Volume 0 (1981), Number 0 pp. 1–8

CPU-GPU Hybrid Real Time Ray Tracing Framework

S.Beck , A.-C. Bernstein , D. Danch and B. Fröhlich

Lehrstuhl für Systeme der Virtuellen Realität, Bauhaus-Universität Weimar, Germany

Abstract

We present a new method in rendering complex 3D scenes at reasonable frame-rates targeting on Global Illumination as provided by a Ray Tracing algorithm. Our approach is based on some new ideas on how to combine a CPU-based fast Ray Tracing algorithm with the capabilities of todays programmable GPUs and its powerful feed-forward-rendering algorithm. We call this approach CPU-GPU Hybrid Real Time Ray Tracing Framework. As we will show, a systematic analysis of the generations of rays of a Ray Tracer leads to different render-passes which map either to the GPU or to the CPU. Indeed all camera rays can be processed on the graphics card, and hardware accelerated shadow mapping can be used as a pre-step in calculating precise shadow boundaries within a Ray Tracer. Our arrangement of the resulting five specialized render-passes combines a fast Ray Tracer located on a multi-processing CPU with the capabilites of a modern graphics card in a new way and might be a starting point for further research.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Ray Tracing, Global Illumination, OpenGL, Hybrid CPU GPU

1. Introduction

Within the last years prospects in computer-graphics are growing and the aim of high-quality rendering and naturalism enters into many disciplines of graphics such as Animation, Film, Games and even Virtual Reality. In a few years reality and virtual reality might be less and less distinguishable. Therefore new methods are introduced and invented in the context of Render Engines for Games and Virtual Reality Systems.

Ray Tracing is one of the oldest and most promising methods to render 3D scenes with respect to Global Illumination in terms of physically correctness and therefore will always be an actual research topic. It is a simple but powerful model for generating shadow, reflection and refraction and even more phenomena like Motion Blur , Depth of Field and others. With respect to naturalism Ray Tracing even is a kind of overkill in simulating lights and its specular reflection, depending on the lighting model, and the depth of recursion when computing reflection on a surface. On the one hand Ray Tracing with its branched algorithm is inefficient and the computational time increases in a non-linear behavior when adding complexity. The lack of hardware acceleration therefore inspired Wald et.al to design a prototype and the results are showing promise. On the other hand a Ray Tracer can compute every pixel of an image separately in parallel which is done in clusters and render-farms and shortens render time. Additionally many acceleration techniques have been introduced that adopt well to Ray Tracing such as BSP and Light Buffers or techniques that try to Generalize Rays.

Radiosity and its correctness in a diffuse Global Illumination should be mentioned and kept in mind as well. But because of its non-trivial algorithm it will remain computational expensive and wonÂt't target on real-time rendering. Though the realism and image quality are remarkable and phenomena like color bleeding can be handled as well as area lights, shadow, reflection and even more.

When concentrating on high complexity scenes and interactive frame-rates the best compromise still is and will be a GPU based feed-forward rendering pipeline such as provided by OpenGL. In addition the programmability of today graphics pipelines, respectively the Vertex and Fragment-Programs, have introduced a new generation of real-time effects and approximations of phenomena like soft-shadows and environment-mapping, multi-texturing and other techniques. These all claim to add realism to a rendered scene. Nevertheless reflection via environment-maps and shadows

generated by e.g.a hardware accelerated Shadow Mapping Technique are fake.

As a result, when analyzing these different rendering methods, we can combine the power of todays graphics cards and the physically correct Global Illumination of a Ray Tracer in a new way. We will therefore explain how to break down the Ray Tracing algorithm into different render-passes treating every generation of rays separately. We will then show which parts can be satisfied by the graphics card and how to integrate a Fast Ray Tracing algorithm to add a Global Illumination component to a rendered scene. With the wish of shifting as much as possible to the GPU starting with the first generation of rays, which a Ray Tracer would fire, we end up in our interleaved and hybrid CPU-GPU Ray Tracing approach which we present in the following sections.

2. Related Work

In [Wal04] Wald describes the *OpenRT* software real-time Ray Tracing system, which is a pure software solution. It achieves interactive performance through the usage of several techniques including a fast triangle intersection, an optimized BSP, fast BSP traversal and the usage of the SIMD extension. Furthermore caching and memory optimization are combined in a wise manner. For efficiency and linear scaling of the OpenRT Renderer, it is possible to add more CPUs by clustering.

