EUROGRAPHICS 2004 / M.-P. Cani and M. Slater
(Guest Editors)

Volume 23 (2004), Number 3

Self-intersections with Cullide

Niels Boldt[†] and Jonas Meyer[‡]

Department of Computer Science, University of Copenhagen, Denmark.

Abstract

We present an image-space technique, which can detect intersections and self-intersections among multiple moving and deforming objects. No preprocessing is needed and the shape of the objects are unconstrained and can be an arbitrarily polygon soup. Compared to other intersection detection algorithms running on graphics hardware the algorithm only make modest use of bandwidth between the CPU and GPU because no buffer readbacks are necessary.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Collision Detection, Image-Space, Graphics Hardware, Self-intersections

1. Introduction

Efficient collision detection is a fundamental problem in physical-based simulation, computer animation, surgery simulators and virtual prototyping and can often be the bottleneck in the mentioned areas. For rigid bodies many efficient approaches, based on precomputed bounding-volume hierarchies have been proposed [Hub93, GLM96, KHM*98].

But as deformable objects are becoming normal and necessary, these hierarchial datastructures can not be precomputed, but must be updated frequently. Some effort has been given to optimize this update [LAM01], but it is still expensive to update the hierarchial datastructures in dynamic environments. An even harder problem is posed if self-intersections in a deforming object must be detected. Use of bounding-volume hierarchies must be abandoned, because traversal of the hierarchies will be slow due to the many overlaps of bounding volumes. For cloth simulation acceptable solutions has been proposed [VMT00, VT95].

The rise of graphics hardware can provide a solution to the self-intersection problem. In the last few years, powerful graphics hardware has become a standard part of a commodity PC. The cheap computing resources found in graphics hardware has spurred an interest in intersection detection based on image-space techniques.

In this paper, an image-space technique for intersection detection of arbitrarily shaped, deforming object is presented. The objects can deform in any way and both open and closed objects can be treated. The approach is based on Cullide, presented in [GRLM03], but in contrast, we adress, and solve, the problem of self-intersections.

Compared to previous approaches, our approach is quite simple and can easily be implemented on commodity graphics hardware. As processing power of graphics hardware grows faster than processing power of general purpose hardware, our approach should be able to scale faster than algorithms implemented on standard CPU's. There is no restriction on the geometry used, except that it should be triangulated. Neither preprocessing nor hierarchial structure are necessary for the correctness of the algorithm. However, a hierarchy based on mesh connectivity can improve the algorithm significantly.

The rest of the paper is organized as follows: We give an description of previous work done on algorithms for detecting self-intersections and intersections using graphics hardware in section 2. We give an overview of the original Cullide algorithm in section 3. We present our modification to Cullide in section 4. In section 5 we discuss problems with image-space precision and compare precision of Cullide and the modified version. In section 6, various tests are performed, to rate the performance of the modified algorithm. In section 7 we summarize our work and give directions for future work.

† email: boldt@diku.dk
‡ email: meyer@diku.dk

2. Previous Work

In this section we will give a brief review of prior work on image-space techniques for intersection detection.

An approach to image-space intersection detection is presented in [BWS99]. The depth layers of two convex objects are rendered into the depth buffer. Any overlaps between the objects is found using the stencil buffer. The method can only handle convex objects and do not consider self-intersections. The main problem is that the stencil buffer must be read back into main memory and processed.

In [HKL*99], graphics hardware is used to compute generalized 2D and 3D Voronoi diagrams, which can be used for proximity queries. For each sample point, closest site and distance to that site is computed, using polygon scan-conversion and depth buffer comparisons. Distance meshes for points, lines, curves and curved surfaces in 2D and 3D can be computed.

A specialized approach to intersection detection for cloth on walking people is presented in [VSC01]. Depth map, normals and velocity for the avatar is computed in image-space. Intersection detection for a particle can be done by transforming to image space and comparing with the depth map. The algorithm relies heavily on buffer readback.

In [HTG04] an image-space technique is used to compute a layered depth image, which can be queried for collisions and self-intersections. The algorithm requires that the geometry is a twofold. The algorithm is implemented in two versions, on CPU and in graphics hardware. For small configurations the CPU-implementation is most efficient. For big configurations with up to 500k faces the GPU-implementation running on graphics hardware is most efficient. Again, the algorithm relies on buffer readback.

