HEIG-VD
Haute Ecole d'Ingénierie et de Gestion
de Canton de Vaud

B A C H E L O R T H E S I S

GPU-accelerated triangle-triangle intersection tester algorithm

Author: Romain Maffina

Thesis Advisor: Stephan Robert

Mandator: Pixelux Entertainement

July the 27th, 2012

Specifications

The goal of the project is to develop a triangle-triangle collision algorithm. A reference triangle is given as well as a variably-sized array of many other triangles. The algorithm must check if one triangle intersects with the reference triangle. That operation has to be led for each "non-reference" triangle with the reference triangle. If one triangle intersects with the reference triangle the algorithm must return a true value, otherwise (if none of the triangles in the array intersects with the reference triangle) it must return a false value. The algorithm has to be as fast as possible. Therefore we are trying to use the capabilities of massively parallel computations with GPUs (Graphics Processing Units). In particular the NVIDIA technology was chosen. Therefore the implementation of the algorithm is done in CUDA C.

Here is the specifications given by *Pixelux Entertainement* on March 1st, 2012 :

> The project is to build a CUDA-accelerated triangle-triangle intersection test. This could benefit our auto-cage tool that constructs watertight "cages" used for tessellation around geometry objects.
>
> The project is defined as such:
>
> 1. Assume you have a triangle. A triangle is represented as a 2D object but it is represented in 3D coordinates. So very simply defined as a set of 3 3D coordinates.
> 2. Assume you have a list of triangles that you will be comparing a single triangle to. I would imagine that in CUDA, this would ideally be represented as some sort of array.
> 3. Write a CUDA function that compares a single triangle to the list of triangles and returns True or False. True if the triangle intersects any of the triangles in the list, and False if it does not.
>
> We prefer the code be written in "C" and, of course, CUDA.

Contents

Abstract

During this thesis, after having firstly developed a testing and viewing program to help me in my task, I've realized the algorithm to test the intersection between two triangles in a 3D space. This algorithm has been then integrated in a higher level function to meet the specifications of computing the intersection for each triangle of an array with a reference triangle. The function was first only developed for the CPU. Then, after many corrections and optimizations too ensure the rightness and speediness of the algorithm, I've tested to port the function to CUDA while forming myself about this technology. The first part of my thesis ends here.

Then, in the second full-time part of my thesis, I've ported the CPU version of the function to CUDA as cleanly as possible but above all trying to optimize it to make it as fast as I could. After that, the remaining time was dedicated to secure the function and make it reliable. At the same time, I've enhanced the functionality of my test program for that purpose.

Chapter 1

Introduction

In the past few years, a new computation power as been "discovered" for general-purpose computing. Discovered is maybe not exactly the good word, in fact, it already exists for a longer time, but no one had the idea to use it this way until a few years ago. The computation power I'm talking about is known as *GPGPU*[1] and is the means of using a *GPU*[2] to perform computation in general-purpose application that would have normally been handled by the *CPU*[3].

Last year, during summer university courses, I had an introduction to *CUDA*[4] technology invented by *Nvidia Corporation*. It's precisely a concrete way to implement *GPGPU* using the NVIDIA compatible end-user GPUs. The first SDK for CUDA as been released on February 2007.[5] The course interested me a lot as well as the idea of *GPGPU*, I thought this technology could be a future unavoidable way of programming and thus I'd like to see it diffused as much as possible.

Therefore, when the time come to me to choose a subject for my thesis, I quickly thought about *GPGPU* and *CUDA* technologies. So I contacted my professor, Stephan Robert, who gave me the CUDA course, and asked him if he would agree to be my thesis advisor for a thesis based on GPGPU. He gladly accepted and began to contact some companies to ask them if they were interested to parallelize with CUDA some part of their algorithm. Unexpectedly, a promising company, *Pixelux Entertainement*, answered affirmatively.

> [1] In Febuary 2004, Pixelux Entertainment S.A. is founded in Geneva, Switzerland with the intent of developing technology that would automate art asset generation for video games through advanced simulation. To that end, the Company creates a unique real-time material physics simulation technology called "Digital Molecular Matter" or DMM.

Because they must handle very heavy tasks in their DMM, they were interested by the idea and they proposed me to build a CUDA-accelerated triangle-triangle intersection test. This could benefit their auto-cage tool that constructs watertight "cages" used for tessellation around geometry objects.

[1] General-purpose computing on graphics processing units[6]
[2] Graphics Processing Unit
[3] Central Processing Unit
[4] Compute Unified Device Architecture

Chapter 2

Setting up

2.1 Some considerations

The task given is very clear and simple to understand. It is however not so simple to realize and here are some important considerations :

First thing, there is not a unique way to realize a triangle-triangle intersection tester but a lot of different ways! The only thing they have in common is that they are all based on geometric tools. Also one can mix some parts of those different algorithms to realize it. Moreover, the algorithm could be more or less complicated, depending on how it must be quick and optimized (i.e. already eliminate triangles that can't intersect for sure) and so on. So I first decided to study some of theses algorithms to have a good overall understanding of what is possible to do and then decide a way to implement it. The searches I've done on the Internet were not very lucky, so I quickly tried to find a good reference book that could explains it to me in a more complete and comprehensive way. The book I've found is called *Geometric Tools for Computer Graphics* [2] and have a section who explain a way to check intersections between two triangles. I've decided to follow the steps they present.

Then, as mentioned before, a lot of geometry need to be used, especially the use of vectors and the associated operations. So I strongly preferred to develop a library for it. However *Pixelux Entertainement* said they prefer C code for the function, which is actually quiet good because CUDA is mostly used with C programming language, but it leads to an extra complication. Indeed C code is not very indicated to develop such a library with basic operations as plus, minus and so on because it don't have the operators overloading, the use of the library become a lot less intuitive.

Finally, I thought that developing such a tough algorithm directly in a CUDA way will probably always lead to a waste of time. It's the case because CUDA brings another abstract layer over the natural trickiness of the problem, in this way a tricky algorithm become a *very* tricky algorithm. Furthermore, as the standard function is written in C code, it's not such a lot of work to port it to CUDA. Finally, this allows me to do then a very interesting and useful thing : compare the same algorithm in both CPU and GPU version to see the gain CUDA brings. This could be very informative for *Pixelux Entertainement* to see if CUDA is interesting for them.

Now that the fullness of the job is better understood, let's organize it in the next section.

2.2 The chosen approach

Because of the considerations made before, I've decided to organize myself in this way :

1. Develop the algorithm as a standard C function using the CPU to compute the collisions
2. Modify then the C function to a CUDA function that will execute itself on the GPU
3. Optimize the algorithm

2.3 First steps

The first development step were to create a testing environment. So I've started by writing a random triangle array generator function and an other function to *printf* them. Then I quickly realized how hard it was to imagine how the triangles were placed in the 3D space and even to visualize the shape of each triangle by only reading a list of coordinates!

So my decision was quickly made : I needed to develop a minimalistic viewer to see the triangles in the 3D space and be able at least to rotate the camera around them. *OpenGL* came immediately to my mind to realize it.

2.3.1 The OpenGL viewer

Because the collision between two triangles can only be checked visually, a viewer is essential and necessary. It will be also very important during the test phase. The goal of this viewer is not to be very user-friendly or beautiful so it's minimalistic.

I think that it's not pertinent to explain how this viewer has been developed as it's only part of the test program. However here are some figures that show how it looks.

The Figure 2.1 shows a standard random test with 10 triangles generated randomly. The reference triangle is painted in blue. The algorithm then checks the collisions and marks each triangle that intersect with the reference triangle in orange and leave the other ones in grey. One can rotate around the Y axis with the mouse and zoom in/out with the keyboard. A new bunch of random triangles can be generated by pressing the space key.

The Figure 2.2 shows a debugging situation where a pair of triangle is isolated. The viewer draws also some helpers : the intersection of each triangle with the line $\mathcal{L}$, the intersection of the triangles planes.

The Figure 2.3 shows another debugging situation where triangle's planes are shown.

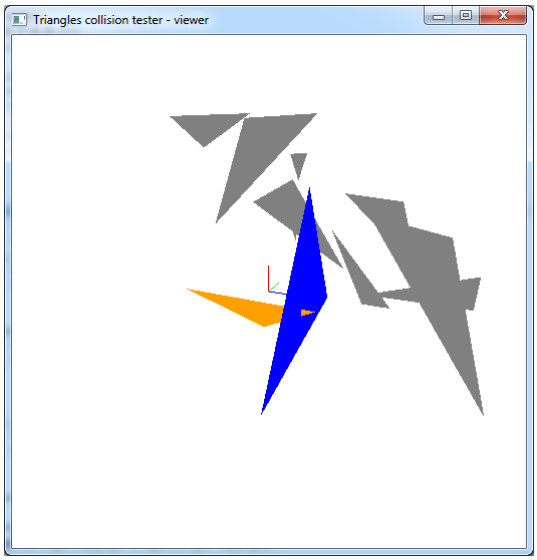


Figure 2.1: Example of the viewer with 10 triangles plus the reference triangle.

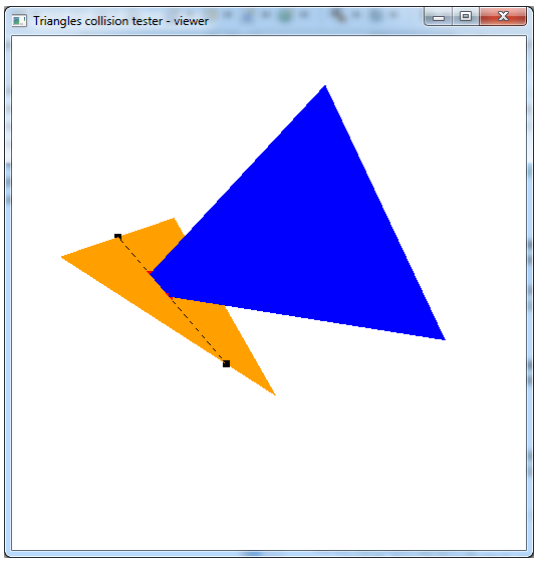


Figure 2.2: Example of the viewer with only a pair of triangles, used to debug.

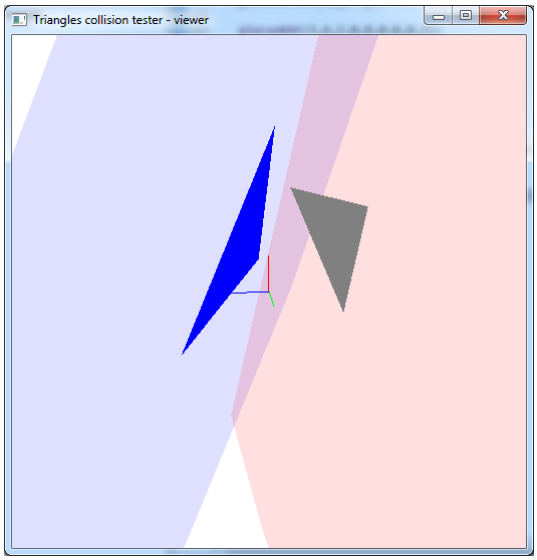


Figure 2.3: Example of the viewer with a pair of triangles and they respective planes, used to debug.

2.3.2 The vector library

As the algorithm will use a lot of geometric concepts, I've decided to create a library to group them. It will contain the types definitions, the operations like dot product and some functions like a function to find the projection of a triangle on a plane, and so on.

I will not present this library here because it's "only" some mathematics tools, the full source code listing can be found in appendix D.1.

Chapter 3

Develop the algorithm as a standard C function

3.1 First version

This implementation is based on the steps proposed in the book *Geometric Tools for Computer Graphics* [2, p.542] and here is an overview of what I will going trough :

quick rejection test : This first step already eliminate with a simple test pair of triangles that cannot intersect.

triangle projection on other triangle plane : The goal of this is to pass from the 3D world to a "simpler" 2D world.

2D intersection test : Now that we're in a 2D world we can perform a "simple" 2D intersection test.

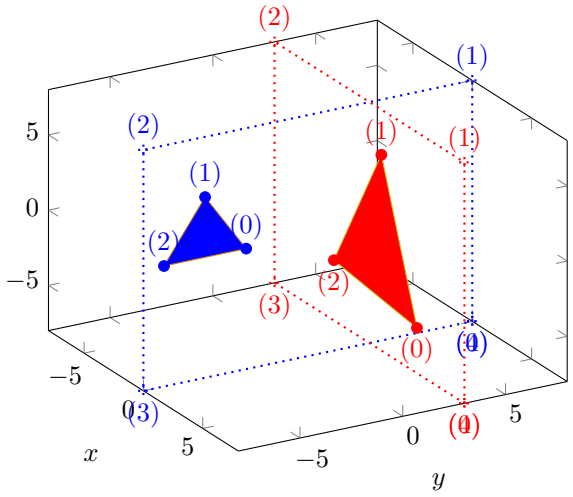
3.1.1 The quick rejection test

Given two triangles $\mathcal{T}_0$ and $\mathcal{T}_1$, the goal is to do an early check and already eliminate pairs of triangles that cannot intersect. Indeed, if all vertices of T_0 are all on the same side of T_1 plane or vice-versa, we already can say that the two triangles can't intersect so it's useless to go further. The figure 3.1 illustrate this : the blue triangle is entirely on the same side of the red triangle's plane so we already can eliminate this pair. Note that in this case you cannot decide to eliminate the pair with the reverse test because the red triangle intersects the blue plane.

To do this check, I compute for each vertex of the triangle the signed distance of this vertex to the plane. [2, p.374] This way, a positive value indicates that the vertex is on side A of the plane, a negative value indicate that the vertex is on side B of the plane and a zero indicate that the vertex lies exactly on the plane. Thus, if the distances are both positive and negative, the pair is kept as well as if one or more distances are zero. Otherwise the pair is rejected.

The signed distance is computed as following : We consider a point Q and a plane $\mathcal{P} : \{P, \vec{n}\}$, where $\vec{n}$

Figure 3.1: Blue triangle is entirely on one side of the red triangle's plane



is a normal vector of the plane and P is a point on $\mathcal{P}$.

$$dist = \vec{n} \cdot \vec{PQ}$$

The triangle's plane is compute as following : We consider a triangle $\mathcal{T}$ and his 3 points : t_0, t_1, t_2. The plane is defined as $\mathcal{P}$: $\vec{n}$, P and d, where $\vec{n}$ is a normal vector of the plane, P is a point on the plane and d the distance from plane to origin.

$$\vec{n} = t_0\vec{t}_1 \times t_0\vec{t}_2$$

$$P = t_0$$

$$d = \textit{distance from plane to origin, see above}$$

3.1.2 Triangle projection on a plane

The projection of the triangle onto the plane is actually a vertex projection onto a plane done three times. [2, p.663] So we consider the projection Q' of a point Q onto a plane $\mathcal{P}$. The plane is defined as $\mathcal{P} : \vec{n}$, P and d, where $\vec{n}$ is a normal vector of the plane, P is a point on the plane and d the distance from plane to origin. The projected Q' point is computed as following :

$$Q' = Q - (Q \cdot \hat{n} + d)\hat{n}$$

Note that here it's important to use $\hat{n}$, the $\vec{n}$ vector normalized.

3.1.3 2D intersection test

The method I used to realize a 2D intersection test is the one called *The method of separating axes*. This method can only test convex polygons and here is the mathematical fundamental on which the method is based :

> [2, p.265] If there exists a line for which the intervals of projection of the two objects onto that line do not intersect, then the objects do not intersect. Such a line is called a *separating line* or, more commonly, a *separating axis*.

Figure 3.2 shows examples of non-overlapping and overlapping projections on a random separating axis.

In fact only a finite number of separation axis needs to be considered for this test. It's only the separation axis defined by each normal of each edge of both triangles. So we have to consider only 6 separation axis as shown in Figure 3.3. On each of them we will project both triangles and check if they are separated or not along the axis. Separated means that the projections of the two triangles don't overlap. If on all 6 tested axis the projections overlap then the two triangles intersect, otherwise, if one or more of the six axis shows non-overlapping projections, then the two triangles don't intersect.

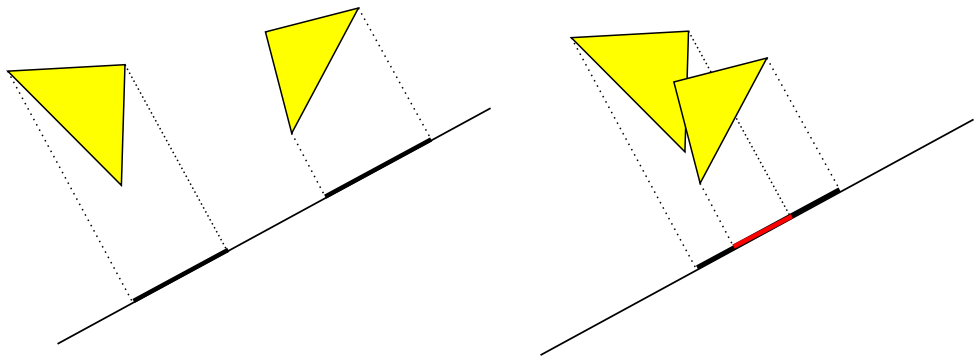


Figure 3.2: Examples for non-overlapping and overlapping projections.

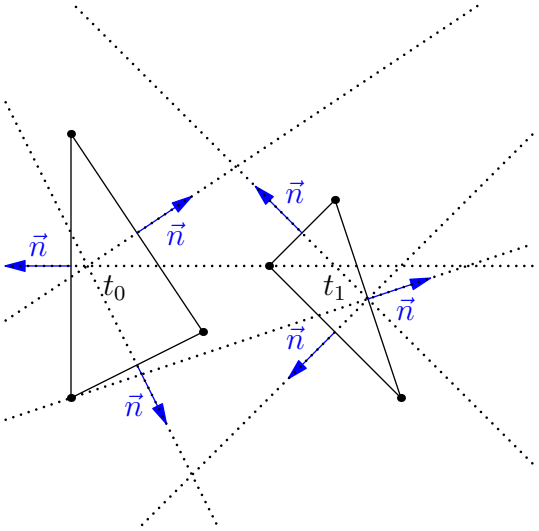


Figure 3.3: Example of 2 triangles and the 6 separations axes to consider for the test.

3.1.4 Problem : Particular case not handled

Coming to the very end of the realization of this algorithm and after testing it a problem appears to me : there is a very particular case in which the algorithm detects a false positive. Figure 3.4 shows one among many other triangles configurations that causes that.

To explain why a false positive is detected let's take again the main steps of the whole algorithm and applies to them our example.

Quick rejection test : $\mathcal{T}_1$ with $\mathcal{P}_0$
Two vertices of $\mathcal{T}_1$ lies on one side of $\mathcal{P}_0$ and the last vertex lies on the other side so the pair is not rejected.

Quick rejection test : $\mathcal{T}_0$ with $\mathcal{P}_1$
Two vertices of $\mathcal{T}_0$ lies on one side of $\mathcal{P}_1$ and the last vertex lies on the other side so the pair is not rejected as well.

2D intersection test on $\mathcal{P}_0$
After projecting $\mathcal{T}_1$ on $\mathcal{P}_0$ we do a 2D intersection test following the method of separating axis. The problem appears when we project $\mathcal{T}_1$ on $\mathcal{P}_0$, indeed while the two triangles do not intersect when they are in a 3D world they do when they are in a 2D world! Thus the 2D intersection test return true and so all the conditions are gathered to mark this pair of triangles as intersecting when they aren't in fact... The problem is exactly the same if you do the reversed 2D intersection test with $\mathcal{T}_0$ projected on $\mathcal{P}_1$.

First I did some attempt to find a workaround to this problem but I had to capitulate because the problem was not in the code but in the theoretical algorithm. So I'll have to change the algorithm partly. The fact is that, as mentioned before, I've followed the algorithm given in the GTCG [2] but I misunderstood it. It may be the fact that it's written in English and so it's a bit harder for me to understand. I also have to say that the triangle-triangle intersection section is not very clear.

Indeed the section is constructed this way : the first part explains with text a method to realize the intersection test (the method is the *interval overlap method* describe by Möller and Haines) and then in the second part they explains the algorithm step-by-step in a summary way. The problem is that the step-by-step summary is quiet messy and do not made clear reference to what was said before in the first part. This was quite confusing for me. Here is the summary I'm talking about[1] :

> [2, p.542] [...]
> An outline of the entire algorithm is as follows:
>
> 1. Determine if either $\mathcal{T}_0$ or $\mathcal{T}_1$ (or both) are degenerate, and handle in application dependent fashion (this may mean exiting the intersection algorithm, or not).
> 2. Compute the plane equation of $\mathcal{T}_0$.
> 3. Compute the signed distances $\text{dist}_{V_{1,i}}$, $i \in \{0, 1, 2\}$, of the vertices of $\mathcal{T}_1$.
> 4. Compare the signs of $\text{dist}_{V_{1,i}}$, $i \in \{0, 1, 2\}$: if they are all the same, return *false*; otherwise, proceed to the next step.
> 5. Compute the plane equation of $\mathcal{T}_1$.

[1] Note that the signed distances of the vertices of $\mathcal{T}_0$ are never computed in thoses steps, it must a forgetting.

6. If the plane equations of $\mathcal{P}_0$ and $\mathcal{P}_1$ are the same (or rather, within $\in$), then compare the d values to see if the planes are coincident (within $\in$) :
   - If coincident, then project the triangles onto the axis-aligned lane that is most nearly oriented with the triangles' plane, and perform a 2D triangle intersection test.[2]
   - Otherwise, the parallel planes are not coincident, so no possible intersection; exit the algorithm.
7. Compare the signs of $\text{dist}_{V_{0,i}}$, $i \in \{0, 1, 2\}$: if they are all the same, return *false*; otherwise, proceed to the next step.
8. Compute intersection line.
9. Compute intervals.
   - If no interval overlap, triangles don't intersect. Return *false*.
   - Otherwise, if intersecting line segment is required, compute it. In any case, return *true*.

First I understood that the part where the intersection test was done was the point 6 and only there. A 2D intersection test using the method of separating axes. This is why I've first done it the way explained above.

Then I realized, reading it again and again trying to understand what was wrong, that the standard intersection test was performed in point 8 and 9. The point 6 explains how the algorithm have to handle the special case of two triangles being in the same plane (*coincident* in a math speaking way) by doing a special 2D intersection test.

Once I realized it I started to change the algorithm partly and code a second version of it. This second version is detailed in the next section.

[2] *The method of separating axes (Section 7.7) can be used to determine whether or not and intersection exists. Generally speaking, [. . .]*

(a) Front view (b) Back view

Figure 3.4: Example of a special triangle configuration that causes the algorithm to detect a false positive.

