多线程，多显示场景图形设计：
一种新的过程模型

作者：Don Burns，2001

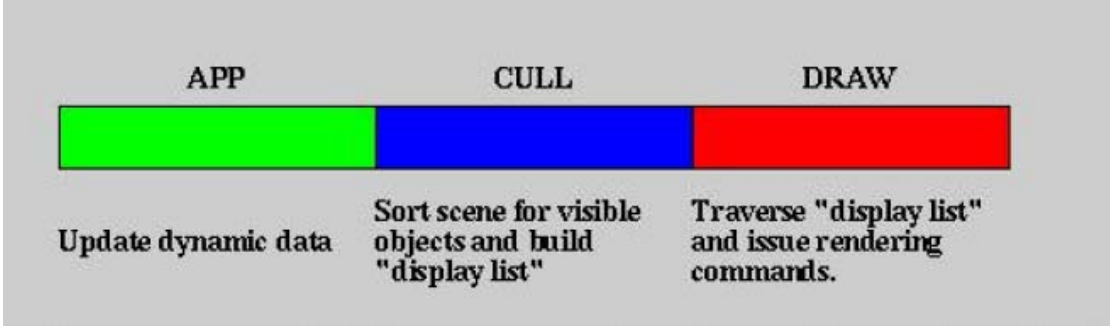
译者：王锐，2008

新的设想

场景图形的主要目的是改善场景优化，渲染状态排序和各种其它操作的性能，降低图形渲染引擎的负荷，并实现复杂场景的“实时”渲染。实时渲染的目标是以足够高的帧速（符合人眼的交互要求）渲染场景。飞行模拟程序通过窗口输出（out-the-window）的图像生成频率要求为 60Hz 或者更高，这样显示的内容才不会发生异常。而 30Hz，20Hz 或者 15Hz 的频率则是“可交互”的，即，视口可以交由用户进行操控，并在合适的时间响应用户的输入。基于这些目标，我们需要参照帧速 60Hz 的仿真要求实现我们的设计。我们假定帧速率恒定，且图形子系统针对垂直消隐时间（vertical blanking time，即电子枪扫描完一帧之后返回原点的时间）与渲染交换缓冲区（rendering buffer swap）执行同步的能力恒定。此外，我们还假定多显示的系统已经实现了同步锁相（genlock），至少实现了帧锁相（frame lock，借助硬件使每个显示屏上的帧实现同步），这样就可以实现垂直回描边界（vertical retrace boundary，即电子枪扫描完一帧之后返回的边界）在所有图形子系统上的同步。

多任务，多显示，单系统绘图的传统方法

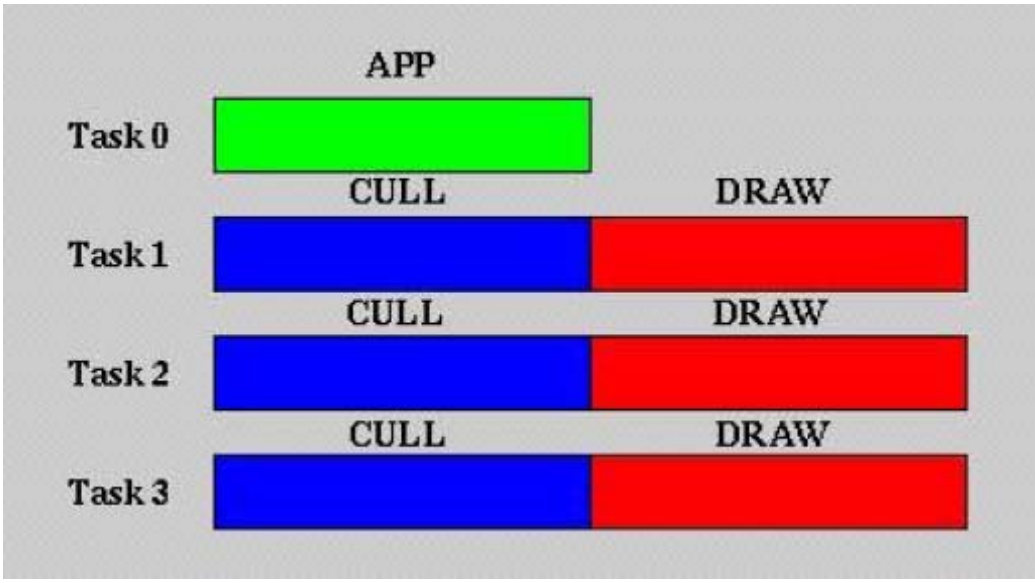
使用场景图形实现实时渲染的“传统”方法是实现多个阶段，即：APP（用户阶段），CULL（拣选阶段），DRAW（绘制阶段）。APP 阶段用于更新所有的动态用户数据，包括相机的位置，以及运动对象的位置和属性变化。CULL 必须跟随在 APP 之后，这一阶段中，首先参照视口锥截体（viewing frustum）中的可见对象，其次参照用于渲染性能的渲染状态，对场景进行排序。CULL 阶段更新摄像机位置的依赖数据，并为其后的 DRAW 阶段构建一个“显示列表”（display list）。DRAW 阶段仅仅是遍历整个显示列表，并执行 OpenGL 的各种调用，将数据传递给图形子系统进行处理。



在一个具备多个图形子系统的系统中，有必要为每个图形子系统设置 CULL 和 DRAW

阶段，因为在假定各个视口锥截体均不同的前提下，只有 CULL 阶段能够为每个子系统构建唯一的“显示列表”。而 APP 阶段则不必设置多个，因为每个视口均会共享同一个 APP 阶段更新的动态数据。

