一种图形光栅的硬件实现算法

郭安泰 郭 立 杨 毅 吴 思
(中国科学技术大学电子科学与技术系,合肥 230027)

摘 要 提出了一种面向嵌入式平台的图形光栅的硬件实现算法。将三角面包围盒内的像素分成多个规则像素块,在像素块基础上进行扫描转换和像素插值以及透视校正。在对算法做了大量优化后,用 FPGA(现场可编程门阵列)对算法进行了实现和验证。与传统的光栅算法相比,提出的算法提高了像素命中率,减小了计算复杂度,降低了硬件成本。验证结果表明,算法渲染的图形质量达到 OpenGL® ES 1.1 渲染效果;在一般场景下的渲染速度达到 30 帧/秒,满足实时渲染要求;在 Xilinx FPGA Vertex2P xc2vp30-7ff89 上的综合资源为 5 545 个 Slice,硬件消耗较小。

关键词 3D 图形 图形光栅 硬件实现 像素块
中图法分类号: TP391.41 **文献标识码**: A **文章编号**: 1006-8961(2009)01-0176-07

A Hardware Algorithm for Graphics Rasterizer

GUO An-tai, GUO Li, YANG Yi, WU Si
(*Department of Electronic Science and Technology, University of Science and Technology of China, Hefei* 230027)

Abstract In this paper, a hardware algorithm for graphics rasterizer on embedded platform is presented. By dividing the pixels in the triangle's bounding box into a number of regular tiles, the algorithm do the scan conversion, pixel interpolation and projective correction on the basis of tile architecture. After a lot of optimization, the algorithm is implemented and tested on FPGA. As compared with the traditional algorithm, the algorithm presented in this paper has increased the pixel hit rate, and reduced the computational complexity, as well as effectively reduced the hardware cost. Testing results show that, the quality of the algorithm's rendered images has reached the level of OpenGL® ES 1.1. In general scene, the rendering speed reached 30 fps, meeting the requirements of real-time rendering. In terms of the synthesized hardware resources, it is small within 5 545 slices on Xilinx FPGA Vertex2P xc2vp30-7ff89.

Keywords 3D graphics, graphics rasterizer, hardware implementation, tile

1 引 言

一般 3D 图形渲染分为 2 个阶段[1-2]:几何阶段和光栅阶段。几何阶段主要负责顶点的几何变换操作,光栅阶段主要负责多边形扫描转换、纹理贴图以及像素测试等,其中光栅处理器负责扫描转换部分。光栅处理器按照其功能可以分为 2 个部分:三角面填充单元和像素插值单元。

三角面填充单元根据三角面顶点的屏幕坐标和属性,扫描三角面,生成所有三角面内部像素的屏幕坐标以及相关像素属性[3]。在早期的图形加速卡中,三角面填充单元一般采用边扫描算法[4],如图 1 所示。首先根据 Bresenham 直线扫描算法[2]扫描三角面的 3 条边,计算出三边与屏幕水平扫描线所有交点的屏幕坐标和像素属性。然后根据水平扫描线上两个端点的屏幕坐标和像素属性,插值计算水平扫描线上两个端点之间所有像素的屏幕坐标和属性,从而完成三角面的填充。该算法对于规则的三角面,填充效率高,但对于特殊三角面,如形状狭长,

收稿日期:2007-07-09;**改回日期**:2007-08-14

第一作者简介:郭安泰(1982~),男。中国科学技术大学电子科学与技术系硕士研究生。主要研究方向为计算机图形学及硬件实现。E-mail:gantai@mail.ustc.edu.cn

