



(12) 发明专利

(10) 授权公告号 CN 102077251 B

(45) 授权公告日 2014. 01. 08

(21) 申请号 200980124880. 0

(22) 申请日 2009. 06. 26

(30) 优先权数据

12/163, 734 2008. 06. 27 US

(85) PCT国际申请进入国家阶段日

2010. 12. 24

(86) PCT国际申请的申请数据

PCT/US2009/048960 2009. 06. 26

(87) PCT国际申请的公布数据

W02009/158679 EN 2009. 12. 30

(73) 专利权人 微软公司

地址 美国华盛顿州

(72) 发明人 M·V·奥内珀 C·佩普尔

A·L·布里斯 J·L·拉普

M·M·莱西

(74) 专利代理机构 上海专利商标事务所有限公

司 31100

代理人 胡利鸣 钱静芳

(51) Int. Cl.

G06T 15/00(2011. 01)

G06F 9/44(2006. 01)

(56) 对比文件

US 2004/0237074 A1, 2004. 11. 25, 说明书 [0008]、[0040]、[0053] 段。

US 6704927 B1, 2004. 03. 09, 说明书摘要及说明书第 6 栏第 44-50 行。

US 6654951 B1, 2003. 11. 25, 说明书第 3, 12 栏的第 1 段 - 最后一段及附图 2。

US 2004/0237074 A1, 2004. 11. 25, 说明书 [0008]、[0040]、[0053] 段。

US 2007153015 A1, 2007. 07. 05, 全文。

审查员 苏菲

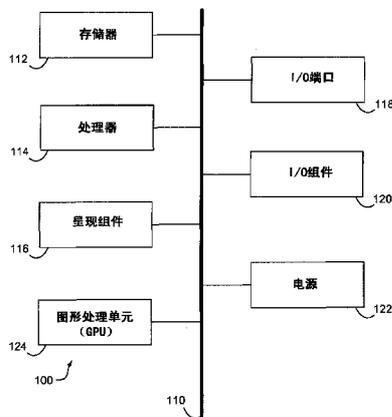
权利要求书2页 说明书10页 附图3页

(54) 发明名称

着色器接口

(57) 摘要

本文描述了处理器对用于着色器的存储器寄存器的分配。对于每一着色器，寄存器基于着色器的复杂度水平来分配。较简单的着色器实例限于较小数量的存储器寄存器。较复杂的着色器实例被分配较多的寄存器。为此，开发人员的高级着色 (HLSL) 语言包括稍后可由复杂或简单版本的着色器替换的着色器的模板类。HLSL 被转换成可用于光栅化计算设备上的像素的字节代码。



1. 一种用于将存储器寄存器分配到着色器实例的方法,所述方法包括:
 - 声明用于定义着色器模板的一个或多个接口,其中着色器的一个或多个类依赖于所述一个或多个接口(302);
 - 在着色器程序中定义用于内联实际着色器实例的变量(304);
 - 用所述实际着色器实例来替换所述着色器程序中的调用(308);
 - 将简单着色器实例和复杂着色器实例绑定到流水线,其中绑定包括将所述简单着色器实例和所述复杂着色器实例两者的每次使用存储在按函数类型和调用位置排列的表中,其中为与所述简单着色器实例或所述复杂着色器实例中的至少一个相关联的着色器代码中调用函数的每一位置发出所述函数的新版本。
2. 如权利要求 1 所述的方法,其特征在于,还包括为所述着色器实例分配存储器。
3. 如权利要求 2 所述的方法,其特征在于,与所述存储相关联并被分配给所述着色器实例的寄存器的数量取决于所述着色器实例的复杂度水平。
4. 如权利要求 1 所述的方法,其特征在于,还包括:
 - 将所述着色器实例转换成字节代码;以及
 - 基于所述字节代码光栅化计算设备上的像素。
5. 如权利要求 4 所述的方法,其特征在于,所述着色器实例由高级着色语言(HLSL)编译器编译成字节代码。
6. 如权利要求 1 所述的方法,其特征在于,所述着色器实例用高级着色语言(HLSL)来编程。
7. 如权利要求 1 所述的方法,其特征在于,所述着色器实例用于确定三维(3D)图形中的操作。
8. 如权利要求 1 所述的方法,其特征在于,还包括:
 - 提供编译的着色器字节代码;
 - 提供指向应用程序接口的指针;
 - 提供指向像素着色着色器的指针;
 - 向图形处理单元(GPU)的设备驱动程序提供所述着色器字节代码;以及
 - 返回对所述着色器实例的引用。
9. 如权利要求 8 所述的方法,其特征在于,还包括:
 - 引用所述着色器的一个或多个类;
 - 将所述一个或多个类存储在存储器位置中;以及
 - 指定指向所述存储器位置的指针。
10. 如权利要求 1 所述的方法,其特征在于,所述流水线由图形处理单元(GPU)执行。
11. 如权利要求 10 所述的方法,其特征在于,所述流水线同时并行地执行多个着色器实例。
12. 如权利要求 1 所述的方法,其特征在于,替换所述着色器程序中的调用还包括:
 - 接收所述着色器的两个或更多着色器实例中的一个;以及
 - 用所述着色器的两个或更多着色器实例中的一个来替换所述着色器程序中的调用。
13. 一种用于将存储器寄存器分配到着色器实例的系统,所述系统包括:
 - 用于声明用于定义着色器模板的一个或多个接口的装置,其中着色器的一个或多个类

依赖于所述一个或多个接口(302)；

用于在着色器程序中定义用于内联实际着色器实例的变量的装置(304)；

用于用所述实际着色器实例来替换所述着色器程序中的调用的装置(308)；以及

用于将简单着色器实例和复杂着色器实例绑定到流水线的装置，其中绑定包括将所述简单着色器实例和所述复杂着色器实例两者的每次使用存储在按函数类型和调用位置排列的表中，其中为与所述简单着色器实例或所述复杂着色器实例中的至少一个相关联的着色器代码中调用函数的每一位置发出所述函数的新版本。

