Raytracing Point Clouds using Geometric Algebra

Crispin Deul[1] Michael Burger[1] Dietmar Hildenbrand[1] Andreas Koch[2]

[1]Interactive Graphics Systems Group
Computer Science Department
TU Darmstadt, Germany
dietmar.hildenbrand@gris.informatik.tu-darmstadt.de

[2]Embedded Systems and Applications
Computer Science Department
TU Darmstadt, Germany
koch@esa.informatik.tu-darmstadt.de

ABSTRACT

Geometric Algebra (GA) supports the geometrically intuitive development of an algorithm with its build-in geometric primitives such as points, lines, spheres or planes. But on the negative side GA has a huge computational footprint. In this paper we study how GA can compete with traditional methods from Linear Algebra (LA) in the field of raytracing. We examine the raytracing algorithm for both GA and LA on the basis of primitive operations. Furthermore we introduce a novel framework for rendering point clouds based on spheres and planes as surface elements. We use this model to benchmark implementations of both algebras. Our results show that depending on the microprocessor architecture like CPUs, FPGAs or GPUs Geometric Algebra and Linear Algebra can raytrace with comparable speed.

Keywords: FPGA, Geometric Algebra, GPGPU, Point Cloud, Raytracing.

1 INTRODUCTION

this paper we investigate how to speed up calculations in Conformal Geometric Algebra to get comparable speed to linear algebra in the field of computer graphics. In the last decades Geometric Algebra (GA) has become a reasonable alternative in describing geometric algorithms compared to other systems like vector algebra.

One can work with Geometric Algebra in a very intuitive way since all objects of the algebra have a geometric meaning. In the conformal model, which we use throughout this paper, one can describe lines, planes and spheres directly as objects of the algebra. Furthermore operations like reflections and rotations can be applied uniformly to all geometric objects. As a result algorithms formulated with GA are very compact compared to systems describing geometry that are usually used in computer graphics.

Besides the new objects like spheres or planes and the uniformly applicable operations GA also includes many other mathematical systems like vector algebra, projective geometry or quaternions that enjoy a widespread use in computer graphics today. GA developers can slightly shift from their knowledge in these systems to the new opportunities introduced by GA while still being able to use their elaborate algorithms.

Figure 1: Max Planck point cloud rendered at 4.5 fps on an AMD HD4850 GPU. The viewport size is 640 by 480. The model consists of 96208 surfels.

While these properties of Geometric Algebra are very exciting especially for people working in graphics, computer vision or animation, there seems to be one major drawback that causes a niche existence of GA in todays applications. The mathematics in the conformal model of GA are based on the calculation of 32 dimensional so called multivectors. These multivectors represent the geometric objects of GA like spheres or planes. Different products between multivectors lead to operations like intersections or reflections. To take the scare here we have to admit that for geometric meaningful objects one often does not need more than ten non-zero entries of these multivectors. As a result mathematical effort reduces a lot in a non-naive implementation of GA [5]. While this still seems to be

a lot of mathematical effort there are algorithms that work faster using GA instead of using LA [10].

There has been a second development in the last years that leads to the results of our paper. After hitting the power wall with their monolithic cores and increasing clocks CPU developers began to put more and more cores onto their chips to increase computational power. Secondly around the same time GPUs have been opened to general purpose computations by the introduction of specialized computing platforms. Today there are lots of parallel computational resources available in commodity hardware [1] [13] [16]. Geometric Algebra benefits from these architectures since multivector entries can be computed independently of each other.

In this paper we chose a field of computer graphics, namely raytracing, to investigate how much overhead there really is in choosing GA in favor of LA by simply counting the needed mathematical operations for the different raytracing primitives. We introduce a novel surfel (surface element) model based on GA spheres and planes to represent the local surface of a point cloud. With the help of this model we fortify our theoretical observations by raytracing point clouds on different microprocessor architectures. Furthermore we investigate the impact of parallelism on both Linear Algebra and Geometric Algebra versions of our algorithms.

