Contact generation for meshes

Pierre Terdiman - v1.2

1) Introduction

This paper presents an algorithm for generating good mesh contacts for rigid body simulation. In particular, it explains how to efficiently prevent invalid contacts with “internal edges”. The algorithm is robust, simple, and memory-efficient.

We use a sphere-vs-mesh example but the same idea is applicable to other shapes.

2) Previous approaches

There have been a lot of different approaches used by different physics engines over the years to generate mesh contacts. It always starts with a set of candidate triangles returned from the *midphase* (i.e. the set of triangles from the mesh that the sphere overlaps for a given frame). But what one does afterwards with these triangles varies a lot for each algorithm.

The first approach is to always use the triangle’s normal for a contact normal. This works well for flat connected triangles (in particular it does not generate contacts with internal edges), but it completely fails in other cases (e.g. see (2) for the “cliff problem”).

The second approach is to compute the closest point between the sphere’s center and the triangle, and use the resulting vector as a contact normal. This works better but it creates undesired and unrealistic motions when the sphere approaches triangle edges, and generates invalid contacts with these internal edges. This is sometimes referred to as the “fear of the wireframe” problem.

The third approach is to take advantage of the render normals (as defined for rendering), and use them for contact normals (1). This produces smooth motions over flat areas of the mesh but also produces undesired and unrealistic motions in some cases. For example a sphere resting on top of a large triangle can start moving towards its edges, because of the triangle’s smoothed vertex normals. But the correct contact normal in that case would be the triangle’s (exactly vertical) normal. The same issue happens for rendering, where smoothing groups are necessary to compute the correct normals and get the correct lighting for objects as simple as a cube. The paper (1) ignores this, and always uses the smoothed normals, producing invalid motions as a result. The approach also increases memory usage for storing these normals, which are otherwise not needed and usually not available in the physics data structures. Sharing them with rendering is not an option since rendering uses non-smoothed normals anyway (a single vertex can have multiple vertex normals here). It might be possible to improve the approach to also use multiple vertex normals, but it is unclear how to do so.

The fourth approach is to use *Voronoi regions*, as in the Bullet physics engine (2). However it is unclear whether their implementation always works correctly since it uses various epsilons and thresholds to tag convex edges, which is usually unreliable. The approach also requires a costly preprocessing step and extra memory to store the precomputed data.

3) Meqon idea

The idea presented in this paper also uses Voronoi regions, but in a fully implicit way. The algorithm only needs vertex references. It does not need a preprocessing pass. It does not rely on fragile convexity thresholds or convex decomposition of the mesh. It does not need extra memory to store additional data. As a result, the code is robust, fast, and memory efficient.

As far as we can tell, it was originally used in the Meqon physics engine. A rough description can still be found online (3), although it is not very clear, and it misses several details that we found are necessary to make the approach work reliably. For example it is unclear at this point if the refinements from Chapter 5 already existed in Meqon's implementation. Generally speaking the details of their implementation are unknown, but what we can say is that the referenced quote (3) was definitely the inspiration for the algorithm described here.

4) Initial implementation

a. First pass: face contacts

The midphase reports triangles touched by the sphere **S**, one triangle at a time. Let **Sc** be the sphere's center in mesh space, and **Sr** the sphere's radius. We backface-cull all triangles w.r.t. **Sc**, i.e. if **Sc** lies in the negative part of the triangle's plane, then the triangle is ignored.

For each incoming triangle we further compute:

- The distance **D** between **Sc** and the triangle.
- The closest point **Cp** between **Sc** and the triangle.
- The feature code **Fc** for the **Cp**, i.e. we determine if the closest feature of the triangle is a vertex (figure 5), an edge (figure 3), or a face (figure 2).

