Advancements in V-Ray RT GPU

GTC 2015

Vladimir Koylazov
Blagovest Taskov

Overview

- GPU renderer system improvements
  - System for managing GPU buffers
  - Texture paging
  - CUDA code debugging on CPU
  - QMC sampling
- Additional features
  - Architectural visualization
    - Light cache improvements
    - Anisotropic reflections
  - Character rendering
    - Hair and hair material
    - Sub-surface scattering
    - Displacement and subdivision surfaces
    - UDIM textures
  - Games
    - Texture baking
- Research efforts
  - More efficient realtime cloud rendering
  - Run-time shader compilation
  - Improved acceleration structures for raytracing

System for managing GPU buffers

- Initially we only had a fixed set of buffers (for geometry, lights, materials, bitmaps)
- However, many features need specialized buffers:
  - Look-up tables for importance sampling (dome/rectangle lights, hair material)
  - Anisotropy needs tangent and binormal vectors on selected geometry
  - Remapping shaders (ramps, Bezier curves) need varying tables for the knots
- We implemented a system for managing arbitrary buffers
  - Somewhat similar to the CUDA runtime transparent memory transfers
    - We don’t use the CUDA runtime (just the driver API)
    - Manual, more coding needed, but works with OpenCL too
  - Lights, materials, textures, geometry can specify and upload arbitrary data for use by the respective GPU code
  - The system handles GPU buffer (de)allocation and data transfer at the appropriate time
  - The system can replace pointers to system memory in uploaded data structures with GPU addresses.

System for managing GPU buffers

We want to transfer this structure to the GPU:

```
struct Test {
        int count;
        float *numbers;
};
```

Our system provides a GPUDataPtr class that can be used instead:

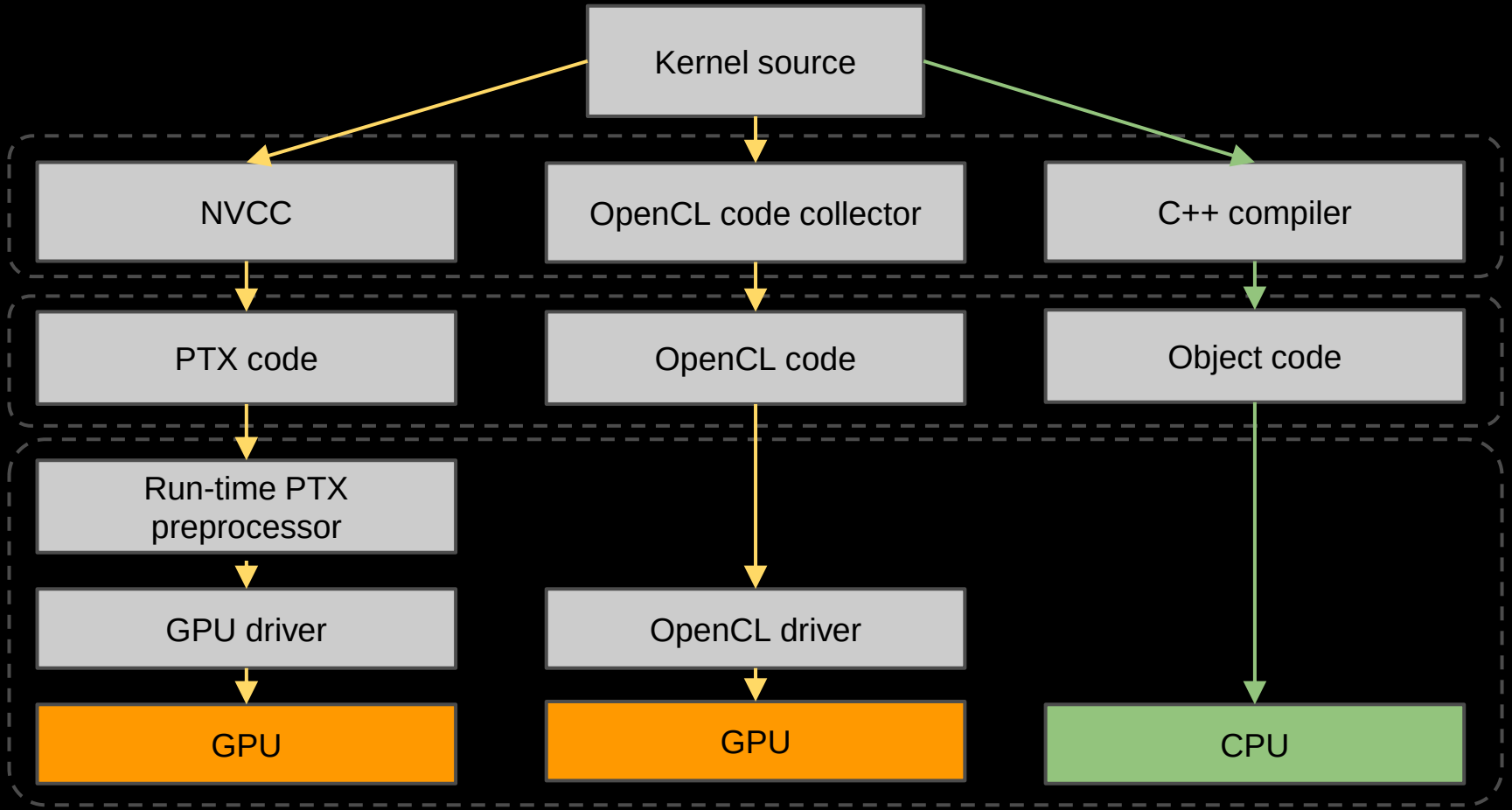
```
struct Test {
        int count;
        GPUDataPtr<float> numbers;
};
```

and provides methods for allocating, deallocating and accessing the numbers array on the CPU and the GPU when transferring the Test structure.

