团结1.1版本小游戏新功能&优化

2023

1.1新增

- → DotnetWasm方案
- → GPU Skinning
- → Shader Warmup
- → Managed Code Stripping - Extreme Level
- → IL2CPP元数据精简
- → 内存分配器优化
- → TextureManager
- → 小游戏宿主（Android）
- → C# Debugging
- → Remapper运行时内存优化
- → Math库支持Wasm SIMD
- → 深度集成微信小游戏SDK

未来...

- 轻量化物理引擎Bullet
- 引擎公共包，加速下载启动
- 渲染线程支持，降低发热
- 在引擎侧充分利用宿主新能力，如：
  - shader binary加载
  - render api扩展
  - 重度资产在宿主侧处理，wasm侧仅发送cmd
- 尝试gpu driven管线
- 从TextureManager逐步走向更多类型的资产打包管理
- 对多线程更好的支持

Unity®

Dotnet Wasm方案

→ IL2CPP方案痛点

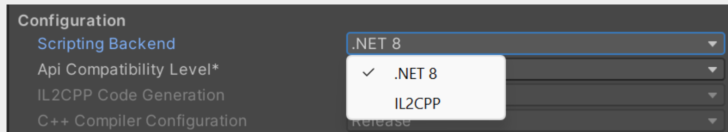
– Wasm

- IL转换为Cpp，编译链接进一个All-in-one的Wasm
- 浏览器加载执行Wasm时，代码编译、运行时指令优化、JIT优化等等，会消耗Wasm体积数倍的内存
- 例如，Wasm体积**50MB**，仅Wasm相关的内存开销就可能多达**500MB**

Dotnet Wasm方案

→ **.Net 8**

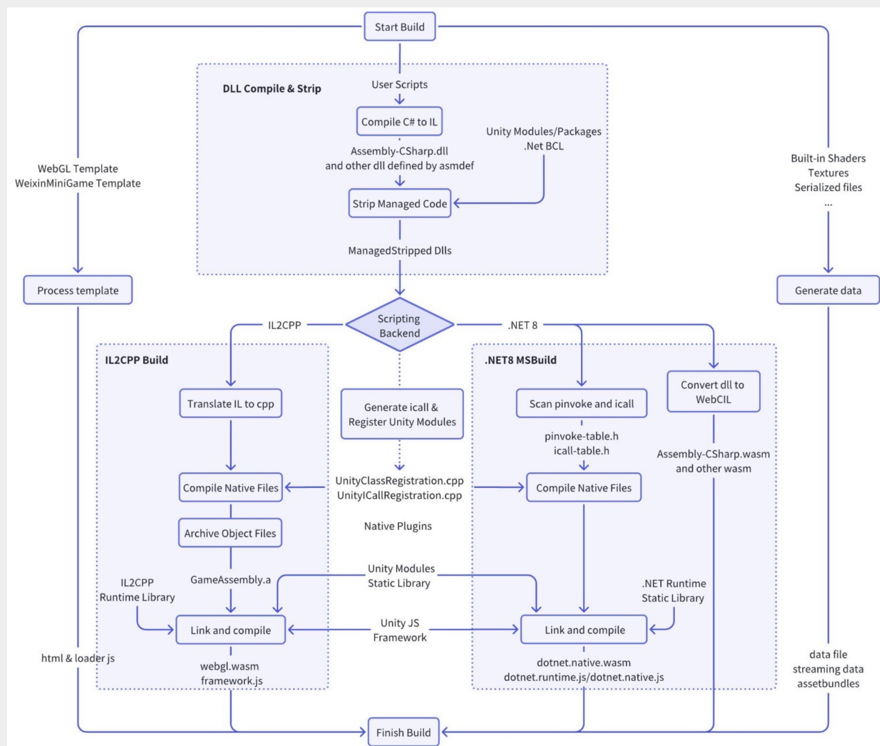
- 2023微软最新发布
  - 稳定支持WebAssembly
- 解释执行IL
  - 将Dll部分从Wasm中剥离
  - 减小WASM体积，显著降低运行内存
- .NET生态
  - AOT，JITerpreter，SGen等新技术
  - 性能提升



Dotnet Wasm方案

→ 构建环节

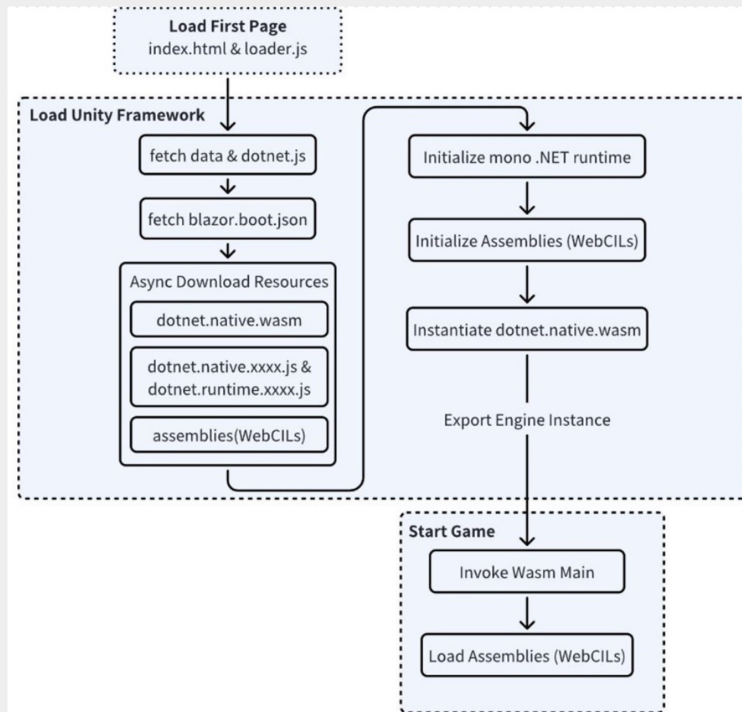
- DLL Compile & Strip
  - 与IL2CPP一致
- Convert DLL
  - In：DLL
  - Out：WebCIL格式的Wasm文件
- Native code Compile & Link
  - In：Native代码 + 引擎 + .NET Runtime
  - Out：dotnet.native.wasm + js胶水代码



