Marching Cubes for Teaching GLSL Programming

Ivaylo Ilinkin[1]

[1]Gettysburg College, USA

Abstract
This paper shares ideas for illustrating GLSL programming based on the classic Marching Cubes *algorithm. The algorithm has a number of appealing aspects: it is feasible to implement as one of the components in a computer graphics course, it motivates naturally a number of GLSL concepts and constructs, and leaves the students with a sense of accomplishment having reproduced original research. The paper suggest possible variations and extensions that could form the basis for final group projects.*

CCS Concepts
• ***Computing methodologies*** $\rightarrow$ ***Computer graphics; Shape modeling;***

1. Introduction

Teaching computer graphics has always faced a challenge in terms of finding the right scaffolding that lets the students discover and experiment with the main concepts while abstracting away details about GUI frameworks and rendering. Unfortunately, the flexibility offered by the modern shader-based pipeline introduces further layers of complexity that require new teaching approaches, assignments, and tools and frameworks. General discussion about transitioning to the modern approach can be found in [FWW12, RME14, AB15]; frameworks that aim to provide the right scaffolding are described in [PPGT14, BSP17, TRK17, ACFV18]; creative assignment for introducing the modern pipeline is presented in [FP18].

The goal of this paper is to share ideas for using the *Marching Cubes (MC)* [LC87] algorithm to illustrate shader programming. The appeal of MC is that it is feasible to implement within the time frame of a regular assignment and offers an opportunity to introduce *geometry shaders*. In fact, our use of MC was motivated by the need to find a non-trivial example that could illustrate the utility of *geometry shaders* in a realistic and natural context. An added benefit to this approach is that it exposes students to published research that is accessible at the undergraduate level—Section 4 in the original paper [LC87] presents clearly the main ideas and can serve on its own as the foundation for basic implementation. The algorithm offers a number of possibilities for variations or extensions that instructors could consider for final group projects.

In the rest of the paper we provide context for the proposed ideas, describe the setup and implementation details, and conclude with a discussion of possible variations and extensions.

2. Context and Objectives

Our computer graphics course is an upper-level elective for computer science majors at a liberal arts college. It is typically taken by juniors and seniors (3rd and 4th year students) and the formal prerequisites are three core courses: *Introduction to Programming*, *Object-Oriented Programming*, and *Data Structures and Algorithms*. The language of instruction in the core courses is Java, while the computer graphics course is taught in C++ to offer exposure to a new language. There is no formal linear algebra and advanced calculus requirement—both C++ and the required mathematical background are introduced as needed during the course.

Previous iterations of the course were based on the OpenGL fixed-function pipeline while the latest version also included the modern shader-based approach replacing raster graphics algorithms for 2D primitives (scan converting lines and circles, filling and clipping polygons, etc.). The *Marching Cubes* algorithm was used at a point when the students had already learned the concepts of *vertex shader* and *fragment shader*, *uniform variables*, and how to prepare geometry and attribute buffers for the shader pipeline. The students were also familiar with 2D and 3D geometric transformations and projections, although in order to reduce complexity these were not used in the MC implementation. The data was set up for an implicit orthographic projection and the only transformation applied to the data was a simple rotation around the y-axis (one could, however, appreciate the results without this rotation). This already hints at a possible variation—instructors could progressively add different projections and transformations/interactions; we did not see the need, since these were already covered in a different context.

As mentioned earlier the motivation for MC was to find a natural and realistic context to introduce *geometry shaders*. This also

DOI: 10.2312/eged.20211008

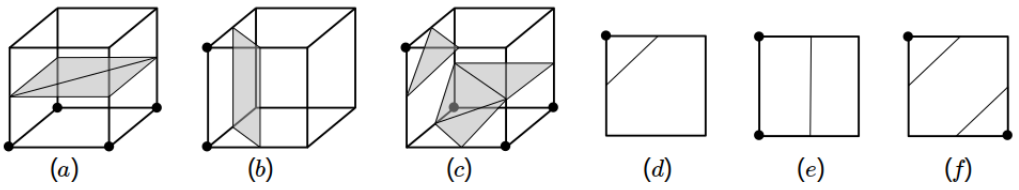

Figure 1: *Representative configurations for the 3D (left) and 2D (right) case.*

offered further illustration of *vertex* and *fragment shaders* through the integration of the three different stages.

