Accelerating Rendering of NURBS Surfaces by Using Hybrid Ray Tracing

Oliver Abert Markus Bröcker Rafael Spring

Institute for Computational Visualistics
University of Koblenz-Landau
Universitätsstraße 1
56070, Koblenz, Germany

abert@uni-koblenz.de mbroecker@uni-koblenz.de spring@uni-koblenz.de

ABSTRACT

In this paper we present a new method for accelerating ray tracing of scenes containing NURBS (Non Uniform Rational B-Spline) surfaces by exploiting the GPU's fast z-buffer rasterization for regular triangle meshes. In combination with a lightweight, memory efficient data organization this allows for fast calculation of primary ray intersections using a Newton Iteration based approach executed on the CPU. Since all employed shaders are kept simple the algorithm can profit from older graphics hardware as well. We investigate two different approaches, one initiating ray-surface intersections by referencing the surface through its child-triangles. The second approach references the surface directly and additionally delivers initial guesses, required for the Newton Iteration, using graphics hardware vector interpolation capabilities. Our approaches achieve a rendering acceleration of up to 95% for primary rays compared to full CPU ray tracing without compromising image quality.

Keywords: Ray Tracing, NURBS, Realtime Ray Tracing, Hybrid Ray Tracing, Object Intersection Buffer

1 INTRODUCTION

Plain ray tracing itself always has been very popular for realistic image synthesis. Recently, it even received more attention by scientists, as it is now possible to ray trace non-trivial scenes in real-time on a single commodity PC. Unfortunately, such ray tracing systems mostly work on data stored as simple triangle meshes. Typical drawbacks of triangular mesh representations are visible edges at surface silhouettes as well as very large primitive counts depending on the scene.

NURBS surfaces, however, have become the de facto standard in most CAD/CAM applications, where they are, for example, especially vital to the automobile industry. Virtual prototypes are nearly always developed using NURBS surfaces. However, rendering such data using common approaches, i.e., rasterization or standard ray tracing, requires the conversion into triangular meshes. On the one hand, such an offline conversion is often expensive and error prone, but on the other hand the rendering process afterwards is relatively fast. Nevertheless, it would be desirable to render NURBS data sets directly and fast. Recently, Abert et al. [AGM06] presented the ray tracing system *Augenblick* which is able to ray trace complex scenes consisting of NURBS surfaces with several frames per second. Obviously, ray tracing NURBS surfaces still can be considerably slower than ray tracing triangles.

In order to further reduce the speed gap, we extended the existing system with a hybrid ray tracing module. This module will speed up the NURBS tracing process by completely avoiding acceleration data structure intersection tests for primary rays. This enhances SIMD-coherence and additionally minimizes the total number of performed intersection tests (see table 6). Additionally, providing optimal starting values for the Newton Iteration, which is the core of the intersection algorithm (see [AGM06, PMS+99] for more details about the actual intersection test), will reduce the time for a single intersection test itself.

Hybrid solutions for ray tracing triangles (see section 2) already exist, however, the combination with NURBS surfaces is novel. Interestingly, hybrid ray tracing is more beneficial to NURBS scenes than to triangle based scenes. Since the NURBS intersection test is extremely expensive compared to simple triangle intersections, every unnecessary intersection test avoided is worth much more compared to classical triangle tests.

The underlying ray tracing system *Augenblick* is a cross platform application, optimized especially for massively parallel execution on both, SIMD units and multiple cores, delivering several frames per second on complex NURBS scenes on a single PC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WSCG'2008, February 4 – February 7, 2008
Plzen, Czech Republic.
Copyright UNION Agency – Science Press

2 RELATED WORK

Ray tracing as well as (general purpose) GPU development have been experiencing a lot of research since

 ISBN 978-80-86943-16-9

their introduction. In this section we shortly summarize other researchers' work that is closely related to our own.

