Technical Brief

Floating-Point Specials on the GPU

Table of Contents

Floating-Point Specials on the GPU

Cem Cebenoyan

cem@nvidia.com

Floating-Point Specials on the GPU

Abstract

Floating-point math sometimes results in the generation of floating-point specials (such as NaNs and Infs), on a CPU as well as on GPUs. These special numbers often result in unintuitive rendering results that are hard to debug. This paper describes when floating-point specials are generated in various stages of a modern GPU pipeline, and discusses methods for handling and protecting against these situations.

What Are Floating-Point Specials?

Floating-point specials arise when the result of a floating-point math operation is either undefined or out of the range of representable values for a given floating-point format. For example, dividing a positive non-zero number by zero will result in a special value of +Inf (positive infinity), and dividing a negative non-zero number by zero will result in –Inf (negative infinity). Dividing zero by zero will result in NaN[1] (not-a-number).

CPUs have been generating these special numbers for years (they are in the IEEE 754 floating-point spec, and more readably explained in [Goldberg 91]), and programmers include code to handle these cases because they have come to expect them for certain operations on the CPU (like divide-by-zero). The generation of these values on the GPU, however, is often surprising, due to a few differences between current GPU and CPU programming environments:

- Not all GPUs fully support floating-point specials. All NVIDIA GPUs that support floating-point math do, but this does not generally apply to all IHVs.
- Due to support for half-precision (fp16) values in shaders, textures, and render targets, overflow conditions are much easier to generate. While a single precision (fp32) floating-point value will overflow (generating +Inf) at around 3.4e38, fp16 values overflow at a more achievable 65504.
- Lack of full-featured GPU hardware debuggers can make these issues more difficult to diagnose.

[1] There is only one NaN format in the GeForce FX and GeForce 6 series. There is no SNaN or QNaN distinction. All NaNs are, essentially, QNaNs.

- Until recently, GPU programming was restricted to using fixed-point arithmetic, so most developers aren't used to handling these cases.

For these reasons, the best way to prevent problems with floating-point specials in your app is to write your GPU algorithms robustly, actively preventing their creation.

Specials in the Rendering Pipeline

Many stages of the GPU pipeline can generate (or at least propagate) specials; we concentrate here on the shading stages (vertex and fragment), texturing operations, and framebuffer access. Keep in mind that specials propagate from one stage to the next, so outputting a special from the fragment shader is guaranteed to result in a special reaching the framebuffer, if a pixel is written.

Vertex and Fragment Shading

Many basic shading operations generate special results depending on input. See tables 1, 2, and 3 for a few specific examples.

Table 1. Result of multiplication and addition of special numbers. (X denotes a non-special number)

A	B	(A * B)	(A + B)
0.0	+/- Inf	NaN	+/- Inf
+Inf	-Inf	-Inf	NaN
+/- Inf	non-zero X	+/- Inf	+/- Inf
NaN	X	NaN	NaN

Table 2. Result of unary operations.
(-X denotes a negative non-special number)

A	rcp(A)	rsq(A)	$\log_2(A)$	$\exp_2(A)$
0.0	+Inf	+Inf	-Inf	1.0
+Inf	0.0	0.0	+Inf	+Inf
-Inf	0.0[2]	NaN	NaN	0.0
-X	1 / -X	NaN	NaN	2^{-X}
NaN	NaN	NaN	NaN	NaN

Table 3. Result of comparisons of special numbers[3]

A	B	max(A, B)	min(A, B)	(A == B)
+Inf	+Inf	+Inf	+Inf	1
+Inf	-Inf	+Inf	-Inf	0
+/- Inf	NaN	NaN	NaN	0
NaN	NaN	NaN	NaN	0

We lump vertex and fragment shading together here because the handling of specials for similar operations are identical, however, there are some important practical differences:

- Fragment shaders can operate on fp16 values (in Direct3D, this is done using the **_pp** instruction modifier for shader assembly, or the **half** data type in HLSL), while vertex shaders do all operations on fp32 values. fp16 values generally operate identically to fp32 values, but will overflow to +/- Inf much more easily. The maximum representable fp16 value is 65504.
- Fragment shaders often output to a different precision floating-point format in the framebuffer, resulting in a conversion that can cause finite values to overflow. For example, when writing out to an fp16 render target, a finite but

[2] Strictly speaking, this computation produces -0.0 (negative zero).

[3] Note that these tables list the results of operations from the point-of view of the GPU hardware. Practically, a compiler could, for example, optimize out or reorder an operation like (A == B) depending on inputs, resulting in behavior that seems different from that listed in the tables above.

large fp32 value in the fragment shader will get converted to +Inf in fp16 format. This overflow generation of Inf does not happen with vertex shaders because a large finite fp32 value in the shader either gets written to a high-precision interpolator (like a texture coordinate) in which case it is preserved, or is converted to a low-precision fixed-point format (like for the diffuse interpolator or the fog distance) in which case it is simply clamped to [0..1].