2 RELATED WORK

The performance of raytracing with conformal GA on a general purpose CPU has already been examined in [3] and [6] but without considering the question of parallelization. Parallel FPGA implementations of raytracers were presented in [21] or [4] but only on the basis of LA. The topic of implementing a general GA processor on special hardware architectures like FPGAs was discussed in [19] and [7] with the first one only implementing one of the products of conformal GA and the second one concentrating on the 4D homogeneous space. Both also without the discussion of optimization techniques and no relation to raytracing. In this paper we try to combine all these topics to optimized GA raytracing on specialized hardware respectivly GPUs.

A number of rendering approaches and surface definitions of point clouds have been proposed in the past. Rusinkiewicz et al. [20] propose a method based on splatting small quadrats or ellipsoids onto the screen for a subset of the point cloud. Ohtake et al. [17] use local low degree implicit functions based surfels to approximate the point cloud. Their approach is close to ours since they use a surfel as a local approaximation and define the region of influence of their surfel by using a bounding sphere. Though their approach is still CPU based. A GPU based extension of the SLIM rendering is presented by Kanai et al. [14]. In contrast to our approach Kanai et al. create their primary rays by rasterizing the bounding box of the surfels. Guennebaud et al. [8] fit spheres into the point cloud similar to our approach. While we use the spheres as a direct representation of the surface Guennebauds spheres are only an intermediate step in finding an algebraic point set surface.

3 SURFEL MODEL

For our surfel model we chose a representation that directly fits to Geometric Algebra. As a result we can take advantage of the primitives and operations that are directly included in the algebra. We chose the two geometric objects plane and sphere as a basis of the surfels. These two objects are the only objects that on their own represent a surface in GA. With planes and spheres we can directly approximate local details of a real-world model. To get a representation of a whole model we simply use several of the surfels that each represent different local features of the model on their own. Since there are no data file sources for our new model representation we have to acquire the data from other representations. One way is to use point clouds as a data basis and to fit the surfel locally into a neighborhood of points. Another way would be to use triangle meshes. One could create the planes by using the plane of a triangle. Spheres could be created by using one vertex and three of its neighbors to describe a sphere of GA with the help of the outer product.

3.1 Building the Model

We build our surface by using point clouds as a data basis. An Algorithm to fit planes and spheres into point clouds has been published in [9]. The algorithm is based on the distance measure of GA between points and spheres. With the distance measure we can define a least squares approach for the point neighbourhood. Based on the least squares approach the eigenvalues of a 5x5 matrix have to be solved where the smallest positive eigenvalue directly includes coefficients of a sphere. A nice property of the algorithm is that we do not have to care whether the point neighborhood is planar since then the result of the algorithm are the coefficients of a plane. In fact in geometric algebra one can think of a plane as a sphere with infinite radius [12].

We use an iterative algorithm to get our final model. We fit surfels into the point cloud as long as there are points that are not represented by one of the already fitted surfels. To get the local neighborhood we randomly chose a non-fitted point which we call the fitting point. With the fitting point we query a kd-tree including the whole point cloud for the k nearest neighbors. The fitting point is always assumed to be fitted by the calculated surfel from the fitting algorithm. For

the k neighbors we calculate the distance to the surfel. We define each neighbor to be fitted if its distance to the surfel is below a previously defined bound ε. Using this approach we can significantly reduce the data amount compared to a naive algorithm where one would fit a surfel for every point of the point cloud.

Furthermore we introduce a bounding mechanism into our surfel model. If you consider spheres with low curvature or planes these objects will usually cover the whole image while often approximating a part of the model data that is relatively small in the image space. To counter this behaviour we introduce the bounding mechanism. A natural choice for the bound is a region with some radius around the fitting point that includes most of the points in the neighborhood that were involved in the fitting process. The translation of this requirement into GA is a sphere centered at the fitting point. We calculate a first radius candidate of the bounding sphere by taking the distance between the fitting point and the farthest point of the neighborhood that is fitted by the surfel. In most cases we can take this candidate as a feasible radius for the bounding sphere but there are cases where the fitted sphere will be similar in size to the bounding sphere if taking this candidate for the radius. A result of the similar size are visual artifacts in the final rendered image. For many models it is sufficient to take a value between the radius down to a quarter of the radius of the fitting sphere as an upper bound for the radius of the bounding sphere to avoid most artifacts.

