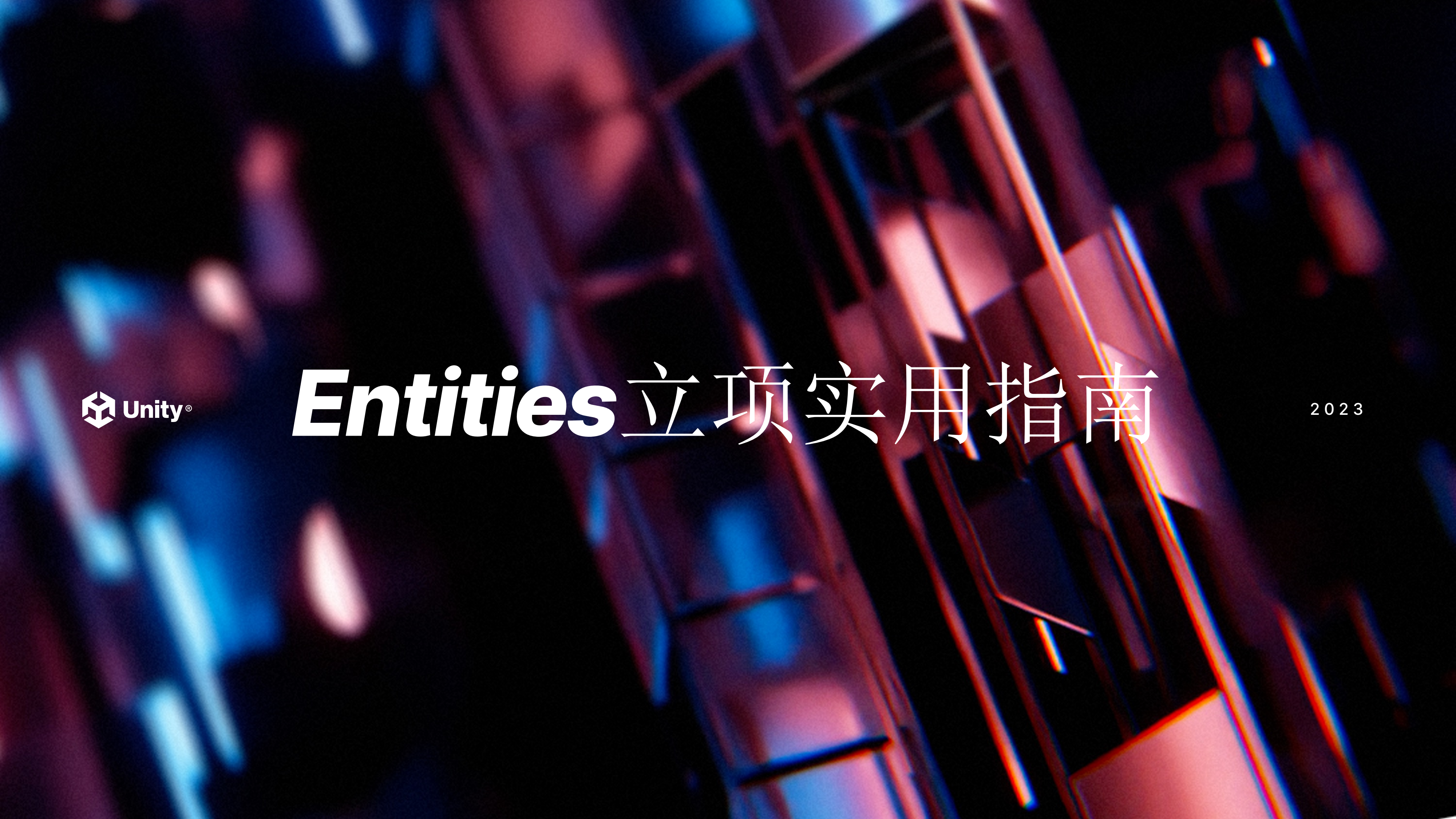


Unity®

Entities立项实用指南

2023

大纲

→ 使用DOTS的策略

→ 使用Entities的策略

使用DOTS的策略

使用DOTS的策略

→ DOTS是一系列技术的组合

- HPC#
- Burst compiler
- C# Jobsystem
- Entities
- 基于Entities的一系列软件包

使用DOTS的策略

→ HPC# & Burst compiler & C# JobSystem

- 发布于Unity 2018
- 到Unity 2023为止，经过了6个大版本迭代
- 相当成熟，功能齐全
- 在Unity packages和Asset Store上被大量使用

使用DOTS的策略

→ Entities & 基于Entities的packages

- 2022.3发布1.0版本
- Entities Component System
- 有了一些基础的package(Entities graphics等)
- 全新的工作流（Baker，Content Management）

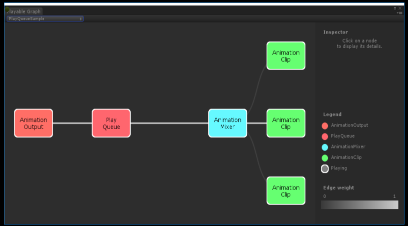
Unity®

使用DOTS的策略

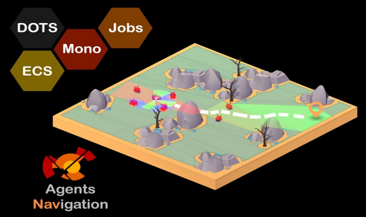
- → 是否使用Entities
  - — C# Job System + Burst Compiler
  - — Entities + C# Job System + Burst Compiler

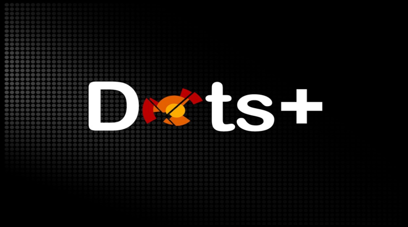
使用DOTS的策略

→ C# Job System + Burst Compiler案例



Click on a node to display its details.





使用DOTS的策略

- Unity正在向用户开放越来越多的底层接口
- 基本都会支持C# Job System
  - NavMeshQuery
  - MeshDataArray
  - TransformAccess
  - RaycastCommand
  - BatchRendererGroup

使用DOTS的策略

→ C# Job System总结

- 解决项目中的性能热点，如物理模拟，寻路等
- 模块相对独立，与其他模块互动比较少
- 适合团队中相对资深的成员

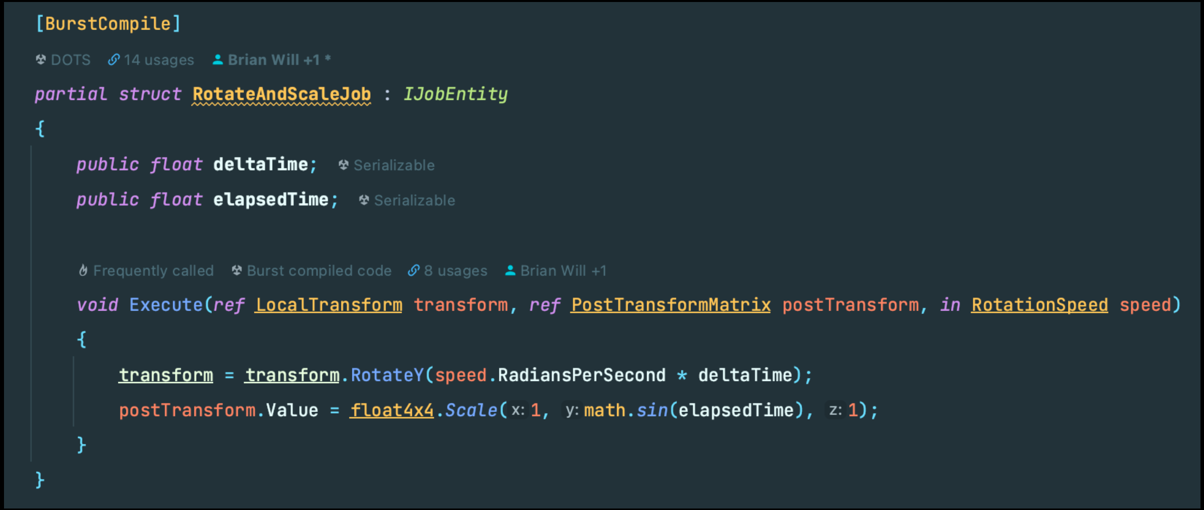
- 不适合编写一般业务逻辑
- Why?

使用DOTS的策略

- 不适合一般业务逻辑
  - 显示管理内存（Native Collections）
  - HPC#（C+，Low Level Language）
  - CPU架构（cache line，多核）
  - 并行编程
  - SIMD
- 能否既要又要?

使用DOTS的策略

- → Entities
  - — 面向数据编程框架
  - — 提供高效编程范式
  - — 隐藏技术细节
  - — 尽量隐藏并行/并发问题
- → Entities很难?

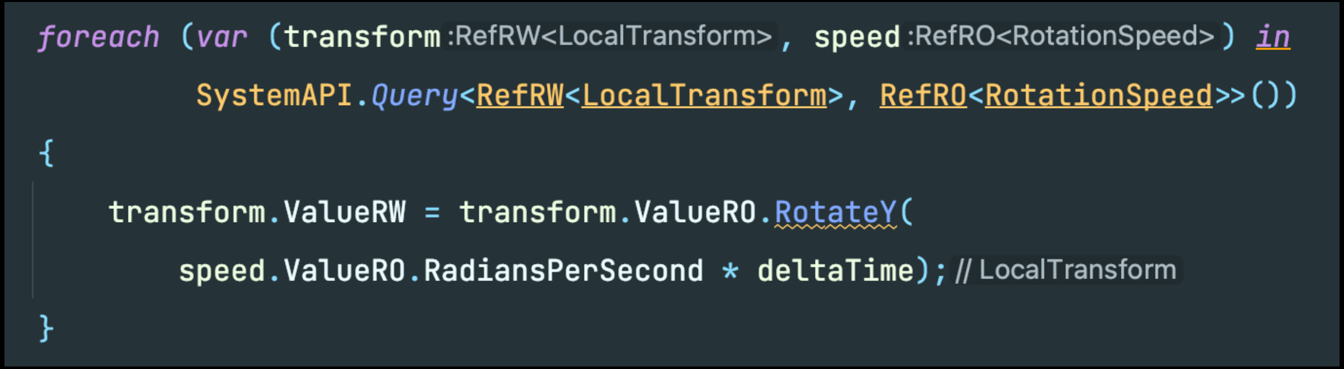


```
[BurstCompile]
partial struct RotateAndScaleJob : IJobEntity
{
    public float deltaTime;
    public float elapsedTime;

    void Execute(ref LocalTransform transform, ref PostTransformMatrix postTransform, in RotationSpeed speed)
    {
        transform = transform.RotateY(speed.RadiansPerSecond * deltaTime);
        postTransform.Value = float4x4.Scale(x: 1, y: math.sin(elapsedTime), z: 1);
    }
}
```

使用DOTS的策略

→ Entities使用难度

- Entities只是和面向对象不同
- 类似于数据库的CRUD操作
- 难度不高于数据库中的CRUD

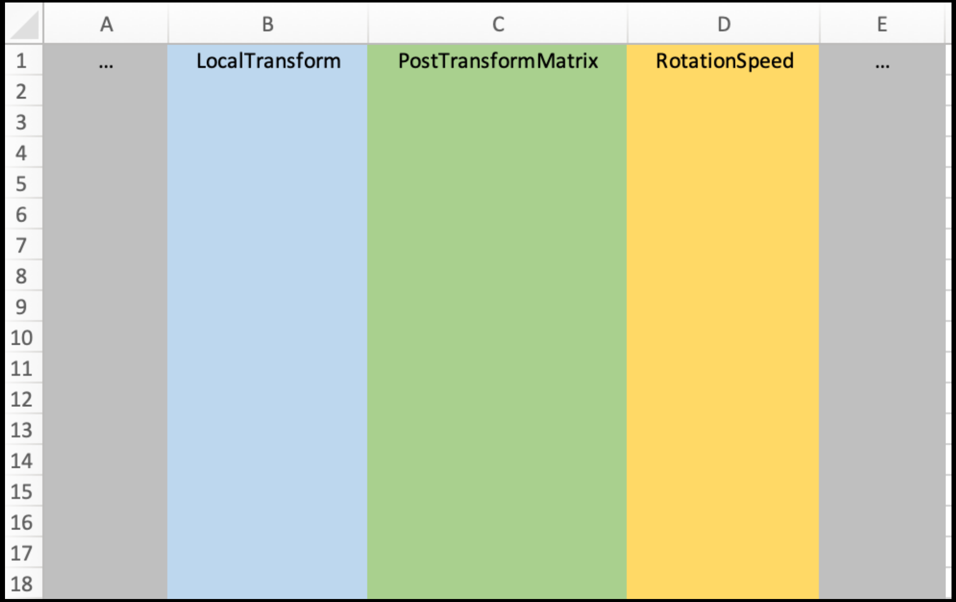


```
foreach (var (transform :RefRW<LocalTransform> , speed :RefRO<RotationSpeed> ) in
         SystemAPI.Query<RefRW<LocalTransform>, RefRO<RotationSpeed>>())
{
    transform.ValueRW = transform.ValueRO.RotateY(
        speed.ValueRO.RadiansPerSecond * deltaTime); // LocalTransform
}
```

Unity®

使用DOTS的策略

- → Entities使用难度
  - — Entities只是和面向对象不同
  - — 类似于数据库的CRUD操作
  - — 难度不高于数据库中的CRUD



	A	B	C	D	E
1	...	LocalTransform	PostTransformMatrix	RotationSpeed	...
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					

Unity®

使用DOTS的策略

→ 为什么大家觉得Entities很难?