Dotnet Wasm方案

→ 启动流程

- 加载引导文件：
  - .Net的data和js + blazor.boot.json
- 下载代码资源：
  - dotnet.native.wasm + WebCIL文件（从CDN）
- 初始化
  - 初始化.NET Runtime
  - 以Buffer形式加载WebCIL文件
  - 加载dotnet.native.wasm
- 进入游戏
  - CallMain
  - 按需加载WebCIL，执行其中方法



Dotnet Wasm方案

→ **SGen GC**

- 专为Mono设计
  - 精确扫描Mono Stack
  - 运行时搬移整理内存碎片
  - 分代回收，回收行为可以分散在每一帧，减少卡顿
- 有一些内存Overhead
  - 预分配40~50MB内存用于维护GC相关的数据结构，同时可能按需上浮
  - 内存比较紧张的项目仍然可以使用Boehm GC

Dotnet Wasm方案

→ **JITerpreter**

- 类似JIT（Just-in-Time）编译优化的特性
  - 将部分热点代码转换为细密的Wasm代码，从而提高运行效率
    - 一段热点代码可能生成多份Wasm文件，每份Wasm不超过4KB
- 性能提升
  - 对解释执行的程序可以带来相当可观的性能提升
  - 也可以用来优化Wasm和JS代码间的相互调用
- .Net8 默认开启

Dotnet Wasm方案

→ 优势总结

- 更小的运行内存
- 更平滑的帧率波动
- 更快的出包时间
- 方便的调试体验
- 热更新的天然支持

→ 实测案例

- 某款重度MMO游戏

	内存（MB）		
	启动峰值	主城均值	战斗均值
Unity - IL2CPP	1147	898	932
团结 - IL2CPP	1018(-129)	831(-67)	858(-74)
团结 - Dotnet	**849(-298)**	**716(-182)**	**749(-183)**

Dotnet Wasm方案

→ **未来计划**

- 进一步完善细节和开发体验
  - 补全少量不支持的API
  - 完善引擎、微信小游戏各项细节功能支持
- 更多新技术加入
  - 支持AOT
- 扩展至更多平台

GPU Skinning

→ **几种原有的Skinning方案**

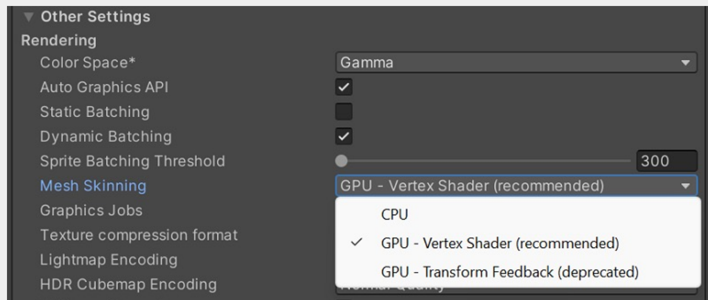
- CPU
  - 最慢
- CPU - SIMD
  - 微信小游戏分包尚不支持
- GPU - Compute Shader
  - WebGL不支持
- GPU - Tranform Feedback
  - 增加一个pass，利用Vertex shader计算，输出到GL_TRANSFORM_FEEDBACK_BUFFER
  - 对某些场景效果不佳甚至是负优化（角色多，每个角色顶点数少）

GPU Skinning

→ Vertex Shader Skinning

- 实现原理
  - Skinning在Vertex shader里计算
  - 不增加pass，处理后的数据直接用于后续MVP坐标转换
- 使用方便，仅需开发者少量介入
  - 引擎内置shader：已全部支持
  - 用户自定义shader：只需添加如下一行

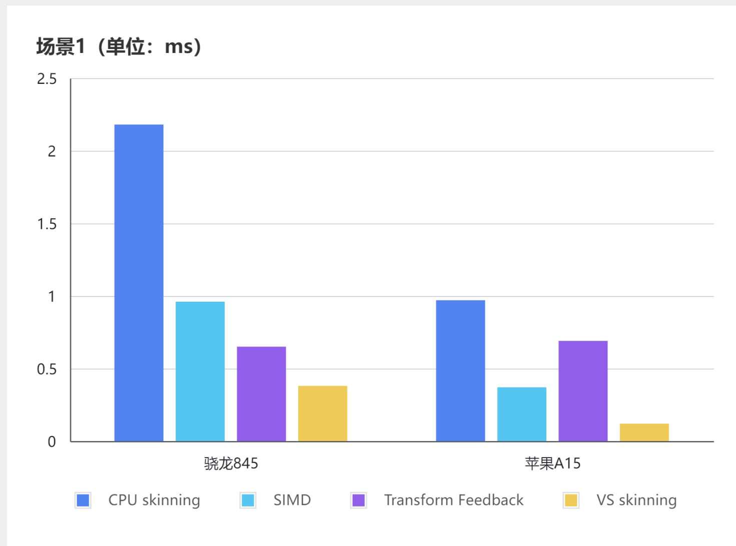
```
#pragma multi_compile _ ENABLE_VS_SKINNING
```

