EFFICIENT INVERSE KINEMATICS ALGORITHM BASED ON CONFORMAL GEOMETRIC ALGEBRA

Using Reconfigurable Hardware

Dietmar Hildenbrand
Research Center of Excellence for Computer Graphics, University of Technology, Darmstadt, Germany
Dietmar.Hildenbrand@gris.informatik.tu-darmstadt.de

Holger Lange, Florian Stock, Andreas Koch
Embedded Systems and Applications Group, University of Technology, Darmstadt, Germany
koch@esa.informatik.tu-darmstadt.de

Keywords: geometric algebra, geometric computing, computer animation, inverse kinematics, hardware acceleration, reconfigurable hardware, runtime performance.

Abstract: This paper presents a very efficient approach for algorithms developed based on conformal geometric algebra using reconfigurable hardware. We use the inverse kinematics of the arm of a virtual human as an example, but we are convinced that this approach can be used in a wide field of computer animation applications. We describe the original algorithm on a very high geometrically intuitive level as well as the resulting optimized algorithm based on symbolic calculations of a computer algebra system. The main focus then is to demonstrate our approach for the hardware implementation of this algorithm leading to a very efficient implementation.

1 INTRODUCTION

Based on an inverse kinematics application we show that algorithms using the geometrically intuitive mathematical language of conformal geometric algebra can be implemented very efficiently using reconfigurable hardware.

Our starting point is the inverse kinematics algorithm for the arm of a virtual character of the paper (Hildenbrand et al., 2006). It introduces the principle of optimizing geometric algebra algorithms using the symbolic calculation feature of Maple. In section 3 we present the optimized algorithm resulting of this approach. In principle, we describe the algorithm based on the computation of the coefficients of the main data structure of conformal geometric algebra, the 32-dimensional multivector. This kind of optimized algorithms uses only basic arithmetic operations.

These properties provide an adequate basis for our hardware implementation as described in section 4. The optimized algorithm of section 3 uses only operations easily performable by hardware and the computations of the coefficients of the multivectors can be executed in parallel. We use reconfigurable hardware in order to have the chance to configure the hardware acceleration depending on the application.

In a nutshell, the approach of this paper promises to combine the easy development of algorithms based on conformal geometric algebra with very efficient implementations in a wide field of computer animation and computer graphics applications.

2 CONFORMAL GEOMETRIC ALGEBRA

Blades are the basic computational elements and the basic geometric entities of the geometric algebra. For example, the 5D Conformal Geometric Algebra provides a great variety of basic geometric entities to compute with. It consists of blades with **grades** 0, 1, 2, 3, 4 and 5, whereby a scalar is a **0-blade** (blade of grade 0). There exists only one element of grade five in the Conformal Geometric Algebra. It is therefore also called the pseudoscalar. A linear combination of blades is called a **k-vector**. So a bivector is a linear combination of blades with grade 2. Other k-vectors are vectors (grade 1), trivectors (grade 3) and quadvectors (grade 4). Furthermore, a linear combination of blades of different grades is called a multivector. Multivectors are the general elements of a Geometric Algebra. Table 2 lists all the 32 blades of Conformal

Table 1: Representations of the conformal geometric entities

entity	standard repr.	direct repr.
Point	$P = \mathbf{x} + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0$	
Sphere	$s = P - \frac{1}{2}r^2 e_\infty$	$s^* = x_1 \wedge x_2 \wedge x_3 \wedge x_4$
Plane	$\pi = \mathbf{n} + d e_\infty$	$\pi^* = x_1 \wedge x_2 \wedge x_3 \wedge e_\infty$
Circle	$z = s_1 \wedge s_2$	$z^* = x_1 \wedge x_2 \wedge x_3$
Line	$l = \pi_1 \wedge \pi_1$	$l^* = x_1 \wedge x_2 \wedge e_\infty$
Point Pair	$P_p = s_1 \wedge s_2 \wedge s_3$	$P_p^* = x_1 \wedge x_2$

Geometric Algebra. The indices indicate 1: scalar, 2..6: vector 7..16: bivector, 17..26: trivector, 27..31: quadvector, 32: pseudoscalar.

Table 1 presents the basic geometric entities of conformal geometric algebra, points, spheres, planes, circles, lines and point pairs. The s_i represent different spheres and the π_i different planes. The two representations are dual to each other. In order to switch between the two representations, the dual operator which is indicated by '*', can be used. For example in the standard representation a sphere is represented with the help of its center point P and its radius r, while in the direct representation it is constructed by the outer product '$\wedge$' of four points x_i that lie on the surface of the sphere ($x_1 \wedge x_2 \wedge x_3 \wedge x_4$). In standard representation the dual meaning of the outer product is the intersection of geometric entities. For example a circle is defined by the intersection of two spheres ($s_1 \wedge s_2$).

