Report project Computer graphics
Mesh simplification

Laurens Diels & Philip Kukoba

May 3, 2021

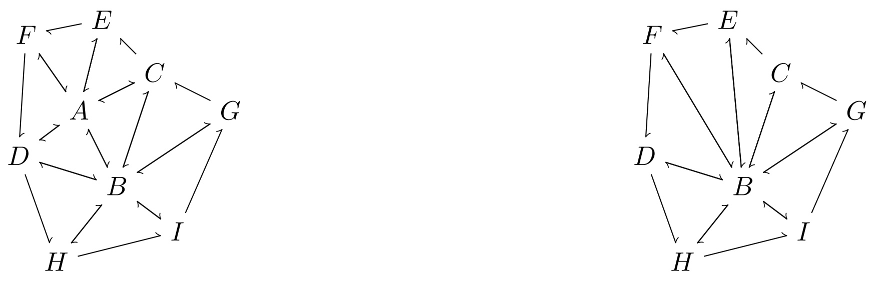
1 Brief introduction

Meshes nowadays can be extremely complicated. On the one hand, this means that models look nice, but on the other hand, we need to expend a lot of GPU power to draw them. But in certain cases such as far away objects such complexity is unnecessary and wasteful. Therefore, in this project we simplify meshes in order to save on compute resources.[1]

In sections 2 to 5 below we will only give conceptual explanations. For more concrete implementation details (such as which representation we are using, and how this affects the edge collapse) we refer to the Appendix (section 7).

2 Edge collapse

Suppose we are given a mesh. To simplify it, we can remove a (directed) edge $\overrightarrow{AB}$ (from a starting vertex A and an ending vertex B).[2] We can do this by 'pulling' A towards B, dragging the edges connected to A along with it. See Figure 1 for an illustration.



(a) Before collapsing $\overrightarrow{AB}$. *(b) After collapsing $\overrightarrow{AB}$.*

Figure 1: The result of collapsing $\overrightarrow{AB}$ in this (flattened) example mesh. The arrows on the edges are relevant only when taking the representation into account. (In the half-edge representation the edges are directed and belong to a triangle. The half-arrows in the figure indicate the direction of a half-edge (although the opposite pointing half-edge also always exists) and the side the half-tip is on indicates the triangle it lies in.)

In doing so, we have deleted one vertex (A), two triangles (ABC and ADB) and 3 bidirectional edges (AB, AC and AD). Note that the edges AE and AF have been replaced by BE and BF.

For a concrete algorithm for our mesh representation, we refer to subsection 7.2 in the Appendix.

[1] This is also known as *mesh decimation*.

[2] We will denote directed edges as $\overrightarrow{XY}$, where X and Y are vertices, and undirected edges as XY. Triangles will be denoted as XYZ (where Z is also a vertex).

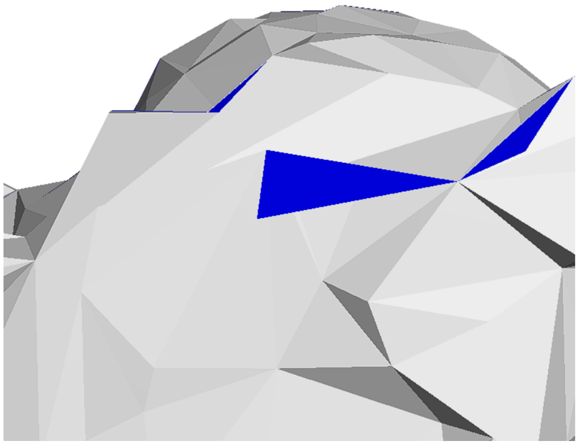


Figure 2: An example of inverted triangles after decimation.

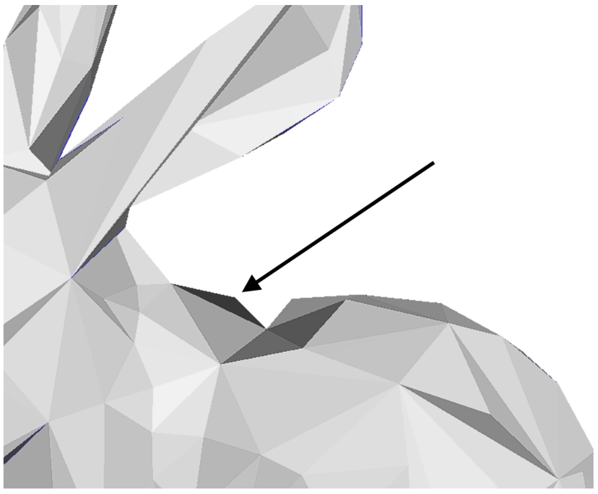


Figure 3: An example of a floating triangle after decimation. The vertex indicated by the arrow is only connected to one (two-sided) triangle.

2.1 Issues

2.1.1 Inverted triangles

It might happen that by collapsing an edge $\overrightarrow{AB}$ a non-deleted triangle previously connected to A, now connected to B, has its orientation changed considerably. This can result in its normal pointing in the opposite direction of its surrounding (correctly oriented) triangles. See Figure 2 for an example. Conceptually this can be fixed by simply negating the triangle's normal, or equivalently, by switching the order of its points around (which gives rise to the triangle's orientation), i.e. to change the order from XYZ to XZY. Unfortunately this is all directly tied to the mesh representation. For the half-edge representation we are using this operation would be problematic, messing up in particular the half-edge's opposites. Therefore we cannot easily correct triangles which after collapse are now inside-out. The best we can do is to compare the normal of an altered triangle after edge collapse with its neighbouring triangles' normals and check if they point in the same direction (i.e. the dot product is positive). If not, then we will say that the edge collapse was illegal and should not have been performed.

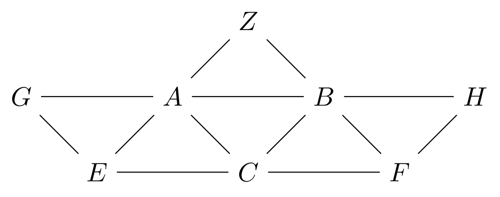
2.1.2 Floating triangles

After edge collapses we might get the degenerate situation that a part of our mesh consists of two triangles in opposite orientation stacked on top of one another. In other words, our mesh contains triangles XYZ and XZY at the same time.[3] The appearance of such a situation is sadly unavoidable in general. Indeed, consider a simple tetrahedron. After collapsing a single edge we are left exactly with two such triangles.