CUDA code debugging on CPU

- Debugging on the GPU is very difficult
  - Finding causes of NaNs in shaders
  - Invalid memory accesses
  - Logical/programming errors
  - NSight helps sometimes, but is slow and often doesn't find the issue
  - Ideally we want to debug CUDA code with the same ease that we have with the CPU code
- Our solution is to compile and run the CUDA code as regular C++ code that can be debugged with traditional tools
  - We already use the same code to target CUDA and OpenCL using #define statements, so we just had to extend these to C++
  - Some data types needed to be defined (float2, float4 etc)
  - This allows the CUDA code to be able to compile and link as regular C++ code
- We already have a generalized class for a compute device that has specializations for CUDA and OpenCL devices
  - We just have to implement a CPU device that executes the compiled CUDA code
- This allows us to execute and debug the code on the CPU
  - Too slow to be useful for anything other than debugging/testing purposes (10+ times slower)
  - Has enormously sped up our GPU development process
    - Syntax highlighting, intellisense
  - Useful for automated unit tests on machines that don't have GPUs

One source - multiple targets

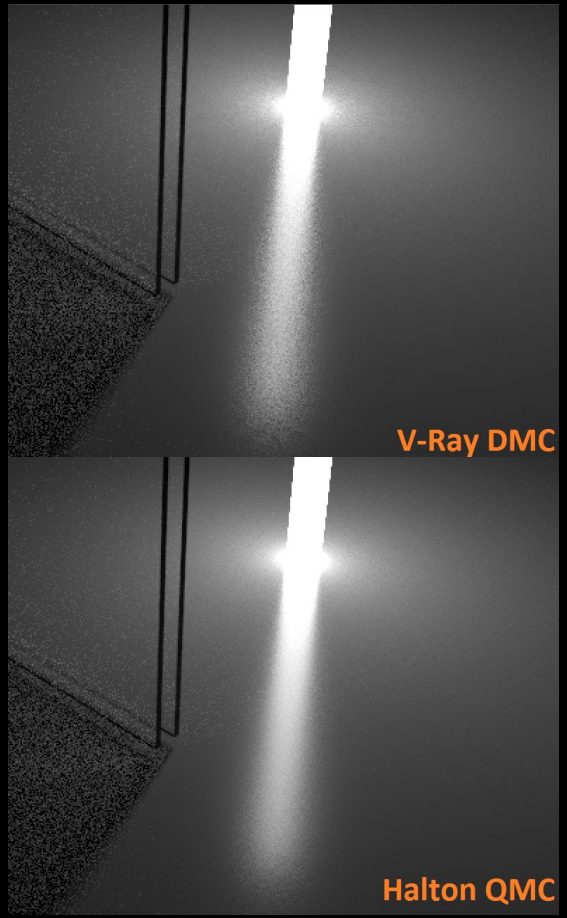


One source - multiple targets

```
_void setMatrix33FromMemory(Matrix33 *M, GLOBAL_CONST(float) ptr) {
    M->col1 = FLOAT4(ptr[0], ptr[1], ptr[2], 0.0f );
    M->col2 = FLOAT4(ptr[3], ptr[4], ptr[5], 0.0f );
    M->col3 = FLOAT4(ptr[6], ptr[7], ptr[8], 0.0f );
}
```

```
#ifdef __CUDACC__
#define _void __device__ void
#define GLOBAL_CONST(type) const type * __restrict
#define FLOAT4 make_float4
#else
#define _void void
#define GLOBAL_CONST(type) __global const type * restrict
#define FLOAT4 (float4)
#endif
```

QMC sampling



- QMC sampling is a way to distribute samples for AA, DOF, moblur, GI etc in an optimal way
- We licensed QMC sampling for V-Ray RT GPU when running on nVidia GPUs with CUDA
- Especially useful for V-Ray RT GPU because it relies on probabilistic Russian roulette sampling much more than the regular V-Ray renderer
- Noise is generally better and in some cases much better

Light cache improvements

- Modify the memory layout of the light cache to improve GPU performance
  - Before we used a strict binary KD tree for nearest lookups where each leaf contained exactly one point
  - Now the leaves can contain a small number of points that are tested in sequence
  - The points are rearranged so that the points in a single leaf occupy sequential addresses in memory
  - Bonus points: improved CPU rendering as well
- Support for motion-blurred geometry
  - Given an intersection point at an arbitrary moment of time, we need to figure out the position of the point at the start of the frame
- Support for hair geometry
  - Same implementation as on the CPU
  - The light cache now contains two types of points
    - Surface points have a normal and describe irradiance on a surface
      - Used for regular geometry
    - Volume points don’t have a normal and describe spherical irradiance at a point
      - Used for hair geometry
- Implement support for retracing if a hit is too close to a surface
  - Increases precision in corners and other areas where objects are close to each other
  - Reduces flickering in animations
  - Reduces light leaks in corners

Anisotropic reflections

- Anisotropic reflections require tangent and binormal vectors
  - Sometimes, a tangent vector is enough, if we assume that the tangent, the binormal and the normal are orthonormal
  - We prefer to compute and store both the tangent and the binormal explicitly
- To avoid discontinuities, those vectors need to be smooth over the surface
- For mesh objects, we compute the tangent and binormal vectors for each vertex based on a UV mapping channel
  - This is done similar to how smooth normals are computed
    - The vectors are computed for each face, and accumulated at each of the vertices for each face
    - In the end, the accumulated results for each vertex are normalized
- Memory concerns
  - Those vectors can take significant amounts of GPU RAM
  - It is impractical to compute and store them for each UV channel of every object
  - We want to compute and upload on the GPU tangent and binormal vectors only for objects that actually have anisotropic materials
- We modified our material descriptor parser to provide a list of UV channels that require tangent and binormal vectors
- Those vectors are only computed and uploaded for objects that have anisotropic materials