Quaternions are embedded in the Conformal Geometric Algebra in a very intuitive way. The main observation is that an arbitrary line through the origin represents the rotation axis for a quaternion if we use the following definitions for the imaginary units

$$i = e_3 \wedge e_2, \tag{1}$$
$$j = e_1 \wedge e_3, \tag{2}$$
$$k = e_2 \wedge e_1. \tag{3}$$

A rotation around the line L as nomalized rotation axis with an angle of ϕ can be computed as the following quaternion:

$$Q = \cos(\frac{\phi}{2}) + L\sin(\frac{\phi}{2}). \tag{4}$$

For example, if $L = i = e_3 \wedge e_2$, the resulting quaternion

$$\begin{aligned} Q &= \cos(\frac{\phi}{2}) + i\sin(\frac{\phi}{2}) \\ &= \cos(\frac{\phi}{2}) + (\mathbf{e_3} \wedge \mathbf{e_2})\sin(\frac{\phi}{2}) \end{aligned}$$

represents a rotation around the x-axis. For efficiency reasons we use an approach to calculate quaternions

Table 2: A multivector in the 5D Conformal Geometric Algebra is a linear combination of 32 blades. On hardware all its coefficients can be computed in parallel.

Index	blade
1	1
2	e_1
3	e_2
4	e_3
5	e_∞
6	e_0
7	$e_1 \wedge e_2$
8	$e_1 \wedge e_3$
9	$e_1 \wedge e_\infty$
10	$e_1 \wedge e_0$
11	$e_2 \wedge e_3$
12	$e_2 \wedge e_\infty$
13	$e_2 \wedge e_0$
14	$e_3 \wedge e_\infty$
15	$e_3 \wedge e_0$
16	$e_\infty \wedge e_0$
17	$e_1 \wedge e_2 \wedge e_3$
18	$e_1 \wedge e_2 \wedge e_\infty$
19	$e_1 \wedge e_2 \wedge e_0$
20	$e_1 \wedge e_3 \wedge e_\infty$
21	$e_1 \wedge e_3 \wedge e_0$
22	$e_1 \wedge e_\infty \wedge e_0$
23	$e_2 \wedge e_3 \wedge e_\infty$
24	$e_2 \wedge e_3 \wedge e_0$
25	$e_2 \wedge e_\infty \wedge e_0$
26	$e_3 \wedge e_\infty \wedge e_0$
27	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty$
28	$e_1 \wedge e_2 \wedge e_3 \wedge e_0$
29	$e_1 \wedge e_2 \wedge e_\infty \wedge e_0$
30	$e_1 \wedge e_3 \wedge e_\infty \wedge e_0$
31	$e_2 \wedge e_3 \wedge e_\infty \wedge e_0$
32	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$

without the need of using trigonometric functions. The angle between two lines or two planes is defined as follows:

$$\cos(\theta) = \frac{o_1^* \cdot o_2^*}{|o_1^*|\,|o_2^*|}, \tag{5}$$

We already know the cosine of the angle. This is why we are able to compute the quaternion in a more direct way using the following two properties of the trigonometric functions

$$\cos(\frac{\phi}{2}) = \pm\sqrt{\frac{1+\cos(\phi)}{2}} \tag{6}$$

Table 3: Input/output parameters of the inverse kinematics algorithm

parameter	meaning
P_w	target point of wrist
ϕ	swivel angle
d_1, d_2	length of the forearm and the upper arm
Q_s	shoulder quaternion
Q_e	elbow quaternion

and

$$\sin(\frac{\phi}{2}) = \pm\sqrt{\frac{1-\cos(\phi)}{2}}, \tag{7}$$

leading to the formulas

$$\cos(\frac{\phi}{2}) = \pm\sqrt{\frac{1+\frac{o_1^* \cdot o_2^*}{|o_1^*||o_2^*|}}{2}} \tag{8}$$

and

$$\sin(\frac{\phi}{2}) = \pm\sqrt{\frac{1-\frac{o_1^* \cdot o_2^*}{|o_1^*||o_2^*|}}{2}}. \tag{9}$$

The signs of these formulas depend on the application.

For the foundations of conformal geometric algebra and its application to kinematics please refer for instance to (L.Dorst et al., 2007), (Rosenhahn, 2003), (Bayro-Corrochano and Zamora-Esquivel, 2004), (Hildenbrand et al., 2005) and to the tutorials (Hildenbrand et al., 2004) and (Hildenbrand, 2005).

