Nicholas Hampshire

BACVA Level H 2008

# Innovations Project Report

# Real-time Collision Detection using Polygonal Patches

# 0.0 Notes

All discussion which assumes processing power of a computer system running the simulations are based on the computer that they were developed, which has the following performance statistics and details:

Macintosh OS 10.4.11
1.67 GHz PowerPC G4 processor
512KB L2 cache
1 GB DDR SDRAM
ATI Mobility Radeon 9700 128MB

Collision detection and physics could be thought of as a very similar process within a simulation. However, there are vast differences in the actual kinds of calculations performed by these two areas. Collision detection has a few core tasks it must specifically undertake: Discover through mathematical test whether two geometric models have intersected or interpenetrated each other; If they have found to be colliding, take relevant action against it by generating "contact" information that typically consists of: A normalised vector that indicates the direction in which the objects should retaliate in order to be expelled from each other; the depth that the objects have interpenetrated each other.
Physics encompasses tasks involving iterating the position of a geometric model over a fixed time step based on the forces acting on it. These forces can be based on virtual environmental influences like gravity and friction, they are also generated from the contact information returned by the collision detection system should it have found objects to be interpenetrating each other, specifically called physical response. Therefore, it is possible to see how the two systems are in-fact separate from each other, but still highly dependent of one another. The collision detection system must be aware of the position of the object, data generated by the physics system. The physics system must be aware of the forces each object is undergoing in order to integrate their position over the next time step.

In the project the focus is on collision detection, therefore a simple physics system has been implemented in order to demonstrate these principles in a way that has attempted to approach a degree of physical realism. The system is based on code that has been re-factored from open source examples (Fiedler, 2007). The system has also only thus far been designed with translating objects, but rotations could be implemented.

# 1.0 Introduction

Real-time physical simulations require a level of collision and physical realism in order to actually be seen as a proper physical simulation, and to be perceived in an appreciable way by the analytical human mind. In this instance, “real-time” should be defined as the rate of frame refresh on a display that is perceived by the eye to give the impression of

continuity, which indicates a rate of refresh at or above 25 frames a second.

Whilst it is commonly understood how real world forces such as friction, bounce and other environmental forces such as gravity can be implemented on an abstract virtual object undergoing such forces in a computer simulation, the real-time analysis of object interaction and collision response can become highly complex and unpredictable, requiring a lot of processing time. A "virtual shape" or "virtual object" refers to a 2 dimensional (2D) geometric polygonal model consisting of vertices and edges, or a 3 dimensional (3D) geometric polygonal model consisting of vertices, edges and faces. This report discusses the steps taken to not only produce a system allowing for detailed interaction between objects of any shape and form in 2D and 3D, but also algorithms that increase the efficiency of an otherwise laborious and slow process. The highs and lows of the design are then discussed and evaluated in context.

## 1.5 Intersection

In physical simulations that occur in such applications as computer games, attaining reasonably high levels of visual realism is no longer a problem in real-time. He (1999, p.1) explains that it is perfectly legitimate for models of well over hundreds of thousands of polygons to be drawn in real-time, and it is likely many more could be drawn.

Sadly however, it is preposterous to suggest a physical simulation where multiple geometrical models of this vast complexity and detail could even begin to have collision detection algorithms performed on them in real-time without extensive overlaying algorithms in place. Indeed, such a process would require substantial processing time, potentially hours, and it is the process of collision detection which would cause this huge expense on the processor.

This problem is widely recognised and has encountered many developments in the past. Lin and Gottschalk (1998, p.1) survey how collision detection is one of the major bottlenecks encountered in computer graphics processing. Solutions for the general problem are focused on various different levels of data processing and arrangement. The lowest level may be polygonal, which sees efforts such as implementing faster mathematical approaches to the calculations applied (Giang and O'Sullivan 2006 cited Dingliana and O'Sullivan 1999).

Next is likely to be the generation of stand-in shapes that replace geometry with simple mathematical forms, which are very faster to evaluate but often provide less desirable results. Giang and O'Sullivan (2006, p.2-3) demonstrate the use of circles to approximate the shape of rectangles in order to produce fast but regularly unpredictable results.

The structural approach is often used to generate trees allowing for quick

discarding of data that need not be included in the low level mathematical evaluation of detection in the first place. The concept of Closest Feature Maps introduced by Giang and O'Sullivan (2006), which explains how trees of spheres can be used to discard large amounts of polygons from analysis, is an excellent example of a structured approach. Other organisational systems like octal trees for 3D systems and quad trees for 2D systems also demonstrate the popularity and effectiveness of the approach, and orientation based systems like that of He (1999).

## 2.0 Generation

In the ideal case it is the intention to maintain as much integrity of the original geometrical shape as possible during collision detection. Considering an object that has been modelled in order to be used in a simulation, there are a number of pieces of data that can be generated at this stage. Not only will this allow for the kind of detailed interaction sought, it will also pre-compute as much as possible in order to save time during run-time evaluation, without currently focusing on the outer structural problem. This was approached in two ways, one for 2D models and another for 3D, in programs that generate an output file prior to running the simulation. The simulation program then loads data from these files into an internal data type. The program that analyses the geometry has been built to import data from the “.obj” file format (Wavefront, 1995), but could be adapted to handle any kind of geometry file containing the necessary information.

In the 2D version, the smallest kind of information held is the vertex, and the highest is the edge. Since the shape will only be recognised in 2D, the original polygon could potentially have inside edges [**Fig. 1.0**] that are unnecessary during collision detection, and therefore will have them removed and discarded from the model [**Fig. 1.1**].

