CREMA: A Parallel Hardware Raytracing Machine

Ulf Ochsenfahrt and Ralf Salomon
Faculty of Computer Science and Electrical Engineering
University of Rostock, 18051 Rostock, Germany
{ulf.ochsenfahrt,ralf.salomon}@uni-rostock.de

Abstract— **A raytracer calculates how a camera would observe a potentially complex scene consisting of numerous objects and light sources. If all the objects are modeled by n primitives, e.g., triangles, the runtime of a *software* raytracer scales at least *logarithmically*. This scaling behavior effectively imposes limitations on the scene's complexity, its size, and the raytracer's real-time capabilities. As an alternative, this paper proposes a parallel *hardware* raytracing machine. A prototypical implementation on a field-programmable gate array, as offered by markets today, validates that this machine achieves rendering in constant time, regardless of both the scene's size and its complexity.**

I. Introduction

Rendering is a well-established technique to calculate how a camera would observe a scene consisting of several objects and light sources. Rather than solving complex equations exactly, virtually all state-of-the-art rendering machines employ *approximation* techniques, which is raytracing [1] in most cases. A raytracer, as the name suggest, shoots rays into the scene. In so doing, it traces their progress, and considers secondary rays in case they touch objects or run into light sources.

In order to be efficient, a real-time raytracer assumes the following two simplifications: (1) it models all the present objects by a total of n primitives, which are simple triangles in most cases [2], and (2) light sources have the size of a mathematical point. Furthermore, it uses a simple and efficient shading model: in case a ray hits a primitive, this ray spawns at least three secondary rays to model the physical effects called *reflection, refraction,* and *shadow.*

A software raytracer spends most of its processing time on the calculation of potential intersections of a ray with any of the n primitives. A naive implementation considers all primitives leading to a computational complexity of $O(n)$ per ray. In order to relieve this bottleneck, previous research has developed acceleration data structures, such as regular grids [3], kd-trees [4], and bounding box hierarchies [5], to mention but a few. These data structures provide a criterion that allows the raytracer to stop the testing process. The pertinent literature [6] suggests that these enhancements reduce the rendering time to about $\Omega(\log n)$ per ray under certain favorable conditions. However, previous research does not provide any performance guarantees or necessary preconditions. That is, it might as well happen that a scene of a certain complexity or size still requires $O(n)$ intersection tests per ray.

The time bound $\Omega(\log n)$ mentioned above is notorious for single-processor software solutions [7], since they can access their memory and perform intersection tests only sequentially. To go beyond these limitations, Section II proposes a new hardware design called Constant-Time Raytracing with Embedded Memory Architecture (CREMA). A key concept of CREMA is that it employs a tiny processor and a few data cells for *every* primitive. These n nano processors perform all intersection tests in parallel, which requires exactly *one* macro cycle. Hence, the computational complexity shrinks to $O(1)$.

In order to validate the feasibility of the proposed architecture, Section III discusses a prototypical implementation. Even though the current implementation is clocked at 28 MHz, it renders images of 256x128 pixels in real-time at 13 frames per second. Finally, Section IV concludes with a brief discussion.

II. CREMA: A Parallel Hardware Renderer

The canonical setup of a rendering machine is depicted in Figure 1. In this approach, the three processes *ray buffer*, *geometry calculation*, and *shading* model the appearance of the scene's objects and the effects of its light sources. In this respect, this paper is mainly concerned with the realization of the *geometry calculation* process. It consists of a large set of simple nano processors as well as a tree-shaped set of selectors. The result of this processing queue is that triangle that is intersected closest to the origin of the ray under consideration. Provided that this architecture features as least one processor for each primitive (see also Subsection II-C for relaxed conditions), this architecture processes all the required intersection tests simultaneously. As a result, the architecture yields one result per time step, called macro cycle, leading to a computational complexity of $O(1)$ per ray.

It should be mentioned here that CREMA's design draws some inspiration from an early approach [8] that was discarded at that time. The remainder of this section describes the design of the nano processors as well as the intersectors in full detail, and wraps up with a few remarks on the required infrastructure, which is also implemented on the same chip.

