高技术通讯 2024 年 第34 卷 第7 期:681-691

doi:10.3772/j.issn.1002-0470.2024.07.002

基于无裁剪图形流水线的三维图形处理器①

赵皓宇② 王重熙 宋鹏皓 章隆兵③

(处理器芯片全国重点实验室(中国科学院计算技术研究所) 北京 100190)

(中国科学院大学 北京 100049)

摘 要 传统的三维图形处理器通过裁剪操作获取三角形的可见区域。然而,裁剪操作的延迟长且硬件开销高,大量的裁剪操作会降低图形处理器的性能。本文设计了一款基于 OpenGL ES 2.0 标准的三维图形处理器芯片,采用了统一渲染架构。该图形处理器采用高效的无裁剪图形流水线结构,消除了裁剪所带来的硬件开销和性能损耗。此外,本文为该图形处理器设计了一个符合 IEEE-754 标准的三维向量内积(DP3)计算单元,用于固定功能流水线,以提高图形处理器的性能,并消除图形渲染过程中浮点乘加操作的误差,增强了图形处理器的图形渲染鲁棒性。该三维图形处理器每秒能够处理 500 M 个顶点和 8 G个纹素,功耗为 1 000 mW,采用了 28 nm 工艺,面积为 7.92 mm^2。实现结果表明,与之前的工作相比,本文设计的图形处理器的性能-功耗比提高了 27.8%。

关键词 三维图形处理器;图形流水线;裁剪;向量内积

在桌面级图形架构中,图形处理器有着数十个处理器内核,称为着色器核,可以同时执行数万个线程来进行高质量的图形渲染[1]。然而,在移动设备中,由于硅片面积和功耗的限制,只能有少量的着色器核同时执行少量的线程,所以图形处理器需要在有限的功耗内产生足够的处理能力。因此,人们提出了高性能-功耗比的嵌入式三维图形处理器。

根据 OpenGL ES 2.0 规范,图形处理器包括可编程流水线和固定功能流水线,可编程流水线包括顶点着色器和像素着色器[2]。一些研究在顶点着色器中采用定点算术单元和对数单元来进行坐标变换和光照处理,以减小面积和降低功耗[3]。然而,在大多数三维图形渲染算法中,数值计算要求动态的表示范围和一定的精度,因此目前的图形处理器普遍采用浮点算术单元。纹理映射是像素着色器的主要功能,而纹理映射需要从内存中读取大量的纹理数据,大量访存操作导致纹理映射是图形处理器的主要功耗开销。一些研究通过采用预取和功耗感知的纹理缓存,来提高纹理映射的效率[4-6]。然而,预取和纹理感知的缓存会引入额外的硬件开销,增大芯片的面积。这些研究集中于对可编程流水线的功耗优化,却忽视了固定功能流水线。

根据 OpenGL ES 2.0 规范,传统的固定功能流水线需要对图元进行裁剪[2],图元包括点、线和三角形。裁剪使用 6 个视平面所包围的视锥体来确定图元的可见部分,剪切出图元在视锥体内的部分。然而,传统的裁剪方法复杂且耗时,不仅会产生额外的顶点,还需要大量计算来对图元的顶点坐标和属性进行插值操作,导致固定功能流水线的性能降低和功耗增大。因此,一些研究分别通过硬件和软件 2 个方面来优化裁剪单元以提高裁剪效率。

一些研究设计专用的硬件裁剪单元,通过消除冗余计算和并行处理需要裁剪和不需要裁剪的图元,来提高裁剪性能[7-9]。然而,这些方法会增加额

① 国家重点研发计划(2022YFB3105103)资助项目。

② 男,1997 年生,博士生;研究方向:计算机系统结构,通用图形处理器设计;E-mail:zhaohaoyu19s@ict.ac.cn。

③ 通信作者,E-mail:lbzhang@ict.ac.cn。

(收稿日期:2023-07-04)

外的硬件开销,从而增加图形处理器的功耗。此外,一些研究通过软件的方式,使用可编程流水线的着色器核来执行裁剪操作。这些方法利用了裁剪算法在顶点属性和裁剪面上的独立性,设计了并行裁剪算法,在着色器核上执行[10-11]。然而,基于着色器核的裁剪算法需要根据直线或者三角形以及不同的填充模式,来执行条件判断和分支指令。因为着色器核通常基于单指令多数据的模式来锁步地执行指令,所以分支分歧会导致着色器核的执行效率降低,从而降低顶点的吞吐率。而顶点的吞吐率是决定三维图形处理器性能的关键因素,因此在着色器核上执行裁剪操作,虽然能减小硬件开销,但是会降低图形处理器的性能。

为了消除裁剪的硬件开销和性能损失,本文实现了一个基于无裁剪图形流水线的三维图形处理器,并且流片。无裁剪的图形流水线的原理是将视平面所包围的视锥体归一化为一个立方体[12],因此可以通过在裁剪坐标系下进行图元建立和视口映射,然后在光栅化阶段根据屏幕空间剔除不可见的像素。本文的三维图形处理器在整个图形渲染的过程中不需要专用硬件进行裁剪,因此不会产生额外的顶点和图元。此外,为了避免光栅化误差,图元建立和视口映射需要一定的精度以避免误差。本文研究发现,图元建立、视口映射和光栅化会频繁地执行三维向量内积(3D dot product,DP3)操作。如果使用传统的浮点乘加计算单元来执行三维向量内积,不仅需要 9 个周期的延迟,而且单精度浮点乘加的舍入操作可能导致光栅化误差。因此,本文设计了一个符合 IEEE 754 标准的三维向量内积的浮点计算单元。该单元只需要 5 个周期的延迟来计算三维向量内积,比使用传统浮点乘加单元的性能更高。此外,该三维向量内积单元不对内部计算的中间结果进行任何舍入操作,因此结果更加精确,增强了图形渲染的鲁棒性。

本文的其余部分组织如下:第 1 节介绍了研究背景;第 2 节描述了无裁剪的图形渲染算法;第 3 节展示了三维图形处理器的架构;第 4 节展示了验证和实现结果;第 5 节对全文进行总结。

