
46. idleAnimType = useZombieAims ? ECS\_UnitAnimType.TypeEnum.dZombie\_Idle : ECS\_UnitAnimType.TypeEnum.dBareHands\_Idle,
47. walkAnimType = useZombieAims ? ECS\_UnitAnimType.TypeEnum.dZombie\_Walk : ECS\_UnitAnimType.TypeEnum.dBareHands\_Walk
48. });
50. entityManager.SetComponentData(entity, **new** QuadrantEntity { typeEnum = QuadrantEntity.TypeEnum.Zombie });
51. entityManager.SetComponentData(entity, **new** FindTargetData { targetRange = 100f });
53. ECS\_Animation.PlayAnimForced(entity, ECS\_UnitAnimType.TypeEnum.dBareHands\_Idle, **new** Vector3(0, -1), **default**);
54. }

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。

1. **private** **void** HandleZombieSpawning()
2. {
3. zombieSpawnTimer -= Time.deltaTime;
4. **if** (zombieSpawnTimer < 0)
5. {
6. zombieSpawnTimer = zombieSpawnTimerMax;
7. zombieSpawnTimerMax = .2f - Time.time \* .002f;
8. zombieSpawnTimerMax = Mathf.Clamp(zombieSpawnTimerMax, .07f, 1f);
10. **int** spawnZombieCount = Mathf.RoundToInt(1 + Time.time \* .05f);
12. **if** (Time.time < 20f)
13. {
14. **for** (**int** i = 0; i < spawnZombieCount; i++)
15. {
16. SpawnZombie(**true**);
17. }
18. }
19. **else**
20. {
21. SpawnZombie(**true**);
22. **for** (**int** i = 0; i < spawnZombieCount; i++)
23. {
24. SpawnZombie(**false**);
25. }
26. }
27. }
28. }

僵尸移动控制，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。

1. **public** **class** UnitMoveSystem : JobComponentSystem
2. {
4. **private** **struct** Job : IJobForEachWithEntity<MoveTo, Translation, Skeleton\_PlayAnim, EntityAnims>
5. {
7. **public** **float** deltaTime;
9. **public** **void** Execute(Entity entity, **int** index, [ReadOnly] **ref** MoveTo moveTo, **ref** Translation translation, **ref** Skeleton\_PlayAnim skeletonPlayAnim, **ref** EntityAnims entityAnims)
10. {
11. **if** (moveTo.move)
12. {
13. **if** (math.distance(translation.Value, moveTo.position) > 1f)
14. {
15. // Move to position
16. float3 moveDir = math.normalize(moveTo.position - translation.Value);
17. translation.Value += moveDir \* moveTo.moveSpeed \* deltaTime;
18. skeletonPlayAnim.PlayAnim(entityAnims.walkAnimType, moveDir, **default**);
19. }
20. **else**
21. {
22. // Already there
23. skeletonPlayAnim.PlayAnim(entityAnims.idleAnimType, float3.zero, **default**);
24. }
25. }
26. }
28. }
30. **protected** **override** JobHandle OnUpdate(JobHandle inputDeps)
31. {
32. Job job = **new** Job
33. {
34. deltaTime = Time.deltaTime,
35. };
36. **return** job.Schedule(**this**, inputDeps);
37. }
39. }

地图模块系统，将整块大地图分块生成，每一个地图区块为一个独立的Entity，会单独计算当中的玩家人物和僵尸的位置信息

1. **using** System.Collections;
2. **using** System.Collections.Generic;
3. **using** UnityEngine;
4. **using** Unity.Entities;
5. **using** Unity.Jobs;
6. **using** Unity.Transforms;
7. **using** Unity.Burst;
8. **using** Unity.Mathematics;
9. **using** Unity.Collections;
11. **public** **struct** QuadrantEntity : IComponentData
12. {
13. **public** TypeEnum typeEnum;
15. **public** **enum** TypeEnum
16. {
17. Marine,
18. Zombie,
19. }
20. }
22. **public** **struct** QuadrantData : IComponentData
23. {
24. **public** Entity entity;
25. **public** float3 position;
26. **public** QuadrantEntity quadrantEntity;
27. }

30. **public** **class** ECS\_QuadrantSystem : ComponentSystem
31. {
32. **public** **static** NativeMultiHashMap<**int**, QuadrantData> quadrantDataHashMap;
34. **public** **static** **void** Init()
35. {
36. quadrantDataHashMap = **new** NativeMultiHashMap<**int**, QuadrantData>(0, Allocator.Persistent);
37. }

