Voxelized Hierarchical Convex Decomposition - V-HACD version 4

John W. Ratcliff : jratcliffscarab@gmail.com
Khaled Mamou : kmamou@gmail.com

What is V-HACD?

- A C++11 header file only library which takes an arbitrary triangle mesh and decomposes it into a set of convex hulls, suitable for submission to a physics engine for real time simulation purposes
- Most physics engines only operate on basic convex shapes, such as boxes, spheres, capsules, and convex hulls for real-time simulation
- V-HACD is tool that can convert an arbitrary triangle mesh into a series of convex hulls which can reasonably approximate it as a solid object
- V-HACD is fast, stable, and robust
- Currently integrated into Omniverse, UE4, Blender, and other projects

Contributors

- The first version was known simply as ACD (approximate convex decomposition) and was written by John W. Ratcliff
- The next two versions (HACD and then V-HACD) were written by Khaled Mamou
- The most recent version 4 refactor of the codebase was done by John W. Ratcliff
  - With version 4 all previous versions are now to be considered deprecated and unsupported
  - Only version 4 of V-HACD will receive technical support going forward
- Miles Macklin : Provided the axis aligned bounding box code
- Julio Jerez : Provided the new high speed and high precision convex hull generation code
- Danny Couture : Optimized some of the voxelization code and added support for user defined threading callbacks
- Khaled Mamou provided the inspiration for running this algorithm in voxel space and implemented the voxelization code

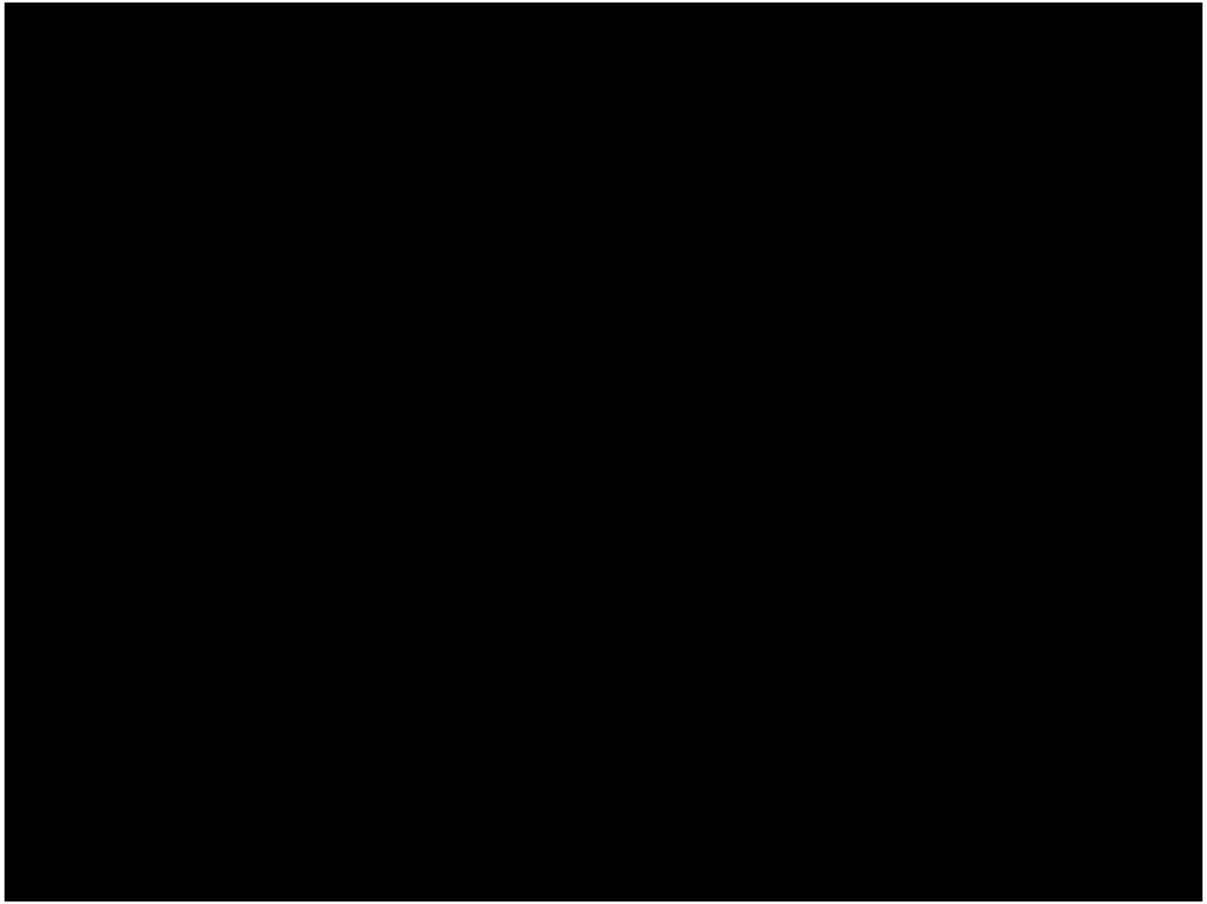
How to use it?