A. Design of the Nano Processor

For every incoming ray, each of the nano processors calculates whether or not the ray hits its stored triangle. In case of an intersection, the nano processor further calculates the distance of the intersection to the ray's starting point.

The basic idea of the intersection test by Segura and Feito [9] is illustrated in Figure 2: Taking the two ray points P and Q as well as the three triangle's vertices A, B, and C, the test calculates the signs of the three tetraeder's volumes $PQAB$, $PQBC$, and $PQCA$. The ray PQ hits the triangle ABC if all

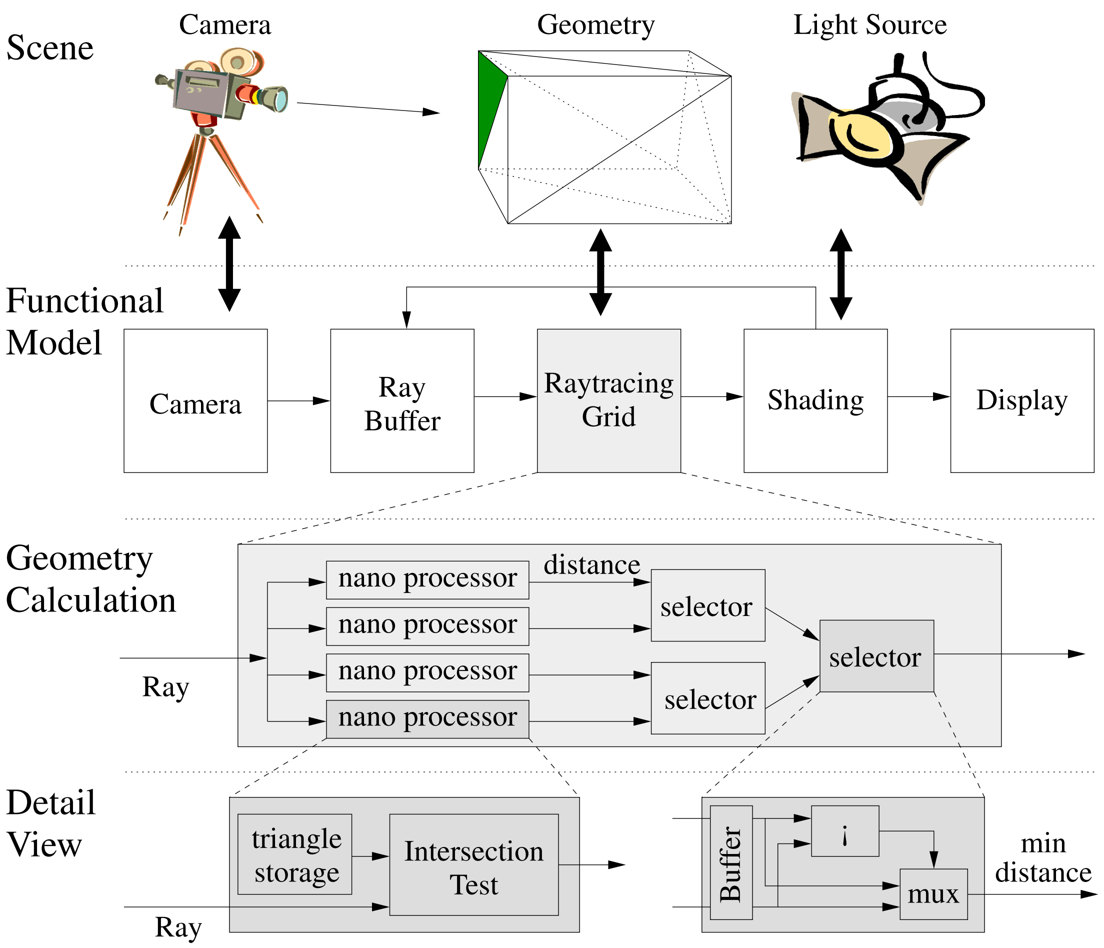

Fig. 1. The CREMA architecture realizes the geometry process as a grid of nano hardware processors, which pass their results to a selector hierarchy. The nano processors and selectors in turn consist of very simple storage and calculation modules.

signs are equal. The volume equation for the first tetraeder is:

$$\operatorname{sign}(V(P,Q,A,B)) = \operatorname{sign}\begin{vmatrix} P_x & P_y & P_z & 1 \\ Q_x & Q_y & Q_z & 1 \\ A_x & A_y & A_z & 1 \\ B_x & B_y & B_z & 1 \end{vmatrix}, \quad (1)$$

which can be easily evaluated with multipliers and adders.