1　研究背景

图 1 展示了传统的图形渲染流水线。图形渲染流水线可以分成 2 个部分,即可编程流水线和固定功能流水线。

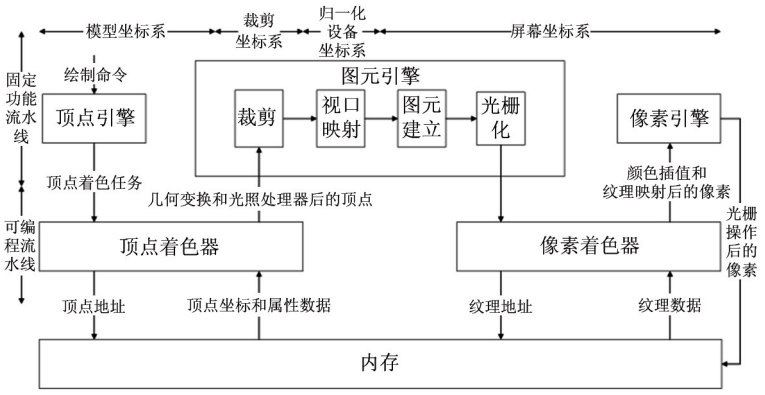


图 1　传统的图形渲染流水线

OpenGL ES 2.0 规定的可编程流水线包含顶点着色器和像素着色器。顶点着色器计算物体在三维空间中的位置,对每个顶点进行几何变换和光照处理。像素着色器通过颜色插值和纹理映射来改变每个像素的颜色。顶点着色器和像素着色器在一个统一的硬件着色器核上执行,2 个着色器共享相同的数据通路。

顶点着色器将处理后的裁剪坐标系下的顶点发送到固定功能流水线的图元引擎。顶点首先被组装为点、线和三角形图元,然后经过裁剪和透视校正,将坐标转换到归一化设备坐标系,最后通过视口映射将坐标转换到屏幕坐标系。屏幕坐标系下的图元

首先在图元建立阶段计算图元的边方程和深度方程;然后在光栅化阶段生成覆盖图元的像素并计算像素的重心坐标;随后像素被像素着色器渲染,输出到像素引擎执行深度测试和颜色混合等光栅操作;最终被写入到片外内存中。

现有的图形流水线必须通过裁剪操作才能将图元的坐标转换到屏幕坐标系。然而,裁剪操作成为了图形流水线的瓶颈,尽管使用专用的硬件可以加速裁剪,但也会引入高昂的硬件开销[7-9]。即使使用着色器核进行软裁剪,也会降低顶点的吞吐率,从而降低图形处理器的性能[10-11]。

2 无裁剪的图形渲染

无裁剪的图形渲染主要包含 3 个阶段,即图元建立、视口映射和光栅化。与图 1 所示的传统图形渲染流水线不同,无裁剪图形渲染先进行图形建立,再执行视口映射。无裁剪图形渲染首先在裁剪坐标系下执行图元建立,计算图元的边方程和深度方程的系数;然后执行视口映射,将边方程和深度方程的系数从裁剪坐标系转换到屏幕坐标系;最后执行光栅化,生成图元覆盖的屏幕像素。无裁剪的图形流水线将视平面所包围的视锥体归一化为一个立方体,该立方体决定了可见的屏幕区域。因此,光栅化可以根据屏幕区域剔除视锥体之外的图元的不可见部分像素。

2.1 图元建立

图元建立过程需要计算图元的边方程和深度方程。对于三角形图元,存在 3 个顶点,经过顶点着色的顶点坐标位于裁剪坐标系下。对于其中的 2 个顶点 $\boldsymbol{V}_0=[x_0, y_0, w_0]$ 和 $\boldsymbol{V}_1=[x_1, y_1, w_1]$,表示裁剪坐标系下的顶点的齐次坐标,设连接这 2 个顶点的边方程为 $ax+by+c=0$,用向量表示为 $\boldsymbol{E}=[a, b, c]$,其中 a、b、c 为边方程的系数,则有:

$$\boldsymbol{E}=\boldsymbol{V}_0\times\boldsymbol{V}_1 \tag{1}$$

对于裁剪坐标系下三角形的 3 个顶点 $\boldsymbol{V}_0=[x_0, y_0, z_0, w_0]$、$\boldsymbol{V}_1=[x_1, y_1, z_1, w_1]$ 和 $\boldsymbol{V}_2=[x_2, y_2, z_2, w_2]$,表示裁剪坐标系下顶点的齐次坐标,设由这 3 点所决定的面方程为 $ax+by+cz+d=0$,用向量表示为 $\boldsymbol{P}=[a, b, c, d]$,其中 a、b、c、d 为面方程的系数,则有:

$$\boldsymbol{P}=\boldsymbol{V}_0\times\boldsymbol{V}_1\times\boldsymbol{V}_2 \tag{2}$$

通过顶点着色器得到顶点在裁剪坐标系下的齐次坐标,那么根据式(1)可以计算三角形的 3 条边的方程,假设三角形的 3 条边的方程通过向量表示,分别为 $\boldsymbol{E}_i=[a_i, b_i, c_i]^{\mathrm{T}}$,$(i=0, 1, 2)$。那么设 $\boldsymbol{C}=[c_0, c_1, c_2]$,$\boldsymbol{W}=[w_0, w_1, w_2]$ 和 $\boldsymbol{Z}=[z_0, z_1, z_2]^{\mathrm{T}}$,其中 c_i 为边方程系数,w_i 和 z_i 为各顶点的投影系数和深度。那么深度方程 $\boldsymbol{D}$ 可以从式(1)和(2)推导出来,可表示为

$$\boldsymbol{D}=\frac{1}{\boldsymbol{C}\cdot\boldsymbol{W}}(\boldsymbol{E}_0, \boldsymbol{E}_1, \boldsymbol{E}_2)\times\boldsymbol{Z} \tag{3}$$

2.2 视口映射

图 1 所示的传统图形处理器的视口映射是将顶点坐标从归一化设备坐标系转换到屏幕坐标系。然而,在无裁剪的图形渲染中,没有从裁剪坐标系到归一化设备坐标系的转换,因此无法直接对顶点坐标进行视口映射。因此,本研究的图形流水线是将边方程和深度方程从裁剪坐标系转换到屏幕坐标系。相比对顶点坐标进行坐标变换,对方程系数进行坐标变换可以在视口映射之前进行预裁剪和背面剔除操作,提前消除不可见的图元,从而减少视口映射的计算量,降低图形处理器的功耗。

视口是屏幕上的矩形的像素可见区域,设视口左下角的屏幕坐标为 (x, y),视口的像素宽度和高度为 w 和 h。如果裁剪坐标系下的边方程为 $\boldsymbol{E}_c=[a_c, b_c, c_c]^{\mathrm{T}}$,屏幕坐标系下的边方程为 $\boldsymbol{E}_s=[a_s, b_s, c_s]^{\mathrm{T}}$,那么边方程的视口映射可以表示为

$$\boldsymbol{E}_s=\begin{bmatrix}2(2x+w) & 0 & 0\\ 0 & 2(2y+h) & 0\\ -2(2x+w)h & -2(2y+h)w & wh\end{bmatrix}\cdot\boldsymbol{E}_c \tag{4}$$

