Cage Generation for Mean-Value Deformation

Quincent Masters, Aaron Hornby, Mikhail Starikov

Abstract

We have implemented a robust object-oriented program to assist in generating cages for various models and a test of the cages through mean-value deformation. The cage generation methodology was drawn from Xian's paper so that given an arbitrary model, we can calculate and find the tightest fitting Oriented Bounding Box for the model.

Oriented Bounding Boxes are widely used in physics and gaming to detect model collision as well as, more recently, being used as an alternative to skeletal-based animation. The boxes are usually not generated when we construct a model within a software (such as Blender). As such, we hope this program will be useful in future iterations.

1 Overview

Our paper is divided into the following sections:

Background: Covers the papers, research, and theory that founded our program.

Implementation: The various subtleties that we used to implement the ideas covered in the **Background**. We explain and disclose some steps we've made that may differ from the original ideas covered in **Background**.

Results: Mostly pictures of the results from the implementation. Some remarks are given to explain the result.

Future Work: Issues that could be resolved or improved are addressed here.

References: Our sources that we consulted throughout the progress of this project.

2 Background

Our paper goes over the effectiveness and rationale for focusing on the cage-based generation of arbitary models. In our paper, we use a pre-generated model and cage, `Armadillo`, as a control to

check our algorithm and application's efficacy. The explications and details in constructing these cages will be generic with constraints so that given an imported model, we can properly generate a cage to use for future cage-based deformations and animations.

2.1 Generating Cages

As the primary section of the project, as we continued to research into this section, we have learned that cage generation is a well-known problem within the video game development community. As such, generating cages for arbitrary models is quite a challenge. For our generation of cages, we have decided to follow a few different papers to efficiently utilize the creation of the cages.

2.1.1 Oriented Bounding Box Generation via Principle Component Analysis

Oriented Bounding Box (OBB) is an alternative to the Axis-Aligned Bounding Box (AABB). In an AABB structure, the faces of the boxes are parallel to the three coordinate planes (de facto X,Y,Z-planes). Rather than having this constraint, OBBs utilize the same set of vertices to generate a box that encapsulates the set of vertices without needing to orient the box so its faces hold the parallel structure. To do this, the use of statistics and numerical linear algebra is utilized to help analyze and generate an appropriate OBB.

When we begin calculating the OBB, due to the use of statistics, the data for our linear system can produce three areas of confusion/trouble:

- Noise - inaccuracy in the measurements or data
- Rotation - lack of knowledge in the underlying variables in the system we are utilizing
- Redundancy - lack of knowledge whether our variables are independent or not (thus leading to multiple redundant calculations and points).

Principle Component Analysis, or PCA, is a technique that uses orthogonal transformation to convert our vertex set into a set of linearly uncorrelated variables, known as principle components. By doing this, PCA allows use to manage the three areas of confusion:

- Isolates noise
- Eliminates rotation effects
- Separates redundant degrees of freedom

To help determine the relative relationship between variables, we use statistics to help determine the relationship. Covariance measures the degree of the linear relationship between two variables. Smaller covariance values dictate the relative independence between two variables. The general equation for covariance is:

$$cov(X_i, X_j) = E[(X_i - \mu_i)(X_j - \mu_j)], \tag{1}$$

where μ_i and μ_j are the expected values (mean) for X_i and X_j, respectively. However, since our vector sets are hardly simply one dimensional, or rarely 2D, we can generalize this concept for multiple dimensions by using a covariance matrix:

$$A = \begin{bmatrix} cov(x,x) & cov(x,y) & cov(x,z) \\ cov(x,y) & cov(y,y) & cov(y,z) \\ cov(x,z) & cov(y,z) & cov(z,z) \end{bmatrix} \tag{2}$$

Notice that the diagonal elements of A represent the variances of our variable set. In fact, the off-diagonal values represent the covariances in our variable set. Large values for the off-diagonal covariances reflect a distortion in our data set. To counteract this distortion, we can redefine our variables as a linear combination of others, effectively changing our matrix A. A simpler way of viewing this is if we change the matrix, so that the off-diagonal values are close to zero. Simply put, we want to diagonalize the matrix.

Notice that the way the matrix is set up, and the behavious of covariance, it is already symmetric. Since most of our coordinates are over $\mathbb{R}$, we guarantee that we get three eigenvectors in the change-of-basis matrix when we diagonalize A. Simply put, the eigenvectors of our covariance matrix make up the orientation of our OBB. Large eigenvalues reflect large variances, so to minimize we align our OBB along the eigenvector corresponding to the largest eigenvalue.

Determining the bounds, size, and center of box can be done once you diagonalize the covariance matrix. To first determine our center, we need to have eigenvectors computed. Projecting our set of vertices onto these vectors allow us to firmly "shift" the origin until there is an equal signed projection length. For that, we cannot simply take the mean value of our coordinates. For example, given the set of variables {-5,4,4,5}, our OBB center will be at 0, not 2.

Computing the half-lengths (the overall size of the bounding boxes), we first compute the vector between each variable in our set and the OBB center. Once the vectors are established, we project them onto our OBB axis. For each axis you want to keep track of the relative extremes. These extremes, relative to your center, determine the lengths of your box. For a better visualization, a 2D example is given below.

Example Given the following ten set of data points, we want to generate an OBB. {(3.7, 1.7), (4.1, 3.8), (4.7, 2.9), (5.2, 2.8), (6.0, 4.0), (6.3, 3.6), (9.7, 6.3), (10.0, 4.9), (11.0, 3.6), (12.5, 6.4)}

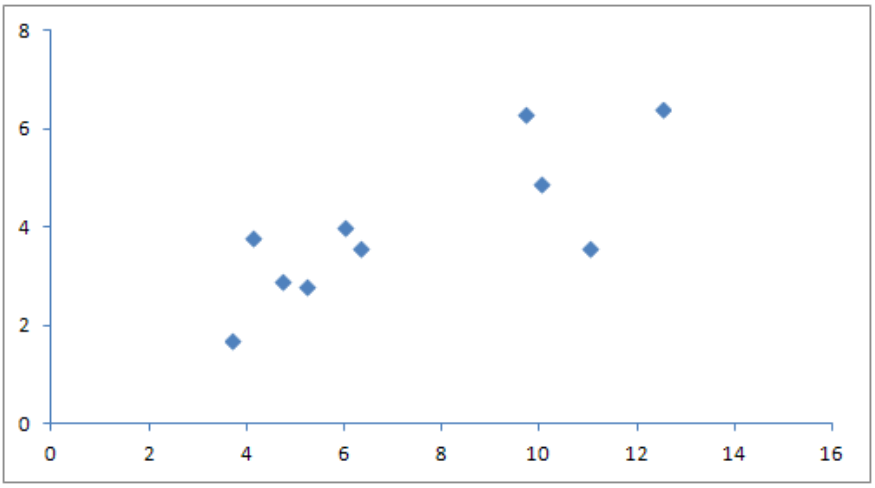


Figure 1: Set of Data Points

Computing the variances and covariances gives us the following 2x2 matrix:

$$A = \begin{bmatrix} cov(x,x) & cov(x,y) \\ cov(x,y) & cov(y,y) \end{bmatrix} = \begin{bmatrix} 9.0836 & 3.365 \\ 3.365 & 2.016 \end{bmatrix}$$

Notice the large off-diagonal values in the matrix. This is due to the rather large "distortion" in our initial data set. To offset this distortion, we diagonalize and apply change-of-basis to the matrix giving us the following result (with eigenvectors):

$$A = \begin{bmatrix} 0.928491 & 0.371355 \\ -0.371355 & 0.928491 \end{bmatrix} \begin{bmatrix} 10.4294 & 0 \\ 0 & 0.670152 \end{bmatrix} \begin{bmatrix} 0.928491 & -0.371355 \\ 0.371355 & 0.928491 \end{bmatrix}$$

From this diagonalization, we get the following eigenvectors:

$$\vec{v}_1 = \begin{bmatrix} 0.928491 \\ 0.371355 \end{bmatrix} \vec{v}_2 = \begin{bmatrix} -0.371355 \\ 0.928491 \end{bmatrix}$$

By following the projection scheme laid out above, we get the following figure for our OBB. Our center is at (8.10, 4.05) with half-lengths of 4.96 and 1.49 units. The span of the OBB axes are marked in green.