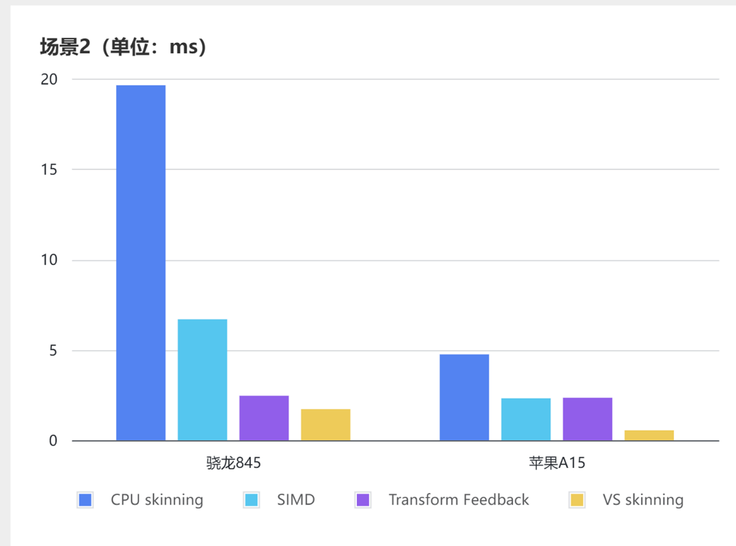


GPU Skinning

→ 性能对比

Vertex Shader Skinning 性能优势明显





Shader Warmup

→ **Shader编译比较耗时**

- WebGL不支持Binary Shader, 单个Shader编译时间通常在几十毫秒
- 现有WarmUp
  - ShaderVariantCollection.WarmUp()：可能会让主线程卡住长达数秒的时间
  - ShaderVariantCollection.WarmupShadersProgressively(1)：分散逐帧编译，但仍可能严重影响帧率

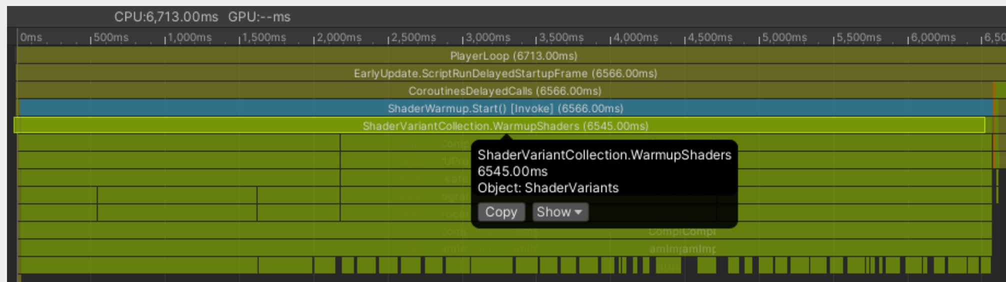
→ **异步Shader Warmup**

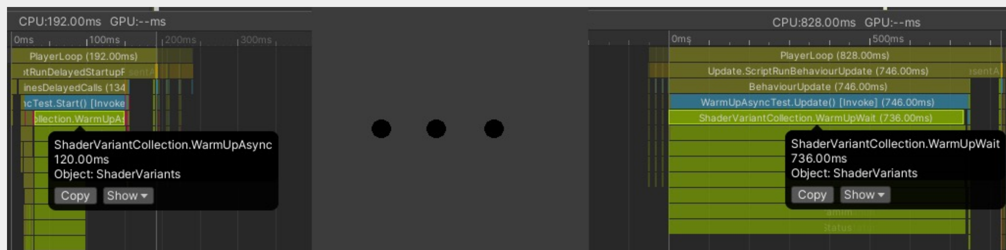
- WebGL的扩展 KHR_parallel_shader_compile 支持将Shader编译异步化
- 基于此扩展，实现了异步的Shader Warmup
  - WarmupAsync：发起异步编译
  - WarmupWait：check status

Shader Warmup

→ **实测案例**

- 原先Shader编译：6545ms
- 使用异步Shader Warmup
  - WarmupAsync：120ms
  - WarmupWait：763ms
- 合计共减少 **86.5%**





Managed code stripping – Extreme level

→ Managed code stripping

- 通过UnityLinker
  - 从DLL中剔除没有使用到的代码
  - 原有最高级别 – High
- 剔除策略仍然偏保守
  - 保留了 MonoBehaviour 和 ScriptableObject 的全部成员
- 可能剔除必要代码
  - 需要用户自行发现，添加到link.xml

Managed code stripping – Extreme level

Extreme level

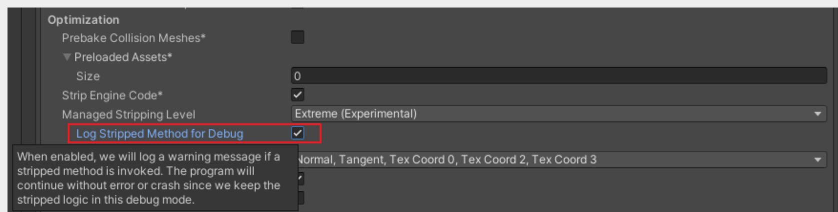
– 使用更激进的剔除规则

- 针对 MonoBehaviour 和 ScriptableObject，仅保留Unity Event Functions和实际被使用到的方法

Assembly type:		Marking rules:
	High	Extreme
.NET Class & Platform SDK and UnityEngine Assemblies	Applies any preservations defined in any link.xml file.	Applies any preservations defined in any link.xml file.
Assemblies with types referenced in a scene	Marks the following: •All methods which have the [RuntimeInitializeOnLoadMethod] or [Preserve] attribute. •Preservations defined in any link.xml file. •Marks all types derived from MonoBehaviour and ScriptableObject in precompiled, package, Unity Script or assembly definition assemblies.	标注这些： •All methods which have the [RuntimeInitializeOnLoadMethod] or [Preserve] attribute. •Preservations defined in any link.xml file. •只标注通过link.xml 保留的 MonoBehaviour 和 ScriptableObject类
All other	Marks the following: •All methods which have the [RuntimeInitializeOnLoadMethod] or [Preserve] attribute. •Preservations defined in any link.xml file. •All types derived from MonoBehaviour and ScriptableObject in precompiled, package, Unity Script or assembly definition assemblies.	标注这些： •All methods which have the [RuntimeInitializeOnLoadMethod] or [Preserve] attribute. •Preservations defined in any link.xml file. •只标注通过link.xml 保留的 MonoBehaviour 和 ScriptableObject类
Test		