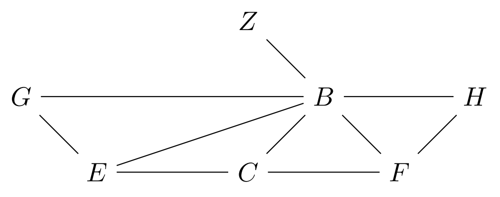
Since one point in such a triangle is detached from the 'main' mesh (and such triangles often defy gravity) we say that such triangles are *floating*. An example can be found in Figure 3. Edge collapses involving floating triangles need extra care. When collapsing an edge $\overrightarrow{AB}$ where A is the top of such a floating triangle, conceptually there are no issues, but there are some technical issues concerning the representation and our collapse algorithm. Therefore this situation is further explored in subsection 7.3 of the Appendix.

Also if A is not the point on the floating triangle 'poking out' we can still run into issues. Consider the situation where we want to collapse the base of the triangle. This situation is illustrated in Figure 4. As can be seen, our floating triangle collapses into an edge, which now no longer lies in any triangle. In this case the top (Z) and its edge (ZB) also needs to be removed. However, on a technical level this is not straightforward at all. Details can again be found in the Appendix (subsection 7.4).

[3]Like inverted triangles these will often also show up in blue in our figures, but this is a limitation of our visualisation. Theoretically both sides of the triangle(-stack) are on the outside. But since the one triangle's inside is the other's outside, drawing the insides in blue also results in blue outsides.



(a) *Before collapsing $\overrightarrow{AB}$.*



(b) *After collapsing $\overrightarrow{AB}$.*

Figure 4: The results of collapsing $\overrightarrow{AB}$ in this (flattened) example mesh with a floating triangle.

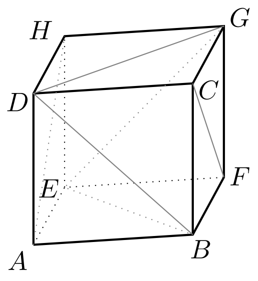
2.1.3 Objects joined by a single vertex

Consider two objects with meshes. Now join them along a single vertex A. For example, consider two cubes with precisely one point in common such as $[-1, 0]^3$ and $[0, 1]^3$. If we then collapse an edge $\overrightarrow{AX}$ we will get into trouble. However, this is not an issue with edge collapse in and of itself, but rather with our implementation for the half-edge representation. Therefore, we do not elaborate further here, but in the Appendix subsection 7.5.

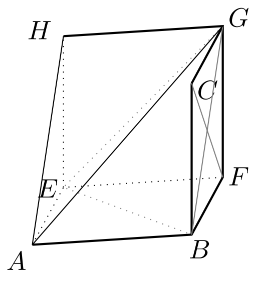
However, this situation is quite artificial and unlikely to occur in a 'real' mesh. It is nonetheless important to note that certain mesh simplification strategies can lead to this very situation. Indeed, in [GH97], on which we will later on base an edge selection approach, the authors explicitly allow collapsing vertex pairs (not just edges). The reasoning is precisely to join together disjoint regions. But as this results in the situation described above, we will restrict ourselves to selecting edges.

2.1.4 Self-intersections

Consider the cube and triangular mesh in Figure 5a. If we collapse the (directed) edge $\overrightarrow{DG}$, then the undirected edge BD will be transformed into BG. But the undirected edge CF will remain unaffected. Therefore we get two intersecting edges CF and BG.



(a) *A mesh of a cube before collapsing $\overrightarrow{DG}$.*



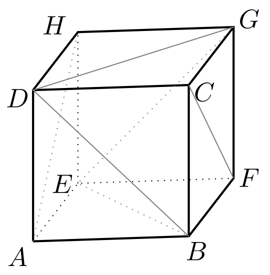
(b) *The result after collapsing $\overrightarrow{DG}$.*

Figure 5: Edge collapses can introduce self-intersections.

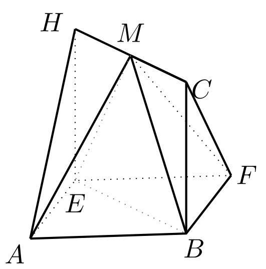
Self-intersections of edges (in contrast to triangles) are rare, however, and will only occur in very regular objects, such as the cube above. Although not a proper solution, we can avoid the issue of self-intersecting edges somewhat by collapsing an edge not to its endpoint, but to an intermediate point, as will be explained below in subsection 2.2. The result in that case is shown in Figure 6, which indeed does not have any self-intersections.

2.2 Modifications

In the Appendix we already mentioned how to deal with some exceptional cases involving floating triangles. Above we also hinted towards collapsing an edge to some other point than the endpoint. Below we continue this line of thought.



(a) Our cube mesh before collapsing $\overrightarrow{DG}$.



(b) The result after collapsing $\overrightarrow{DG}$ to its midpoint M.

Figure 6: Edge self-intersections can be alleviated by collapsing to midpoints instead of endpoints.

Instead of collapsing an edge $\overrightarrow{AB}$ to its endpoint B, we can collapse it to any other point on the line segment AB. Intuitively this means that instead of moving A to B, dragging all connections with it, we now drag both A and B to this point in between. This might sound a lot more complicated than the algorithm in subsection 7.2 for the endpoints, with also cycling around B's triangles this time. But luckily we can do this very easily: after collapsing $\overrightarrow{AB}$ to its endpoint B, just translate B to the point in between.[4]

Although talking about collapsing the edge $\overrightarrow{AB}$ implies we should collapse it to some point in between A and B, in the strategy above we can translate B to an arbitrary point in space. But if we are not careful about this point, we will very easily introduce intersecting triangles. Therefore, we do stick with taking a point in between A and B, the midpoint being the canonical option.

Apart from avoiding self-intersections to some extent, using midpoints will yield more rounded results than using endpoints. This will normally mean that the quality of the mesh degrades less. We will evaluate this later in section 6.

3 Choosing an appropriate edge to collapse

So far we have not tackled the issue of choosing which edge to collapse. A first approach would be to manually select one.

3.1 Manual edge selection

We implemented a vertex picker using ray casting. This can be activated by pressing the 'v' key on the keyboard. When two vertices are selected, and if they form an edge, this edge can be subsequently collapsed by pressing 'r'.

The idea is that we convert the 2D screen coordinates of the clicked-on point to a 3D direction vector of the line between the intended 3D point and the camera center. Details can be found in subsection 7.6. Here we also explain how to calculate the distance from a point to a line in 3D Euclidean space.

To then find the vertex in our mesh corresponding to the point on the screen we clicked on, we simply iterate over all vertices in our mesh, calculate the distance to this line and take the closest one. Note that we then also select a point, even when none is nearby. This can be fixed by using a threshold on the distance, but this depends on the scales involved and we opted to not implement this.[5] Instead, if the user is not content with the selected point, (s)he can just select another one by again pressing 'v'.

