Game Developers Conference

Direct3D API Issues: Instancing and Floating-point Specials

Cem Cebenoyan

NVIDIA Corporation

SAN FRANCISCO CA MAR 7-11 GDC 05

Agenda

- Really two mini-talks today
- Instancing API
  - Usage
  - Performance / pitfalls
- Floating-point specials

DirectX 9 Instancing API

- **What is it?**
  - Allows you to avoid DIP calls and minimize batching overhead
  - Allows a single draw call to draw multiple instances of the same model
- **What is required to use it?**
  - Microsoft DirectX 9.0c
  - VS/PS 3.0 hardware

DirectX 9 Instancing API - Basic Idea

- **Multiple streams:**
  - Primary stream is a single copy of the model data
  - Secondary streams contain per instance data
- **Primary stream loops**
  - stream_index = index % instance_size
- **Secondary streams increment per-instance**
  - stream_index = index / instance_size

DirectX 9 Instancing API

- Controlled by a single API entry-point:

```
IDirect3dDevice9::SetStreamSourceFreq
 (UINT StreamNumber, UINT Setting)
```

- Setting parameter can be one of:
  - `D3DSTREAMSOURCE_INDEXEDDATA`
  - `D3DSTREAMSOURCE_INSTANCEDATA`
  - Bitwise OR with a particular value

DirectX 9 Instancing API

- **D3DSTREAMSOURCE_INDEXEDDATA**
  - This setting controls the number of instances to draw
  - Set on the primary stream
  - For example, to render 10 instances:

```
d3dDevice->SetStreamSourceFreq(0,
    D3DSTREAMSOURCE_INDEXEDDATA | 10);
```

DirectX 9 Instancing API

- D3DSTREAMSOURCE_INSTANCEDATA
  - This setting controls over how many instances the pointer on the stream is incremented
    - Almost always set to 1
  - Set on the instanced stream
  - For example:

```
d3dDevice->SetStreamSourceFreq(1,
  D3DSTREAMSOURCE_INSTANCEDATA | 1);
```

DirectX 9 Instancing - An Example

- 100-vertex trees
  - Stream 0 contains just the one tree model
  - Stream 1 contains model world transforms
    - Possibly calculated per frame
  - Vertex Shader is the same as usual, except you use the matrix from the vertex stream instead of the matrix from VS constants
- You want to draw 50 instances

DirectX 9 Instancing API - Dataflow

Vertex Stream 0

0	$(x_0\ y_0\ z_0)\ (n_{x0}\ n_{y0}\ n_{z0})$
1	$(x_1\ y_1\ z_1)\ (n_{x1}\ n_{y1}\ n_{z1})$
...	...
99	$(x_{99}\ y_{99}\ z_{99})\ (n_{x99}\ n_{y99}\ n_{z99})$

