Scientific Visualization
Project I

Milian Wolff

November 14, 2011

In the first project for the scientific visualization class by Eugene Zhang I got some first hand experience in 3D visualization algorithms, Java programming and JavaView. While the former was very interesting and gave many opportunities to play with parameters, models and algorithms, the latter two I could have done without. Especially the severe lack of up-to-date documentation for the required beta release of JavaView along with its horrible API made me waste a lot of time.

The source code of my exercise solutions can be found online under

`https://github.com/milianw/scivi`

Contents

1 Euler Characteristics

The relationship between the Euler characteristic $V - E + F$ and the number of handles turns out to be

$$V - E + F = 2 - 2H. \tag{1.1}$$

This is in so far intuitive as 2 is the Euler characteristic of a sphere, and 0 that of a torus. Now every time we add another handle, we can say we add a torus - and hence decrease the Euler characteristic by two.

The code for these results can be found in the class `Ex1_1`. When executed, the user can select an `*.obj` file and the program will show it's Euler characteristic as computed by JavaView's `PgPointSet::getEulerCharacteristic()` method. It then continues to iterate over all sibling files with the `*.obj` extension and repeats the procedure.

1.1 Sphere

The sphere has no handles, as can be seen in fig. 1.1a. As such its Euler characteristic is of course 2.

1.2 Torus

The torus has a single handle and hence an Euler characteristic of 0. See also fig. 1.1b.

1.3 Bunny

The bunny as seen in fig 1.1c has no handles and is thus topologically equivalent to the sphere with an Euler characteristic of 2.

1.4 Hand

The hand I first suspected to be handle-free. But JavaView calculated an Euler characteristic of 0, which made me look for a handle. And indeed, there is a small hole between index and middle finger, which can be seen in fig. 1.1d.

1.5 Feline

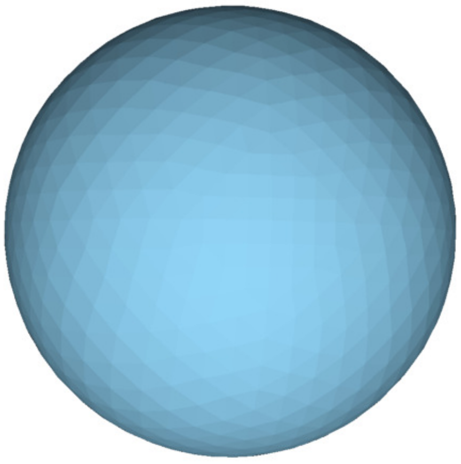
The feline has two handles at its tail, which is equivalent to an Euler characteristic of -2. See also fig. 1.1f.

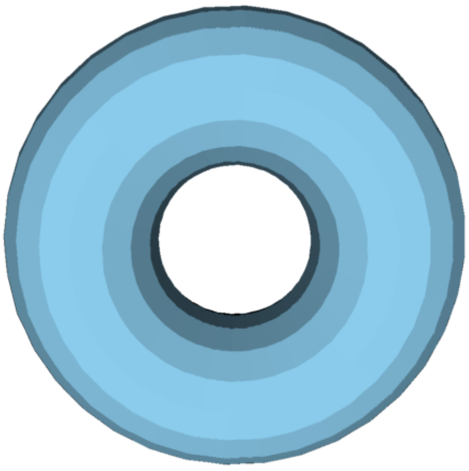
1.6 Dragon

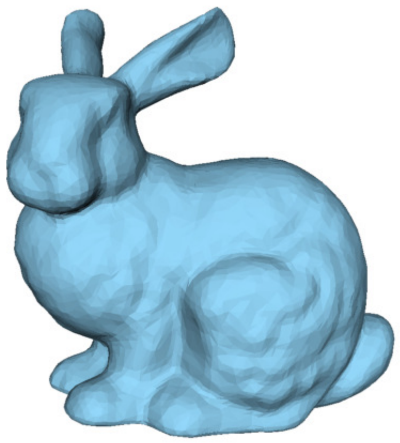
The dragon's tail creates a handle near its end. As such it has an Euler characteristic of 0 and is topologically equivalent to a torus. See also fig. 1.1e.

1.7 Buddha

The Buddha has the most handles of the objects provided. Its Euler characteristic is -10, and as such we should find six handles in total. Five of them are easily visible in fig. 1.1g while the last one should be seen from the rotated view in fig. 1.1h.


(a) Sphere


(b) Torus


(c) Bunny

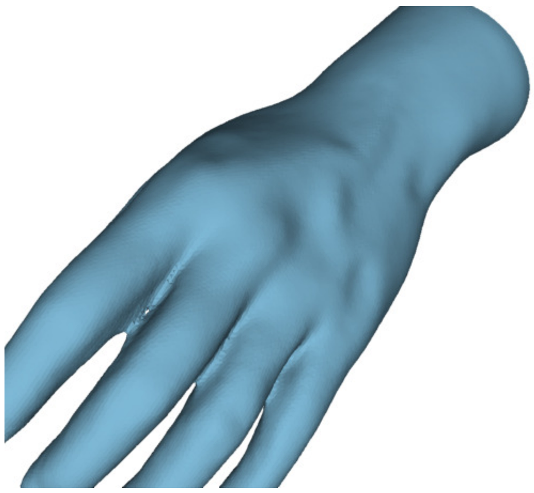
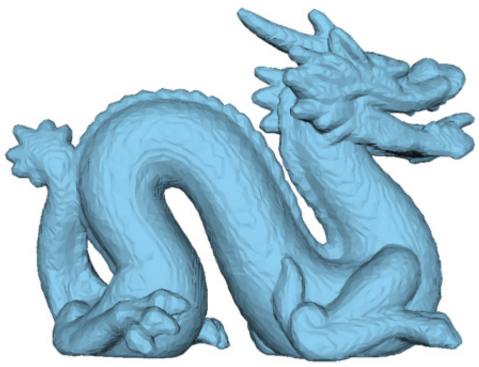
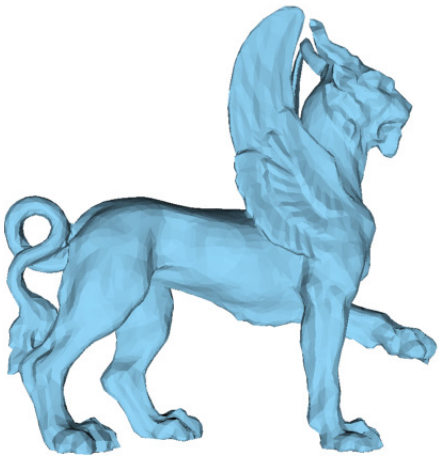
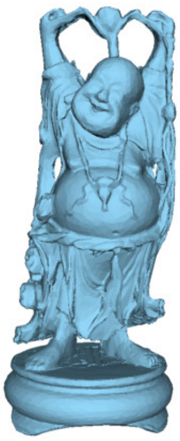

(d) Hand

Figure 1.1: Handles in Sphere, Torus, Bunny and Hand objects


(e) Dragon


(f) Feline


(g) Buddha

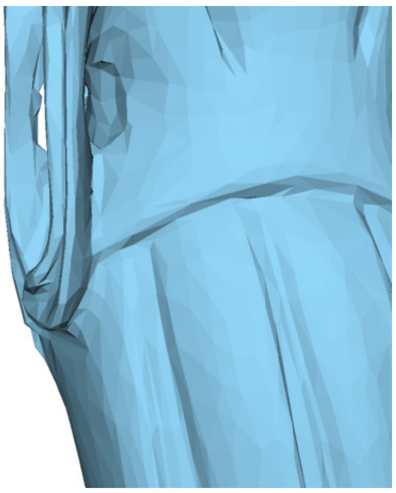

(h) Buddha

Figure 1.1: Handles in Dragon, Feline and Buddha objects

2 Mesh Coloring

2.1 Normal Mapping

This exercise was straight forward to solve. One just iterates over the polygons, or "elements" in JavaViews terminology, and assigns a color following the equation as given on the exercise sheet:

$$R = |N_x|, G = |N_y|, B = |N_z| \tag{2.1}$$

`Java.awt.Color` comes with a constructor taking three `float` arguments. Alternatively we would multiply the absolute value with 255 and round off, to get an integer RGB color component value.