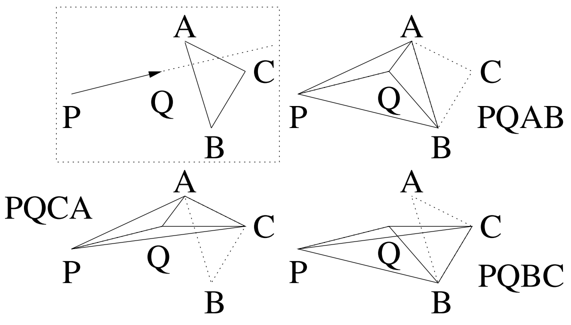

Fig. 2. The ray *PQ*, the triangle *ABC*, and the three tetraeders *PQAB*, *PQCA*, and *PQBC*. If all three tetraeder's volumes have the same sign, the ray intersects the triangle.

In case of an intersection, the nano processor has to further calculate the distance of its intersection to the ray's starting point P. Generally, the distance t at which the ray hits the triangle can be calculated from the plane and ray equations. With A denoting one of the triangle's vertices, N denoting the triangle normal, and P and Q denoting distinct points on the ray, the distance can be determined as follows:

$$(X-A)*N = 0 \qquad X = P + t*(Q-P),$$

thus arriving at the distance equation

$$t = \frac{NA - NP}{NQ - NP}\,. \quad (2)$$

B. Design of the Selector

After all nano processors have performed their intersection tests and have done - as far as necessary - their distance calculations, the selection hierarchy is implemented as a binary tree of simple comparator/multiplexer modules. That is, a selector compares two inputs t_i and t_j according to Equation (2), and sets its output $o = \min\{t_i, t_j\}$ to the minimum of the inputs. The comparison can be done in two different ways: either the selector can compare distances itself. In that case, the nano processors would have to perform the division.

Or, the selectors can use the enumerator and denominator of Equation (2) directly as follows:

$$t_i = \frac{N_i A_i - N_i P}{N_i Q - N_i P} = \frac{a_i}{b_i}, \quad t_j = \frac{N_j A_j - N_j P}{N_j Q - N_j P} = \frac{a_j}{b_j}\,.$$

The case $b_i, b_j > 0$ is then handled as follows:

$$t_i < t_j \quad \Leftrightarrow \quad \begin{vmatrix} a_i & b_i \\ a_j & b_j \end{vmatrix} < 0\,. \tag{3}$$

This can be represented as the sign of the determinant of the matrix of t_i and t_j. If $b_i, b_j > 0$ and the sign of the determinant is negative, then $t_i < t_j$ and the i^{th} triangle is hit before the j^{th} triangle; vice versa if the sign is positive. The generalization to all other cases is straightforward. If one of b_i or b_j is negative, the selector has to flip the sign before the test.

Due to their structure, equations (1), (2), and (3) can all be handled by the same multiply-and-accumulate logic, which greatly reduces the number of required logic elements.

C. Enhancements

CREMA's basic design can be improved by various enhancements. Two of them are described here, since they reduce the required chip area and accelerate the runtime performance.

The first enhancement consists of splitting the intersection and distance calculations into two stages, and storing common subexpressions in the nano processor's local memory. For example, in the distance equation (2) $t = (NA - NP)/(NQ - NP)$ the nano processors can first pre-calculate all terms that involve the ray's starting point P, i.e., NA, NP, and $NA - NP$. Afterwards, the nano processors perform all operations involving the ray's second point Q. Similarly, the terms of the volume equation (1) can be split into two subsequent phases. This optimization requires more local memory cells for storing common subexpressions, but it improves the rendering time by more than a factor of two for rays with a common origin; this applies at least to all rays starting at the camera.