或面积极小的三角面,算法需要做单独的判断处理,这导致算法的硬件实现效率低下。基于边界方程的半空间填充算法[5-7]能很好地解决这个问题,如图 2 所示。半空间填充算法利用三角面 3 条边的边界方程来进行三角面扫描转换。边界方程是根据矢量叉乘原理求出的边的解析方程,以边 p_0p_1 的边界方程 e_2 为例,如式(1)~(3)所示。

$$e_2 = \overrightarrow{p_0p_1} \times \overrightarrow{p_0p} \tag{1}$$

$$e_2 = (x_1 - x_0)(y - y_0) - (x - x_0)(y_1 - y_0) \tag{2}$$

$$e_2(x,y) = ax + by + c \tag{3}$$

边界方程有如下性质:

$$\begin{cases} e_2(x,y) < 0, \text{像素}(x,y)\text{位于}\overrightarrow{p_0p_1}\text{的外面} \\ e_2(x,y) \geqslant 0, \text{像素}(x,y)\text{位于}\overrightarrow{p_0p_1}\text{的里面} \end{cases} \tag{4}$$

当三角面的 3 个顶点按固定顺序排列时(本文为逆时针),只有 3 条边的边界方程值都大于零的像素为三角面内部像素。

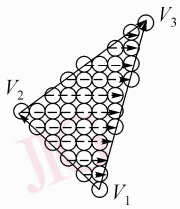


图 1　边扫描算法示意图

Fig. 1　Diagram of edge walking algorithm

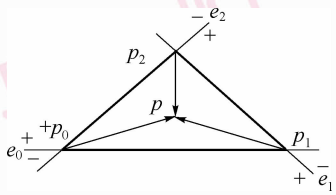


图 2　半空间填充算法示意图

Fig. 2　Diagram of half-space filling algorithm

像素插值单元负责计算三角面内部像素的属性,包括像素坐标、颜色、深度、纹理坐标等。与边扫描算法相适应的像素插值一般采用线性插值[3],如式(5)和式(6)所示。根据 2 个端点的屏幕坐标 x_1,x_0 和属性 s_1,s_0,首先计算出属性的梯度增量 $\partial s/\partial x$,然后进行线性插值。

$$\frac{\partial s}{\partial x} = \frac{s_1 - s_0}{x_1 - x_0} \tag{5}$$

$$s(x) = \frac{\partial s}{\partial x}(x - x_0) \tag{6}$$

多个像素属性插值需要多次除法,不利于硬件实现,而且误差较大;平面插值算法是一种性能更佳的算法,根据 3 个顶点的屏幕坐标 (x_0,y_0),(x_1,y_1),(x_2,y_2) 和像素属性 (s_0,s_1,s_2),预先计算出属性的平面插值系数 (k_1,k_2,k_3),然后根据像素的屏幕坐标进行平面线性插值。

$$\begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \end{bmatrix} \tag{7}$$

$$s(x,y) = k_1x + k_2y + k_3 \tag{8}$$

所有的平面插值系数可以在三角面建立阶段预先生成,从而避免了扫描阶段的实时除法运算,而且相邻像素的插值可以通过简单递增得到。

$$s(x+1,y) = s(x,y) + k_1 \tag{9}$$

$$s(x,y+1) = s(x,y) + k_2 \tag{10}$$

如何将上述算法移植到嵌入式平台,满足其面积小、功耗低的特性成为当前研究热点。最近,PowerVR体系结构[8]的提出使得上述算法的移植成为可能。PowerVR 体系结构是一种基于像素块处理的结构,它将屏幕划分为相同尺寸的多个矩形像素块,每个像素块相互独立,这是一种较低成本的图形绘制架构。本文结合二者的长处,提出了一种优化后的光栅处理器硬件算法。

2　光栅处理器算法

2.1　基于像素块的半空间填充算法

根据 PowerVR 体系结构特点,为了加快三角面填充速度,将三角面包围盒内像素划分为多个像素块(4×4),如图 3 所示,根据像素块和三角面的位置关系,像素块可以分为两类:完全在外的像素块(像素块 1)和不完全在外的像素块(像素块 2,3,4)。

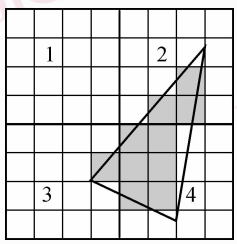


图 3　像素块分类示意图

Fig. 3　Diagram of tile's classification

算法流程如下:

(1) 根据三角面包围盒划分像素块;

(2) 对像素块逐个进行分类判断;