Havran [Hav00] did an intensive study on hierarchical subdivision methods. His optimized kd-tree adopts well to complex scenes and has a compact storage of inner and outer nodes and an efficient traversal algorithm. Indeed his researches constitute that a kd-tree is an optimal BSP and introduce a cost-function-based algorithm for the position of the splitting plane.

Algorithms for adding shadows to 3D scenes in the context of Real Time Rendering are research topics in many publications. Beside Shadow Volumes and other well known techniques the image based Shadow Mapping algorithm was improved by Stamminger and Drettakis in [SD02]. Their issue in how to reduce the so called perspective aliasing artefacts which are implied by uniform shadow maps gave a starting point for many other researchers. The basic idea is to distribute the resolution of the depth range in a non-linear manner using a perspective transform during the generation of the shadow map. Wimmer and Scherzer have introduced a light space perspective Shadow Mapping algorithm, shortened *LiSPSM*, which is based on this general idea in [WSP04]. Indeed they argued that any Perspective Transform could be used to achieve a wise distribution of depth in the Shadow Map. They also give a fundamental overview and analysis of the different occurring aliasing errors and provide a robust and quasi-optimum solution for generating shadows with Shadow Mapping.

Several approaches exist to map the Ray Tracing algorithm to the GPU. The increasing programmability of GPUs makes it possible to implement Ray Tracing using shader programs which can be found in [PBMH02] [Chr05]. In addition to pure software solutions, several hardware designs were introduced e.g. [CRR04], [JS02], [SWW*04]

3. Our concept

3.1. Motivation

We want to render 3D scenes of medium to high complexity at reasonable frame-rates and resolutions including Global Illumination as provided by a Ray Tracing algorithm. On the other hand a Real Time Ray Tracing approach without respect to the power of todays graphics cards would lead to low performance without clustering workstations. Therefore we developed a hybrid CPU-GPU Real Time Ray Racing Framework by combining the power and possibilities of todays programmable rendering pipelines with a "fast as possible" Ray Tracing algorithm located on the CPU.

3.2. Analysis of the Ray Tracing Algorithm

When analyzing a classical recursive Ray Tracing algorithm we end up in so called *Generations of Rays* which have to be processed for each pixel of the Image. Launching a generation of primary (camera) rays followed by their shadow rays and subsequent generations of rays depending on the properties of the hidden surfaces would be a typical overall Algorithm.

In our approach we will show that the first generation of rays can be represented by classical feed forward rendering in terms of the OpenGL rendering pipeline. In addition the corresponding shadow rays can also be processed on the graphics card. Mapping further generations of rays to an algorithm located on the graphics card is not suitable on the other hand. This is mainly caused by non existent acceleration techniques and the huge amount of data to be processed for these generations.

3.3. Naming the Render-tasks

In the following subsection we describe our concept of spreading up the different stages of a classical Ray Tracing algorithm as independent tasks for the GPU and CPU:

- **Primary Shadow Rays**: To generate the shadows for the camera rays we use a light space perspective shadow mapping algorithm which is located on the graphics card. For better results we can additionally refine the shadow boundaries, which are typically aliased, in two post processing steps: We first blur all shadow boundaries in image space within a short fragment-shader. This blur step would mark all boundaries as "not clean" and we are then capable to perform a traditional shadow test with shadow rays located on the CPU for the boundaries.

