Project CUDC Report

OUNAN DING
Student ID: 861194909
oding001@ucr.edu

Contents

1 Overview

This report is for the final project of CS/EE-217 GPU Architecture and Programming. Implementations of dual contouring will be presented and analyzed.

We provide a CPU implementation written in C++ first as our baseline, then move on to introduce the CUDA implementations. We will see what the bottle neck is in our CPU version, and how we make trade off in the CUDA version to overcome that bottle neck. A visualizer is also provided as a Blender add-on to preview the results and check the correctness.

Finally we will provide some performance testing and detailed analysis in the last part of this report.

2 Design and Implementation

2.1 General Overview

In this section we will provide a general overview of dual contouring. The architecture-independent perspectives of this algorithm will be discussed.

The primary paper used for this project is [JLSW02]. [SWU] also provides supplemental materials on how to construct and solve the Quadratic Error Function(QEF).

For the first step in dual contouring, given an implicit function such as $(x^2 + y^2 - 1)^3 - x^2y^3$ in 2D, or $x^2 + y^2 + z^2 - 1$ in 3D, we will sample the function on a uniform grid. Here is an example in figure 1, where we sample the function $x^2 + y^2 - 1.7$. The 0 level set of this function is plotted as a circle of radius 1.7. The samples which have a negative value are drawn with filled dots.

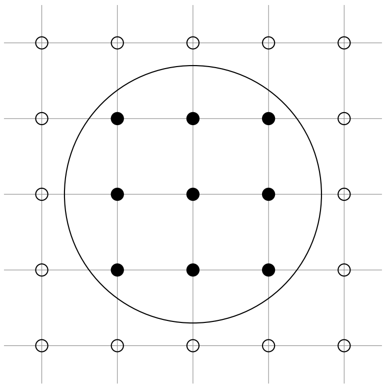

Figure 1: Sampling of $x^2 + y^2 - 1.7$

Every edge which exhibits opposite signs at its ends has a potential intersection with the 0 level set of the function. We highlight these edges in figure 2.

The intersections of these edges can be found by a binary search. And the gradients at the intersections (the Hermitian data in the glossary of [JLSW02]) can be computed by finite differencing. The result of this step is illustrated in figure 3.

After the Hermitian data is ready, the QEFs can be built.

$$
\begin{aligned}
E[x] &= \sum_i (n_i \cdot (x - p_i))^2 \\
&= x^T A^T A x - 2x^T A^T b + b^T b
\end{aligned}
$$

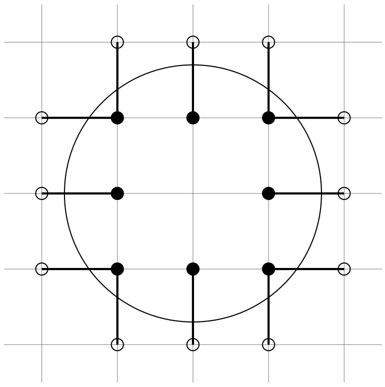

Figure 2: The intersected edges

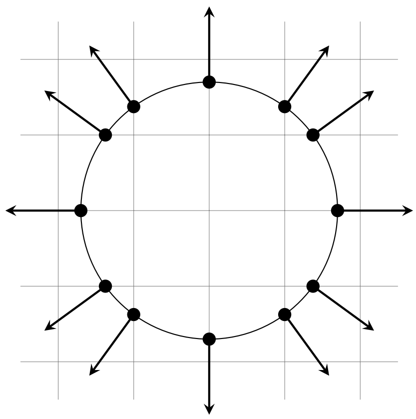

Figure 3: Intersections and their Hermitian data

In 2D dual contouring, each QEF can be represented by 7 floating point numbers. Since $A^T A$ is a 2×2 symmetrical matrix, it requires 3 floats. The $A^T b$ is a 2×1 column vector, so it requires 2 floats. We also store 2 floats for the mass point, which is a 2×1 column vector and will be used in mass point projection (see [SWU] for details about mass point projection).

We solve the QEF by computing the pseudo-inverse as suggested in [JLSW02]. And compute a 2×2 SVD for the pseudo-inverse. This method is described in [Bli03].

