Fast Distance Queries for Triangles, Lines, and Points using SSE Instructions

Evan Shellshear Robin Ytterlid
FCC

Abstract

This paper presents a suite of routines for computing the distance between combinations of triangles, lines and points that we optimized for the x86 SSE SIMD (vector) instruction set. We measured between two and seven times throughput improvement over the naive non-SSE optimized routines.

1. Introduction

In recent years, SIMD utilization has become a major theme of high performance computing. Modern CPUs support SIMD extensions that can be used to gain some of the benefits of many-core computation while avoiding some of the drawbacks. SIMD extensions are CPU implementations of the SIMD architecture described in Flynn's Taxonomy [Flynn 1972], and they allow a CPU core to perform a single instruction on multiple pieces of data in parallel, while maintaining the low response time inherent in CPU processing. A benefit of SIMD is the possibility to combine SIMD extensions with threaded multi-core CPU processing for even greater exploitation of parallelism. Intel's Streaming SIMD Extensions (SSE) is a SIMD instruction set that is supported by the great majority of modern processors, and it utilizes 128-bit registers that allows a core to perform up to four single-precision floating point operations at once. Although the SSE instruction set can process at most four floating-point values at a time, with the more recent AVX and AVX2 instructions it is possible to process up to eight floating-point values in parallel. Thakkur and Huff [1999] gives a more in-depth description of SSE and SIMD extensions in general.

When it comes to basic, low-level computer graphics routines, there is great interest in exploiting SSE, with articles exploring fast SIMD-based ray-triangle intersection tests [Havel and Herout 2010], sphere-box intersection tests [Larsson et al. 2007], etc. Mostly, such tests focus on using SSE in a more serial fashion where one

 ISSN 2331-7418

exploits SSE functions while testing one object against another. The potential problem is that compilers are growing so much better at optimizing code that hand-written serial optimizations may soon be unnecessary (depending on the application). This is why we focus on a more parallel approach to SIMD, whereby instead of simply trying to improve a single test, we use SIMD to be able carry out up to four distance tests at once. The benefit of such an approach is a more natural utilization of all SIMD registers (for three dimensional problems) than utilizing three registers for coordinate data and wasting the final register, [Havel and Herout 2010].

In this article we present new SIMD-based triangle-triangle distance routine to efficiently exploit the x86 SSE instructions. Achieving this goal, however, required us to develop a number of other SSE based distance routines such as triangle-point distance, line segment-line segment distance and triangle-triangle collision. We present all the other necessary algorithms before presenting our SSE based triangle-triangle distance routine. In addition, we avoid bloating this paper with comments more than is absolutely necessary because the basic idea behind each routine is not new. We have simply taken existing C++ code and converted it (with the necessary small modifications and optimizations) to SSE friendly code using a Structure-of-Arrays (SoA) approach. Where changes are made that significantly effect the performance of the code and these changes deviate from the original code then we point this out. Otherwise the reader is referred to the original articles/code to see detailed explanations of the code flow and meaning.

The motivation for our routines comes from scenarios where it is important to have fast distance queries between triangle meshes for path planning [Hermansson et al. 2013; Spensieri et al. 2008] in robotics, [Spensieri et al. 2013]. Routines such as those presented here can be used for bounding volume hierarchies to speed up the final distance routines, [Shellshear et al. 2013].

In the next section we describe our new SSE based primitive routines. After that, in Section 3 we present our experimental setup and benchmark cases. In Section 4 we then present our results and in Section 5 we conclude. We also supply supplementary code for our routines in Section 6.

2. SIMD-Based Geometry Routines

We approach our SIMD based distance tests by using a SoA data layout, [Klimovitski 2001], instead of the Array-of-Structures (AoS) data layout. This means that instead of storing one array of three dimensional coordinates specifying our points, lines or triangles, we store three arrays containing sets of x, y and z coordinates, respectively. The approach we use (for SSE) is to load the x coordinates from four triangles into one SSE register, and the y and z coordinates from the same triangles into two other registers, then process all four triangles in parallel. All code described in this section

can be found in the code listings of Section 6.

SSE code can be written directly using inline assembly language instructions or using compiler intrinsics. However, to simplify the coding and the notation in this article, the Embree package (version 1.1) [Intel 2014] was used to provide basic vector functions via operator overloading, etc. In code listing 5, we present the definitions of the basic data SSE structures and their respective operations. These data structures are from the Embree package and are the basic building blocks from which all other data structures are built. For the sake of illustration, some of the basic functions included in these data structures are also given. To see all the included functions, the reader is referred to the Embree project source.

From the `ssef` data structure defined in code listing 5, we create the SoA that is used to store the data for multiple points, lines and triangles. These collections of primitives are in turn based on another data structure given in Embree, `Vec3.` `Vec3` has a number of overloaded functions to allow one to use SIMD data structures in the template. Examples of a few such functions are also given in code listing 6.

The two basic types `ssef` and `Vec3` were used to define other data structures such as our SSE points, lines and triangles. See the code listing 7 for the definition of each data type.

Each of our distance routines is based on rewriting the fastest known (to the authors) distance tests into a SIMD friendly format. We compared well-known routines as well as those published at Geometric Tools website [Eberly 2007], with the ones presented here. The line segment-line segment and triangle-point distance tests were adapted from the routines presented in the book *Real-Time Collision Detection* [Ericson 2004]. This turned out to be faster than basing the code on the routines at the Geometric Tools website and was mostly likely due to the significant branching present in the routines there. The triangle-triangle distance test is taken from the PQP [Larsen et al. 2014] source code, as our SIMD version of this code turned out to be faster than our SIMD version of the triangle-triangle distance routine presented on the Geometric Tools website. The authors are unaware of the triangle-triangle distance routine from the PQP source code having been published via any outlet other than as source code on the previously mentioned reference.

In the following subsections we briefly describe the algorithms used in the original code from *Real-Time Collision Detection* and PQP, and the changes that we made to make it more SSE friendly and to improve performance. We used the SSE intrinsic instructions via the Embree wrapper functions to write the code and used instructions from versions up to and including SSE4. All routines return the squared distance between the primitives, hence the 2 after each name. The only remaining function needed to understand the code in the subsections is our clamp function in code listing 8.

2.1. Line Segment-Line Segment Distance Test

The first distance test we present code for is the line segment-line segment distance routine in code listing 9. The SIMD version is a simple adaption of the well-known distance test presented in the book *Real-Time Collision Detection* [Ericson 2004].