3. Description and Implementation Details

Here we sketch the main ideas of MC and refer the reader to Section 4 in [LC87] for the complete details. The algorithm extracts an isosurface from volume data for a user-supplied intensity value (*isovalue*) using a sliding cube over the data. For each placement of the sliding cube the algorithm thresholds to 0/1 the voxel intensities at the cube vertices against the given isovalue and infers a surface configuration by emitting a collection of triangles. For example, if all four vertices of a single face of the cube are thresholded to 1 (Figure 1*a*), then we can infer that we have a transition from inside the volume to the outside and that the face represents the volume boundary. Thus, we can emit triangles that represent the face for rendering. Similarly, if the two vertices of a single edge of a face are thresholded to 1 (Figure 1*b*) we can infer a transition point and emit triangles that separate the edge (inside) from the other six vertices (outside). In the extreme either all vertices of the cube are thresholded to 1 (or to 0), in which case no triangles are emitted since we are entirely inside (or outside) the volume. Altogether 15 canonical cases (up to symmetry) are identified in [LC87] of which we have reproduced three representative cases in Figure 1*a-c*.

We can now see the natural application of *geometry shaders* in the algorithm implementation. The *geometry shader* receives as input the 8 vertices of the cube and emits a variable number of triangles depending on the 0/1 state of the vertices. This is also a place where the instructor can introduce (or reintroduce) *triangle strips*, since many of the configurations have this natural representation.

The correspondence between cube configuration and triangle primitives is achieved via a lookup table. Each configuration can be identified by a bit sequence representing the state of the cube vertices. The integer value of the bit sequence is used as an index in the lookup table to retrieve an encoding that allows for constructing the triangle primitives. In our implementation the encoding was a sequence of pairs of integers, each pair indicating which of the 8 vertices received by the shader represent 0—1 edge; the value -2 was used to separate triangle strips within a configuration; and -1 was used to terminate the encoding (the lookup table has fixed dimensions, but the encodings have different lengths). For example, using a top-to-bottom, back-to-front labelling of the vertices, the bit sequence, the index, and the encoding for the configuration shown in Figure 1*c* are 10110100, 180, and {2,0,2,3,2,6,-2,5,1,4,0,7,3,4,6,7,6,-1,..0s...}, respectively. The vertices of the emitted triangles represent the constant isovalue, i.e. the intersection of the isosurface with the cube edges, so the vertex positions should be interpolated along the edges based on the endpoint intensities. Our implementation assigned the average intensity to add variation and avoid the flat appearance due to the absence of lighting or other cues.

There are 256 possible configurations (8 vertices in two possible states) and the longest encoding for our scheme is 28, so the lookup table was represented as `int[256][28]`. This can be reduced to `int[256][16]` if the encoding is based on labelling the midpoints of the edges, with a bit of extra work (e.g. additional lookup in `int[12][2]` table) to identify the vertices of the midpoint's edge. Other encoding are possible as well, so this could be an opportunity for instructors to discuss the issue of representation in computer graphics and space-time trade-off.

One implementation consideration is how to communicate the lookup table to the *geometry shader*. This presents an opportunity to introduce *uniform buffer objects (UBO)* or *shader storage buffer objects (SSBO)* to communicate a large block of GLSL *uniform data*. UBO might be a bit more natural given prior familiarity with *uniform variables*, but for this application there is essentially no difference in the setup and use of either structure. The only practical consideration is that UBO have a guaranteed limit of 16KB (which is exceeded by the `int[256][28]` representation), but the hardware is likely to allow larger sizes.

Finally, we mention the data format although instructors are likely to have different means of loading image and volume data. The data was read from `PGM` image files (grayscale raw `PPM` variant) and each volume data was stored in a separate directory with individual images for each volume slice. Internally, the slices were stored consecutively in row-major form in a 1D array (*C++ vector*) as a 4-tuple that assigned to each pixel/voxel its coordinates and grayscale intensity. During loading the coordinates were set to be in $[-1,+1]^3$ at equal intervals along each axis to eliminate the need for transformations. In the shader pipeline the thresholded 0/1 intensity values were used to identify each configuration. Finally, the positions and intensity values for the emitted triangle primitives were computed as mentioned earlier.

