Independent Polyhedric Decomposition

Chase Grajeda

I – Abstract

This project explores a computational geometry algorithm designed to decompose some mesh into uniform subdivisions along a 3D grid. This algorithm offers a tool that is applicable to a wide variety of domains that utilize destruction, erosion, and physics. The algorithm operates by taking some input mesh ϕ and a smaller input mesh ψ, then repeatedly translating ψ along the bounding box of ϕ. After each translation, the intersection of points contained by ϕ and ψ is calculated and added to a list as a new mesh. Once all translations are complete, the list of subdivisions is returned, which represents the domain of ϕ filled with ψ. Key features of this algorithm include preserving internal structures of 3D objects, direct support for external rendering software, and dynamic input support for both ϕ and ψ. Through practical demonstrations and experimentations, the algorithm's applications and results are showcased across a couple of simple scenarios. This exploration may contribute as a valuable tool to the 3D modeler's toolkit in the building of realistic 3D environments.

II – Introduction

As years progress, it's obvious that the ability and scale of technology has been vastly increasing. Especially with the rise of computing, 3D modeling has been an important aspect to design, engineering, and simulations. An important tool we have used ourselves quite often in 3D modeling is taking a 3D object and adding edges or faces to it. This allows us to perform a variety of new transformations such as extrusion. Another application is the simplification of meshes, which can be done with methods such as Triangulation, which (briefly) takes a large face or collection of faces and turns them into faces triangles. However, this really only operates on the surface of an object. Sometimes if may be helpful to begin subdividing regions not only on the surface, but on the object's internal volume as well.

Another interesting application of 3D modeling is physics simulation. Simulation programs sometimes offer a physics engine that allows researchers to experiment with objects with gravity, forces, and motion. An extension of this is examining how an object reacts to destructive forces such as explosions, or to see how an object may crumble. To provide a tool for this, we are proposing an algorithm for researchers to use. This algorithm will take a input mesh and subdivide it uniformly by a specified shape. This allow users to gain more control over their meshes and apply smarter physics processes in their applications. Further, this will allow more flexibility when subdividing very large meshes.

The remainder of the paper is organized as follows: Section III will detail the task and implementation of the algorithm, as well as provide pseudocode and simple equations. Section IV will provide a range of examples of the algorithm at work and the results there of. Section V will briefly provide the time complexity for the implementation. Lastly, Section VI will conclude the paper, holding a discussion summarizing the paper, the implementation, results, drawbacks, and future applications.

III – Task and Algorithm

Let ϕ be the primary object to be operated over and ψ be the subdivision object. To reiterate, the goal of the application is to iteratively subdivide the domain of ϕ into a uniform distribution of ψ. Further, each subdivision will be a disconnected object from the overall domain. In the end, the algorithm will return a vector of meshes that describe the domain of ϕ as a whole. This allows complex transformations to be applied to ϕ, including creating expanded views, simulating crumbling or destruction, and having more control over small regions within ϕ.

The algorithm utilizes C++ and CGAL to successfully generate a domain subdivision of ϕ. From CGAL, the Polygon Mesh Processing, Affine Transform, and Surface Mesh packages are used. Please note that the exact predicates inexact constructions kernel is utilized. The program begins by loading a source

object to be subdivided (ϕ), and a space filling object to subdivide the domain (ψ). Both of the inputs are .obj files passed in by the command line: "./marching_polyhedrons.exe <path to ϕ> <path to ψ>." Note, ϕ must be larger than ψ on all axes. Next, the inputs are triangulated to allow finer controls for calculating intersections. The bounding box of ϕ is now calculated. From the bounding box, the x, y, and z widths are determined. The same is done for ψ. Now, the uniform grid over which ϕ will be operated on is determined. The question being asked here is simply, "how many ψ can uniformly fit in ϕ?" The question is tacked individually on each axis, the results from which become a translation offset on ψ. This is represented by the following equations:

$$x_{offset} = ceil\left(\frac{\phi_{xwidth}}{\psi_{xwidth}}\right)$$

$$y_{offset} = ceil\left(\frac{\phi_{ywidth}}{\psi_{ywidth}}\right)$$

$$z_{offset} = ceil\left(\frac{\phi_{zwidth}}{\psi_{zwidth}}\right)$$

Let's use an example first in 2D. Say we have a 3x3 square ϕ that we'd like to fill with a 1x1 square ψ. Then, there will be 3 partitions along the x axis, and 3 partitions along the y axis. This gives 9 partitions in total. Extended to 3D, say we have a 3x3x3 cube ϕ that we'd like to fill with a 1x1x1 cube ψ. The x, y, and z axes will have 3 partitions along them respectively, resulting in 27 partitions total. This essentially creates a uniform mapping over ϕ, given the calculated offsets.