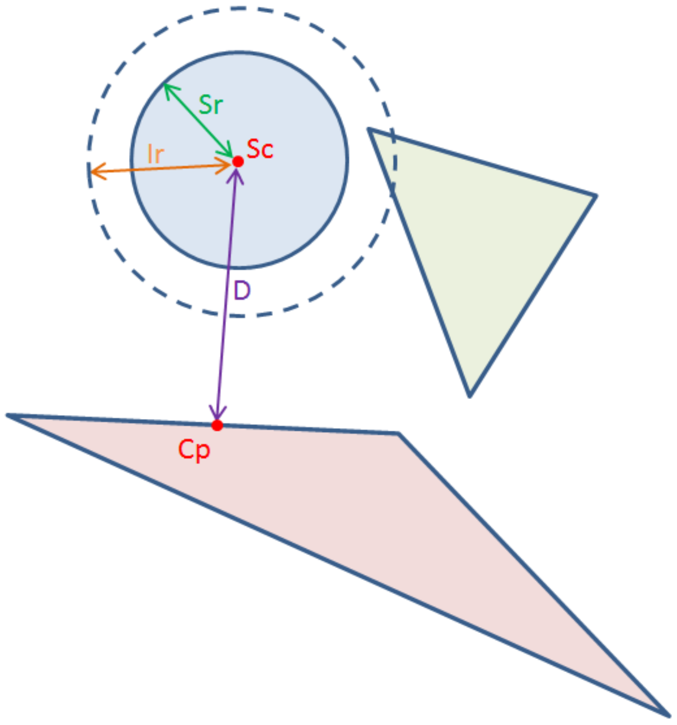

Figure 1: red triangle is ignored (D > Ir)

We define an inflated radius **Ir** = **Sr** + a user-defined contact distance. If **D > Ir**, the triangle is ignored (Figure 1). Otherwise we have a *potential contact*.

If this potential contact is a face contact, we keep it and register a rigid body contact immediately. In this case the contact normal is the triangle's normal. The contact point is defined by the intersection between the sphere's surface and the line joining **Sc** and **Cp**. The contact depth is the difference between **D** and **Sr**.

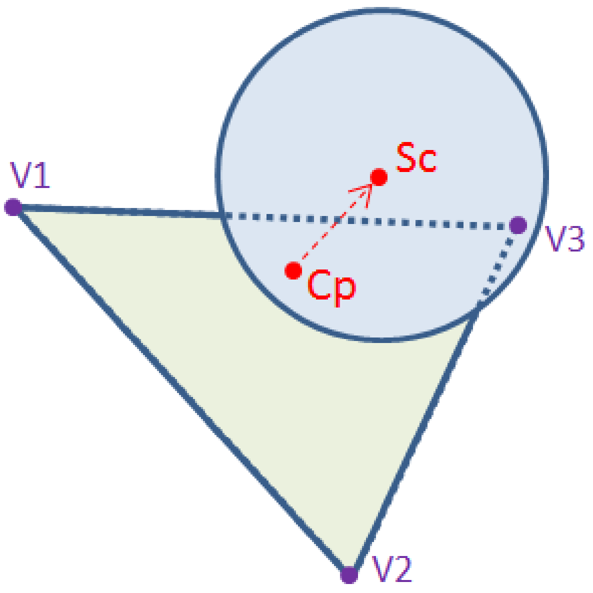

Figure 2: face contact

If the potential contact if not a face contact, we record the relevant data for further processing – we do not register a rigid body contact yet.

For each face contact we also record the vertex indices (V1, V2 and V3 in Figure 2) of the triangle for which the contact has been generated. This dataset **V** (for *Voided* features) will be used in subsequent passes to validate or ignore edge and vertex contacts –a.k.a. *delayed contacts*. **V** only contains triangles. An edge contact will be voided if any of the triangles in **V** contains the edge. A vertex contact will be voided if any of the triangles in **V** contains the vertex.

All face contacts are kept. There are no triangle configurations in which ignoring a face contact is the correct thing to do.

b. Second pass: edge contacts

After the midphase finishes, and all face contacts have been generated, we go over the set of delayed contacts that we recorded in the first pass. In this second pass we only consider edge contacts.