Vertex Stream 1

0	worldMatrix_0
1	worldMatrix_1
...	...
49	worldMatrix_{49}

Instance 0

$(x_0\ y_0\ z_0)$
$(n_{x0}\ n_{y0}\ n_{z0})$
worldMatrix_0

Vertex_0

$(x_1\ y_1\ z_1)$
$(n_{x1}\ n_{y1}\ n_{z1})$
worldMatrix_0

Vertex_1

⋮

Instance 1

$(x_0\ y_0\ z_0)$
$(n_{x0}\ n_{y0}\ n_{z0})$
worldMatrix_1

Vertex_0

$(x_1\ y_1\ z_1)$
$(n_{x1}\ n_{y1}\ n_{z1})$
worldMatrix_1

Vertex_1

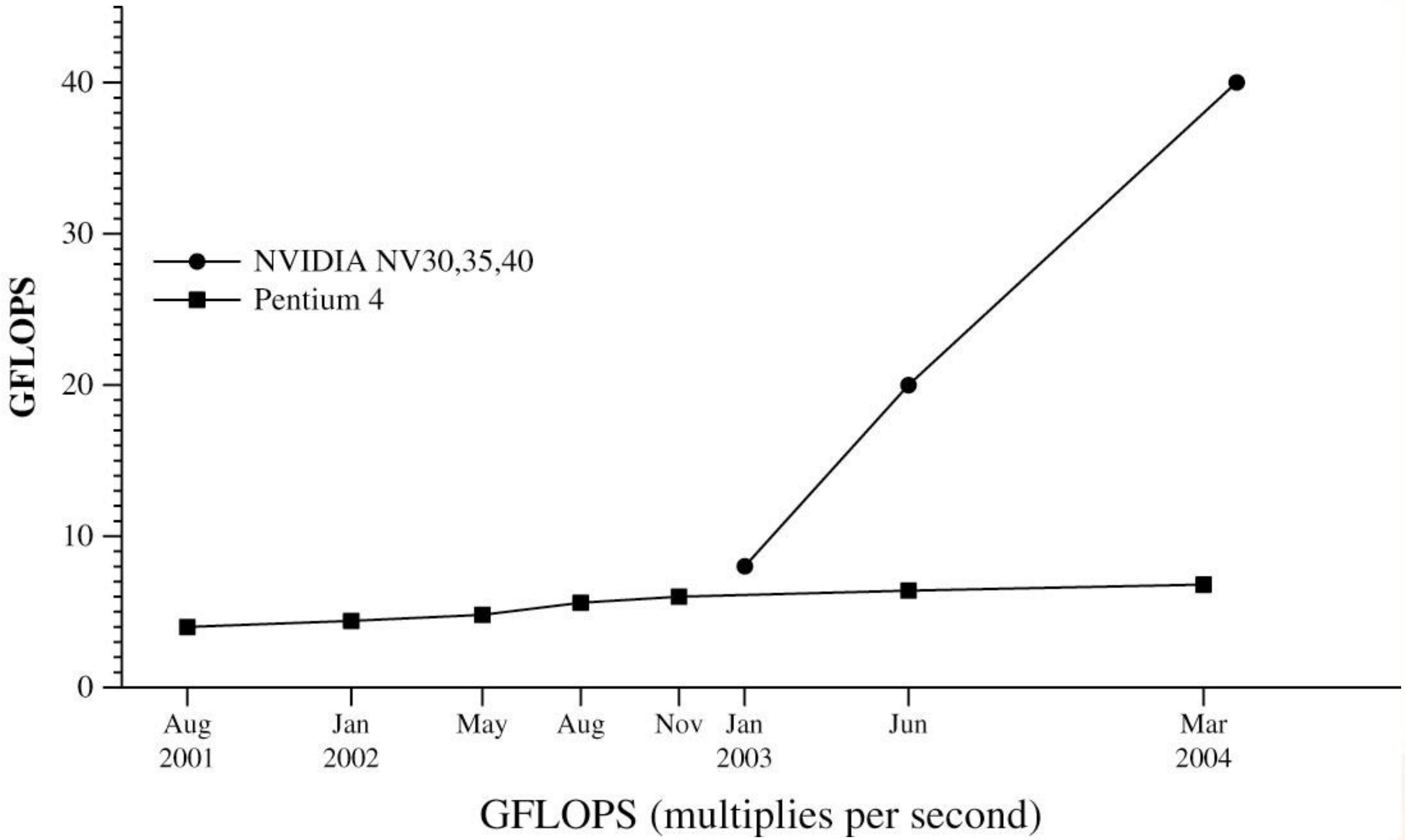
⋮

...

Why use instancing?

- **Batching is still the #1 performance issue in modern games**
  - **And it's only getting more important**
- **Instancing minimizes the `DrawIndexedPrimitive()`call overhead**
  - **In the Direct3D runtime, the operating system, the driver, and the hardware**