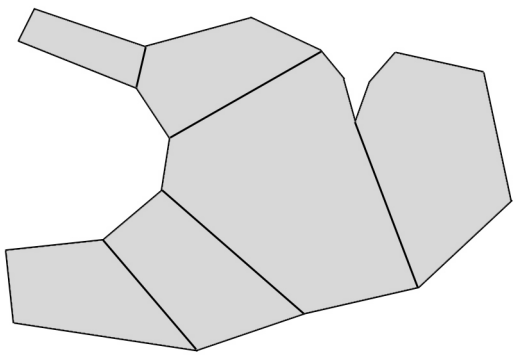

Figure 1.0: A polygon that has been modelled with inside edges.

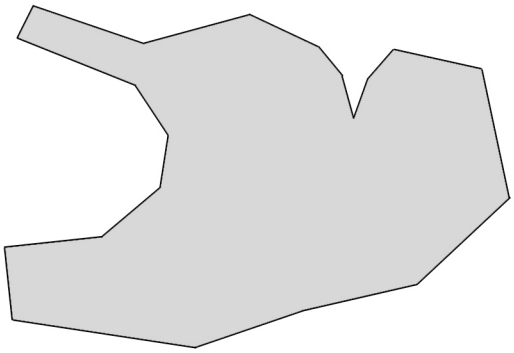

Figure 1.1: The polygon has had all unnecessary inside edges removed, retaining the important outer shell.

Initially, every edge has a normal vector established for it. This normal will be perpendicular to the direction vector of the edge and be directed outwards from the model, thus defining a continuous boundary for the polygon. To do this, a line is cast from the centre of the edge at right angles to it [**Fig. 1.2**]. It is tested for intersections (Comninos, 2006) against all other edges on the polygon to find the orientation of the normal for that edge [**Fig. 1.3**].

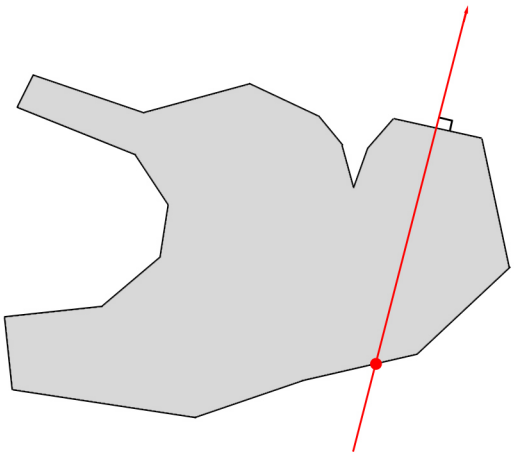

Figure 1.2: The active edge at the top has a perpendicular line cast and tested for intersections against all other edges in the polygon

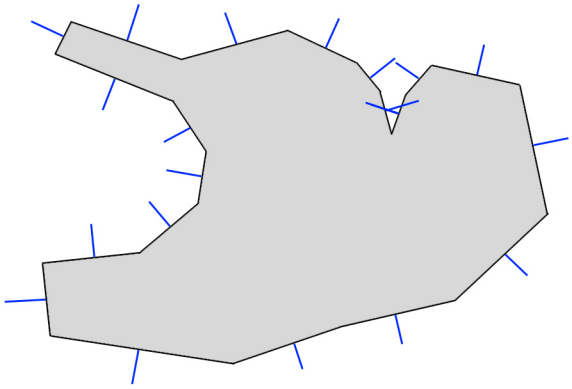

Figure 1.3: A line cast from the edge that has an odd number of edge intersections to one side of it will indicate that side to be the on the inside. The blue lines now indicate the correctly orientated normals.

For each edge, two indices to the vertices that construct it are established, and in addition to it's normal vector, a direction unit vector along the edge is calculated, and edge length. This direction unit vector is always calculated from the first constructive vertex towards the second. Each vertex also has two indices to the two edges that surround it. Next, the vertices "type" is established. This is based on the orientation of the two edges surrounding it, and leads to a vertex being named as neutral (the surrounding edges have parallel normals), concave (the surrounding edges have inward facing normals) or convex (the surrounding edges have outward facing normals). The final important addition necessary for each vertex is a so-called "safe point" (SP). Since there is no need to test vertices of concave or neutral type for intersections, they need not have SP's calculated. The SP is calculated by first casting a line along the normalised average of the two edge normals surrounding it. The line is then tested for intersections against all other edges in the model to find the nearest. This point is then stored as the vertices safe point [**Fig. 1.4**]. All of this data is required and now ready to perform collision detection between two geometric models of any type [**Fig. 1.5**].

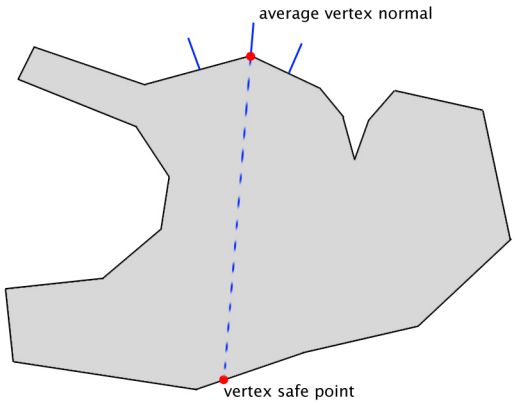

Figure 1.4: The convex vertex at the top has first an averaged normal calculated between its two constructing edges, then an SP found on the far side of the polygon.