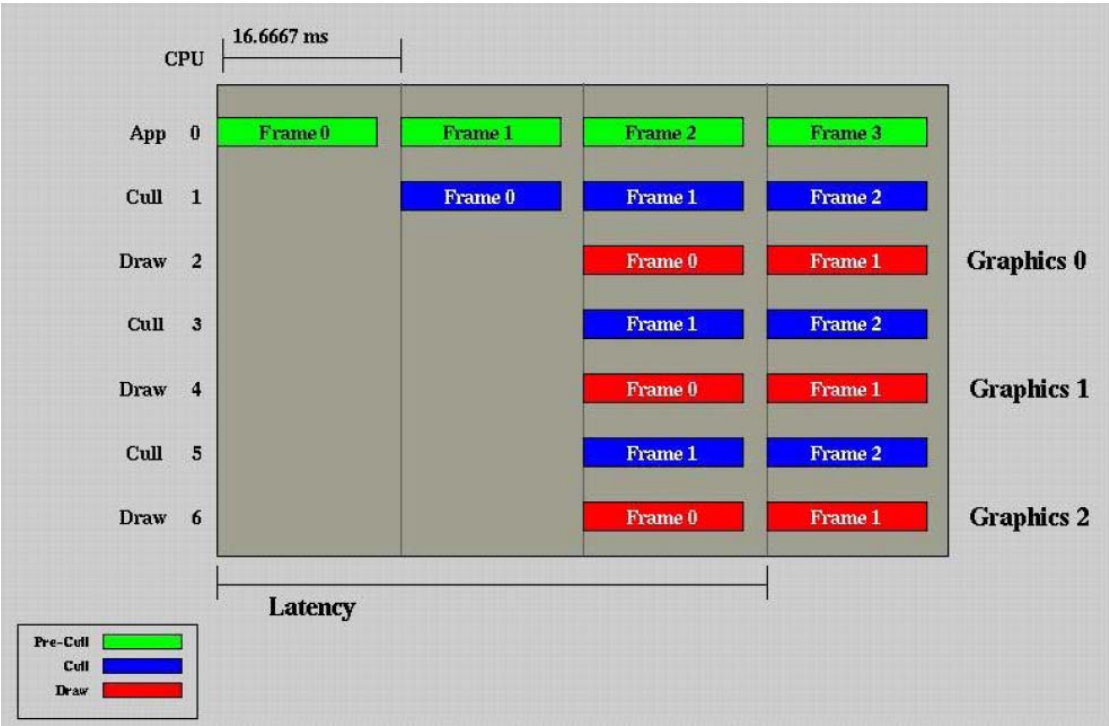
鉴于此，一个具备多图形子系统的系统，要实现多任务的机制，则需要定义过程如下：一个单处理器的系统需要顺序地执行各个阶段（例如，APP，CULL_0，DRAW_0，CULL_1，DRAW_1，CULL_2，DRAW_2），而一帧的时间也就相当于这些阶段耗费的总时间。因此，我们需要实现的任务主要有两个：（1）唯一的 APP 任务，（2）每个图形子系统各自的 CULL/DRAW 任务。



在多处理器系统中，如果处理器树木足够多的话，那么每一个任务都可以在某一个处理器上并行地执行。此外，CULL/DRAW 任务也可以分离为两个任务并行地运行。

并行多处理器环境的模型实现主要有两个目标。（1）分离并行化（Task Division Parallelization）：将一个大型的任务分解成多个可以并行运行的小型任务，以削减运行时间；（2）集成并行化（Task Aggregation Parallelization）：将一个任务 N 次倍乘后，并行地运行它的每个实例，而不增加运行时间。将 CULL/DRAW 阶段从 APP 阶段分离出来，然后将 CULL 和 DRAW 分离成独立并行的任务，这是一个“分离并行化”的例子。将多个 CULL/DRAW 组合的任务添加给各个图形子系统，这是一个“集成并行化”的例子。

并行地运行各个阶段可能会带来一些问题。首先，各个阶段必须顺序地处理数据。换句话说，APP 阶段完成对数据的处理之后，CULL 阶段才可以开始使用这些数据。同样，DRAW 阶段也不可以在 CULL 完成数据生成之前使用这些数据。不过，APP 阶段不需要等待 CULL 和 DRAW 阶段的工作完成，就可以开始下一帧的数据处理工作，因此，系统管线流程的设计如下图所示。



此外，各个阶段之间共享的数据也需要进行保护和缓存。由之前的阶段写入的数据，不可以被同时发生的其它并行阶段读取。这样场景图形软件就需要引入复杂的数据管理功能。

以上所述是 SGI Iris Performer 在九十年代提出的一种框架结构。在当时它是合适的，但是现在已经有些过时。

实时且具备窗口输出（out-the-window）的飞行模拟系统需要 60Hz 的帧速率，也就是说，每个阶段只有 16.667 毫秒的时间来完成其任务。1990 年，SGI 开始开发实时图形系统，它所使用的处理器速度是现在处理器的 1/60。由于图形系统与图形处理器的能力成比例相关，APP 和 CULL 阶段所需的时间负荷并不是相同的。上图所示的系统设计中，APP 和 CULL 阶段被假定可能占用整整一帧的时间进行处理。

后来，系统频宽的增长降低了基于本地的图形分配的消耗，而 DRAW 阶段需要分离成两个独立的执行线程：一个在主机上运行，另一个在图形子系统上运行。这一改变将在后面的部分详述。