— 社区还没有建立起来
- API变动导致很多教程已经过期
- 缺少最佳实践
- 官方也缺少实用案例（已经很大改善）

— 核心组件缺失
- 必须使用hybrid模式带来额外的复杂性
- 相比Mono Behaviour需要造一些轮子

— HPC#在使用上有更多限制

使用DOTS的策略

→ 总结

- 保守做法
  - HPC# + Burst + C# Job System
  - 解决性能痛点
  - 局限于一些复杂度不高/封闭的模块
- 架构演进
  - 充分分析需求确保需要的feature都被支持
  - Hybrid模式
- 切勿激进！

使用Entities的策略

使用Entities的策略

→ 使用Entities常见问题
  - 寻找平替
  - 工作流
  - 开发策略

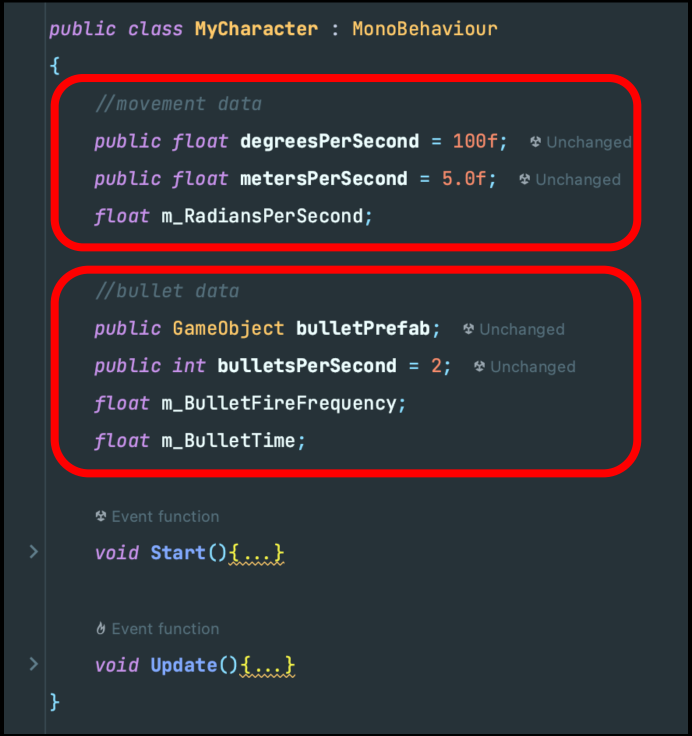
使用Entities的策略

- 平替
  - 面向对象
  - 多态
  - 事件
  - 回调

Unity®

使用Entities的策略

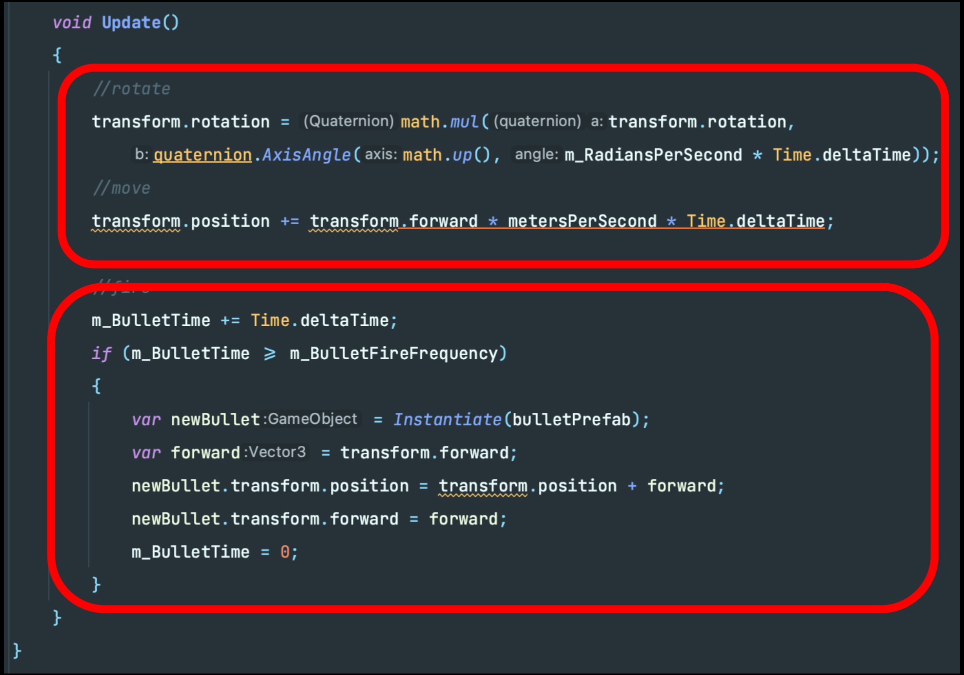
→ 面向对象



```
public class MyCharacter : MonoBehaviour
{
    //movement data
    public float degreesPerSecond = 100f;  Unchanged
    public float metersPerSecond = 5.0f;  Unchanged
    float m_RadiansPerSecond;

    //bullet data
    public GameObject bulletPrefab;  Unchanged
    public int bulletsPerSecond = 2;  Unchanged
    float m_BulletFireFrequency;
    float m_BulletTime;

     Event function
    void Start(){...}

     Event function
    void Update(){...}
}
```

使用Entities的策略

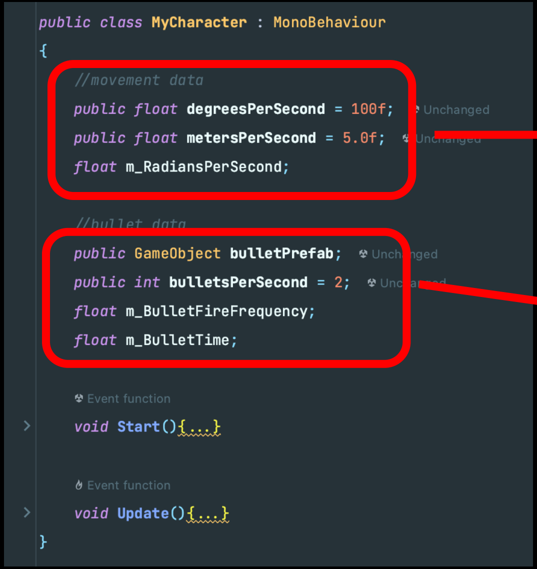
→ 面向对象



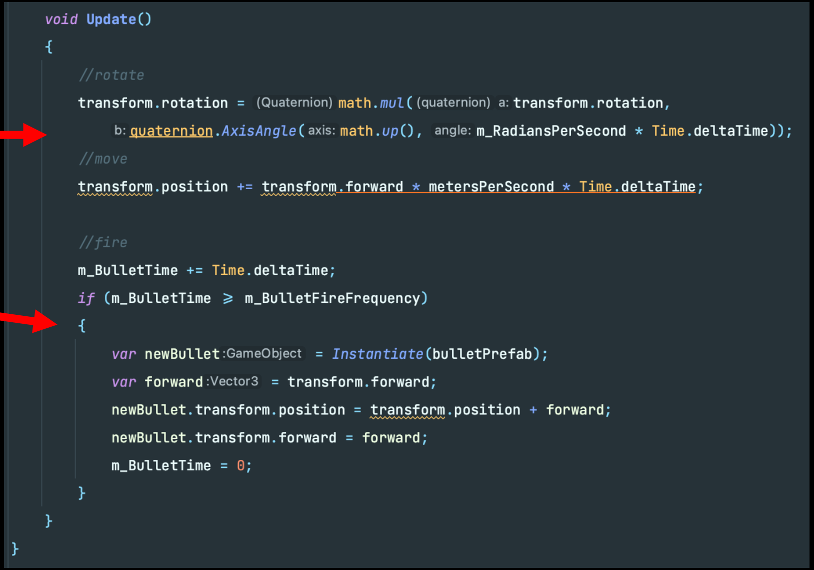
```
    void Update()
    {
        //rotate
        transform.rotation = (Quaternion) math.mul( (quaternion) a: transform.rotation,
            b: quaternion.AxisAngle( axis: math.up(), angle: m_RadiansPerSecond * Time.deltaTime));
        //move
        transform.position += transform.forward * metersPerSecond * Time.deltaTime;

        //fire
        m_BulletTime += Time.deltaTime;
        if (m_BulletTime ≥ m_BulletFireFrequency)
        {
            var newBullet:GameObject = Instantiate(bulletPrefab);
            var forward:Vector3 = transform.forward;
            newBullet.transform.position = transform.position + forward;
            newBullet.transform.forward = forward;
            m_BulletTime = 0;
        }
    }
}
```

使用Entities的策略

→ 面向对象



```
public class MyCharacter : MonoBehaviour
{
    //movement data
    public float degreesPerSecond = 100f;  Unchanged
    public float metersPerSecond = 5.0f;  Unchanged
    float m_RadiansPerSecond;

    //bullet data
    public GameObject bulletPrefab;  Unchanged
    public int bulletsPerSecond = 2;  Unchanged
    float m_BulletFireFrequency;
    float m_BulletTime;

    Event function
    void Start(){...}

    Event function
    void Update(){...}
}
```

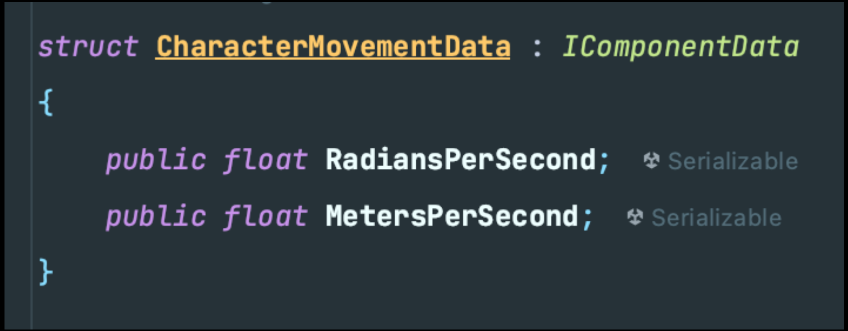


```
    void Update()
    {
        //rotate
        transform.rotation = (Quaternion) math.mul( (quaternion) a: transform.rotation,
            b: quaternion.AxisAngle( axis: math.up(), angle: m_RadiansPerSecond * Time.deltaTime));
        //move
        transform.position += transform.forward * metersPerSecond * Time.deltaTime;

        //fire
        m_BulletTime += Time.deltaTime;
        if (m_BulletTime >= m_BulletFireFrequency)
        {
            var newBullet :GameObject = Instantiate(bulletPrefab);
            var forward :Vector3 = transform.forward;
            newBullet.transform.position = transform.position + forward;
            newBullet.transform.forward = forward;
            m_BulletTime = 0;
        }
    }
}
```

