GPU TECHNOLOGY CONFERENCE

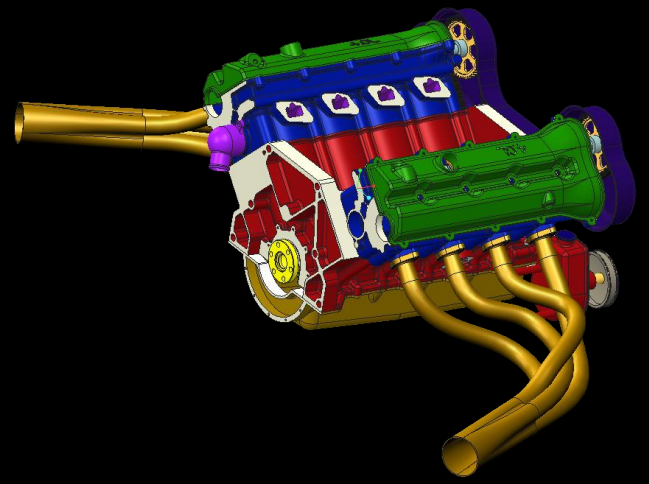
NVIDIA

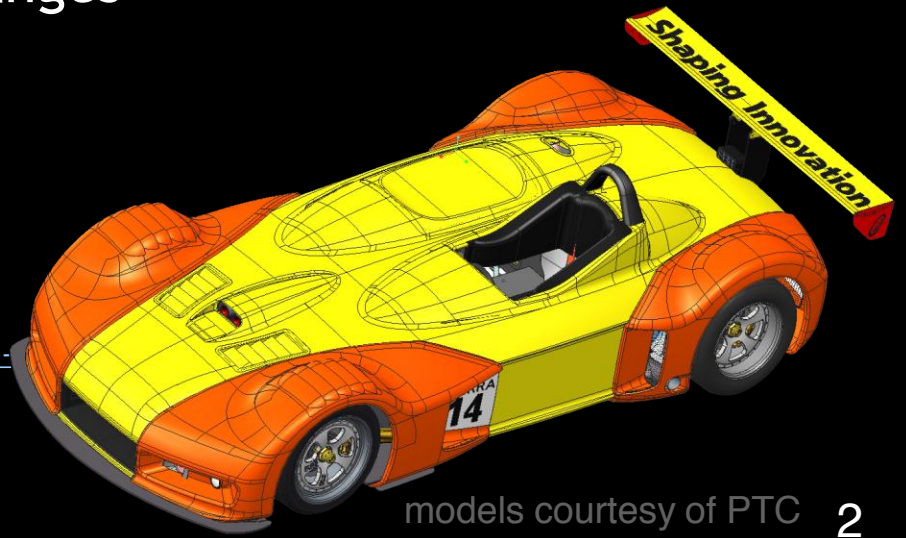
OpenGL 4.4
Scene Rendering Techniques

Christoph Kubisch - ckubisch@nvidia.com
Markus Tavenrath - matavenrath@nvidia.com

Scene Rendering

- Scene complexity increases
  - Deep hierarchies, traversal expensive
  - Large objects split up into a lot of little pieces, increased draw call count
  - Unsorted rendering, lot of state changes
- CPU becomes bottleneck when rendering those scenes
- Removing SceneGraph traversal:
  - http://on-demand.gputechconf.com/gtc/2013/presentations/S3032-Advanced-Scenegraph-Rendering-Pipeline.pdf

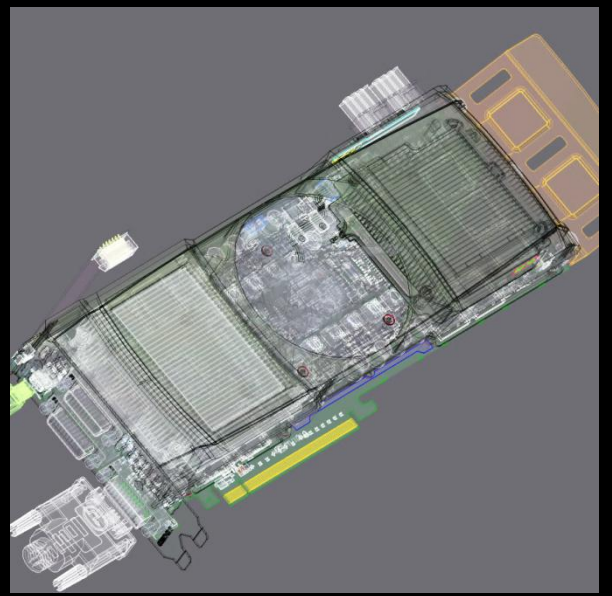




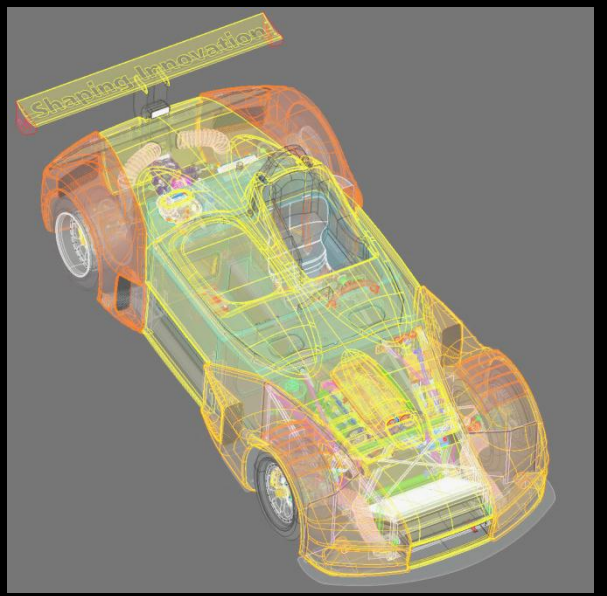
models courtesy of PTC

Challenge not necessarily obvious

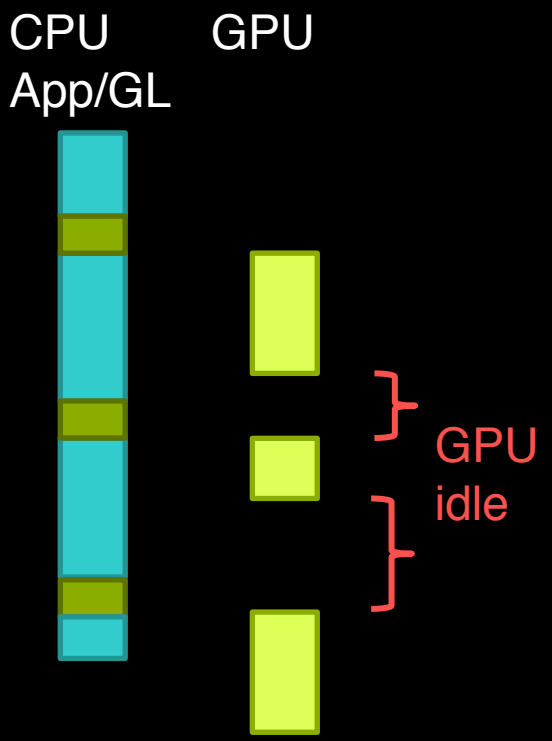
- Harder to render „Graphicscard“ efficiently than „Racecar“



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

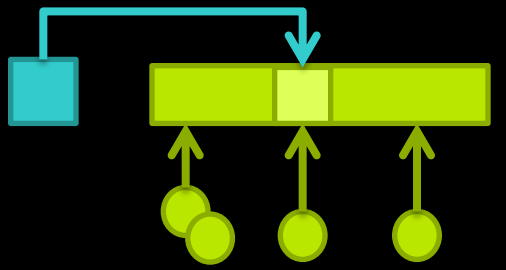


- 3 700 000 Triangles
- 98 000 Parts
- ~ 37 Triangles per part

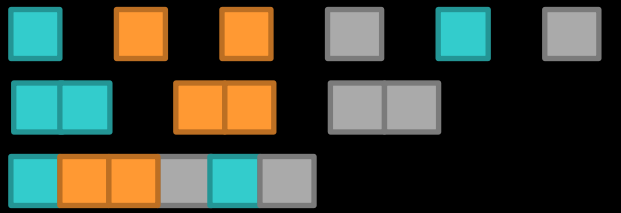


Enabling GPU Scalability

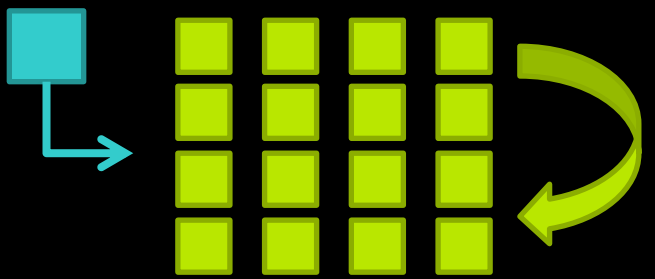
- Avoid data redundancy
  - Data stored once, referenced multiple times
  - Update only once (less host to gpu transfers)