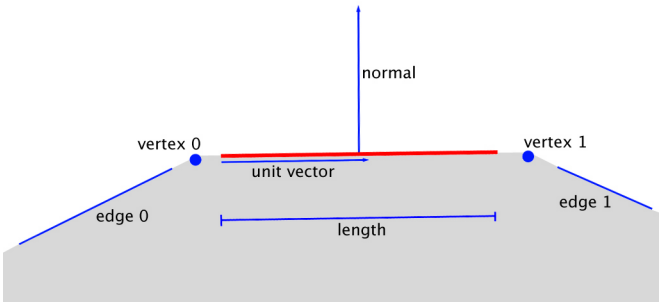


Figure 1.5: The information and indices held by an edge in a 2D shape, highlighted in red.

In 3D, the lowest kind of information held is the vertex, then the edge, and the highest is the face. Each of these entities is stored within separate arrays and contains indices to the other entities that are associated with them. Faces are always considered triangulated, and any that are not will be truncated to the first three vertices associated with them. They also store indices to the three edges from which they are constructed. Faces have a unit length normal vector calculated which is perpendicular to their infinite plane [**Fig. 1.6**]. As with the 2D version, each edge stores indices to it's two vertices, and has a unit length vector and length calculated for it between their first and second vertex. They also have indices to the two faces that surround them.

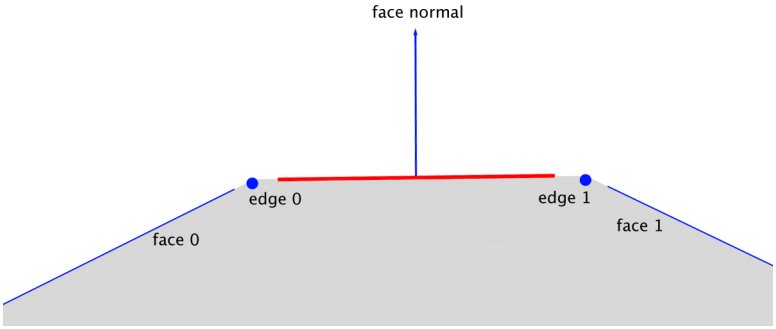


Figure 1.6: A side view of a face, highlighted in red, showing the first two of its three connected vertices and edges.

As part of the collision detection system, each face is required to hold a collection of other normals that are used at certain stages. There are six in total, two for each relative edge attached to it. The first, "edge face" (EF) normals [**Fig. 1.7**], are calculated by finding the vector product (Comninos 2006) of the face normal and edge unit vector, and then orientated so that it directs outward from the centre of the face.

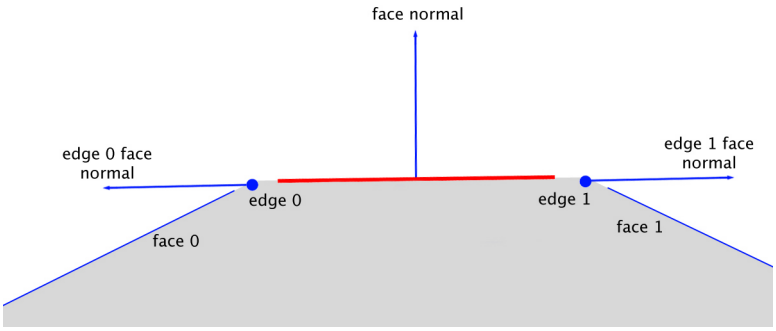


Figure 1.7: The EF normals are parallel to the plane of the face, and perpendicular to each edge of that face respectively.

The second normal, "edge face safe" (EFS) normals [**Fig. 1.8**], are abstract in orientation. Just as vertices had their "type" established in the 2D version, edges have their type established in the same way, except the surrounding face normals are used to determine them instead. Neutral and Concave edge need not have EFS normals calculated for the same reason. For all convex edges on the face, EFS normals are calculated by first finding the average of the two face normals surrounding them. The EFS normal is then calculated by finding vector product of the edge unit normal and the averaged face normals, and then orientated so that it is directed away from the centre of the face.

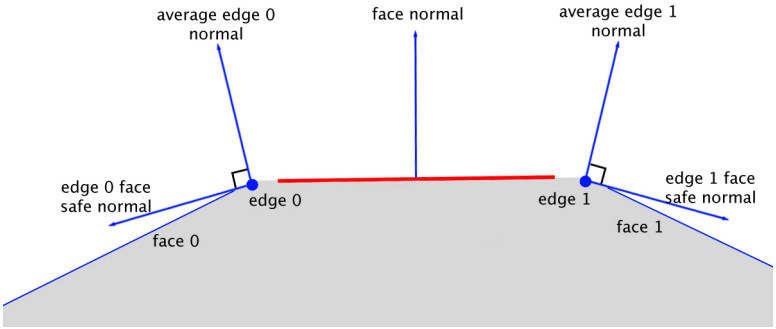


Figure 1.8: The EFS normals are a result of the linear collision detection methodology used in real-time processing. Since the objects position would be integrated by a fixed step, often they find themselves intersecting in awkward ways, requiring predictions to be made as to how they got there.

At this stage, there is now enough data to perform accurate collision detection between two complex shapes and retain physical and collision realism, since appropriate contact normals and depth can be calculated should a collision occur. This is outlined in the following explanation of the 3D collision detection function. This function occurs inside the program that features the physical environment, a program that also imports the data generated by the analytical program previously explained.

## 3.0 Calculation

For every potential case to be covered, during the testing of one “proxy”

shape against another (a polygonal shape of any kind), every convex edge on the first proxy must be tested against every face on the second, and every convex edge on the second must also be tested against every face on the first, for intersections. Mathematically, these test are very simple, but also very numerous.