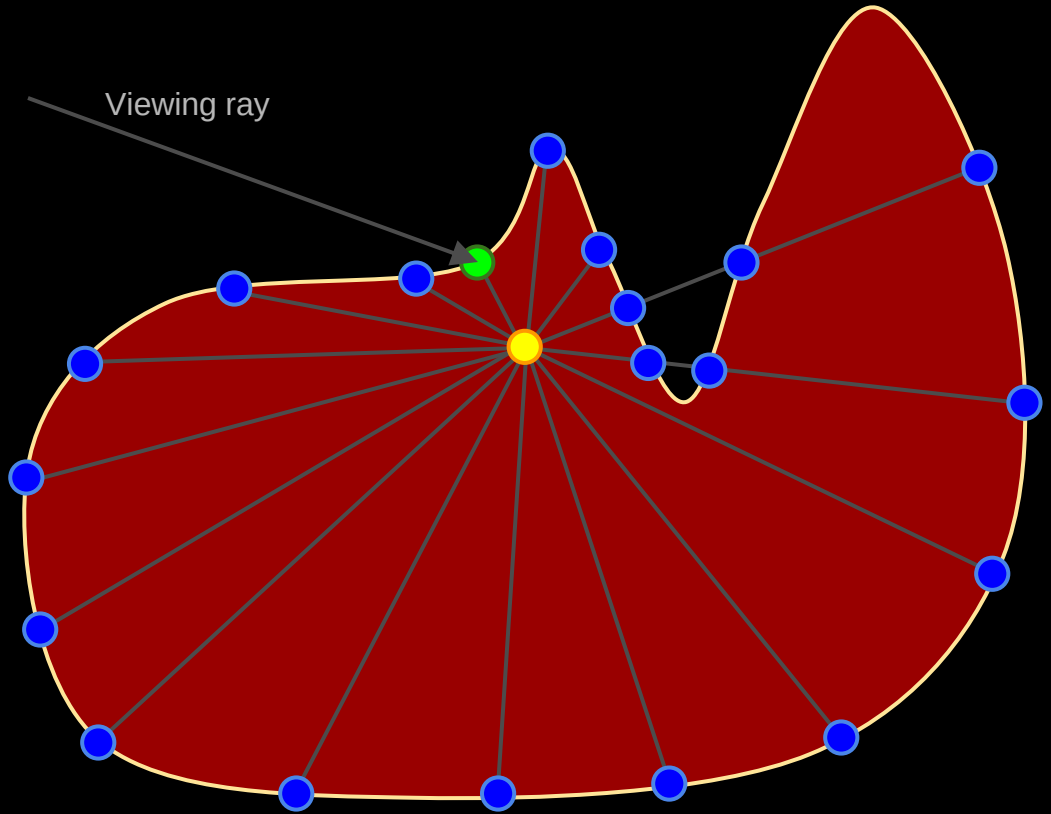
Hair and hair material

- Ray intersection with hair strands
  - Hair strands are represented as sequences of straight segments
    - View-dependent spline tessellation is used to smooth the curves as they get closer to the camera
  - We use the same KD trees for static and motion-blurred hair segments as we do in our CPU renderer
- Hair material
  - The hair shader is different from surface shaders as it can be illuminated from any direction
    - The code for lights had to be modified to take that into account
  - The hair shader uses look-up tables to generate directions for importance sampling
    - These must be uploaded on the GPU only if there are hair shaders in the scene
  - The light cache can be used to accelerate secondary GI bounces
  - Four-component shader model
    - Two specular components
    - A transmission component
    - A diffuse component

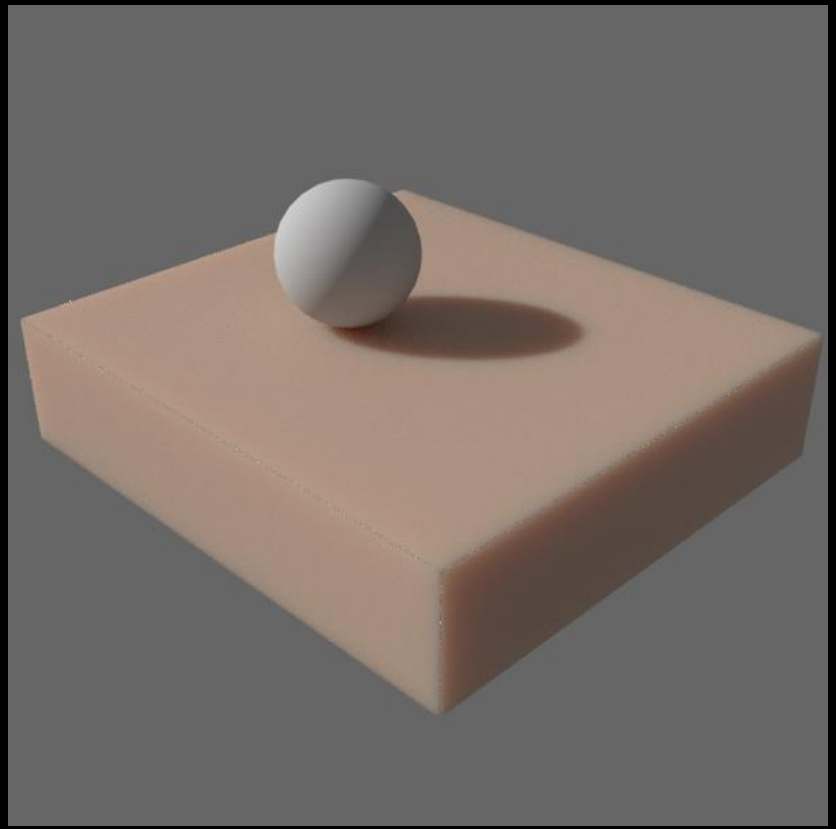
Sub-surface scattering

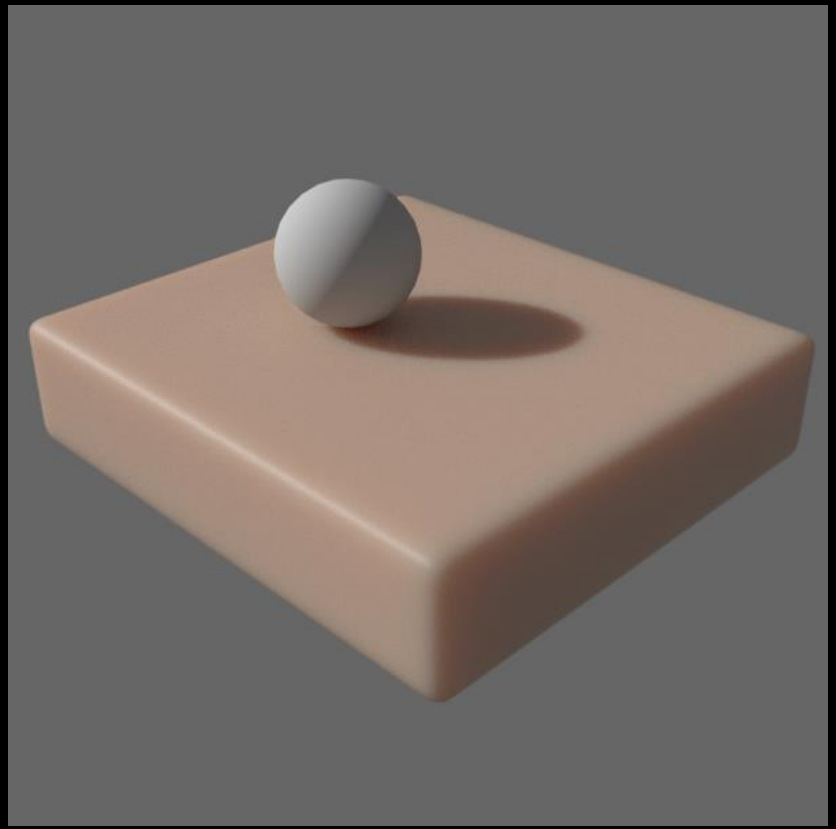
- Based on dipole BSSRDF model
  - Integrates the lighting over the entire surface of an object, convolved with a diffusion kernel
- On the CPU, we can use prepasses to precompute an illumination map at points on the surface
  - We can’t do that on the GPU - we can only use raytracing
  - We need to figure out a way to generate points on the surface only with raytracing
    - Preferably in such a way that the distribution approximates the diffusion kernel
    - We use spherical sampling starting with a point that is one mean free path below the surface
      - Works well for smooth surfaces; very noisy at sharp corners
      - Improvements pending
    - We sample the three color components separately and combine with MIS
- We need recursive calls to evaluate illumination (both direct and GI) at surface points
  - This is only possible in CUDA, so we don’t support SSS in OpenCL right now
  - Does not work with the texture paging system.
  - Needs to be reworked to remove the need for recursive calls.