深度方程的视口映射与边方程类似。

2.3 光栅化

图 1 所示的传统的图形流水线首先进行裁剪,将图元在视锥体之外的部分剪切掉,然后再进行光栅化。然而,本文提出的图形流水线通过将视锥体归一化为一个立方体[12],使得光栅化能够在像素层

面上剔除位于屏幕区域外的不可见像素,即通过光栅化间接进行裁剪。

光栅化包括 3 个步骤:包围框计算、块级光栅化和像素级光栅化。包围框是完全覆盖图元的最小矩形屏幕区域,因此不需要对包围框外的区域进行光栅化。块级光栅化会遍历包围框内的 4 ×4 像素块,如果图元覆盖该像素块,则对该像素块进行像素级光栅化。像素级光栅化首先判断像素块内每个像素的可见性,然后计算每个像素的深度和重心坐标。将像素坐标 (x, y) 代入边方程 $ax + by + c$,可以得到边方程在该像素点上的值。由式(1)可以计算 3 个边方程的系数,设得到的 3 个边方程在某个像素点上的值分别为 e_0、e_1 和 e_2,那么该像素点的重心坐标 $\boldsymbol{B}$ 可以表示为

$$\boldsymbol{B} = \left(\frac{1}{e_0 + e_1 + e_2}\right)(e_0, e_1, e_2) \tag{5}$$

3　三维图形处理器系统架构

在图 1 所示的传统图形流水线中,顶点着色器和像素着色器分别通过独立的硬件单元执行。而本文的图形处理采用了统一着色器核,在单个核中同时执行顶点着色器和像素着色器。图 2 显示了本文设计的三维图形处理器的整体架构,只包含一个统一着色器核,统一着色器核避免了顶点和像素着色的负载不均衡问题,提高了图形处理器的性能。该主机处理器包括着色器核、任务调度器、命令处理器、顶点引擎、图元引擎和像素引擎。图形流水线可划分为固定功能流水线和可编程流水线。固定功能流水线由命令处理器、顶点引擎、图元引擎和像素引擎组成,而可编程流水线由着色器核和任务调度器组成。

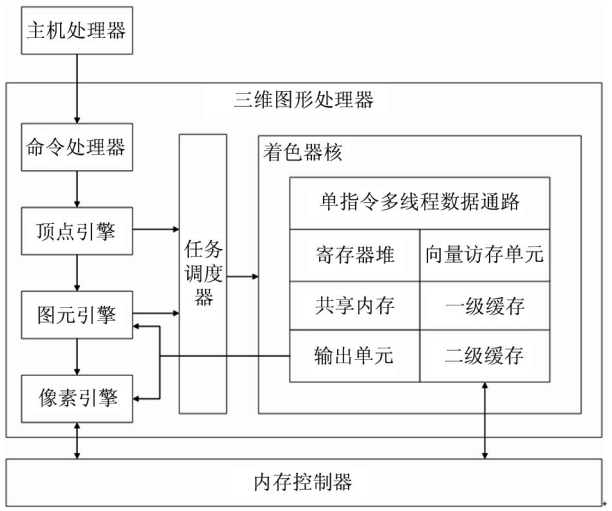


图 2　基于无裁剪图形流水线的三维图形处理器架构

3.1　统一渲染架构

该三维图形处理器采用统一渲染架构,顶点着色器和像素着色器在同一个着色器核中同时执行,因此它们共享相同的数据通路。用户程序输入的顶点被打包成每 32 个顶点为一组的顶点着色任务,这些任务被发送到任务调度器进行调度。任务调度器根据着色器核的内部执行状态来调度这些顶点着色任务。经过顶点着色的顶点被发送到图元引擎,图元引擎将这些顶点组装成三角形、点或线段图元。图元经过光栅化后生成覆盖的像素,并计算像素的重心坐标。像素以 2 ×2 的像素块为基本单元,每 8 个 2 ×2 的像素块被打包成一个像素着色任务,这些任务也被发送到任务调度器,交由着色器核执行。像素在像素着色器中进行渲染,进行纹理映射或颜色插值。由于纹理映射需要访问外部存储器,因此使用数据缓存来提高性能。顶点和像素着色器在同一个着色器核的相同运算单元中交替处理。每个顶点或像素的处理对应着色器核中的一个线程,每个着色任务中的 32 个线程锁步执行。

在统一渲染架构中,存在 3 个层次的并行性:(1)顶点和像素数据级的并行性;(2)着色器指令级的并行性以及顶点;(3)像素任务级的并行性。

顶点和像素数据级的并行性指的是顶点之间或像素之间的计算并行。在这种并行性下,输出顶点或像素不需要反馈给输入顶点或像素,它们之间不存在依赖关系。因此,顶点或像素可以同时进行处理。为了利用这种并行性,通常将每 32 个顶点或像素打包为一个着色任务,这些任务以单指令多线程的方式在着色器核上执行。一个着色器核最多可以同时执行 40 个任务,从而实现顶点和像素数据级的并行计算。