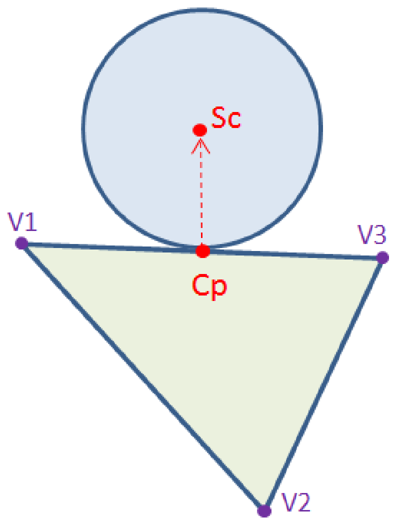

Figure 3: edge contact

Each edge contact references two vertex indices - for example V1 and V3 in Figure 3. If the set of voided features **V** contains (V1;V3), i.e. if that edge belongs to one of the triangles for which we already generated a contact, then the potential edge contact is ignored (Figure 4).

Otherwise we register a rigid body contact as we did in the first pass, but with the normalized vector between **Sc** and **Cp** as a contact normal. If that vector is null, i.e. if **Sc** and **Cp** are the same, then we instead use the triangle's normal as a contact normal.

The triangle containing the edge is then unconditionally added to **V** (whether or not a new rigid body contact has been created). These new additions to **V** will be used to validate or ignore vertex contacts in the next pass.

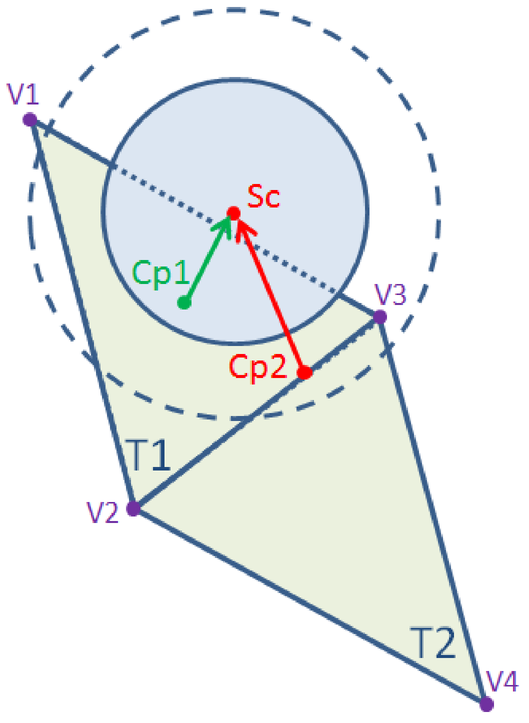

Figure 4: "fear of the wireframe" issue

Figure 4 shows the typical “fear of the wireframe problem” and how the algorithm deals with it. We first process triangle T1, which has a face contact (closest point Cp1 is on the triangle’s interior). Vertices V1, V2 and V3 are added to **V**. In the second pass we process triangle T2, which has an edge contact (closest point Cp2 is on the edge V2;V3). Since **V** contains the V2;V3 edge, the edge contact is ignored.

We see here why it is important to process face contacts first: if we would have processed triangle T2 first, the dataset **V** would have been empty, and the edge contact would have been kept.

c. Third pass: vertex contacts

After the second pass finishes, and all edge contacts have been generated, we go over the set of delayed contacts that we recorded in the first pass. In this third pass we only consider vertex contacts.

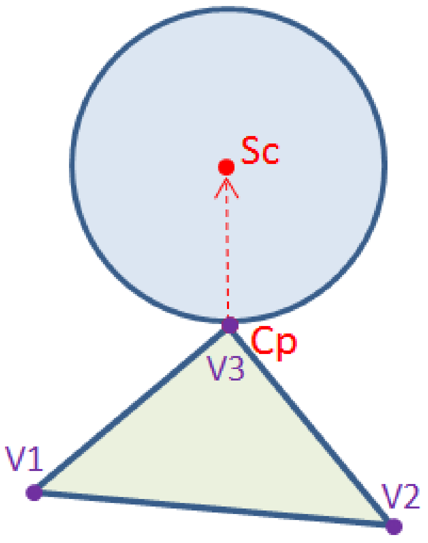

Figure 5: vertex contact

Each vertex contact references one vertex index - for example V3 in Figure 5. If the set of voided features **V** contains V3, i.e. if that vertex belongs to one of the triangles for which we already generated a contact, then the potential vertex contact is ignored (Figure 6).