3.2 Second version

Well as explained in the previous section, I misunderstood the *Geometric Tools for Computer Graphics* book [2] I'm following to realize this triangle intersection tester and so my first version of the algorithm, quiet logically, didn't worked at 100%. Thus I started to modify it to get it working. This section will present the second, corrected, version of the algorithm. I will now use, according to the GTCG [2, p.539], the *interval overlap method* described by Möller and Haines as the main method to detect intersections. The method used before in the first version of the algorithm is not lost and will be used to compute intersections when the triangles are in the same plane. Indeed the *interval overlap method* don't handle this special case.

Let's have a overview of the steps involved :

quick rejection test : As the first version, this first step already eliminates with a simple test pair of triangles that cannot intersect. See section 3.1.1 for more details.

Intersection of $\mathcal{P}_0$ and $\mathcal{P}_1$ resulting in a line $\mathcal{L}$: The goal is to obtains the line corresponding to the intersection of $\mathcal{P}_0$ and $\mathcal{P}_1$. If the planes are parallels there is no intersection and so no line is returned.

If the line exists we use the *interval overlap method* to check the intersections : Both triangles will now be projected on $\mathcal{L}$. Prior to the projection the vertices of each triangles needs to be ordered. Each triangle's projection will result in an interval on the line. If the intervals overlap, then the triangles intersect, otherwise, they do not.

Else we use the *method of separating axes* : If $\mathcal{L}$ doesn't exists this means that the planes are parallels. Thus we are in 2D world and we can use a 2D intersection test to check the intersections. The method is used exactly as in the first version.

I will now present in the next sections the new parts of this second version.

3.2.1 Intersection of $\mathcal{P}_0$ and $\mathcal{P}_1$ resulting in a line $\mathcal{L}$

The planes are defined as $\mathcal{P} : \vec{n}$, P and d, where $\vec{n}$ is a normal vector of the plane, P is a point on the plane and d the distance from plane to origin. The line $\mathcal{L}$ is defined as a point P and a direction $\vec{d}$.

The direction $\vec{d}$ can be easly computed by doing the cross product of the two plane's normals :

$$\vec{d} = \vec{n_1} \times \vec{n_2}$$

Now we need a point on this line to completely specify it. Assuming that this point will be a linear combination of $\vec{n_1}$ and $\vec{n_2}$ so :

$$P = a\vec{n_1} + b\vec{n_2}$$

We can also say that the point will be on both planes so :

$$\vec{n_1} \cdot P = s_1$$

$$\vec{n_2} \cdot P = s_2$$

s_1 and s_2 are the distance to origin for both planes.

We substitute and solve for a and b :

$$a = \frac{s_2\vec{n_1} \cdot \vec{n_1} - s_1||\vec{n_2}||^2}{(\vec{n_1} \cdot \vec{n_2})^2 - ||\vec{n_1}||^2||\vec{n_2}||^2}$$

$$b = \frac{s_1\vec{n_1} \cdot \vec{n_1} - s_2||\vec{n_1}||^2}{(\vec{n_1} \cdot \vec{n_2})^2 - ||\vec{n_1}||^2||\vec{n_2}||^2}$$

The line $\mathcal{L}$ is now fully defined.

3.2.2 Projection of $\mathcal{T}_0$ and $\mathcal{T}_1$ on the line $\mathcal{L}$

Here is used the *interval overlap method* described by Möller and Haines. The projection of each triangle is done by projecting each of it vertices on the line $\mathcal{L}$ computed in the section 3.2.1.

Assuming the following : $V_{0,0}$, $V_{0,1}$ and $V_{0,2}$ are the vertices of $\mathcal{T}_0$, the projection of each vertex is done as following :

$$V'_{0,i} = \hat{d} \cdot (V_{0,i} - P), \qquad i \in \{0, 1, 2\}$$

Note that $\vec{d}$, which is the direction vector of the line, must be normalized first.

3.2.3 Order the vertices of each triangles prior to the projection on $\mathcal{L}$

To get the Möller and Haines method working the vertices of each triangle must be ordered in a special way. Indeed the algorithm assumes that in a standard situation vertices V_0 and V_1 lie on one side of $\mathcal{L}$, and the vertex V_2 on the other side. Moreover the special cases (one point is on the line i.e.) need some attention as well. So to respect theses constraints the vertices needs to be ordered prior to compute the interval. The Figure 3.5 shows all possible placements relative to the line with the particular cases. The working vertex configuration is shown in green for each case. Note that V_0 and V_1 can always be swapped. Contrariwise V_2 has to be placed exactly as shown in every cases.

The vertex configurations for the special cases are absolutely not treated in the book so I had to do a lot of testing to see which configuration works with the algorithm and which don't.

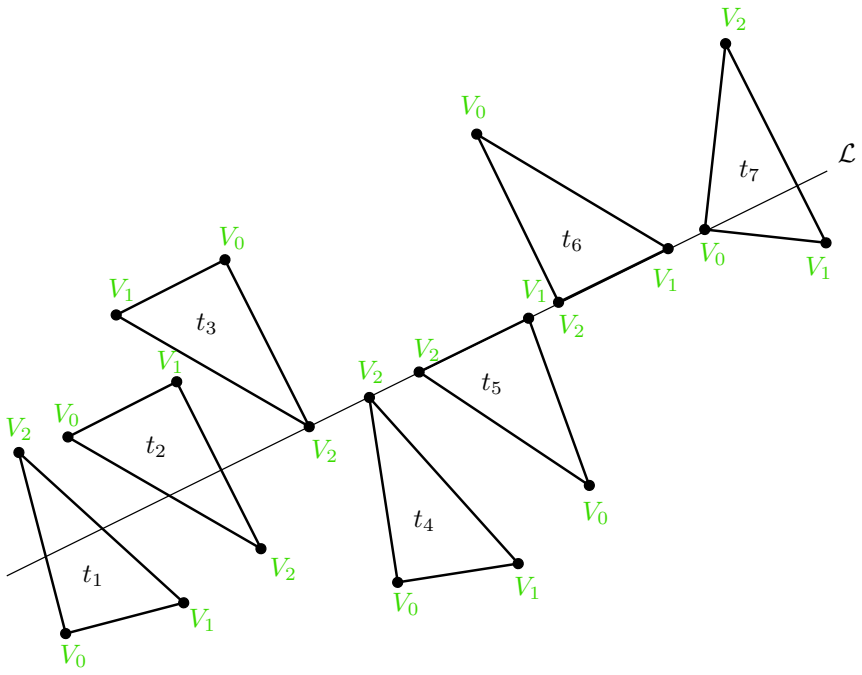


Figure 3.5: All possible placements for a triangle relative to the line.

3.2.4 Compute each triangle interval on $\mathcal{L}$

Assuming the following : $V_{0,0}$, $V_{0,1}$ and $V_{0,2}$ are the vertices of $\mathcal{T}_0$, the interval $t_{0,0}$ to $t_{0,1}$ of $\mathcal{T}_0$ is compute as following :

$$t_{0,i} = V'_{0,i} + (V'_{0,2} - V'_{0,i}) \frac{dist_{V_{0,i}}}{dist_{V_{0,i}} - dist_{V_{0,2}}}, \qquad i \in \{0, 1\}$$

The same is done to compute the interval $t_{1,0}$ to $t_{1,1}$ of $\mathcal{T}_1$. Now that we have both interval on $\mathcal{L}$ we simply have to check if they overlap or not. If the answer is yes this means that the triangles intersect otherwise not.

3.3 Test bench of the algorithm

Testing well this algorithm is quite a tricky job. Indeed the only way to approve the rightness of the algorithm is to do a visual check. This means in my case using my eyes to check everything. Moreover the number of possibilities is infinite for two triangles to be placed in a 3D space. This leads to the fact that I simply cannot test everything. So to cover as much problematic possibilities as possible I've decided to split my test bench in 2 parts. A random part and a special cases part.

3.3.1 Random tests

I've proceeded this way to check a maximum number of configurations : In my testing program, I generate a fixed number of triangles (randomly sized) like 10 to 50 (while I can clearly see the intersections with the reference triangle) at random positions around the reference triangle. Then I check if the triangles that the algorithm has marked as *intersecting* (repaint in a orange color) are indeed visually intersecting. Then the same is done for the triangles marked as not intersecting by the algorithm (grey color). When I've checked everything and it's O.K. I generate a new bunch of random triangles and check again.

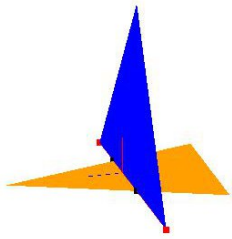
When I see a problem, I store the coordinates of the problematic triangle and I change the testing program to generate only the reference triangle and the problematic triangle. Then I try to locate the problem using the debug mode and some visual helpers drawn with OpenGL like a triangle's plane, the projected triangle or the projection on the line, etc.

3.3.2 Special cases

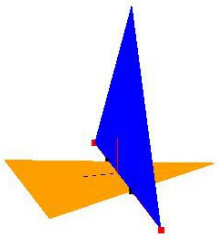
To control more in depth the rightness of my algorithm, I've tested all the special cases mentioned in the Figure 3.5 which shows all the vertex configurations of a triangle relative to the line $\mathcal{L}$. This is because the projection of the vertices is the critical part of the algorithm. The cases are presented in the figures below using the minimalistic viewer I've developed (see Section 2.3.1 for more details about the viewer). I've done a extra test in which the triangles are in the same plane because in this case the algorithm don't use the standard intersection test (the Möller and Haines one) but the 2D intersection test (using the separating axes method) who has to be tested as well.

Note that all the cases tested are presented below while intersecting but of course I've controlled as well

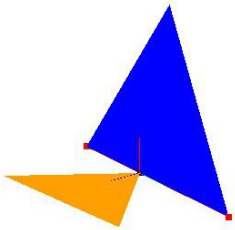
that, by moving the triangle step-by-step, once the triangles don't visually intersect the algorithm "see" the same as me and repaint the array triangle in grey.



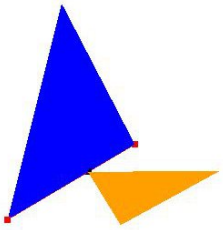
(a) t_1 case.



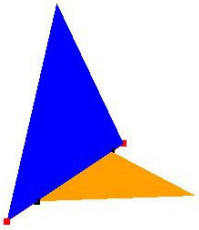
(b) t_2 case.



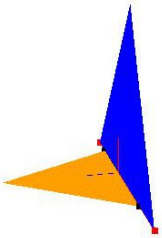
(c) t_3 case.



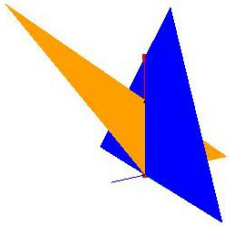
(d) t_4 case.



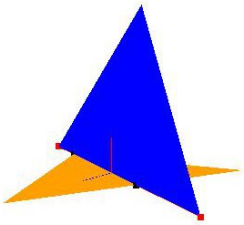
(e) t_5 case.



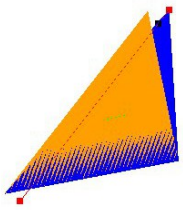
(f) t_6 case.



(g) t_7 case.



(h) t_7 case.



(i) Both triangles are in the same plane.

Figure 3.6: Special cases tested relative to the configurations presented in the Figure 3.5.

Chapter 4

Port the C function to a CUDA function

4.1 Toolkit version used

Currently the last stable version of the CUDA toolkit is the 4.2. Therefore I'v developed using this version. See appendix A for more details on how to install CUDA.

4.2 Introduction to CUDA

4.2.1 Important keywords

Prior to anything it's important to introduce a bit of vocabulary :

host : The CPU and the system's memory

device : The GPU and its memory

kernel : A function that executes on the device

To write CUDA application we in fact write some standard C code but with some CUDA extensions, here are explained some function qualifiers :

___global___ : To qualify a kernel, a function that executes on the device

___device___ : To qualify a function that will be called from a kernel

___host___ : To qualify a function that will be called from a host function

Here is a example of use : `__device__ void sum(int a, int b, int *c){...}`

4.2.2 Device properties

The CUDA API provide a way to recover all the device properties. Here is a code snippet to demonstrate how to recover for example the name of the device :

```
1 cudaDeviceProp devProp;
2 cudaGetDeviceProperties(&devProp, 0);
3 //...
4 char name[256] = devProp.name;
5 //...
```

Also an important aspect for a device is his *Compute Capability*, it's a term used by NVIDIA to describes the features supported by a CUDA-capable device. The compute capability specific properties can be found on the Wikipedia page of CUDA [5].

4.2.3 General pattern of a CUDA application

Because the device has his own memory and he cannot access the host memory, the data need to be copied from the host memory to the device memory prior to the kernel launch. For the same reason, the resulting data needs then to be copied back to the host. All of this implies of course to allocate the memory on the device before anything. So generally, in all CUDA application the general pattern is always the same :

1. `cudaMalloc()` - We allocate memory on the device for in/out data according to our needs
2. `cudaMemcpy(cudaMemcpyHostToDevice)` - We copy data from host to device in the space we allocated before
3. `kernel<<<blocks,threads>>>()` - We launch the kernel
4. `cudaDeviceSynchronize()` - We wait for the device to finish computation
5. `cudaMemcpy(cudaMemcpyDeviceToHost)` - We copy the result data back from device to host
6. `cudaFree()` - We free the memory on the device

You can see that the kernel launch is like a normal function call but with extras brackets. It's here that happens all the magic. The parameters passed using the brackets, the number of blocks and the number of threads, configure the parallel execution that will be executed on the device. The next section will talk a bit more about the kernel.

4.2.4 The kernel

As explained before a *kernel* is basically a function that executes on the device. Now to understand where is hidden all the massive parallelism power let's have a example.

We have a list of sums to compute, therefore we have 3 arrays of the same size. The first contains the operand a, the second the operand b and the last array the result of each sum as c. The following snippet shows a possible function to realize this using standard C code :

```
1 void add(int *a, int *b, int *c, int size){
2     int i;
3     for(i=0;i<size;i++)
4         c[i]=a[i]+b[i];
5 }
```

The CPU will therefore execute each sum one after the other until he have computed everything.

The total theoretical execution time of this function is :

$$T_{total} = T_{sum} * size$$

Now here is how the same function looks with CUDA C to take advantage of the massive parallelism power of the GPU :

```
1  __global__ void add(int *a, int *b, int *c, int size){
2      int i = threadIdx.x;
3      if(i<size)
4          c[i]=a[i]+b[i];
5  }
6
7  void main(void{
8      //...
9      kernel<<<1,size>>>(a,b,c,size);
10     //...
11 }
```

Let's review, one after the other, the changes relative to the CPU version. First, the `__global__` qualifier in front of the function header, this qualifier alerts the compiler that the following function is intended to execute on the device instead of the host.

Then you may have noticed the lack of for loop! It has been replaced by a simple check to ensure that we are not out of bound. Therefore you may ask yourself how can the array be fully computed if no more for loop iterate them.

Well, let's have a look at the kernel launch. As you can see two extras parameters are passed to the function using the brackets, it's respectively the number of blocks and the number of threads to use to execute the function. I will give more details later about the blocks, for now we will stay with something more familiar to every programmer : the threads. So by doing `<<<1,size>>>` I'm telling to the compiler that I want to launch *size* numbers of parallel threads on the device to execute my function.

Now the last unexplained line in the code is `int i = threadIdx.x;` and hopefully this is the glue that links everything. The threadIdx built-in CUDA variable indicates for each thread executing this code his id. Assuming that size=4 figure 4.1 shows a representation of the 4 threads launched executing the code.

So each thread is responsible for computing one sum of the array.

Now the total theoretical execution time of this function is only the time spent to compute one sum :

$$T_{total} = T_{sum}$$

Obviously this is a theoretical time and the memory allocation and copy plus other things are not taken in account.

```
1 __global__ void
2 add(int *a, int *b, int *c, int size){
3     int i = 0;
4     if(i<size)
5         c[i]=a[i]+b[i];
6 }
```

(a) Code executed by thread 0

```
1 __global__ void
2 add(int *a, int *b, int *c, int size){
3     int i = 1;
4     if(i<size)
5         c[i]=a[i]+b[i];
6 }
```

(b) Code executed by thread 1

```
1 __global__ void
2 add(int *a, int *b, int *c, int size){
3     int i = 2;
4     if(i<size)
5         c[i]=a[i]+b[i];
6 }
```

(c) Code executed by thread 2

```
1 __global__ void
2 add(int *a, int *b, int *c, int size){
3     int i = 3;
4     if(i<size)
5         c[i]=a[i]+b[i];
6 }
```

(d) Code executed by thread 3

Figure 4.1: Representation of 4 GPU threads executing the code.

4.2.5 Using blocks and threads together

In the previous section I talked about blocks but without explaining anything, let's see now what blocks are :

Block 0	Thread 0	Thread 1	Thread 2	Thread 3	...
Block 1	Thread 0	Thread 1	Thread 2	Thread 3	...
Block 2	Thread 0	Thread 1	Thread 2	Thread 3	...
Block 3	Thread 0	Thread 1	Thread 2	Thread 3	...
...	...	...	...	...	...

So a block can simply be seen as a collection of threads. It here to simplify the threads management, for the developer as well as for the hardware. However one can ask why can't I simply always use threads? The answer is simple : you can't launch as much threads as you want, each device have a maximum threads per block limitation. For example on the machine I'm currently developing, I have a NVS 4200M, the limitation is 1024. Therefore if I want to launch more than 1024 parallel running threads I need to use blocks as well.

Still on my NVS 4200M the max number of blocks I can launch on a 1D grid[1] is 65535. Therefore by combining blocks and threads I can launch up to :

$$maxBlocks * maxThreads = 65535 * 1024 = 67'107'840 \text{ parallel threads}$$

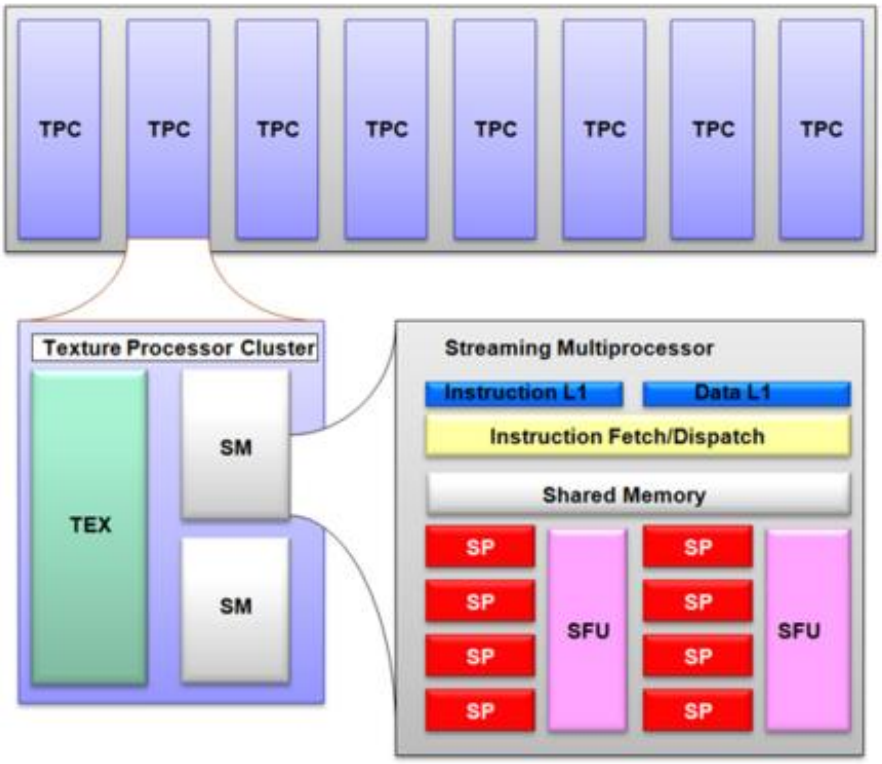
Which is now a comfortable number of potential[2] parallel computations!

[1] CUDA also allows to index a grid in 2D or 3D depending on the compute capabilities of the device. I only use 1D indexing in my project so I won't explain more about it.

[2] depending on the compute capabilities of the device

4.2.6 Hardware architecture

Until now, I've presented CUDA from the software point of view, it's time now to go a bit further and see how the blocks, threads and resources are handled from the hardware point of view. Here is a representation of the GPU architecture from `http://www.tomshardware.com/` :



Thus the GPU is composed of SMs (stream processors) and each SM contains a number of SPs (processors or also called CUDA cores by NVIDIA). The number of SM is dependant of the device hardware and the number of SP per SM is defined by the *Compute Capability* of the device. Here is the list of corresponding number of SPs relative to the *Compute Capability*:

CC	SPs
1.0	8
1.1	8
1.2	8
1.3	8
2.0	32
2.1	48
3.0	192

When a kernel is launched the corresponding grid of blocks is passed to the GPU. Each SM will then take care of one or more blocks until its maximum capacity, the remaining blocks are queued. A block is completely treated by one and only one SM until it finish. He can then free the slot for one of the remaining blocks waiting. This is done until all blocks of the grid have been treated. One can now

understand that despite an already important parallelization, not all the blocks will run concurrently. This level of parallelization depends on the hardware capabilities of the device, its number of SM and SP.

Inside the SM the block threads will be split into *warp*. A *warp* is a set of 32 threads. Each SP will be assigned witha *warp* to handle concurrently.

An also important thing to know is that each SM as a limited number of 32-bit registers and shared memory to store the data used by the threads. The threads will need to share all these resources. This can be a limitation to take in account to avoid the kernel launch to fail.

4.3 The ported function

The first CUDA version of the function simply use the general pattern for CUDA application presented before and uses both blocks and threads to computes in parallel the collisions. Thus each thread will be assigned with a triangle from the array and will compute the collision between this triangle and the reference triangle to finally return the result.

See chapter 6 for detailed performance analysis.