着色器指令级的并行性是指在顶点或像素着色器内部的指令级别的并行计算。在顶点和像素着色器程序中,存在一些无依赖关系的指令组,例如几何

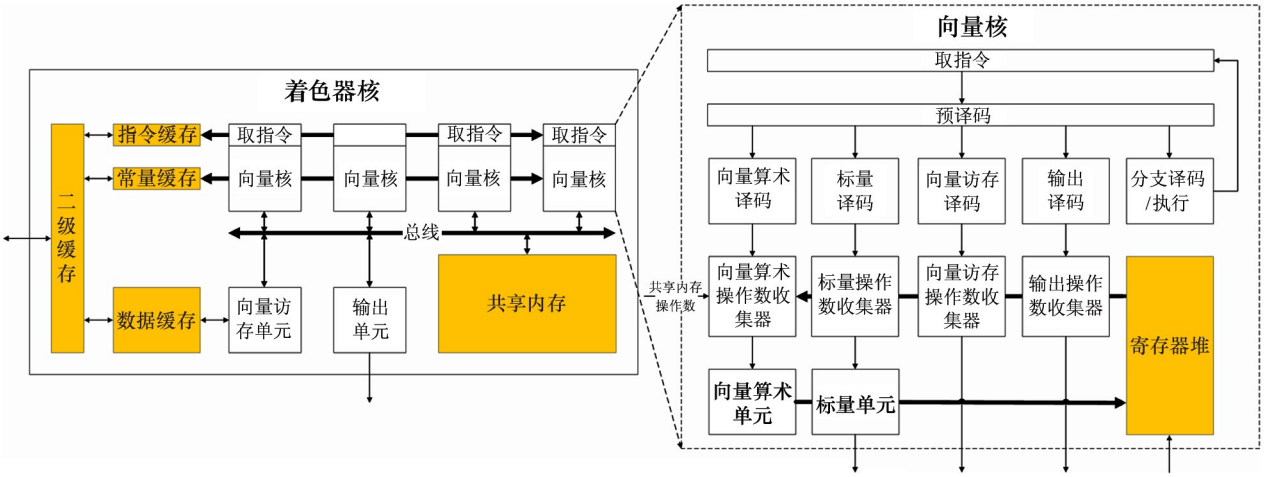
变换和光照处理等操作。这些指令可以在着色器程序中同时执行,从而提高着色器的运行效率。另外,由于一个顶点或像素是由许多属性组成的,如位置、颜色和纹理坐标等,着色器程序可以并行地处理这些属性。

顶点和像素任务级的并行性体现在顶点和像素程序的独立性。顶点和像素着色器之间相互独立,可以使用着色器核来并行处理它们。这种并行性允许顶点和像素任务同时进行,从而进一步提高并行计算的效率。

综合利用这 3 种并行性,统一渲染架构可以实现高效的图形渲染,充分发挥图形处理器的计算能力。

3.2 统一渲染架构的着色器核实现

图 3 展示的着色器核架构是一个单指令多线程处理器,用于可编程流水线的三维图形渲染。它支持 OpenGL ES 2.0 和 GLSL 1.0 标准。着色器核包括的主要组件有 4 个向量核、1 个向量访存单元、1 个输出单元和 1 个共享内存。每个向量核具有独立的寄存器堆、向量算术单元、标量单元和分支单元。这意味着每个向量核可以独立执行指令和计算,它们能够并行处理多个线程的工作负载,从而提高计算效率。共享内存、向量访存单元、输出单元、指令缓存、常量缓存、数据缓存和二级缓存是 4 个向量核所共享的资源。



(有色区域为着色器核的存储单元)

图 3 基于无裁剪图形流水线的图形处理器着色器核内部架构

为了实现顶点和像素着色器的并行执行,本文的图形处理器采用了一种并行架构。在这个架构中,向量核为每个着色任务分配了独立的寄存器堆和共享内存。每个向量核最多可以同时执行 10 个着色任务,因此一个着色器核可以并行地执行 40 个着色任务。

在取指令阶段,着色器核采用轮询调度的方式选择一个着色任务,并从指令缓存中取出该任务的指令。预译码器根据指令的类型将其发送到相应的译码器上。译码器进一步对指令进行译码,并通过记分板的方式解决指令之间的相关性。无相关的指令被发射到相应的执行单元的操作数收集器上。在译码阶段,如果某个着色任务因为指令相关性而无法发射,调度器可以发射其他着色任务的指令,以掩盖相关性引起的延迟。由于每个向量核最多可以调度 10 个着色任务的指令,并且这些任务的指令之间没有相关性,因此大多数的延迟可以被掩盖。操作数收集器从寄存器堆和共享内存中读取操作数,并将其发射到相应的执行单元上。每个向量核包含一个 8 路的向量算术单元,而每个着色任务负责处理 32 个顶点或像素。因此,一条向量算术指令至少需要 4 个周期才能完成。本文的图形处理器的频率为 500 MHz,因此该处理器的算术性能达到了 16 GFLOPs。这样的性能可以为三维图形渲染提供强大的计算能力。

向量访存单元除了执行常规的访存指令,还负

责执行纹理采样指令和进行纹理过滤。当着色器核执行顶点着色器时,它对矩阵和顶点坐标进行几何运算,并对顶点属性进行光照处理。矩阵系数通过标量访存指令从常量缓存中获取,而顶点坐标和属性则通过向量访存指令从数据缓存中获取。当着色器核执行像素着色器时,它会进行颜色插值和纹理映射等操作。对于双线性纹理过滤,纹理单元每个周期可以处理 16 个纹素(texel),并生成 4 个像素。为了保证纹理过滤的吞吐率,数据缓存由 4 个双端口静态随机存取存储器(static random-access memory,SRAM)组成,每个 SRAM 每个周期可以读出 32 字节的数据。在 500 MHz 的时钟频率下,着色器核可以每秒最多处理 8 G 个纹素。着色器核完成着色任务后,输出单元将处理后的顶点和像素分别输出到图元引擎和像素引擎,进一步进行后续处理。这样的架构设计能够高效地支持三维图形渲染,并实现高吞吐率的纹理过滤和像素处理。

在着色器核中,向量算术单元和向量访存单元是核心资源,对于三维图形渲染的性能起着重要作用。这 2 个单元具有快速的算术处理和纹理获取能力。通过优化向量算术单元和向量访存单元的设计和性能,着色器核能够提供高效的算术处理和纹理采样能力,从而支持实时的三维图形渲染需求。

3.3　固定功能流水线

3.3.1　命令处理器

命令处理器是主机和三维图形处理器之间的接口,用于实现它们之间的交互。用户程序通过命令处理器发起绘制任务,并将相关的建模数据、流水线配置和着色器代码存储在内存中,这些数据一起构成了一次完整的绘制任务。当用户程序发送绘制命令后,驱动程序首先将相应的数据准备好并存储在内存中;然后主机将绘制命令发送到图形处理器的命令队列中,并唤醒命令处理器。命令处理器接收到命令后,开始控制图形处理器来完成该绘制命令。

3.3.2　顶点引擎

顶点引擎负责生成和调度顶点着色器任务。它将每 32 个顶点的内存索引打包成一个顶点着色任务,并将其发送到着色器核进行执行。顶点着色程序根据索引从内存中提取顶点数据并进行处理,最终将结果发送到图元引擎。

3.3.3　图元引擎

图元引擎的结构框图如图 4 所示。它集成了几个模块,用于加速三维图形的渲染。相比传统的图形渲染流水线,该三维图形处理器没有裁剪单元,并且在图元建立之后执行视口映射。图元引擎的关键任务是在裁剪设备坐标中计算图元的边方程和深度方程,从而进行图元建立。在图元建立和视口映射阶段,根据式(1)、(3)和(4)计算图元的边方程和深度方程。由于没有裁剪单元,相较于传统的图形流水线,该图形处理器的硬件开销更小。视口映射在图元剔除之后执行,这样可以减少计算量,并降低图形处理器的功耗。