There are a few functions used in the 3D detection system. The first is short, and merely calls another that tests edges of the first argument against faces of the second. It makes this call twice with reversed arguments in order to cover the clause mentioned previously, ensuring all cases are covered.

This next function then loops through every edge on proxy one, and then with a nested loop, through every face on proxy two. Within both these loops, it calculates the signed distance of the start point and end point of the active edge from the plane of the active face. If there are differences in the polarity of the distances, there is a collision. Otherwise, the face is iterated and the distances re-calculated.

In the case of a collision, the intersection point of the line and the plane is calculated. The intersection point is then tested thrice (once for each edge) to find if it is inside the boundaries of the polygon. This stage uses the EF normals, in which for each edge, the distance of the intersection point is calculated from a plane defined by a point on it and its EF normal. If the distance is greater than zero on any one of these three edge tests, it is outside the region of the polygon, and so the loop iterates as there is no collision.

If it is inside, the distance of the point that was behind the plane (causing the collision) is retained and a new point is project from the intersection point on the surface of the plane inwards along the face normal to the depth of the penetrating point. This point is therefore always going to lie inside the infinite prism-shaped projection of the face, no matter how deep the intersection. At this stage, a collision is guaranteed. Therefore the normal of retaliation and penetration depth must be found. Using the EFS normals, the projected point is now classified, by finding its distance from another set of three planes. A point on each edge of the face and its EFS normal defines these planes respectively. If the position of the point is behind each of the three planes, then the point must be hitting the front of the face, and so can be rejected along the faces normal. Otherwise, the collision must be shearing off one of the edges to the face, and will have an altered angle of retaliation based on whichever edge it is closest to. **Figures 1.9** through to **2.2** highlight the problems encountered with the system before the EFS normals were implemented.

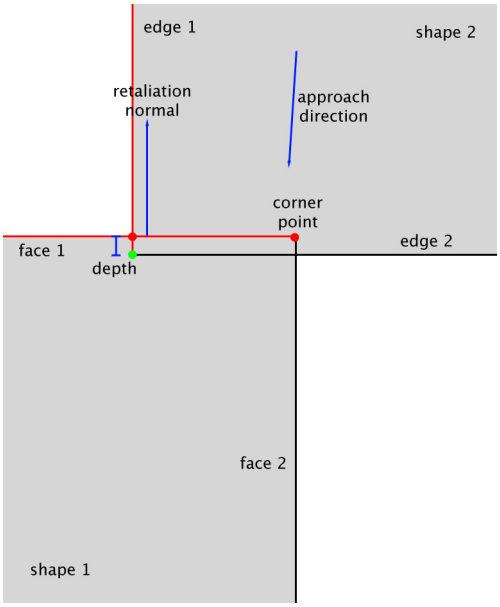


Figure 1.9: Under these circumstances, it is clear that shape 2 will be rejected in the correct manner. The intersecting point is shown in green. The active edge and face are highlighted red.

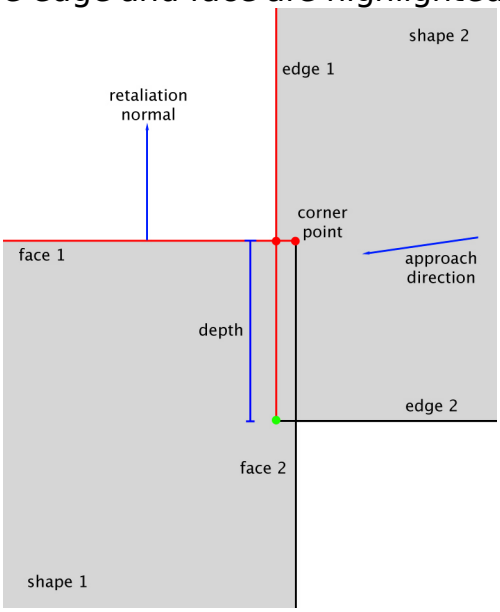


Figure 2.0: However, if the edge and face were evaluated in the same order when the angle of incidence had changed, an extortionate depth would have been calculated, hurling the object outward with not only excessive force, but also in the wrong direction.

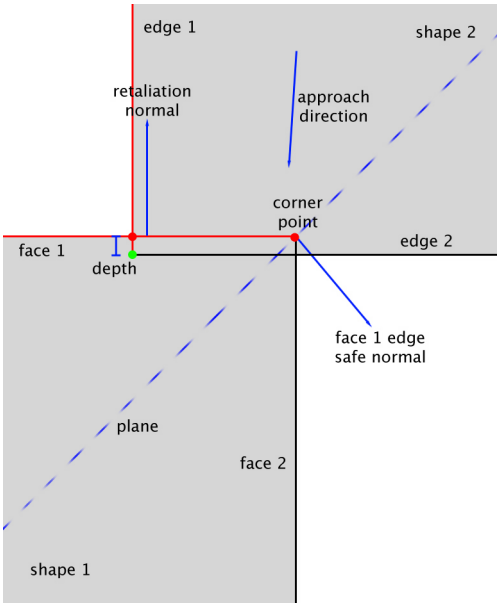


Figure 2.1: Using the extra evaluation granted by the EFS normal, the green point is further checked. In this instance, it remains behind all of the planes created, and so continues to be rejected properly.

