NVPRO-PIPELINE
A RESEARCH RENDERING PIPELINE

MARKUS TAVENRATH - MATAVENRATH@NVIDIA.COM
SENIOR DEVELOPER TECHNOLOGY ENGINEER, NVIDIA
NVPRO-PIPELINE

Peak Double Precision FLOPS

- GPU perf improved better than CPU perf
- In the past apps were GPU bound
- Today apps tend to become CPU bound
- nvpro-pipeline started as research platform to address this issue
- http://github.com/nvpro-pipeline

<table>
<thead>
<tr>
<th>Year</th>
<th>GPU perf</th>
<th>CPU perf</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>500</td>
<td>100</td>
</tr>
<tr>
<td>2009</td>
<td>1000</td>
<td>200</td>
</tr>
<tr>
<td>2010</td>
<td>1500</td>
<td>300</td>
</tr>
<tr>
<td>2011</td>
<td>2000</td>
<td>400</td>
</tr>
<tr>
<td>2012</td>
<td>2500</td>
<td>500</td>
</tr>
<tr>
<td>2013</td>
<td>3000</td>
<td>600</td>
</tr>
<tr>
<td>2014</td>
<td>3500</td>
<td>700</td>
</tr>
</tbody>
</table>
## CPU Boundedness Reasons

<table>
<thead>
<tr>
<th>Application</th>
<th>Pipeline for application like experiments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scene traversal</td>
<td></td>
</tr>
<tr>
<td>Culling</td>
<td></td>
</tr>
<tr>
<td>Other, i.e. animation, simulation</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Driver</th>
<th>Pipeline for OpenGL techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inefficient functionality like glBegin/glEnd</td>
<td></td>
</tr>
<tr>
<td>Functionality which is yet optimized</td>
<td></td>
</tr>
<tr>
<td>CPU-&gt;GPU data transfer</td>
<td></td>
</tr>
</tbody>
</table>
# NVPRO-PIPELINE MODULES

<table>
<thead>
<tr>
<th>SceneGraph</th>
<th>RiX (Renderer)</th>
<th>Effect System</th>
<th>Utilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>[dp::sg]</td>
<td>[dp::rix]</td>
<td>[dp::fx]</td>
<td>[dp::util]</td>
</tr>
<tr>
<td>Algorithms</td>
<td>GL Backend [dp::rix::gl]</td>
<td>XML Based for GLSL</td>
<td>Math library [dp::math]</td>
</tr>
<tr>
<td>SceneTree (XBAR)</td>
<td>Vulkan backend planned</td>
<td>[dp::fx::xml]</td>
<td>Culling [dp::culling]</td>
</tr>
<tr>
<td>Loaders/Savers</td>
<td></td>
<td></td>
<td>Windowing [dp::ui]</td>
</tr>
<tr>
<td>Renderer for RiX::GL</td>
<td></td>
<td></td>
<td>Manipulators [dp::ui::manipulator]</td>
</tr>
</tbody>
</table>

- **SceneGraph [dp::sg]**
- **RiX (Renderer) [dp::rix]**
- **Effect System [dp::fx]**
- **Utilities [dp::util]**
RENDERING PIPELINE

- **SceneGraph**
  - Scene abstraction, algorithms, loaders, savers, …

- **SceneTree (XBAR)**
  - Scene Traversal

- **Rendering Algorithm**
  - Developers code with rendering algorithm

- **EffectFramework**
  - Shader abstraction

- **RiX**
  - OpenGL abstraction, hides VAB, UBO, bindless, …
RENDERING PIPELINE

- **SceneGraph**
  - Scene abstraction, algorithms, loaders, savers, ...

- **SceneTree (XBAR)**
  - Scene Traversal

- **Rendering Algorithm**
  - Developers code with rendering algorithm

- **EffectFramework**
  - Shader abstraction

- **RiX**
  - OpenGL abstraction, hides VAB, UBO, bindless, ...
SCENEGRAPH

- Simplified version of SceniX SceneGraph
  - GeoNodes, Groups, Transforms, Billboards, Switches still available
  - Animated* objects have been removed to make development easier
  - New property based animation system prepared, but not yet active (LinkManager)
SCENEGRAPH TRAVERSAL COST

- **Memory cost**
  - Objects scattered in RAM
    - Latency when accessing an object
  - Objects are big
    - Traversing one object might touch multiple cache-lines

- **Instruction calling cost**
  ```c
  void processNode(Node *node) {
    // function call
    switch (node->getType()) {
      // branch misprediction
      case Group:
        handleGroup((Group*)node); // virtual function call
        break;
      case Transform:
        handleTransform((Transform*)node);
        break;
      case GeoNode:
        handleGeoNode((GeoNode*)node);
        break;
    }
  }
  ```

- **Transformation Cost**
  - Compute accumulated transformations during traversal

- **Hierarchy Cost**
  - Deep hierarchy adds 'needless' traversal cost (5/14 nodes in example of interest)
RENDERING PIPELINE

- SceneGraph
  - Scene abstraction, algorithms, loaders, savers, ...