(3) 对不完全在外的像素块进行像素块内扫描;

在判断像素块类型时,只有像素块 4 个顶点都位于三角面某条边的外面时才表明该像素块为完全在外的像素块,如图 4 所示。一般先计算出像素块 4 个顶点的同一边界方程值 $e_i(A)$,$e_i(B)$,$e_i(C)$,$e_i(D)$,$i=0,1,2$。只有 4 个边界方程值都小于零时才表明该像素快位于三角面内部。这样计算复杂度较大,需要预先计算出 12 个边界方程值。

根据三角面和矩形的相交测试原理[1],可以对算法做进一步优化。由边界方程 $e=ax+by+c$ 可知,矢量 $\boldsymbol{n}=(a,b)$ 为边的法线方向,因为顶点 A 在法线上的投影点距离边 p_0p_1 最近,只要顶点 A 位于边 p_0p_1 的外面,其他 3 个顶点肯定位于边 p_0p_1 的外面,所以只需计算顶点 A 对于边 p_0p_1 的边界方程值。同理对其余的 2 条边也只需各计算 1 个顶点的边界方程值,就可判断出像素块是否为完全在外的像素块。

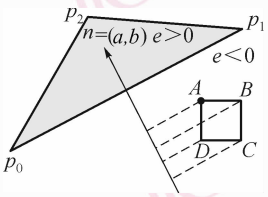


图 4 像素块和三角面的相交测试

Fig. 4 Intersection testing for tile and triangle

而根据边的法线方向(n_x,n_y),即可得到每条边需要测试的像素块顶点位置。

$$\begin{cases} n_x<0,n_y<0,\text{左下角顶点} \\ n_x>0,n_y<0,\text{右下角顶点} \\ n_x<0,n_y>0,\text{左上角顶点} \\ n_x>0,n_y>0,\text{右上角顶点} \end{cases} \tag{11}$$

这使得原来需要计算 12 次边界方程的计算量减少到了 3 次,提升了像素块的处理速度。

2.2 Zigzag 扫描顺序

像素块的扫描顺序很大程度上决定三角面的填充效率。通过对各种常见扫描算法的分析及实验测试,采用 Zigzag 扫描算法[6,9],如图 5 所示。算法首先从底部的像素块出发,水平扫描当前像素块行,直到遇到第 1 个完全在外的像素块,然后 Y 坐标增加一个像素块高度,再次反向扫描当前像素块行,依此反复扫描。当前像素块行的 Y 坐标决定当前行的扫描方向。如 Y 坐标为奇数就向左扫描,如为偶数则向右扫描。该算法计算复杂度小,算法结构简单,易于硬件的快速实现。

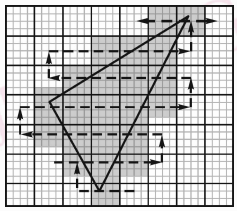


图 5 Zigzag 扫描曲线示意图

Fig. 5 Diagram of Zigzag scanning curve

像素块内部扫描一般采用逐行扫描策略。在 Zigzag 像素块扫描模块给出的边界方程值的基础上,利用递推能快速求得内部各像素的边界方程值,从而判断出像素是否位于三角面内部。

2.3 重心坐标插值

采用的像素插值算法基于重心坐标,是平面插值算法的一种优化改进。基于当前像素和三角面 3 个顶点所围成子三角形的面积 A_0,A_1,A_2,如图 6 所示,将 3 个顶点的属性 s_0,s_1,s_2 以子三角形面积进行加权平均,即得到当前像素该属性的插值,插值公式为

$$s(p)=\frac{A_0s_0+A_1s_1+A_2s_2}{A_\Delta} \tag{12}$$

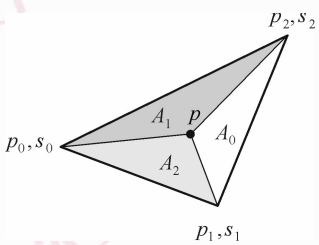


图 6 重心坐标插值示意图

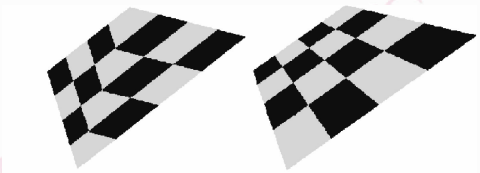
Fig. 6 Diagram of barycentric coordinates interpolation