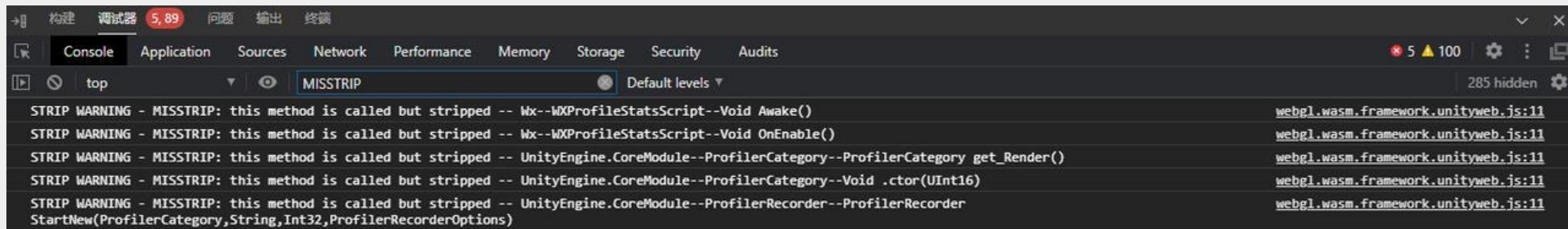
Rule Target	Action at each stripping level	
	High	Extreme
MonoBehaviour	The Unity linker marks all members of a MonoBehavior type when it marks the type.	所有的Unity Event Functions(Awake, OnEnable, Start等）以及它们的依赖
ScriptableObject	The Unity linker marks all members of a ScriptableObject type when it marks the type.	所有的Unity Event Functions(Awake, OnEnable, Reset等）以及它们的依赖

Managed code stripping – Extreme level



→ **Dryrun 模式**

- Extreme level同样可能剔除必要代码
  - 主要是涉及反射
- Log Stripped Method for Debug
  - 代码不会真的剔除，而是在待剔除的方法处插入警告的Log
  - 游戏运行时，如果待剔除的方法被调用，Console中会打印警告信息，开发者可以快速定位与收集误剔除的方法



Managed code stripping – Extreme level

→ **对IL2CPP和Dotnet Runtime都有效**

→ **实测案例**

- 在某MMO游戏上，由High level切换到Extreme level：
  - Wasm体积由**49.5MB降至44.8MB**
  - global-metadata.dat文件体积由**15.3MB降至13.3MB**

IL2CPP元数据精简

→ IL2CPP元数据

- IL2CPP运行时依赖元数据来获知C#类型、方法等信息
- 默认的元数据结构可以支持超过21亿个类型或方法
- 但在小游戏来说，方法的数量通常在万这个级别

→ 元数据精简

- 打包小游戏时，根据方法的数量自动选择满足当前要求的、精简的元数据结构
- global-metadata.dat体积缩减约 **15%**

内存分配器优化

→ 引擎分配器内存Overhead

- 出于Profile、平台兼容等原因，会记录每次分配的信息，并且由此会在内存分配器层面多引发一次内存对齐，Release版本也是如此
- 微信小游戏内存比较紧张，去除了这部分开销