- V-HACD version 4 is delivered as a header file only library
- Simply include it in one of your CPP files with following #define declared
  - #define ENABLE_VHACD_IMPLEMENTATION 1
  - #include "VHACD.h"
- The define should be declared prior to including VHACD.h and then only once

An overview of how version 4 is used

Voxel Resolution

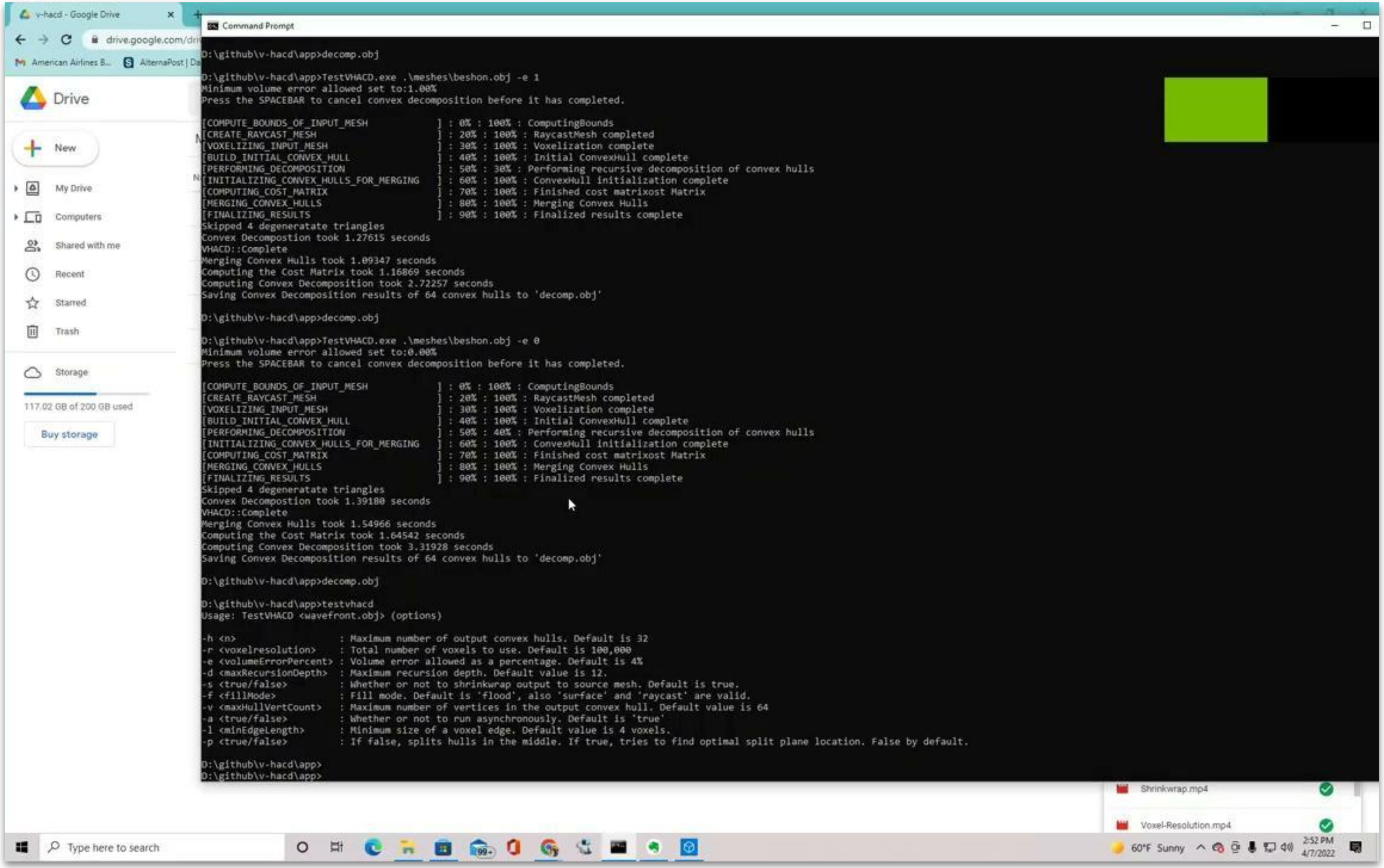
Shrinkwrapping Convex Hull Results



Maximum Output Convex Hulls

Volume Error Percentage

Maximum Convex Hull Vertices



```
D:\github\v-hacd\app>decomp.obj

D:\github\v-hacd\app>TestVHACD.exe .\meshes\beshon.obj -e 1
Minimum volume error allowed set to:1.00%
Press the SPACEBAR to cancel convex decomposition before it has completed.

[COMPUTE_BOUNDS_OF_INPUT_MESH            ] : 0% : 100% : ComputingBounds
[CREATE_RAYCAST_MESH                     ] : 20% : 100% : RaycastMesh completed
[VOXELIZING_INPUT_MESH                   ] : 30% : 100% : Voxelization complete
[BUILD_INITIAL_CONVEX_HULL               ] : 40% : 100% : Initial ConvexHull complete
[PERFORMING_DECOMPOSITION                ] : 50% : 30% : Performing recursive decomposition of convex hulls
[INITIALIZING_CONVEX_HULLS_FOR_MERGING   ] : 60% : 100% : ConvexHull initialization complete
[COMPUTING_COST_MATRIX                   ] : 70% : 100% : Finished cost matrixost Matrix
[MERGING_CONVEX_HULLS                    ] : 80% : 100% : Merging Convex Hulls
[FINALIZING_RESULTS                      ] : 90% : 100% : Finalized results complete
Skipped 4 degeneratate triangles
Convex Decompostion took 1.27615 seconds
VHACD::Complete
Merging Convex Hulls took 1.09347 seconds
Computing the Cost Matrix took 1.16869 seconds
Computing Convex Decomposition took 2.72257 seconds
Saving Convex Decomposition results of 64 convex hulls to 'decomp.obj'

D:\github\v-hacd\app>decomp.obj

D:\github\v-hacd\app>TestVHACD.exe .\meshes\beshon.obj -e 0
Minimum volume error allowed set to:0.00%
Press the SPACEBAR to cancel convex decomposition before it has completed.

[COMPUTE_BOUNDS_OF_INPUT_MESH            ] : 0% : 100% : ComputingBounds