Since our meshes contain a manageable number of vertices, this simple approach is sufficiently fast. In very dense meshes it could be sped up using binary space partitioning. For example, if we build an octree or 3-D tree on the vertices, we can recursively (top-down) find the intersected cubes or boxes and ignore all vertices in non-intersected regions.

[4]Note that collapsing the edge $\overrightarrow{AB}$ to its endpoint is not as easy as translating A to B. In that case the resulting mesh would look identical to the real collapse, but it would not be any simpler, as we would not have deleted anything.

[5]However, this is not too hard: we can just use the standard deviation (i.e. the square root of the average L^2-distance to the mean vertex), or the size of the object's bounding box.

3.2 Automatic edge selection

We will not be able to significantly reduce the complexity of a mesh if we always need to select the next edge to collapse, as this would take way too much time.[6] A simple alternative approach would be to just take a random edge. But we can do better. Collapsing long edges is likely to deform the geometry considerably, whereas collapsing short ones will likely have only a small effect. But in both cases the collapse decreases the complexity of the mesh by the same amount (removing 1 vertex, 2 triangles and 6 directed edges). Therefore, it is a good idea to collapse the shortest edges in the mesh first. More generally, we can assign a collapse quality (or cost) to each edge and first collapse edges with the highest quality (minimal cost).

3.2.1 Finding the shortest edge

Collapsing the shortest edge is an easy way (if the shortest edge is known) to roughly preserve the shape of the object. A naive approach to find the shortest edge is to iterate over all edges in the mesh. This has a complexity of $\mathcal{O}(n)$ to find a single edge to collapse, where n is the number of edges in the mesh. This is very inefficient and thus an unusable approach.

A more efficient data structure can aid in getting the shortest edge. The approach used is a min heap, wherein the shortest edge is on top. The min heap can easily be constructed in C++ STL, using a priority queue. Due to the structure of the existing code, it is crucial to store pointers to Edges and not edges themselves. Because of that, a custom Comparer class is needed (only known in the Mesh class) to compare Edge pointers.

```
priority_queue<Edge*,vector<Edge*>,EdgeComparer>* edges;
```

A small trade-off with the priority queue is that it requires more work to construct it ($\mathcal{O}(\log(n))$ per added edge, yielding a total of $\mathcal{O}(n\log(n))$). However this is done when loading the mesh, and since the mesh is only loaded once, it is not a big issue. Retrieving the shortest edge is $\mathcal{O}(1)$ and removing it is $\mathcal{O}(\log(n))$, since the min heap has to be restructured.

3.2.2 Quadric error metric

Quadric error metrics is a modern approach to mesh simplification, introduced by Garland and Heckbert in [GH97]. It uses a heuristic that calculates a geometric error for each vertex. This error or cost for a vertex $\mathbf{v} = (v_x, v_y, v_z, 1)^T \in \mathbb{R}^4$ is defined by the sum of squared distances to each connected plane:

$$\Delta(\mathbf{v}) = \sum_{\mathbf{p}\in\text{planes}(\mathbf{v})} (\mathbf{p}^T\mathbf{v})^2$$

In the above formula, $\mathbf{p} = (a, b, c, d)^T \in \mathbb{R}^4$ represents the plane defined by the equation $ax + by + cz + d = 0$, where $a^2 + b^2 + c^2 = 1$, and planes($\mathbf{v}$) is the set of planes containing $\mathbf{v}$ as a vertex.[7] The error metric can be rewritten in quadratic form:

$$\begin{aligned}\Delta(\mathbf{v}) &= \sum_{\mathbf{p}\in\text{planes}(\mathbf{v})} (\mathbf{v}^\mathrm{T}\mathbf{p})(\mathbf{p}^\mathrm{T}\mathbf{v}) \\ &= \sum_{\mathbf{p}\in\text{planes}(\mathbf{v})} \mathbf{v}^\mathrm{T}(\mathbf{p}\mathbf{p}^\mathrm{T})\mathbf{v} \\ &= \mathbf{v}^\mathrm{T}\Big(\sum_{\mathbf{p}\in\text{planes}(\mathbf{v})} \mathbf{K_p}\Big)\mathbf{v}\end{aligned} \tag{1}$$

where $\mathbf{K_p}$ is the matrix

$$\mathbf{K_p} = \mathbf{p}\mathbf{p}^\mathrm{T} = \begin{pmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{pmatrix}$$

[6] In addition, selecting valid vertices (forming an edge) on a dense mesh will be quite difficult.

[7] It follows that $\Delta(\mathbf{v}) = 0$, i.e. *original* vertices in the mesh have no cost. But this will change during simplification: we will (normally) have $\Delta(\overline{\mathbf{v}}) > 0$ using the notation below.

This matrix is the fundamental error quadric $\mathbf{K_p}$ for a plane $\mathbf{p}$. It can be used to find the squared distance of any point in space to the plane $\mathbf{p}$. We can sum these fundamental quadrics together and represent an entire set of planes connected to a vertex $\mathbf{v}$ by a single matrix $Q = \sum_p K_p$, as used below in the algorithm.

We use a quadric error metric algorithm based on [GH97]:

1. Compute the Q matrices for all the initial vertices.
2. For each edge[8] compute the optimal contraction target $\overline{\mathbf{v}}$ according to

$$\overline{\mathbf{v}} = \begin{pmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \tag{2}$$

   at least if this matrix is invertible. Otherwise we let $\overline{\mathbf{v}}$ be the midpoint or endpoint of the edge. Here the q_{ij} are the entries of the Q-matrix we associate with $\overline{\mathbf{v}}$, namely the sum of the Q-matrices of the edge's vertices. Use it to calculate the cost $\Delta(\overline{\mathbf{v}})$.

   The equation for $\overline{\mathbf{v}}$ minimises this cost, see [GH97].
3. Store all edges in a min heap, ordered by the cost of the associated contraction target.
4. Iteratively remove the least cost edge from the heap, collapse it to its target $\overline{\mathbf{v}}$ and update the contraction targets and costs of all edges affected by the collapse.

The computation of the initial Q-matrices and optimal contraction targets is done when loading the mesh, by iterating over the triangles (set the Q property for every vertex) and then iterating over the edges (set the cost and $\overline{\mathbf{v}}$ properties). A priority queue is used to store all edges based on their cost.

