Floating Point and CPU vs GPU Rendering

Sam Moore, David Gow

July 24, 2014

Abstract

We qualitatively illustrate differences between floating point operations on the x86-64 CPU and several GPUs by rendering filled ellipses.

1 Introduction

The IEEE Standard for Floating-Point Arithmetic [1] has been widely adopted by hardware manufacturers of CPUs and programming language standards.

Although it is well known that the behaviour of GPU drivers is inconsistent, there is little formal academic research into the behaviour of floating point operations using such drivers.

In 2004 Hillesland and Lastra adapted Kahan's well known program for testing floating point arithmetic on CPUs during the 1980s "Paranoia" for GPUs and found that many GPUs did not appear to be compliant with IEEE-754[2].

Given the recent interest in use of the GPU for vector graphics[3] the behaviour of GPUs when performing floating point operations is worthy of closer investigation.

Using a straight forward filled ellipse rendering algorithm implemented in C/C++ and GLSL we show inconsistent floating point behaviour when comparing the x86-64 CPU, an nVidia GPU[1], an intel GPU[2] and an AMD/ATI GPU[3].

2 Algorithm

For each pixel position (x, y) normalised relative to the bounding rectangle, if $x^2 + y^2 \leq 1$ then (x, y) should be filled.

Although x and y may be treated as integers on the CPU, since the OpenGL API requires floating point vertex coordinates, our CPU implementation also normalises the coordinates relative to the bounding rectangle; this way we can compare the performance of floating point operations on the CPU and GPU(s).

[1] ??? using the nVidia driver
[2] ??? using the intel driver
[3] Whistler LE (Radeon HD 6610M/7610M) using the fglrx driver

2.1 GLSL Fragment Shader

```
#version 140
// Fragment shader (others omitted)

in vec2 objcoords; // Coordinates x, y, relative to bounding rectangle (from other shaders)
out vec4 output_colour;

uniform vec4 colour;

void main()
{
        if ((objcoords.x)*(objcoords.x) + (objcoords.y)*(objcoords.y) > 1.0)
        {
                discard;
        }
        output_colour = colour;
}
```

2.2 CPU Rendering Algorithm (simplified)

```
// where bounds = {x,y,w,h} gives the bounding rectangle in integer pixel positions
// and centre = {x,y} is the centre of the circle
// and pixels[][] is the display buffer
for (int x = bounds.x; x < bounds.x+bounds.w; ++x)
{
        for (int y = bounds.y; y < bounds.y+bounds.h; ++y)
        {
                float dx = 2.0*(float)(x - centre.x)/(float)(bounds.w);
                float dy = 2.0*(float)(y - centre.y)/(float)(bounds.h);
                if (dx*dx + dy*dy <= 1.0)
                {
                        pixels[x][y] = true;
                }
        }
}
```

Note: The `pixels` buffer is uploaded directly to the GPU after CPU rendering is completed.

3 Results

Figure 1 shows the edge of a unit radius circle viewed under a magnification of approximately 5×10^6 as rendered using the CPU.

4 Conclusion

nVidia looks qualitatively similar to the CPU rendering. Frankly I was just happy fglrx didn't segfault. Wierd shit happens with intel. If anyone isn't obeying IEEE-754 here, it is probably intel.

References

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[2] Karl E Hillesland and Anselmo Lastra. Gpu floating-point paranoia. *Proceedings of GP 2004*, 2004.

[3] Mark J Kilgard and Jeff Bolz. GPU-accelerated path rendering. *ACM Transactions on Graphics (TOG)*, 31(6):172, 2012.

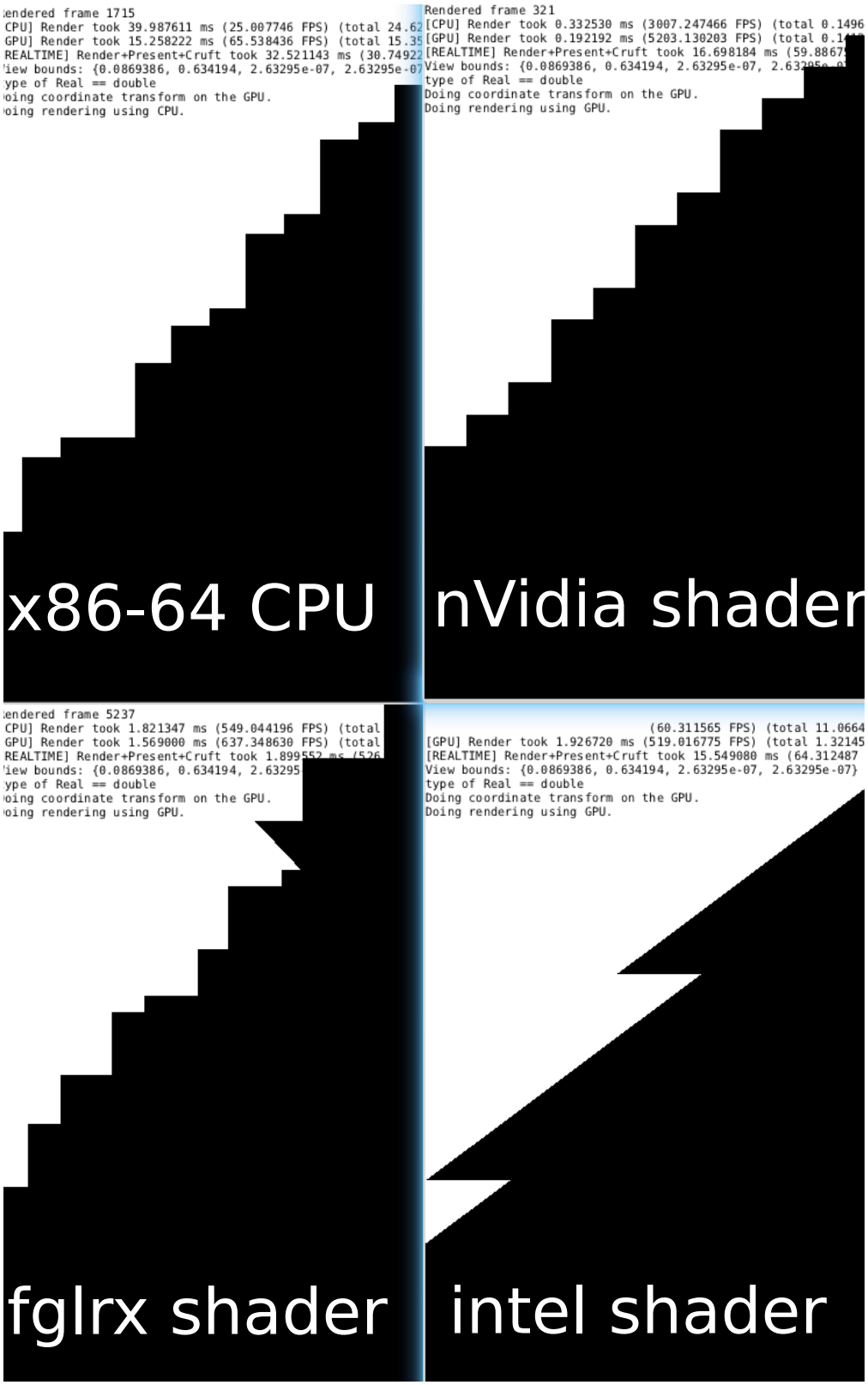

Figure 1: The edges of a unit circle viewed through bounds (x,y,w,h) = (0.0869386,0.634194,2.63295e-07,2.63295e-07)