3rd Workshop in Virtual Reality Interactions and Physical Simulation "VRIPHYS" (2006)
C. Mendoza, I. Navazo (Editors)

Collision Detection between a Complex Solid and a Particle Cloud assisted by Programmable GPU

Juan J. Jiménez, Carlos J. Ogáyar, Rafael J. Segura & Francisco R. Feito [1]

[1]Departamento de Informática de la Universidad de Jaén. Campus las Lagunillas s/n. 23071 Jaén (Spain)
{juanjo, cogayar, rsegura, ffeito}@ujaen.es

Abstract

In this work the problem of collision detection (CD) between a Complex Solid and a Particle cloud with autonomous movement is studied. In order to do this, some algorithms and data have been adapted to suit new extensions of the new generations of programmable graphics cards. These types of graphics cards allow more flexible programming in order to solve problems not-related to visualization process. We use a representation based on simplicial coverings as well as a structure named Tetra-Tree in order to represent and classify the simplices of complex objects. With this type of representation the operations carried out in CD are more robust and efficient than those used in classic algorithms, so it is not necessary to decompose the complex object into more simple pieces. We also propose some implementation alternatives and give a study of their performance.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Geometric algorithms, languages and systems; Curve, surface, solid, and object representations. I.3.6 [Computer Graphics]: Methodology and Techniques; Graphics data structures and data types.

Additional Key words and Phrases: barycentric coordinates, collision detection, covering by means of tetrahedra, programmable GPU, tetra-tree.

1. Introduction

The possibility of programming graphics cards arose several years ago. The number of instructions and the limitations on program size and the number of control structures available have restricted the use of such programmable graphics cards for rendering.

However, the appearance of the Geforce 6 series of NVidia supposed a great advance due to the elimination of some restrictions. These include:

- The elimination of the limitations on the number of operations per program (vertex and fragment programs).
- The incorporation of the iteration control structure in the fragment programs.
- The possibility of accessing textures from the vertex programs.

To these improvements we must add the greater calculating power of the new graphics processors and the high bandwidth of the memory transferences within the graphics card [KF05].

These and other developments have led many researchers towards the possibility of using the GPU (Graphics Processing Unit) as one more general processor, so more and more solutions to general problems are being achieved by means of the use of graphics processors.

The problem here, however is that graphics cards and programming languages are designed more for rendering than for general purpose computing, and so a (usually expensive) conversion of the initial problem to a graphics one becomes necessary. And even when an adaptation of an algorithm to the GPU is found, the change in the approach and in the operation mode can cause the CPU solution to be more efficient, although still less powerful.

A particle system can be defined as a set of particles in which their dimension is not important, only their position, mass, movement and behaviour toward diverse phenomena

[Ebe04]. The movement of these particles can be chaotic or can be induced by certain physical laws like gravity, pressure, etc.

For most applications, a particle system can be seen in two ways: as a representation of a point in a continuous medium, or as a description of the dynamic state of a solid.

In the context of modelling, particles are used fundamentally in the first way, so that a particle system describes a continuous medium in a non-continuous (discreet) way [TF88] [Sta99] [JP02]. Particles can also be used for implicit surfaces sampling [WH94].

In this work the problem of collision detection between a particle system and a solid has been solved by using a specific representation of the solid based on simplicial coverings [FT97a] as well as a new space decomposition named tetra-tree [JFSO06] that uses tetra-cones.

The main objective of this work consists of verifying the possibility of implementation of geometric algorithms for collision detection in the GPU, and the posterior comparison with the implementation in the CPU. In order to do this, we have extended and adapted a point in solid inclusion algorithm so it can be used in the collision detection between a particle system and a solid in the GPU.

The paper is organized in the following way. Firstly, we will review some methods used by other authors for collision detection using the GPU and the CPU. Next, we will describe the foundations and the algorithm used in collision detection. In the Implementation section we will explain different methods of using the GPU to solve this problem. Later, we will compare the solutions developed by using the GPU with the CPU-based solution. Finally, in the Conclusions section we will summarize the most important achievements of this method as well as the future work to be undertaken.

2. Previous work

Most collision detection techniques for particle systems using graphics hardware are image-based because they are oriented towards the visualization of particles. Thus, Kolb [KLR04] proposes an algorithm based on depth maps which represent the solid boundary, distance values and normal vectors are stored in this data structure. These maps are codified on 8-bits textures.

Knott [Kno03] proposes a method for collision detection with generic particle systems, based on the use of graphics hardware, although he uses only one stage in the GPU pipeline and restricts the number of particles to the number of cycles supported by the GPU.

Baciu [BW03] suggests an image-based collision detection method between solids. The Z-buffer algorithm is modified so that the depth values are stored in a matrix structure, needing more than one iteration for the collision determination.

Govindaraju [GLM05] presents a collision detection algorithm based on the occlusion extensions of new graphics cards.

Jiménez [JFSO06] proposes a robust method for collision detection between a particle and a complex solid. In order to do this they decompose the solid (represented by means of simplicial coverings) using a tetrahedra hierarchy and use barycentric coordinates for the particle-in-solid inclusion determination. The authors affirm that a GPU implementation is possible, but they do not provide information on how this could be achieved.

Additional information concerning different methods and techniques for collision detection can be consulted in [TKH*04] [LG98] [JTT01].

3. Collision detection

The collision detection between a particle and a solid goes through several stages:

- In pre-processing, a simplicial covering of the polyhedron is generated in linear time and by means of tetrahedra. This is followed by the classification in a tetra-tree of these tetrahedra of the covering. A tetra-tree [JFSO06] is a spatial decomposition structure based on tetra-cones.
- In the first stage, particles are classified in the tetra-tree (obtaining the greater depth tetra-cones where particles are located). A particle is discarded if it is outside the bounding tetrahedron of the tetra-cone where it is situated.
- In the second stage, particles not-discarded in the previous stage are classified with regard to tetrahedra classified in the corresponding tetra-cone. The inclusion of the particle in this part of the polyhedron is determined and the result is extended to the complete polyhedron.