→ 内存Alignment

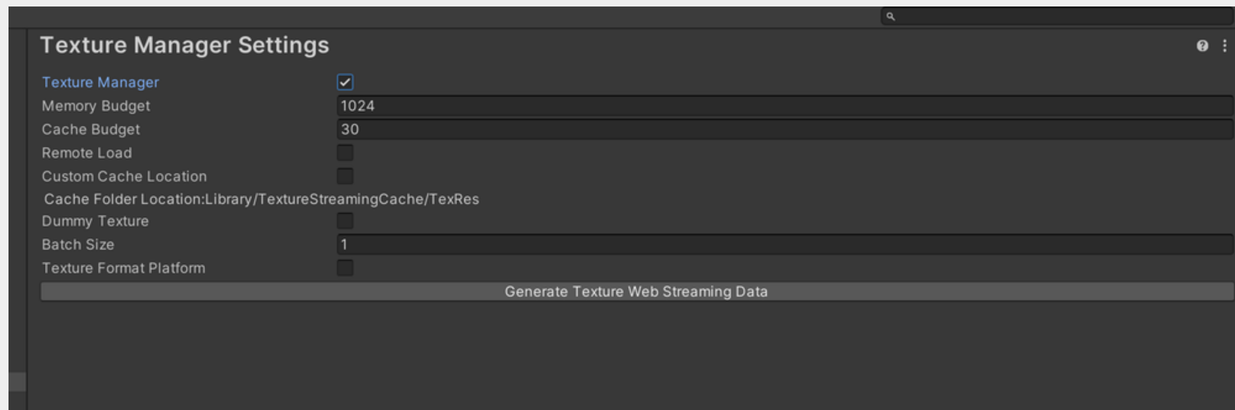
- 引擎默认内存Aligment是16，在微信小游戏平台上我们优化为4字节

→ 实测案例

- 某款中重度游戏，上述优化减少内存占用约 **10~12MB**

TextureManager

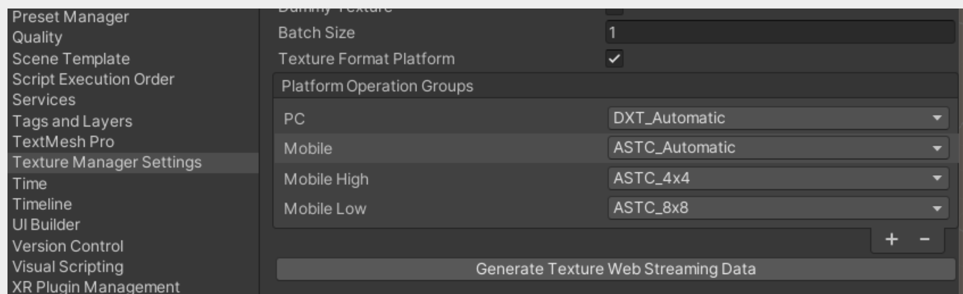
- 多纹理压缩格式支持
- 显存预算控制
- 生命周期管理
- 纹理重映射

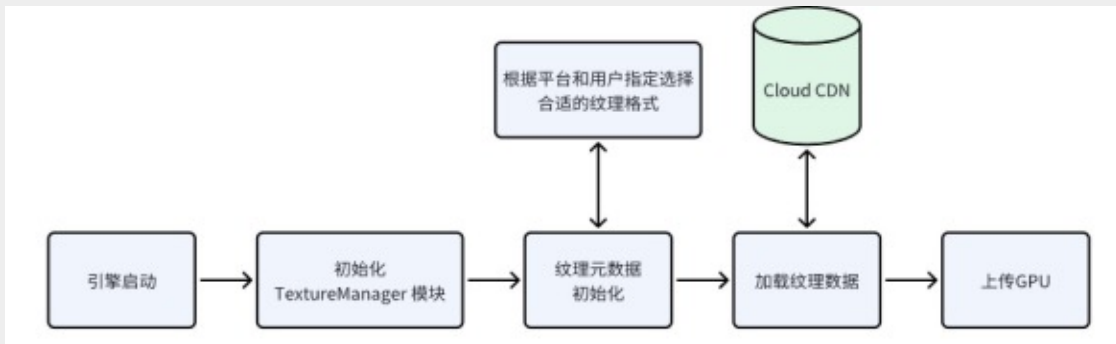


TextureManager 多纹理压缩格式支持

→ 一套AssetBundle支持多套纹理（DTX + ASTC 4x4 + ASTC 8x8等)

- 根据运行平台选择
  - PC使用DXT
  - 手机使用ASTC
- 根据设备性能选择，例如
  - 高端机使用ASTC 4x4
  - 低端机使用ASTC 8x8



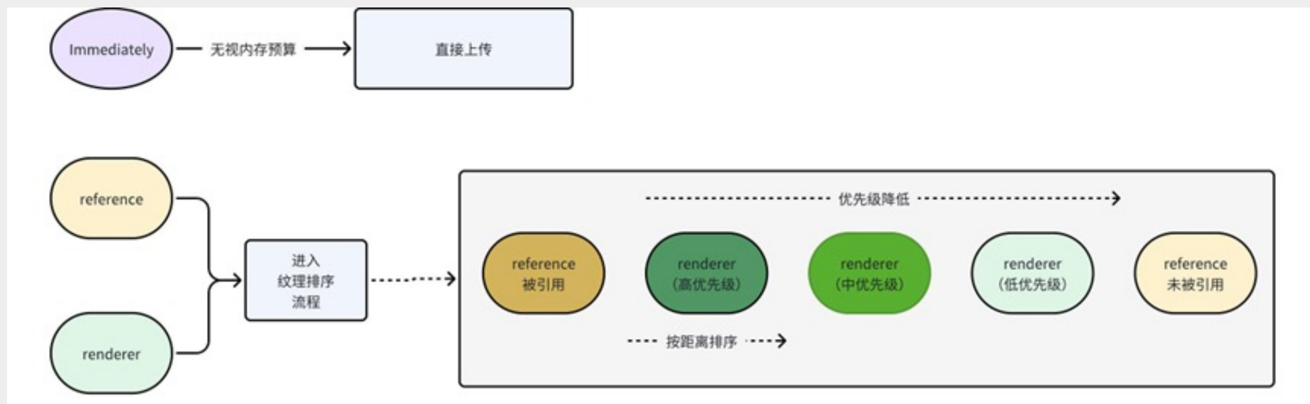


TextureManager 显存预算控制

→ 支持设置纹理显存用量上限，从GPU剔除重要性低的纹理

→ 纹理重要性计算方式

- 纹理上传模式（Immediately, Reference, Renderer）
- 纹理优先级（Renderer模式）
- 纹理Renderer可见性或者被UI引用情况
- 纹理Renderer与相机的距离



TextureManager 生命周期管理

→ **纹理生命周期**

- 调用 Assetbundle.LoadAsset(Async)、SceneManager.LoadScene(Async)加载资产时，引擎执行
  - 创建纹理对象
  - 读纹理内容进内存
  - 并将纹理内容上传至GPU

- 调用Assetbundle.Unload(true)、Resource.UnloadUnusedAssets()卸载资产，引擎执行
  - 从GPU上释放纹理显存
  - 销毁纹理对象

