**虚幻引擎心得**

一.地形系统的深入浅出

使用虚幻引擎的地貌系统可以创建广袤的、开放的世界环境，这得益于地貌系统强劲的LOD系统和高效的内存使用，可以创建大于terrain地形系统几个数量级的地形地貌。虚幻引擎现可处理最大精度为 8192x8192 的高度图，支持巨型室外世界场景，无需对引擎或工具进行修改即可快速便捷地创建出辽阔庞大的室外环境。

地形比静态网格体更适合创建巨大地形。地形的顶点数据为每个顶点 4 字节，静态网格体以 12 字节矢量储存位置，切线 X 和 Z 矢量每个均装入 4 字节，16 位或 32 位浮点 UV 每个顶点为 24 或 28 字节。这意味着在相同的顶点密度下，静态网格体占用的内存是地形的6或7倍。地形还将其数据保存为纹理，可为远处场景流式输出未使用的LOD层级，并在靠近远处场景时从背景磁盘中进行加载。地形使用恒定的高度场，因此可更高效地存储碰撞数据。

地形系统在 GPU 内存中将把地形渲染数据存储为纹理，使数据可在顶点着色器中进行查找。数据被装入 32 位纹理，高度以 R 和 G 通道的形式占据 16 位，X 和 Y 偏移在 B 和 A 通道中存储为 8 位数值。

地形被划分为多个组件，即为虚幻引擎的渲染、可视性计算和碰撞的基础单元。地形中的组件尺寸均相同，且为方形。地形创建时将决定地形组件的尺寸，选择取决于需要创建的地形尺寸和细节。

地形由组件组成，最大支持规格为32 x 32，组件的比例可以任意搭配；组件由分段组成，分段的规格为1 x 1或2 x 2 ；分段由方格组成，支持方格的规格为7 x 7到255 x 255；地形的维度基于每个分段中的四边形数量、每个组件中的分段数量、以及地形中的组件数量。地形 Actor 拥有颜色编码，可以明确分辨每个分段的作用。地形的边缘以黄色高亮，每个组件的边缘为浅绿色，分段边缘为中绿色，方格四边形为深绿色。

组件可被划分为 1 个或 4 个（2x2）子分段。这些分段是地形 LOD 计算的基础单元。使用 4 个（2x2）子分段选项获得的高度图与在一个子分段上使用 4 次组件所获得的高度图相同，但使用的组件次数越少，运行性能越好。每个分段的尺寸（以顶点数为单位）必须为 2 的幂次方（最大为 256x256）。因此不同的 LOD 层级可存储在纹理的 mipmap 中。这将使一个组件中四边形的数量为 2 的幂次方减 1（如每个组件有一个分段）或 2 的幂次方减 2（如每个组件有四个分段）。

对组件尺寸和组件总数量进行选择是为保证性能进行的平衡。组件尺寸越小，LOD 的过渡越快，并对更多地形构成遮挡，但需要的组件数量越多。

每个组件均有渲染线程 CPU 处理消耗，每个分段均为绘制调用，因此需要将它们的数量控制到最低。Epic 推荐使用的巨型地形最大组件数量为 1024。

创建地形有两种方式，一种为自定义地形，一种为导入高度图地形。自定义地形方便程序人员快速搭建地形进行测试。但通常为了使地形更加逼真或者达到理想效果，会由专业美术人员使用GlobalMapper、World Machine等工具制作高度图来作为地形。高度图的格式为16bit，灰度，以小字节序排列的RAW格式。

导入的地形难免会根据需求进行微调，虚幻引擎提供了一套强大的工具集，包括雕刻、平滑、平整、斜坡、腐蚀、水力侵蚀、噪点、重新拓扑、可见性等功能，完全满足工作的需要。

在处理高度图时应注意分辨率对地形的影响。斜坡与平地的交汇处会因分辨率的过大变得突兀，此处的过大指的是一手获得的卫图与处理后的高度图的分辨率差值。当分辨率差距过大造成过渡不平滑时，可以在处理高度图时进行平滑处理。处理后的高度图最高点和最低点不会改变，但是过度的平滑处理会使得除最高点和最低点之外的所有点变平，在进行平滑处理时应注意实时观察。如果除最高点和最低点外有重要的高山或盆地不建议使用平滑处理，因为会改变它们的海拔。如果显得突兀的边缘较少的话，可以用引擎自带的地形修改器进行修改。

高度图的每个点的高度肯定是无法与虚幻引擎中地形的每个点一一对应。因为在地形的两次软件的转换中势必存在误差，还有在UE4中计算Scale时省略过小数字产生的误差，在测量时产生的误差，但是可以保证99%的还原，此结论是经过多次细致的测量得到的。最高点与最低点完全可以对上，其他的点要取决于在World Machine里是否进行过平滑处理，如果处理过那么就无法对应上，没有处理过要看分辨率，当地形的分辨率大于等于卫片的分辨率就可以对应上，当地形的分辨率小于卫片的分辨率就无法对应。

使用单一地形存在帧率较低，内存占用较高，不能对某一地段的精度进行修改等问题。同时在程序实际运行时我们可能并不需要加载所有地形，在地形较大视野较小时更是如此，作为一个成熟的游戏引擎，虚幻引擎为此提供了流关卡和世界组件。流关卡因为细腻灵活而更适合于较小区域的动态加载，例如从一个房间到另一个房间；而世界组件以地图为操作单位，加载大地图、多分段时，世界组件更易于编写和判断，在制作大地形时可以实现大型地图的分段加载以达到减少资源浪费的目的。