The first and second stages are iterated for each movement of a particle. In case of a particle cloud, these stages are applied to each particle.

3.1. Pre-processing

From a solid representation by means of a simplicial covering [FT97a] we made a hierarchical space decomposition based on a tetra-tree [JFSO06].

The simplicial covering of a solid allows the decomposition of a complex solid (with non-convex faces, holes, etc.) into more simple pieces like tetrahedra each with an associated sign. This decomposition will allow us to perform the calculations involved in collision detection with the tetrahedra of the covering instead of performing these calculations with the solid. These calculations between tetrahedra

are simpler and the algorithms obtained are more robust and efficient [FT97b].

A tetra-tree is a hierarchical data structure formed by eight initial tetra-cones. Each tetra-cone S_0 (with an associated tetrahedron $T(P_0P_1P_2P_3)$) is divided into four sub-tetra-cones $(S_{00}, S_{01}, S_{02}, S_{03})$, Figure 1.a. These eight initial tetra-cones cover the whole space without overlapping. Each sub-tetra-cone is recursively divided into four tetra-cones in the same way (Figure 1.b).

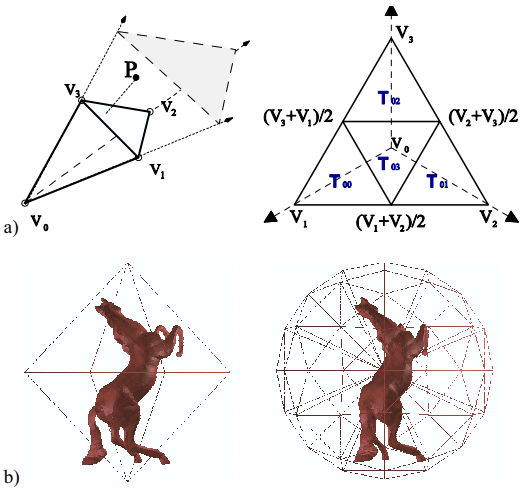


Figure 1: *A tetra-cone and sub-tetra-cones (a). A tetra-tree of level 1 and level 3 (b).*

A polyhedron is decomposed by a tetra-tree with the original vertex situated on the centroid of the solid, so tetrahedra of the covering of the polyhedron are classified in the tetra-cones of this tetra-tree. The tetra-tree depth can be fixed or a tetra-cone can not be subdivided when the number of tetrahedra classified in that tetra-cone is smaller than a given value.

We use this data structure because it fits to the solid more precisely than other structures like an octree, its calculation being faster [JFSO06]. Furthermore, we can use the barycentric coordinates of a point with regard to a tetrahedron for calculating the inclusion of a point in a tetrahedron or a tetra-cone, as well as for the intersection between tetrahedra and tetra-cones, needed for tetrahedra classification.

3.2. First stage: classification and prune

In order to determine whether a collision between a particle and a polyhedron takes place or not, this particle is first recursively classified within the tetra-tree, obtaining a max depth tetra-cone where the particle is located.

This information is stored between frames, so coherence in the movement of particles is used. In the next frame, it is probable that the particle will be in the same tetra-cone as in the previous frame, so we check first if this occurs.

Each tetra-cone has an associated bounding tetrahedron, so if a point is not inside this tetrahedron, the point will not be inside any tetrahedra of the covering classified in this tetra-cone, and this particle is discarded.

3.3. Second stage: inclusion in a tetra-cone field

Particles that passed the first stage must be classified with regard to the tetrahedra of a tetra-cone.

In order to detect the inclusion of a particle in these tetrahedra, we use the inclusion algorithm developed by [FT97b]. This algorithm has been modified and adapted, so barycentric coordinates are used in order not only to determine whether a point is inside a tetrahedron, but to determine the position of the particle (if it is in a vertex, edge, face or in the interior of the tetrahedron).

The barycentric coordinates of a point with regard a tetrahedron are well known. Given four points that form a tetrahedron, a point $P \in \Re^3$ satisfies that $P = \alpha A + \beta B + \gamma C + \delta D$, being A, B, C, D the vertices of the tetrahedron and $\alpha, \beta, \gamma, \delta \in \Re^3$ four unique values so that $\alpha + \beta + \gamma + \delta = 1$. The numbers $\alpha, \beta, \gamma, \delta$ are the barycentric coordinates of P with regard to the tetrahedron $ABCD$, and can be calculated as follows:

$$\alpha = |PBCD|/|ABCD|$$
$$\beta = |PCBA|/|ABCD|$$
$$\gamma = |PABD|/|ABCD|$$
$$\delta = |PADC|/|ABCD|$$

We can determine the inclusion of a point in a tetrahedron by using the barycentric coordinates. The point is inside the tetrahedron iff [Bad90]:

$$0 \leq \alpha, \beta, \gamma, \delta \leq 1$$

Additionally, the barycentric coordinates allow us to determine the position of the point in the tetrahedron, that is, if a point is on a concrete vertex, edge or face [JFSO06], see Figure 2.

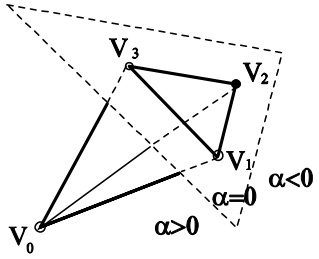


Figure 2: *Geometric interpretation of the barycentric coordinates of a point.*

The inclusion algorithm allows us to determine if the particle is in the part of the solid classified in a tetra-cone by obtaining the inclusion of the particle in each tetrahedron of this tetra-cone. In order to do this, it performs a signed summation with the tetrahedra in which this particle is included. A particle will be inside the polyhedron if this sum is equal to one.