2.1 Hybrid Ray Tracing

For conventional ray tracing it is mandatory to employ an hierarchical acceleration data structure. Such a data structure reduces the complexity for ray-scene intersection tests from $O(n)$ to $O(\log n)$, where n is the number of primitives. Nevertheless, a significant amount of computation time is spend for the traversal of this data structure (sometimes up to 70% of the total processing time). Another approach was introduced with hybrid ray tracing. Primary ray first-hit calculations can be replaced by rasterization using the graphics hardware. This was already described by Lamparter et al. in 1990 [LMW90] and by Paik in 1999 [Pai99]. Due to the enormous advances in the field of hardware-accelerated graphics, this shortcut works well even for scenes with high polygon count (though it is known that rasterization has $O(n)$ complexity). Thus most standard- or especially gaming-scenes do not have a primitive count critical to rasterization speed.

In 2005 Beck et al. [BBDF05] presented a GPU-CPU hybrid ray tracing framework targeting at interactive framerates for medium- and high-complexity scenes. They used the GPU to compute primary ray intersections along with shadow maps for simple shadows and standard Phong shading. Additionally, GPU and CPU rendering calculations were computed with a one-frame-offset, thus enabling parallel execution with minimal synchronization overhead. While they achieved good results for mostly diffuse scenes, their approach suffered from less efficient secondary ray computations on the CPU, resulting in sub-interactive rendering times for scenes with lots of reflective materials. Finally, only triangles were supported as geometric primitive.

Two years later, Reiter et al. [HSHH07] presented a kd-tree based ray tracer running entirely on the GPU. In contrast to [BBDF05], all rays (i.e., primary and secondary as well as shadow rays) were computed on the GPU. They were able to achieve nearly real time frame rates on an ATI X1900 XTX graphics board for non-trivial scenes. Limitations were inflicted by non-coherent secondary rays, which reduced the frame rate considerably. Again, only triangles were supported.

2.2 NURBS Ray Tracing

In general all ray-NURBS intersection algorithms work either iteratively or by subdivision, since a universal analytic solution does not exist.

The work of Kajiya [Kaj82] and Toth [Tot85] in 1982 and 1985 respectively, solved the problem generally. However, as they employed arithmetics with complex numbers in an iterative manner, their approaches were very slow, especially the latter one, and therefore unsuitable for real time applications.

Sweeney et al. [SB86] used a subdivision based technique to compute the intersection point. By refining the initial control point mesh successively, they were able to perform the intersection test with a triangle generated from such a refined part of the mesh.

A different method, called *Bézier Clipping*, was suggested by Nishita et al. in [NSK90]. By exploiting the convex hull property of Bézier surfaces, whole regions of the surface could be found that are known not to intersect the ray. Unfortunately, his approach can not be mapped to more complex B-Spline surfaces directly, since the Bézier convex hull property does not hold for them. [1] However, it would be possible to convert the NURBS surfaces to a Bézier representation and apply Bézier clipping for them. We do not consider this approach since high degree Bézier surfaces are very inefficient due to their global control property.

In contrast, Martin et al. [MCFS00] employed tiny bounding volumes covering very small regions of the parametric domain. For these regions, they successfully assumed that any enclosed value will suffice as an initial guess to start off the Newton Iteration. As their approach scales nicely with SIMD units, this approach was adopted for the *Augenblick* ray tracing system.

3 GENERAL APPROACH

In this section, we will shortly summarize the steps employed in our approach. We present two variants of our hybrid ray tracing technique. For convenience we call the first variant *ID processing* (i.e., storing surface ID numbers in the graphics buffer), while the second is called *u/v processing* (i.e., additionally storing uv parameters to start the Newton Iteration). We store triangle meshes as well as various kinds of free-form surfaces together in one scene while both primitive types benefit from the hybrid technique.

Standard ray tracing approaches, which do not employ the GPU for rendering, mostly operate on scene-data stored in main memory only. Because our system benefits from the high geometry throughput- and fill-rates of today's rasterization hardware, geometry data must be available to the GPU during rendering and thus has to be loaded into GPU memory as a step previous to rendering.