3 THE OPTIMIZED INVERSE KINEMATICS ALGORITHM

In this section we present the optimized inverse kinematics algorithm for the arm of a virtual character as described in (Hildenbrand et al., 2006). We especially use its optimization approach based on Maple in order to get the most elementary relationship between the input and output parameters of the algorithm. The goal of the inverse kinematics algorithm is to compute the quaternions Q_s and Q_e based on the target point P_w and the parameters ϕ, d_1, d_2 (see table 3).

3.1 Compute the swivel plane

First of all, we compute the swivel plane. According to (Tolani et al., 2000) we use the swivel angle ϕ as one free degree of redundancy. The swivel plane is the plane rotated by ϕ around the line L_{sw} through shoulder (at the origin) and P_w (see figure 1). The

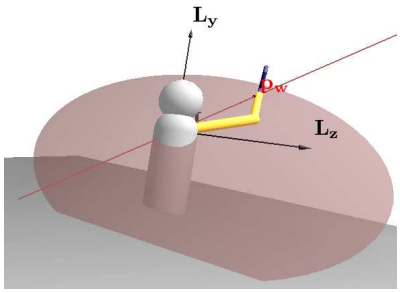

Figure 1: Swivel plane

length of L_{sw} can be computed as

$$|L_{sw}| = \sqrt{p_{wx}^2 + p_{wy}^2 + p_{wz}^2}. \tag{10}$$

The coefficients of the swivel plane optimized by Maple are:

$$\begin{aligned}
\pi_{swivel\,x} =& (2\cos\frac{\phi}{2}\sin\frac{\phi}{2}p_{wz}p_{wx} - p_{wy}|L_{sw}| + \\
& 2p_{wy}|L_{sw}|\cos\frac{\phi}{2}^2)/|L_{sw}| \\
\pi_{swivel\,y} =& (2\cos\frac{\phi}{2}\sin\frac{\phi}{2}p_{wz}p_{wy} + p_{wx}|L_{sw}| - \\
& 2p_{wx}|L_{sw}|\cos\frac{\phi}{2}^2)/|L_{sw}| \\
\pi_{swivel\,z} =& \frac{-2\sin\frac{\phi}{2}\cos\frac{\phi}{2}(p_{wx}^2 + p_{wy}^2)}{|L_{sw}|}
\end{aligned} \tag{11}$$

3.2 The elbow point P_e

With the help of the two spheres $S_1 = P_w - \frac{1}{2}d_2^2 e_\infty$ and the sphere $S_2 = e_o - \frac{1}{2}d_1^2 e_\infty$ with center points P_w and e_o and radii d_2, d_1 we are able to compute the circle $Z_e = S_1 \wedge S_2$ determining all the possible locations of the elbow as the intersection of the spheres (see table 1). The intersection with the swivel plane delivers the

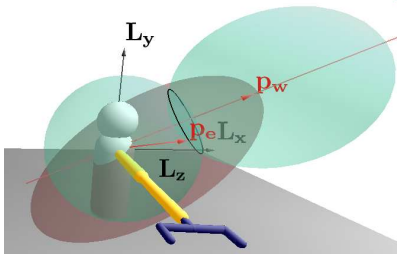

Figure 2: Compute the elbow point

point pair $Pp = Z_e \wedge \pi_{swivel}$.

Its coefficients based on the optimizations of Maple are as follows

$$
\begin{aligned}
PP_i &= \frac{1}{2}\pi_{swivel_x}(p_{w_x}^2 + p_{w_z}^2 + p_{w_y}^2 + d_1^2 - d_2^2)\\
PP_j &= \frac{1}{2}\pi_{swivel_y}(p_{w_x}^2 + p_{w_z}^2 + p_{w_y}^2 + d_1^2 - d_2^2)\\
PP_k &= \frac{1}{2}\pi_{swivel_z}(p_{w_x}^2 + p_{w_z}^2 + p_{w_y}^2 + d_1^2 - d_2^2)\\
PP_{14} &= \frac{1}{2}(1 - d_1^2)(p_{w_y}\pi_{swivel_z} - p_{w_z}\pi_{swivel_y})\\
PP_{15} &= \frac{1}{2}(1 + d_1^2)(p_{w_z}\pi_{swivel_y} - p_{w_y}\pi_{swivel_z})\\
PP_{24} &= \frac{1}{2}(1 - d_1^2)(p_{w_z}\pi_{swivel_x} - p_{w_x}\pi_{swivel_z})\\
PP_{25} &= \frac{1}{2}(1 + d_1^2)(p_{w_x}\pi_{swivel_z} - p_{w_z}\pi_{swivel_x})\\
PP_{34} &= \frac{1}{2}(d_1^2 - 1)(p_{w_y}\pi_{swivel_x} - p_{w_x}\pi_{swivel_y})\\
PP_{35} &= \frac{1}{2}(1 + d_1^2)(p_{w_y}\pi_{swivel_x} - p_{w_x}\pi_{swivel_y}) \quad (12)
\end{aligned}
$$