Once the uniform mapping is calculated, ϕ and ψ are translated such that their min-most corners of their respective bounding boxes are aligned. Now that the objects are aligned, we can begin subdividing ϕ by ψ. The algorithm does this incrementally over each axis. To perform a subdivision, or a "cut," the intersection of ϕ and ψ is calculated. Here, the intersection is the collection of points shared between ϕ and ψ. The intersection is saved as a new mesh and added to a list of surface meshes. Then,

ψ is translated ψ_{xwidth} units along the x-axis. We'll repeat this for x_{offset} times. Then, we'll translate along the y-axis and return the x-axis to the origin. Generally, this is translating ψ over the 3D uniform grid and calculating the intersection at each stop. Below the pseudocode for the algorithm:

Subdivisions = [ ]

For (int z = 0; z < z_{offset}; z++):

For (int y = 0; y < y_{offset}; y++):

For (int x = 0; x < x_{offset}; x++):

Mesh intersection = Intersection(ϕ, ψ)

Subdivisions.add(intersection)

TranslateTo(ψ, x = ψ.x + ψ_{xwidth}, y = ψ.y, z = ψ.z)

TranslateTo(ψ, x = 0, y = $\psi.y + \psi_{ywidth}$, z = $\psi.z$)

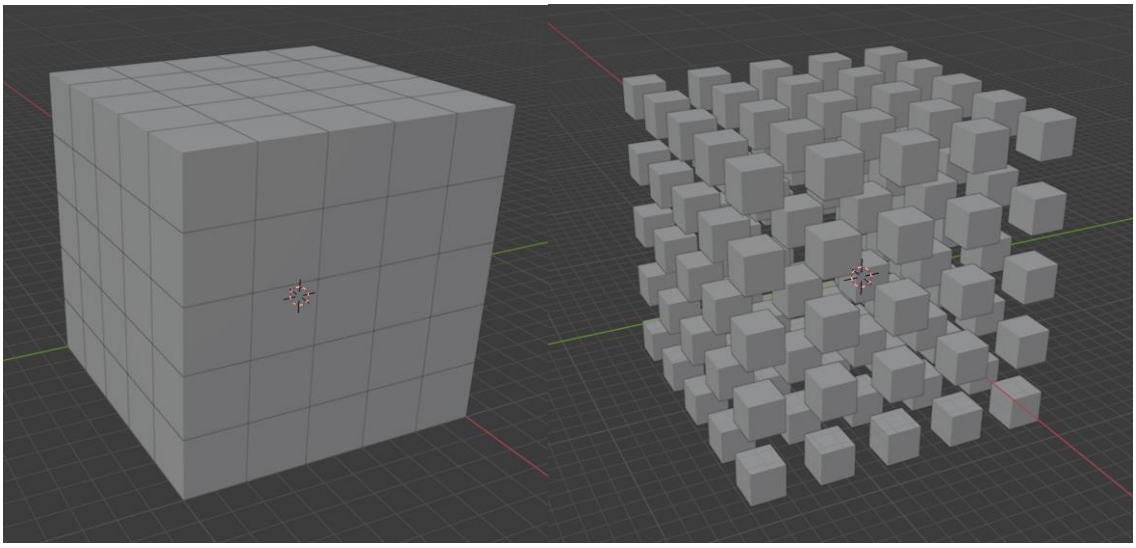
TranslateTo(ψ, x = 0, y = 0, z = $\psi.z + \psi_{zwidth}$)

After the walk over the 3D uniform grid is complete, a list of indiviudal surface meshes remains. This list represents the domain of ϕ by filled with ψ. To visualize this, all of the members within the list will be translated away from the center of the domain, creating an "exploded" view. This isn't necessary, but is important for visualization purposes. Lastly, the objects are all saved to a .obj file.

