WebGL Instanced Transform/Feedback for Continuous Iso-surface Rendering and Other Bespoke Processing

Aaron Watters
awatters@flatironinstitute.org
Flatiron Institute
New York City, New York, USA

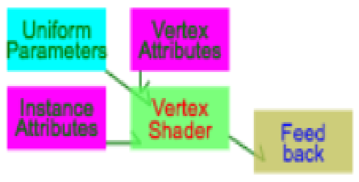

Figure 1: Instanced vertex shader processing element inputs with feedback output.

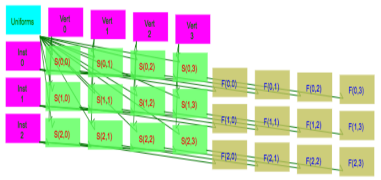

Figure 2: Instanced transform feedback vertex processing parallel pipeline.

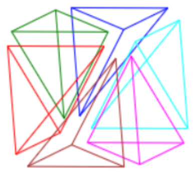

Figure 3: Each voxel cube is tiled by 6 tetrahedra which each generate 0, 1, or 2 triangles on an iso-surface.

ABSTRACT

WebGL2 provides high performance graphics in web browsers but it also also exposes the raw processing power of the underlying graphics processing unit. WebGL instanced rendering allows parallel processing of multiple instances of similar objects of high complexity and the WebGL Transform/Feedback mechanism allows outputs from vertex processing to be captured and extracted or fed back into other processing stages. This paper describes the FeedWebGL2 Javascript library which encapsulates WebGL2 instancing and transform/feedback in a convenient framework. The library includes use cases for the encapsulated features including tools for transforming dense 3 dimensional arrays into contour line segments or iso-surfaces. The primatives generated by the processing can be captured and fed into other graphics libraries such as three.js [3]. Contours and iso-surfaces may be used to implement interactive medical and scientific imaging on web pages.

CCS CONCEPTS

• **Mathematics of computing** → *Mathematical software performance*; • **Computer systems organization** → *Single instruction, multiple data*; • **Human-centered computing** → **Visualization toolkits**; • **Information systems** → **Browsers**.

KEYWORDS

WebGL2, HTML5, Transform, Feedback, contour, isosurface, visualization

ACM Reference Format:
Aaron Watters. 2020. WebGL Instanced Transform/Feedback for Continuous Iso-surface Rendering and Other Bespoke Processing. In *Web3D 2020 Conference, June 24–26, 2020, Seoul, South Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

WebGL [7] is a feature of HTML5 which allows web pages running in browsers to use the graphics processing unit (GPU) of the client computer to display high quality performant graphics. Javascript programs configure WebGL components by compiling, configuring, and running shader programs written in the GLSL shader language. The shader programs run in parallel on a large number of processors in the GPU in "single-instruction multiple-thread" mode. The WebGL processing typically generates raster images which are often directly displayed on the user's screen.

WebGL2 [4] adds a "transform/feedback" feature – the ability to capture intermediate processing outputs from a WegGL2 program run. These captured outputs may be copied into the Javascript context, or pipelined to other processing on the GPU.

This paper describes a Javascript library, FeedWebGL2 [8], which streamlines the process for implementing transform/feedback processing stages using WebGL2. The paper also describes included use cases for transform/feedback which implement contour generation and isosurface rendering, and how those components can be integrated with other Javascript graphics libraries such as three.js [2].

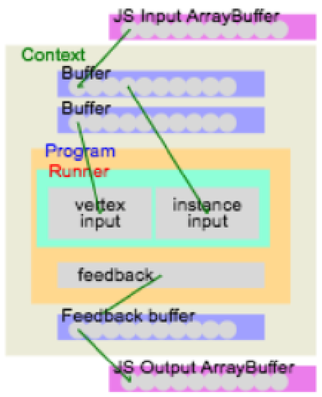

Figure 4: Conceptual object wrapper structure for FeedWebGL2 encapsulation of instanced transform/feedback.