As all current rasterization hardware only supports triangles as geometric primitives, we cannot transfer free-form models to GPU memory directly, but must first triangulate every free-form model. Since this model-tessellation is a costly operation requiring sophisticated algorithms for good results, and due

[1] The stronger B-Spline k-point convex hull property can not be exploited that efficiently for this algorithm, though it might be an interesting option

to the fact that we are rendering rigid models only, these computations are performed offline during scene loading. Since triangulated models are just an approximation, this entails small errors in the resulting rasterization output and will be described in sections 5.3 in detail.

Note, that the original free-form data must not be discarded, since it is still required for the intersection test.

Along with the regular triangle meshes provided by scene data, we store the triangulation output in GPU memory using display lists. Additionally, we provide u/v parameter data (i.e., u/v NURBS surface parameters to start the Newton Iteration) for every triangle corner using regular texture coordinates. When choosing *u/v processing* this is necessary for further processing of the free-form surfaces in the rendering step.

During scene rendering, our approach operates on a per-frame basis. All triangle meshes, now available in GPU memory, are at first rasterized into two buffers using the multiple render targets capability of today's graphics boards. Since we use the GPU only for generating first-order hits and interpolating u/v vector data, all remaining computation steps are done by the CPU, using the buffer's contents. GPU buffer-content thus has to be transferred into main memory as an intermediate step.

Subsequentially, we process the acquired buffer data and calculate actual ray-primitive intersections. These operations highly depend on the employed variant: The *ID processing* method only uses the contents of the color-buffer for primary-hit-information. However, for intersecting rays with freeform-surfaces using Newton Iteration, additional parametric u/v vector-data must be provided for each ray-patch intersection. u/v parameters reference a point on the surface to serve as an approximate guess for the actual ray-patch intersection. This vector is thus acquired in an additional function-call when choosing *ID processing*.

Since we did already pass the parametric u/v data to the GPU using texture-coordinates, correctly interpolated u/v data is available for each pixel (or ray, respectively) in an additional buffer, too. The *u/v processing*-variant processes data from this buffer and passes the resulting intersection guesses to the Newton Iteration.

Note, that the tessellated models are used solely for accelerating the overall ray tracing process. Resulting images do not contain the tessellated models as rendered geometry. Hence, the goal of the tessellation process is to generate meshes that are most suitable for accelerating NURBS ray tracing rather than generating high-quality smooth meshes for direct screen rendering.

4 PREPROCESSING

In this section the required preprocessing steps are further outlined.

4.1 Free-Form Model Tessellation

Triangle-meshes can only serve as a linear approximation to continuously curved surfaces. Naturally, when it comes to tessellation, this involves a trade-off between tessellation accuracy and primitive count. This trade-off involves some more significant aspects in the context of our hybrid ray tracing approach. We will focus on these in this section.

Triangulating coarsely will obviously result in a smaller mesh size, which saves on overall memory resources. Of course, for rasterization hardware, less triangle transformations implies faster rendering, too. For the *ID processing*-mode, which processes every triangle in the rasterized image, a coarser mesh results in less cache misses, since rasterized triangles, once needed for intersection calculations, reside in L1 cache and can often be reused for following ray intersections.

However, choosing a too coarse tessellation does not allow for accurate and efficient ray-patch intersections. More accurate guesses lead to more efficient iterations (and thus ray-surface intersections), whereas inaccurate guesses cause a render time penalty or even iteration divergence leading to image artifacts. Since triangles can only store accurate u/v parameters on a per-vertex basis, the guesses will become too inaccurate when using extremely coarse meshes.

To avoid seam holes between surfaces, triangulation should be computed as a per-model operation rather than tessellating each surface individually. In section 5.3 we describe possible artifacts resulting from tessellation and our approaches to avoid these errors.

4.2 Data Preparation

Necessary data for the rasterization process is transfered to GPU memory upon scene loading. Each rigid model is stored in a display list containing geometry-, color- and u/v-coordinate-information. For both, triangle models as well as tessellated free-form models, the geometry information is passed straightforward as ordinary triangles.