In [GRLM03], occlusion queries are used to prune a set of potentially overlapping triangles. No preprocessing is needed and arbitrary polygon soups can be handled. An advantage is that buffer readbacks are not needed, implying minimal use of bandwidth.

3. Cullide

In this section we will give an overview of Cullide as presented in [GRLM03]. An object is defined to be a collection of one or more triangles. In the following we assume an environment O of n objects $o_1, o_2, \ldots, o_n$.

In the following, we assume that all objects are divided into some form of hierarchy. We will ignore the form of the hierarchy because it only influences performance and not correctness.

3.1. Overview

Given a collection of objects, the purpose of a collision detection algorithm, is commonly to determine either of the following

- A set of pairs of intersecting triangles.
- A set of pairs of closest features.

Most collision detection algorithms determine either one or both of the above.

Cullide is different in this aspect, as it does not directly determine any of the two results mentioned above. Instead, the algorithm takes as input a set O, of n objects. The algorithm reduces this set, and returns a new set, called the PCS, *potentially colliding set*, which is a subset of the original set.

For clarity, we consider Cullide consisting of two parts. The first part manages the graphics hardware and controls what should be sent to it. It also controls the usage of object hierarchies. The second part is a simple operation, `reducePCS`, which can be implemented using graphics hardware. The operation take as input a set of objects O, and prune some of the non-intersecting objects away. The operation does not guarantee that all non-intersecting objects will be removed, but it does guarantee that no intersecting objects will be pruned from the set.

The `reducePCS` operation can be considered a primitive operation - There are few variations on to how it can be implemented. The performance of the algorithm depends primarily on the first part of the algorithm - we can not improve `reducePCS`, because it depends solely on the performance of the graphics hardware.

3.2. Part 1: Usage of the object hierarchies

The usage of the hierarchies can be relatively simple. We start by running `reducePCS` at the object level, pruning non-intersecting objects. We then replace all the non-leaf objects in the PCS, with their children and repeat the operation. The algorithm is finished once the PCS is empty or all the objects in the PCS are leaf nodes. The following pseudo code describes the algorithm:

```
cullide( Objects )
  PCS = Objects
  do
    PCS = reducePCS( PCS )
    for each node in PCS
      if !is_leaf( node )
         PCS.remove( node )
         PCS.insert( node.children )
  while has_non_leaf_nodes( PCS )
  return PCS
```

Figure 1: *A basic version of Cullide.*

3.3. Part 2: The `reducePCS` operation

The `reducePCS` operation is implemented on graphics hardware using occlusion queries, see [GRLM03] for elaboration of the details.

The visibility information gained from occlusion queries is used to prune objects from the PCS. If all fragments pass the depth test, when rendering an object, the object is said to be fully visible, with respect to the set of objects already rendered to the depth buffer. Thus we get the first lemma:

Lemma 3.1 If all fragments generated, when rendering an object, pass the depth test, the object is fully visible. When an object is fully visible, the object does not intersect with any of the objects already rendered to the depth buffer.

The proof is trivial. If there is an intersection with any of the objects already rendered, the intersection will cause some of the pixels to fail the depth test.

An object can be pruned from the PCS if it is determined that the object is fully visible, with respect to all other objects in the set. A naive approach to using the algorithm would require rendering each object n times, thus yielding a time complexity proportional to $\mathbf{O}(n^2)$. The solution to the problem lies in the following lemma

Lemma 3.2 Given a set, O, of objects $o_1, o_2 \dots o_n$. The object o_k , $k \in [1,n]$ is fully visible with respect to $O \setminus o_k$, iff it is fully visible with respect to $O'_k = o_1 \dots o_{k-1}$ and fully visible with respect to $O''_k = o_{k+1} \dots o_n$.

Again, the proof is trivial. If o_k is fully visible to both O'_k and O''_k, it must be fully visible to the union of the two sets, $O \setminus o_k$.

Using lemma 3.2, we see that the two subproblems exhibits an optimal substructure. That is, when testing object o_k, against the set O'_k, we note that $O'_k = O'_{k-1} \cup \{o_{k-1}\}$. Thus we can reuse the rendered data. Likewise for O''_k.