其中整个三角面的面积倒数 $1/A_\Delta$ 只需要在三角面建立阶段计算一次,而当前像素的 3 个边界方程值 e_0,e_1,e_2 正是子三角形面积 A_0,A_1,A_2 的两倍,如式(13)所示,所以子三角形面积不需要额外计算。故重心坐标插值的计算复杂度小,易于硬件高效实现。

$$e_2=\overrightarrow{p_0p_1}\times\overrightarrow{p_0p}=2A_2 \tag{13}$$

2.4　透视投影校正

在透视投影下,同样大小的物体离屏幕越远,在屏幕上显示的越小。因此,为了达到逼真的图形渲染效果,如图 7 所示,必须对线性插值的重心坐标算法做出调整才能满足透视投影的非线性特点。



(a) 无透视投影校正　　(b) 有透视投影校正

图 7　纹理透视校正效果

Fig. 7　The effect of texture projective correction

文献[1]中给出的一般透视投影校正做法,如式(14)~式(16)所示,先计算出 3 个顶点的属性 s_i 和齐次因子 w_i 的商 s_w^i,然后分别对商 s_w 和齐次因子倒数 w_r 进行线性插值,最后将插值得到的商 $s_w(x,y)$ 和齐次因子倒数 $w_r(x,y)$ 相除即可得到正确插值结果 $s(x,y)$。这涉及到每个像素每个属性 1 次的除法,这样的计算复杂度在嵌入式设备上是不可取的。

$$s_w^i = \frac{s_i}{w_i} \qquad i = 0,1,2 \tag{14}$$

$$w_r^i = \frac{1}{w_i} \qquad i = 0,1,2 \tag{15}$$

$$s(x,y) = \frac{s_w(x,y)}{w_r(x,y)} \tag{16}$$

为了避免每个像素 1 次的除法运算,并且和基于像素块的扫描算法结合起来,如图 8 所示,提出一种基于局部线性插值的算法优化方案:

(1) 根据纹理透视校正算法计算出像素块 4 个子区域中心的真实纹理地址 (u_{c_i}, v_{c_i}), $i = 0,1,2,3$,需要 4 次除法;

(2) 根据子区域中心纹理地址,如式(17)和式(18)示,计算纹理地址的局部梯度增量 $\mathrm{d}u/\mathrm{d}x$, $\mathrm{d}u/\mathrm{d}y$, $\mathrm{d}v/\mathrm{d}x$, $\mathrm{d}v/\mathrm{d}y$;

$$\frac{\mathrm{d}u}{\mathrm{d}x} = \frac{u_{c_1} - u_{c_0}}{2} \tag{17}$$

$$\frac{\mathrm{d}u}{\mathrm{d}y} = \frac{u_{c_2} - u_{c_0}}{2} \tag{18}$$

(3) 根据像素块子区域中心纹理地址和纹理地址的局部梯度增量,如式(19)和式(20)所示,通过线性插值计算像素块内所有像素的纹理地址;

$$u_1 = u_{c_0} - 0.5 \times \frac{\mathrm{d}u}{\mathrm{d}x} - 0.5 \times \frac{\mathrm{d}u}{\mathrm{d}y} \tag{19}$$

$$u_2 = u_{c_0} + 0.5 \times \frac{\mathrm{d}u}{\mathrm{d}x} - 0.5 \times \frac{\mathrm{d}u}{\mathrm{d}y} \tag{20}$$