Batching is getting more important…



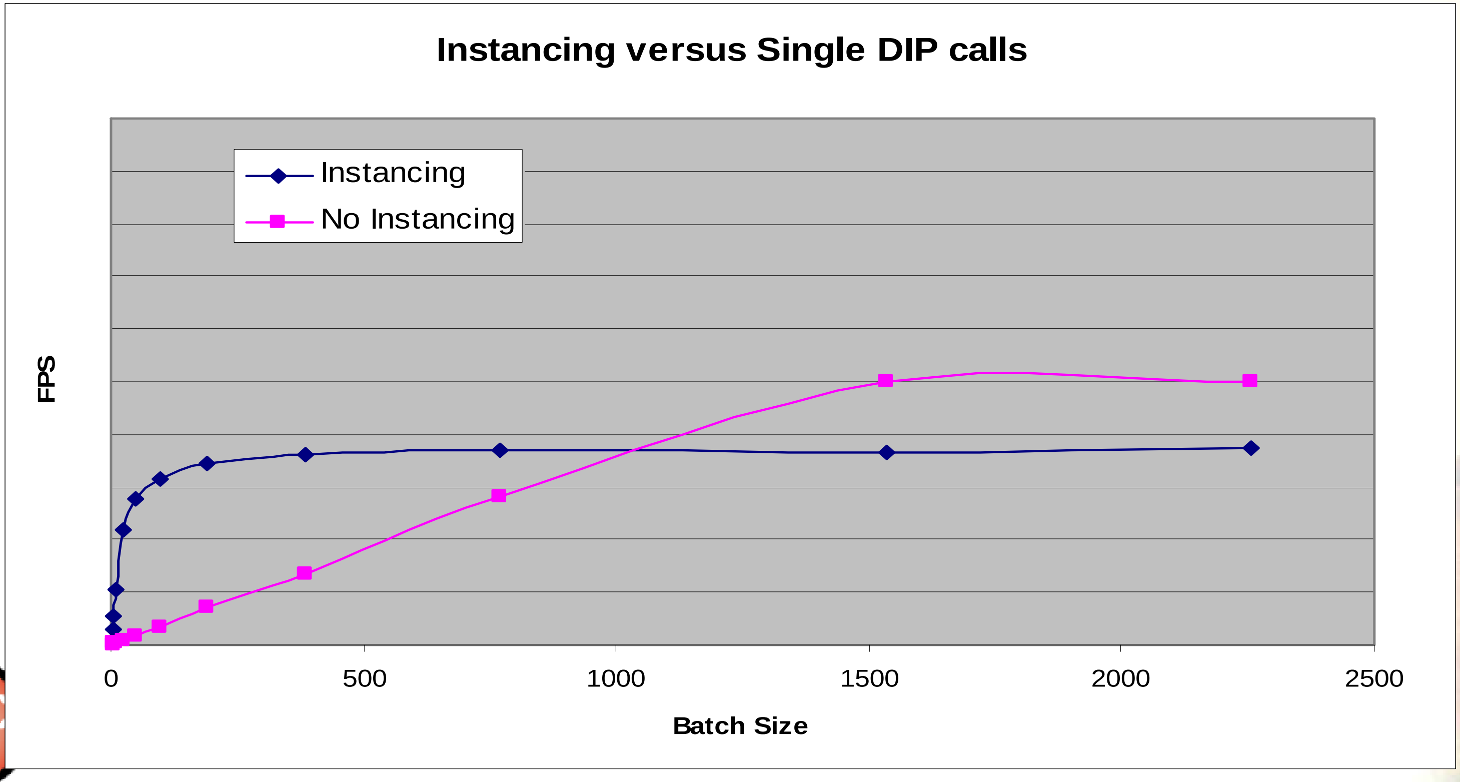
Courtesy Ian Buck, Stanford University

When to use instancing?