3.2 Raytracing the Surfel Model

In Raytracing we want to know where the nearest intersection between a ray shot from the camera through a pixel and the surface is located in space. We represent our rays by lines of Geometric Algebra. We create our rays by taking the camera origin and the 3D position of the pixel on the image plane. With these two points we build the outer product with the point at infinity to get a line. We can intersect the lines with both spheres and planes. Since planes in GA are spheres with infinite radius we can use our intersection indicator and intersection point algorithms for both of these objects. The algorithm to find the nearest intersection point for one pixel looks as follows:

1. **Find a candidate fit** We use both brute force and spacial data structure based approches depending on the use of our results.

2. **Intersect the bounding sphere** The bounding sphere indicates the region in space where the fitted surfel is feasible regarding the model data. Additionally in most cases the bounding sphere is much smaller in screen space than the fitted surfel. To reduce the intersection operations with fitted surfels we first calculate the intersection indicator of the bounding sphere.

3. **Test for intersection** The effort of calculating the intersection points can be saved in some cases when the ray intersects the bounding sphere but is tangential to the surfel. Furthermore the calculation of the intersection indicator imposes no extra cost since we can reuse the calculated value in the calculation of intersection points. Vielleicht noch dazu dass beim betrachten eines modells the bspheres wie ein matel um das modell sind und diese Fälle dann ausgeschlossen werden.

4. **Calculate intersection points** This operation is somewhat more involved in GA than in LA. In LA you can simply calculate the ray parameter t1 and t2. Using the parameters, the origin and direction of the ray one can easily calculate the intersection points. In GA the result of the intersection is a point pair which includes both of the intersection points. We have to dissect the point pair to get the intersection points. After dissection we have to normalize the intersection points so that the following tests work in the right way.

5. **Test intersection points against bounding sphere** We have to introduce this test before we can expect the intersection points to be feasible points of our model. There are cases where a ray intersects both the bounding sphere and the fitting sphere but only one of the possible two intersection points or none of them is feasible concerning our model description. In figure 2 you can see three of the cases depicted in 2D that can occure when a ray intersects both spheres. The top case is the usual case where both of

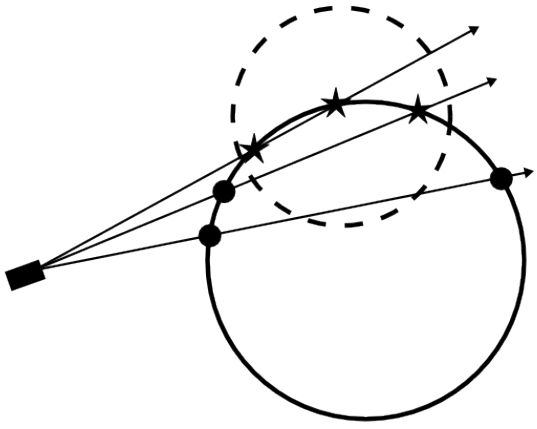

Figure 2: Three cases of the intersection of a ray with both the bounding sphere and the fitted surfel. The surfel is depicted in black while the bounding spheres outline is stippled. There are three feasible intersection points depiected by the stars and three infeasible intersection points represented by the black filled circles

the intersection points are feasible and we take the nearest one for further calculations. In the middle case only one of the intersection points is inside the bounding sphere. The lower case shows an example of the intersection of both spheres where none of the intersection points is feasible for our model.

6. **Test intersection points to be the nearest points** We use the inner product of Geometric Algebra to get the distance measure between an actually found intersection point and a point from a previous iteration of the algorithm.

7. **Repeat 1. - 6. for other candidate fits**

After finding the nearest intersection point for the ray we shade the pixel using the Phong lighting model. To calculate the diffuse part of the shading the Phong model needs the normal at the intersection point. Calculating the normal is the first time we have to branch depending on the type of our surfel. The normal of a plane surfel can be read directly from the GA coefficients. The normal of a sphere surfel can be calculated from the center of the sphere to the intersection point.