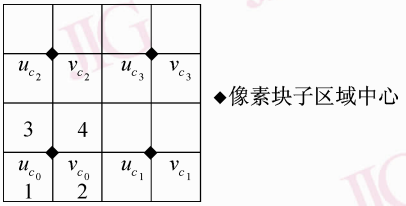


图 8　纹理的局部线性插值

Fig. 8　Locally linear interpolation of texture

这种算法渲染的透视校正效果和传统透视校正效果基本相同,但是除法运算由原来的每个像素块 16 次减少到了 4 次,节省了硬件面积。

3　光栅处理器硬件实现

3.1　硬件框图

按照功能,光栅处理器的硬件架构[10,11]可以分为 3 部分:三角面建立部分、三角面扫描部分和像素插值部分,其中三角面扫描部分可以分为像素块扫描模块和像素块内部扫描模块,而像素插值部分按照是否具有透视投影校正功能,又可以分为平面插值模块和纹理插值模块等。光栅处理器的硬件框图如图 9 所示。

其中透视校正模块,如图 10 所示,主要负责纹理透视校正。根据像素块第 1 个子区域中心 C_0 的纹理地址和齐次因子的商 $(u_{c_0}/w_{c_0}, v_{c_0}/w_{c_0})$ 和齐次因子的倒数 $1/w_{c_0}$,如图 8 所示,计算出 C_0 的真实纹理地址 (u_{c_0}, v_{c_0}),然后通过递推计算子区域中心 C_1, C_2 纹理地址和齐次因子的商 $(u_{c_1}/w_{c_1}, v_{c_1}/w_{c_1})$, $(u_{c_2}/w_{c_2}, v_{c_2}/w_{c_2})$ 和齐次因子的倒数 $1/w_{c_2}$, $1/w_{c_3}$,继而求出子区域中心 C_1, C_2 的真实纹理地址 (u_1, v_1), (u_2, v_2)。根据 3 个相邻子区域中心的真实纹理地址,计算出纹理地址的局部梯度增量 $\mathrm{d}u/\mathrm{d}x$, $\mathrm{d}u/\mathrm{d}y$, $\mathrm{d}v/\mathrm{d}x$, $\mathrm{d}v/\mathrm{d}y$。最后根据子区域中心的纹理地址和纹理地址局部梯度增量插值计算像素块内所有像素的纹理地址。在实际操作中,为了提高硬件速度,倒数单元和乘法器都分多级流水线操作。

