GPU Collision Detection in Conformal Geometric Space

Eduardo Roa[1] Víctor Theoktisto[1,2] Marta Fairén[2] Isabel Navazo[2]

[1]Universidad Simón Bolívar, Caracas, Venezuela
[2]Universitat Politécnica de Catalunya, Barcelona, Spain †

ABSTRACT

We derive a conformal algebra treatment unifying all types of collisions among points, vectors, areas (defined by bivectors and trivectors) and 3D solid objects (defined by trivectors and quadvectors), based in a reformulation of collision queries from $\mathbb{R}^3$ to conformal $\mathbb{R}^{4,1}$ space. The algebraic formulation in this 5D space is then implemented in GPU to allow faster parallel computation queries. Results show expected orders of magnitude improvements computing collisions among known mesh models, allowing interactive rates without using optimizations and bounding volume hierarchies.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors, parallel processing, I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Boundary representations, Collision detection, I.1.2 [Computer Graphics]: Algorithms—Algebraic algorithms

1. Introduction

Certain application domains, such as animation and haptic rendering, involve real-time interactions among detailed models, requiring fast computation of massive numbers of collisions. Diverse formulations and optimizations have been developed that individually target specific object representations for collision queries.

In the next sections we present a unified geometric algebra treatment with a SIMD implementation that shifts collision detection from euclidean $\mathbb{R}^3$ to conformal $\mathbb{R}^{4,1}$ space. Line segments, spheres and polygons (and therefore, meshes) are treated as similar conformal entities by a shared core of CUDA kernels running in the GPU.

Results show expected orders of magnitude improvements when computing collisions and intrusions among known mesh models, without using any hierarchical collision pre-filtering schemes.

2. Collision Detection Computation

Basically, a collision is the result of a spatial query asking whether two geometric objects intersect at some point in time. A likely scenario is rigid body collision detection, highly used in haptic manipulation [ORC07, TFN10] and animation.

Most techniques avoid exhaustive detection by enclosing objects into hierarchies of fast-to-discard *bounding volumes* [Eri05]: Axis-Aligned and Oriented Bounding Boxes (AABB, OBB), Rectangular Swept Spheres (RSS), Convex Hulls, kd-trees, and BSP trees.

2.1. GPU-assisted parallel computation

General-Purpose Computation on Graphics Hardware [OLG*07] harnesses programmable graphics processors to solve vastly complex problems, sending data as texture memory to shader programs for some number crunching instead of image rendering. On top of that, the Compute Unified Device Architecture (CUDA™) API/SDK [SK10] provides a SIMD parallel programming framework, with concurrent threads/simultaneous kernel execution at the GPU streaming processors. A current survey of GPU-assisted applications, including collision detection, can be found at [LMM10].

† Co-financed in part by Project TIN2010-20590-C02-01 of the Spanish Ministry of Science (MEC)

2.2. Geometric Algebra

A recent formalism in Computer Graphics is the Geometric Algebra approach described by Dorst *et al* [DFM07], with the extensions to conformal geometric spaces added by Vince [Vin08]. The fundamental algebraic operators in this approach are the *inner product* $(x \cdot y)$, the *outer product* $(x \wedge y)$, and the *geometric product* (xy).

Definition 2.1 For vectors a and b, the *outer product* $a \wedge b$ defines an *oriented hyperplane*, or *bivector*. Its magnitude is the signed area of the parallelogram $\|a \wedge b\| = \|a\|\|b\| \sin\theta$. The sign will be positive if a folds onto b *counterclockwise*, and negative otherwise.

Definition 2.2 For vectors a and b, its *geometric product* ***ab*** is the sum of the *inner* dot product and the *outer* bivector product.

$$ab = a \cdot b + a \wedge b, \begin{cases} \text{anticommutative}, & ba = -ab \\ \text{associative}, & a(bc) = (ab)c = abc \\ \text{distributive}, & a(b+c)d = abd + acd \end{cases}$$

2.3. Conformal Geometry

Conformal Geometry [DFM07, DL09, BCS10] describes an elegant algebraic space for geometric visualization in $\mathbb{R}^3$, since it is homogeneous, supports points and lines at infinity, preserves angle and distance, and can represent points, circles, lines, spheres and planes.