4 MINIMIZING OPERATIONS

One subgoal of this investigation was to compare our GA raytracing approach to existing solutions which are based on Linear Algebra and their amount of primitive operations like additions and multiplications. We looked at the following raytracing subtasks:

- Determining whether a ray intersects an object in the scene, in our case spheres
- Calculating the intersection point of a ray and an object
- Finding the surface normal at the intersection point
- Calculating the reflection vector which is needed for lighting and recursive raytracing

The number of operations for the case of LA was taken from [22] where a GPU raytracing method based on quadrics is introduced. Our algorithm was developed and tested with CluCalc [18]. To analyse and increase the performance of our algorithm we used the tool Gaalop [11] to symbolically optimize our GA formulas. The output of Gaalop was then searched for further potential of optimization with the intend of reducing the total number of multiplications and additions/substractions. This way differs from the approach of optimization in [6] where a GA raytracer is implemented on a CPU with the help of Gaigen2 [5]. Gaigen2 increases the performance of GA algorithms by using specialized objects instead of standard 32 entry multivectors. This leads to the advantage that only those entries are considered in calculations which really belong to an object. For example a sphere will always contain only non-zero coefficients for e_1, e_2, e_3, e_{inf} and e_0. In general we used three ways to reach our goal of less primitive operations:

1. We searched for constant values of variables which could be excluded from the calculation. Especially constant values of zeroes and ones lead to simplifications.

2. Gaalop sometimes calculates coefficients which don't belong to the object. These calculations can be removed completely.

3. Sometimes the same multiplications appear in more than one coefficient or more than one time in the same coefficient. These parts were factored out. They can be precalculated in a previous step and the result is used in the computation of the coefficients.

All optimizations were done under the point of view that the algorithm should run on a parallel platform and especially on GPUs and FPGAs. Point 3 of the list above could be important for implementing the algorithm on a FPGA, because it implies a pipeline architecture where the result is calculated in some single steps. This type of architecture can be computed on a FPGA in an efficient way if it is assured that the pipeline can be filled constantly with new data. This requirement is fullfilled in raytracing applications because of the high number of pixels for which the raytracing procedure must be executed. Another intend during the developement of the algorithm was to avoid the use of square roots and divisions because they cause lot of computational effort. In the LA case most divisions and roots are caused by the normalization operation. In GA we are in most cases able to use unnormalized objects because a scaled multivector represents the same geometric object in the conformal space. To demonstrate our approach we describe in detail the inspection of the ray-sphere intersection and reflections in the following two sections.

4.1 Ray-Sphere Intersection

In the GA case we have two objects. The sphere S and the ray R, which is represented by a line. The surface normal is represented by a line, too. In the following $*$ denotes the geometric product of two entities, while the inner product is represented by the . operator. S and R are intersected through the outer product $S \wedge R$. The result of this operation is the point pair Pp. We have to extract the point P from Pp which is nearest to the eyepoint. For this extraction the following formula is used:

$$P = (\sqrt{Pp.Pp} \pm Pp) * (einf.Pp)$$

This is a variation of the extraction formula from [12, p. 74, 6.11] where the division through $einf.PP$ is replaced by a geometric product. This represents the same object in GA like described above.

The inner product $Pp.Pp$ indicates whether the ray intersects the sphere or not. If it is positive there exists an intersection. The calculation of this value consists of a long sum of products and can't take advantage of parallelism. In LA it is necessary to compute the discriminant of a quadratic equation which leads to 8 additions and 20 mutliplications. In GA we need 14 additions and 22 multiplications. So the effort for the decision of intersection is comparable in both algebras. Considering all subtasks, the inspection of [22] and counting of operations lead to the amount of calculations summarized in table 1.

	GA	LA
additions/substractions	29	12
multiplications	42	23
divsions	0	1
square roots	1	1

Table 1: operations for intersection

So in general GA needs two times more operations than LA. But a point consists of five non-zero coefficients which can be calculated in parallel. In LA there are only three entries in the result vector which can be computed simultaneously. As a result it seems possible that the GA computations can be performened in the same time as those from LA on a parallel hardware.