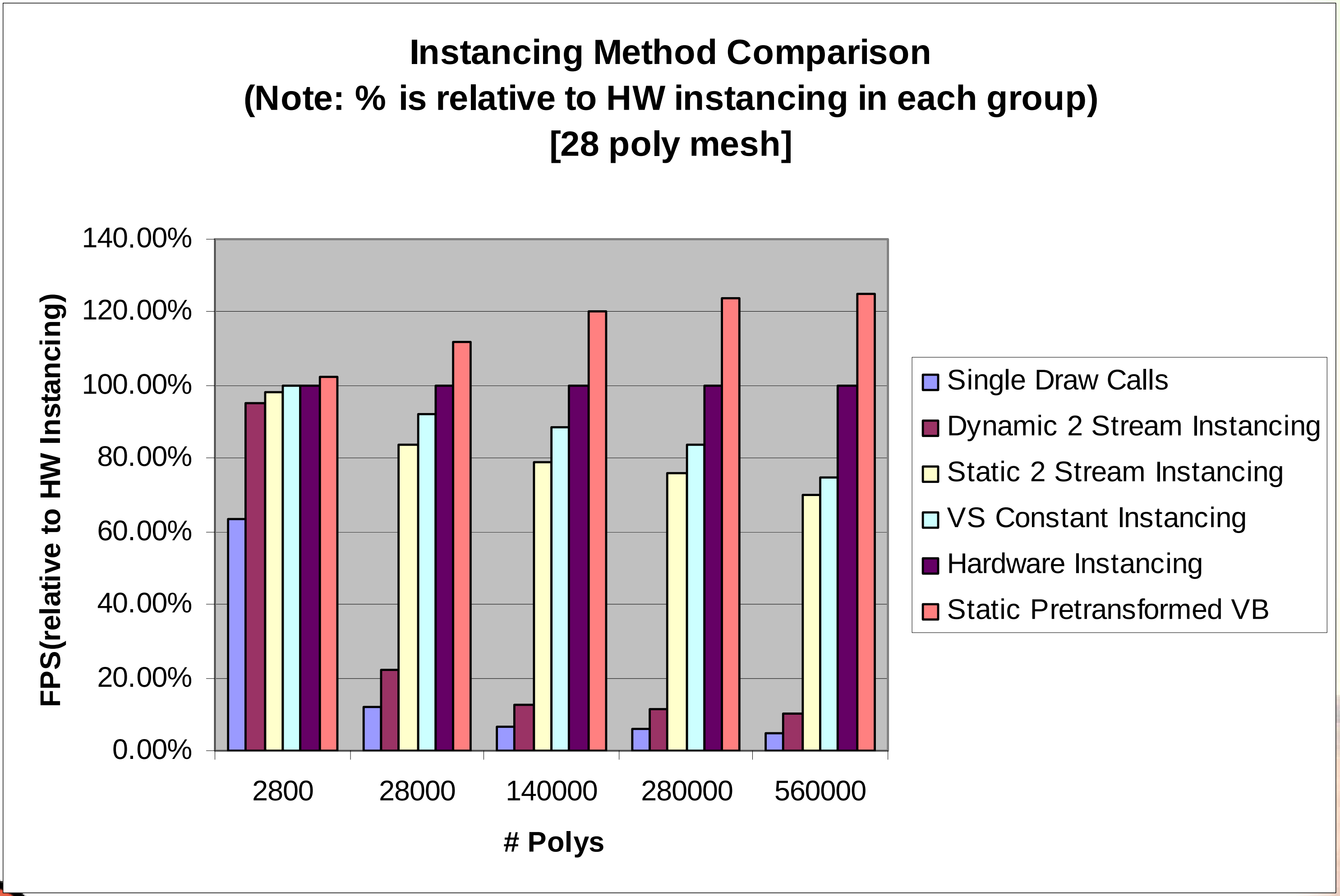
- Scene contains many instances of the same model
  - Forest of Trees, Particles, Sprites
- If you can encode per instance data in 2nd streams. i.e instance transforms, model color, indices to textures/constants.
- Less useful if your batch size is large
  - >1k polygons per draw
  - There is some fixed overhead to using instancing

Some Test Results

- Test scene that draws 1 million diffuse shaded polys
- Changing the batch size, changes the # of drawn instances
- For small batch sizes, can provide an extreme win as it gives savings PER DRAW CALL.
- There is a fixed overhead from adding the extra data into the vertex stream
- The sweet spot will change based on many factors (CPU Speed, GPU speed, engine overhead, etc)