Definition 2.3 A conformal space $\mathbb{R}^{p+1,q+1}$ of $p+1$ *positive* dimensions and $q+1$ *negative* dimensions is built from a $\mathbb{R}^{p,q}$ space. A point $x = ue_1 + ve_2 + we_3$ in $\mathbb{R}^3$ maps to a null vector X in $\mathbb{R}^{4,1}$ $(X \cdot X = 0,\ X \neq 0)$, having the orthonormal base $\{e_1, e_2, e_3, e, \bar{e}\}$.

$$e_1 \cdot e_1 = e_2 \cdot e_2 = e_3 \cdot e_3 = 1, \qquad e \cdot e = 1, \quad \bar{e} \cdot \bar{e} = -1$$
$$n = e + \bar{e}, \quad \bar{n} = e - \bar{e}$$
$$X = P(x) = 2x + x^2 n - \bar{n} \quad (2.1)$$

Table 1: *The 32 (1+5+10+10+5+1) blade terms of the canonical base for the conformal* $\mathbb{R}^{4,1}$ *space*

	Base Elements	Blade components				
1	scalar			λ		
5	vectors	e_1,	e_2,	e_3,	e,	$\bar{e}$
10	bivectors	$e_1 \wedge e_2$, $e_2 \wedge e_3$,	$e_3 \wedge e_1$, $e_1 \wedge \bar{e}$,	$e_2 \wedge \bar{e}$, $e_3 \wedge \bar{e}$,	$e_1 \wedge e$, $e_2 \wedge e$,	$e_3 \wedge e$, $\bar{e} \wedge e$
10	trivectors	$e_1 \wedge e_2 \wedge e_3$,	$e_1 \wedge e_2 \wedge \bar{e}$,	$e_1 \wedge e_2 \wedge e$,	$e_3 \wedge e_1 \wedge \bar{e}$,	$e_3 \wedge e_1 \wedge e$,
		$e_2 \wedge e_3 \wedge \bar{e}$,	$e_2 \wedge e_3 \wedge e$,	$e_1 \wedge \bar{e} \wedge e$,	$e_2 \wedge \bar{e} \wedge e$,	$e_3 \wedge \bar{e} \wedge e$
5	quadvectors	$e_1 \wedge e_2 \wedge e_3 \wedge \bar{e}$,	$e_1 \wedge e_2 \wedge e_3 \wedge e$,	$e_1 \wedge e_2 \wedge \bar{e} \wedge e$,	$e_3 \wedge e_1 \wedge \bar{e} \wedge e$,	$e_2 \wedge e_3 \wedge \bar{e} \wedge e$
1	pseudoscalar (I)			$e_1 \wedge e_2 \wedge e_3 \wedge \bar{e} \wedge e$		

Table 2: *Algebraic primitives built from blades in the* $\mathbb{R}^{4,1}$ *conformal space, which also includes scalars, points and vectors*

Primitive	**Blade type**	**Algebraic representation**	**Geometric interpretation**
Circle	*trivector*	$C = P_1 \wedge P_2 \wedge P_3$	Three noncollinear points delimit the perimeter of the circle
Line	*trivector*	$L = P_1 \wedge P_2 \wedge n$	Two nonidentical points define a segment plus the point at *infinity*
Sphere	*quadvector*	$S = P_1 \wedge P_2 \wedge P_3 \wedge P_4$	Four noncoplanar points delimit the surface of the sphere
Plane	*quadvector*	$\Pi = P_1 \wedge P_2 \wedge P_3 \wedge n$	Three noncollinear points define a triangle plus the point at *infinity*

with n and $\bar{n}$ representing the null vectors at *infinity* and at the *origin*. From these null vectors are derived the primitives shown on Table 2.

Definition 2.4 The outer product of k vectors is called a *k-blade*:

$$v_1 \wedge v_2 \wedge \ldots \wedge v_{k-1} \wedge v_k = V \in \mathbb{R}^{n,m}, \;\; k \leq n+m \qquad (2.2)$$

The highest order k-blade of a $\mathbb{R}^{n,m}$ space is called a *pseudoscalar*, denoted by I ($I^2 = -1$), for its similarity as a *rotor* to the complex number $\boldsymbol{i}$. Thus, IX is a $\frac{\pi}{2}$ *counterclockwise* rotation of X. Likewise, XI is a $\frac{\pi}{2}$ *clockwise* rotation of X.