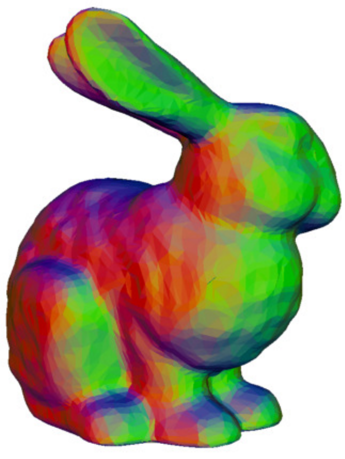
The results can be seen in fig. 2.1a, 2.2a, 2.3a and 2.4a, and the source code is located in `Ex1_2::setNormalMappingColors()`.

2.2 3D Checkerboard

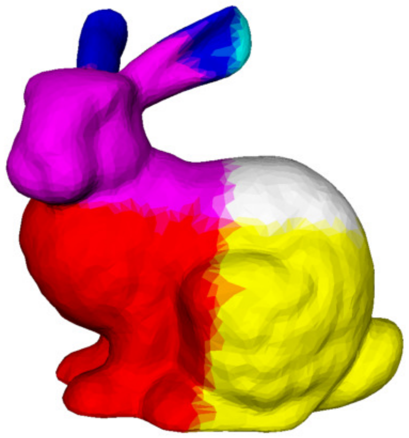
Again, the solution to this exercise was not difficult to program. Again we map the RGB values to 8-bit integer values, just set the vertex colors this time. The elements are then colored by calling $PgElementSet :: showElementFromVertexColors(true)$ on the selected geometry.

The results can be seen in fig. 2.1b, 2.2b, 2.3b and 2.4b, and the source code is located in `Ex1_2::set3DCheckerboardColors()` and `Ex1_2::f()`, while the latter combines two steps from the exercise sheet, namely the division by L followed by flooring the result and the computation denoted by $f(n)$ on the exercise sheet.

Problems with this algorithm are the jagged edges of the single checkerboard tiles. Especially for lower L this gets worse, as can be seen in e.q. fig. 2.4b. This problem gets even more prominent when applying the colorization algorithm to an Icosahedron, as seen in fig. 2.5a and 2.5b. The reason for both is the relatively low number of vertices compared to the size of the polygons. If we would remesh the objects and add more points, the checkerboard algorithm should work better again.

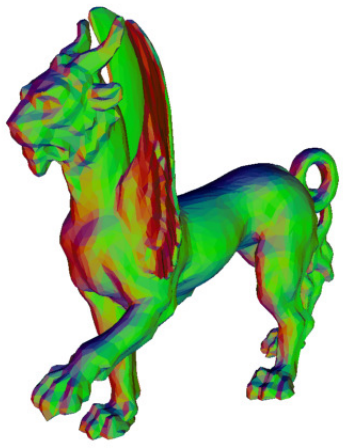


(a) normal mapping

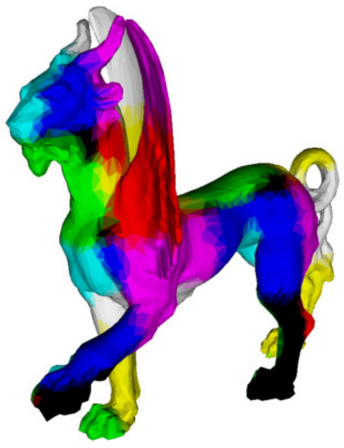


(b) checker board, $L = 1.0$

Figure 2.1: Bunny



(a) normal mapping



(b) checker board, $L = 0.5$

Figure 2.2: Feline

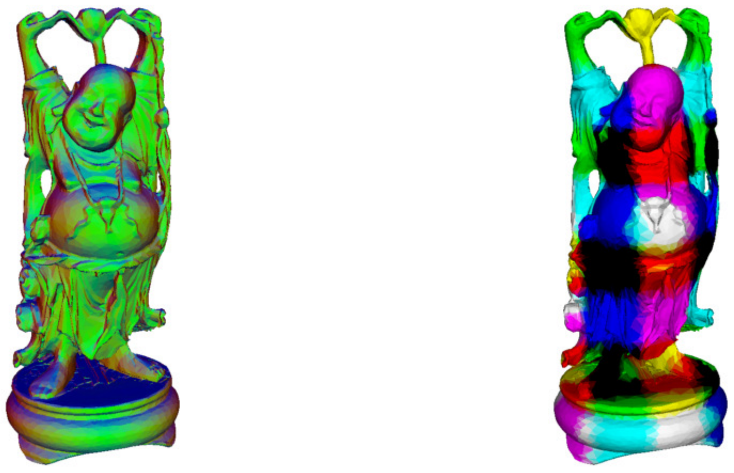

(a) normal mapping (b) checker board, $L = 0.25$

Figure 2.3: Buddha

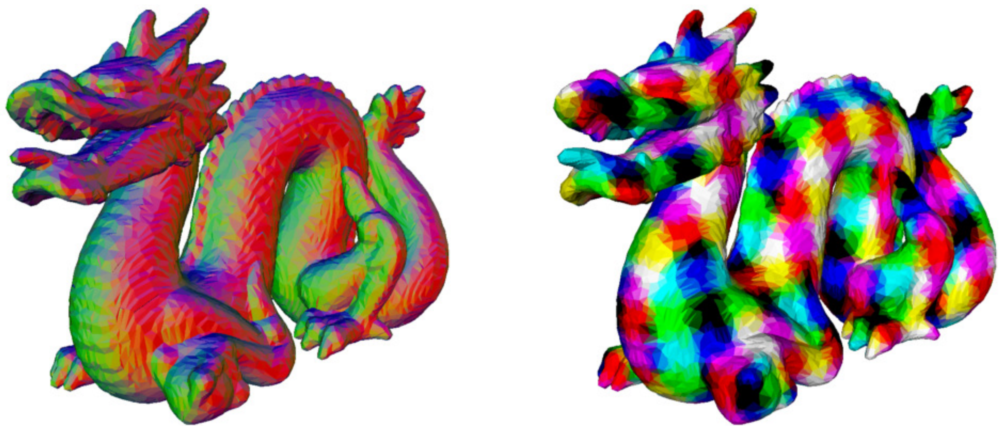
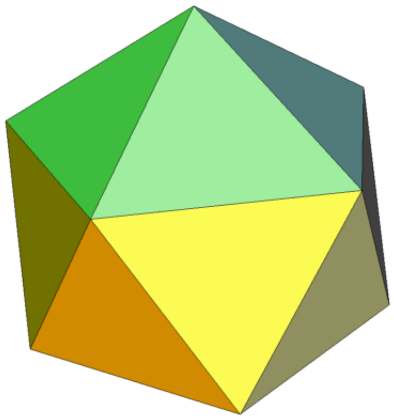
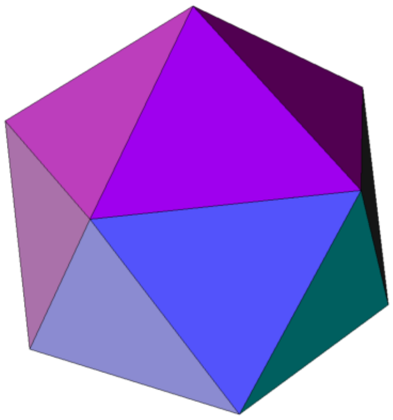

(a) normal mapping (b) checker board, $L = 0.125$

Figure 2.4: Dragon



(a) checker board, $L = 0.5$



(b) checker board, $L = 1.0$

Figure 2.5: Icosahedron

2.3 Polygon ID

In this task we are asked to come up with a bijective mapping of an integer polygon ID to some color. The task is a preparation for the following two and is not required to give visually pleasing results, but should rather be simple.