- SceneTree (XBAR)
  - Scene Traversal

- Rendering Algorithm
  - Developers code with rendering algorithm
  - EffectFramework
    - Shader abstraction
  - RiX
    - OpenGL abstraction, hides VAB, UBO, bindless, …
SCENETREE REQUIREMENTS

- Generate on the fly from SceneGraph
- Incremental updates
  - Minimal amount of work on changes
- Caching mechanism per path
  - No recomputation of 'unchanged' values
- Flat list of GeoNodes
  - Get rid of traversal
- Memory efficient
  - Don't copy data, keep references
SCENETREE CONSTRUCTION

Event based updates

Flat List

S0  S1  S2  S1'  S2'
SCENETREE CONSTRUCTION
SCENETREE CONSTRUCTION

Event: Node Removed

Event: GeoNode Removed

Flat List

S0  S1
SCENETREE CONSTRUCTION

Event: Node added

Event: GeoNode added

Flat List
SCENETREE CONSTRUCTION

Event: Property Matrix
Transform changed

Event: Transform Changed (2x)

Construction:
S3032 Advanced Scenegraph Rendering Pipeline (GTC 2013)
The rendering pipeline involves:

- **SceneGraph**: Scene abstraction, algorithms, loaders, savers, ...
- **SceneTree (XBAR)**: Scene Traversal
- **RiX**: Developers code with rendering algorithm
- **EffectFramework**: Shader abstraction
- **OpenGL abstraction, hides VAB, UBO, bindless, ...**
SCENERENDERER

- Observe SceneTree to track GeoNodes in arrays
- `dp::sg::renderer::rix::gl` is 'example' renderer

<table>
<thead>
<tr>
<th>Render Scene</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update resources</td>
</tr>
<tr>
<td>Compute near/far plane</td>
</tr>
<tr>
<td>Frustum culling</td>
</tr>
<tr>
<td>Depth pass</td>
</tr>
<tr>
<td>Opaque pass</td>
</tr>
<tr>
<td>Transparent pass</td>
</tr>
</tbody>
</table>
**RENDERING PIPELINE**

- SceneGraph ➔ Scene abstraction, algorithms, loaders, savers,…
- SceneTree (XBAR) ➔ Scene Traversal
- Rendering Algorithm ➔ Developers code with rendering algorithm
- EffectFramework ➔ Shader abstraction
- RiX ➔ OpenGL abstraction, hides VAB, UBO, bindless, …
# ANATOMY OF A SHADER

<table>
<thead>
<tr>
<th>Shader Part</th>
<th>Source Code Example</th>
<th>Pipeline Module</th>
</tr>
</thead>
</table>
| **Version Header**            | // version header & extensions
#version 330
#extension GL_NV_shader_buffer_load : enable                                    | Renderer                      |
| **Uniforms**                  | // Uniforms
uniform struct Parameters{
    float parameter;
};                                                                              | Material description          |
| **Attributes**                | // vertex attributes (vertex shader)
layout(location = 0) in vec4 attrPosition;                                         | (Material description)        |
| **Shader Stage variables**    | in/out vec3 varPosition;                                                            | Hardcoded or generated        |
| **Library functions**         | Bsdf*(params);
determineMaterialColor();
determineNormal();                                                           | User provided to generator    |
| **User Implementation**       | void main()
{
    // some code
}                                                                 | Material description or rendering system |
## PARAMETER GROUPING

<table>
<thead>
<tr>
<th>EffectSpec</th>
<th>ParameterGroupSpecs</th>
<th>Binding Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Shader independent globals, i.e. camera</td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>Shader dependent globals, i.e. environment map</td>
<td>rare</td>
</tr>
<tr>
<td></td>
<td>Light, i.e. light sources and shadow maps</td>
<td>frequent</td>
</tr>
<tr>
<td></td>
<td>Material parameters without objects, i.e. float, int and bool</td>
<td>frequent</td>
</tr>
<tr>
<td></td>
<td>Material parameters with objects, i.e. textures and buffers</td>
<td>always</td>
</tr>
<tr>
<td></td>
<td>Object parameters, i.e. position/rotation/scaling</td>
<td>always</td>
</tr>
</tbody>
</table>
EFFECT FRAMEWORK GOALS

- Unique shader interface with support of multiple rendering APIs
- Code generation for different kind of parameter techniques, i.e.