40. **public** **struct** EntityWithPosition
41. {
42. **public** Entity entity;
43. **public** float3 position;
44. }
46. **private** **const** **int** quadrantYMultiplier = 1000;
47. **private** **const** **int** quadrantCellSize = 70;
49. **private** **static** **int** GetPositionHashMapKey(float3 position)
50. {
51. **return** (**int**)(math.floor(position.x / quadrantCellSize) + (quadrantYMultiplier \* math.floor(position.y / quadrantCellSize)));
52. }
54. **public** **static** **void** DebugDrawQuadrant(float3 position)
55. {
56. Vector3 lowerLeft = **new** Vector3(math.floor(position.x / quadrantCellSize) \* quadrantCellSize, math.floor(position.y / quadrantCellSize) \* quadrantCellSize);
57. Debug.DrawLine(lowerLeft, lowerLeft + **new** Vector3(+1, +0) \* quadrantCellSize);
58. Debug.DrawLine(lowerLeft, lowerLeft + **new** Vector3(+0, +1) \* quadrantCellSize);
59. Debug.DrawLine(lowerLeft + **new** Vector3(+1, +0) \* quadrantCellSize, lowerLeft + **new** Vector3(+1, +1) \* quadrantCellSize);
60. Debug.DrawLine(lowerLeft + **new** Vector3(+0, +1) \* quadrantCellSize, lowerLeft + **new** Vector3(+1, +1) \* quadrantCellSize);
61. }