14. 如权利要求 13 所述的系统，其特征在于，还包括：

用于提供编译的着色器字节代码的装置；

用于提供指向应用程序接口的指针的装置；

用于提供指向像素着色着色器的指针的装置；

用于向图形处理单元(GPU)的设备驱动程序提供所述着色器字节代码的装置；以及

用于返回对所述着色器实例的引用的装置。

15. 如权利要求 13 所述的系统，其特征在于，基于与所述着色器实例相关联的复杂度来确定要分配到所述着色器实例的寄存器的数量。

16. 如权利要求 13 所述的系统，其特征在于，所述着色器实例用于确定三维(3D)图形中的操作。

17. 一种由计算设备实现的用二维再现三维图形的相似度的方法，所述计算设备包括带有一个或多个存储器寄存器的存储器单元(112)，所述方法包括：

图形处理单元 GPU (124)基于先前定义的着色器类来分配与所述着色器相关联的所述一个或多个存储器寄存器，包括：

所述 GPU 声明用于定义着色器模板的一个或多个接口，其中着色器的一个或多个类依赖于所述一个或多个接口；

所述 GPU 在着色器程序中定义用于内联实际着色器实例的变量；

所述 GPU 用所述实际着色器实例来替换所述着色器程序中的调用；以及

所述 GPU 将简单着色器实例和复杂着色器实例绑定到流水线，其中绑定包括将所述简单着色器实例和所述复杂着色器实例两者的每次使用存储在按函数类型和调用位置排列的表中，其中为与所述简单着色器实例或所述复杂着色器实例中的至少一个相关联的着色器代码中调用函数的每一位置发出所述函数的新版本。

18. 如权利要求 17 所述的计算设备实现的方法，其特征在于，所述 GPU 执行用于基于所述实际着色器实例来着色像素的光栅化器。

19. 如权利要求 17 所述的计算设备实现的方法，其特征在于，所述实际着色器实例确定一个或多个图元的位置。

着色器接口

[0001] 背景

[0002] 如今的图形处理单元 (GPU) 主存要在计算机屏幕上生成高质量图形所需的所有计算,从而让计算机设备的中央处理单元 (CPU) 可用于其他任务。具体地,GPU 通过处理多个被称为“着色器”的程序来在计算机屏幕上再现图形。简言之,着色器是执行用于再现二维 (2D) 或三维 (3D) 图形的操作的专用计算机程序。在现代 GPU 中,真实的场景是通过用由着色器控制的各种虚拟材料再现几何形状来生成。这些材料在着色器程序代码中表示,着色器程序代码处理各种输入 (包括纹理地图、光位置和其他数据) 来生成视觉结果。通过使用着色器,开发人员可通过结合不同的顶点着色、图元着色和像素着色来实质上控制任何图形或图形效果。

[0003] 当前用于实时地再现复杂的 3D 图形场景的方法包括结合自定义逻辑单元支持并行体系结构的处理器通过将开销分布在多个并行单元上来隐藏等待时间。在提供了对如点、线段、或三角形等线性图元集合的高级 3D 描述时,围绕将该集合转换或光栅化为投影的像素表示的图元光栅化流水线来设计所使用的流水线。在现有 3D 硬件技术中,被称为“着色器”的小程序用于定义再现算法的某些阶段的操作,如变换图元的顶点或计算屏幕上单个像素的颜色。着色器定义要在大型并行执行批中执行的、常常跨在图形处理单元 (GPU) 上的许多专用处理器分布的少量工作。

[0004] 着色器的创建通过被设计成可以硬件体系结构可用为目标的高度专用编程语言来完成,并且等效编译器可处理代码并将其简化为硬件和相关联的设备驱动程序可使用的指令。开发人员使用这种技术以便将再现流水线定制为仅是特定应用程序所需的行为。例如,如果开发人员正在创建执行对非常复杂的主题的非照片般逼真的 3D 再现的应用程序时,开发人员可将着色器优化成非常简单以便最大化场景的复杂度。相反,如果开发人员希望将非常高保真度的材料属性和光照应用于较不复杂的场景,则开发人员可创建高度定制化的着色器来创建可能非常复杂的非常真实的效果。此外,着色器被编译成设备驱动程序映射其以供硬件运行的抽象二进制形式。

[0005] 为了说明这点,考虑其中人物暴露于多个光源的游戏场景。光源中的一个可以是简单地来自夜晚的月亮的环境光。另一个光源可以从沿着街道的灯柱延伸的。有了来自月亮的第一光源,可将着色器编写成控制从月亮发射的光。在这种情况下,光是恒定的且只需要由简单的程序来表示以将光分散到整个场景中。然而,灯柱可能较复杂。有了灯柱,光可被配置成仅在特定方向上发光;然而,来自灯柱的光可能无法绕过转角。因此,被编写成控制来自灯柱的光的着色器可能需要比被编写成控制来自月亮的光的着色器更复杂的计算。在任一场景中,GPU 必须根据来自每一着色器的底层计算来光栅化像素。

[0006] GPU 的常见体系结构通过使系统上的资源变得灵活来提供场景复杂度和着色器复杂度之间的折衷。为了执行,着色器通常需要处理单元、着色器实例的数据、全局资源 (例如,纹理图像)、用于执行计算的中间寄存器库和一组输出寄存器。对于简单着色器——意味着着色器需要相对较少的寄存器来计算,可同时运行多得多的着色器,从而导致底层应用程序或游戏得到较高的帧速率,因为可以并行地完成更多的工作。对于较复杂的着色

器——意味着着色器需要较多寄存器来计算,可并行地执行较复杂的着色器的较少实例,因为正在使用较多的寄存器。换言之,对寄存器的分配对可并行地处理的着色器的数量具有直接决定权。因为再现图形所需的时间取决于着色器的并行处理,所以尽可能多地处理着色器是有益的,且因此对寄存器的分配对于性能而言是关键的。

[0007] 概述

[0008] 提供本概述是为了以精简的形式介绍将在以下详细描述中进一步描述的一些概念。本概述不旨在标识所要求保护的的主题的关键特征或本质特征,也不旨在用于帮助确定所要求保护的的主题的范围。

[0009] 本发明的一个方面允许单个着色器具有改变其中的复杂度的替换路径。在这一方面,按照允许高效的寄存器分配的方式提供了对通过着色器的特定路径的选择。存储器寄存器基于着色器路径或实例的复杂度水平来分配。在一个实施例中,着色器实例是着色器开发人员所开发的着色器程序或其部分。较简单的着色器路径或实例可限于较少数量的存储器寄存器。较复杂的着色器路径或实例可被分配较多的寄存器。本发明的另一方面涉及被配置成按此方式分配存储器寄存器的 GPU。