We decide for one of the two possible elbow points and compute the three components of the elbow point p_{ex}, p_{ey}, p_{ez} from the point pair PP:

$$
\begin{aligned}
einf\_PP =& (PP_{35} - PP_{34})^2 + (PP_{14} - PP_{15})^2\\
&+ (PP_{25} - PP_{24})^2\\
tmp_1 =& -PP_i^2 - PP_j^2 - PP_k^2 - PP_{14}^2 +\\
&PP_{15}^2 - PP_{24}^2 + PP_{25}^2 - PP_{34}^2 + PP_{35}^2\\
tmp_{sqrt} =& \sqrt{tmp_1}\\
p_{ex} =& (PP_j(PP_{34} - PP_{35}) + PP_k(PP_{25} - PP_{24}\\
&+ tmp_{sqrt}(PP_{15} - PP_{14}))/einf\_PP\\
p_{ey} =& (PP_i(PP_{35} - PP_{34}) + PP_k(PP_{14} - PP_{15})\\
&+ tmp_{sqrt}(PP_{25} - PP_{24}))/einf\_PP\\
p_{ez} =& (PP_j(PP_{15} - PP_{14}) + PP_i(PP_{24} - PP_{25})\\
&+ tmp_{sqrt}(PP_{35} - PP_{34}))/einf\_PP \quad (13)
\end{aligned}
$$

3.3 Calculate the elbow quaternion Q_e

The elbow angle θ_4 is computed with the help of the line $L_{se} = (e_o \wedge P_e \wedge e_\infty)^*$ through the shoulder and the elbow and the line $L_{ew} = (P_e \wedge P_w \wedge e_\infty)^*$ through the shoulder and the wrist. Based on these two lines we are able to compute the angle between them ($c_4 = \cos(\theta_4) = \frac{L_{se}^* \cdot L_{ew}^*}{|L_{se}^*||L_{ew}^*|}$) according to equation (5).

Now, we are able to compute the quaternion $Q_e = \cos(\theta_4/2) + \sin(\theta_4/2)i$ according to equation (1). It represents a rotation around the local x-axis with the angle θ_4. The optimized version of this quaternion is $Q_e = \sqrt{\frac{1+c_4}{2}} - \sqrt{\frac{1-c_4}{2}}i$, according to (8) and (9). This

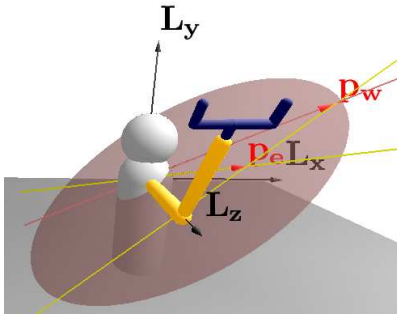

Figure 3: Use the elbow quaternion

quaternion rotates the upper arm corresponding to the angle between the two yellow lines as shown in figure 3.

The quaternion Q_e of the rotation at the elbow joint:

$$
\begin{aligned}
Q_e =& \sqrt{\frac{1 + \frac{p_{e_x}^2 - p_{e_x}p_{w_x} + p_{e_y}^2 - p_{e_y}p_{w_y} - p_{e_z}p_{w_z} + p_{e_z}^2}{d_1 d_2}}{2}}\\
&- \sqrt{\frac{1 - \frac{p_{e_x}^2 - p_{e_x}p_{w_x} + p_{e_y}^2 - p_{e_y}p_{w_y} - p_{e_z}p_{w_z} + p_{e_z}^2}{d_1 d_2}}{2}}\, i
\end{aligned}
\quad (14)
$$

3.4 Rotate to the elbow position

At first we calculate the middle line L_m through the origin within the same distance from the points P_e and $P_{ze} = d_1 e_3 + \frac{1}{2}d_1^2 e_\infty + e_o$. We will need this line L_m in the next step in order to rotate around this line.
To compute L_m, we use the middle plane as the difference of the two points P_e and P_{ze} ($\pi_m = P_{ze} - P_e$) and the plane through the origin and the points P_e and P_{ze} as $\pi_e^* = e_o \wedge P_{ze} \wedge P_e \wedge e_\infty$ and intersect them ($L_m = \pi_e \wedge \pi_m$).