4. Setup

The students received the following handouts—only `geometryShader.glsl` was new, since the rest had been used in prior assignments on GLSL:

- `shaderutils.(h|cpp)` — simple utilities for loading a shader or creating a shader program

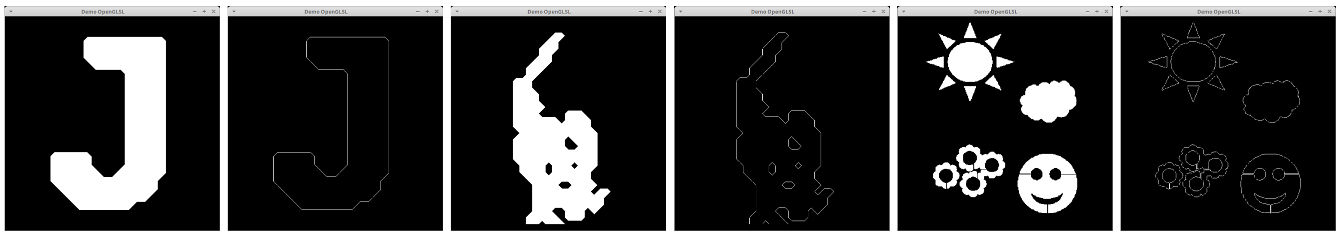

Figure 2: Marching Squares *results showing the original image and the extracted contours.*

- `main.cpp` — OpenGL/GLUT setup for rendering a square
- `vertexShader.glsl` and `fragmentShader.glsl` — basic pass-through shaders
- `geometryShader.glsl` — simple example that receives a vertex and emits a diamond shape centered at the vertex; when enabled it changes the rendering of the square in `main.cpp` to a rendering of four diamonds at the square vertices

The assignment was completed over the course of two weeks and the major components are described in the following sections. Results are shown in Figures 2 and 3, respectively.

4.1. Marching Squares

The first task was to implement a variation called *Marching Squares (MS)* for extracting a contour from 2D image data, instead of a surface from 3D volume data. The algorithm works exactly as described in Section 3, but slides a square over the data emitting line segments depending on the 0/1 state of the vertices (Figure 1*d-f*). The number of possible configurations is reduced significantly (256 to 16) which shrinks the lookup table from `int[256][28]` to `int[16][28]`. This 2D variation has the same conceptual and implementation structure as the original, but its 2D nature aids significantly in grasping and visualizing the details of the algorithm.

For the MS component the students completed the following tasks:

- *pixel/voxel representation*: Each pixel was a 4-tuple that stored the pixel coordinates (at this stage `z` was 0) and intensity. In the shader the intensity was retrieved via the `.w` component, which simplified the data and attribute setup, but instructors could consider different representations.

- *image implementation*:

  - `constructor` — read the data from `PGM` file
  - `setup()` — create data buffers and describe attributes
  - `draw()` — activate the buffers and draw the data

  The main focus here was on method `setup()` which had the task of initializing the data and attribute buffers, and computing correctly the strides and offsets to ensure that the *geometry shader* received data for 4 adjacent pixels in a square configuration within the image.

- *lookup table generation*: The lookup table was generated by hand based on careful examination of the 16 possible configuration producing the pair encoding described in Section 3. The work is made even more manageable, since there is only rotational symmetry and therefore symmetric cases can be obtained successively via +1 increment.

  Since the lookup table was quite small, at this stage it was included directly in the *geometry shader* bypassing the need for UBO/SSBO.

- *geometry shader modification*: The *geometry shader* form the handout was modified to receive the 4 vertices of the sliding square and emit the *line strips* for the current 0/1 configuration:

  1. threshold the intensity value stored in `.w` against the isovalue
  2. form the configuration index using simple bit operations to combine the thresholded values into a binary sequence
  3. use the index to retrieve the encoding from the lookup table
  4. traverse the encoding emitting a vertex for each edge pair; complete the primitive at -2; stop at -1

To aid testing and debugging the students were advised to use initially a hardcoded 2x2 grid and assign 0/1 intensity values to test each of the 16 configurations. Once all configurations were tested thoroughly any dataset was likely to work correctly.