A brief outline of the distance test is as follows. The closest points between two segments is found by finding a vector perpendicular to both line segments. This can always be found by infinitely extending the lines in both directions. If the two found closest points lie within the line segments then the computation is finished. If not, then one needs to be careful when clamp the closest points to the segments so that the nearest points on the line segments are found. More details can be found in the book *Real-Time Collision Detection*.

To make the code more SIMD friendly, a lot of the branching for handling degenerate line segments has been replaced by hard minimum and maximum values in our SSE implementation to prevent unnecessary branching. Although this can result in slightly different results in degenerate cases, in all tests performed here (which included such degenerate cases) we noticed no difference in the final results. In numerous places we also utilized the `select` function provided by Embree which is a wrapper for the *_mm_blendv_ps* intrinsic to make sure the values are chosen for the correct triangles. However, the Embree version swaps the order of the first two arguments in *_mm_blendv_ps*, i.e. $\_mm\_blendv\_ps(a,b,mask) = select(mask,b,a)$. By doing so we managed to avoid code such as in code listing 1 which can be SIMD unfriendly due to branching. Such code was replaced with much more SIMD friendly code such as that in code listing 2. Note that the code in code listing 1 has been optimized for the different t values which is why there is no t value in the computation for s.

```
if (t < 0.0f) {
  t = 0.0f;
  s = clamp(-c / a, 0.0f, 1.0f);
} else if (t > 1.0f) {
  t = 1.0f;
  s = clamp((b - c) / a, 0.0f, 1.0f);
}
```

Listing 1. Standard clamping code for segment-segment distance

```
const simdFloat newT = clamp(t, simdFloat(zero), simdFloat(one));
simdBool mask = (newT!=t);
s = select(mask, clamp((newT*b - c) / a, simdFloat(zero),
    simdFloat(one)), s);
```

Listing 2. SIMD friendly clamping code for segment-segment distance

Finally, the code also uses the Embree defined constant ulp which is defined by $std :: numeric\_limits < float >:: epsilon()$, e.g. a minimal, positive value, to avoid divisions by zero.

2.2. Triangle-Point Distance Test

The second distance test we present code for is the triangle-point distance routine in code listing 10. The SSE version is a simple adaption of the well-known distance test presented in the book *Real-Time Collision Detection*. A brief outline of the distance test is as follows. To find the closest point on a triangle to a given point, we first project the given point onto the triangle's plane. If the point orthogonally projects inside the triangle, then the projection point is the closest point to the given point. If the given point projects outside the triangle, then the closest point must instead lie on one of the triangle's edges. To find which edge, one computes which of the triangle's Voronoi feature regions the given point is in.Once determined, only the orthogonal projection of the given point onto the corresponding feature must be computed to find the closest point.

The main difference between the original version and our version is that after each computation of the closest point for each Voronoi region of a triangle, we then check if that point is the closest point, closest points found so far and if so, set the corresponding bit in a mask variable (maskX, X = 1,...,6). We compute the bitwise OR of this mask with all previous masks (six in total) until all closest points have been found (all mask bits set) or until we reach the end of the function. So instead of the code in code listing 3 we have the code in code listing 4 with additional bitwise OR operations for each future test.

```
1 if (d1 <= 0.0f && d2 <= 0.0f) return a; // barycentric coordinates
      (1,0,0)
2 ...
3 if (d3 >= 0.0f && d4 <= d3) return b;  // barycentric coordinates
      (0,1,0)
```

Listing 3. Branching example from triangle-point distance code

```
1   const simdBool mask1 = (d1 <= simdFloat(zero)) & (d2 <= simdFloat
        (zero));
2   oTriPoint = iTri[0];
3   simdBool exit(mask1);
4   if(all(exit))
5     return length2(oTriPoint - iPoint);  // barycentric coordinates
          (1,0,0)
6   ...
7   exit |= mask2;
8   oTriPoint = select(mask2, iTri[1], oTriPoint);
9   if(all(exit))
```

```
10      return length2(oTriPoint - iPoint);  // barycentric coordinates
            (0,1,0)
```

Listing 4. SIMD friendly branching example from triangle-point distance code

2.3. Triangle-Triangle Distance Test

The final routine we present code for is the triangle-triangle distance routine. This function is adapted from the C++ code in the PQP [Larsen et al. 2014] source code, to SSE optimized code. It performs similar steps as in the original code but the routine has been optimized for SIMD style computations. A brief outline of the distance test is as follows. The triangle-triangle distance test is based on an initial fast test to see if the closest points are between an pair of edges of the two triangles. If this is unsuccessful then we know that no edge pairs contain the closest points. Hence, we know that either

- the closest points must be between the vertex of a triangle and the face of the other triangle,
- the triangles are intersecting,
- the edge of one triangle is parallel to the other's face (in this case we can take the closest points from the initial computation),
- the triangles are degenerate.

Each of these cases is then tested in turn to determine which case we are in.

To be able to carry out the distance test we require a triangle-triangle intersection function which we present in code listing 11. This function was also adapted from the triangle-triangle intersection function in the PQP software, which takes two sets of three triangle vertices as input and outputs whether the two triangles are colliding or not via separating axis tests. Our version of the function simply has each of the C++ data types and functions swapped for SSE data types and functions. Again we are unaware of the original algorithm being published anywhere apart from in the aforementioned software. We also do not claim this to be the fastest routine for triangle-triangle intersection, however, the routine is simple to convert to SSE code due to the only branching occurring as early-outs. Additionally, this routine contributed less than five percent of total computation time in the triangle-triangle distance routine. Hence we leave it to future research to investigate the quality of this routine in comparison to SSE versions of other routines [Möller 1997; Guigue and Devillers 2003], as intersection testing is not the focus of this paper.

The triangle-triangle distance test uses a number of smaller subfunctions that carry out a number of repetitive tasks. In code listing 13, we present the triangle-

triangle distance function first and then two smaller subfunctions. The first subfunction, `closestEdgeToEdge`, computes for four triangle pairs the closest distance between all edges in the first triangle and one particular edge in the second triangle. The second subfunction, `closestVertToTri`, computes distances between four triangles and four single triangle vertices.

As is the case with the line segment-line segment and triangle-point distance tests, the major differences between the original triangle-triangle distance code and that presented here is in the judicious use of masks and the select function to avoid branching and checking whether all triangles now fulfill a given early out criterion.

3. Benchmarks

To test the performance of the distance functions, we created a number of benchmarks that encompass the cases commonly encountered by the authors. Three benchmarks were used to test the performance of both the SIMD and non-SIMD versions. The geometries used in the benchmarks are presented in Figures 1, 2 and 3. The first geometry is a CAD geometry that contains many irregular triangles (33k in total) and called cembox here. This geometry is a proprietary CAD geometry and was chosen due to the high number of degenerate and irregular triangles. The second geometry is the well-known armadillo, Figure 2, which contains much more regular triangles (345k in total) as seen in Figure 3 and comes from the Stanford 3D scanning repository, [2014].

