Partitioning Large 3D Objects for Use in 3D Printing

Jason Kraft
kraftj3@rpi.edu

Uyen Uong
uongu@rpi.edu

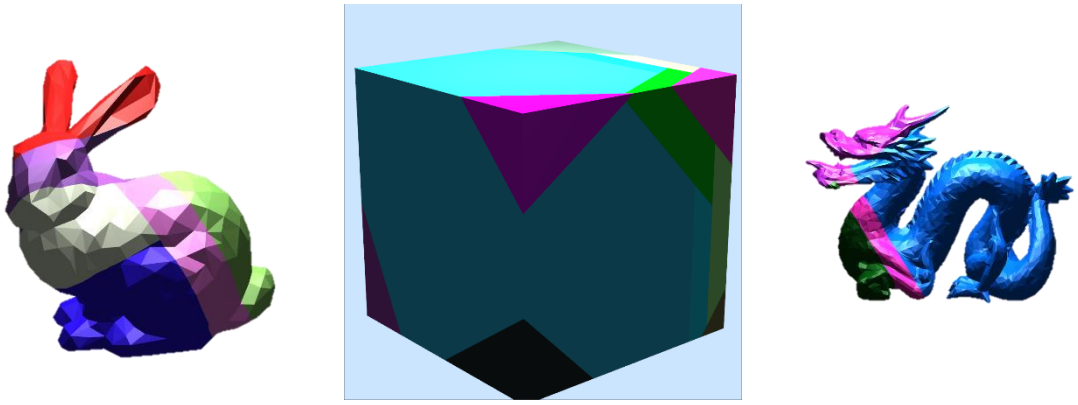

Figure 1: *Partitions are highlighted on each mesh in different colors. Of all our tests, the bunny model performed the best.*

Abstract

While the cost of 3D printing technology has decreased substantially for both professional and hobbyist consumers, the printing of large 3D objects remains one of the greatest challenges to wide-scale adoption. We present an algorithm to automatically partition large 3D models into printable subcomponents. This algorithm optimizes the partitioning scheme based on several desirable criteria including printability, assemblability, and efficiency.

1 Introduction

3D printing is rapidly becoming a critical component to the design and manufacturing process. For professionals, 3D printing presents a new method for rapid prototyping. For hobbyists, it is a medium for flourishing creativity and art. Nevertheless, attempting to print 3D objects larger than the working volume of a printer remains a difficult, if not impossible, problem to solve. Solutions usually involve producing a smaller scale model of the desired component (which is undesirable for many situations) or manually breaking the object down into several pieces (which is often very time consuming for the designer).

We present a framework for the automatic partitioning of large 3D objects. This algorithm searches for an optimal partitioning scheme based on several key criteria:

- Printability: whether the parts can fit inside the printing volume.
- Assemblability: whether the parts can be reassembled easily after printing.
- Efficiency: minimizing the number of subcomponents required to be printed and maximizing the part volume to printer volume ratio.

These criteria, which we will formally define as objective functions, can be weighted to control their influence on the final output.

2 Related Work

Various methods have been developed that try to optimize the subdivision of models while enabling the subcomponents to be reassembled through connectors.

This kind of method was first proposed in "Automatic Subdivision and Refinement of Large Components for Rapid Prototyping Production" [Medellin et al. 2006]. Rather than decomposing a 3D model into thin sheets that previous works had done, this paper proposes a new technique for decomposing 3D models into various sizes and geometries. The system uses an iterative method of defining 3D units to divide the model. An initial 3D unit set is created using a regular 3D lattice and then refined through design operators that check for manufacturing difficulties. These operators include checking for thin sections, cusps and undersized volumes. A random pattern of spheres is used for the connectors to ensure that each connection has a unique orientation to aid in the assemblage. However, because of the lattice cuts and the limitations of the operators on evaluating manufacturability, the system may produce undesirable results, relying on user intervention to correct them.