4.2 Reflections

The field of reflections was analysed with the most effort of the four subtasks. First we looked at reflections in the 5D conformal space. To reflect an incoming ray on a sphere we construct a temporary plane PL through the intersection point P in the direction of the surface normal N in P. With the help of two geometric products we can calculate the reflected ray R_{ref} by the formula:

$$R_{ref} = -Pl * R * PL$$

The resulting C-code created by Gaalop contained over 8 times more primitive operations than the LA solution. By hand optimizations lead to a solution which is still between 5 and 6 times larger than the existing LA solution with the help of the formula. Further optimization seems not to be possible. Another approach was to use a rotation around the normal. This is possible because our normal is represented as a line and not as a direction vector like in LA. But this way also leads to results comparable to the temporary plane solution. This is due to the characteristic of 5D GA that all calculations are done free in space and not like in LA related to the origin what leads to a clearly higher computational effort. So we analyzed the reflection in 3D GA. Our investigation was based on [23, p. 108f] and took into account two different variants of the GA description of reflection. The first one is the equivilant to our 5D solution and taken from [23, p. 109, 8.28]. The refleced ray is calculated through two geometric products of the ray R and the normal N of the plane PL.

$$R_{ref} = -N * R * N$$

This leads to C-code with 3 times higher operation count than LA. After by hand optimization the operation count can be reduced to the same amount as in LA. But to do this we had to presume that the constructed normal has unit length, which has to be achieved through cost intensive normalization. However because of our use of the Phong model we need the normalized direction of the reflected ray anyway, so that this is not drawback.

	GA 5D	GA 3D	LA
additions/substractions	25	5	5
multiplications	37	7	7

Table 2: operations for reflection

So the GA and LA solution have the same amount of operations in 3D. Like in LA the result vector contains 3 elements which can be computed in parallel.

5 IMPLEMENTATION DETAILS

In our fitting process we find the neighbors of the fitting points by using a knn-search of the ANN library [15]. We use the newmat library [2] to calculate the eigenvectors of the 5x5 matrix computed from the point neighborhood.

For our final implementation we use the OpenCL environment together with an AMD Radeon HD4850 graphics card to raytrace views of our scenes. We create a 256 by 256 array of threads which is the maximum for our hardware in the OpenCL environment. The array of threads is slided across the image domain so that every pixel is covered once by one thread. Every thread of the thread array calculates the raytracing algorithm for the covered pixel independent of the other threads.

We derive the rays by calculating a GA line using the origin of the camera and a point on the image plane. The point on the image plane is interpolated bilinearly in euclidean space from the 3D coordinates of the image planes corners depending on the pixel position and the image resolution.

To speed up the calculation of intersections we use a modified kd-tree. When we split a node of the kd-tree in the building process we can not cut surfels that cover the splitting plane of the kd-tree node. We decided to enlarge the axis aligned bounding boxes (AABB) of the resulting child nodes so that every surfel is enclosed completely in exactly one of the child nodes. We assign the surfels to the child node that is covered by most of

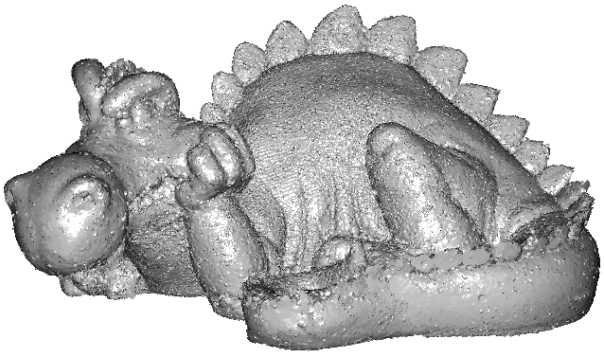

Figure 3: Phlegmatic Dragon point cloud rendered at 2.6 fps on an AMD HD4850 GPU. The viewport size is 640 by 480. The model consists of 166162 surfels.

the surfels AABB volume. A result of our kd-tree building process is that several of the trees node AABBs will intersect each other which does not happen in a real kd-tree. Furthermore we have to use a larger memory footprint than is necessary for the usual kd-tree because we save six coordinates for two corner points of the nodes AABBs instead of only saving one coordinate for the splitting plane and its direction. To traverse the kd-tree we use a stack in OpenCLs shared memory. On our hardware shared memory is emulated in global memory. A result of this is that on our hardware the large memory footprint is not such a big problem. Instead of reading the AABB information from the stack like one would do it with the usual kd-tree we read the coordinates from our kd-tree data structure. Both data sources are in global memory.