It is important to note that in each case when we compared the given SIMD routine against a non-SIMD routine, we always chose the fastest non-SIMD routine. In all cases, this happened to be the one our SIMD variant was based on. For both the triangle-point distance and the line segment-line segment routine, according to our tests, the fastest non-SIMD routines are from the book *Real-Time Collision Detection* [2004]. The fastest non-SIMD triangle-triangle routine is the one found in the PQP package [2014].

In each of the benchmark cases we moved one object via 10 steps from an initial non-colliding position through the other object to another non-colliding position. During the movements the moving object rotated 360 degrees and we computed the distance from a set of 10,000 triangles in the moving object to a set of 10,000 triangles in the static object. Note that when using SIMD instructions, branch divergence among the data can cause severe performance losses. In order to test the quality of our code with respect to branch divergence we ran two sets of tests. In the first three tests, we proceeded by testing four triangles at a time that all share a vertex in the moving geometry, against a single triangle from the static geometry. In the final three tests we tested four random triangles from the moving geometry against a single triangle in the static geometry. When computing line segment-line segment and triangle-point

distances we proceeded as with the triangle-triangle distances but we computed all combinations of edge (line segment) distances and triangle-point distances.

It is important to note that the data layout for SIMD tests is different to the normal data layout for single distance tests. Hence, we decided also to test the non-SIMD routines with our SIMD data layout to see the effect of this on performance. This data layout was the SoA data layout as explained in Section 2.

All benchmarks were run on a 64 bit 2.67 GHz Intel Core i7-920 CPU (Nahelem architecture, Bloomfeld model) with 8 GB RAM under Windows 7 using Microsoft Visual Studio 2012 with the $\backslash O2$ optimization on for all tests. All data was tested using 4 byte floats meaning that it was possible to pack four floats into a single SSE register.

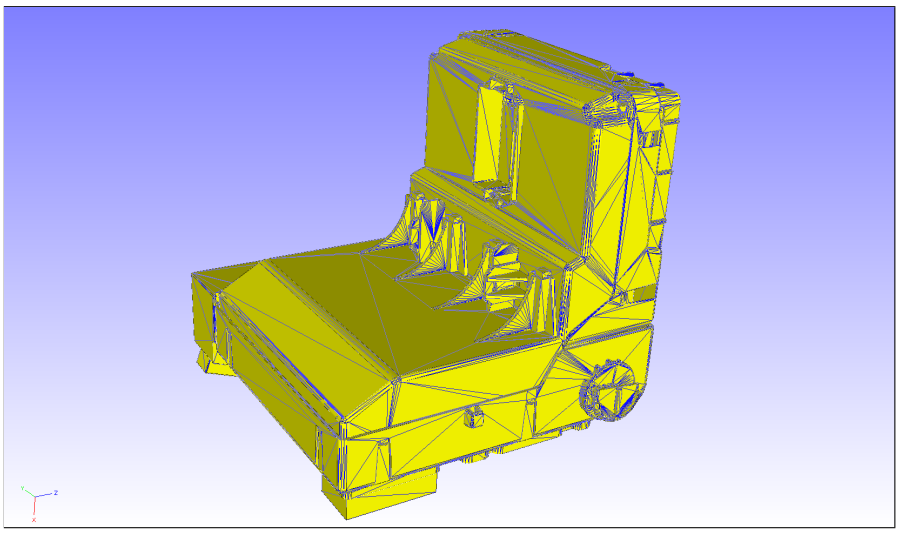

Figure 1. Cembox showing triangles imposed on the geometry. Photo courtesy of Volvo Cars.

Figure 2. Armadillo.

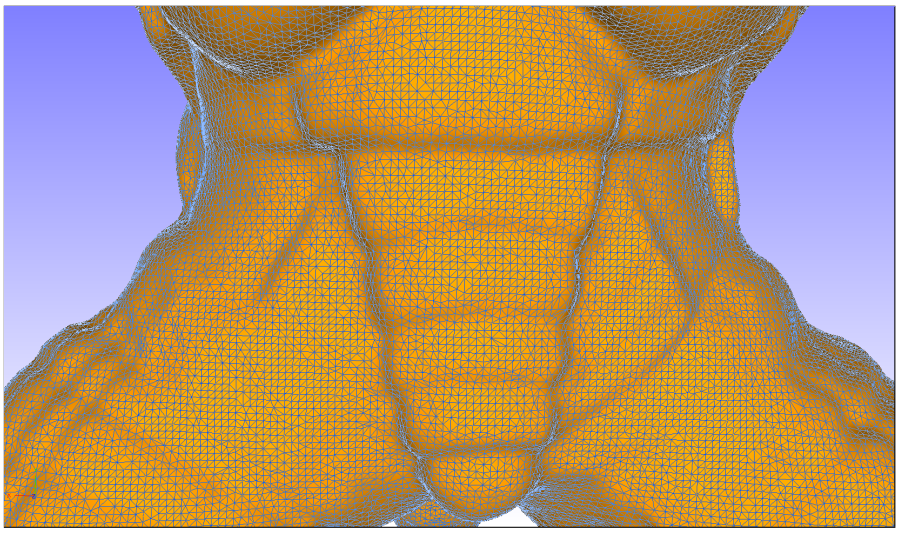

Figure 3. A close up of Armadillo showing triangles

4. Results

In the following tables we present our results. In each table Tri stands for triangle and Seg stands for line segment. In each case, each entry is the number of 10^7 tests/s per second. We present results for the non-SIMD code both with and without SIMD style data layout. For these non-SIMD routines, we called the non-SIMD layout test "non-SIMD layout" and the SIMD style data layout test "SIMD layout". We also present results whereby all optimizations are switched off (the $\backslash Od$ command for command

line optimization in the Visual Studio compiler) and add a no-opt to each test name to signify the tests without optimizations.

Cembox vs Cembox neighboring triangles						
Test Type	Tri-Point	Seg-Seg	Tri-Tri	Tri-Point no-opt	Seg-Seg no-opt	Tri-Tri no-opt
SIMD	3.33	2.94	2.63	0.166	0.148	0.035
SIMD layout	2.44	2.04	0.8	0.196	0.157	0.090
non-SIMD layout	2.08	1.6	0.79	0.132	0.111	0.075

Table 1. Results from the Cembox-Cembox benchmark with neighboring triangles.