Chapter 5

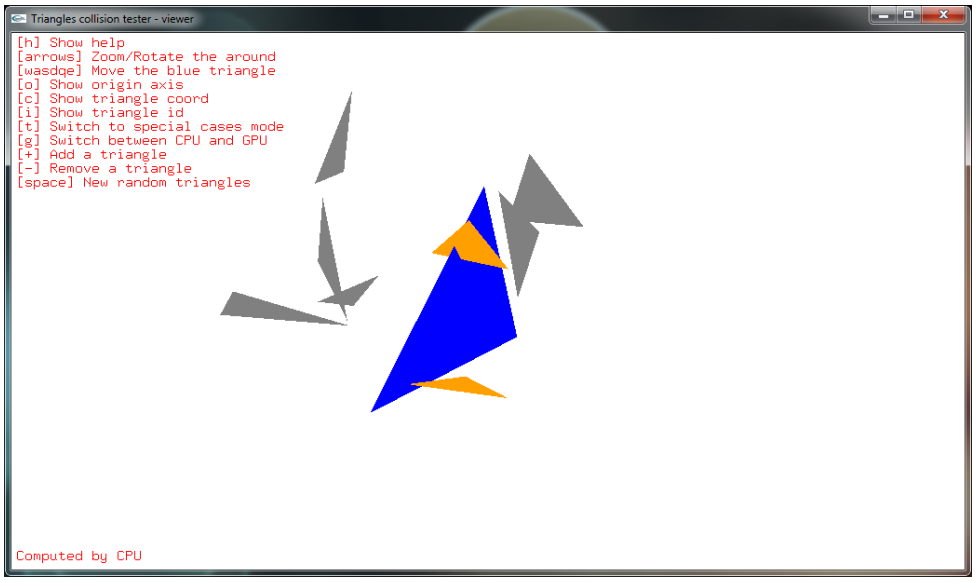
Optimize the function

5.1 A proper testing program

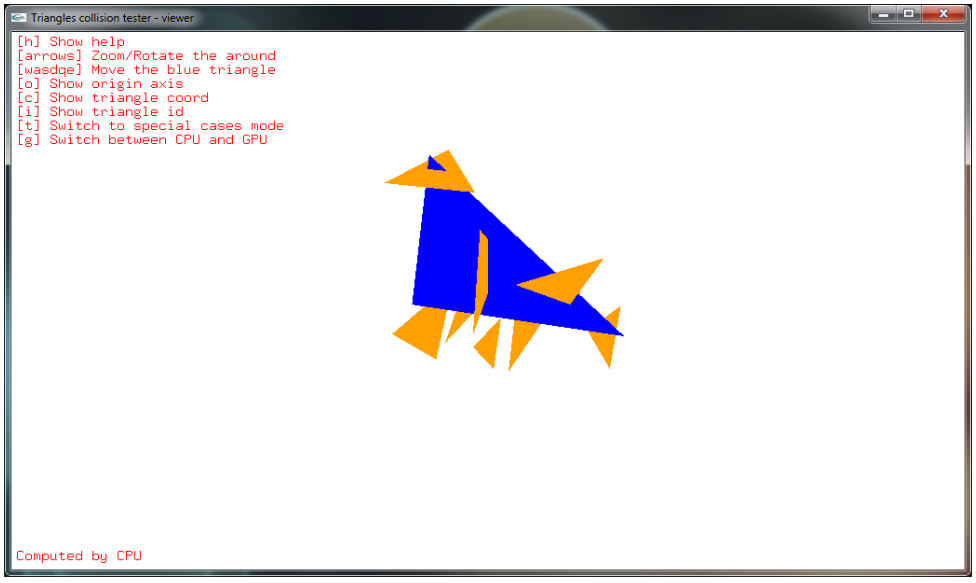
Before going any further I've decided to improve my way of testing the functions because it was becoming too slight. First, I've improved my OpenGL viewer to help me developing and debugging more easily. Here are the new features I've included :

- A help menu listing all the features and the corresponding shortcut
- the reference triangles can be moved around
- The origin axis can be displayed or hidden
- The triangle's coordinates can be displayed
- The triangle's id can be displayed
- A special cases mode as been includes to test directly all special cases configurations at once
- The user can switch between CPU and GPU for the collisions computation
- The number of triangles can be increased or decreased on the fly

Let's have a look at this new version :



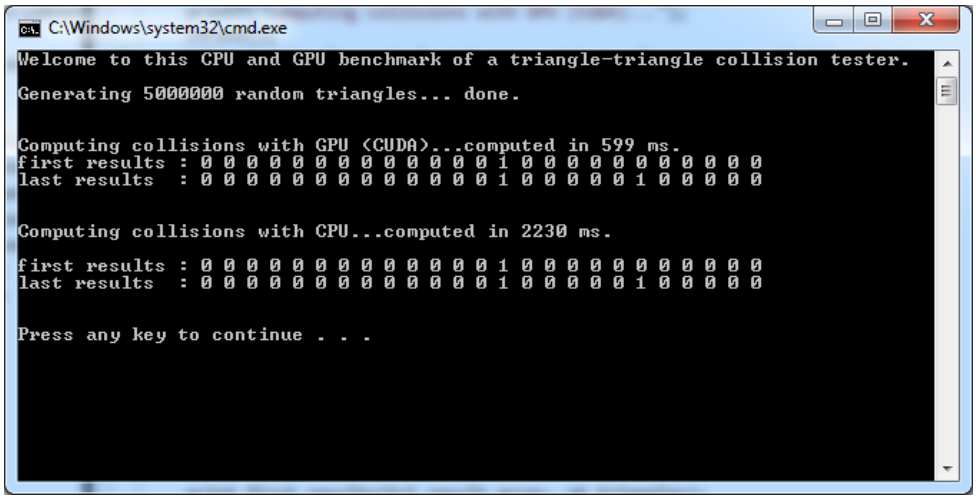
And here is the special cases[1] testing mode :



Then I've also developed a console benchmark to measure performances and being able to compare different optimizations. The benchmark is simple : it generates a number of random triangles, measures the time the GPU use to compute all the collisions with a reference triangle and then do the same for the CPU. Finally the results are printed.

Here is how the console benchmark looks like :

[1] Based on the considerations made in the section 3.3.2



In a sake of simplicity, the console benchmark and the OpenGL viewer has been merged in a single testing program and the user can chose by command line which program to launch and with how much triangles. More details on how to use the test program can be found in appendix C.

5.2 Optimize the common algorithm

You can optimize some code in multiple ways: memory footprint, executable size, speed, exception handling, etc. In my case the only optimization I'm interested in is *speed*.

To let me appreciate the improvement that one or another modification brings I need to compare the execution time after each of them. The benchmark is done by computing the collisions for 5M (millions) triangles on a *Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz* while measuring the execution time between the moment just before calling the function and the moment the program have just returned from the function. The time elapsed is measured using the library `<time.h>` and the `clock_t clock ( void );` function from it.

The results will be presented this way :

version	*M1*	*M2*	*M3*	*M4*	*M5*	*AVG*	*delta gain*	*total gain*

The *M1-5* are the measures taken, *AVG* is the average of the 5 measures, *delta gain* is the gain between the version just before and the one presented. Finally the *total gain* is the global gain compared to the reference version v1.0.

The command line used to launch the test program is :

```
1 > test-program.exe --benchmark -t 5000000
```

5.2.1 v1.0

This is the first working version of the algorithm. It's already quite efficient because it's based on good techniques thanks to the book [2] I used. However some things can still be optimized, the following sections will present you the optimizations done.

As this is the first version of the algorithm it will be used as reference performance index.

version	*M1*	*M2*	*M3*	*M4*	*M5*	*AVG*	*delta gain*	*total gain*
v1.0	7756 ms	7718 ms	7779 ms	7725 ms	7757 ms	**7747.0 ms**	-	-

5.2.2 v1.1

The different algorithm steps were slit into multiple functions. Here a new unique function is doing all the job in a more comprehensive way. The structure of the function is even more based on the steps given in the book [2, p.542].

version	*M1*	*M2*	*M3*	*M4*	*M5*	*AVG*	*delta gain*	*total gain*
v1.1	7701 ms	7694 ms	7653 ms	7732 ms	7671 ms	**7690.2 ms**	0.74%	**0.74%**

5.2.3 v1.2

The distance between a vertex of a triangle and the plane of the other triangle was computed more than one time in multiple functions while the value was always the same. Those signed distances are now computed only once in the main algorithm function.

version	*M1*	*M2*	*M3*	*M4*	*M5*	*AVG*	*delta gain*	*total gain*
v1.2	7298 ms	7271 ms	7246 ms	7278 ms	7249 ms	**7268.4 ms**	5.80%	**6.58%**

5.2.4 v1.3

The side of each vertex of a triangle relative to the plane of the other triangle (found by checking the sign of the distance) is used in multiple places in the algorithm, in the quick reject test and then while ordering the vertex. Sadly it was also computed in both places, now it is computed only once in the main algorithm function.

version	*M1*	*M2*	*M3*	*M4*	*M5*	*AVG*	*delta gain*	*total gain*
v1.3	7084 ms	7203 ms	7131 ms	7101 ms	7103 ms	**7124.4 ms**	2.02%	**8.74%**

5.2.5 v1.4

The planes of both triangles are needed in multiples places in the algorithm. However they were also computed more than once, so they are now computed only once in the main algorithm function and passed to the functions that need them.

version	*M1*	*M2*	*M3*	*M4*	*M5*	*AVG*	*delta gain*	*total gain*
v1.4	6712 ms	6847 ms	6733 ms	6720 ms	6729 ms	**6748.2 ms**	5.57%	**14.80%**

5.3 Optimize the CUDA part

To let me appreciate the improvement that one or another modification brings I need to compare the execution time after each of them. The benchmark is done by computing the collisions for 20M (millions) triangles while measuring the execution time between the moment just before calling the function and

the moment the program have just returned from the function. The time elapsed is measured using the library `<time.h>` and the `clock_t clock ( void );` function from it.

For the CUDA optimizations the machine used to test and compare the optimizations differ from the previous section :

- Intel i7-2600K @ 3.4GHz (4 cores)
- 8.00 RAM
- $2x$ 590 GTX (bi-GPU)

On this machine, with the first working CUDA version based on the v1.4 of the common algorithm and without any optimizations on the CUDA side the performances are already quite impressive[2] :

	M	*gain*
CPU	8959 ms	-
GPU	2462 ms	**264%**

See chapter 6 for detailed and complete performance analysis.

5.3.1 Split kernel launches

This is not really a performance optimization but it's a trick I'm using to avoid a *Windows* security system. This security system is part of the *Windows Display Driver Model* and is called *Timeout Detection and Recovery*. It's a protection to avoid frozen display in case of a GPU problem. It triggers if the OS have no response of the GPU driver for 2 seconds. This is a good functionality for normal use, but because a kernel occupies the device at 100% the TDR triggers and resets the GPU driver and therefore kills the running kernels to recover the graphic card if it's running for more than 2 seconds. It's quiet annoying because this limits a lot the interest of GPU computing.

The best solution is to deactivate this timeout when computing kernel with the GPU. To see how to do this see appendix A.4.

However because not every one can maybe do it (no administrator rights, etc.) I've decided to find a seamless solution using code. The idea is to split the unique kernel call that takes more than 2 seconds into multiples sequential kernel calls that take less than 2 seconds each. But because kernel calls are asynchronous, a `cudaDeviceSynchronize();` must be placed after each kernel call to wait the end of the kernel execution and therefore to ensure that the GPU driver as a little time between calls to response again to the OS to avoid the protection to trigger.

Because the time a kernel takes to execute depends on the speed of the GPU, a kernel may be split into too many calls if I don't adapt it to the speed of the device. To do that I've based myself on the number of CUDA cores of the device (which can be found in the devices properties, see section 4.2.2 for more details) to theoretically evaluate the capabilities of the device. This allows me to always split the kernel call into smaller ones that takes always just under 2 seconds each.

[2] Especially as the CPU of this machine is already very quick

Using split kernels will also helps us to avoid the maximum blocks limitation of 65535 which can be reached. Indeed with 128 threads per blocks for example we can compute $128 * 65535 = 8.4$M triangles only.

The performance are the same using split kernel calls or unique kernel call which is a good thing. This is due to the fact that, in reality on the hardware, the blocks (if a lot) will not all execute concurrently on the device but only some. It depends on the hardware capabilities of the device. Thus the other ones are queued waiting for being treated. So, when using split kernel call, we will simply do as the hardware do, but using code.

5.3.2 Maximize hardware utilization

Unlike the title may indicates, this is not really an optimization. It's more a wish of adapting the kernel launch to run as well as possible no matter which device it is on. For that, the hardware specifications of the device must be taken into account.

In the section 4.2.6 covering the hardware architecture a part is explaining the fact that each thread have a limited number of registers to store his data. Because each thread will fully execute the kernel code, each variable declaration in it will use some registers. For example declaring a 32-bit integer somewhere in the code will use one of the 32-bit register, and so on. As the threads are running concurrently they will need some registers at the same time.

To be more precise each stream processor, unit that will handle a block of threads, have a number of registers available to share between all the threads running. Therefore more the kernel uses registers less the number of threads launched per block could be high. Precisely in my case this limitation causes my kernel launch to fail on some devices. Let's see why :

I launch a kernel with 1024 threads per block on a NVS 4200 which is *Compute Capability* 2.1 to compute the triangle-triangle collisions. The compiler option : `--ptxas-options=-v` allows me to find the number of registers used by my kernel. In my case the algorithm uses **48 registers**. So each thread will need 48 registers to run correctly. The NVS 4200 has, according to it *Compute Capability* 2.1, 32K of registers per stream processor. So the 32K (32768) of registers will be shared between the threads.

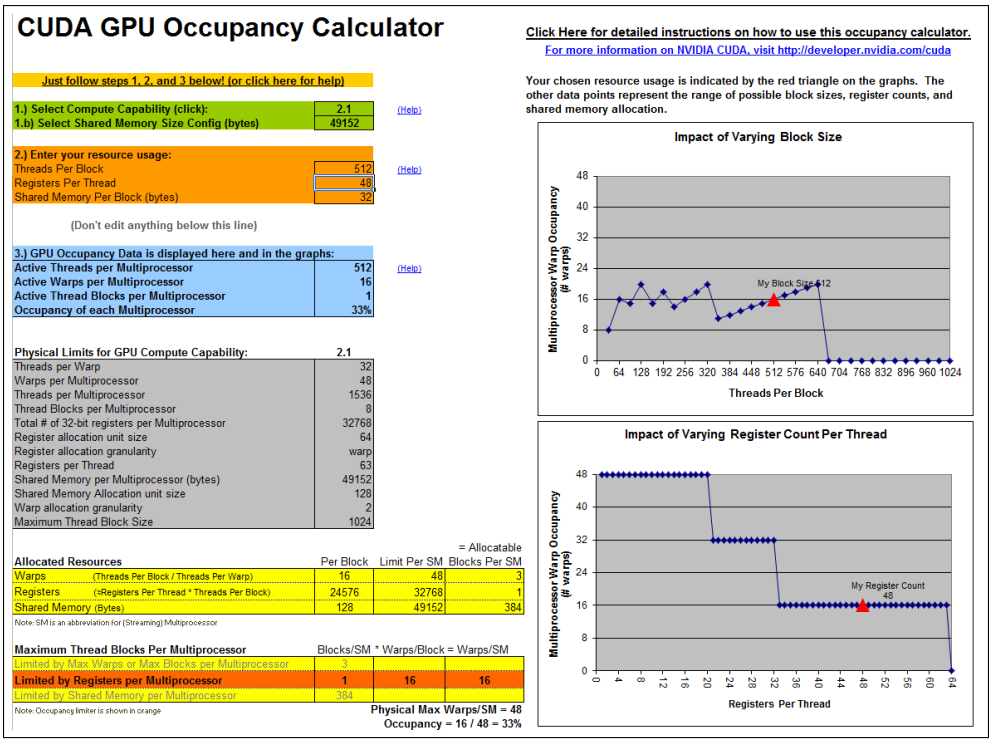
$$(32768/1024 = 32) < 48?$$

The answer is clearly no. I don't have enough registers available per thread. I have to either lower the number of registers or to lower the number of threads launched. Because I can't use less registers that I'm already doing, I will need to lower the number of threads per block. In my case the maximum number of threads per block is :

$$32768/48 = 682$$

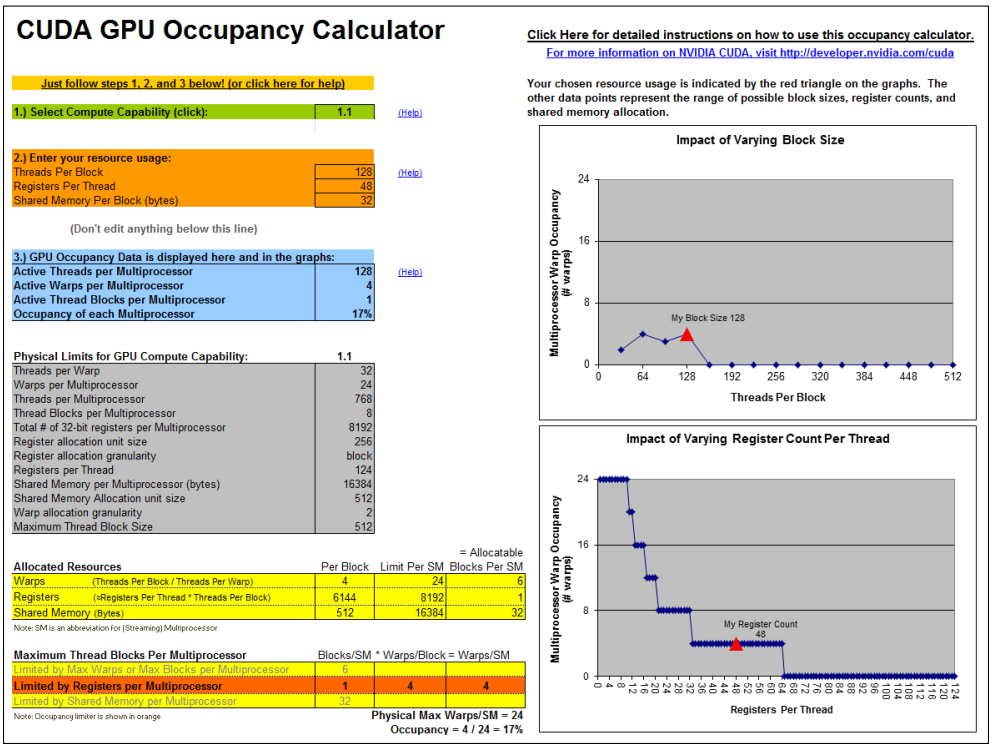
Therefore if I launch blocks of 682 threads each, my kernel will now work. However this configuration may not be optimal depending on the device, so let's go a bit further.

To allows developer to maximize the performances of their CUDA applications, NVIDIA have provided a tool to find the best launch configuration for each particular kernel and depending on the hardware. This tool is called *Occupancy Calculator* and it can be found in the CUDA toolkit under `tools/` or on their website. Here is how it looks like :



The most interesting plot of this tool is the one called *Impact of Varying Block Size* which allows to see, for our specific kernel usage values, the best threads per block launch configuration. On the x-axis the number of threads per block is represented while on the y-axis the multiprocessor occupancy is shown. The higher the occupancy is, better it is. The screenshot above shows the plot for my kernel with my NVS 4200. As you can see the occupancy at the limit of 682 threads calculated before is good. So I could launch the kernel with this much threads.

However for an older CUDA-capable device, a 8800GT, the conclusion is not the same :



The device have 8192 registers per multiprocessor so the maximum number of threads per block is :

$$8192/48 = 170$$

If I want to launch the kernel with this value as I've done before the occupancy would be very low this time!

The solution I've chosen after having tested a variety of configuration is to use the power of two just smaller than the limit calculated. So for the NVS 4200 it would be 512 threads per block and for the 8800GT it would be 128 threads per block. I found this solution very good because it optimizes well the GPU occupancy for any kind of devices.

5.3.3 Use of host pinned memory - not applicable

When dynamically allocating data on the host, the use of `malloc()` is quite standard. When developing CUDA applications it can be interesting to use CUDA version of the `malloc()`, `cudaHostAlloc()`. Multiple types of memory can be allocated using this function but the default behaviour is to allocate pinned memory, memory that cannot be swapped to the hard-drive. Being sure that the memory allocated this way will never be swapped out, the CUDA driver tracks the memory addresses used and therefore the device can directly access this memory to read and write faster than it would with standard `malloc()` memory.

The use of `cudaHostAlloc()` is necessary to use streams (see next section) and also for the zero-copy feature that allows to avoid the data copies from host to device and backward. The device directly access to the host memory. It's more efficient on devices which share the host memory with the CPU like integrated graphic cards.

However in my case all of theses features can't be used because I don't know how the host memory has been allocated when my function is called. It is certainly static memory allocation, or malloc allocation but certainly not a memory space allocated with `cudaHostAlloc()` as I need it.

5.3.4 Staged concurrent copy and execute with streams - not applicable

This could be a very efficient way to reduce the global execution time of an algorithm running on the GPU. The book I read on CUDA reported in his example a 21% improvement over the original version.[4, p.210]

The principle is to create and launch on the device multiple streams of operations like memcpy, kernel execution, etc. that will execute serially within a stream but simultaneous between the streams. Thus every operation (even memcpy) are launched asynchronously at the beginning and then the device will try, if it is capable, to run a memcpy of a stream at the same time as a kernel execution in another stream.

So one understand that it's a useful feature if the memcpy takes a considerable time to execute relative to the kernel execution. Sadly this is not my case, indeed I've run my program with different number of triangles 0.1M, 1M, 10M, etc. and all memcpy operations take always less than 1% time of the kernel execution time.

Therefore I've decided not to implement streams in my function as it would be a waste of time compared to the improvement I may have.

5.3.5 Multi-GPU

CUDA allows the developer to use more than one graphic card if available. Prior to the version 4 of CUDA this was a bit restrictive because a host CPU thread was required for each graphic device. I agree, first this seams a good thing because each device management is well separated and the management code is cleaner. However this means more complex code because of the threads creation and also a heavier code to execute, it's quiet long to create a host thread. From the version 4.0 and ahead a single host thread can handle more than one graphic device. This is what I've chosen in my case because I prefer fast code as clean code.

The general pattern to manage multiple devices is the following :

```
1 int device_count;
2 cudaGetDeviceCount(&device_count);
3 //...
4 int i;
5 for(i=0;i<device_count;i++){
6     cudaSetDevice(i);
7     //Specific operations on selected device (memcopy, kernel launch, etc.)
8 }
```

However doing it exactly like this will not takes advantage of the multiple devices. Indeed let's assume the operations are the following (in that order) :

1. memory allocation on the device
2. copy data from host to device
3. launch the kernel
4. copy result back from device to host
5. free memory

This leads us to the following code :

```
1 int device_count;
2 cudaGetDeviceCount(&device_count);
3 //...
4 int i;
5 for(i=0;i<device_count;i++){
6     cudaSetDevice(i);
7
8     cudaMalloc();
9     cudaMemcpy(cudaMemcpyHostToDevice);
10    kernel<<<blocks,threads>>>();
11    cudaDeviceSynchronize();
12    cudaMemcpy(cudaMemcpyDeviceToHost);
13 }
```

So as you can see in the code snippet above the for loop will run through each CUDA-capable device thanks to the call to `cudaSetDevice()` at the beginning of each iteration. This call will define the active

device for the following `cudaXXX()` calls. Therefore the first iteration of this loop will execute the 5 operations enumerated before for the device 0, then again for the device 1, and so on.

But if we have 4 CUDA-capable devices and we want to take advantage of the multi-GPU capabilities we need to have the 4 kernels running *in parallel*. The solution with only one host thread that execute the code sequentially is to do only asynchronous calls. And here is the problem, in the five operations enumerated before only the kernel launch is asynchronous. However, because of other synchronous calls (memory allocation, copy, etc.), the kernel launches will be delayed one from each other.