SSS surface area sampling



SSS surface area sampling





SSS examples

Displacement and subdivision surfaces

- We already have a good view-dependent tessellator for the CPU renderer
  - We wanted to reuse as much code from that
- However, the CPU renderer generates geometry on the fly at render time
  - The tessellated result is never explicitly stored as a mesh
  - Some things are computed on the fly (tessellated UV coordinates, normals)
  - We didn’t want to do that for the GPU - instead all geometry is tessellated at the start of a frame
- We had to rework the tessellator to allow the explicit generation of tessellated geometry, UVs, normals
  - The result is a regular mesh that can be uploaded on the GPU as any other mesh
- The tessellator is view-dependent, but only with respect to the camera position when the mesh is generated
  - Interactive adjustments of the camera do not cause retessellation

UDIM textures

- Important for characters and large assets
- The CPU renderer can load textures on demand as it gets to them
- We can do that on the GPU only if texture paging is enabled
  - In that case, we use the regular bitmap manager to resolve, load and sample the correct tile based on the UV coordinates
- If texture paging is disabled
  - If a texture has a <UDIM> tag in the file name, we load all textures that match the UDIM template and make a map of the IDs of the individual textures for each UV tile; then we mark the texture as a UDIM texture
  - The GPU code checks if a bitmap is a UDIM texture and uses the UV coordinates and the UDIM tile map to read the correct texture

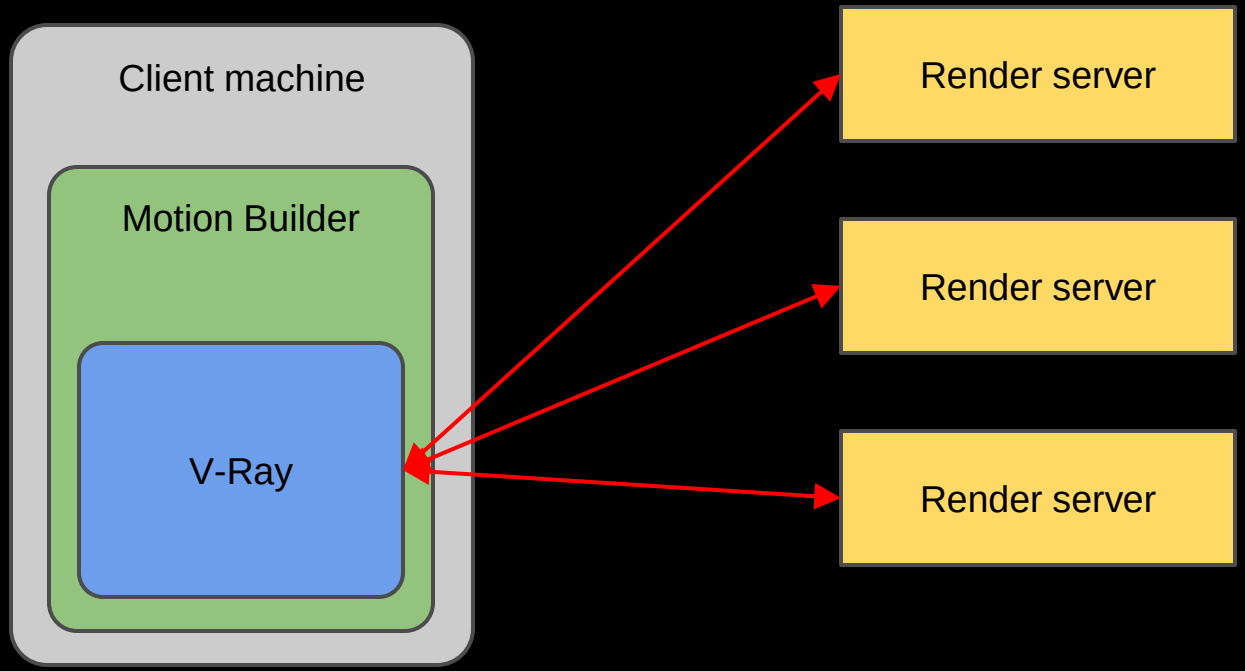
Texture baking

- Replaces the code for generating camera rays with generating a point on the surface of the baked object based on UVs
- We support render elements, so we can bake out different lighting components
  - Direct lighting
  - Global illumination
  - Diffuse texture filter
  - Normal maps
  - Other material components (reflectivity, transparency etc)