Cembox vs Armadillo neighboring triangles						
Test Type	Tri-Point	Seg-Seg	Tri-Tri	Tri-Point no-opt	Seg-Seg no-opt	Tri-Tri no-opt
SIMD	3.57	2.94	2.63	0.161	0.15	0.111
SIMD layout	2.17	1.92	0.35	0.196	0.158	0.042
non-SIMD layout	1.96	1.54	0.35	0.13	0.14	0.039

Table 2. Results from the Cembox-Armadillo benchmark with neighboring triangles.

Armadillo vs Armadillo neighboring triangles						
Test Type	Tri-Point	Seg-Seg	Tri-Tri	Tri-Point no-opt	Seg-Seg no-opt	Tri-Tri no-opt
SIMD	3.33	2.94	2.5	0.151	0.149	0.023
SIMD layout	2.22	1.79	0.59	0.197	0.158	0.063
non-SIMD layout	2.04	1.56	0.52	0.132	0.114	0.056

Table 3. Results from the Armadillo-Armadillo benchmark with neighboring triangles.

Cembox vs Cembox random triangles						
Test Type	Tri-Point	Seg-Seg	Tri-Tri	Tri-Point no-opt	Seg-Seg no-opt	Tri-Tri no-opt
SIMD	3.45	3.03	2.56	0.162	0.15	0.033
SIMD layout	1.56	1.59	0.78	0.196	0.155	0.09
non-SIMD layout	1.3	1.25	0.76	0.135	0.115	0.076

Table 4. Results from the Cembox-Cembox benchmark with random triangles.

Cembox vs Armadillo random triangles						
Test Type	Tri-Point	Seg-Seg	Tri-Tri	Tri-Point no-opt	Seg-Seg no-opt	Tri-Tri no-opt
SIMD	3.57	3.13	2.5	0.159	0.15	0.111
SIMD layout	1.59	1.6	0.35	0.198	0.156	0.042
non-SIMD layout	1.31	1.33	0.34	0.123	0.131	0.039

Table 5. Results from the Cembox-Armadillo benchmark with random triangles.

Armadillo vs Armadillo random triangles						
Test Type	Tri-Point	Seg-Seg	Tri-Tri	Tri-Point no-opt	Seg-Seg no-opt	Tri-Tri no-opt
SIMD	3.28	3.2	2.38	0.154	0.15	0.019
SIMD layout	1.59	1.6	0.53	0.2	0.157	0.064
non-SIMD layout	1.35	1.31	0.49	0.125	0.113	0.057

Table 6. Results from the Armadillo-Armadillo benchmark with random triangles.

4.1. Discussion of results

The first thing that one notices in each of the above benchmark cases is that the SSE based routines are twice as fast in the triangle-point and segment-segment routines and up to seven times as fast in the triangle-triangle distance routine for each case with full optimization on (i.e. the $\backslash O2$ setting). This result is quite counterintuitive because the maximum performance gain that one would expect is about a four times speed up (due to similar data layout). We examine this phenomenon in more detail at the end of this section.

When it comes just to the data layout part of the speed up, we can see, on average, about a twenty percent performance improvement just based on the SIMD style data layout for the non-SIMD code.

When analyzing the unoptimized code using typical performance analysis tools, one finds the performance differences due to the `closestEdgeToEdge` from listing 13. In the case of cembox vs cembox or armadillo vs armadillo, one needs to go through numerous cases and the `closestEdgeToEdge` takes about 95 percent of the computation with half of the time being spent on the first call and the other half on the next two. For the case of cembox vs armadillo (in both cases), the closest distances can be quickly determined by the aforementioned routine, which takes about 90 percent of the computation. In the cembox vs armadillo case, however, the first `closestEdgeToEdge` accounts for already 85 percent of all the computation. This result is due to the small armadillo triangles being tested against the usually much larger and narrower cembox triangles and demonstrates the benefits of data coherence

that the fully optimized program seems to be able to even out.

In general, one can see that the non-SIMD code tends to perform better on the more irregular triangles, whereas the SIMD based code varies little. In the cases where the triangles are randomly chosen, we do not notice a performance loss based on the lack of geometric proximity between triangles. We can see that the SIMD code does not seem to suffer from additional branch divergence when the triangles are spread out instead of clustered together. The two different classes of tests (random vs neighboring triangles) also indicate that the tests are not memory bandwidth limited due to the similar results in both cases. Hence, it is the computations themselves that appear to limit the performance.

The most interesting part of the results, however, seems to be when one compares the non-optimized code with the optimized code. For the non-optimized code, the SSE based routines are often slower for all types of distance routine except for the cembox vs armadillo test case (where the SSE code is about three times faster). However, once optimization is turned on, the compiler is obviously able to optimize the SSE based code much better resulting in between a twenty and more than one hundred times speed up. However, the non-SIMD based code only improves about ten times more in each case. The final high quality results seem to be based on, in part, the ability of the compiler to optimize the code much better. Hence, the seven time speed up can be seen to be due mainly to the ability of the compiler to optimize the SIMD code much better than the non-SIMD code.

5. Conclusion and future work

In this paper we have demonstrated the ability to significantly accelerate distance computations between geometric primitives such as points, lines and triangles. Our results show that up to seven times performance gains are possible when using simple SIMD adaptations to available algorithms.

Possible future work for the results presented here would be to extend the tests to the new AVX and AVX2 instructions.

Also given the improvements resulting from the compiler optimizations, it would be interesting to retest these results with a number of different compilers to see how much of the effect seen here depends on the Microsoft Visual Studio compiler.

6. Supplementary code

In this section we present the details of the routines mentioned earlier in the text. We have also provided the code for our own functions listed in the code listings inside a supplementary header and source file called `ssedistance.h` and `ssedistance.cpp` respectively. To run this code it is necessary to download and link the files to the Embree package [2014]. The armadillo geometry can be downloaded from the Stanford

```
1  struct sseb
2  {
3    // data
4    union  { __m128 m128; int32 v[4]; };
5  };
6  // operations
7  const sseb operator &( const sseb& a, const sseb& b ) { return
       _mm_and_ps(a, b); }
8  const sseb operator |( const sseb& a, const sseb& b ) { return
       _mm_or_ps (a, b); }
9  const sseb operator ^( const sseb& a, const sseb& b ) { return
       _mm_xor_ps(a, b); }
10 bool all ( const sseb& b ) { return _mm_movemask_ps(b) == 0xf; }
11 bool any ( const sseb& b ) { return _mm_movemask_ps(b) != 0x0; }
12 bool none ( const sseb& b ) { return _mm_movemask_ps(b) == 0x0; }
13
14 struct ssef{
15   // data
16   union { __m128 m128; float v[4]; int i[4]; };
17 };
18 // operations
19 const ssef operator +(const ssef& a, const ssef& b) {return
       _mm_add_ps(a.m128, b.m128);}
20 const ssef operator -(const ssef& a, const ssef& b) {return
       _mm_sub_ps(a.m128, b.m128);}
21 const ssef min(const ssef& a, const ssef& b) {return _mm_min_ps(a.
       m128, b.m128);}
22 const ssef sqr(const ssef& a) {return _mm_mul_ps(a, a);}
23 const ssef sqrt(const ssef& a) {return _mm_sqrt_ps(a.m128);}
24 const ssef select(const sseb& mask, const ssef& t, const ssef& f) {
       return _mm_blendv_ps(f, t, mask);}
```