- Increase GPU workload per job (batching)
  - Further cuts API calls
  - Less driver CPU work



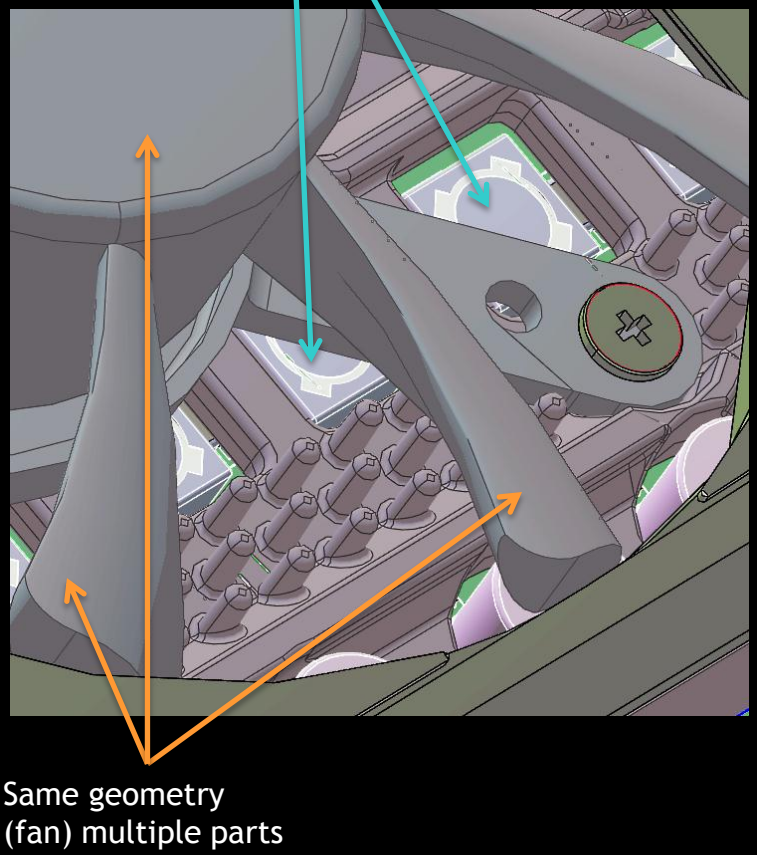
- Minimize CPU/GPU interaction
  - Allow GPU to update its own data
  - Low API usage when scene is changed little
  - E.g. GPU-based culling



Rendering Research Framework

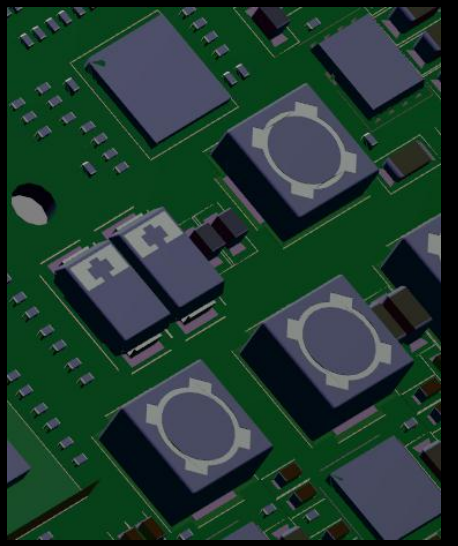
- Avoids classic SceneGraph design
- Geometry
  - Vertex & Index-Buffer (VBO & IBO)
  - Parts (CAD features)
- Material
- Matrix Hierarchy
- Object

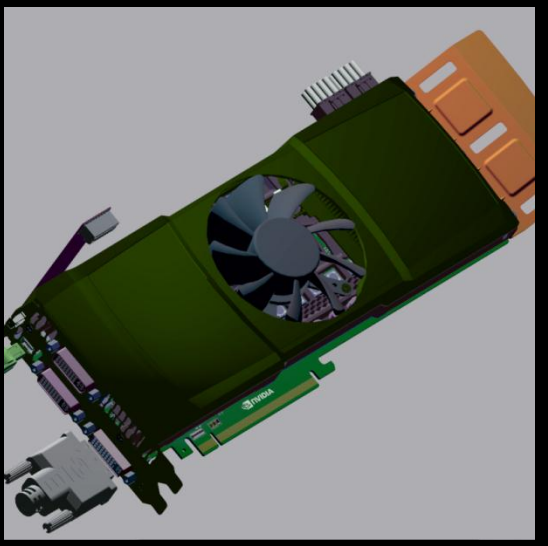
  References Geometry, Matrix, Materials



Performance baseline

- Benchmark System
  - Core i7 860 2.8Ghz
  - Kepler Quadro K5000
  - Driver upcoming this summer
- Showing evolution of techniques
  - Render time basic technique 32ms (31fps), CPU limited
  - Render time best technique 1.3ms (769fps)
  - Total speedup of **24.6x**





110 geometries, 66 materials

2500 objects

Basic technique 1: 32ms CPU-bound

- Classic uniforms for parameters
- VBO bind per part, drawcall per part, 68k binds/frame

```
foreach (obj in scene) {


    setMatrix (obj.matrix);

    // iterate over different materials used
    foreach (part in obj.geometry.parts) {
        setupGeometryBuffer (part.geometry); // sets vertex and index buffer
        setMaterial_if_changed (part.material);
        drawPart (part);
    }
}
```

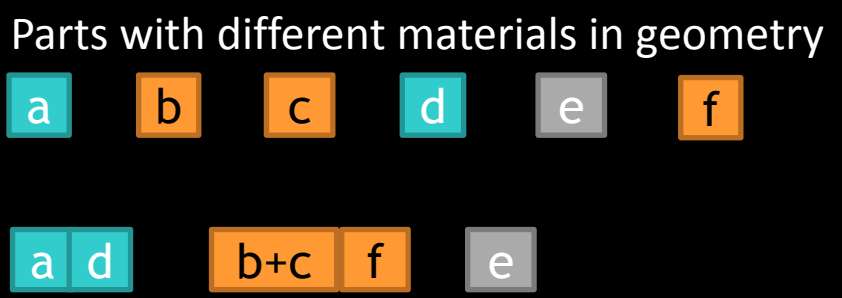
Basic technique 2: 17 ms CPU-bound

- Classic uniforms for parameters
- VBO bind per geometry, drawcall per part, 2.5k binds/frame

```
foreach (obj in scene) {

    setupGeometryBuffer (obj.geometry); // sets vertex and index buffer
    setMatrix (obj.matrix);

    // iterate over parts
    foreach (part in obj.geometry.parts) {

        setMaterial_if_changed (part.material);
        drawPart (part);
    }
}
```

Drawcall Grouping

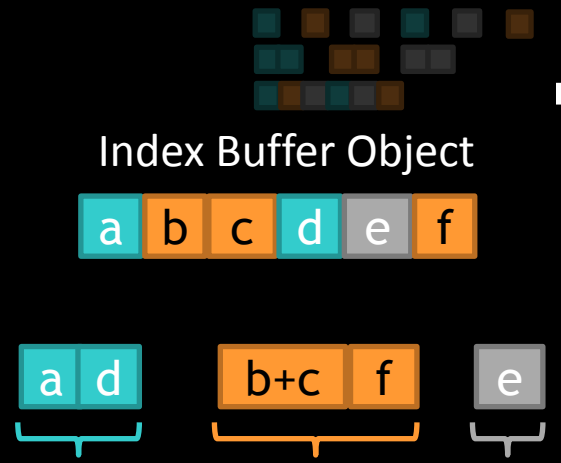
- Combine parts with same state
  - Object‘s part cache must be rebuilt based on material/enabled state




```
foreach (obj in scene) {
     // sets vertex and index buffer
    setupGeometryBuffer (obj.geometry);
    setMatrix (obj.matrix);

    // iterate over material batches: 6.8 ms ☺ -> 2.5x
    foreach (batch in obj.materialCache) {

        setMaterial (batch.material);
        drawBatch (batch.data);
    }
}
```

glMultiDrawElements (GL 1.4)