To avoid this, the solution I chosen is to use the following pattern :

```
1  int device_count;
2  cudaGetDeviceCount(&device_count);
3  //...
4  int i;
5  for(i=0;i<device_count;i++){
6      cudaSetDevice(i);
7
8      cudaMalloc();
9      cudaMemcpy(cudaMemcpyHostToDevice);
10 }
11
12 for(i=0;i<device_count;i++){
13     cudaSetDevice(i);
14
15     kernel<<<blocks,threads>>>();
16 }
17
18 for(i=0;i<device_count;i++){
19     cudaSetDevice(i);
20
21     cudaDeviceSynchronize();
22     cudaMemcpy(cudaMemcpyDeviceToHost);
23 }
```

As the kernel execution is clearly the most time consuming operation in my case, this solution is almost optimal. Indeed the synchronous memory allocation and data copy are quickly executed on each devices. Then the second for loop allows to asynchronously launch all the kernels, they will therefore execute in parallel. Finally the last for loop will synchronize each device with the host and then copy the result back.

	M	*gain from single GPU*	*gain from single CPU*
GPU (4x)	1163 ms	111%	**670%**

See chapter 6 for detailed and complete performance analysis.

5.3.6 Debug to Release

When time came for me to prepare my code for release I had a very good surprise : the equivalent execution times were a lot lower then I expected In single GPU computation I have a stratospheric gain of 289% so quite 4 times faster! Here is an outline[3] :

[3]Strangely in Release the multi-GPU version become useless as the single-GPU version is even quicker for all the values tested. It's however strongly dependant on the hardware and maybe on an other configuration the multi-GPU version could be interesting.

	M	*gain from GPU (1x)*	*gain fromGPU (4x)*	*gain from single CPU*
GPU (1x) release	633 ms	289%	84%	**1315%**
GPU (4x) release	701 ms	251%	66%	**1178%**

I have been surprised to see so much speed differences between *Debug* and *Release* compiled version so I've tried to inform myself to find what was making the *Debug* version so slow.

Honestly it was hard to find a lot of informations but I was able to highlight two possible reasons :

- Some stack checks can be performed in Debug mode which can slow down the execution time a lot
- The code execution on the GPU in Debug mode is in some way synchronized with the host to simplify the debugging so it became slower

Chapter 6

Performance analysis

6.1 Algorithm optimizations

This is performance outline of the optimizations done to the common algrotihm in the section 5.2.

> The benchmark is done by computing the collisions for 5M (millions) triangles on a *Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz* while measuring the execution time between the moment just before calling the function and the moment the program have just returned from the function. The time elapsed is measured using the library `<time.h>` and the `clock_t clock ( void );` function from it.

version	*M1*	*M2*	*M3*	*M4*	*M5*	*AVG*	*delta gain*	*total gain*
v1.0	7756 ms	7718 ms	7779 ms	7725 ms	7757 ms	**7747.0 ms**	-	-
v1.1	7701 ms	7694 ms	7653 ms	7732 ms	7671 ms	**7690.2 ms**	0.74%	**0.74%**
v1.2	7298 ms	7271 ms	7246 ms	7278 ms	7249 ms	**7268.4 ms**	5.80%	**6.58%**
v1.3	7084 ms	7203 ms	7131 ms	7101 ms	7103 ms	**7124.4 ms**	2.02%	**8.74%**
v1.4	6712 ms	6847 ms	6733 ms	6720 ms	6729 ms	**6748.2 ms**	5.57%	**14.80%**

6.2 CUDA optimizations

This is performance outline of the CUDA optimizations done in the section 5.3.

> To let me appreciate the improvement that one or another modification brings I need to compare the execution time after each of them. The benchmark is done by computing the collisions for 20M (millions) triangles while measuring the execution time between the moment just before calling the function and the moment the program have just returned from the function. The time elapsed is measured using the library `<time.h>` and the `clock_t clock ( void );` function from it.

For the CUDA optimizations the machine used to test and compare the optimizations was this :

- Intel i7-2600K @ 3.4GHz (4 cores)
- 8.00 RAM
- 2x GTX 590 (bi-GPU)

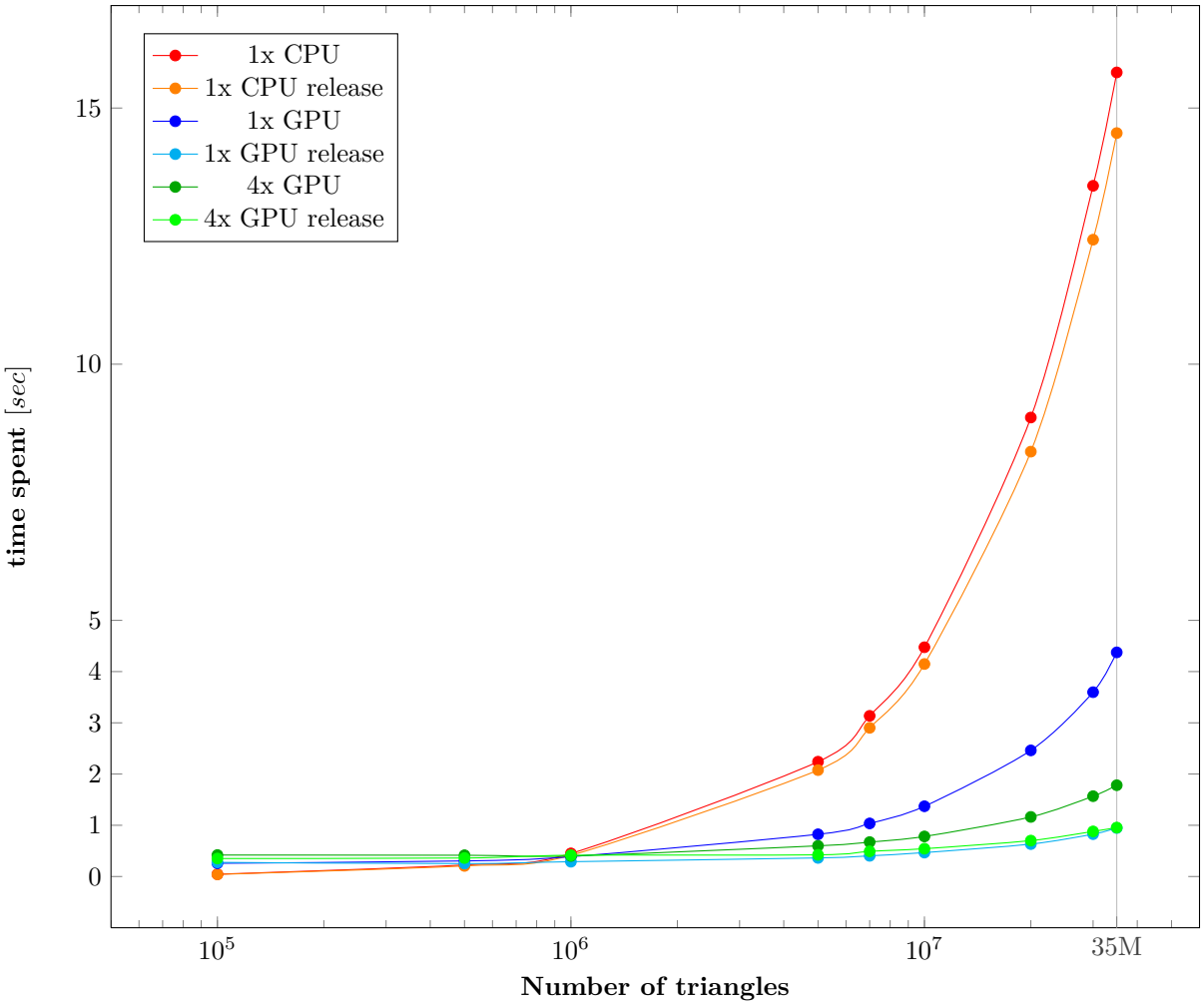
	M	*diff gain from previous*	*total gain*
1x CPU	8959 ms	-	-
1x GPU	2462 ms	264%	**264%**
4x GPU	1163 ms	111%	**670%**
1x GPU release	633 ms	84%	**1315%**
4x GPU release	701 ms	-10%	**1178%**

As mentioned in the section 5.3.6 the multi-GPU in release version is a bit slower than the single-GPU release. I can't understand well why but I imagine that, while single-GPU performances have increased so much, the kernel computation become not enough heavy so the extra execution time due to the multi-device management become too much weighty compared to the multi-GPU benefits. So the total gain from single-CPU to single-GPU for 20M triangles computation is a whopping 1315% of speed gain, therefore is more then 14 times quicker! The gap even increases with more triangles, the maximal gain reaches 1432% (so more than 15 times quicker) when computing 35M between release version of the single-CPU and the single-GPU.

6.3 Overview on a high-end machine

The measures have been taken on the following machine :

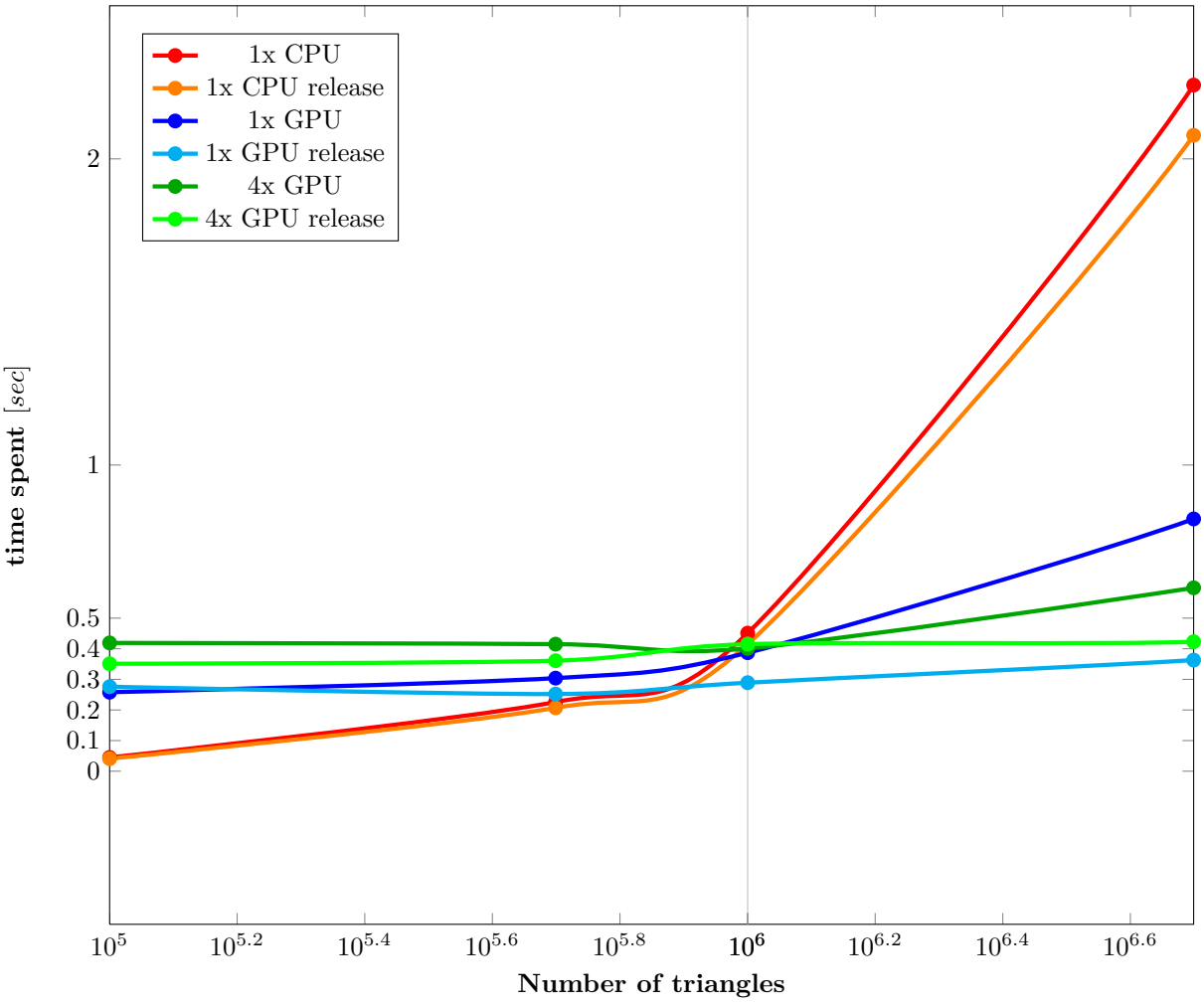
- Intel i7-2600K @ 3.4GHz (4 cores)
- 8.00 RAM
- 2x GTX 590 (bi-GPU)



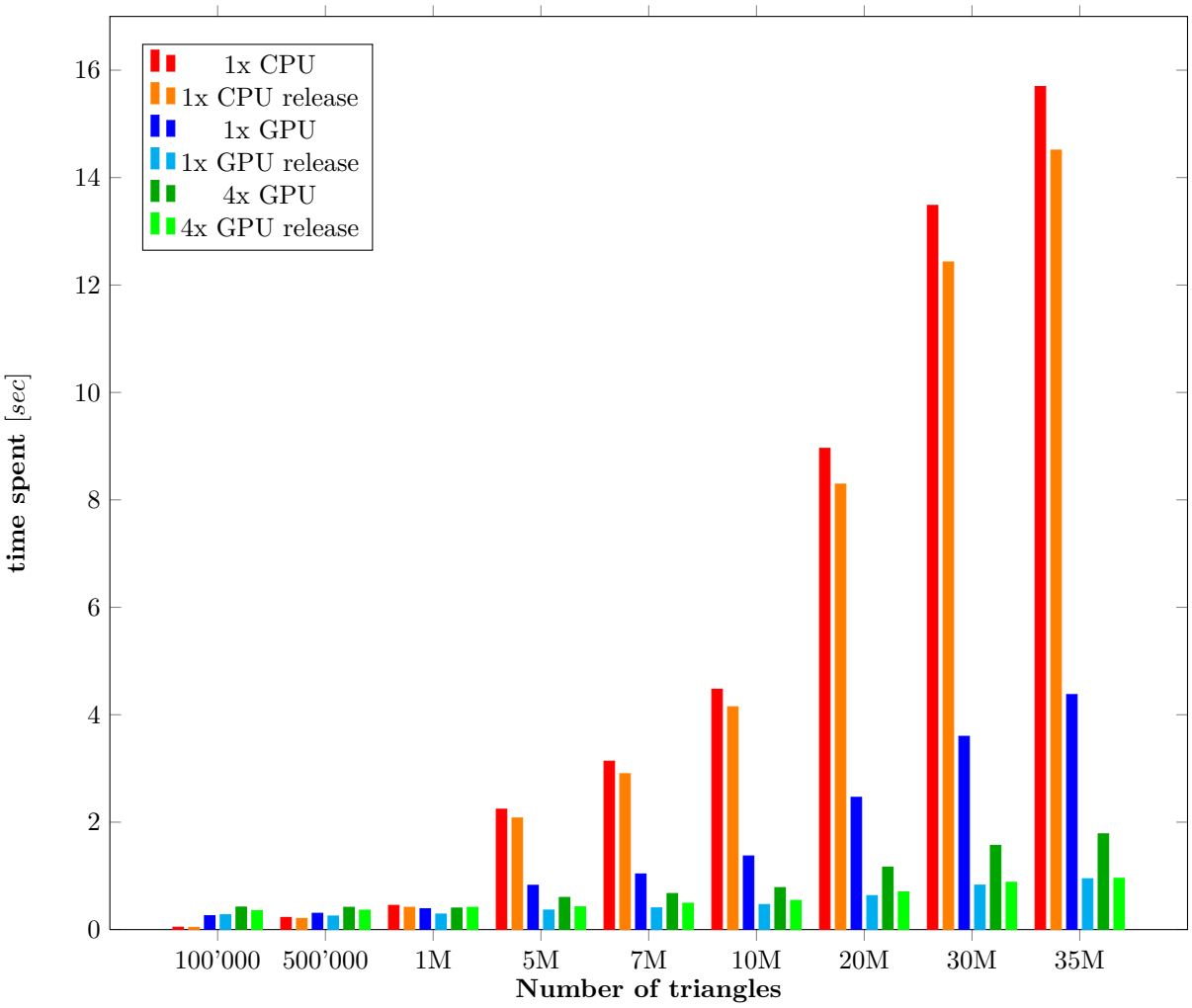
This semilogx plot let's well appreciate the astonishing performance of GPU computation. Until 1M triangles the performances are quiet the same (even a little disadvantage for the GPUs) but from there, while the time spent by the CPU is flighting away, all GPU versions keep a very acceptable time to compute the quickly increasing number of triangles. Notice the great speed improvement of the Release version of the code. The plot also clearly indicates that for bigger numbers then 35M the gap between the CPU and the GPU would still grow. This is easily explained by the fact that the CPU evolution will be linear because it is computing the collision sequentially while an added triangle collision to compute for the GPU will simply be executed in parallel with the other ones so no extra time is spent. Practically, even if very high, the number of parallel running threads on the GPU is though limited.[1]

It may be interesting to zoom on the left of the previous plot just to see how it is for low triangle count. As you can see for 10'000 triangles the results are clearly disadvantaging the GPU version : 45 ms for the CPU, 258 ms for the single-GPU version and 419 ms for the multi-GPU version. This clearly indicates that, although the GPU versions are a lot quicker for heavier tasks, it's very time consuming to prepare the computation on the GPU (memory allocation, data copy, etc.) and so the CPU is quicker if we want to compute a few collisions (under 1M).

[1] see section 4.2.6 for more details on the hardware architecture.



This last chart let's appreciate the results under a different light :



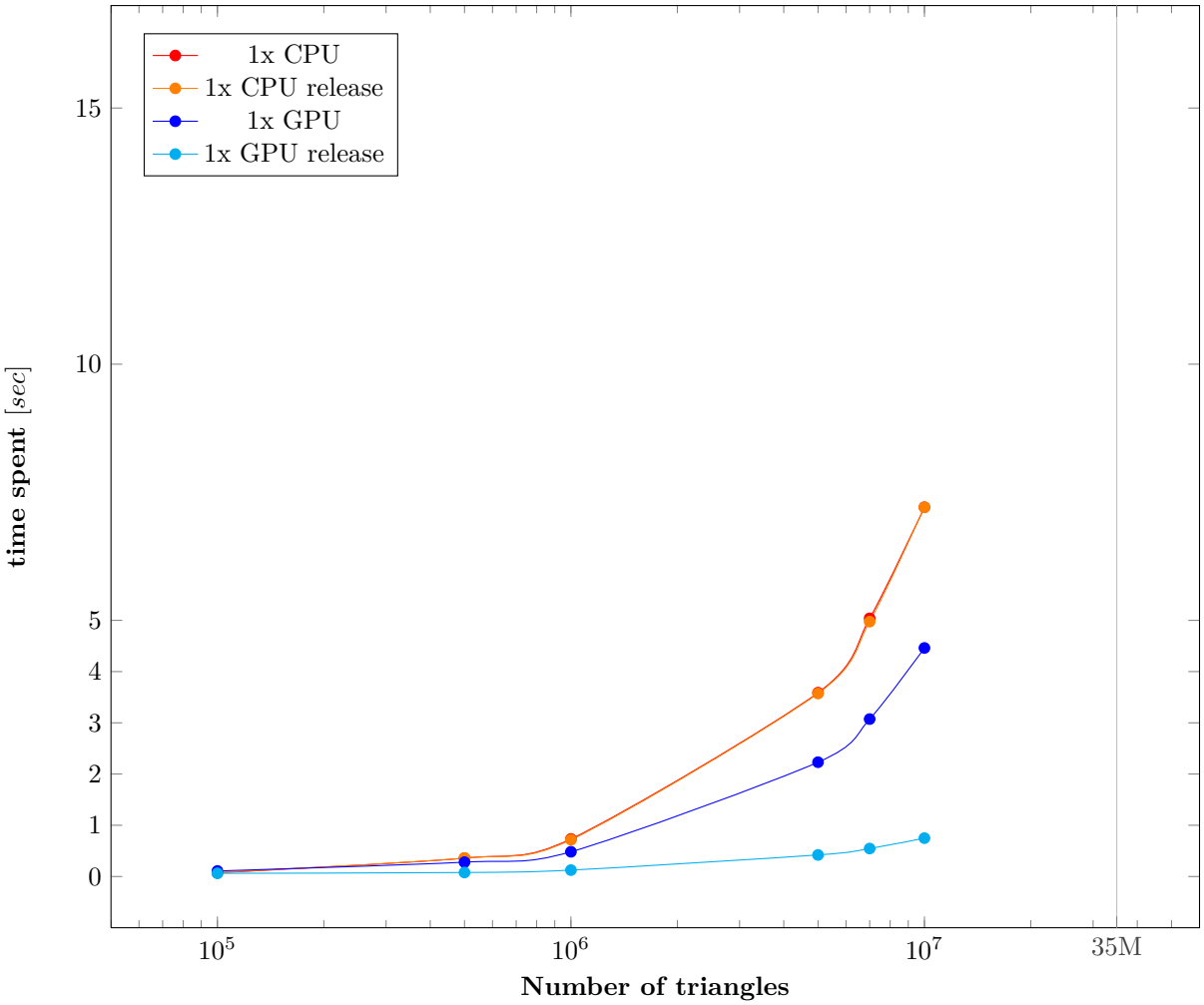
6.4 Overview on a 5 years old machine

Now let's see how it looks on a 5 years old machine with one of the first CUDA-capable device.

