Realtime Mesh Generation via Marching Cubes Implemented in CUDA

Nicolas Mignucci, George Ralph

URL:

https://github.com/gdr22/15-466-Final-Project

Summary:

We are going to attempt to fully implement the Marching Cubes mesh generation algorithm in CUDA on the GPUs in the GHC servers, with hopes of being able to generate meshes in real-time. This implementation will ideally produce identical results to the one we had previously written for the Magic Leap One headset. Assuming we are able to get a functional implementation in CUDA, we may port the algorithm to HLSL on the Magic Leap and profile the performance of the new and old implementations.

Background:

The marching cubes algorithm is a method to turn a 3D scalar field into a planar mesh of the zero-intersection level surface of the scalar field. The field is discretized to a 3D grid of samples. Each voxel created by eight neighboring samples potentially contributes triangles to the output mesh. The triangle arrangement itself is determined simply by the signs of the eight samples, and with 256 configurations, can be stored as a lookup table. Generating the mesh points however relies on linearly interpolating the zero crossing between two adjacent samples.

A naïve approach to parallelizing this algorithm may attempt to exploit the lack of dependence between results for each voxel in the problem and compute each voxel in parallel. This approach, while correct, fails to notice that not all voxels contribute triangles to the mesh. In practice, very few do. Thus, a considerable amount of work performed is entirely unnecessary.

Additionally, the generation of a mesh generally assumes a contiguous array of both vertices and triangles in the output. While this is trivial in an iterative algorithm, a CUDA algorithm will require some combination of scan and scatter operations to recreate this output format. Additional complexity in a similar vein is added when we also insert triangle faces for all voxels on the border of the grid.

The Challenge:

Marching cubes is composed of largely independent operations on neighborhoods of 8 samples, thus there are some segments which are embarrassingly parallel. In terms of avoiding useless or redundant work however, we will need to divide our algorithm into multiple passes in order to ensure effective work division. Given that we are working on a GPU, this means some higher level structuring of our program beyond naively computing each voxel in parallel will need to be considered.

As a single edge can exist in up to 4 adjacent voxels, it will likely be advantageous to avoid recomputing this value for each voxel, thus additional work will be necessary to ensure that values are not computed redundantly. In a similar vein, storing rather than recomputing these redundant computations opens the door for more complex memory accesses, thus we will likely consider different work division patterns which could help to minimize the number of global memory accesses our algorithm performs.

Resources:

The original Marching Cubes algorithm is described here in the following 1987 paper. We will refer to this as a primary point of reference.

https://dl.acm.org/doi/10.1145/37402.37422

We have an existing largely sequential reference implementation written in C# and HLSL which we will attempt to match our results against. For this project, we will be rewriting the algorithm from scratch in C++ and CUDA. After improving work division and memory accesses on the GHC machines' GPUs, we may have enough time to attempt porting our new algorithm back into HLSL to run on the Magic Leap, using the existing framework written to support the original implementation. We currently have access to the Magic Leap hardware, thus no additional resources will be necessary.

Goals and Deliverables:

- PLAN TO ACHIEVE
  - Borderless, no-vertex-sharing marching cubes in CUDA with more efficient work sharing
  - Border meshing
  - Vertex sharing
  - Performance profiling based on voxel access patterns
  - Better utilization of shared memory
  - Ideally, we should be able to achieve real-time (60FPS) mesh generation for grids over size 32x32x32
- HOPE TO ACHIEVE
  - Port the CUDA code to HLSL on Magic Leap
  - Profile HLSL version against original implementation
    - We should see much higher speedup

    - Ideally, for the poster session, we would be able to demonstrate our algorithm producing high resolution meshes in realtime on the device using the existing UI framework demonstrated here

Platform Choice:

Ideally, we would like our marching cubes implementation to be computable in real-time, as this has useful applications in interactive data visualization. Because of this constraint, plus the relatively small amount of compute for each voxel, and the fact that the number of voxels scales cubically with our selected sample resolution, we feel that a GPU would be the most appropriate platform to implement our algorithm.

Due to their availability and power, we will start our development on the GPUs included in the GHC machines. After achieving a stable and efficient implementation on these systems though, we may take advantage of some of the existing marching cubes interactive visualization code we had previously written for the Magic Leap One augmented reality system, to test our code in an existing use case compared to a less parallelized implementation.

Schedule:

- **(11/8/21)** Complete non-CUDA scaffolding code (input scalar field, output mesh file)
- **(11/15/21)** Complete v1 CUDA code (interior faces, no vertex sharing, improved work division)
    - Interior faces refers to the fact that stock marching cubes will not generate faces for the boundaries of the cube of samples.
    - The reference implementation reuses some of the results of our marching cubes implementation to fill in these holes with marching squares, a lower dimensional version of the same algorithm. (This shares many computations with the original algorithm, thus there isn't much in terms of difficulty)
- **(11/22/21)** Add border faces and vertex sharing (Milestone)
    - Vertex sharing refers to the fact that adjacent voxels share some vertex position results in terms of computation. By identifying and sharing these vertices, we will be able to reduce the size of our mesh, and avoid redundant computations.
- **(11/26/21)** Investigate memory access patterns
- **(12/3/21)** Port CUDA code to HLSL/Integrate with Magic Leap code
- **(12/6/21)** Profile performance on ML with original implementation
- **(12/10/21)** Poster presentation session