[0010] 若干附图的简述

[0011] 下面参考附图详细描述本发明,附图中:

[0012] 图 1 是供在实现本发明的一实施例时使用的示例性操作环境的框图;

[0013] 图 2 是示出根据本发明的一实施例的对计算设备的存储器寄存器的分配的示图;以及

[0014] 图 3 是根据本发明的一实施例的用于基于着色器复杂度来分配寄存器的流程图。

[0015] 详细描述

[0016] 用具体细节呈现此处所描述的主题以满足法定要求。然而,此处的描述并非旨在限制本专利的范围。相反,可以设想,所要求保护的的主题还可结合其它当前或未来技术按照其它方式来具体化,以包括不同的步骤或类似于本文中所描述的步骤的步骤组合。此外,尽管术语“框”可在此处用于指示所采用的方法的不同元素,但该术语不应被解释为暗示此处公开的各个步骤之中或之间的任何特定顺序。

[0017] 此外,在本说明书全文中使用各种技术术语。这些术语的定义可以在 H. Newton 编写的 Newton' s Telecom Dictionary(《牛顿电信词典》)(2005 年第 21 版)中找到。这些定义旨在提供对本发明中所公开的的想法的更清楚的理解但不旨在限制本发明的范围。这些定义和术语应该在允许任何在上述参考文献中所提供的词的的含义的意义上宽泛地和不受限制地解释。

[0018] 本发明可一般地被描述为用于提供动态代码联接来基于着色器的复杂度水平最优地分配用于着色器的寄存器的一种或多种系统、方法和计算机存储介质。在一个实施例中,开发人员可创建其自己的着色器类并且基于复杂度将寄存器分配给每一着色器。较简单的着色器被分配较少的寄存器,而复杂的着色器被分配附加的寄存器。

[0019] 如本领域技术人员可以理解的,本发明的各实施例可尤其被具体化为:在一种或多种有形计算机可读介质上体现的方法、系统或计算机程序产品。因此,这些实施例可采用硬件实施例、软件实施例或者组合软件和硬件的实施例的形式。

[0020] 动态代码绑定提供了用于从函数的消费者中抽象函数的实现的一种机制,这种机

制通过提供函数调用和实现之间的间接水平来提供。通常,这种间接通过首先查看虚拟函数表来找到要执行的函数的位置来执行。在执行应用程序时,用函数实现的位置来填充先前为空的表(“链接”的实际动作),由此允许应用程序实际按需执行函数。在一个实施例中,提供了减少专用着色器的排列的数量同时仍然提供跨抽象边界的全局优化的子集动态联接。

[0021] 作为向每一抽象函数提供单个编译的实现的替代,实施例按照这样的方式来生成编译的代码:即编译对着色器的特定实例的每一次使用,就像是代码中内联的一样,并随后将其存储在按函数类型和调用位置排列的表中。理解以下是重要的:此处描述的各实施例与常规联接的不同之处在于:在运行时不使用调用约定。相反,每一次应该调用函数时,发出函数的版本来匹配调用点的寄存器状态和其他状态。因为为着色器代码中调用函数的每一位置发出函数的新版本,所以在内联时使用的所有优化都适用,除了该函数代码必须在功能上与主着色器代码保持分开之外。因为此处描述的各实施例与“真实”联接有所区别,所以此处描述的各实施例所生成的代码量可能相当大。在多个调用点之间不发生代码共享。如果代码大于代码高速缓存,则不隐藏源自高速缓存未命中的等待时间的惩罚。

[0022] 此处描述的某些实施例使用可选内联。可选内联允许系统生成不仅接近于最优指令使用而且为给定任务的每一调用使用最少所需寄存器的着色器实例。在该实施例中,特定着色器调用所需的寄存器的总数可由设备驱动程序快速地计算并相应地进行分配。这阻止了非常复杂的计算影响寄存器使用,除非计算实际上正在执行。

[0023] 为了维护优化,各实施例被配置成每调用点发出每个所用方法的不同版本,如同该方法是内联的一样以允许跨方法调用边界的优化。这具有空间的折衷,并且不像标准联接只为每一方法创建一个编译的版本,此处描述的各实施例创建许多版本,潜在可能导致较大的二进制文件。

[0024] 此处描述的各实施例一般参考 Windows®操作系统(OS)的各个版本中包括的 Direct3D API。各实施例不限于 Direct3D API。本领域技术人员将理解,不同 OS 中的各种 API 提供与此处描述的调用和例程类似的功能。然而,为了清楚起见,此处参考 Direct3D。

[0025] 在继续下一步之前,应该定义若干关键定义。虽然以下定义应该帮助读者理解此处描述的各实施例,但这些定义仅仅出于解释目的而提供。

[0026] “图元”是用于描述形状的基本单元。在计算机图形中,三角形通常被认为是基本图元,因为所有可能的 2D 和 3D 形状都可由三角形组成。如本领域技术人员将理解的,在再现图形时可替换地将其他形状用作图元。

[0027] “着色器”是执行再现计算的某一方面的小型专用计算机程序。着色器负责在典型再现流水线中的多个主要方面。这些方面特别地包括,顶点着色、图元(或几何)着色和像素着色。本领域技术人员将理解,顶点着色指的是确定图元的顶点的位置和定向——例如,将 2D 中的三角形的顶点放置在何处,从而使得该三角形表现为 3D。图元着色描述用于单个图元的表面操作。而像素着色基于所再现的图元来着色每一像素,以便将图元绘制到屏幕。

[0028] 可以开发或编程着色器来处理实质上游戏体验的任何方面。例如,可以编写着色器来基于人物的颜色、时间、光照或其他相关变量来控制人物的皮肤的光反射。如先前所描述的,某些着色器相对简单;而其他的可能需要较复杂的计算。在某些实施例中,简单着色器可能需要与复杂着色器相比更少的存储器寄存器来处理。

[0029] “光栅化器”是获取由诸如线、点和三角形等高阶图元组成的图像并将图像转换成光栅图像（即，像素）以供在视频显示器上输出的组件。光栅图像是带有颜色的图元的位图表示。在一个实施例中，光栅化器是被配置成根据着色器所产生的图元来对像素着色的由 GPU 执行的软件。