Phong  Car paint  PBR

Uniforms  Uniform Buffer Objects  Shader Storage Buffer Objects  Shader Buffer Load  Other Graphics API
PARAMETER SHADER CODE GENERATION

**ParameterGroup phong_fs**

- `vec3` ambient
- `vec3` diffuse
- `vec3` specular
- `float` specularExp

**Uniforms**

- `uniform vec3 ambient;`
- `uniform vec3 diffuse;`
- `uniform vec3 specular;`
- `uniform float specularExp;`

**UBO**

```glsl
layout(std140)
uniform ubo_phong_fs {
    uniform vec3 ambient;
    uniform vec3 diffuse;
    uniform vec3 specular;
    uniform float specularExp;
} sys_phong_fs;
```

**shaderbufferload**

```glsl
struct sbl_phong_fs {
    uniform vec3 ambient;
    uniform vec3 diffuse;
    uniform vec3 specular;
    uniform float specularExp;
}

uniform sbl_phong_fs *sys_phong_fs;

#define ambient sys_phong_fs->ambient
#define diffuse sys_phong_fs->diffuse
#define specular sys_phong_fs->specular
#define specularExp sys_phong_fs->specularExp
```
RENDERING PIPELINE

- SceneGraph
  - Scene abstraction, algorithms, loaders, savers, ...
- SceneTree (XBAR)
  - Scene Traversal
- Rendering Algorithm
  - Developers code with rendering algorithm
- EffectFramework
  - Shader abstraction
- RiX
  - OpenGL abstraction, hides VAB, UBO, bindless, …
RIX

- Rendering API abstraction with OpenGL backend in place
- Hide implementation details which generate all kind of (OpenGL) streams

<table>
<thead>
<tr>
<th>Bindless</th>
<th>Vertex Attribute</th>
<th>Parameter Updates</th>
<th>Buffer Upload</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generic Attributes (2.1)</td>
<td>glUniform</td>
<td>glBufferSubData</td>
</tr>
<tr>
<td></td>
<td>Vertex Array Objects (VAO, 3.0)</td>
<td>glBufferSubData</td>
<td>Batched</td>
</tr>
<tr>
<td></td>
<td>Vertex Attrib Binding (VAB, 4.3)</td>
<td>glBufferAddressRangeNV</td>
<td>Persistent Mapped</td>
</tr>
<tr>
<td></td>
<td></td>
<td>glBindBufferRange</td>
<td></td>
</tr>
</tbody>
</table>
**Render Pipeline Using RIX**

- **Render Scene**
  - Depth Pass
  - Opaque Pass
  - Transparent Pass
  - Post-Processing

- **RenderGroup**
  - Depth
  - Opaque
  - Transparent

- **Same objects**

**RenderGroup per render pass**

- Rendering cache can be optimized for pass
- Depth-Pass might require only positions, but not normals and texture coordinates -> smaller cache
  - Fewer OpenGL calls than opaque pass with optimized cache
- Transparent pass might or might not require ordering
GeometryInstance can only be referenced by single RenderGroup
RENDER GROUP

RenderGroup

ProgramPipelineGroupCache

Group by Program

Sort by ContainerData
ProgramPipelineGroupCache<VertexCache, ParameterCache>

AttributeCacheEntry

GeometryInstanceCacheEntry
  'solid'
  'solid'
  'solid'

ContainerCacheEntry

std::vector<unsigned char> uniforms;
dp::gl::Buffer bufferData; // UBO, SSBO

dp::gl::Buffer bufferData; // UBO, SSBO
**BENCHMARK**

- GLUTAnimation
  - 100x100 Spheres
  - Geometry duplication
  - 5 different materials
  - Each sphere has own 'color'
## CPU TIME VERTEX TECHNIQUES

<table>
<thead>
<tr>
<th>Technique</th>
<th>Rendertime (ms)</th>
<th>Bindless</th>
<th>Bindless</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBO</td>
<td>5.7</td>
<td>1.8</td>
<td>2</td>
</tr>
<tr>
<td>VAB</td>
<td>4.9</td>
<td>1.6</td>
<td>1.8</td>
</tr>
<tr>
<td>VAO</td>
<td>7.5</td>
<td>3.2</td>
<td>3.2</td>
</tr>
<tr>
<td></td>
<td>1 stream</td>
<td>2 stream</td>
<td></td>
</tr>
</tbody>
</table>
CPU TIME PARAMETER TECHNIQUES

Time (ms)

- bindless bindRange / UBO: 1.6
- bindRange / UBO: 2.2
- bindRange / SSBO: 4.6
- subdata / ssbo: 10
- subdata / ubo: 3.1
- uniform: 5.4

glBindBufferRange Speedup

- future: 7.8
- r340: 4.3
- 332.21: 1.4
- 314.07: 1.0
PARAMETERS UPDATE HANDLING

- Each RenderGroup has a set of ContainerDatas
  - Map of containerData -> cache position (IMAGE)
- How to manage dirty state per RenderGroup efficient?
  - Set of ContainerData
CONTAINERDATA UPDATE HANDLING