- **Reflection and Refraction Rays**: For the calculation of the reflective and refractive parts of the image we use a fast Ray Tracing algorithm which is located on the CPU. In a pre-step we generate a special id-frame-buffer by encoding a unique number for each triangle primitive of the scene, rendering this special geometry set on the standard OpenGL graphics pipeline as shown in Figure 2. This first step gives us the intersection result of all camera rays with the scene. After a read-back this frame-buffer enables a fast decision whether to calculate a reflection and or refraction ray for a single pixel when iterating the id-frame-buffer on the CPU. The reflective and refractive contribution of a pixel with respect to shadow can be calculated by our fast Ray Tracing algorithm then.

```
// Pass #1 Shadow-map Generation
shadowmap->update(eyepos,viewdir,lightdir);
shadowmap->beginShadowMapGeneration();
scene->drawRAW();
shadowmap->endShadowMapGeneration();
glClear(DEPTH_BUFFER_BIT);
// Pass #2 Id-Shading and Shadow Test
shadowmap->start();
idshader->start();
scene->drawID();
idshader->stop();
shadowmap->stop();
// Pass #3 Blurring the Alpha Channel
copy_framebuffer_to_texture();
screen->setupOrtho();
blurshader->start();
screen->draw();
blurshader->stop();
screen->resetState();
// Pass #4 Perform Ray Tracing on CPU
copy_framebuffer_to_hostmemory();
resetRayGenerations();
for( all pixels in framebuffer){
  id = get_id(pixel);
  checkIDandfillRayGenerations(id);
  clearRGB(pixel);
}
generateRayGenerations();
scene->intersect(raygenerations);
metashader->shade(raygenerations);
// Pass #5 Primary Rays + Global Illumination
copy_framebuffer_to_gpumemory();
globalshader->start();
scene->draw();
globalshader->stop();
# display the framebuffer
glutSwapbuffers();
```

Figure 1: *Pseudocode of our five render-passes.*

- **Primary Rays**: As mentioned above the surfaces hit by the camera rays can completely be shaded on the graphics card. With the opportunity of defining individual shader within a programmable rendering pipeline we can apply the same phong-shading algorithm for the camera rays on the GPU as we would in a traditional Ray Tracer on a CPU. Therefore the overall biggest computational part of the image generation, which are the primary rays, can be completely done by the GPU.

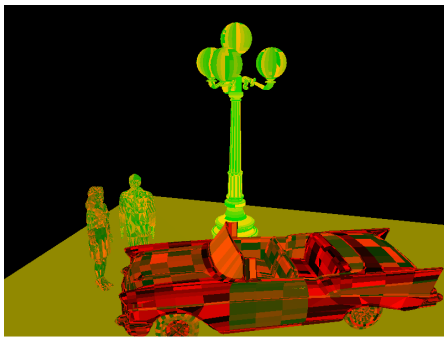

Figure 2: *Visualization of the ID-Shading step: The number of a triangle is encoded in its color value. Black means no triangle hit by a camera ray.*

3.4. Resulting Render-passes

Summarizing the above render-tasks and combining them appropriately we result in five render-passes as illustrated in pseudocode in Figure 1 and outlined in the following listing:

1. In a first pass we generate a Shadow Map. Therefore we update the Shadow-map settings appropriately and just send down a vertex-only geometry set of our scene the rendering pipeline. No additional information like materials or vertex normals are needed. This pass affects only the Depth Buffer.
2. After this pass we can perform the shadow mapping for our scene and combine it with the id-shading step. Therefore we first setup the OpenGL Texture State with a Projection- and Modelview-Matrix in Light Space. Now we render our special id-geometry set writing the encoded triangle number as RGB color and simultaneous do a shadow map depth-test within a fragment-shader and write the result of this test into the alpha channel of the frame-buffer. As a temporary result we get a frame-buffer with the triangle number for each camera ray encoded in the RGB part and the information of shadow in the alpha channel. This step is done by the following special id-fragment-shader:

```
// id fragment-shader
uniform sampler2DShadow depthtexture;
varying vec4 id;
void main (void)
{
 // ...we do the projective depth test
 vec4 depthz = shadow2DProj(depthtexture,
                            gl_TexCoord[0]);
 // ...and write the id and result
 // of depth test to the fragment
```

```
 gl_FragColor = vec4(id.rgb, depthz.r);
}
```

3. In a following blur pass we just alter the alpha channel as described in the subsection above for marking the shadow boundaries. The blurring is done by a conventional convolution filter. We therefore copy the frame-buffer to a texture and perform a weighted texture lookup in a fragment-shader and write the new alpha value for each fragment as result and leave the RGB channels untouched:

```
// blur fragment-shader
uniform sampler2DRect tex;
void main (void)
{
 // ...we define our Convolution Kernel
 // and an Offset
 ...
 // ...calculate the new blurred shadow value
 float shadow = 0.0;
 for(int i = 0; i < KernelSize; i++)
  shadow += KernelValue[i]*
            texture2DRect(tex,
            gl_FragCoord.xy + Offset[i] ).a;
 // and get the id which will be unchanged
 vec4 id = texture2DRect(tex,gl_FragCoord.xy);
 gl_FragColor = vec4(id.rgb, shadow);
}
```