→ **卸载纹理显存，需要销毁纹理对象本身。**

TextureManager 生命周期管理

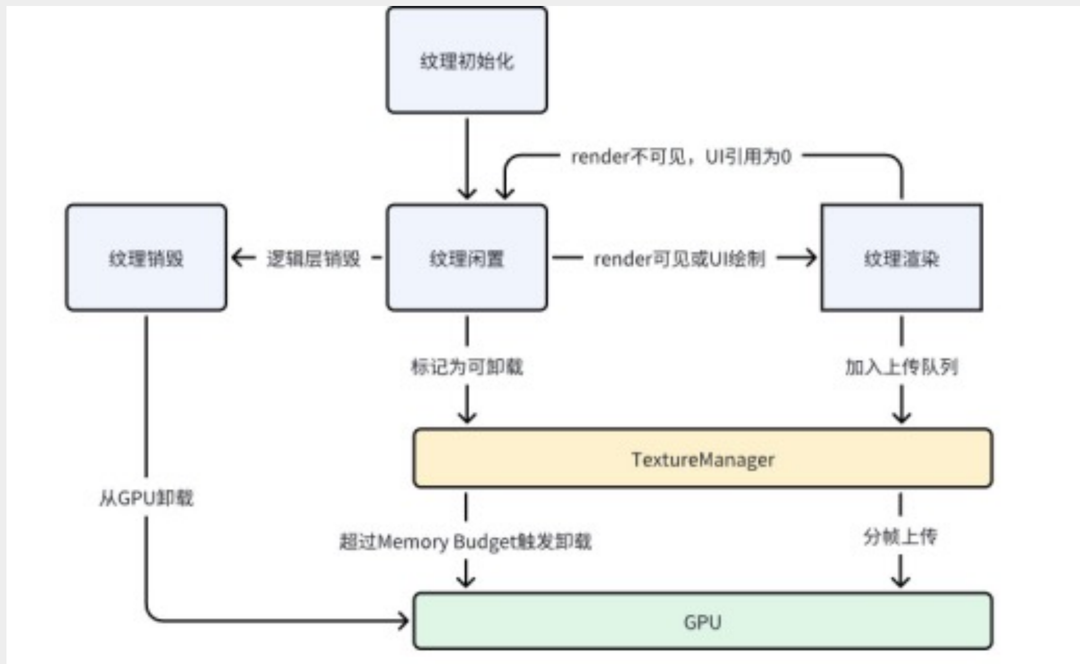
- 为控制显存，需要开发者
  - 做好资产分包，对复杂项目来说是个很大的挑战；
  - 维护一个资产管理系统，采用引用计数来判断等来判断资源是否可以卸载。

- 在小游戏平台存在的问题
  - Resource.UnloadUnusedAssets()容易导致卡顿；
  - AssetBundle.Unload(true)需要等待AB所有资源闲置；
  - 因为资源依赖难以卸载所有未使用的纹理。

TextureManager 生命周期管理

→ 纹理生命周期管理

- 仅对纹理的GPU显存进行管理，不改变纹理原有的生命周期；
- GPU上传延迟至纹理绘制可见；
- 进入渲染状态时，自动上传；
- 纹理闲置时，自动标记为可卸载。



→ 解决闲置纹理浪费显存问题

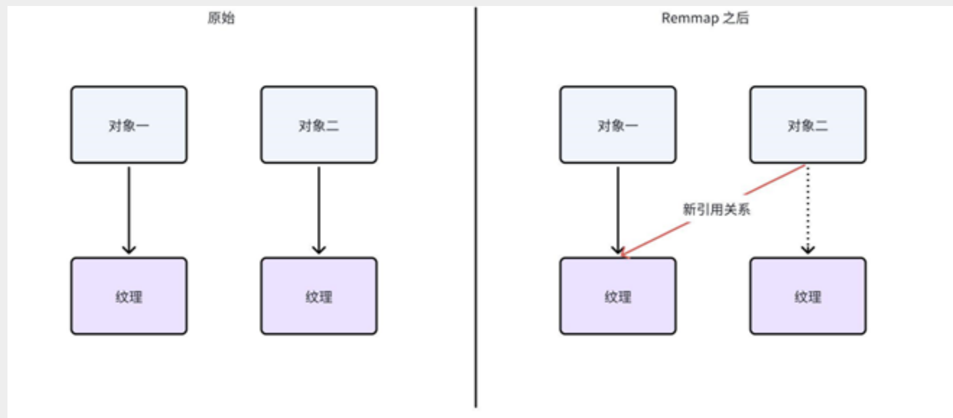
TextureManager 纹理重映射

→ 纹理显存冗余问题

- 不合理的AB组织结构
  - AB中纹理冗余
- 不合理的AB加载流程
  - AssetBundle.Unload(false)

→ 纹理重映射

- TextureManger上传GPU前会
  - 检查纹理是否可读写
  - 对比纹理数据是否相同
- 映射后，共用GPU上的一份纹理数据



小游戏宿主（Android）

→Connect App 小游戏宿主能力

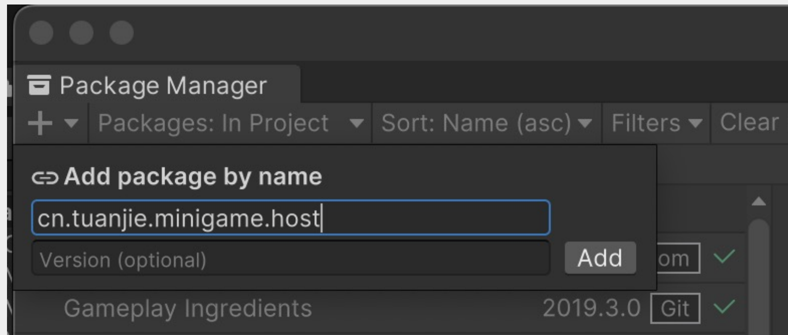
- 基于v8开发；
- 提供CPU使用率、帧率、内存、VConsole、启动时间等信息；
- 引擎内一键打包上传，扫码即可运行；
- **支持更高的Profiling时钟精度；**
- **提供C# Debugging的能力。**

