Parallel Processing within a Host Application

Andy Lin, Joe Longson, Gene Lee, Noel Villegas
• Rig interaction speed is paramount to our animators (>24fps)

• Real-time feedback = more iterations = higher quality animation

• Artists work within multiple existing host applications (e.g. Maya, Houdini, Nuke)

GOAL

Seamless real-time feedback within host applications
Course Overview

1. Improve your viewport render performance (Nitro)
2. Cache character poses (RigCache)
3. Predict and pre-cache future poses (Parade)
Nitro
Viewport Renderer
Nitro Goals

1. Faster than 24fps responsiveness (goal 60-90fps for manipulation)

2. Smooth and seamless host integration (i.e. Maya/Houdini/etc.)

3. Adequate representative visual quality (allows artistic judgements)
Multi-threading
Friendly
Data Structures

• Safe/efficient for threaded traversal
• Lightweight to construct, implicitly cached + shared
• Natively instanced, Lazily constructed
• Performance + throughput first, generality second
• Tuned for most commonly used production data
Data Structure Tradeoffs

**Traditional Host DAG**
- Consistent structure
- Extensible representation
- Not easy to make thread-safe

**Nitro Shadow DAG**
- Separable structure
- Limited extensibility
- Designed to be Thread-safe
Data Types

Primary Structures

• Root Node: Entry point for DAG
• Group Node: Organization + threading fan-out
• Single-Instance: Single hero instance of geometry
• Instance-Table: Packed collection of geometry instances

Secondary Structures

• Node Iterator: Consistent search+traverse
• Transforms: Cached TRS+VIS+BBOX
• LOD: Lazy loaded + packed geometry
Geometry Packing

Traditional DAG

- Data tightly packed + reindexed
- Face/Curve boundaries retained
- Explicit sharing between samples
- Goal: Reduces number of draw calls
- This is the **Nitro Data API**

Nitro DAG
Instance Handle

Numerical Representation

<table>
<thead>
<tr>
<th>InstID</th>
<th>NodeID</th>
<th>PrimID</th>
<th>HierID</th>
</tr>
</thead>
</table>

128bit unsigned integer (vec4ui)

String Representation

“Set : Element : Instance# : InstanceCode : Mesh : Prim”

Selection Set

IH IH IH ...

- Root has list of all created Nodes (for fast hash-based lookup)
- Used by GPU-based picking & filtering
- Can be stored in EXRs (i.e. ObjectID render)
- Efficient conversion to/from string format
- Selection Sets for efficient draw-time filtering
Design Alternatives

• "Bucket of Polygons"
  • Single large table of instances. Easy to share and de-duplicate data
  • Drawback: Data update and mutex sync too expensive, not easily threadable.

• "Abstract Drawing API"
  • Easy to insulate/optimize drawing code and abstracts drawing from callers
  • Drawback: All caching delegated to callers, lots of duplicate effort to optimize, users had to learn a non-standard API
Load Balancing

- Threaded Traversal
- Asynchronous Loading
- Managing Threading Overhead
- Performance Gotchas
Threaded Traversal

- TBB Tasks provide parallelism
- Root is processed as a single task
- Fan-out when Group.children() > 1
- Additional Tasks within Instance-Tables
- Single-Instance/Instance-Table location chosen explicitly via heuristic
Asynchronous Loading

- When BBox hit, ask Loader for LOD
- Load can be (optionally) asynchronous
- When load complete, notifications sent
- Loader cache is the only mutex-locking code
- Logic generalized for any type of Loader (e.g. Geometry/Texture/Graph)
Managing Thread Overhead

- TBB tasks good for millions of tasks
- Tasks not really great for File IO, nor easy to cancel
- Avoid blocking/waiting on locks in tasks
- Utilize continuation tasks whenever possible
- Allow tasks to be re-used vs reallocated when possible
Performance Gotchas!

- Add performance counters
- TBB tasks + ref-counting
- Avoid non-data intensive tasks
- Caching + threading = challenging
- Revenge of the python GIL
Low-Level Optimizations

- Ptex texture atlas construction
- Image processing
- Surface tracking for fur
- Vectorization
- Lessons learned
Ptex Atlas Packing

Bottleneck

56% - Convert Ptex to Image
42% - Embed Pixels in Atlas
2% - Other

Convert Ptex - Solution

tbb::task* ConvertPtexTask::execute {
    tbb::parallel_for(faces, grainsize) {
        convertPtexToImage(...)
    }
}

