GPU TECHNOLOGY CONFERENCE

GPU-DRIVEN LARGE SCENE RENDERING NV_COMMAND_LIST

Pierre Boudier, Quadro Software Architect

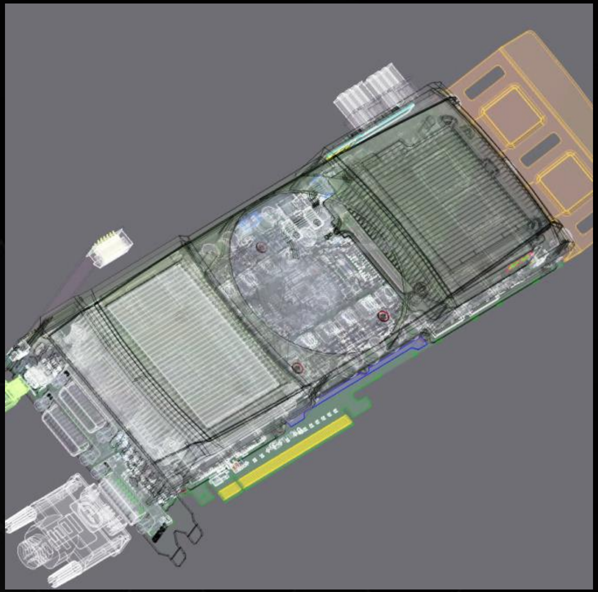
Christoph Kubisch, Developer Technology Engineer

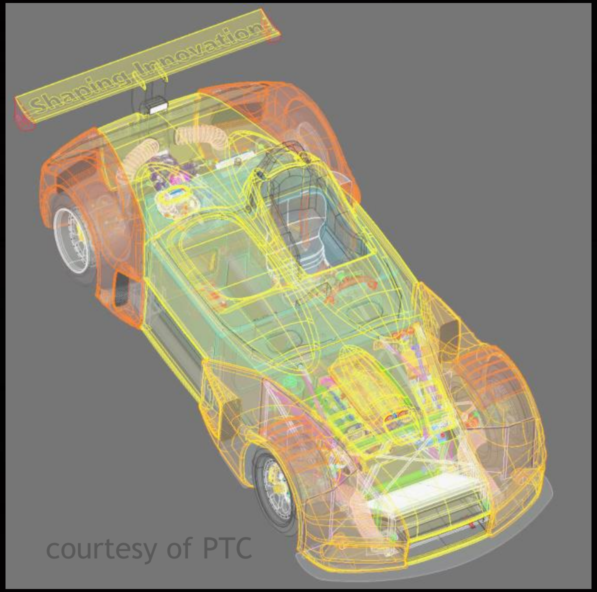
MOTIVATION

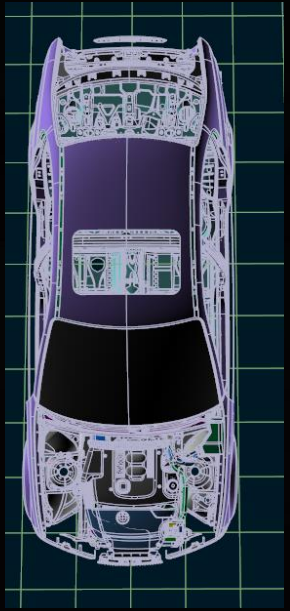
- Modern GPUs have a lot of execution units to make use of
  - Quadro 4000: 256 cores
  - Quadro K4000: 768 cores
  - Quadro K4200: 1344 cores
  - Quadro M6000: 3072 cores
- How to leverage all this power?
  - Efficient API usage and rendering algorithms
  - APIs reflecting recent hardware designs and capabilities

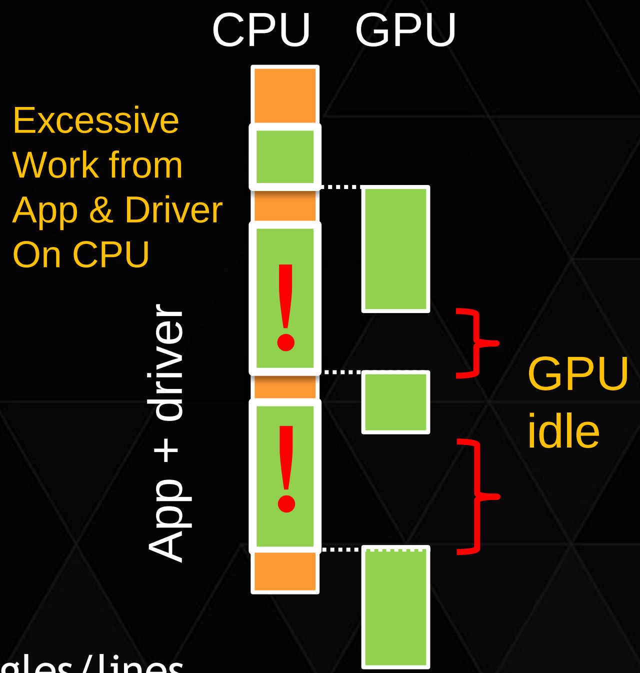
CHALLENGE OF ISSUING COMMANDS

- Issuing drawcalls and state changes can be a real bottleneck









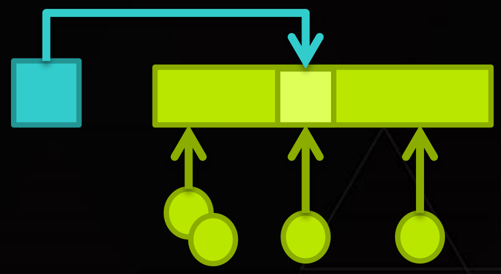
- 650,000 Triangles
- 68,000 Parts
- ~ 10 Triangles per part

- 3,700,000 Triangles
- 98,000 Parts
- ~ 37 Triangles per part

- 14,338,275 Triangles/lines
- 300,528 drawcalls (parts)
- ~ 48 Triangles per part

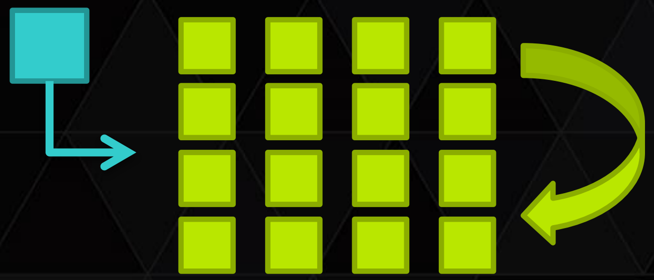
ENABLING GPU SCALABILITY

- Avoid data redundancy
  - Data stored once, referenced multiple times
  - Update only once (less host to gpu transfers)



- Increase GPU workload per job
  - Further cuts API calls
  - Less CPU work

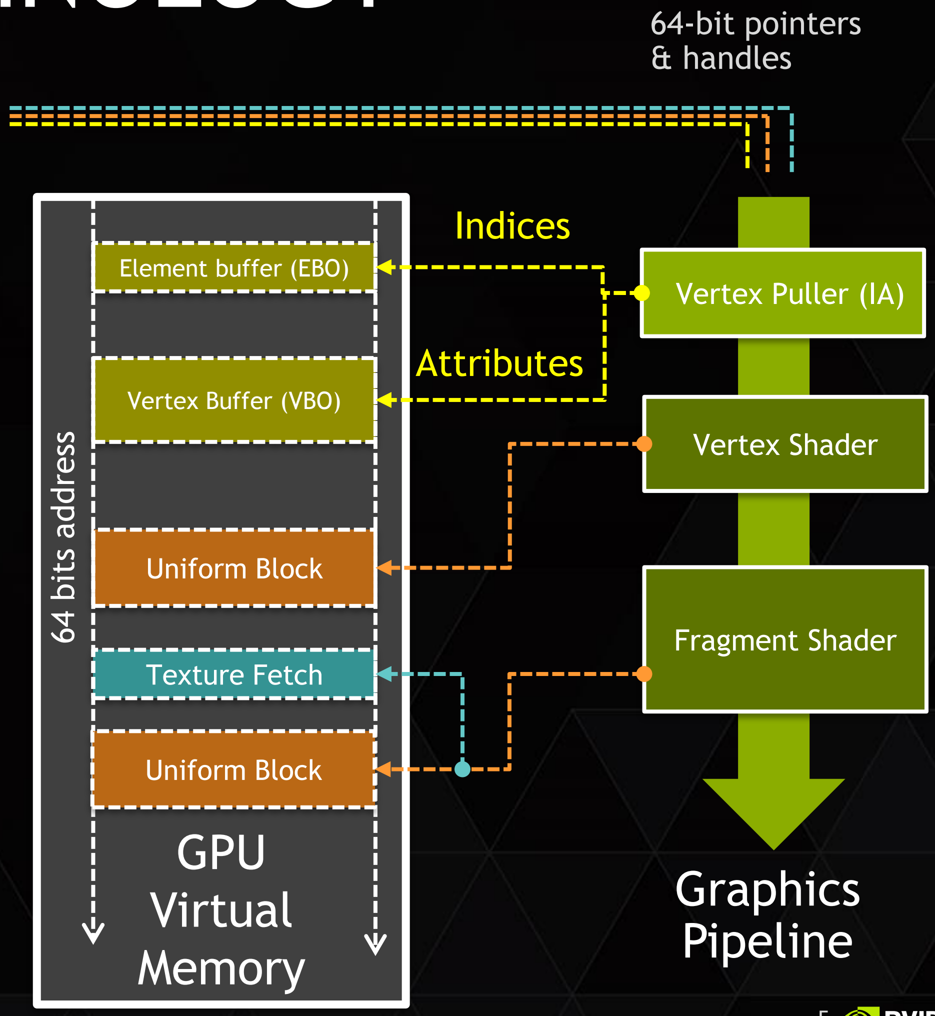
- Minimize CPU/GPU interaction
  - Allow GPU to update its own data
  - Low API usage when scene is changed little
  - E.g. GPU-based culling, matrix updates...



