# 对游戏引擎架构的理解

## 游戏引擎的由来和历史

今天看来，由于unity和unreal这类商业引擎的存在，自己制作一个小游戏似乎已经不是什么遥不可及的事情，一般的大学生甚至高中生在学习了一些基本的编程和美术之后都能一定程度上实现出一款应用程序，尽管质量参差不齐，但是这些应用程序也能达到被称为游戏的要求。

回望昨日，在没有这些被称之为引擎的软件支撑开发的前提下，每一个游戏都需要从底层做起，游戏制作者需要自己负责跟操作系统层面的代码打交道，实现物理碰撞、三维场景渲染、音频系统，游戏逻辑等等诸多方面的功能。这无疑是难以想象的工作量，尤其是在每一款游戏都几乎需要重新制作这些内容的情况下。

懒惰是计算机科学的发展动力，程序员们发现不停的造轮子带来的巨大损耗令人难以承受，并且所有游戏都确实有共同的功能需求，最典型的有玩家输入、渲染、音频等。

于是，几乎是必然的，游戏引擎的概念出现了，负责把游戏创作者和底层的系统隔开，专门处理这些重复的，模板化的但是又不得不考虑的繁琐工作。

## 引擎具体是什么？

游戏引擎这个概念是在开发过程中，人们发现有一些共同的开发部分，这些部分与底层强相关，但是与游戏本身的玩法关联性并不大，因此逐渐将游戏创作和底层的代码系统剥离从而得到的。

可以说，早期的几乎每一个游戏都有一个属于自己的引擎，但是到了今天，常常是一个引擎服务多个游戏，游戏和游戏引擎的对应关系从一对一变成了多对一。

但是让我们回到最原点，引擎还是游戏的一部分，那么电子游戏是什么呢？从纯粹理性的角度来说，电子游戏就是一个具有输入输出的计算机应用程序，是一个能够在玩家输入操作数据基础上，对计算机内部数据进行有机修改并依此输出图形画面和声音的应用程序。彼时，引擎就是这个应用程序的一部分。

现在，应该说所有游戏都在复用引擎，那么引擎其实就是生产电子游戏这类计算机应用程序的一个巨大的代码库和工具链，更具体的,对于今天的许多情况而已，引擎可以是一个用来生产实时3d渲染的应用程序的框架，它已经不必拘泥于游戏本身，相反因为引擎这个事物演变所产生的高度泛用性和目前的计算机性能已经能满足多数任务的要求，引擎已经能够作为一般应用程序的开发环境。

应该说，游戏引擎起源于电子游戏，但必定不会终于电子游戏，或者说，电子游戏的定义在今天已经不应该局限于以往，而应该被拓宽到互动媒体应用程序的定义上。

## 引擎里有什么？以Unity为例

引擎的内容是一片广阔的知识海洋。

自底向上，游戏首先需要决定它自己在什么平台上运行。根据平台和硬件设备的不同会有对应的驱动程序，在这之上会有操作系统，然后才是游戏引擎需要接入的地方。

一般而言，引擎会建立在许多第三方库和api上，例如图形编程需要调用GPU，那么就需要通过open GL或者directX等图形接口来跟显卡打交道，在mac系统上可能还需要使用metal图形接口；在物理实现上，可能也要依赖于havok或者PhysX等物理引擎。

### 资源管理系统

由于引擎首先是一个应用程序的开发环境，所以在能够处理多媒体输入输出之前，首先要进行数据的管理，我们说计算机程序就是数据结构加算法，因此如何合理的存储游戏中需要使用的资源（通常是巨量的、类型多变的、复杂的）是引擎要做的首要任务。当然，在引擎构建初期，这一任务可以先藉由操作系统和C++语言本身的文件处理功能进行简单的图像文本等数据的读取，但是进行到一定程度之后就一定需要定义引擎开发者自己约定的数据格式，并使用一个单独的软件层和代码库来进行管理。