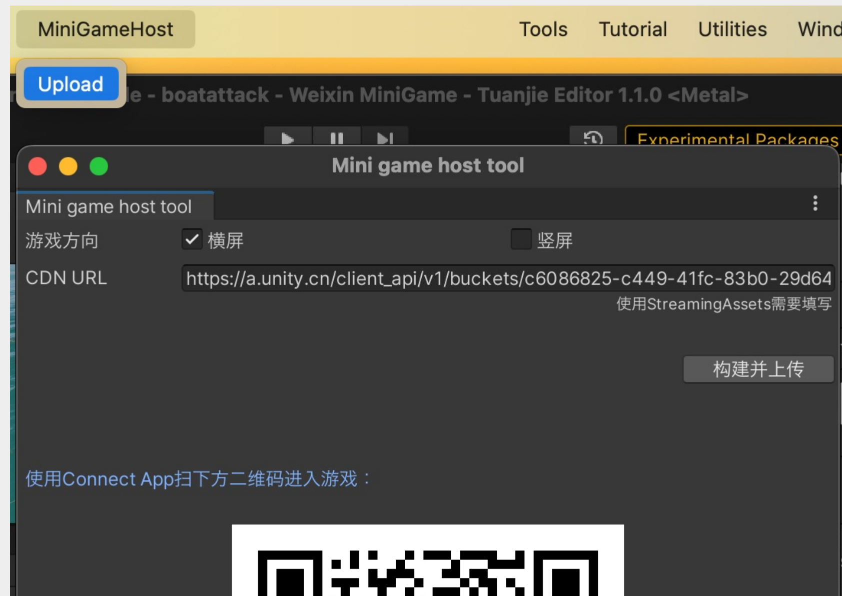
Connect App下载

小游戏宿主（Android）

→打包到Connect 小游戏宿主

- 从package manager 安装cn.tuanjie.minigame.host package
- 从MiniGameHost -> Upload 窗口打包





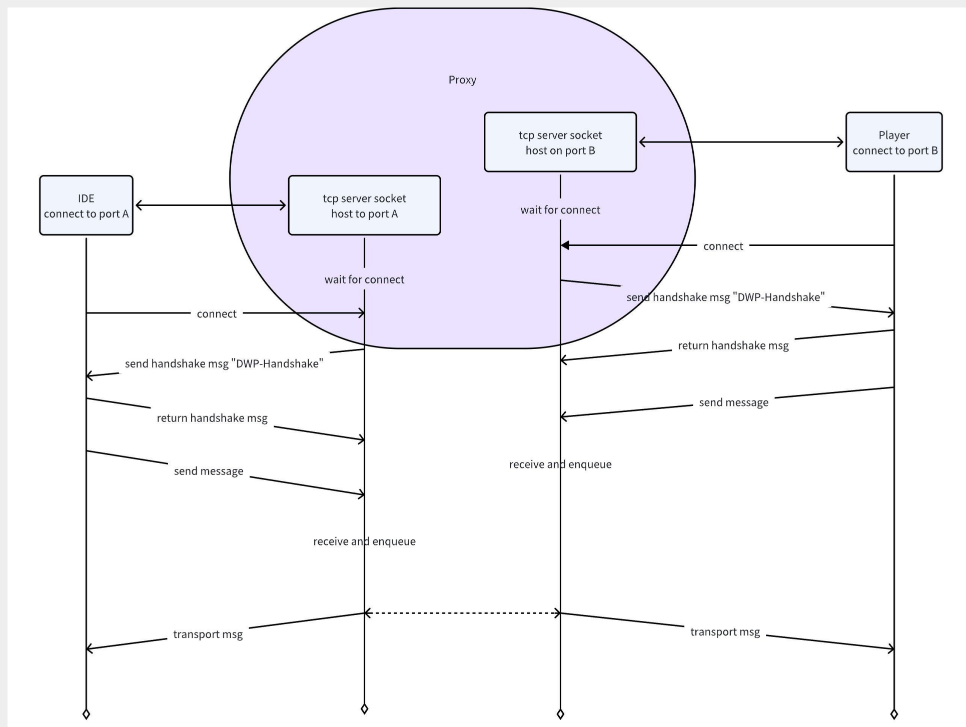
C# Debugging

→**Unity WebGL未支持代码调试的原因**

- 多线程受限
- 不支持Socket
- 无法发送广播
- 无法监听端口等原因

→**微信小游戏平台的解决方案**：

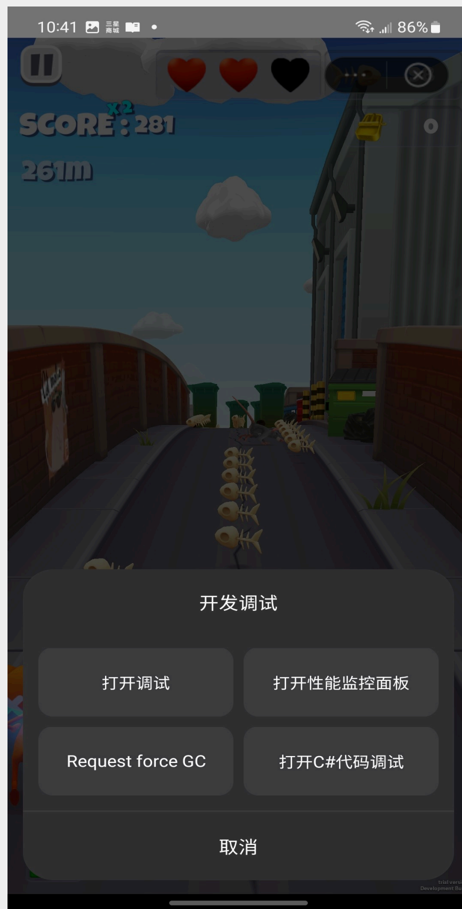
- 微信小游戏平台支持多线程；
- 小游戏宿主实现了广播、监听能力；
- 小游戏宿主增加中间代理，
  桥接WebSocket和Socket。



C# Debugging

→**使用Connnect小游戏宿主调试代码**

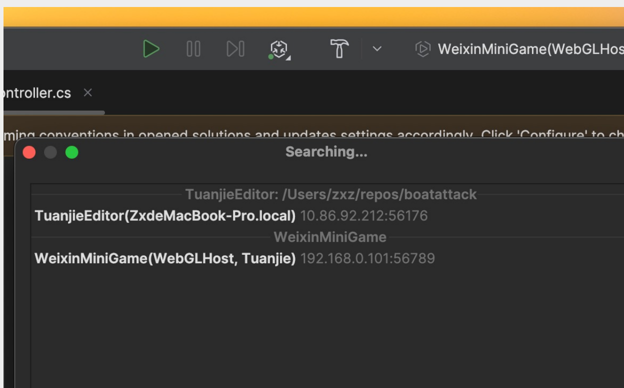
- 打开Development Build和Script Debugging
- 宿主打开C#代码调试
- IDE和宿主运行在同一局域网内
- 支持多个 IDE
  - Visual studio
  - Rider