There are special cases when a feature of a tetrahedron (vertex, edge or face) is shared with other tetrahedra. Given a tetrahedron with ordered vertices as we can see in Figure 3; we use a variable for accumulating some values:

- If the particle is on edge V_2V_3 then the particle is inside the polyhedron and it is not necessary to check additional tetrahedra.
- If the particle is strictly inside a tetrahedron or in a non-shared face $(V_1V_2V_3)$, we accumulate the value $+1$ or -1 according to the sign of the tetrahedron.
- If the particle is inside a face shared by two tetrahedra or on an edge of the 2D covering of a face of the polyhedron (faces and edges shared only by two tetrahedra), we accumulate the values $+1/2$ or $-1/2$ according to the sign of the tetrahedron.
- In other cases, the particle is on an edge or vertex shared by an indeterminate number of tetrahedra. It is necessary to accumulate $+1$ only one time per positive feature and -1 one time per negative feature. Thus, we create two sets of features, one for positive features and one for negative ones. When a particle is on one feature, the set of features is consulted according to the sign of the tetrahedron, and a value $+1$ or -1 is accumulated once. If the feature of a given set has been accumulated previously, no further operation is performed.

The algorithm that shows the second stage of the collision detection between a particle and a tetra-cone can be seen in Algorithm 1.

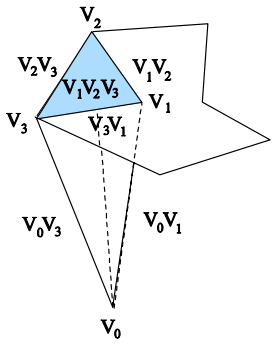


Figure 3: *Features of a tetrahedron of the simplicial covering of a face.*

1. Initialize positive and negative features sets to empty. Initialize $sum = 0$.
2. For each k tetrahedron in the Tetra-Cone:
   a. If the point is on edge V_2V_3 or on vertices V_2 or V_3: return IN.
   b. If the point is strictly inside the tetrahedron or on the outer face: $sum+ = 2 \cdot sign(tetrahedron)$
   c. If the point is on an inner face or on edge V_1V_2 or edge V_3V_1: $sum+ = sign(tetrahedron)$
   d. In other cases (inner edge or vertex V_0 or vertex V_1)
      i. If $sign(tetrahedron) = 1$: Check whether the related feature is in the set of positive features, if not: $sum+ = 2$ and the feature is included in the set of positive features.
      ii. If $sign(tetrahedron) = -1$: Check whether the related feature is in the set of negative features, if not: $sum- = 2$ and the feature is included in the set of negative features.
3. If $sum = 2$ return IN; otherwise, return OUT.

Algorithm 1: *Second stage of the collision detection between a particle and a tetra-cone. Calculations are multiplied by two in order to use integer arithmetic.*

4. Implementation

Nowadays it is possible to program graphics cards so that operations related to rendering are programmed in vertex shaders and fragment shaders. These programs can be implemented by means of a high-level language specialized and similar to C, denominated Cg. This language contains a series of specific and optimized instructions for these tasks. These instructions are suitable for geometric treatment, so we think they are appropriate for some geometric problems in which some tasks can be split and performed in parallel. These capacities lead us to use this language in the implementation of collision detection between a particle cloud and a polyhedron in the GPU.

4.1. Data codification

In order to perform this collision detection it is necessary to send some information about the polyhedron and particles to the GPU.

The idea consists of providing the GPU with each particle on which collision detection will be performed, as well as information about the polyhedron geometry and tetrahedra of the tetra-tree.

The result from the GPU is obtained in the frame-buffer, codified in RGBA components which represents the pixel colour. For this reason, information of the 2D coordinate of a pixel in the frame-buffer will be sent. This information represents the position of the result of a particle/polyhedron collision detection. Therefore, variable information will be sent to the GPU for each particle. This consists of the frame-buffer position of the result, the position of the particle and

the tetra-cone in which the particle was located in the previous frame.

As particles move from frame to frame, it is more appropriate to store this information in data structures which allow modifications between frames. With OpenGL, the position of the result and the previous tetra-cone are stored in a Vertex-Array, and the particles positions in Normal-Array, so we can use the glDrawArrays function from OpenGL.

The general concept of this implementation consists of sending particles to the vertex shader. Each particle causes only one program execution in the fragment shader, in order to obtain the collision result of this particle with the part of the polyhedron classified in a tetra-cone.

The way to send information on the polyhedron and tetra-tree consists of codifying this information in textures. For each one two textures are used, one storing indices and the other vertices.

We use a texture of vertices for the codification of the geometry of the polyhedron: Vertex[Rows,Columns,3]; so in Vertex[i,j] three coordinates of a vertex are stored. A texture of indices is also used: Index[Rows,Columns,4]; so in Index[i,j] three indices of three external vertices (V_1, V_2, V_3) of a tetrahedron of the covering of the polyhedron and the sign of this tetrahedron in the fourth component are stored. The original vertex of the covering (V_0) is sent to GPU as a shared uniform parameter.

In each row of the Index texture, the indices of tetrahedra of the covering classified in a tetra-cone are stored. Each row represents a tetra-cone with the tetrahedra classified in it. The first column of a row is reserved and represents the number of tetrahedra classified in that tetra-cone.

The tetra-tree has been stored in a similar way. We use a vertex texture for the vertices which represents the tetra-tree: VertexTT[Rows,Columns,3]; and an index texture: IndexTT[Levels,Tetra-Cone,3]; in this, each row represents a level of the tetra-tree, that is to say, stores the tetra-cones of a level. Thus, a position in the texture, IndexTT[3,48] for example, stores three indices of the vertices of the tetrahedron of tetra-cone 48 of level 3. Tetrahedra that represent tetra-cones are bounding tetrahedra of the tetrahedra classified in that tetra-cone.

4.2. Vertex & Fragment Shader implementation

The GPU programmable pipeline, divided into two programmable stages, vertex shader and fragment shader, along with the nature of the previous algorithm, lead us to implement, in a natural way, the collision between a particle cloud and a solid by implementing each stage of the proposed algorithm in each programmable stage in the GPU.

The vertex program classifies particles in a tetra-tree and obtains the tetra-cone in which the particle is located (if this particle is not inside the same tetra-cone from the previous frame). Additionally it discards particles that are outside the bounding tetrahedron of this tetra-cone.