2.4. Intersections in conformal space

A *multivector* is a linear combination of the $\sum_{k=1}^{n+m}\binom{n+m}{k} = 2^{n+m}$ canonical base of blades for the conformal $\mathbb{R}^{n,m}$ space. Table 1 shows the 32 blade terms of the canonical base in $\mathbb{R}^{4,1}$.

The *meet* operator ($\vee$) denotes the *intersection multivector*, having pseudoscalar $I = e_1 e_2 e_3 e \bar{e}$ for that space. Intersections among multivector in the conformal model [DL09] are specified by the same equation for all multivectors, as shown in Table 3.

$$B \;=\; (X \vee Y) = (IX) \cdot Y, \;\text{ with square norm } B^2 = \|B\|^2 \qquad (2.3)$$

$$\text{and } B \;=\; \beta_0 + \beta_{e_1} e_1 + \ldots + \beta_{e_1 e_2} e_1 e_2 + \ldots + \beta_{e_1 e_2 e_3 e \bar{e}} e_1 e_2 e_3 e \bar{e}.$$

The work of Roa [Roa11] states the following criteria for B^2:

If $B^2 > 0$, X and Y intersect at least at two points.
If $B^2 = 0$, X and Y intersect at one point (a tangent).
If $B^2 < 0$, X and Y do not intersect.

Table 3: *Primitive intersections in* $\mathbb{R}^{4,1}$ *conformal space*

Primitive	Primitive	Conformal representation
Line	Plane	$B = \Pi_1 \vee L_1 = (I\Pi_1) \cdot L_1$
Line	Sphere	$B = S_1 \vee L_1 \;\; = (IS_1) \cdot L_1$
Plane	Plane	$B = \Pi_1 \vee \Pi_2 = (I\Pi_1) \cdot \Pi_2$
Plane	Sphere	$B = S_1 \vee \Pi_1 = (IS_1) \cdot \Pi_1$
Sphere	Sphere	$B = S_1 \vee S_2 \;\; = (IS_1) \cdot S_2$

3. Kernels for the Intersection Algorithms

For the next sections we present the algorithms, results and conclusions of the CUDA implementation for collision detection, using meshes from the Stanford University repository at http://www.graphics.stanford.edu/data/3Dscanrep/. Neither bounding volume hierarchies nor optimizations were employed, just raw collisions.

The obtained algorithms of interest are Ray–Plane (Line Segment–Triangle), Plane–Plane (Triangle–Triangle), and Sphere–Sphere. The different B multivectors and their B^2 norms were algebraically derived for each CUDA kernel.

3.1. Line Segment (Ray)–Triangle (Plane) intersection

The intersection between a line segment and a triangle is a collision query between the ray passing along the segment and the plane of the triangle, and later checking whether boundaries meet. The intersection multivector $B = (\Pi_1 \vee L_1) = (I\Pi_1) \cdot L_1$ evaluates to

$$\begin{aligned}
B \;&=\; (\omega_2\beta_3 + \omega_1\beta_4 - \omega_4\beta_1)e_1 e + (\omega_2\beta_3 + \omega_1\beta_4 - \omega_4\beta_1)e_1\bar{e} \\
&+\; (\omega_3\beta_1 - \omega_2\beta_2 + \omega_1\beta_5)e_2 e + (\omega_3\beta_1 - \omega_2\beta_2 - \omega_1\beta_5)e_2\bar{e} \\
&+\; (-\omega_3\beta_3 + \omega_4\beta_2 + \omega_1\beta_6)e_3 e + (-\omega_3\beta_3 + \omega_4\beta_2 + \omega_1\beta_6)e_3\bar{e} \\
&+\; (-\omega_3\beta_4 - \omega_4\beta_5 - \omega_2\beta_6)e\bar{e} \qquad (3.1) \\
B^2 \;&=\; (\omega_3\beta_4 + \omega_4\beta_5 + \omega_2\beta_6)^2 \qquad (3.2)
\end{aligned}$$

where the βs and the ωs are the coefficients of the corresponding multivectors for L_1 and Π_1. A nonnegative B^2 signals a potential collision. The segment is then tested against the triangle's edges, to detect crossings and an effective collision. Algorithm 1 describes the complete procedure to compute intersections.