[CREATE_RAYCAST_MESH                     ] : 20% : 100% : RaycastMesh completed
[VOXELIZING_INPUT_MESH                   ] : 30% : 100% : Voxelization complete
[BUILD_INITIAL_CONVEX_HULL               ] : 40% : 100% : Initial ConvexHull complete
[PERFORMING_DECOMPOSITION                ] : 50% : 40% : Performing recursive decomposition of convex hulls
[INITIALIZING_CONVEX_HULLS_FOR_MERGING   ] : 60% : 100% : ConvexHull initialization complete
[COMPUTING_COST_MATRIX                   ] : 70% : 100% : Finished cost matrixost Matrix
[MERGING_CONVEX_HULLS                    ] : 80% : 100% : Merging Convex Hulls
[FINALIZING_RESULTS                      ] : 90% : 100% : Finalized results complete
Skipped 4 degeneratate triangles
Convex Decompostion took 1.39180 seconds
VHACD::Complete
Merging Convex Hulls took 1.54966 seconds
Computing the Cost Matrix took 1.64542 seconds
Computing Convex Decomposition took 3.31928 seconds
Saving Convex Decomposition results of 64 convex hulls to 'decomp.obj'

D:\github\v-hacd\app>decomp.obj

D:\github\v-hacd\app>testvhacd
Usage: TestVHACD <wavefront.obj> (options)

-h <n>                    : Maximum number of output convex hulls. Default is 32
-r <voxelresolution>      : Total number of voxels to use. Default is 100,000
-e <volumeErrorPercent>   : Volume error allowed as a percentage. Default is 4%
-d <maxRecursionDepth>    : Maximum recursion depth. Default value is 12.
-s <true/false>           : Whether or not to shrinkwrap output to source mesh. Default is true.
-f <fillMode>             : Fill mode. Default is 'flood', also 'surface' and 'raycast' are valid.
-v <maxHullVertCount>     : Maximum number of vertices in the output convex hull. Default value is 64
-a <true/false>           : Whether or not to run asynchronously. Default is 'true'
-l <minEdgeLength>        : Minimum size of a voxel edge. Default value is 4 voxels.
-p <true/false>           : If false, splits hulls in the middle. If true, tries to find optimal split plane location. False by default.

D:\github\v-hacd\app>
D:\github\v-hacd\app>
```