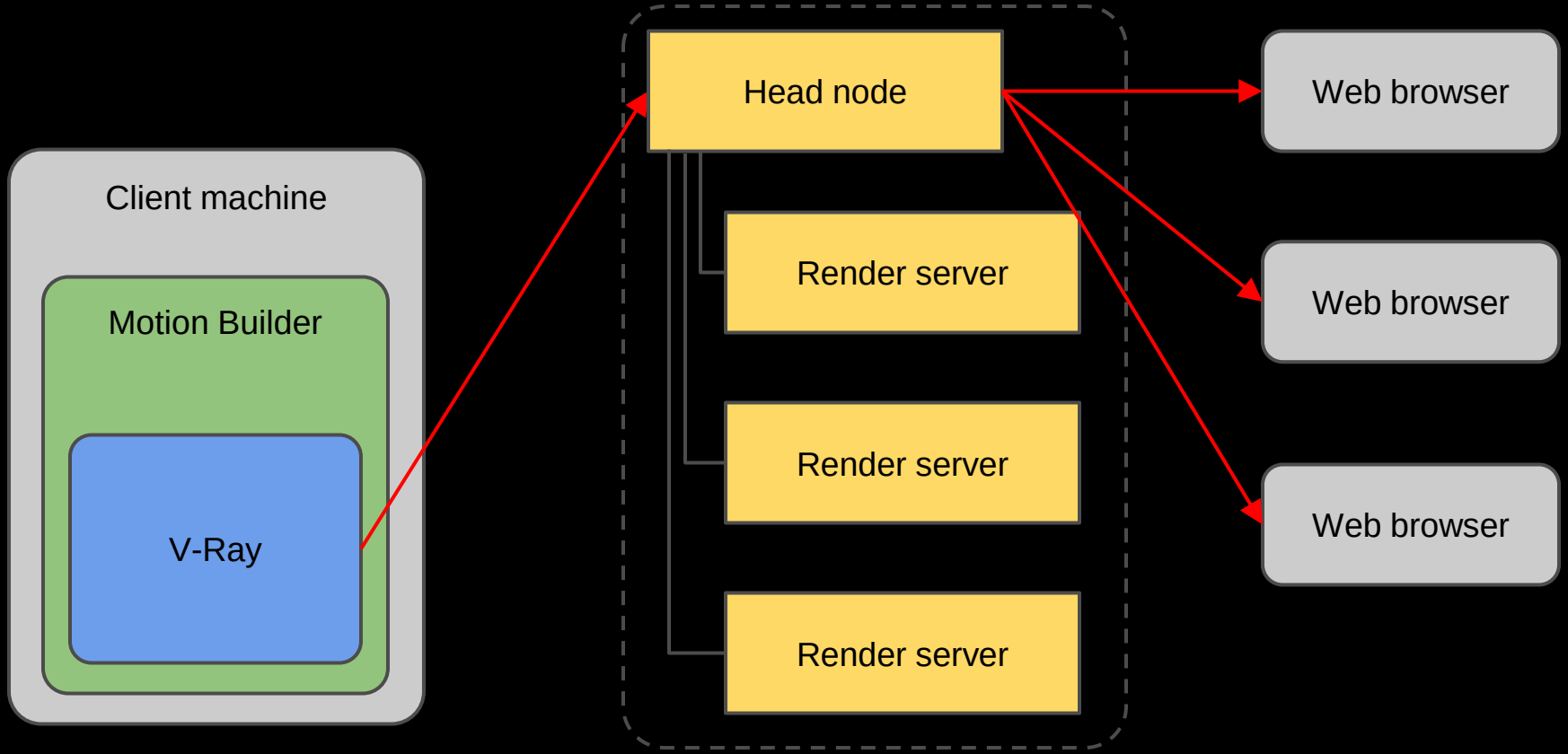
More efficient cloud rendering

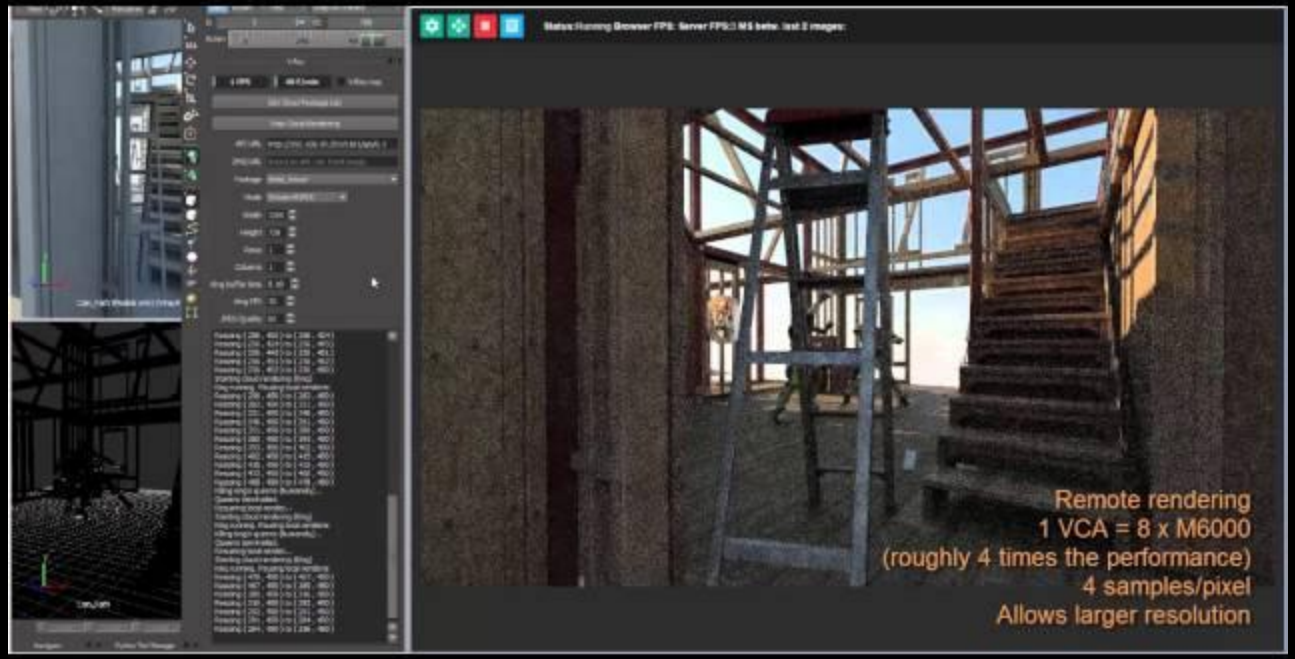
- Continuation of the research we did on running V-Ray inside Motion Builder
  - Higher resolution and better quality previews requires (way) more computing power
  - There is only a limited amount of GPUs that can be put into a single machine
  - We wanted to research the possibility of using a VCA render server or a cluster of VCA render servers
- V-Ray supports distributed rendering, however:
  - It is not well-suited for realtime rendering as each node directly communicates with the client machine
  - The required network traffic is huge and completely saturates Ethernet networks
- V-Ray Cloud on the other hand is specifically designed for rendering on a cluster of machines
  - We used the nVidia VCA cluster, where machines are interconnected with Infiniband
  - The idea is to hook up V-Ray for Motion Builder to the V-Ray Cloud system
- One machine in the cluster is designated as a head node
  - Accepts scene changes from V-Ray running inside Motion Builder
  - Distributes those changes to render servers in the cluster
  - Collects the rendered results and composes a final image
  - Includes a HTTP server that generates a stream of JPEG images viewable in any browser
- Results
  - Eventually works decently
  - Further research is needed to find optimal performance
  - At these speeds, we found many bottlenecks in our code that needed to be resolved