C# Debugging

→案例：

—在Rider中调试BoatAttack



tj1.0.1/tutorial/dev　　WeixinMiniGame(WebGLHost, Tuanjie)

SimpleCameraController.cs　　AiController.cs

```
   Event function   Averozer Wu +5
36  private void Start()
37  {
        int boatid = gameObject.GetComponent<Boat>().playerIndex;   boatid: 0       '@18765960'
39      _boatAddLevel = RaceManager.raceData.boats[boatid].boatType;
40      int playerboatlevel = MenuCtrl.Getlevelfromname(RaceManager.raceData.boats[0].boatType);   playerboatle
41      int addindex = MenuCtrl.Getlevelfromname(_boatAddLevel) + SceneManager.GetActiveScene().buildIndex - 1
42      if (addindex > playerboatlevel + SceneManager.GetActiveScene().buildIndex + 2)
43          addindex = playerboatlevel + SceneManager.GetActiveScene().buildIndex + 2;
44      _accPower = 1.3f * (1f + addindex * _addpoweracc);
45      engine.horsepower = engine.horsepower * (1 + addindex * _addpowervel);
46      RaceManager.raceStarted += StartRace;
47      _islevel2 = (SceneManager.GetActiveScene().name == "level_Island2");
48      navPath = new NavMeshPath();
49  }
```

Assembly-CSharp › BoatAttack › AiController › Start

Console　Debug Output　Parallel Stacks　Memory　Modules

AiController.Start() in BoatAttack

- Active scene = Scene
- this.gameObject = 洛里安 (active: true, layer: 11)
- this = {AiController} enabled: true, gameObject: 洛里安
- addindex = {int} 0
- boatid = {int} 0
- playerboatlevel = {int} 0

Remapper运行时内存优化

- → **Remapper**
  - – **序列化位置**和**InstanceID**之间双向映射关系的数据结构。
- → 优化前
  - – InstanceID每次加二，不复用
  - – 哈希Map，稀疏，内存翻倍增长
- → 优化后
  - – InstanceID每次加一，复用
  - – 数组，紧凑，内存线性增长

▼ PersistentManager.Remapper (1)	16.0 MB
Remapper	9.9 MB
▶ Rendering (10)	2.0 MB

▼ PersistentManager.Remapper (1)	3.0 MB
Remapper	3.0 MB
Objects	

Remapper目前仅在微信小游戏平台开启，后续会逐步向其他平台开放

Math库支持Wasm SIMD

→ Wasm SIMD

- Emscripten支持 WebAssembly SIMD功能，SSE2 和 ARM_NEON SIMD 指令集的指令可以通过编译转化成在 wasm 虚拟机下的指令进行模拟，从而获得比普通标量化运算更好的性能
- 团结早些版本已经在微信小游戏平台的Skinning模块接入了WebAssembly SIMD

→ Math库

- 为引擎各个模块提供基础的数学运算
- 其中的向量和矩阵等相关的运算天生适合使用SIMD进行优化
- 使用WebAssembly SIMD intrinsics重写了Math库中的实现
- 由此引擎整体都获得了来自SIMD的性能提升

AssetBundle打包优化

→ **优化BuildPipeline.BuildAssetBundles()接口**

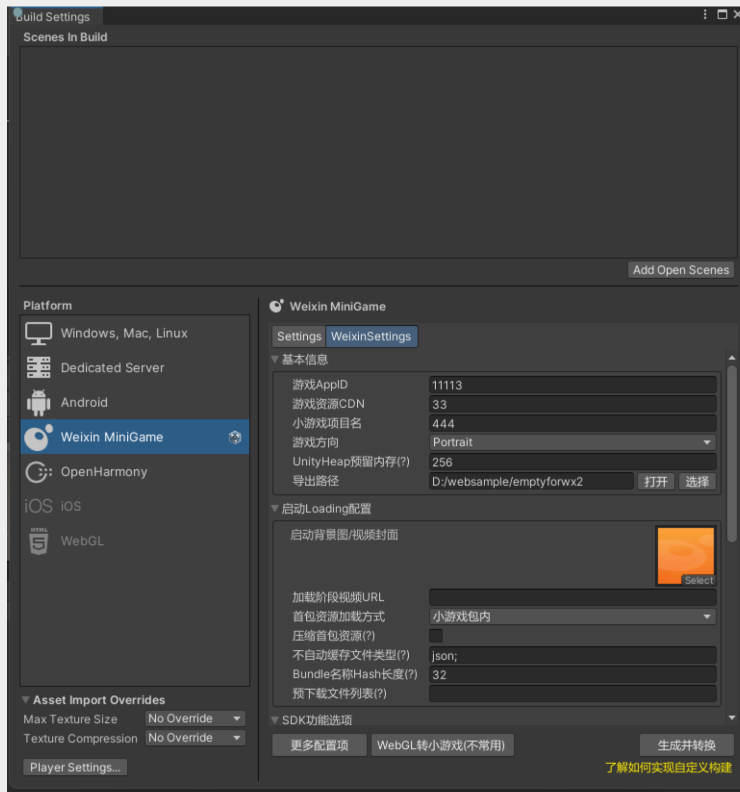
- 改进了AB打包流程中图集相关的逻辑
- 该优化目前仅在微信小游戏平台开启，后续会逐步向其他平台开放
- 实测案例
    - 2.5万个AB，打包AB时间从优化前的**160分钟减少为70分钟**

深度集成微信小游戏SDK

→ 切换至Weixin MiniGame平台，引擎自动安装WXSDK package；

→ 原微信小游戏转换工具面板内嵌至BuildSettings界面中；

→ 后续将支持引擎内选择微信SDK版本。



Thank you!

Let's keep in touch - liang.zhao@unity.cn