Embed Pixels - Solution

tbb::parallel_for(faces, grainsize) {
    Image::embed(...)
}

5.8x improvement
Image Processing

OpenImageIO for Textures & FrameBuffers

- Abstracts IO logic
- Efficient memory packing
- Built-in threading support

Bottleneck

- OIIO uses threads vs. tasks
- Too high thread overhead (no benefit)

Solution

tbb::parallel_for(pixels, grainsize)
{
    image.getPixel(...)
    processPixel()
    image.setPixel(...)
}

2.4x improvement
Surface Tracking for Fur

**Bottleneck**

- **Surface Evaluation**
  (per-primitive position/orient evaluation needed)
- **Primitive Update**
  (compute delta from rest pose to deformed pose)

**Solution**

tbb::parallel_for(description, grainsize)

tbb::parallel_for(prim, grainsize) {
  evaluateSurface(prim)
  updatePrim(prim)
}
SSE Vectorization

Convert Matricies

- TRS/Camera as double, GL wants float
- Conversion needed with each draw() call
- Casting is not free!

```cpp
inline void convertDoubleMatrixToFloat( float result[18], const Matrix4<double>& mat )
{
    // NOTE: the SIMD Load operation transfer 2 doubles per operation,
    // while the SIMD store operation transfers 4 floats. Thus, we need an array
    // of size 18 in order to convert 16 double to 16 floats without data overrun.
    const double* tmp = mat.ptr();
    _mm_storeu_ps( &result[0], _mm_cvtpd_ps(_mm_loadu_pd(&tmp[0])) );
    _mm_storeu_ps( &result[2], _mm_cvtpd_ps(_mm_loadu_pd(&tmp[2])) );
    _mm_storeu_ps( &result[4], _mm_cvtpd_ps(_mm_loadu_pd(&tmp[4])) );
    _mm_storeu_ps( &result[6], _mm_cvtpd_ps(_mm_loadu_pd(&tmp[6])) );
    _mm_storeu_ps( &result[8], _mm_cvtpd_ps(_mm_loadu_pd(&tmp[8])) );
    _mm_storeu_ps( &result[10], _mm_cvtpd_ps(_mm_loadu_pd(&tmp[10])) );
    _mm_storeu_ps( &result[12], _mm_cvtpd_ps(_mm_loadu_pd(&tmp[12])) );
    _mm_storeu_ps( &result[14], _mm_cvtpd_ps(_mm_loadu_pd(&tmp[14])) );
}
```
Lessons Learned

• Simple Optimization Process
  (Guess - try - measure - repeat)

• Get to know and love your profilers
  (Used Intel’s VTune, Zoom, and perf-counters)

• Compile-time switches are your friends
  (Make sure you can turn optimizations off)
GPU Integration

- Avoiding a YAAPI (yet another API)
- Keep feeding the beast (i.e. the GPU)
- How to play nice with others
Nitro GPU APIs

- Avoids abstracting most GPU APIs
- Easy data packing for better performance
- Isolates serial nature of GPU draw submit
- Simple helpers common structures (e.g. Texture, Shader, Render Passes)
- Still allows full OpenGL API access (though most don’t need it)
Render Commands

- Commands built in DAG (threaded)
- Lock-free construction, mutex lock on submit
- Contains just enough information to draw
Render Engine

- While TBB Tasks traverse DAG ... (threaded build+submit command-streams)
- Engine serially processes commands (manages OpenGL state/memory)
- Shader handles OpenGL draw() calls
GPU Shaders

- **Statically allocated**
  (User overridable without performance impact)
- **FeatureSet for conditional compile**
  e.g. for shadows, gpu-picking, transparency, etc.
- **API callbacks for Render Engine (but no state)**
  activate/deactivate()
  preInstance/postInstance()
  preMesh/postMesh()
  draw()
- **Helper APIs to lookup/get/set GLSL uniforms**

This is the **Nitro GPU API**
Non ThreadSafe Processing

- Some DAG nodes aren’t safe for threading
- Push node reference to unsafe-queue
- Engine processes serially from unsafe-queue
- DAG node then submits to command-queue
- Can use a unsafe_thread (if supported)
Host Integration

• Each has their own rich plugin eco-system
• Everyone wants to exclusively own the GPU
• Nitro is fundamentally host agnostic
• Challenge: How to play nice with others
Key Areas of Integration

• Cameras
  • MVP, Viewport, Clipping Planes
• Lighting Conditions
  • Lights, ColorSpace (Lin/Mon)
• Material Properties
  • PBS, Phong/Blinn, Transparency, etc.

• Drawing State
  • Wire/Flat/Smooth Shaded
  • Lighting Enabled/Disabled
  • Two-sided Lighting, Shadows
  • Selected/Highlighted/Hidden
“Do No Harm”

• Most important principle!