- glMultiDrawElements supports multiple index buffer ranges



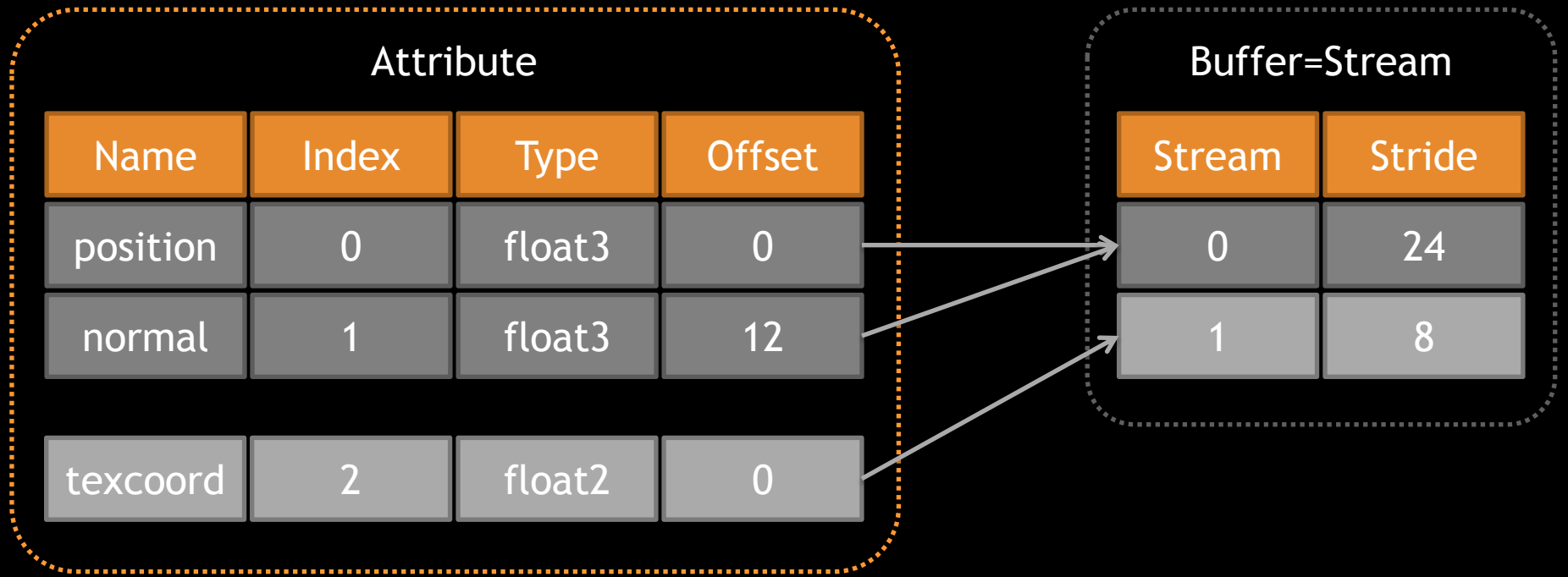
offsets[] and counts[] per batch for glMultiDrawElements

```
drawBatch (batch) { // 6.8 ms
    foreach range in batch.ranges {
      glDrawElements (GL_.., range.count, .., range.offset);
    }
}

drawBatch (batch) { // 6.1 ms ☺ -> 1.1x
    glMultiDrawElements (GL_.., batch.counts[], .., batch.offsets[],
       batch.numRanges);
}
```

Vertex Setup

```
foreach (obj in scene) {

    setupGeometryBuffer (obj.geometry);
    setMatrix (obj.matrix);

    // iterate over different materials used
    foreach (batch in obj.materialCache) {
        setMaterial (batch.material);
        drawBatch (batch.geometry);
    }
}
```

Vertex Format Description



Vertex Setup VBO (GL 2.1)

- One call required for each attribute and stream
- Format is being passed when updating ‚streams‘
- Each attribute could be considered as one stream

```
void setupVertexBuffer (obj) {
  glBindBuffer (GL_ARRAY_BUFFER, obj.positionNormal);
  glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 24,  0); // pos
  glVertexAttribPointer (1, 3, GL_FLOAT, GL_FALSE, 24, 12); // normal

  glBindBuffer (GL_ARRAY_BUFFER, obj.texcoord);
  glVertexAttribPointer (2, 2, GL_FLOAT, GL_FALSE, 8, 0);   // texcoord
}
```

Vertex Setup VAB (GL 4.3)

- ARB_vertex_attrib_binding separates format and stream

```
void setupVertexBuffer(obj) {

  if formatChanged(obj) {
    glVertexAttribFormat  (0, 3, GL_FLOAT, false,  0); // position
    glVertexAttribFormat  (1, 3, GL_FLOAT, false, 12); // normal
    glVertexAttribFormat  (2, 2, GL_FLOAT, false,  0); // texcoord
    glVertexAttribBinding (0, 0); // position -> stream 0
    glVertexAttribBinding (1, 0); // normal   -> stream 0
    glVertexAttribBinding (2, 1); // texcoord -> stream 1
  }

  //                  stream, buffer,              offset, stride
  glBindVertexBuffer (0     , obj.positionNormal, 0     , 24    );
  glBindVertexBuffer (1     , obj.texcoord      , 0     , 8     );
}
```

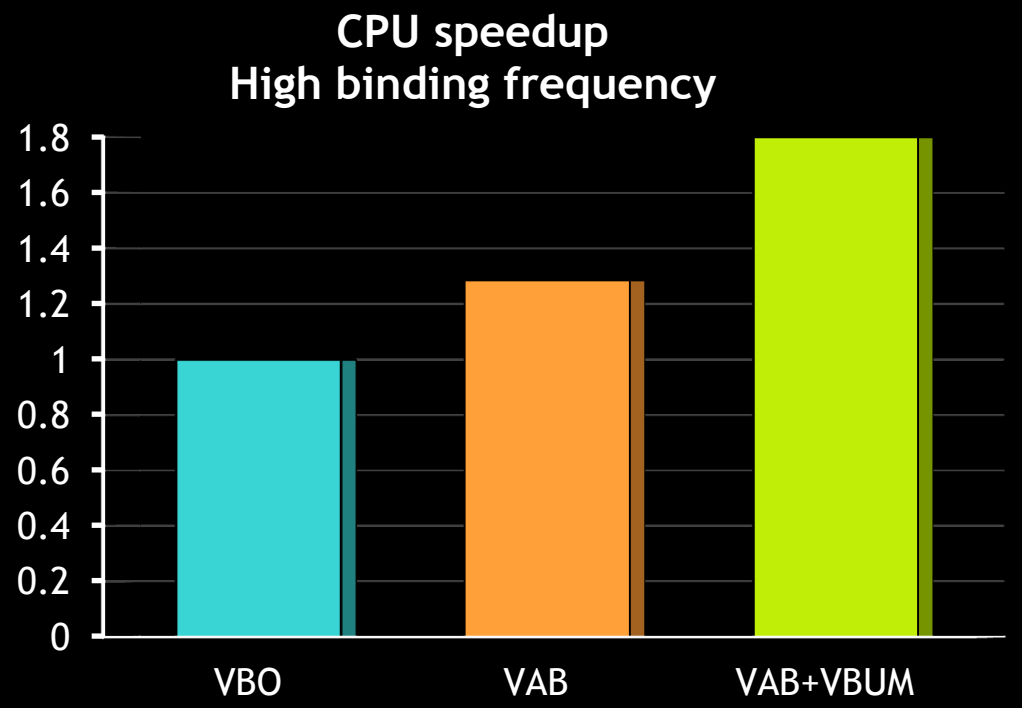
Vertex Setup VBUM

- NV_vertex_buffer_unified_memory uses buffer addresses

```
glEnableClientState (GL_VERTEX_ATTRIB_UNIFIED_NV); // enable once

void setupVertexBuffer(obj) {
  if formatChanged(obj) {
    glVertexAttribFormat (0, 3, . . .

    //                stream, buffer, offset, stride
    glBindVertexBuffer (0,    0,        0,      24); // dummy binds
    glBindVertexBuffer (1,    0,        0,      8);  // to update stride
  }
  // no binds, but 64-bit gpu addresses              stream
  glBufferAddressRangeNV (GL_VERTEX_ARRAY_ADDRESS_NV, 0, addr0, length0);
  glBufferAddressRangeNV (GL_VERTEX_ARRAY_ADDRESS_NV, 1, addr1, length1);
}
```

Vertex Setup

- Framework uses only one stream and three attributes
- VAB benefit depends on vertex buffer bind frequency



Parameter Setup

```
foreach (obj in scene) {

    setupGeometryBuffer (obj.geometry);
    setMatrix (obj.matrix); // once per object

    // iterate over different materials used
    foreach (batch in obj.materialCaches) {
        setMaterial (batch.material); // once per batch
        drawBatch (batch.geometry);
    }

}
```

Parameter Setup

- Group parameters by frequency of change
- Generate GLSL shader parameters

```
Effect "Phong" {
  Group "material" {
    vec4 "ambient"
    vec4 "diffuse"
    vec4 "specular"
  }
  Group "object" {
    mat4 "world"
    mat4 "worldIT"
  }
  Group "view" {
    vec4 "viewProjTM"
  }
... Code ...
}
```