A more robust system was later developed and proposed in "Chopper: Partitioning Models into 3D-Printable Parts" [Luo et al. 2012]. This paper describes the framework, Chopper, which divides a given 3D model into parts that are feasible to create within the printing volume, while optimizing the partitions based on a set of criteria that more thoroughly describe the needs of the user. This criteria includes the ability to be printed and assembled, minimizing the number of partitions and maximizing the use of the printing volume, ensuring seams are in areas of low stress and unobtrusive to the design of the model, and optimizing the connector locations. The planar cuts that divide the model are represented using a Binary Space Partition Tree, allowing for non-axis aligned cuts unlike in the previous method by Medellin et al.

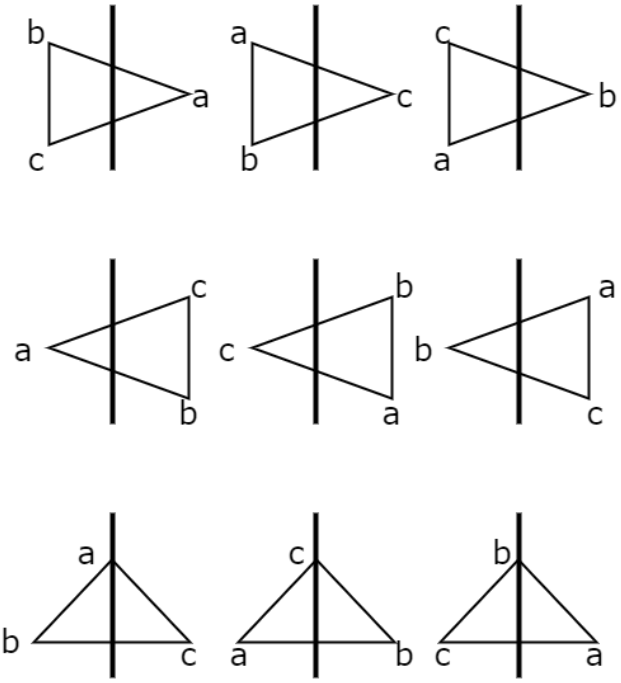

Figure 2: *All possible orientations of triangles intersecting with a plane.*

Simulated annealing was used to determine the placement of the hexagonal prism connectors.

Another method of automatic partitioning was proposed in "Level-Set-Based Partitioning and Packing Optimization of a Printable Model" [Yao et al. 2015], which not only optimizes partitions based on printable sizes but also aims to reduce printing time and packing space. This method for partitioning uses level sets to represent the model and focuses on minimal stress load, interface area and packed size, surface detail alignment, printability, and assembling. Cylinders were used as connectors, utilizing the same method as Luo et al. to determine the placement of the connectors.

For our purposes, we chose to closely follow the partitioning scheme described in Luo et al.'s paper. The criteria that Chopper used exactly matched our needs and avoided the limitations from Medellin et al.'s implementation. The partitioning system proposed in Yao et al.'s paper included more functionality and optimizations than what we wanted.

3 Approach

In developing our partitioning algorithm, we encountered several key challenges. These challenges include deciding how to efficiently divide a mesh with a plane, creating a optimization search algorithm that is less prone to local minima, deciding how to generate and evaluate cut planes on a given submesh, and defining the objective functions for our search algorithm. We chose handle each challenge, described in sections 3.1 through 0, in the order in which they appear in this paper. In doing so, we wish to convey the thought process used to designing each component.