For my solution I simply interpret the polygon ID P as a base-255 number with the RGB color components as coefficients:

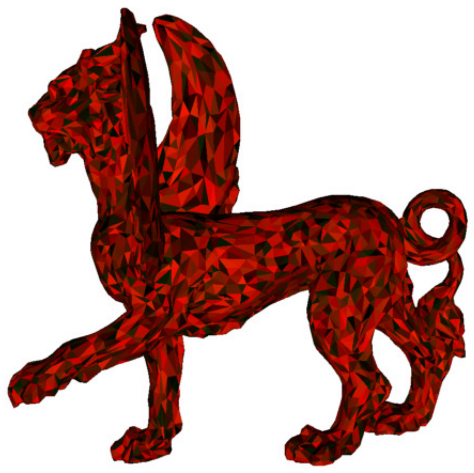
$$P = R \cdot 255^0 + G \cdot 255^1 + B \cdot 255^2 \qquad 0 \leq R, G, B \leq 255 \tag{2.2}$$

This of course will only work for meshes with less than $255 + 255^2 + 255^3 = 16646655$ polygons. Thankfully, this is the case for the objects relevant to this exercise. As an additional restriction we reserve the last color, i.e. white, to be used as the background color in the following two exercises.

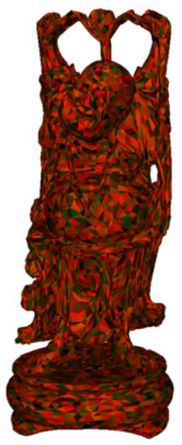
The results can be seen in fig. 2.6a, 2.6b, 2.6c and 2.6d. I was surprised to see that the polygon IDs are apparently random and do not follow any geometric route. As such, the colorization looks random as well, which can be seen as an interesting side effect for this algorithm. If one tries to create non-randomly sorted polygon IDs, he could verify whether he succeeded easily by applying this colorization scheme.

The source code for this solution is located in `Ex1_2::setPolygonIdColors()`.

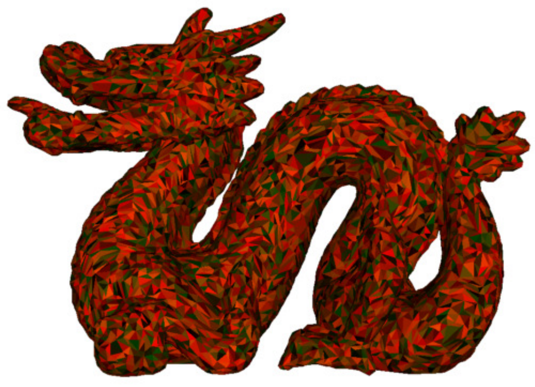
(a) Bunny



(b) Feline



(c) Buddha



(d) Dragon

Figure 2.6: Polygon ID colorization

2.4 Surface Visibility

This exercise was frustrating to say the least... Especially that we got the hints on how to use the crude JavaView API just in the last week, made me waste lots of time trying - and failing - to get something done for this exercise. I deeply hope that this experience won't be repeated in future exercises.

The algorithm is simple to write down in textual form:

1. apply Polygon ID colorization to geometry
2. enclose geometry in sphere S
3. iterate over vertices V of S
4. rotate camera to point from V to center of geometry C
5. render scene
6. find unique set of colors in rendered scene
7. map colors to polygon ID and increase counter for found IDs
8. optionally weight by dot product of polygon normal and normalized view vector
9. continue for all vertices V of S
10. normalize result

But as often when implementing a numerical algorithm, the problems lay in the dirty details, like hitting the wall that is the JavaView-API...

2.4.1 Enclosing Sphere

Just one detail of the algorithm, yet one which took me quite some time to get done. We are looking for a discretized sphere with at least 200 nodes. Furthermore it should be as unbiased as possible, meaning the vertices should be placed evenly on the sphere surface without favoring some areas, like the poles.

JavaView Sphere

The obvious first choice was to use `PgElementSet::computeSphere(15, 15, r)` which wraps a plane of 225 vertices to a sphere. This mesh is biased towards the poles though, where the node density is much larger compared to the equator, as can be seen in fig. 2.7.

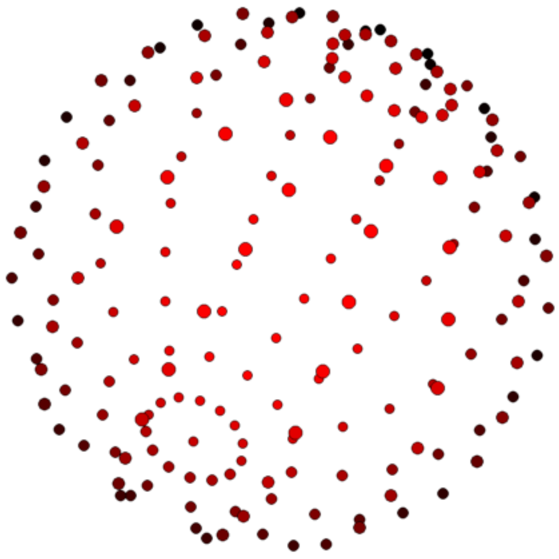


Figure 2.7: Discretized sphere by wrapping a plane of 225 vertices

Subdivided Platonic Solid Take 1

The exercise sheet suggests to take a platonic solid, e.g. an octahedron, and apply a loop subdivision scheme. Sadly, the are no details given on how to do this exactly, leaving me no choice but to try different alternatives.

As a first solution, I picked the aparrently simple route, by iteratively adding vertices in the center D of each polygon ABC. Then we subdivide the polygon into three polygons ABD, BCD and ACD. Since we are looking for a discretized sphere, we have to scale D such that its distance to the center of the sphere equals the radius of our sphere.

While kind of straight forward to implement, it turned out that this approach was not good: Since it only adds points in the middle of the polygons in each step, the long edges from the initial octahedron will persist. This leads to an uneven and hence biased distribution of the points on the sphere surface, which can be seen in fig. 2.8.

Subdivided Platonic Solid Take 2

In my second approach I refined the previous algorithm by additionally adding vertices in the middle of the edges of each polygon. Even though this sounds rather simple, JavaView once again managed to make my life hard by preventing me from iterating over the edges of an geometry... Anyhow, after some time I got it working, as can be seen in `Ex1_2::getViewPointsPlatonic()`.

Sadly, the result is also not optimal, resulting again in biased areas with higher

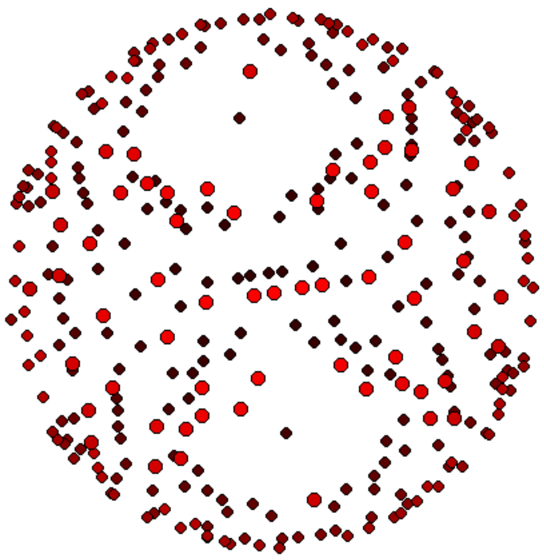


Figure 2.8: Discretized sphere by iteratively subdeviding an Octohedron, adding vertices in the center of each polygon in each step

vertex density than others, as can be seen in fig. 2.9. Another disadvantage is that the number of vertices increases exponentially with each iteration step. As such it is no wonder that we end up with a sphere of over 800 vertices, even though 200 are supposed to be enough.