The transferred triangle colors are dependent on the chosen variant: For *ID processing* (see figure 1) a triangle color represents the 32-bit triangle's main memory address for both regular triangle meshes or tessellated NURBS models.

For *u/v processing* (see figure 2) triangle colors for tessellated NURBS models refer directly to the memory address of the parent NURBS surface (the one it was tessellated from).

5 RUNTIME COMPUTATIONS

In this section we will give an in-depth description of the steps performed on a per-frame basis which differ from standard ray tracing.

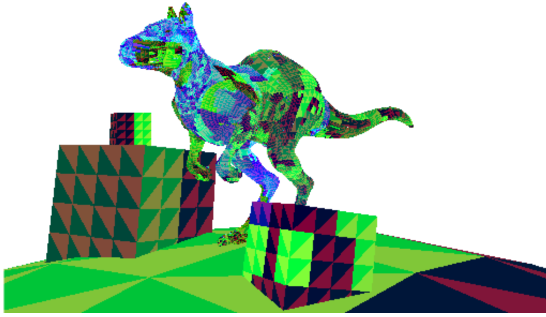

Figure 1: Rasterization output: The color buffer showing a scene with triangle-colors representing their address in main memory.

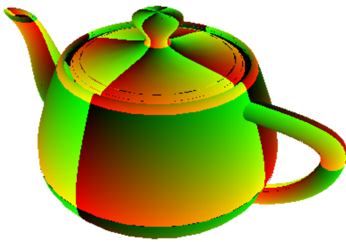

Figure 2: Rasterization output: The auxiliary buffer showing a tessellated teapot with colors which are u/v coordinate values, interpolated over the triangle's corners. U values are encoded in red, v values in green.

5.1 Scene Rasterization and Read-back

Acquiring primary-ray intersections through rasterization techniques is beneficial in two ways: First of all, costs for traversing the acceleration data structure are completely avoided. Furthermore, for each (non-background) pixel only a single intersection test is required, exploiting the z-buffer filled during rasterization. This way a significant number of intersection tests can be saved, as can be seen in section 6. Note, that in general no traversal scheme for bounding volume hierarchies can guarantee only a single intersection test per pixel.

In order to be able to separate the colors for both the object-reference and the parametric u/v-coordinates (if desired) into different buffers in one pass, we use the multiple render targets capability of today's graphics hardware. As mentioned before, rendering to more than a single buffer is only required for *u/v processing*. *ID processing* uses the standard, shaderless fixed-function-pipeline only.

However, to keep the overall process simple, our system uses a single shader, always rendering reference- and u/v-information into two buffers simultaneously. Rasterization speed is hardly affected, therefore it is not a limiting factor to overall processing speed. Likewise for sake of simplicity, u/v coordinates are always passed to the GPU, even if the underlying processing variant is not using them in the intersection step.

Camera and projection information for the rasterizer is simply drawn from the original scene data, since the ray traced and the rasterized scene need to be identical from the camera's point of view, obviously.

After the rasterization step, buffer data is read back into main memory. For both the intersection and the u/v coordinate-buffer we use data-aligned memory blocks for further SIMD-friendly processing.

5.2 Calculating ray-patch intersections

By processing the acquired buffer data, exact ray-primitive intersections are calculated:

As in standard ray tracing of primary rays, the image is rendered tile-wise using multiple threads, which exploits cache coherence and the power of modern computers with several cores per system. The tile size is variable, however we found a tile-size of 32x32 pixels to be the most efficient setting for the majority of scenes and hardware architectures. For each tile, four rays corresponding to four pixels in buffer-memory are processed simultaneously since all operations are performed using SIMD instructions.

The intersection-buffer stores 32-bit-wide RGBA color values, which represent memory addresses of the same width. Therefore, four references can be held in a SIMD-vector. The buffer that stores the interpolated u/v-values holds a 128-bit data word for each pixel. Two 32-bit words are used for floating point coordinate values of both parametric coordinates. 64-bit are wasted, however, we still found this to be the fastest solution, since a slightly higher memory consumption trades off well compared to costly data transformations.