最后一个需要提及的话题是等待时间（latency，即获得系统任何形式响应所需的最小时间）。飞行模拟系统可以允许有大约三帧的视觉反应延迟时间。这一时间延迟是符合真实人体行为研究的结果的，因此我们不得不参照上述过程模型的要求进行折衷的设计。

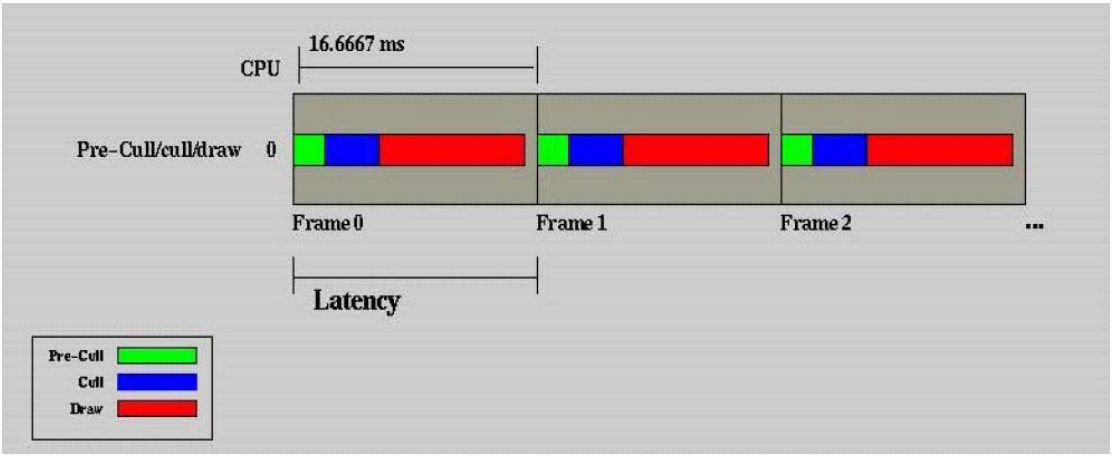
新的实现方案

在目前的硬件条件（以 60Hz 为帧速率标准）下运行的大多数应用程序，其 CULL 预阶段（即 APP 阶段）和 CULL 阶段的处理时间分别限制在少于 1 毫秒和少于 3-5 毫秒的范围内。这样的话，各个阶段要各自占用完整的一帧，或者占用一个完整的 CPU 时间，恐怕是浪费且不切实的。下面的图表反映了这一事实。

有一种提议是，交由 CULL 预阶段和 CULL 阶段来执行复杂的任务，以增加了它们的

运行时间。但是，大多数执行复杂操作的应用程序任务，其更好的实现方案是与帧的实现部分同步运行。

首先我们考虑单处理器，单图形子系统的系统模型。当 CULL 预阶段和 CULL 阶段的计算需求降低时，各阶段的运行相位可以如图进行设计：



这一方案的好的方面是，所有的三个阶段可以在一帧执行，响应时间也只有一帧。不利的方面是，DRAW 阶段分配的时间较以前大大减少，其启动时间也在一帧的中部位置。使用场景图形的另一个益处是，CULL 阶段可以去除不可见的场景，以降低主机图形子系统的带宽压力，同时它还负责按照状态变化值对所有对象进行排序，以优化图形流水线的性能。

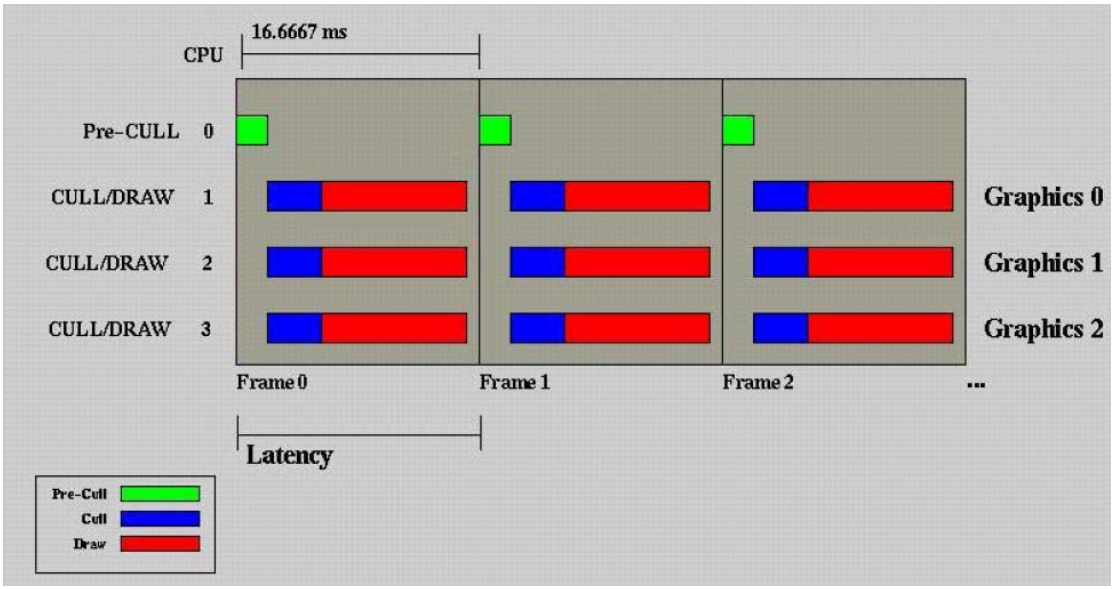
由于系统带宽和图形性能的不断提高，那些运行在老式硬件平台下的应用程序，其需求可能较以前有所降低。分配给渲染的时间也许是充足的。这样的话，场景图形系统也就不必再考虑数据保护和管理上的特殊需要了。