- OpenGL 2 uniforms
- OpenGL 3.x, 4.x buffers

Uniform

- glUniform (2.x)
  - one glUniform per parameter (simple)
  - one glUniform array call for all parameters (ugly)

```
// matrices
uniform mat4 matrix_world;
uniform mat4 matrix_worldIT;

// material
uniform vec4 material_diffuse;
uniform vec4 material_emissive;
...

// material fast but „ugly“
uniform vec4 material_data[8];
#define material_diffuse material_data[0]
...
```

Uniform

```
foreach (obj in scene) {
    ...
    glUniform (matrixLoc,   obj.matrix);
    glUniform (matrixITLoc, obj.matrixIT);

    // iterate over different materials used
    foreach ( batch in obj.materialCaches) {
        glUniform (frontDiffuseLoc, batch.material.frontDiffuse);
        glUniform (frontAmbientLoc, batch.material.frontAmbient);
        glUniform (...)
        ...

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

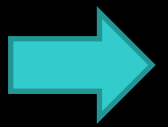
BufferSubData

```
glBindBufferBase (GL_UNIFORM_BUFFER, 0, uboMatrix);
glBindBufferBase (GL_UNIFORM_BUFFER, 1, uboMaterial);

foreach (obj in scene) {
    ...
[   glNamedBufferSubDataEXT (uboMatrix, 0, maSize, obj.matrix);


    // iterate over different materials used
    foreach ( batch in obj.materialCaches) {
[       glNamedBufferSubDataEXT (uboMaterial, 1, mtlSize, batch.material);



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

Uniform to UBO transition

- Changes to existing shaders are minimal
  - Surround block of parameters with uniform block
  - Actual shader code remains unchanged
- Group parameters by frequency

```
// matrices
uniform mat4 matrix_world;
uniform mat4 matrix_worldIT;


// material
uniform vec4 material_diffuse;
uniform vec4 material_emissive;
...
```



```
layout(std140,binding=0) uniform matrixBuffer {
  mat4 matrix_world;
  mat4 matrix_worldIT;
};

layout(std140,binding=1) uniform materialBuffer {
  vec4 material_diffuse;
  vec4 material_emissive;
  ...
};
```

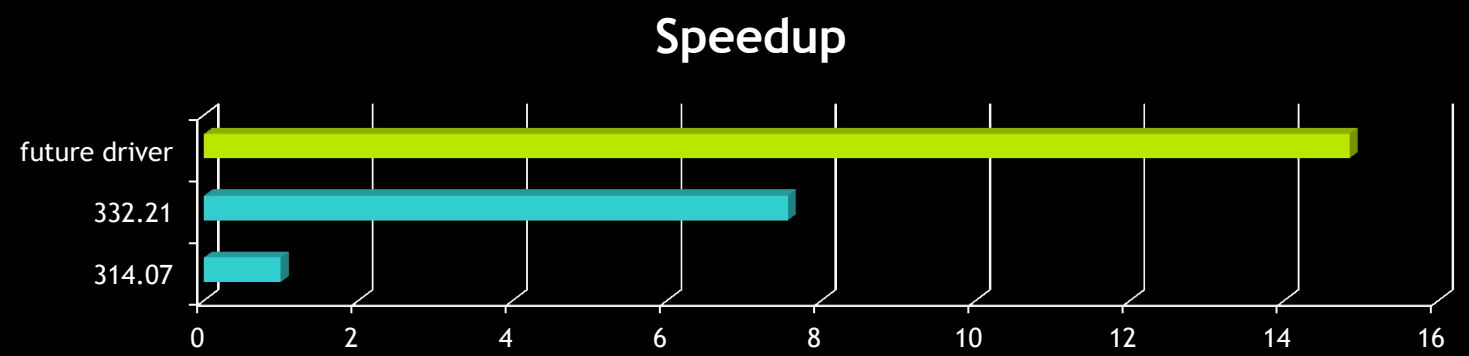
Performance

- Good speedup over multiple glUniform calls
- Efficiency still dependent on size of material

Technique	Draw time
Uniform	5.2 ms
BufferSubData	2.7 ms 1.9x

BufferSubData

- Use glBufferSubData for dynamic parameters
- Restrictions to get effcient path
  - Buffer only used as GL_UNIFORM_BUFFER
  - Buffer is <= 64kb
  - Buffer bound offset == 0 (glBindBufferRange)
  - Offset and size passed to glBufferSubData be a multiple of 4



BindBufferRange

```
UpdateMatrixAndMaterialBuffer();

foreach (obj in scene) {
    ...

[   glBindBufferRange (UBO, 0, uboMatrix, obj.matrixOffset, maSize);

    // iterate over different materials used
    foreach ( batch in obj.materialCaches) {
[     glBindBufferRange (UBO, 1, uboMaterial, batch.materialOffset, mtlSize);

      glMultiDrawElements (...);
    }

}
```

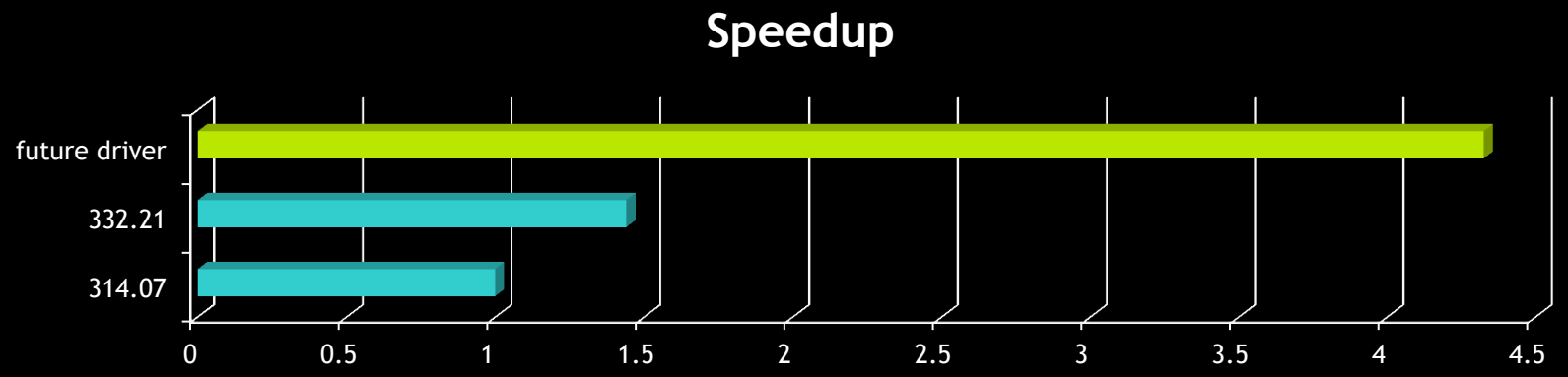
Performance

- glBindBufferRange speed independent of data size
  - Material used in framework is small (128 bytes)
  - glBufferSubData will suffer more with increasing data size

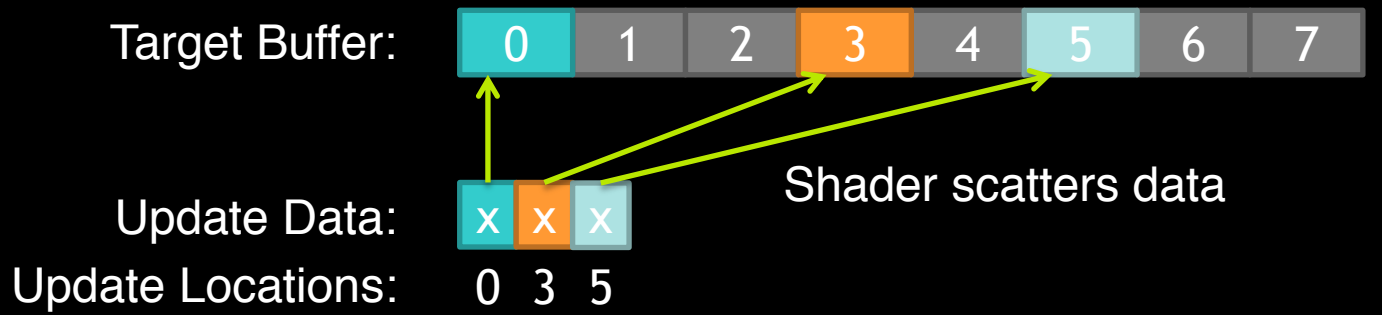
Technique	Draw time
glUniforms	5.2 ms
glBufferSubData	2.7 ms 1.9x
glBindBufferRange	2.0 ms 2.6x

BindRange

- Avoid expensive CPU -> GPU copies for static data
- Upload static data once and bind subrange of buffer
  - glBindBufferRange (target, index, buffer, offset, size);
  - Offset aligned to GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT
  - Fastest path: One buffer per binding index