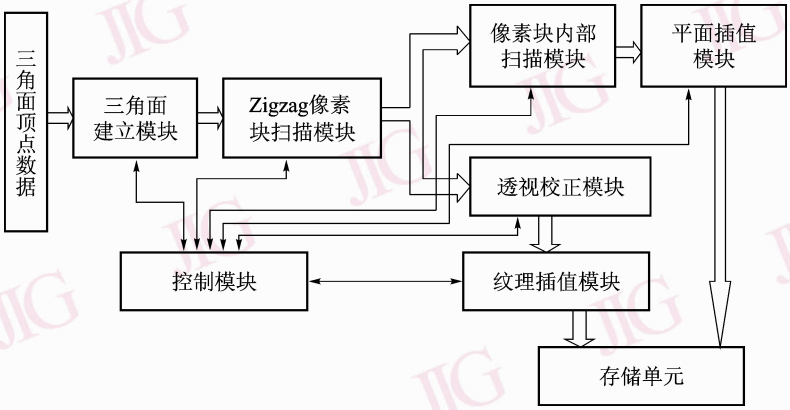


图 9 光栅处理器硬件架构

Fig. 9 Block diagram of rasterizer architecture

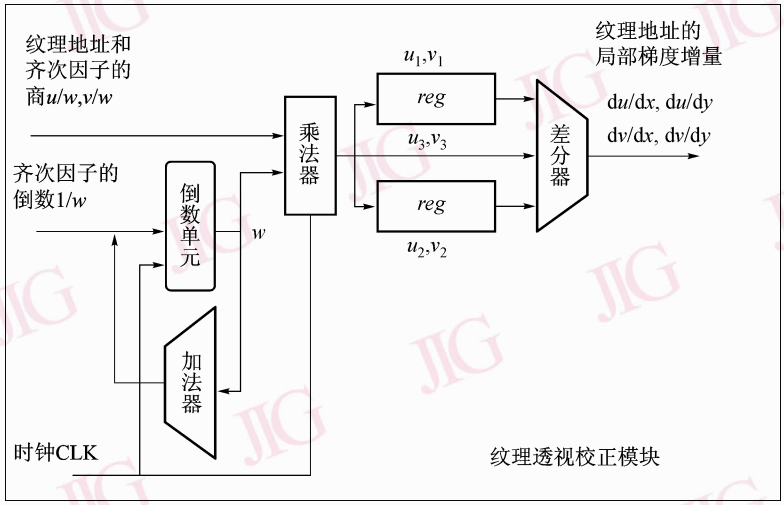


图 10 纹理透视校正单元

Fig. 10 Texture projective correction unit

3.2 模块描述

三角面建立模块负责预先计算出三角面扫描时需要的所有常量,输入数据为三角面的 3 个顶点,输出数据为三角面的边界方程和属性的平面插值系数。

Zigzag 像素块扫描模块负责筛选出有效像素块,快速剔除三角面外的像素块,输入数据为三角面的 3 个顶点坐标及边界方程,输出数据为有效像素块的屏幕坐标及相关的边界方程值。

像素块内部扫描模块负责判断出有效像素,并计算相应的边界方程值,输入数据为像素块的屏幕坐标和边界方程值,输出数据为有效像素的屏幕坐标及对应的边界方程值。

平面插值模块[12]负责根据边界方程值,线性插值计算出像素属性,输入数据为像素的边界方程值,属性的平面插值系数,输出数据为像素的属性。

透视校正模块负责纹理地址的透视校正,输入数据为 3 个顶点处的纹理地址和齐次因子,像素块的边界方程值,输出数据为像素块 4 个子区域中心的真实纹理地址以及纹理地址的局部梯度增量。

纹理插值单元负责在像素块内对纹理地址进行局部线性插值,输入数据为像素块 4 个子区域中心的真实纹理地址以及纹理地址的局部梯度增量,输出数据为像素纹理地址。

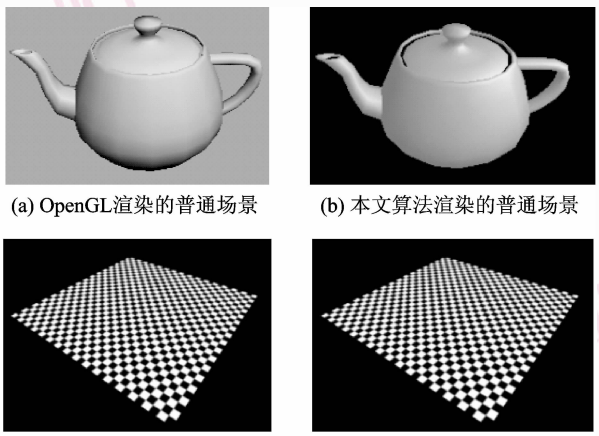
4 实验结果与分析

为了测试图形光栅硬件实现算法的性能,首先用 C/C ++ 语言对算法进行了软件模拟[13],开发出

支持 OpenGL® ES 1.1 标准[14]的 3D 图形引擎[15-16]，然后用 FPGA(Xilinx Vertex2P xc2vp30ff-896)验证了算法的 RTL 代码(Verilog HDL)。

4.1　渲染质量

将同一场景分别用 OpenGL® ES 1.1 和本文算法进行渲染,渲染效果如图 11 所示,从图中可以看出,算法的渲染效果和 OpenGL® ES 1.1 几乎一致,这说明算法在功能上正确,渲染的图形质量达到 OpenGL® ES 1.1 的渲染水平。