Updating the affected edges during edge collapse is done with the helper methods of `addEdge` and `RemoveEdge`, used in the basic edge collapse algorithm.[9] These methods also update other data structures to ensure that switching between simplification methods is possible.[10]

4 Progressive meshing

In the introduction we mentioned it is wasteful to draw meshes in its full complexity when the camera is far away. As a consequence it is natural to automatically decrease the mesh quality when moving away from an object, and increasing it again when approaching. We implemented this using a very simple strategy where we distinguish 11 levels of details for the mesh (level 0 being full quality, and level 10 being maximally simplified[11]) and assigning a level of detail for an object based on the distance between the object and the camera. We also take into account the scale of the object.

More concretely, we measure the distance between the camera center and the center of the bounding box of the object, and subtract this distance for the initial camera position (when the object is in full view). Then we divide by the maximal dimension of this bounding box. For quantitatively more pleasant results we additionally scale by a constant, which determines at what normalised distances we should change level of detail.[12] The level of detail number is then simply the floor of the resulting quantity, capped between 0 (at around the initial distance or closer) and 10 (when very far away).

The mesh at level of detail $i + 1$ is generated from mesh i using a decimation to decrease the number of triangles by 10%. The result is cached by saving to disk. Thus if we want to show the mesh at detail level i, we first check if it already exists and is saved, in which case we just load it in, and if not, we

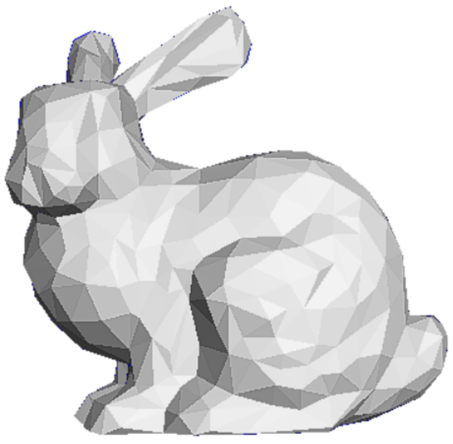
[8]In [GH97] they use all possible edge pairs. But we already explained in subsubsection 2.1.3 why we will restrict ourselves to edges.

[9]Originally we saved the connectivity information of all vertices in a 2D array, and used it to identify the affected edges after completing the basic edge collapse. However, using the integrated method using `addEdge` and `RemoveEdge` is considerably simpler and more elegant. However, we should note that previously we were able to avoid repeating the calculations for the opposite edge, which is now less straightforward.

[10]Of course in practice we would stick to only one edge selection method and only need to keep track of one data structure.

[11]Note in particular that somewhat confusingly in our terminology the highest detail level yields the least detail and vice versa.

[12]For `bunny_1k.obj` we determined the constant 0.25 by simply trying out a number of values. But similar to the situation in subsection 7.6, this constant seems to be scene dependent.



(a) Detail level 0 *(*452 *vertices,* 2700 *edges and* 900 *triangles).*

(b) Detail level 10 *(*173 *vertices,* 1026 *edges and* 342 *triangles).*

Figure 7: The `bunny_1k.obj` *model at distances corresponding to the lowest and highest detail level. Here we collapsed shortest edges to their midpoints.*

decimate a number of times from the lowest detail level which we know (i.e. exists on disk, or the current mesh in memory). Of course if we previously were at detail level i and still are now, we need not compute anything.

5 Further optimisations

We next present some optimisations we either have implemented, but were not worth mentioning before, or possible future optimisations we did not have time for.

5.1 Selecting random edges

In the supplied `bag.h` code the approach to randomly select an element was as follows. At its most fundamental level a bag is implemented using a simple array, elements of which can be set or not. To select a valid element, randomly generate an index and check if it corresponds to a valid entry. If so, return it. If not, try again.

If we have an array of size m, where only n entries are set, then the probability of selecting a valid element in this manner is n/m. The probability that after k attempts we still have not selected a valid element then is $(1 - n/m)^k$. It follows that the number of attempts k to get a valid entry with a probability of more than p (i.e. k failures with a probability of at most $1 - p$) then is

$$k = \left\lfloor \frac{\log(1-p)}{\log\left(1 - \frac{n}{m}\right)} \right\rfloor.$$

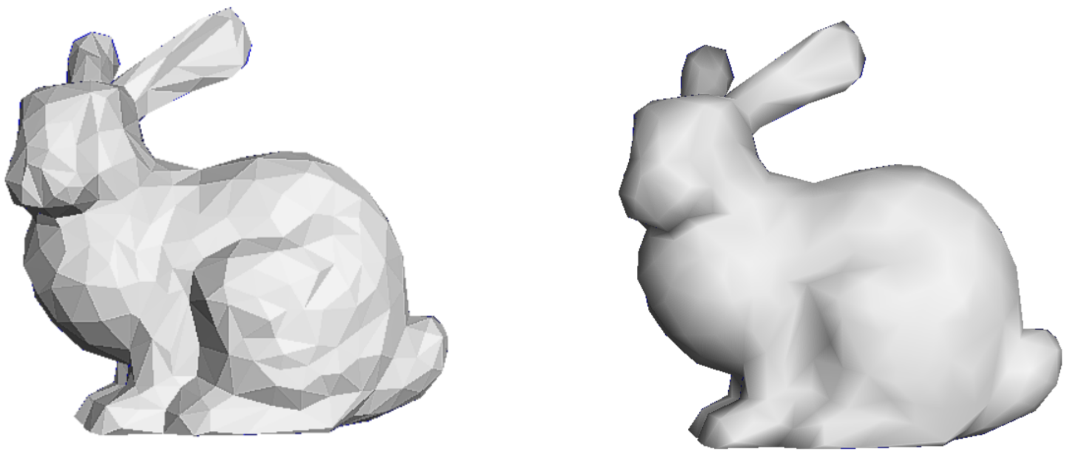
Clearly, if $m \gg n$ (i.e. our array is mostly empty) this will become very large. For example, with $m = 10001$ and $n = 36$ (the amount of half-edges in a cube) we get $k = 638$ for a probability of $p = 90\%$.

But if we are extremely unlucky, we might need an extremely long time ($k \to \infty$ for $p \to 1$). Our random edge collapse based decimation operator did in fact sometimes run noticeably slow because of this reason. We then improved the situation by selecting a random integer b between 1 and $m-1$, and by then iteratively checking the multiples $ib \bmod m$ for $i = 1, 2, \ldots, m$. In this manner we are guaranteed to find a valid entry after at most $m - n$ iterations (in the worst case scenario). Indeed, this follows from the fact that every element $b \not\equiv 0 \bmod m$ is a generator of $\mathbb{Z}/m\mathbb{Z}$. The only downside of this approach is that we are biased against checking the 0-th entry, as we will only do this at the very end, when $i = m$.