Listing 5. Basic Data Types

3D Scanning Repository [2014]. The cembox used here is proprietary and unfortunately cannot be distributed.

```
template<typename T> struct Vec3{
  // data
  T x, y, z;
}
// operations
template<typename T> Vec3<T> operator+(const Vec3<T>& a, const Vec3
    <T>& b) {return Vec3<T>(a.x + b.x, a.y + b.y, a.z + b.z);}
template<typename T> Vec3<T> operator-(const Vec3<T>& a, const Vec3
    <T>& b) {return Vec3<T>(a.x - b.x, a.y - b.y, a.z - b.z);}
template<typename T> Vec3<T> rcp(const Vec3<T>& a) {return Vec3<T>(
    rcp  (a.x), rcp  (a.y), rcp  (a.z));}
template<typename T> Vec3<T> rsqrt(const Vec3<T>& a) {return Vec3<T
    >(rsqrt(a.x), rsqrt(a.y), rsqrt(a.z));}
template<typename T> T dot(const Vec3<T>& a, const Vec3<T>& b) {
    return a.x*b.x + a.y*b.y + a.z*b.z;}
template<typename T> T length2(const Vec3<T>& a) {return dot(a,a);}
```

Listing 6. Vec3 and some of the overloaded functions it provides

```
typedef sseb                          simdBool;
typedef Vec3<ssef>                    simdFloatVec;
typedef std::array<simdFloatVec, 3>   simdTriangle_type;
typedef std::array<simdFloatVec, 2>   simdLine_type;
typedef simdFloatVec                  simdPoint_type;
```

Listing 7. Typedefs used throughout the paper

```
template<typename T> T clamp(const T& x, const T& lower, const T&
    upper) { return max(lower, min(x,upper)); }
```

Listing 8. Templated clamp function

```
simdFloat simdSegmentSegment2(simdFloatVec& oLine1Point,
    simdFloatVec& oLine2Point, const simdLine_type& iLine1, const
    simdLine_type& iLine2){
  const simdFloatVec dir1 = iLine1[1] - iLine1[0];
  const simdFloatVec dir2 = iLine2[1] - iLine2[0];
  const simdFloatVec lineDiff = iLine1[0] - iLine2[0];
  //The following are a set of values to assist computations
  const simdFloat a = dot(dir1, dir1);
  simdFloat e = dot(dir2, dir2);
  const simdFloat f = dot(dir2, lineDiff);
  const simdFloat c = dot(dir1, lineDiff);
  const simdFloat b = dot(dir1, dir2);

  //s and t are the parameter values from iLine1 and iLine2.
  simdFloat s,t;
  simdFloat denom = a*e-b*b;
  denom = max(denom, ulp);
  s = clamp((b*f - c*e) / denom, simdFloat(zero), simdFloat(one));
  e = max(e, ulp);
  t = (b*s + f) / e;
  //If t in [0,1] done.  Else clamp t, recompute s for the new
      value of t and clamp s to [0, 1]
  const simdFloat newT = clamp(t, simdFloat(zero), simdFloat(one));
  simdBool mask = (newT!=t);

  //Now choose correct values for s based on what positions the
      line segments were in.
  s = select(mask, clamp((newT*b - c) / a, simdFloat(zero),
      simdFloat(one)), s);
  //Compute closest points and return distance.
  oLine1Point = iLine1[0] + dir1 * s;
  oLine2Point = iLine2[0] + dir2 * newT;
  return length2(oLine1Point - oLine2Point);
}
```

Listing 9. Segment-Segment Distance

```
1  const simdFloat simdTriPoint2(simdFloatVec& oTriPoint, const
       simdTriangle_type& iTri, const simdPoint_type& iPoint){
2    const simdFloatVec ab = iTri[1] - iTri[0];
3    const simdFloatVec ac = iTri[2] - iTri[0];
4    const simdFloatVec ap = iPoint - iTri[0];
5    const simdFloat d1 = dot(ab, ap);
6    const simdFloat d2 = dot(ac, ap);
7    const simdBool mask1 = (d1<= simdFloat(zero)) & (d2<= simdFloat(
         zero));
8    oTriPoint = iTri[0];
9    simdBool exit(mask1);
10   if(all(exit))
11     return length2(oTriPoint - iPoint);
12
13   const simdFloatVec bp = iPoint - iTri[1];
14   const simdFloat d3 = dot(ab, bp);
15   const simdFloat d4 = dot(ac, bp);
16   const simdBool mask2 = (d3 >= simdFloat(zero)) & (d4 <= d3);
17   //Closest point is the point iTri[1].  Update if necessary.
18   oTriPoint = select(exit, oTriPoint, select(mask2, iTri[1],
         oTriPoint));
19   exit |= mask2;
20   if(all(exit))
21     return length2(oTriPoint - iPoint);
22
23   const simdFloatVec cp = iPoint - iTri[2];
24   const simdFloat d5 = dot(ab, cp);
25   const simdFloat d6 = dot(ac, cp);
26   const simdBool mask3 = (d6>=simdFloat(zero)) & (d5<=d6);
27   //Closest point is the point iTri[2].  Update if necessary.
28   oTriPoint = select(exit, oTriPoint, select(mask3, iTri[2],
         oTriPoint));
29   exit |= mask3;
30   if(all(exit))
31     return length2(oTriPoint - iPoint);
32
33   const simdFloat vc = d1*d4 - d3*d2;
34   const simdBool mask4 = (vc<=simdFloat(zero)) & (d1>=simdFloat(
         zero)) & (d3<=simdFloat(zero));
35   const simdFloat v1 = d1 / (d1 - d3);
36   const simdFloatVec answer1 = iTri[0] + v1 * ab;
37   //Closest point is on the line ab.  Update if necessary.
38   oTriPoint = select(exit, oTriPoint, select(mask4, answer1,
         oTriPoint));
39   exit |= mask4;
40   if(all(exit))
41     return length2(oTriPoint - iPoint);
42
43   const simdFloat vb = d5*d2 - d1*d6;
```

```
44   const simdBool mask5 = (vb<=simdFloat(zero)) & (d2>=simdFloat(
         zero)) & (d6<=simdFloat(zero));
45   const simdFloat w1 = d2 / (d2 - d6);
46   const simdFloatVec answer2 = iTri[0] + w1 * ac;
47   //Closest point is on the line ac.  Update if necessary.
48   oTriPoint = select(exit, oTriPoint, select(mask5, answer2,
         oTriPoint));
49   exit |= mask5;
50   if(all(exit))
51     return length2(oTriPoint - iPoint);
52
53   const simdFloat va = d3*d6 - d5*d4;
54   const simdBool mask6 = (va<=simdFloat(zero)) & ((d4 - d3)>=
         simdFloat(zero)) & ((d5 - d6)>=simdFloat(zero));
55   simdFloat w2 = (d4 - d3) / ((d4 - d3) + (d5 - d6));
56   const simdFloatVec answer3 = iTri[1] + w2*(iTri[2] - iTri[1]);
57   //Closest point is on the line bc.  Update if necessary.
58   oTriPoint = select(exit, oTriPoint, select(mask6, answer3,
         oTriPoint));
59   exit |= mask6;
60   if(all(exit))
61     return length2(oTriPoint - iPoint);
62
63   const simdFloat denom = simdFloat(one) / (va + vb + vc);
64   const simdFloat v2 = vb * denom;
65   const simdFloat w3 = vc * denom;
66   const simdFloatVec answer4 = iTri[0] + ab * v2 + ac * w3;
67   const simdBool mask7 = length2(answer4 - iPoint) < length2(
         oTriPoint - iPoint);
68   //Closest point is inside triangle.  Update if necessary.
69   oTriPoint = select(exit, oTriPoint, select(mask7, answer4,
         oTriPoint));
70   return length2(oTriPoint - iPoint);
71 }
```