To take advantage of this, we first test all objects against their respective O'_k, and then test all objects against their O''_k. The effect of this, is that the running time of the algorithm is reduced to $\mathbf{O}(n)$. Pseudo code for the `reducePCS` algorithm is given in figure 2.

```
reducePCS( Objects )

  ClearDepthBuffer()
  for each object in Objects
    DepthTest( GREATER_EQUAL )
    DepthMask( FALSE )
    BeginOcclusionQuery()
    object.render()
    object.fullyvisible =
      EndOcclusionQuery() == 0
    DepthTest( LESS )
    DepthMask( TRUE )
    object.render()

  reverse( Objects )

  ClearDepthBuffer()
  for each object in Objects
    DepthTest( GREATER_EQUAL )
    DepthMask( FALSE )
    BeginOcclusionQuery()
    object.render()
    object.fullyvisible =
      EndOcclusionQuery() == 0
      && object.fullyvisible
    DepthTest( LESS )
    DepthMask( TRUE )
    object.render()
  remove_fully_visible( Objects )
  return Objects
```

Figure 2: *Pseudo code for reducePCS. As* `EndOcclusionQuery()` *returns the number of fragments passing the inverted depth test, a object is fully visible with respect to the objects rendered into the depth buffer if this query returns* 0.

4. Self-intersections

A drawback of Cullide, is that the algorithm is unable to detect self-intersections. This is due to the construction of the `reducePCS` operation where object o_k only are tested for intersection against the other $n-1$ objects. So when we remove an object from the PCS, we know that it does not intersect with other objects, but we do not know if any self-intersection exists in the removed object. In the following we will explain how to modify Cullide such that the algorithm becomes capable of detecting self-intersections.

Algorithm

A simple idea, would be to test the object against itself, while testing against either O'_k or O''_k. This is not immediately possible, since the depth test function is less than. If we render an object to the depth buffer, and then render it again, doing an occlusion query, it will fail no matter what, since none of the fragments will generate lesser depth values. This happens because we submit the same object twice - during the second pass, the rasterizer will generate exactly the same fragments.

This can easily be remedied, by changing the function to less than or equal. This ensures that *some* of the fragments will pass the depth test. We thus have an opportunity to prune away subparts of the object. Faces closest to the viewer will be pruneable. Faces obscured, fully or partially, by other faces can not be pruned before the faces in front of them are pruned.

In order to see how we detect self-intersections, we must consider the objects visibility as a whole. We describe this visibility with reference to the object itself as self-visibility. The following definition formalizes it:

Definition 4.1 An object is fully self-visible, if all fragments generated by rendering the object to a depth buffer in which the object has already been rendered, pass the modified depth test.

The crucial observation, is similar to what appeared in the original Cullide algorithm - Fully visibility means no intersections. We restate it regarding self-intersections and self-visibility:

Lemma 4.2 An object which is fully self-visible never has any self-intersections.

The proof of this lemma is obvious. If an object is fully self-visible all fragments generated can be seen if we look in the direction of projection. A necessary condition for self-intersection is that a fragment generated must be hidden behind another fragment. Therefore we conclude that no self-intersection can exist. Notice that the lemma above is vague - fully self-visibility implies no self-intersections, but objects that are not fully self-visible are not necessarily self-intersecting.

We need to expand our rule to sub-objects, otherwise, we will be unable to prune many objects. The following lemma describes self-visibility recursively, as a function of the self-visibility of the sub-objects. This allows recursive testing, and pruning objects by the use of their hierarchies.

Lemma 4.3 An object is fully self-visible, if all its sub-objects are fully self-visible and fully visible with reference to all other sub-objects.

Again, the proof is trivial. If an sub-object does not intersect with any other sub-object, and neither intersects with itself, it can not be part of any self-intersection. If this applies for all sub-objects, the object can not contain any self-intersections.

Turning back to the original version of `reducePCS` (Fig. 1), we can see that it does already test all sub-objects for visibility against each other - We must therefore modify it to test against itself.