Algorithm 1: Line Segment-Triangle intersection

```
kernel Segment_Triangle_Intersect(segment L1, plane P1)
  Normalize(L1);  Normalize(P1)
  [a,e3er] = ConformalIntersectLinePlane(L1,P1)
  L3 = Line(L0.point1,L0.point2)
  L2 = Line(P1.point3,P1.point1)
  [out1,ind1] = ConformalIntersectSegmentSegment(L2, L3 )
  L2 = Line(P1.point1,P1.point2)
  [out2,ind2] = ConformalIntersectSegmentSegment(L2, L3 )
  L2 = Line(P1.point3,P1.point2)
  [out3,ind3] = ConformalIntersectSegmentSegment(L2, L3 )
    // out# = 1 (segments intersect); 0 (they do not)
    // ind#: scalar coefficient of e vector
  if (a == 0) then   //line and plane parallel
    if e3er = 0 then //line lies on the triangle's plane
      //verify segment intersection with other triangles
      return (out1==1) or (out2==1) or (out3==1)
    else return 0    // No intersection found
    end if
  else //whether both points are in same side of plane
    sign1 = trivector(P1.point2-P1.point1,P1.point3-P1.
            point1,L1.point1-P1.point1)
    sign2 = trivector(P1.point2-P1.point1,P1.point3-P1.
            point1,L1.point2-P1.point1)
    if (sign1 == sign2) then
      return 0         // Segment does not touch plane
    end if             // Segment crosses the plane
    return result = (ind1>0 and ind2>0 and ind3<0) or
                    (ind1<0 and ind2<0 and ind3>0);
  end if
```

3.2. Triangle (Plane)–Triangle (Plane) intersection

A triangle–triangle is the most interesting collision to define, since is most commonly used. B is the following term

$$\begin{aligned}
B \;&=\; (\omega_4\lambda_2 - \omega_2\lambda_4)e_1 e\bar{e} + (\omega_2\lambda_3 - \omega_3\lambda_2)e_2 e\bar{e} + (\omega_3\lambda_4 - \omega_4\lambda_3)e_3 e\bar{e} \\
&+\; (\omega_2\lambda_1 - \omega_1\lambda_2)e_1 e_2 e + (\omega_3\lambda_1 - \omega_1\lambda_3)e_2 e_3 e + (\omega_4\lambda_1 - \omega_1\lambda_4)e_3 e_1 e \\
&+\; (\omega_2\lambda_1 - \omega_1\lambda_2)e_1 e_2\bar{e} + (\omega_3\lambda_1 - \omega_1\lambda_3)e_2 e_3\bar{e} + (\omega_4\lambda_1 - \omega_1\lambda_4)e_3 e_1\bar{e} \qquad (3.3) \\
B^2 \;&=\; (\omega_4\lambda_2 - \omega_2\lambda_4)^2 + (\omega_2\lambda_3 - \omega_3\lambda_2)^2 + (\omega_3\lambda_4 - \omega_4\lambda_3)^2 \qquad (3.4)
\end{aligned}$$

Instead of plane intersections, it is much faster to implement a Triangle–Triangle intersection (see Table 3) for three Segment–Triangle intersections, as shown in Algorithm 2. Any one segment colliding with the opposite triangle triggers detection.

Algorithm 2: Triangle-Triangle intersection

```
kernel Segment_Plane_Intersect(triangle P1, triangle P2 )
  //Verify if all points are at same side of plane
  if not VerifySameSidePoints(P1) then
    return 0 // No Intersection
  end if
  Normalize(P1);  Normalize(P2)
  r1 = Line(P2.point1 ,P2.point2)
  r2 = Line(P2.point2 ,P2.point3)
  r3 = Line(P2.point3 ,P2.point1)
  // out# = 1, intersection exists , 0 no intersection ,
  out1 = ConformalIntersectSegmentPlane(r1 , P1 )
  out2 = ConformalIntersectSegmentPlane(r2 , P1 )
  out3 = ConformalIntersectSegmentPlane(r3 , P1 )
  if (out1 == 1) or (out2 == 1) or (out3 == 1) then
    return 1  // intersection exists
  end if
  r1 = Line(P1.point1 ,P1.point2)
  r2 = Line(P1.point2 ,P1.point3)
  r3 = Line(P1.point3 ,P1.point1)
  out1 = ConformalIntersectSegmentPlane(r1 , P2 )
  out2 = ConformalIntersectSegmentPlane(r2 , P2 )
  out3 = ConformalIntersectSegmentPlane(r3 , P2 )
  return  (out1 == 1) or (out2 == 1) or (out3 == 1)
```