6 RESULTS

We chose two different architectures to benchmark our algorithms. We implemented a CPU version to measure the impact of choosing one of the algebras for the raytracing application directly. Our second architecture are AMD 45xx series GPUs. The AMD GPUs can be seen as a parallel processor for one invocation of the raytracing algorithm for one pixel. For further details we refer to section 6.2.

6.1 Raytracing on CPUs

Our CPU implementation of the raytracing algortihm is written in plain C/C++. We do not use any vector extensions like SSE since we are interested in the performance of LA and GA based on their mathematical effort. To speed the algorithm invocation up we split the calculations for the pixel array of the final image up among multiple CPU cores with OpenMP pragmas.

6.2 Raytracing on GPUs

We decided to use the AMD Stream Technology for our benchmarks on GPUs to get two advantages. First the AMD processing elements that compute a single thread of our GPU raytracing kernel have a 5-way VLIW Design (very long instruction word). The processing elements can compute up to five floating point additions or multiplications simultaneously which make up most of the calculations of our GA raytracing algorithm. Second with AMD Stream Kernel Analyzer we can disassemble the compiled code for the GPU. Let's look at the example of calculating the reflection in Linear Algebra. In listing 1 you can see the according kernel written in the Brook+ language.

```
kernel void reflection(float4 incident_ray<>,
  float4 normal<>, out float4 reflected_ray<>){
      float factor;
      float4 ret;
      factor = 2.f*(incident_ray.x*normal.x +
      incident_ray.y*normal.y+
      incident_ray.z*normal.z);
      ret.x = incident_ray.x - normal.x * factor;
      ret.y = incident_ray.y - normal.y * factor;
      ret.z = incident_ray.z - normal.z * factor;
      ret.w = 0.f;
      reflected_ray = ret;
}
```

Listing 1: Brook+ kernel that calculates the reflection in Linear Algebra

The disassembly of the compiled code can be seen in listing 2. There are five instructions denoted by the numbers 2 to 6. The characters x, y, z, w and t show which of the five ALUs are active in one instruction. In instruction 2 there are four active ALUs of which three calculate a multiplication while ALU t issues a move operation. In contrast instruction 3 has only one active ALU.

```
2  x: MUL_e*2 T0.x, R1.z, R0.z
2  z: MUL_e*2 ____, R1.y, R0.y
2  w: MUL_e*2 ____, R1.x, R0.x
2  t: MOV R2.w, 0.0f
3  y: ADD ____, PV2.w, PV2.z
4  w: ADD ____, PV3.y, T0.x
5  x: MUL_e ____, R0.z, PV4.w
5  y: MUL_e ____, R0.y, PV4.w
5  z: MUL_e ____, R0.x, PV4.w
6  x: ADD R2.x, R1.x, -PV5.z
6  y: ADD R2.y, R1.y, -PV5.y
6  z: ADD R2.z, R1.z, -PV5.x
```

Listing 2: The computational part of the reflection dissassembly

With the advantages of parallel execution of operations inside a thread and the possibility to dissassmble the

compiled code we can directly measure the impact of the parallel nature of GA multivector calculations compared to an algorithm in Linear Algebra. The measurement is possible both in terms of instruction count by using the disassemblies of the according kernels and in terms of execution time during a benchmark. The results in table 3 show that our theoretical considerations which lead to the conclusion that the GA algorithm has advantages on parallel architectures compared to LA were right.

	LA inst	GA inst	GA/LA inst	GA/LA op
II	9	12	1.33	1.29
IP	13	17	1.3	2.0
RF	5	12	2.1	5.2

Table 3: GA/LA instructions and operations in comparision for Intersection Indicator (II), Intersection Point (IP) and Reflection (RF)

The table compares the instruction count of both algebras (LA inst and GA inst). Furthermore the ratio for the instruction count (GA/LA inst) and the observed ratio for the operation count (GA/LA op), which was derived in section 4, between them is shown. It is obvious that the instruction ratio for the intersection point and especially for the reflection vector is significantly smaller than the operation ratio. So the GA multivectors can profit from AMDs architecture that puts up to five operations into one instruction. The value for the intersection indicator doesn't change because its computation can't be parallelized like shown in section 4.