By changing the depth function and adding a test against the object itself, while testing against O'_k, we get a modified version of `reducePCS`, which does not prune self-intersecting objects. This is the only change necessary, to enable Cullide to handle self-intersections. The modified version is shown in figure 3.

```
reducePCS( Objects )

  ClearDepthBuffer()
  for each object in Objects
    DepthMask( TRUE )
    DepthTest( LESS )
    object.render()
    DepthMask( FALSE )
    DepthTest( GREATER )
    BeginOcclusionQuery()
    object.render()
    object.visible =
      EndOcclusionQuery() == 0

  reverse( Objects )

  ClearDepthBuffer()
  for each object in Objects
    DepthMask( FALSE )
    DepthTest( GREATER )
    BeginOcclusionQuery()
    object.render()
    object.visible =
      EndOcclusionQuery() == 0
      && object.visible
    DepthMask( TRUE )
    DepthTest( LESS )
    object.render()

  remove_fully_visible( Objects )
  return Objects
```

Figure 3: *Our modified version of `reducePCS` where objects containing self-intersection are not pruned from the PCS.*

Drawbacks

There are drawbacks to the method. The first comes from the change of depth function. The original Cullide algorithm was able to detect contacts. Our method can not do this, but is limited to detecting penetrations. This limits our method to detecting penetrations. We do not believe that it will be a problem. The primary reason is, that there are no way of distinguishing between contacts and penetrations. If an algorithm is to use our method (or the original Cullide), it would have to utilize some algorithm to distinguish between faces penetrating, and faces in contact. Further, when detecting contacts, algorithms provide some means to set a minimum detection distance, a collision envelope, such that all primitives closer than this minimum distance are treated as contacts. This is not possible with Cullide either, but it is implicitly set, by the precision of the depth buffer and the resolution of the screen.

Another drawback appears when considering closed objects. It should be obvious, that all closed objects will *not* be fully visible with respect to themselves - Some object parts are bound to obscure other parts. Thus they are guaranteed to stay in the PCS. This does not affect the correctness of the algorithm, since the algorithm descends to the lower levels of the hierarchy of the object, and prunes on the lower levels of the objects.

Finally, we must consider the case where a node of the hierarchy contains a set of triangles, which obscure each other, no matter where they are viewed from. Closed objects are one instance of such a problem, another is shown in figure 4. This will prevent the algorithm from pruning the object. As can be seen in figure 4, it is a rather obscure example - sub-meshes composed of few triangles rarely exhibit these forms.

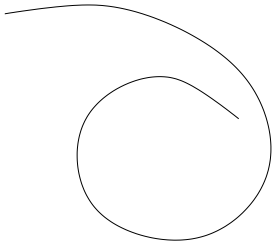


Figure 4: *Illustration of a bad mesh combination. The Figure shows a sample combination of triangles, that will always fail to be pruned, when placed in a leaf. It fails to do so, because no matter from where we observe the sub-object, we will unable to see all fragments generated by rendering the object. Note that the illustration is a two dimensional version of the actual three dimensional problem.*

5. Precision issues

In order to correctly utilize the algorithm, it is necessary to understand the implications of using the graphics hardware. In this section we will shortly explain what has an effect on the precision of the algorithm. The discussion will generally apply to the original Cullide and our version, and we will mention when they differ.

The algorithm can be considered as a hardware accelerated discrete approach to a brute force solution. Thus the precision issues with the algorithm occurs when transforming our exactly formulated problem into a discrete one. This happens when using the rasterizer of the graphics hardware.

The first problem is regarding the rounding of the values used in the depth buffer. Here our approach differs from Cullide. In Cullide, the strict less than test ensures that if two polygons map to the same depth, they will generate an intersection. Since our method changes the depth test function, it requires objects to penetrate by a certain amount, before the depth test will ensure detection of intersections. Given an orthographic projection with clipping planes given by $near, far, far > near$, and a depth buffer with n bits of precision, they need to penetrate by at least

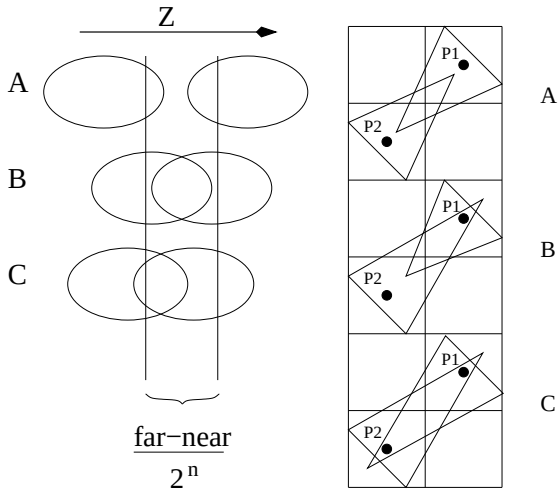
$$\frac{far - near}{2^n}$$

to guarantee detection.