The measures have been taken on the following machine :

- Intel Core2Duo E6850 @ 3.00GHz
- 4.00 RAM
- GeForce 8800 GT

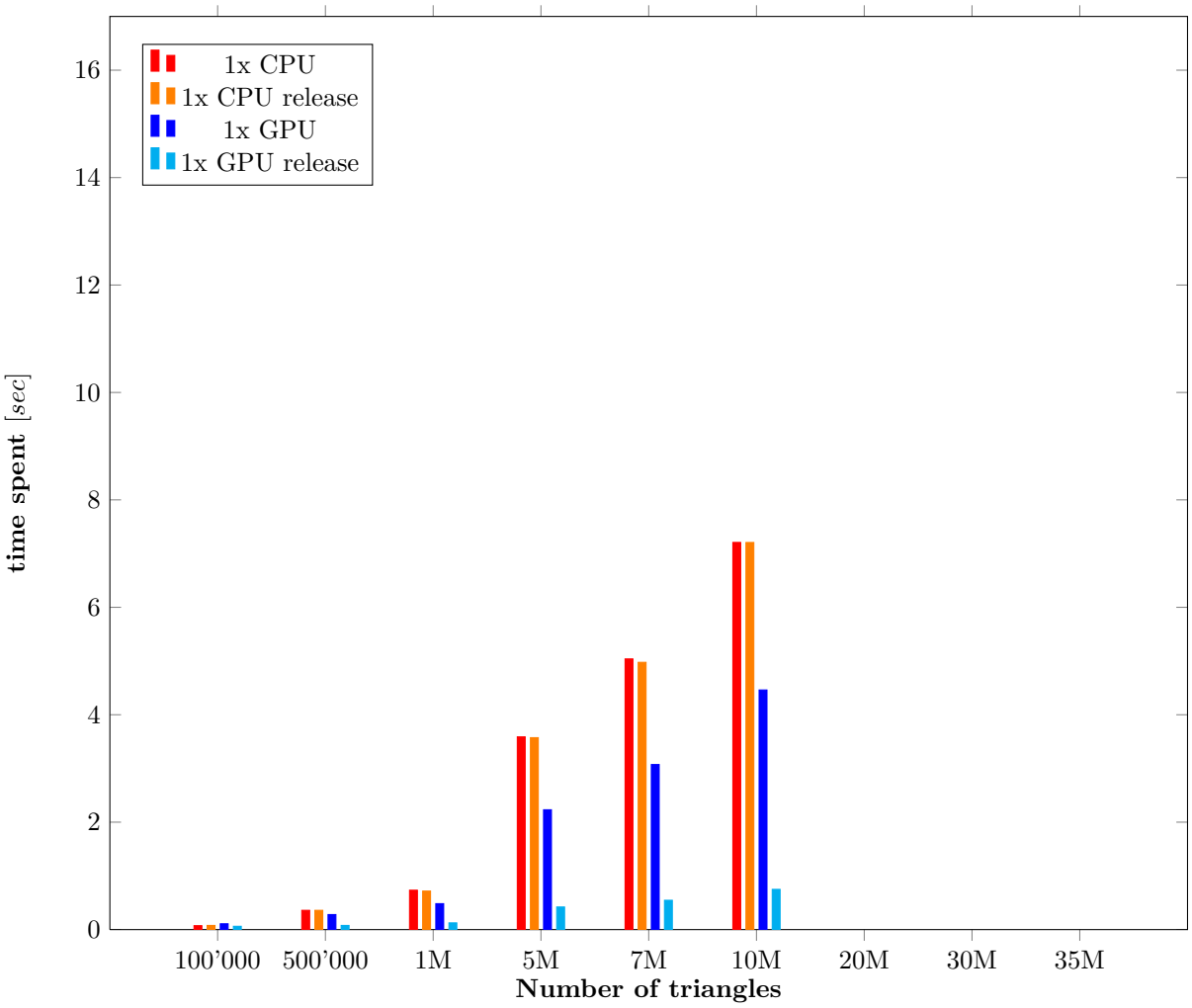
The following plot and chart are voluntary plotted with the same view and ranges to help the comparison with the previous ones.



The results are here logically under the high-end computer speed but still the GPU gives very good results especially in the release version. The gap between the Debug and the Release version is here even bigger then in the previous section. In all cases this means that the function may take advantage of the GPU over the CPU (if same age) no matter which CUDA-capable device it is.

Note that the plot stop at 10M triangles only because this GPU as too few memory to handle more than that.

Like in the previous section here is the corresponding chart :



The maximal gain here in release of the GPU version is 860% (so quite 10 times quicker). Note that compared to the high-end machine with his huge GTX 590, this 8800GT isn't doing much slower in release version. For 10M triangles the GTX 590 is only 60% quicker so this quite old card is still doing well.

Chapter 7

Conclusion

7.1 Problems encountered

7.1.1 CUDA tools have no linker

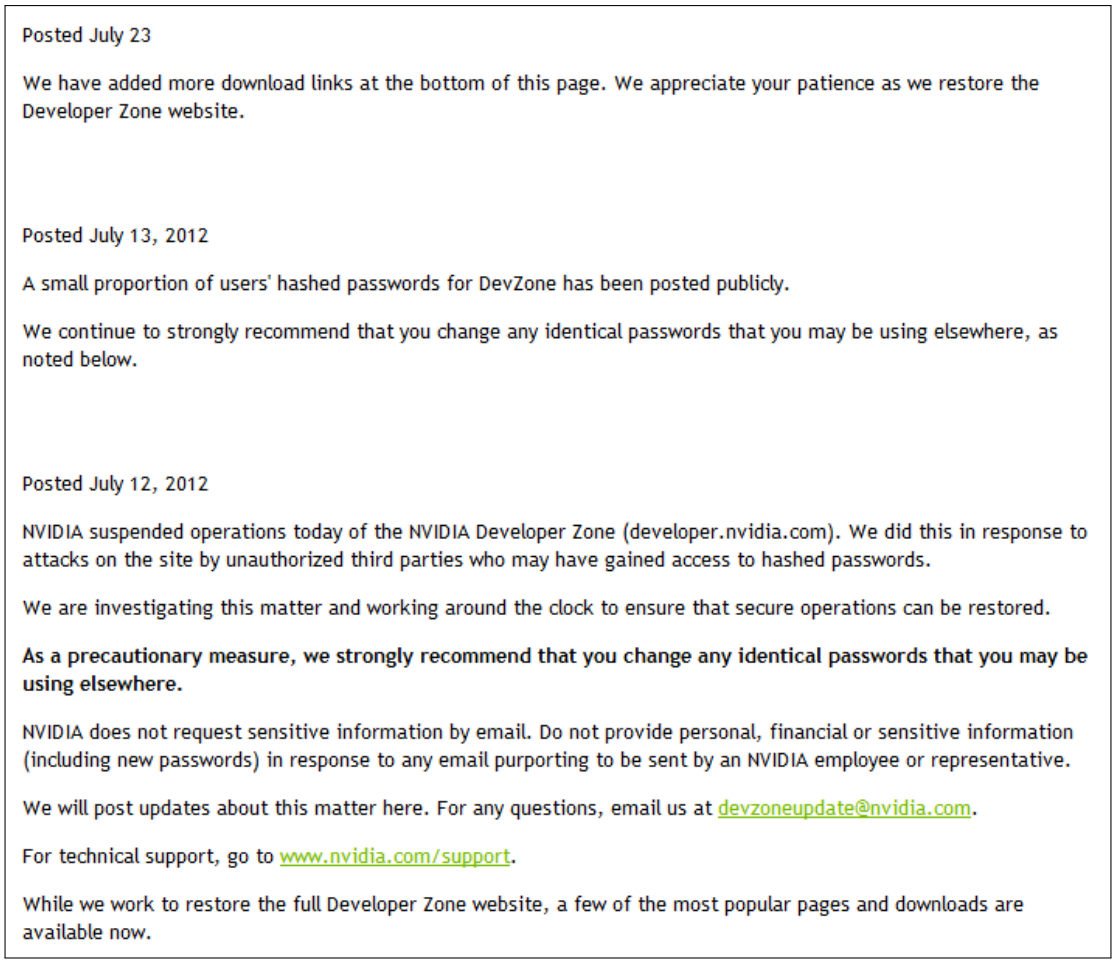
A known limitation of the current stable CUDA toolkit version 4.2 is the lack of linker, this leads to the fact that it's not possible to split the functions used by the kernel into multiples files. All the functions must be in the same file and qualified with `__device__`[1] for CUDA compiler agrees to use them in the kernel. I was annoyed by this because I wanted to use the same geometric functions I've developed as well as the same intersection test algorithm for both CPU and GPU functions.

The workaround I've found is done in two steps : First to qualify the functions with both `__host__` and `__device__` to indicates the CUDA compiler that theses functions are designed to run on the host as well as on the device. Secondly to include the source file *.cu containing all the geometric functions directly into the file that contains the kernel, this way the CUDA compiler have all the needed function at the same place. This way the host function can also use the tools and they still are in a separate file for better comprehension. Note that the file included must not be compiled because he already is through the inclusion.

7.1.2 NVIDIA Developer Zone down

Another inconvenience is that from about July the 4 or 5 all the NVIDIA Developer Zone as been closed due to attacks on the site. Here is the current message displayed when trying to access the Developer Zone :

[1] This is a protection to prevent the possibility of calling any standard host library function on the device.



Posted July 23

We have added more download links at the bottom of this page. We appreciate your patience as we restore the Developer Zone website.

Posted July 13, 2012

A small proportion of users' hashed passwords for DevZone has been posted publicly.

We continue to strongly recommend that you change any identical passwords that you may be using elsewhere, as noted below.

Posted July 12, 2012

NVIDIA suspended operations today of the NVIDIA Developer Zone (developer.nvidia.com). We did this in response to attacks on the site by unauthorized third parties who may have gained access to hashed passwords.

We are investigating this matter and working around the clock to ensure that secure operations can be restored.

As a precautionary measure, we strongly recommend that you change any identical passwords that you may be using elsewhere.

NVIDIA does not request sensitive information by email. Do not provide personal, financial or sensitive information (including new passwords) in response to any email purporting to be sent by an NVIDIA employee or representative.

We will post updates about this matter here. For any questions, email us at devzoneupdate@nvidia.com.

For technical support, go to www.nvidia.com/support.

While we work to restore the full Developer Zone website, a few of the most popular pages and downloads are available now.

This problem as been very painful for me because all forum topics are inaccessible and this was my main information source when I was encountering problems. Therefore some times I've passed more time to solve a problem just because the Developer Zone was inaccessible. The site was still down on July the 24.

7.1.3 Float rounding differences between CPU and GPU

I've encountered a quiet strange problem with floating point numbers that I was not able to solve. Indeed in my console benchmark program I've tried to compared the results computed by both CPU and GPU for the same triangle and this for to whole array to check if the results were always the same as it should. Unfortunately, I discovered that it was not always the case. Indeed for a bunch of 1000 triangles I have in average one mismatching result. By displaying each problematic triangle found I've discovered that it always concern triangles who are close to touching or closely touch the reference triangle. It's clearly indicates a rounding problem. This may be caused by differences between floating precision of the CPU and the GPU. With this idea in mind I checked that both would be working on single-precision floats but despite the effort the problem remains. However good CUDA is IEEE 754 compliant for floating operations.

Trying to find more informations about it I found a document written by NVIDIA about this subject. [3] Here is extract of it:

> When porting numeric code from the CPU to the GPU of course it makes sense to use the

> x86 CPU results as a reference. But differences between the CPU result and GPU result must be interpreted carefully. Differences are not automatically evidence that the result computed by the GPU is wrong or that there is a problem on the GPU.

More in details they explains also the following thing :

> As we have explained, there are many reasons why the same sequence of operations may not be performed on the CPU and GPU. The GPU has fused multiply-add while the CPU does not. Parallelizing algorithms may rearrange operations, yielding different numeric results.

I conclude that it may not be a solvable problem and above all that it may not be so dramatic. Indeed the problematic triangles are the ones who are already at the limit, so finally it doesn't matter if they are considered as intersecting or not because both results may be right as they are so closed.

7.2 Possible improvements

7.2.1 Multi-core CPU version

The CPU version of the function is now only a single-core version. So when comparing the result with a multi-GPU version of the CUDA function this is not fully "comparable" even if it's still informative. Idealy a multi-core CPU version could have been interesting.

7.2.2 WDDM TDR workaround also available for multi-GPU

For now the workaround that allows to avoid the WDDM TDR protection to trigger is only available in single-GPU version (see section 5.3.1). This is because after each launch of a split kernel a `cudaDeviceSynchronize()` is placed. Because currently a single host is launching all the kernels on the devices, the calls needs to be asynchronous to take advantage of the multiple devices. This would not be the case if a `cudaDeviceSynchronize()` is placed between each launch.

Therefore the only solution to avoid this problem is to have one host thread per device to manage. Thus the split kernels workaround could be used even in multi-GPU execution of the CUDA function.

7.2.3 OpenCL

It has been discussed with *Pixelux Entertainement* to try to port also the algorithm to another technology then CUDA if I had enough time. Sadly it was not the case but I though inform myself a bit on OpenCL. OpenCL is a open standard to develop application that take advantage of parallel hardware. It like CUDA do, but not only with NVIDIA GPU devices, but also with AMD/ATI hardware and so on.

For what I've seen the performances between the same CUDA code running in OpenCL are quite the same. Moreover porting CUDA code to OpenCL code is not so much work because OpenCL is itself based on CUDA. Tools even exist (for what I've seen only for linux at the moment) to port it automatically. I've however not tested it.

Thus this could be an interesting improvement of this function.

7.3 Last words

When I launched myself in this thesis I had no idea if the results would be interesting and even if the whole function could works on the GPU, therefore I'm very satisfied with the results! The algorithm is robust and reliable and the CUDA function achieves great performances compared to the CPU version. The best performance ratio achieved is more then **15 times quicker** for the GPU between a *i7@3.4GHz* and a *GTX 590* with release code. An also good aspect is that the performances are still very interesting even on an older machine : a ratio of 10 times quicker for the GPU is achieved between a *Core2Duo E6850@3.00GHz* and a *GeForce 8800GT* with release code. Finally this project has been very informative and interesting to realize. One of it great aspect is that it's so simply defined while involving many different aspects and being very complete and expansible.

Bibliography

[1] Pixelux Entertainement. About. http://www.pixelux.com/company.html, 2012.

[2] Philip J.Schneider and David H.Eberly. *Geometric Tools for Computer Graphics*. Morgan Kaufmann Publishers, 2003.

[3] NVIDIA. Floating point and ieee 754 compliance for nvidia gpus. http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf.

[4] Jason Sanders and Edward Kandrot. *CUDA By example*. Edwards Brothers, 2011.

[5] Wikipedia. Cuda — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=CUDA&oldid=500305004, 2012. [Online; accessed 21-July-2012].

[6] Wikipedia. Gpgpu — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=GPGPU&oldid=502022507, 2012. [Online; accessed 24-July-2012].

Appendix A

Installing CUDA

A.1 Do you have a CUDA-capable device?

NVIDIA graphic cards built since 2006 and the GeForce 8800GTX are CUDA-capable, for the full list see the Wikipedia page [5].

A.2 To use CUDA application

If the NVIDIA graphic card is CUDA-capable, then it's sufficient to install the lastest available official driver from NVIDIA website, the CUDA driver is included in it. However just in case it can be downloaded separately at `http://www.nvidia.com/content/cuda/cuda-downloads.html`.

A.3 To develop CUDA applications

A.3.1 Minimum

By going to this url : `http://www.nvidia.com/content/cuda/cuda-downloads.html` again one can download and install the CUDA toolkit which contains the compiler `nvcc`, libraries, headers, etc. Optionally the CUDA SDK can be downloaded and installed as well, it contains a lot of useful code examples.

Then, because a CUDA application runs on both CPU and GPU, a standard host C compiler is needed as well. When we want to compile a application that contains CUDA C code, `nvcc` will obviously handle on his own the device CUDA C code but he will then feed the standard C compiler with the regular C host code.

A.3.2 With a IDE integration

NVIDIA provide also a integration of CUDA tools with *Microsoft Visual Studio* and *Eclipse* (at the moment for *Linux* only) as well. It's called *NSight* and can be downloaded by following this url :

`http://www.nvidia.fr/object/nsight.html`

A.4 How to deactivate Windows GPU driver timeout

If the screen blink during kernel computation and a message is displayed by *Windows* to indicates that the video driver wasn't responding and has been reset, this means the WDDM TDR has trigger. See section 5.3.1 for more details about this protection.

Here is two ways to deactivate this protection :

NSight installed This is the easier way

1. Open *NSight Monitor* application
2. Go to the options - General tab
3. Under *Windows Display Driver* set false for WDDM TDR enabled

NSight not intalled It's a *Windows* registry modification

1. *Windows key + R* to open the execute windows
2. Type *regedit*
3. Go to HKLM\SYSTEM\CurrentControlSet\Control\GraphicsDrivers\
4. Add a new key DWORD named "TdrLevel" with value 0 or modify existing one

Appendix B

How to use the functions

Both CPU and GPU version of the function are provided to compute the collisions. Here are the steps needed to use them in an existing program :

1. Add the sources files to the project or link the already compiled library provided
2. Include the `detect-tri2tris-intersection.h` to gain access to the functions
3. Configure the functions launches using the `void config_set(uint8_t config);` function (optional, see below)
4. Call the functions to compute the collisions, both the CPU or the GPU version (if the machine have a CUDA-capable device) can be used.

By default the functions will stop once the first collision is detected and the CUDA function will use the workaround to avoid the WDDM TDR to trigger. However the following flags can be set :

COMPUTE_ALL_INTERSECTION Force the two intersect fct to compute intersections for the whole array even if an intersection as already been detected

VERBOSE_MODE The functions will printf some text when error occurs

NO_WDDM_TDR_WORKAROUND By default the function avoid the Microsoft TDR to trigger that kill the cuda kernel if it's running for more then 2 sec by using a workaround. This flag force the function to not use the workaround. Note that the workaround only working for a single GPU computation.

FORCE_SINGLE_GPU_COMPUTATION Allow to force the use of a single device to compute the intersections even if there is more then one CUDA-capable device, the first device will be used

Finally here is a example to demonstrate how to use the configuration function :

```
1 config_set( COMPUTE_ALL_INTERSECTION | VERBOSE_MODE );
```

Appendix C

How to use the test program

The test program is composed of two smaller programs : a OpenGL viewer and a console benchmark. The user can select, using the command line, which program to launch and with how much triangles. The command line options are the following :

-h or –help to shows help.

-t or –triangles to specify the number of triangles. (default value is 10 triangles)

-b or –benchmark to run benchmark mode to compare CPU and GPU algorithm.

Here is a example of command line :

```
1 > test-program.exe --benchmark -t 5000000
```

If launched without any options the test program will launch the viewer with 10 triangles.

The viewer on his side shows the reference triangle in blue and the other triangles in gray. When a collision is detected between the reference triangle and one of the triangles in array this last is painted in orange. The viewer have the following shortcut :

h Show help

arrows Zoom/Rotate around

wasdqe Move the blue triangle

o Show origin axis

c Show triangle coord

i Show triangle id

t Switch to special cases mode

g Switch between CPU and GPU

+ Add a triangle

- Remove a triangle

space bar New random triangles

Appendix D

Listing

D.1 Triangle-triangle intersection test algorithm

geometric-types.h

```
1  /**
2   * \file        geometric-types.h
3   * \author      Romain Maffina
4   * \version     1.0
5   * \brief       Types definitions for geometric tools.
6   */
7
8  #ifndef GEOMETRIC_TYPES_H_
9  #define GEOMETRIC_TYPES_H_
10
11 #include <stdint.h>
12
13 /**
14  * @struct coord_t
15  * @brief Represent a point/vertex in a 3D space system.
16  */
17 typedef struct{
18     float x,y,z;
19 } coord_t;
20
21 /**
22  * @struct vector_t
23  * @brief Represent a vector in a 3D space system.
24  */
25 typedef struct{
26     float x,y,z;
27 } vector_t;
28
29 /**
30  * @struct triangle_t
31  * @brief Represent a triangle with 3 coordinates.
32  */
33 typedef struct{
34     coord_t p1,p2,p3;
35 } triangle_t;
```

```
36
37 /**
38  * @struct line_t
39  * @brief Represent a line defined by a point and a direction.
40  */
41 typedef struct{
42     vector_t direction;
43     coord_t p;
44 } line_t;
45
46 /**
47  * @struct plane_t
48  * @brief Represent a plane defined by a point, a normal and the distance to zero.
49  */
50 typedef struct{
51     //!a point on the plane
52     coord_t p;
53     //!non-normalized
54     vector_t normal;
55     //!distance to zero coord
56     float d;
57 } plane_t;
58
59 #endif
```

geometric-tools.h

```
1  /**
2   * \file        geometric-tools.h
3   * \author      Romain Maffina
4   * \version     1.0
5   * \brief       Toolbox with geometric tools.
6   */
7
8  #ifndef GEOMETRIC_TOOLS_H_
9  #define GEOMETRIC_TOOLS_H_
10
11 #include <math.h>
12 #include "geometric-types.h"
13
14 //!Dot product between two point/vector
15 #define dot(u,v)     ((u).x * (v).x + (u).y * (v).y + (u).z * (v).z)
16 //! Norm of a vector
17 #define norm(v)      sqrtf(dot(v,v))
18 //! Normalize a vector
19 #define normalize(r,v)  (r).x = (v).x/norm(v);\
20                         (r).y = (v).y/norm(v);\
21                         (r).z = (v).z/norm(v);
22 //! Add a point/vector with another point/vector
23 #define add3d(r,a,b)         (r).x = (a).x+(b).x;\
24                              (r).y = (a).y+(b).y;\
25                              (r).z = (a).z+(b).z;
26 //! Substract a point/vector with another point/vector
27 #define sub3d(r,a,b)         (r).x = (a).x-(b).x;\
28                              (r).y = (a).y-(b).y;\
29                              (r).z = (a).z-(b).z;
30
31 #ifdef __cplusplus
32 extern "C"{
```

```
33 #endif
34
35 void triangle_from_triangle(triangle_t r, triangle_t *t);
36 void triangle_from_coords(coord_t p1, coord_t p2, coord_t p3, triangle_t *t);
37 void triangle_from_value(float p1x, float p1y, float p1z,
38                     float p2x, float p2y, float p2z,
39                     float p3x, float p3y, float p3z,
40                     triangle_t *t);
41
42 vector_t vector_by_scalar(vector_t v, float c);
43 vector_t vector_from_points(coord_t m, coord_t n, vector_t *mn);
44 vector_t cross(vector_t a, vector_t b, vector_t *result);
45 float point_to_plane_dist(coord_t p, plane_t pl);
46 plane_t triangle_to_plane(triangle_t t, plane_t *pl);
47 triangle_t triangle_proj_on_plane(triangle_t tri, plane_t pl, triangle_t *proj);
48 coord_t point_proj_on_line(coord_t p, line_t l, coord_t *p_bis);
49 void triangle_proj_on_line(triangle_t t, vector_t d, float *min, float *max);
50 uint8_t intersection_test_2d(triangle_t t0, triangle_t t1, plane_t pl);
51 uint8_t intervals_overlap(triangle_t T0, triangle_t T1, plane_t P0, plane_t P1, line_t L,
       float dist_T0[], float dist_T1[], uint8_t signs_T0[], uint8_t signs_T1[]);
52 uint8_t planes_intersection(plane_t p1, plane_t p2, line_t *line);
53 void triangle_line_intersection_interval(triangle_t t, float dist[], uint8_t signs[],
       line_t l, coord_t *t0, coord_t *t1);
54 void order_points(coord_t p[], float dist[], uint8_t sides[]);
55
56 #ifdef __cplusplus
57 }
58 #endif
59
60 #endif /* GEOMETRIC_TOOLS_H_ */
```