[0030] Direct3D 是 Windows® 中提供的用于再现 2D 和 3D 场景的应用程序编程接口 (API)。Direct 3D 包括带有可编程阶段的图元光栅化器，其允许开发员在 Windows® 平台内将定制的程序加载到 GPU 上以便进行再现。当前正在使用多个版本的 Direct3D，并因此应该为本领域技术人员所理解。

[0031] 高级着色语言 (HLSL) 是出于开发着色器目的而设计的编程语言（例如，C、C++、C#、Java 等）的变体。具体地，开发员用 HLSL 对着色器进行编程。在操作上，HLSL 被编译成与图形应用程序联用的中间语言 (IL)。

[0032] IL 是着色器阶段应该执行的操作的低级的基于指令的二进制表示。其用作编译器和图形硬件的本机指令集的中间优化步骤。因此，IL 将由开发员通过使用 HLSL 指定的指令转换成图形硬件（即 GPU）为了再现图形所需的字节代码。

[0033] 即时 (JIT) 指的是对着色器 IL 执行的用于将着色器转换成图形硬件的本机指令集的快速处理编译。JIT 简单地指代在编程和处理中动作何时发生的时间点。本领域技术人员将理解，JIT 编译如何在诸如 Python 和 C# 等其他编程语言中工作。

[0034] 在一个实施例中，本发明采用包括其上包含计算机可使用指令的一个或多个计算机可读介质的计算机程序产品的形式。计算机可读介质包括易失性和非易失性介质以及可移动和不可移动介质。作为示例而非限制，计算机可读介质包括计算机存储介质。计算机存储介质，即机器可读介质，包括以用于存储信息的任何方法或技术来实现的介质。

[0035] 存储的信息的示例包括计算机可使用指令、数据结构、程序模块以及其它数据表示。计算机存储介质示例包括，但不限于，随机存取存储器 (RAM)、只读存储器 (ROM)、电可擦除可编程只读存储器 (EEPROM)、独立于不同的存储介质或者与之相结合来使用的闪存，这些不同的存储介质诸如例如，紧致盘只读存储器 (CD-ROM)、数字多功能盘 (DVD)、全息介质或其他光盘存储、磁带盒、磁带、磁盘存储以及其他磁存储设备。这些存储器组件可瞬时、临时或永久地存储数据。

[0036] 在简要描述了此处描述的各实施例的概览后，以下描述一示例性计算设备。最初具体参考图 1，示出了用于实现本发明的示例性操作环境，并将其笼统指定为计算设备 100。计算设备 100 只是合适的计算环境的一个示例，并且不旨在对本发明的使用范围或功能提出任何限制。也不应该将计算设备 100 解释为对所示出的任一组件或其组合有任何依赖性要求。在一个实施例中，计算设备 100 是常规计算机（例如，个人计算机或膝上型计算机）。

[0037] 本发明的一个实施例可以在计算机代码或机器可使用指令的一般上下文中描述，计算机代码或机器可使用指令包括诸如程序模块等由计算机或其他机器执行的计算机可执行指令。一般而言，包括例程、程序、对象、组件、数据结构等的程序模块指的是执行特定任务或实现特定抽象数据类型的代码。此处所描述的各实施例可以在各种系统配置中实施，这些系统配置包括手持式设备、消费电子产品、通用计算机、更专用计算设备等。此处所描述的各实施例还能在其中任务由通过通信网络链接的远程处理设备完成的分布式计算

环境中实现。

[0038] 继续参考图 1, 计算设备 100 包括直接地或间接地耦合以下设备的总线 110: 存储器 112、一个或多个处理器 114、一个或多个呈现组件 116、输入 / 输出端口 118、输入 / 输出组件 120、以及说明性电源 122。总线 110 可以是一条或多条总线 (诸如地址总线、数据总线、或其组合)。尽管为了清楚起见用线条示出了图 1 的各框, 但是在现实中, 各组件的划界并不是那样清楚, 并且按比喻的说法, 更精确而言这些线条将是灰色的和模糊的。例如, 可以将诸如显示设备等的呈现组件认为是 I/O 组件。而且, 处理器具有存储器。本领域的技术人员可以理解, 这是本领域的特性, 并且如上所述, 图 1 的图示只是例示可结合本发明的一个或多个实施例来使用的示例性计算设备。在诸如“工作站”、“服务器”、“膝上型计算机”、“手持式设备”等等之类的类别之间不进行区别, 因为所有这些都都在图 1 的范围内并都被称作“计算设备”。

[0039] 计算设备 100 通常包括各种计算机可读介质。作为示例而非局限, 计算机可读介质可包括 RAM、ROM、EEPROM、闪存或其他存储器技术、CDROM、DVD 或其他光学或全息介质、磁带盒、磁带、磁盘存储或其他磁存储设备、或可被配置成存储与此处所描述的各实施例相关的数据和 / 或执行的类似的有形介质。

[0040] 存储器 112 包括易失性和 / 或非易失性存储器形式的计算机存储介质。存储器可以是可移动的、不可移动的、或其组合。示例性硬件设备包括固态存储器、硬盘驱动器、高速缓存、光盘驱动器等。计算设备 100 包括从诸如存储器 112 或 I/O 组件 120 之类的各种实体读取数据的一个或多个处理器。呈现组件 116 向用户或其他设备呈现数据指示。示例性呈现组件包括显示设备、扬声器、打印组件、振动组件等等。

[0041] I/O 端口 118 允许计算设备 100 在逻辑上耦合至包括 I/O 组件 120 的其他设备, 其中某些设备可以是内置的。说明性组件包括话筒、操纵杆、游戏垫、圆盘式卫星天线、扫描仪、打印机、无线设备等等。

[0042] 计算设备 100 还包括能够同时处理并行线程中的多个着色器的 GPU 124。为此, GPU 124 可被配备成具有各种设备驱动程序并且实际上包括多个处理器。

[0043] 图 2 是示出根据本发明的一实施例的对计算设备的存储器寄存器进行分配的图。如图 2 所示, 计算设备上有多个存储器寄存器 200 可用。并列地呈现相同的寄存器 200 的块来说明如何为简单着色器实例 202、204、206 和 208 以及复杂着色器实例 210 和 212 中的任一个分配存储器。实例可包括着色器的程序表示或着色器的字节代码重现中的任一个。此外, 对寄存器 200 的分配可由计算设备上的 GPU 或 CPU 来执行。