Regular V-Ray DR

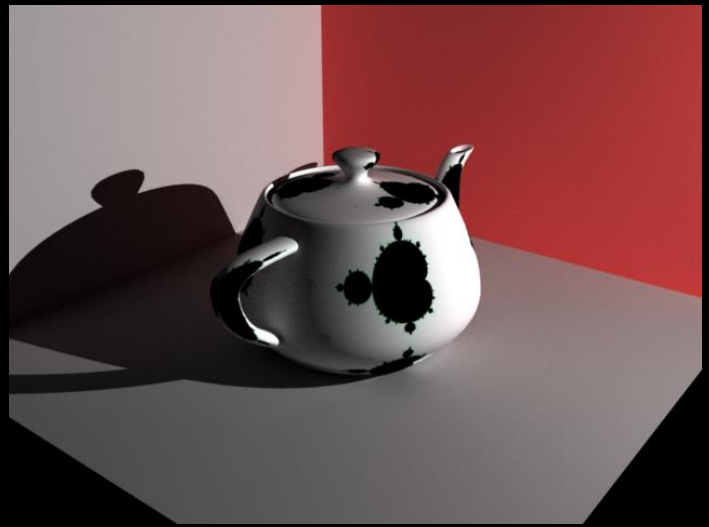


V-Ray DR through the V-Ray Cloud




Remote rendering
1 VCA = 8 x M6000
(roughly 4 times the performance)
4 samples/pixel
Allows larger resolution

Runtime shader compilation

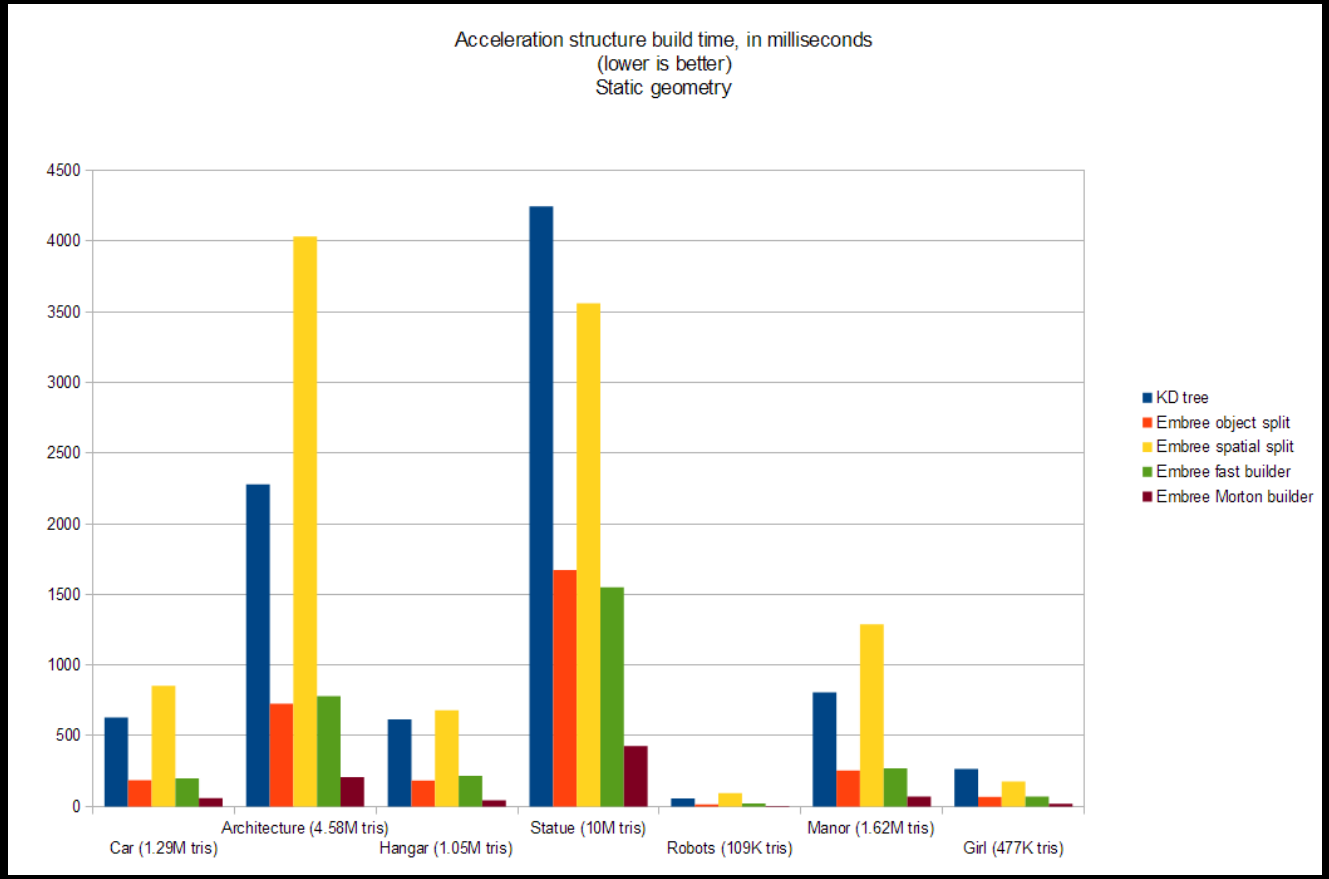


- Why this is interesting?
  - Custom shaders - users can write their own shaders that run on the GPU
  - Optimizations - compiling a shader network for a specific set of input parameters can eliminate dead code and improve performance
- Experiment 1: Through the V-Ray GLSL implementation
  - We have had support for GLSL shaders (with V-Ray extensions) in the CPU renderer for a while
    - Initially compiled down to byte code and interpreted by a virtual machine (GLVM)
    - Later on LLVM was hooked up to compile the byte code to x64 code
  - We used this work as foundation to generate PTX code and glue it into the mega kernel
    - LLVM has a PTX back-end which worked for this purpose
    - We already had in place a system that edits PTX code and passes it to the driver to generate the final mega kernel code
  - The advantage of this is that customers can write shaders in (V-Ray) GLSL directly and have then executed on the GPU
- Experiment 2: Through NVRTC.
  - Most of our shaders are not written in GLSL. Instead, material and texture plugins generate material descriptor code for the shading virtual machine in the mega kernel, which is more high-level than the GLVM byte code
  - CUDA 7 includes a library for run-time compilation of CUDA code (NVRTC)
  - We wrote a system that takes the material descriptor, generates the respective CUDA code for the shader (based on snippets of code from the GPU mega kernel itself), and produces PTX for the shader through NVRTC
  - The code is then glued into the mega kernel as with the GLSL version