Otherwise we register a rigid body contact as we did in the second pass. The triangle to which the vertex belongs is also unconditionally added to **V** (whether or not a new rigid body contact has been created). These new additions to **V** will be used to ensure that a given vertex does not generate more than one rigid body contact.

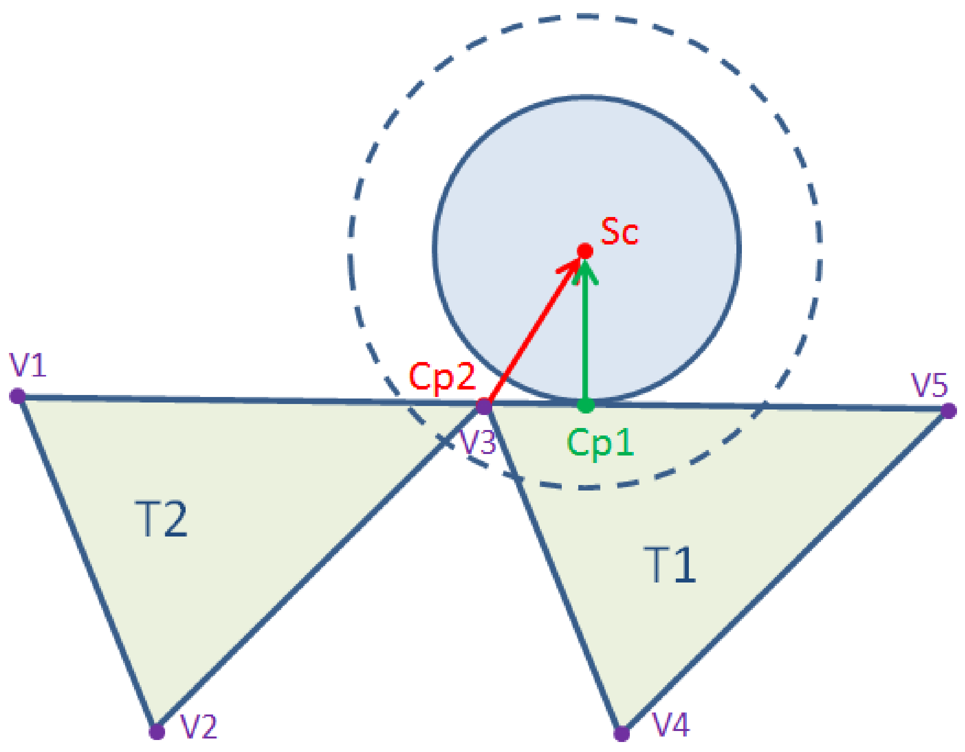

Figure 6: ignored vertex contact

Figure 6 shows how the algorithm works for edges and vertices. We first process triangle T1, which has an edge contact (closest point Cp1 is on the edge V3;V5). Vertices V3, V4 and V5 are added to **V**. In the second pass we process triangle T2, which has a vertex contact (closest point Cp2 is equal to V3). Since **V** contains the V3 vertex, the vertex contact is ignored.

We see here why it is important to process edge contacts before vertex contacts: if we would have processed triangle T2 first, the dataset **V** would have been empty, and the vertex contact would have been kept.

5) Improved implementation

The algorithm as proposed above works fine most of the time, but it still produces invalid contacts in certain cases. This chapter explains the problems, how to solve them, and also why it was necessary to update **V** unconditionally in the edge and vertex passes.

a. Large contact distance

The initial implementation sometimes generates invalid contacts, especially when large contact distances are used.