http://on-demand.gputechconf.com/gtc/2014/presentations/S4379-opengl-44-scene-rendering-techniques.pdf

BINDLESS TECHNOLOGY

- What is it about?
  - Work from native GPU pointers/handles
  - Less validation, less CPU cache thrashing
  - GPU can use flexible data structures
- Bindless Buffers
  - Vertex & Global memory since pre-Fermi
- Bindless Constants (UBO)
  - Support for Fermi and above
- Bindless Textures
  - Since Kepler



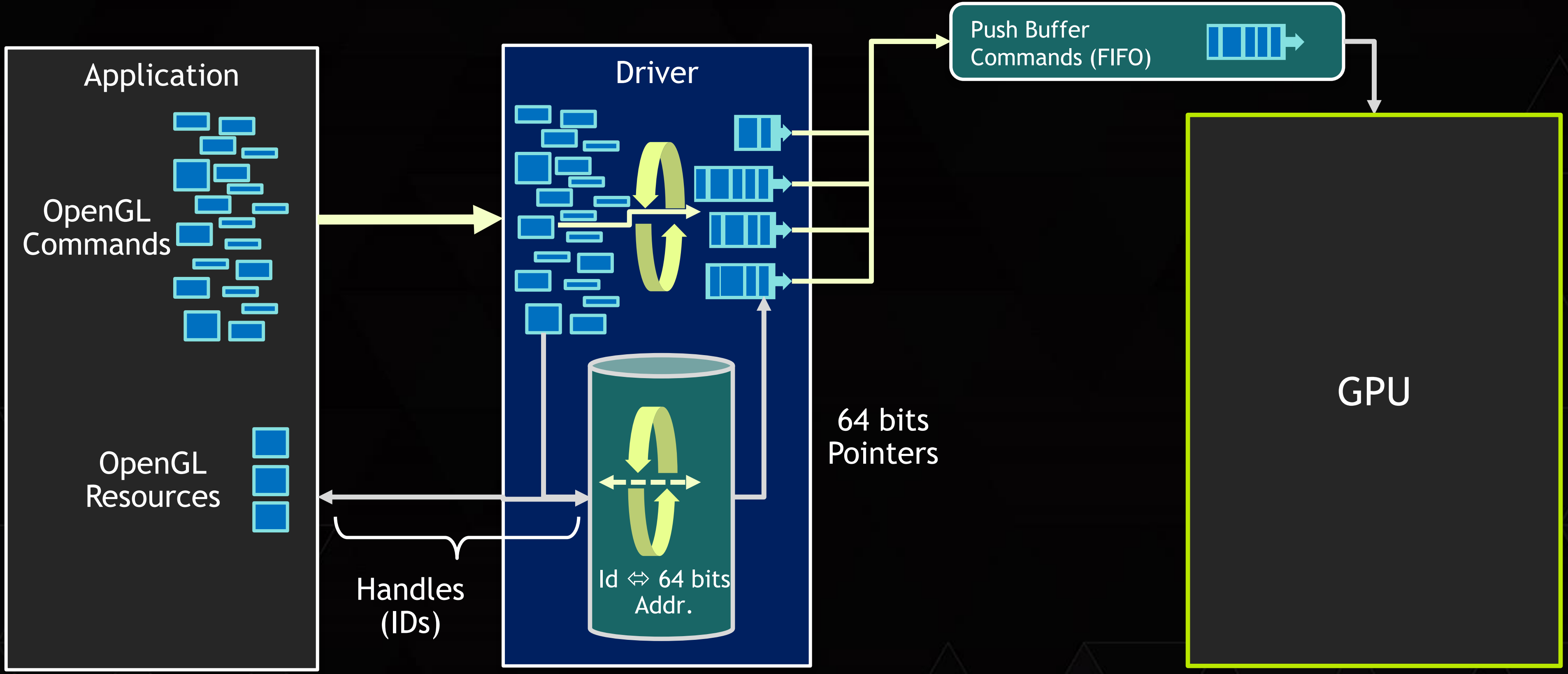
BINDLESS DRAWING LOOP

```
UpdateBuffers();
glBufferAddressRangeNV(UNIFORM..., 0, addrView, ...);

// redundancy filters not shown
foreach (obj in scene) {
    glBufferAddressRangeNV(VERTEX..,   0, obj.geometry->addrVBO, ...);
    glBufferAddressRangeNV(ELEMENT..., 0, obj.geometry->addrIBO, ...);

    glBufferAddressRangeNV(UNIFORM..., 1, addrMatrices + obj.mtxOffset, ...);

    // iterate over cached material groups
    foreach ( batch in obj.materialGroups) {
      glBufferAddressRangeNV(UNIFORM, 2, addrMaterials + batch.mtlOffset, ...);

      glMultiDrawElements (...);
    }
}
```

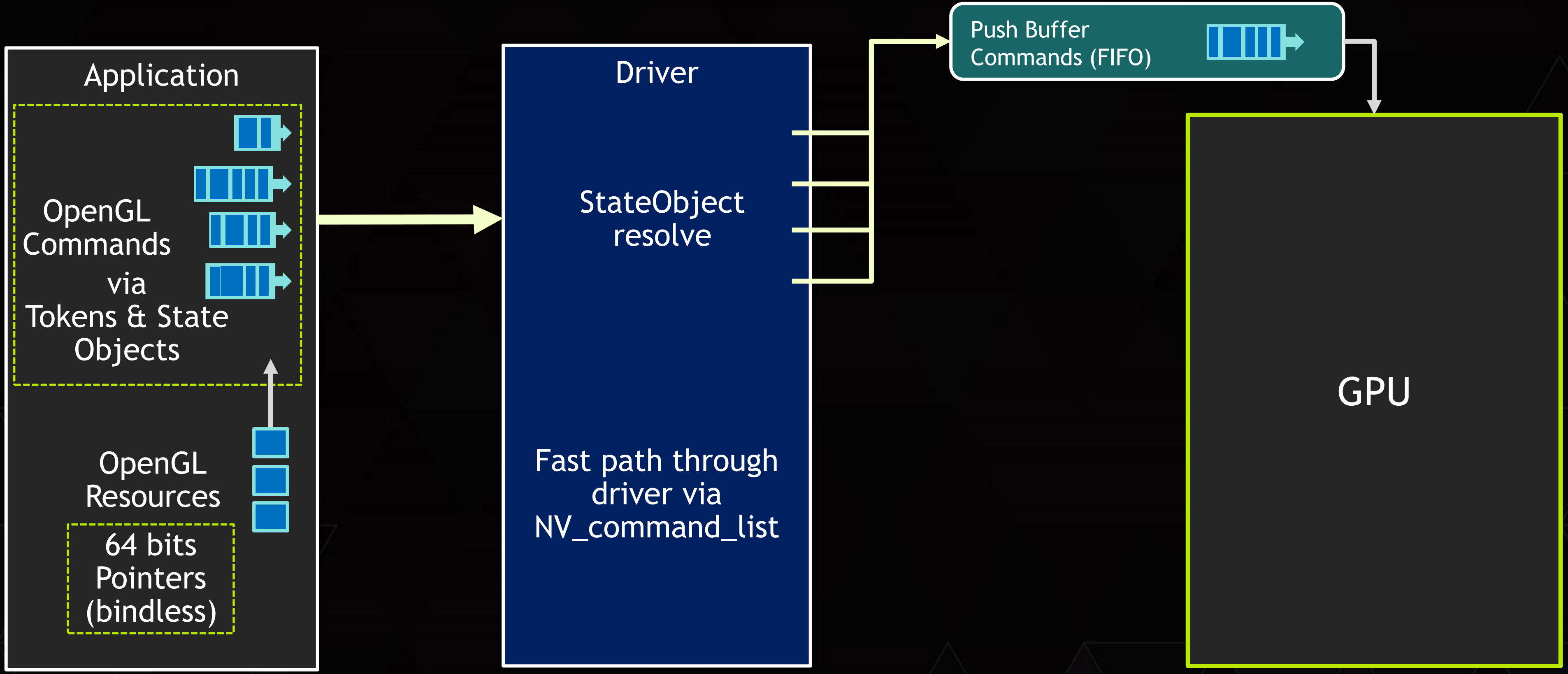
NV_COMMAND_LIST – KEY CONCEPTS

- Tokenized Rendering (GPU modifiable command buffers):
  - Simple state changes and draw commands are encoded into binary data stream
  - Leverages bindless resources
- State Objects (pre-validated)
  - Macro state (program, blending, fbo-config...) is captured into an object
  - Control over when costly validation happens, later reuse of objects is very fast
- Compiled Command List (alternative to token buffer)
  - Display list like usage, however buffer addresses are referenced, therefore their content (matrices, vertices...) can still be modified.