3.3. Sphere–Sphere intersection

$$\begin{aligned} B &= (\mu_3\lambda_1-\mu_1\lambda_3)e_1e_2e+(\mu_4\lambda_1-\mu_1\lambda_4)e_2e_3e+(\mu_5\lambda_1-\mu_1\lambda_5)e_3e_1e \\ &+ (\mu_3\lambda_2-\mu_2\lambda_3)e_1e_2\bar{e}+(\mu_4\lambda_2-\mu_2\lambda_4)e_2e_3\bar{e}+(\mu_5\lambda_2-\mu_2\lambda_5)e_3e_1\bar{e} \\ &+ (\mu_5\lambda_3-\mu_3\lambda_5)e_1e\bar{e}+(\mu_3\lambda_4-\mu_4\lambda_3)e_2e\bar{e}+(\mu_4\lambda_5-\mu_5\lambda_4)e_3e\bar{e} \\ &+ (\mu_1\lambda_2-\mu_2\lambda_1)e_1e_2e_3 \end{aligned} \tag{3.5}$$

As before, B^2 will determine if a collision occurs. Algorithm 3 turned out to be as simple as it is in $\mathbb{R}^3$: one of the spheres is placed at the origin and the others moved accordingly, enabling fast computation of huge numbers of colliding spheres.

Algorithm 3: Sphere-Sphere intersection

```
kernel Sphere_Sphere_Intersect(sphere S1, sphere S2 )
  // origin set at the center of sphere S1
  ChangeCoordinatesSphere(S1)
  ChangeCoordinatesSphere(S2)
  return ConformalIntersectSphereSphere(S1,S2)
    // 1 = spheres intersect , 0 they do not
```

4. CUDA Implementation and Results

An initial implementation phase was devised in which the $\mathbb{R}^3$ to $\mathbb{R}^{4,1}$ algebraic mappings and algebraic algorithms were prototyped in MatLab™ and linked to AutoDesk Maya™. After checking for algebraic correctness, they were migrated to a CUDA implementation. All trials were performed at a 3 Ghz Dual Core Intel 2 CPU with a 64 cores NVIDIA 9800GT GPU.

GPU performance tests were executed on intersections and collisions in conformal space among several standard meshes to gather statistics. On average, they show a three order of magnitude improvement for all implemented algorithms from a pure (single core) CPU implementation of conformal space.

4.1. Mesh-Mesh Collisions

The basic CUDA procedure allows for querying whether an object, in this case a polygonal mesh, collides against any of the three primitives: line segments, triangles, and spheres. All CUDA kernels share the conformal collision query procedure, with a different post-processing phase. A typical computed collision between a large triangle (green) and the Stanford Bunny mesh (red) can be seen in Figure 1, with the intersected mesh triangles shaded in yellow to show where the plane (triangle) cuts the mesh.

Here are shown the number of collided triangles at the Bunny and Armadillo meshes in different resolutions:

Bunny - Armadillo	Bunny	Armadillo
1K - 5K	187	491
1K - 20K	185	996
1K - 100K	194	2215
1K - 345K	195	3968
1K - 1000K	107	5498
1K - 5000K	191	15063

The table corresponds to collisions of the Bunny – Armadillo meshes (Figure 2) and measured times (Figure 3). Intersected triangles are bright yellow (*Bunny*) and pink (*Armadillo*).