2 FEEDBACK ENCAPSULATION

The FeedWebGL2 library structures the primary objects of the WebGL2 transform/feedback mechanism using higher level wrappers which manage object relationships and bookkeeping. Figure 4 illustrates the primary object wrappers and their interconnections. The encapsulation provided by the library greatly reduces the conceptual load required of the programmer when implementing instanced transform/feedback processing.

WebGL rendering is broken into a number of processing stages. For the purposes of this paper we are only interested in the vertex processing stage where instancing and transform/feedback are implemented. In vertex rendering each processing element runs a vertex shader program which receives a small number of input parameters (often including a vertex location and color) and generates a small number of output parameters (often including a modified vertex location and possibly altered color). Figure 1 illustrates a single processing element where the vertex shader sees uniform parameters shared across all elements, and instance attributes shared by all elements for this instance, and vertex attributes shared by all elements for this vertex.

The WebGL vertex processor is implemented using a special purpose programming language called GLSL. The FeedWebGL2 library streamlines the process of configuring and running a GLSL vertex shader.

Instanced rendering is designed to generate primatives for a large number of similar object instances where the object similarities are captured as vertex attributes broadcast across all instances and the object differences are broadcast across all vertices for the instance. Figure 2 illustrates the processing pipeline where there are three instances which each have four vertices. One might use instancing to implement many dancing stick figures by capturing the lengths of the arms and legs as vertex attributes (shared by all stick figure instances) but broadcasting the positions of the arms and legs for each individual figure as instance attributes.

In the context of transform/feedback the inputs and outputs of the shader need not be directly related to graphical primatives, but may implement any sort of computation where few inputs produce few outputs

To create an instanced feedback program using feedWeGL2, instantiate a context manager and use it to load WebGL buffers with data needed for the inputs. Use the context to compile the vertex shader GLSL program, identifying the feedback variables to extract from the program runs.

To run the program create a runner object from the program, specifying shared uniform parameters and their initial values. Then specify vertex and instance inputs and attach the inputs to data in the context buffers. Note that inputs may be attached to a buffer at an offset, and it is often useful to attach several inputs to the same buffer. In iso-surface processing, for example, the function values at each of the 8 corners of each voxel are attached to the same "values" data buffer. The runner will automatically configure output buffers for feedback variables specified for the program.

Once the runner has been configured with attached data as illustrated in Figure 2 initiate the run to process the broadcasted inputs. Often the "rasterization" stages for the program are disabled if only the feedback output is desired and rasterization is not required.

After the run the feedback buffers may be copied out of the graphics contexts (as shown in Figure 2) or to other buffers in the graphics context.

A configured runner can be run many times with different values assigned to the uniform parameters and sometimes changes to the input buffers. For example the iso-surface processing stage described below is usually run many times with different cut-off values to produce different iso-surfaces from the same underlying matrix in real time.

3 CONTOUR PROCESSING

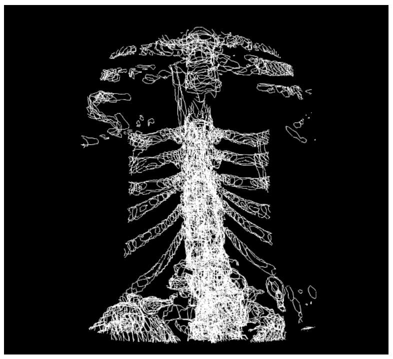

Figure 5: Contours generated from a CT-scan image stack captured from a human torso.

The FeedWebGL2 library was created in part to facilitate the presentation of 3 dimensional arrays. The library implements both 2d contour generation (Figure 5) and 3d isosurface generation.

The contour generation stage is implemented using "marching triangles" which is a simplification of the "marching cubes" method [6] for generating iso-surfaces. The input to the contour generator is a 3d array of floating point values and a target "cut-off" value for the contours. The output of the generator is a sequence of line segments approximating points where the array "crosses" the target value, and also "degenerate" line segments where pixels of the array do not cross the target value as illustrated in Figure 6

In order to implement marching triangles using feedWebGL2, consider each pixel of the input array to be an "instance" and load the input array into a single buffer. Index each of the four corners of each pixel of the input array using separate instance input variables. For each pixel instance divide the square cornered at the pixel into two triangles. If the corner of a triangle crosses the target value then that triangle generates a "real" line segment, but if not it generates a degenerate line segment. This results in 2 line segments for each pixel of the input for a total of 4 vertices per pixel, many of which are typically degenerate.