COMMAND PIPELINE



COMMAND PIPELINE



TOKENIZED RENDERING

```
// bindless scene drawing loop
foreach (obj in scene) {
      glBufferAddressRangeNV(VERTEX..,   0, obj.geometry->addrVBO, ...);
      glBufferAddressRangeNV(ELEMENT..., 0, obj.geometry->addrIBO, ...);
      glBufferAddressRangeNV(UNIFORM..., 1, addrMatrices + obj.mtxOffset, ...);
      foreach ( batch in obj.materialCaches) {
        glBufferAddressRangeNV(UNIFORM, 2, addrMaterials + batch.mtlOffset, ...);
        glMultiDrawElements(...)
      }
}
```

All these commands (hundreds of thousands) for the entire scene can be replaced by a single call to API!

```
glDrawCommandsNV (TRIANGLES,
   tokenBuffer, offsets[], sizes[], count);
             //         {0}, {tokensSize}, 1
```

TOKENIZED RENDERING

- Tokens are tightly packed structs in linear memory

```
*CommandNV  {
     GLuint     header; // glGetCommandHeaderNV(type,…)
     ... command specific payload
};
```

```
ELEMENT_ADDRESS_COMMAND_NV
ATTRIBUTE_ADDRESS_COMMAND_NV
UNIFORM_ADDRESS_COMMAND_NV
```

```
TERMINATE_SEQUENCE_COMMAND_NV
NOP_COMMAND_NV
```

DRAW tokens allow mixing strips, lists, fans, loops of same base mode (TRIANGLES, LINES, POINTS) in single dispatch

```
DRAW_ELEMENTS_COMMAND_NV
DRAW_ARRAYS_COMMAND_NV
DRAW_ELEMENTS_STRIP_COMMAND_NV
DRAW_ARRAYS_STRIP_COMMAND_NV

DRAW_ELEMENTS_INSTANCED_COMMAND_NV
DRAW_ARRAYS_INSTANCED_COMMAND_NV
```

```
BLEND_COLOR_COMMAND_NV
STENCIL_REF_COMMAND_NV
LINE_WIDTH_COMMAND_NV
POLYGON_OFFSET_COMMAND_NV
ALPHA_REF_COMMAND_NV
VIEWPORT_COMMAND_NV
SCISSOR_COMMAND_NV
FRONTFACE_COMMAND_NV
```

TOKENIZED RENDERING

```
// single drawcall, tokens encoded into raw memory buffer!
glDrawCommandsNV (..., tokenBuffer, offsets[], sizes[],      count);
                                 //        {0}, {bufferSize}, 1
```

VBO	EBO	UBO Matrix	UBO Material	Draw	UBO Material	Draw	Draw		

```
AttributeAddressCommandNV
{
  GLuint    header;

  GLuint    index;
  GLuint64  address;
}
```

```
ElementAddressCommandNV
{
  GLuint    header;

  GLuint64  address;
  GLuint    typeSizeInByte;
}
```

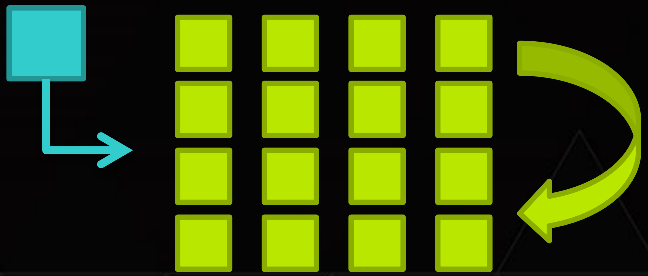
```
UniformAddressCommandNV
{
  GLuint     header;

  GLushort   index;
  GLushort   stage;
   // glGetStageIndexNV(VERTEX..)
  GLuint64   address;
}
```

```
DrawElementsCommandNV
{
  Gluint    header;

  GLuint    count;
  GLuint    firstIndex;
  GLuint    baseVertex;
}
```

GPU TECHNOLOGY CONFERENCE

TOKENIZED RENDERING

- What is so great about it?
  - It‘s crazy fast (see later) and tokens are popular in render engines already
  - The tokenbuffer is a „regular“ GL buffer
    - Can be manipulated by all mechanisms OpenGL offers
    - Can be filled from different CPU threads (which do not require a GL context)
  - Expands the possibilities of GPU driving its own work without CPU roundtrip



GPU TECHNOLOGY CONFERENCE

STATE OBJECTS

- StateObject
  - Encapsulates majority of state (fbo format, active shader, blend, depth ...), but no bindings! (use bindless textures passed via UBO...)
    - `glCaptureStateNV ( stateobject, GL_TRIANGLES );`
  - Less rendertime variability, explicit control over validation time
- Render entire scenes with different shaders/fbos... in one go
  - Driver caches state transitions

```
// single drawcall, multiple shaders, fbos...
glDrawCommandsStatesNV (tokenBuffer, offsets[], sizes[], states[], fbos[], count);
```

GPU TECHNOLOGY CONFERENCE

STATE OBJECTS

```
// single drawcall, multiple shaders, fbos...
glDrawCommandsStatesNV (tokenBuffer, offsets[], sizes[], states[], fbos[], count);

for i < count {
  if (i == 0) set state from states[i];
  else        set state transition states[i-1] to states[i]

  if (fbo[i]) glBindFramebuffer( fbo[i] ) // must be compatible to states[i].fbo
  else        glBindFramebuffer( states[i].fbo )

  ProcessCommandSequence(... tokenBuffer, offsets[i], sizes[i])
}
```

- Can reuse tokens & state with different fbos (e.g. shadow passes)
- Compatibilty depends on fbo‘s drawbuffers, texture formats… but not sizes

STATE OBJECTS

```
// single drawcall, multiple shaders, fbos...
glDrawCommandsStatesNV (tokenBuffer, offsets[], sizes[], states[], fbos[], count);
                      // {0,sizeA}, {sizeA, sizeB}, {A,B},      {f,f},     2
```

tokenBuffer: VBO | IBO | Matrix UBO | Material UBO | Draw | Material UBO | Draw | Draw ¦ Draw | Draw | Draw

Sequence A (e.g. triangles) ¦ Sequence B (lines)

[0] State Object A | FBO f | VBO | IBO | Matrix UBO | Material UBO | Draw | Material UBO | Draw | Draw

[1] State Object B | FBO f | Draw | Draw | Draw

- Within glDrawCommandsStatesNV state set by tokens is inherited across sequences

COMPILED COMMAND LIST

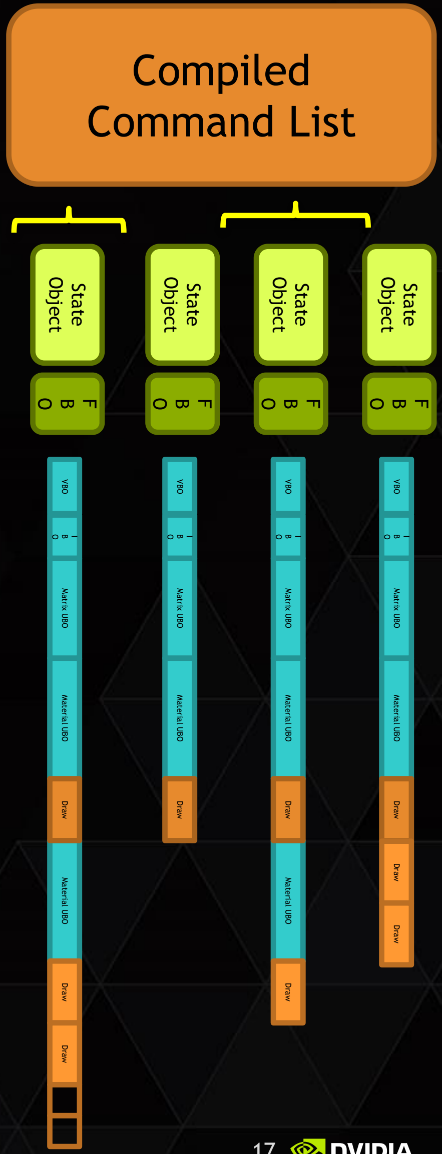
- Combine multiple segments into CommandList object
  - Tokens provided by system memory

```
glListDrawCommandsStatesClientNV( list, segment,
   void* tokencmds[], sizes[], states[], fbos[], count);
```

- Less flexibilty compared to token buffer
  - Token content, state and fbo assignments are deep-copied
  - List is immutable, needs recompile if pointers/state changes