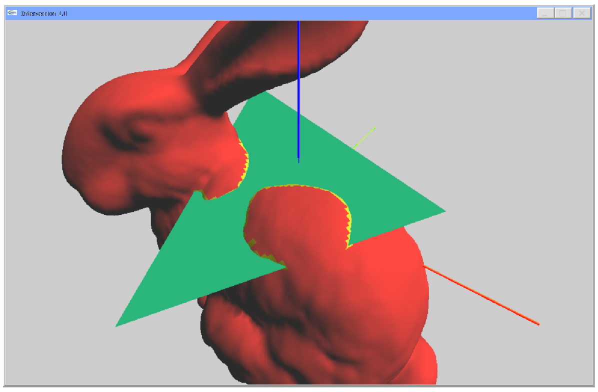

Figure 1: *Triangle - Bunny Mesh Intersection*

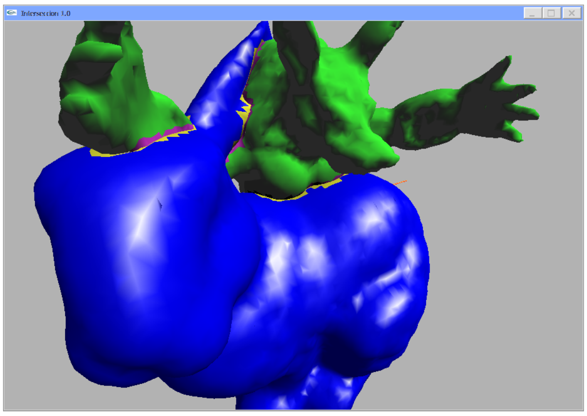

Figure 2: *Bunny mesh - Armadillo mesh Intersection*

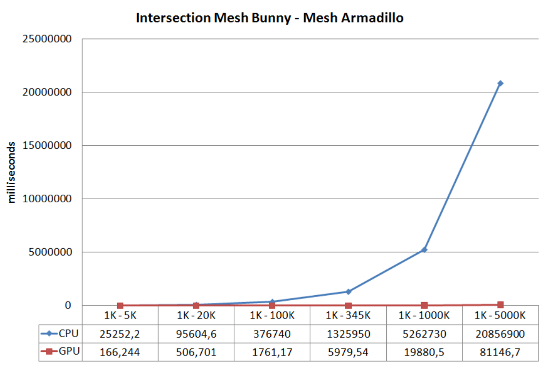

Figure 3: *Bunny mesh - Armadillo Intersection times*

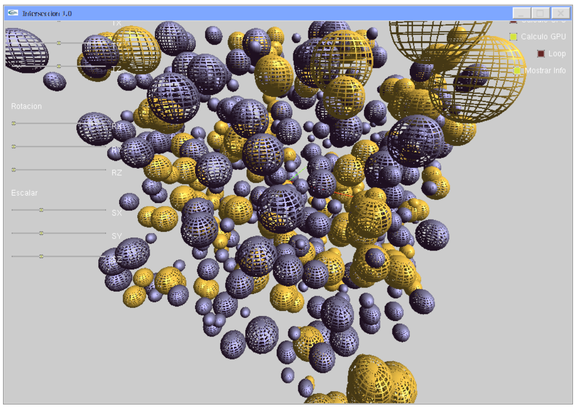

Figure 4: *Spheres - Spheres Intersection*

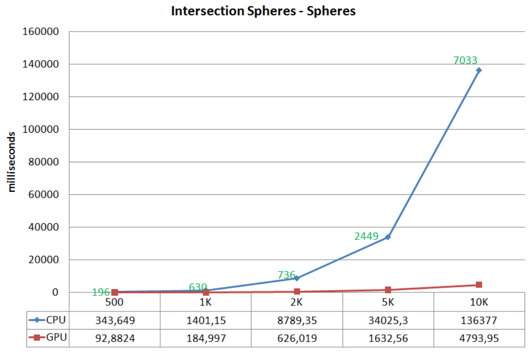

Figure 5: *Spheres - Spheres Intersection times*

4.2. Sphere – Sphere Collisions

In this setup, 500, 1K, 2.5K, 5K and 10K randomly generated spheres are intersected against each other, as seen in Figure 5 and Figure 4. The spheres that are colliding are shaded in orange. As can be appreciated, the GPU implementation offers dramatic speedups, and its curve grows much slower than the CPU implementation.

	Triangles	Milliseconds	Seconds
CPU	4000K	30034,4	30,034
GPU	4000K	715,872	0,715

For example, taking the last values of the Bunny – Line Segment intersection, even when intersecting a line segment with nearly 4 million triangles, GPU computations are still under 1 second.