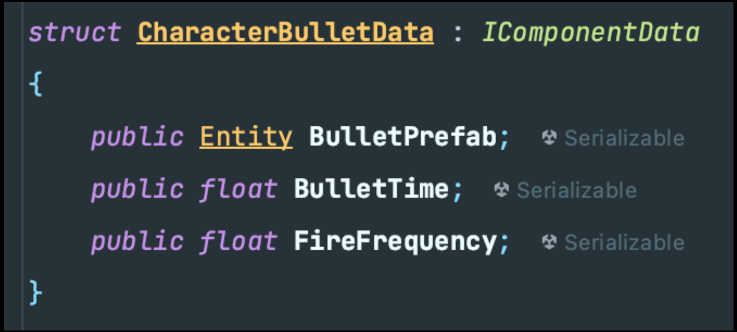
Unity®

使用Entities的策略

→ 面向数据



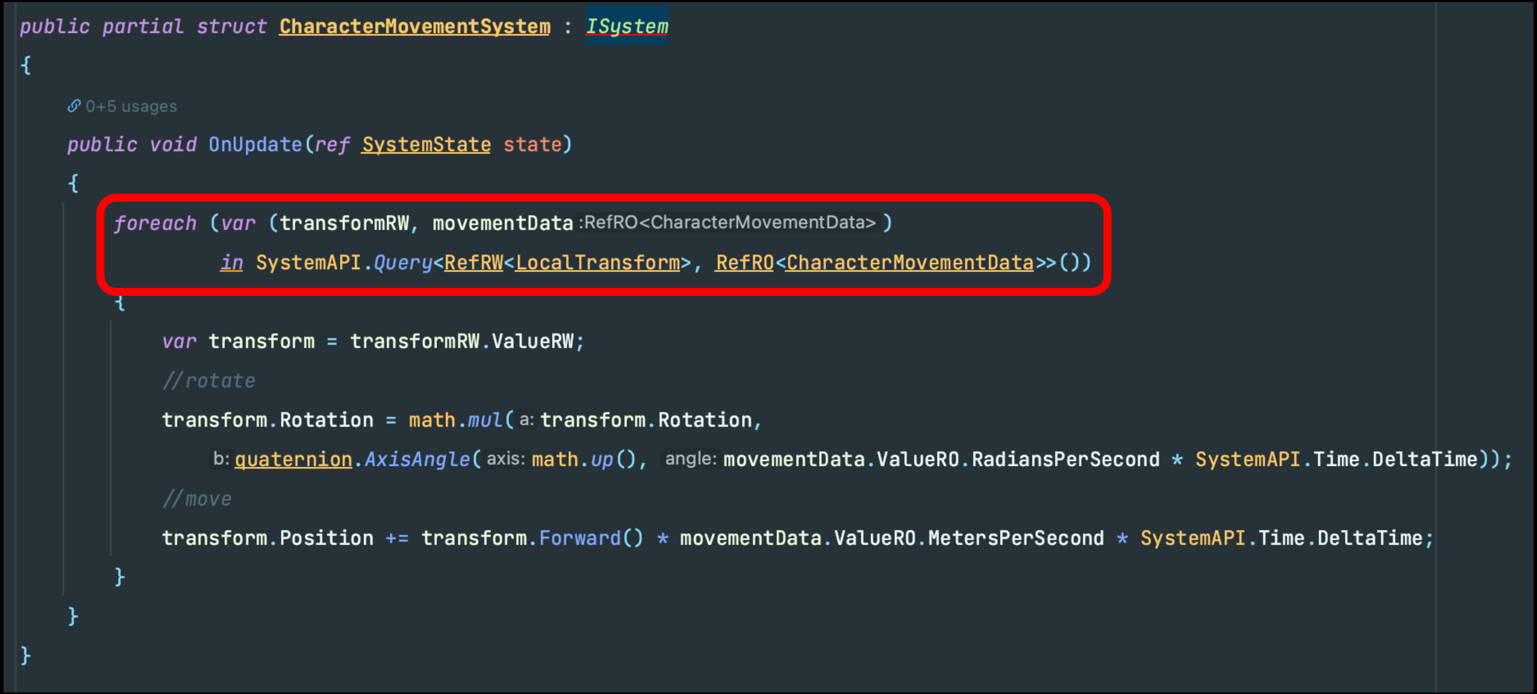
```
struct CharacterMovementData : IComponentData
{
    public float RadiansPerSecond;  Serializable
    public float MetersPerSecond;  Serializable
}
```



```
struct CharacterBulletData : IComponentData
{
    public Entity BulletPrefab;  Serializable
    public float BulletTime;  Serializable
    public float FireFrequency;  Serializable
}
```

使用Entities的策略

→ 面向数据

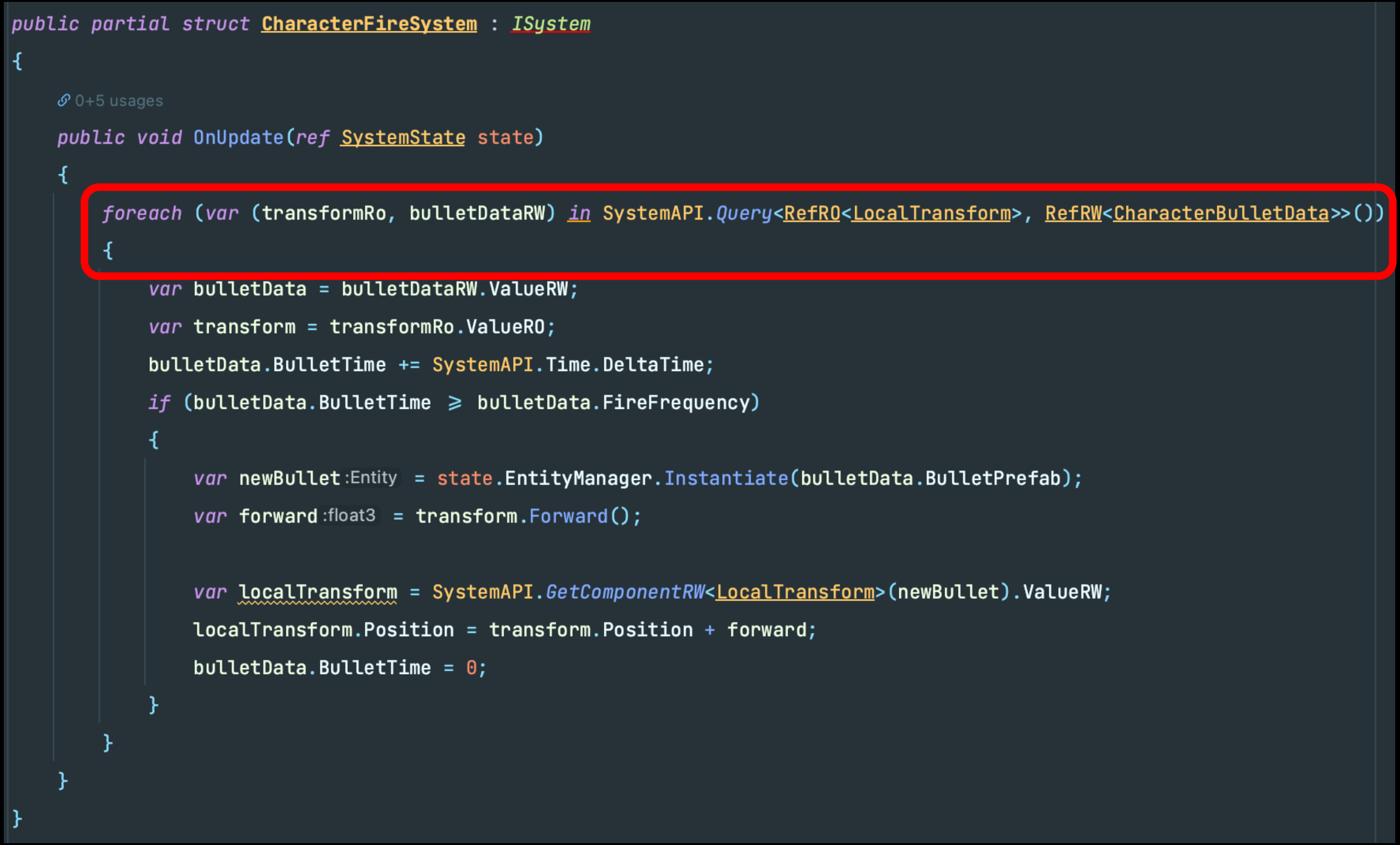


```
public partial struct CharacterMovementSystem : ISystem
{
    0+5 usages
    public void OnUpdate(ref SystemState state)
    {
        foreach (var (transformRW, movementData :RefRO<CharacterMovementData> )
                 in SystemAPI.Query<RefRW<LocalTransform>, RefRO<CharacterMovementData>>())
        {
            var transform = transformRW.ValueRW;
            //rotate
            transform.Rotation = math.mul( a: transform.Rotation,
                b: quaternion.AxisAngle( axis: math.up(),  angle: movementData.ValueRO.RadiansPerSecond * SystemAPI.Time.DeltaTime));
            //move
            transform.Position += transform.Forward() * movementData.ValueRO.MetersPerSecond * SystemAPI.Time.DeltaTime;
        }
    }
}
```

Unity®

使用Entities的策略

→ 面向数据

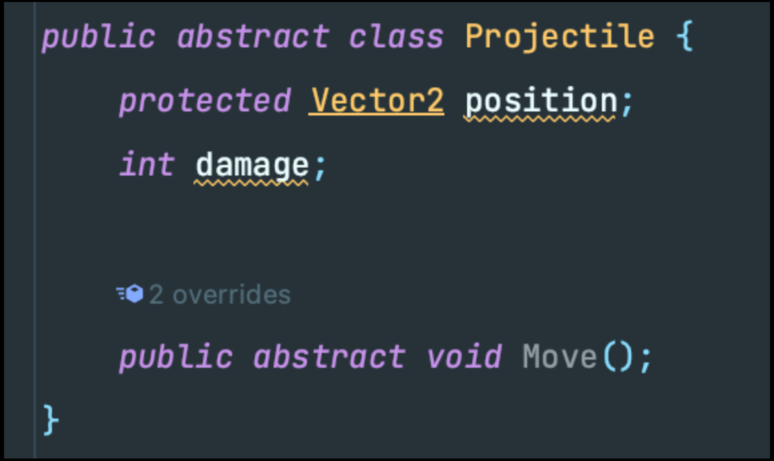


```
public partial struct CharacterFireSystem : ISystem
{
    0+5 usages
    public void OnUpdate(ref SystemState state)
    {
        foreach (var (transformRo, bulletDataRW) in SystemAPI.Query<RefRO<LocalTransform>, RefRW<CharacterBulletData>>())
        {
            var bulletData = bulletDataRW.ValueRW;
            var transform = transformRo.ValueRO;
            bulletData.BulletTime += SystemAPI.Time.DeltaTime;
            if (bulletData.BulletTime ≥ bulletData.FireFrequency)
            {
                var newBullet :Entity = state.EntityManager.Instantiate(bulletData.BulletPrefab);
                var forward :float3 = transform.Forward();

                var localTransform = SystemAPI.GetComponentRW<LocalTransform>(newBullet).ValueRW;
                localTransform.Position = transform.Position + forward;
                bulletData.BulletTime = 0;
            }
        }
    }
}
```

Unity®

使用Entities的策略

→ 多态

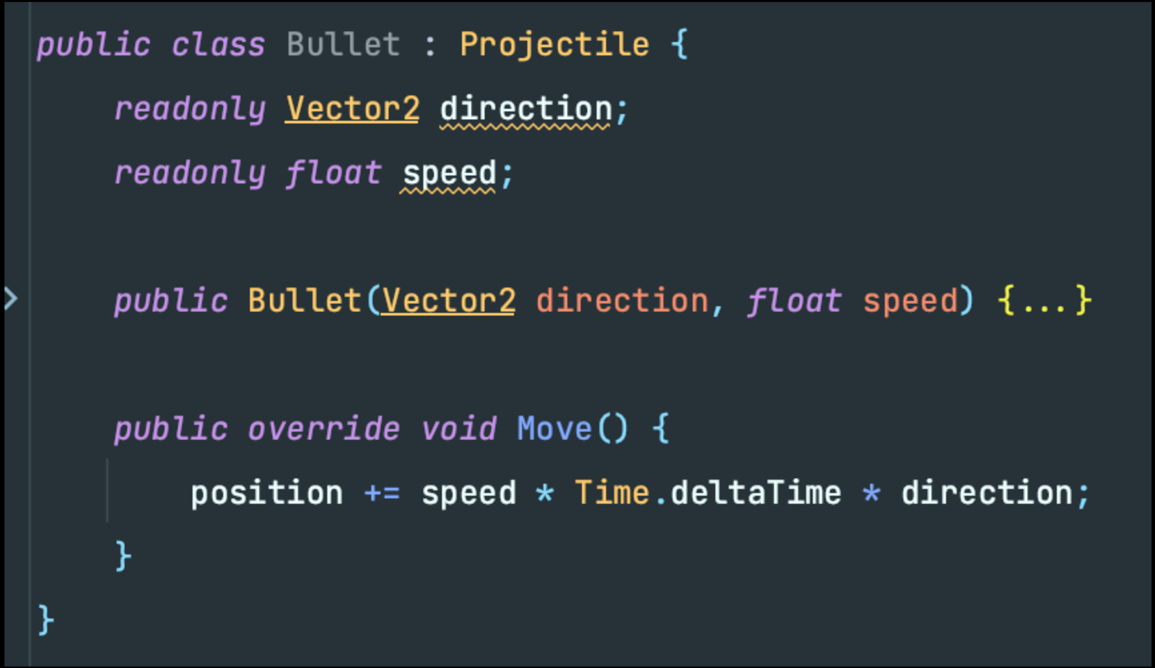


```
public abstract class Projectile {
    protected Vector2 position;
    int damage;

    2 overrides
    public abstract void Move();
}
```

示例来源：https://coffeebraingames.wordpress.com/2019/09/15/replicating-polymorphism-in-ecs/

使用Entities的策略

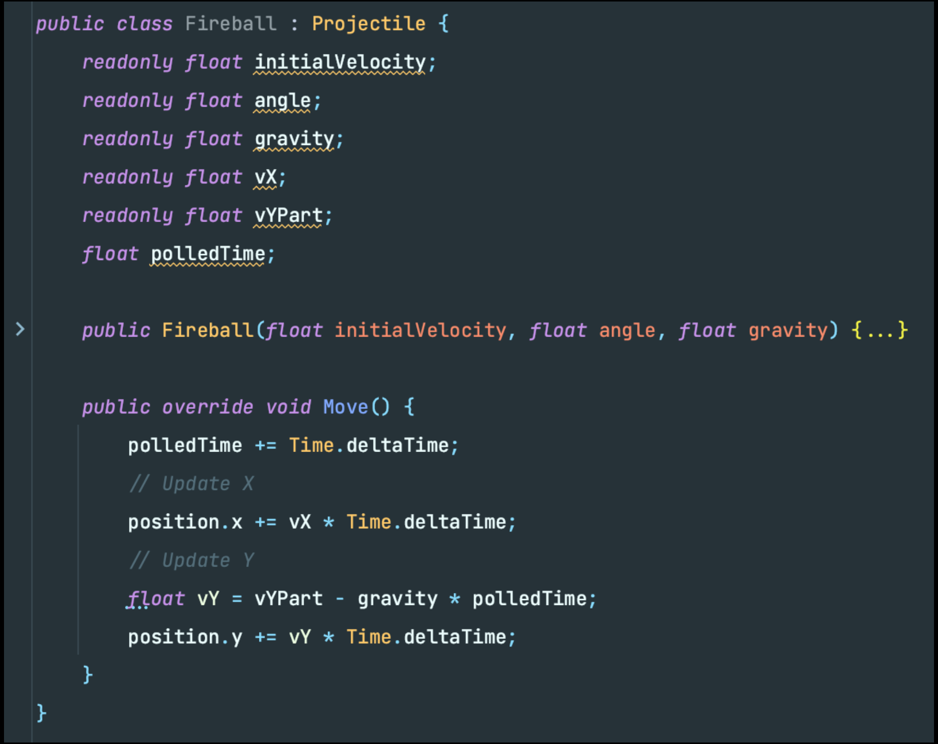
→ 多态



```
public class Bullet : Projectile {
    readonly Vector2 direction;
    readonly float speed;

    public Bullet(Vector2 direction, float speed) {...}

    public override void Move() {
        position += speed * Time.deltaTime * direction;
    }
}
```