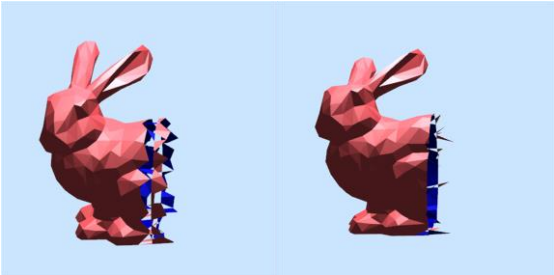

Figure 4: *Several attempts were necessary to account for all edge cases when dividing triangles that intersected with the cut plane.*

3.1 Dividing a Mesh

Our mesh representation uses the Doubly Connected Edge List, more commonly known as the HalfEdge, data structure first proposed by Muller and Preparata [1]. When dividing a mesh into two subcomponents given a plane, represented by a normal and an offset, each subcomponent must not have duplicate vertices and the triangles that are cut must be oriented the correct way. Because of this, there are nine different cases that we needed to account for (**Figure 2**) regarding the orientation of the intersected triangle relative to the cutting plane. Multiple attempts were necessary to correctly compute all the cases (**Figure 4**).

Our first implementation of dividing the mesh did not check for the first case, however, and ended up with multiple duplicate vertices and disconnected triangles (**Figure 5**). When given a normal and an

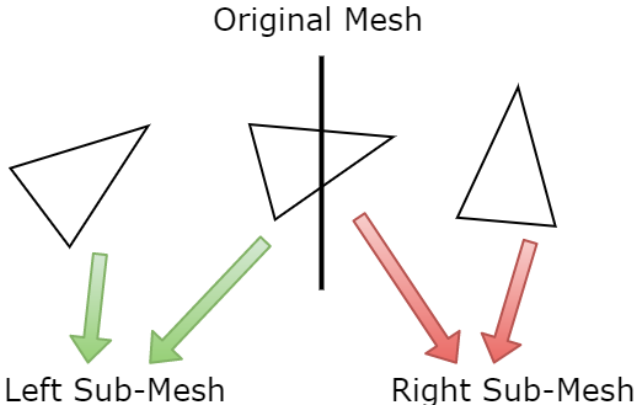

Figure 3: *Dividing the triangles of a mesh between two submeshes separated by a plane. Triangles that intersect the cut plane are added to both submeshes and later pruned.*

offset, a point on the plane was calculated by multiplying the normal with the offset, and we searched through all the triangles of the mesh. Triangles that were completely to the left and to the right of the plane, determined by taking the dot product of the normal with the difference between each vertex and the point on the plane, were added to their respective submeshes (**Figure 3**). The intersecting triangles were cut up according to their orientation relative to the plane, and the divided triangles were added to their respective meshes. This was all done within the same loop.

Our second, improved implementation (**Figure 5**), while similar to the first implementation, ensures duplicate vertices are not added to the submeshes by pairing the vertices from the original mesh with the vertices of the submeshes to avoid having to search for them. In addition, triangles intersecting

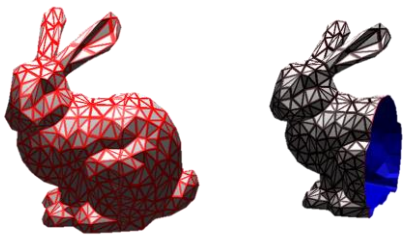

Figure 5: *Dividing the mesh without accounting for duplicate vertices (left). The red lines indicate that each edge in our mesh is a boundary edge. Our second implementation properly accounts for duplicate vertices and boundary edges (right).*

Algorithm 1: *Beam search*

```
input  : O is the object, b is the beam width
output : T is the BSP tree that partitions the object
function T = BeamSearch(O, b)
  currentBSPs ← ∅
  while not AllAtGoal(currentBSPs)
    newBSPs ← ∅
    foreach T that is not at the goal
      remove T from currentBSPs
      P ← LargestPart(O, T)
      insert all results from EvalCuts(T, P) into newBSPs
    while |currentBSPs| < b
      insert HighestRanked(newBSPs) into currentBSPs
```

Algorithm 2: *Evaluating a cut*

```
function ResultSet = EvalCuts(T, P)
  N ← OctahedronNormals()
  foreach n_i ∈ N
    maxOffset ← maxPosition(n_i)
    for d_j ← minPosition(n_i) to maxOffset
      T' ← Chop(P, n_i, d_j)
      foreach objective function f_k
        f(T') += a_k f_k(T')
  resultSet ← ∅
  foreach T sorted by f(T)
    if SufficientlyDifferent(T, resultSet)
      resultSet += T
  return resultSet
```

the cutting plane are added to both submeshes and are split up after all the triangles in the original mesh have been added to the submeshes. This made the algorithm more readable and easier to manage.