In figure 5(a), three different cases of overlap in depth values are illustrated. The vertical lines indicate where the rounded value changes. There is no overlap in A, and penetration on both B and C. Cullide detects an intersection in all three cases. Our algorithm only detects an intersection in case C.

The second problems is regarding to the rasterizer. The general rule that rasterizers follows, is to fill only pixels having centers completely inside the triangle being rendered. This has the effect, that triangles can overlap in screen-space, without actually filling the same pixels. The bound for this overlap is, given a screen with a pixel of size $width \times height$, $\sqrt{width^2 + height^2}$.

Figure 5(b), shows 12 pixels of the screen that we render three cases to, here illustrating the problems with the resolution of the screen. The dots mark the center of the pixels. In all three cases, the triangles intersect. In case A, neither Cullide or our method will detect an intersection. In case B, Cullide will detect an intersection iff the triangles depth values at P1 are rounded to the same value. Our method will not detect an intersection in case B. In case C Cullide will always detect an intersection, our method will detect one if the distance between the depth of the fragments at P1 and P2 are at least $\frac{far-near}{2^n}$.



(a) depth value rounding (b) rasterization

Figure 5: *Difference in precision between Cullide and our method*

Given a triangle, we observe, that the closer the normal of the triangle is to being perpendicular to the direction of projection, the smaller will the projected triangle be. In fact, a triangle with normal perpendicular to the direction of the projection should not generate any pixels at all. This does unfortunately cause erroneous pruning of triangles.

6. Results

We have implemented our system on a 2.8 GHz Pentium 4 with a GeForceFX 5900 Ultra graphics card. We have tested our implementation, using modified meshes from [Bra], in the following environments:

- A deformed Stanford Bunny with a single intersection area. We have tested with triangles count from 5000

Figure 6: *This figure shows our deformed Stanford bunny. The intersecting triangles are shown with green.*

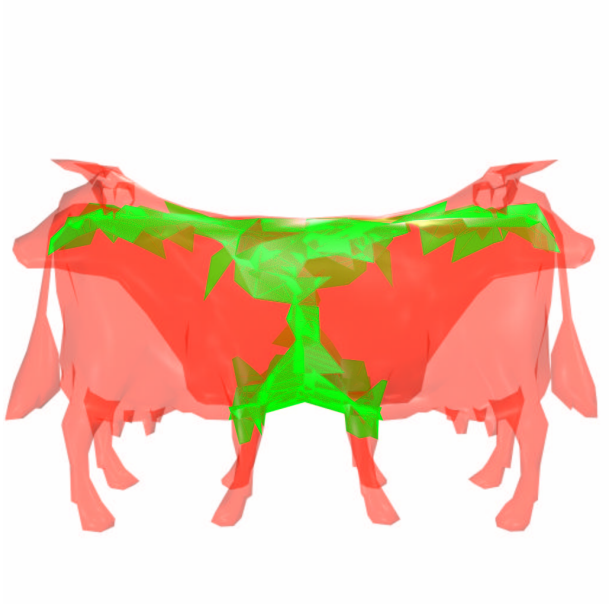


Figure 7: *This figure shows our intersecting cows. The intersecting triangles are shown with green.*

triangles to 270000 triangles. We are able to prune these models in 20 to 600 ms. The model is shown in figure 6. A figure showing the bunny in the pruning process can be found in figure 12

- Two cows combined to one object such that many self-intersections exists. We have tested with triangle count from 3000 to 270000 triangles. We are able to prune this model in approximately 35 ms to 1000 ms. The model is shown in figure 7.

A comparison of pruning times is presented in figure 8. This is done at a resolution of 500x500. It can be seen that it is significantly faster to prune the bunny compared to the cow. This is caused by the large amount of self-intersections in the cow, which make the algorithm descend deeply into the hierarchy.