4. After copying the frame-buffer to the host memory the next pass is done with our fast Ray Tracer on the CPU. By iterating the frame-buffer, respectively the triangle ids in the RGB part of a pixel and the value for the shadow in the alpha part. The overall Ray Tracing algorithm generates a reflection ray when identifying triangle number as reflective and performs a Ray Tracing for each of these. The reflective contribution then overwrites the id in the frame-buffers RGB channels. The refractive pixels are handled in the same way by generating refractive rays respectively. On the other hand non-reflective and non-refractive pixels will be set to zero. For the marked shadow boundaries, which are non-zero and non-full-byte values in the alpha part, shadow rays are generated and tested for intersection with the scene. As a result of this additional Ray Tracing shadow test the value in the alpha channel is set to 0 for shadow or 255 for no shadow respectively. These two tasks modify the frame-buffer with respect to a reflective and refractive contribution and an ensurance of shadow or illumination of each pixel in the resulting image.
5. After copying this new frame-buffer to a texture a last render-pass can produce the final image. By rendering the geometry with vertex-normals and materials states the GPU can shade the primary rays with our individual defined illumination model and can combine the information of shadow or not for each fragment and furthermore adds the contribution of reflection and refraction, calculated by the former CPU Ray Tracing, to the final pixel. This can be done in one single pass within a Global Illumination fragment-shader by calculating the fragments color with respect to the values in the frame-buffer texture supplied by the former CPU pass:

```
// Global Illumination fragment-shader
uniform sampler2DRect tex;
varying vec3 normalVec;
varying vec3 vertPosition3;

void main(void){

 ...

 coords = gl_FragCoord.xy;
 normal = normalize(normalVec);
 light = vec3(gl_LightSource[0].position.xyz);
 light = normalize(light);
 eye   = normalize(-vertPosition3);
 half  = normalize(light + eye);

 // get shadow from alpha
 shadow = texture2DRect(texture, coords).a;
 // calculate local Illumination...
 col = gl_LightSource[0].ambient *
       gl_FrontMaterial.ambient;
 // ...with respect to Shadow
 col += shadow *
        gl_LightSource[0].diffuse *
        gl_FrontMaterial.diffuse *
        max( dot(NLightVec,NNormal), 0.0 );
 shiny = pow(max( dot(normal, half), 0.0 ),
        gl_FrontMaterial.shininess);
 col += shadow * gl_LightSource[0].specular *
        gl_FrontMaterial.specular * shiny;
 MyColor.a = 1.0;
 // get ray traced contribution from rgb
 rtcol = texture2DRect(texture, coords).rgb;
 // add it...
 col.rgb += rtcol.rgb;
 gl_FragColor = col;
}
```

Figure 3 gives an overview of the above five render-tasks and outlines the different stages of the frame-buffer, respectively the RGB and alpha channel. The resulting image then includes precise shadow boundaries for the primary and proximate generations of rays and correct reflective and refractive parts.

3.5. Interleaving and Multi-threading the Render-passes

As the CPU and the GPU are individual processors we can accelerate the overall rendering by interleaving the render-passes appropriately. The CPU pass depends on the first three GPU passes and can then be done separately without the need of any GPU computations. Therefore the CPU and GPU can run in separate threads and the different render-passes can be interleaved as shown in Figure 4:

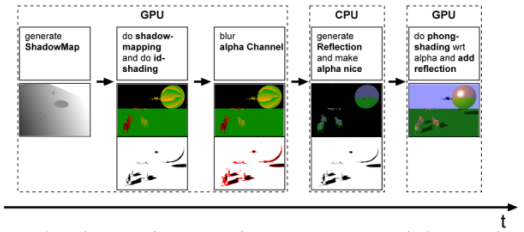

Figure 3: *The resulting render-passes: From left to right the small image in middle shows the content of the RGB part and the lower one the state of the alpha channel of the frame-buffer respectively. Only the fourth render-pass is done by the CPU whereas the others are done by the graphics card.*

