Mesh Smoothing for Teaching GLSL Programming

Ivaylo Ilinkin[1]

[1]Gettysburg College, USA

Abstract

This paper shares ideas for effective assignment that can be used to introduce a number of advanced GLSL concepts including shader storage buffer objects, transform feedback, and compute shaders. The assignment is based on published research on mesh smoothing which serves as a motivating factor and offers a sense of accomplishment.

CCS Concepts

• ***Computing methodologies*** → ***Computer graphics; Shape modeling;***

1. Introduction

The modern graphics pipeline introduces a number of pedagogical challenges. The amount of required code for a basic application and the difficulties of conceptualizing the flow of program execution can be quite overwhelming for the students. These challenges and efforts to address them have been described in a substantial body of recent research; see for example [FWW12, RME14, AB15, PPGT14, BSP17, TRK17, ACFV18]. Effective assignments that introduce GLSL concepts in creative ways are presented in [FP18, Ili21].

The contribution of this paper is to share ideas for effective assignment that can be used to introduce a number of advanced GLSL concepts including *shader storage buffer objects*, *transform feedback*, and *compute shaders*. The assignment is based on the work of Vollmer et al. [VMM99] that describes algorithms for mesh smoothing. The algorithmic ideas are fairly straightforward to understand and implement, so the paper provides an engaging context for the GLSL concepts and does not detract from the main goals of the assignment.

2. Assignment Brief

The assignment was used in a first course on computer graphics for 3rd and 4th year students in a Bachelor's program at a liberal arts college in the US (four courses constitute a full-time load per semester and class size is 10–16 students). The course had thirteen weekly assignments (to be completed individually) and one exam.

This was a two-part assignment that was completed at the end of the semester: Tasks 1 and 2 in Assignment 12, Tasks 3 and 4 in Assignment 13 (see Sections 2.3– 2.6). It was designed to introduce advanced GLSL concepts after the students had already implemented in a previous assignment a GLSL application that renders a 3D mesh with a simple lighting model and *uniform* variables for rotation around the primary axes.

The assignment was introduced with this high level description:

The goal of this assignment is to implement the algorithms for smoothing triangle meshes presented in the following paper:

> *Improved Laplacian Smoothing of Noisy Surface Meshes.*
> *Vollmer, Mencl, and Müller. Computer Graphics Forum '99.*

The focus is on Sections 3 and 4. Skim through Sections 1 and 2.

This assignment is designed around the following topics:

- *shader storage buffer objects*
- *transform feedback*
- *compute shaders*

The introduction was followed by detailed explanations of the individual tasks as described in the rest of the paper.

2.1. Algorithms Overview

This section summarizes briefly the algorithmic aspects presented in [VMM99] that are relevant for the implementation tasks in the assignment.

Each iteration of the smoothing algorithm transforms the current vertex position, p_i', into the next vertex position, p_i'', according to one of the following choices:

$$\mathbf{p}_i'' = \frac{1}{n_i} \sum_{j \in J_i} \mathbf{p}_j' \tag{1}$$

$$\mathbf{p}_i'' = \frac{1-\alpha}{n_i} \sum_{j \in J_i} \mathbf{p}_j' + \alpha \mathbf{p}_i' \tag{2}$$

$$\mathbf{p}_i'' = \frac{1-\alpha}{n_i} \sum_{j \in J_i} \mathbf{p}_j' + \alpha \mathbf{p}_i \tag{3}$$

DOI: 10.2312/eged.20221044

where $\alpha \in [0..1]$ is a *weight factor* and n_i, p_i, J_i are respectively the *degree*, *original position*, and *adjacent indices* for vertex i.

Algorithm 1 simply updates each vertex position as the average of its neighbors, while Algorithms 2 and 3 also add the current and original position, respectively, as *pull anchor*. The pull aims to counteract a shrinkage effect that is observed when using Algorithm 1 — the mesh gradually shrinks and in the limit converges to a point (Figure 2).