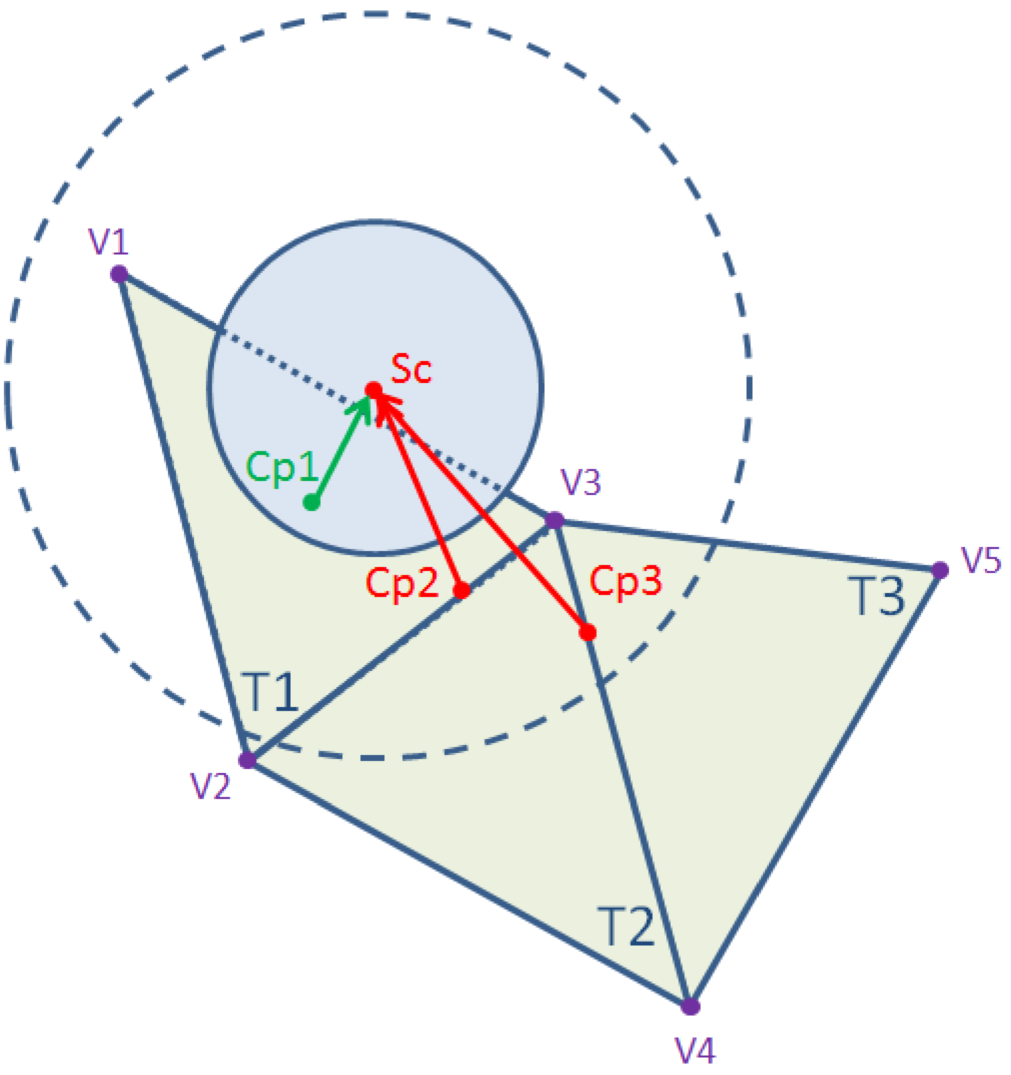

Figure 7: large contact distance

In figure 7, we receive 3 triangles from the midphase. Some of them are quite far from the sphere, due to the large contact distance involved. Triangle T1 is a face contact, processed first. Vertices V1, V2 and V3 are added to **V**.

Triangles T2 and T3 are delayed edge contacts. As such, they are both processed in an arbitrary order in the second pass. If we process T2 first, we see that Cp2 is on the V2;V3 edge, which is contained in **V**, and thus the contact is ignored. If we do not update **V** with triangle T2 in that case, then when we process T3 we see that Cp3 is on the V3;V4 edge, it is *not* contained in V, and the contact is kept. Which is wrong. However if we would have updated **V** with T2, it would contain the edge and the T3 contact would be ignored. Thus, we need to update **V** with triangle T2 even if T2 does not generate a contact.

Now this works if T2 is processed first. But if we process T3 first, then **V** does not contain the V3;V4 edge yet, and the contact is wrongly kept. A simple solution to make sure that T2 is processed before T3 is to sort delayed contacts according to the distance **D** before running the second and third passes.

Moreover, sorting the contacts has the nice side effect that vertex contacts usually end up after edge contacts. So once the sorting is done, the second and third passes can actually be merged into a unique “delayed contacts” pass.