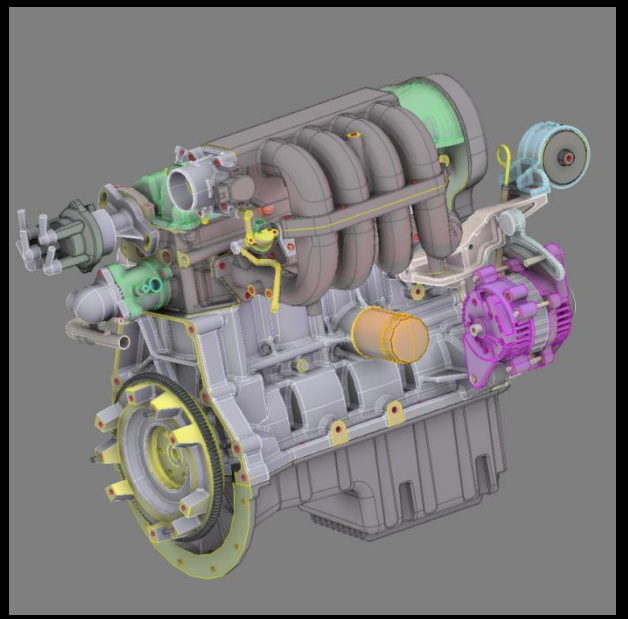
Incremental Buffer Updates

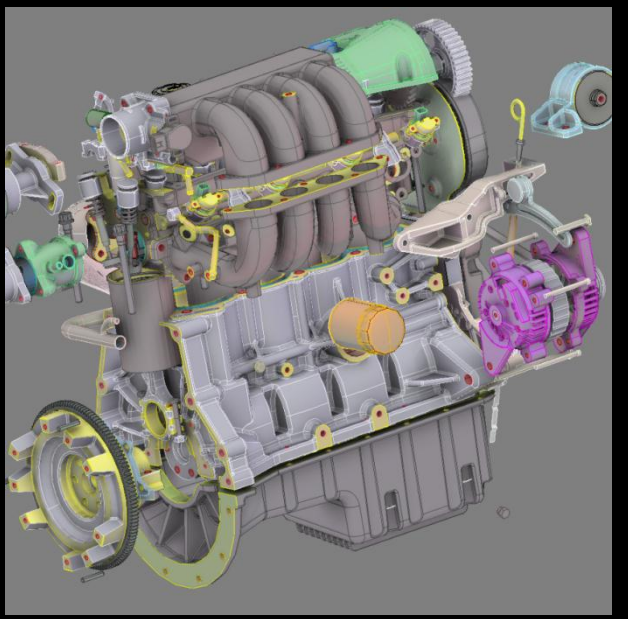


- Buffer may be large and sparse
  - Full update could be ‚slow‘ because of unused/padded data
  - Too many small glBufferSubData calls
- Use Shader to write into Buffer (via SSBO)
  - Provides compact CPU -> GPU transfer

Transform Tree Updates

- All matrices stored on GPU
  - Use ARB_compute_shader for hierarchy updates ☺
  - Send only local matrix changes, evaluate tree

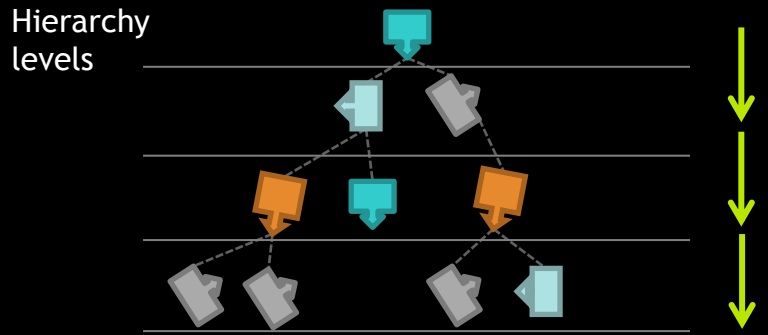




model courtesy of PTC

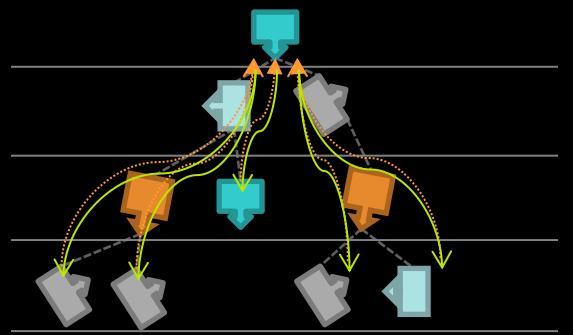
Transform Tree Updates

- Update hierarchy on GPU
  - Level- and Leaf-wise processing depending on workload
  - world = parent.world * object



Level-wise waits for previous results

- Risk of little work per level



Leaf-wise runs to top, then concats path downwards per thread

- Favors more total work over redundant calculations

Indexed

- TextureBufferObject (TBO) for matrices
- UniformBufferObject (UBO) with array data to save binds
- Assignment indices passed as vertex attribute or uniform
- Caveat: costs for indexed fetch

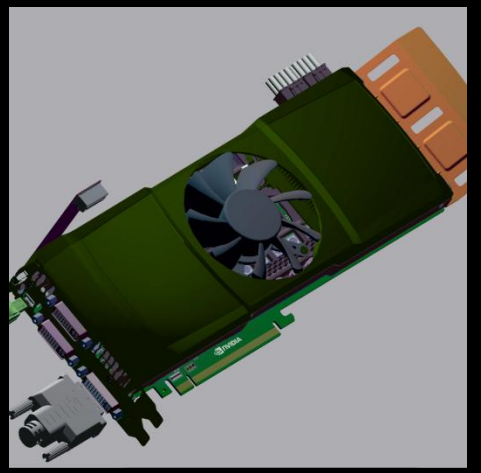
```
in vec4 oPos;

uniform  samplerBuffer matrixBuffer;

uniform  materialBuffer {
  Material materials[512];
};

in ivec2 vAssigns;
flat out ivec2 fAssigns;

// in vertex shader
  fAssigns = vAssigns;

  worldTM = getMatrix (matrixBuffer,
                       vAssigns.x);
  wPos = worldTM * oPos;
...
// in fragment shader

  color = materials[fAssigns.y].color;
...
```

Indexed

```
setupSceneMatrixAndMaterialBuffer (scene);

foreach (obj in scene) {
    setupVertexBuffer (obj.geometry);

    // iterate over different materials used
    foreach ( batch in obj.materialCache ) {

      glVertexAttribI2i (indexAttr, batch.materialIndex, matrixIndex);

      glMultiDrawElements (GL_TRIANGLES, batch.counts, GL_UNSIGNED_INT ,
                           batch.offsets,batched.numUsed);
    }
  }
}
```

Indexed

- Scene and hardware dependent benefit



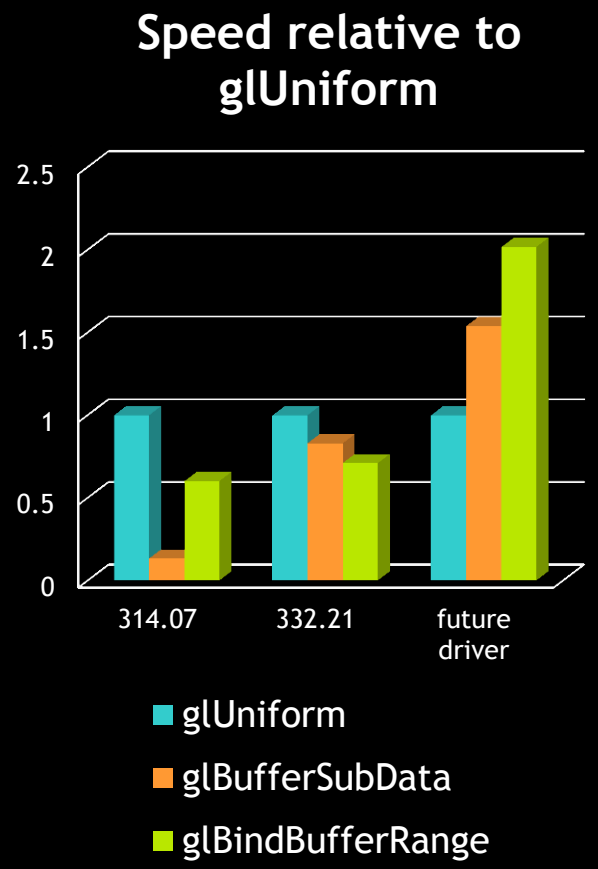
avg 55 triangles per drawcall

avg 1500 triangles per drawcall

Timer	Graphicscard		Racecar	
Hardware	K5000	K2000	K5000	K2000
BindBufferRange GPU	2.0 ms	3.3 ms	2.4 ms	7.4 ms
Indexed GPU	1.6 ms 1.25x	3.6 ms 0.9x	2.5 ms 0.96x	7.7 ms 0.96x
BindBufferRange CPU	2.0 ms		0.5 ms	
Indexed CPU	1.1 ms 1.8x		0.3 ms 1.6x	