Basically, instead of repeatedly randomly taking one index out of m, we generate an entire permutation of size m. But we limit ourselves to translation permutations in the cyclic group $\mathbb{Z}/m\mathbb{Z}$ as these can be generated extremely efficiently.

5.2 Gouraud shading

We implemented Gouraud shading by computing for each vertex the mean normal vector of its surrounding triangles and simply using OpenGL's `glNormal3f` (see Figure 8). To do this efficiently, iterate over the triangles, and update for each vertex in each triangle the mean normal vector (so far) using this triangle's normal. But we recompute these normals every time we need to draw to the screen, even though after decimation most of the triangles and vertices will not have changed. This did not lead to any performance issues, but nevertheless we could improve it by storing computed normals and updating them during edge collapse itself (when changing the triangles).



(a) With flat shading. *(b) With Gouraud shading.*

Figure 8: The `bunny_1k.obj` model with flat shading and with Gouraud shading.

5.3 Progressive meshing

As previously explained we cache simplified meshes by saving them to disk. If we have a fast and large storage and a slow processor it might be beneficial to precompute the mesh at different detail levels. This way no simplification has to take place while manipulating the camera. Or if the system has sufficient amounts of memory, we could also store the simplified meshes (or certain 'key' detail levels (e.g. every other one)) in memory. But a bigger performance/memory/storage gain can be obtained by using a more fine-grained approach. In particular, as long as camera movements are relatively smooth, we should only need to care about going from detail level i to $i+1$, and conversely going from detail level $i+1$ to level i. The former can be done without needing to store anything by using our decimation implementation. For the latter our approach is currently to load in the entire mesh of level i. But (like in subsection 5.2) much of the mesh will not been altered. Therefore, it would be more efficient to be able to simply undo an edge collapse. Of course, we still need to save some local information in the neighbourhood of the collapsed edge, but this is quite limited. Going from level $i+1$ to level i can then be achieved by undoing all edge collapses we did when going from level i to level $i+1$, in reverse order.

6 Evaluation

We now have three methods for edge selection,[13] random edge, shortest edge and the best edge according to the quadric error metric, and in the case of the first two also two natural points to collapse to, the endpoint and the midpoint.[14] We gave arguments why generally random edge selection should perform

[13]By pressing the 'M' key on the keyboard, you can switch between these methods.
[14]Switching between these modes is possible by using the 'L´ key.

poor and why midpoints should be better than endpoints. Let us now put this to the test. For our evaluation we will compare the original `bunny_1k.obj` mesh to the reduced mesh after 5 decimations removing 10% of the triangles.[15] The comparison is done by varying the decimation operator and by computing the Hausdorff distance

$$d_H(M_1, M_2) = \max\left\{ \sup_{x_1\in M_1} \inf_{x_2\in M_2} \|x_1 - x_2\|_2 \,,\ \sup_{x_2\in M_2} \inf_{x_1\in M_1} \|x_1 - x_2\|_2 \right\}$$

(for Euclidian distance) where $M_1, M_2 \subseteq \mathbb{R}^3$ are the full and decimated meshes. In practice this is approximated using sampling. We will use the Metro tool[16] which will take care of the concrete implementation for us. For the case of the random edge collapse we present the mean Hausdorff distance and standard deviation of 5 runs. The other selection methods are deterministic, so one run suffices.[17] The results are shown in Table 1.

	Random edge	Shortest edge	QEM best edge
Endpoint	3.2527 ± 0.64763	2.0686	2.5283
Midpoint	2.4706 ± 0.48587	1.3121	

Table 1: Hausdorff distances between the full `bunny_1k.obj` mesh and that mesh decimated 5 *times using various decimation operators. For readability the values are scaled by a factor of* 100 *and rounded up to* 5 *significant digits. Note that the quadric error metric (QEM) approach also includes finding the best point to collapse to, so that we do not have results for Endpoint or Midpoint.*

Ignoring the quadric error metric results, this is indeed consistent with our expectations: collapsing to midpoints is better than collapsing to endpoints, and shortest edge selection is better than random edge selection. However, the quadric error metric result are unexpected: they are worse than both shortest edge results and comparable to collapsing random edges to their midpoints. This seems to disagree with the conclusions in [GH97]. A possible reason for this is that they consider all possible pair of vertices as candidates to collapse, while we restrict ourselves to edges, as previously explained. They note that one of the major strengths of their approach is then its generality, as it can also deal with multiple disconnected objects closely together (e.g. the skeleton examples in the paper). Depending on the use case our inability to deal with such situations might be a deal-breaker. But for our situation this is not an issue.

To also get a qualitative feel for these results, we present the decimated meshes in Figure 9.

7 Appendix: In more detail

In the Appendix we give addenda to the main text. These will be more technical in nature. In particular we will take into account the concrete mesh representation we use.

7.1 Mesh representation: via half-edges

In this subsection we explain how the half-edge representation for meshes works. The reader might find it helpful to keep a picture like Figure 1a in mind.

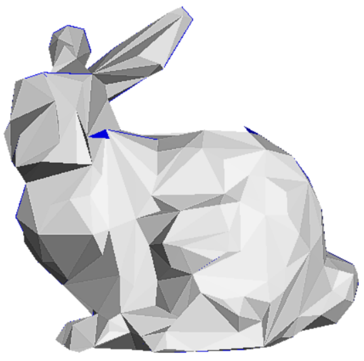
Assume we have a triangle T consisting of points A, B and C. We then represent the directed edge $\overrightarrow{AB}$ as a *half-edge* by the endpoint B and a pointer to the next (half-)edge $\overrightarrow{BC}$. In its turn this is represented by its endpoint C and a pointer to the next half-edge $\overrightarrow{CA}$, which of course will be stored as its endpoint A, together with a pointer to the half-edge $\overrightarrow{AB}$ we started from.

In order to know how half-edges are combined into triangles, we also store for each half-edge the triangle it belongs to. Note that since we are working with directed edges, the triangles also have a direction, that is, the triangle ABC built from (half-)edges $\overrightarrow{AB}$, $\overrightarrow{BC}$ and $\overrightarrow{CA}$ is different from the triangle

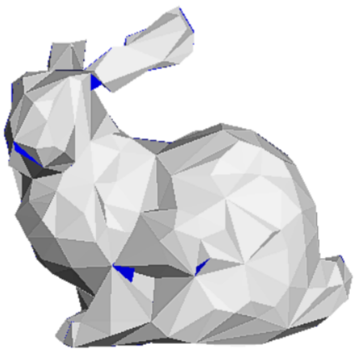
[15]Because of our choices of constants in subsection 5.3 this is equivalent to comparing the detail level 5 mesh to the full detail (level 0) mesh.