64. [BurstCompile]
65. [RequireComponentTag(**typeof**(QuadrantEntity))]
66. **private** **struct** SetEntityHashMapJob : IJobForEachWithEntity<Translation>
67. {
68. **public** NativeMultiHashMap<**int**, Entity>.Concurrent nativeMultiHashMap;
70. **public** **void** Execute(Entity entity, **int** index, **ref** Translation translation)
71. {
72. **int** hashMapKey = GetPositionHashMapKey(translation.Value);
73. nativeMultiHashMap.Add(hashMapKey, entity);
74. }
75. }
77. [BurstCompile]
78. [RequireComponentTag(**typeof**(QuadrantEntity))]
79. **private** **struct** SetEntityWithPositionHashMapJob : IJobForEachWithEntity<Translation>
80. {
81. **public** NativeMultiHashMap<**int**, EntityWithPosition>.Concurrent nativeMultiHashMap;
83. **public** **void** Execute(Entity entity, **int** index, **ref** Translation translation)
84. {
85. **int** hashMapKey = GetPositionHashMapKey(translation.Value);
86. nativeMultiHashMap.Add(hashMapKey, **new** EntityWithPosition { entity = entity, position = translation.Value });
87. }
88. }
90. [BurstCompile]
91. [RequireComponentTag(**typeof**(QuadrantEntity))]
92. **private** **struct** SetQuadrantDataHashMapJob : IJobForEachWithEntity<Translation, QuadrantEntity>
93. {
94. **public** NativeMultiHashMap<**int**, QuadrantData>.Concurrent nativeMultiHashMap;
96. **public** **void** Execute(Entity entity, **int** index, **ref** Translation translation, **ref** QuadrantEntity quadrantEntity)
97. {
98. **int** hashMapKey = GetPositionHashMapKey(translation.Value);
99. nativeMultiHashMap.Add(hashMapKey, **new** QuadrantData
100. {
101. entity = entity,
102. position = translation.Value,
103. quadrantEntity = quadrantEntity
104. });
105. }
106. }
108. [BurstCompile]
109. [RequireComponentTag(**typeof**(QuadrantData))]
110. **private** **struct** FindClosestTargetJob : IJobForEachWithEntity<Translation, QuadrantEntity, FindTargetData>
111. {
112. [ReadOnly] **public** NativeMultiHashMap<**int**, QuadrantData> targetHashMap;
113. **public** NativeHashMap<Entity, QuadrantData>.Concurrent unitTargetHashMap;
115. **public** **void** Execute(Entity entity, **int** index, **ref** Translation translation, **ref** QuadrantEntity quadrantEntity, **ref** FindTargetData findTargetData)
116. {
117. **int** unitHashMapKey = GetPositionHashMapKey(translation.Value);
118. float3 unitPosition = translation.Value;
119. Entity targetEntity = Entity.Null;
120. float3 targetPosition = **new** float3(0, 0, 0);
121. TrySetClosestTarget(unitHashMapKey, unitPosition, quadrantEntity.typeEnum, **ref** targetEntity, **ref** targetPosition);
122. TrySetClosestTarget(unitHashMapKey - 1, unitPosition, quadrantEntity.typeEnum, **ref** targetEntity, **ref** targetPosition); // Left
123. TrySetClosestTarget(unitHashMapKey + 1, unitPosition, quadrantEntity.typeEnum, **ref** targetEntity, **ref** targetPosition); // Right
124. TrySetClosestTarget(unitHashMapKey + quadrantYMultiplier - 1, unitPosition, quadrantEntity.typeEnum, **ref** targetEntity, **ref** targetPosition); // Up Left
125. TrySetClosestTarget(unitHashMapKey + quadrantYMultiplier, unitPosition, quadrantEntity.typeEnum, **ref** targetEntity, **ref** targetPosition); // Up Center
126. TrySetClosestTarget(unitHashMapKey + quadrantYMultiplier + 1, unitPosition, quadrantEntity.typeEnum, **ref** targetEntity, **ref** targetPosition); // Up Right
127. TrySetClosestTarget(unitHashMapKey - quadrantYMultiplier - 1, unitPosition, quadrantEntity.typeEnum, **ref** targetEntity, **ref** targetPosition); // Down Left
128. TrySetClosestTarget(unitHashMapKey - quadrantYMultiplier, unitPosition, quadrantEntity.typeEnum, **ref** targetEntity, **ref** targetPosition); // Down Center
129. TrySetClosestTarget(unitHashMapKey - quadrantYMultiplier + 1, unitPosition, quadrantEntity.typeEnum, **ref** targetEntity, **ref** targetPosition); // Down Right
131. **if** (math.distance(unitPosition, targetPosition) > findTargetData.targetRange)
132. {
133. targetEntity = Entity.Null;
134. }
136. **if** (targetEntity != Entity.Null)
137. {
138. unitTargetHashMap.TryAdd(entity, **new** QuadrantData { entity = targetEntity, position = targetPosition });
139. }
140. }
142. **private** **void** TrySetClosestTarget(**int** quadrantHashMapKey, float3 unitPosition, QuadrantEntity.TypeEnum unitTypeEnum, **ref** Entity targetEntity, **ref** float3 targetPosition)
143. {
144. QuadrantData targetQuadrantData;
145. NativeMultiHashMapIterator<**int**> nativeMultiHashMapIterator;
146. **if** (targetHashMap.TryGetFirstValue(quadrantHashMapKey, **out** targetQuadrantData, **out** nativeMultiHashMapIterator))
147. {
148. **do**
149. {
150. **if** (targetQuadrantData.quadrantEntity.typeEnum != unitTypeEnum)
151. {
152. **if** (targetEntity == Entity.Null)
153. {
154. targetEntity = targetQuadrantData.entity;
155. targetPosition = targetQuadrantData.position;
156. }
157. **else**
158. {
159. **if** (math.distance(unitPosition, targetQuadrantData.position) < math.distance(unitPosition, targetPosition))
160. {
161. targetEntity = targetQuadrantData.entity;
162. targetPosition = targetQuadrantData.position;
163. }
164. }
165. }
166. } **while** (targetHashMap.TryGetNextValue(**out** targetQuadrantData, **ref** nativeMultiHashMapIterator));
167. }
168. }
169. }
171. **private** **struct** SetTargetJob : IJobForEachWithEntity<Translation>
172. {
173. [ReadOnly] **public** NativeHashMap<Entity, QuadrantData> entityTargetHashMap;
174. **public** EntityCommandBuffer.Concurrent entityCommandBuffer;
176. **public** **void** Execute(Entity entity, **int** index, **ref** Translation translation)
177. {
178. QuadrantData quadrantData;
179. **if** (entityTargetHashMap.TryGetValue(entity, **out** quadrantData))
180. {
181. entityCommandBuffer.RemoveComponent(index, entity, **typeof**(HasTarget));
182. entityCommandBuffer.AddComponent(index, entity, **new** HasTarget { targetEntity = quadrantData.entity, targetPosition = quadrantData.position });
183. }
184. }
185. }
187. **private** **static** **int** GetEntityCountInHashMap(NativeMultiHashMap<**int**, QuadrantData> quadrantDataHashMap, **int** quadrantHashMapKey)
188. {
189. QuadrantData quadrantData;
190. NativeMultiHashMapIterator<**int**> nativeMultiHashMapIterator;
191. **int** count = 0;
192. **if** (quadrantDataHashMap.TryGetFirstValue(quadrantHashMapKey, **out** quadrantData, **out** nativeMultiHashMapIterator))
193. {
194. **do**
195. {
196. count++;
197. } **while** (quadrantDataHashMap.TryGetNextValue(**out** quadrantData, **ref** nativeMultiHashMapIterator));
198. }
199. **return** count;
200. }