现在我们考虑多图形子系统，多处理器的系统模型。为了使用多处理器系统的优势，我们需要设置一个主线程，用于运行 CULL 预阶段的任务，并为每个图形子系统设置 CULL/DRAW 线程。为此，我们要针对数据管理考虑以下两个方面：

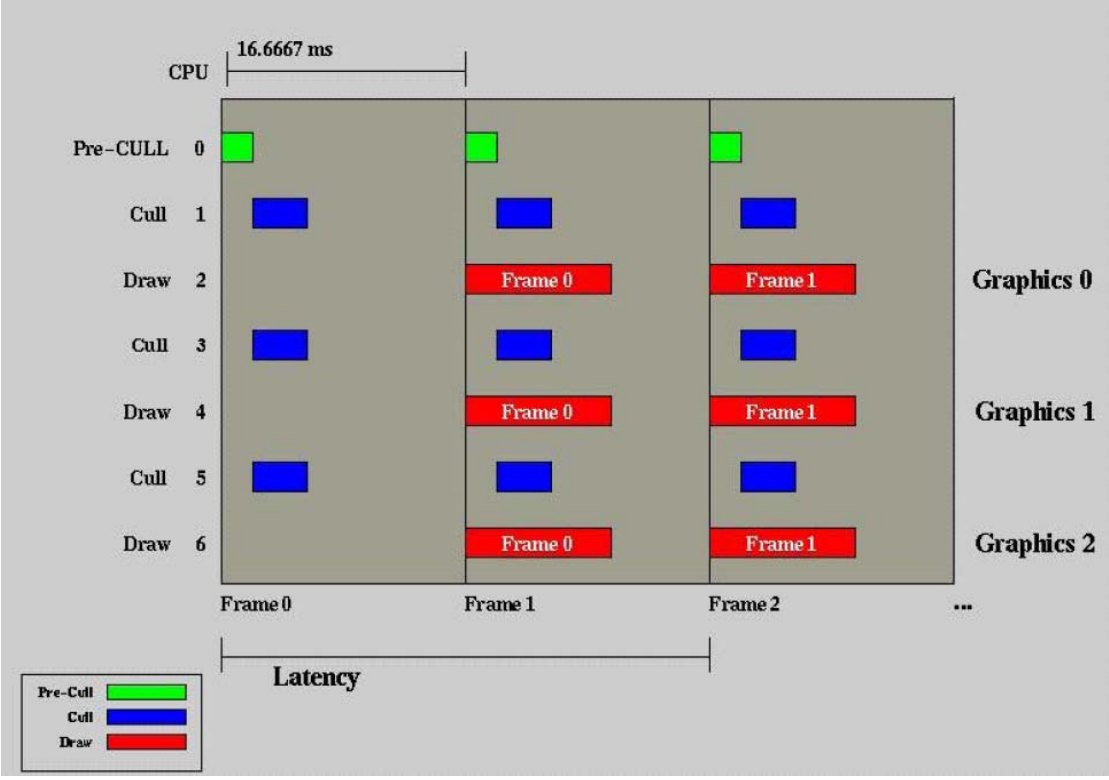
（1）CULL 预阶段的公共数据写入。

（2）CULL 阶段生成的内部数据，分别复制到各个 CULL/DRAW 阶段中。

然后，我们才可以安全地执行各个阶段，如下图所示：



我们已经解决了集成并行化所存在的问题，但是还没有解决 DRAW 阶段少于一帧时间的问题。我们必须将 CULL 和 DRAW 阶段分割成不同的处理线程来实现这一目标。因此我们需要考虑如何保护和缓存数据，这些数据由 CULL 阶段生成，由 DRAW 阶段处理。下面的部分将讨论这一主题，其阶段图如下所示。