[16]`http://vcg.isti.cnr.it/vcglib/metro.html`

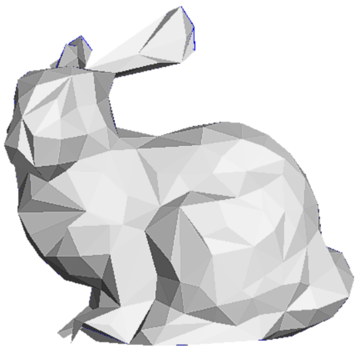
[17]Because of the sampling strategy for approximating the Hausdorff distance, the presented distances could in principle be stochastic, depending on the sampling strategy. However, in any case should have quite a low variance due to the number of samples involved (20000), hence we will ignore this. We will also limit ourselves to the standard bunny data set.



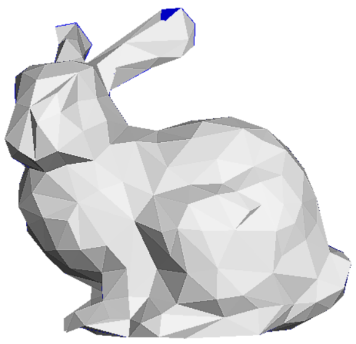
(a) Random edge collapses to their endpoints.



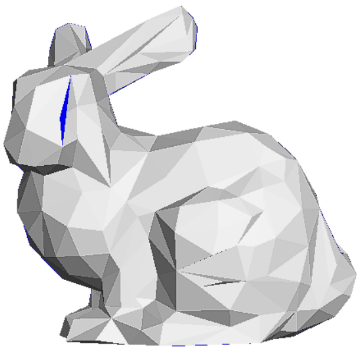
(b) Random edge collapses to their midpoints.



(c) Shortest edge collapses to their endpoints.



(d) Shortest edge collapses to their midpoints.



(e) QEM edge collapses.

Figure 9: Qualitative comparison between the different decimation strategies applied five times on the `bunny_1k.obj` *mesh.*

ACB, built from $\overrightarrow{AC}$, $\overrightarrow{CB}$ and $\overrightarrow{BA}$. This difference in direction results in a difference in orientation: the surface normal of ABC will be the opposite of the one of ACB.

Another consequence is that $\overrightarrow{AB}$ and $\overrightarrow{BA}$ belong to different triangles. But these triangles are connected, since they have the non-directed edge AB in common. Since we need such local connectivity information, we will also store for each half-edge $\overrightarrow{AB}$ a pointer to its opposite $\overrightarrow{BA}$.

To summarise, edges e are represented by their end vertex, and by pointers to the triangle T that contains it, to the edge in T following e, and to e's opposite edge.

Triangles will be represented simply by pointing to one of its edges (which also points back). Vertices are represented directly using the half-edges themselves.

7.2 Edge collapse algorithm

Recall the example edge collapse in section 2. Note that this operation collapsed vertex A to vertex B, and changed edges $\overrightarrow{AP}$ to $\overrightarrow{BP}$, and $\overrightarrow{QA}$ to $\overrightarrow{QP}$ for any vertices P and Q connected to A. Because the triangles ABC and ADB are collapsed to edges, we must remove them. Similarly the edge $\overrightarrow{AB}$ is collapsed to the vertex B, hence needs to be removed. Finally, $\overrightarrow{AD}$ and $\overrightarrow{AC}$ are transformed to $\overrightarrow{BD}$ and $\overrightarrow{BC}$ which already existed. Thus one of these instances needs to be removed. The same holds also for the opposite edges.

The achieve this algorithmically we will do the following. Let ABC be the triangle which contains the half-edge $\overrightarrow{AB}$ we are collapsing. Note that C can be found as the (end) vertex associated with the next half-edge $\overrightarrow{BC}$ of $\overrightarrow{AB}$. Further let ADB be the triangle containing its opposite half-edge $\overrightarrow{BA}$. Again D can be obtained as $\overrightarrow{BA}$'s next's endpoint. We will then remove triangles ABC and ADB and all associated edges $\overrightarrow{AB}$, $\overrightarrow{BC}$, $\overrightarrow{CA}$, $\overrightarrow{BA}$, $\overrightarrow{AD}$ and $\overrightarrow{DB}$ from the mesh. Also A will evidently be removed. But of course this corrupts the mesh as now e.g. $\overrightarrow{CA}$ does not have an associated vertex and $\overrightarrow{CB}$ does not have an opposite. To remedy this, we will cycle through all other triangles containing A. This can be achieved by starting from $\overrightarrow{AD}$ (which is $\overrightarrow{AB}$'s opposite $\overrightarrow{BA}$'s next) and iteratively applying getting the opposite's next half-edge. The iteration process ends when we again end up at $\overrightarrow{AB}$, as we have then fully looped around A.[18]

Let $\overrightarrow{AP}$ be from an intermediate step in the iteration loop, lying in a triangle APQ (e.g. $P = E$, $Q = F$ in Figure 1). When collapsing A to B, edges $\overrightarrow{AP}$, $\overrightarrow{PQ}$ and $\overrightarrow{QA}$ are transformed to $\overrightarrow{BP}$, $\overrightarrow{PQ}$ and $\overrightarrow{QB}$. Note that $\overrightarrow{PQ}$ seems to remain unaffected, but in the half-edge representation, we need to take into account its next edge and triangle, which have both changed (the opposite does remain the same). Thus we need to make three new edges. Setting the next edge is easy, as is setting the triangle. Indeed we just remove triangle APQ and replace it by a new triangle pointing to (say) the new edge $\overrightarrow{BP}$. Setting the opposites is more tricky, however. For the new $\overrightarrow{PQ}$ we can just use the old $\overrightarrow{PQ}$'s opposite $\overrightarrow{QP}$. But for $\overrightarrow{BP}$ and $\overrightarrow{QB}$ we will need to 'cross the iteration boundaries'. More precisely, to set these in iteration i, we need the edges created in iterations $i-1$ and $i+1$ (for example to set the opposite of $\overrightarrow{FB}$ in triangle BEF in Figure 1, we need $\overrightarrow{BF}$ from the previous triangle BFD). Indeed, note that the previous 'cycle' edge, i.e. the edge of which $\overrightarrow{AP}$ is the opposite's next, is $\overrightarrow{AP}$'s previous's opposite. Thus this is $\overrightarrow{QA}$'s opposite $\overrightarrow{AQ}$. This means that if in iteration i we have vertices P_i and Q_i, then we have $P_{i-1} = Q_i$ and consequently also $P_i = Q_{i+1}$. Therefore, to set the new $\overrightarrow{BP_i}$'s opposite, we need to look forward to the next iteration's $\overrightarrow{Q_{i+1}B}$, and to set the new $\overrightarrow{Q_iB}$'s opposite back to the previous iteration's $\overrightarrow{AP_{i-1}}$. In this manner we can also set all the opposites, except at the very start and end of the iteration loop.

