Mesh Simplification

Course Project 1, Computer Graphics, Tsinghua University

Our task is to implement a classical Mesh Simplification algorithm "Surface Simplification Using Quadric Error Metrics", SIGGRAPH 97.

Run My Code

I implement the algorithm in C++ for efficiency and use `Makefile` as the build tool. Make sure your environment has a C
++ compiler (`g++` or `clang++`). Then run the following command in the root directory. The binary `ms` (short for `mesh simplify`) will be compiled and built.

```
bash build.sh
```

Run examples under `obj/`, using:

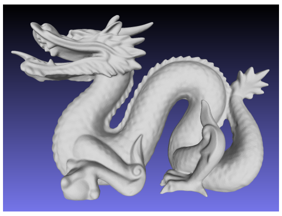
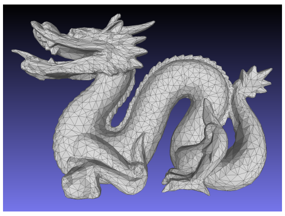
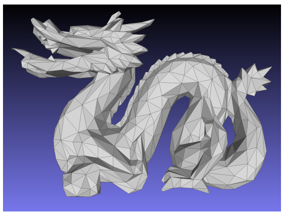
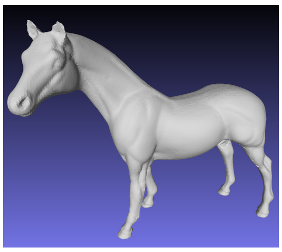
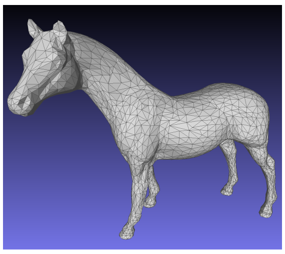
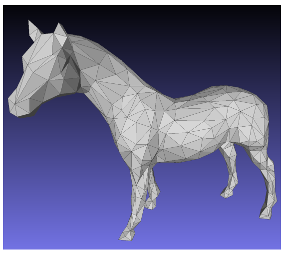
```
./ms obj/Input/Dragon.obj obj/MyOutput/Dragon_0.1.obj 0.05
```

Here `0.05` is the simplification ratio (the number of faces of the result / the original faces number). The threshold used in pair selection is hardcoded as `0.01` in `main.cpp`, which you can also modify.

Result

Note: `threshold=0.01` in all following settings.

Visualization of Some Examples

	Original Mesh	ratio=0.05	ratio=0.01
Dragon.obj			
	Vertices: 104855, Faces: 209227	Vertices: 5236, Faces: 10460	Vertices: 1047, Faces: 2091
Horse.obj			
	Vertices: 48485, Faces: 96966	Vertices: 2426, Faces: 4848	Vertices: 486, Faces: 968

Efficiency

ratio=0.1, Optimization choose y-axis to sort (Block.obj, Cube.obj, Dinosaur.obj and Sphere.obj are too simple so we don't include it)

Object	Total Time (s)	Avg. Error
Arma	2.27	4.06e-6
Buddha	7.25	5.99e-05
Bunny	91.53	7.24e-07
Dragon	30.74	2.17e-05
Horse	5.26	2.15e-05
Kitten	2.30	0.099

Implementation

First I try to implement in Python using `trimesh` library to load/store Mesh, but soon I found Python is too slow for this task (Although we can use Torch/Triton to parallelize some computation in GPU, the bottleneck of this algorithm (Select a pair from the top of heap, contract them, repeatedly) is hard to parallelize. So finally I embrace C++.

Load/Store Mesh

The format of the mesh is not complicated. Each line represents a Vertex (start with `v`) or a Face (start with `f`). For a Vertex, the following three numbers are the coordinates (`x`, `y`, `z`); For a Face, the following three integers represent the indices of three endpoints of this triangle (Each face is a triangle).

Data Structures

I implement several basic data structures which are necessary for the algorithm: `class Vertex`, `class Triangle` and `class VertexPair`. And the whole `Mesh` is wrapped as a `class Mesh` so that we can operate it easily.

We allocates memory for each `Vertex` and `Triangle` (a.k.a `Face`) when loading them, and reference them using C++ pointers to avoid necessary copy.

Algorithm Implementation

Most part follows the original paper. There are some points worth mentioning:

- For the heap, I use `std::priority_queue` in C++ STL by overriding the `<` operator of `class VertexPair`. And since it's hard to perform `delete` and `update` operations in `priority_queue`, I do this in a **lazy manner**:
    - For `delete`, I mark the corresponding vertex as removed by setting its index to `-1`. And each time we pop a pair from the heap, we will check whether the pair contains deleted vertex. If so, discard it and pop another pair.
    - For `update`, I maintain a `timestamp` in each pair and record the newest timestamp for each `vertex_id` pair. If I found the pair we pop is not the newest, a.k.a it's expired, we will discard it too.

- For calculating `\overline{v}` from `v1` and `v2`, we need to calculate the determinant and inverse of a 4th order matrix. I calculate it directly by violently expanding to achieve a better performance. if the matrix is not invertible, we use `(v1 + v2) / 2` as the contracted position.

Optimization

After finished the algorithm and check its correctness by visualization in MeshLab, I try to optimize its efficiency. So first I investigate the breakdown of my initial version algorithm (before optimized): (ratio=0.01, threshold=0.01)

Breakdown Time (s) (Before Optimized)	Horse.obj	Arma.obj	Dragon.obj
Load Mesh	0.15	0.08	0.39
Calculate Q Matrix	0.02	0.01	0.06
Select Valid Pairs	39.69	8.79	191.31
Simplify (Aggregation)	4.42	1.95	24.68
Total Time	44.28	10.84	216.43

We can observe that **"Select Valid Pairs" is the bottleneck** which usually takes `>=80%` of total time.
The reason is that the time complexity of this part is O(n^2), where `n` is the number of vertices.

But we can first sort the vertices by certain coordinate (e.g. by `x`) beforehand. Then we can prune our search space: if we find `v[j].x - v[i].x > threshold`, we can directly break the `for-j` loop according to the monotonicity of `x`.

Evaluation of brute-force and sorting by different coordinates (`x`, `y` and `z`) are as follows (threshold=0.01)

Time of Select Pairs (s)	Horse.obj	Arma.obj	Dragon.obj
Brute-force	39.69	8.79	191.31
Sort by x	1.76	0.45	6.25
Sort by y	1.13	0.45	7.17
Sort by z	1.20	0.48	8.75

We can see that this pruning greatly reduces the runtime and sorting by `x`, `y`, `z` shows similar speed up. After this optimization, "Select Valid Pairs" does not dominate the runtime any more and "Simplify (Aggregation)" takes `>=70%` time. Further optimization can focus on this part.

Breakdown Time (s) (After Optimized)	Dragon.obj
Load Mesh	0.38
Calculate Q Matrix	0.06
Select Valid Pairs	8.77
Simplify (Aggregation)	24.98
Total Time	34.19

Implementation References

- https://github.com/aronarts/MeshSimplification
- https://github.com/xianyuggg/Mesh-Simplification
- https://github.com/granitdula/mesh-simplification