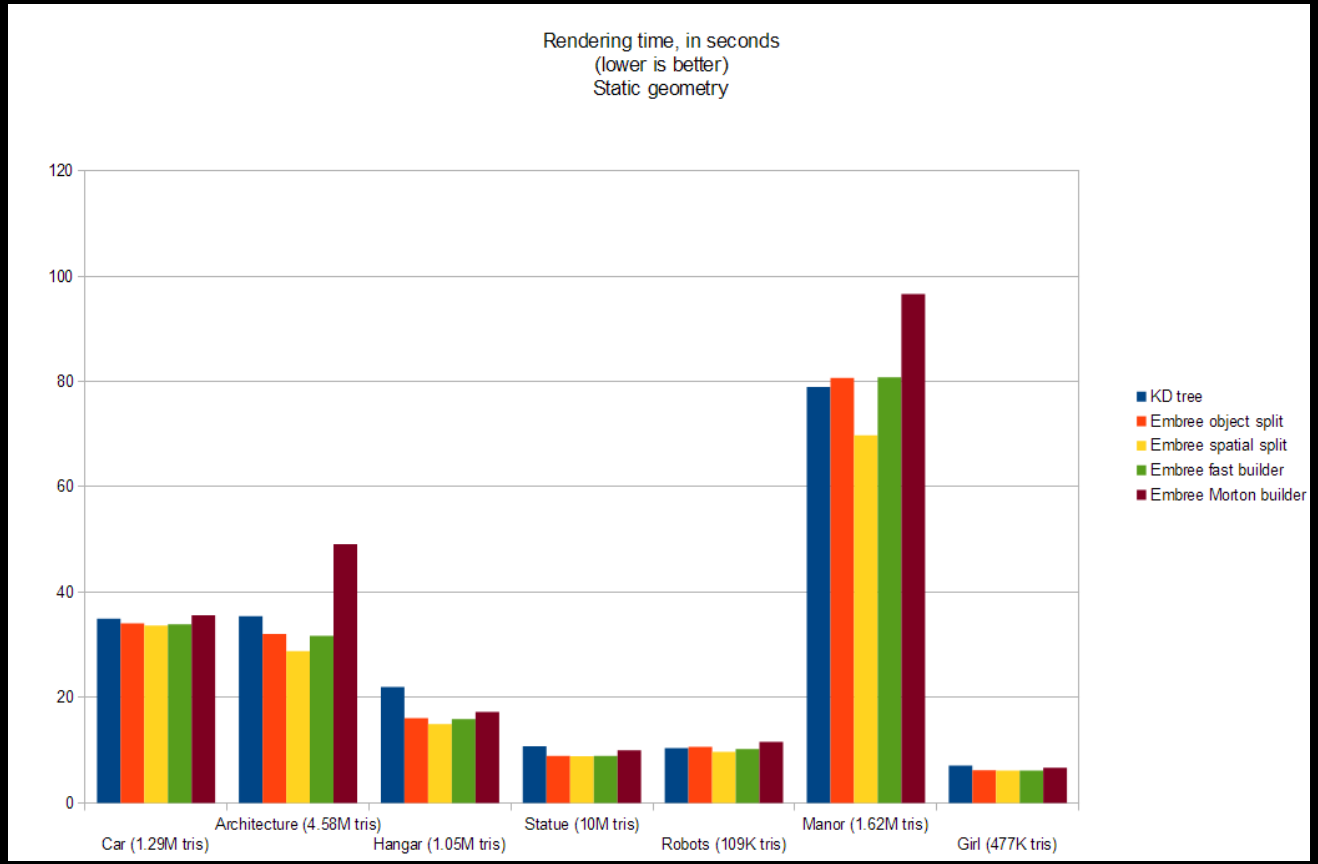
Improved raytracing acceleration

- Two goals:
  - To hopefully improve overall raytracing speed
  - To achieve faster updates for scenes with deforming geometry (as needed f.e. for skinned meshes from Motion Builder)
- Our original code for the GPU is using KD trees, because this is what we had in the CPU renderer
- More recent research seems to indicate that BVH approaches are more performant
  - We wanted to verify if this is still the case, and to what extent, both when intersecting rays and when building the BVH
- We already have Embree support in our CPU renderer
  - We modified the GPU code to upload the Embree BVH on the GPU
  - Wrote a traversal for the Embree tree on the GPU
    - The approach used on the CPU didn't work well because it relied on SSE2 instructions
    - It took a while to figure out an efficient way to do that on the GPU
- Results
  - Raytracing performance is varying
    - Static geometry performs on average 15% faster when using highest build quality
    - Dynamic geometry is about 5% slower even with highest quality builder (trees are smaller?)
  - Build performance is varying, from being the same speed for highest build quality, to 10 times faster when using the Morton builder (although in that case raytracing performance suffered).
  - More test are needed to determine if we should move to BVH completely.

KD tree vs BVH build times



KD tree vs BVH raytracing times on GPU



Continued with texture paging

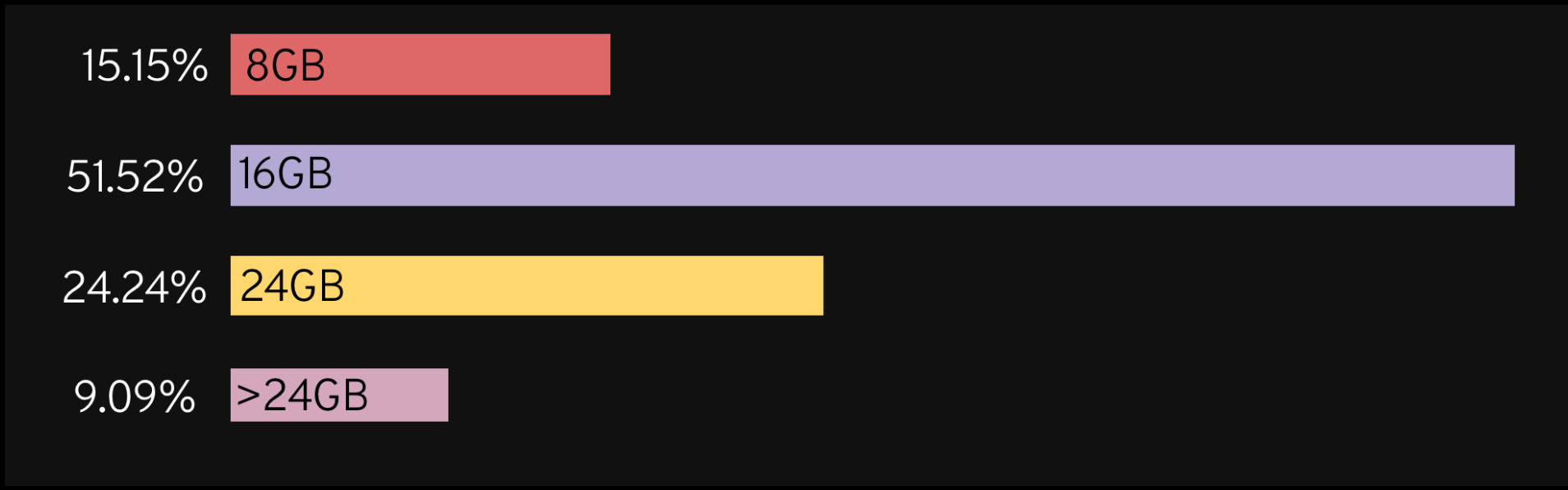
GPU memory today

- 4GB as a standard
- 12GB as a premium
- 24GB+ soon

How much does a scene take up?