Recap

- glUniform
  - Only for tiny data (<= vec4)
- glBufferSubData
  - Dynamic data
- glBindBufferRange
  - Static or GPU generated data
- Indexed
  - TBO/SSBO for large/random access data
  - UBO for frequent changes (bad for divergent access)



Multi Draw Indirect

- Combine even further
  - Use MultiDrawIndirect for single drawcall
  - Can store array of drawcalls on GPU

Grouped and „grown“ drawcalls

Single drawcall with material/matrix changes

```
DrawElementsIndirect
{
   GLuint      count;
   GLuint      instanceCount;
   GLuint      firstIndex;
   GLint       baseVertex;
   GLuint      baseInstance;
}

DrawElementsIndirect  object.drawCalls[ N ];
```

encodes material/matrix assignment

Multi Draw Indirect

- Parameters:
  - TBO and UBO as before
  - ARB_shader_draw_parameters for gl_BaseInstanceARB access
  - Caveat:
    - Currently slower than vertex-divisor technique shown GTC 2013 for very low primitive counts (improvement being investigated)

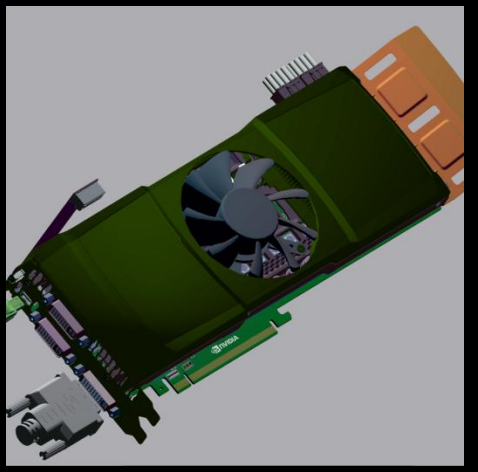
```
uniform  samplerBuffer matrixBuffer;

uniform  materialBuffer {
  Material materials[256];
};

// encoded assignments in 32-bit
ivec2 vAssigns =
   ivec2 (gl_BaseInstanceARB >> 16,
          gl_BaseInstanceARB & 0xFFFF);
flat out ivec2 fAssigns;

// in vertex shader
  fAssigns = vAssigns;

  worldTM = getMatrix (matrixBuffer,
                       vAssigns.x);
...
// in fragment shader
  color = materials[fAssigns.y].diffuse...
```

Multi Draw Indirect

```
setupSceneMatrixAndMaterialBuffer (scene);

glBindBuffer (GL_DRAW_INDIRECT_BUFFER, scene.indirectBuffer)

foreach ( obj in scene.objects ) {
   ...

   // draw everything in one go
   glMultiDrawElementsIndirect ( GL_TRIANGLES, GL_UNSIGNED_INT,
                                 obj->indirectOffset, obj->numIndirects, 0 );
}
```

Performance

- Multi Draw Indirect (MDI) is primitive dependent



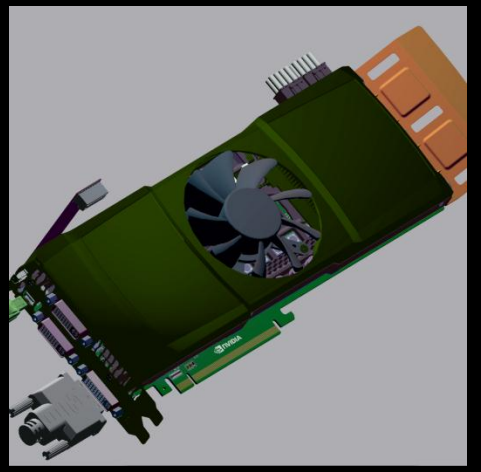
avg 55 triangles per drawcall

avg 1500 triangles per drawcall

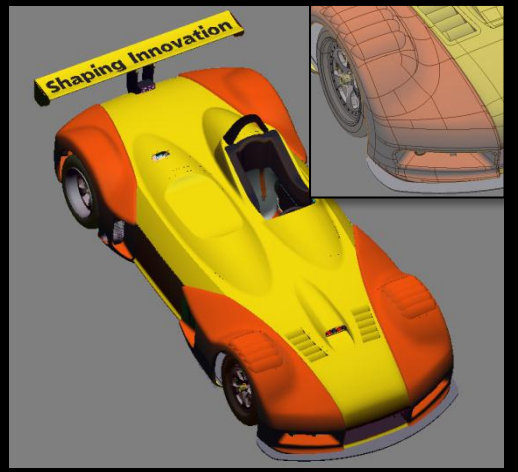
Timer	Graphicscard	Racecar
Indexed GPU	1.6 ms	2.5 ms
MDI w. gl_BaseInstanceARB	2.0 ms **0.8x**	2.5 ms
MDI w. vertex divisor	1.3 ms **1.5x**	2.5 ms
Indexed CPU	1.1 ms	0.3 ms
MDI	0.5 ms **2.2x**	0.3 ms

Performance

- Multi Draw Indirect (MDI) is great for non-material-batched case



68.000 drawcommands



98.000 drawcommands

Timer	Graphicscard	Racecar
Indexed (not batched) GPU	6.3 ms	8.7 ms
MDI w. vertex divisor (not batched)	2.5 ms **2.5x**	3.6 ms **2.4x**
Indexed (not batched) CPU	6.4 ms	8.8 ms
MDI w. vertex divisor (not batched)	0.5 ms **12.8x**	0.3 ms **29.3x**

NV_bindless_multidraw_indirect

- DrawIndirect combined with VBUM

```
DrawElementsIndirect
{
   GLuint       count;
   GLuint       instanceCount;
   GLuint       firstIndex;
   GLint        baseVertex;
   GLuint       baseInstance;
}

BindlessPtr
{
   Gluint       index;
   Gluint       reserved;
   GLuint64     address;
   GLuint64     length;
}
```

```
MyDrawIndirectNV
{
  DrawElementsIndirect  cmd;
  GLuint                reserved;
  BindlessPtr           index;
  BindlessPtr           vertex;  // for position, normal...
}
```

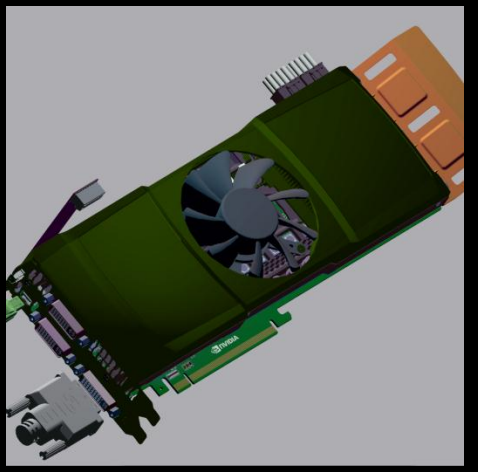
- Caveat:
  - more costly than regular MultiDrawIndirect
  - Should have > 500 triangles worth of work per drawcall

NV_bindless_multidraw_indirect

```
// enable VBUM vertexformat
...

glBindBuffer (GL_DRAW_INDIRECT_BUFFER, scene.indirectBuffer)

// draw entire scene one go ☺
// one call per shader

glMultiDrawElementsIndirectBindlessNV
  (GL_TRIANGLES, GL_UNSIGNED_INT,
   scene->indirectOffset, scene->numIndirects,
   sizeof(MyDrawIndirectNV),
   1  // 1 vertex attribute binding);
```

Performance

- NV_bindless_multi... is primitive dependent



avg 55 triangles per drawcall

avg 1500 triangles per drawcall

Timer	Graphicscard	Racecar
MDI w. gl_BaseInstanceARB GPU	2.0 ms	2.5 ms
NV_bindless..	2.3 ms 0.86x	2.5 ms
MDI w. gl_BaseInstanceARB CPU	0.5 ms	0.3 ms
NV_bindless..	0.04 ms 12.5x	0.04 ms 7.5x

Textured Materials

- Scalar data batching is „easy“, how about textures?
  - Test adds 4 unique textures per material
  - Tri-planar texturing, no additional vertex attributes

Textured Materials

- ARB_multi_bind aeons in the making, finally here (4.4 core)

```
// NEW ARB_multi_bind

glBindTextures (0, 4, textures);

// Alternatively EXT_direct_state_access

glBindMultiTextureEXT ( GL_TEXTURE0 + 0, GL_TEXTURE_2D, textures[0]);
glBindMultiTextureEXT ( GL_TEXTURE0 + 1, GL_TEXTURE_2D, textures[1]);
...

// classic selector way

glActiveTexture (GL_TEXTURE0 + 0);
glBindTexture   (GL_TEXTURE_2D, textures[0]);
glActiveTexture (GL_TEXTURE0 + 1 ...
...
```