Listing 10. Point-Triangle Distance

```
1  bool simdTriContact(const simdFloatVec &P1,const simdFloatVec &P2,
       const simdFloatVec &P3,const simdFloatVec &Q1,const simdFloatVec
        &Q2,const simdFloatVec &Q3) {
2
3    const simdFloatVec p1 = 0; //P1 - P1;
4    const simdFloatVec p2 = P2 - P1;
5    const simdFloatVec p3 = P3 - P1;
6
7    const simdFloatVec q1 = Q1 - P1;
8    const simdFloatVec q2 = Q2 - P1;
9    const simdFloatVec q3 = Q3 - P1;
10
11   const simdFloatVec e1 = P2 - P1;
12   const simdFloatVec e2 = P3 - P2;
13
14   const simdFloatVec f1 = Q2 - Q1;
15   const simdFloatVec f2 = Q3 - Q2;
16
17   simdBool mask(true);
18
19   const simdFloatVec n1 = cross(e1, e2);
20   mask &= simdProject6(n1,p1,p2,p3,q1,q2,q3);
21   if(none(mask)) return 0;
22   const simdFloatVec m1 = cross(f1, f2);
23   mask &= simdProject6(m1,p1,p2,p3,q1,q2,q3);
24   if(none(mask)) return 0;
25   const simdFloatVec ef11 = cross(e1, f1);
26   mask &= simdProject6(ef11,p1,p2,p3,q1,q2,q3);
27   if(none(mask)) return 0;
28   const simdFloatVec ef12 = cross(e1, f2);
29   mask &= simdProject6(ef12,p1,p2,p3,q1,q2,q3);
30   if(none(mask)) return 0;
31
32   const simdFloatVec f3 = q1 - q3;
33   const simdFloatVec ef13 = cross(e1, f3);
34   mask &= simdProject6(ef13,p1,p2,p3,q1,q2,q3);
35   if(none(mask)) return 0;
36   const simdFloatVec ef21 = cross(e2, f1);
37   mask &= simdProject6(ef21,p1,p2,p3,q1,q2,q3);
38   if(none(mask)) return 0;
39   const simdFloatVec ef22 = cross(e2, f2);
40   mask &= simdProject6(ef22,p1,p2,p3,q1,q2,q3);
41   if(none(mask)) return 0;
42   const simdFloatVec ef23 = cross(e2, f3);
43   mask &= simdProject6(ef23,p1,p2,p3,q1,q2,q3);
44   if(none(mask)) return 0;
45
46   const simdFloatVec e3 = p1 - p3;
47   const simdFloatVec ef31 = cross(e3, f1);
```

```
48   mask &= simdProject6(ef31,p1,p2,p3,q1,q2,q3);
49   if(none(mask)) return 0;
50   const simdFloatVec ef32 = cross(e3, f2);
51   mask &= simdProject6(ef32,p1,p2,p3,q1,q2,q3);
52   if(none(mask)) return 0;
53   const simdFloatVec ef33 = cross(e3, f3);
54   mask &= simdProject6(ef33,p1,p2,p3,q1,q2,q3);
55   if(none(mask)) return 0;
56   const simdFloatVec g1 = cross(e1, n1);
57   mask &= simdProject6(g1,p1,p2,p3,q1,q2,q3);
58   if(none(mask)) return 0;
59   const simdFloatVec g2 = cross(e2, n1);
60   mask &= simdProject6(g2,p1,p2,p3,q1,q2,q3);
61   if(none(mask)) return 0;
62   const simdFloatVec g3 = cross(e3, n1);
63   mask &= simdProject6(g3,p1,p2,p3,q1,q2,q3);
64   if(none(mask)) return 0;
65   const simdFloatVec h1 = cross(f1, m1);
66   mask &= simdProject6(h1,p1,p2,p3,q1,q2,q3);
67   if(none(mask)) return 0;
68   const simdFloatVec h2 = cross(f2, m1);
69   mask &= simdProject6(h2,p1,p2,p3,q1,q2,q3);
70   if(none(mask)) return 0;
71   const simdFloatVec h3 = cross(f3, m1);
72   mask &= simdProject6(h3,p1,p2,p3,q1,q2,q3);
73   if(none(mask)) return 0;
74   return 1;
75 }
```