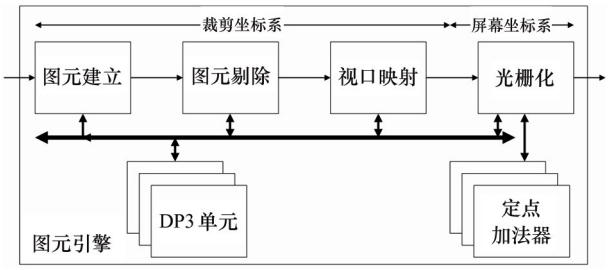


图 4　基于无裁剪图形流水线的图元引擎架构

光栅化器通过递归的方式进行层次化遍历包围框。在计算图元的包围框后,该图元被压入栈中。每个周期,光栅化器首先从栈中弹出一个图元,并生成该图元包围框的 4 个子区域。其次,它判断这 4 个子区域是否与该图元有重叠,并只将与图元有重叠的子区域压入栈中。该过程重复进行,直到生成的子区域的像素宽度小于等于 4。最后,光栅化器将覆盖这些子区域的 4 ×4 像素块压入队列中。在像素级遍历阶段,像素级光栅化器从队列中取出 4 ×4像素块。首先它判断每个像素的可见性,然后根据式(5)计算每个像素的重心坐标,最后图元引擎将每 8 个 2 ×2 的像素块打包成一个像素着色任务,并发送给着色器核执行像素着色器。经过多像素着色后,结果被发送到像素引擎处理。

3.3.4　像素引擎

像素引擎负责执行深度测试和颜色混合等光栅操作。对于需要执行深度测试和颜色混合的像素块,像素引擎首先从内存中读取相应位置的帧缓冲

区数据,然后进行相应的计算,最后将计算结果写回到帧缓冲区中。

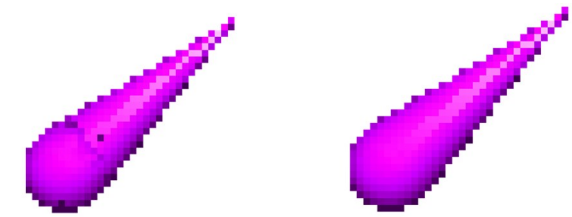
由于光栅操作通常属于典型的读-改-写型原子操作,为了提高像素引擎的性能,使用了一个 4 kB 的缓存来临时存储从内存中读取的数据。通过使用缓存,可以减少对内存的频繁读写操作,提高数据的访问效率。

3.4 三维向量内积单元

本小节介绍无裁剪图形流水线图形处理器的内部优化方法。如图 4 所示,图元引擎通过硬件集成了三维向量内积(DP3)单元,来加速第 2 节所述的无裁剪图形渲染算法。式(1)和式(3)~(5)包含向量内积操作,其计算在由图元引擎的 DP3 单元执行。在图形应用中,每一帧经常需要处理数万个图元,而每个图元都会在图元引擎中执行式(1)及式(3)~(5)所示的计算。因此图元引擎执行向量内积的性能,将影响图形处理器的图元吞吐率。而且,像素着色器的执行依赖于式(1)、(3)~(5)的计算结果,如果在计算过程中浮点计算的误差逐渐累积,最终会导致绘制图像出现错误,所以需要保证向量内积计算的准确性。因而,向量内积是图形处理器中频繁且重要的操作。

式(1)可以转化为二维向量内积操作,而式(3)涉及 4 个三维向量内积操作,式(4)则表示视口映射需要 3 个三维向量内积操作,式(5)表明每个像素需要进行一次三维向量求和操作。这些操作都可以被转换为三维向量内积操作。这些计算的准确性对光栅化的鲁棒性至关重要,而延迟则决定了图形处理器的性能。因此,快速且准确地计算向量内积是决定图形处理器性能和鲁棒性的关键因素。为此,设计了一个专门用于浮点三维向量内积的单元,以取代传统的浮点加法器和乘法器。实验发现该单元能够提升性能并增强图形渲染的鲁棒性。

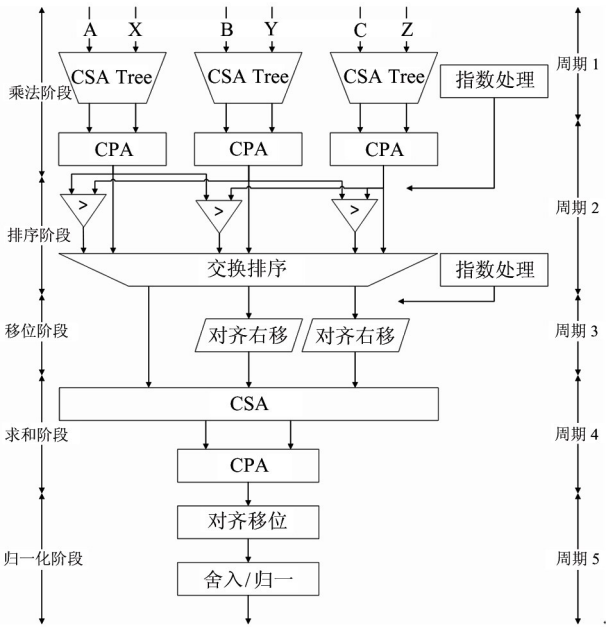
图 5 展示了使用单精度浮点乘加器和使用 DP3 单元的绘制结果。在图 5(a)中,观察到的错误是由于计算深度方程时引入的误差所致。图中绘制了一个圆锥体和一个球体,它们的表面相切。理论上,切线位置的深度应该相等。然而,由于计算深度方程时的误差,原本在锥体后面的球面被绘制出来,导致了错误的结果。这个误差的主要来源是浮点数计算中的舍入造成的精度损失。当 2 个数值接近的单精度浮点乘法结果相减时,浮点计算中的舍入误差尤其明显。在浮点乘法的最后舍入操作中,低位有效数字被丢失。因此,当 2 个相近的浮点乘法的结果相减时,高位的有效数字会减至 0。然而,由于低位有效数字在舍入过程中丢失,导致移位后的结果不准确。