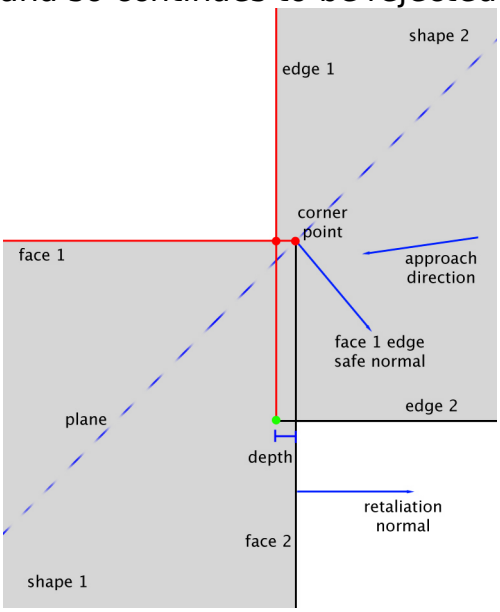


Figure 2.2: Now it can be seen that since the colliding point is in front of the EFS normal here, it will be expelled with the normal of a different face other than that which it was detected to be colliding with. This is made available with the EFS normal plane checks, and made possible due to the face carrying indices to the other faces around it.

## 4.0 Response

Currently, the data type is capable of performing collision detection between two objects, but it is extremely slow due to the exhaustive nature of continually testing every edge and face. At this stage, it would be possible to generate a structural tree that organises the polygons enabling faster evaluation. Despite the obvious and extensive advantages of doing this, the algorithm is likely to begin the same type of polygon analysis once the final colliding regions have been discovered.

It is this stage of the processing that the patching mechanism (PM) is proposed for. The PM is also therefore scalable and intended to be completely general, potentially being implemented underneath any type of more complex structuring system.

Due to the nature of the PM, it increases running speed but still sees virtual shapes being able to collide as explained without any degree of degradation based on stand-in shapes or geometry approximation/simplification. It can analyse geometrical models in either a 2D or 3D sense, building on the evaluative programs explained earlier. It does this prior to their use in the collision detection algorithm. When first imported into the pre-processing program, the geometrical information can be considered a "polygon soup" (Lin and Gottschalk 1998, p.3), where there is no greater knowledge about connectivity, and each polygon is considered a separate independent entity.

Based on a set of rules, the PM can create up to 3 different types of geometry patch. A patch can simply be described as a collection of polygons that are procedurally grouped based on these rules. The types of patch are concave, convex and mixed.

In 2D, the patches are generated at an edge level. Before the PM is run, each edge must have it's "type" confirmed. An edge that is defined by two concave vertices will have type concave. If two convex vertices define it, it will have type convex. If it is defined by any combination of concave and convex vertices, its type will be mixed. Any combination of one neutral vertex and one concave or convex vertex will see the edge type becoming that of the non-neutral vertex, and any purely neutral edge will by default become convex. Once this has been established, it is possible to begin the PM [**Fig. 2.3**]. The rules which apply are as follows: Each edge begins as an individual patch of it's own; The process will continue indefinitely until a cycle where no more changes to the patch families occur; Two neighboring edges of the same type should become of the same patch, and thus the size of the patch will grow out until all connected edges of the same type on an edge strip will be joined; An edge will be removed from a patch it has become part of if: It is part of a concave patch yet there are more than no vertices of any other edge in it's patch family which are behind the plane defined by a point on the edge and it's normal, or it is part of a convex patch yet there are more than no vertices of any other edge in it's patch family which are in front the plane defined by a point on the edge and it's normal, creating a so called "allergy".

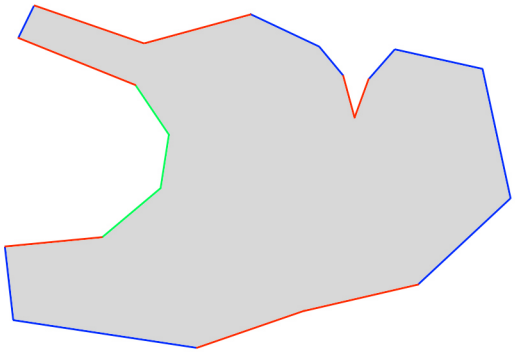

Figure 2.3: Green indicates concavely grouped edges, blue indicates convexly grouped edges and red indicates mixed edges.

In 3D patches are generated at a face level, and the same rules apply to the construction, but instead of vertices defining face patching, it is the edges of the face. Allergies also occur in the same way, except the planes are defined by a point on the plane of the face and the normal to the face.

The PM is completed for the edge or face respectively, and the data is written to an output file, which is then imported by the simulation program, where each patch family is stored as an individual dynamic array of indices to the elements in the patch. Therefore, and object may consist of many different types of patches. These programs are named ‘makeNSF2’ for 2D shapes, and ‘makeNSF3’ for 3D shapes.

In order to increase efficiency, many algorithms require closed manifold convexity (Lin and Gottschalk 1998, p.3) of a polygonal object, or exploit this aspect. Whilst this will naturally increase efficiency due to ease of discarding, it is very likely that the integrity of the original object will have been lost in the process of adapting it's shape to enable this closed convexity, since objects are rarely convex in every way. However, using the PM, every detail is maintained whilst the object is also adapted to discard in the fastest possible way.

Within the collision detection function therefore, a few structural changes are made to the loops. The original edge loop is retained, but the face loop is replaced with three loops. They operate on concave, convex and mixed patches respectively, and within these outer three loops is contained one more to iterate over the faces of the patches. The other processes stay intact, except now, iterations of the loop can be entirely skipped in the correct circumstances: If both vertices of the edge being tested are found to be behind a face which is in a concave patch, every other face in the patch can be skipped; If both vertices of the edge being tested are found to be in front of a face which is in a convex patch, every other face in that patch can be skipped.