2.2. Data Representation

This section gives an overview of the data representation and setup of the shader program. The mesh data is stored in a file with custom format designed to be easy to read. It is shown here only as a reference, but other standard options could be used instead:

```
vertices N               // number of vertices
...
v  i                     // vertex index
p  x  y  z               // position
n  x  y  x               // smooth normal
c  r  g  b               // color
a  n  j0 j1 ... jn-1     // indices of n adjacent
...
```

The mesh data is sent for processing via the standard setup with *vertex buffer object* (*vbo*) and *vertex array object* (*vao*). Thus, each vertex shader has access to the *original* vertex position (needed in Algorithm 3) and the adjacency information (needed in all algorithms).

Section 2.1 suggests that three buffers are needed for the *position* data: *original*, *current*, and *updated*. The original positions are communicated via *vbo+vao* while the other two are set up via *shader storage buffer objects* (*ssbo*). The first *ssbo* provides the vertex shader with random access to the *current* positions of the adjacent vertices and the second *ssbo* eliminates the need for synchronization within the shader since the *updated* positions are stored in a separate buffer.

Finally, a separate buffer of indices is needed to store the adjacency lists. Since the vertices have different number of neighbors that information cannot be communicated via the *vbo*. Instead, in our implementation all adjacency lists are stored sequentially into a read-only *adjacent ssbo*. Thus, each vertex only needs to store two indices, a_0 and a_1, that indicate where its adjacency list begins and ends within the *adjacent* buffer. These two indices are sent via the *vbo* to the vertex shader and together with the *adjacent* buffer provide access to the data.

Figure 1 provides a conceptual diagram that illustrates the above ideas.

2.3. Task 1: Smooth Normals

The first task in the assignment is designed to introduce the students to the data representation (the adjacency structure, in particular) and the general setup of *ssbo*s. At this stage the students are only asked to recompute the smooth normals at the vertices using the simple strategy of adding the flat normals of the adjacent faces. This task introduces *ssbo*s since it requires access to the *adjacency lists* and the *current* coordinates of the neighbors. The initialization step of setting up *vbo+vao* is extended with code for requesting and loading the new buffers. Separately, the new buffers are attached to the vertex shader. Finally, a function that computes the smooth normal is implemented in the vertex shader.

This intermediate task is not required for the smoothing algorithms but is included for the following reasons:

- The students can focus on a task they are familiar with and defer reading the paper.
- The key step in both the smoothing algorithms and the normal computation is the ability to work with the adjacency lists which is represented by the summation terms in Algorithms 1–3.
- It is easier to verify correctness of the implementation and gain confidence for the later tasks since the computed normals should produce a visual effect similar to the supplied smooth normals. The result is not identical since the supplied normals are computed with different algorithms.

In fact, computing the normals is slightly more complicated than the smoothing algorithms (one has to process pairs of neighbors with wraparound), but much of the code can be reused in later stages.

2.4. Task 2: Algorithm 1 and Transform Feedback

At this point the students have a good understanding of the data representation and adjacency list processing. Adapting the code from Section 2.3 to implement a function for Algorithm 1 is now straightforward.

The main focus of this task is to guide the students in setting up *transform feedback* with an associated *updated* buffer and with specification of variables to record. An *out* variable is added to the vertex shader that represents the new position of its associated vertex (p_i'' in Algorithms 1–3). As with other aspects of working with the modern pipeline, setting up *transform feedback* is a matter of following a recipe of function calls with their correct placement.

The updated vertex positions recorded by the *transform feedback* become current positions for the next rendering cycle by exchanging the roles of the *current* and *updated* buffer.

2.5. Task 3: Algorithm 2 and 3

This is a simple extension of Task 2. It involves implementation of two new functions for Algorithms 2 and 3 which are minor variations on Algorithm 1. This fairly effortless task offers the satisfaction of having three different algorithms that can be compared on a variety of meshes.

In terms of GLSL content, this task reinforces (or introduces) *uniform* variables for representing the parameter α. Instructors could also consider introducing *shader subroutines* for selecting which algorithm to run during the rendering cycles.

2.6. Task 4: Algorithm 4, Compute Shader

Algorithm 4 is discussed separately in this section. While most of the elements in Algorithm 4 will already be familiar to the students, its structure requires a different approach that in the context of GLSL invites consideration of *compute shaders*.