(a) OpenGL渲染的普通场景　(b) 本文算法渲染的普通场景

(c) OpenGL渲染的纹理透视校正　(d) 本文算法渲染的纹理透视校正

图 11　本文算法和 OpenGL 渲染图像的对比

Fig. 11　The comparison between OpenGL and our algorithm

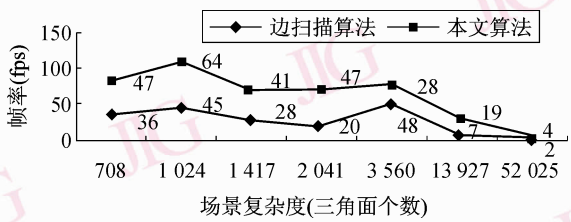


图 12　不同算法软件渲染速度对比

Fig. 12　The speed comparison between different algorithms

4.2　渲染速度

通过将具有不同场景复杂度的场景分别用本文算法和边扫描算法进行渲染,如图 12 所示,本文算法的渲染速度明显比边扫描算法快,尤其在场景复杂度小时。这说明本文算法在保持较好渲染效果的同时,有效地减小了算法的计算复杂度。

同时根据实验发现,基于嵌入式平台的一般 3D 图形渲染中,场景的平均复杂度为每帧 10 万个像素块左右,而本文算法在 FPGA 上每个时钟渲染 1 个像素,渲染 1 个像素块(4×4)需要 16 个时钟,而要达到实时渲染,帧率必须不小于 30 帧/秒,此时需要的时钟频率至少达到 48 MHz,而本文算法在 FPGA 上的综合频率为 60.87 MHz,能够满足实时性要求。如果用 ASIC 实现,模块速度能进一步提升,再加上多通道的并行渲染,完全满足一般的基于嵌入式平台的 3D 图形实时渲染要求。

4.3　硬件消耗资源

算法在 FPGA 上综合主要消耗了以下硬件资源,如表 1 所示。

表 1　光栅处理器综合后的 FPGA 资源

Tab. 1　The FPGA synthesis resources of rasterizer

单元名称	位宽	个数
乘法器	18×18	23
寄存器	1	2 016
4 输入 LUT	/	2 666
BRAM	16×256	2

4.4　仿真波形

综合后用 Modelsim 进行仿真,仿真波形图如图 13 所示,仿真结果和软件模拟完全一致。

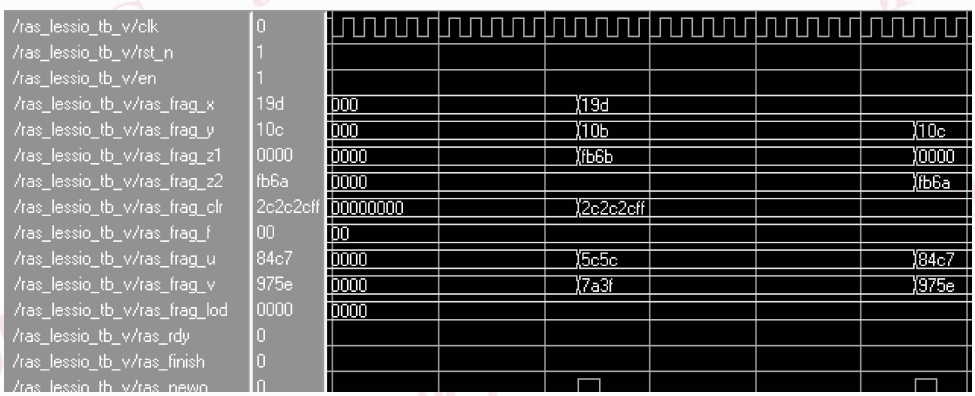


图 13　具有 2 个像素的三角面扫描仿真波形图

Fig. 13　Simulated waveform of a triangle with 2 pixels

5 结 论

在 PowerVR 体系结构和半空间填充算法的基础上,提出了一种适合嵌入式平台的图形光栅的硬件实现算法。该算法的核心是在基于像素块的半空间填充算法和 Zigzag 扫描算法的基础上,利用重心坐标进行像素插值,并且采用优化后的纹理透视校正算法。与传统的光栅算法相比,本文算法加快了三角面的扫描速度,提高了像素命中率,减小了计算复杂度,更加易于硬件实现。

FPGA 验证结果表明,算法渲染的图形质量达到了 OpenGL® ES 1.1 渲染效果;在一般场景下的渲染速度达到 30 帧/秒,满足实时渲染要求;在 FPGA Vertex2P xc2vp30-7ff89 上的综合资源为 5 545 个 Slice,硬件消耗小,满足嵌入式平台的 3D 图形渲染要求。在将来的工作中,可以加入早期深度测试和反走样测试信息等以进一步完善模块功能。

参考文献(References)

1 Tomas Akenine-Möller, Eric Haines. Real-Time Rendering[M]. 2nd ed. Translated by Pu Jian-tao. Beijing: Peking University Press, 2004. [Tomas Akenine-Möller, Eric Haines 著. 实时计算机图形学[M]. 第 2 版. 普建涛译. 北京:北京大学出版社,2004.]