From this point on, the algorithm basically consists of two successive steps of which the first differs depending on whether *ID-* or *u/v processing* is chosen.

Step one prepares all data for fast and unified intersection.

ID-processing The colors of the primary hit image reference triangle addresses in main memory. Triangles that were generated from free-form surface models have u/v information stored at each vertex. First, all of the packet's pixels acquire initial guesses for the Newton Iteration. This is implemented using polymorphic function calls. It immediately returns for regular triangles. In any other case it computes the average of the u/v values associated to the triangle vertices. Second, the four reference values taken from the intersection-buffer are replaced by the address values pointing to the parent surface of each referenced triangle. The parent surface for a regular triangle, for example, is the triangle itself, since it was provided by scene data itself. The parent surface for a triangle generated by tessellation is the surface it was tessellated from (see figure 3

 ISBN 978-80-86943-16-9

for an illustration of this). Processing surface data this way greatly enhances SIMD coherence: Because whole free-form surfaces (or their tessellated counterparts, respectively) are more likely to cover bigger chunks of space in the rasterized image, variance of references in a SIMD-vector is reduced. Often, four pixels in a block refer to the same patch which can then be intersected by at best four rays in parallel with only one call to the intersection routine.

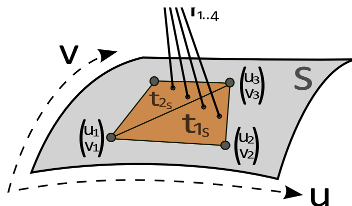

Figure 3: Triangles t_{is} with their parent surface s. u/v coordinate values are stored for each corner. A packet of primary rays $r_{1..4}$ hitting both of the triangles whereas the (also hit) parent surface s is the same for both triangles.

u/v-processing A 4x4 matrix containing four u/v values is read from the u/v-buffer and then transposed to acquire the initial guesses for a packet of four rays. Since with this variant, all tessellated parent surfaces are rasterized directly, colored with their 32-bit address, no further processing is necessary to obtain parent surfaces.

In step two the actual intersection is calculated. At this point, all values in our reference buffer are pointers to surfaces that can be intersected right away. The polymorphic intersection method is then called for each distinct reference in the SIMD-packet. Of course, any arbitrary intersection algorithm can be employed for the different geometric primitives. We employed the Möller-Trumbore algorithm [MT97] for triangles and the approach by Abert et al. [AGM06] for NURBS surfaces.

The overall algorithm used to calculate the exact primary hits of rays in a tile using the rasterization output is shown in Algorithm 1. Note that the $setupPacket()$ method is polymorphic and differs on the selected mode - either *ID Processing* or *u/v Processing*.

5.3 Artifact handling

Visible artifacts mainly originate from the tessellation of free-form surfaces. Since free-form surfaces are continuously shaped and triangle meshes are a piecewise linear approximation, there are always cases in which rasterized primary hits do not exactly match the ray-traced hits. If that was not true, there would have been no reason to ray trace the scene in the first place. Missed hits at convex surface boundaries do not impose a problem: if the background was drawn instead of a surface, as we can simply use standard ray tracing in that case. Concave surfaces are different: there might be cases in which a hit is indicated by rasterization while the actual free-form surface has a slight offset in one direction, like illustrated in figure 4. In this case, responding to the rasterization output, an intersection test is performed, resulting in iteration divergence. However, if another surface, lying behind, is actually hit by this ray, we must perform an intersection test for this surface but are not able to do this directly since rasterization output does not provide correct primary hit information for this ray. Therefore, as a second step, all rays that have both a hit indication by rasterization and an intersection fail are conventionally traced through the scene. This scheme also works for displaced patch borders through tessellation: If primary hits from rasterization involve ray-patch-intersection tests on the wrong patch, iteration will diverge and the corresponding rays are traced. As mentioned before, this scheme also handles very coarsely or noisy tessellated surfaces as well as displaced rasterization output. More accurate rasterization output gradually leads to shorter rendering times since less rays have to be traced through the hierarchy.