- **First approach**
  - RenderGroup holds std::set<ContainerData> of dirty objects
  - std::map<ContainerData, CacheLocation> for ContainerData->CacheLocation mapping

- Profiling revealed this was a bad idea
  - Dirty phase
    - std::set::insert, top hotspot in GLUTAnimation
    - Binary search, allocation, large amount of ‘random memory access‘ ops
  - Update Phase
    - std::map<ContainerData*,CacheLocation>::find()
    - Binary search, ‘random memory access‘
CONTAINERDATA UPDATE HANDLING

- Second approach
  - Assign each Container a unique id, keep unique ids as dense as possible

![Diagram showing BitArray Dirty, CacheInfos, and Uniforms/UBOs for Containers 1, 2, ..., n.]

- BitArray Dirty
- CacheInfos
- Uniforms/UBOs

Unique Ids

Offset
CONTAINERDATA UPDATE HANDLING

- Update phase: Set bits in dirty array
- Process update phase: Get offset from CacheInfos Array
- Constant time operations

<table>
<thead>
<tr>
<th>BitArray Dirty</th>
<th>Container 1</th>
<th>Container 2</th>
<th>...</th>
<th>Container n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unique Ids

Offset
## RESULTS

<table>
<thead>
<tr>
<th></th>
<th>Time STL (ms)</th>
<th>Time BitArray (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do Updates</td>
<td>4.8</td>
<td>2.5</td>
</tr>
<tr>
<td>Process Update</td>
<td>4.0</td>
<td>0.9</td>
</tr>
<tr>
<td>Total Time</td>
<td>8.8</td>
<td>3.6</td>
</tr>
</tbody>
</table>

Profiler Hotspot:
- Event handling
- Cache update
BITARRAY::TRAVERSEBITS

- Linear memory -> cache efficient
- Works on size_t type, skips 32/64 bits if no bit is set in a element
- Uses ctz (count trailing zeroes) intrinsics
  - No branch misprediction issues on 01001101 pattern
- 1M bits need 122kb, ~0.4us traversal time if no bit set
- As comparison
  - Red-Black treenode has 3 ptrs and a color, at least
    - 64-bytes per node + payload
    - 1953 nodes need more memory than 1m bits
- BitTree would solve linear problem during traversal
SPARSE UBO/SSBO UPDATES

- Efficient algorithm to handle changed containers -> done
- Assuming thousands of Containers referencing UBOs are dirty
  - How to execute an efficient update?
  - One map/unmap call for the UBO?
    - No, too much data transfer between CPU and GPU
  - One mapRange/unmapRange per update?
    - No, mapRange/unmapRange create sync points
  - glBufferSubData?
    - If glBindBufferRange is being used it’ll be slow too!
SPARSE UBO/SSBO UPDATES

- **dp::gl::BufferUpdater**
  - Supports updates of any block-size which is a multiple of 16
  - Gathers all updates, uploads them as compact buffer and scatters on the GPU

<table>
<thead>
<tr>
<th>Data</th>
<th>Offset SSBO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>49152</td>
</tr>
<tr>
<td>4096</td>
<td>53280</td>
</tr>
<tr>
<td>7168</td>
<td>512</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time <code>glBufferSubData</code> (ms)</th>
<th>Time batched (ms)</th>
<th>100x speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>0.07</td>
<td></td>
</tr>
</tbody>
</table>
CULLING

- dp::culling abstract API for frustum culling
  - CPU & OpenGL compute backend

Culling Group

Object 1  Object 2  Object 3  Object 4  Object 5  Object 6

Axis Aligned Bounding Box  Transform  Payload
foreach (object : group) {
    isVisible = result->isVisible(object->culling);
    setVisible(object->rix, isVisible);
}

expensive 'query' and update call for each object

Solution: ResultObject. Cull(group, result, viewProjection);

<table>
<thead>
<tr>
<th>Old visibility</th>
<th>Object 1</th>
<th>Object 2</th>
<th>Object 3</th>
<th>Object 4</th>
<th>Object 5</th>
<th>Object 6</th>
<th>New visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

BitArray::TraverseBits on XOR result
RESULTS

- Scene traversal can be avoided for static scene parts
- Rendering time depends a lot on used OpenGL methods
  - VAB + glBindBufferRange UBO good, in combination with bindless best
- BitArrays can be a good tool to avoid maps/sets
- Try to batch small updates to GPU memory

- Still CPU bound?
  - S5135 - GPU-Driven Large Scene Rendering in OpenGL (Tue 16:00, LL21B)

- GPU bound?
  - S5291 - Slicing the Workload: Multi-GPU Rendering Approaches (Web 10:00, LL21B)
THANK YOU
MATAVENRATH@NVIDIA.COM
JOIN THE CONVERSATION
#GTC15  
http://github.com/nvpro-pipeline