3.2 Search Algorithm

Our search algorithm is an adaptation of a beam search as described by Luo et al. [2]. Briefly stated, a beam search is a semi-greedy optimization algorithm where, upon each iteration, we select top n results (referred to as a beam) and discard the rest (**Figure 6**). The main purpose to using a beam search is to efficiently converge on a potential solution while lowering the risk of selecting locally optimal cuts that may lead to suboptimal outputs.

Though the overall pseudocode for our implementation is identical to that of Luo et al.'s [2], we found the wording in their paper to be particularly vague. As a result, it took multiple attempts and iterations to obtain a reasonable result.

3.2.1 Beam Search

The algorithm (**Algorithm 1**) begins by defining a specified beam width b (we chose a beam width of 4 in most instances). This beam width defines the size of an array, which we will refer to as *currentBSPs*, which holds the best candidates for our BSP tree. For the initial iteration, we store the object in a BSP tree and place this tree into the first index of our array. We then check each tree stored in *currentBSPs* to see if they have met our final goal (our goal being defined as whether each subcomponent of our original object can fit within the working volume of our printer). For each tree that does not meet this criterion, we remove it from *currentBSPs*, find its largest part P using the algorithm described in section 3.3.1, and evaluate all potential cuts of P and store them in *newBSPs*.

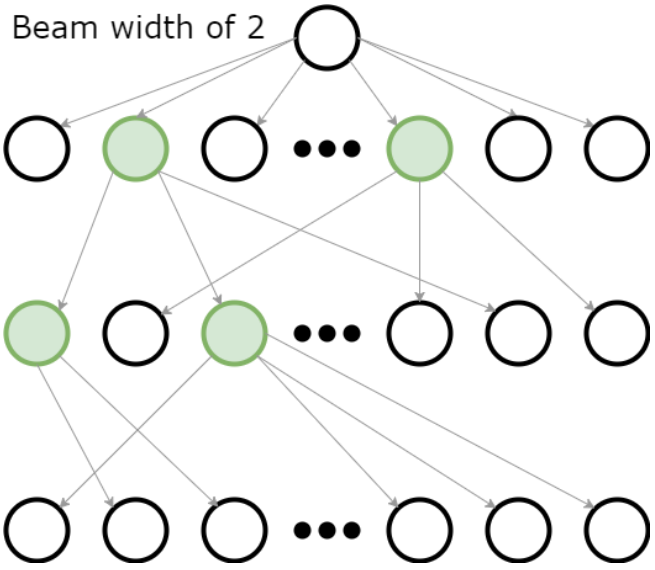

Figure 6: *A simplified diagram of a beam search. Each node represents a BSP tree in our search algorithm. Each row represents one iteration of cutting and evaluation. The green nodes represent the highest ranking BSP trees for each iteration.*

After completing this process for all BSP trees in *currentBSPs* that do not meet our goal, we replace these trees with the best results in *newBSPs*. Note that every tree in *newBSPs* is the result of adding a single cut to one of the BSP trees in *currentBSPs*. This process of subdivision continues until all trees in *currentBSPs* meet the final goal of fitting in our print volume.

3.2.2 Evaluating a Cut

Given a tree *T* and its largest part *P*, we must find the best possible cuts of *P* and grade them based on our objective functions (**Algorithm 2**). Since there are an infinite number of ways to cut our part, we restrict our search space to a handful of plane normals and offsets. The directions were obtained by normalizing the vertices of a thrice subdivided octahedron centered at the origin [2]. After discarding any duplicate normals (normals that point in the same direction but with opposite sign), we are left with 129 unique directions.