Here is a sketch of the main ideas:

foreach *vertex i*: `1st pass`

$$\mathbf{b}_i = \frac{1}{n_i}\sum_{j\in J_i}\mathbf{p}'_j - (\alpha\mathbf{p}_i + (1-\alpha)\mathbf{p}'_i)$$

foreach *vertex i*: `2nd pass`

$$\mathbf{p}''_i = \frac{1}{n_i}\sum_{j\in J_i}\mathbf{p}'_j - (\beta\mathbf{b}_i + \frac{1-\beta}{n_i}\sum_{j\in J_i}\mathbf{b}_j)$$

The main difference is that this is a two-pass algorithm. The first pass computes more sophisticated *pull anchors* denoted b_i based on the *current positions*. The second pass computes the *updated* positions by including the *pull anchors*.

The two-pass structure does not allow direct implementation in the vertex shader since the first pass must complete before the final positions can be computed. This offers an opportunity to introduce *compute shader* as a preprocessing stage that implements the first pass using three buffers only the last of which is new — *adjacency lists*, *current positions*, and *pull anchors*. The *pull anchors* are also attached to the vertex shader, so that after the compute shader pass concludes, their values can be used to update the vertex positions.

The appealing aspect of this task is that the implementation details for both passes in Algorithm 4 are minor variations on previous functionality. The modification of the vertex shader is fairly minimal and involves implementation of a new function for updating the vertex positions that is similar to the functions for Algorithms 1–3. This allows the students to focus on the details of setting up a compute shader that is executed as a separate shader program. In our implementation we used a simple linear arrangement for *work groups*, i.e. $(X = |V|, Y = 1, Z = 1)$, and *local size* of 1, i.e. $(x = 1, y = 1, z = 1)$. This simplifies the reasoning about the flow of computation and indexing for each shader invocation reduces to using $i = gl\_GlobalInvocationID[0]$. As an exercise instructors can vary *work groups* and *local size* to build intuition about their relationship and prepare the students for applications of compute shaders in other domains.

2.7. Discussion

Figure 2 shows representative images from a student submission for smoothing a mesh with three of the algorithms.

Algorithm 1 leads to substantial smoothing, but it suffers considerably from shrinkage effect. Algorithm 2 produces similar result.

Algorithm 3 uses the *original* positions as pull anchors which counteracts the shrinking effect at the expense of smoothing quality. This is controlled by the parameter α: $\alpha = 0$ degenerates Algorithm 3 into Algorithm 1; $\alpha = 1$ fixes the output to the original mesh.

Algorithm 4 uses more sophisticated pull anchors to balance the tradeoff between shrinkage and smoothing quality. As can be seen in Figure 2 Algorithm 3 is not free from shrinkage.

Algorithms 3 and 4 can be shown to converge. For the example in Figure 2 and choice of parameters the convergence is fairly quick. The progression of the algorithms can be appreciated better in the accompanying video.

3. Conclusions

This paper presented ideas for effective assignment based on the work of Vollmer et al. [VMM99] that can be used to introduce a number of GLSL concepts, including *shader storage buffer objects*, *transform feedback*, and *compute shaders*. The appealing aspect of [VMM99] that makes it possible to serve as the basis for a course assignment is that the algorithmic ideas are fairly straightforward to understand and implement. This enables the students to focus on the complexities of GLSL concepts and thus strengthens their understanding. The fact that the assignment is based on published research further serves as a motivating factor and provides a sense of accomplishment.

4. Acknowledgements

The images and accompanying video are based on the assignment submission of Doug Harsha.