It would actually be possible to handle all contacts in the same “contact generation pass” (i.e. the face contacts as well), but that would require storing all triangles from the midphase first, then sort everything, then generate the contacts. It works, but it increases the memory needed for storing temporary data, and it increases the cost of sorting contacts (since we have more contacts to sort). It is more efficient to handle face contacts on-the-fly.

b. No face contact

In regular cases it is correct that processing contacts in the face->edge->vertex order gives the best results. But this is not always the case. In the following singular case (Figure 8), the sphere overlaps all triangles. The sphere’s center is exactly located above the common vertex shared by triangle T1 to T6. In such a configuration the initial pass does not generate any face contacts.

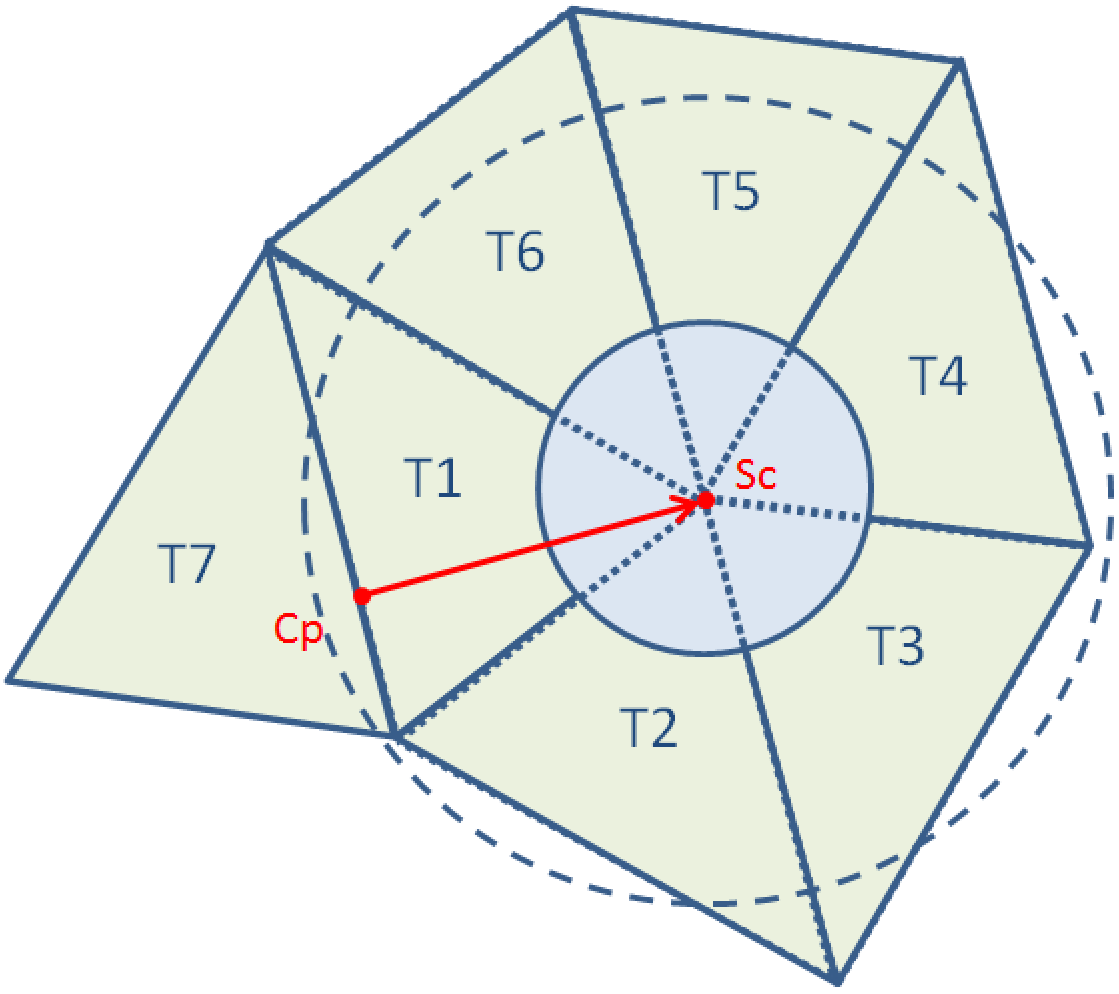

Figure 8: vertex contacts should be first

In the delayed contacts pass we have six vertex contacts from triangles T1 to T6, and one edge contact from triangle T7. Processing the edge contact first would generate the invalid contact Cp on the edge of T7. In this case we actually need to process the vertex contacts first, invalidating the edge shared edge between T1 and T7.