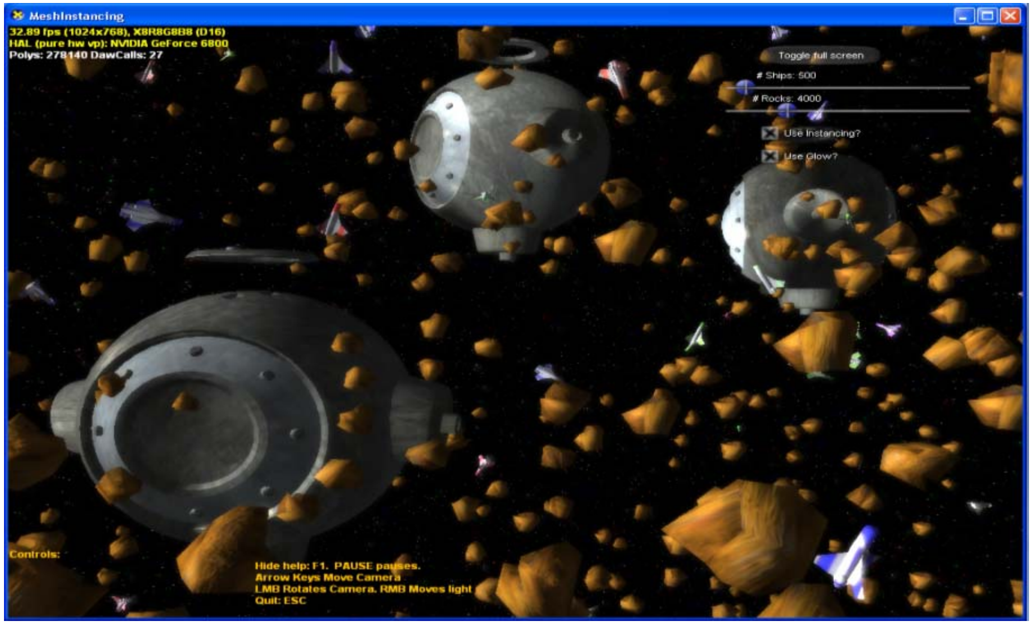


Instancing – Variations



Instancing Demo

- Space scene with 500+ ships, 4000+ rocks
- Complex lighting, post-processing
  - Some simple CPU collision work as well
- Dramatically faster with instancing



Instancing – Caution!

- It seems there are two factors that can hurt your performance with instancing
  - Becoming bus bandwidth bound
  - Becoming “attribute bound”
- But in reality there is only one
- This explains the slowdowns at the limit in the previous graphs

Instancing – Bandwidth Cost

- It seems that additional vertex data may have to be transferred over AGP / PCIE / local FB with instancing
- But, in reality, not an issue
  - The instanced streams have super locality
  - You hit the same data over and over!
- Attribute boundedness is what gets you

Instancing – Attribute Fetch Cost

- On modern HW, even given infinite bandwidth, vertices are not pulled infinitely fast
- Speed here is a function of the number of attributes in the input stream
- Pack input attributes as tightly as possible
  - Do not send data per-vertex that is constant for the whole scene!

Floating-point Specials

- What are floating-point specials?
- When and where can they occur on a GPU?
  - Following discussion pertains to all GeForceFX and GeForce6 GPUs
- What can you do about it?

What are FP Specials?

- Special numbers generated when fp math goes wrong
  - + / - Inf
  - NaN – Not a number
- Have been generated by CPUs for years
  - Defined by IEEE
- Now GPUs can generate them as well
  - Note that the following does not necessarily apply to all GPUs

FP Specials - Where?

- Shaders!
  - Vertex and Pixel
- Code like this can generate a +Inf:

```
//grab half angle vector
float3 vec = HalfAngleVec.xyz;

//compute length
float vecLen = length(vec);

//normalize (could divide by zero!)
vec /= vecLen;
```

FP Specials – Where?

- Some common shader operations:

(0.0 * Inf)	NaN
(+Inf + -Inf)	NaN
(NaN + anything)	NaN
rsq(0.0)	+Inf
log2(-1.0)	NaN
(NaN == NaN)	false

FP Specials – Where?

- **Texturing can also generate and propagate specials**
  - **Not usually a big issue**
  - **And finite inputs mean finite outputs**

FP Specials – Where?

- ROP access can also generate specials
- Especially important for overflow
- Remember, fp16 overflows at a meager 65504
  - Write out a value greater than that, and you get +Inf in the fb!
  - Do additive fp blending (ONE:ONE), overflowing result means +Inf!

FP Specials - How can you tell?

- In general, specials show up as follows:
  - +Inf – white pixel
  - -Inf – black pixel
  - NaN – black pixel
- Convolution / blurring has a tendency to propagate this over the whole screen
  - Write out a single +Inf due to overflow and your whole screen can be hosed

FP Specials – How can you tell?

- There are also HLSL functions to help you out
  - `isnan()`
  - `isinf()`
  - `isfinite()`

- But there can be driver issues

- These should be used for debugging only

FP Specials – What can you do?

- Key to solving these issues is dealing with them proactively

FP Specials – What can you do?

- **Change previous example to:**

```
//grab half angle vector
float3 vec = HalfAngleVec.xyz;

//compute length
float vecLen = length(vec);

//safely normalize
if (vecLen != 0.0f)
   vec /= vecLen;
```

FP Specials – What can you do?

- **Ditto for overflow specials:**

```
//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
return float4(color, worldSpacePos.z);
```

Questions?

- cem@nvidia.com