Poll on ***forums.chaosgroup.com***

"HOW MUCH MEMORY WOULD BE ENOUGH?"



We already have that, so there is no problem. Good.

v-ray by CHAOSGROUP

… yes, but

The top 10% would like more memory

in fact

UNLIMITED

amount of memory

Hundreds of GBs ? … *"It is like falling off a cliff"*

We ask ourselves ...

"Can we make it more like falling down the stairs?"

STANDARD APROACH

- Multi kernel (implies slowdown).
  - *Megakernel + code morphing (with the register pressure) is still much faster than multikernel approaches (due to kernel call overhead, raystates being in the global memory, etc)*

- *Create LoDs & tiles.*

- *Trace and store tiles usage.*

- *Transfer tiles (update cache), trace, repeat.*

So what is wrong with that ?

CUSTOM MADE TEXTURES!

WITH 1X1 TEXTURE TILES, WE DO NOT USE THE BUILT-IN TEXTURE FUNCTIONS ANYWAY...

Here is how it works (forget everything you know about texture management)

APROACH #(N-1)

1. Intersect.
2. Check what ***PIXELS*** are needed.
3. Transfer only them (bump is tricky to do).
4. Shade.

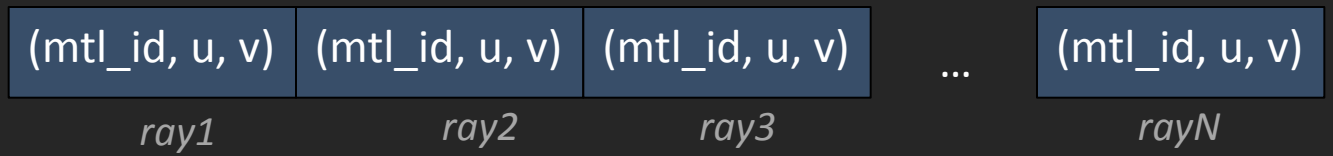
AND NOW, THE BEST PART...

The memory footprint for all needed pixels is <10MB.

We use **pinned memory** and **async transfers**!

The GPU is rendering one part of the scene, while it transfers what is going to be needed for the next.

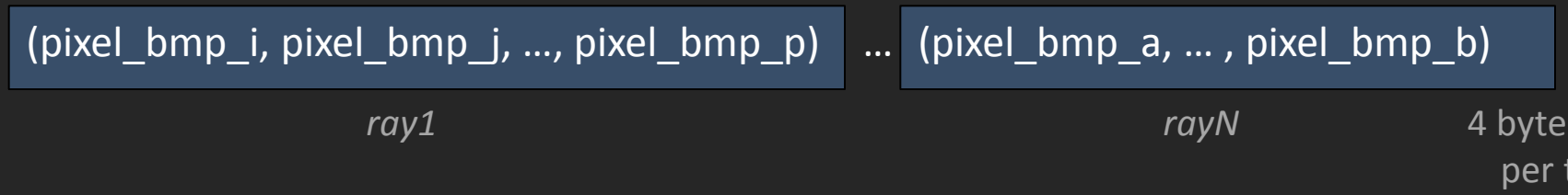
1. Call ***intersect*** kernel and fill the paging data in the format:



10 bytes per ray per map channel

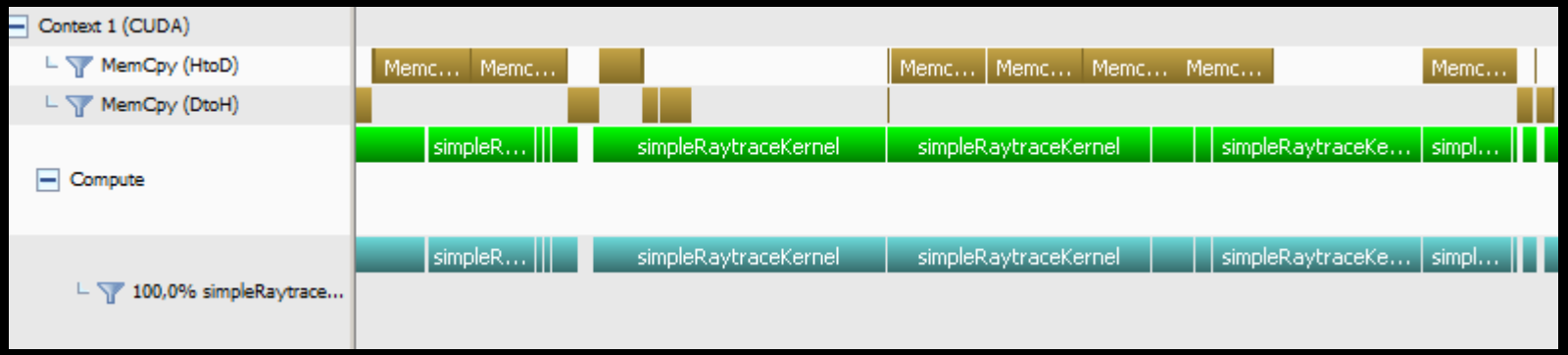
2. **Async** transfer this data to the CPU.

3. On the CPU, prepare pixel buffer in the format:



4 bytes per ray per texture

4. **Async** transfer that to the GPU.
5. Call **shade** kernel (all needed pixels are already transferred).
6. Repeat (the loop is guaranteed to be finite, since every ray advances with one bounce per *intersect* call).



Texture size does not slows down the rendering. You can **render terabytes**!

It is an abstraction, so the other parts of the rendering do not know about it.

It is so **awesome**!
And we gave it a name – ***"Pixel Streaming"***

Will work even better in the future with **nVlink**!

SO HOW MUCH SLOWER IS IT?

55s

149s

3X AT MOST*

**constant slowdown,*
compared to the megakernel!

If the data can fit into the caches, it goes up to just 1.3x times slower.

QUESTIONS?

email : blagovest.taskov@chaosgroup.com

** Please complete the Presenter Evaluation sent to you by email or through the GTC Mobile App. Your feedback is important!*