6.3 Performance

We use artificial scenes and real world point clouds for our benchmarks. The artificial scenes are designed to show the impact of different stages in our raytracing algorithm. For the intersection indicator we create a scene consisting of 400 spheres. The spheres are placed in a screen aligned 2D grid with a distance of their center equal to 1.4 times their radius. As a result every ray through a pixel of the image intersects at most two spheres. The scene designed for the intersection consists of 100 screen filling spheres that are placed one behind the other to get a high depth complexity. A single screen filling sphere is used to benchmark the impact of the reflection calculation for shading. The real world scenes are covered by the Egea model and a reduced version of the chameleon point cloud with around 4500 points.

The CPU implementation is consistent with our theoretical observations in section 4. Table 4 shows that the GA needs more time to render the same scene than LA. The difference is not that high like in theory because there is a mixture of different parts of the algorithm even if the scene is designed to show the impact of one special part of the raytracing algorithm. Furthermore the timings show the need for a spacial data structure to render real world scenes.

	II	IP	RF	CH	EG
GA	2320	2190	270	21700	39089
LA	2000	1901	140	19580	34931
GA/LA	1.2	1.2	1.9	1.1	1.1

Table 4: Timings in milliseconds for different scenes on an AMD Athlon 5600+ dual core at 2.8 GHz. Intersection Indicator (II), Intersection Point (IP) and Reflection (RF) are artificial scenes to benchmark the different parts of the raytracing algorithm. The real world scenes are Chameleon (CH) and Egea (EG). The viewport size is 320 by 240

	II	IP	RF	Chameleon	Egea
GA	52	49	14	770	1402
LA	68	62	32	810	1322
GA/LA	0.76	0.79	0.43	0.95	1.06

Table 5: Timings in milliseconds for different scenes on an AMD HD4850 GPU of the brute force approach. Intersection Indicator (II), Intersection Point (IP) and Reflection (RF) are artificial scenes to benchmark the different parts of the raytracing algorithm. The viewport size is 640 by 480

Looking at the results of the GPU brute force implementation presented in table 5 we were somehow surprised. The GA algorithm does not only perform comparable to the LA implementation but is in some cases even faster. An investigation of the dissasemblies of both kernels shows a 5,26% increase in ALU instructions and a 28% increase in control flow instructions for the LA implementation compared to GA. The increase in control flow instructions seems to affect only the artificial scenes with its low object count. The real world scenes are rendered with nearly equal speed.

	Bunny	Max Planck	Dragon
surfel count	19336	96208	166162
time (ms)	160	220	380

Table 6: Performance of the final implementation using OpenCL on an AMD HD4850 GPU. The surfel models were computed from the Standford Bunny, Max Planck and original version of the Phlegmatic Dragon point clouds. The viewport size is 640 by 480

The performance of our final OpenCL implementation presented in table 6 shows that we get interactive frame rates for small and medium sized scenes.

7 CONCLUSION

We were able to show that it is possible to optimize GA code in a way, that its amount of primitive operations is

comparable to that of LA solutions, with a little drawback for GA. In some cases we had to do some handwork to improve the results obtained by Gaalop even further. In the field of reflections it does not seem to be possible to reach a comparable count of operations in the case of conformal space. To clearly increase the performance we have to use the 3D case with the drawback to transform between spaces and to loose some part of the elegance and compactness of GA. So there exists always the task of finding a trade-off between performance and elegance/compactness depending on the kind of application which is developed.

Furthermore by using parrallel architectures one can reach comparable speed for algorithm in GA and LA even without additional handwork. Though the difference in speed for both algebras does not only depend on mathematical effort but on other factors like the amount of control flow instructions.

8 FUTURE WORK

We aim to implement our GA raytracing procedure on a FPGA. Important questions to answer will be how to partition the algorithm between FPGA and a CPU, what spacial data structure to use and the relation between pipeline and parallel architecture. Existing raytracing procedures and GA co-processors on FPGAs reached the capacity of the hardware very fast. So the challenge will be to combine both tasks on a single component.