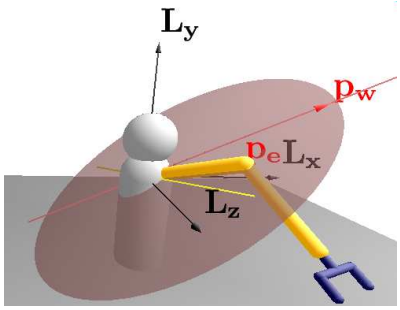

Figure 4: Rotate to the elbow position

In order to rotate the elbow towards our already computed point P_e we have to rotate around the middle line of the previous step with angle π. This results

in a quaternion identical with the normalized middle line ($Q_{12} = \frac{L_m}{|L_m|}$). Figure 4 shows this rotation from the z-axis L_z to the elbow point with the help of the yellow middle line.

The result for the quaternion Q_{12} optimized by Maple can be computed as follows:

$$\begin{aligned}
tmp_2 =& d_1^4 p_{e_y}^2 - 2d_1^3 p_{e_y}^2 p_{e_z} + d_1^2 p_{e_y}^2 p_{e_z}^2 + \\
& d_1^4 p_{e_x}^2 - 2d_1^3 p_{e_x}^2 p_{e_z} + d_1^2 p_{e_x}^2 p_{e_z}^2 + \\
& d_1^2 p_{e_x}^4 + 2d_1^2 p_{e_x}^2 p_{e_y}^2 + d_1^2 p_{e_y}^4 \\
|L_m| =& \sqrt{tmp_2} \\
q_{12_i} =& \frac{d_1 p_{e_x}(d_1 - p_{e_z})}{|L_m|} \\
q_{12_j} =& \frac{d_1 p_{e_y}(d_1 - p_{e_z})}{|L_m|} \\
q_{12_k} =& \frac{d_1 (p_{e_x}^2 + p_{e_y}^2)}{|L_m|}
\end{aligned} \tag{15}$$

3.5 Rotate to the wrist location

The angle θ_3 (and the resulting quaternion Q_3) is computed with the help of the y-z-plane rotated by the quaternion Q_{12} and the swivel plane. The plane in y and z direction (with normal vector e_1 and zero distance to the origin), is computed by $\pi_{yz} = e_1$. The rotated plane π_{yz2} results in $\pi_{yz2} = Q_{12}\ \pi_{yz}\ \tilde{Q}_{12}$. Based

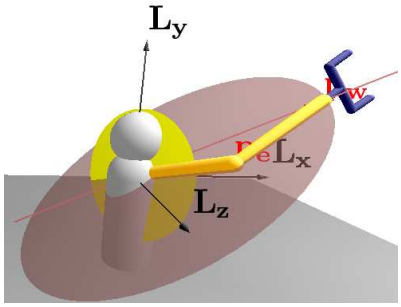

Figure 5: Rotate to the wrist location

on these two planes we are able to compute the angle between them as $c_3 = \cos(\theta_3) = \frac{\pi^*_{yz2} \cdot \pi^*_{swivel}}{|\pi^*_{yz2}||\pi^*_{swivel}|}$ according to equation (5) and we get the quaternion $Q_3 = \cos(\theta_3/2) + \sin(\theta_3/2)k$. It represents a rotation around the local z-axis with the angle θ_3. The optimized version of this quaternion is

$$Q_3 = \pm\sqrt{\frac{1+c_3}{2}} + \sqrt{\frac{1-c_3}{2}}k, \tag{16}$$

according to (8) and (9).
Note: the sign of this quaternion depends on which side of the plane π_{yz2} the point P_w is lying. This can be easily computed with the help of the inner product $\pi_{yz2} \cdot P_w$. This quaternion rotates the arm to the wrist location as shown in figure 5.

The rotation at the shoulder joint optimized by Maple can be computed as follows:

$$\begin{aligned}
c_3 =& (\pi_{swivel_x}(q_{12_k}^2 + q_{12_j}^2 - q_{12_i}^2) - \\
& 2q_{12_i}(q_{12_j}\pi_{swivel_y} + q_{12_k}\pi_{swivel_z})) \\
& / \sqrt{\pi_{swivel_x}^2 + \pi_{swivel_y}^2 + \pi_{swivel_z}^2} \\
sign =& p_{w_x}(q_{12_i}^2 - q_{12_k}^2 - q_{12_j}^2) + \\
& 2q_{12_i}(q_{12_k}p_{w_z} + q_{12_j}p_{w_y}) \\
sign =& \frac{sign}{|sign|} \\
q_{3_{scalar}} =& \sqrt{\frac{1+c_3}{2}} \\
q_{3_k} =& \sqrt{\frac{1-c_3}{2}}\, sign
\end{aligned} \tag{17}$$