使用Entities的策略

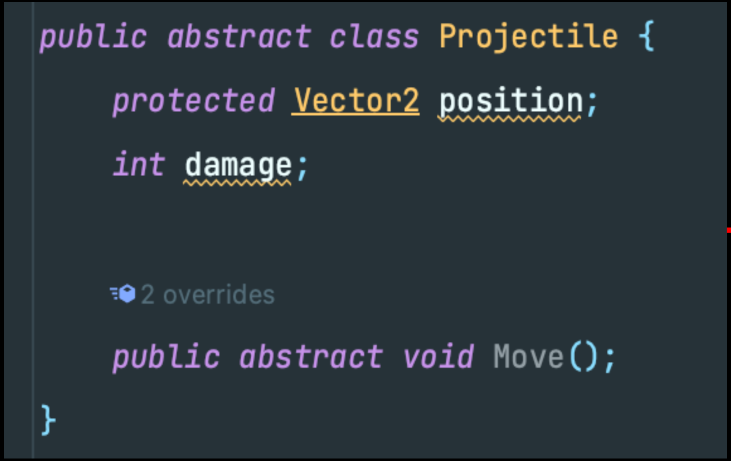
→ 多态



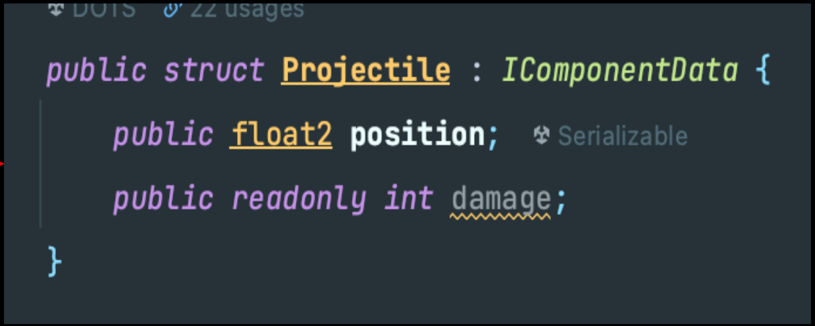
```
public class Fireball : Projectile {
    readonly float initialVelocity;
    readonly float angle;
    readonly float gravity;
    readonly float vX;
    readonly float vYPart;
    float polledTime;

    public Fireball(float initialVelocity, float angle, float gravity) {...}

    public override void Move() {
        polledTime += Time.deltaTime;
        // Update X
        position.x += vX * Time.deltaTime;
        // Update Y
        float vY = vYPart - gravity * polledTime;
        position.y += vY * Time.deltaTime;
    }
}
```

使用Entities的策略

→ 多态ECS版本



```
public abstract class Projectile {
    protected Vector2 position;
    int damage;

    2 overrides
    public abstract void Move();
}
```

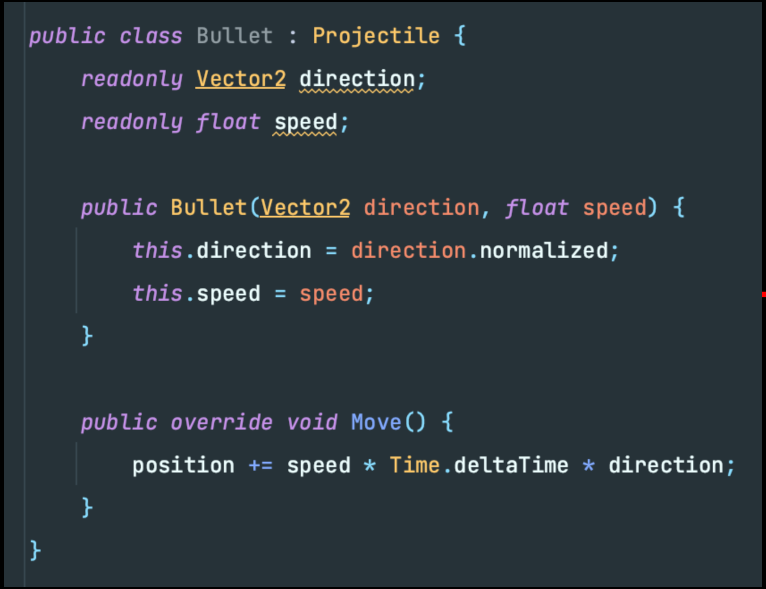


```
DOTS   22 usages
public struct Projectile : IComponentData {
    public float2 position;  Serializable
    public readonly int damage;
}
```

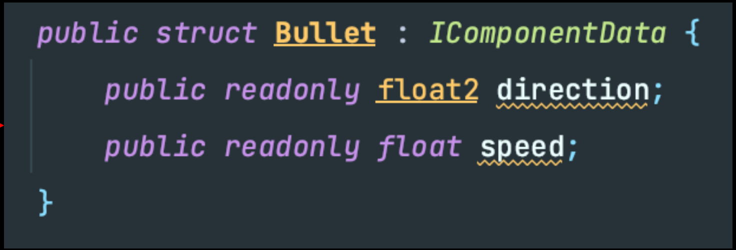
Unity®

使用Entities的策略

→ 多态ECS版本



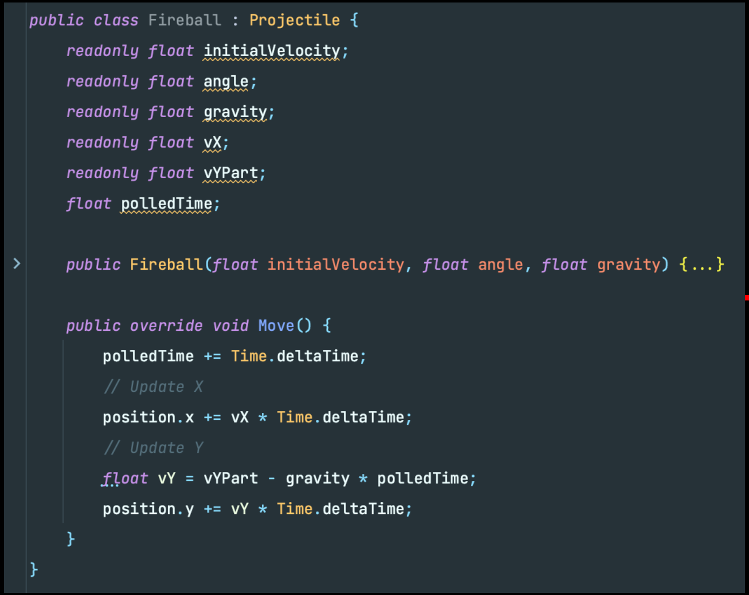
```
public class Bullet : Projectile {
    readonly Vector2 direction;
    readonly float speed;

    public Bullet(Vector2 direction, float speed) {
        this.direction = direction.normalized;
        this.speed = speed;
    }

    public override void Move() {
        position += speed * Time.deltaTime * direction;
    }
}
```



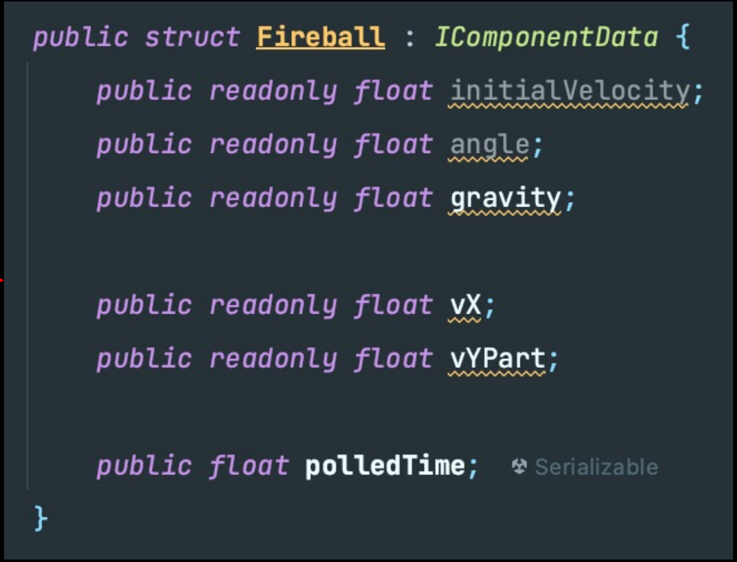
```
public struct Bullet : IComponentData {
    public readonly float2 direction;
    public readonly float speed;
}
```

使用Entities的策略

→ 多态ECS版本



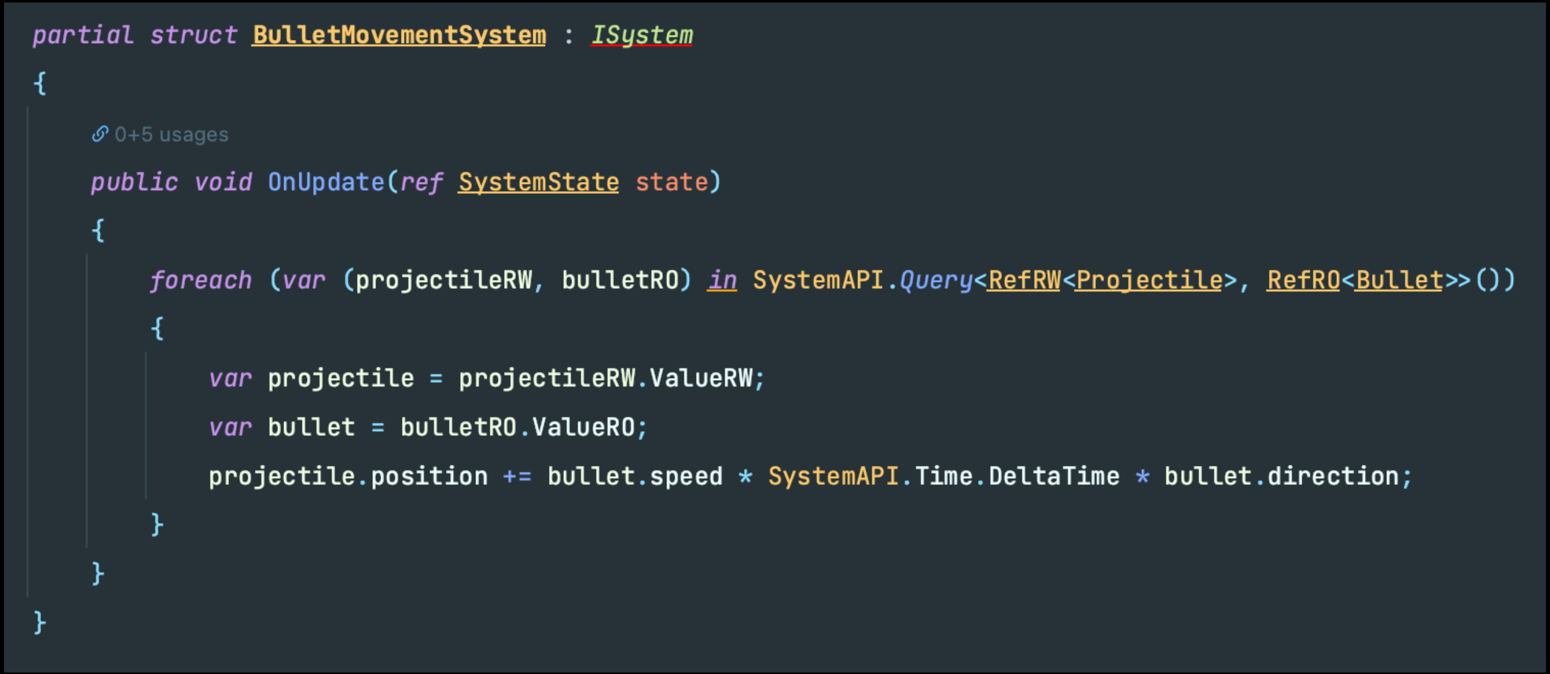
```
public class Fireball : Projectile {
    readonly float initialVelocity;
    readonly float angle;
    readonly float gravity;
    readonly float vX;
    readonly float vYPart;
    float polledTime;

    public Fireball(float initialVelocity, float angle, float gravity) {...}

    public override void Move() {
        polledTime += Time.deltaTime;
        // Update X
        position.x += vX * Time.deltaTime;
        // Update Y
        float vY = vYPart - gravity * polledTime;
        position.y += vY * Time.deltaTime;
    }
}
```



```
public struct Fireball : IComponentData {
    public readonly float initialVelocity;
    public readonly float angle;
    public readonly float gravity;


    public readonly float vX;
    public readonly float vYPart;


    public float polledTime;
}
```