(a) 基于单精度浮点乘加器　　(b) 基于 DP3 单元

图 5　基于不同浮点运算单元的绘制结果

图 6 展示了本文设计的 DP3 单元,它支持 IEEE 754 单精度格式。DP3 单元采用了 3 路浮点单指令多数据结构。该单元不仅可以生成 DP3 的结果,还可以生成 DP2 或基本的乘法和加法运算的结果。为了消除浮点运算舍入导致的误差,DP3 单元内部不对中间结果进行任何舍入操作。此外,DP3 单元在两两相乘的结果上进行排序,根据绝对值大小进



(CSA:进位保持加法器;CPA:进位传播加法器)

图 6　浮点三维向量内积单元

行排序。如果绝对值较大的 2 个乘积之和为 0,则最小的乘积不会进行移位对齐操作,以避免精度损失。整个计算过程包括以下几个周期:(1)周期 1,计算浮点尾数乘积;(2)周期2,对乘积结果进行排序;(3)周期 3,通过移位操作对乘积结果进行对齐;(4)周期 4,对乘积结果进行求和;(5)周期 5,对结果进行归一化。通过这样的设计,能够提高计算的精度并减小舍入误差的影响。

传统的乘加单元通常具有 3 个周期的延迟,这意味着完成一次 DP3 操作需要 9 个周期。然而,通过使用 DP3 单元,可以将执行时间缩短至 5 个周期。在图元引擎中,不同的模块配置了不同数量的 DP3 单元,这样可以确保图形流水线能够以每 4 个周期一个三角形的吞吐率进行工作。利用 DP3 单元的高效性能,能够加快图形处理器的处理速度,使其能够更快地完成复杂的图形计算任务。这种优化设计的目标是保持良好的吞吐率,并确保图形流水线能够高效地处理每个图元。

4 验证和芯片实现

本文验证的测试集包括 glmark2[13]、Piglit[14]、mesa-demos[15] 和 XScreenSaver[16]。这些测试集共计包含 2 011 个测试样例,用于验证本文所设计的图形处理器的性能和正确性。glmark2 是一个开源的基准测试程序,用于测试 OpenGL|ES 2.0 的性能,它涵盖了图形处理器性能的多个方面,包括缓冲、建模、光照和纹理等。Piglit 是一个用于测试 OpenGL 实现的自动化测试集,它包括 Glean 测试、从 Mesa 改编的测试以及针对特定错误的回归测试。Piglit 的目的是通过广泛的测试覆盖来验证 OpenGL 实现的正确性。mesa-demos 是一个包含了 OpenGL 和 Mesa 的演示和测试程序集,它提供了各种用例和场景,用于测试和展示 OpenGL 和 Mesa 的功能和性能。本实验还使用了 XScreenSaver 作为测试集,以验证图形处理器在真实应用下的正确性。XScreenSaver 是一个在 Linux 和 Unix 系统上运行的 X11 窗口系统的标准屏幕保护应用程序。通过在这些测试集上执行大量的测试样例,就能够评估本文所设计的图形处理器在不同方面的性能和功能,并验证其在真实场景下的正确性。

4.1 验证

图 7 展示了本文在实际图形应用程序中使用的框架和流程,用于收集应用程序编程接口(application programming interface, API)调用并进行验证。GLInterceptor 记录应用程序发出的所有 OpenGL 命令及其参数值、着色器代码、纹理和顶点缓冲区数据,并将这些数据存储在一个输出文件中。同时,所有的 OpenGL 命令和数据也传递给 OpenGL 驱动程序以继续应用程序的执行。为了验证记录的 API 调用的完整性和真实性,构建了一个模拟器来模拟实现 GPU 的功能。该模拟器是基于功能的验证,不提供任何结构上的参考,并未实现包括流水线、缓存和内存系统在内的结构设计。本文将 GLInterceptor 抓取的数据转换成三维图形处理器可理解的内存镜像格式。使用模拟器读取该镜像并执行,以验证抓取的 API 调用的正确性;内存镜像被初始化并输入到寄存器传输级仿真中。在每个绘制任务结束后,当前的帧缓冲以图像的形式输出。最终将生成的图像与模拟器生成的参考图像进行比较,以验证仿真的正确性。通过这一流程,就能够收集并验证应用程序中的 API 调用,确保它们的完整性和正确性。模拟器和寄存器传输级仿真帮助验证设计的功能,并生成参考图像进行比较。这样可以确保本文的图形处理器在实际应用中能够正确地执行 OpenGL 命令,并生成预期的图像输出。

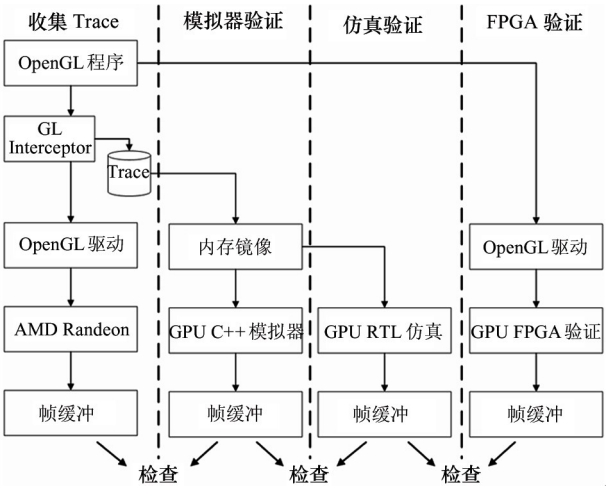
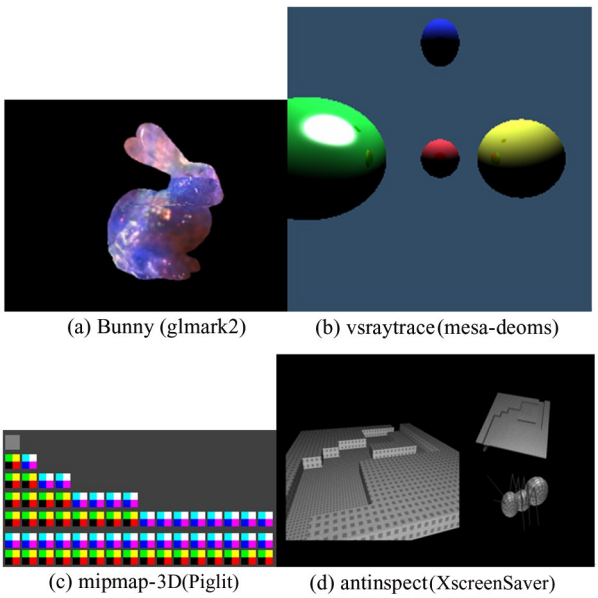