In figure 9 and 10 we have compared pruning time with triangles rendered for each pass of a single collision query on the cow and bunny. A pass includes several calls to `reducePCS` with different direction of projections. In figure 11 we compare the relative performances of the two collision queries on the cow respectively the bunny. For the cow approximately 40% of the used time, is spent pruning 2-3% of the pruned triangles. The same tendency is exhibited with the bunny.

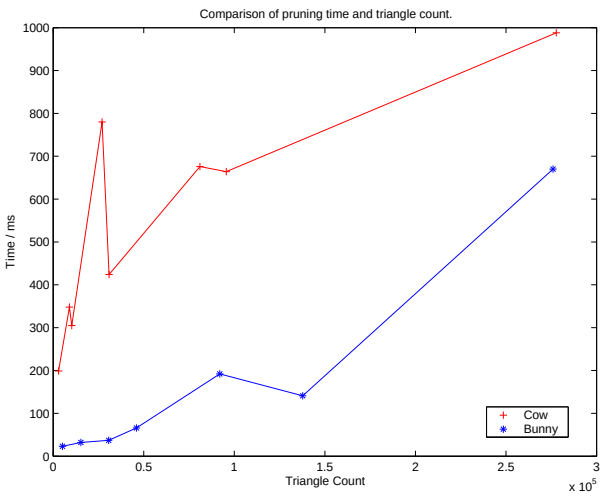


Figure 8: *A comparison of pruning time for bunny and cow with varying triangle count. The peak for the cow is due to an extra number of passes required to prune the PCS compared to other instances of this model.*

6.1. Performance Analysis

From the test results we can conclude that the performance of the algorithm depends on triangle count, number of

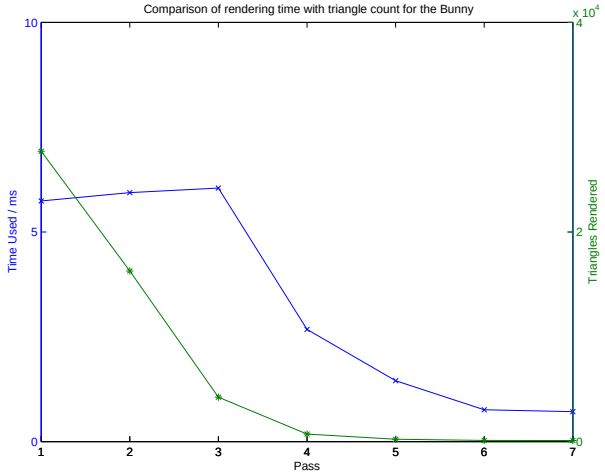


Figure 9: *The time used and triangles rendered is plotted, for each pass, for the bunny with 5000 triangles. The used time and number of triangles rendered does not necessarily depend linearly on each other, because the number of occlusion queries differ in the passes.*

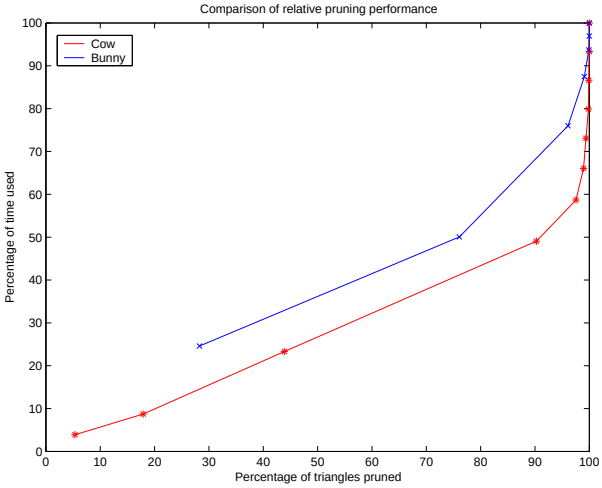


Figure 11: *A comparison of the relative pruning performance for a occlusion query on the cow and bunny. On the x-axis the accumulated percentage of triangles pruned can be read, while the percentage of the accumulated used time can be read from the y-axis. The starting point of the lines denote the percentage of triangles pruned in first pass.*