4.3. $\mathbb{R}^3$ (CPU) collisions vs $\mathbb{R}^{4,1}$ (CPU) collisions

For a measuring framework of the conformal approach, Möller's optimized CPU approach for Triangle-Triangle intersection [Möl97] was implemented. The $\mathbb{R}^{4,1}$ conformal model was also implemented purely in CPU and a performance evaluation was obtained. We can appreciate in Figure 6 that Möller's CPU implementation (in **red**) is 4 times faster than the conformal model in CPU, attributed to the extra dimensionality of the latter. Thus, it is expected that a GPU implementation of the conformal would be orders of magnitude more efficient in this respect.

5. Conclusions

We have derived a unified treatment of collisions detection in conformal space, based in a reformulation of collision queries from euclidean ($\mathbb{R}^3$) to conformal space ($\mathbb{R}^{4,1}$), sharing a parallel GPU implementation of core CUDA kernels implementing collision detection as algebraic operations that indistinctly determine intersections among lines, circles, polygons and spheres.

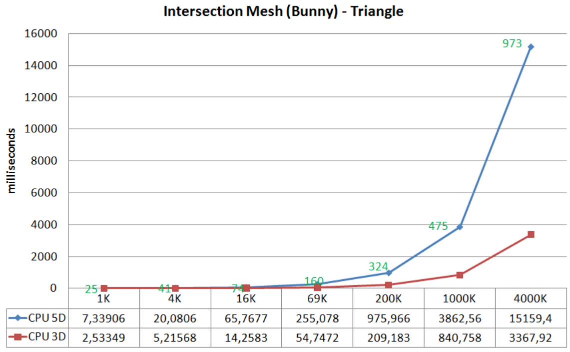

Figure 6: *Fast CPU vs. Conformal CPU Triangle Intersections*

In the results we increase throughput by two or more orders of magnitude in collision benchmarks among known mesh models, computed in blind *all vs. all* manner without any bounding volume collision pre-filtering. This means that realtime collisions of complex objects in conformal space can be computed at interactive rates.

Given that any hierarchical approach will remove large swathes of data from the computations, these values are to be considered as absolute upper limits on a worst case scenario.

Since our model does not use any acceleration techniques, it clearly signals that radical performance improvements will be observed when incorporating higher Bounding Volume Hierarchies for early pruning and other accelerating techniques to the GPU programming.

References

[BCS10] Bayro-Corrochano E., Scheuermann G. (Eds.): *Geometric Algebra Computing in Engineering and Computer Science*, first ed. Springer, 2010. 1

[DFM07] Dorst L., Fontijne D., Mann S.: *Geometric Algebra for Computer Science*. Morgan Kaufmann, San Francisco, CA, 2007. 1

[DL09] Doran C., Lasenby A.: *Geometric Algebra for Physicists*. Cambridge, Cambridge, UK, 2009. 1, 2

[Eri05] Ericson C.: *Real Time Collision Detection*. Morgan Kaufmann, San Francisco, CA, 2005. 1

[LMM10] Lauterbach C., Mo Q., Manocha D.: gProximity: Hierarchical GPU-based operations for collision and distance queries. *Computer Graphics Forum 29* (2010), 419–428. 1

[Möl97] Möller T.: A Fast Triangle-Triangle Intersection Test. *Journal of graphics, gpu, and game tools 2*, 2 (1997), 25–30. 4

[OLG*07] Owens J. D., Luebke D., Govindaraju N., Harris M., Krüger J., Lefohn A. E., Purcell T. J.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 27* (2007), 375–384. 1

[ORC07] Ortega M., Redon S., Coquillart S.: A six degree-of-freedom god-object method for haptic display of rigid bodies with surface properties. *IEEE Transactions on Visualization and Computer Graphics* (May/June 2007), 458–469. 1

[Roa11] Roa E.: *Operaciones de Cómputo Gráfico en el Espacio Geométrico Conforme 5D usando GPU*. Master's thesis, Universidad Simón Bolívar, Venezuela, February 2011. `http://www.ldc.usb.ve/~vtheok/thesis/gpuconformal.pdf`. 2

[SK10] Sanders J., Kandrot E.: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, Boston, 2010. 1

[TFN10] Theoktisto V., Fairen M., Navazo I.: Hybrid Rugosity Mesostructures (HRMs) for fast and accurate rendering of fine haptic detail. *CLEI Electronic Journal 13* (December 2010), 1–12. paper 6. 1

[Vin08] Vince J.: *Geometric Algebra for Computer Graphics*. Springer, London, UK, 2008. 1