Sophe Grid

Thus I resorted to another subdivision scheme I encountered during my bachelors thesis: The XSophe Grid [1]. `Ex1_2::getViewPoints()` shows how easy the implementation is. The number of nodes in a thusly created mesh can be calculated via

$$|V| = 2 + 8\sum_{n=1}^{N} n - 4N = 2 + 8 \cdot N/2 \cdot (N+1) - 4N = 2 + 4N^2. \tag{2.3}$$

This equation is easy to understand: We have two points that represent the south and north pole. The we sum the points per quarter hemisphere and multiply it by 8. This way we count the points on the equator twice though, which is why we substract them again.

In our case though we want to know N for a given $|V| = 200$:

$$N = \frac{\sqrt{|V| - 2}}{2} \tag{2.4}$$

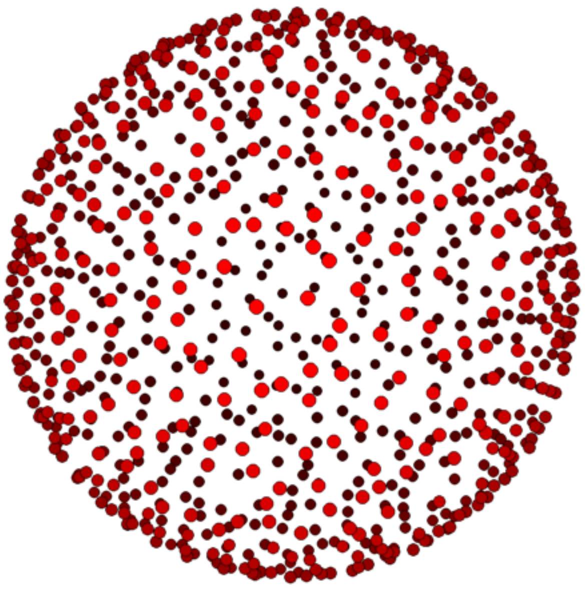


Figure 2.9: Discretized sphere by iteratively subdeviding an Octohedron, adding vertices in the center of each polygon and edge in each step

Besides being easy to implement and understant, the XSophe Grid also creates an unbiased discretization of the sphere, as can be seen in fig. 2.10. Due to these advantages, I decided to use this algorithm for the calculation of the surface visibility.

2.4.2 Finding Visible Polygons

Now we iterate over the vertices of the sphere and rotate our camera accordingly. Then we let JavaView render the image into an offscreen buffer. This part was again a huge waste of time due to the unintuitive API of JavaView and was nearly impossible until we finally got the neccessary hints in the recitation.

Once we have the image buffer though we simply have to iterate over all pixels and read the image RGB color at this point. Using eq. 2.2 we directly get the associated polygon ID. We then find the unique list of colors in the buffer and increase our counter for each of these.

For this all to work as expected though one must setup JavaView. Otherwise unexpected colors might show up in the image, confusing the surface visibility algorithm. For example one must register JavaView to prevent the “missing license” warning text. Furthermore we must only paint elements with the special `PvLightIf.MODEL_SURFACE` lightning model set. Additionally antialiasing must be disabled.

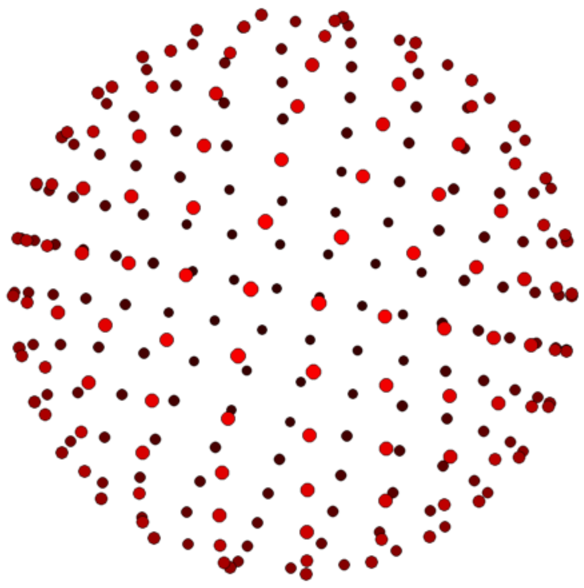


Figure 2.10: Discretized Sphere following the Sophe Grid algorithm

2.4.3 Weighting and Normalization

For the unweighted surface visibility we increase our counter by one for each encountered color in each view point step. For normalization we then devide these by the number of view points. As such the maximum surface visibility will be 1, which can only occur though in the rare case of a single always visible polygon.

If we are calculating the weighted surface visibility though, we increase the counter not by one in each step, but by the projection of the polygon's normal on the normalized view vector. The question then is how to normalize the final counter values. The exercise only requests that the final maximum value will be the same as in the unweighted surface visibility, i.e. 1. I decided to achieve this by remembering the maximum projection value for each view point, then deviding the counter values by the summed maximum projection values.

2.4.4 Results

The final results of my hours of work on the surface visibility are rather unsatisfying. Especially for the icosahedron or sphere one expects equal surface visibility for each polygon, which clearly is not the case as can be seen in figures 2.11 and 2.12.

For the larger models, like e.g. the dragon in fig. 2.13, the results look rather random. Adjacent polygons get highly different surface visibilities, which is rather bogus.

Comparing the weighted and unweighted surface visibility algorithms it is obvious that my choice of normalization in the weighted case should be improved. It turns out that it yields much lower absolute surface visibility numbers which makes comparisons harder.

Additionally I find it noteworthy that the performance of both algorithms is very bad. Especially the $O(n^2)$ algorithm for finding the colors in the image buffer must be improved, as it is repeated for every view point. If one could make JavaView paint to an image with an indexed colorspace of variable size, with a direct getter for the used colors, the performance should be much better. Sadly, I did not find any way to achieve this in Java.

The source code for this exercise can be found in the following methods:

- `Ex1_2::setUSVColors()`: set unweighted surface visibility colors on given geometry
- `Ex1_2::setWSVColors()`: set unweighted surface visibility colors on given geometry
- `Ex1_2::getSurfaceVisibility()`: get map of polygon id to surface visibility using either the weighted or unweighted algorithm
- `Ex1_2::renderOffscreen()`: render scene to offscreen buffer
- `Ex1_2::getViewPoints()`: get view points resembled as vertices in the returned geometry
- `Ex1_2::getViewPointsSphere()`: get view points using `PgElementSet.computeSphere()`.
- `Ex1_2::getViewPointsPlatonic()`: get view points using subdivision of octahedron
- `Ex1_2::getViewPointsSophe()`: get view points using Sophe Grid algorithm.

Finally I can only repeat that this exercise was very frustrating to work on, especially due to the bad JavaView API and the unsatisfying results in the end.