2 Foley James D, Dam Andries van, Feiner Steven K, *et al*. Computer Graphics: Principles and Practice, Second Edition in C [M]. 2nd ed. Translated by Tang Ze-sheng, Dong Shang-hai. Beijing: Mechine Press, 2004. [James D. Foley 等著. 计算机图形学原理及实践:C 语言描述[M]. 第 2 版. 唐泽圣,董上海等译. 北京:机械工业出版社,2004.]

3 Akeley Kurt, Jermoluk Tom. High Performance polygon rendering [J]. Computer Graphics, 1988, **22**(4): 239-246.

4 Kugler Ander. The setup for triangle rasterization[A]. In: Proceedings of the 11th Eurographics Workshop on Computer Graphics Hardware [C], Poitiers, France, 1996: 49-58.

5 Yu Jiang-yi. Scan Converting Triangles-Lecture 06 CISC 440/640 Spring 2007 [EB/OL]. http://www.cis.udel.edu/~yu/Teaching/CISC4 40_07S/handouts/Lecture06.pdf.

6 Pineda Juan. A parallel algorithm for polygon rasterization [A]. In: Proceedings of ACM SIGGRAPH 1988[C], Atlanta, Georgia, USA, 1988: 17-20.

7 Waller Marcus, Ewins Jon, White Martin, *et al*. Efficient primitive traversal using adaptive linear edge function algorithms [J]. Computer Graphics, 1999, **23**: 365-375.

8 PowerVR. PowerVR White Paper: 3D Graphical Processing[EB/OL]. http://www.powervr.com/pdf/TBR3D.pdf.

9 McCormack Joel, McNamara Robert. Tiled polygon traversal using half-plane edge functions [A]. In: Proceedings of the 2000 SIGGRAPH/ EUROGRAPHICS Workshop on Graphics Hardware [C], Interlaken, Switzerland, 2000: 15-21.

10 Park Woo-chan, Lee Kil-whan, Kim Il-san, *et al*. An effective pixel rasterization pipeline architecture for 3D rendering processors [J]. IEEE Transactions on Computers, 2003, **52**(11): 1501-1508.

11 ATI Technologies Incorporation Radeon X800: 3D Architecture Whitepaper [EB/OL]. http://ati.de/products/radeonx800/RadeonX800 Architecture WhitePaper.pdf.

12 Oberman Stuart, Siu Michael. A high-performance area-efficient multifunction interpolator [A]. In: IEEE Symposium on Computer Arithmetic[C], Cape Cod, MA, USA, 2005: 272-279.

13 Moya Victor, González Carlos, Roca Jordi, *et al*. A cycle-level execution-driven simulator for modern GPU architectures [A]. In: IEEE International Symposium on Performance Analysis of Systems and Software[C], Austin, Texas, USA, 2006: 231-241.

14 Blythe David, Munshi Aaftab, Leech Munshi Joe. OpenGL® ES Common/Common-Lite Profile Specification Version 1.1.10 (Full Specification) [EB/OL]. http://www.khronos.org/registry/gles/specs/1.1/es_full_spec.1.1.10.pdf.

15 Akenine Möller Tomas, Ström Jacob. Graphics for the masses: A hardware rasterization architecture for mobile phones [J]. ACM Transactions of Graphics, 2003, **22**(3): 801-808.

16 Crisu Dan, Cotofana Sorin D, Vassiliadis Stamatis, *et al*. GRAAL - A development framework for embedded graphics accelerators [A]. In: Proceedings of Design, Automation and Test in Europe[C], Paris, France, 2004: 1366-1367.