The FeedWebGL2 contour and isosurface implementations avoid conditional branching in the GLSL code by using indexing into static arrays whereever possible in order to accelerate the single-instruction multiple-thread execution.

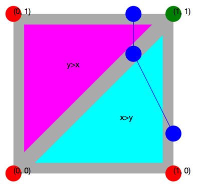

Figure 6: In marching triangles each pixel of the input layer is divided into 2 triangles which each contain 1 or 0 crossing line segments approximating the target value. Here the red vertices are above the target and the green vertices are below the target and the blue points are interpolated approximations where the value is achieved.

4 ISOSURFACE PROCESSING

FeedWebGL2 implements isosurface generation (Figure 7) using "marching tetrahedra" [9] which is also a simplification of "marching cubes". In this case the input is a 3d array of floating point values with a specified cutoff value and the output is a sequence of triangles where the array approximates the cutoff (and degenerate triangles where no approximation was found in a voxel).

In this case each voxel of the input array is tiled into 6 tetrahedra (Figure 3). Each tetrahedron in turn generates one (Figure 8) or two (Figure 9) or zero non-degenerate triangles approximating the surface crossing the cutoff value. For each valid triangle the vertex shader also generates a "unit normal" vector approximating a vector pointing away from the surface. Normals are often used for lighting effects in three.js, for example.

In the FeedWebGL2 context we consider each voxel of the input matrix to be an "instance" divided into 6 tetrahedra with two triangles each for a total of 36 vertices per instance many of which will be degenerate.

5 INTEGRATION WITH THREE.JS

FeedWebGL2 processing outputs may be pipelined into other javascript libraries such as three.js in order to take advantage of the extensive features available in external libraries. At present the pipelining

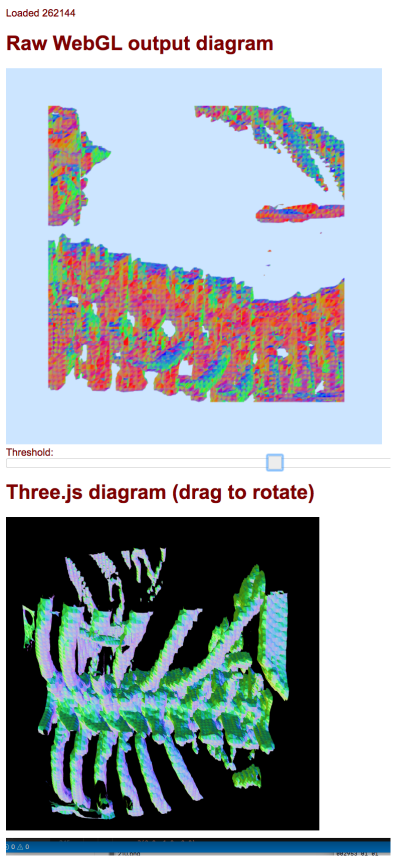

Figure 7: Iso-surface generated from a 3d CT-scan image stack. The surface is computed using feedWebGL2 and pipelined to a three.js presentation.

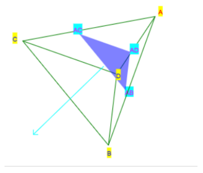

Figure 8: In marching tetrahedra a tetrahedron with one "high" or "low" vertex generates one non-degenerate triangle.

method involves copying the feedback buffer context out of the GPU context and into the main computer memory and then back into a different GPU context controlled by the three.js library. By using the BufferGeometry feature of three.js the feedback output may be used directly without (for example) removing degenerate elements.

Although the buffer copying is fast, the copy operation does become a bottleneck at sufficiently large data sizes and it should be theoretically possible to pipeline feedback outputs directly on the

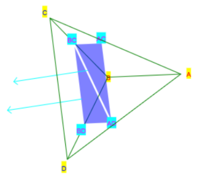

Figure 9: In marching tetrahedra a tetrahedron with two "high" or "low" vertices generates two non-degenerate triangles.