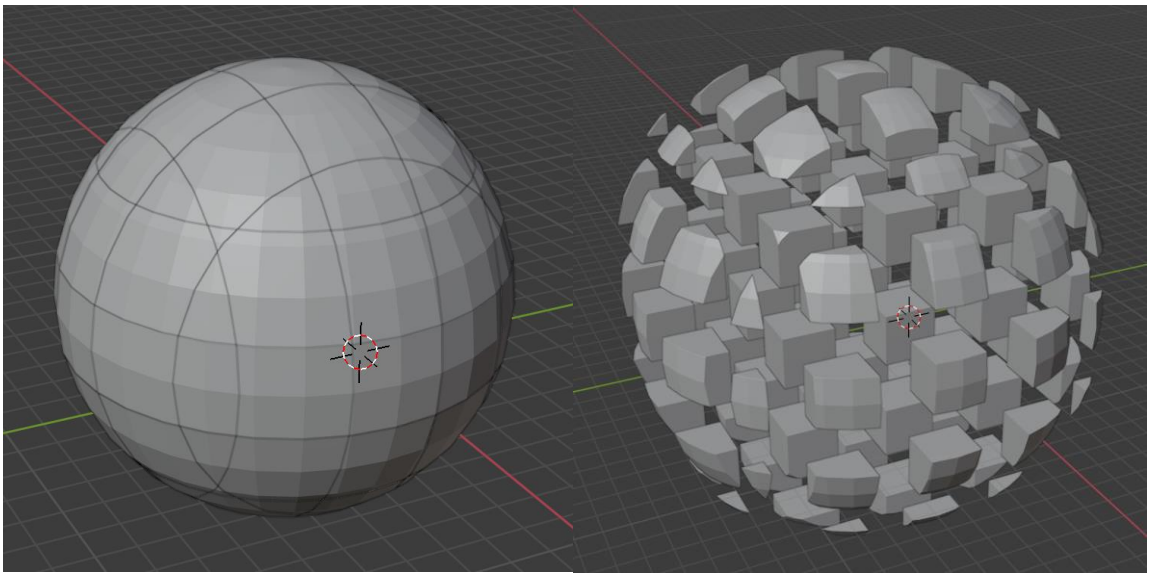
IV – Experiments

To demonstrate the use of this algorithm, as series of test scenarios have been generated. Three shapes will be used: Cube, Cone, and Sphere. Below are a few combinations of setting ϕ and ψ to the input shapes.

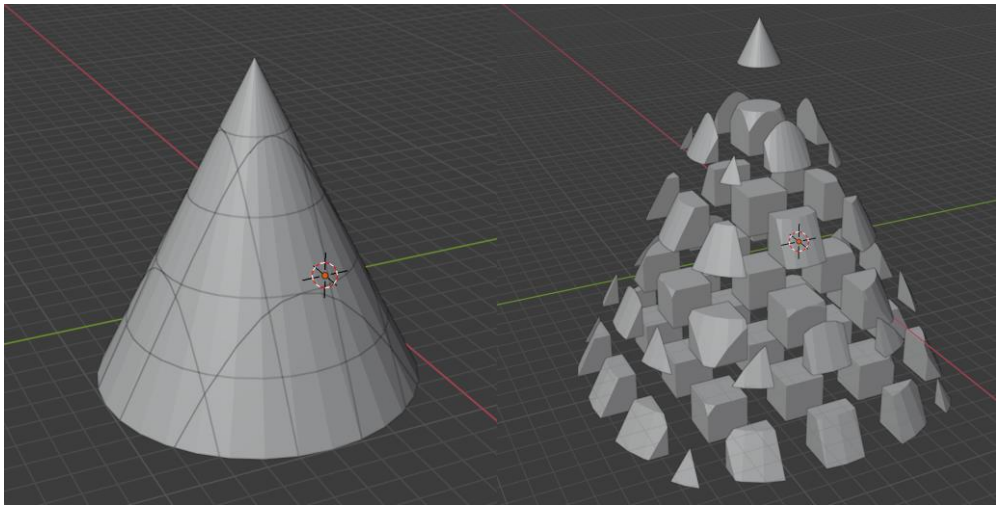
Cube filled with cubes (left: regular, right: exploded):