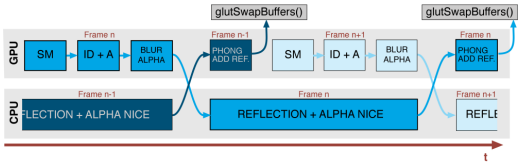

Figure 4: *The interleaving of the render-passes: The two time-lines, respectively graphics card and the CPU symbolize the different render-passes and the interleaving, note that CPU and GPU operate on different frames. SM: Shadow Map generation; ID+A: ID-Shading and Shadow Depth-test in one step; BLUR ALPHA: Shadow Boundaries are blurred; PHONG ADD REF: combining the results of the former CPU-pass within a phong-fragment-shader as described in the text in a final render-pass; REFLECTION+ALPHANICE: the CPU Ray Tracing pass and shadow-boundary test as described in the text.*

While the GPU does the shadow-map generation, the Shadow Mapping and the blurring of the shadow boundaries for frame n as described in the subsection above, the CPU can work on the frame-buffer of frame $n-1$ at the same time. When the CPU work is done for frame $n-1$ the GPU can go on with the last pass for frame n-1 and in parallel the CPU can go on with frame n. With this concept we can ensure that at least the CPU, which is the bottleneck, is always working to full capacity, while the GPU might still be under-worked.

To enable this interleaving technique we developed a double-buffer state for each entity which could change in between two frames. These are the camera, the light and the frame-buffer. Holding two states for each of them we can ensure that the two parallel processes always work on the correct state of the scene. As a consequence the out-coming final image will have a delay of one frame.

On machines with dual-processors and or hyper-threading-CPUs we can additionally split the CPU part into multiple threads. Because the relevant computations are independent for each pixel there are no crossovers between the parallel threads. Therefore each thread can individually iterate through a subset of the frame-buffer and perform the render-pass on its subset as described before. The multi-threading capability thus additionally shortens the overall computation time of the CPU part.

4. Implementation

We built and tested our framework on two PCs with following configurations:

- ***testSysI***: graphics card: GeForce® 6600 GT (driver NVIDIA 1.0-6629); processor: AMD® Athlon XP 2800™ CPU 2.0 GHz, RAM: 1024 MByte ; operating system: Linux Fedora Core 2
- ***testSysII***: graphics card: Quadro® 3400 (driver NVIDIA 1.0-7162); processor: Intel® Xeon™ Dual CPU 3.20GHz, RAM: 4096 MByte ; operating system: Linux Fedora Core 3

Our Framework is written completely in *C++* and *GLSL* and is targeted to Linux with support for the Intel compiler, version 8.1 and above, and also the gnu compiler collection gcc, in version 3.3.3 and above. For optimal performance we use the Intel C++ Compiler on testSysI.

As mentioned before we implemented a kd-tree as BSP of our 3D scenes. For an optimal placing of the splitting plane we used a cost function as explained in [Hai]. The size of the kd-Tree nodes is squeezed to 8 byte [Wal04] for optimal memory usage and cache alignment.

As aforementioned several classes are double-buffered in order to interleave the CPU and GPU render-passes. Such are abstractions for camera,light and the frame-buffer. To handle the vertex and fragment programs we use the *libGLSL* developed at our university, which provides a simple interface for defining and loading shader from a file without recompiling. For the generation of the Shadow Map we used LiSPSM-Algorithm introduced by [WSP04].

In order to provide hardware accelerated Shadow Mapping and to run our different Vertex and Fragment Programs the following OpenGL Extensions are necessary: *GL_ARB_shadow, GL_ARB_shader_objects, GL_ARB_shading_language_100, GL_ARB_depth_texture and GL_ARB_texture_rectangle*.

Furthermore two additional libraries were used: The *boost libraries* in Version 1.32 which provides threads, respectively *boost::thread*. And due to the need of a 16 byte alignment of all C++ objects our *MemoryManager* uses *boost::pool*. This guarantees that all objects are memory aligned for optimal cache alignment and the correct support for of the CPUs SIMD-Extension, which we use in our math-library for vector- and matrix-arithmetics.

5. Results and Discussion

The following Results and a Discussion on these emphasizes on the overall performance, such as render-time and scalabil-

ity with respect to the number of secondary rays corresponding to the amount of reflection within a 3D scene. Indeed our new approach gives a starting point of testing the two different architectural as well as algorithmic parts, respectively the two opposite rendering techniques, namely a fast CPU Ray Tracing algorithm and the classical feed-forward-rendering provided by OpenGL. Beside this performance test we also recommended to test the accuracy and quality of the shadow generation within our algorithm.