geometric-tools.cu

```
1
2  /**
3   * \file        geometric-tools.cu
4   * \author      Romain Maffina
5   * \version     1.0
6   * \brief       Toolbox with geometric tools.
7   */
8
9  #include <stdio.h>
10 #include <math.h>
11
12 #include "geometric-tools.h"
13
14
15 //const coord_t zero_p = {0,0,0};
16 //const vector_t zero_v = {0,0,0};
17
18 /**
19  * @brief Create a triangle from another triangle.
20  *
21  * @param r the triangle to copy
22  * @param *t the resulting triangle
23  */
24 __host__ __device__ void triangle_from_triangle(triangle_t r, triangle_t *t){
25     t->p1.x=r.p1.x; t->p1.y=r.p1.y; t->p1.z=r.p1.z;
26     t->p2.x=r.p2.x; t->p2.y=r.p2.y; t->p2.z=r.p2.z;
```

```
27      t->p3.x=r.p3.x; t->p3.y=r.p3.y; t->p3.z=r.p3.z;
28  }
29
30  /**
31   * @brief Create a triangle from 3 coords.
32   *
33   * @param p1 first coord
34   * @param p2 second coord
35   * @param p3 third coord
36   * @param *t the resulting triangle
37   */
38  __host__ __device__ void triangle_from_coords(coord_t p1, coord_t p2, coord_t p3,
        triangle_t *t){
39      t->p1 = p1; t->p2 = p2; t->p3 = p3;
40  }
41
42  /**
43   * @brief Create a triangle from float values.
44   *
45   * @param p1x
46   * @param p1y
47   * @param p1z
48   * @param p2x
49   * @param p2y
50   * @param p2z
51   * @param p3x
52   * @param p3y
53   * @param p3z
54   * @param *t the resulting triangle
55   */
56  __host__ __device__ void triangle_from_value(float p1x, float p1y, float p1z,
57                      float p2x, float p2y, float p2z,
58                      float p3x, float p3y, float p3z, triangle_t *t){
59      t->p1.x=p1x; t->p1.y=p1y; t->p1.z=p1z;
60      t->p2.x=p2x; t->p2.y=p2y; t->p2.z=p2z;
61      t->p3.x=p3x; t->p3.y=p3y; t->p3.z=p3z;
62  }
63
64  /**
65   * @brief Multiply a vector by a scalar.
66   *
67   * @param v the vector
68   * @param c the scalar
69   * @return the resulting vector
70   */
71  __host__ __device__ vector_t vector_by_scalar(vector_t v, float c){
72      v.x *= c;
73      v.y *= c;
74      v.z *= c;
75      return v;
76  }
77
78  /**
79   * @brief Project a triangle on a plane.
80   *
81   * @param tri the triangle to project
82   * @param pl the plane to project on
83   * @param *proj the projected triangle
84   * @return the projected triangle
```

```
85   * @see GTCG, section 12.1, page 663
86   */
87  __host__ __device__ triangle_t triangle_proj_on_plane(triangle_t tri, plane_t pl,
         triangle_t *proj){
88      vector_t n;
89
90      //the plane normal needs to be normalized first
91      normalize(pl.normal,n);
92
93      sub3d(proj->p1,
94              tri.p1,
95              vector_by_scalar(n, dot(tri.p1,n)+pl.d) );
96
97      sub3d(proj->p2,
98              tri.p2,
99              vector_by_scalar(n, dot(tri.p2,n)+pl.d) );
100
101     sub3d(proj->p3,
102             tri.p3,
103             vector_by_scalar(n, dot(tri.p3,n)+pl.d) );
104
105     return *proj;
106 }
107
108 /**
109  * @brief Project a point on a line.
110  *
111  * @param p the point
112  * @param l the line to project on
113  * @param *p_bis the projected point
114  * @return the projected point
115  * @see GTCG, section 11.5.4
116  */
117 __host__ __device__ coord_t point_proj_on_line(coord_t p, line_t l, coord_t *p_bis){
118     vector_t v, d;
119     float t;
120
121     //the line normal needs to be normalized first
122     normalize(d, l.direction);
123
124     t = dot(d, vector_from_points(l.p,p,&v));
125     add3d(*p_bis, l.p, vector_by_scalar(d, t));
126
127     return *p_bis;
128 }
129
130 /**
131  * @brief Create a vector from two points.
132  *
133  * @param m the start point
134  * @param n the end point
135  * @param *mn the resulting vector
136  * @return the resulting vector
137  */
138 __host__ __device__ vector_t vector_from_points(coord_t m, coord_t n, vector_t *mn){
139     mn->x = n.x-m.x;
140     mn->y = n.y-m.y;
141     mn->z = n.z-m.z;
142
```

```
143     return *mn;
144 }
145
146 /**
147  * @brief Cross product between two vectors
148  *
149  * @param a the first vector
150  * @param b the second vector
151  * @param *r the resulting vector
152  * @return the resulting vector
153  */
154 __host__ __device__ vector_t cross(vector_t a, vector_t b, vector_t *r){
155     r->x = a.y*b.z - a.z*b.y;
156     r->y = a.z*b.x - a.x*b.z;
157     r->z = a.x*b.y - a.y*b.x;
158
159     return *r;
160 }
161
162 /**
163  * @brief Point to plane distance.
164  *
165  * @param p the point
166  * @param pl the plane
167  * @return the distance
168  * @see GTCG, section 10.3.1, page 374
169  */
170 __host__ __device__ float point_to_plane_dist(coord_t p, plane_t pl){
171     vector_t v;
172     vector_from_points(pl.p,p,&v);
173
174     return dot(pl.normal, v);
175 }
176
177 /**
178  * @brief Obtains the plane of a triangle.
179  *
180  * @param t the triangle
181  * @param *pl the corresponding plane
182  * @return the corresponding plane
183  */
184 __host__ __device__ plane_t triangle_to_plane(triangle_t t, plane_t *pl){
185     vector_t v1, v2;
186     coord_t zero_p = {0.0f,0.0f,0.0f};
187
188     //Compute triangle's normal and define the plane by cross product
189     vector_from_points(t.p1, t.p2, &v1);
190     vector_from_points(t.p1, t.p3, &v2);
191     cross(v1, v2, &(pl->normal));
192
193     //A triangle's point is arbitrary chosen to define the plane
194     pl->p = t.p1;
195
196     //the distance to origin is computed
197     pl->d = point_to_plane_dist(zero_p, *pl);
198
199     return *pl;
200 }
201
```

```
202 /**
203  * @brief Obtains the projection of a triangle on a line.
204  *
205  * @param t the triangle
206  * @param d the direction of the line
207  * @param *min the proj starting point
208  * @param *max the proj ending point
209  */
210 __host__ __device__ void triangle_proj_on_line(triangle_t t, vector_t d, float *min, float
        *max){
211
212     int i;
213     float temp;
214     coord_t tri_points_array[3];
215
216     tri_points_array[0] = t.p1; tri_points_array[1] = t.p2; tri_points_array[2] = t.p3;
217
218     //base define of min and max
219     *min = *max = dot(d, t.p1);
220
221     //find the projection interval
222     for(i=1;i<3;i++){
223         temp = dot(d, tri_points_array[i]);
224         if(temp < *min)
225             *min = temp;
226         else if (temp > *max)
227             *max = temp;
228     }
229 }
230
231 /**
232  * @brief Obtains the interval on the line corresponding with the triangle-line
        intersection. It's
233  * part of the Moller and Haines algorithm.
234  *
235  * @param t the triangle
236  * @param dist[] the corresponding signed distances of the 3 points
237  * @param signs[] the side of each point relative to te line
238  * @param l the line
239  * @param *t0 the intersection starting point
240  * @param *t1 the intersection ending point
241  * @see order_points()
242  */
243 __host__ __device__ void triangle_line_intersection_interval(triangle_t t, float dist[],
        uint8_t signs[], line_t l, coord_t *t0, coord_t *t1){
244
245     vector_t v;
246     float scalar;
247     coord_t p[3];
248
249     //Project the first triangle points on the line
250     point_proj_on_line(t.p1, l, &p[0]);
251     point_proj_on_line(t.p2, l, &p[1]);
252     point_proj_on_line(t.p3, l, &p[2]);
253
254     //Order points (necessary for the equations below to works)
255     order_points(p, dist, signs);
256
257     //Compute t0 coord
```

```
258     vector_from_points(p[0], p[2], &v);
259     scalar = dist[0] - dist[2]==0?0:dist[0] / ( dist[0] - dist[2] );
260     v = vector_by_scalar(v, scalar);
261     add3d(*t0, p[0], v);
262 
263     //Compute t1 coord
264     vector_from_points(p[1], p[2], &v);
265     scalar = dist[1] - dist[2]==0?0:dist[1] / ( dist[1] - dist[2] );
266     v = vector_by_scalar(v, scalar);
267     add3d(*t1, p[1], v);
268 }
269 
270 /**
271  * @brief Realize a 2D intersection test between two triangles.
272  * Thus the test is realized on the same plane.
273  *
274  * This intersection test uses the method called The method of separating axes:
275  * First we need to find each normal of each edge of both triangles. Then we project
276  * both triangles on each normal (in fact the axis defined by the normal). If the
277  * projections overlap for the all 6 axis then the triangles intersect othewise not.
278  *
279  * @param t0 the first triangle
280  * @param t1 the second triangle
281  * @param pl the plane on which to project
282  * @return 1 = intersect, 0 = no intersection
283  */
284 __host__ __device__ uint8_t intersection_test_2d(triangle_t t0, triangle_t t1, plane_t pl){
285     int i; vector_t axis;
286     float min0, min1, max0, max1;
287 
288     vector_t edges_array[6];
289 
290     //We find and store each edge of both triangles to compute then,
291     //in the for loop, the normal of each edge.
292     vector_from_points(t0.p1, t0.p2, &edges_array[0]);
293     vector_from_points(t0.p2, t0.p3, &edges_array[1]);
294     vector_from_points(t0.p3, t0.p1, &edges_array[2]);
295     vector_from_points(t1.p1, t1.p2, &edges_array[3]);
296     vector_from_points(t1.p2, t1.p3, &edges_array[4]);
297     vector_from_points(t1.p3, t1.p1, &edges_array[5]);
298 
299     for(i=0;i<6;i++){
300         //we compute the line on which we'll project
301         cross( edges_array[i], pl.normal, &axis );
302         triangle_proj_on_line(t0, axis, &min0, &max0);
303         triangle_proj_on_line(t1, axis, &min1, &max1);
304 
305         //check if the intervals don't overlap and if so exit
306         if(max1<min0 || max0 < min1)
307             return 0;
308     }
309     return 1;
310 }
311 
312 /**
313  * @brief Prepare (order) the three points of a triangle for the Moller and Haines
        algorithm.
314  *
315  * @param p[] the triangle points
```

```
316  * @param dist[] the corresponding distances from each point to the line
317  * @param sides[] the side of each point relative to the line
318  * @see triangle_line_intersection_interval()
319  */
320 __host__ __device__ void order_points(coord_t p[], float dist[], uint8_t sides[]){
321
322     coord_t p_tmp;
323     float dist_tmp;
324     int i=0;
325
326     //No point on the line = standard ordering (the point alone on his side is placed in
            last position)
327     if(sides[2] == 0){
328
329         //if the point alone is negative
330         if(sides[0] == 2){
331             for(i=0;i<3;i++){
332                 if(dist[i]<0){
333                     p_tmp=p[2]; dist_tmp=dist[2];
334                     p[2]=p[i]; dist[2]=dist[i];
335                     p[i]=p_tmp; dist[i]=dist_tmp;
336                     break;
337                 }
338             }
339         }
340         //if the point alone is positive
341         else if(sides[1] == 2){
342             for(i=0;i<3;i++){
343                 if(dist[i]>0){
344                     p_tmp=p[2]; dist_tmp=dist[2];
345                     p[2]=p[i]; dist[2]=dist[i];
346                     p[i]=p_tmp; dist[i]=dist_tmp;
347                     break;
348                 }
349             }
350         }
351
352     }
353
354     //One point is on the line
355     else if(sides[2] == 1){
356         //the other points are on the same side of the line
357         if(sides[0] == 2 || sides[1] == 2){
358             for(i=0;i<3;i++){
359                 if(dist[i]==0){
360                     p_tmp=p[2]; dist_tmp=dist[2];
361                     p[2]=p[i]; dist[2]=dist[i];
362                     p[i]=p_tmp; dist[i]=dist_tmp;
363                     break;
364                 }
365             }
366         }
367         //One point is on one side and the other point on the other side
368         else if(sides[1]==1 && sides[0]==1){
369             //The last point is the one on the line, need to swap
370             if(dist[2]==0){
371                 p_tmp=p[2]; dist_tmp=dist[2];
372                 p[2]=p[0]; dist[2]=dist[0];
373                 p[0]=p_tmp; dist[0]=dist_tmp;
```

```
374             }
375         }
376     }
377     //Two point are on the line
378     else if(sides[2] == 2){
379         for(i=0;i<3;i++){
380             if(dist[i]==0){
381                 p_tmp=p[2]; dist_tmp=dist[2];
382                 p[2]=p[i]; dist[2]=dist[i];
383                 p[i]=p_tmp; dist[i]=dist_tmp;
384                 break;
385             }
386         }
387     }
388 }
389
390 /**
391  * @brief Test the intersection between two triangles using the Moller and Haines :
392  * the "interval overlap method".
393  *
394  * @param T0 the first triangle
395  * @param T1 the second triangle
396  * @param P0 the plane of T0
397  * @param P1 the plane of T1
398  * @param L the line on which to compute the interval
399  * @param dist_T0[] the signed distance for each point of T0 to P1
400  * @param dist_T1[] the signed distance for each point of T1 to P0
401  * @param signs_T0[] the side of each T0 point respect to P1
402  * @param signs_T1[] the side of each T1 point respect to P0
403  * @return 1 = intersection, 0 = no intersection
404  */
405 __host__ __device__ uint8_t intervals_overlap(triangle_t T0, triangle_t T1, plane_t P0,
        plane_t P1, line_t L, float dist_T0[], float dist_T1[], uint8_t signs_T0[], uint8_t
        signs_T1[]){
406
407     uint8_t overlap=0;
408     vector_t v, d;
409     coord_t T00, T01, T10, T11;
410     float t00, t01, t10, t11;
411     float min, max;
412
413     //the line direction vector needs to be normalized here first
414     normalize(d, L.direction);
415
416     //Compute for both triangles intersection with the line
417     triangle_line_intersection_interval(T0, dist_T0, signs_T0, L, &T00, &T01);
418     triangle_line_intersection_interval(T1, dist_T1, signs_T1, L, &T10, &T11);
419
420     /*
421      * here we compute each value for t00, t01, .. as the (signed) norm of the vector
422      * defined by the point P of the line and each intersection point computed before
423     */
424     vector_from_points(L.p, T00, &v);
425     t00 = dot(d,v);
426
427     vector_from_points(L.p, T01, &v);
428     t01 = dot(d,v);
429
430     vector_from_points(L.p, T10, &v);
```

```
431     t10 = dot(d,v);
432
433     vector_from_points(L.p, T11, &v);
434     t11 = dot(d,v);
435
436     /* Check if the intervals computed before overlap or not */
437     min = t00<t01?t00:t01;
438     max = t01>t00?t01:t00;
439
440     if( (t10 >= min && t10 <= max) ||
441         (t11 >= min && t11 <= max)
442             ){
443         overlap=1;
444     }
445
446     min = t10<t11?t10:t11;
447     max = t11>t10?t11:t10;
448
449     if( (t00 >= min && t00 <= max) ||
450         (t01 >= min && t01 <= max)
451             ){
452         overlap=1;
453     }
454
455     return overlap;
456 }
457
458 /**
459  * @brief Find the resulting line of two planes intersection.
460  *
461  * @param pl1 the first plane
462  * @param pl2 the second plane
463  * @param *line the resulting line
464  * @return  1 indicate that the planes intersect
465  *          0 indicate that the planes don't intersect (parallels)
466  * @see GTCG, section 11.5.1, page 529
467  */
468 __host__ __device__ uint8_t planes_intersection(plane_t pl1, plane_t pl2, line_t *line){
469     vector_t v;
470     float n1_n2_dot, n1_norm_sqr, n2_norm_sqr, a, b;
471
472     //the line direction
473     cross(pl1.normal, pl2.normal, &v);
474
475     //if planes are parallels -> return
476     if(v.x==0.0f && v.y==0.0f && v.z==0.0f){
477         return 0;
478     }
479     else
480     {
481         line->direction = v;
482
483         n1_n2_dot = dot(pl1.normal, pl2.normal);
484         n1_norm_sqr = dot(pl1.normal, pl1.normal);
485         n2_norm_sqr = dot(pl2.normal, pl2.normal);
486
487         a=0;b=0;
488         //scalars used to find a point on the line (it's a linear combination of the two
              planes normals)
```

```
489         if((n1_n2_dot*n1_n2_dot - n1_norm_sqr * n2_norm_sqr)!=0){
490             a = (pl2.d * n1_n2_dot - pl1.d * n2_norm_sqr) / (n1_n2_dot*n1_n2_dot -
                    n1_norm_sqr * n2_norm_sqr);
491             b = (pl1.d * n1_n2_dot - pl2.d * n1_norm_sqr) / (n1_n2_dot*n1_n2_dot -
                    n1_norm_sqr * n2_norm_sqr);
492         }
493
494         //we define the line point
495         add3d(
496                 line->p,
497                 vector_by_scalar(pl1.normal, a),
498                 vector_by_scalar(pl2.normal, b)
499                 );
500
501         //Correction
502         line->p.x = -line->p.x;
503         line->p.y = -line->p.y;
504         line->p.z = -line->p.z;
505     }
506
507     return 1;
508 }
```

detect-tri2tris-intersection.h

```
1  /**
2   * \file        detect-tri2tris-intersection.h
3   * \author      Romain Maffina
4   * \version     1.0
5   * \brief       Provide tools to detect intersections between triangles.
6   *
7   */
8
9  #ifndef __DETECT_TRI2TRIS_INTERSECTION_
10 #define __DETECT_TRI2TRIS_INTERSECTION_
11
12 #include "geometric-types.h"
13
14 //! Force the two intersect fct to compute intersections for the whole array even if an
       intersection as already been detected
15 #define COMPUTE_ALL_INTERSECTION 0x01
16 //! The functions will printf some text when error occurs
17 #define VERBOSE_MODE 0x02
18 /** By default the function avoid the Microsoft TDR to trigger that kill the cuda kernel if
       it's running for more then 2 sec by using a workaround.
19  This flag force the function to not use the workaround. Note that the workaround only
       working for a single GPU computation.
20  To manually deactivate the TDR trigger : HKLM\\SYSTEM\\CurrentControlSet\\Control\\
       GraphicsDrivers\\ --> DWORD : TdrLevel = 0 */
21 #define NO_WDDM_TDR_WORKAROUND 0x04
22 //! Allow to force the use of a single device to compute the intersections even if there is
       more then one CUDA-capable device, the first device will be used
23 #define FORCE_SINGLE_GPU_COMPUTATION 0x08
24
25 #ifdef __cplusplus
26 extern "C"{
27 #endif
28
29 void config_reset();
```

```
30 void config_set(uint8_t config);
31 int8_t tri2tris_intersection_detect_gpu(triangle_t ref, triangle_t tri_array[], int
       array_sz, uint8_t result_array[]);
32 int8_t tri2tris_intersection_detect_cpu(triangle_t ref, triangle_t tri_array[], int
       array_sz, uint8_t result_array[]);
33
34 #ifdef __cplusplus
35 }
36 #endif
37
38 #endif
```