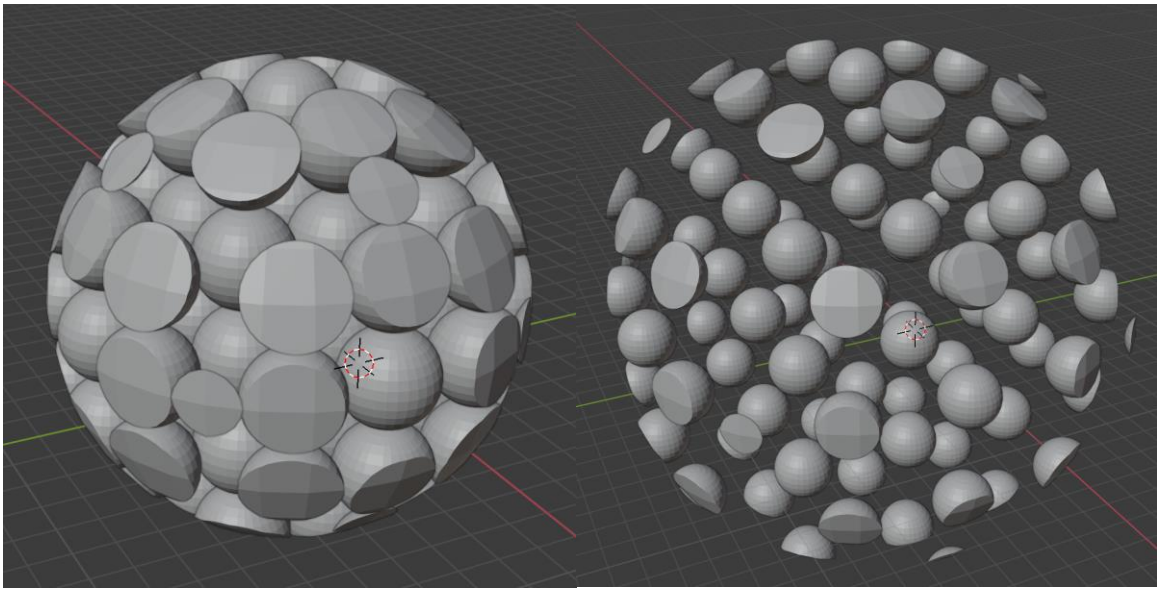
Sphere filled with cubes (left: regular, right: exploded):



Cone filled with cubes (left: regular, right: exploded):



Sphere filled with spheres (left: regular, right: exploded):



V – Time Complexity

The core algorithm can be quite expensive depending on the size of the input objects. Given the algorithm loops over the 3D uniform grid by the xyz_{offest} calculated earlier, we can derive the time complexity from there. With respect to the program as a whole, assume the CGAL 3D triangulation function has complexity $O(n^2)$ where n is the number of input points, and the intersection function has complexity $O(\phi_t * \psi_t)$ where ϕ_t and ψ_t are the total number of triangles of ϕ and ψ respectively. The nested for-loop provided by the pseudocode has complexity $O\left(z_{offset} * y_{offset} * x_{offset}\right)$, ignoring the intersection algorithm. Thus, we have the following time complexity:

$$O\left(n^2 + \left(z_{offset} * y_{offset} * x_{offset} * (\phi_t * \psi_t)\right)\right)$$

VI – Discussion

The presented algorithm and implementation provide a wide range of applications and neat ways to edit the domain of an object. However, the algorithm in its current state has some limitations. First off, the algorithm has some room for optimization. Currently, if a user attempts to subdivide a very large object, say a 1000x1000x1000 cube by a 1x1x1 cube, the program will take a long while fulfilling this request, given it has to write 1000^3 objects. Thus, this algorithm is not suitable for decomposing large objects at run time, and is better being used for preprocessing. The algorithm also has some bugs. Whether its due to implementation error or a behavior from the used CGAL libraries, the application only supports convex 3D objects as inputs. This means any objects with holes or surface deformations may cause the program to crash. Ideally, this bug can be fixed to support all kinds of 3D objects in the future.

Another abnormality that caused frustration was getting all of the subdivisions to be saved to a single file. We failed to find a CGAL library or package that merged two surface mesh objects and

supported drawing. One exploration was using the mesh domains CGAL package, but the only file we could export to was MEDIT. However, the generated file was buggy and would always fail to render. In order to have some form of output, all of the subdivisions are saved to a list and exported as individual .obj files. While this is not great to have in applications, it allowed us to export the results to a rendering software such as Blender. A feature that we would like to add to the program is supporting an input list of .obj to subdivide by. This will allow us to randomly choose between a $\psi_1, \psi_2, \ldots \psi_n$ to intersect the mesh with, creating a more complex and interesting domain.

Despite the current limitations, the program has many useful applications. Firstly, the program can be used to prepare meshes for simulating destruction and erosion. Since the output is a set of indidivual polyhedrons, individual meshes can be manipulated. Especially when a Physics engine is applied, individual subdivisions can be pushed or pulled in any direction and the forces will propagate throughout the domain. Another application is prepping an object for carving. Say you have a general object ϕ that you would like to carve out, whether its creating a simple tunnel through (bullet hole, cave) or wanting to create a complex tunnel system. Since the program returns the list of individual subdivisions, the user has the power to delete objects from the output. Effectively, this allows them to create ψ-sized holes in an external editor.