5.1. Test Scenes

We tested our framework with a scene of medium complexity with about 500.000 triangles which includes reflective surfaces as shown in Figure 5. The depth of reflection is currently limited to one and due to the current implementation our framework provides only one directional light. To test our framework with respect to the Shadow Generation we tested a different scene at medium complexity as seen in a following subsection. All tests have been rendered at a resolution of 1024×764 pixels, where the camera is positioned in such a way that approximately all objects lie inside the viewing frustum.

Figure 5: *Medium test scene: 500.000 Triangles, containing 10% reflection,* 1024×768 *pixel, 8 fps.*

5.2. Performance and Render-time

Indeed our approach concentrates on the idea of taking advantage of both rendering-techniques and settles a claim on Gloabal Illumination, respectively reflection and refraction, a Phong shading model and the generation of accurate shadows. Ray Tracing as a method and solution for this claim has a non-linear time-complexity related to scene-complexity and in general does all computations sequentially. Therefore a Ray Tracer cannot reach render-times like feed-forward OpenGL rasterization of nowadays graphics cards.

The overall performance of our framework closely depends on the amount of secondary rays fired by the Ray Tracer, while the GPU will mostly remain under-worked. As a final result Figure 6 illustrates the computation times of each render-pass and faces GPU and CPU. As expected the render-time is limited by the CPU which obviously is the bottle-neck of our algorithm. Concerning scalability we recommend to test whether this bottleneck would swap to the GPU when increasing the number of triangles. Though we are not able to test our framework with more than 1M triangles until now the OpenRT Renderer confirms this assumption.

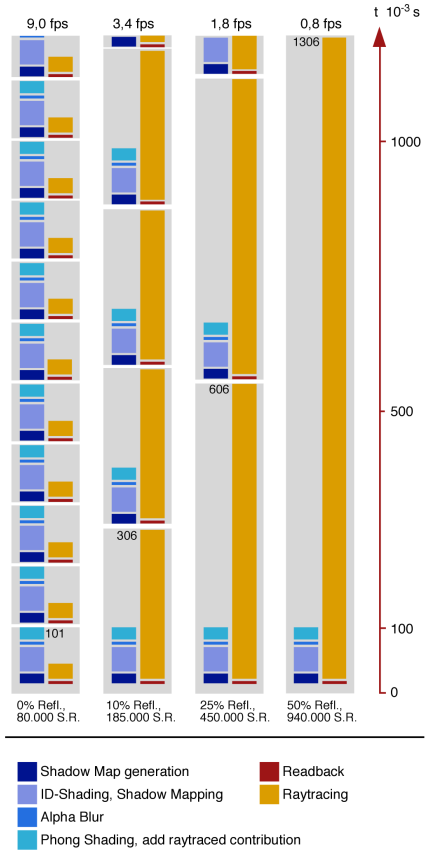

Figure 6: *Render-times of the five render-passes and the read-back on testSysII: 500.000 Triangles,* 1024×768 *Pixel, increasing reflection from left to right. The GPU remains under-worked. The Ray Tracing has none-linear scaling.*

5.3. Multi-threading

As described before our framework is able to distribute the CPU part in a multi-threading manner. The number of threads therefore was adjusted to the configuration of each test system and accordingly to the well known overhead of synchronization time of all concurrently running programs.

A comparison of the overall render-time, rendering our medium test scene as shown in Figure 5, with testSysI and testSysII gives Figure 7. As result we found that testSysII reduces the render time by factor two as expected when running with two CPU threads instead of one. We received the

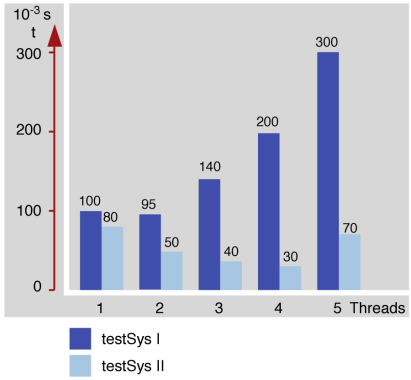

Figure 7: *Comparison of testSysI and testSysII: overall render-time in milliseconds, number of Threads.*

best results with testSysII and four threads for the CPU Ray Tracing pass. Indeed the hyper-threading dual-processor PC runs at full capacity with four threads.