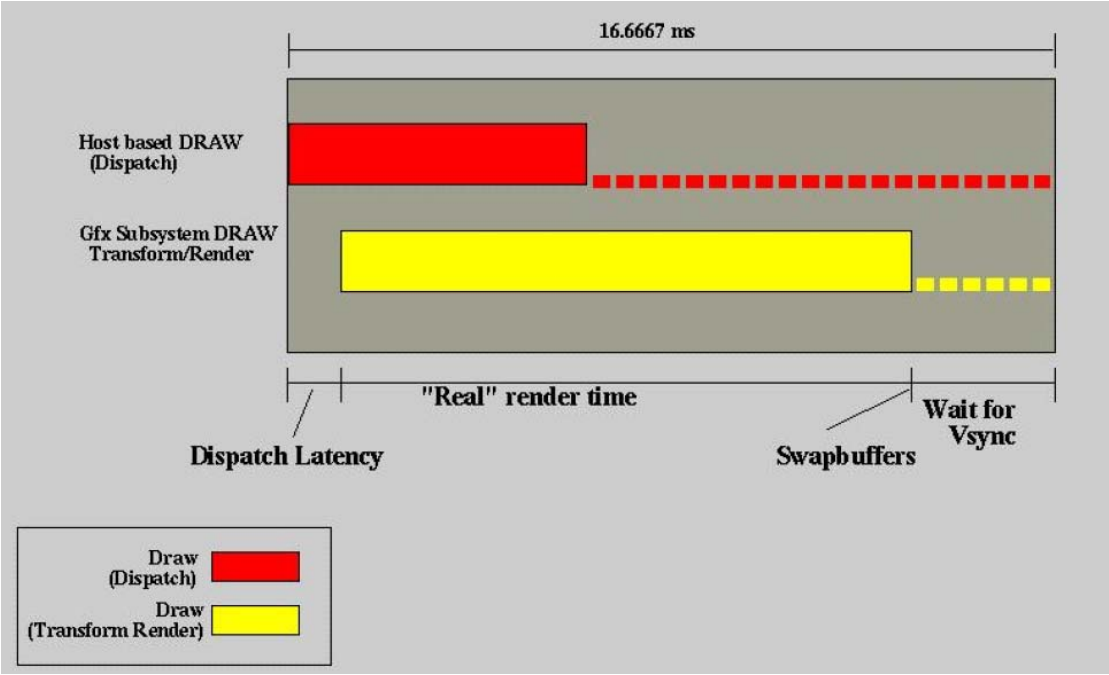


对于硬件厂商来说，上图所示的情景一定是十分诱人的，因为它使用了 7 个 CPU 来控制 3 个图形子系统。而且还有一种说法是，如果保留 CPU 0 来执行操作系统任务，从 CPU 1 开始执行仿真任务，那么我们还需要第八个 CPU。但是，对于工程人员来说，上图中存在了太多的空闲空间。同时，我们还增加了每一帧的响应时间。不过这还是好过旧式模型的三帧的响应时间。

主机绘制 vs. 图形子系统绘制

到这里为止，我们一直都把 DRAW 作为一个独立的阶段，或者一个独立的线程（进程）进行介绍。在旧式的系统上，由于绘制（DRAW）过程受到主机向图形子系统传输的带宽和图形处理速度的影响，这一工作模型应当说是合理的。但是如今，我们需要认识到，当 DRAW 阶段运行于主机的某个专有 CPU 上时，它同时与图形子系统上的另一个并行处理器产生交互。OpenGL 程序所做的仅仅是封装了 OpenGL 协议（以数据和信息流的方式），并传递给图形子系统，后者处理数据和信息流的内容并执行实际的矩阵变换，并实现结果的渲染。主机的绘制（DRAW）过程略微提前于图形子系统的绘制过程开始，并略微提前（有时可能会大幅提前）结束处理。通过使用基于主机的计时工具来进行图形基准测试，即可获得这一结论。

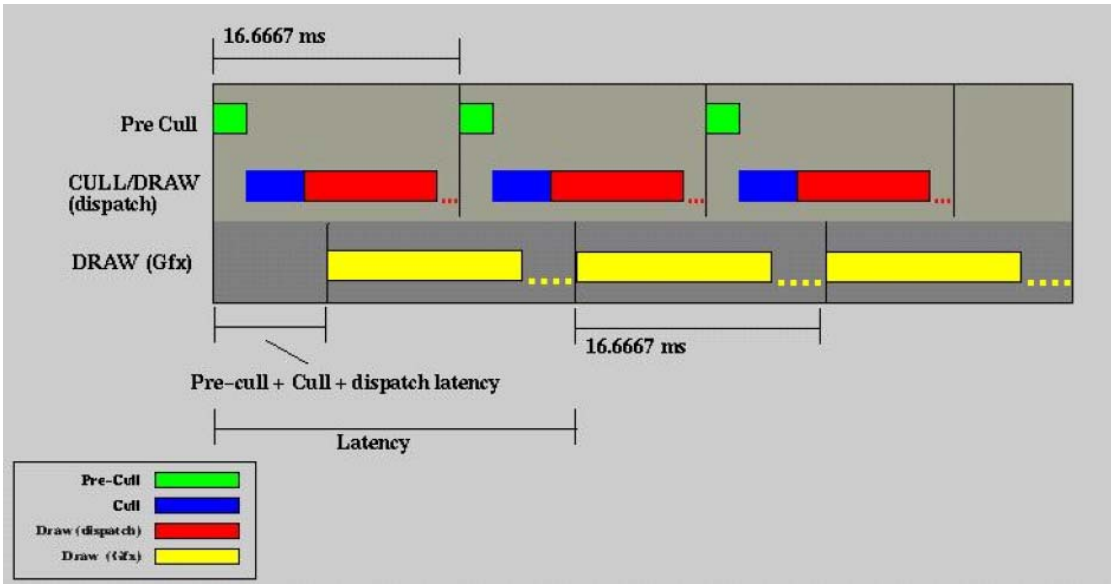
下图所示为主机 DRAW（也称作分派）阶段和图形子系统 DRAW 阶段的实时运作流程



从上图可以看出，一帧时间内，主机 DRAW（分派）过程在帧的边界开始。而主机 DRAW 分派 OpenGL 调用和图形子系统开始处理这些调用之间的时间区域，被称作分派时间（Dispatch latency）。黄色的条带表示图形子系统完成数据流的读入和处理，完成矩阵变换，渲染，并执行渲染缓存的交换所需的时间。由于交换缓存在下一次垂直回描（vertical retrace）消隐之前不会开始，因此图形子系统在这段时间里处于空闲。

注意，由于 DRAW 分派过程在图形子系统的处理完成之前已经结束。为了实现应用程序与图形子系统的同步，大多数主流的图形软件都会选择在下一帧之前等待一个指示“交换缓存已经执行”的信号。这可以说是一个优化主机运行时间的机会。