203. **protected** **override** **void** OnDestroy()
204. {
205. quadrantDataHashMap.Dispose();
206. }
208. **protected** **override** **void** OnUpdate()
209. {
210. EntityQuery entityQuery = Entities.WithAll<QuadrantEntity, Translation, FindTargetData>().ToEntityQuery();
212. quadrantDataHashMap.Clear();
213. **if** (entityQuery.CalculateLength() > quadrantDataHashMap.Capacity)
214. {
215. quadrantDataHashMap.Capacity = entityQuery.CalculateLength();
216. }
218. NativeHashMap<Entity, QuadrantData> entityTargetHashMap = **new** NativeHashMap<Entity, QuadrantData>(entityQuery.CalculateLength(), Allocator.TempJob);
220. SetQuadrantDataHashMapJob setQuadrantDataHashMapJob = **new** SetQuadrantDataHashMapJob
221. {
222. nativeMultiHashMap = quadrantDataHashMap.ToConcurrent(),
223. };
224. JobHandle setEntityHashMapJobHandle = JobForEachExtensions.Schedule(setQuadrantDataHashMapJob, entityQuery);
225. setEntityHashMapJobHandle.Complete();

228. // Cycle through all Units and FindTarget
229. FindClosestTargetJob findClosestTargetJob = **new** FindClosestTargetJob
230. {
231. targetHashMap = quadrantDataHashMap,
232. unitTargetHashMap = entityTargetHashMap.ToConcurrent(),
233. };
234. JobHandle findClosestTargetJobHandle = JobForEachExtensions.Schedule(findClosestTargetJob, entityQuery);
235. findClosestTargetJobHandle.Complete();

238. SetTargetJob setTargetJob = **new** SetTargetJob
239. {
240. entityTargetHashMap = entityTargetHashMap,
241. entityCommandBuffer = PostUpdateCommands.ToConcurrent(),
242. };
243. JobHandle setTargetJobHandle = JobForEachExtensions.Schedule(setTargetJob, entityQuery);
244. setTargetJobHandle.Complete();
246. entityTargetHashMap.Dispose();
248. //此为测试模块，查看每一个地图区块的范围
249. //DebugDrawQuadrant(UtilsClass.GetMouseWorldPosition());
250. }
251. }

玩家人物瞄准系统，当僵尸进入玩家人物的瞄准区域内，会自动攻击最近的一个僵尸

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. }

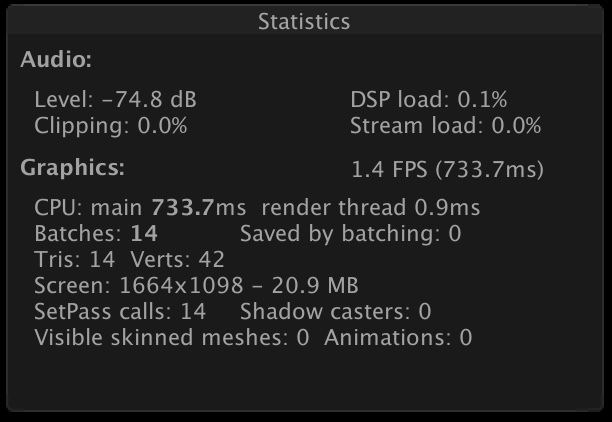
队列系统，游戏场景内生成的各类人物，都会自动加入队列协同的对象池中，对象池会单独计算每个Entity的生命值，根据生命值得出该人物是否已经死亡，并赋予其相应的状态。