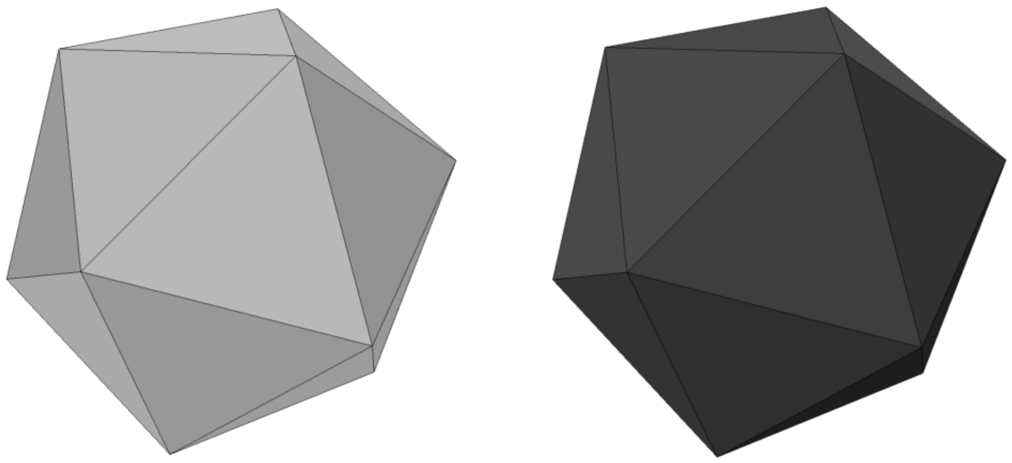

(a) Unweighted (b) Weighted

Figure 2.11: Surface Visibility Calculation for the Icosahedron

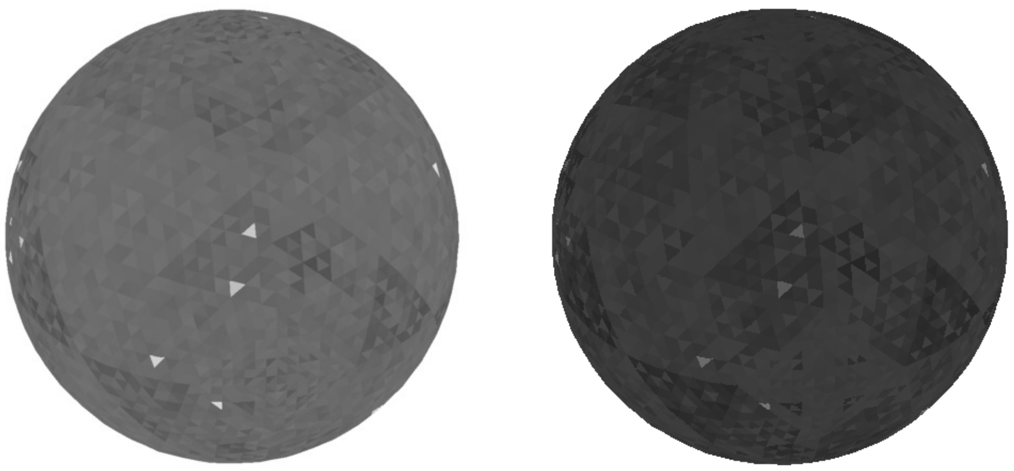
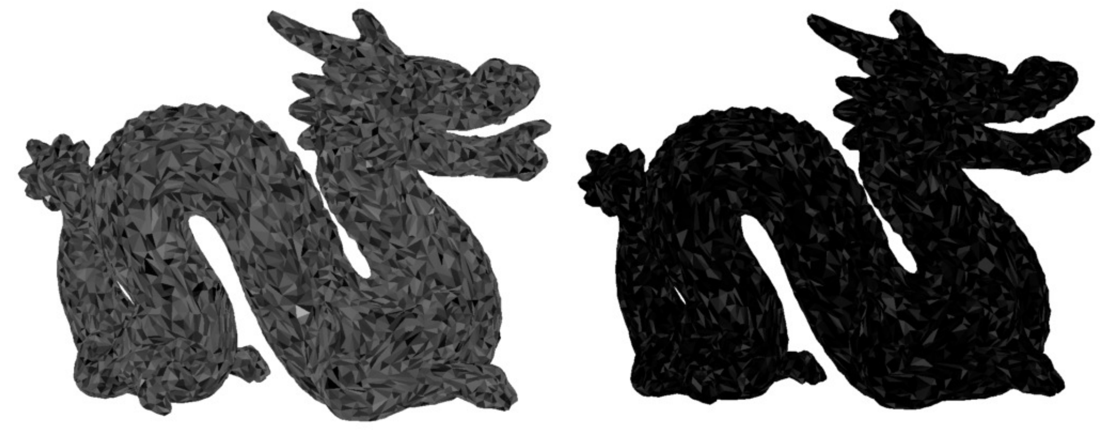

(a) Unweighted (b) Weighted

Figure 2.12: Surface Visibility Calculation for the Sphere



(a) Unweighted (b) Weighted

Figure 2.13: Surface Visibility Calculation for the Dragon

3 Bounding Boxes

This exercise was a nice change after the frustrating work on the surface visibility of exercise two. Our task was to construct bounding boxes for the given geometries using different algorithms.

3.1 General Algorithm

To construct a bounding box one generally follows the same steps. First, one finds the orientation of the bounding box, which is basically the only part that differs by the different approaches.

JavaView provides a bounding box that is axis aligned, meaning the edges of the bounding box will always be parallel to the views x-, y- and z-axes. The algorithms we should implement on the other hand all align the bounding box along the eigenvectors of a covariance matrix, with elements

$$M_{i,j} = \sum_{n=1}^{N} (i_n - \bar{i})(j_n - \bar{j}) \qquad i, j = x, y, z. \tag{3.1}$$

Once the alignment of the bounding box is found, one iterates over the vertices of the target geometry and finds the pair of largest and lowest projection on each of the three alignment axis. Using these, one can then construct the bounding box as can be seen in `Ex1_3::getBoundingBox`.

3.2 Covariance of Vertex Location

The first algorithm constructs the covariance matrix from the vertex locations. The matrix elements can be calculated using eq. 3.1, with $N = |V|$ being the number of available vertices and i_n being the i-th coordinate of the vertex n.

This approach has an advantage over the axis aligned algorithm for objects which are not already aligned with the view axes, yielding boxes with smaller volumes. This is apparent when e.g. comparing fig. 3.7a and 3.7d. For most other objects though, the difference is not so obvious.

3.3 Weighted Covariance of Vertex Location

The second approach goes a different way and takes the surface of the polygons into account. We still iterate over the vertices, but now i_n is the sum of a third of the area of all adjacent polygons. The area of a triangular polygon can be calculated using any two of the three edges $\vec{e}_{1,2,3}$

$$a = \frac{1}{2} \cdot |\vec{e}_1 \times \vec{e}_2|. \tag{3.2}$$

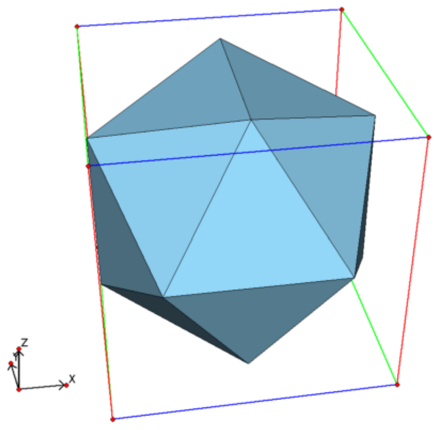
As suggested by the exercise, this algorithm results in smaller bounding boxes for objects with unevenly distributed surface vertices. This seems to be the case for the feline if one compares 3.7a and 3.7b. In most of the other cases though the difference is negleglible.

3.4 Covariance of Polygon Normals

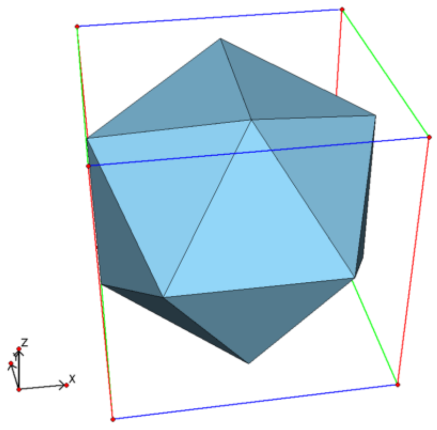
For our last approach we now build the covariance matrix using the polygon normals weighted by their area. This approach results in unexpectedly large bounding boxes, esp. when comparing e.g. 3.6b and 3.6c. Either there is an error in my implementation or this algorithm does not yield good results.