使用Entities的策略

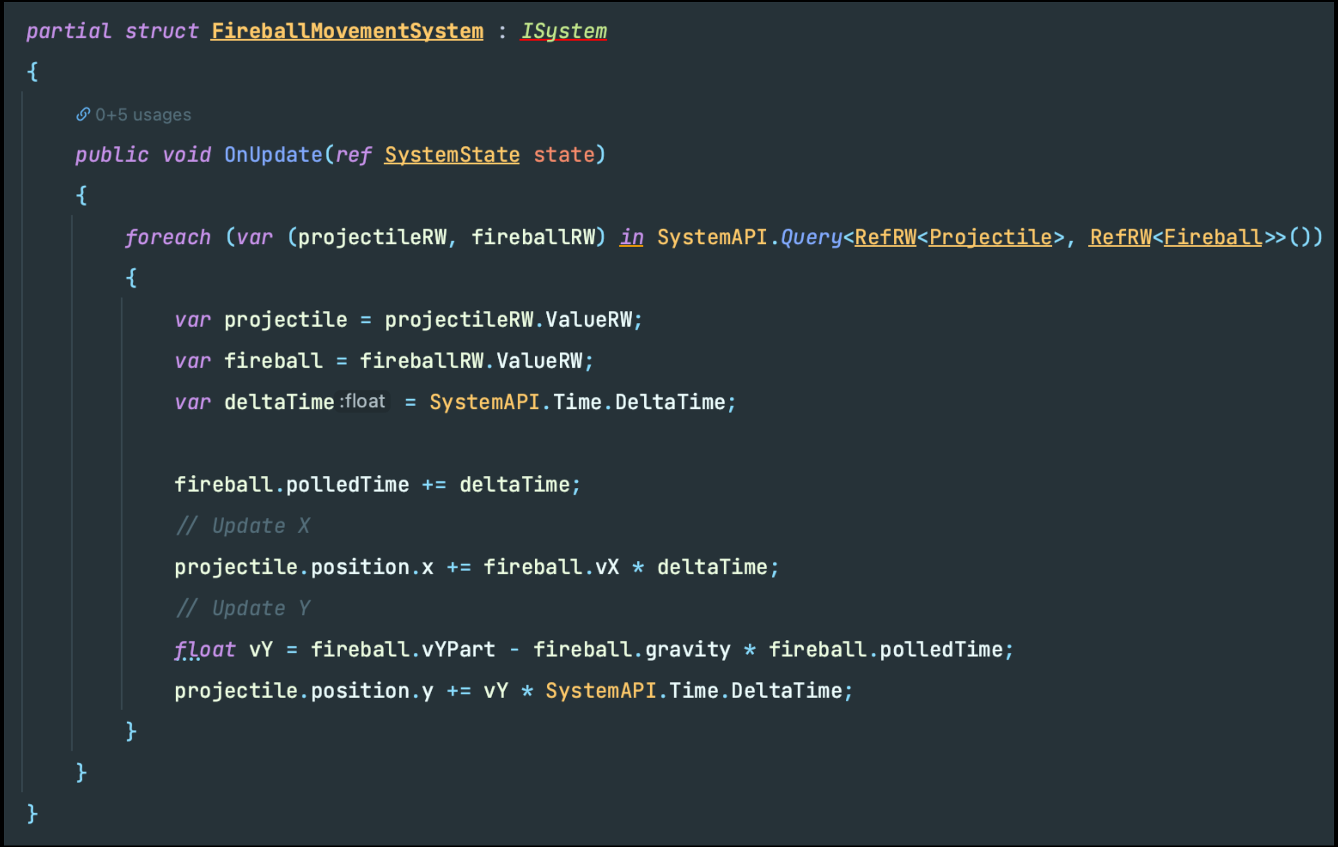
→ 多态ECS版本



```
partial struct BulletMovementSystem : ISystem
{
    0+5 usages
    public void OnUpdate(ref SystemState state)
    {
        foreach (var (projectileRW, bulletRO) in SystemAPI.Query<RefRW<Projectile>, RefRO<Bullet>>())
        {
            var projectile = projectileRW.ValueRW;
            var bullet = bulletRO.ValueRO;
            projectile.position += bullet.speed * SystemAPI.Time.DeltaTime * bullet.direction;
        }
    }
}
```

使用Entities的策略

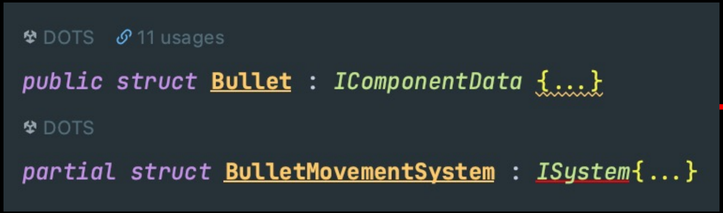
→ 多态ECS版本



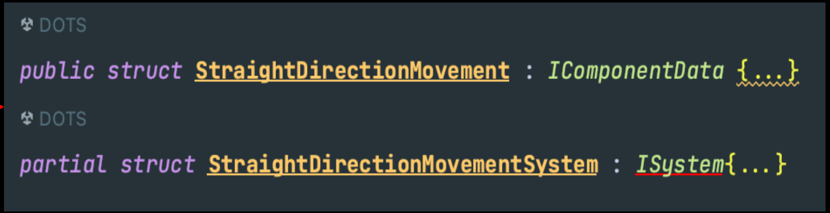
```
partial struct FireballMovementSystem : ISystem
{
    0+5 usages
    public void OnUpdate(ref SystemState state)
    {
        foreach (var (projectileRW, fireballRW) in SystemAPI.Query<RefRW<Projectile>, RefRW<Fireball>>())
        {
            var projectile = projectileRW.ValueRW;
            var fireball = fireballRW.ValueRW;
            var deltaTime :float  = SystemAPI.Time.DeltaTime;

            fireball.polledTime += deltaTime;
            // Update X
            projectile.position.x += fireball.vX * deltaTime;
            // Update Y
            float vY = fireball.vYPart - fireball.gravity * fireball.polledTime;
            projectile.position.y += vY * SystemAPI.Time.DeltaTime;
        }
    }
}
```

使用Entities的策略

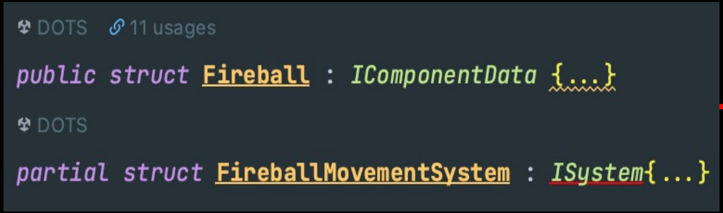
→ 多态ECS版本



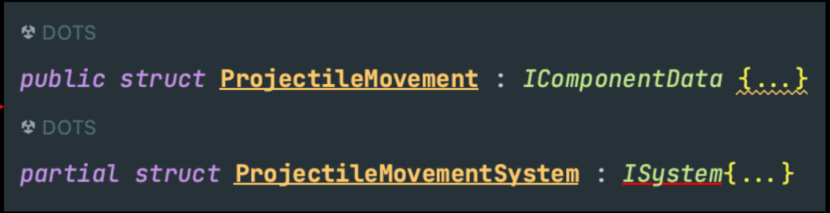
```
DOTS   11 usages
public struct Bullet : IComponentData {...}
DOTS
partial struct BulletMovementSystem : ISystem{...}
```



```
DOTS
public struct StraightDirectionMovement : IComponentData {...}
DOTS
partial struct StraightDirectionMovementSystem : ISystem{...}
```



```
DOTS   11 usages
public struct Fireball : IComponentData {...}
DOTS
partial struct FireballMovementSystem : ISystem{...}
```



```
DOTS
public struct ProjectileMovement : IComponentData {...}
DOTS
partial struct ProjectileMovementSystem : ISystem{...}
```

Unity®

使用Entities的策略

→ 多态ECS版本

— 策略模式

— 功能可以插拔/替换

— 代码复用难度更低

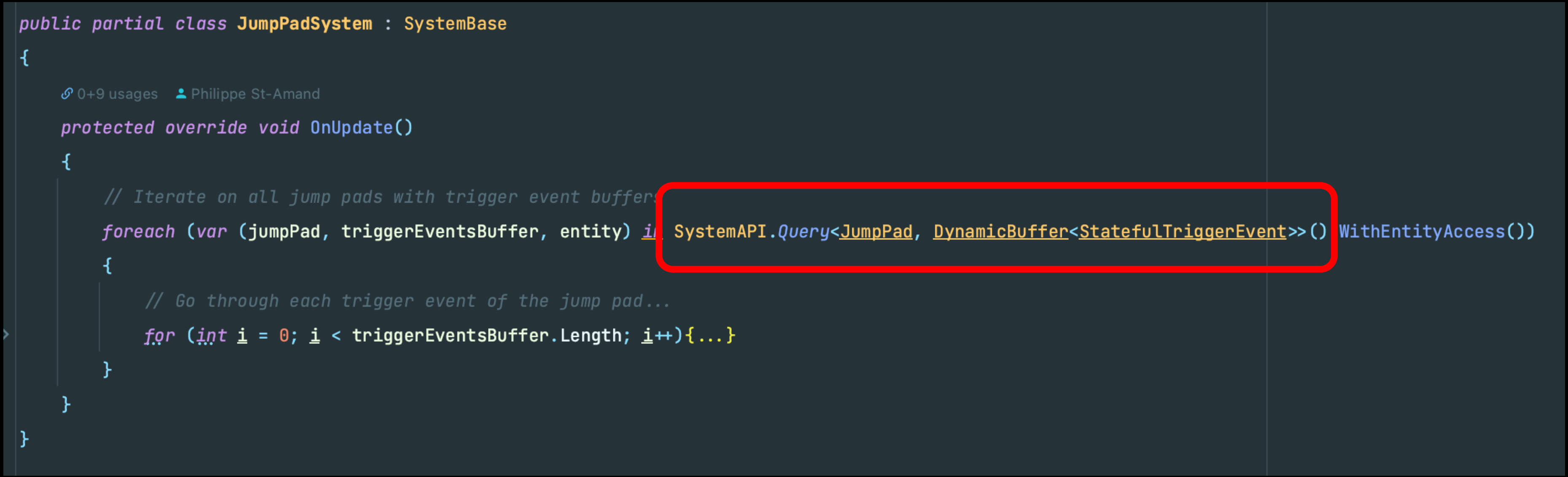
— 满足开闭原则

使用Entities的策略

- 事件/消息/生产者消费者模式
  - Entity
    - 一个事件一个Entity
    - 问题：structural change
  - Dynamic Buffer
  - SharedStatic C#->HPC#通信

使用Entities的策略

→ 事件/消息



```
public partial class JumpPadSystem : SystemBase
{
    0+9 usages   Philippe St-Amand
    protected override void OnUpdate()
    {
        // Iterate on all jump pads with trigger event buffers
        foreach (var (jumpPad, triggerEventsBuffer, entity) in SystemAPI.Query<JumpPad, DynamicBuffer<StatefulTriggerEvent>>() WithEntityAccess())
        {
            // Go through each trigger event of the jump pad...
            for (int i = 0; i < triggerEventsBuffer.Length; i++){...}
        }
    }
}
```

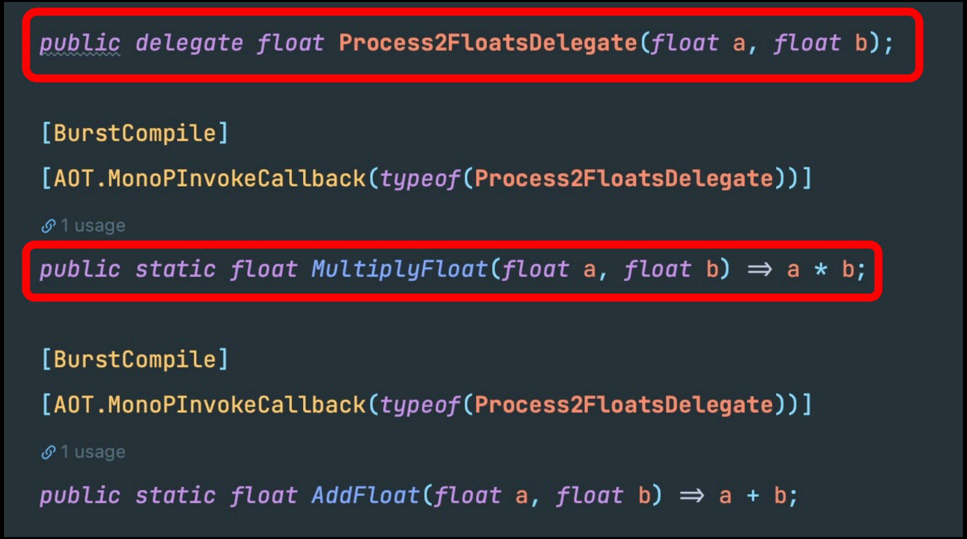
Unity®

使用Entities的策略

- 回调/delegate/监听者模式
  - FunctionPointer<T>(有限制)