We also want to compare our introduced surfel model to other surface representations to render point clouds. We aim to improve the visual quality of our model by interpolating neighboring surfels to get a smooth surface without discontinuities. To enhance the rendering speed, which is about an order of magnitude below other algorithms to render point clouds, we will investigate which spatial data structure is suited better to our model than the presented kd-tree.

REFERENCES

[1] AMD. The AMD Stream Technology home page. HTML document http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx, 2009.

[2] Robert Davies. The Newmat home page. HTML document http://www.robertnz.net/nm_intro.htm, 2009.

[3] L. Dorst, D. Fontijne, and S. Mann. *Geometric Algebra for Computer Science, An Object-Oriented Approach to Geometry*. Morgan Kaufman, 2007.

[4] Joshua Fender. A high-speed ray tracing engine built on a field-programmable system. In *Proc. Int. conf. on Field-Programmable Technology, IEEE*, pages 188–195, 2003.

[5] D. Fontijne, T. Bouma, and L. Dorst. Gaigen 2: A geometric algebra implementation generator. http://staff.science.uva.nl/ fontijne/gaigen2.html.

[6] Daniel Fontijne. *Efficient Implementation of Geometric Algebra*. PhD thesis, University of Amsterdam, 2007.

[7] S. Franchini, A. Gentile, M. Grimaudo, C.A. Hung, S. Impastato, F. Sorbello, G. Vassallo, and S. Vitabile. A sliced coprocessor for native Clifford algebra operations. In *Euromico Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, 2007.

[8] Gaël Guennebaud and Markus Gross. Algebraic point set surfaces. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 23, New York, NY, USA, 2007. ACM.

[9] D. Hildenbrand. Geometric computing in computer graphics using conformal geometric algebra. *Computers & Graphics*, 29(5):802–810, 2005.

[10] D. Hildenbrand, H. Lange, Florian Stock, and Andreas Koch. Efficient inverse kinematics algorithm based on conformal geometric algebra using reconfigurable hardware. In *GRAPP conference Madeira*, 2008.

[11] D. Hildenbrand and Joachim Pitt. The Gaalop home page. HTML document http://www.gaalop.de, 2008.

[12] Dietmar Hildenbrand. *Geometric Computing in Computer Graphics and Robotics using Conformal Geometric Algebra*. PhD thesis, Darmstadt University of Technology, 2006.

[13] Intel. The Ct: C for Throughput Computing home page. HTML document http://techresearch.intel.com/articles/Tera-Scale/1514.htm, 2009.

[14] Takashi Kanai, Yutaka Ohtake, Hiroaki Kawata, and Kiwamu Kase. Gpu-based rendering of sparse low-degree implicit surfaces. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 165–171, New York, NY, USA, 2006. ACM Press.

[15] David M. Mount and Sunil Arya. The ANN home page. HTML document http://www.cs.umd.edu/ mount/ANN/, 2009.

[16] NVIDIA. The CUDA home page. HTML document http://www.nvidia.com/object/cuda_home.html, 2009.

[17] Yutaka Ohtake, Alexander Belyaev, and Marc Alexa. Sparse low-degree implicit surfaces with applications to high quality rendering, feature extraction, and smoothing.

[18] C. Perwass. The CLU home page. HTML document http://www.clucalc.info, 2008.

[19] C. Perwass, C. Gebken, and G. Sommer. Implementation of a Clifford algebra co-processor design on a field programmable gate array. In R. Ablamowicz, editor, *CLIFFORD ALGEBRAS: Application to Mathematics, Physics, and Engineering*, Progress in Mathematical Physics, pages 561–575. 6th Int. Conf. on Clifford Algebras and Applications, Cookeville, TN, Birkhäuser, Boston, 2003.

[20] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, pages 343–352, July 2000.

[21] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. Saarcor – a hardware architecture for ray tracing. In *Proceedings of the conference on Graphics Hardware 2002*, pages 27–36. Saarland University, Eurographics Association, 2002. available at http://www.openrt.de.

[22] C. Stoll, S. Gumhold, and H.-P. Seidel. Incremental raycasting of piecewise quadratic surfaces on the gpu. *Symposium on Interactive Ray Tracing*, 0:141–150, 2006.

[23] John Vince. *Geometric algebra: An algebraic system for computer games and animation.* London: Springer. xviii, 195 p, 2009.