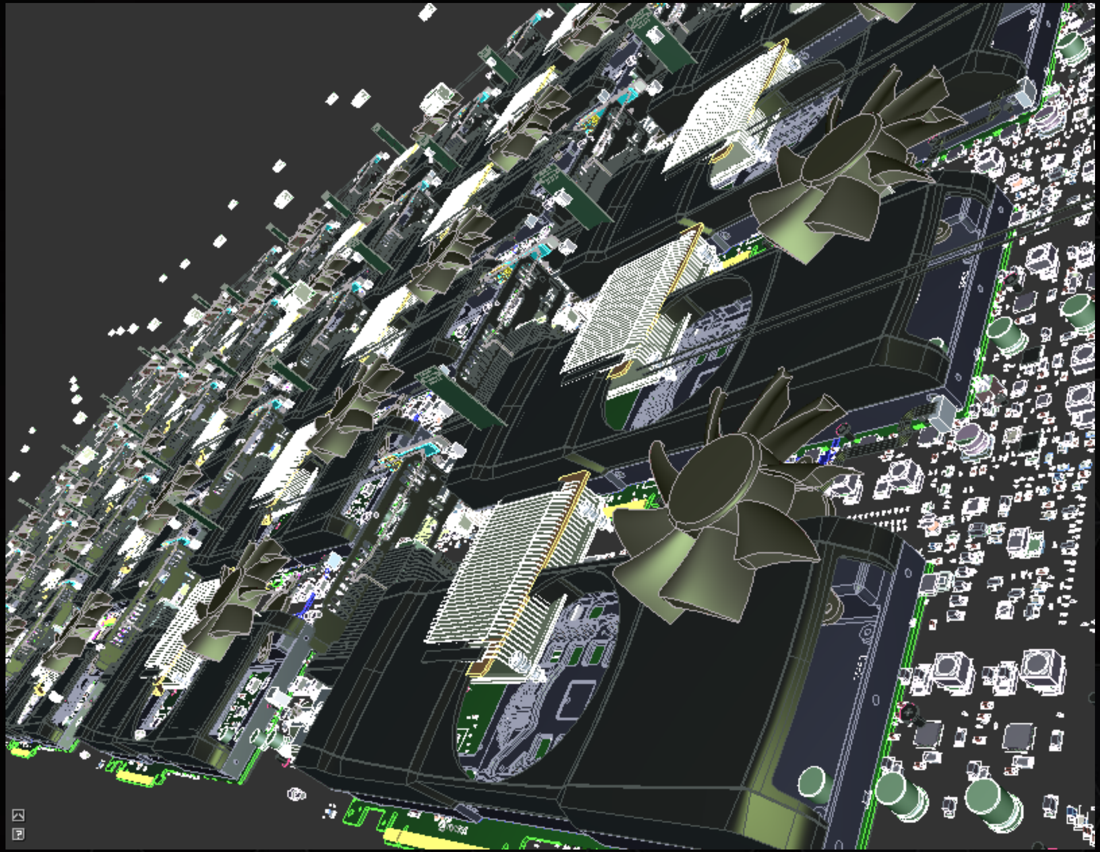
```
glCompileCommandListNV( list );
```

- Allows even faster state transitions
  - All key data is known to the driver



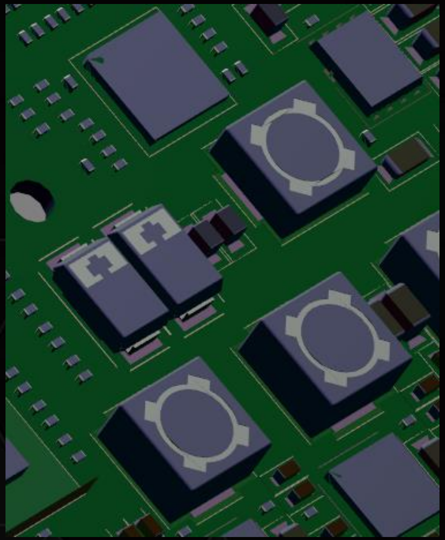
RESULTS

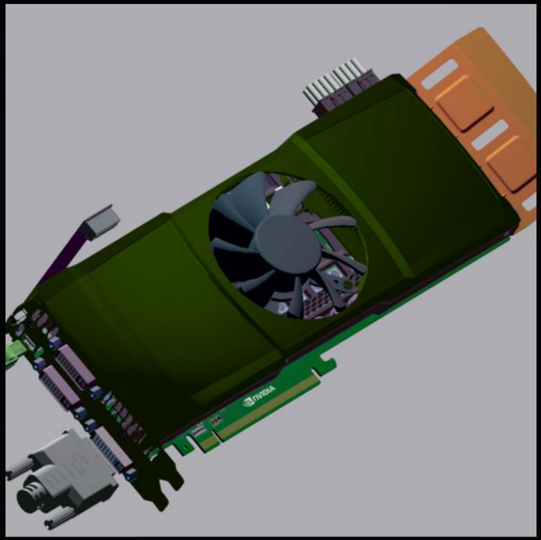
- High scene complexity
  - No instancing used, true copies
  - Each object unique and editable
- 90 000 objects
  - Each drawn with triangles & lines
  - Raw: 4.8m drawcalls
    - Standard GL: 2 fps
    - Commandlist: 20 fps



RENDERING RESEARCH FRAMEWORK

- Render test with „Graphicscard“ model
  - Many low-complexity drawcalls (CPU challenged)