熟悉以上内容后对于地形系统便会有一个较为全面的认识，实际操作起来也会得心应手。

二.编程心得

虚幻引擎每一个对象都有基于Uobject，由此可以使每一个对象可追踪，提供通用的属性和接口，统一的内存分配释放，统一的序列化模型，统计功能，调试的便利，为反射提供便利，UI编辑的便利。虚幻引擎以此构建了一个Object运行的世界。

虚幻引擎包含着组件化开发的思想，UactorCompoent就这么应运而生了。UactorCompoent作为功能的载体，具备一定程度的嵌套组装能力。这样做使得功能划分更佳清晰，可以按组件分配开发任务；更好排查问题，某个组件出现问题，直接对组件进行处理；项目可维护性更强，提高开发效率。

虚幻引擎有了组件那么必定有容器，在虚幻引擎中AActor充当容器，无论是实体还是移动，它的功能都是由Components拼装而成。

对于游戏引擎而言，“显示”指的是游戏的UI，或是手柄上的输入和震动，是与玩家直接交互的载体；“数据”指的是Mesh，Material，Actor，Level等各种元素组织起来的内存数据表示；“算法”可以是各种渲染算法，物理模拟，AI寻路，。抽象这三个变化，并归纳关系，就是典型的MVC模式了。作为上帝的Controller就承担着“算法”的功能，Controller不能是一个Component，一是因为Component的层级太低，表达的是功能的概念而非逻辑；二是Component必须依附于Actor存在，而我们的Controller希望能独立存在。 Actor具有配置动态生成、输入事件响应、Tick、可继承、可容纳Component、可在世界里出现、可在网络间同步的能力，只需要再加个控制Pawn的能力就能达到“算法”的要求，所以Controller由Actor分化而来。

了解以上内容后对虚幻引擎的结构就有了初步、大体的认识，这对于蓝图和C++编程是十分有必要的。同时，对引擎节点的熟悉也是十分有必要的，看源代码，查文档，逛论坛会成为学习虚幻引擎不可或缺的一部分。

良好的编码规范在保证代码在高质量完成需求的同时具备良好的可读性、可维护性。如成员变量前要加下划线，首字母要大写；成员函数要首字母大写并且有意义，注意语态；系统头文件应用#include <xxx.h>；自定义同文件应用#include "xxx.h"；头文件对变量函数声明，CPP文件进行定义；局部变量小写，开头不加下划线等要点。

三.性能分析

我制作了一个可以机动、瞄准、射击开火、寻路、搜索敌人的坦克，在较小的场景中使用这样的几个低模坦克可以达到一百二十帧，当在一百平方公里贴上高精度卫片随机地点生成两百个高模坦克时只有不到二十帧。为了提高性能，我开始对性能进行分析，采用控制因素的方法,把多因素的问题变成多个单因素的问题,而只改变其中的某一个因素,从而研究这个因素对事物影响,分别加以研究,最后再综合解决。

在一百平方公里贴上高精度卫片情况下，帧率达到60。

在一百平方公里贴上高精度卫片随机地点生成两百个低模坦克时，帧率达到68左右。此时逻辑线程为18毫秒左右，渲染线程为9毫秒左右。此时的逻辑线程耗时大于渲染线程，最终帧数与逻辑线程耗时相近，根据短板效应可以主要考虑逻辑线程。使用性能分析工具发现此时的计算面为18万，基准图耗时2.7毫秒，动态原始绘图2.7毫秒，移动组件消耗了8.4毫秒，数据更新消耗3.6毫秒。由此可见，两百个低模坦克因网格体、特效和运行逻辑所带来的计算对性能影响还是十分明显的。

在一百平方公里贴上高精度卫片随机地点生成两百个高模坦克时，帧率达到20左右。此时逻辑线程为42毫秒左右，渲染线程为14毫秒左右。此时的逻辑线程耗时大于渲染线程，最终帧数与逻辑线程耗时相近，根据短板效应可以主要考虑逻辑线程。使用性能分析工具发现此时的计算面为一亿三千面，基准图耗时4.7毫秒，动态原始绘图4.7毫秒，移动组件消耗了36毫秒，数据更新消耗22毫秒。

经过分析带来此问题的原因是因为一个高模大约一万多面，低模只有十几个面；从结构上比较，高模比低模多出两个履带，十八个轮子；高模有精细的UV贴图，低模只有简单的单色贴图。

注意渲染视图也会产生较大的开销，大约为8毫秒左右，在考虑问题时应将此纳入其中。

为了提高性能，经过测试发现，减少屏幕上的UI可以提高帧数，具体提高多少帧应以UI处理前后的复杂程度而定；独立窗口播放要比编辑器视口播放提高7帧左右；将实体炮弹及时销毁可以大幅提高性能，将炮弹从之前的不销毁改为一秒后销毁，帧数上提高6帧左右；添加距离场修改物体将会大大减少面数，减少逻辑线程时间。

优化后帧数提升到30帧左右，但实际上开销最大的还是运动组件方面，34毫秒的总帧时间，运动组件占到了27毫秒。通过对运行数据进行采集，然后逐帧分析，追踪到引擎中的代码，发现在这其中包含着物理模拟、组件碰撞、碰撞更新，这几个开销较大，目前还在尝试在此方面进行优化。