For each direction, we find the scalar position along the normal of each vertex in our mesh defined by the dot product of the normal and the vertex position. We then restrict our offset search space by setting our starting offset at the minimum position and incrementing by some small amount. This amount is subjective and can vary depending on the size of the mesh. For an object of length 10cm, for instance, we would set our offset increment to 5mm.

For each normal-offset pair, we cut *P* into two parts using the algorithm described in section 3.1. We then grade the resultant BSP tree using the objective functions in section 0, storing this grade as a member variable in the root node of our tree. We place these candidate trees in a priority queue sorted by grade (lowest grade at the top of the tree).

Afterwards, we iterate through each candidate tree *T* in order of grade, adding them to our *resultSet* list only if the grade of *T* is sufficiently different based on the Root-Mean-Square distance, defined as

$$E_{RMS}(\hat{x}) = \sqrt{\frac{\sum_{t=1}^{n}(\hat{x} - x_t)^2}{n}} \qquad (1)$$

where $\hat{x}$ is the grade of *T* and $x_1 \cdots x_n$ are the grades of BSP trees already in *resultSet*. Lastly, we return *resultSet*.

3.3 Objective Functions

Once we have our potential BSPs, we would like to evaluate the quality of each new cut using several objective functions. These objective functions take in a BSP tree as their input and output a value based on how well the tree adheres to the functions' corresponding criteria. To control how much influence each objective function has on the final output, we multiply each function by a weighting coefficient and sum their results.

$$f(T) = \sum_k a_k f_k(T) \qquad (2)$$

Each function f_k in equation (2) is a unique objective function and each constant a_k is its weighting coefficient. Since we are treating our search as a minimization problem, the best BSP trees are the ones with the lowest grade.

3.3.1 Number of Parts

Because we wish to minimize the total number of partitions created, we try to minimize the number of printing volumes necessary to cover each part, denoted by $\Theta(P)$. In order to do so, we use Luo et al.'s calculation of the f_{part} objective function [2] that compares the sum of all the printing volumes needed for each partition to the number of printing volumes needed to tile the original mesh, Θ_O.

$$f_{part}(T) = \frac{1}{\Theta_O}\sum_{P \in T} \Theta(P) \qquad (3)$$

To calculate $\Theta(P)$, Luo at al. created an Oriented Bounding Box around the partition to calculate the number of printing volume tiles. However, we used an Axis-Aligned Bounding Box in our calculation, which still produced reasonable results. Calculating

Table 1: *Results*

Model	Number of Vertices	Printer Dimensions	Offset Increment	Run Time	Number of Parts
Cube	8	0.5×0.5×0.5cm	0.1cm	4.51s	10 (did not converge)
Stanford Bunny	1000	0.2×0.2×0.2cm	0.05cm	7.16s	7
Stanford Dragon	~5000	5.0×5.0×5.0cm	1.0cm	1m24s	10 (did not converge)

f_{part} requires a simple recursive traversal of the BSP tree, computing $\Theta(P)$ whenever we hit a leaf node and summing up our results from both child nodes.

3.3.2 Utility Objective

We also wish to use the printing volume as effectively as possible. Small partitions are undesirable because they make assembling more difficult. We adopt Luo et al.'s f_{util} objective function that compares the volume of the partition, V_p, to the printing volume, V.

$$f_{util}(T) = \max_{P \in T} \left(1 - \frac{V_P}{\Theta(P)V}\right) \qquad (4)$$

This can also be easily calculated through a recursive traversal of the BSP Tree, computing f_{util} at each leaf node and taking the maximum of the results from the children nodes.