5.4. Usage of the SIMD Extension

As mentioned in a former section we use the CPUs SIMD Extension for vector and matrix-arithmetics. Comparing our math library with a non-SIMD version we have measured a consistently at least 10 percent shorter render-time throughout all tests. This speedup is still remarkable because in opposite to OpenRT we do not use SIMD for ray-triangle-intersections, but indeed this would be an improvement for a further implementation of our framework.

5.5. Shadow Boundary Refinement

Our approach presents a hybrid method to generate precise and smooth shadow boundaries, as achieved by a classical ray-based shadow test in a Ray Tracer, and additionally uses the hardware accelerated Shadow Mapping algorithm for the Umbra and the Illuminated Parts of a 3D scene. With this new method we can generate exact shadows with a minimum cost in CPU time.

Figure 8 pictures our test scene rendered with the LiSPSM-Algorithm presented in [WSP04]. The same scene rendered with our Shadow Refinement algorithm and a convolution kernel of 5×5 pixels for the alpha blurring is shown in Figure 9 comparatively. As a result of our algorithm the shadow boundaries are smooth and accurate and the well-known artefacts of Shadow Mapping are completely eliminated by our Ray Tracing post process. Obviously the choice of the kernel size is a trade-off between overall performance and quality. Nevertheless artefacts with an area larger than the blurring-kernel cannot be removed.

Figure 8: *Shadow test scene,* 1024×768 *pixel, 500.000 Triangles: shadow boundaries without CPU-refinement including artefacts, 7 fps.*

Figure 9: *Shadow test scene,* 1024×768 *pixel, 500.000 Triangles: precise shadow boundaries with CPU-refinement and a blurring-kernel of* 5×5 *pixel, 6 fps.*

6. Conclusion and Future Work

We presented a new approach on rendering medium to complex scenes with regard to Global Illumination. Our framework takes advantage of a fast Ray Tracing algorithm located on the CPU and the programmable feed-forward OpenGL rendering pipeline of actual GPUs. The analysis of a 3D scene rendered by a Ray Tracer was a starting point for our approach which leads to a new algorithm by splitting the overall rendering into five render-passes. We therefore use the GPU wherever possible and only assign our fast CPU Ray Tracing algorithm where needed. In other words we exploit the capabilities of the GPU most suitable our Ray Tracer contributes in the terms of Global Illumination to a final high quality rendered image. Using a compact and fast kd-tree and the SIMD-Extension CPU-side our implemented

framework renders medium to high complexity scenes at moderate frame rates, nevertheless we could not achieve interactivity.

Our hybrid render technique identifies the differences between a CPU- and a GPU-based rendering algorithm and shows how to combine them appropriate by taking the best of each. Though the CPU bottleneck frustrates interactive frame-rates our approach could be a starting point for further research topics. In a future work we would like to map more of the overall computations within our framework to the GPU. New abstractions and appropriate algorithms in context of Ray Tracing and Global Illumination and also new data-structures have to be designed. Obviously actual researches concentrate on shifting even more algorithms to the GPU and the possibilities of multi-GPU PCs and also clustering will encourage us to keep our results in mind.

References

[Chr05] CHRISTEN M.: *Ray Tracing on GPU*. Master's thesis, University of Applied Sciences Basel, 2005.

[CRR04] CASSAGNABERE C., ROUSSELLE F., RENAUD C.: Path tracing using the ar350 processor. In *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and Southe East Asia* (New York, NY, USA, 2004), ACM Press, pp. 23–29.

[Hai] HAINES E.: Bsp plane cost function revisited. *ACM TOG 17*, 1.

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, 2000.

[JS02] J"ORG SCHMITTLER INGO WALD P. S.: Saarcor – a hardware architecture for ray tracing.

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM Press, pp. 703–712.

[SD02] STAMMINGER M., DRETTAKIS G.: Perspective shadow maps. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM Press, pp. 557–562.

[SWW*04] SCHMITTLER J., WOOP S., WAGNER D., PAUL W. J., SLUSALLEK P.: Realtime ray tracing of dynamic scenes on an fpga chip. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), ACM Press, pp. 95–106.

[Wal04] WALD I.: *Realtime Raytracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group Saarland University Saarbr"ucken, 2004.

[WSP04] WIMMER M., SCHERZER D., PURGATHOFER W.: Light space perspective shadow maps. In *Proceedings of Eurographics Symposium on Rendering 2004* (2004).