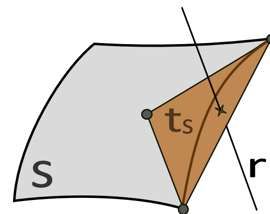

Figure 4: A primary ray r hitting a triangle t_s from a tessellated surface while missing the original (parent) surface s.

6 RESULTS

We implemented the algorithm using C++ and OpenGL with GLSL as a module for the existing real-time NURBS ray tracer *Augenblick*. Since *Augenblick* is cross-platform and currently supports OS X and Linux, OpenGL was chosen as hardware interface, although a DirectX implementation would be possible. However, DirectX is not an option for non-Windows systems.

The difference in speed between normal and hybrid ray tracing (see figure 7 for an example) depends on the complexity of the scenes and on the number of cores/threads used. Because our hybrid approach still involves additional constant buffer-readback times, highly parallelized standard ray tracing on multiple cores still begins to outperform our hybrid approach at some point (compare table 6). These additional readback times become most evident when using *u/v-Processing*. The cost of reading two buffers back into main memory (instead of one for *ID-Processing*) is

Algorithm 1: Primary Intersection Algorithm

```
Method processTile ()
  Data: tile : Tile, idBuffer : IDBuffer
  foreach RayPacket rays in tile do
    HitPacket hits = rays.getHits ();
    PixelPacket p = idBuffer.getPacket (rays);
    computeHits (rays, hits, p);
end Method

Method computeHits (RayPacket rays, HitPacket hits, PixelPacket p)
  if containsBackgroundColor (p) then
    trace (rays, hits);
  else
    GuessesPacket g = p.setupPacket ();
    foreach distinct reference r in p do
      r.intersect (rays, hits, g);
    RayPacket failedRays = raysContainingFailedIntersection (rays);
    if not isEmpty (failedRays) then
      trace (failedRays, hits);
end Method

Method PixelPacket::setupPacket () - Approach 1
  GuessesPacket g;
  forall distinct references r in this do
    g.set (r.acquireGuesses ());
    r = r.getParentSurface ();
  return g
end Method

Method PixelPacket::setupPacket () - Approach 2
  Data: uvBuffer
  GuessesPacket g;
  UVBufferData uvd = uvBuffer.getNextPacket ();
  uvd.transpose ();
  g[U] = uvd[U];
  g[V] = uvd[V];
  return g
end Method
```

evident in table 6, as the framerate drops about almost one third in many cases.

As a consequence of the slight shape distortion done by tessellation there also might be artifacts due to geometric inaccuracies (leading to intersection calculations on wrong patches) which our artifact handling cannot catch. This might happen especially when surfaces with very coarse tessellation intersect (as shown in figure 6). An exact scheme to catch these types of artifacts seems infeasible and would require additional ray tracing on (self-)intersecting objects to determine correct z-locations. However, almost all of these artifacts are avoided when tessellating not too coarsely.

The presented algorithm will work only on 32-bit systems, as the pointer size must fit into a 32-bit color. Extending the presented technique to support 64-bit pointers would be possible, assuming graphics hardware that can write to multiple buffers. A single memory address could then be split into two buffers. However, we found such an extension to be not reasonable at this point of time, since scenes with triangle-data exceeding 4 GB can neither be rasterized easily nor fast even using latest techniques and high-end graphics processors. In such a case direct ray tracing would be the more efficient choice in the first place.

During tests with float values, we found that 32-Bit values passed to the graphics hardware and then to be read back are not guaranteed to arrive "untouched". Naturally, bad pointer data for the intersection test

Statistics	Easy	Cornell Box	Teapot	Killeroo	Mercedes
NURBS Surfaces	1	18	32	90	1212
Triangles	112	576	16792	170712	1389582
Screen Coverage	27%	100%	21%	14%	18%
BVH Boxes	81	445	13018	131006	1012680
Intersection count (standard)	33800	92239	64186	89121	165551
Intersection count (hybrid)	27317	69006	28342	32750	48972
Frames/Second (standard, 1 thread)	7.2	3.1	5.2	3.3	2.1
Frames/Second (standard, 8 threads)	45.2	22.2	34.2	22.4	15.1
Frames/Second (hybrid, 1 thread)	7.9	4.0	8.0	5.9	4.1
Frames/Second (hybrid, 8 threads, *ID*-processing)	42.9	25.5	41.7	30.8	14.3
Frames/Second (hybrid, 8 threads, *u/v*-processing)	27.5	19.5	25.5	22.1	12.0