The resulting quaternion for the shoulder rotation can now be computed as the product $Q_s = Q_{12}Q_3$. The final result of Q_s optimized by Maple can be computed as follows:

$$\begin{aligned}
Q_s =& -q_{12_k} \cdot q_{3_k} + \\
& (q_{12_i} \cdot q_{3_{scalar}} + q_{12_j} \cdot q_{3_k})i - \\
& (q_{12_i} \cdot q_{3_k} - q_{12_j} \cdot q_{3_{scalar}})j + \\
& (q_{12_k} \cdot q_{3_{scalar}})k
\end{aligned} \tag{18}$$

4 HARDWARE IMPLEMENTATION

The previous section describes the inverse kinematics algorithm, originally developed based on conformal geometric algebra, with the help of the equations (10) to (18) using only basic arithmetic operations. In principle they describe the computation of the components of the 32-dimensional multivectors of the algebra. Equation (12) for instance computes the 9 components of a point pair. On hardware all these computations can be executed in parallel.

These equations provide an adequate basis for the following hardware implementation.

4.1 The Hardware Platform

The accelerated hardware implementation targets a Field Programmable Gate Array (FPGA). These reconfigurable devices allow the implementation of custom circuits, but provide only limited resources. The

Xilinx Virtex 2/4 series of FPGAs comprises an array of lookup-tables (LUTs) with one associated Flip-Flop (FF) per LUT. Each LUT can implement an arbitrary function of up to four independent 1-bit inputs.

To fit the calculation on the FPGA and speed it up, we decided to use only fixed point calculations instead of floating point operations (which is possible but not as efficient as fixed point). As a side effect of calculating in dedicated hardware, the cost of operations drastically changes: e.g. multiplication and addition need the same calculation time, whereas reciprocal value, division and square root are really costly in both execution time and FPGA resources.

4.2 Preparatory Optimization

The equations described in the previous section were the starting point for further optimization. They where manually optimized to reduce the number of operations, especially the expensive ones. E.g. we changed the last six equations of (12) to

$$\begin{aligned} PP_m^+ &= PP_{m5} + PP_{m4} \\ &= 2d_1^2(p_{w_{...}}\pi_{swivel_{...}} - p_{w_{...}}\pi_{swivel_{...}}) \\ PP_m^- &= PP_{m5} - PP_{m4} \\ &= 2(p_{w_{...}}\pi_{swivel_{...}} - p_{w_{...}}\pi_{swivel_{...}}) \\ & \quad m = 1,2,3 \end{aligned} \tag{19}$$

These six equations comprise the same number of (constant) multiplications and additions, so the calculations seem as expensive as before. However, now the PP_m^+ and PP_m^- share a common subexpression, so 2 multiplications and 1 addition are saved. Furthermore, the constant multiplication in PP_m^+ is now a constant multiplication with a power of 2, which is implemented in hardware efficiently through simply shifting wire connections by the exponent value. Since only pure wiring is involved, no additional ressources are required for this kind of multiplication.

In the following computation of the elbow points in (13), the frequent reference to PP_{m4} and PP_{m5} is clearly obvious. However, as the references occur only in a sum or difference, the new PP_m^+ and PP_m^- can be used instead, thus eliminating an addition each time. Only tmp_1 does not refer directly to a sum or difference, but it can be expressed as

$$tmp_1 = \ldots + PP_m^+ PP_m^- + \ldots,$$

eliminating another 3 multiplications and 3 additions.

For a comparison of the required operations before and after optimization see table 4.

4.3 Fixed Point Conversion

The optimized equations from the previous section (4.2) were manually transformed into dataflow graphs. For conversion into fixed point format suitable for efficient hardware realization, we need to define an as small as possible word length for every variable and intermediate result in these graphs (e.g., see figure 6 calculation of p_{e_x} from equation (13)). While automatic conversions from floating point to fixed point exist (Han, 2006), we preferred a manual translation nevertheless, as automatic conversions are still limited (e.g. not all operations are supported). We used two approaches for minimizing the word length while retaining sufficient precision.

	$+,-$	$\times$	$\sqrt{}$	$\frac{a}{b}, \frac{1}{b}$
unoptimzed	147	87	8	16
optimized	69	42	8	10

Table 4: Comparison of required operations

Analytic approach: The input parameters have a given precision and value range which were propagated forward through the data flow graphs. On the other hand, the expected precision and value range of the *result* were propagated backwards (e.g. the precision of a sum is the maximum of the precision of the summands). Figure 6 shows the resulting value ranges as annotations to the nodes.

Moreover, we can benefit from dedicated knowledge of the problem and inequalities to narrow the value range even further. Referring to the graph in figure 6, we know that the resulting point p_{e_x} is the elbow. Hence, the distance is given by d_1, which also implies that $|p_{e_x}| \leq d_1$, $|p_{e_y}| \leq d_1$ and $|p_{e_z}| \leq d_1$ (narrowed value ranges are shown inside dashed frames at the nodes in figure 6).