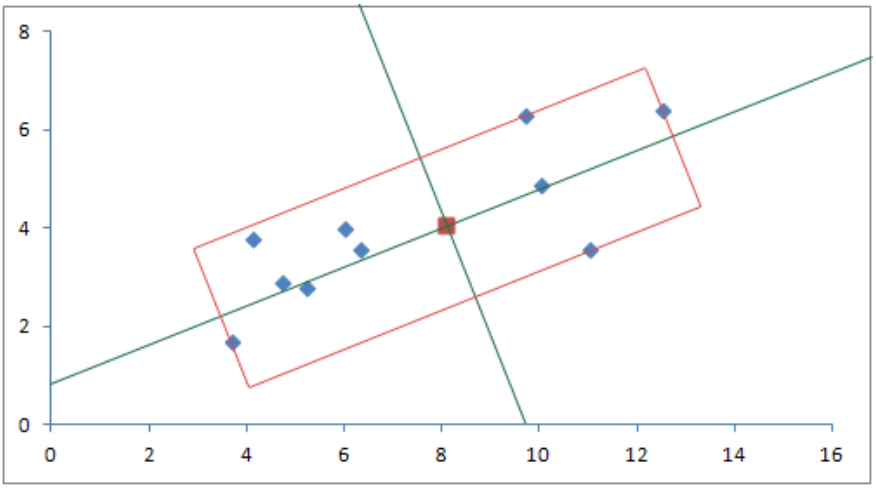


Figure 2: Set of Data Points with OBB

2.1.2 Automatic Cage Generation

Automatic cage generation tackles many of the issues that come with the various model types and previous cage generation techniques that are available. First, suppose that the model is complex.

Using an sketch-based cage generation could take quite some time depending on both the complexity of the model and the targeted deformation. In addition, the rigidity and spacing of the cage relative to the model is usually quite large, relative to model size. As a result, sketch-based generation limits the effectiveness of generating cages for complex models. Due to the issues with the other methods of generation (sketch-based or subdivision), Xian employs a more robust method of tightly enclosing the cages around their models and generating a complete cage, which are encompassed within four key parts.

Voxelization and Point-Set Generation The first step in the automatic cage generation is generalizing voxels and a point set for our model's mesh. To voxelize the model, M, first compute the OBB, O, of the model using PCA. After generating the O, split the box into voxels of pre-determined voxel size s. After the redefining of the OBB, the voxels should be categorized into one of three categories:

- Feature Voxels - voxels that intersect with the mesh
- Inner/Outer Voxels - defining inner/outer characteristic is obtained from applying the scan-conversion algorithm

From here, we generate our point set for the later applications of this cage generation. We do this by first adding all of our mesh vertices to the set, P. In addition to the mesh vertices, we add in the barycenters of the inner voxels to our set. However, this situation primarily applies to closed models! For open models, they should be repaired to be closed in advance. From our point set, we recalculate our PCA OBB using P as our data set. The resulting OBB is more closely aligned with the original model than just PCA OBB on M. The figure below shows the difference.

Improved OBB Slicing Rule The slicing rule we will follow would be the one implemented by Xian. Comparing the results from their paper, their slicing rule allows for a more effective and clean slice when dealing with complex models with multiple curves or sharp edges. As such the slicing rule, both original and amendment, will be covered.

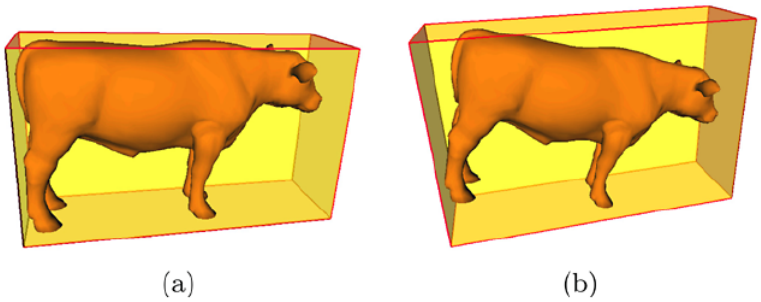


Figure 3: Comparison of PCA results on the point set P and the mesh vertices. (a) PCA result on the point set P. (b) PCA result on the mesh vertices.

The original proposed OBB slicing rule is that we split the OBB by a plane that is perpendicular to the longest edge and passes through the barycenter point of the original OBB. One problem that arises with this method is for complex models with multiple depths and curves, the barycenter slicing gives us awkward and loose fitting cages. This method ignores the original shape of the model and just looks at the OBB. The goal, as seen in the figure below, is to geometrically slice the model at the point where the model/shape changes the most.

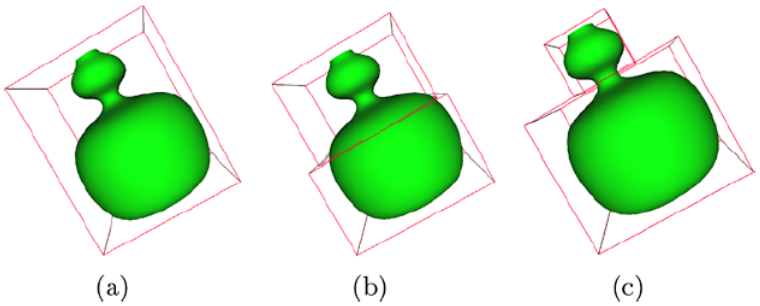


Figure 4: (a) General OBB for the vase (b) Original OBB slicing scheme (c) Xian's improved slicing scheme.

The idea behind this seems sound, wanting to separate complex and drastic changes to be kept in separate sections of the subdivided OBB. To begin sketching this method, we first take the lower left corner of the OBB as the origin. From this origin, we construct a *local* Cartesian coordinate with the longest, second longest, and shortest edges of the OBB as the x, y, z axes, respectively.

From this local Cartesian coordinate system, we want to define the minimum cross section area function, $f(x)$, where the cross section is the intersection between the model and the splitting plane at x. If the cross section only has one part, $f(x)$ is simply the area of the part. However, for more complex models, if a cross section contains more than one part, $f(x)$ is defined as follows:

$$f(x) = \min_i\{\text{area}(s_i^x)|i = 1, 2, \ldots, k\}, \tag{3}$$

which simply returns the smallest cross section area if there are multiple cross sections in the slicing plane. From this equation, we could graph the minimum cross section area function. Geometrically, the area with the greatest change occurs at the location where there is the greatest jump in values. First we compute the local minima and maxima of the function. Then by simple comparison, we can find a value x^s, the x coordinate for slicing, that would maximize the jump between the two values. The rational for creating a local Cartesian coordinate system is if we do not find any clean cuts along the x axis, we move onto the y, and finally the z axis. However, if we fail to find a "jump" in any of the three axes, we revert and employ the default OBB slicing method. This case would only occur is *all* the cross section area functions are completely smooth and flat.

In the method employed by Xian, since we voxelize our model, we can simplifiy calculation of the cross section area as to just "counting" the number of inner and feature voxels that intersect with the splitting plane at x. Below is an example the slicing method on a model of a cow along with the cross section area plotted.

Termination Conditions As we continuously subdivide an OBB, how do we know when to stop? What would help us determine when the cage is "refined" enough for export or other applications? We can ask the user for a defined value η, ζ that allows us to utilize a calculation method in Xian's paper to determine when the OBB subdivision should end. Of these two values, we look at two differnt constraints: the shape of the model and the shape of the OBB.