# 5.0 Restitution

This system allows for sub-structured tree discarding, with absolutely no loss of object shape quality. A branch of a structured tree could now contain not just the polygon soup it would have before, but an organised set allowing for minimal CPU effort once the correct node had been discovered. The nature of the algorithm is such that it recognises the outside of an object to be so due to the direction of the normals of the edges or faces in 2D or 3D respectively. Fascinatingly, if you simply interpret the normal direction to be inverted, thus substituting the actual outside for the physical inside, and merely reverse the algorithm so that it uses what were previously concave patch handling methods with convex ones and vice-versa, the system runs in perfect symmetry, allowing for multi-sided collision objects without any kind of additional processing, beautifully adopting the design philosophy "form follows function".

The algorithm is in fact actually efficient enough to run in real-time without any kind of structural tree yet implemented, paying homage to the design. The 2D version is considerably faster due to the nature of the simpler calculations but the 3D is still quite capable of maintaining high frame rates with simple objects. It has been tested to compare its impact on performance under an altering physics time step. A smaller time step is most desirable since it will mean the simulation will retain accuracy, but will also mean increased collision detection cycles, reducing the frame rate. It was first tested with a stand-in spherical collision shape that uses a simpler collision detection function with very similar principles as the former [**Fig. 2.4**], then with the new algorithm [**Fig. 2.5**].

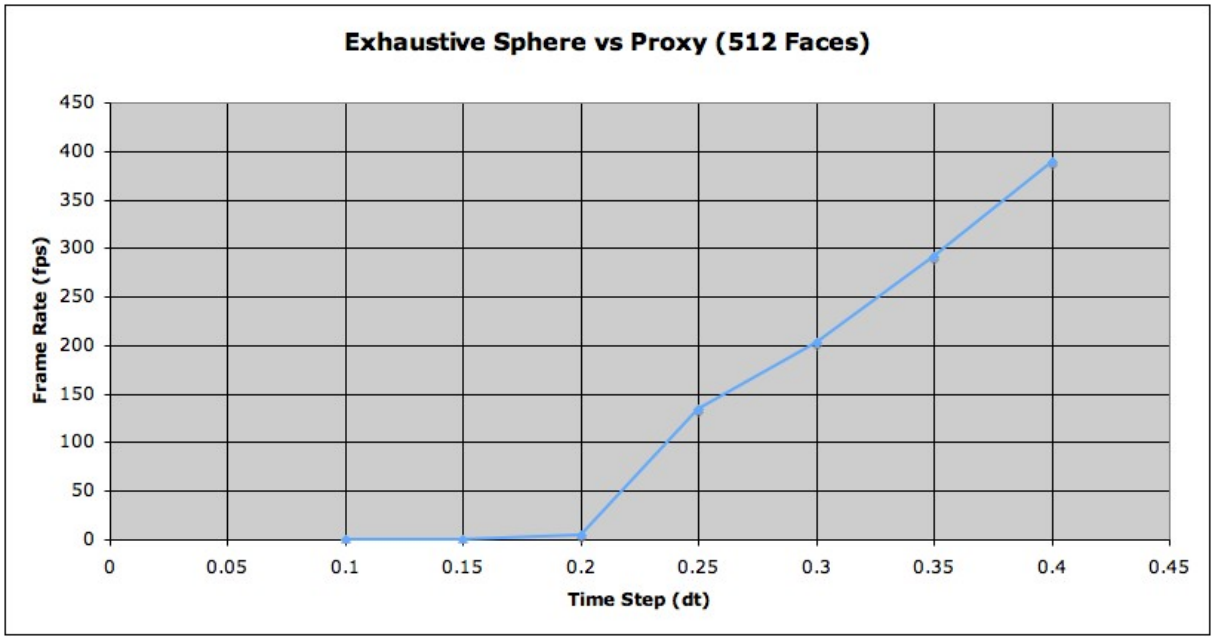


Figure 2.4: A sphere tested against a proxy shape with 512 faces. The sphere was tested against every single face, every convex edge and every vertex despite of its position and arrangement.