图 7 三维图形处理器仿真和验证流程

除了模拟器和仿真验证,还在 Xilinx XCVU440 现场可编程门阵列(field programmable gate array, FPGA)上成功验证了本文设计的三维图形处理器。由于 FPGA 的大小限制,对图形处理器架构进行了裁剪,最终实现的图形处理器只包含 2 个向量单元。尽管裁剪后的处理器能够执行与完整图形处理器相同的操作,但由于计算资源较少,其吞吐量较低。在 FPGA 上进行图形处理器的原型设计提供了一种有效的验证方法,可以验证设计在物理硬件实现方面的可行性和正确性。

经过测试,验证了本文设计的图形处理器在 glmark2、Piglit、mesa-demos 和 XScreenSaver 4 个测试集上的正确性,部分测试结果如图 8 所示。



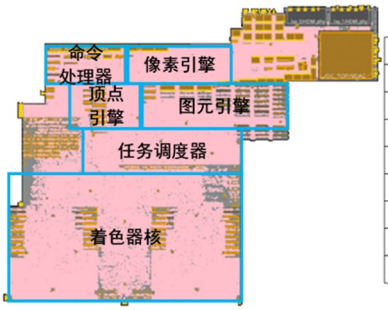
(a) Bunny (glmark2) (b) vsraytrace(mesa-deoms)

(c) mipmap-3D(Piglit) (d) antinspect(XscreenSaver)

图 8 图形处理器在部分测试样例的渲染结果

4.2 芯片实现结果

除了验证之外,还将该处理器进行了流片,图 9 展示了该芯片的物理设计布局和实现结果。在物理设计上,着色器核上占了超过 50% 的面积开销,这是因为着色器核为了支持 1 280 个线程并行执行,不仅采用了共 256 kB 的 SRAM 组成的寄存器堆以及 64 kB 的共享内存,而且同时调度 40 个着色任务来提高着色器核的利用率,因此着色器核的硬件开销远高于其他模块。在剩余模块中,图元引擎的面积和硬件开销最大,图元引擎是图元渲染流水线中固定功能管线的主要算法实现模块,需要足够的 DP3 单元与定点加法器来保证顶点的吞吐率,因此图元引擎有着较大的面积开销。



工作电压	1.0 V
制程	28 nm
面积	7.92 mm²
晶体管数	3.5 M
工作频率	500 MHz
纹素填充率	8 Gtexels/s
FP32 性能	16 GFLOPs
API	OpenGL ES 2.0
功耗	1 000 mW

图 9 图形处理器芯片的物理设计布局和实现结果

为了测试该图形处理器芯片的性能,本文测试了该芯片在 glmark2 上的性能,表 1 展示了测试性能所用的参数。glmark2 是不同方面进行性能测试的基准性能测试程序集,其中 effect2d 用来测试图形处理器在二维贴图上的性能;texture-cube 顶点数较少,但是有深度测试;texture-cat 和 jellyfish 是有大量顶点的三维建模图形应用,其中 texture-cat 有深度测试。为了能测试纹素填充率,所选择的程序都采用了纹理映射。

表 1 测试应用的参数

测试应用	顶点数	分辨率	深度测试
effect2d	4	800×600	无
texture-cube	36	800×600	有
jellyfish	13 200	800×600	无
texture-cat	43 044	800×600	有

图 10 展示了本文所设计的图形处理器在测试应用上的纹素填充率和顶点吞吐率。可以看到,随

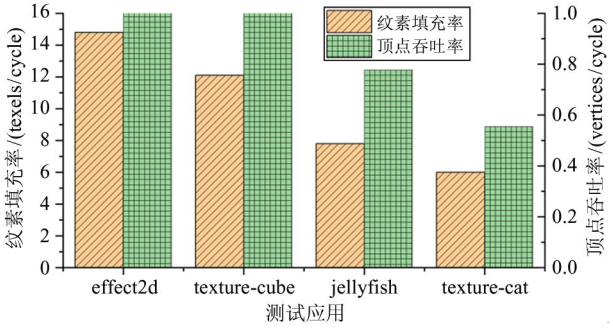


图 10 图形处理器芯片在不同测试应用上的纹素填充率和顶点吞吐率

着应用的顶点数增加,纹素填充率和顶点吞吐率在下降。这是因为着色器核同时执行顶点和像素着色器,因此顶点和像素着色器会竞争着色器核资源,包括寄存器堆、缓存、运算单元、访存带宽等。在 effect2d 上,由于顶点数量少,顶点着色器的任务较少,不会和像素着色器竞争着色器核资源,所以纹素填充率接近理论上的峰值性能——每周期 16 个纹素。对于 texture-cube,由于有深度测试,处理器需要从内存中读写深度数据,与纹理映射竞争访存带宽,所以纹素填充率下降。对于 texture-cat 和 jelly-fish,首先,大量顶点的三维图形建模的三角形通常只覆盖几个像素,所以三角形光栅化所生成的 2 ×2 像素块中存在不可见的像素,导致这些像素块所打包生成的像素着色任务中存在无效的线程。由于着色器核锁步执行的特点,这些无效的线程虽然执行但是不会进行纹理映射,所以导致纹素填充率降低。其次,顶点着色器需要从内存中读取大量顶点的数据,与像素着色器竞争缓存和访存带宽,使得纹素填充率进一步降低。

表 2 对比了本文所设计的三维图形处理器与以前工作在峰值性能和功耗方面的表现。本文的图形处理器采用了无裁剪的图形流水线,使用了 28 nm 工艺制造。其面积为 7.92 mm^2,工作电压为 1.0 V,频率为 500 MHz。本文的芯片经过测试,在满负载的情况下功耗为 1 000 mW,峰值性能可以达到每秒 500 M 个顶点和 8 G 个纹素的吞吐率,计算性能达到 16 GFLOPs。与以前的工作相比,本文的图形处理器采用了更大的寄存器、缓存和共享内存,同时在着色器核上支持更多的线程并行执行,这使得本文的图形处理器在功耗上更高并且性能更强。由于采用了无裁剪的图形流水线,相比传统的图形流水线,本文的图形处理器没有裁剪模块,面积更小。而且本文的图形处理器使用了 DP3 单元,在性能上每个 DP3 单元等价于 3 个浮点乘加器,而在面积上 DP3 单元的开销小于 3 个浮点乘加器,更进一步减小了面积开销。在相同性能的情况下,本文的图形处理器面积更小、功耗更低。因此,本文的图形处理器能够更高效地执行图形渲染,提高性能-功耗比。实现结果表明,本文所设计的图形处理器的性能-功耗比比以前的工作提高了 27.8% 。