Extraneous Tuning Parameters

How the algorithm works, step by step

- The following slides will, for the first time, describe how the V-HACD algorithm works in some detail
- This information is not actually necessary to use the library, but it is important that it be documented for future work and reference
- Note that internally all of V-HACD uses double precision floating point numbers. For geometric mesh processing algorithms the highest precision possible is recommended for the best results

Step #1 : Cleaning up the input mesh

- The first thing V-HACD does is clean the input mesh
- It computes the bounding volume of the input vertices and normalizes them to fit inside a cube based on the longest edge
- Each vertex is then indexed to remove any duplicates which might exist
- It looks for degenerate triangles and removes them as well

Step #2 : Create a RaycastMesh instance of the source mesh

- Once we have normalized and cleaned the input mesh, we next create an axis aligned bounding volume tree for raycasting and nearest point testing
    - In the final step of producing the convex hull output, the user may want the voxel vertices to be ‘shrink wrapped’ to fit exactly on the surface of the original source mesh; creating an instance of RaycastMesh allows us to do this efficiently
    - If the user wishes to use the ‘raycast fill’ option, or the option to try to find the optimal splitting plane, then the raycast mesh representation is used for that purpose as well
- The AABB code used by V-HACD was generously contributed by Miles Macklin

Step #3 : Voxelize the source mesh

- We next convert the source mesh into a voxelized grid at a resolution specified by the user
- The voxelization code was first written by Khaled Mamou and later performance improvements were made by Danny Couture
- The voxelization code is fairly straightforward; it simply converts each triangle into voxel space and plots the points
- The voxel representation is direct, one byte per voxel in three dimensions (no hash table)
- A voxel resolution anywhere between 100,000 and 10,000,000 can be used, but the default value of 400,000 is more than sufficient for most general use cases
- The higher the voxel resolution then the more fine details you can capture but, in exchange, the slower the process takes

Step #4 : Fill the interior voxels

- When performing convex decomposition, in most cases, we wish to treat the source mesh as being a solid object. For this to work the input mesh should be ‘watertight’ with no holes in it
- By default V-HACD will find all of the interior voxels by performing a flood-fill operation. However, if the source mesh is not 100% perfectly watertight, the flood-fill will fail
- If the source mesh isn’t perfectly watertight, the user can try the raycast fill option, which will determine interior voxels by raycasting towards the source mesh
- Finally, in some rare cases, a user might actually want the source mesh to be treated as if it were hollow, in which case they can skip generating interior voxels entirely
- Once this step is complete, all voxels which correspond to the surface of a triangle on the source mesh will be defined and all voxels which comprise the ‘interior’ will be defined as well

Step #5 : Create the initial VoxelHull instance

- The core V-HACD algorithm operates by taking a set of voxels and recursively subdividing them until they meet a specific completion criteria
- For the first VoxelHull we begin with all of the voxels created by the voxelization step
- First, all of the surface voxels are copied into a container
- Next, all of the interior voxels are copied into a container
- Next, a triangle mesh representation of all of the surface voxels is created by converting each voxel into a twelve triangle box
- Finally, a convex hull is created which can enclose all of the vertices from that triangle mesh representation
- The volume of the convex hull is calculated as well as the volume of all of the voxels which contributed to it

Step #6 : See if the input mesh is already sufficiently convex

- Most of the V-HACD algorithm works off of the concept of ‘volume conservation’. Meaning, if the volume of the convex hull is sufficiently the same as the volume of the voxels which contributed to it, then we can assume that the convex hull is a reasonable approximation for the voxels
- The user provides a ‘percentage allowed volume error’ to determine whether any hull is a close enough match for the voxels which comprise it
- No voxel representation is an exact match for the original mesh; there will always be some precision loss due to it being converted into voxel space
- If the volume of the hull surrounding the input mesh is very near the volume of all of the voxels comprising it, then we can ‘early out’ and determine that the input mesh was itself convex and a single convex hull is a reasonable approximation