For each particle, as input to the vertex program, the following information is provided:

Variable for each particle:

- With POSITION semantics, the 2D coordinates in the frame-buffer of the collision result, as well as the tetra-cone from previous frame: (*x,y,tetra-cone*).
- With NORMAL semantics, the 3D position of the particle: (*x,y,z*).

Constant for all particles (uniform):

- Modelling and Viewing transformation, and Object transformation.
- Centroid of the polyhedron (Vertex V_0). This vertex is shared with all tetrahedra of the covering and with all tetrahedra of the tetra-tree.
- Maximum level of the tetra-tree.
- Textures with the geometry of the polyhedron and tetra-tree.

A general representation of the inputs and outputs between CPU and GPU is shown in Figure 4. This diagram is valid for all implementations proposed in this paper.

The vertex program returns the particle position and the tetra-cone in which the particle is located, using TEXCOORD0 semantics. This information is the input to the fragment program. COLOR semantics can not be used because the output range is restricted to the interval $[0.0-1.0]$. It is necessary to return the particle position to the fragment program because this information can not be obtained directly from CPU.

Particles that are outside the bounding tetrahedra can be discarded by applying a transformation that moves them, for example, behind the back-plane, so that the pipeline excludes them from later processes. In this case, it is not possible to obtain the tetra-cone in which the particle is classified for the next frame. Therefore, we send an exit flag using TEXCOORD1 semantics which indicates if the particle is discarded or not in the following stage.

The second stage of the algorithm has been implemented in the fragment shader. The input contains information about the particle position and the tetra-cone in which it is classified. We have included a flag in order to discard particles not included in the bounding tetrahedron. The geometry of the polyhedron is stored in textures.

In this stage, the inclusion of each particle in the tetrahedra that are classified in the input tetra-cone is studied by applying Algorithm 1. If the particle is inside the polyhedron, the R (red) component of the output pixel is written with the value 1. Obviously, the frame-buffer has been initialized to 0. Special cases (when sets of positive and negative features are needed) are implemented in the CPU and

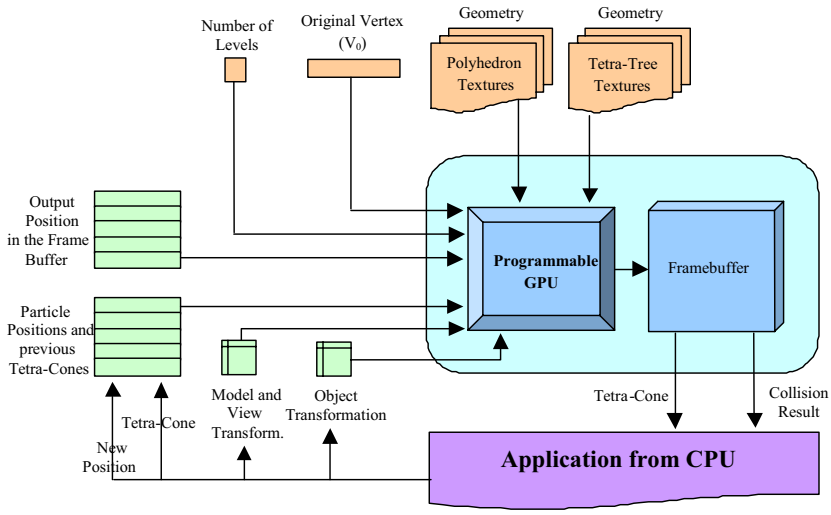


Figure 4: *Information flow between elements from CPU and GPU.*

$1/2$ value is written in the R component. This is due to the impossibility of implementing sets or similar structures in the GPU because of the language used. In these cases the collision is calculated in the CPU. Nevertheless, the probability of these cases is very small (<1%) and the performance is not severely affected.

The output of this stage consists of a set of particles that collide with the polyhedron, stored in the R component of the frame-buffer. The tetra-cone in which each particle is classified is also obtained in the GB components (green and blue) and this tetra-cone is used in the next frame. This information is codified in these two components because the range in the frame-buffer is within the $[0.0-1.0]$ interval and 8 bits per component are used.

The previous process, consisting of two stages, is summarized in Figure 5. The vertex program and fragment program are shown in the Appendix section.

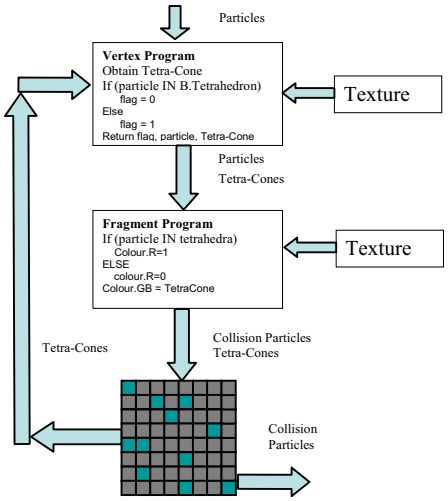


Figure 5: *Solution based on a Vertex and Fragment Shader implementation.*

4.3. Fragment Shader implementations

We could think that the previous implementation using both, vertex and fragment shaders for the algorithm stages would be optimal. However, the times obtained in collision detection show that this is not completely correct. The problem which arises is due to a synchronization fault in the texture access present in current graphics cards. This occurs when both fragment and vertex programs access simultaneously to textures and is due to the new characteristics of the Shader Model 3.0 which allow the access to texture from vertex programs, an operation that slows the global process down considerably.

On the other hand, the fragment shader must wait for the output from the vertex shader, the fragment shader being faster than the vertex shader. In this two-stage configuration there is one execution of the fragment program per execution of the vertex program, and the first must wait for the tetra-cones and particles positions obtained from the vertex program. Thus it is not possible to take advantage of the parallelism of several kernel units and the high speed of the fragment shader.