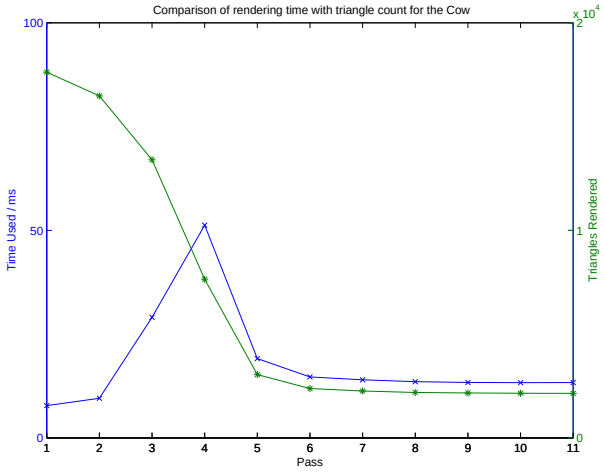


Figure 10: *The time used and triangles rendered is plotted, for each pass, for the cow with 3000 triangles.*

occlusion queries, and number of intersections. Another important observation, we made during development of the modified Cullide is the importance of the hierarchy. Initially we used a randomly built hierarchy, which performed poorly. Instead we used a hierarchy built on mesh connectivity, which improved performance significantly. All of the tests are based on hierarchies built this way.

7. Conclusion and further work

We have presented Cullide, and shown how to augment it, to handle self-intersections.

The modified Cullide seems reasonably easy to implement, but this is not the case if performance is of importance. To maximize performance of the modified Cullide clever strategies for rendering and construction of hierarchies must be made.

Performancewise many things could be done. Systematic methods for finding the best direction of projection could be based on assumptions of temporal coherence or object orientation. A heuristic for finding the time, at which pruning should be turned to the CPU can improve the algorithm.

Regarding the precision issues, some work is necessary to make the algorithm stable, and to ensure that no intersecting triangles can be pruned. Testing how close a triangles normal is to being perpendicular to the direction of projection, could be used to estimating which triangles that

are too small to be pruned. This is definitely possible, but we believe it will be hard to implement on current graphics hardware, without imposing some restrictions to the layout of the triangles. Future graphics hardware is very likely to provide capabilities, allowing the process to be fully automated.

References

[Bra] BRADSHAW G.:. http://isg.cs.tcd.ie/spheretree/. 5

[BWS99] BACIU G., WONG W. S.-K., SUN H.: Recode: an image-based collision detection algorithm. *The Journal of Visualization and Computer Animation 10* (december 1999), 181 – 192. 2

[GLM96] GOTTSCHALK S., LIN M. C., MANOCHA D.: OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics 30*, Annual Conference Series (1996), 171–180. 1

[GRLM03] GOVINDARAJU N. K., REDON S., LIN M. C., MANOCHA D.: Cullide: interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2003), Eurographics Association, pp. 25–32. 1, 2

[HKL*99] HOFF III K. E., KEYSER J., LIN M., MANOCHA D., CULVER T.: Fast computation of generalized Voronoi diagrams using graphics hardware. *Computer Graphics 33*, Annual Conference Series (1999), 277–286. 2

[HTG04] HEIDELBERGER B., TESCHNER M., GROSS M.: Detection of collisions and self-collisions using image-space techniques. *WSCG 12*, 1-3 (februrary 2004). 2

[Hub93] HUBBARD P. M.: Interactive collision detection. In *IEEE Symposium on Research Frontiers in Virtual Reality* (1993), pp. 24–32. 1

[KHM*98] KLOSOWSKI J. T., HELD M., MITCHELL J. S. B., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of *k*-DOPs. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (/1998), 21–36. 1

[LAM01] LARSSON T., AKENINE-MÖLLER T.: Collision detection for continuously deforming bodies. Eurographics, pp. 325–333. 1

[VMT00] VOLINO P., MAGNENAT-THALMANN N.: *Virtual Clothing, Theory and Practice*. Springer-Verlag Berlin Heidelbarg, 2000. 1

[VSC01] VASSILEV T., SPANLANG B., CHRYSANTHOU Y.: Fast cloth animation on walking avatars. *Computer Graphics Forum 20* (2001). 2

[VT95] VOLINO P., THALMANN N. M.: Collision and self-collision detection : Robust and efficient techniques for highly deformable surfaces. *Eurographics Workshop on Animation and Simulation* (1995). 1