使用Entities的策略

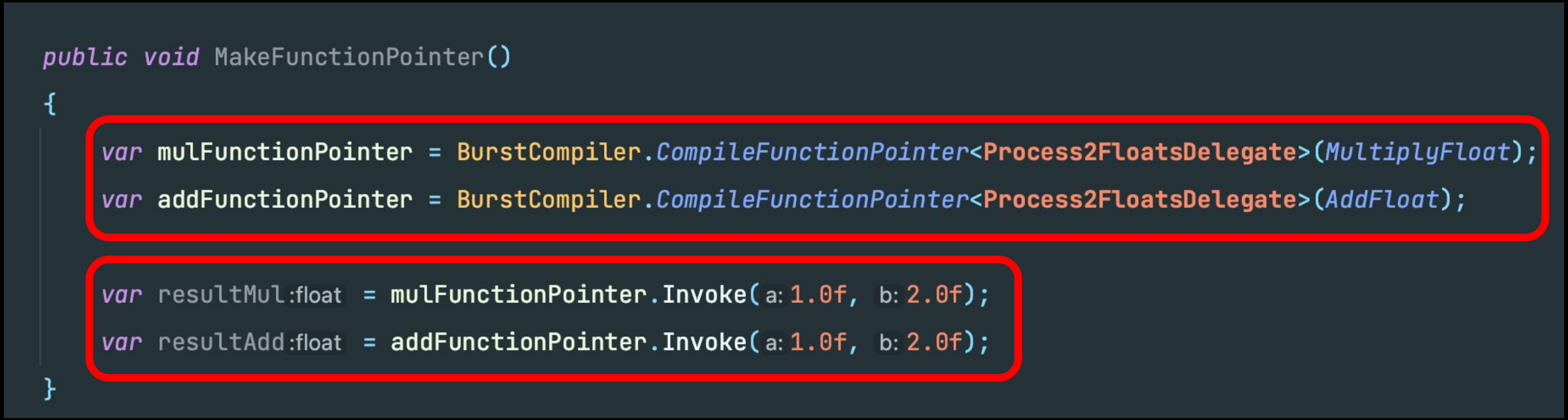
→ FunctionPointer<T>



```
public delegate float Process2FloatsDelegate(float a, float b);

[BurstCompile]
[AOT.MonoPInvokeCallback(typeof(Process2FloatsDelegate))]
1 usage
public static float MultiplyFloat(float a, float b) ⇒ a * b;

[BurstCompile]
[AOT.MonoPInvokeCallback(typeof(Process2FloatsDelegate))]
1 usage
public static float AddFloat(float a, float b) ⇒ a + b;
```

使用Entities的策略

→ FunctionPointer<T>



```
public void MakeFunctionPointer()
{
    var mulFunctionPointer = BurstCompiler.CompileFunctionPointer<Process2FloatsDelegate>(MultiplyFloat);
    var addFunctionPointer = BurstCompiler.CompileFunctionPointer<Process2FloatsDelegate>(AddFloat);

    var resultMul:float  = mulFunctionPointer.Invoke( a: 1.0f,  b: 2.0f);
    var resultAdd:float  = addFunctionPointer.Invoke( a: 1.0f,  b: 2.0f);
}
```

使用Entities的策略

- → 回调/delegate/监听者模式
  - — 尽量使用消息机制代替
  - — FunctionPointer<T>可以一定程度上替代

使用Entities的策略

→ 工作流

— Hybrid工作流

— 设计数据

— System时序

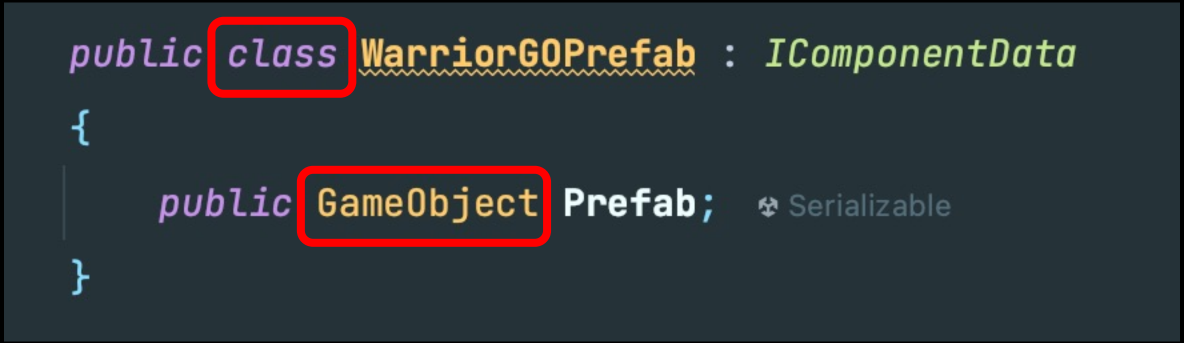
使用Entities的策略

→ Hybrid工作流

- MonoBehaviour和ISystem协同工作
- Input、UI、动画等

使用Entities的策略

→ Hybrid工作流

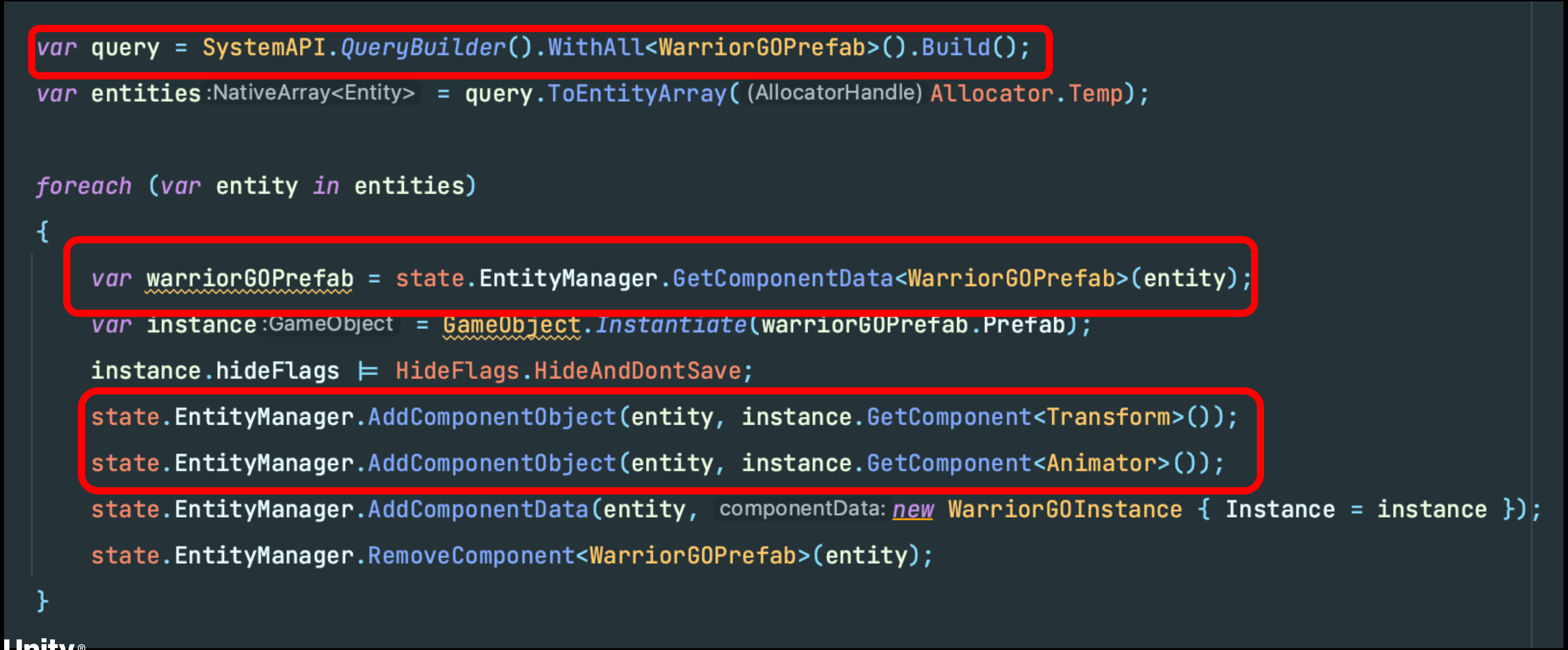


```
public class WarriorGOPrefab : IComponentData
{
    public GameObject Prefab;  Serializable
}
```

案例来源：EntitiesSamples/Assets/Miscellaneous/AnimateGameObject

使用Entities的策略

→ Hybrid工作流

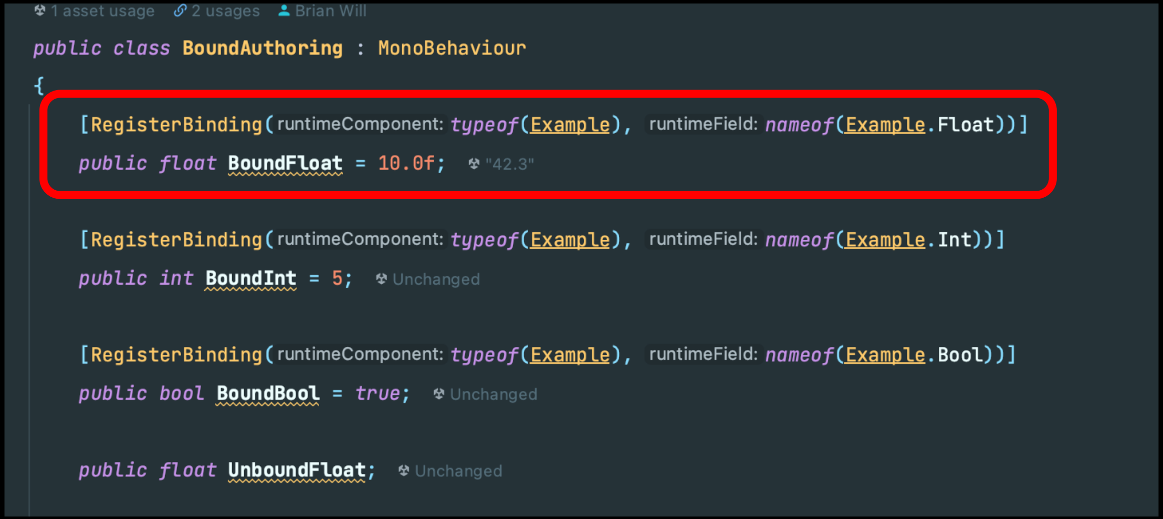


```
var query = SystemAPI.QueryBuilder().WithAll<WarriorGOPrefab>().Build();
var entities :NativeArray<Entity> = query.ToEntityArray( (AllocatorHandle) Allocator.Temp);

foreach (var entity in entities)
{
    var warriorGOPrefab = state.EntityManager.GetComponentData<WarriorGOPrefab>(entity);
    var instance :GameObject = GameObject.Instantiate(warriorGOPrefab.Prefab);
    instance.hideFlags |= HideFlags.HideAndDontSave;
    state.EntityManager.AddComponentObject(entity, instance.GetComponent<Transform>());
    state.EntityManager.AddComponentObject(entity, instance.GetComponent<Animator>());
    state.EntityManager.AddComponentData(entity, componentData: new WarriorGOInstance { Instance = instance });
    state.EntityManager.RemoveComponent<WarriorGOPrefab>(entity);
}
```

使用Entities的策略

→ Hybrid工作流

— BindingRegistry



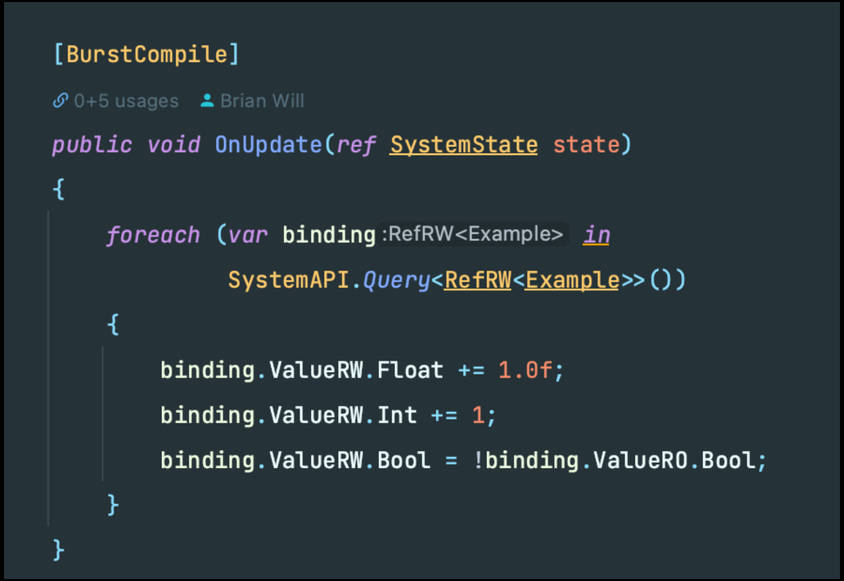
```
1 asset usage   2 usages   Brian Will
public class BoundAuthoring : MonoBehaviour
{
    [RegisterBinding(runtimeComponent: typeof(Example),  runtimeField: nameof(Example.Float))]
    public float BoundFloat = 10.0f;  "42.3"

    [RegisterBinding(runtimeComponent: typeof(Example),  runtimeField: nameof(Example.Int))]
    public int BoundInt = 5;  Unchanged

    [RegisterBinding(runtimeComponent: typeof(Example),  runtimeField: nameof(Example.Bool))]
    public bool BoundBool = true;  Unchanged

    public float UnboundFloat;  Unchanged
```