4 Results

We ran our program against three models of varying size and complexity and compared them based on run time and number of parts required to partition the mesh (**Table 1**). The models in increasing order of complexity were a cube with just 8 vertices, followed by a 1000 vertex bunny and a dragon of roughly 5000 vertices. To our surprise, the bunny significantly outperformed the other two models, finishing its search it just over 7 seconds with a total of 7 partitions. The cube, while fast (finishing its search after 4 seconds), did not converge on a viable set of cuts after a 10-iteration limit. The dragon was the worst performer, taking over a minute to complete and failing to converge as well after hitting a 10-iteration limit. **Figure 1** shows the three models after running them through our program.

In the case of the dragon, we suspect the model might have converged had we given it more resources and time to run. The cube's failure was less obvious, however. The algorithm continually attempted to cut the mesh along the corners rather than down the middle. This could be due to a lack of enough objective functions or simply an error in the logic of our code.

5 Conclusion

The algorithm that we have proposed automatically partitions a mesh based on printability, assemblability, and efficiency. Although the algorithm does not work for all types of models, the produced results appear to be plausible mesh partitions.

There are many possibilities for future work on the project that mainly focus on improving the efficiency of the algorithm as well as improving its robustness.

We were unable to parallelize the evaluation of the numerous possible cuts, which severely slows down the algorithm. The information stored within individual Mesh objects were accessed and modified by different threads, causing multiple copies of the same edge to be added to a mesh that resulted in assertion errors. With the limited amount of time we had left, we decided our time was better spent on other tasks. However, getting this to work would significantly improve the runtime.

Because our algorithm uses an Axis-Aligned Bounding Box, rather than an Oriented Bounding

Box, we are not as effectively using the printing volume as we would like. Extending our program to calculate the OBB would make our objective functions more accurate.

In addition, considering the connector feasibility of the cuts would also greatly improve the placement of the cuts selected and the assemblability of the model. This requires closing open faces of the mesh through triangulation along with projecting the vertices along the boundary edges of the cross section onto a 2D plane and computing the convex hull to determine if connectors can be placed on that cross section. We started to implement this objective function but due to time constraints were unable to finish it.

Limitations in restricting the cuts to be planar prevent more natural cross sections on the object that could be desirable from the users. Cross sections that are not flat can add more stability to the assembled model as well.

While the algorithm that we have presented needs significant improvements, we are still proud of what were able to accomplish.

6 Division of Labor

In general, we worked together on developing and understanding the algorithms necessary to accomplish the project, coding alongside each other and sometimes together to debug and ensure we were on the right track. Here is a list of the individual tasks and who primarily worked on what. In total, we approximate that the combined man hours spent on the project was 125 hours.

- Setting up the project: Jason
- Adapting code from past homework assignments for use in our project: Uyen
- Developing a basic BSP Tree data structure to represent the mesh: Jason
- Basic rendering to use different colors for partitions: Uyen
- Rewriting the code to handle multiple Mesh objects for storing and rendering: Jason
- Initial mesh cutting algorithm: Jason
- Improved cutting algorithm: Uyen
- Beam search algorithm: Jason
- Evaluating cuts algorithm: Jason
- f_{part} objective function: Uyen
- f_{util} objective function: Uyen

References

[1] D. E. Muller and F. P. Preparata, "Finding the Intersection of Two Convex Polyhedra," *Theoretical Computer Science,* vol. VII, no. 2, pp. 217-236, 1978.

[2] L. Luo, I. Baran, S. Rusinkiewicz and W. Matusik, "Chopper: Partitioning Models into 3D-Printable Parts," in *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 2012.

[3] H. Medellin, T. Lim, J. Corney, J. Ritchie and J. Davies, "Automatic Subdivision and Refinement of Large Components for Rapid Prototyping Production," *Journal of Computing and Information Science in Engineering,* vol. VII, no. 3, pp. 249-258, 2007.

[4] M. Yao, Z. Chen, L. Luo, R. Wang and H. Wang, "Level-Set-Based Partitioning and Packing Optimization of a Printable Model," *ACM Transactions on Graphics,* vol. XXXIV, no. 6, p. 11, 2015.