References

[AB15] ACKERMANN P., BACH T.: Redesign of an Introductory Computer Graphics Course. In *EG 2015 - Education Papers* (2015), Bronstein M., Teschner M., (Eds.), The Eurographics Association. doi:10.2312/eged.20151021. 1

[ACFV18] ANDUJAR C., CHICA A., FAIRÉN M., VINACUA A.: GL-Socket: A CG Plugin-based Framework for Teaching and Assessment. In *EG 2018 - Education Papers* (2018), Post F., Žára J., (Eds.), The Eurographics Association. doi:10.2312/eged.20181003. 1

[BSP17] BÜRGISSER B., STEINER D., PAJAROLA R.: bRenderer: A Flexible Basis for a Modern Computer Graphics Curriculum. In *EG 2017 - Education Papers* (2017), Bourdin J.-J., Shesh A., (Eds.), The Eurographics Association. doi:10.2312/eged.20171023. 1

[FP18] FOURQUET E., PENTECOST L.: A Creative First Assignment in the Modern Graphics Pipeline. In *EG 2018 - Education Papers* (2018), Post F., Žára J., (Eds.), The Eurographics Association. doi:10.2312/eged.20181006. 1

[FWW12] FINK H., WEBER T., WIMMER M.: Teaching a Modern Graphics Pipeline Using a Shader-based Software Renderer. In *Eurographics 2012 - Education Papers* (2012), Gallo G., Santos B. S., (Eds.), The Eurographics Association. doi:10.2312/conf/EG2012/education/073-080. 1

[Ili21] ILINKIN I.: Marching Cubes for Teaching GLSL Programming. In *Eurographics 2021 - Education Papers* (2021), Sousa Santos B., Domik G., (Eds.), The Eurographics Association. doi:10.2312/eged.20211008. 1

[PPGT14] PAPAGIANNAKIS G., PAPANIKOLAOU P., GREASSIDOU E., TRAHANIAS P.: glGA: an OpenGL Geometric Application Framework for a Modern, Shader-based Computer Graphics Curriculum. In *Eurographics 2014 - Education Papers* (2014), Bourdin J.-J., Jorge J., Anderson E., (Eds.), The Eurographics Association. doi:10.2312/eged.20141026. 1

[RME14] REINA G., MÜLLER T., ERTL T.: Incorporating modern opengl into computer graphics education. *IEEE Computer Graphics and Applications 34*, 4 (2014), 16–21. doi:10.1109/MCG.2014.69. 1

[TRK17] TOISOUL A., RUECKERT D., KAINZ B.: Accessible GLSL Shader Programming. In *EG 2017 - Education Papers* (2017), Bourdin J.-J., Shesh A., (Eds.), The Eurographics Association. doi:10.2312/eged.20171024. 1

[VMM99] VOLLMER J., MENCL R., MÜLLER H.: Improved Laplacian Smoothing of Noisy Surface Meshes. *Computer Graphics Forum* (1999). doi:10.1111/1467-8659.00334. 1, 3

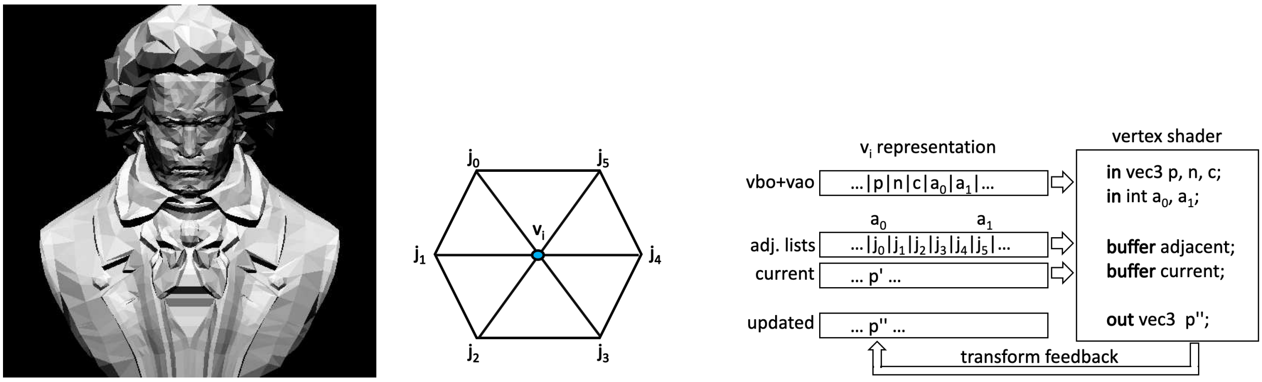

Figure 1: *Left: Original Mesh. Right: Vertex representation and a sketch of vertex shader setup with the various buffers.*