Shape of the Model: Given the model, we can determine when the OBB should end. Since we know that the cross section area function gives us strictly positive values and it reflects the variety

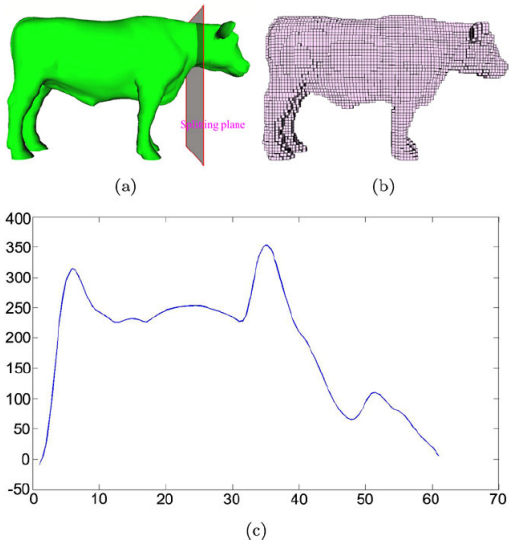


Figure 5: (a) Cow model and place with greatest change (b) Voxelization of the cow (c) Minimum cross section area function.

of shapes within the model, we can ask for η to be strictly within the range $[0, 1]$. This construction allows us to generate a function, T_1, that computes when the OBB slicing is done based on the shape of the model:

$$T_1 = \frac{\min f(x)}{\max f(x)}.$$

This equation is strictly bound between the range (0,1]. It follows that if $T_1 > \eta$, we do not need to split further, since the mesh contained in the OBB varies very little. Otherwise, the mesh still has quite a bit of variance (determined by user input value, η) within the OBB. However, if our shape/model is very regular, then the value of T_1 will be 1 from the very beginning (for example, a cube). Since we can have regular shapes with convoluted structures, we must consider having another way of checking on top of the shape of the model!

Shape of the OBB Given some shapes that have more regular features, T_1 and η comparison may not be sufficient on its own for determining when the OBB should stop slicing. On user input, the user defines a value between (0,1] for ζ. This ζ value determines when to stop the slicing of the OBB based on the OBB shape. Consder the shape of the OBB and the relative terinating condition

defined as follows:

$$T_2 = \frac{l_{min}}{l_{max}},$$

where l_{min} is the shortest length of the OBB and l_{max} the longest. Once again, like the first terminating condition, the values for the secondary terminating condition is also strictly positive. As the value of T_2 approaches 1, we approach our termination.

As such, we terminate the slicing and subdiving of the OBB when $T_1 > \eta$ and $T_2 > \zeta$ holds.

OBB Registration and Mesh Generation The OBBs generated by the previous methods can be disjoint and oriented in opposing directions (their normals intersect). In addition, the cages are uneven and the distances between the cages and the model are heavily varied, with the variance widely increasing. To handle this situation, consider the following scenario (figure taken from Xian's paper):

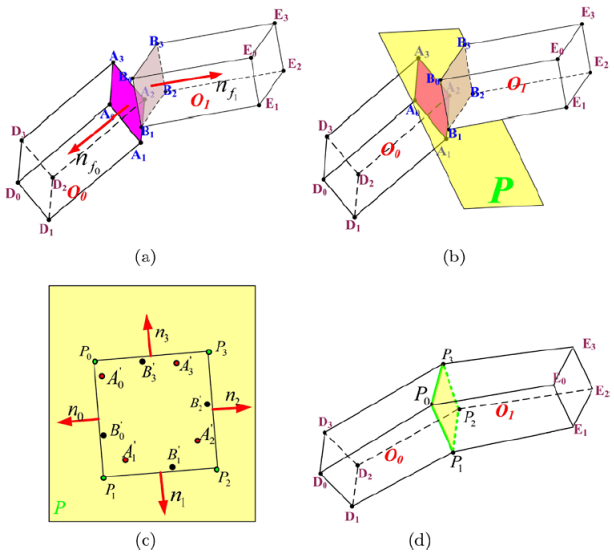


Figure 6: (a) Two adjacent OBBs with the two faces for merging to highlight. (b) The Plane (c) The projection point of the OBBs on the plane. (d) The merged OBB

To begin with, suppose we jave two adjacent OBBs, O_0, O_1 with the target faces we want to merge. Each of the target faces is "valid" for merging, however, Xian lists the constraints for the the merging of the OBBs to tbe successful:

- s_0, the area of the face on O_0, and s_1, the area of the face on O_1, have a ratio of at least 0.6. More explicitly, $\frac{s_0}{s_1} > 0.6$
- The angle between the normals of the faces targeted for the merge, the angle must be at least $\frac{2}{3}\pi$, or 120 degrees.
- Lastly, and most importantly, the sub-models contained within these OBBs are indeed connected!

One things to note is that the above conditions do not take into account the sizes of the OBB. It should be accounted for that if the adjcent OBBs in question are too varied in size, or the distance is too far apart, the merging of the OBBs may not occur.

Should the two OBBs meet the above criteria, they can be merged. The process for doing the merge can be daunting, at first, but overall quite simple as well. When we register the OBBs, we first must register the face of each OBB that we are trying to merge. When selecting a face, we must register the centers of each face, C_0 andC_1, respectively. To appropriately generate a plane, P, that will "house" the two faces, we can construct/find a plane, through linear algebra, that passes through the point $\frac{C_0+C_1}{2}$ and is perpendicular to C_0C_1. If a plane is found, we can project the vertices of the faces onto the plane and construct a 2D OBB for the vertices of the faces. Once the 2D OBB for the projection of the faces is constructed, we need to register and create a correspondence between the vertices of the faces to "link" them up.

Labeling our 2D OBB, F_b, we can begin to test the proper correspondence between the OBB faces, F_0 andF_1. For this case, we will go over the correspondence of F_0 and F_b. For our OBB, O_0, we want to compute the four side face normals. Similarly, we want to compute the four edge normals for F_b. By selection, there are four possible correspondence orientations for the adjoining of F_0 and F_b. As such, we want to select the one that maximizes normals between the two faces. As such

$$\max_{j=0,1,2,3} \Sigma_{i=0}^{3} n_i \cdot n^0_{((j+i) mod 4)}$$

After registering, the adjoining process is simply merging the two OBBs with the respective coordinate orientations that resulted in the maximized normals. Lastly, an intersect check is performed to ensure the newly merged OBBs do not intersect the model or other OBBs. Upon finishing this merging operation wherever it is possible, the final part of the algorithm is to return all the key corners that are outside of the model to generate/return a frame for future uses.

2.2 Mean-Value Deformation

In addition to generating cages, we have chosen to implement and use mean-value cage-based deformation to test the quality and accuracy of the cages generated. The common problem within computer graphics was defining a function from the vertices of the cage to its interior. Simply put we have our set of cage vertices, $V = \{v_0, v_1, v_2, \ldots, v_m\}$ and the set of points on our model, in this example, the Armadillo, is denoted as $C = \{c_0, c_1, c_2, \ldots, c_i\}$. The construction of defining the function between the cage and the vertices of the target object are developed carefully by [Ju, Schaeffer, and Warren 2005].

To begin, suppose we have a triangle with given vertices $\{p_1, p_2, p_3\}$ with intensities $\{f_1, f_2, f_3\}$. The intensity of the interior of the triangle (currently 2D) can be expressed by the following equation:

$$\hat{f}[v] = \frac{\Sigma_j w_j f_j}{\Sigma_j w_j} \tag{4}$$

where w_j is the area of the sub-triangle $\{v, p_{j-1}, p_{j+1}\}$. The above equation normalizes the intensity of the interior of the triangle. However, our meshes often includes vertices rather than just points and intensities, it is suffice to know that we can represence the vertex, v, as an affine combination of our vertex points, $\{p_1, p_2, p_3\}$. Extending from this, Ju defines for us Floater's interpolant for defining the weight functions:

$$w_j = \frac{\tan[\frac{\alpha_{j-1}}{2}] + \tan[\frac{\alpha_j}{2}]}{|vertp_j - v|}, \tag{5}$$

where α_j is the angle formed by the vector $p_j - v$ and $p_{j+1} - v$. The resule of this implementation is that by normalizing each weight function, w_j, by the sum of all weight functions generates the *mean-value coordinates* of v with respect to p_j. Further work shows that by maintaining a consistent

orienation for a 2D polygon and treating α_j as signed angles, the mean-value coordinates produce linear functions everywhere. More precisely, $\frac{w_j}{\Sigma_j w_j}$ is the affine combination such that

$$v = \frac{\Sigma_j w_j p_j}{\Sigma_j w_j}. \tag{6}$$

From this generation of mean-value coordinates in 2D, we can shift and begin the construction for closed triangular meshes in 3D. Given a closed surface P in $\mathbb{R}^3$, we want to construct a function $\hat{f}[v]$, for $v \in \mathbb{R}^3$, that interpolates an auxilary function $f[x]$ over P. As the paper states, to construct $\hat{f}[v]$, we first project a point $p[x]$ onto the unit sphere, S_v, centered at v. Next, weight the point's associated $f[x]$ value (if any) by $\frac{1}{|p[x]-v|}$ and integrate the new weighted function over the sphere. Placing the pieces together, we get a *mean-value interpolant* of the form

$$\hat{f}[v] = \frac{\int_x w[x,v] f[x] dS_v}{\int_x w[x,v] dS_v}, \tag{7}$$

where $w[x,v] := \frac{1}{|p[x]-v|}$. As mentioned in the paper, the new equation is an integral version of the discrete formula in Equation 1 as well as the discrete weights in Equation 3 and this equation differ only by a numerator. As for retaining the features from Floater's formula with $\tan[\frac{\alpha_j}{2}]$ becomes present when we take the integration over the whole sphere. From this new equation, we can see that the mean-value interpolant satisfies three properties, pointed out in Ju's paper:

Interpolation: As $v \to p[x]$, $\hat{f}[v] \to f[x]$. This can be seen as the weight function, $w[x,v] \to \infty$ as $p[x] \to v$.

Smoothness: $\hat{f}[v]$ is well-defined and smooth for all v not on P. This follows since the projection of $f[x]$ onto S_v is continuous in the position of v and taking the integral of this continuous process/function yields a smooth function.

Linear Precision: If $f[x] = p[x]$ for all x, then the interpolant $\hat{f}[v]$ is identically v for all v. This happens since the integral of a unit normal over a sphere is exactly 0.

Mean-value coordinates for closed meshes, more specifically triangular meshes, are a natural generalization of the 2D mean-value coordinates. For calculating the mean-value coordinates, we need

to include two key equations: the *mean vector equation*:

$$m = \Sigma_i \frac{\theta_i n_i}{2}, \tag{8}$$

where given a spherical triangle $\bar{T}$, let θ_i be the length of the i^{th} edge and n_i be the inward unit normal towards the i^{th} edge, as well as the weights equation:

$$w_i = \frac{n_i \cdot m}{n_i \cdot (p_i - v)} \tag{9}$$

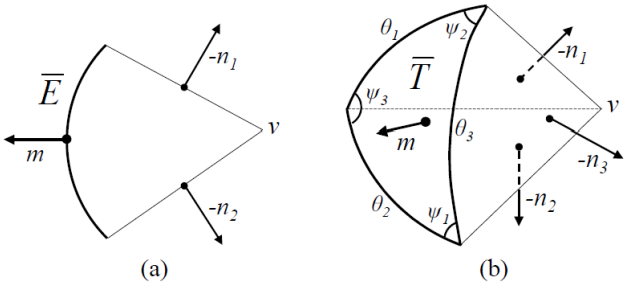


Figure 7: Spherical Triangle and its appropriately labeled parts

The issues following using these two equations for generating the mean-value coordinates involve two specific cases: the stability case (for when computing weights, we may get a denominator of 0) and the co-planar case. Each of these cases are listed out below in terms of how the paper describes handling them.

Stability: The case of stability comes with the chance that when we compute the weights for our cage mesh, we may end up with a calculation error that has a zero in the denominator. By substituting Equation 8 into 9, we get the following equation:

$$w_i = \frac{\theta_i - \cos[\psi_{i+1}]\theta_{i-1} - \cos[\psi_{i-1}]\theta_{i+1}}{2\sin[\psi_{i+1}] sin[\theta_{i-1}] |p_i^k - v|}, \tag{10}$$

where k is the edge numeration. This monster of an equation is a little tough to look at, but all variables mentioned in the equation, apart of ψ should be recognizable. ψ_i, here, are the

dihedral angles between the faces with normals n_{i-1} and n_{i+1}, see Figure 7. To calculate $\cos[\psi]$ without computing normals, we can simply use Beyer's formula for half-angles on spherical triangles to simplify the amount of memory usage we would need:

$$\cos[\psi_i] = \frac{2\sin[h]\sin[h-\theta_i]}{\sin[\theta_{i+1}]\sin[\theta_{i-1}]}, \tag{11}$$

where $h = \frac{\theta_1+\theta_2+\theta_3}{2}$. Apply substitutiong of Equation 8 into 7 and we get a concrete formula for computing w_i that only involves using $|p_i - v|$ and θ_i.

Co-planar: For testing the co-planar cases we have two different scenarios to check: the first is if the point v is on the plane inside our triangle, T. We can easily check this by simply checking the equality $\Sigma_i\theta_i = 2\pi$. However, if the point(s) lie on the plane outside of T, then we can use Equation 8 to generate a stable computation for $\sin[\psi_i]$. In this scenario, v lies on the plane outside of T if any $\sin[\psi_i] = 0$.

To apply a model deformation in these new mean-value coordinates, the paper suggests that we use the following model to compute the new set of vertices for our model. Given our set of points for our cage, $V = \{v_0, v_1, v_2, \ldots, v_m\}$, and the mean-value weight functions w_j for each vertex v_j with respect to model vertex c, the set of new positions for vertices in our model would be defined as

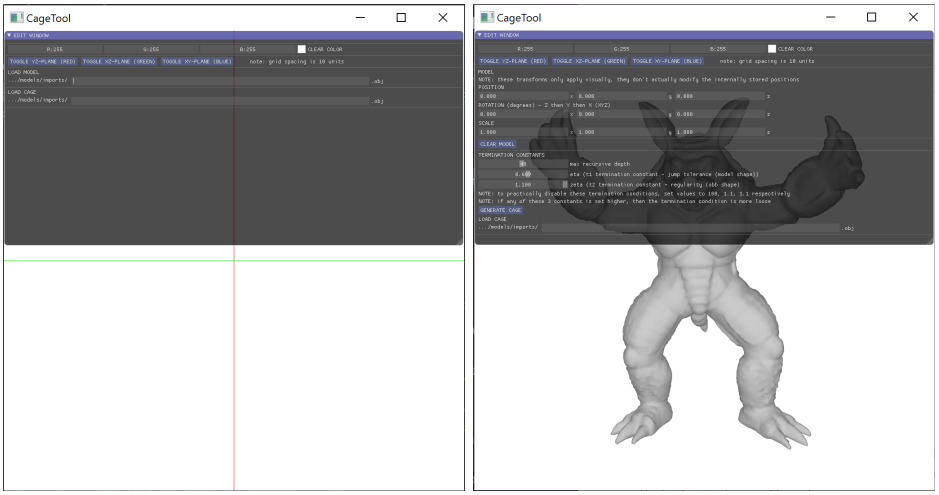
$$\hat{c} = \frac{\Sigma_j w_j \hat{p}_j}{\Sigma_j w_j}, \tag{12}$$

where $\hat{p}_j$ is the new position of the deformed cage vertex p_j. Consequentially, due to linear precision, if $\hat{p}_j = p_j$, then $\hat{c} = c$.

3 Implementation

3.1 Main Application

We've elected to use ImGui as our defacto interaction for loading in files as well as some of the interactions within the application. We are not very versed in creating applications ourselves, so this helps alleviate some of the technical problems with constructing a well-designed, functioning UI.



(a) Initial load of the program. (b) UI after successfully loading in a model.

When loading, we initially are zoomed in to the origin of the plane. When we load in our models, we restrict our models to have triangulated. The application moves from the initial UI to the next UI (shown above) **only** if the application has successfully loaded in the model. Depending on the coordinates of the `obj` file, the application may move to the next phase even if the model has not appeared. This is simply due to either the model being too small or veered far from the origin in the original `obj` file. Some key features in the UI:

- Clear Model: removes all model verts and objects to allow the user to load in a new model.
- Recursive Depth: Allows the user to control how many "slices" occur in the OBB generation if available.
- Generate Cage: Generates an OBB for the user.
- Compute Cage Weights: Computes the cage weights via mean-value coordinates for deformation.

In addition, there is only one restriction to the `obj` files that the program can utilize: the faces must be triangles. Although this is a simple restriction, our `obj` files are formatted in the exported manner that Blender typically provides.