GPU with no external copy operation required – but this would require a patch to three.js (see future work below).

6 IMPLEMENTING OTHER PROCESSING STAGES

FeedWebGL2 will also be useful for implementing other sorts of processing stages. Many of these processing stages will be most useful when combined into longer workflows The following is a list stages to be added to the standard collection included with the library:

Cut by plane: Given a sequence of points, line segments, or triangles, "cut out" those elements which do not fall on one side of a 3 dimensional plane. Here "cut out" means "transform to degenerate".

Interpolate time step: Given two buffers of float values and a mixing parameter, interpolate a "mixed" buffer. This will be useful for combining iso-surface rendering with animation in the context of 3d light microscopy.

Min, max, and combine: From two buffers of floats, generate the min, max, or linear combination of the buffer values.

Apply color interpolation: From a buffer of floating point values generate a buffer of 3 dimensional color interpolations using an interpolation rule.

Shrink triangles: Generate a "shrunken" sequence of triangles from a sequence in input point triples. This will allow the user to "look through" an object without using color transparency (which often doesn't work well).

7 CONCLUSIONS AND FUTURE WORK

This paper described FeedWebGL2, a Javascript library for building special purpose processing stages using the instanced transform/feedback features of WebGL2. We plan to explore the following directions, among others:

Texture/feedback support: The library currently has not implemented features for mapping textures into the graphics context for use by the vertex shader.

Experiment with point based processing: FeedWebGL2 could facilitate the implementation of point based methods [1] for an alternative presentation of 3 dimensional data.

Better three.js integration: The three.js library currently does not provide direct external access to internal WebGL buffers managed by the library. Since direct GPU-to-GPU communication of feedback outputs into three.js structures requires buffer access, we intend to develop a pull request to the three.js repository which exposes the internal buffers.

External export: The library will also include methods for extracting the non-degenerate values from the feedbacks for external storage or other use.

Jupyter integration: We plan to make the FeedWebGL2 functionality available in Jupyter [5] using the Jupyter widget framework.

ACKNOWLEDGMENTS

This work was inspired by discussions with many scientists at the Flatiron Institute who described diverse ideas for interactively visualizing data sets, including Rachel Sealfon, Nick Carriero, Pat Gunn, Manuel Zingl, Danxun Li and others.

REFERENCES

[1] Alex Laier Bordignon, Luana Sá, Hélio Lopes, Sinésio Pesco, and Luiz Henrique De Figueiredo. 2013. Technical Section: Point-based Rendering of Implicit Surfaces in R4. *Comput. Graph.* 37, 7 (Nov. 2013), 873–884. https://doi.org/10.1016/j.cag.2013.06.005

[2] C. Cabello. 2017. three.js. https://github.com/mrdoob/three.js.

[3] J. Dirksen. 2013. *Learning Three.js: The JavaScript 3D Library for WebGL.* Packt Publishing. https://books.google.com/books?id=6TVeAQAAQBAJ

[4] F. Ghayour and D. Cantor. 2018. *Real-Time 3D Graphics with WebGL 2: Build interactive 3D applications with JavaScript and WebGL 2 (OpenGL ES 3.0), 2nd Edition.* Packt Publishing. https://books.google.com/books?id=Qel1DwAAQBAJ

[5] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.

[6] C. Maple. 2003. Geometric design and space planning using the marching squares and marching cube algorithms. In *2003 International Conference on Geometric Modeling and Graphics, 2003. Proceedings.* 90–95. https://doi.org/10.1109/GMAG.2003.1219671

[7] Tony Parisi. 2012. *WebGL: Up and Running* (1st ed.). O'Reilly Media, Inc.

[8] A. Watters. 2020. feedWebGL2. https://github.com/flatironinstitute/feedWebGL2.

[9] Michael Wehle and Heinrich Müller. 1997. Visualization of Implicit Surfaces Using Adaptive Tetrahedrizations. *Dagstuhl '97 - Scientific Visualization Conference* 00, undefined (1997), 243. https://doi.org/doi.ieeecomputersociety.org/10.1109/DAGSTUHL.1997.10005