3.5 Conclusion

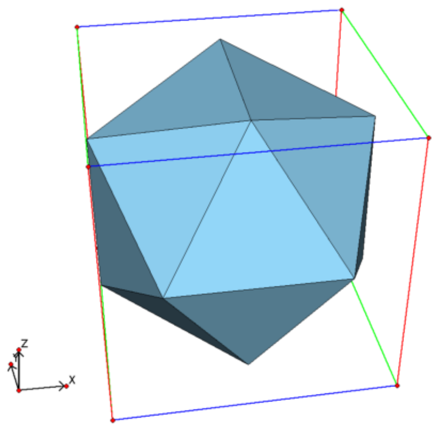
In general we see that calculating bounding boxes is inherently a simple task, yet finding the perfect bounding box looks like an extremely hard task. None of the investigated algorithms is optimal for all objects.



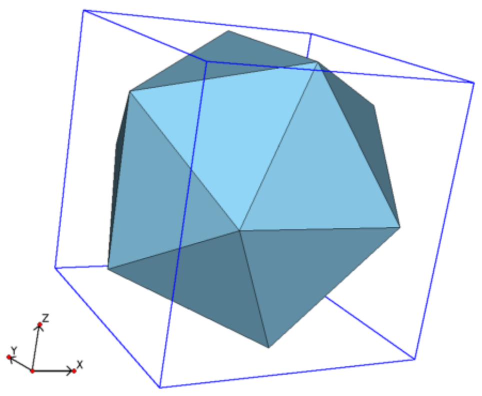
(a) Variant 1, $V \approx 4.92$



(b) Variant 2, $V \approx 4.92$

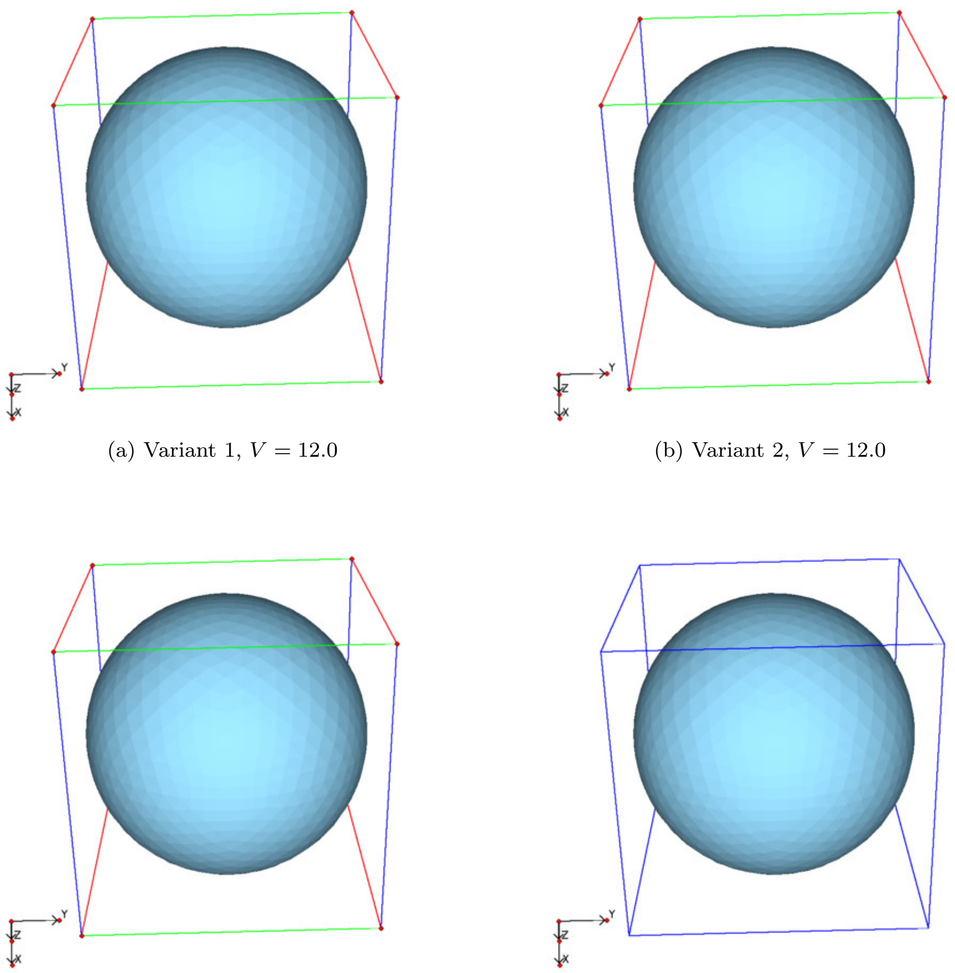


(c) Variant 3, $V \approx 4.92$



(d) JavaView

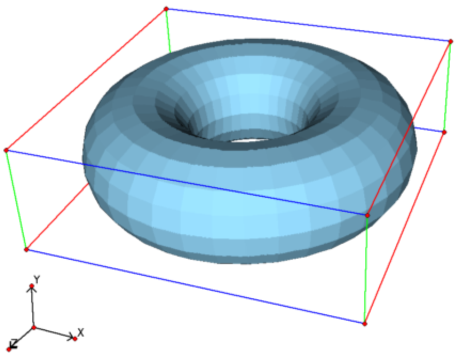
Figure 3.1: Bounding Boxes with volume V of the Icosahedron



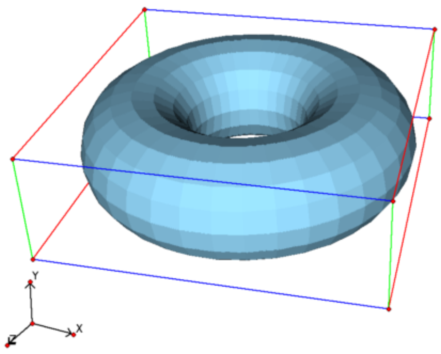
(a) Variant 1, $V = 12.0$ (b) Variant 2, $V = 12.0$

(c) Variant 3, $V = 12.0$ (d) JavaView

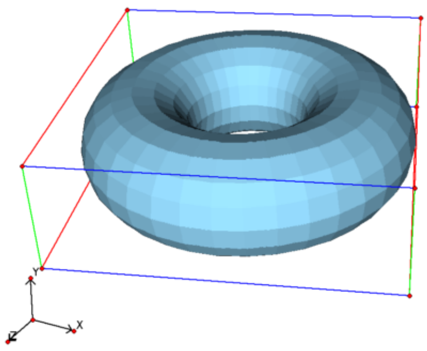
Figure 3.2: Bounding Boxes with volume V of the Sphere