综上，再考虑到主机和图形系统的并行特性，我们可以设想如下图所示的过程模型。



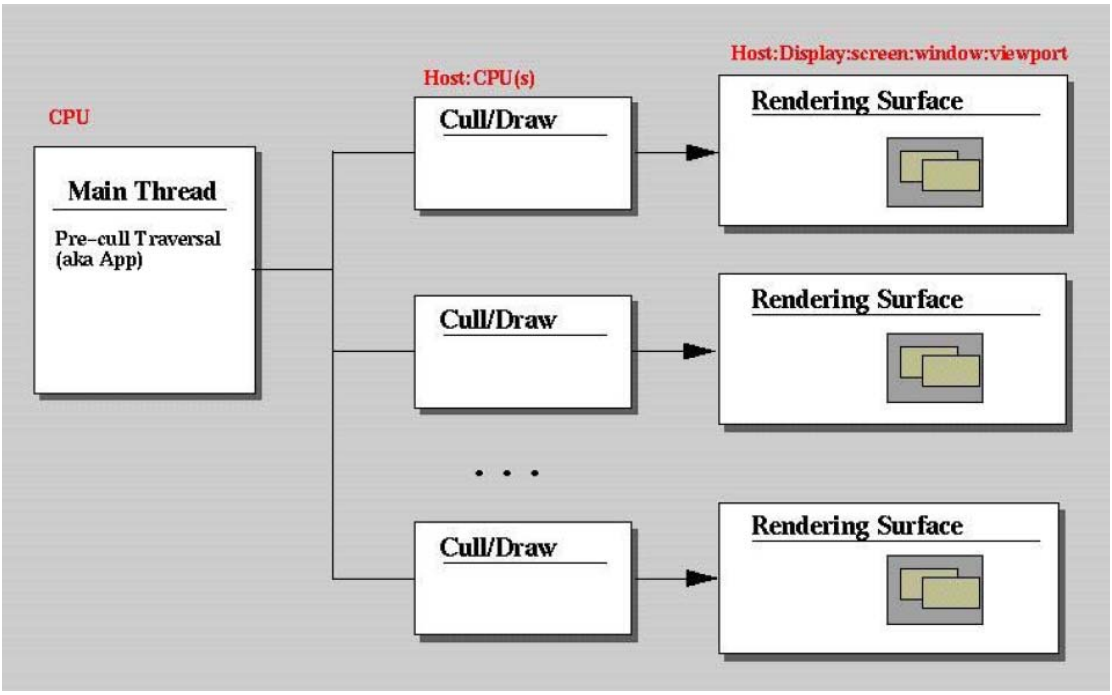
在这个模型中，我们根据图形子系统垂直回描信号的精确时间，规划主机上的帧调度工作。不过，我们可以控制时间产生轻微的摆动，这样就可能在垂直回描之前，在主机上开始

新的一帧。当垂直回描信号产生，同时图形子系统的处理重置之时，我们在主机上完成 CULL 预阶段，CULL 阶段，并向下传递由主机 DRAW 过程生成的 OpenGL 协议。这一工作的起始时间与图形子系统的帧边界尽量贴近。注意，CULL 和 DRAW（分派）位于同一线程中，执行串行处理。其结果是，原本用于等待垂直回描信号而浪费的主机时间，现在得到了有效的节约。

这一模型意味着场景图形的内存管理计算强度得到了改善，同时给图形系统的 DRAW 阶段提供了最大的渲染时间。此外，响应时间也降低到少于两帧。

开放场景图形多处理器模型的设计

开放场景图形多处理器（Open Scene Graph Multi Processor）模型的设计如图所示。



图中所示的矩形块表示的是一种抽象的概念，它不是与硬件紧密结合的，也无法直接加以实现。模型的实现方法将在本文中陆续给出。红色的字体表示模型配置文档和实现时所用到的专有名词。线和箭头表示数据在系统中的流向，最终完成在显示器上的渲染。

主线程（Main Thread）

主线程是运行 CULL 预阶段的进程或线程。其内容包括了用于运行此线程的 CPU。我们假设主线程在它被触发的主机上开始运行。我们使用配置管理器来启动并初始化上图的每一项内容，而主线程则运行在配置管理器所运行的主机上。

拣选/绘制线程

拣选（CULL）/绘制（DRAW）可以作为一个线程来执行，也可以像之前章节所述的那样，运行于不同的线程上。它可以指定两个参数：一个是系统运行的主机名称，另一个是在该主机上负责调度线程的 CPU 序号。如果 CPU 的数目为复数（不大于 2），那么我们假设拣选/绘制线程可以作为不同的线程来运行。

渲染表面

渲染表面描述了最终渲染结果显示的屏幕空间。其中定义了

- 主机（Host）：执行显示的系统所在的主机名称；
- 显示设备（Display）：即图形子系统，此处假设在 XWindow 系统下使用显示设备；
- 屏幕（Screen）：此处假设在 XWindow 系统下使用屏幕；
- 窗口（Window）：此处假设在 XWindow 系统下使用窗口；
- 视口（Viewport）：视口指的是窗口内的一个矩形区域，用于安置渲染的最终结果。

以上所述在配置文档中均为实际的实现细节。

配置

上面所述的内容可以按照三个独立的环境进行配置。

（1）单系统映像（Single System Image）

如果主机域名始终保持不变，那么我们的系统将在同一个主机上进行初始化。然后我们就可以根据 CPU 域的定义来设置线程的参数。

（2）图像集群（Graphics Cluster）

如果 CULL/DRAW 阶段所在的主机域与 CULL 预阶段所在的主机域不同，那么在 CULL/DRAW 的主机上需要启动一个 CULL 预阶段的代理器，它用于执行 CULL 预阶段（另一台主机上）生成的动态数据集的同步。如果数据同步没有完成，那么这个代理器会阻塞 CULL 阶段的运行。