• Leave the GL state as you found it (i.e. push/pop)

• Enable/disable GL state per need (i.e. don’t assume anything)

• Always use “Modern OpenGL” (i.e. avoid fixed-function)

• Also, strive for OpenGL core-profile
Selection

• Need a select_callback() from the host application
  x,y - position
  rx, ry - region of interest (i.e. marquee)

• Need to update the host application’s “Selection List”

• Support for ‘fit on select’
  TIP: Nitro uses two ‘components’ in Maya to allow fit on selection’s BBox
Maya Integration

- **Proxy**: MPxComponentShape (Maintains Nitro selection list)
- **Shader**: MPxHwShaderNode (Uses Nitro GPU APIs)
- **Maya Viewport 2.0 supported**
- **Python for Disney pipeline glue**
- **Custom Outliner for Nitro DAG**

**Challenges**

- Avoiding Maya traversals tricky (both in DG and EMP)
- Python+MEL in Rigs (can’t run MEL/Python in any thread other than main)
Houdini Integration

- Draw: `GR_Primitive/GUI_PrimitiveHook` (No legacy GL2 support)
- Select: `DM_MouseEventHook` (Maintains Nitro selection list)

Challenges
- Still some artifacts with double-draw below `/obj`
- Marquee selection logic challenging
- Issues with selection stability
RigCache
A Pose-Based Caching System
Introduction

• Rigs are Complex
  • 350 rig controls
  • 15,000 nodes
  • 50,000 connections

• Complexity RISING!
Problem

- **Slowness:**
  - evaluation
  - rendering
- **Preview:**
  - via playblast
Objective

Real-Time PLAYBACK (with Multiple Characters)

24 FPS
Complexity in DG Evaluation
Pose-Based Geometry Caching
Without Caching

Evaluation Engine

POSE

GEOMETRY
With Caching

- Rig Ctrl A
- Rig Ctrl B
- Rig Ctrl C

- Compute
- Compute
- Compute
- Compute

- PoseCache
  - Pose ID

- RigCache

- Nitro
Hashing a Pose

```cpp
static const double FX_PT_SCALE(4096.0);
namespace std {
    template<>
    struct hash<Pose> {

        size_t operator()(const Pose &pose) const
        {
            // NOTE: Similar poses that are not exactly identical due to floating point inexactness
            // will hash to the same value.
            // rigCode is a unique string to identify the rig
            uint64_t key(std::hash<std::string>()(pose.rigCode()));
            const size_t size = pose.size(); // number of rig control values for this pose
            for (size_t i=0; i<size; ++i) {
                const double val = pose[i]*FX_PT_SCALE; // account for small floating point inexactness
                key = boost::hash_combine(key, int64_t(val));
            }
            return key;
        }
    };
}
```
Hash Table

KEYS (Pose IDs)

VALUES (Geoms)
Populate hash table first time we encounter a pose (full compute)
Populate hash table first time we encounter a pose (full compute)
Populate hash table first time we encounter a pose (full compute)
Rig Cache → Nitro

Pull from hash table when we see a pose again (fast compute)
Time-Based Pose Caching

- **Problem:** Still need to compute the pose at every frame
- **Solution:** Cache the poses
## Playback Performance

<table>
<thead>
<tr>
<th>Caching Type</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Caching</td>
<td>1x</td>
</tr>
<tr>
<td>Geometry Caching</td>
<td>36x</td>
</tr>
<tr>
<td>Geom+Pose Caching</td>
<td>80x</td>
</tr>
</tbody>
</table>
Parade

A Distributed Cache Evaluation System
Initiate background worker sessions
FG Maya

Send per-frame poses to background

BG 1  BG 2  BG 3  BG 4
Evaluate per-frame mesh results
FG Maya

Populate the Rig Cache

BG 1  BG 2  BG 3  BG 4
Shared Memory

Status

Commands: C1 C2

Pose:

Buffers:

Tasks:

RigCache

Gathering Thread

Parade BG

BG Rig

FG

BG
Shared Memory

- boost::interprocess::managed_shared_memory
- BG creates a Shared Memory Object (named w/ BG’s pid)
- FG finds the Shared Memory Object by the BG’s pid
- Construct Named C++ Objects (e.g. std::vector) in Shared Memory
- Access to Objects in Shared Memory must be guarded by boost::interprocess::interprocess_mutex
Analysis

- Shared Memory Size: 1 GB per BG Maya
- Automatically Launch/Terminate BGs
- Automatically Load/Unload Rigs in BGs
- 600,000 Poses Evaluated in BGs per Day
- 150,000,000 Cache Hit per Day
Handling In-Shot Deformer