[0044] 简单寄存器 202、204、206 和 208 被分配到可用块 214、216、218 和 220。复杂着色器被分配到可用块 222 和 224。被分配到简单着色器的可用块 (即, 块 214、216、218 和 220) 包含比被分配到复杂着色器的可用块 (即, 块 222 和 224) 少的寄存器。

[0045] 在一个实施例中, 通过为各场景 (简单的和复杂的) 指定两个分开的着色器并在需要时加载适当的场景来在代码中分配寄存器的块。为此, 可以为给定着色器设置指针。

[0046] 图 3 是根据本发明的一实施例的用于基于着色器复杂度来分配寄存器的流程图。最初, 用 HLSL 声明定义了可从其实例化多个着色器类的模板的接口, 如 302 处所指示的。用于内联着色器实现的变量也在着色器程序中定义, 如 304 处所指示的。在一个实施例中, 着色器实现是例程或子例程。通过指向存储关于着色器实现的数据的表来指定实际实例, 如

306 处所指示的。且用实际类实例（例程或子例程）来替换着色器程序中的方法调用，如 308 处所指示的。为了说明上述步骤，下文呈现并详细讨论了可与 Direct3D 集成的代码。

[0047] Direct3D 包含多个分立着色器阶段，每一阶段为了再现流水线中的单独目的。这六个阶段创建了其中开发人员编写代码来控制每一着色器或阶段的操作的再现流水线。为了以这些阶段为目标，开发人员使用 HLSL 并将 HLSL 代码转换成经优化的着色器字节代码的相关联 HLSL 编译器。如上所述，字节代码是用于图形硬件中的图形设备驱动程序的经编译的 HLSL 代码的低级表示。

[0048] 在一个实施例中，HLSL 包含允许将函数和如变量和纹理等独立资源编组成类的面向对象构造。在该范例中，可以声明定义了可从其实例化多个类的模板的接口。在用这种方式定义时，从接口继承的类定义将使用动态联接来链接的实现。为了定义着色器程序中可被插入诸实现中的一个的点，开发人员创建接口变量并且可在不参考可能的类推断的情况下使用该接口的定义的方法。在一个实施例中，选择单个着色器实现并将其内置到着色器中。在替换性实施例中，HLSL 代码中使用接口的特定点定义了所有实现是内联的位置，并且所有实现主体被插入到着色器中。随后，当着色器实际运行时，在编译很久之后，选择特定实现来执行。

[0049] 下列代码示出可以如何选择实际类实例并用着色器代码来替换方法调用。

```
[0050] 1 interface Light
[0051] 2 {
[0052] 3     float3 Calculate(float3Position, float3Normal) ;
[0053] 4 }
[0054] 5
[0055] 6 class AmbientLight:Light
[0056] 7 {
[0057] 8     float3 Calculate(float3 Position, float3 Normal)
[0058] 9     {
[0059] 10         return AmbientValue ;
[0060] 11     }
[0061] 12
[0062] 13     float3 AmbientValue ;
[0063] 14 }
[0064] 15
[0065] 16 class DirectionalLight:Light
[0066] 17 {
[0067] 18     float3 Calculate(float3 Position, float3 Normal)
[0068] 19     {
[0069] 20         float3 LightDir = normalize(Position-LightPosition) ;
[0070] 21         float LightContrib = saturate(dot(Normal, -LightDir)) ;
[0071] 22         return LightColor*LightContrib ;
[0072] 23     }
```

```
[0073] 24
[0074] 25         float3 LightPosition ;
[0075] 26         float3 LightColor ;
[0076] 27 }
[0077] 28
[0078] 29 AmbientLight My Ambient ;
[0079] 30 DirectionalLight MyDirectional ;
[0080] 31
[0081] 32 float4 main(Light MyInstance, float3 CurPos:CurPosition,
[0082] 33         float3 Normal:Normal):SV_Target
[0083] 34 {
[0084] 35     float4 Ret ;
[0085] 36     Ret.xyz = MyInstance.Calculate(CurPos, Normal) ;
[0086] 37     Ret.w = 1.0 ;
[0087] 38
[0088] 39     return Ret ;
[0089] 40 }
```

[0090] 上述示例是用 HLSL 来编写的,且字节代码的实际表示是二进制形式的。上述代码的解释在以下段落中给出。

[0091] 1-4 行定义被称为 Light(光)的接口,其是在该示例中定义的类的父接口。在第 3 行中,定义了 Calculate(计算)方法的原型。Calculate 必须由 Light 的任何子类来实现。6-14 行定义 AmbientLight(环境光),其是 Light 接口的简单实现(即,简单着色器定义)。18-23 行示出带有与 Light::Calculate(光的计算)相同的签名但带有比 DirectionalLight::Calculate(方向光的计算)更复杂的代码(即,复杂着色器定义)的 Calculate 方法的实现。

[0092] 25-26 行示出 DirectionalLight(方向光)的操作所需的描绘局部类变量。29-30 行示出两个变量的类实例定义:MyAmbient(我的环境)(简单着色器定义)和 MyDirectional(我的方向)(复杂着色器定义)。这些变量用作至再现流水线的绑定点并标识可被选择来在以下描述的 MyInstance(我的实例)变量的位置中使用的可能的实现。

[0093] 32-40 行示出程序的主着色器部分。第一个自变量是类型 Light 的通用接口变量。在使用 Light 的点处,插入特殊调用指令(在一个实施例中)。结果,所有实现主体将出现在着色器字节代码中。表随后可将调用链接到其可能执行的主体。其余的参数是标准 Direct3D 着色器中使用的标准再现流水线变量。

[0094] 在操作上,上述代码可用 HLSL 代码来编写并被发送到 HLSL 编译器以便转换成字节代码。进而将字节代码提供给 GPU 上的驱动程序以便被设置为在 HLSL 代码中描述的着色器阶段的程序。在 Direct3D 的先前版本中,此字节代码由着色器所需的输入、输出和依赖资源的低级描述以及定义着色器阶段的操作的汇编样式指令构成。就此处描述的各实施例而言,Direct3D 字节代码还包括定义用于子例程的输入、输出和操作指令的子例程以及定义抽象接口方法的使用点和哪些方法定义可被内联到着色器中的不同点的表。注意到以