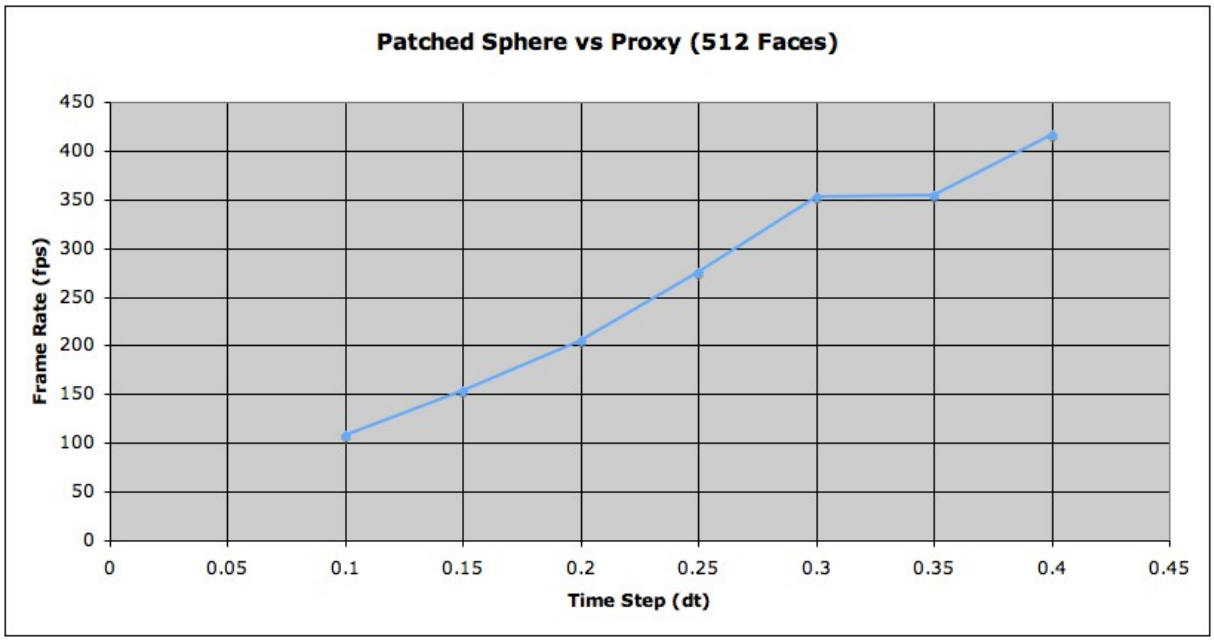


Figure 2.5: The same objects were tested using the new algorithm, which will discard at every opportunity. In the same time step scale, much higher frame rates were attained.

It was also tested with two proxy shapes with low resolution geometry [**Fig. 2.6, 2.7**].

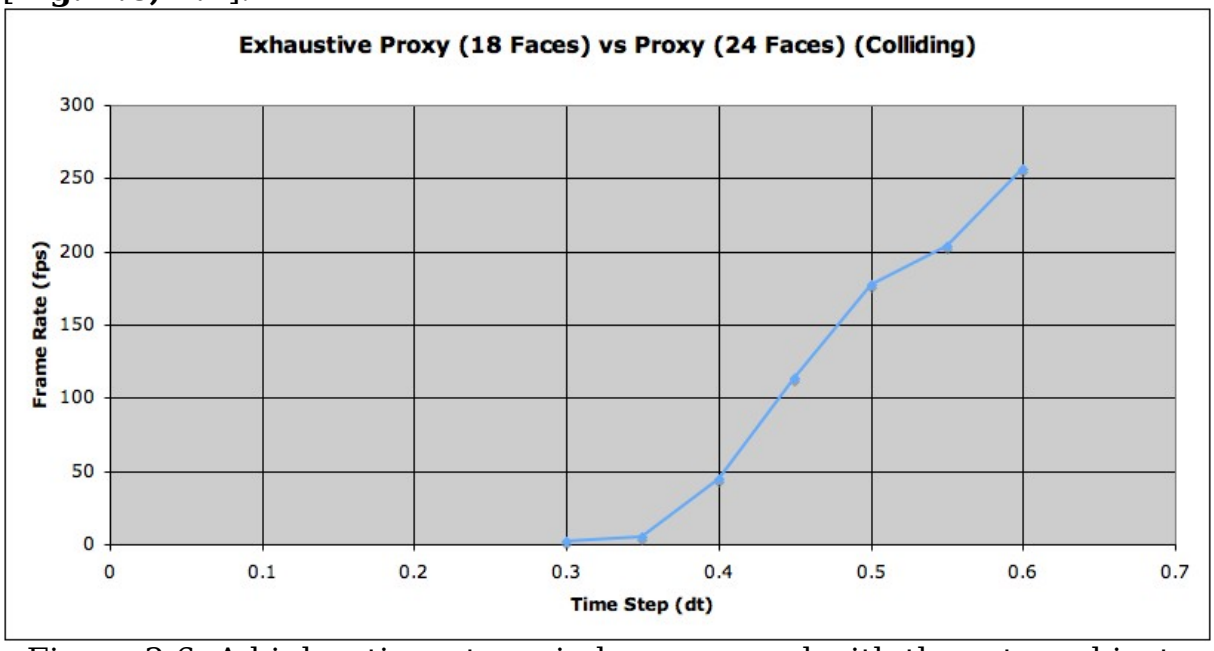


Figure 2.6: A higher time step window was used with these two objects since it is a much heavier process.

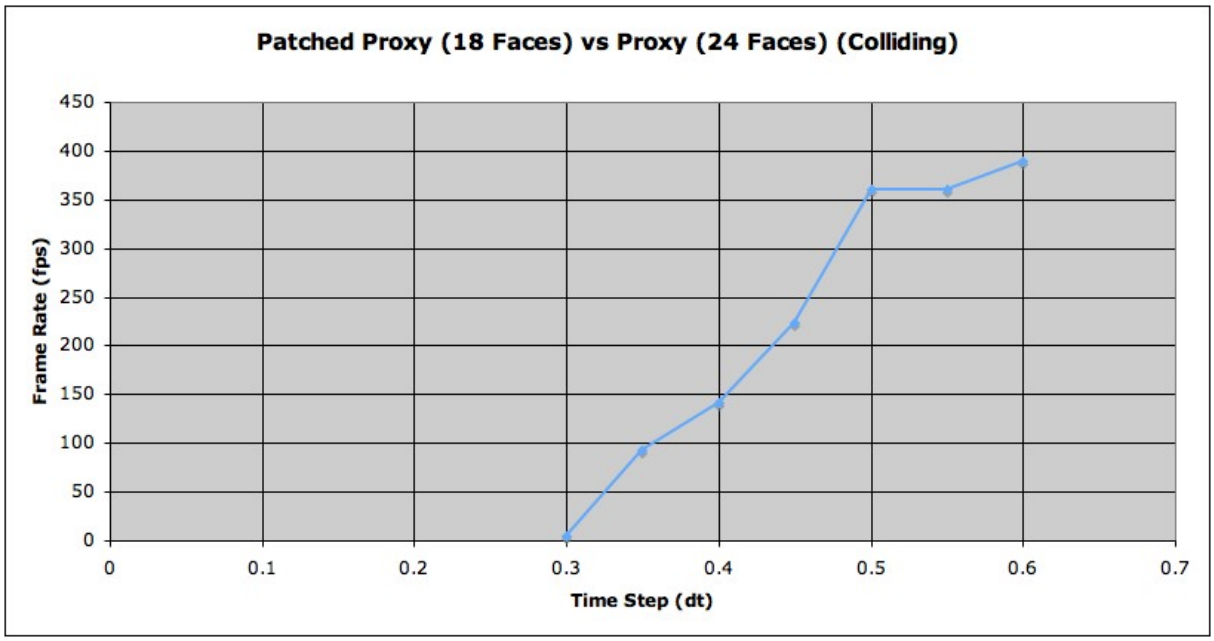


Figure 2.7: The same two proxy objects now using the discarding algorithm.