Once the model has been loaded there are two options: generating a cage or loading a cage file. For loading a cage, we can choose to load in another `obj` file that contains the relevant cages for `Armadillo` in the below, Figure 9. If there is a successful load of the cage, the UI again shifts and has an option to calculate the mean-value coordinates of the cage and the model.

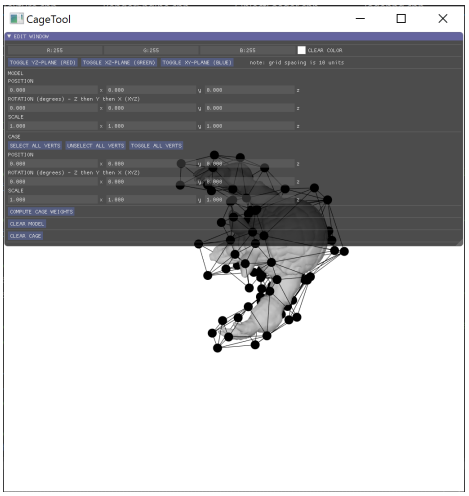
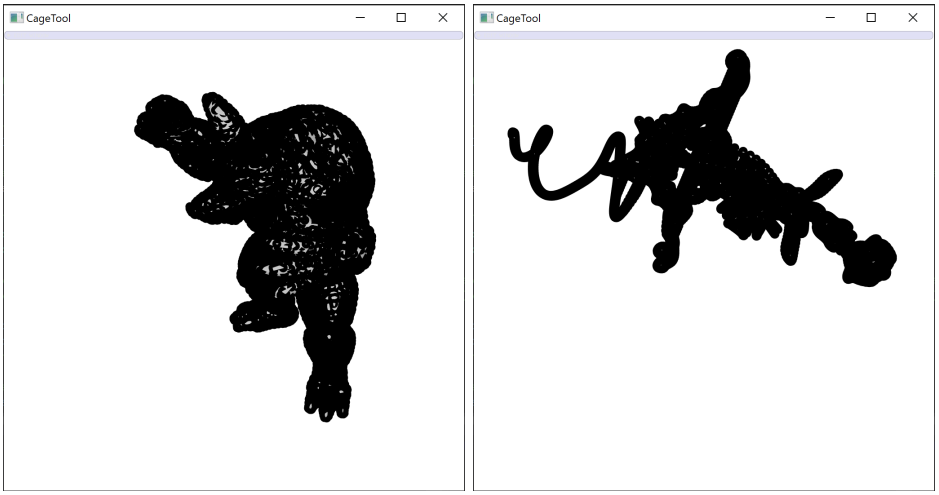


Figure 9: Successful load of a cage for Armadillo

3.1.1 Algorithm for Generating Cages

Once a model has been successfully loaded in, the cage generation goes through the first main step of the paper to begin calculating the point set P for the OBB. In this step, the program first applies a voxelization of the model.

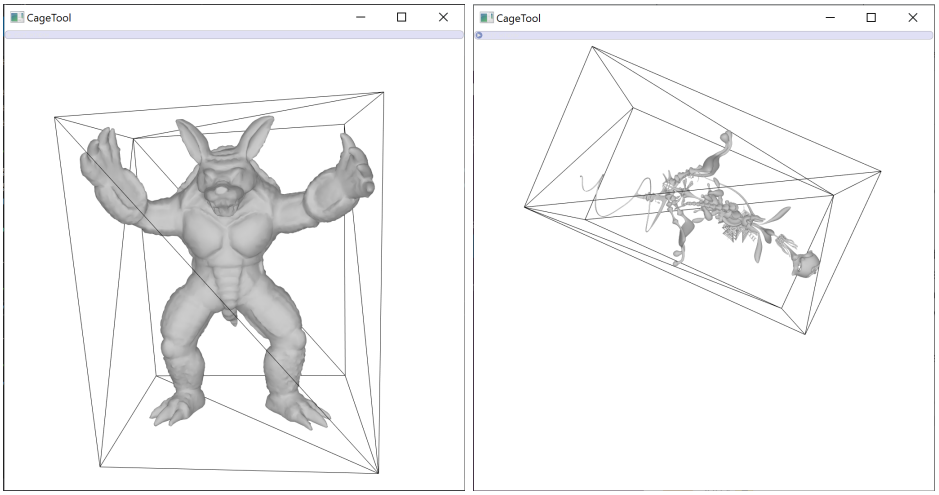
Due to the fine nature of `Genus 131`, the voxelization of the model simply looks like a massive shadow outline of the model. From the voxelization we do a scan check, utilizing the tips from the Wolfire blog (link under references), to categorize the voxels appropriately. From this check, we are able to construct our point set P from the original mesh vertices, inner voxels, and feature voxels. This addition of feature voxels is a judgement call made for the models that have very thin lines that do not have any inner voxels available, like `Genus 131`. From here, we can construct the initial OBB for each of the models via PCA.



(a) Voxelization of the `Armadillo` (b) Voxelization of `Genus 131`

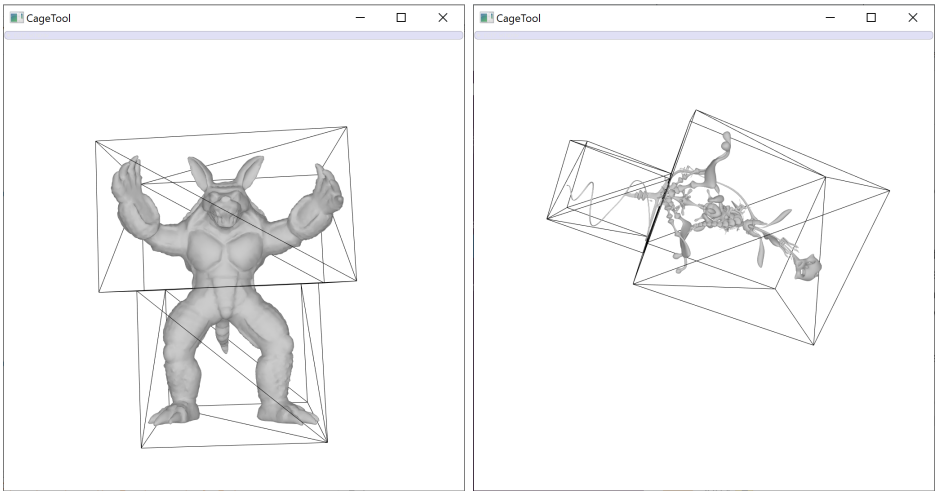
Figure 10: Voxelization of model samples

From the internal construction of the OBB, we can set end conditions on the recursion of slicing the OBB, to recursively split each OBB until the count condition ends. Included in the UI are explanations how to run and use the various variables that we mentioned in the background section. More precisely, we included a slider for η and ζ to adjust the values. If the user has no preference for the values of η and ζ, we recommend setting the value to be 1.1, the default setting for η and ζ.



(a) `Armadillo` (b) `Genus 131`

Figure 11: Initial OBBs of our test models



(a) `Armadillo` (b) `Genus 131`

Figure 12: First split of initial OBBs of our test models

More explicitly, our program implements the following algorithm for generating the cages:

1. Load in a model M. Upon successful load, the user is given the option to load in a cage or generate a cage.

2. From our model M, generate an initial OBB using PCA. However, rather than shifting the origin, we use the barycenter (mean) as our origin for the OBB.

3. Voxelize the volume of the OBB and categorize the voxels as either inner, outer, or feature voxels. Save the voxel size, s.

4. Construct our point set P to consist of inner voxels, feature voxels, and mesh vertices.

5. Recompute the OBB using our point set P via PCA. From this computation, we also add to the OBB three basis vectors $v_3 \geq v_2 \geq v_1$. We add an extra voxel on each half-side to reduce boundary issues.

6. Voxelize the new OBB.

7. We can apply the slicing rule. Unlike Xian's, if we fail to slice along all three axes, v_3, v_2, v_1, we skip this OBB. We have a check to make sure the split is not too close to any boundary to reduce the chance of really thin boxes appearing.