Texturing

Surprisingly, simple texturing can also generate and propagate specials. Obviously, if your floating-point textures contain special numbers, for textures fetched from the vertex or fragment shader, these are passed through to the shader upon texture fetch. More subtly, floating-point texture filtering (currently supported by GeForce 6800 and 6600 GPUs in the fragment shader for fp16 texture formats) can generate special numbers, depending on input.[4]

It may seem strange for special values to exist inside a texture, but this actually occurs frequently when doing framebuffer post-processing effects, such as blur, on a floating-point buffer. In this case, if any specials exist in the framebuffer (such as an Inf due to overflow in a previous rendering pass) these are fetched by texture in the blur pass and, due to the nature of many blur algorithms, propagated over the (entire) screen.

Framebuffer Operations

Framebuffer operations can have especially subtle interactions with floating-point specials. There are a couple of specific gotchas to look out for when dealing with the framebuffer:

- As stated earlier, outputting a finite value from the shader can cause an overflow if the render target format is of lower precision than the shader output.
- Alpha blending operations are able to generate specials on hardware that supports alpha blending with floating-point surfaces (the GeForce 6 series supports fp16 per-component alpha blending). So the shader rules for arithmetic apply here as well. For example, if you output an Inf in the RGB channels, a zero in the alpha channel, and set SRCALPHA:INVSRCALPHA blending, you will generate a NaN while blending which gets written to the framebuffer. Another, more subtle example: if you are summing lights into a floating-point buffer (ONE:ONE blending) and (SRCCOLOR + DESTCOLOR) is greater than the maximum representable value of the framebuffer format, you will get overflow, resulting in +Inf being output. Note that this can happen even if all inputs (SRCCOLOR and DESTCOLOR) are finite values.

[4] Filtering (and texture fetch in general) will not, however, generate specials if specials do not exist in the source texture. Finiteness is preserved by texture.

Visualizing Specials

One of the tell-tale signs of a floating-point special problem is if your application periodically has all or part of the screen turn black or white. Since you cannot directly visualize a floating-point surface on a GeForce6 GPU, most algorithms dealing with floating-point data convert them to fixed-point at some point. In this step, a +Inf becomes white (1.0), and –Inf and NaN become black. Since most every value involved in an arithmetic expression with a special becomes a special (see Table 1), a single special value ends up propagating over a significant number of pixels in the final frame buffer. This is especially true for algorithms where a single destination pixel is influenced by many source pixels (for example, a blur).

Protecting Against Specials

There's only one way to eradicate specials from your GPU algorithms: make sure they do not get created in the first place. There are two basic ways to create a special – with undefined arithmetic and due to overflow. We cover each in turn.

Arithmetic Specials

A special can be generated arithmetically with a simple bit of HLSL like this:

```
float3 vec = HalfAngleVec.xyz; //grab half angle vector
float vecLen = length(vec);    //compute length
vec /= vecLen;                 //normalize (could divide by zero!)
```

If vec is of zero length, this snippet will cause vec to be the three vector (+Inf, +Inf, +Inf) which will likely wreak havoc on the rest of the shader.

A simple solution to add robustness and avoid a special is to test for the zero-length condition, just as you would in a CPU algorithm:

```
float3 vec = HalfAngleVec.xyz; //grab half angle vector
float vecLen = length(vec);    //compute length
if (vecLen != 0)
       vec /= vecLen;          //normalize, more safely
```

Note that this additional test can increase the cost of your shader, so it should only be done if the boundary condition is actually practically possible.

Overflow Specials

Specials generated due to overflow conditions are trickier. As previously stated, these can happen inside a shader, but are more likely to occur due to a range mismatch between values inside a fragment shader and the output render target format. Testing for this case is easy: simply add code to the end of your fragment shader to test for exceeding the bounds of the render target format:

```
//compute world-space position
float3 worldSpacePos = mul(objPosition, WorldTransform);

//perform lighting <snipped>

//clamp z to fp16 range
worldSpacePos.z = min(worldSpacePos.z, 65504);

//store world space depth in alpha, output is 4xfp16 surface
return float4(color, worldSpacePos.z);
```

Note that this technique of clamping on output is very powerful– it not only solves the case of overflow, but it also reduces the chance of a special getting generated by framebuffer blending, and also localizes specials generated arithmetically (like the divide by zero example in the previous section). After all, if what happens in the shader stays in the shader – and it does if you keep from writing specials to the output – you guarantee that specials do not propagate throughout your scene.

Conclusion

Floating-point specials are a relatively new concern in GPU programming -- before modern programmable GPUs with full floating-point throughout the pipeline, they just weren't something programmers had to worry about. They are now a very real issue, however, and creating robust rendering algorithms requires thinking about the corner cases where specials can appear, understanding how the GPU handles them, and adding code to check and handle these situations.

References

[Goldberg 91] D. Goldberg,What Every Computer Scientist Should Know About Floating-Point Arithmetic, ACM Computing Surveys, Volume 23 Issue 1, p. 5-48, 1991

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com