## 6.0 Conclusion

As the testing has proven, there are some significant performance gains when using the algorithm. Not only is the overall frame rate able to climb higher, but also it seems the degradation of frame rate over a decreasing time-step is made smoother and more even in some cases. Whilst it is clear that there are some obvious advantages to the performance gained by the system, there are also some unfortunate side effects due to the level of pre-processing. Also, there are assumptions made within the PM that seem unavoidable, which are not always the most logical solution when observed. One of the most notable problems encountered in the PM that can cause problems and result in a large drop in later efficiency are the handling of faces/edges with neutral type. Imagining a collection of neutral faces all geometrically connected, it is likely that they should be placed in a convex patch since this is the fastest to discard. However, this assumes that the algorithm will be interpreting the actual outside as the physical outside, but if it were inverted as discussed, the patch would instead be considered concave and result in a performance pit. Similarly, imagining a collection of neutral faces and concave edges geometrically connected, it is likely that since the concave faces will spread and occupy the neutral faces amongst them, the resulting patch will be a larger concave patch as opposed to what instead could be a concave patch just incorporating the concave faces and a convex patch incorporating the neutral faces, again making the assumption that the actual outside of the shape will be interpreted as the physical outside.
It is likely that various other solutions could be found to solve these problems, but it is unlikely that a level of assumption will never be needed or used, very often during the handling of neutral edges/faces.

Another disadvantage of the system is the memory footprint that the data

structure will have because of the large amount of pre-processed data required when running the true polygon-polygon collision detection system. In order to properly detect and analyse a collision, as seen in the explanations of the algorithm, a large amount of data is called upon to try and resolve the issue as quickly as possible. This leads to a common pitfall of more complex systems whereby if additional performance is gained then the memory requirement is likely to increase. To confirm this, it is of note that many of the normal vectors generated, for example in 3D the EFS normals, could be calculated during the collision detection process and not stored in memory, thus freeing what is currently three 3D vectors per face of the object.

To sustain a level of mathematical certainty, the 3D system is also currently clamped to the use of triangulated faces, since faces of any other number of vertices can be constructed in a non-planar fashion. Whilst it is safest to use triangulated faces for this reason, it often also causes unnecessary performance overheads. For example, a cube within the collision system, at it's lowest possible topology, will need to have 12 triangulated faces in order to have closed manifold (Lin and Gottschalk, 1998). Whilst this is also very safe, it is far more sensible to assume four sided faces and instead of 12 use 6. Simply put, limiting the system to triangulated faces will very commonly cause an excessive bloat in the testing bottleneck. The algorithm could be redesigned to incorporate faces of four sides, if sufficient validation systems where in place prior to the PM that would reject individual faces which do not have a convex boundary and planar construction qualities.

Since the system is capable of handling any kind of object, it is possible to create very awkward interactions. Adding to problem is the truth that simulations will need to use a fairly high time step in order to sustain a real-time environment. This can circumstantially cause objects to intersect in way that simply cannot be properly resolved, resulting in “explosions” of undefined collision depth and direction. Ideally the time step should remain tiny in order to prevent this, but that is rarely possible. Therefore the system only compensates to a feasible degree.

# References

[1] Fiedler, G. 2007. *Gaffer on Games.* Available from: www.gaffer.org. [Accessed 01.01.2008, Unfortunately the website is currently under re-design and missing certain pages concerning physics and time steps].

[2] He, T. 1999. *Fast Collision Detection Using QuOSPO Trees. In:* Symposium on Interactive 3D Graphics Atlanta. 55-62.

[3] Lin, M.C. and Gottschalk, S. 1998. *Collision Detection Between Geometric Models: A Survey.* University of North Carolina.

[4] Giang, T. and O'Sullivan, C. 2006. *Approximate Collision Response Using Closest Feature Maps.* Computer and Graphics 30. Image Synthesis Group, Trinity College, Dublin, Ireland.

[5] Wavefront 1995. *The OBJ Specification.* Available at: http://netghost.narod.ru/gff/vendspec/waveobj/obj_spec.txt [Accessed 12.03.2008]

[6] Comninos, P. 2006. *Mathematical and Computer Programming Techniques for Computer Graphics.* 84-86,109-110. London: Springer-Verlag.

[7] Arbib, M.A. and Alagic, S. 1979. *Proof Rules for Gotos. In:* Acta Informatica. Vol: 11/2. 139-148.

[8] Miroshnichenko, E. 2003. *Structured Programming: Myths and Facts.* Computer Engineering Dept. Tomsk Polytechnic University. In: Proceedings Korus 2003. 396-400

[9] Vince, J. 2005. *Mathematics for Computer Graphics*. 2nd Ed. London: Springer-Verlag.

[10] Shreiner, D and Woo, M. 2007. *The OpenGL Programming Guide.* 6th Ed. Addison Wesley.