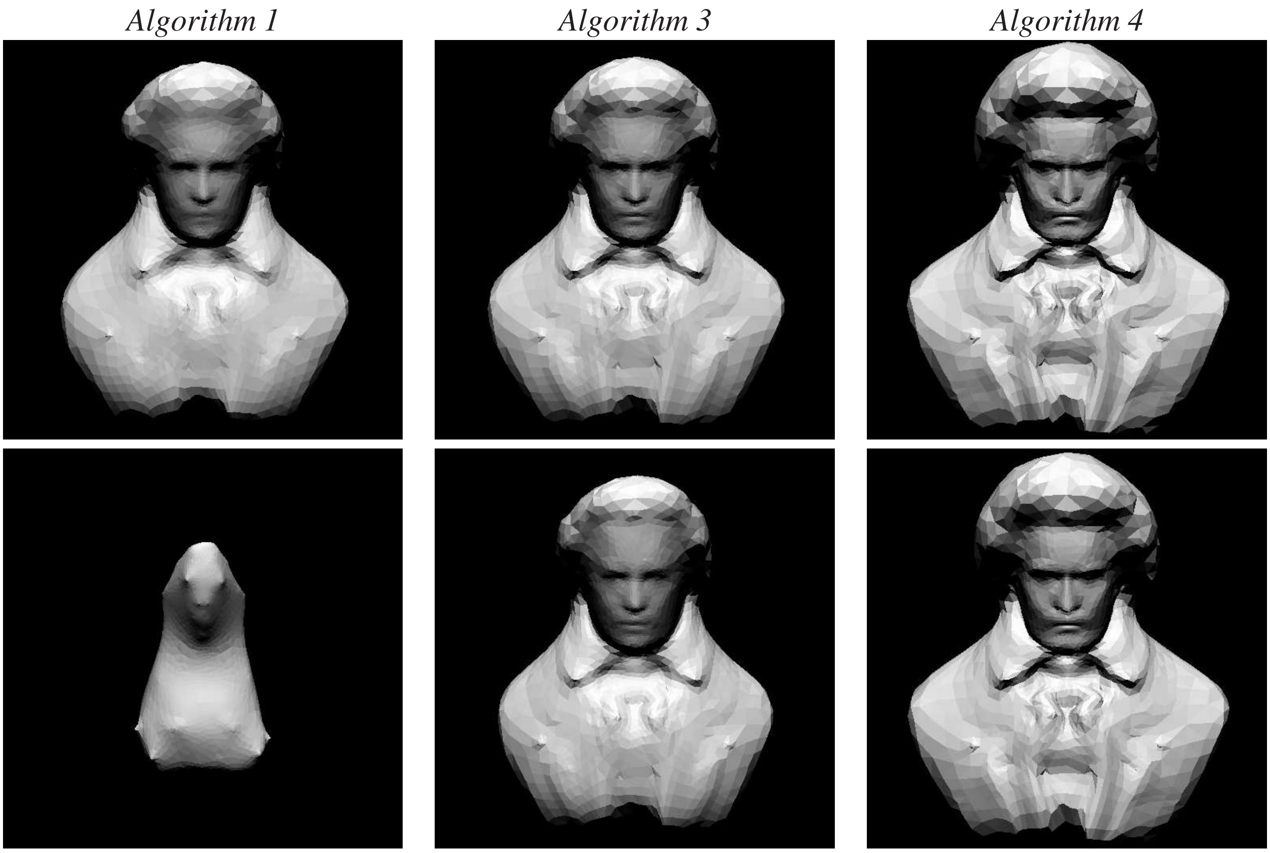

Figure 2: *Representative images based on a student submission for $\alpha = 0.1$ and $\beta = 0.2$ for the mesh shown in Figure 1. The top row shows 5 iterations of each algorithm, while the bottom row shows 100, 30, and 10 iterations, respectively (note that Algorithms 3 and 4 have achieved convergence at this level). Algorithm 2 is not shown since it produces similar effect as Algorithm 1.*