Empiric approach: To verify the analytical analysis (and obtain results where the analytic approach fails), all graphs were implemented in *MATLAB*. Then random valid values were supplied as inputs to the equations, which were calculated subsequently with default Matlab double precision (= 64 bits) floating point arithmetic as well as the *Fixed-Point Toolbox* (The MathWorks, 2007). The Fixed-Point Toolbox allows exact specification of the number format used for fixed point arithmetic (sign bit, integer and fractional word length) for each variable.

The analyses show that it is sufficient to use a total word length of 32 bits or less.

4.4 Hardware Realization

The dataflow graphs were implemented as a fully *spatial pipeline* in the hardware description language Verilog. *Spatial parallelism* means parallel execution of operations that can be performed simultane-

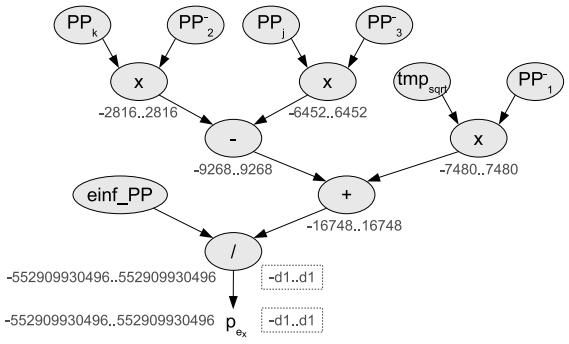

Figure 6: Dataflow graph for p_{ex}

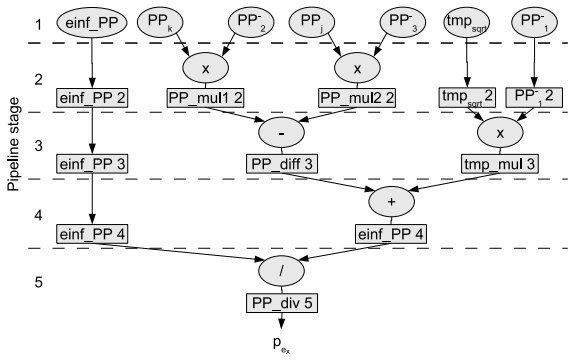

Figure 7: Pipeline schedule for p_{ex} (cf. figure 6)

ously due to independent subexpressions. *Pipelining* further extends parallel execution to simultaneous calculation of sequentially dependent operations. To this end, the parallel-sequential dataflow graph is partitioned into many sequential execution steps, also known as *pipeline stages*. The results of the first execution step are used as inputs for the second step. This scheme is repeated for all subsequent operation steps. Hence, parallel execution for *all* operations is exploited if possible.

As a benefit, the execution time for each pipeline stage is minimized with respect to the whole sequential dataflow graph. The faster pipeline stages, executing in *parallel*, allow for a higher clock frequency of the hardware, thus increasing overall throughput with a set of results delivered at each clock cycle. The *latency* is only experienced twice at pipeline startup and shutdown, or respectively, in single shot mode.

We chose to optimize the pipeline for maximum throughput to demonstrate the potential of accelerating the algorithm in hardware. Nevertheless other optimization goals (minimum use of hardware resources/area, low latency, low power consumption) are possible as well, but not discussed here.

Many subexpression trees have different heights

Number of LUTs	$\approx$ 30000 (45% of 2VP70 max.)
Clock frequency	100 MHz
Throughput	1 set of results per clock cycle
Latency	$\approx$ 550 clock cycles

Table 5: Hardware mapping results

which mostly correspond to inhomogeneous numbers of pipeline stages from common input values to intermediate values, the latter to be merged for subsequent calculations. To account for this difference, the pipeline was balanced with additional flip-flop stages on the shorter subtrees, hence providing intermediate results synchronously at such *merge points*. The exact pipline timing (also known as *schedule*) for p_{ex} (cf. 13) is shown in figure 7. The rectangles represent flip-flops which temporarily store the intermediate results of the 5 pipline stages (separated by dashed horizontal lines). Note the balancing flip-flops for $einf\_PP$, tmp_{sqrt} and PP_1^-.

In contrast to a fully spatial pipeline which is customized for the specific application, general purpose instruction-based execution models as employed in CPUs or Digital Signal Processors (DSPs) only provide for a very limited exploitation of both spatial and sequential (pipelining) parallelism. E.g., a CPU pipeline has to be flushed and refilled on every (unpredicted) branch in the control flow of the software code. What is more, even a superscalar CPU hardly executes more than half a dozen instructions per clock cycle (Hennessy and Patterson, 2007, chapter 3.6).