下是重要的：以下示例是用被称为反汇编的半可读文本来编写的，并且字节代码的实际表示是二进制形式的。

[0095] 字节代码旨在表示对着色器阶段的状态和预期执行的高度优化的表达式定义。在 Direct3D 10 之前的各个 Direct3D 版本中，该字节代码匹配于可在图形硬件上执行的准确指令。因为 Direct3D 10 中的发散体系结构（例如，CPU 上的流水线仿真），修订字节代码来改为提供中间表示——即 IL。在一个实施例中，向 GPU 的设备驱动程序提供 IL 并将代码转换成在 JIT 进程中的合适的本机指令。在另一实施例中，以 JIT 操作可用优化或重新格式化的最小需求来执行的方式来设计 IL。另外，可在创建着色器定义时——而非在联接发生时——对独立的类实例执行 JIT 操作以便确保在链接时联接可如平凡的内联般执行。上述可以在以下示例性代码中看出。应该注意，以下呈现的代码不旨在限制本发明的各实施例。也可以另选地使用其他代码。

```
[0096] 1 dcl func_table ft0{fb0}
[0097] 2
[0098] 3 dcl func_table ft1{fb1}
[0099] 4
[0100] 5 dcl_func_ptr fp0[1][1] = {ft0, ft1};
[0101] 6
[0102] 7 fb0:in:(const iv0.xyzw,
[0103] 8         const iv1.xyzw,
[0104] 9         const iv2.xyzw),
[0105] 10    out:(oo0.xyzw)
[0106] 11 mov oo0, cb[iv0.x][iv0.y]
[0107] 12 ret
[0108] 13
[0109] 14 fb1:in:(const iv0.xyzw,
[0110] 15         const iv1.xyzw,
[0111] 16         const iv2.xyzw),
[0112] 17    out(oo0.xyzw)
[0113] 18 add r0.xyz, iv1.xyzx, -cb[iv0.x][iv0.y].xyzx
[0114] 19 dp3 r0.w, r0.xyzx, r0.xyzx
[0115] 20 rsq r0.w, r0.w
[0116] 21 mul r0.xyz, r0.xyzx, r0.wwww
[0117] 22 dp3_sat r0.x, iv2.xyzx, -r0.xyzx
[0118] 23 mul oo0.xyz, r0.xxx, cb[iv0.x][iv0.y+1].xyz
[0119] 24 ret
[0120] 25
[0121] 26 main:
[0122] 27 fcall fp0[0][0], in:(cb14[0], v0, v1), out:(o0)
[0123] 28 ret
```

[0124] 第 1 行描绘了用于 AmbientLight 的类实例表并列出了用于特定类实例的所有函数实现。在上述代码中,只有一个函数 Calculate,其在主着色器代码中被调用一次。因此,只存在一个实现 fb0。也可以引用其他方法、函数、例程或对类 AmbientLight 中的现有方法的调用。此外,第 3 行示出参考先前的代码所讨论的变量 DirectionalLight 的类实例表。

[0125] 用于经由 MyInstance 来分派的表接口在第 5 行上。第一个数组维度指示接口变量是否是一个数组。在此情况中,只存在一个元素,所以维度被给予值 1。第二个数组维度是接口的调用点的数量。在此情况中,只有一个方法 Calculate,所以维度是一。最后,括号中的列表是该接口变量可使用的类实例表的列表。因为 ft0(AmbientLight) 和 ft1(DirectionalLight) 都是从 Light 继承的,所以列出了两个表。

[0126] 7-12 行示出为 27 行处的调用点优化的用于 Calculate 的 AmbientLight 的实现的 IL。如果存在使用 AmbientLight 类的 Calculate 函数的附加调用点,则将存在为特定调用点优化的与这个相类似的多个块。注意,被标记为“iv”和“ov”的寄存器被用来代替标准 HLSL 寄存器如 x、s 或 cb。如果多个调用点发出相同的指令集,则可以移除冗余块且各个调用点将指向单个块。这意味着调用点枚举寄存器,从而要求寄存器的替代需要作为内联进程的一部分来发生。另外,14-24 行示出为 27 行处的调用点优化的用于 Calculate 的 DirectionalLight 的实现的 IL。

[0127] 主着色器代码块在行 26-28 中呈现。在 27 中,fcall 指令指示数组元素并为用于变量 MyMaterial(我的材料)的 Calculate 例程定义调用点。第一个参数指示要使用的接口表(fp0)。第一个分类索引定义方法索引。在这种情况下,只有一个方法调用点,所以索引是一。第二个分类索引定义正在执行的调用点的索引。在这种情况下,只有对 Calculate 方法的一个调用,所以该索引是零。“in(入)”和“out(出)”指示该调用使用的寄存器。第一个“in”参数总是指代常量存储器中存储类实例变量的位置——在这种情况下是 cb14,元素零。

[0128] 在一个实施例中,fcall 指令指的是方法将通过提供索引来调用而不是定义要调用的准确实现。在生成 IL 以及随后的用于程序执行的本机硬件指令时,发出代码到 fcall 例程并且部分地高速缓存寄存器的当前状态和其他着色器状态并围绕实现生成来还原。用于第一实现的代码通过从寄存器分配、临时寄存器等的当前状态开始来生成。一旦该生成步骤完成,状态就被还原到高速缓存的状态,且为下一可能的实现重复生成和编译。重复该循环直到所有实现都生成了代码。最后,当前状态被还原到高速缓存的状态并且将 fcall 的输出影响应用于当前状态,而在 fcall 之后继续代码生成。所生成的所得方法是用 IL 来定义的并在类实例表中被引用,这些类实例表与接口定义的结构匹配并具有对所创建的每一编译的函数版本的引用。

[0129] 为了在 C API 中实现以上所述的,对 Direct3D 作出最小改动。在一个实施例中,添加用于引用着色器所提供的类实例的新的 API。另外,创建另一 API 来引用类实例。

[0130] 在操作上,当该着色器对象被绑定到流水线时,应用程序具有提供其希望用于着色器中的可用绑定点的特定类实例的列表的机会。为此,通过向引用类实例的 API 提供用于着色器的 HLSL 类实例的名称来获得列表对象。