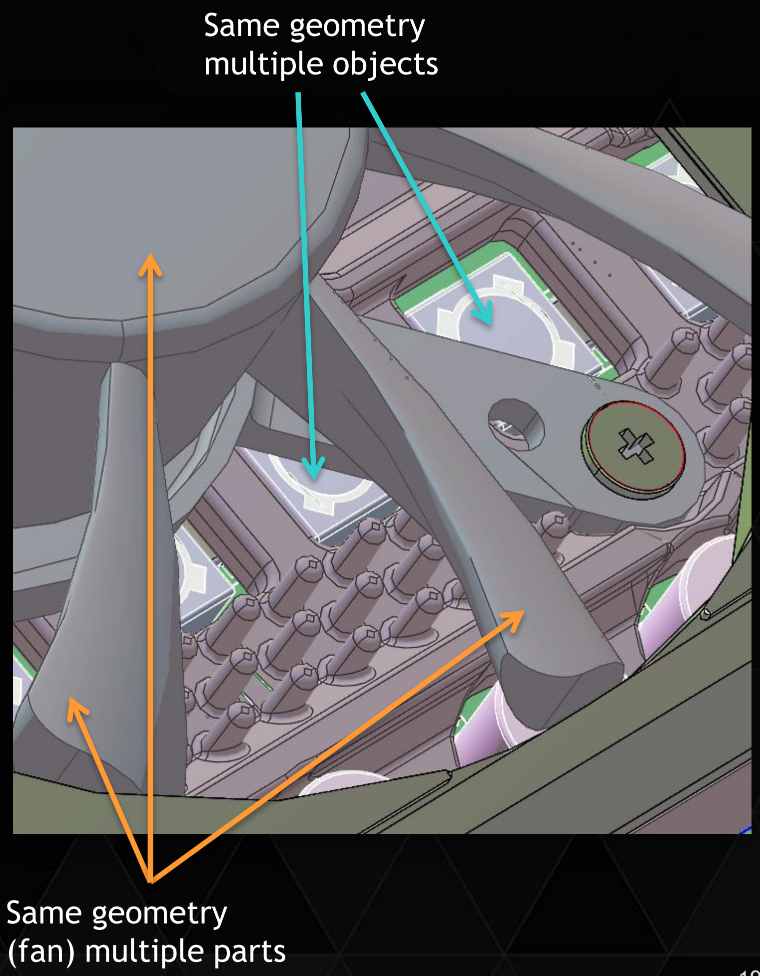




110 geometries, 66 materials

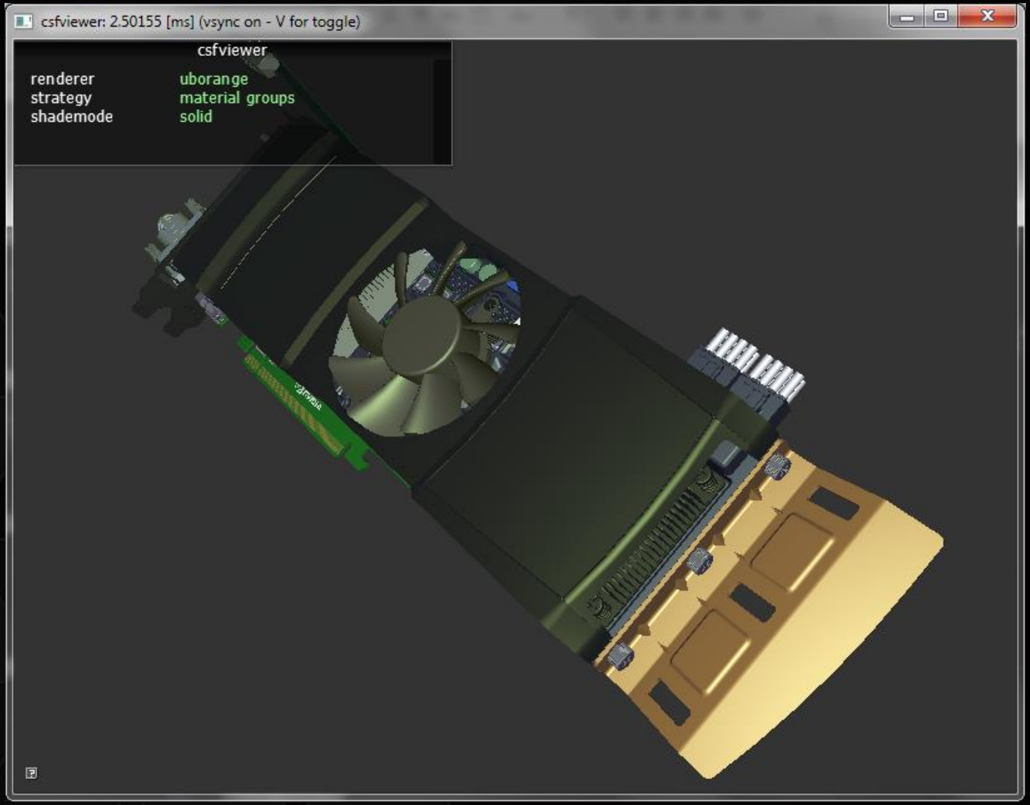
68 000 parts

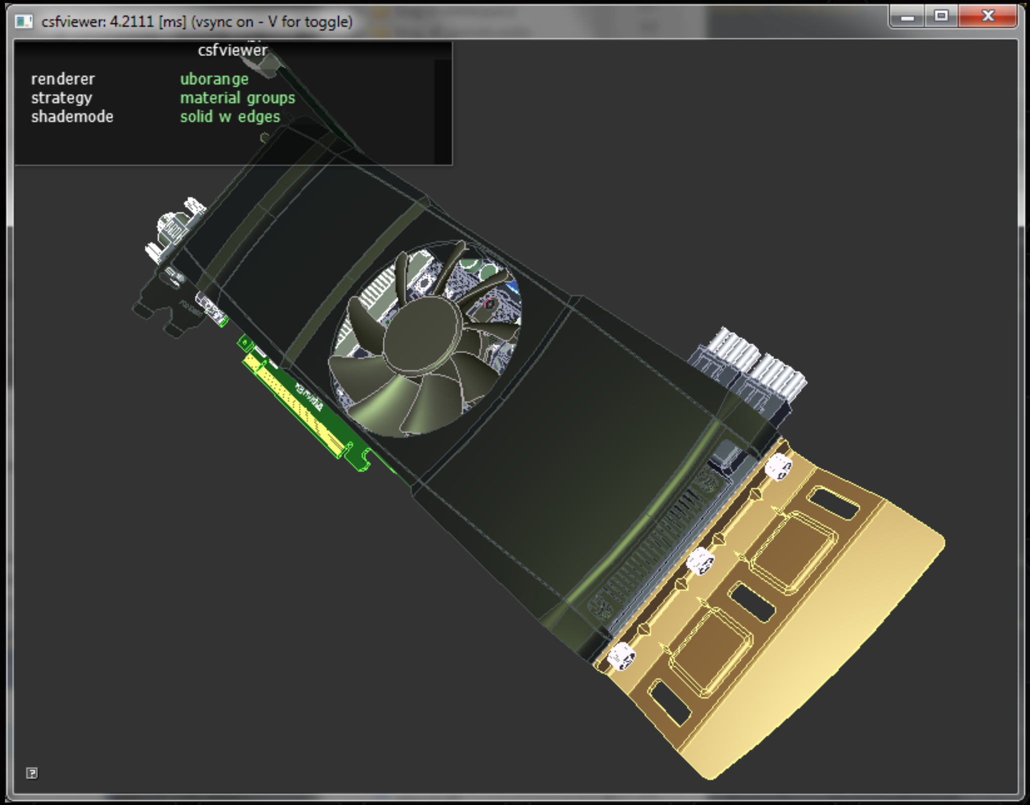
2500 objects



SCENE STYLES

- „Shaded“ and „Shaded & Edges“





GPU TECHNOLOGY CONFERENCE

SCENE DRAWING (GROUPED)

```
UpdateBuffers();
glBindBufferBase (UBO, 0, uboView);

foreach (obj in scene) {
    // redundancy filter for these (if (used != last)...
    glBindVertexBuffer (0, obj.geometry->vbo, 0, vtxSize);
    glBindBuffer (ELEMENT, obj.geometry->ibo);
    glBindBufferRange (UBO, 1, uboMatrices, obj.matrixOffset, maSize);

    // iterate over cached material groups
    foreach ( batch in obj.materialGroups) {
      glBindBufferRange (UBO, 2, uboMaterial, batch.materialOffset, mtlSize);


      glMultiDrawElements (...);
    }
}
```

~ 2 500 api drawcalls
~11 000 drawcalls
~55 triangles per call

SCENE DRAWING (INDIVIDUAL)

```
UpdateBuffers();
glBindBufferBase (UBO, 0, uboView);

foreach (obj in scene) {
    ...




    // iterate over all parts individually
    foreach ( part in obj.parts) {
      if (part.material != lastMaterial){
        glBindBufferRange (UBO, 2, uboMaterial, part.materialOffset, mtlSize);
      }
      glDrawElements (...);
    }
}
```

~68 000 drawcalls
~10 triangles per call

PERFORMANCE SHADED

- Render all objects as triangles
  - GROUPED: ~ 300 KB (22 k tokens, ~11k buffer related, 11 k for drawing)
  - INDIVIDUAL: ~ 1 MB (79 k tokens, ~68 k for drawing)

Technique	Draw time 11k draws		Draw time 68k draws	
Timer	GPU	CPU	GPU	CPU
Core OpenGL	0.4	1.4	3.1	6.7
NV bindless	0.3	0.7 2 x	1.9	3.8 1.7 x
TOKEN buffer	0.3	~0 BIG x	0.9	~0 BIG x

Preliminary results M6000

PERFORMANCE SHADED

- Removing buffer redundancy filtering

	Material UBO	Draw	Material UBO	Draw	Material UBO	

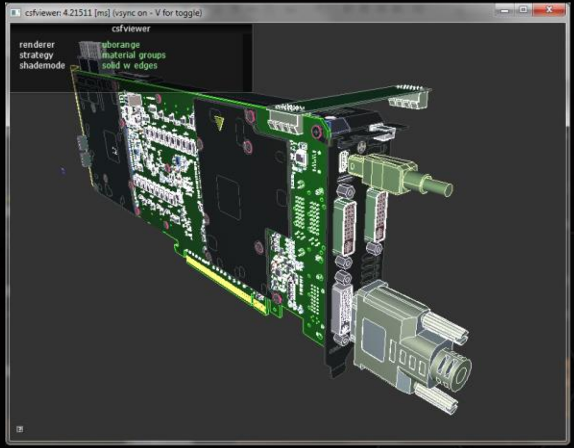
  - adds 60 k UBO, and 3.4k EBO & VBO tokens; total 144 k tokens

Technique	**Draw time 68k draws**	
Timer	GPU	CPU
Unfiltered Core OpenGL	5.6	11.1
Core OpenGL	3.1 1.8 x	6.7 1.6 x
Unfiltered TOKEN buffer	1.9 2.9 x	~0 BIG x

Preliminary results M6000

PERFORMANCE SHADED & EDGES

- For each object render triangles then lines
- Frequent alternation between two state objects (TRIANGLES/LINES) (~5000 times)
  - GROUPED: 540 KB (~ 40k tokens)
  - INDIVIDUAL: 2 MB (~ 160k tokens)



Technique	Draw time 11k*2 draws		Draw time 68k*2 draws	
Timer	GPU	CPU	GPU	CPU
Core OpenGL	0.8	2.4	11.5	14.3
NV bindless	0.8	1.4 1.7 x	6.5	8.0 1.7 x
TOKEN buffer	0.8	0.4 6 x	2.1	0.4 35 x

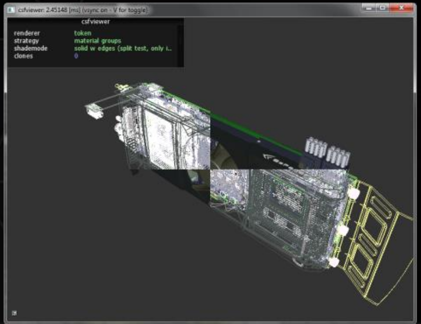
Preliminary results M6000

EXAMPLE USE CASES

- 5 000 shader changes: toggling between two shaders in „shaded & edges“

Timer	GPU	CPU
NV bindless	12.3	15.1
TOKEN buffer	1.6 7.7 x	0.4 37 x
Compiled TOKEN list	1.5 8.2 x	0.005 BIG x

- 5 000 fbo changes: similar as above but with fbo toggle instead of shader
  - Almost no additional cost compared to rendering without fbo changes

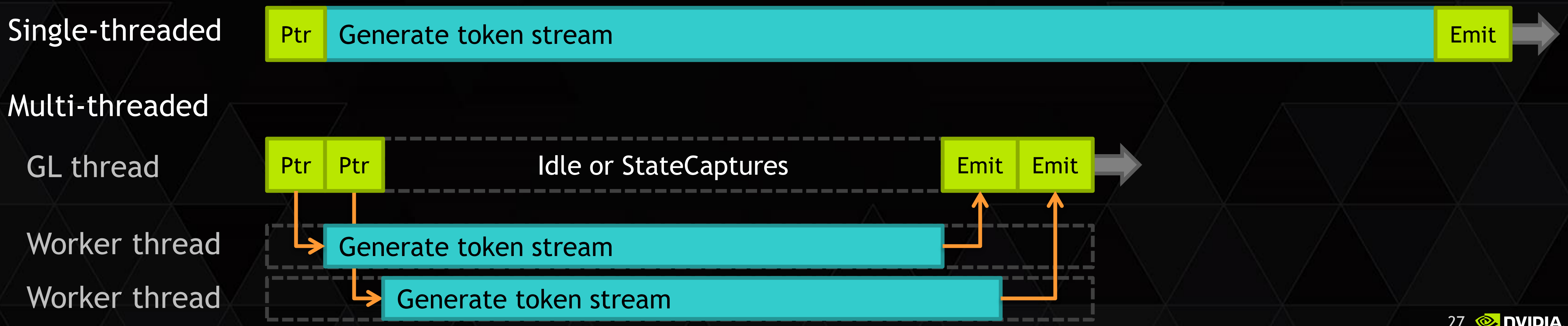


Timer	GPU	CPU
NV bindless	57.0	59.0
TOKEN buffer	1.1 51 x	0.9 65 x
Compiled TOKEN list	0.8 71 x	0.022 BIG x

Preliminary results on M6000

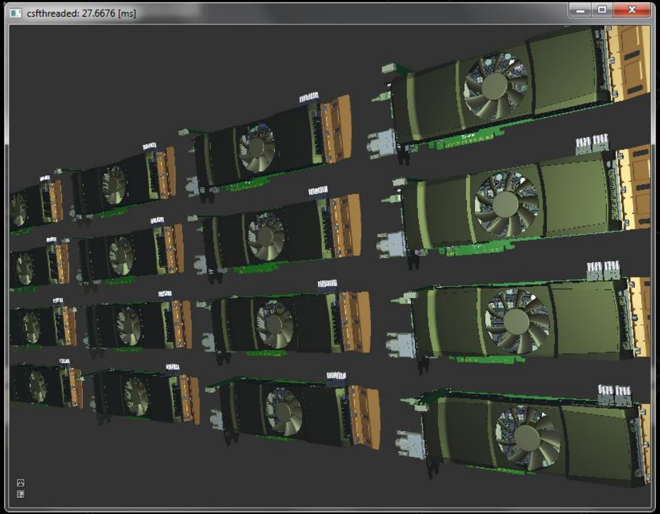
TOKEN STREAMING

- In case token buffer cannot be reused, fill tokens every frame
  - Fill & emit from a single thread or multiple threads
    - Pass command buffer pointers to worker threads, that do not require GL contexts
    - Handle state objects in GL thread, or pass what is required to generate between threads (GL thread captures state, while worker fills command buffer)



