Paper: Convex Covering

Jan “Scymtym” Moringen

Yukari “Shinmera” Hafner

jmoringe@techfak.uni-bielefeld.de

shinmera@tymoon.eu

Shirakumo.org

Zürich, Switzerland

ABSTRACT

A

CCS CONCEPTS

• **Computing methodologies** → **Mesh geometry models**; **Computer graphics**; • **Applied computing** → Media arts.

KEYWORDS

Common Lisp, Convex Decomposition, Games, Video Games, Computer Graphics, Experience Report

ACM Reference Format:

Jan “Scymtym” Moringen and Yukari “Shinmera” Hafner. 2024. Paper: Convex Covering. In *Proceedings of the 17th European Lisp Symposium (ELS’24)*. ACM, New York, NY, USA, 3 pages.

1 INTRODUCTION

Many geometrical applications such as games require efficient intersection tests between various geometrical shapes. Efficient algorithms for such intersection tests such as GJK[2] or MPR[6] require the involved shapes to be convex, however. This is a difficult constraint, as most shapes encountered in nature and in environments designed for games are non-convex. A simple solution to this problem is to compute a convex-hull over the concave shape. However, this creates imprecision in the collision result, which may not always be permissible. Especially for highly non-convex objects such as terrain or building interiors, merely computing a convex-hull is not feasible. Instead, we need to decompose the shape into multiple convex shapes.

Since computing an optimal convex decomposition is NP-hard[5] all used algorithms to perform such decompositions operate on heuristics and various relaxations of the constraints. A frequent relaxation is to allow the resulting shapes to be “approximately convex”. This is usually permissible as the intersection test algorithms still deliver results that are accurate enough even in the presence of very small concavities. Another relaxation is to allow the convex hulls to not have to match the input surface exactly. However, for the case of environment geometry in games, this approximation means that the resulting shapes can produce geometry that leads to collisions that would not be present if a test were performed against the input mesh instead. Such protrusions and inaccuracies can be significant enough to be noticeable to the player.

For efficiency reasons we also desire a couple of additional, competing qualities:

- The number of vertices in a convex hull should be small. GJK degrades in performance as the number of vertices on a convex hull increases.
- The number of convex hulls should be small. If we keep trading vertices per hull for more convex hulls, we instead trivialise the problem and create too many potential candidates for which the costly fine intersection test needs to be performed.
- The size of convex hulls should be small. If the resulting convex hull has a large extent, the broad-phase intersection test will include it for many cases, leading to more candidates on average.

In this paper we present an algorithm that produces a convex decomposition that does not produce additional collisions and can be tuned to optimise the number and complexity of convex hulls. It is also usable as an automatic off-the-shelf component that will generate an acceptable result without manual tuning of its parameters.

2 RELATED WORK

Liu et al.[3]’s work serves as the baseline for our implementation. Unfortunately their descriptions of the algorithm’s details aren’t entirely precise, making it difficult to reproduce their results exactly. We were also unable to find any publication of source code at all, let alone a working implementation.

Mamout et al.[4]’s work and their open implementation, “V-HACD”, provide a high-quality *approximate* convex decomposition algorithm. Their algorithm relies on a voxelisation step, which forces the source mesh into a watertight 2-manifold representation, and introduces deviations from the source mesh’s vertices. This can lead to noticeably different collision behaviour for terrain than the source mesh would produce. It can also easily drastically increase the total number of vertices compared to the source mesh, degrading collision performance. Their algorithm does allow tuning the complexity and number of produced hulls, but doing so requires human evaluation and there is no good general case behaviour.

Wei et al.[7] present a much improved method for approximate convex decomposition that is especially tuned for collision handling. However, their approach is extremely complex and difficult to implement, relying on many other algorithm implementations, such as 2D triangulation and monte-carlo tree search, which we would have had to reproduce in Lisp. Liu et al.'s original work only requires on an implementation of the Quickhull[1] algorithm.