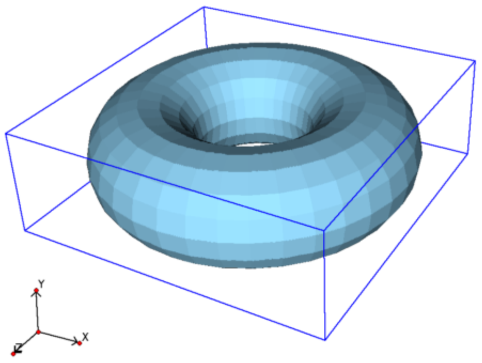
(a) Variant 1, $V = 8.0$



(b) Variant 2, $V = 8.0$

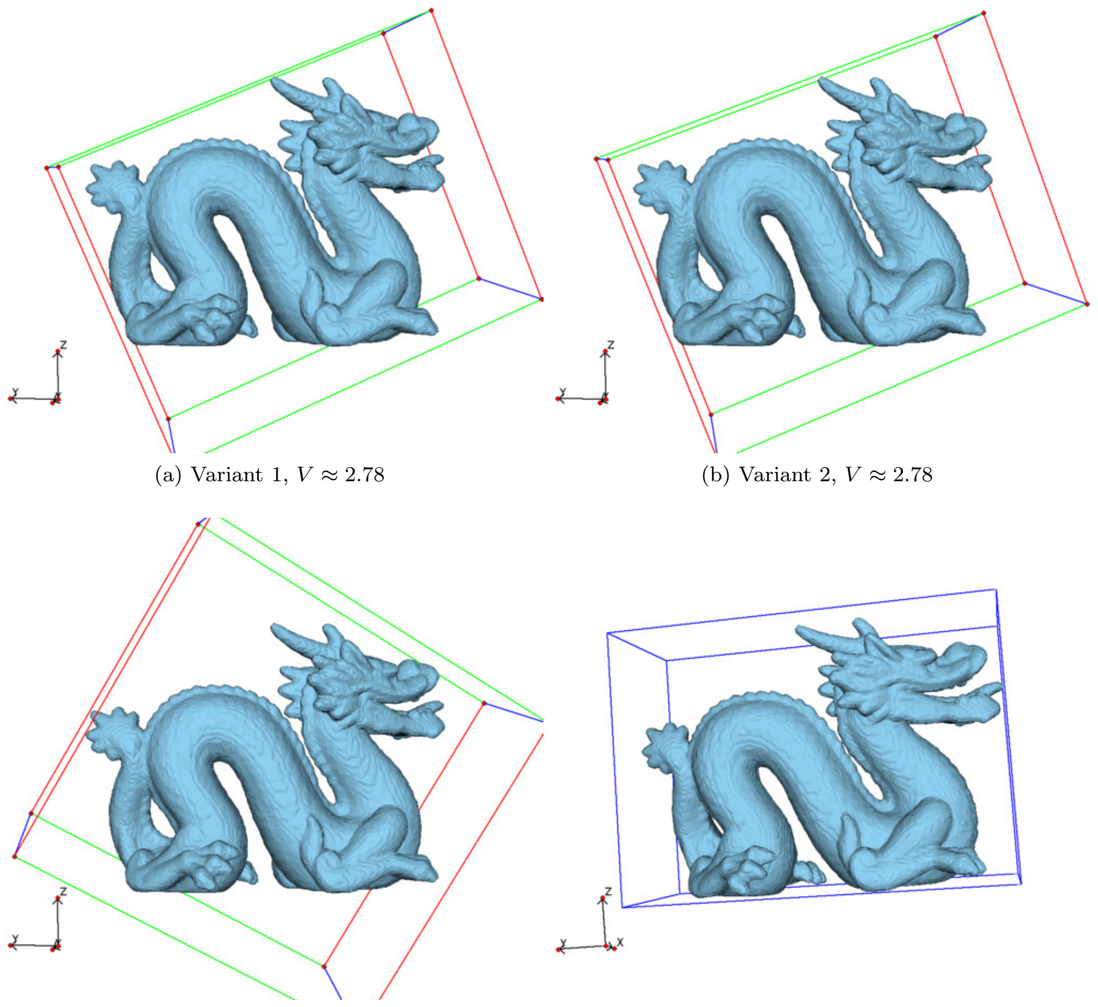


(c) Variant 3, $V = 8.0$



(d) JavaView

Figure 3.3: Bounding Boxes with volume V of the Torus



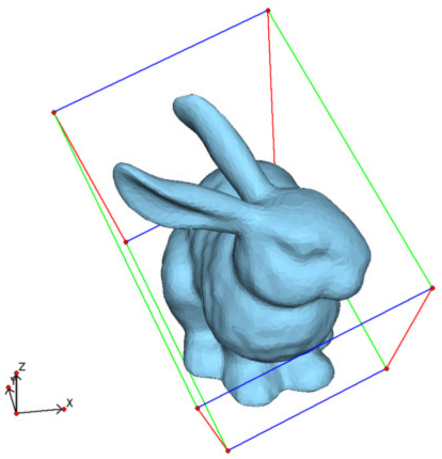
(a) Variant 1, $V \approx 2.78$

(b) Variant 2, $V \approx 2.78$

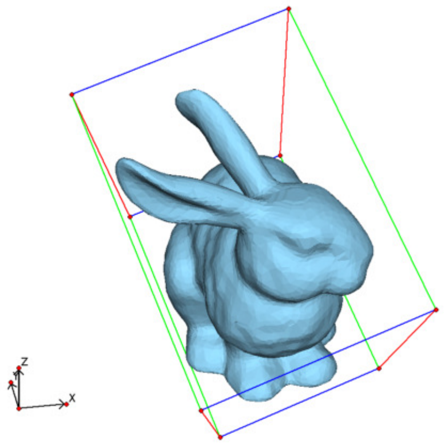
(c) Variant 3, $V \approx 3.19$

(d) JavaView

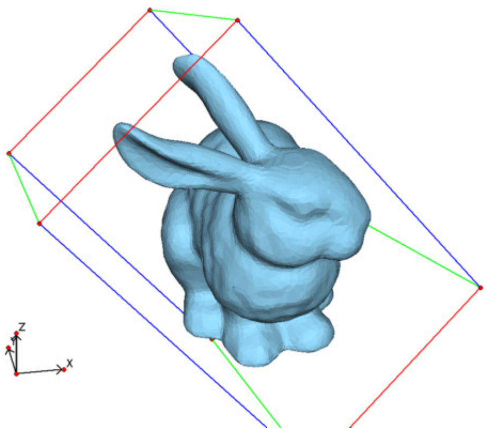
Figure 3.4: Bounding Boxes with volume V of the Dragon



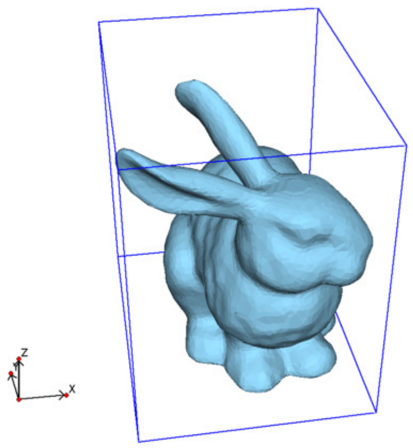
(a) Variant 1, $V \approx 6.09$



(b) Variant 2, $V \approx 6.09$

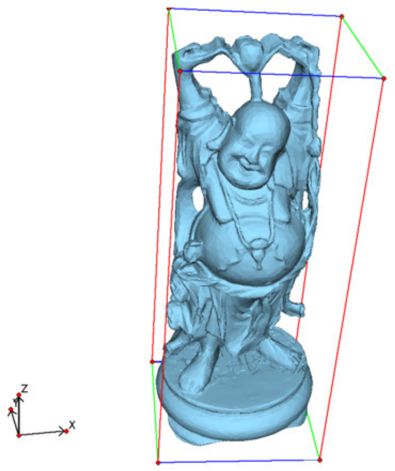


(c) Variant 3, $V \approx 7.37$

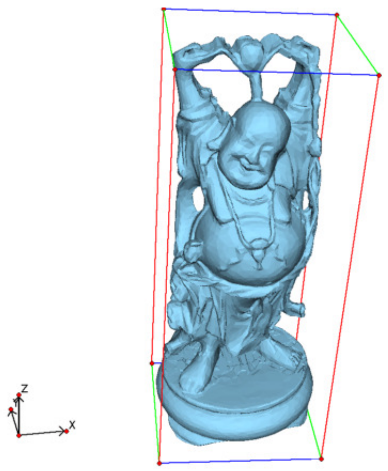


(d) JavaView

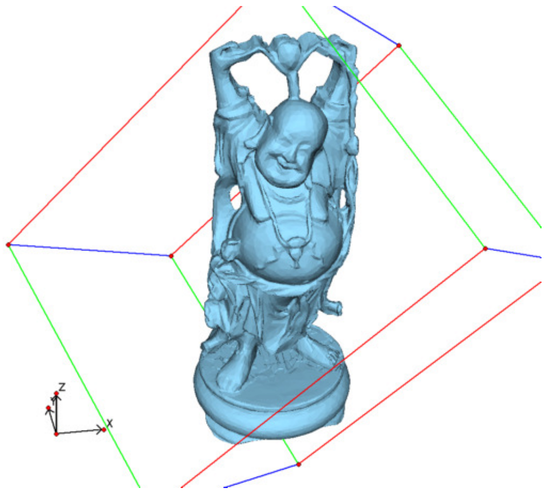
Figure 3.5: Bounding Boxes with volume V of the Bunny