For this reason, we have performed some alternative im-

plementations in order to solve this problem efficiently, although in a less natural form. In these implementations we use the vertex shader to pass the input information to the fragment shader. These implementations are the following:

- First stage implemented in the CPU, and the second in the fragment shader (Figure 6.a).
- A fragment program implementation of the first stage and a fragment program implementation of the second stage. There is a change of the program used in the fragment shader between stages (Figure 6.b).
- Implementation of the first and second stages combined in one fragment program, with no program change between stages (Figure 6.c).
- Implementation of the first and second stages combined in one fragment program, without feed-back, that is without information about the tetra-cone from the previous frame (Figure 6.d).

5. Performance study

A 1.6 GHz Pentium IV processor with a Nvidia GeForce 6600 graphics card has been used. The algorithms have been implemented in a Linux platform with C++, Cg and OpenGL.

In the mixed solution based on CPU + Fragment-Shader (CPU+FS) we have obtained promising results, mainly due to two factors: Firstly, because there is no interference in the texture access between fragment and vertex shaders. This feature is common to all the fragment-based implementations. Second, the tetra-cones between frames are stored in main memory, and the reading of this data from the frame-buffer is not necessary. Frame-buffer reading is very expensive because transferences to the main memory are not optimized as in the other direction, i.e. from the main memory to the frame-buffer.

The Fragment-Shader + Fragment-Shader (FS+FS) solution presents a set of disadvantages that make this method slower than expected. In this case, it is necessary to read information from the frame-buffer (tetra-cones obtained in stage 1) in order to send this information to stage 2 after loading its associated program in the fragment shader. Therefore, it is necessary to read information from the frame-buffer one time and to load a fragment program two times per frame. Loading a program in the GPU supposes a small additional cost, but this is only taken into account for implementations which produce a change of the fragment program.

In order to solve some of the previous problems, a Fragment-Shader implementation with only one stage has been performed (FS with f-b). This stage is composed of the two aforementioned stages. So the fragment program is always in memory. The information on tetra-cones from frame-buffer feeds back the vertex shader that passes this

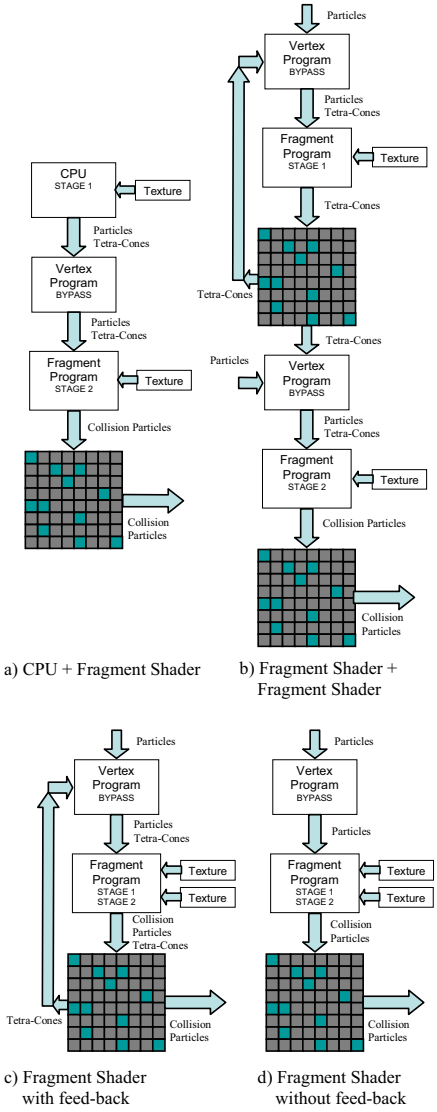


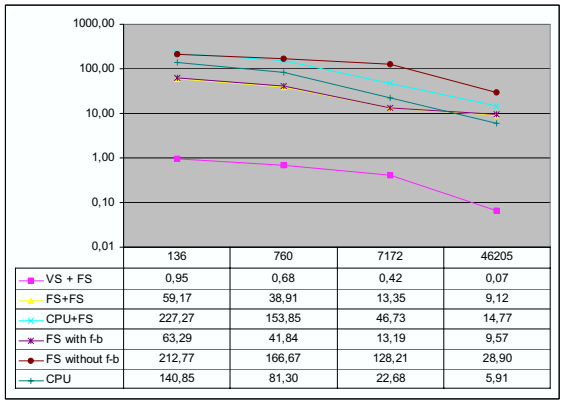
Figure 6: *Solutions based on Fragment-Shader implementations.*

information to the fragment shader. This solution is named "Fragment-Shader with feed-back".

Finally, a solution based on "Fragment-Shader without feed-back" has been implemented (FS without f-b). This solution is similar to the previous one, but no feed-back is performed. So tetra-cones are always calculated and the algorithm does not take advantage of the coherence in the movement of the particles. Due to the high cost of the transferences between frame-buffer and main memory, this algorithm is more efficient than the others, as shown in Figures 7 to 9.

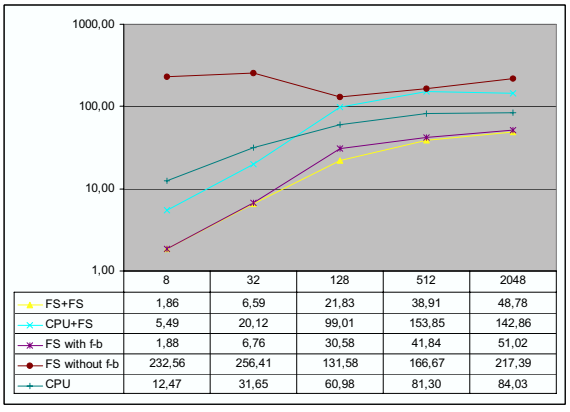
In these Figures we measure the number of frames per second using a logarithmic scale.

We performed several tests to measure the speed of the different implementations. The number of tetrahedra of the covering (Figure 10), the number of subdivisions performed with the tetra-tree (according to different levels), and the number of particles are parameters which we have measured. We have generated particles inside the bounding-box of each polyhedron and they move randomly. We have measured the number of frames per second in the collision detection.