8. Check the termination conditions of T_1, T_2, and recursive definition. All of these are user defined to help focus the generation of the cages to be user tailored. *Note: The checks that occur in step 7 when we voxelize override any user-defined termination clauses.*

3.1.2 Cage-Based Deformation

Although we ambitiously stated the goal of implementing all the cage-based deformations, we stuck with the Mean-Value Deformation implementation and focused on the cage-generation aspect of the application. That being said, our implementation of the Mean-Value Deformation. The application handles both the calculation and applying the deformation in two function calls: `computeCageWeights()` and `deformModel()`.

`computeCageWeights()`: The function utilizes the paper mentioned in Section 2.2 and their provided algorithm to roughly implement the calculation of the mean-value coordinates. The function behaves in the following steps:

1. Initiating the cage vertices weights for each model vertex `i`
2. Draw a sphere centered at `i`
3. For each triangular face mesh, save the vertices
4. Calculate the project of the points onto the sphere
5. Calculate the arc-lengths fo the spherical triangle
6. Check the influence of the faces by finding the center of the sphere and which face it passes through
7. Zero the weights for all other faces
8. The selected face will only have influence on the model vertex `i`
9. Compute the weight distribution per vertex
10. Normalize and assign the weights for the model vertex

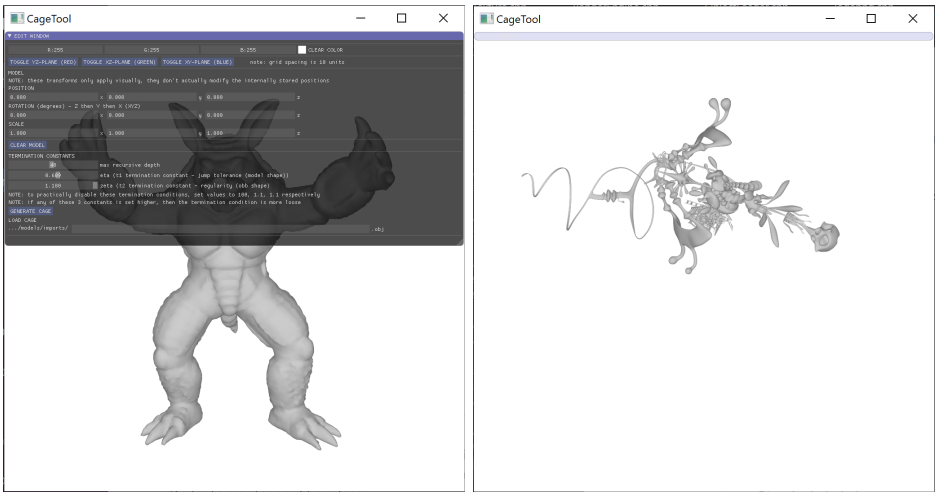
The more mathematical/computational explanation is within Section 2.2.

`deformModel()`: We apply the deformation formula, see Section 2.2, to recalculate every model vertices' new position. After which, we recalculate all the normals for the model after the shift and change in the model's deformation.

4 Results

4.1 Cage Generation

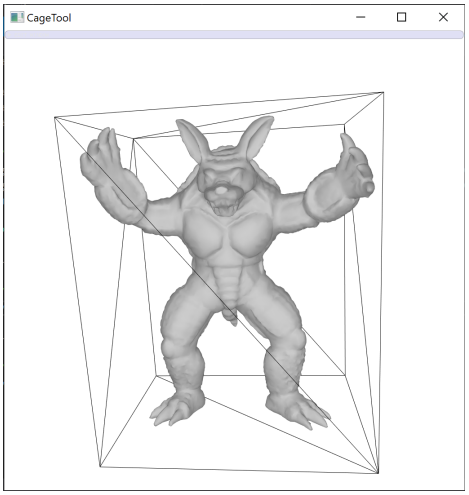
As with our cage generation, we use two core objects to test our algorithm and implementation on: the classic `Armadillo` and a model we found whose genus is 131, `Genus_131`.



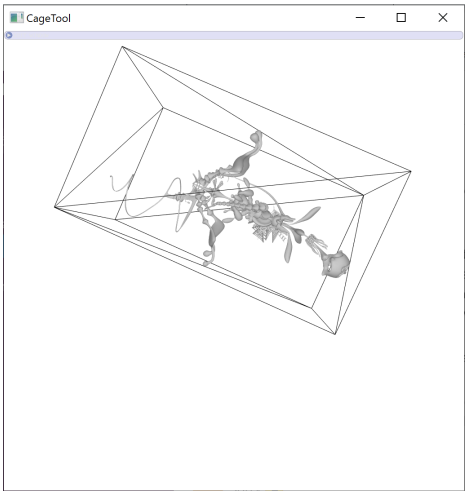
(a) Armadillo Model. (b) Genus 131.

Figure 13: Selection of test models

One unexpected result that we have encountered is that the algorithm and program handles "holes" quite well, surprisingly. Initially, we wanted to restrict the models we input to have no holes. However, the biggest issue we have encountered thus far are the intersecting and colliding OBBs within the model as we recursively split the initial OBB. Below the sequence of splitting our test models n times. In addition, due to the recursive nature of our slicing algorithm, we can expect a maximum of 2^n OBBs for a given model M with recursive depth n.

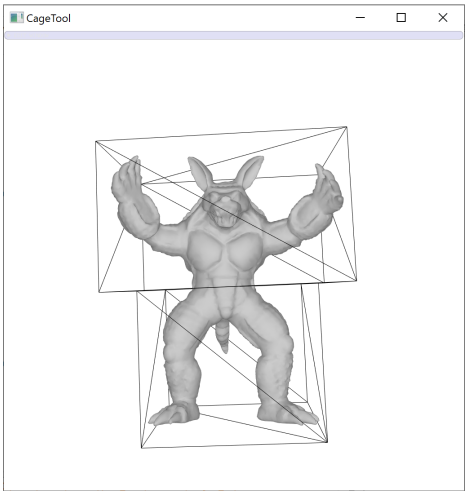


(a) Armadillo Model.

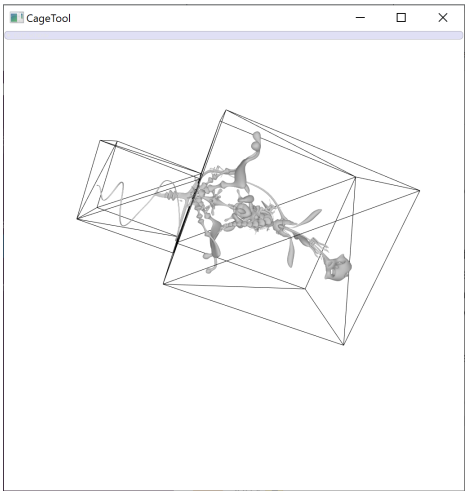


(b) Genus 131.

Figure 14: $n = 0$ splitting (Initial OBB)



(a) Armadillo Model.



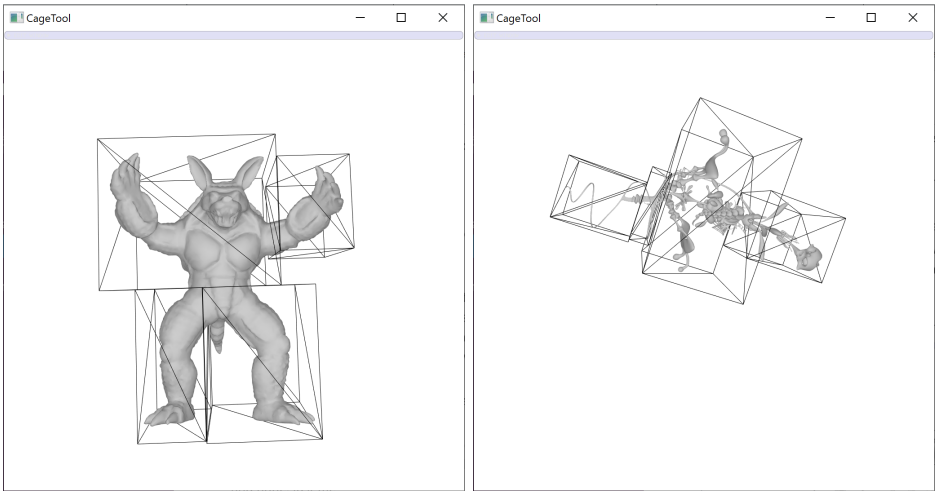
(b) Genus 131.