detect-tri2tris-intersection.cu

```
1  /**
2   * \file        detect-tri2tris-intersection.cu
3   * \author      Romain Maffina
4   * \version     1.0
5   * \brief       Provide tools to detect intersections between triangles.
6   *
7   */
8
9  #include <stdio.h>
10 #include "detect-tri2tris-intersection.h"
11 #include "geometric-tools.cu" // Limitation, CUDA have no linker
12
13 /**********/
14 /* GLOBAL */
15 /**********/
16
17 //!configuration flags. @see config_set()
18 uint8_t flags = 0x00;
19
20 /**
21  * @brief Reset the configuration.
22  * @see config_set()
23  */
24 void config_reset(){
25     flags = 0x00;
26 }
27
28 /**
29  * @brief Set some configuration.
30  * @param config the configuration
31  * @see COMPUTE_ALL_INTERSECTION, VERBOSE_MODE, NO_WDDM_TDR_WORKAROUND,
       FORCE_SINGLE_GPU_COMPUTATION, config_reset()
32  *
33  * Here is a example of use : config_set(VERBOSE_MODE | NO_WDDM_TDR_WORKAROUND);
34  *
35  * By default the tools run silent (no printf), try to compute on multiples GPU if possible
       ,
36  * don't handle the TDR trigger problem and return once a intersection has been detected.
37  */
38 void config_set(uint8_t config){
39     flags |= config;
40 }
41
42 /**
43  * @brief Test the intersection between two triangles.
```

```
44  *
45  * @param T0 the first triangle
46  * @param T1 the second triangle
47  * @return  0=don't intersect
48  *          1=intersect
49  * @see The book "Geometric Tools for Computer Graphics", p.542
50  *
51  * This function is based on the outline given in the book.
52  */
53 __host__ __device__ uint8_t intersect(triangle_t T0, triangle_t T1){
54
55     plane_t P0, P1;
56     float dist_T0[3], dist_T1[3];
57     uint8_t signs_T0[3] = { 0 }, signs_T1[3] = { 0 };
58     line_t L;
59
60     /* 2.Compute the plane equation of T0 */
61     triangle_to_plane(T0, &P0);
62
63     /* 3.Compute the signed distances of T1 to P0 */
64     dist_T1[0] = point_to_plane_dist(T1.p1, P0);
65     dist_T1[1] = point_to_plane_dist(T1.p2, P0);
66     dist_T1[2] = point_to_plane_dist(T1.p3, P0);
67
68     /* Compute side of T1 vertices relative to P0 */
69     if( dist_T1[0] > 0) signs_T1[0]++; else if(dist_T1[0]<0) signs_T1[1]++; else signs_T1
           [2]++;
70     if( dist_T1[1] > 0) signs_T1[0]++; else if(dist_T1[1]<0) signs_T1[1]++; else signs_T1
           [2]++;
71     if( dist_T1[2] > 0) signs_T1[0]++; else if(dist_T1[2]<0) signs_T1[1]++; else signs_T1
           [2]++;
72
73     /* 4. Quick reject if all vertices of T1 are on one side of P0 */
74     if(signs_T1[0] == 3 || signs_T1[1] == 3 )
75         return 0;
76
77     /* 5. Compute the plane equation of T1 */
78     triangle_to_plane(T1, &P1);
79
80     /* 6. See if planes are coincident to choose the correct intersection test */
81     if(planes_intersection(P0, P1, &L) == 0){
82     //planes are coincident = 2D intersection test by Separated Axes method
83
84         return intersection_test_2d(T0, T1, P0);
85     }
86     else{
87     //planes are not coincident = standard "interval overlap method" intersection test
88
89         /* Compute the signed distances of T0 to P1 */
90         dist_T0[0] = point_to_plane_dist(T0.p1, P1);
91         dist_T0[1] = point_to_plane_dist(T0.p2, P1);
92         dist_T0[2] = point_to_plane_dist(T0.p3, P1);
93
94         /* Compute side of T0 vertices relative to P1 */
95         if( dist_T0[0] > 0) signs_T0[0]++; else if(dist_T0[0]<0) signs_T0[1]++; else
               signs_T0[2]++;
96         if( dist_T0[1] > 0) signs_T0[0]++; else if(dist_T0[1]<0) signs_T0[1]++; else
               signs_T0[2]++;
```

```
97          if( dist_T0[2] > 0) signs_T0[0]++; else if(dist_T0[2]<0) signs_T0[1]++; else
                signs_T0[2]++;
98
99          /* 7. Quick reject if all vertices of T0 are on one side of P1 */
100         if(signs_T0[0] == 3 || signs_T0[1] == 3 )
101             return 0;
102
103         /* 8. Compute intersection line */
104         //already done in if
105
106         /* 9. Compute intervals */
107         return intervals_overlap(T0, T1, P0, P1, L, dist_T0, dist_T1, signs_T0, signs_T1);
108     }
109
110 }
111
112 /*************/
113 /* CPU STUFF */
114 /*************/
115
116 /**
117  * @brief CPU VERSION - Test the 3D intersection between a given triangle and each triangle
          contained in the array passed
118  *
119  * @param ref the ref triangle
120  * @param tri_array[] the triangles array
121  * @param array_sz the size of the array
122  * @param result_array[] the result for each triangle in tri_array
123
124  * @return  0=no triangle of the array intersects the ref triangle,
125  *          1=at least one triangle of the array intersects the ref triangle
126  *          -1=error
127  *
128  * This function is using exactly the same algorithm as tri2tris_intersection_detect_gpu()
129  * but it should be used only if the amound of triangles in array is less then 1M approx
         then the tri2tris_intersection_detect_gpu()
130  * function become a lot quicker. It's running on 1 core only.
131  */
132 int8_t tri2tris_intersection_detect_cpu(triangle_t ref, triangle_t tri_array[], int
        array_sz, uint8_t result_array[]){
133     uint8_t x=0; //To remember a intersection occured, default=0
134
135     int i;
136     for(i=0;i<array_sz;i++){
137         result_array[i]=0;
138         if(intersect(ref, tri_array[i])){
139             x=result_array[i]=1;
140             if( !(flags & COMPUTE_ALL_INTERSECTION) )
141                 break;
142         }
143     }
144
145     return x;
146 }
147
148 /**************/
149 /* CUDA STUFF */
150 /**************/
151
```

```
152 /* Device variables */
153 //! Global on device (shared between GPU as well) intersect variable to know when a
        intersection as been found
154 __device__ uint8_t dev_intersect;
155 //! Used to store the configuration on the device at function call
156 __constant__ uint8_t dev_flags;
157
158 /* Prototypes */
159 int ConvertSMVer2Cores(int major, int minor);
160 void slit_kernel_call(triangle_t* dev_ref, triangle_t* dev_tri_array, int size, int
        nb_threads, uint8_t* dev_result_array);
161 __global__ void kernel(const triangle_t *ref, const triangle_t *tri_array, int array_sz,
        uint8_t *result_array);
162
163
164 /**
165  * @brief Give the amount of CUDA cores of a device.
166  *
167  * @param major the major of the CUDA capability of a device
168  * @param minor the minor of the CUDA capability of a device
169  *
170  * Taken from the CUDA SDK Examples
171  */
172 int ConvertSMVer2Cores(int major, int minor){
173         // Defines for GPU Architecture types (using the SM version to determine the # of
                cores per SM
174         typedef struct {
175                 int SM; // 0xMm (hexidecimal notation), M = SM Major version, and m = SM
                        minor version
176                 int Cores;
177         } sSMtoCores;
178
179         sSMtoCores nGpuArchCoresPerSM[] =
180         { { 0x10,  8 },
181           { 0x11,  8 },
182           { 0x12,  8 },
183           { 0x13,  8 },
184           { 0x20, 32 },
185           { 0x21, 48 },
186           {   -1, -1 }
187         };
188
189         int index = 0;
190         while (nGpuArchCoresPerSM[index].SM != -1) {
191                 if (nGpuArchCoresPerSM[index].SM == ((major << 4) + minor) ) {
192                         return nGpuArchCoresPerSM[index].Cores;
193                 }
194                 index++;
195         }
196         printf("MapSMtoCores undefined SMversion %d.%d!\n", major, minor);
197         return -1;
198 }
199
200 /**
201  * @brief This function split a kernel call into multiple calls. This is used to avoid
        Microsoft WDDM TDR to trigger.
202  *
203  * @param dev_ref the ref triangle
204  * @param dev_tri_array[] the triangles array
```

```
205  * @param size the size of the array
206  * @param dev_result_array[] the result for each triangle in tri_array
207  *
208  * The WDDM TDR (Windows Display Driver Model - Timeout Detection and Recovery) is a
         protection to avoid freezed display
209  * in case of a display driver fail. It trigger after 2sec if the graphic driver can't
         recover the hand on the graphic device.
210  * Because a kernel occupies the device at 100% the TDR trigger and kill the driver and the
          running kernels to recover the
211  * graphic card. Thus this function allows to execute kernels longer then 2 secondes.
212  */
213 void slit_kernel_call(triangle_t* dev_ref, triangle_t* dev_tri_array, int size, int
        nb_threads, uint8_t* dev_result_array){
214
215     cudaDeviceProp devProp;
216     cudaGetDeviceProperties(&devProp, 0);
217
218     /* Adapt values to the device for kernel launches */
219     int max_triangle_pro_kernel = ConvertSMVer2Cores(devProp.major,devProp.minor) * devProp
            .multiProcessorCount*20000;
220     int nb_calls = (size+max_triangle_pro_kernel-1)/max_triangle_pro_kernel;
221
222     /* Launch the split kernels synchronously (due to the cudaThreadSynchronize) */
223     int i, launch_size = max_triangle_pro_kernel;
224     for(i=0;i<nb_calls;i++){
225
226         //If it's the last call and a rest exist --> the size must be corrected
227         if(i==nb_calls-1 && size%max_triangle_pro_kernel != 0)
228             launch_size = size%max_triangle_pro_kernel;
229
230         //Launch the current split kernel and synchronize to avoid WDDM TDR to trigger
231         kernel<<<(launch_size+nb_threads-1)/nb_threads,nb_threads>>>(dev_ref, dev_tri_array
                +i*max_triangle_pro_kernel, launch_size, dev_result_array+i*
                max_triangle_pro_kernel);
232         cudaDeviceSynchronize();
233     }
234 }
235
236
237 /**
238  * @brief The kernel = the function that will execute on the device.
239  *
240  * @param ref the ref triangle
241  * @param tri_array[] the triangles array
242  * @param array_sz the size of the array
243  * @param result_array[] the result for each triangle in tri_array
244  */
245 __global__ void kernel(const triangle_t *ref, const triangle_t *tri_array, int array_sz,
        uint8_t *result_array){
246
247     //if config is set to compute all intersection then the if block is always done,
248     //otherwise the computation is skipped once a intersection have been found
249     if( (dev_flags & COMPUTE_ALL_INTERSECTION) || !dev_intersect ){
250         //Calculate the index of the triangle for the current thread to compute
                intersection
251         int i = blockIdx.x * blockDim.x + threadIdx.x;
252
253         //Control that we're not out of the array
254         if(i<array_sz){
```

```
255             //Default value = not intersecting
256             result_array[i]=0;
257             //Intersection test between the ref triangle and the current triangle from
                    array
258             if(intersect(*ref, tri_array[i]))
259                 dev_intersect=result_array[i]=1;
260         }
261     }
262
263 }
264
265 /**
266  * @brief CUDA VERSION - Test the 3D intersection between a given triangle and each
        triangle contained in the array passed
267  *
268  * @param ref the ref triangle
269  * @param tri_array_full[] the triangles array
270  * @param array_sz_full the size of the array
271  * @param result_array_full[] the result for each triangle in tri_array
272  *
273  * @return  0=no triangle of the array intersects the ref triangle,
274  *          1=at least one triangle of the array intersects the ref triangle
275  *          -1=error
276  *
277  * This function is using exactly the same algorithm as tri2tris_intersection_detect_cpu()
278  * but it should be used only if the amound of triangles in array is more then 1M approx.
        For smaller arrays
279  * the CPU version should be quicker. It can run on multi-GPU.
280  */
281 int8_t tri2tris_intersection_detect_gpu(triangle_t ref, triangle_t tri_array_full[], const
        int array_sz_full, uint8_t result_array_full[]){
282
283     uint8_t intersect = 0;
284     uint8_t *ptr_dev_intersect;
285
286     triangle_t **tri_array = NULL;
287     uint8_t **result_array = NULL;
288     int *array_sz = NULL;
289     int *nb_threads = NULL;
290
291     triangle_t **dev_ref = NULL;
292     triangle_t **dev_tri_array = NULL;
293     uint8_t **dev_result_array = NULL;
294
295     int device_count;
296     cudaError_t cudaStatus;
297     char str_debug[100] = "";
298
299     // Get the device (=GPU) count to exploit them all
300     cudaGetDeviceCount(&device_count);
301     if(device_count==0)
302         goto Error;
303
304     // Force single GPU computation if configured
305     if(flags & FORCE_SINGLE_GPU_COMPUTATION)
306         device_count=1;
307
308     // Allocate arrays of host pointers to dataset for each GPU to compute
309     tri_array = (triangle_t**)calloc(device_count, sizeof(triangle_t));
```

```
310     result_array = (uint8_t**)calloc(device_count, sizeof(uint8_t));
311     array_sz = (int*)calloc(device_count, sizeof(int));
312     nb_threads = (int*)calloc(device_count,sizeof(int));
313
314     // Allocate arrays of device pointers for the parameters
315     dev_ref = (triangle_t**)calloc(device_count, sizeof(triangle_t));
316     dev_tri_array = (triangle_t**)calloc(device_count, sizeof(triangle_t));
317     dev_result_array = (uint8_t**)calloc(device_count, sizeof(uint8_t));
318
319
320     /* Allocate memory on devices and then copy contents */
321     int i;
322     for(i=0;i<device_count;i++){
323
324         /* Compute device-specific pointers and sizes */
325         //if last device AND array size cannot by fully devided by the number of devices
326         if(i==device_count-1 && array_sz_full%device_count!=0)
327             array_sz[i] = array_sz_full/device_count + array_sz_full%device_count;
328         else
329             array_sz[i] = array_sz_full/device_count;
330
331         tri_array[i] = tri_array_full + (i*(array_sz_full/device_count) );
332         result_array[i] = result_array_full + (i*(array_sz_full/device_count) );
333
334
335         // Set the target device for the next cudaXXX calls
336         if( (cudaStatus = cudaSetDevice(i)) != cudaSuccess ){
337             sprintf(str_debug, "cudaSetDevice failed!"); goto Error;
338         }
339
340         cudaDeviceProp dev_prop;
341         cudaGetDeviceProperties(&dev_prop, i);
342         int max_threads=dev_prop.regsPerBlock/48;
343         nb_threads[i]=1;
344         while(nb_threads[i]*2<max_threads){
345             nb_threads[i] *= 2;
346         };
347
348         // Allocate ref triangle on device
349         if( (cudaStatus = cudaMalloc((void**)&dev_ref[i], sizeof(triangle_t))) !=
                cudaSuccess ) {
350             sprintf(str_debug, "cudaMalloc failed for ref triangle!"); goto Error;
351         }
352
353         // Allocate array of triangles on device
354         if( (cudaStatus = cudaMalloc((void**)&dev_tri_array[i], array_sz[i] * sizeof(
                triangle_t))) != cudaSuccess ) {
355             sprintf(str_debug, "cudaMalloc failed for array of triangles!"); goto Error;
356         }
357
358         // Allocate array for results
359         if( (cudaStatus = cudaMalloc((void**)&dev_result_array[i], array_sz[i] * sizeof(
                uint8_t))) != cudaSuccess) {
360             sprintf(str_debug, "cudaMalloc failed for results array!"); goto Error;
361         }
362
363         // Copy ref triangle from host to device
364         if( (cudaStatus = cudaMemcpy(dev_ref[i], &ref, sizeof(triangle_t),
                cudaMemcpyHostToDevice)) != cudaSuccess) {
```

```
365             sprintf(str_debug, "cudaMemcpy failed for ref triangle!"); goto Error;
366         }
367
368         // Copy array of triangles from host to device
369         if( (cudaStatus = cudaMemcpy(dev_tri_array[i], tri_array[i], array_sz[i] * sizeof(
                triangle_t), cudaMemcpyHostToDevice)) != cudaSuccess) {
370             sprintf(str_debug, "cudaMemcpy failed for triangles array!"); goto Error;
371         }
372     }
373
374     // Initialize global device intersect variable
375     cudaGetSymbolAddress((void **)&ptr_dev_intersect, "dev_intersect");
376     cudaMemset(ptr_dev_intersect,0,sizeof(uint8_t));
377
378     // Copy config flags to device
379     cudaMemcpyToSymbol("dev_flags", &flags, sizeof(flags));
380
381     /* Launch kernels */
382     if(device_count==1 && !(flags & NO_WDDM_TDR_WORKAROUND) ){
383         slit_kernel_call(dev_ref[0], dev_tri_array[0], array_sz[0], nb_threads[0],
                dev_result_array[0]);
384     }else{
385
386         for(i=0;i<device_count;i++){
387             cudaSetDevice(i);
388
389             kernel<<<(array_sz[i]+nb_threads[i]-1)/nb_threads[i], nb_threads[i]>>>
390                 (dev_ref[i], dev_tri_array[i], array_sz[i], dev_result_array[i]);
391         }
392     }
393
394     /* Synchronize host with devices and copy results back to host */
395     for(i=0;i<device_count;i++){
396         cudaSetDevice(i);
397
398         // cudaDeviceSynchronize waits for the kernel to finish
399         if( (cudaStatus = cudaDeviceSynchronize()) != cudaSuccess) {
400             sprintf(str_debug, "cudaDeviceSynchronize failed after launching addKernel!\n")
                    ; goto Error;
401         }
402
403         // Copy results array from device to host
404         if( (cudaStatus = cudaMemcpy(result_array[i], dev_result_array[i], array_sz[i] *
                sizeof(uint8_t), cudaMemcpyDeviceToHost)) != cudaSuccess) {
405             sprintf(str_debug, "cudaMemcpy failed for results array!"); goto Error;
406         }
407     }
408
409     // Recover the value of the global intersect variable
410     cudaMemcpyFromSymbol(&intersect, "dev_intersect", sizeof(uint8_t), 0,
            cudaMemcpyDeviceToHost);
411
412 Error:
413     /* Free allocated memory on devices and handle errors if some */
414     for(i=0;i<device_count;i++){
415         cudaSetDevice(i);
416
417         cudaFree(dev_ref[i]);
418         cudaFree(dev_tri_array[i]);
```

```
419         cudaFree(dev_result_array[i]);
420     }
421     if(flags & VERBOSE_MODE){
422         if(device_count==0){
423             printf("\n\nError : No devices found!\nExiting...\n\n");
424             return -1;
425         }
426         if (cudaStatus != cudaSuccess){
427             printf("\n\nCUDA have reported the following error : \"%s\"\n",
                    cudaGetErrorString(cudaStatus));
428             printf("Details : \"%s\"\nExiting...\n\n", str_debug);
429             return -1;
430         }else if(cudaPeekAtLastError() != cudaSuccess){
431             printf("\n\nCUDA have reported the following error : \"%s\"\n",
                    cudaGetErrorString(cudaGetLastError()));
432             return -1;
433         }
434     }
435
436     return intersect;
437 }
```

D.2 Test program

main.c

```
1  /**
2   * \file        main.c
3   * \author      Romain Maffina
4   * \version     1.0
5   * \brief       Main, test program for triangle-triangle collisions.
6   */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <string.h>
11
12 #include "test-tools.h"
13 #include "console.h"
14 #include "viewer.h"
15
16 int main(int argc, char** argv){
17 int nb_triangles=10;
18 uint8_t console=0, help=0;
19
20     int i;
21     for(i=1;i<argc;i++){
22         if(!strcmp(argv[i], "-t") || !strcmp(argv[i], "--triangles") )
23             nb_triangles = atoi(argv[(i++)+1]);
24         else if(!strcmp(argv[i], "-b") || !strcmp(argv[i], "--benchmark") )
25             console=1;
26         else if(!strcmp(argv[i], "-h") || !strcmp(argv[i], "--help") )
27             help=1;
28         else{
29             printf("Error when parsing command line... exiting. Enter -h or --help
                    parameter.");
30             fflush(stdout);
```

```
31              return EXIT_FAILURE;
32          }
33      }
34
35      if(help){
36          printf("This test program is composed of two smaller programs : a OpenGL viewer and
                a console benchmark. It is intended to test the CPU and GPU (with CUDA)
                version of a triangle-triangle intersection test algorithm.\n\n"
37                  "Parameters:\n"
38                  "\t -h or --help to shows help.\n"
39                  "\t -t or --triangles to specify the number of triangles.\n"
40                  "\t -b or --benchmark to run benchmark mode to compare CPU and GPU
                        algorithm.\n");
41          return EXIT_SUCCESS;
42      }
43      else if(console){
44          if(start_benchmark(argc, argv, nb_triangles)<0)
45              return EXIT_FAILURE;
46          return EXIT_SUCCESS;
47      }
48      else{
49          start_viewer(argc, argv, nb_triangles);
50          return EXIT_SUCCESS;
51      }
52  }
```

test-tools.h

```
1  /**
2   * \file        test-tools.h
3   * \author      Romain Maffina
4   * \version     1.0
5   * \brief       Toolbox associated with the test-program.
6   */
7
8  #ifndef TEST_TOOLS_H_
9  #define TEST_TOOLS_H_
10
11 #include "geometric-types.h"
12
13 #define MAX_X 8
14 #define MAX_Y 8
15 #define MAX_Z 8
16 #define TRIANGLE_SZ 5
17
18 extern triangle_t *tri_array;
19 extern uint8_t *tri_result_array;
20 extern triangle_t ref;
21
22 int chrono();
23 int tri_generator(uint32_t size);
24 void free_memory();
25
26 #endif /* TEST_TOOLS_H_ */
```

test-tools.c

```
1  /**
```

```
 * \file        test-tools.c
 * \author      Romain Maffina
 * \version     1.0
 * \brief       Toolbox associated with the test-program.
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#include "test-tools.h"
#include "geometric-tools.h"


/* Global */
triangle_t *tri_array = NULL;
uint8_t *tri_result_array = NULL;
triangle_t ref;

/* Models */
triangle_t ref_base = { { 2.0f,-2.0f,4.0f}, { 1.0f,5.0f,-2.0f}, {1.0f,-3.0f,-3.0f} };
triangle_t ref_special_cases = { { 0,-2,6}, { 0,5,0}, {0,-3,0} };
triangle_t special_cases_array[10] = {
        { {-2,-2,0}, {2,-3,0}, {3,-1,0} },
        { {2,-1,0}, {-2,-1.5,0}, {-3,0,0} },
        { {0,0,0}, {-2,0.5,0}, {-3,2,0} },
        { {0,0.5,0}, {2,0,0}, {3,1,0} },
        { {3,1.5,0}, {0,1,0}, {0,2,0} },
        { {-3,3,0}, {0,2.5,0}, {0,3.5,0} },
        { {-2,4.7,0}, {0,3.7,0}, {2,4.7,0} },

        { {1.5,-3,5.5}, {-2,-2.00,5.5}, {1.5,0,5.5} }, //fully inside
        { {2,0,0.5}, {-2,0,1.5}, {1,0,4} }, //fully inside

        { {0,3,1}, {0,1,1.5}, {0,4,3} } //same plane
};

int chrono(){
    static int call_count;
    static clock_t start;

    call_count++;

    //second call, return the time lapse until last call
    if(call_count%2==0){
        return (clock()-start);
    }
    //first call, store the start time
    else{
         start = clock();
    }
    return 0;
}

//int tri_generator(triangle_t *tri_array[], uint8_t *tri_result_array[], uint32_t size){
int tri_generator(uint32_t size){
    static uint32_t nb_calls, last_size;
    uint8_t special_cases_mode=0;
```

```
61      unsigned int i;
62      float x,y,z;
63
64      if(nb_calls==0)
65          srand(time(NULL));
66
67      //define reference triangle
68      triangle_from_triangle(ref_base, &ref);
69
70      //Handle special cases mode
71      if(size==0){
72          special_cases_mode=1;
73          size = sizeof(special_cases_array)/sizeof(triangle_t);
74          triangle_from_triangle(ref_special_cases, &ref);
75      }
76
77      if(size != last_size){
78          do{
79              tri_array = (triangle_t*)realloc(tri_array, sizeof(triangle_t)*size);
80          }while(tri_array==NULL && (size-=10));
81
82          tri_result_array = (uint8_t*)realloc(tri_result_array, sizeof(uint8_t)*size);
83          if(tri_result_array==NULL) return -1;
84      }
85
86      //define triangles
87      for(i=0;i<size;i++){
88          tri_result_array[i]=0;
89
90          if(special_cases_mode){
91              tri_array[i].p1.x = special_cases_array[i].p1.x; tri_array[i].p1.y =
                    special_cases_array[i].p1.y; tri_array[i].p1.z = special_cases_array[i].p1.
                    z;
92              tri_array[i].p2.x = special_cases_array[i].p2.x; tri_array[i].p2.y =
                    special_cases_array[i].p2.y; tri_array[i].p2.z = special_cases_array[i].p2.
                    z;
93              tri_array[i].p3.x = special_cases_array[i].p3.x; tri_array[i].p3.y =
                    special_cases_array[i].p3.y; tri_array[i].p3.z = special_cases_array[i].p3.
                    z;
94          }else{
95              //gen base coords
96
97              x=(rand()%(MAX_X*10)-MAX_X*10/2)/10.0f;
98              y=(rand()%(MAX_Y*10)-MAX_Y*10/2)/10.0f;
99              z=(rand()%(MAX_Z*10)-MAX_Z*10/2)/10.0f;
100             //assign coords to the triangle using base coord plus variance
101             tri_array[i].p1.x = x+rand()%TRIANGLE_SZ; tri_array[i].p1.y = y+rand()%
                    TRIANGLE_SZ, tri_array[i].p1.z = z+rand()%TRIANGLE_SZ;
102             tri_array[i].p2.x = x+rand()%TRIANGLE_SZ; tri_array[i].p2.y = y+rand()%
                    TRIANGLE_SZ, tri_array[i].p2.z = z+rand()%TRIANGLE_SZ;
103             tri_array[i].p3.x = x+rand()%TRIANGLE_SZ; tri_array[i].p3.y = y+rand()%
                    TRIANGLE_SZ, tri_array[i].p3.z = z+rand()%TRIANGLE_SZ;
104         }
105     }
106
107     last_size=size;
108     nb_calls++;
109
110     return size;
```

```
111 }
112
113 void free_memory(){
114     free(tri_array);
115     free(tri_result_array);
116 }
```