Step #7 : Recursive decomposition of voxel hulls

- If the first convex hull is not a reasonable approximation of the source mesh, we must then start recursively splitting each into two smaller pieces
- First we must choose an axis aligned splitting plane
- The default behavior is to use the midpoint of the longest edge
- There is an optional mode to attempt to split along the greatest point of ‘concavity’ found on the longest edge. This feature is considered ‘experimental’, may have artifacts, and is not strictly needed. It is currently disabled by default
- The next slide will discuss how the plane splitting operation works

Step #8 : Split a single convex hull and voxel patch into two

- First we create two new VoxelHull instances representing the positive and negative half of the splitting plane
- We copy all surface voxels from the original into either the positive or negative child depending on which volume they intersect
- Next we copy all of the interior voxels the same way, however, we check to see if any of the interior voxels we are copying lie directly on the split plane and, if so, they are instead treated as new ‘surface’ voxels for the child
- Next we build a triangle mesh representing the surface voxels only
- Finally we build the convex hull which best fits the triangle mesh representation of the voxels
- This operation can be run in parallel to increase overall performance

Step #9 : Determine if either child is ‘complete’

- A single VoxelHull patch is considered ‘finished’ if it meets the following criteria:
    - All 3 sides are below the minimum width threshold. Once a set of voxels becomes too small, it is not reasonable to expect a hull to approximate it well. The default value is a width of 4 voxels
    - The recursion depth has reached a maximum limit specified by the user. The default value is 12; which sets a maximum of 4,096 convex hulls. This can be set to an even higher value if needed
    - The percentage volume error between the voxels and the convex hull is below the threshold specified; the default value is an acceptable error of 4%. It is perfectly valid to set the error to zero, and just let the algorithm reach maximum recursion depth on all leaf nodes
    - If any of these termination conditions are true, then this hull fragment is retained
    - If not, then we continue recursively splitting this node

Step #10 : Merging convex hull results by first computing the N squared cost basis matrix

- Once the recursion is complete it is completely normal and valid to have hundreds, if not thousands, of convex hulls which have been produced
- The next step revolves around merging the convex hulls based on proximity and volume conservation until we reach the target number of hulls specified by the user
- The first step of this operation is to precompute the 'cost basis' of merging *any two hulls* with each other
- To do this, we create a new convex hull which is comprised of all of the points in HullA and all of the points in HullB. The 'cost' is determined as the ratio of the difference in the volume between HullA + HullB versus the volume of CombinedAB. If the volume of the combined hull is nearly the same as the two separate, then the combined hull is considered very 'low cost'
- As a performance optimization, we don't actually generate the merged convex hull if the two source hulls do not overlap by their axis aligned bounding boxes inflated by 10%. If they don't overlap, we can do a faster error computation based on just the AABB volumes
- Computing the merged convex hull and volume error can be run in a background thread to improve performance
- The cost matrix results are placed into a priority queue sorted by the lowest cost to merge first

Step #11 : Merging convex hulls

- Once the cost matrix has been calculated we can begin merging convex hulls
- We begin by popping the lowest cost pair from the priority queue
- We next see if the two convex hulls in this pair still exist (they could no longer exist due to previous merge operations). If either of the hulls do not exist, then we can skip this pair and move on to the next
- Since we have determined that these two hulls have the lowest merge cost, we go ahead and merge them. We remove the two hulls which existed before, and now add the new merged hull. We don't remove their references from the priority queue, as these will get cleaned up automatically
- We must now compute the cost matrix entries for this new hull we have just created against the outstanding hulls; so we do this accordingly for just the new hull we created
- Finally, this process simply repeats until enough hulls have been merged to reach the desired output amount specified by the user

Step #12 : Finalizing the output

- The final step is to make sure that the output convex hulls do not contain more vertices than specified by the user
- As well, the user may have enabled the ‘shrinkwrap’ option (true by default) which attempts to translate all of the vertices from voxel space back into mesh space by using a ‘closest point’ algorithm
- The last step is to scale the output convex hulls back into the space of the original input mesh