After solving all of the QEFs, we get all vertices of the result mesh. We can build the topology by connecting vertices from adjacent grid cells.

Figure 4 illustrates dual contouring results of a heart, a squircle($x^4 + y^4 - 1$), and the subtraction of them.

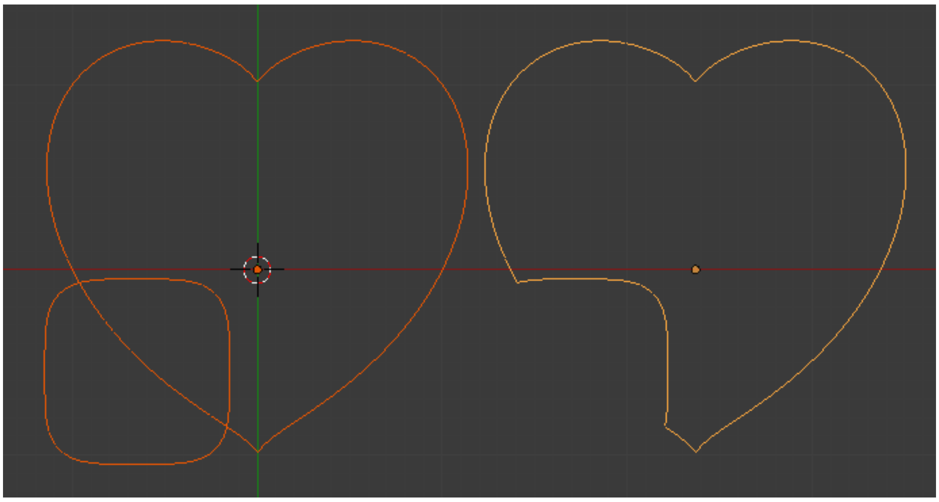

Figure 4: 2D dual contouring example

2.2 The CPU Implementation in C++

There are several considerations in our implementation of dual contouring on CPU.

First, a trivial implementation which stores the whole mesh grid suggests a spatial complexity of $\mathcal{O}(n^2)$(or $\mathcal{O}(n^3)$ in 3D). We do not want the memory consumption be a bottle neck, so we do the sampling in a row by row manner. Then only one row has to be maintained during sampling, and one extra row has to be maintained for the topology construction since we have to connect edges to the vertices in the previous row. Thus the spatial complexity can be reduced to $\mathcal{O}(n)$ in 2D(or $\mathcal{O}(n^2)$ in 3D).

Second, we think the sampling would occupy the majority time of the whole computation(However this turns out to be *false*. See our later performance report and analysis). So we want to reduce the computation as much as we can. Observing that each sampling vertex has 4 adjacent grid cells, we can do the sampling only once, cache it in memory, and reuse it for all adjacent grid cells.

The computing scheme is illustrated in figure 5. Each dots indicate a sampling point. And the dashed rectangle indicates the row we are working on. The hatched area indicates a 4 elements wide SIMD pack if SSE is used.

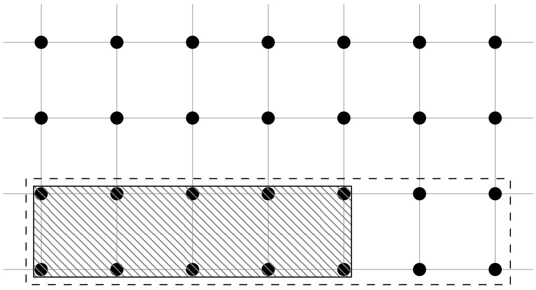

Figure 5: The computing scheme in C++

2.3 The CUDA Implementation

3 Analysis

4 Conclusion

References

[Bli03] J. Blinn. *Jim Blinn's Corner: Notation, Notation, Notation.* Jim Blinn's corner. Morgan Kaufmann Publishers, 2003.

[JLSW02] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. *ACM Transactions on Graphics (TOG)*, 21(3):339–346, 2002.

[SWU] Scott Schaefer, Joe Warren, and Rice UniversityE. Dual contouring: The secret saucec.