TOKEN STREAMING

- Rendering the model 16 times (176k draws)
  - StateObjects are reused, tokens regenerated and submitted in chunks (~22 per frame)
  - Framework is „too simple“ in terms of per-thread work to show greater scaling



Technique	Draw time 11k*16 draws	
Timer	GPU	CPU
Core OpenGL (1 thread)	22	27
TOKEN 1 worker thread	4.3	3.5 7.7 x
TOKEN 2 worker threads	4.3	2.1 12 x
TOKEN 3 worker threads	4.3	1.7 15 x

Preliminary results M6000

MIGRATION STEPS

- All rendering via FBO (statecapture doesn‘t support default backbuffer)
- No legacy state use in GLSL
  - Shader driven pipeline, use generic glVertexAttribPointer (not glTexCoordPointer and so on), use custom uniforms no gl_ModelView...
- No classic uniforms, all in UBO
  - ARB_bindless_texture for texturing
- Bindless Buffers
  - ARB_vertex_attrib_binding combined with NV_vertex_buffer_unified_memory
- Organize for StateObject reuse
  - Can no longer just „glEnable(GL_BLEND)“, avoid many state captures per frame

GPU TECHNOLOGY CONFERENCE

MIGRATION TIPS

- Vertex Attributes and bindless VBO
  - http://on-demand.gputechconf.com/siggraph/2014/presentation/SG4117-OpenGL-Scene-Rendering-Techniques.pdf (slide 11-16)
- GLSL

```
// classic attributes





...

  normal      = gl_Normal;
  gl_Position = gl_Vertex;
```

```
// generic attributes
// ideally share this definition across C and GLSL

#define VERTEX_POS    0
#define VERTEX_NORMAL 1

in layout(location= VERTEX_POS)    vec4 attr_Pos;
in layout(location= VERTEX_NORMAL) vec3 attr_Normal;
...

  normal      = attr_Normal;
  gl_Position = attr_Pos;
```

MIGRATION TIPS

- UBO Parameter management
  - http://on-demand.gputechconf.com/siggraph/2014/presentation/SG4117-OpenGL-Scene-Rendering-Techniques.pdf (18-27, 44-47)
  - Ideally group by frequency of change

```
// classic uniforms



uniform samplerCube viewEnvTex;




uniform vec4 materialColor;
uniform sampler2D materialTex;


...
```

```
// UBO usage, bindless texture inside UBO, grouped by change
layout(commandBindableNV) uniform;
layout(std140, binding=0) uniform view
{
  samplerCube viewEnvTex;
};

layout(std140, binding=1) uniform material {
  vec4 materialColor;
  sampler2D texMaterialColor;
};

...
```

MIGRATION TIPS

- StateObject
  - Sample provides „statesystem.cpp/hpp“ that showcases most of the commonly used state being captured, also useful for emulation.
  - Does not capture what can be modified by tokens (e.g. Viewport & Scissor)

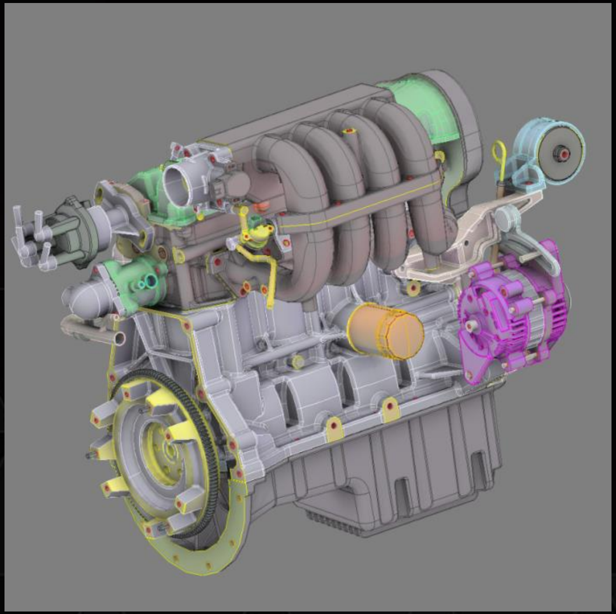
```
State
{
    EnableState              enable;         PrimitiveState           primitive;
    EnableDeprecatedState    enableDepr;     SampleState              sample;
    ProgramState             program;        RasterState              raster;
    ClipDistanceState        clip;           RasterDeprecatedState    rasterDepr;
    AlphaState               alpha;          DepthRangeState          depthrange;
    BlendState               blend;          MaskState                mask;
    DepthState               depth;          FBOState                 fbo;
    StencilState             stencil;        VertexState              vertex;
    LogicState               logic;          VertexImmediateState     verteximm;
}
```

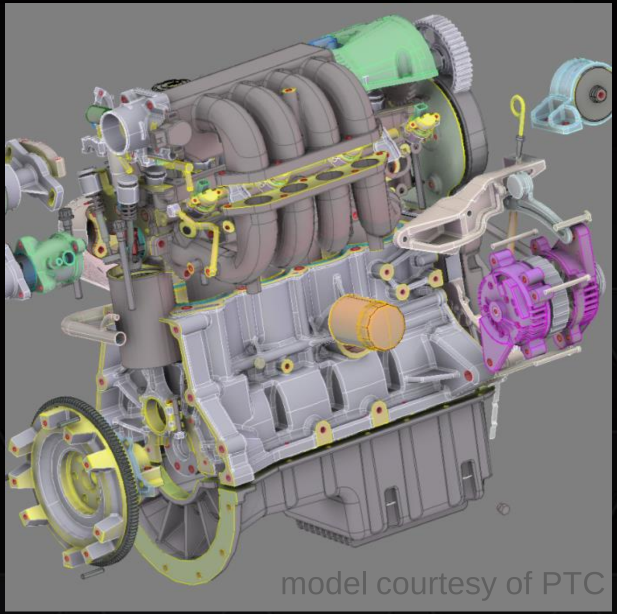
LET GPU DO MORE WORK

- When data is only referenced, we can:
  - Still change vertices, materials, matrices... from CPU
  - Perform updates based on additional knowledge on GPU
    - Object data (matrices, materials animation)
    - Geoemtry data (deformation, skinning, morphing...)
    - Occlusion Culling
    - Level of Detail

TRANSFORM TREE UPDATES

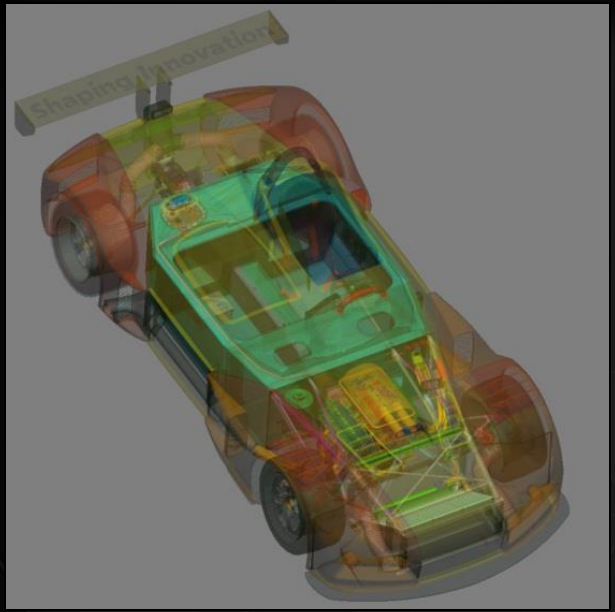
- All matrices stored on GPU
  - Use ARB_compute_shader for hierarchy updates, send only local matrix changes, evaluate tree
  - http://on-demand.gputechconf.com/siggraph/2014/presentation/SG4117-OpenGL-Scene-Rendering-Techniques.pdf (29-30)





OCCLUSION CULLING

- Try create less total workload
- Many occluded parts in the car model (lots of vertices)



GPU CULLING BASICS

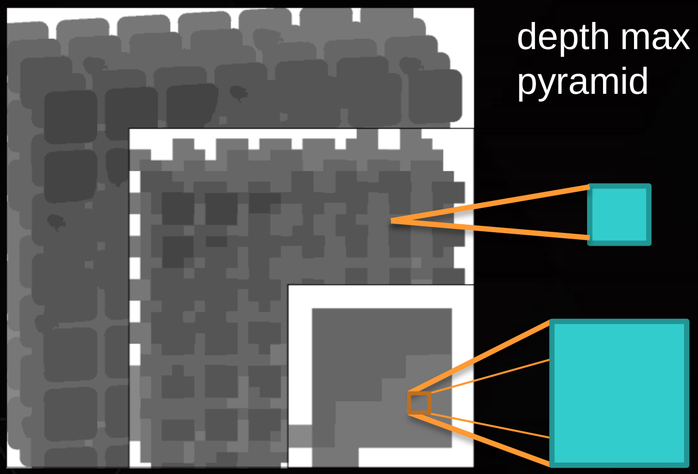
- GPU friendly processing
  - Matrix, bbox and object (matrixIdx + bboxIdx) buffers
  - More efficient than occ. queries, as we test many objects at once
- Results
  - Readback: GPU to Host
    - GPU can pack bit stream
  - Indirect: GPU to GPU
    - E.g. DrawIndirect‘s instanceCount to 0 or 1

```
0,1,0,1,1,1,0,0,0
```