Table 1: Rendering times table. The FPS are the computed average over 5 frames and rounded to the first decimal place. The intersection count gives the number of calls to the NURBS intersection method. All tests were run on a Dual 3.0 GHz quad-core Apple Mac Pro with 2 GB RAM and an ATI X1900 XT graphics board. The rendered image has a size of 512x512 pixels.

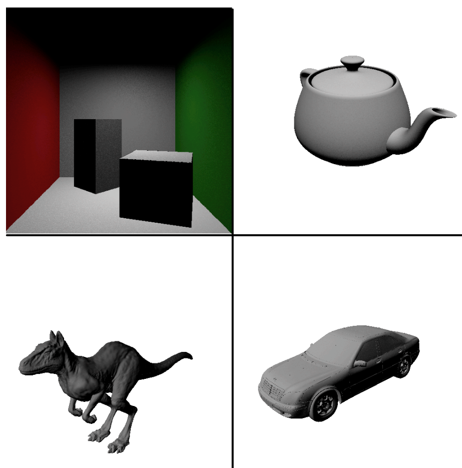

Figure 5: From left to right and top to bottom: Cornell, Teapot, Killeroo, Mercedes

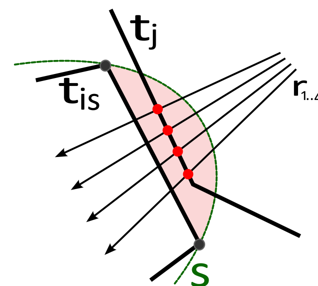

Figure 6: A packet of rays $r_{1..4}$ intersecting the wrong surface t_j. The altered shape t_{is} of the surface s causes objects in the shaded area to be drawn instead of culled and vice versa (if rays origin in the opposite direction).

will cause severe problems, one of which most likely a crash of the application. Fortunately, since we pass 32-bit references as an RGBA color value and graphics hardware uses 128-bit to store color values, our system is robust against falsification through internal floating-point-conversions on bus transfers. Thus we did not encounter any system crashes on erroneous pointer data. Regardless of this, we can, of course, never guarantee an error free bus transfer on all hardware layouts using float values.
Using the OpenGL integer texture extension which allows for direct writing of unsigned integers to the graphics buffer has not been tested yet. This could overcome possible corruption of memory addresses using float values while also using a smaller bandwidth to transfer data to the graphics hardware.

Rasterizing speed, which will carry more weight for very high primitive counts, could be further improved by using techniques such as view frustum culling or occlusion culling. To take full advantage of this, one must not upload and reference objects to the GPU as a whole, but rather piecewise. A bounding volume hierarchy for ray tracing already exists and could be reused for visibility culling purposes. This is not yet implemented in our system, since the rendering time for rasterization has not presented itself as a bottleneck, yet.

However, one of the biggest bottlenecks of our approach lies in memory readbacks from the graphics hardware to the main memory. At least one buffer must be readout. If using u/v coordinate interpolation this will be two buffers. Additionally readback speed can be compromised by CPU-sided data-conversion, when the employed graphics board does not support hardware-sided data-conversion.