The second enhancement lets every nano processor host a pair of triangles that share an edge. This step is generally applicable, since according to the Euler-Poincaré Formula [10], all usual geometries have an even number of triangles. Furthermore, Peterson's graph theorem [11] provides a guideline of how an object's primitives should be grouped into pairs. This second enhancement has the following three advantages: (1) it saves memory space, as only four as opposed to six points are necessary to describe two triangles, (2) it saves processing time, because it reduces the number of necessary calculations, and (3) it significantly saves silicon area, since it reduces the number of selectors by a factor of two.

D. The Infrastructure

CREMA's infrastructure consists of the four modules *camera*, *ray buffer*, *shading*, and *display* (Figure 1), which can be found in *all* rendering machines in some flavor. Each of these processes is mapped onto a distinct hardware module. The *camera* module generates rays and feeds each of them to the *ray buffer*. At the same time, the *ray buffer* distributes a selected ray to all of the the nano processors, which forward their results to a hierarchy of selectors. These selectors propagate their selections from one level to the next, resulting in one distinct result, which is passed on to the *shading* module. All hardware modules work simultaneously and their data exchanges are clocked. The *shading* module is the only exception, since it must be able to pass multiple rays to the *ray buffer* within one meta cycle. CREMA utilizes additional hardware to setup and modify the scene between frames. As in all software raytracers, these hardware modules still work sequentially.

E. Summary

This section has introduced the core concepts of CREMA. Its main property is a grid of very small, rather rudimentary, nano processors, which all simultaneously determine whether or not the current ray hits their respective triangles. Then, a binary tree of selection modules determines the triangle that is closest to the ray's origin. After an initial delay of $\tau \sim \log_2 n$, CREMA yields one result per macro cycle, since the processing of all intersection tests is done in parallel. From an abstract architectural perspective, the approach presented here is a massively parallel single instruction multiple data (SIMD) machine, which, in a sense, corresponds to developments in the area of ultra-high performance computing [12], [13].

III. Prototypical Implementation

From a technical point of view, CREMA is best implemented as an ASIC with embedded DRAM. ASIC designs, however, are way too expensive for University research. Therefore this paper uses a much cheaper field-programmable gate array (FPGA) prototype that is based on a Xilinx Virtex-E 1000 chip. Even though the number of triangles is severely limited due to the small FPGA size, the prototype allows for the validation of the basic concepts.

A. Latency and Throughput

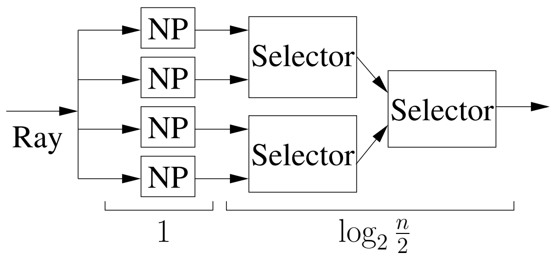

Fig. 3. In CREMA, latency of the nano processors (NP) is constant and independent of their actual number, since they all work in parallel. The selectors then add a logarithmic *latency*, since they are organized as a tree.

Latency is defined as the time the architecture requires to fully process every single ray. In software, the latency τ is strongly linked to the throughput $p = 1/\tau$. For example, if a ray takes $\tau(n) = \Omega(\log n)$ time for being processed, the maximal throughput is $p(n) = O(1/\log n)$ rays per time unit.

The parallel raytracer CREMA yields much different performance figures: According to Figure 3, both the nano processors and the selection hierarchy impose delays $\Theta(1)$ and $\Theta(\log_2 n/2)$ leading to an overall delay $\tau(n) = \Theta(\log n)$.

In CREMA's geometry processing module, all operations are pipelined. That is, the nano processors perform a complete ray-triangle intersection test, and simultaneously, all selectors do the comparisons and forward their results to the next layer, and thus yields one result (pixel value) per macro cycle. Due to internal clocking, one macro cycle requires 64 clock cycles.

The prototype renders images of 256x128 pixels and directly outputs them to a VGA monitor. Even with a clock rate as low as 28 MHz, the prototype renders 13 frames per second. It renders only primary rays with simple shading: either depth-based coloring, or shading based on one light source and an approximation of the Banks reflection model.