Textured Materials

- NV/ARB_bindless_texture
  - Manage residency

    `uint64 glGetTextureHandle (tex)`

    `glMakeTextureHandleResident (hdl)`

  - Faster binds

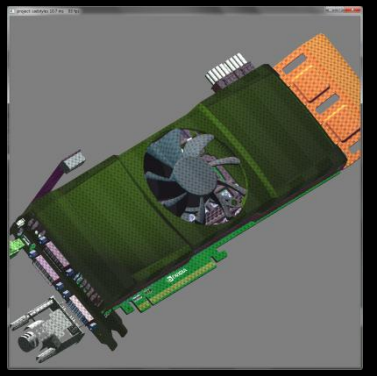
    `glUniformHandleui64ARB (loc, hdl)`

  - store texture handles as 64bit values inside buffers

```
// NEW ARB_bindless_texture stored inside
buffer!

struct MaterialTex {
  sampler2D tex0; // can be in struct
  sampler2D tex1;
  ...
};

uniform  materialTextures {
  MaterialTex texs[128];
};

// in fragment shader

flat in ivec2 fAssigns;
...
color = texture ( texs[fAssigns.y] .tex0,
                  uv);
```

Textured Materials

- CPU Performance
  - Raw test, VBUM+VAB, batched by material



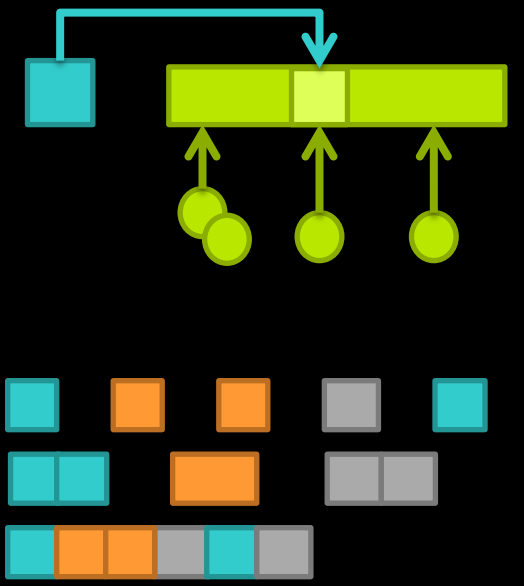
~11.000 x 4 texture binds
66 x 4 unique textures

~2.400 x 4 texture binds
138 x 4 unique textures

Timer	Graphicscard		Racecar
glBindTextures	6.7 ms	(CPU-bound)	1.2 ms
glUniformHandleui64 (BINDLESS)	4.3 ms 1.5x	(CPU-bound)	1.0 ms 1.2x
Indexed handles inside UBO (BINDLESS)	1.1 ms 6.0x		0.3 ms 4.0x

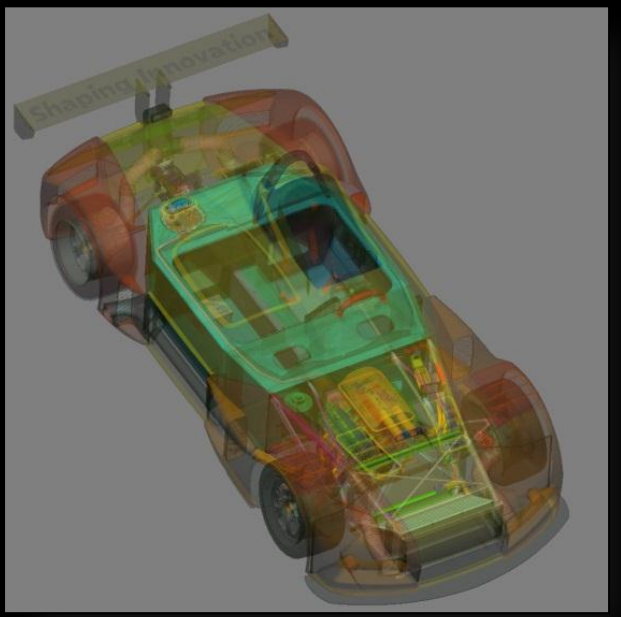
Recap

- Share geometry buffers for batching
- Group parameters for fast updating
- MultiDraw/Indirect for keeping objects independent or remove additional loops
  - BaseInstance to provide unique index/assignments for drawcall
- Bindless to reduce validation overhead/add flexibility

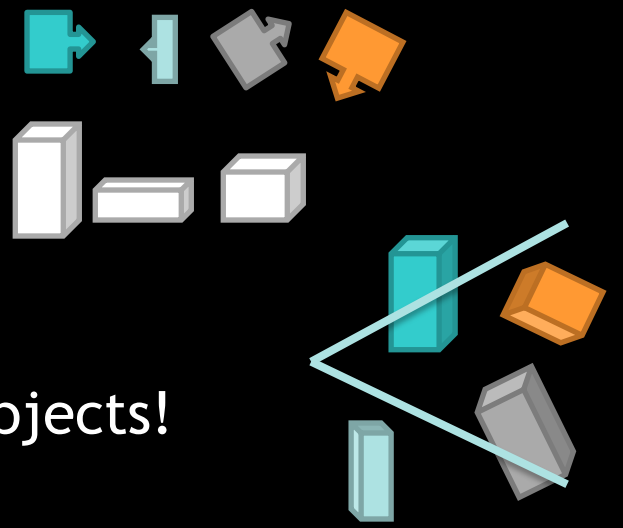


Occlusion Culling

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



GPU Culling Basics



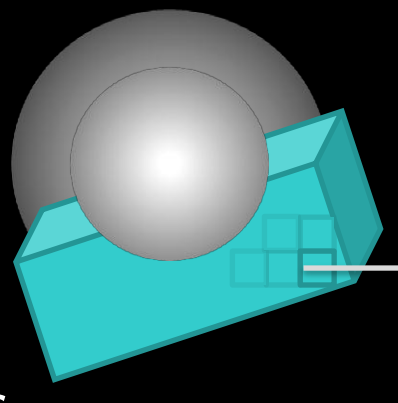
- GPU friendly processing
  - Matrix and bbox buffer, object buffer
  - XFB/Compute or „invisible“ rendering
  - Vs. old techniques: Single GPU job for ALL objects!
- Results
  - „Readback“ GPU to Host
    - Can use GPU to pack into bit stream
  - „Indirect“ GPU to GPU
    - Set DrawIndirect‘s instanceCount to 0 or 1