This is a case that clearly fails with the initial implementation. It would also fail with separate vertex and edge passes, even if the contacts are sorted. This case only works with sorted contacts and a unique "delayed contacts" pass, in which the (closer) vertex contacts are processed before the invalid edge contact.

Thus the final (and as far as we can tell, correct) algorithm is:

- Create face contacts in a first pass
- Sort delayed (vertex and edge) contacts
- Process delayed contacts in sorted order, unconditionally voiding triangle features along the way

6) Extension to disjoint meshes

The dataset **V** contains integer vertex indices, which are compared against each-other to determine if **V** contains a given edge or vertex. These integer comparisons are fast, but they only work with clean collision meshes. For example in Figure 7 the algorithm only works if the (V2;V3) edge is properly shared between triangles T1 and T2. This is not a given. These two triangles could reference duplicate vertices, i.e. vertices sharing the same exact location but duplicated within the mesh data structures.

Similarly, these duplicated vertices could in fact belong to different meshes – for example different pieces of a race track in a racing game. The meshes could have been authored that way to make the artists' lives easier, or maybe for technical reasons – e.g. if the different parts of the mesh are streamed in and out of the game's database at different times.

In that case, an edge that visually appears to be shared between two triangles can in fact be a "boundary edge", i.e. an edge belonging to one triangle only for a given mesh. The algorithm would see the edge as such, and would generate an (incorrect) edge contact.

It is easy to make the algorithm work across meshes though.

First, the midphase must be modified so that triangles from different meshes are gathered in a common input set (i.e. the complete set of triangles surrounding the sphere, regardless of which mesh object the triangles belong to).

Then, the algorithm must be modified so that it operates on actual floating-point vertex positions, rather than integer vertex indices. A position is 3 floats, but for exact comparisons they can be handled as their integer binary representation. So the performance cost is relatively low here: 3 integer comparisons instead of 1 previously. Memory requirement for the dataset **V** also becomes 3 times larger, which can be an issue on platforms with limited local memory.

Since all candidate triangles are first gathered in the same input buffer, all contacts are now generated in an independent pass after the midphase. Thus, the face contacts pass can now be merged with the delayed contacts pass, giving birth to the unique contact generation pass mentioned at the end of the 5a paragraph.

7) References:

(1) http://www.crcpress.com/product/isbn/9781568814742, Game Physics Pearls book, "Smooth Mesh Contacts with GJK"

(2) https://**bullet**.googlecode.com/files/GDC10_Coumans_Erwin_Contact.pdf

(3) Peter Tchernev (ex-Meqon, ex-Ageia), Bullet Physics forums. Relevant quote below. http://www.bulletphysics.org/Bullet/phpBB3/viewtopic.php?f=9&t=1814&p=8387#p8387

"The general solution to this problem is to ignore contacts whose normals are outside the voronoi region of the triangle feature that they are located on. It is possible to do it without explicitly computing the voronoi regions by keeping a list of "voided" edges and vertices for each collision pair / detector. For each contact we need to know the closes feature on the the triangle of the mesh. We do this using information from GJK. If a face is the closest feature of the triangle all it's edges and vertices are added to the "voided" sets. If an edge is the closest feature of the triangle all other of the triangle edges are and vertices are added to the "voided" sets. If a vertex is the closest feature of the triangle all other of the triangle edges are and vertices are added to the "voided" sets. Then a pass is made to validate all the contacts. Any face contacts are passed. Any edge contacts that do not appear in the voided edges set are passed and their vertices are added to the voided vertices set. Any vertex contacts that do not appear in the voided vertices set are passed. The tricky bit is to recognize common edges and vertices. Can be done (correctly) using indices if available or (somewhat ad-hoc) using a hash code based on coordinates."