	136	760	7172	46205
VS + FS	0,95	0,68	0,42	0,07
FS+FS	59,17	38,91	13,35	9,12
CPU+FS	227,27	153,85	46,73	14,77
FS with f-b	63,29	41,84	13,19	9,57
FS without f-b	212,77	166,67	128,21	28,90
CPU	140,85	81,30	22,68	5,91

Figure 7: *Frames/sec. in the collision of 1024 particles and some polyhedra. X axis shows the number of tetrahedra of the covering of each polyhedron.*

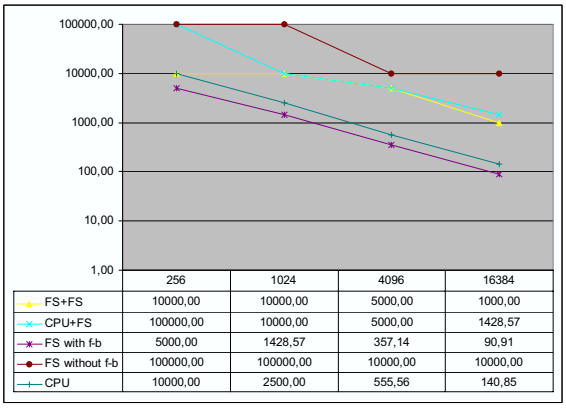


	8	32	128	512	2048
FS+FS	1,86	6,59	21,83	38,91	48,78
CPU+FS	5,49	20,12	99,01	153,85	142,86
FS with f-b	1,88	6,76	30,58	41,84	51,02
FS without f-b	232,56	256,41	131,58	166,67	217,39
CPU	12,47	31,65	60,98	81,30	84,03

Figure 8: *Frames/sec. in the collision of 1024 particles and the polyhedron named "Cat". X axis shows the number of subdivisions performed with the tetra-tree (level).*

The tests performed are:

- Collision detection in frames/second for different objects and a particle cloud formed by 1024 particles. We have fixed the tetra-tree depth to 2048 subdivisions, see Figure 7.



	256	1024	4096	16384
FS+FS	10000,00	10000,00	5000,00	1000,00
CPU+FS	100000,00	10000,00	5000,00	1428,57
FS with f-b	5000,00	1428,57	357,14	90,91
FS without f-b	100000,00	100000,00	10000,00	10000,00
CPU	10000,00	2500,00	555,56	140,85

Figure 9: *Frames/sec. in the collision of some particles and the polyhedron named "Horse". X axis shows the number of particles.*

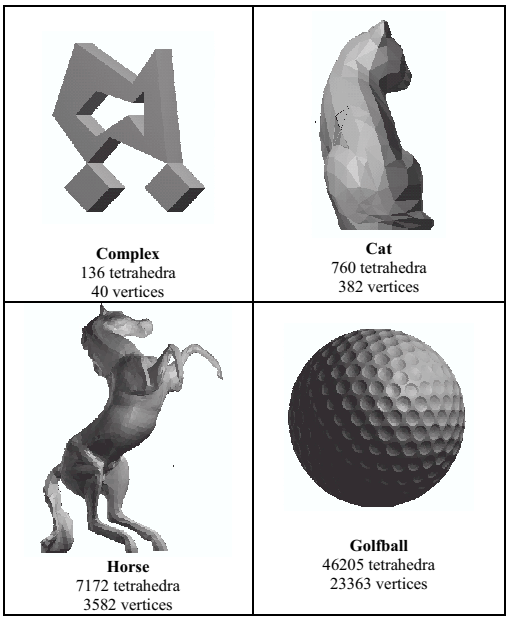


Figure 10: *Polyhedra used in the tests.*

- Collision detection in frames/second for the object "Cat" and a particle cloud formed by 1024 particles. We have change the tetra-tree depth (number of subdivisions), see Figure 8.
- Collision detection in frames/second for the object "Horse" and a particle cloud formed by a varying number of particles. We have fixed the tetra-tree depth to 2048 subdivisions, see Figure 9.

Particle movement has been calculated in CPU, so it is

Figure	Complex	Cat	Horse	Golfball
Tetrahedra	136	760	7172	46205
Vertices	40	382	3582	23363
CPU+FS	61%	89%	106%	150%
FS without f-b	51%	105%	465%	389%

Table 1: *Improvement percentages of some implementations with regard to CPU implementation.*

possible to send GPU a direction vector for each particle and actualize their positions in the vertex or fragment shader.

Table 1 shows the improvement percentage of the most important implementations with regard to the CPU implementation for the first test (Figure 7).

6. Conclusions and Future work

On one hand we have developed an exact and robust object-based method for collision detection between a complex object and a particle cloud by using the programmable GPU. On the other hand, we have shown the advantages of using several implementations based on programmable GPU.

The solution proposed does not need to decompose a complex object into convex pieces, the simplicial covering being a simple, elegant and fast solution to deal with this type of objects.

We can conclude that at present, it is less efficient to access texture information simultaneously from the vertex and fragment shader. In the first stage of the problem posed it would be more adequate to develop a hash function [THM*03] in order to obtain the tetra-cone in which the particle is included, it not being necessary to recursively classify the particle in the tetra-tree. Additionally, it is necessary to provide a mechanism in order to avoid the one-to-one vertex program and fragment program execution and thus to achieve the maximum performance of the fragment shader.

In addition, it is necessary to avoid information transference between frame-buffer and main memory, as well as program changing between frames in the GPU, in order to obtain efficient implementations.

The algorithms developed can be easily modified to use spherical particles. We can use tetrahedra with an offset equal to the sphere radius and use point-based particles [JFSO06].

We can extend this algorithm in order to obtain the collision determination, that is to say, the parts of the polyhedron which are involved in the collision. In order to do this we can use a p-buffer that allows a wider output range.

7. Acknowledgments

This work has been partially funded by the Ministry of Science and Technology of Spain and the European Union by means of the ERDF funds, under the research project TIN2004-06326-C03-03.