对外，引擎的资源管理接受来自计算机各个位置或是网络上的资源，对内，引擎把这些资源或者是资源的索引暴露给游戏开发者，中间使用自己定义的数据格式。

在Unity中，在资源导入时，一般都需要进行资源导入选项的设置，这就是在帮助Unity按照正确的方式接收并构建外部资源在引擎内部的数据结构表达方式。

### 渲染和物理系统

现代的游戏引擎一般都需要为玩家处理渲染和物理模拟等工作，当然一般而言游戏引擎提供的物理模拟不会像离线渲染里的物理结算一样复杂，而是集中在刚体动力学和碰撞检测上。渲染是一个非常有意思的领域，首先引擎要在应用层定义好场景，然后把场景的各种信息通过opengl等图形接口转交给GPU，并告诉GPU如何绘制，这些信息包括但不限于场景物体的点表、线表、使用的贴图、场景光源位置、相机位置、光照类型等，其中相机等信息需要CPU抽象后传输一个MVP矩阵给GPU，对每个顶点进行矩阵变换。这个通常称为Drawcall的过程一般很复杂，CPU首先需要通知GPU准备显存接受数据，并通过Shader（当然需要引擎先编译链接shader程序）告诉GPU如何绘制这些数据，还要管理GPU的各种算法选项。一般而言对于任何一个物体都要经历一遍这个过程，而现代引擎通常会尽量让共有这一过程的物体打包一起渲染，从而减少CPU和GPU之间传输数据的次数，避免渲染速度的bottleneck。

由于shader并不是一个面向艺术家的概念，所以一般而言引擎会提供material类型，来方便艺术家理解，同时在Unity中，material类还整合了一个物体所需要的渲染信息。例如Unity定义了自己的shaderlab语言，在其中将Uniform类型放入到property表中，并在Inspector面板上暴露出来，除此之外shaderlab还支持调整许多渲染选项，例如Z-Buffer的写入模式、Z-Test启用与否、像素混合模式等等。

这些原本需要使用图形接口的函数来调整的功能，Unity都通过自定义shader文件并提取其中用户设置的方式，来帮游戏制作者完成了。

除了最基本的材质系统，Unity引擎自然还要考虑现代的实时渲染方式，例如抗锯齿，实时层级式阴影，全局光，反射折射，PBR，天空盒，次表面散射，粒子系统以及屏幕后处理等各种用于实现真实感渲染的技术都被集成在Unity中。

物理碰撞跟渲染本质上都属于图形学范畴，引擎在物理引擎上，需要支持各种类型的物体碰撞和求交过程，无论在物理还是渲染上，这类算法都起到极大的作用。

### 动画系统

除了物理系统之外，动画系统也是让游戏的画面动起来的重要系统之一，Unity里提供了多种动画系统，包括序列帧动画，骨骼动画等等。同样，在动画层面Unity引擎也给游戏开发者们留出了接口，不需要自己去编写IK和FK动画的原理，也不需要去考虑动画文件的数据结构，这些已经被资源加载系统解析过了，开发者们需要做的就是在DCC软件内生成美术内容，然后在引擎内通过状态机来控制动画的播放效果。

### 输入输出系统

处理输入输出也是引擎非常重要的一个方面。游戏中面临的输入设备一般而言有键盘鼠标，手柄，触摸屏等等，所以引擎需要对这些设备的可能输入进行枚举和定义，并通过事件机制监听这些输入事件的发生。这些事件监听常常跟图形渲染的循环结合在一起，但是也有的做法是将逻辑层和表现层的帧率分开处理，因而具体的处理视游戏项目和引擎而定。

## 总结

引擎的内容十分广泛，我只能依据我自己相对熟悉的部分进行一个简单的介绍，但是归结起来，引擎本身和其他软件系统所做的事情并无二至，都是遵循着一种尽可能降低不同软件层的研究人员对其他层知识的了解要求的设计理念，将用户和硬件驱动操作系统等复杂又专业的知识分开。

就像图形渲染的子系统一样，即便底层需要做大量的工作去保证图形流水线的正常高效运转，但是到了用户的层面，分层的设计使得顶层的开发者几乎不需要管理如何把物体在窗口里绘制出来这件事。

同样的思想会被应用在各种领域，尽管在计算机领域内可以说是随处可见这样的设计思想。虽说实现一个具有完全普遍适用性的引擎几乎是一个不可能的任务，但是今天的商业引擎例如unity和unreal无疑是在朝着这个方向进发，现代引擎已经逐渐涵盖了游戏开发的各个方面，除了上面提到的系统以外，还包含了计算机网络、人工智能甚至是高性能GPU计算的功能。知海无涯，谨当上下而求索。