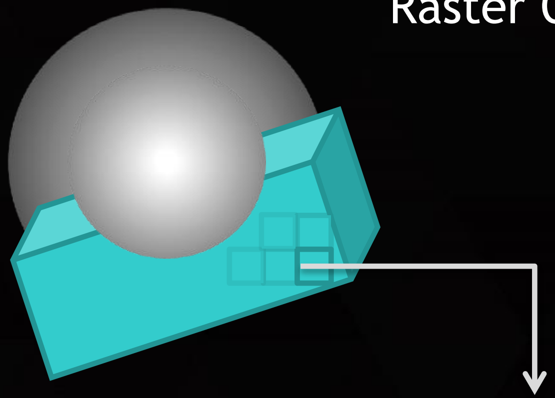
```
buffer cmdBuffer{
  Command cmds[];
};
...
cmds[obj].instanceCount = visible;
```

OCCLUSION CULLING

HiZ occlusion



Raster Occlusion




```
// rendered without depth or color writes

// GLSL fragment shader
// from ARB_shader_image_load_store
layout(early_fragment_tests) in;
...

void main()
{
  visibility[objectID] = 1;
  // could use atomicAdd for coverage
}
```

- Raster gives more accurate results
- Both benefit from temporal coherence usage to avoid a dedicated depth pass

http://on-demand.gputechconf.com/siggraph/2014/presentation/SG4117-OpenGL-Scene-Rendering-Techniques.pdf (slide 49 - 54)

RESULTS VIA READBACK

- Use dedicated buffers for readback
  - One for GPU processing only (ensures best memory type used)
  - N for readbacks (for example 4 to avoid sync points)
    - `glCopyNamedBufferSubData (gpuresult, readbacks[ frame % N ]...)`
    - Readback could be mapped persistently via GL_ARB_buffer_storage
- Ideally delay access of readback for a few frames
  - Avoids need for synchronization, but can introduce visible artefacts
  - Readback older frames to give CPU additional knowledge, but use GPU indirect methods for rendering

RESULTS VIA COMMANDLIST

- Commandlist culling needs several buffers
  - Token commandstream (input & output): variable size
  - Token attributes (input & output): size, offset, object ID
    - Can use negative objectID to encode tokens that must always be added
- Algorithm:
  - First compute output sizes using object ID and visbility
    - output.sizes [token] = visible [objectID] ? input.sizes[token] : 0
  - Run a scan operation to compute output offsets
  - Build output tokenstream

RESULTS VIA COMMANDLIST

- Multiple squences may be stored in the tokenstream (different stateobjects..)

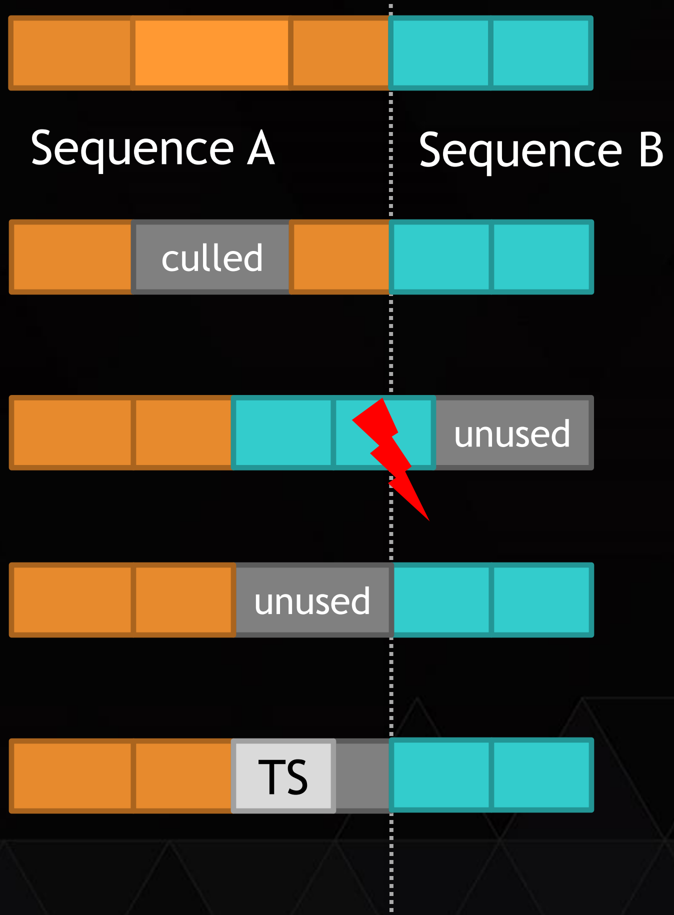
Original token stream

Find out which tokens to cull

Token offsets from scan are global

Correct output offset based on sequence‘s start offset

Insert terminate sequence when: last token‘s offset != original offset



The sequence separation is provided by CPU, which we can‘t alter

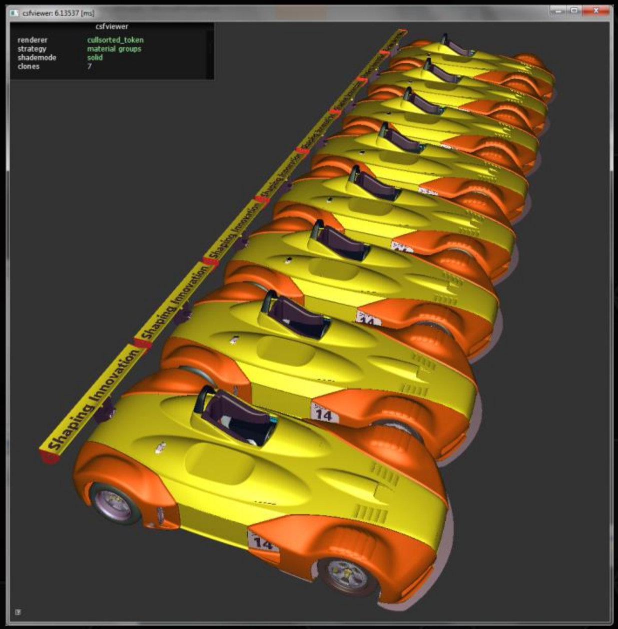
RESULTS

- Now overcomes deficit of previous methods

http://on-demand.gputechconf.com/gtc/2014/presentations/S4379-opengl-44-scene-rendering-techniques.pdf

Technique	Draw time 11k draws	
Timer	GPU	CPU
glBindBufferRange	6.2	3.4
TOKEN native	6.2	~0 BIG x
CULL Old Readback (stalls pipe)*	3.1 2 x	3.1 1.1 x
CULL Old Bindless MDI*	3.7 1.6 x	0.7 4.8 x
CULL NEW TOKEN buffer	2.5 2.5 x	0.2 17 x

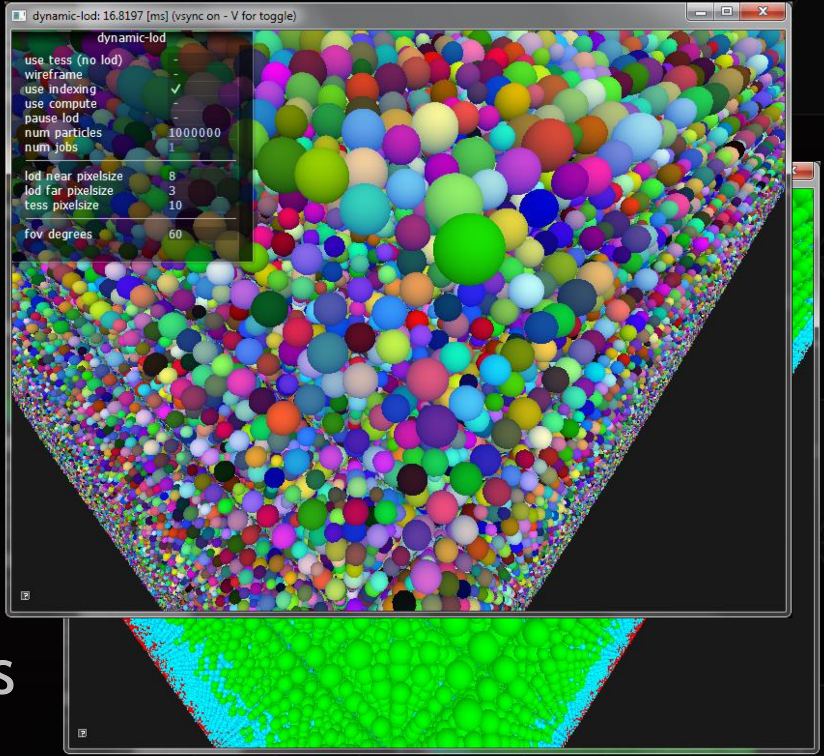
Preliminary results M6000, * taken from slightly differnt framework

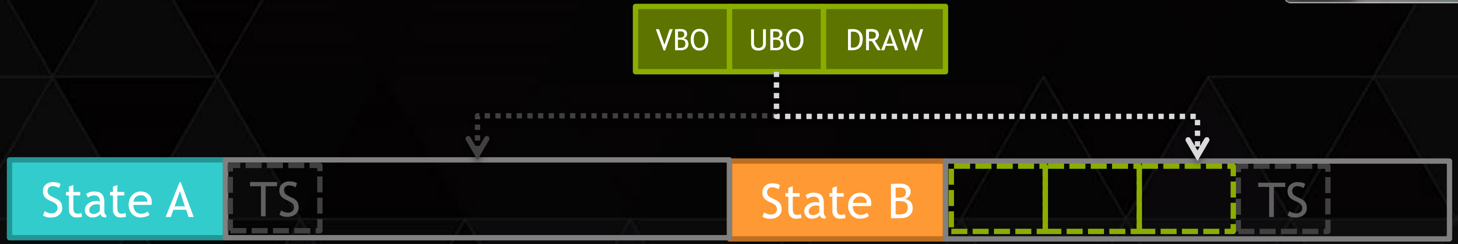


```
No instancing! (data replication)

SCENE:
  materials:            138
  objects:           13,032
  geometries:         3,312
  parts:            789,464
  triangles:     29,527,840
  vertices:      27,584,376
```

