MESH DECIMATION

MODELING AND ANIMATION, LAB 2

Mark Eric Dieckmann   Erik Junholm   Emma Broman   Robin Skånberg

Wednesday 22nd March, 2023

Abstract

In computer graphics it is often necessary to use highly detailed models in order to achieve a convincing level of realism. However, using the full detail representation all the time is a waste of resources since lower detail can often suffice. Low resolution models can automatically be obtained from high resolution ones using a decimation algorithm. This lab will deal with one such algorithm which uses quadrics to measure the error introduced when reducing the model.

1 Introduction

Complex and highly detailed models increase the realism in computer graphics. Unfortunately, the computational cost for working with the full detail model can be very high. One obvious way of dealing with this problem is to only use the full model when necessary and use simpler versions of the model whenever possible. Of course one can proceed to model lower resolution versions by hand but a lot of time and effort can be saved if this can be done automatically.

1.1 Quadric based mesh decimation

This lab will be based on the paper *Surface Simplification Using Quadric Error Metrics* by Michael Garland and Paul S. Heckbert [2] (available on the course webpage). **Read this paper**. It contains the information about quadric based mesh decimation that you will need during the lab.

2 Decimation and half-edge mesh data structures

In [2], Garland and Heckbert present a general decimation algorithm based on quadric error metrics. The approach is general in the sense that it allows *both* edge and non-edge contractions. In this lab however, we will only handle edge contractions, since this simplifies the implementation drastically. Note that non-edge contractions can produce non-manifold surfaces, which requires special considerations in the mesh data structure. By only allowing edge contractions we maintain the topology of the surface and can readily use our half-edge data structure gained from the previous lab.

2.1 Implementing edge contractions

In order to keep the focus of the lab on the quadric metric, you will be given code in the class `DecimationMesh` for performing edge contractions (edge collapses) on the half-edge data structure. This class has a function `Initialize()` which computes all *valid* collapses for the mesh. It also has a function `Decimate()` that performs the actual collapse operation. For efficiency, it uses a heap to select the minimum cost collapse as the decimation progresses.

To complete this lab, you need to finalize a decimation algorithm by implementing the function `ComputeCollapse()` which specifies the error (or

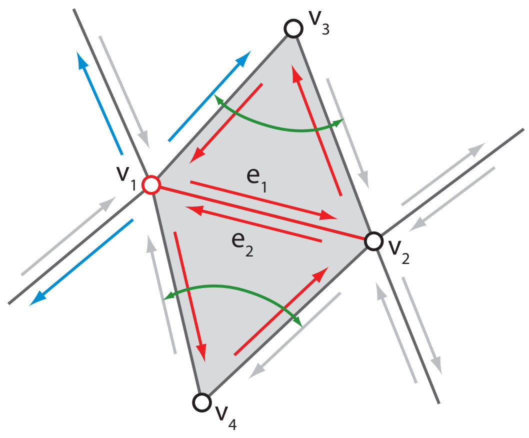

Figure 1: Edge collapse operation

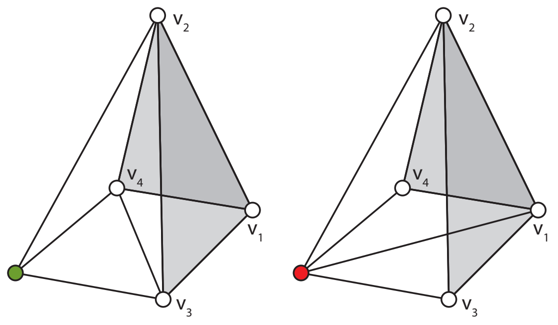

Figure 2: Valid and invalid edge collapses. The left figure shows a valid collapse along edge (v_1, v_2) whereas the right figure shows an invalid collapse.

cost) of a collapse and the resulting new vertex position. As an example, we have implemented `SimpleDecimationMesh` which uses a very simple metric. When collapsing an edge, we want to replace the vertices at the edge's endpoints with a single vertex. In the simple scheme, we simply place this vertex half-way along the edge, and compute the error of the collapse as the distance between this vertex and the two old ones. Thus, we will always collapse the *shortest* edge of the mesh, an approach which is not very useful for preserving fine appearances. Part of your task in this lab is to implement more refined cost functions and new vertex positions.

An edge collapse using the half-edge data structure consists in a number of steps where edge pointers need to be correctly updated to define the new neighborhood after the collapse. We illustrate the scenario depicted in Figure 1, where we want to collapse the edge between vertices v_1 and v_2. Literally speaking, collapsing an edge means moving its endpoints v_1 and v_2 to the exact same position. In practice, this leads to storing duplicate vertices which is not an efficient strategy. Instead we want to remove one of the vertices and connect all relevant edges to the remaining one. In our implementation, we decide to remove v_1. Note that all elements marked as red in Figure 1, are elements which need to be removed during the collapse. The algorithm proceeds as follows:

1. The edges marked as blue are connected to v_1, so we connect these to v_2.
2. There is a possibility that either of v_2, v_3 and v_4 point to one of the removed (red) edges. In such cases, we need to verify that v_2, v_3 and v_4 are redirected to valid edges.
3. Two pair pointers need to be updated, marked by green in Figure 1.
4. We move v_2 to its new position (currently half-way along the collapsed edge).
5. Collapsing edge (v_1, v_2) also removes the possibility to collapse (v_1, v_3) and (v_1, v_4) (since these are now represented as edges (v_2, v_3) and (v_2, v_4)).
6. Next, since edges connected to v_2 have changed, we update the properties of neighboring vertices and faces (such as normals) and update the cost of collapsing these edges.
7. Finally, we make sure that all edges connected to v_2 are valid for collapsing.