Shifting GPU and CPU calculations by one frame and thus eliminating data dependencies (and idle-times) on the CPU as in [BBDF05] would certainly be a good

 ISBN 978-80-86943-16-9

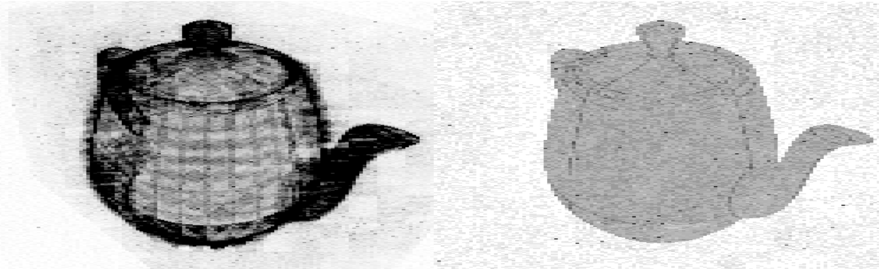

Figure 7: Time comparison for primary scene-intersection. The color of a pixel shows the time it took to compute the shaded result. A darker color means a longer ray tracing time. Left: standard ray tracing. Right: our hybrid approach using u/v-Processing.

solution to that problem. However, as recent developments in CPU- and GPU evolution show, the CPU and GPU will become much more tightly integrated in future systems which might have a major positive impact on bus-transfer- and readback-times, too.

Of course, our system only accelerates object-intersection with primary rays. All generated secondary rays, as needed for shadows or reflective materials, must be traced through the scene with standard traversal schemes.

7 CONCLUSION AND FUTURE WORK

We have presented a simple and efficient scheme to speed up the generation of first-order hits in realtime ray tracing of NURBS surfaces. The presented algorithm also accelerates conventional ray tracing of triangle-data at no additional cost.

Since GPU readback times are still a major speed issue, extending the scheme to support data-independent, frame shifted rendering as in [BBDF05] can certainly be regarded useful with respect to rendering time.
Another consideration applies to running the costly NURBS intersections on GPU.

This could be done by shifting both intersections and the overall ray-scene traversal onto the GPU as in [HSHH07]. In contrast to their work, a BVH traversal scheme would have to be employed. Obviously, this is only necessary if secondary rays are traced.

Finally, as a third research topic, all shading computations could be done on the GPU for triangle data and then be remapped onto the correctly intersected NURBS surface using interpolated u/v coordinates. This would completely avoid the need for an acceleration data structure. Of course, shading techniques specific to ray tracing, such as shadows or photon mapping, would not be available then.

REFERENCES

[AGM06] Oliver Abert, Markus Geimer, and Stefan Müller. Direct and Fast Ray Tracing of NURBS Surfaces. *IEEE Symposium on Interactive Ray Tracing 2006*, pages 161–168, September 2006.

[BBDF05] S. Beck, A.-C. Bernstein, D. Danch, and B. Fröhlich. CPU-GPU Hybrid Real Time Ray Tracing Framework. 2005.

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM.

[Kaj82] James T. Kajiya. Ray tracing parametric surfaces. *Computer Graphics (Proceedings of SIGGRAPH 82)*, 16(3):245–254, July 1982.

[LMW90] Bernd Lamparter, Heinrich Mueller, and Jörg Winckler. The ray-z-buffer-an approach for ray tracing arbitrarily large scenes. In *Eurographic seminars; tutorials and perspectives in computer graphics on Advances in computer graphics hardware*, pages 141 – 145, 1990.

[MCFS00] William Martin, Elaine Cohen, Russell Fish, and Peter Shirley. Practical ray tracing of trimmed NURBS surfaces. *journal of graphics tools*, 5(1):27–52, 2000.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray/triangle intersection. In *Journal of graphic tools*, pages 21–28, 1997.

[NSK90] Tomoyuki Nishita, Thomas W. Sederberg, and Masanori Kakimoto. Ray tracing trimmed rational surface patches. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):337–345, August 1990.

[Pai99] Samuel Paik. Z-buffer and raytracing. *Ray Tracing News Guide*, 1999.

[PMS+99] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. In *Proceedings of ACM Symposium on Interactive 3D Graphics*, pages 119–126, April 1999.

[SB86] Michael Sweeney and Richard Bartels. Ray tracing free-form B-spline surfaces. *IEEE Computer Graphics and Applications*, 6(2):41–49, February 1986.

[Tot85] Daniel L. Toth. On ray tracing parametric surfaces. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):171–179, July 1985.