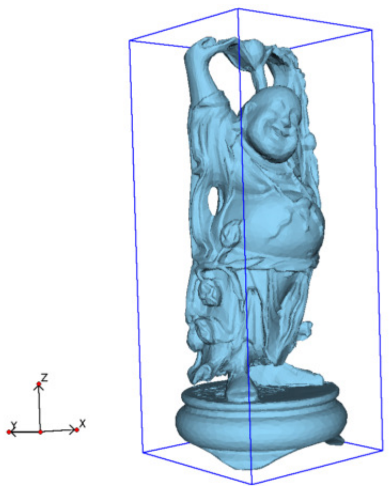
(a) Variant 1, $V \approx 1.37$



(b) Variant 2, $V \approx 1.37$

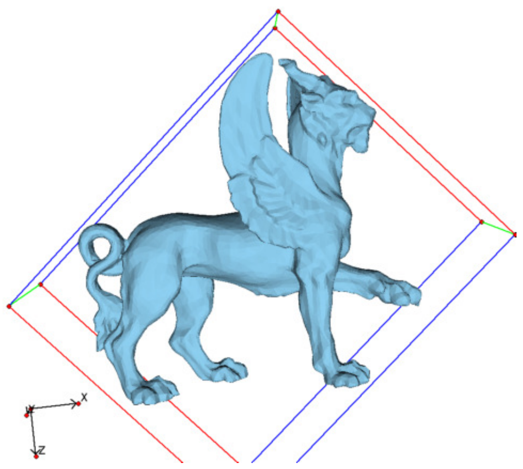


(c) Variant 3, $V \approx 3.76$

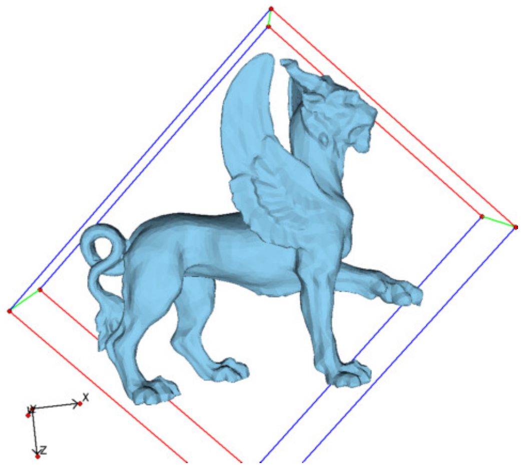


(d) JavaView

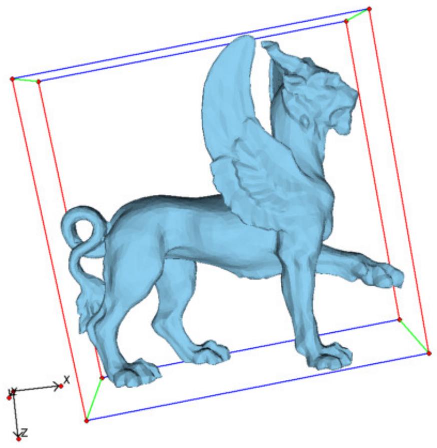
Figure 3.6: Bounding Boxes with volume V of the Buddha



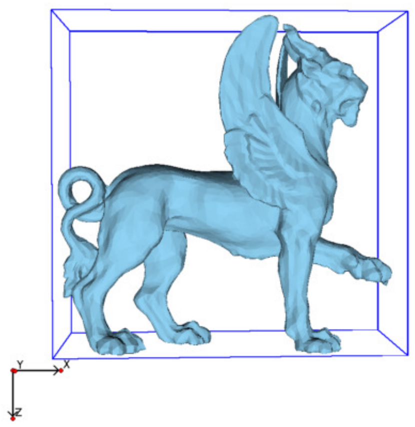
(a) Variant 1, $V \approx 3.26$



(b) Variant 2, $V \approx 3.24$

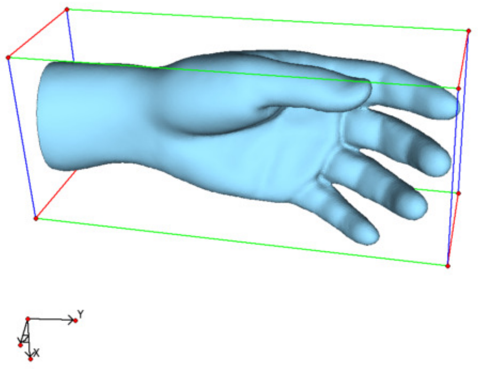


(c) Variant 3, $V \approx 3.27$

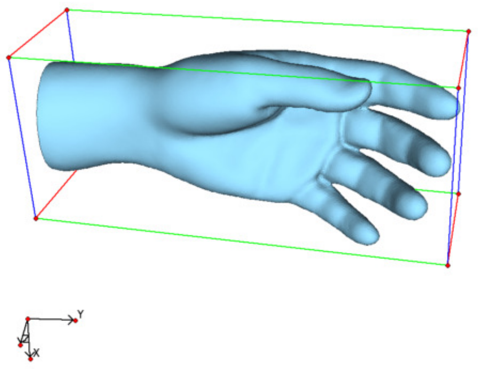


(d) JavaView

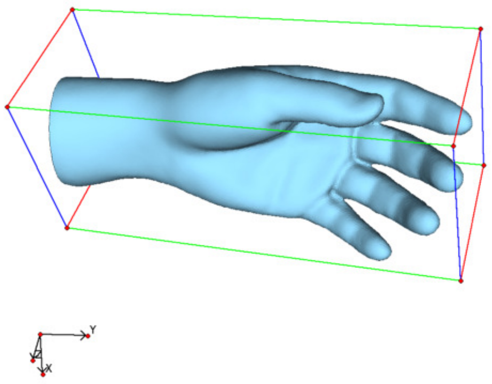
Figure 3.7: Bounding Boxes with volume V of the Feline



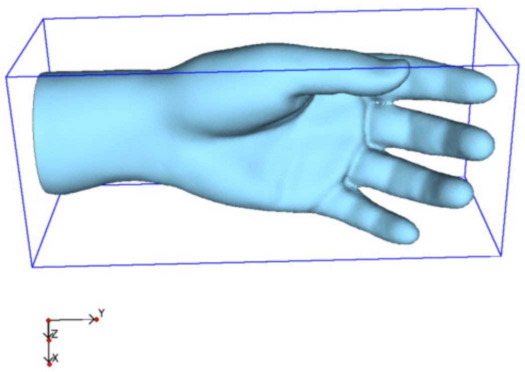
(a) Variant 1, $V \approx 118E6$



(b) Variant 2, $V \approx 118E6$



(c) Variant 3, $V \approx 136E6$



(d) JavaView

Figure 3.8: Bounding Boxes with volume V of the Hand

Bibliography

[1] Graeme R. Hanson, Kevin E. Gates, Christopher J. Noble, Mark Griffin, Anthony Mitchell, and Simon Benson. Xsophe-sophe-xeprview®. a computer simulation software suite (v. 1.1.3) for the analysis of continuous wave epr spectra. *Journal of Inorganic Biochemistry*, 98(5):903 – 916, 2004. <ce:title>Contributions from the 11th International Conference on Biological Inorganic Chemistry</ce:title>.