表 2　基于无裁剪图形流水线的图形处理器与之前工作的比较

	TVLSI'11[6]	JSSC'08[17]	TCE'13[18]	本文芯片
频率/MHz	143	100	200	500
功耗/mW	367.0	195.0	153.3	1 000.0
峰值性能 (Mvertices/s / Mtexel/s)	143/2 300	9.1/400.0	25 Mtriangles/s	500/8 000
性能-功耗比 (Mvertices/s/mW / Mtexel/s/mW)	0.38/6.26	0.04/2.05	——	0.50/8.00
面积/mm^2	4.5 ×4.5	3.3 ×3.0	4.97	7.92
制程	180 nm	130 nm	65 nm	28 nm
主要特点	双核两路 VLIW; 4D 向量数据路径; 功耗感知的纹理单元	对数算术的光照引擎;专门的光照指令	利用屏幕空间区域进行高效率的光栅化	无裁剪的图形流水线; 高速且精确的三维向量内积运算单元

5　结　论

本文提出了一种基于无裁剪图形流水线的三维图形处理器,其采用统一渲染架构,实现了 OpenGL ES 2.0 标准。为了提高能效,采用了无裁剪的图形流水线,并为图形处理器的固定功能流水线设计了一个三维浮点向量内积单元。该单元能够消除传统浮点乘加操作引入的误差,从而提高了图形渲染的鲁棒性。在大规模的测试集上验证了该图形处理器芯片的正确性。实验结果显示,相较于之前的工作,

本文所设计的图形处理器在性能-功耗比方面提高了27.8%。

参考文献

[1] CHOQUETTE J, LEE E, KRASHINSKY R, et al. 3.2 the A100 datacenter GPU and ampere architecture[C]// 2021 IEEE International Solid-State Circuits Conference. San Francisco, USA: IEEE, 2021,64:48-50.

[2] Khronos Group. OpenGL ES 2.0[EB/OL]. (2000-01-01)[2023-06-07]. https://khronos.org.

[3] NAM B G, LEE J, KIM K, et al. A 52.4 mW 3D graphics processor with 141 Mvertices/s vertex shader and 3 power domains of dynamic voltage and frequency scaling [C]//2007 IEEE International Solid-State Circuits Conference. San Francisco, USA: IEEE, 2007:278-603.

[4] 田泽,张骏,许宏杰,等. 图形处理器低功耗设计技术研究[J]. 计算机科学, 2013,40(S1):210-216.

[5] 韩孟桥,蒋林,杨博文,等. 移动图形处理器的纹理Cache设计[J]. 电子技术应用, 2019,45(5):17-22.

[6] YOON J S, YU C H, KIM D, et al. A dual-shader 3D graphics processor with fast 4-D vector inner product units and power-aware texture cache[J]. IEEE Transactions on Very Large Scale Integration Systems, 2010,19(4):525-537.

[7] BAE J, KIM D, KIM L S. An 11M-triangles/sec 3D graphics clipping engine for triangle primitives[C]// 2005 IEEE International Symposium on Circuits and Systems. Kobe, Japan: IEEE, 2005:4570-4573.

[8] CHUNG K, KIM Y J, KIM S H, et al. Clipping-ratio-independent 3D graphics clipping engine by dual-thread algorithm[C]// 2008 IEEE International Symposium on Circuits and Systems. Seattle, USA: IEEE, 2008:3534-3537.

[9] HSIAO S F, CHANG Y N, TIEN T C, et al. Efficient pre-clipping and clipping algorithms for 3D graphics geometry computation[C]// The 2008 IEEE Asia Pacific Conference on Circuits and Systems. Macao, China: IEEE, 2008:522-525.

[10] SCHNEIDER B O, VAN WELZEN J. Efficient polygon clipping for an SIMD graphics pipeline[J]. IEEE Transactions on Visualization and Computer Graphics, 1998, 4(3):272-285.

[11] McGUIRE M. Efficient triangle and quadrilateral clipping within shaders[J]. Journal of Graphics, GPU, and Game Tools, 2011,15(4):216-224.

[12] SKALAV. Barycentric coordinates computation in homogeneous coordinates[J]. Computers and Graphics, 2008, 32(1):120-127.

[13] Github Inc. glmark2[EB/OL]. (2017-11-01)[2023-06-07]. https://github.co m/glmark2/glmark2.

[14] Gitlab Inc. Piglit[EB/OL]. (2007-07-06)[2023-06-07]. https://gitlab.freedesktop.org/mesa/piglit.

[15] Gitlab Inc. mesa-demos[EB/OL]. (2007-01-23)[2023-06-07]. https://gitlab.freedesktop.org/mesa/demost.

[16] ZAWINSK J. XScreenSaver[EB/OL]. (1992-01-01)[2023-06-07]. https://www.jwz.org/xscreensaver/.

[17] WOO J H, SOHN J H, KIM H, et al. A 195 mW/152 mW mobile multimedia SoC with fully programmable 3-D graphics and MPEG4/H.264/JPEG[J]. IEEE Journal of Solid-State Circuits, 2008,43(9):2047-2056.

[18] LAI Y K, CHUNGY C. 3D graphics processor unit with cost-effective rasterization using valid screen space region [J]. IEEE Transactions o Consumer Electronics, 2013, 59(3):705-713.

A 3D graphics processor with clip-free graphics pipeline

ZHAO Haoyu, WANG Chongxi, SONG Penghao, ZHANG Longbing

(State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

(University of Chinese Academy of Sciences, Beijing 100049)

Abstract

Traditional 3D graphics processors rely on clipping operations to determine the visible area of triangles. However, clipping operations introduce latency, high hardware overhead, and can negatively impact performance when dealing with a large number of clipping operations. This paper presents the design of a 3D graphics processor with a unified rendering architecture based on the OpenGL ES 2.0 specification. This proposed graphics processor incorporates an efficient clip-free graphics pipeline architecture, effectively eliminating the hardware overhead and performance loss associated with clipping. Furthermore, an IEEE-754 compliant 3D vector inner product unit is introduced into the fixed function pipeline of the proposed processor, enhancing performance and eliminating errors in floating-point multiplication and addition operations during graphics rendering, improving the robustness of the graphics rendering. The implemented 3D graphics processor achieves a processing capacity of 500 M vertices and 8 G texels per second, consuming 1 000 mW of power. Utilizing a 28 nm process with an area of 7.92 mm^2, the proposed implementation demonstrates a 27.8% improvement in the performance-to-power ratio compared with previous work.

Key words: 3D graphics processor, graphics pipeline, clipping, vector inner product