viewer.h

```
1  /**
2   * \file        viewer.h
3   * \author      Romain Maffina
4   * \version     1.0
5   * \brief       Minimalistic openGL viewer for triangle-triangle collision tester.
6   */
7
8  #ifndef OPENGL_TOOLS_H_
9  #define OPENGL_TOOLS_H_
10
11 #include <stdint.h>
12
13 extern void start_viewer(int argc, char** argv, uint32_t nb_tri);
14
15
16 #endif /* OPENGL_TOOLS_H_ */
```

viewer.c

```
1  /**
2   * \file        viewer.c
3   * \author      Romain Maffina
4   * \version     1.0
5   * \brief       Minimalistic openGL viewer for triangle-triangle collision tester.
6   */
7
8  #include <stdio.h>
9  #include <GL/freeglut.h>
10 #include "viewer.h"
11 #include "test-tools.h"
12 #include "detect-tri2tris-intersection.h"
13 #include "geometric-tools.h"
14
15 /**
16  * @struct color_t
17  * @brief Represent a color in a RGB way.
18  */
19 typedef struct{
20     uint8_t R,G,B;
21 } color_t;
22
23 void draw_axis(void);
24 void opengl_draw_plane_from_triangle(triangle_t t, color_t c);
25 void opengl_draw_triangle(triangle_t t, color_t c, char* str);
26 void opengl_draw_line(line_t l, color_t c, uint8_t coeff);
27 void opengl_draw_segment(coord_t pt1, coord_t pt2);
28 void opengl_text_overlay(float x, float y, char *str);
29 void opengl_text_label(float x, float y, float z, void *font, const char *str);
30
```

```
31 const color_t COLOR_BLUE =      {0,0,255};
32 const color_t COLOR_GREY =      {128,128,128};
33 const color_t COLOR_ORANGE =    {255,160,0};
34 const color_t COLOR_RED =       {255,0,0};
35 const color_t COLOR_BLACK =     {0,0,0};
36
37 uint8_t cfg_show_help=0;
38 uint8_t cfg_show_origin=0;
39 uint8_t cfg_show_tri_coord=0;
40 uint8_t cfg_show_tri_id=0;
41 uint8_t cfg_show_planes=0;
42 uint8_t cfg_special_cases=0;
43 uint8_t cfg_use_cuda=0;
44
45 static uint32_t nb_triangles;
46
47 float angle_around_Z=0;
48 float zoom = 0.15f;
49
50 /**
51  * Draw at point (0;0;0) the 3 axis X,Y and Z.
52  */
53 void draw_axis(void){
54     glLineWidth(1);
55     glPushMatrix();
56     //glScalef(1,1,1);
57     glBegin(GL_LINES);
58     glColor3ub(0,0,255);
59     glVertex3f(0,0,0);
60     glVertex3f(1,0,0);
61     glColor3ub(0,255,0);
62     glVertex3f(0,0,0);
63     glVertex3f(0,1,0);
64     glColor3ub(255,0,0);
65     glVertex3f(0,0,0);
66     glVertex3f(0,0,1);
67     glEnd();
68     glPopMatrix();
69 }
70
71 /**
72  * Draw a corresponding triangle's plane.
73  *
74  * @param t the triangle
75  * @param c the color of the plane
76  */
77 void opengl_draw_plane_from_triangle(triangle_t t, color_t c){
78     vector_t v1, v2;
79     vector_from_points(t.p1, t.p2, &v1);
80     vector_from_points(t.p1, t.p3, &v2);
81
82     glBegin(GL_QUADS);
83     glColor4ub(c.R,c.G,c.B, 32);
84         glVertex3f(t.p1.x + 10*(v1.x + v2.x), t.p1.y + 10*(v1.y + v2.y), t.p1.z + 10*(v1.z
                + v2.z));
85         glVertex3f(t.p1.x + 10*(v1.x - v2.x), t.p1.y + 10*(v1.y - v2.y), t.p1.z + 10*(v1.z
                - v2.z));
86         glVertex3f(t.p1.x + 10*(-v1.x - v2.x), t.p1.y + 10*(-v1.y - v2.y), t.p1.z + 10*(-v1
                .z - v2.z));
```

```
87          glVertex3f(t.p1.x + 10*(-v1.x + v2.x), t.p1.y + 10*(-v1.y + v2.y), t.p1.z + 10*(-v1
                .z + v2.z));
88      glEnd();
89  }
90
91  /**
92   * Draw a triangle.
93   *
94   * @param t the triangle
95   * @param c the triangle's color
96   * @param *str the triangle name
97   */
98  void opengl_draw_triangle(triangle_t t, color_t c, char* str){
99      char str_coord[100];
100     coord_t mid;
101
102     glBegin(GL_TRIANGLES);
103         glColor3ub(c.R,c.G,c.B);
104         glVertex3d(t.p1.x, t.p1.y, t.p1.z);
105         glVertex3d(t.p2.x, t.p2.y, t.p2.z);
106         glVertex3d(t.p3.x, t.p3.y, t.p3.z);
107     glEnd();
108
109     if(cfg_show_tri_coord){
110         sprintf(str_coord, "(%.2f;%.2f;%.2f)", t.p1.x, t.p1.y, t.p1.z);
111         opengl_text_label(t.p1.x, t.p1.y, t.p1.z, GLUT_BITMAP_HELVETICA_10, str_coord);
112         sprintf(str_coord, "(%.2f;%.2f;%.2f)", t.p2.x, t.p2.y, t.p2.z);
113         opengl_text_label(t.p2.x, t.p2.y, t.p2.z, GLUT_BITMAP_HELVETICA_10, str_coord);
114         sprintf(str_coord, "(%.2f;%.2f;%.2f)", t.p3.x, t.p3.y, t.p3.z);
115         opengl_text_label(t.p3.x, t.p3.y, t.p3.z, GLUT_BITMAP_HELVETICA_10, str_coord);
116     }
117
118     if(cfg_show_tri_id && str!=NULL){
119         glColor3ub((c.R-100)%255,(c.G-100)%255,(c.B-100)%255);
120         mid.x = (t.p1.x + t.p2.x + t.p3.x)/3;
121         mid.y = (t.p1.y + t.p2.y + t.p3.y)/3;
122         mid.z = (t.p1.z + t.p2.z + t.p3.z)/3;
123         opengl_text_label(mid.x, mid.y, mid.z, GLUT_BITMAP_9_BY_15, str);
124     }
125
126     if(cfg_show_planes)
127         opengl_draw_plane_from_triangle(t, c);
128 }
129
130 /**
131  * Draw a line.
132  *
133  * @param l the line
134  * @param c the color of the line
135  * @param coeff size of the line
136  */
137 void opengl_draw_line(line_t l, color_t c, uint8_t coeff){
138
139     coord_t pt1, pt2;
140     vector_t d;
141
142     d.x = l.direction.x;
143     d.y = l.direction.y;
144     d.z = l.direction.z;
```

```
145
146     //d = vector_by_scalar(l.direction, coeff);
147     d.x *= coeff;
148     d.y *= coeff;
149     d.z *= coeff;
150
151     add3d(pt1, l.p, d);
152     sub3d(pt2, l.p, d);
153
154     glBegin(GL_LINES);
155             glColor3ub(c.R,c.G,c.B);
156             glVertex3d(pt1.x, pt1.y, pt1.z);
157             glVertex3d(pt2.x, pt2.y, pt2.z);
158     glEnd();
159 }
160
161 /**
162  * Draw a segment.
163  *
164  * @param pt1 the first point
165  * @param pt2 the second point
166  */
167 void opengl_draw_segment(coord_t pt1, coord_t pt2){
168
169     glPointSize(7);
170     glBegin(GL_POINTS);
171         glColor3ub(255,0,0);
172         glVertex3d(pt1.x, pt1.y, pt1.z);
173         glVertex3d(pt2.x, pt2.y, pt2.z);
174     glEnd();
175     glBegin(GL_LINES);
176         glColor3ub(255,0,0);
177         glVertex3d(pt1.x, pt1.y, pt1.z);
178         glVertex3d(pt2.x, pt2.y, pt2.z);
179     glEnd();
180 }
181
182 /**
183  * Print an overlay string to the screen.
184  *
185  * @param x the x coordinate
186  * @param y the y coordinate
187  * @param *str the string to print
188  *
189  * @see From http://www.lighthouse3d.com
190  */
191 void opengl_text_overlay(float x, float y, char *str) {
192     char *c;
193
194     // switch to projection mode
195     glMatrixMode(GL_PROJECTION);
196
197     // save previous matrix which contains the
198     //settings for the perspective projection
199     glPushMatrix();
200
201     // reset matrix
202     glLoadIdentity();
203
```

```
204     // set a 2D orthographic projection
205     gluOrtho2D(0, glutGet(GLUT_WINDOW_WIDTH), glutGet(GLUT_WINDOW_HEIGHT), 0);
206
207     // switch back to modelview mode
208     glMatrixMode(GL_MODELVIEW);
209
210     //set_orthographic_projection(glutGet(GLUT_SCREEN_WIDTH),glutGet(GLUT_SCREEN_HEIGHT));
211     glPushMatrix();
212     glLoadIdentity();
213
214     glColor3ub(255,0,0);
215     glRasterPos3f(x, y,0);
216     for (c=str; *c != '\0'; c++) {
217         glutBitmapCharacter(GLUT_BITMAP_9_BY_15, *c);
218     }
219     //restore_perspective_projection();
220
221     glMatrixMode(GL_PROJECTION);
222     // restore previous projection matrix
223     glPopMatrix();
224
225     // get back to modelview mode
226     glMatrixMode(GL_MODELVIEW);
227 }
228
229 /**
230  * Label a point in the 3D space.
231  *
232  * @param x the x coordinate
233  * @param y the y coordinate
234  * @param z the z coordinate
235  * @param *font the font to use
236  * @param *str the string to use
237  *
238  * @see GLUT_BITMAP_ for available fonts
239  * @see From http://www.lighthouse3d.com
240  */
241 void opengl_text_label(float x, float y, float z, void *font, const char *str)
242 {
243     glDisable(GL_TEXTURE_2D);
244     glDisable(GL_DEPTH_TEST);
245     glRasterPos3f(x, y,z);
246
247     while(*str)
248     {
249         glutBitmapCharacter(font, *str);
250         str++;
251     }
252
253     glEnable(GL_TEXTURE_2D);
254     glEnable(GL_DEPTH_TEST);
255 }
256
257
258 void new_random_triangles(){
259     nb_triangles = tri_generator(nb_triangles);
260     if(cfg_use_cuda)
261         tri2tris_intersection_detect_gpu(ref, tri_array, nb_triangles, tri_result_array);
262     else
```

```
263         tri2tris_intersection_detect_cpu(ref, tri_array, nb_triangles, tri_result_array);
264 }
265
266 void new_specials_triangles(){
267     nb_triangles = tri_generator(0);
268     if(cfg_use_cuda)
269         tri2tris_intersection_detect_gpu(ref, tri_array, nb_triangles, tri_result_array);
270     else
271         tri2tris_intersection_detect_cpu(ref, tri_array, nb_triangles, tri_result_array);
272 }
273
274 // ------------------------------------
275 //             HANDLERS
276 // ------------------------------------
277
278 void keyboard_handle(unsigned char key, int x, int y){
279     static uint32_t previous_size;
280     switch(key){
281         case 'g':
282             cfg_use_cuda=cfg_use_cuda==0?1:0;
283             break;
284         case 0x20:
285             if(cfg_special_cases==0)new_random_triangles();
286             break;
287         case 'h':
288             cfg_show_help=cfg_show_help==0?1:0;
289             break;
290         case 'c':
291             cfg_show_tri_coord=cfg_show_tri_coord==0?1:0;
292             break;
293         case 'i':
294             cfg_show_tri_id=cfg_show_tri_id==0?1:0;
295             break;
296         case 'o':
297             cfg_show_origin=cfg_show_origin==0?1:0;
298             break;
299         case 'p':
300             cfg_show_planes=cfg_show_planes==0?1:0;
301             break;
302         case 't':
303             cfg_special_cases=cfg_special_cases==0?1:0;
304             if(cfg_special_cases){
305                 previous_size=nb_triangles;
306                 new_specials_triangles();
307             }else{
308                 nb_triangles=previous_size;
309                 new_random_triangles();
310             }
311             break;
312         case '+':
313             if(!cfg_special_cases){
314                 nb_triangles++;
315                 new_random_triangles();
316             }
317             break;
318         case '-':
319             if(!cfg_special_cases && nb_triangles>1){
320                 nb_triangles--;
321                 new_random_triangles();
```

```
322             }
323             break;
324 
325         /* Move ref triangle */
326         case 'e':
327             ref.p1.z += 0.0625; ref.p2.z += 0.0625; ref.p3.z += 0.0625;
328             break;
329         case 'q':
330             ref.p1.z -= 0.0625; ref.p2.z -= 0.0625; ref.p3.z -= 0.0625;
331             break;
332         case 's':
333             ref.p1.x += 0.0625; ref.p2.x += 0.0625; ref.p3.x += 0.0625;
334             break;
335         case 'w':
336             ref.p1.x -= 0.0625; ref.p2.x -= 0.0625; ref.p3.x -= 0.0625;
337             break;
338         case 'a':
339             ref.p1.y -= 0.0625; ref.p2.y -= 0.0625; ref.p3.y -= 0.0625;
340             break;
341         case 'd':
342             ref.p1.y += 0.0625; ref.p2.y += 0.0625; ref.p3.y += 0.0625;
343             break;
344     }
345 
346     if(cfg_use_cuda)
347         tri2tris_intersection_detect_gpu(ref, tri_array, nb_triangles, tri_result_array);
348     else
349         tri2tris_intersection_detect_cpu(ref, tri_array, nb_triangles, tri_result_array);
350 
351     glutPostRedisplay();
352 }
353 
354 void special_key_handle(int key, int x, int y){
355 
356     switch(key){
357     case GLUT_KEY_UP:
358         zoom += 0.03f;
359         break;
360     case GLUT_KEY_DOWN:
361         zoom -= 0.03f;
362         break;
363     case GLUT_KEY_LEFT:
364         angle_around_Z -= 0.1f;
365         break;
366     case GLUT_KEY_RIGHT:
367         angle_around_Z += 0.1f;
368         break;
369     }
370 
371     glutPostRedisplay();
372 }
373 
374 void mouse_motion_handle(int x,int y){
375     static int last_x;
376 
377     angle_around_Z += last_x-x<0 ? 3 : -3;
378     last_x = x;
379 
380     glutPostRedisplay();
```

```
381 }
382
383 void resize_handle(int w, int h) {
384     //Lighthouse
385
386     float ratio;
387     // Prevent a divide by zero, when window is too short
388     // (you cant make a window of zero width).
389     if(h == 0)
390         h = 1;
391     ratio = 1.0f* w / h;
392
393     // Use the Projection Matrix
394     glMatrixMode(GL_PROJECTION);
395
396         // Reset Matrix
397     glLoadIdentity();
398
399     // Set the viewport to be the entire window
400     glViewport(0, 0, w, h);
401
402     // Set the correct perspective.
403     gluPerspective(45,ratio,1,1000);
404
405     // Get Back to the Modelview
406     glMatrixMode(GL_MODELVIEW);
407 }
408
409 void display_handle(){
410     unsigned int i;
411     char str[100];
412     color_t c;
413
414     glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
415     glMatrixMode( GL_MODELVIEW );
416     glLoadIdentity( );
417     glEnable(GL_BLEND) ;
418     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) ;
419
420     gluLookAt(3,0,2,0,0,0.1,0,0,1);
421
422     glScalef(zoom, zoom, zoom);
423     glRotatef(angle_around_Z,0,0,1);
424
425     //We draw the ref triangle and his plane
426     opengl_draw_triangle(ref, COLOR_BLUE, "");
427
428     //We draw all the triangles in the array
429     for(i=0;i<nb_triangles;i++){
430         if(tri_result_array[i]) c=COLOR_ORANGE; else c=COLOR_GREY;
431         sprintf(str, "%d", i);
432         opengl_draw_triangle(tri_array[i], c, str);
433     }
434
435     //We draw  the axis in 0,0,0
436     if(cfg_show_origin)
437         draw_axis();
438
439     opengl_text_overlay(5, 15,"[h] Show help");
```

```
440     if(cfg_show_help){
441         opengl_text_overlay(5, 30,"[arrows] Zoom/Rotate around");
442         opengl_text_overlay(5, 45,"[wasdqe] Move the blue triangle");
443         opengl_text_overlay(5, 60,"[o] Show origin axis");
444         opengl_text_overlay(5, 75,"[c] Show triangle coord");
445         opengl_text_overlay(5, 90,"[i] Show triangle id");
446         opengl_text_overlay(5, 105,"[t] Switch to special cases mode");
447         opengl_text_overlay(5, 120,"[g] Switch between CPU and GPU");
448 
449         if(!cfg_special_cases){
450         opengl_text_overlay(5, 135,"[+] Add a triangle");
451         opengl_text_overlay(5, 150,"[-] Remove a triangle");
452         opengl_text_overlay(5, 165,"[space] New random triangles");
453         }
454     }
455     sprintf(str, "Computed by %s", cfg_use_cuda?"GPU using CUDA":"CPU");
456     opengl_text_overlay(5, glutGet(GLUT_WINDOW_HEIGHT)-10, str);
457 
458     //force display
459     glFlush();
460 }
461 
462 // ------------------------------------
463 //             INIT
464 // ------------------------------------
465 
466 void start_viewer(int argc, char** argv, uint32_t nb_tri){
467 
468     nb_triangles = nb_tri;
469 
470     new_random_triangles();
471 
472     /* Setting up configuration */
473     config_set( COMPUTE_ALL_INTERSECTION | VERBOSE_MODE | NO_WDDM_TDR_WORKAROUND);
474 
475     printf("Starting OpenGL viewer...");
476     fflush(stdout);
477 
478     /* OPENGL */
479     glutInit(&argc, argv);
480     glutInitWindowSize(glutGet(GLUT_SCREEN_WIDTH)/2,glutGet(GLUT_SCREEN_HEIGHT)/2);
481     glutInitWindowPosition(
482             (glutGet(GLUT_SCREEN_WIDTH)-glutGet(GLUT_INIT_WINDOW_WIDTH))/2,
483             (glutGet(GLUT_SCREEN_HEIGHT)-glutGet(GLUT_INIT_WINDOW_HEIGHT))/2);
484     glutCreateWindow("Triangles collision tester - viewer");
485 
486     glEnable(GL_DEPTH_TEST); //zdepth
487     glClearColor(1.0f, 1.0f, 1.0f, 0.0f); //white background
488 
489     // callbacks
490     glutKeyboardFunc(keyboard_handle);
491     glutSpecialFunc(special_key_handle);
492     glutDisplayFunc(display_handle);
493     glutMotionFunc(mouse_motion_handle);
494     glutReshapeFunc(resize_handle);
495 
496     glutMainLoop();
497 }
```

console.h

```
/**
 * \file        console.h
 * \author      Romain Maffina
 * \version     1.0
 * \brief       Console testbench for triangle-triangle collision tester.
 */

#ifndef CONSOLE_H_
#define CONSOLE_H_

#include <stdint.h>

int start_benchmark(int argc, char** argv, uint32_t nb_triangles);

#endif /* CONSOLE_H_ */
```

console.c

```
/**
 * \file        console.c
 * \author      Romain Maffina
 * \version     1.0
 * \brief       Console testbench for triangle-triangle collision tester.
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "console.h"
#include "test-tools.h"
#include "geometric-tools.h"
#include "detect-tri2tris-intersection.h"

void print_first_results(uint8_t results[], int size);
void print_last_results(uint8_t results[], int size);


void print_first_results(uint8_t results[], int size){
    int i, count;

    count=size<25?size:25;
    printf("first results : ");
    for(i=0;i<count;i++)
         printf("%d ", results[i]);
    printf("\n");
}
void print_last_results(uint8_t results[], int size){
    int i, count;

    count=size<25?size:25;
    printf("last results  : ");
    for(i=count;i>0;i--)
         printf("%d ", results[size-i]);
    printf("\n");
}
```

```
39
40 int start_benchmark(int argc, char** argv, uint32_t nb_triangles){
41     uint8_t *tri_result_array_gpu = NULL;
42     triangle_t *tri_mismatch_array = NULL;
43     uint8_t *tri_mismatch_result_array = NULL;
44
45     uint32_t nb_triangles_last = nb_triangles;
46
47     printf("Welcome to this CPU and GPU benchmark of a triangle-triangle collision tester.\
           n\n");
48
49     /* Generate random triangles and init result array */
50     printf("Generating %d random triangles...", nb_triangles);
51     fflush(stdout);
52
53     if( (nb_triangles=tri_generator(nb_triangles))<0 ){
54         printf("\n\nERROR : Host memory allocation failed!\n");
55         return -1;
56     }
57     if(nb_triangles_last==nb_triangles)
58         printf(" done.\n\n\n");
59     else
60         printf(" only %d triangles have been generated.\n\n\n", nb_triangles);
61     fflush(stdout);
62
63     tri_result_array_gpu = (uint8_t*)calloc(nb_triangles,sizeof(uint8_t));
64
65     /* Setting up configuration */
66     config_set( COMPUTE_ALL_INTERSECTION | VERBOSE_MODE );
67
68     /* Collisions computed with GPU */
69     printf("Computing collisions with GPU (CUDA)...");
70     chrono();
71     if(tri2tris_intersection_detect_gpu(ref, tri_array, nb_triangles, tri_result_array_gpu)
           != -1){
72         printf("computed in %d ms.\n", chrono() );
73
74         print_first_results(tri_result_array_gpu, nb_triangles);
75         print_last_results(tri_result_array_gpu, nb_triangles);
76     }
77     else{
78         chrono();
79         printf("failed! An error occured.\n", chrono() );
80         return -1;
81     }
82
83     fflush(stdout);
84
85     printf("\n\n");
86
87     /* Collisions computed with CPU */
88     printf("Computing collisions with CPU...");
89     fflush(stdout);
90     chrono();
91     if(tri2tris_intersection_detect_cpu(ref, tri_array, nb_triangles, tri_result_array) !=
           -1){
92         printf("computed in %d ms.\n", chrono() );
93
94         print_first_results(tri_result_array, nb_triangles);
```

```
95          print_last_results(tri_result_array, nb_triangles);
96      }
97      else{
98          chrono();
99          printf("failed! An error occured.\n", chrono() );
100         return -1;
101     }
102     printf("\n\n");
103     fflush(stdout);
104
105     free_memory();
106
107     return 0;
108 }
```