3 ALGORITHM

3.1 Overview

The basic idea of the algorithm is quite simple as illustrated by the pseudo-code in Listing 1

```
(defun decompose (faces)
  (let ((patches (mapcar #'make-patch faces)))
    (loop for candidates = (find-merge-candidates patches)
          while candidates
          do (let ((best (sort candidates #'<
                               :key #'merge-cost)))
               (merge-patches best patches)))
    patches))

(defun find-merge-candidates (patches)
  (loop for patch in patches
        append (loop for neighbour in (patch-neighbours patch)
                     when (merge-possible-p patch neighbour)
                     collect (cons patch neighbour))))
```

Listing 1: A pseudo-code illustration of the basic decomposition algorithm

We consider a set of "patches", each of which forms a convex hull. Initially each face is simply turned into its own patch, after which we try to iteratively merge patches where possible. A merge is only valid if the resulting patch is still convex and, most importantly, does not introduce new faces that would cause collision behaviour to differ. In order to merge the actual patch geometry, we simply run the Quickhull algorithm, which gives us a minimal convex hull of the two input patches.

Since the result of the algorithm depends on the order in which patches are merged, we also perform a heuristic merge cost estimation to select the best merge candidate in each iteration. The exact behaviour of this merge cost estimation has great consequences on the quality of the resulting mesh and the speed at which the algorithm converges.

3.2 Merge Criteria

3.2.1 Vertex and Edge Criterion. Differences from paper:

- Vertices and edges handled in the same test
- Heuristic for "touching" constellation gives rise to tolerance (see below)

3.2.2 Matching Normals Criterion. See figure 1.

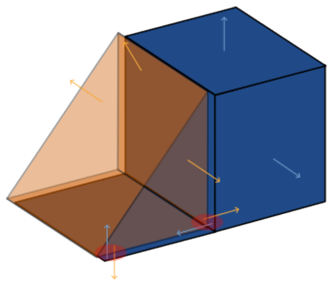

Figure 1: Illustration of the *Matching Normals Criterion*: The orange patch hull (drawn at an offset) is being tested for validity. For two hull facets and two mesh faces, the normals of the hull do not match the mesh normals. The mismatches are indicated in red.

3.2.3 Boundary Constraint Bars Criterion.

3.2.4 Touching Triangles on Negative Side Criterion.

3.3 Merge Cost

3.4 Flat Patches

3.5 Optimisations

3.5.1 Spatial Indices. As mentioned in Section **??**, the merge criteria within the patch validity computation involve predicates over all vertices, edges and faces of the input mesh. However, only predicate results computed on mesh features that spatially close to the patch feature in question actually contribute to the result. For this reason, the computation can be restricted to such mesh features from the beginning by querying spatial indices which support efficient spatial queries for mesh features. We use a modified kd-tree [] which supports AABB intersection queries TODO explain.

3.5.2 Priority Queue for Merging. Cite implementation? Describe cost to integer conversion?

4 EXTENSIONS

4.1 Parallelisation

- Initial computation of index structures is parallelized using `laparallel:plet`: vertex, edge and face indices as well as boundary edge index
- When two patches are merged, new patch links with the new patch at one end of the link are generated which requires expensive computations for evaluating the validity of the merged patch as well as computing the convex hull of the merged.

4.2 Merge Tolerance

5 RESULTS

Illustration of some generic results

Illustration of "tolerance"?

Run times for some input meshes?

6 CONCLUSION

7 FURTHER WORK

8 ACKNOWLEDGEMENTS

REFERENCES

[1] C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

[2] Elmer G Gilbert, Daniel W Johnson, and S Sathiya Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation*, 4(2):193–203, 1988.

[3] Rong Liu, Hao Zhang, and James Busby. Convex hull covering of polygonal scenes for accurate collision detection in games. In *Graphics interface*, pages 203–210, 2008. URL https://www.cs.sfu.ca/~haoz/pubs/liu_zhang_gi08.pdf.

[4] Khaled Mamou, E Lengyel, and A Peters. Volumetric hierarchical approximate convex decomposition. In *Game Engine Gems 3*, pages 141–158. AK Peters, 2016.

[5] J. O'Rourke and K. Supowit. Some np-hard polygon decomposition problems. *IEEE Transactions on Information Theory*, 29(2):181–190, 1983. doi: 10.1109/TIT.1983.1056648.

[6] Gary Snethen. Xenocollide: Complex collision made simple. In *Game programming Gems 7*, pages 165–178. Course Technology, 2008.

[7] Xinyue Wei, Minghua Liu, Zhan Ling, and Hao Su. Approximate convex decomposition for 3d meshes with collision-aware concavity and tree search. *ACM Transactions on Graphics (TOG)*, 41(4):1–18, 2022. URL https://dl.acm.org/doi/pdf/10.1145/3528223.3530103.