At the start of the loop we have $Q = D$, so setting the new $\overrightarrow{QB} = \overrightarrow{DB}$'s opposite can be done by using the old (scheduled for removal) $\overrightarrow{DB}$'s opposite. And at the end we have $\overrightarrow{AP} = \overrightarrow{AB}$ (by the stopping criterion). Thus we are now in triangle ABC with $P_i = B$ and $Q_i = C = P_{i-1}$. So as we still need to set the last iteration's $\overrightarrow{BP_{i-1}} = \overrightarrow{BC}$'s opposite, this can simply be taken from the $\overrightarrow{BC}$ from the original mesh.

7.3 Issues: floating triangles with top A

Above we always assumed that $C \neq D$. But when we have a triangle which is only connected to the 'main' mesh using a single undirected edge (or no edge at all), it can happen that $C = D$. In this case A is poking out of the mesh and its triangle is also poking out. The other triangle connected to AC is the same, just flipped (as it is built using the opposites of the three edges of ABC). In fact, in the most

[18]Alternatively we could cycle using the next's next's (i.e. the previous's) opposite. In that case the roles of $\overrightarrow{AD}$ and $\overrightarrow{CA}$ are reversed.

extreme case we will simply have two flipped triangles stacked on top of one another. This is precisely the situation of floating triangles we alluded to in subsubsection 2.1.2. The problem in this case is that after the main loop in the algorithm we set the last $\overrightarrow{BP}$'s opposite. But in this case there is no iteration and therefore no previous $\overrightarrow{BP}$, causing the programme to crash. But if we do not do anything special we get the issue that $\overrightarrow{CB}$'s opposite (on the 'main' mesh) will no longer be valid. The same holds for $\overrightarrow{DB}$. But as $C = D$ we can just set $\overrightarrow{CB}$'s opposite to $\overrightarrow{DB}$'s opposite.

It is important to note that our algorithm (perhaps opposed to the intuitive idea behind edge collapse) does not yield residual $\overrightarrow{BC}$, as the original $\overrightarrow{BC}$ always gets deleted, and we also remove (and ignore) the only candidate $\overrightarrow{AC}$ to be transformed to $\overrightarrow{BC}$. The same holds for the opposite $\overrightarrow{CB}$ in the floating triangle ABC.

Note that this entire solution also works in the situation where the two flipped triangles containing A are topologically disconnected from any other parts of the mesh (i.e. we cannot move from the floating triangle to the main mesh using nexts or opposites). In that case $\overrightarrow{CB}$ and $\overrightarrow{DB}$ will simply be deleted, as they lie in this triangle-pair. In fact, then $\overrightarrow{CB} = \overrightarrow{DB}$ as half-edges, i.e. their triangles etc. are also the same.

7.4 Issues: floating triangles with A as a base point

We also revisit the situation from subsubsection 2.1.2 in case we want to collapse the base $\overrightarrow{AB}$ of a floating triangle. We distinguish two cases depending on whether this triangle is topologically disconnected from the rest of the mesh.

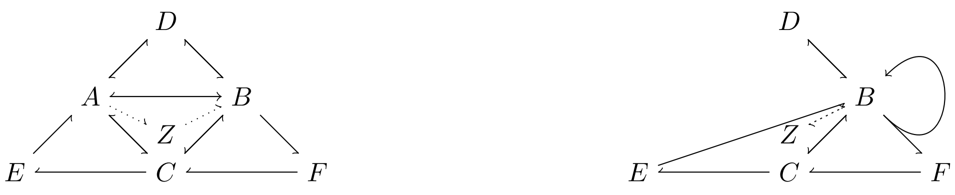
The topologically disconnected case In this case our edge $\overrightarrow{AB}$ to collapse either lies in the floating triangle ABZ to collapse, or it lies in a triangle ABC different from it on the main mesh. Both situations face the same problem, so we will only focus on the latter case. Our edge collapse will work well locally, but we will never move into the floating triangle and change e.g. $\overrightarrow{AZ}$. Thus we cannot fix it by changing it to $\overrightarrow{BZ}$ (although this would also not work, see the topologically connected case). After deleting A we then end up with a corrupted invalid edge $\overrightarrow{AZ}$. There is only one way to satisfactorily solve this problem: we need to abandon our local approach (using opposite and next). We can then search for all instances of $\overrightarrow{AB}$ in different triangles and collapse each of them locally (postponing fully deleting A). This will end up fully deleting ABZ, thanks to our fix in subsection 7.3 (as indeed A is also a top of the fully isolated triangle ABZ which just happens to touch the main mesh at AB). The bad news is that we need to explicitly find all instances of $\overrightarrow{AB}$, sacrificing performance.

Internally we use bags built on hash tables[19], which means finding an edge $\overrightarrow{AB}$ given its vertices A and B (with their indices) can be done very efficiently, since the hash of an edge is easy to compute given the indices of its vertices. It follows that $\overrightarrow{AB}$ in ABC and $\overrightarrow{AB}$ in ABZ have the same hash. Therefore they lie in the same bin. So we simply need to linearly search this bin.

The topologically connected case Consider Figure 10a (which is the half-edge analogue of Figure 4, but with omitted G and H, and added D for clarity). Now we want to collapse $\overrightarrow{AB}$ in triangle ABC. Its opposite $\overrightarrow{BA}$ lies in the floating triangle ADB (if it would lie in AZB in the main mesh we would be in the topologically disconnected case). It can then happen that the first iteration's $\overrightarrow{AP}$ ($\overrightarrow{AD}$'s opposite's next) is different from $\overrightarrow{AB}$ even though $P = B$. This might sound paradoxical, but keep in mind $\overrightarrow{AP}$ and $\overrightarrow{AB}$ are half-edges, and thus are only equal when their endpoints are the same, *and* they lie in the same triangle and have the same next and opposite edges. Note that this implies we will create a new edge $\overrightarrow{BB}$! This situation is shown in Figure 10a. There are two versions of $\overrightarrow{AB}$: the one in ABC and the one in ABZ. The algorithm removes the first, but changes the second to $\overrightarrow{BB}$. There are also two $\overrightarrow{BA}$s, as is to be expected. The one in ADB is removed, the one in AZB will also be changed to $\overrightarrow{BB}$. The result after collapsing is shown in Figure 10b. Here $\overrightarrow{BB}$ appears twice, and although not shown it is assumed that Z is not the top of a floating triangle itself, so is connected to other vertices. But D is isolated.