（3）WireGL 设置

渲染表面包括一个“主机”域。它可以用于实现 WireGL（一种集群渲染系统）的执行，以处理主机 DRAW 阶段传递的 OpenGL 协议。这种配置调度的方便之处在于，它允许上述配置之间互相“混合搭配”（mix-and-match）。例如，某个应用程序可以在三个本地图形子系统上运行其窗口输出的渲染，同时为仿真工作站（Instructor Operator Station）提供多集群的显示，并在 WireGL 集群系统上实现各个显示结果的最终合成。

多处理器（MP）模型

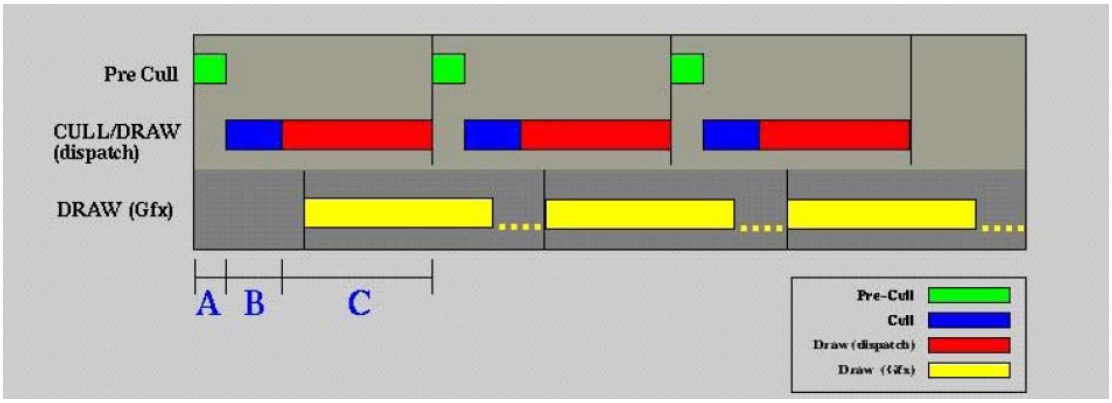
前面的章节叙述了两种类型的 MP 模型，用于实现多任务，多显示的开放场景图形系统

的实现。其不同点可以归结为 CULL/DRAW 作为一个线程还是分开处理的问题。如果考虑到同一主机上，存在移相的帧的调度，那么把 CULL/DRAW 分别进行处理可能就是不妥的。此外，忽略其带来的优越性的话，上述实现方法所引入的内存管理问题也可能导致性能降低。