案例来源：EntitiesSamples/Assets/Streaming/BindingRegistry/BoundAuthoring.cs

使用Entities的策略

→ Hybrid工作流

— BindingRegistry



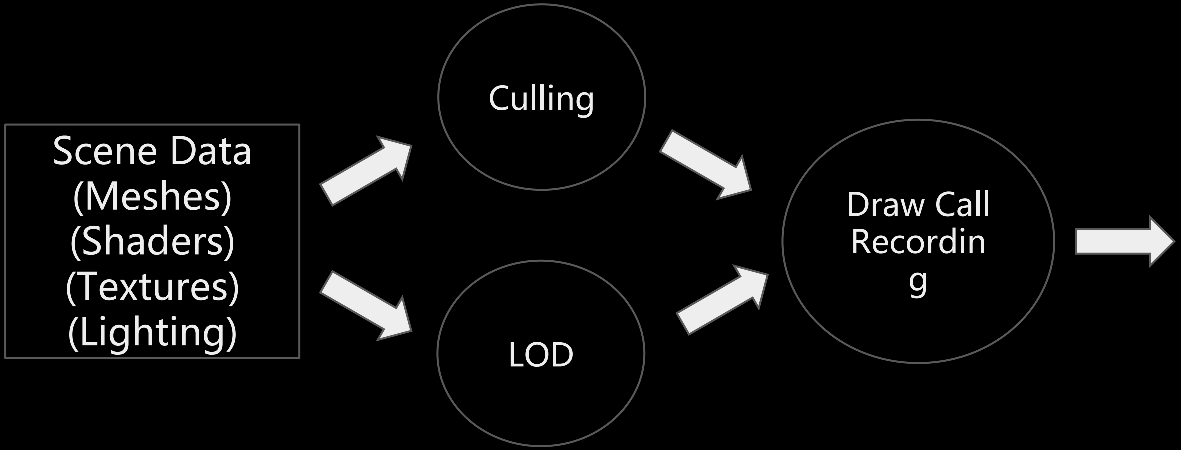
```
[BurstCompile]
0+5 usages   Brian Will
public void OnUpdate(ref SystemState state)
{
    foreach (var binding :RefRW<Example> in
                SystemAPI.Query<RefRW<Example>>())
    {
        binding.ValueRW.Float += 1.0f;
        binding.ValueRW.Int += 1;
        binding.ValueRW.Bool = !binding.ValueRO.Bool;
    }
}
```

Unity®

使用Entities的策略

- 设计数据
  - 如果你不清楚数据的流向，说明需求没有明确
  - 好的数据设计可以帮助更好的解耦

The purpose of all programs, and all parts of those programs, is to transform data from one form to another.



Scene Data
(Meshes)
(Shaders)
(Textures)
(Lighting)

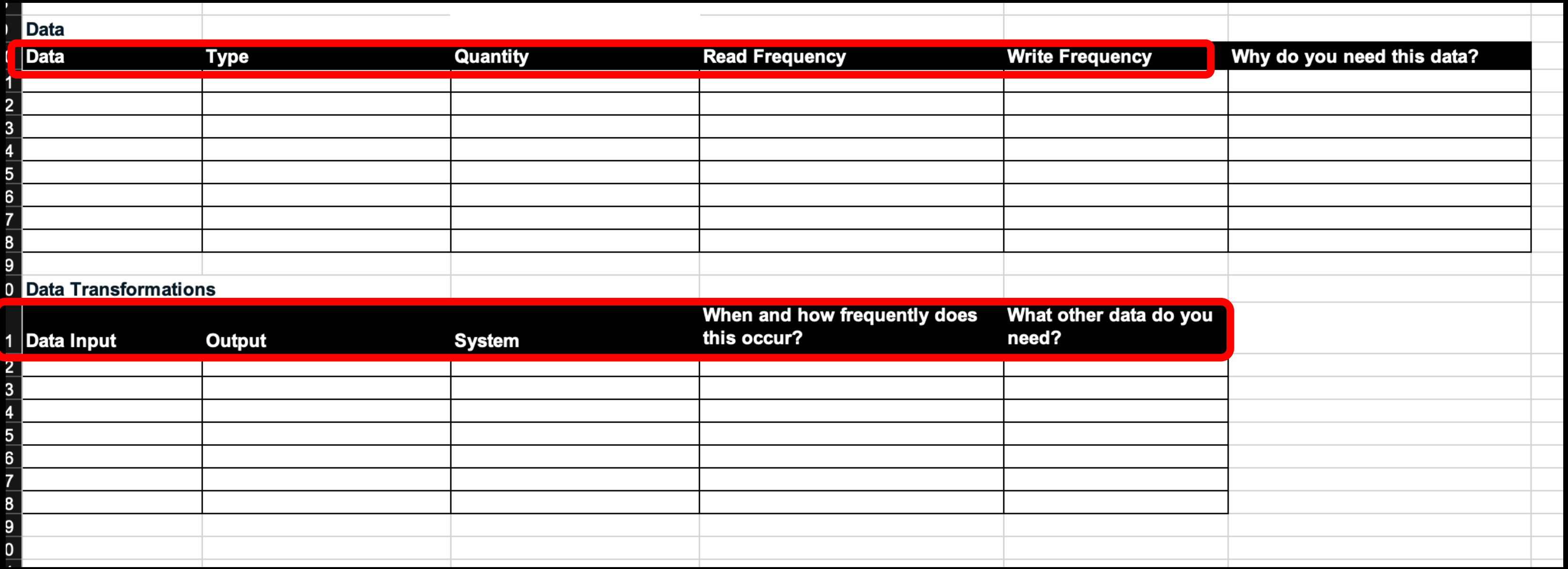
Culling

LOD

Draw Call
Recordin
g

使用Entities的策略

→ 如何设计数据转换?



Data

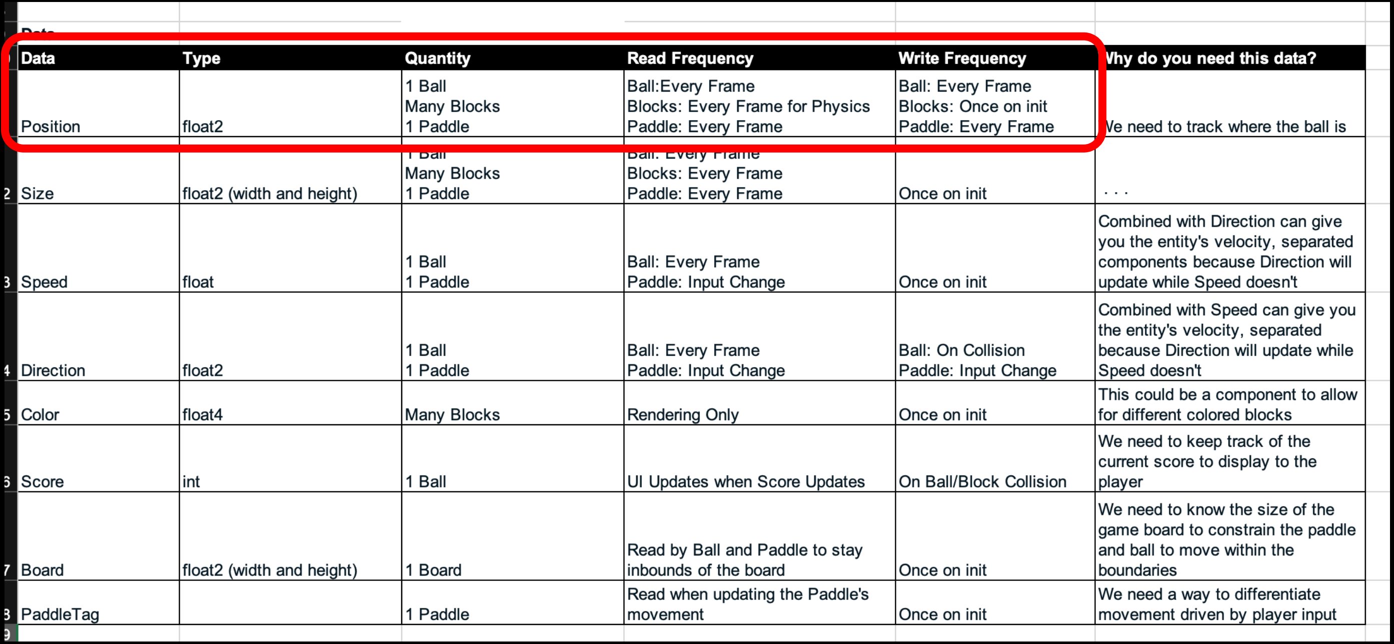
Data	Type	Quantity	Read Frequency	Write Frequency	Why do you need this data?

Data Transformations

Data Input	Output	System	When and how frequently does this occur?	What other data do you need?

使用Entities的策略

→ 如何设计数据转换?

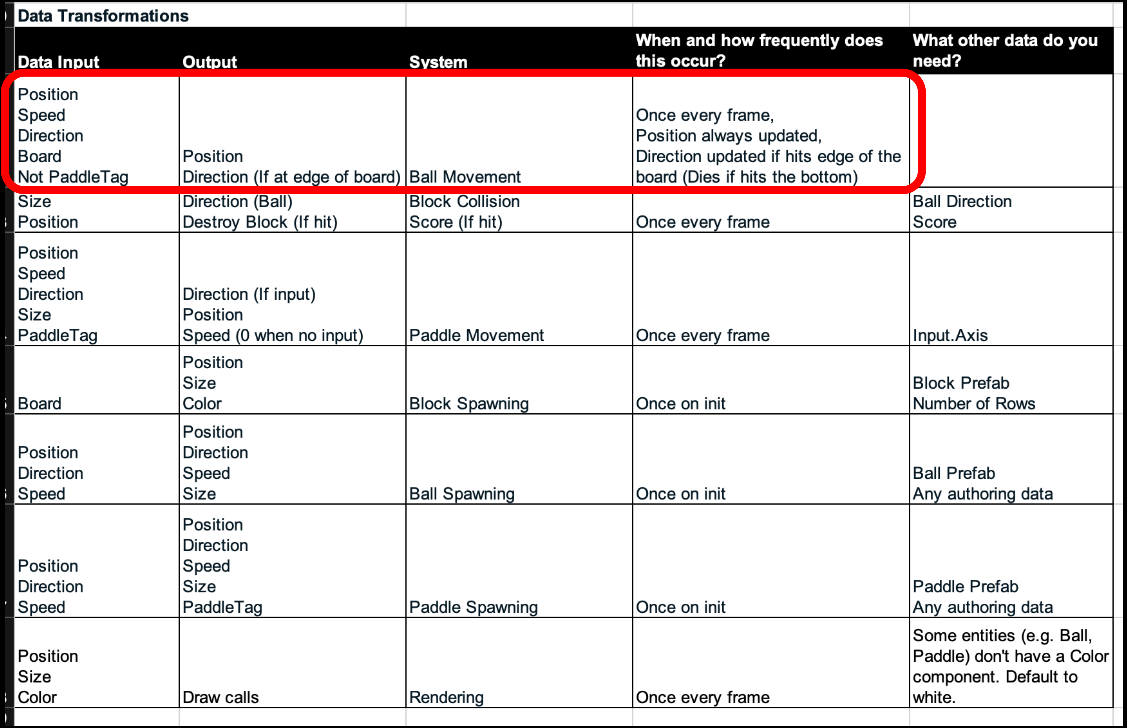


Data

Data	Type	Quantity	Read Frequency	Write Frequency	Why do you need this data?
Position	float2	1 Ball Many Blocks 1 Paddle	Ball:Every Frame Blocks: Every Frame for Physics Paddle: Every Frame	Ball: Every Frame Blocks: Once on init Paddle: Every Frame	We need to track where the ball is
Size	float2 (width and height)	1 Ball Many Blocks 1 Paddle	Ball: Every Frame Blocks: Every Frame Paddle: Every Frame	Once on init	. . .
Speed	float	1 Ball 1 Paddle	Ball: Every Frame Paddle: Input Change	Once on init	Combined with Direction can give you the entity's velocity, separated components because Direction will update while Speed doesn't
Direction	float2	1 Ball 1 Paddle	Ball: Every Frame Paddle: Input Change	Ball: On Collision Paddle: Input Change	Combined with Speed can give you the entity's velocity, separated because Direction will update while Speed doesn't
Color	float4	Many Blocks	Rendering Only	Once on init	This could be a component to allow for different colored blocks
Score	int	1 Ball	UI Updates when Score Updates	On Ball/Block Collision	We need to keep track of the current score to display to the player
Board	float2 (width and height)	1 Board	Read by Ball and Paddle to stay inbounds of the board	Once on init	We need to know the size of the game board to constrain the paddle and ball to move within the boundaries
PaddleTag		1 Paddle	Read when updating the Paddle's movement	Once on init	We need a way to differentiate movement driven by player input

使用Entities的策略

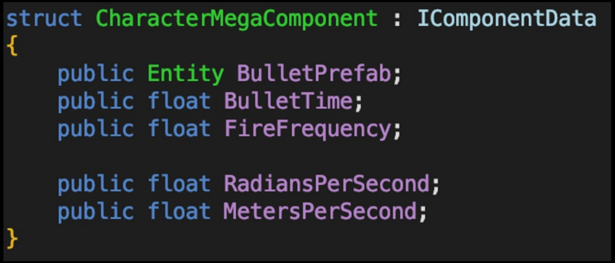
→ 如何设计数据转换?