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

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

Algorithm by
Evgeny Makarov, NVIDIA

Occlusion 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

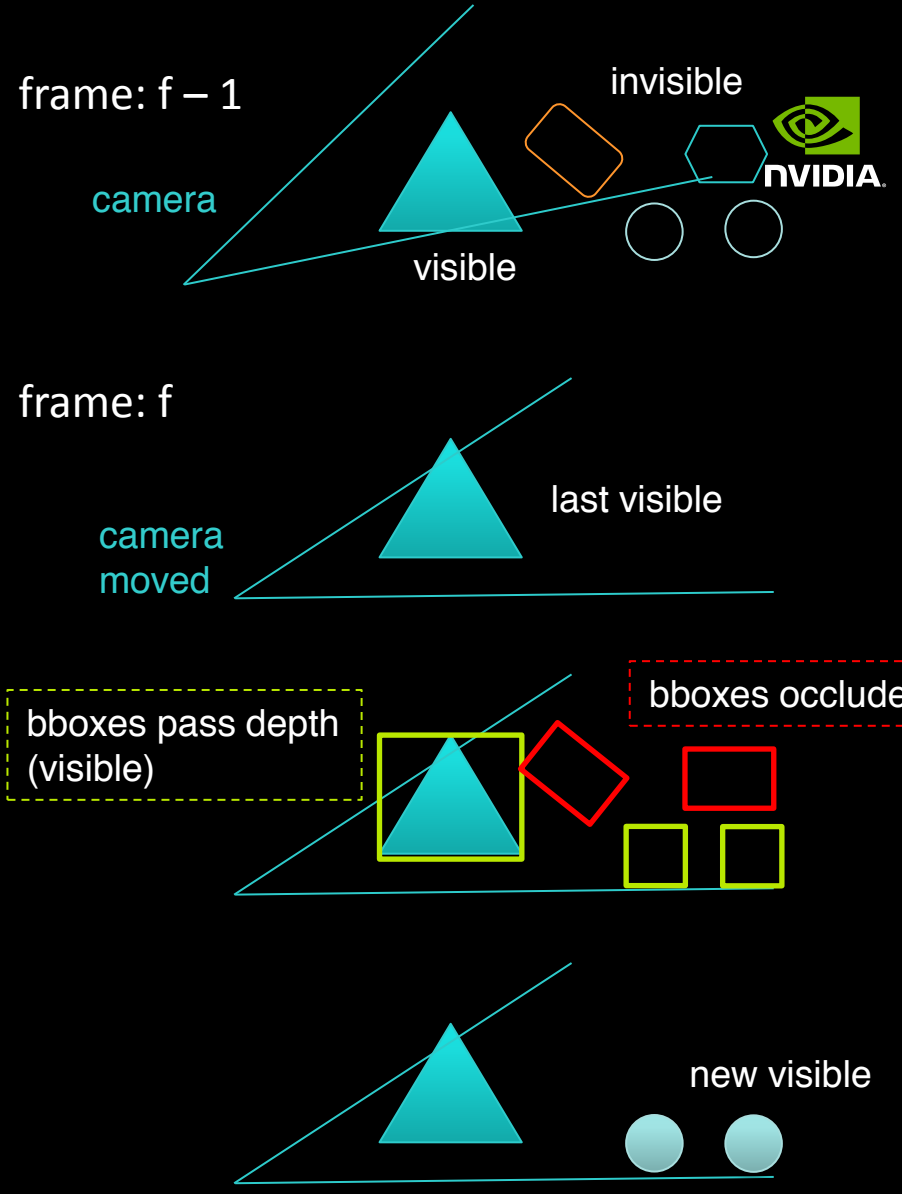



```
// GLSL fragment shader
// from ARB_shader_image_load_store
layout(early_fragment_tests) in;

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
Markus Tavenrath, NVIDIA

Temporal Coherence

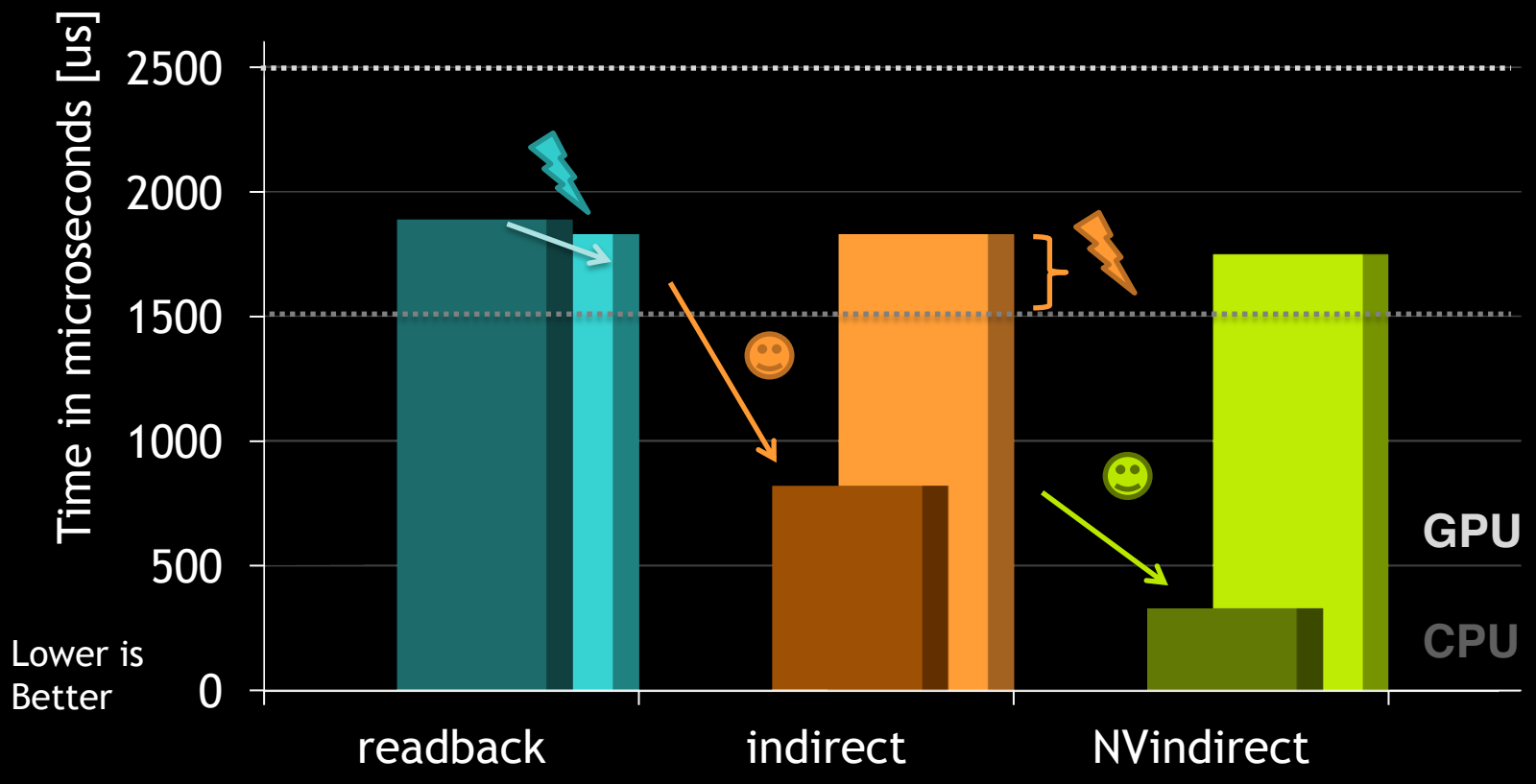
- Few changes relative to camera
- Draw each object only once
  - Render last visible, fully shaded
    `(last)`
  - Test all against current depth:
    `(visible)`
  - Render newly added visible:
    none, if no spatial changes made
    `(~last) & (visible)`
  - `(last) = (visible)`



Culling Readback vs Indirect

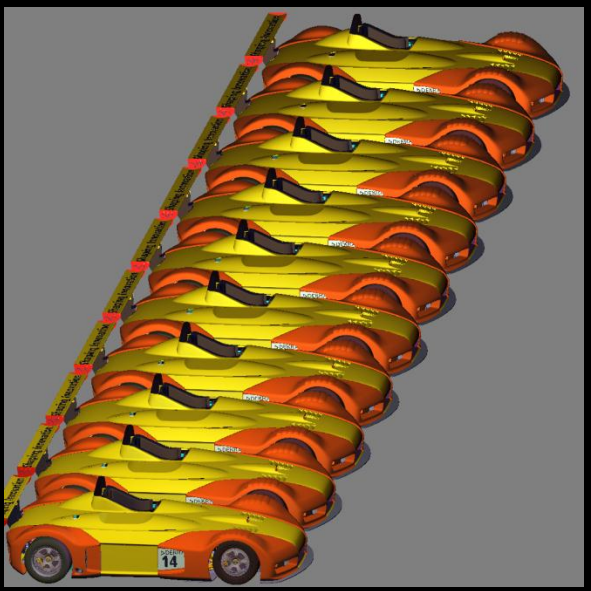
For readback results, CPU has to wait for GPU idle, and GPU may remain idle until new work

Indirect not yet as efficient to process „invisible“ commands



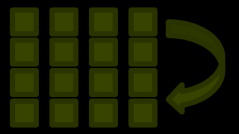
Will it scale?

- 10 x the car: 45 fps
  - everything but materials duplicated in memory, NO instancing
  - 1m parts, 16k objects, 36m tris, 34m verts



- Readback culling: 145 fps **3.2x**
  - 6 ms CPU time, wait for sync takes 5 ms
- Stall-free culling: 115 fps **2.5x**
  - 1 ms CPU time using NV_bindless_multidraw_indirect

Culling Results



- Temporal culling
  - very useful for object/vertex-boundedness
- Readback vs Indirect
  - Readback should be delayed so GPU doesn‘t starve of work
  - Indirect benefit depends on scene (#VBOs and #primitives)
  - „Graphicscard“ model didn‘t benefit of either culling on K5000
- Working towards GPU autonomous system
  - (NV_bindless)/ARB_multidraw_indirect, ARB_indirect_parameters as mechanism for GPU creating its own work

glFinish();

- Thank you!
  - Contact
    - ckubisch@nvidia.com (@pixeljetstream)
    - matavenrath@nvidia.com

Glossary

- VBO: vertex buffer object to store vertex data on GPU (GL server), favor bigger buffers to have less binds, or go bindless
- IBO: index buffer object, GL_ELEMENT_ARRAY_BUFFER to store vertex indices on GPU
- VAB: vertex attribute binding, splits vertex attribute format from vertex buffer
- VBUM: vertex buffer unified memory, allows working with raw gpu address pointer values, avoids binding objects completely
- UBO: uniform buffer object, data you want to access uniformly inside shaders
- TBO: texture buffer object, for random access data in shaders
- SSBO: shader storage buffer object, read & write arbitrary data structures stored in buffers
- MDI: Multi Draw Indirect, store draw commands in buffers