The way to fix this is to remove[20] $\overrightarrow{BD}$ and its opposite $\overrightarrow{DB}$, together with its 'triangle' BBD, and the same for $\overrightarrow{BB}$, its opposite $\overrightarrow{BB}$ and its 'triangle' BBB.

[19]At least until we switch to edge selection using shortest edges or minimal cost edges according to a quadric error metric.
[20]or not create in the first place



(a) Before collapsing $\overrightarrow{AB}$.

(b) After collapsing $\overrightarrow{AB}$.

Figure 10: The results of collapsing $\overrightarrow{AB}$ in this (flattened) example mesh with a floating triangle ADB.

7.5 Issues: objects joined by a single vertex

Recall the situation from subsubsection 2.1.3 where we have two object meshes which meet at a single vertex A (not two vertices which just happen to occupy the same position) and we want to collapse any edge $\overrightarrow{AX}$. Our algorithm cannot easily handle this situation. Indeed, the vertex X only belongs to one of these objects, and our cycling loop from the algorithm above will then only cycle around triangles in this object. The triangles of the other object will never be reached and therefore the corruption resulting from deleting A will not be corrected. That is, if Y is a vertex in the other object connected with A, then the edge $\overrightarrow{YA}$ is not replaced by $\overrightarrow{YX}$ but instead still references the now deleted vertex A.

7.6 Vertex selection details

When we click on the screen we get a 2D point (x, y), where x and y are non-negative integers ranging from 0 (inclusive) to the screen (window) width w and height h (exclusive), respectively. The point $(0, 0)$ is at the top left, and by increasing x we move to the right, by increasing y we move downwards. As a first step we convert such a point (x, y) to camera coordinates. Note that we have access to the camera location in world coordinates, not in view reference coordinates. In fact we also do not have a view reference point at all, which explains why we will use camera coordinates instead.

Since the camera z-direction (the look vector) is forward, and the (3D) y-direction is up, it follows that (in a right-handed coordinate system) the x-direction points to the left. Therefore, first of all we need to flip the directions of the 2D axes. After centralising so that $(0, 0)$ is at the middle of the screen, we get $(-(x - w/2), -(y - h/2))$. Note that the unit here is pixels. Let f be the camera focal length, i.e. the distance between the camera and the image plane, in world units. We also allow negative values, in which case the image plane is behind the camera (at the negative z-direction) . Further let α be the horizontal field of view. Consider the situation at $y = 0$ in camera coordinates. The triangle formed by the 3D points corresponding to $(-w/2, 0)$ (at the right side of the screen) and $(w/2, 0)$ (at the left side of the screen) and the camera center $(0, 0, 0)$ is isosceles[21] with top $(0, 0, 0)$ and top angle α. Moreover, the other points lie on the image plane $z = f$. Because of symmetry they have 3D coordinates $(\pm t, 0, f)$ for some $t \in \mathbb{R}$. See Figure 11.

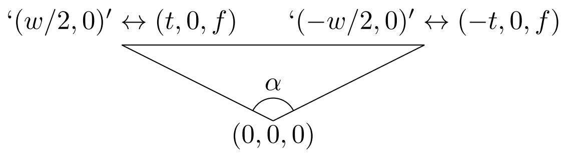


Figure 11: The field of view triangle.

By trigonometry we find $t = f \tan \frac{\alpha}{2}$. Therefore, to convert from pixels to world units (assuming square pixels) we need to multiply with $2t/w = \frac{2f}{w} \tan \frac{\alpha}{2}$. Combining all of the above, our original screen coordinates (x, y) are transformed into camera coordinates

$$\left(-\left(x - \frac{w}{2}\right)\frac{2f}{w}\tan\frac{\alpha}{2}, -\left(y - \frac{h}{2}\right)\frac{2f}{w}\tan\frac{\alpha}{2}, f\right).$$

[21] *Gelijkbenig* in Dutch

Of course a single point on the 2D screen corresponds to an entire line in 3D passing through the camera centre. By scaling the above equation by a factor of $\frac{w}{2f\tan\frac{\alpha}{2}}$ we end up with

$$\left(-\left(x-\frac{w}{2}\right),-\left(y-\frac{h}{2}\right),\frac{w}{2\tan(\alpha/2)}\right)$$

on the same 3D line. In this manner we have eliminated f. However we do need to know the horizontal field of view. Canonically this would be $\alpha = 90°$ (with $\tan(\alpha/2) = 1$), but this does not seem to be the case here. For every scene we need to recalculate the horizontal field of view.[22] In other words, calibration per scene is required.

Since the line passes through $(0, 0, 0)$ this tuple is also the (unnormalised) direction vector v_c of this line. Now we transform this from camera coordinates to world coordinates. Given the look at vector and the view up vector, we first orthogonalise the latter with respect to the former (à la Gram-Schmidt). Next we compute the horizontal vector as the cross product of this vector with the look at vector. Let $C \in \mathbb{R}^3$ consist of the camera centre in world coordinates. Then the transformation T from camera coordinates to world coordinates is done using the 4×4 matrix whose top 3×4 block's columns consist of these three direction vectors and C. The transformation matrix's fourth row is $(0, 0, 0, 1)$. To transform our direction vector v_c's camera coordinates to world coordinates v_w we use $T(v_c|0)^T = (v_w|0)^T$ (i.e. we add a zero component (row) to $v_c \in \mathbb{R}^{3\times1}$, multiply it with T from the left, and remove the last component (which is again 0) of the resulting vector). It is also useful to normalise v_w resulting in $v := \frac{v_w}{\|v_w\|_2}$.

Given the normalised direction vector v of a line L and a point $P \in \mathbb{R}^3$ on that line, the Euclidian distance from a point Q to L is given by

$$\|Q - P - ((Q - P) \cdot v)v\|_2 .$$

That is, we first substract the v-component from $Q - P$. What is left is orthogonal to v, hence its length is the orthogonal distance from Q to L. In our case we can take the base point $P \in L$ to be the camera center (in world coordinates).

References

[GH97] Michael Garland and Paul S. Heckbert, *Surface Simplification Using Quadric Error Metrics*, In Proceedings of the 24th annual conference on Computer graphics and interactive techniques (SIGGRAPH '97), pp. 209–216, 1997.

[22]By working out an example in the `cube.obj` case we found a horizontal field of view of $2\arctan\left(\frac{5}{12}\right) \approx 22.61°$. But for `torus.obj` this did not work, even with the same x, y, w, h and camera pose. Indeed, on-screen what for `cube.obj` was $x = -1$ is now at the same screen position as $x = -3$. This realisation is consistent with an increased field of view in `torus.obj`.