References

[Bad90] BADOUEL F.: *An efficient Ray-Polygon intersection, Graphics Gems.* Academic Press, 1990. 45

[BW03] BACIU G., WONG W.: Image-based techniques in a hybrid collision detector. *IEEE Transaction on graphics 9*, 2 (2003). 44

[Ebe04] EBERLY D. H.: *Game Physics.* Morgan Kaufmann publishers, Elsevier, 2004. 44

[FT97a] FEITO F. R., TORRES J. C.: Boundary representation of polyhedral heterogeneous solids in the context of a graphic object algebra. *The Visual Computer 13* (1997), 64–77. 44

[FT97b] FEITO F. R., TORRES J. C.: Inclusion test for general polyhedra. *Computer & Graphics 21*, 1 (1997), 23–30. 45

[GLM05] GOVINDARAJU N., LIN M., MANOCHA D.: Quick-cullide: Fast inter- and intra-object collision culling using graphics hardware. In *Proc. IEEE VR* (2005). 44

[JFSO06] JIMÉNEZ J. J., FEITO F. R., SEGURA R. J., OGAYAR C. J.: Particle oriented collision detection using simplicial coverings and tetra-trees. *Computer Graphics Forum 25*, 1 (2006), 53–68. 44, 45, 51

[JP02] JAMES D. L., PAI D. K.: Dynamic response textures for real time deformation simulation with graphics hardware. *ACM Transactions on Graphics 21*, 3 (2002), 582–585. 44

[JTT01] JIMENEZ P., THOMAS F., TORRAS C.: 3d collision detection: a survey. *Computer & Graphics 25* (2001), 269–285. 44

[KF05] KILGARIFF E., FERNANDO R.: *The Geforce 6 Series GPU Architecture, GPU Gems 2.* NVidia, 2005. 43

[KLR04] KOLB A., LATTA L., REZKSALAMA C.: Hardware-based simulation and collision detection for large particle systems. *Graphics Hardware* (2004). 44

[Kno03] KNOTT D.: *Collision and Interference Detection in Real Time Using Graphics Hardware.* PhD, University of British Columbia., 2003. 44

[LG98] LIN M., GOTTSCHALK S.: Collision detection between geometric models: A survey. In *Proc. IMA Conf. on Mathematics of Surfaces.* (1998). 44

[Sta99] STAM J.: Stable fluids. In *Proc. SIGGRAPH 99.* (1999), pp. 121–128. 44

[TF88] TERZOPOULOS D., FLEISCHER K.: Deformable models. *The Visual Computer. 4*, 6 (1988). 44

[THM*03] TESCHNER M., HEIDELBERGER B., MULLER M., POMERANETS D., GROSS M.: Optimized spatial hashing for collision detection on deformable objects. In *Proc. VMV* (2003). 51

[TKH*04] TESCHNER M., KIMMERLE S., HEIDELBERGER B., ZACHMANN G., RAGHUPATHI L., FUHRMANN A., CANI M.-P., FAURE F., MAGNETAT-THALMANN N., STRASSER W., VOLINO P.: Collision detection for deformable objects. *Computer Graphics Forum. 23*, 1 (2004), 61–81. 44

[WH94] WITKIN A. P., HECKBERT P. S.: Using particles to sample and control implicit surfaces. In *Proc. SIGGRAPH 94.* (1994), pp. 269–278. 44

```
#define MAX_COL_VERTICES 1024

#define getVertex(tetracone, level) { \
      v = texRECT( indexTT, float2( tetracone, level )).xyz; \
      v_fil = v / MAX_COL_VERTICES; \
      v_col = v % MAX_COL_VERTICES; \
      v1 = texRECT( vertexTT, float2( v_col.x, v_fil.x )).xyz; \
      v2 = texRECT( vertexTT, float2( v_col.y, v_fil.y )).xyz; \
      v3 = texRECT( vertexTT, float2( v_col.z, v_fil.z )).xyz; \
      v1 = mul(transf,float4(v1.xyz,1)).xyz; \
      v2 = mul(transf,float4(v2.xyz,1)).xyz; \
      v3 = mul(transf,float4(v3.xyz,1)).xyz; \
}

int is_in_tetra_cone ( float3 v0, float3 v1, float3 v2, float3 v3, float3 particle ) {

      int s1, s2, s3;
      int s0;
      float3 v0_particle, v1_particle, v2_particle, v3_particle;

      v0_particle = v0 - particle;
      v1_particle = v1 - particle;
      v2_particle = v2 - particle;
      v3_particle = v3 - particle;

      int result = 0;   // out of tetra-cone
      s1 = sign( determinant( float3x3( v0_particle, v3_particle, v2_particle )));
      if ( s1>=0 ) {
            s2 = sign( determinant( float3x3( v3_particle,v0_particle, v1_particle )));
            if ( s2>=0 ) {
                  s3 = sign( determinant( float3x3( v0_particle, v2_particle,
                                          v1_particle )));
                  if ( s3>=0 ) {
                        s0 = sign( determinant( float3x3( v1_particle, v2_particle,
                                   v3_particle )));
                        if ( s0 < 0 ) {
                              result = 2;//out of bounding tetra., but in tetra-cone
                        } else
                              result = 1;//in bounding tetra, in tetra-cone
                  }
            }
      }
      return result;
}

void main (
            float3 position : POSITION,//(x,y,0) frame-buffer output position
            float4 particle : NORMAL,  //(x,y,z,tetracone) particle pos. & tetra-cone

            uniform float4x4 ModelViewProj,
            uniform float4 v0,      //original vertex (tetrahedra) and its inclusion
            uniform float level,    //level of tetra-tree
            uniform float4x4 transf,//Transformation matrix of polyhedron & tetra-tree

            const samplerRECT indexTT : texunit3,//Indices of vertices for tetra-cones
                                                 //(iv1,iv2,iv3)
            const samplerRECT vertexTT : texunit4,//Vertex coordinates (x,y,z)

            out float4 oPosition  : POSITION,//frame-buffer output coord. (x,y,0,1)
            out float4 oParticle : TEXCOORD0,//Particle coord. & tetra-cone
                                             //(x,y,z,tetra-cone)
            out float4 oInfo : TEXCOORD1 )   //(x,0,0,0), in or out the bounding tetra.
{
      int tetracone, tcone, lev, I, is_in;
      float3 v1, v2, v3;
      int3 v, v_fil, v_col;

      oPosition = mul(ModelViewProj,float4(position.xy,0,1));
      oInfo.w = 0;
      tetracone = particle.w;
      v0 = mul(transf,float4(v0.xyz,1));

      getVertex( tetracone, level );
      is_in = is_in_tetra_cone( v0.xyz, v1, v2, v3, particle.xyz );
      if ( is_in == 0 ) {
            for (I=0;I<8;I++) {
                  getVertex( I, 0 );
                  is_in = is_in_tetra_cone( v0.xyz, v1, v2, v3, particle.xyz );
                  if ( is_in >=1 ) {
                        tetracone = I;
                        break;
                  }
            }
            for (lev = 1; lev<=level; lev++) {
                  for (I=0;I<4;I++) {
                        tcone = 4 * tetracone + i ;
                        getVertex( tcone, lev );
                        is_in = is_in_tetra_cone( v0.xyz, v1, v2, v3, particle.xyz );
                        if ( is_in >= 1 ) {
                              tetracone = tcone;
                              break;
                        }
                  }
            }
      }

      if (is_in == 2) oInfo.x = 1; //out of the bounding tetra.
      OParticle = float4( particle.xyz, tetracone );
}
```