4.2. Marching Cubes

The MC stage was only a short step away from MS. Essentially, no extra code had to be written, but instead it was only necessary to make simple adjustments to process 8 elements instead of 4. Importantly, note that steps 3 and 4 in Section 4.1 require no modification—switching the *geometry shader* to emit *triangle strips* completes the transformation:

- extend method `setup()` to compute the correct strides and offsets to ensure that the *geometry shader* receives data for 8 adjacent voxels in a cube configuration within the volume

- extend the *geometry shader* to compute a bit sequence (index) from 8 thresholded intensities; each encoding is processed as before since there is no change in the description of vertices to emit

- send to *geometry shader* the lookup table via UBO or SSBO; here it is not practical to generate the lookup table by hand, so we suggest that instructors provide it

Students were again advised to use a hardcoded dataset—2x2x2

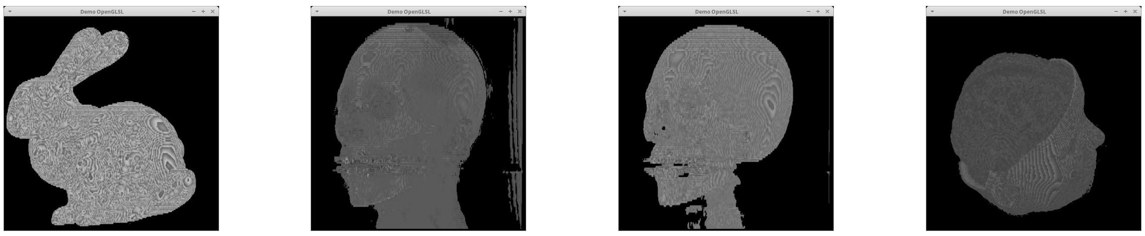

Figure 3: Marching Cubes *results showing the extracted surface. The images in the middle represent different levels of intensity thresholding.*

single cube/volume—and check a few representative configurations. At this stage it is not practical to check all configurations, but assuming the lookup table is provided and the previous stage is implemented correctly, the main issues are likely to be confined to the code repetition and adjustment steps above.

For our implementation the students were required to write a separate program to generate the lookup table suggested in [LC87]. The details are beyond the scope of this short paper and it is not clear whether much is gained by requiring this component. It is perhaps better to include it as an optional exercise or as a final project on the challenges in creating a lookup table that correctly handles the ambiguities that are present in the original table.

5. Discussion

This paper presented ideas for illustrating the modern shader pipeline using *Marching Cubes* which has a number of appealing aspects: it is feasible to implement as a single unit within a computer graphics course; offers a natural context that motivates *geometry shaders* and advanced data communication via UBO/SSBO; and reinforces ideas such as data and attribute buffer use and initialization, *vertex* and *fragment shaders*, and *triangle/line strip* primitives.

Instructors could consider only the *Marching Squares* implementation which can be completed as a one-week assignment and has all the elements of the main algorithm, including the practical application of computing a dataset boundary. Alternatively, the full algorithm can be completed over the course of two weeks by adding additional features including projections and geometric transformations for interacting with the dataset, or varying the threshold to reveal different regions in the data set (the last extension was included in our implementation). An interesting follow-up would be to investigate whether this could be incorporated in the *ShaderLabFramework* [TRK17], which supports *geometry shaders* and allows for interactive editing of shader code in an IDE-like interface.

Other possible ideas include adding lighting and normals computation. This was not included in out implementation, but will be considered in the future. In the setup presented here computing smooth normals is not possible, since connectivity is lost during the *geometry shader* stage. However, it is trivial to compute per-triangle normals, so that could be a simple extension. A feasible adjustment for computing smooth normals within the given framework is to send more data to the *geometry shader*.

Finally, instructors could use this as a starting point for final projects. The presentation here was based on the original paper [LC87], but there has been a significant amount of follow-up work that can form the basis for further exploration.

6. Acknowledgments

We thank Dr. Sunghee Kim at Gettysburg College for suggesting *Marching Cubes*. The results shown in this paper are from the work of Yidan (Tracy) Tang.

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

[LC87] Lorensen W. E., Cline H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (Aug. 1987), 163–169. URL: https://doi.org/10.1145/37402.37422, doi:10.1145/37402.37422. 1, 2, 4

[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, 4