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. }

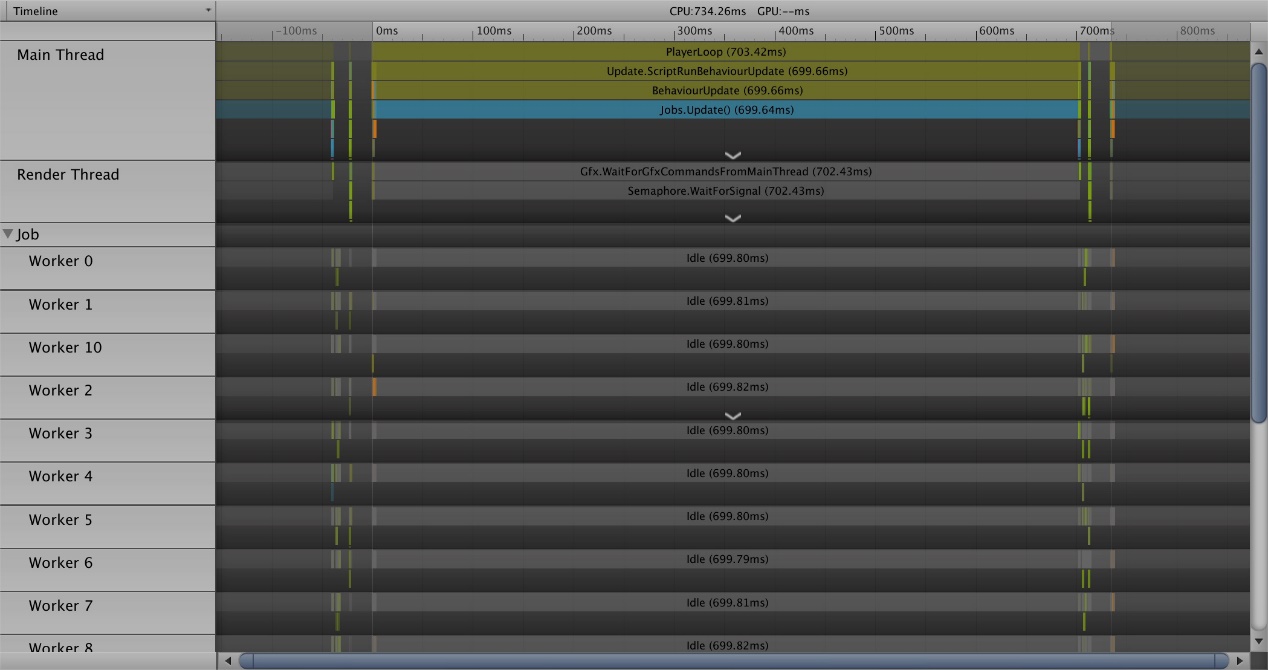
第五章 性能测试

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。

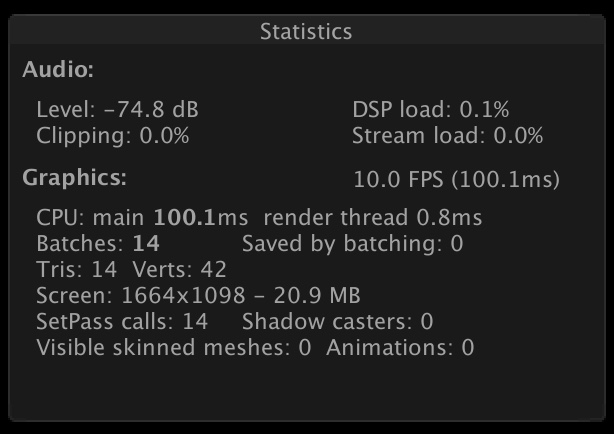
当游戏中出现相同数量的网格、粒子时，使用传统开发模式的数据情况如下图：



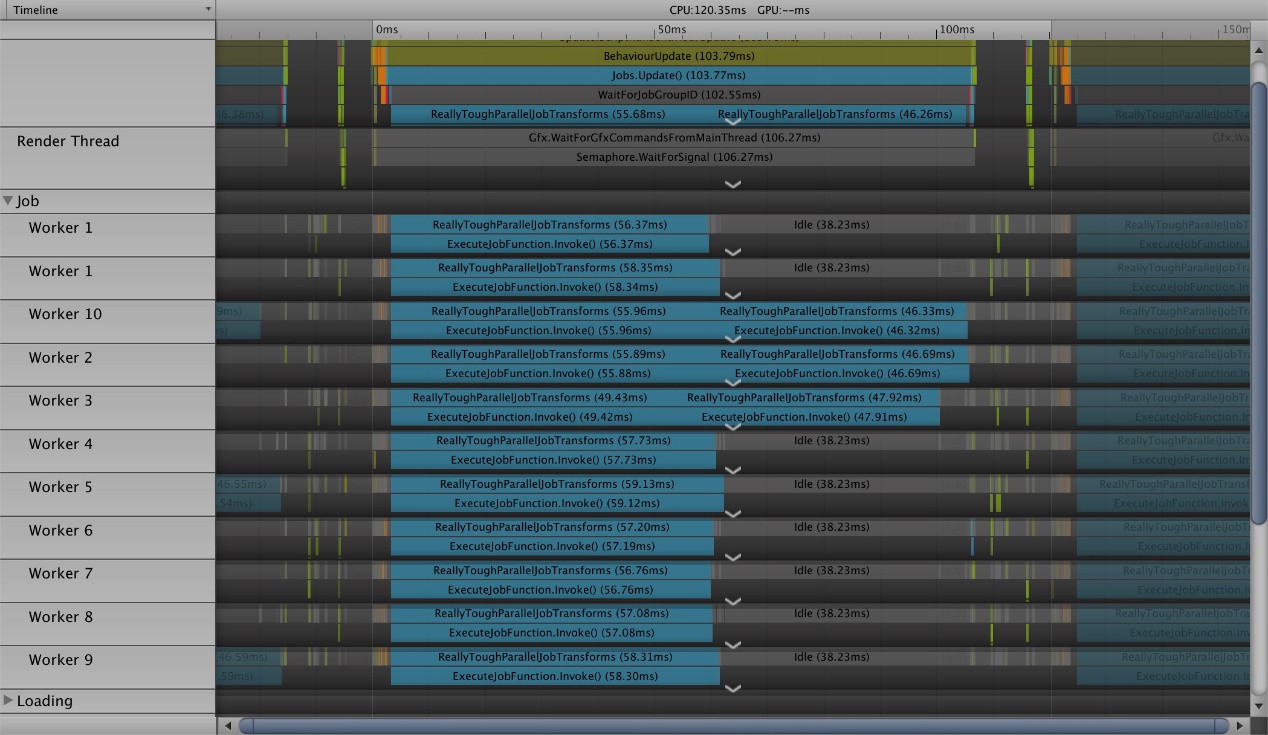
Unity Profiler中多线程使用情况：



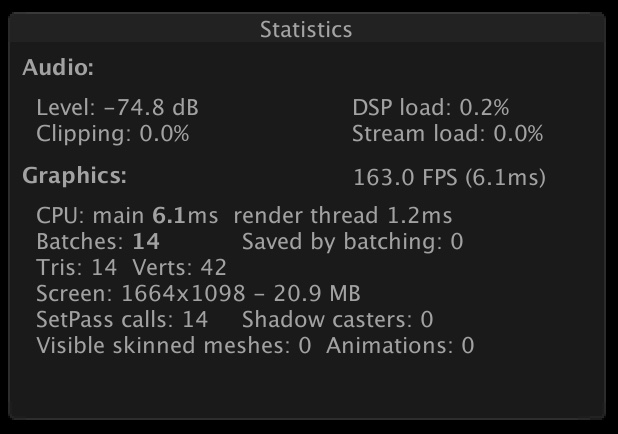