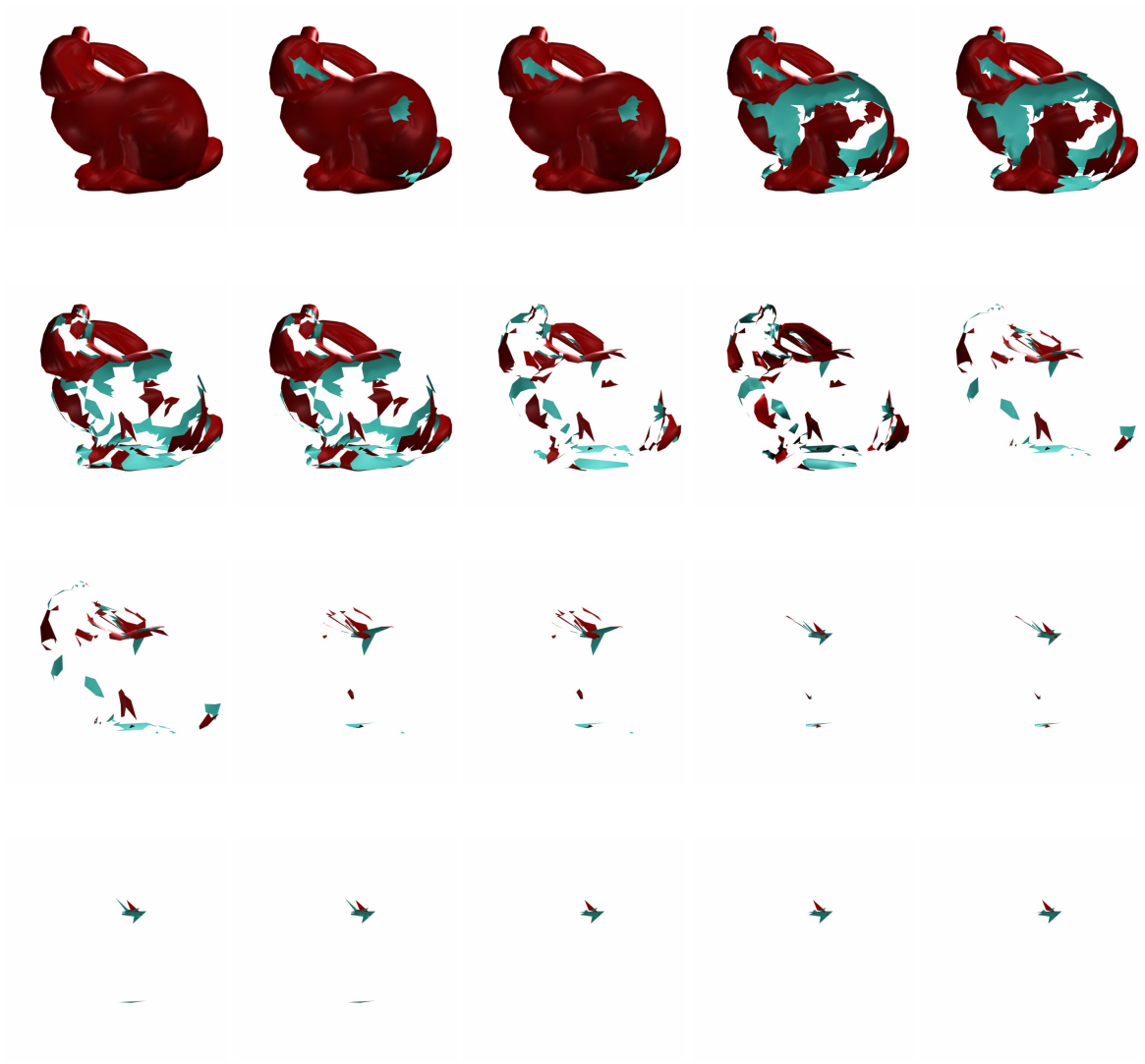


Figure 12: *This sequence shows how the* `reducePCS` *operation works on the bunny with 5000 faces. The first picture shows the original model and in each succeding picture the PCS is shown after the PCS in the previous picture has been pruned.*