To expand on the last item, there might be cases which produce "double edges", meaning several edges connected between two vertices. Such cases make the mesh non-manifold. We detect these cases by assuring that edges connected to v_1 and v_2 do not share any vertices except v_3 and v_4. To see this, study Figure 2 which shows two possible triangulations of a pyramid. Consider again collapsing the edge (v_1, v_2). In the left case, the green vertex is only connected to v_2, and the collapse will produce a valid tetrahedron. For the right case however, the red vertex is connected to both v_1 and v_2 and the collapse will create a zero-volume shape with a "double edge" along v_2 and the red vertex.

Again, to keep the focus of the lab on the quadric metric, we have implemented this (rather tedious) operation for you in the class `DecimationMesh`. Study the functions and make sure you understand and correlate the code with the text in this section. To maintain all possible edge collapses, and efficiently pick the collapse with the least cost, we use a *heap* as indicated in [2]. This is reflected in the code using the `Heap` class which handles `Heapable` objects. The `EdgeCollapse` structure extends `Heapable` to store references to the actual edge to collapse (or half-edge pair) and the resulting new vertex position.

3 Assignments

3.1 Grading

The assignments marked with (3) are mandatory to complete in order to pass the lab, grade 3. You will need some of that code for the next labs. To achieve grade 4 you need to additionally complete the assignment marked with (4). For grade 5 you need to complete the assignments marked with (4,5).

3.2 Assignments

First, study the code in `DecimationMesh` and `SimpleDecimationMesh` and make sure you understand all functionality. You can load a mesh in `GUI` as a `SimpleDecimationMesh` and run `Decimate()` to try out the simple error metric. How well does it work for the cow model (`cow.obj`)?

Implement mesh decimation using quadrics (3)

We want to improve the result by using a better error metric, namely the quadric as presented by Garland and Heckbert [2]. Since the actual edge collapse operation is already implemented, you need to focus on creating and using the quadrics. The outline of the algorithm is presented in Section 4.1 in [2]. Implement step 1 and 3 in `QuadricDecimationMesh` using the functions `CreateQuadricForVert()` (which is called from `Initialize()`), `CreateQuadricForFace()` and `ComputeCollapse()`. Study `DecimationMesh` to see how these functions are called in the overall algorithm. You can use the "Collapse cost" visualization mode to color map the cost of collapsing each edge.,

For the examination you should also derive the expression for the optimal vertex position (Equation (1) in the paper). As mentioned in the paper, this is done by solving $\partial\Delta/\partial x = \partial\Delta/\partial y = \partial\Delta/\partial z = 0$, where $\Delta = \mathbf{v}^T\mathbf{Q}\mathbf{v}$. Hint: Read footnote 2 in the paper.

Implement other cost heuristics (4)

In the first assignment you implemented a decimation algorithm based on a specific error metric. This was motivated by a general assumption that the new vertices should be placed as close as possible to the original mesh in order to preserve overall shape. However, an error metric can also be motivated by prior knowledge about the application. For example, when rendering large terrain data it can be beneficial to consider screen-space coverage in the error metric. Then, objects placed far from the observer can be decimated more heavily than nearby objects. Another example can be fly-by animations when basically only the top of an object needs detail. Implement your own error metric or heuristic in `SimpleDecimationMesh` or `QuadricDecimationMesh`. You should be able to motivate or describe a situation where your metric makes sense. Remember that decimation often is used as an iterative process, so try not to make your metric dependent on absolute values.

Visualize the quadric error metric (5)

Consider the vertex $\mathbf{v}$ with the error quadric $\mathbf{Q}_v$. If $\mathbf{v}$ is moved to the new position $\mathbf{p}$ the error introduced is measured by the product $\mathbf{p}^T\mathbf{Q}_v\mathbf{p}$. Given some error tolerance ϵ all positions $\mathbf{p}$ that satisfies $\mathbf{p}^T\mathbf{Q}_v\mathbf{p} \leq \epsilon$ will be acceptable for the vertex $\mathbf{v}$.

Your task is to visualize the iso-surface $\mathbf{p}^T\mathbf{Q}_v\mathbf{p} = \epsilon$. Use the technique described in [1] chapter 4.1.2 (page 63) which can be found at address *http://mgarland.org/research/thesis.html*.

The Util.h file has a method `CholeskyFactorization` that computes the upper triangular matrix of the cholesky factorization for a 4x4 matrix. The method returns true if the factorization is successful. You do not have to visualize the error for a vertex if the factorization fails for that vertex.

Hint: The technique is based on transforming a unit sphere centered around the origin into an ellipsoid centered around the vertex of interest. Think about what transformations are needed - some are contained in the $\mathbf{Q}_v$ matrix and some needs to be handled by you.

Hint: Glu has a function `gluSphere` that can be used to draw a sphere.

4 Acknowledgements

The lab scripts and the labs were originally developed by Gunnar Läthén, Ola Nilsson, Andreas Söderström and Stefan Lindholm.

References

[1] Michael Garland. *Quadric-Based Polygonal Surface Simplification*. 1999.

[2] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.