同样场景下，使用了Unity Job System + ECS时的数据如下图：



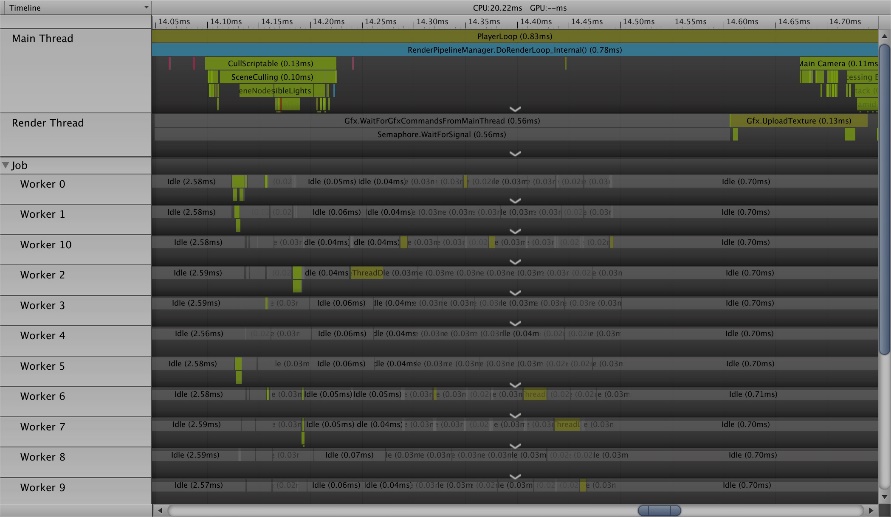
Unity Profiler中多线程使用情况：



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



Unity Profiler中多线程使用情况：



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 |

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，在大学学习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。