Figure 15: $n = 1$ splitting; max: 2 OBBs

4.2 Mean-Value Deformation

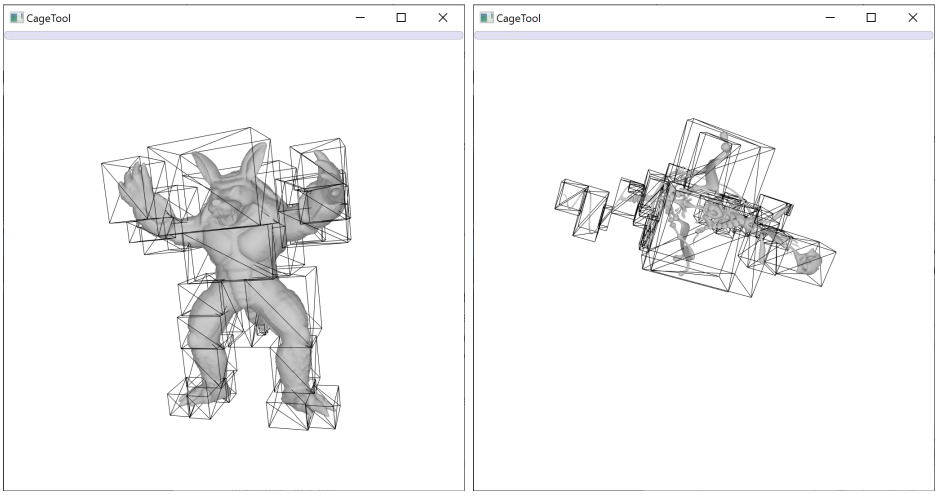
For mean-value deformation, we use both the original pre-generated cage for the `Armadillo` to test the deformation as well as our generated cages to see how our cages fair in the deformation of the model.

The original mean-value deformation on the pre-generated cage applies quite efficiently in the



(a) Armadillo Model. (b) Genus 131.

Figure 16: $n = 2$ splitting; max: 4 OBBs



(a) Armadillo Model. (b) Genus 131.

Figure 17: $n = 5$ splitting; max: 32 OBBs

direction of the three axes as seen in Figure 18. Following Figure 18 are the mean-value deformations on the bottom right-most OBB. As you can see in the deformation, we can move the outer-most OBB vertices without affecting any other part of the model. However, selecting one of the OBB vertices that has an edge intersecting with the model, we violate the constraints of mean-value deformation, thus create some bizarre deformations. However, for the test model, `Genus 131`, the complex nature

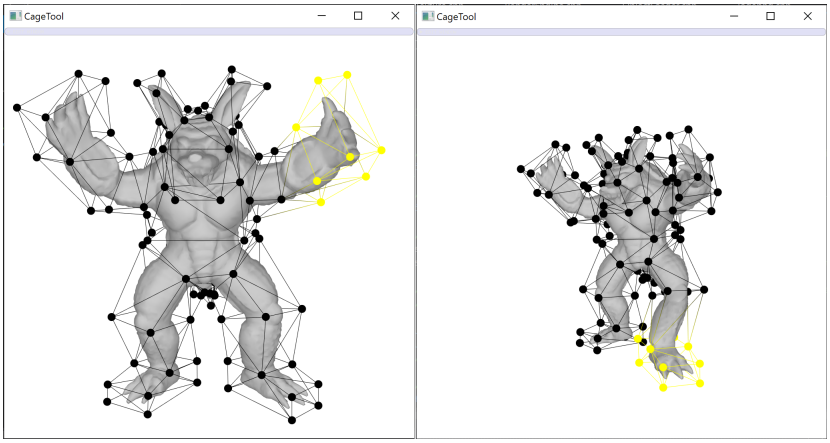
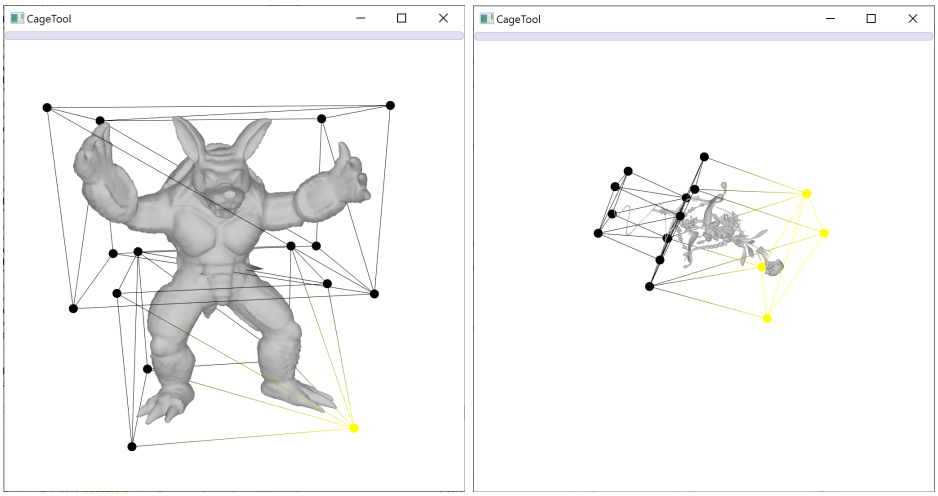


Figure 18: *Left:* Selection of right arm vertices. *Right:* Minor deformation of right leg.

of the model itself created some very close neighboring OBBs resulting in some weird deformations (See Figure 20

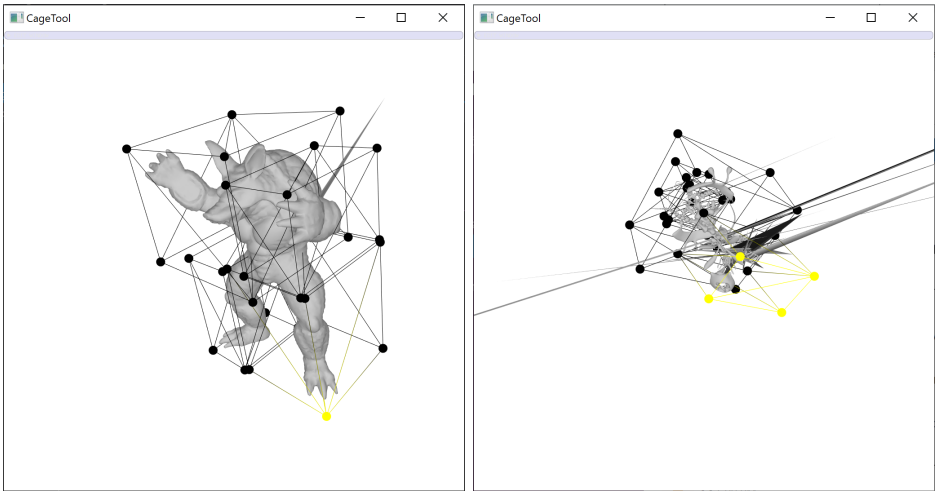


(a) Armadillo Model. (b) Genus 131.

Figure 19: $n = 1$ splitting; max: 2 OBBs

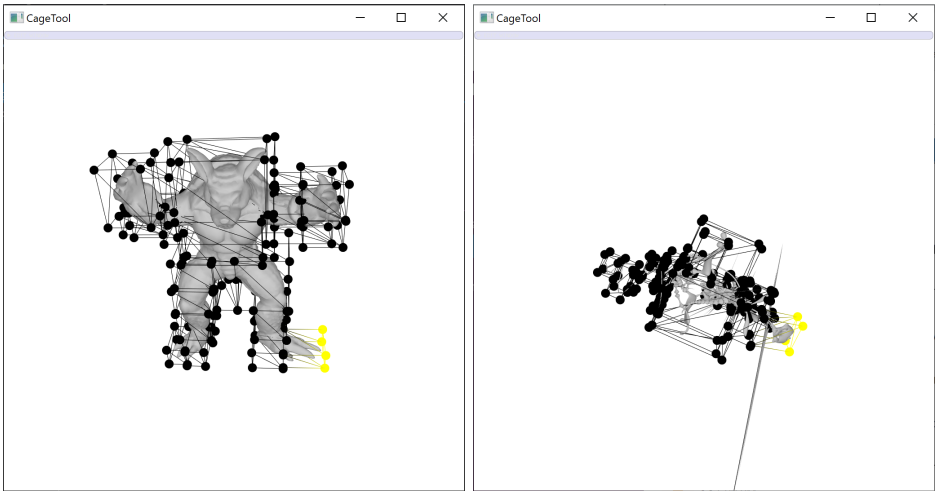
4.3 Rudimentary User Cage-Refinement

After generating the OBB from the user's input of recursion depth, η, and ζ values, the user can electively choose to edit the cages themselves. This refinement can occur by selecting the cage



(a) Armadillo Model. (b) Genus 131.