[0131] 类引用 API 可只允许按批量机制的交互,这意味着应用程序只能改变绘制调用组之间的再现流水线的状态,而非如像素的再现之间那样更细粒度地改变。然而,某些实施例

可改变图元集之间的状态,从而使得类实例只在批量图元之间是可改变的。这可以在以下代码中看出。

```
[0132] 1 pDevice->CreatePixelShader(pShaderCode, pMyClassLibrary, &pMyPS);
[0133] 2
[0134] 3 pMyDirectionalLight = pMyClassLibrary->GetClassInstance(" MyDirect
ional" );
[0135] 4 pMyAmbientLight = pMyClassLibrary->GetClassInstance( " My
Ambient" );
[0136] 5
[0137] 6 while(true)
[0138] 7 {
[0139] 8     if(DirectionalLighting)
[0140] 9         pDevice->PSSetShader(pMyPS, &pMyDirectionalLight,1);
[0141] 10 else
[0142] 11     pDevice->PSSetShader(pMyPS, &pMyAmbientLight,1);12
[0143] 13     RenderScene();
[0144] 14 }
```

[0145] 第 1 行示出例程 CreatePixelShader(创建像素着色器),该例程被提供在 pShaderCode(指向着色器代码的指针)中包含经编译的着色器字节代码的串参数、指向引用类实例的 API 的指针(pMyClassLibrary(指向我的类库的指针))、以及指向像素着色器的指针的指针(pMyPS(指向我的像素着色器的指针))。例程检查字节代码并用关于着色器中什么接口和类实例可用的信息来填充 pMyClassLibrary。另外,还向 GPU 的设备驱动程序提供字节代码,该设备驱动器执行将代码转换成本机表示的 JIT 转换并内部存储所转换的代码。最后,引用返回到着色器 pMyPS 以供稍后 API 使用。

[0146] 在第 3 行中,指向引用类实例的 API 的指针(pMyDirectionalLight(指向我的方向光的指针))被设置为引用着色器中的 MyDirectional 类实例。在第 4 行中,指向引用类实例的 API 的指针(pMyAmbientLight(指向我的环境光的指针))被设置为引用着色器中的 MyAmbient 类实例。6-14 行描绘了将重复地再现场景直到程序退出的循环。

[0147] 第 8 行示出用于基于全局输入 DirectionalLighting(方向光照)来选择要使用什么调用实例的代码。基于第 8 行中作出的选择,用 pMyPS 中的着色器对象以及可应用于 HLSL 变量 MyInstance 的两个可能的类实例中的一个作出调用。最后的自变量指示第二个自变量中提供的数组的长度,因为在任一个着色器中可能存在一个以上的要解析的接口。最后,在第 13 行中示出对用于再现场景的几何的函数的调用。

[0148] 尽管用对结构特征和方法动作专用的语言描述了本主题,但可以理解,所附权利要求书中定义的主题不必限于上述具体特征或动作。相反,上述具体特征和动作是作为实现权利要求的示例形式公开的。例如,除此处描述的之外的采样速率和采样周期也可以由权利要求书的范围来捕捉。