5 RESULTS

The Verilog HDL description of the dataflow graph was mapped to a Xilinx Virtex 2VP70 FPGA (Xilinx, 2005) using the Xilinx ISE tools. For synthesis, we generated dedicated dividers and square-root cores with the Xilinx Core Generator (part of the ISE tools). Multiplications were automatically mapped to *multiplier units*, which are specialized, non-reconfigurable fast hardware blocks provided within the FPGA fabric. Table 5 shows the mapping and performance results for the complete dataflow graph.

Our test scenario was the computation of the motion of the arm of the virtual character based on 100 steps of inverse kinematics computations. To evaluate the hardware performance compared with a pure software implementation, 100 data sets residing in main memory which represent target points in space are converted to quaternions by both our fully spa-

	execution time
SW on CPU	860 us
HW pipeline	6 us (5 us until first data set + 100 * 10 ns remaining sets)

Table 6: SW and HW execution times for 100 data sets

tial pipeline running at 100 MHz clock speed and a Intel Centrino M715CPU running at 1.5 GHz. After conversion, the resulting data sets are written back to memory. Table 6 shows the accelerated HW and pure SW execution times.

6 Comparison to GPU Realisation

The fined-grained parallelism exploited by our FPGA realisation is very different from the coarser-grained one accessible when implementing the algorithm on a modern GPU. A typical example, such as the NVidia G80 architecture (NVIDIA Corp., 2007), can process just 128 operations in parallel, using a SIMD (small-vector) paradigm. In our approach, however, each of the 550 pipeline stages has $4 \ldots 10$ operators executing in parallel, leading to a total parallelism of *thousands* of operations. Additionally, the GPU processing elements have a fixed architecture and cannot be tailored to specific constants or precision requirements. Furthermore, the GPU computation has to be *manually* partitioned into units of parallel execution (so-called threads or warps). In our approach, the mathematical formulation itself directly determines the hardware structure of the accelerator, no additional partitioning is required. While we have not implemented the algorithm on such a GPU yet, we expect the FPGA realisation (especially one using a more recent chip than the one in the current prototype) to be competitive with a modern GPU implementation despite the clock speed differences (550 MHz FPGA vs. 1350 MHz GPU). Such an evaluation is planned in a further refinement of this work.

7 CONCLUSION

We presented a way to implement algorithms based on conformal geometric algebra on hardware. After having developed an algorithm easy and geometrically intuitive on a very high level we are using the symbolic calculation functionality of Maple as a software optimization procedure. While this approach is already leading to very efficient code we presented an approach to further improve the performance using configurable hardware. This implementation turned out to be more than 100 times faster. These results were shown based on an inverse kinematics algorithm but we are convinced that this approach can be used very advantageously in a lot of computer animation and computer graphics applications.

REFERENCES

Bayro-Corrochano, E. and Zamora-Esquivel, J. (2004). Inverse kinematics, fixation and grasping using conformal geometric algebra. In *IROS 2004, September 2004, Sendai, Japan*.

Han, K. (2006). *Automating Transformations from Floating-point to Fixed-point for Implementing Digital Signal Processing Algorithms*. PhD thesis, Dept. of Electrical and Computer Engineering, The University of Texas at Austin.

Hennessy, J. L. and Patterson, D. A. (2007). *Computer architecture*. Kaufmann [u.a.], Amsterdam [u.a.].

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

Hildenbrand, D., Bayro-Corrochano, E., and Zamora-Esquivel, J. (2005). Advanced geometric approach for graphics and visual guided robot object manipulation. In *proceedings of ICRA conference, Barcelona, Spain*.

Hildenbrand, D., Fontijne, D., Perwass, C., and Dorst, L. (2004). Tutorial geometric algebra and its application to computer graphics. In *Eurographics conference Grenoble*.

Hildenbrand, D., Fontijne, D., Wang, Y., Alexa, M., and Dorst, L. (2006). Competitive runtime performance for inverse kinematics algorithms using conformal geometric algebra. In *Eurographics conference Vienna*.

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

NVIDIA Corp. (2007). *NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 1.0*. NVIDIA Corp.

Rosenhahn, B. (2003). *Pose Estimation Revisited*. PhD thesis, Inst. f. Informatik u. Prakt. Mathematik der Christian-Albrechts-Universität zu Kiel.

The MathWorks (2007). *Fixed-Point Toolbox 2, Reference*. The MathWorks.

Tolani, D., Goswami, A., and Badler, N. I. (2000). Real-time inverse kinematics techniques for anthropomorphic limbs. *Graphical Models*, 62(5):353–388.

Xilinx (2005). *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet (DS083)*. Xilinx.