B. Chip Area

In a software architecture, both CPU and memory require silicon area. The CPU requires a constant amount of silicon, and can process scenes of arbitrary size and complexity. The memory size, on the other hand, is dependent on the scene complexity, since n triangles require at least $\Omega(n)$ cells. In addition, the software raytracer's acceleration structures consume $\Theta(n)$ additional cells in the best possible case.

CREMA requires silicon area for the infrastructure, the nano processors, and the selectors. Since the CPU is much simpler than in the traditional architecture, the total silicon area mainly amounts for the nano processors and selectors. For n triangles, the architecture has to employ $n/2$ nano processors and $n/2-1$ selectors. The consumed silicon area thus scales with $\Theta(n)$, as in the software approach.

IV. Conclusion

This paper has argued that the runtime of software raytracers is bound to the sequential nature of conventional CPU/memory-based architectures. Even though many acceleration data structures are able to significantly speed up raytracers under certain favorable conditions, current research cannot give any performance guarantees. As an alternative, this paper has proposed a new parallel hardware rendering architecture, called CREMA, that processes all the time consuming rendering operations fully in parallel; regardless of the scene's complexity and size, CREMA requires only constant processing time $O(1)$ per ray.

Since ASIC designs are way too expensive for regular University research, this paper has also presented a prototypical implementation that is based on the Xilinx Virtex-E 1000 field-programmable gate array. This prototype is clocked at 28 MHz and yields 256x128 pixels at a rate of 13 images per second.

Initiated by the prototype, future research will try to further simplify the intersection tests in terms of hardware resources. Furthermore, future research will explore to what extent the parallel hardware raytracer would benefit from the incorporation of traditional acceleration data structures. The ultimate goal will be the development of an ASIC design that renders practically relevant scenes and image sizes in real time.

Acknowledgements

The authors gratefully thank Ralf Joost for his numerous comments and fruitful discussions. Particular thanks are also due to Prof. Dirk Timmermann for his continuous support. Part of this research has been supported by a fellowship from the Landesgraduiertenförderung Mecklenburg-Vorpommern. Part of this research has been supported by the German Research Foundation (Deutsche Forschungsgemeinschaft), as part of the MuSAMA Research Training Group, grant 1424.

References

[1] T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, June 1980.

[2] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek, "Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip," in *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association, August 2004.

[3] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker, "Ray Tracing Animated Scenes using Coherent Grid Traversal," *ACM Transactions on Graphics*, 2006, (Proceedings of ACM SIGGRAPH 2006, to appear).

[4] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1176–1185, 2005.

[5] I. Wald, S. Boulos, and P. Shirley, "Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies (revised version)," *Technical Report, SCI Institute, University of Utah, No UUSCI-2006-023*, 2006.

[6] V. Havran, "Heuristic ray shooting algorithms," Ph.D. dissertation, Czech Technical University, Prague, Czech, 2000, http://www.cgg.cvut.cz/˜havran/phdthesis.html.

[7] L. Szirmay-Kalos, V. Havran, B. Balázs, and L. Szécsi, "On the Efficiency of Ray-shooting Acceleration Schemes," in *Proceedings of the 18th Spring Conference on Computer Graphics*. ACM Press, 2002, pp. 97–106.

[8] M. K. Ullner, "Parallel Machines for Computer Graphics," Ph.D. dissertation, Cal. Inst. of Technology, Passadena, California, USA, 1983.

[9] R. J. Segura and F. R. Feito, "An Algorithm for Intersection Segment-Polygon in 3D," *Comp. & Graphics*, vol. 22, no. 5, pp. 587–592, 1998.

[10] H. S. M. Coxeter, *Regular Polytopes*, 3rd ed. Dover Publications, June 1973, ch. 9, pp. 165–172, Poincaré's Proof of Euler's Formula. [Online]. Available: http://mathworld.wolfram.com/PoincareFormula.html

[11] R. Diestel, *Graph Theory*. Springer-Verlag Heidelberg, 2005.

[12] Z. Baker and V. Prasanna, "Performance modeling and interpretive simulation of pim architectures and applictions," *Euro-Par 2002.*, 2002. [Online]. Available: citeseer.ist.psu.edu/baker02performance.html

[13] S. Tomashot, "An Embedded DRAM Approach," 2003, IBM Corporation. http://www-306.ibm.com/.