Listing 11. Triangle-Triangle Collision

```
1  //A common subroutine for each separating direction
2  inline simdBool simdProject6(const simdFloatVec &ax,const
       simdFloatVec &p1,const simdFloatVec &p2,const simdFloatVec &p3,
       const simdFloatVec &q1, const simdFloatVec &q2,const
       simdFloatVec &q3) {
3    simdFloat P1 = dot(ax, p1);
4    simdFloat P2 = dot(ax, p2);
5    simdFloat P3 = dot(ax, p3);
6
7    simdFloat Q1 = dot(ax, q1);
8    simdFloat Q2 = dot(ax, q2);
9    simdFloat Q3 = dot(ax, q3);
10
11   simdFloat mx1 = max(P1,P2,P3);
12   simdFloat mn1 = min(P1,P2,P3);
13   simdFloat mx2 = max(Q1,Q2,Q3);
14   simdFloat mn2 = min(Q1,Q2,Q3);
15
16   return (mn1 <= mx2) && (mn2 <= mx1);
17 }
```

Listing 12. Triangle-Triangle Collision Projection Subroutine

```
1  simdFloat simdTriTri2(simdFloatVec& oTri1Point, simdFloatVec&
       oTri2Point, const simdTriangle_type& iTri1, const
       simdTriangle_type& iTri2){
2
3    //The three edges of the triangle.  Keep orientation consistent.
4    const simdLine_type tri1Edges[3] = {{iTri1[1], iTri1[0]}, {iTri1
         [2], iTri1[1]}, {iTri1[0], iTri1[2]}};
5    const simdLine_type tri2Edges[3] = {{iTri2[1], iTri2[0]}, {iTri2
         [2], iTri2[1]}, {iTri2[0], iTri2[2]}};
6
7    simdFloatVec tri1Vector, tri2Vector;
8    simdBool isFinished(False);
9
10   simdFloat minDistsTriTri = closestEdgeToEdge(isFinished,
         oTri1Point, oTri2Point, tri1Edges, tri2Edges[0], iTri2[2]);
11   if(all(isFinished))
12     return minDistsTriTri;
13
14   simdFloat tmpMinDist = closestEdgeToEdge(isFinished, tri1Vector,
         tri2Vector, tri1Edges, tri2Edges[1], iTri2[0]);
15   simdBool mask = tmpMinDist < minDistsTriTri;
16   minDistsTriTri = select(mask, tmpMinDist, minDistsTriTri);
17   oTri1Point = select(mask, tri1Vector, oTri1Point);
18   oTri2Point = select(mask, tri2Vector, oTri2Point);
19   if(all(isFinished))
20     return minDistsTriTri;
21
```

```
22   tmpMinDist = closestEdgeToEdge(isFinished, tri1Vector, tri2Vector
         , tri1Edges, tri2Edges[2], iTri2[1]);
23   mask = tmpMinDist < minDistsTriTri;
24   minDistsTriTri = select(mask, tmpMinDist, minDistsTriTri);
25   oTri1Point = select(mask, tri1Vector, oTri1Point);
26   oTri2Point = select(mask, tri2Vector, oTri2Point);
27   if(all(isFinished))
28     return minDistsTriTri;
29
30   //Now do vertex-triangle distances.
31   tmpMinDist = closestVertToTri(tri2Vector, tri1Vector, iTri2,
         iTri1);
32   mask = tmpMinDist < minDistsTriTri;
33   oTri1Point = select(mask, tri1Vector, oTri1Point);
34   oTri2Point = select(mask, tri2Vector, oTri2Point);
35   minDistsTriTri = select(mask, tmpMinDist, minDistsTriTri);
36
37   tmpMinDist = closestVertToTri(tri1Vector, tri2Vector, iTri1,
         iTri2);
38   mask = tmpMinDist < minDistsTriTri;
39   oTri1Point = select(mask, tri1Vector, oTri1Point);
40   oTri2Point = select(mask, tri2Vector, oTri2Point);
41
42   minDistsTriTri = select(mask, tmpMinDist, minDistsTriTri);
43   //We need to rule out the triangles colliding with each other.
         Hence test for collision.
44
45   simdBool colliding = simdBool(simdTriContact(iTri1, iTri2));
46   return select(colliding, simdFloat(zero), minDistsTriTri);
47 }
48
49 //Compute the distance between a triangle vertex and another
       triangle
50 simdFloat closestVertToTri(simdFloatVec& oTriAPoint, simdFloatVec&
       oTriBPoint, const simdTriangle_type& iTriA, const
       simdTriangle_type& iTriB) {
51   simdFloatVec Ap, Bp, Cp;
52
53   const simdFloatVec edge[2] = {iTriA[1] - iTriA[0], iTriA[2] -
         iTriA[1]};
54   simdFloatVec TriNormal = cross(edge[1], edge[0]);
55   const simdFloat norm2 = length2(TriNormal);
56
57   const simdFloat A = simdTriPoint2(Ap, TriNormal, norm2, iTriA,
         iTriB[0]);
58   const simdFloat B = simdTriPoint2(Bp, TriNormal, norm2, iTriA,
         iTriB[1]);
59   const simdFloat C = simdTriPoint2(Cp, TriNormal, norm2, iTriA,
         iTriB[2]);
```

```
60
61    const simdBool AB = A < B;
62    const simdFloat ABdist = select(AB, A, B);
63    const simdFloatVec ABp = select(AB, Ap, Bp);
64
65    const simdBool ABC = ABdist < C;
66    oTriAPoint = select(ABC, ABp, Cp);
67    oTriBPoint = select(ABC, select(AB, iTriB[0], iTriB[1]), iTriB
          [2]);
68
69    return select(ABC, ABdist, C);
70 }
71
72 //Compute the distance between a triangle edge and another
       triangle's edges
73 simdFloat closestEdgeToEdge(simdBool& oIsFinished, simdFloatVec&
       oTriAPoint, simdFloatVec& oTriBPoint, const simdLine_type (&
       iTriAEdges)[3], const simdLine_type& iTriBEdge, const
       simdFloatVec& iTriBLastPt){
74    //Test the triangle edge against all three edges of the triangle
          iTriA.
75    simdFloatVec A2p, A3p, B2p, B3p, separatingDir;
76
77    const simdFloat A = simdSegmentSegment2(oTriAPoint, oTriBPoint,
          iTriAEdges[0], iTriBEdge);
78    //Test to see if the distances found so far were the closest:
79    separatingDir = oTriBPoint - oTriAPoint;
80    oIsFinished |= closestEdgePoints(iTriAEdges[1][0], oTriAPoint,
          iTriBLastPt, oTriBPoint, separatingDir);
81    if(all(oIsFinished))
82        return A;
83
84    const simdFloat B = simdSegmentSegment2(A2p, B2p, iTriAEdges[1],
          iTriBEdge);
85    separatingDir = B2p - A2p;
86    oIsFinished |= closestEdgePoints(iTriAEdges[2][0], A2p,
          iTriBLastPt, B2p, separatingDir);
87
88    const simdBool AB = A < B;
89    const simdFloat ABdist = select(AB, A, B);
90    oTriAPoint = select(AB, oTriAPoint, A2p);
91    oTriBPoint = select(AB, oTriBPoint, B2p);
92
93    if(all(oIsFinished))
94      return ABdist;
95
96    const simdFloat C = simdSegmentSegment2(A3p, B3p, iTriAEdges[2],
          iTriBEdge);
97    separatingDir = B3p - A3p;
```

```
98    oIsFinished |= closestEdgePoints(iTriAEdges[0][0], A3p,
          iTriBLastPt, B3p, separatingDir);
99
100   const simdBool ABC = ABdist < C;
101   oTriAPoint = select(ABC, oTriAPoint, A3p);
102   oTriBPoint = select(ABC, oTriBPoint, B3p);
103
104   return select(ABC, ABdist, C);
105 }
106 //Find a direction that demonstrates that the current side is
        closest and separates the triangles.
107 simdBool closestEdgePoints(const simdFloatVec& iTri1Pt, const
        simdFloatVec& iClosestPtToTri1, const simdFloatVec& iTri2Pt,
        const simdFloatVec& iClosestPtToTri2, const simdFloatVec&
        iSepDir){
108   simdFloatVec awayDirection = iTri1Pt - iClosestPtToTri1;
109   const simdFloat isDiffDirection = dot(awayDirection, iSepDir);
110
111   awayDirection = iTri2Pt - iClosestPtToTri2;
112   const simdFloat isSameDirection = dot(awayDirection, iSepDir);
113
114   return (isDiffDirection <= simdFloat(zero)) & (isSameDirection >=
          simdFloat(zero));
115 }
```