DYNAMIC LEVEL OF DETAIL

- For example particle LOD
  - GPU classifies how to render particles based on screen area, without CPU envolved (GL 4.3)
    - Point sprite
    - Simple mesh via enhanced instancing
    - Adaptive tessellation
- Tokens allow same for more complex objects





CONCLUSION

- Leverage GPU to full extent
  - Modern software approaches (command buffers, stateobjects…) found in many new graphics APIs (DX12, Vulkan…) or extended OpenGL
  - Higher fidelity (e.g. multiple scene passes) or interactivity for even larger scenes
  - Save CPU time (power/battery, other work…)
- GPU can do more than „just“ rendering
  - Drive decision making (culling, LOD, interactive scientific data brushing…)
  - Compute auxiliary data (matrices, materials…)
  - NV_command_list and NVIDIA‘s bindless enable workflows beyond core api

GPU TECHNOLOGY CONFERENCE

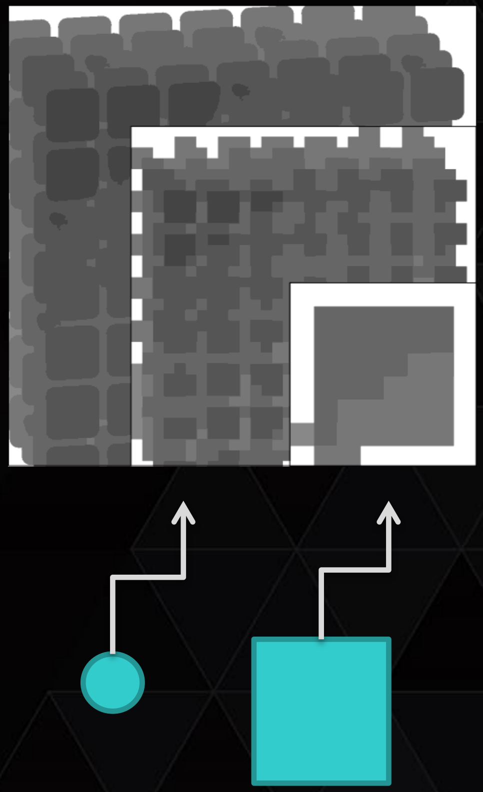
THANK YOU

- Contact: ckubisch@nvidia.com @pixeljetstream
- Sample code
  - https://github.com/nvpro-samples
- Past presentations
  - http://www.slideshare.net/tlorach/opengl-nvidia-commandlistapproaching-zerodriveroverhead
  - http://on-demand.gputechconf.com/siggraph/2014/presentation/SG4117-OpenGL-Scene-Rendering-Techniques.pdf
  - http://on-demand.gputechconf.com/gtc/2013/presentations/S3032-Advanced-Scenegraph-Rendering-Pipeline.pdf
- OpenGL work creation references
  - http://rastergrid.com/blog/2010/10/gpu-based-dynamic-geometry-lod/
  - http://rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling/

BACKUP

HIZ CULLING

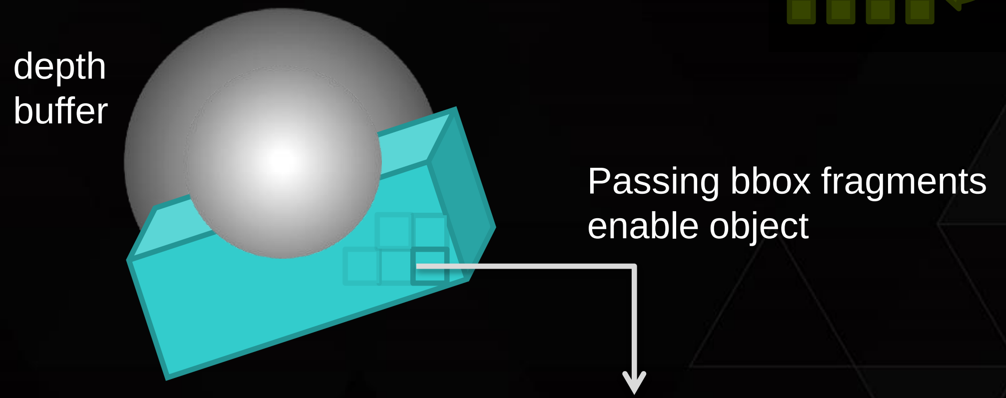
- OpenGL 3.x/4.x
  - Depth-Pass
  - Create mipmap pyramid, MAX depth
    - GM2xx supports GL_EXT_texture_filter_minmax
  - „invisible“ vertex shader or compute
    - Compare object‘s clipspace bbox against z value of depth mip
    - The mip level is chosen by clipspace 2D area



Projected size determines depth mip level
→ mip texels cover object

RASTER CULLING

- OpenGL 4.2+
  - Depth-Pass
  - Raster „invisible“ bounding boxes
    - Disable Color/Depth writes
    - Geometry Shader to create the three visible box sides
    - Depth buffer discards occluded fragments (earlyZ...)
    - Fragment Shader writes output: visible[objindex] = 1




```
// GLSL fragment shader
// from ARB_shader_image_load_store
layout(early_fragment_tests) in;

buffer visibilityBuffer{
  int visibility[];   // cleared to 0
};

flat in int objectID; // unique per box

void main()
{
  visibility[objectID] = 1;
  // no atomics required (32-bit write)
}
```

Algorithm by
Evgeny Makarov, NVIDIA

TEMPORAL COHERENCE

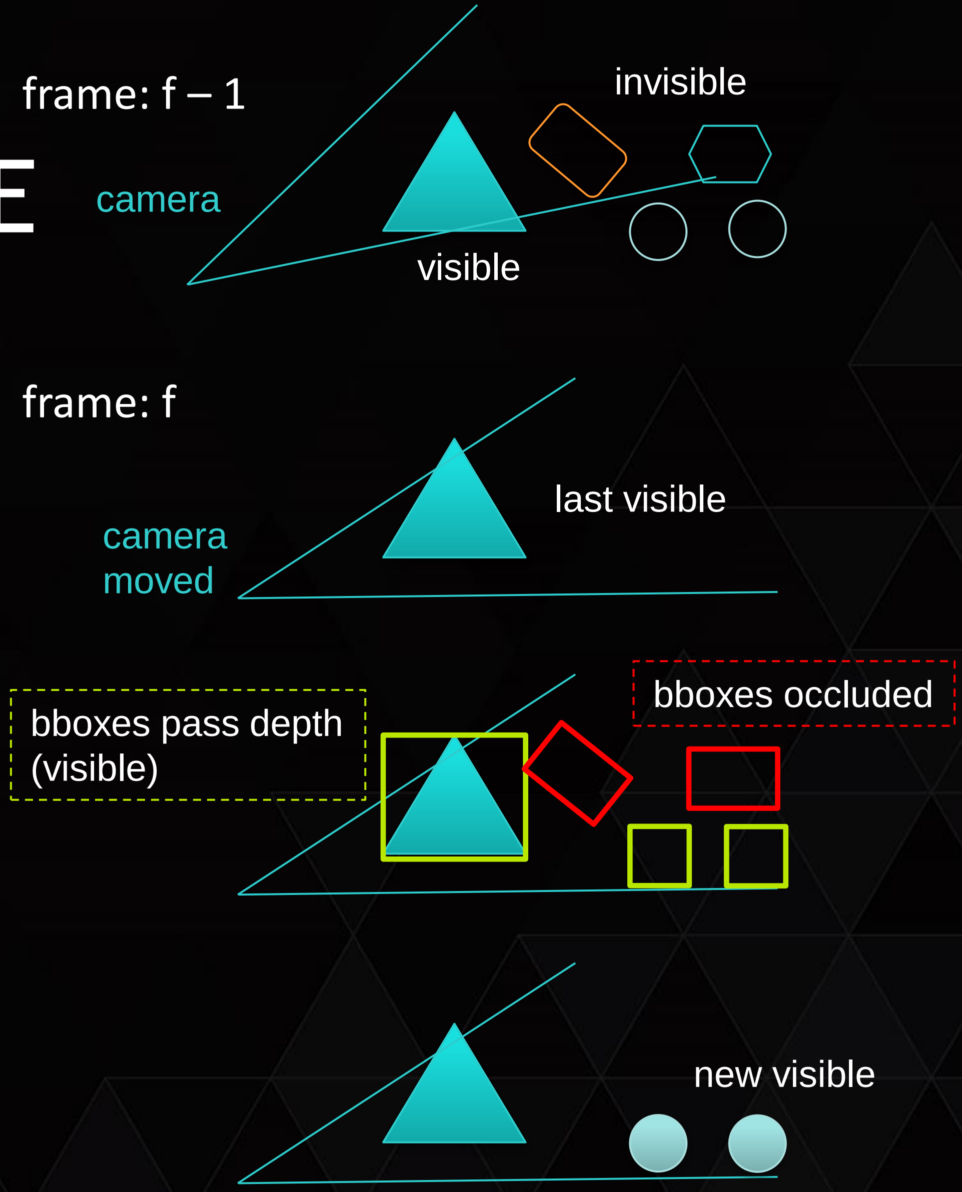
- Few changes relative to camera
- Draw each object only once
  - Render last visible, fully shaded

    `(last)`
  - Test all against current depth:

    `(visible)`
  - Render newly added visible:

    none, if no spatial changes made

    `(~last) & (visible)`
  - `(last) = (visible)`



Algorithm by
Markus Tavenrath, NVIDIA