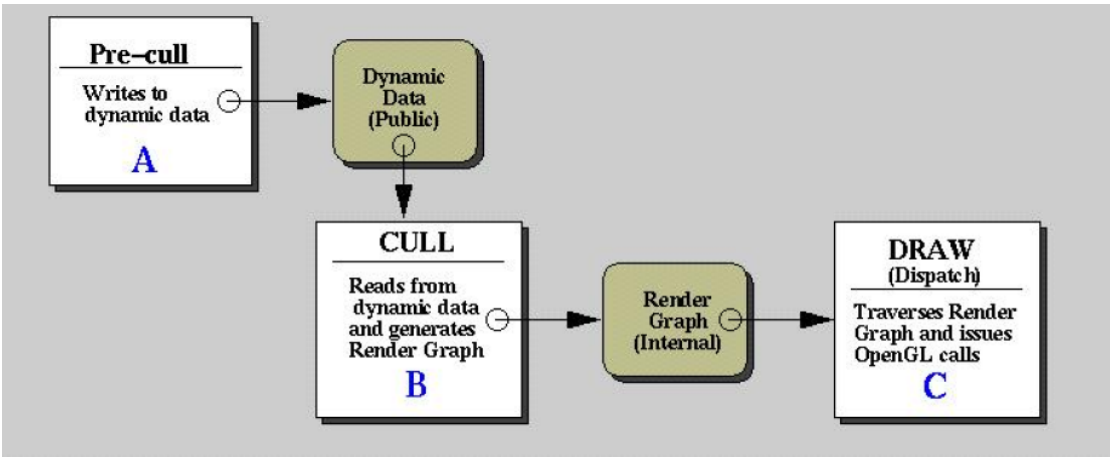
不过，我们仍然会在这里依次讨论这两种模型。

MP 模型 A - 数据流

如前面的章节所述，这里我们假设一个单一的，基于主机的 APP/CULL/DRAW 流水线，注意这里可能存在多个 CULL/DRAW 过程。



这个模型中假设有一个基于主机的可微调的帧调度机制，一个单一的线程来执行 CULL/DRAW。时间线 A，B，C 表示下一幅图中数据流的活动时间。



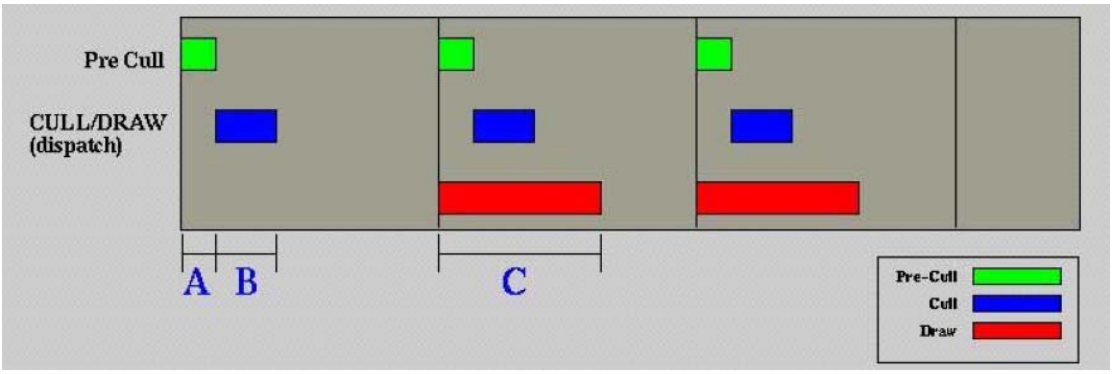
如前文所述，CULL 预阶段负责更新场景图形中的动态数据。这些动态数据包括摄像机位置，场景中的移动物体位置，时间戳，帧数，消耗时间，以及其它一些数据管理的参量。我们假设这些数据已经被正确分配，且都是应用程序可以访问的公共量。当 CULL 预阶段结束时，它向 CULL 阶段发送一个信号，使其进入运行状态。CULL 负责读取更新的动态数据，并生成内部的数据（应用程序无法访问它们），供 DRAW 阶段使用。这些数据是串行进行处理的。DRAW 阶段将遍历这些内部数据并传递相应的 OpenGL 调用。

这个模型较为简练，它只需要简单地实现主机上帧发生相移（phase shifted）时调度的实时性即可。OpenSceneGraph 本身已经包括了多显示系统中，对多重渲染上下文（rendering context）的支持。

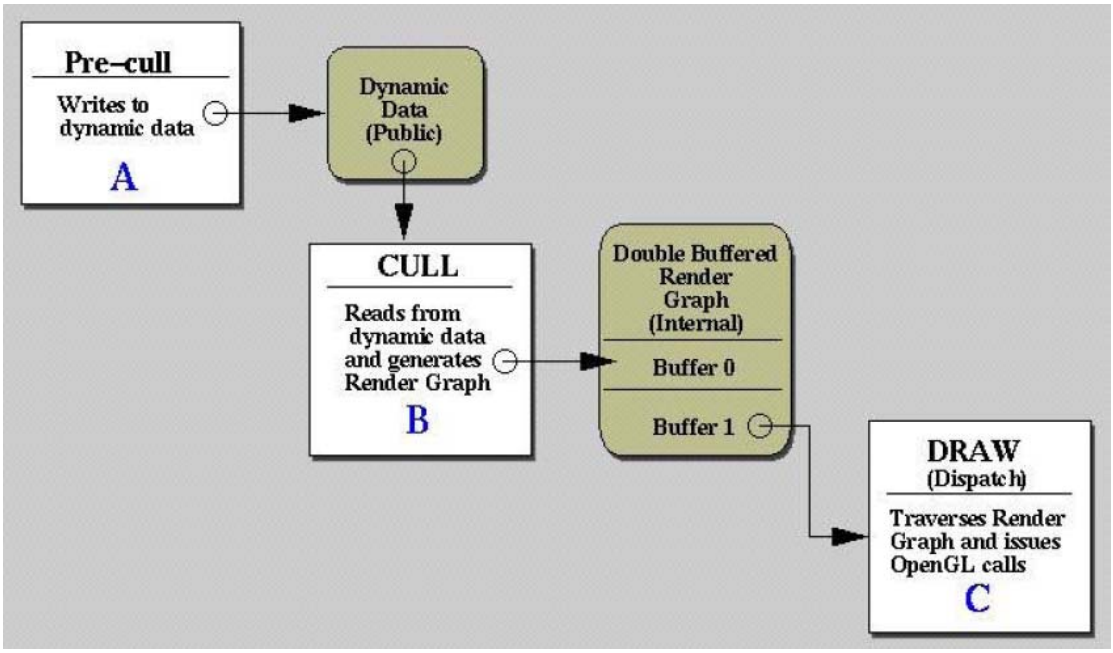
当 CULL/DRAW 作为一个线程运行时，不需要做特殊的更改。

MP 模型 B - 数据流

下面我们讨论 CULL/DRAW 分别在不同线程上运行的情况。注意，下图中并没有包括图形子系统的 DRAW 部分。此模型假设不存在相移，且主机已经处理了与图形子系统的同步问题。



此模型的数据流程如下图所示。



上图与单线程 CULL/DRAW 过程图的区别在于，从 CULL 传递到 DRAW 的内部数据需要经过双缓存的过程。CULL 生成的数据将被写入“缓存 0”，此时 DRAW 从“缓存 1”读取数据。到达 CULL 和 DRAW 线程的同步点时，执行两个缓存的交换。

这种方法需要编写内部数据的双重缓存，以及 CULL 和 DRAW 线程的同步位置实现代码。

总结

OpenSceneGraph 的设计用于实现多任务，多处理器和多显示的功能。它的实现方法是先进的，并且充分利用了现有硬件环境的优势。开放场景图形（Open Scene Graph）已经在 SGI 的 MPK 上测试成功，执行结果令人满意。开放场景图形的开发者迫切希望实现一个跨

平台的，灵活、透明地执行于图形集群（graphics cluster）上的解决方案。在了解了目前的困难之后，相信一款多显示，多处理器的实时开放场景图形系统将会在不久之后诞生。

原文参见: http://andesengineering.com/OSG_ProducerArticles/OSGMP/index.html