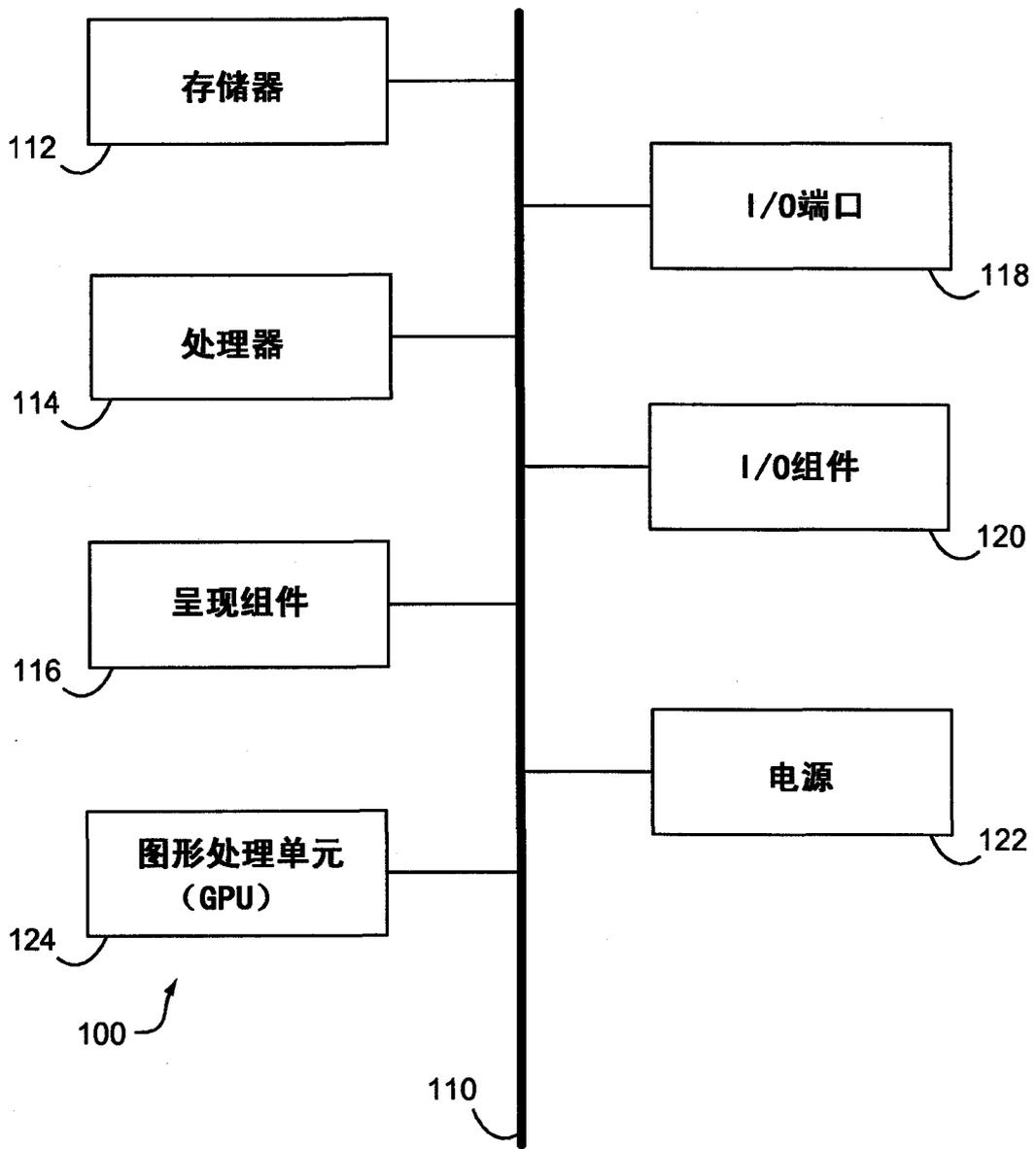


图 1

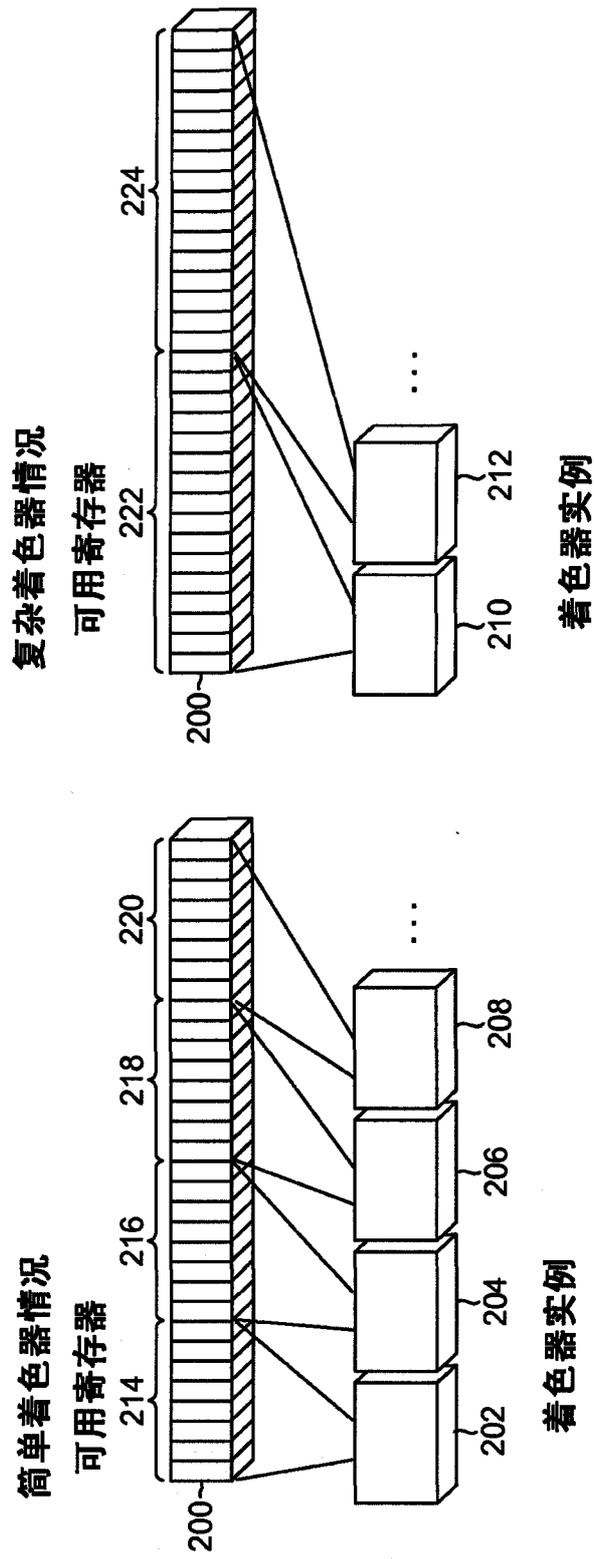


图 2

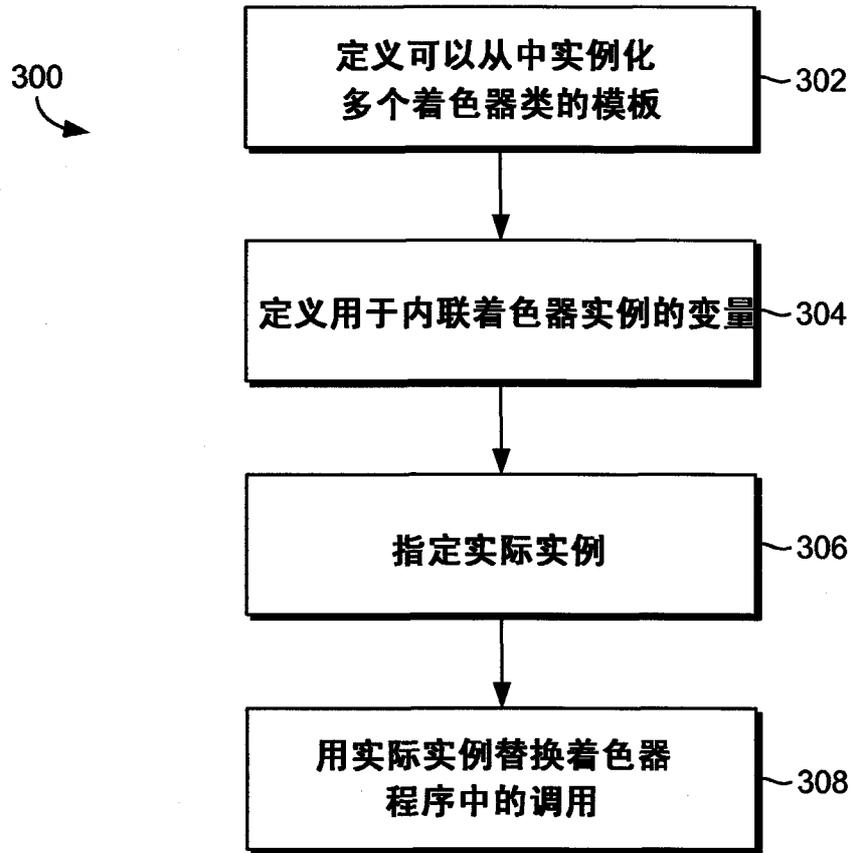


图 3