Listing 13. Triangle-Triangle Distance

7. Acknowledgements

This work was carried out at the Wingquist Laboratory VINN Excellence Centre, and is part of the Sustainable Production Initiative and the Production Area of Advance at Chalmers University of Technology. It was supported by the Swedish Governmental Agency for Innovation Systems. The authors are deeply grateful for the insightful and helpful comments of the reviewers that significantly improved the presentation and results in this paper.

References

EBERLY, D. H. 2007. *3D game engine design: a practical approach to real-time computer graphics*. Taylor & Francis US. 88

ERICSON, C. 2004. *Real-time collision detection*. CRC Press. 88, 89, 92

FLYNN, M. J. 1972. Some computer organizations and their effectiveness. *IEEE Transactions on Computers 100*, 9, 948–960. URL: http://ieeexplore.

ieee.org/xpl/articleDetails.jsp?arnumber=5009071, doi:10.1109/TC.1972.5009071. 86

GUIGUE, P., AND DEVILLERS, O. 2003. Fast and robust triangle-triangle overlap test using orientation predicates. *Journal of graphics tools 8*, 1, 25–32. URL: www.tandfonline.com/doi/abs/10.1080/10867651.2003.10487580, doi:10.1080/10867651.2003.10487580. 91

HAVEL, J., AND HEROUT, A. 2010. Yet faster ray-triangle intersection (using SSE4). *IEEE Transactions on Visualization and Computer Graphics 16*, 3, 434–438. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5159346, doi:10.1109/TVCG.2009.73. 86, 87

HERMANSSON, T., BOHLIN, R., CARLSON, J. S., AND SÖDERBERG, R. 2013. Automatic assembly path planning for wiring harness installations. *Journal of Manufacturing Systems*. URL: http://www.sciencedirect.com/science/article/pii/S0278612513000393, doi:10.1016/j.jmsy.2013.04.006. 87

INTEL, 2014. Embree. http://embree.github.io/. 88, 97

KLIMOVITSKI, A. 2001. Using SSE and SSE2: Misconceptions and reality. *Intel developer update magazine*, 1–8. 87

LABORATORY, S. C. G., 2014. The stanford 3d scanning repository. http://graphics.stanford.edu/data/3Dscanrep/. 92, 98

LARSEN, E., GOTTSCHALK, S., LIN, M. C., AND MANOCHA, D., 2014. PQP. http://gamma.cs.unc.edu/SSV/. 88, 91, 92

LARSSON, T., AKENINE-MÖLLER, T., AND LENGYEL, E. 2007. On faster sphere-box overlap testing. *Journal of graphics, gpu, and game tools 12*, 1, 3–8. URL: http://www.tandfonline.com/doi/abs/10.1080/2151237X.2007.10129232#.VIq9i3urPRg, doi:10.1080/2151237X.2007.10129232. 86

MÖLLER, T. 1997. A fast triangle-triangle intersection test. *Journal of graphics tools 2*, 2, 25–30. URL: http://www.tandfonline.com/doi/abs/10.1080/10867651.1997.10487472#.VIq9bnurPRg, doi:10.1080/10867651.1997.10487472. 91

SHELLSHEAR, E., BITAR, F., AND ASSARSSON, U. 2013. PDQ: Parallel distance queries for deformable meshes. *Graphical Models 75*, 2, 69–78. URL: http://www.sciencedirect.com/science/article/pii/S1524070313000027, doi:10.1016/j.gmod.2012.12.002. 87

SPENSIERI, D., CARLSON, J. S., BOHLIN, R., AND SÖDERBERG, R. 2008. Integrating assembly design, sequence optimization, and advanced path planning. In *ASME Conference Proceedings*, ASME, 73–81. URL: http://link.aip.org/link/abstract/ASMECP/v2008/i43253/p73/s1, doi:10.1115/DETC2008-49760. 87

SPENSIERI, D., BOHLIN, R., AND CARLSON, J. S. 2013. Coordination of robot paths for cycle time minimization. In *Automation Science and Engineering (CASE), 2013 IEEE International Conference on*, 522–527. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6654032, doi:10.1109/CoASE.2013.6654032. 87

THAKKUR, S., AND HUFF, T. 1999. Internet streaming SIMD extensions. *Computer 32*, 12, 26–34. 86

Author Contact Information

Robin Ytterlid
FCC
Sven Hultins Gata 9D, Gothenburg, Västra Götaland,
41288 Sweden
robin.ytterlid@fcc.chalmers.se

Evan Shellshear
FCC
Sven Hultins Gata 9D, Gothenburg, Västra Götaland,
41288 Sweden
evan.shellshear@fcc.chalmers.se

Robin Ytterlid and Evan Shellshear, Fast Distance Queries for Triangles, Lines, and Points using SSE Instructions, *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 4, 86–110, 2014
http://jcgt.org/published/0003/04/05/

Received:	2014-06-19		
Recommended:	2014-09-12	Corresponding Editor:	Marc Olano
Published:	2014-12-13	Editor-in-Chief:	Morgan McGuire