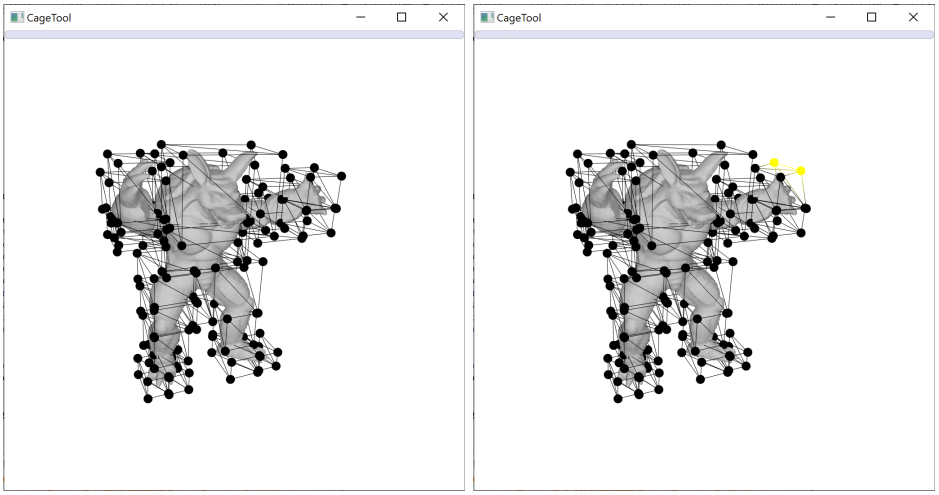
Figure 20: $n = 2$ splitting; max: 4 OBBs



(a) Armadillo Model. (b) Genus 131.

Figure 21: $n = 5$ splitting; max: 32 OBBs

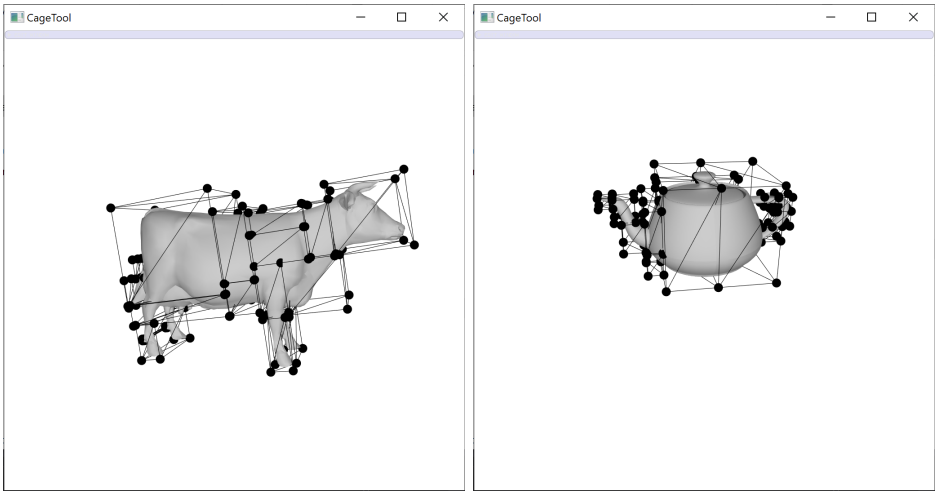
vertices themselves, without the weights computed, and simply shift their location. An example of the before and after editing the cage vertices on the right hand of the `Armadillo` is shown below in Figure 22.



(a) Before (b) After; highlighted yellow

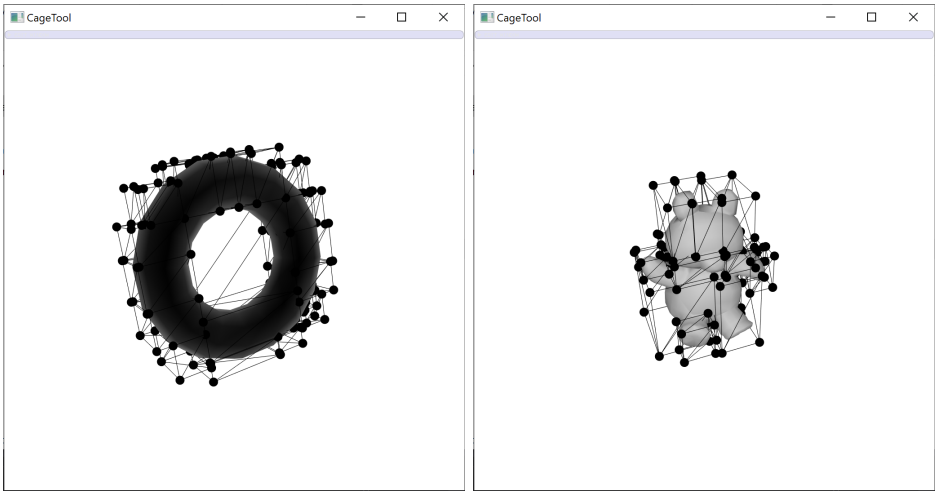
Figure 22: $n = 5$ splitting; max: 32 OBBs

5 Extra OBB Generation for various Models



(a) Cow (b) Teapot

Figure 23: $n = 5$ splitting; max: 32 OBBs



(a) Torus (b) Teddy

Figure 24: $n = 5$ splitting; max: 32 OBBs

6 Future Work

Some future work that can be continued is to optimize the program to better handle UI interactions (shift away from ImGui). This handling will ultimately allow the models to be opened from the user's desired location as well as optimize the general interface to be less abbrasive in the scheme. In addition to shifting away from ImGui, we would like to be able to handle a multitude of `obj` input file formats and accomodate for the various model structures in the modeling world, as currently we've restricted the `obj` files to only handle triangles.

For generating concise OBBs, it would be helpful in the future to implement the last two steps in the paper by Xian to further optimize the OBB generation: Register and boolean union of OBBs and mesh improvement. These last two key points would help alleviate the odd deformation results that appear in Figures 20 and 21. The issue that results in these erratic results are formed from the inner sections of the OBBs within the models and their respective intersections.

Another key part is to optimize many of the algorithms within the program to help reduce the time in calculation. Depending on the resolution of the model, the time it takes to generate cages or even apply a deformation can cause a "lag" effect.

For deformations, we only have deformations (displacement) along the three coordinate axes at the moment. One possible extent would be to apply rotations in the three directions as well.

References

[1] Pascal Laube and Georg Umlauf. "A Short Survey On Recent Methods for Cage Computation", SInCom 2016. Accessed via: http://www-home.htwg-konstanz.de/~umlauf/Papers/cagesurvSinCom.pdf

[2] Chuhua Xian, Hongwei Lin, and Shuming Gao. "Automatic Cage Generation by Improved OBBs for Mesh Deformation", Springer 2011. Accessed via: https://tinyurl.com/snm6vua.

[3] Tao Ju, Scott Schaefer, and Joe Warren. "Mean Value Coordinates for Closed Triangular Meshes", SIGGRAPH 2005. Accessed via: https://www.cse.wustl.edu/~taoju/research/meanvalue.pdf

[4] Wolfire Games Blog. Accessed via: http://blog.wolfire.com/2009/11/Triangle-mesh-voxelization

[5] Genus 131 Model sample. Accessed via: https://www.cs.cmu.edu/~kmcrane/Projects/ModelRepository