Data Transformations

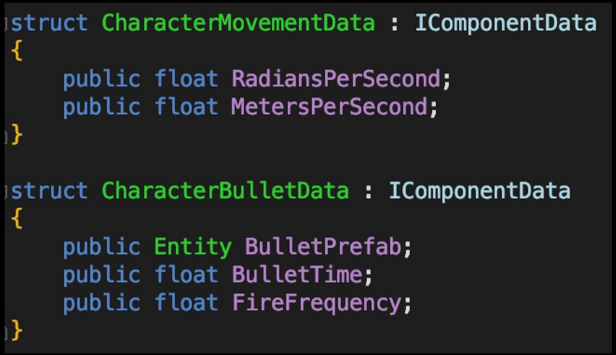
Data Input	Output	System	When and how frequently does this occur?	What other data do you need?
Position Speed Direction Board Not PaddleTag	Position Direction (If at edge of board)	Ball Movement	Once every frame, Position always updated, Direction updated if hits edge of the board (Dies if hits the bottom)	
Size Position	Direction (Ball) Destroy Block (If hit)	Block Collision Score (If hit)	Once every frame	Ball Direction Score
Position Speed Direction Size PaddleTag	Direction (If input) Position Speed (0 when no input)	Paddle Movement	Once every frame	Input.Axis
Board	Position Size Color	Block Spawning	Once on init	Block Prefab Number of Rows
Position Direction Speed	Position Direction Speed Size	Ball Spawning	Once on init	Ball Prefab Any authoring data
Position Direction Speed	Position Direction Speed Size PaddleTag	Paddle Spawning	Once on init	Paddle Prefab Any authoring data
Position Size Color	Draw calls	Rendering	Once every frame	Some entities (e.g. Ball, Paddle) don't have a Color component. Default to white.

MegaComponent



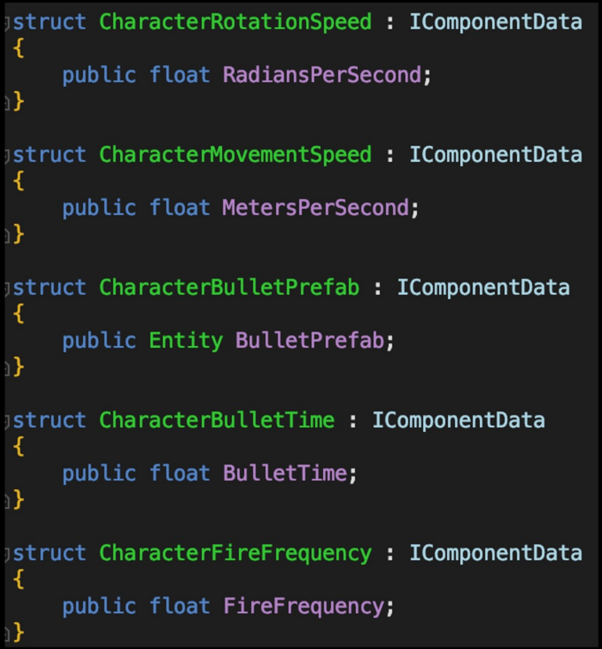
```
struct CharacterMegaComponent : IComponentData
{
    public Entity BulletPrefab;
    public float BulletTime;
    public float FireFrequency;

    public float RadiansPerSecond;
    public float MetersPerSecond;
}
```

Few Components



```
struct CharacterMovementData : IComponentData
{
    public float RadiansPerSecond;
    public float MetersPerSecond;
}

struct CharacterBulletData : IComponentData
{
    public Entity BulletPrefab;
    public float BulletTime;
    public float FireFrequency;
}
```

Many Components



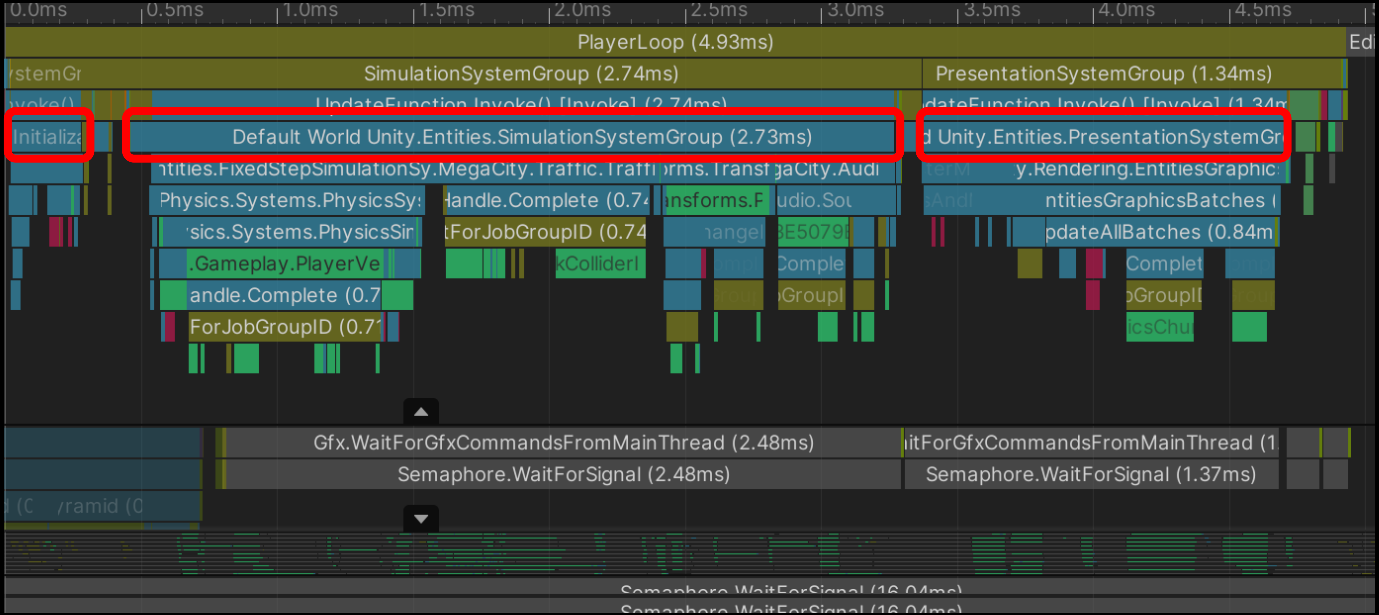
```
struct CharacterRotationSpeed : IComponentData
{
    public float RadiansPerSecond;
}

struct CharacterMovementSpeed : IComponentData
{
    public float MetersPerSecond;
}

struct CharacterBulletPrefab : IComponentData
{
    public Entity BulletPrefab;
}

struct CharacterBulletTime : IComponentData
{
    public float BulletTime;
}

struct CharacterFireFrequency : IComponentData
{
    public float FireFrequency;
}
```

50k Character Entities

- MegaComponent: 4.09ms
- Few, larger components: 3.77ms
- Many, small components: 4.10ms

使用Entities的策略

→ System时序


0.0ms
0.5ms
1.0ms
1.5ms
2.0ms
2.5ms
3.0ms
3.5ms
4.0ms
4.5ms
PlayerLoop (4.93ms)
SimulationSystemGroup (2.74ms)
PresentationSystemGroup (1.34ms)
UpdateFunction.Invoke() [Invoke] (2.74ms)
dateFunction.Invoke() [Invoke] (1.34m
Initializa
Default World Unity.Entities.SimulationSystemGroup (2.73ms)
d Unity.Entities.PresentationSystemGr
ntities.FixedStepSimulationSy.MegaCity.Traffic.Traffi
rms.Transf
gaCity.Audi
y.Rendering.EntitiesGraphic
Physics.Systems.PhysicsSy
Handle.Complete (0.74
nsforms.F
udio.Sou
ntitiesGraphicsBatches
sics.Systems.PhysicsSir
tForJobGroupID (0.74
BE5079
pdateAllBatches (0.84m
.Gameplay.PlayerVe
kColliderI
Comple
Complet
andle.Complete (0.7
Group
GroupI
ForJobGroupID (0.7
icsChu
Gfx.WaitForGfxCommandsFromMainThread (2.48ms)
itForGfxCommandsFromMainThread (1
Semaphore.WaitForSignal (2.48ms)
Semaphore.WaitForSignal (1.37ms)
Semaphore.WaitForSignal (16.04ms)
Semaphore.WaitForSignal (16.04ms)

Unity®

使用Entities的策略

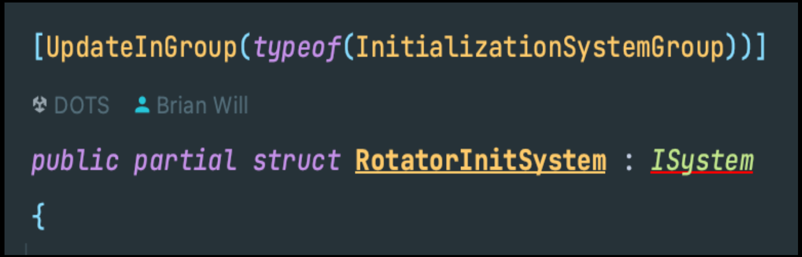
- System时序
  - 业务对先后顺序有要求
  - Safety Checks机制要求读写不能同时进行
  - 并行度要求

使用Entities的策略

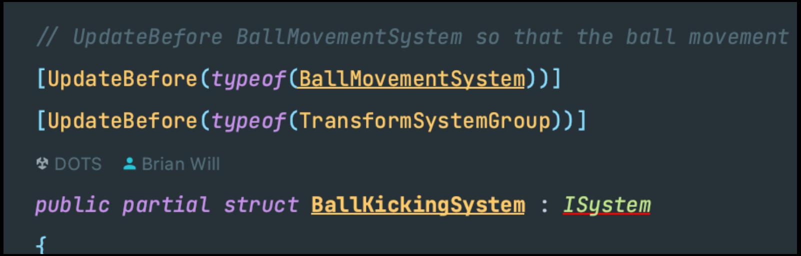
→ System时序

— 业务上自定义SystemGroup

— SystemGroup内部使用[UpdateBefore] [UpdateAfter]进行细节调整



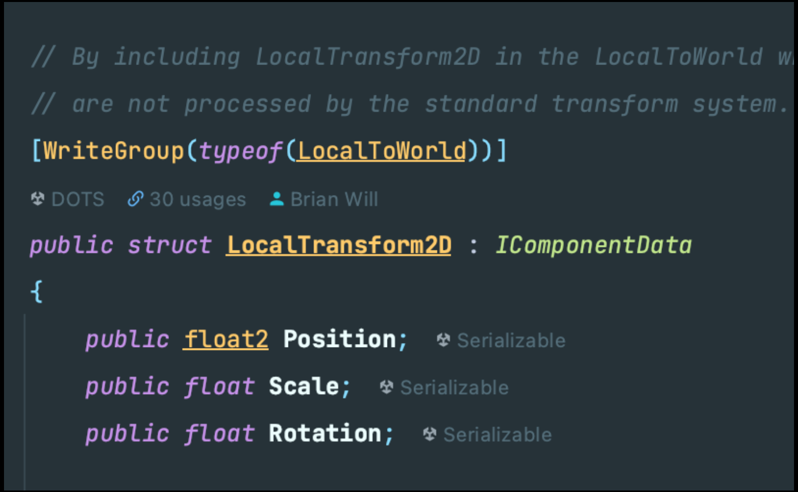
```
[UpdateInGroup(typeof(InitializationSystemGroup))]
DOTS  Brian Will
public partial struct RotatorInitSystem : ISystem
{
```



```
// UpdateBefore BallMovementSystem so that the ball movement
[UpdateBefore(typeof(BallMovementSystem))]
[UpdateBefore(typeof(TransformSystemGroup))]
DOTS  Brian Will
public partial struct BallKickingSystem : ISystem
{
```

使用Entities的策略

- System时序
  - WriteGroup
  - 覆盖默认System实现



```
// By including LocalTransform2D in the LocalToWorld w
// are not processed by the standard transform system.
[WriteGroup(typeof(LocalToWorld))]
DOTS  30 usages  Brian Will
public struct LocalTransform2D : IComponentData
{
    public float2 Position;  Serializable
    public float Scale;  Serializable
    public float Rotation;  Serializable
```

案例来源：EntitiesSamples/Assets/Miscellaneous/CustomTransforms

使用Entities的策略

→ 开发策略

- 使用SystemAPI的单线程版本
- 开启Burst
- 找到性能痛点（2-8原则）
- 性能调优
  - 用新的System替代掉旧的System
  - 并行化
  - 修改数据协议减少structural change

使用Entities的策略

→ 总结

- 确认项目需要的feature被Entities支持
- Hybrid工作流
- 先串行后并行

使用Entities的策略

- 总结
  - 确认项目需要的feature被Entities支持
  - Hybrid工作流
  - 先串行后并行

使用Entities的策略

→ 总结

- 确认项目需要的feature被Entities支持
- Hybrid工作流
- 先串行后并行

使用Entities的策略

→ 总结

- 确认项目需要的feature被Entities支持
- Hybrid工作流
- 先串行后并行

使用Entities的策略

→ 总结

- 确认项目需要的feature被Entities支持
- Hybrid工作流
- 先串行后并行