Figure 11: *Appendix A: Vertex Program*

```
void main (
            float4 particle : TEXCOORD0,//(x,y,z,tetra-cone) particle pos.& tetra-cone
            float4 info : TEXCOORD1,   // (x,0,0,0), in or out the bounding tetra.

            uniform float4 v0,      // Original vertex (tetrahedra) and its inclusion
            uniform float level,    // Tetra-Tree depth
            uniform float4x4 transf,// Polyhedron and Tetra-Tree transformation

            const samplerRECT index: texunit1,  //Indices of vertices for tetrahedra
                                                //(i1,i2,i3,sign)
            const samplerRECT vertex: texunit2, //Vertex coordinates (x,y,z)

            out float4 oResult  : COLOR)  // (r,g,b,0) r=collision, gb=tetra-cone code
{
      int tetracone = particle.w;
      oResult = float4( 0, (tetracone / 256) / 256.0, (tetracone % 256) / 256.0, 0 );
      if (info.x==0) {
            int4 index_v;           //v1,v2,v3 indices
            float3 v[4];
            float3 v0_particle, v1_particle, v2_particle, v3_particle;
            int sign_, pos_x, sum;
            int s0, s1, s2, s3;
            float numTetra = texRECT( index, float2( 0, tetracone )).r;// (y,x).r
            v[0] = mul(transf,float4(v0.xyz,1)).xyz;
            sum = 0;
            for (int i=1;i<=numTetra;i++) {
                  index_v = texRECT( index, float2( i, tetracone ));
                  pos_x = index_v.x / 1024;
                  index_v = index_v % 1024;
                  sign_ = index_v.w;
                  if ( sign_ == 2 ) sign_ = -1;
                  v0_particle = v[0] - particle.xyz;
                  v[1] = texRECT( vertex, float2( index_v.x, pos_x )).rgb;
                  v[1] = mul(transf,float4(v[1],1)).xyz;
                  v[2] = texRECT( vertex, float2( index_v.y, pos_x )).rgb;
                  v[2] = mul(transf,float4(v[2],1)).xyz;
                  v[3] = texRECT( vertex, float2( index_v.z, pos_x )).rgb;
                  v[3] = mul(transf,float4(v[3],1)).xyz;
                  v1_particle = v[1] - particle.xyz;
                  v2_particle = v[2] - particle.xyz;
                  v3_particle = v[3] - particle.xyz;
                  float b0 = determinant( float3x3( v1_particle, v2_particle,
                                          v3_particle ));
                  s0 = sign_ * sign(b0);
                  if ( s0 >= 0 ) {
                        float b1 = determinant( float3x3( v0_particle, v3_particle,
                                                v2_particle ));
                        s1 = sign_ * sign(b1);
                        if ( s1 >= 0 ) {
                              float b2 = determinant( float3x3( v3_particle,
                                                      v0_particle, v1_particle ));
                              s2 = sign_ * sign(b2);
                              if ( s2 >= 0 ) {
                                    float b3 = determinant( float3x3( v0_particle,
                                                            v2_particle, v1_particle ));
                                    s3 = sign_ * sign(b3);
                                    if ( s3 >= 0 ) {
                                          int vert = -1;
                                          // inside v0v1v2v3, face v1v2v3
                                          if ( s1 > 0 && s2 > 0 && s3 > 0 )
                                                sum += 2 * sign_;
                                                // vert. v2, v3, edge v2v3
                                          else if ( s0==0 && s1==0 ) {
                                                oResult.r = 1;
                                                break;
                                          }
                                                // vert. v1, edge v0v1
                                          else if ( s1>0 && s2==0 && s3==0 )
                                                vert = 1;
                                                // edge v0v2
                                          else if ( s0>0 && s1==0 && s2>0 && s3==0 )
                                                vert = 2;
                                                // edge v0v3
                                          else if ( s0>0 && s1==0 && s2==0 && s3>0 )
                                                vert = 3;
                                                // vert. v0
                                          else if ( s0>0 && s1==0 && s2==0 && s3==0 ) {
                                                oResult.r = v0.w;
                                                break;
                                          }
                                          // Face v3v2v0,v3v0v1,v0v1v2, edge v1v2,v2v1
                                          else sum += sign_;
                                          // Special cases, calculated in CPU
                                          if ( vert != -1 ) {
                                                oResult.r = 0.5;
                                                sum = 0;
                                                break;
                                          }
                                    }
                              }
                        }
                  }
            }
            if ( sum==2 ) oResult.r = 1;
      }
}
```

Figure 12: *Appendix B: Fragment Program*