MEPP2: a generic platform for processing 3D meshes and point clouds - Supplementary Materials

Vincent Vidal, Eric Lombardi , Martial Tola , Florent Dupont and Guillaume Lavoué

Université de Lyon, CNRS, LIRIS, Lyon, France

1. Comparing data structures

1.1. Experiment models

The 3D mesh models used for performance comparison in Figures 1 and 2 and in Tables 1, 2, 3, and 4 are presented in Figures 3, 4, 5, and 6. Bimba (resp. monkey, neptune, and happy) model has 8857 vertices (resp. 50503, 249425, and 543652).

1.2. Detailed computations

We ran under the Linux OS the progressive compression and decompression of several 3D meshes, using four different data structures: CGAL Polyhedral Surface, CGAL Surface Mesh, CGAL Linear Cell Complex, and OpenMesh. The algorithm being generically implemented, the same code was used with all data structures. We compared the computation times and the memory footprints[†]. The results, presented in Figures 1 and 2 indicate that when considering the compression (resp. decompression) computation time criterion, OpenMesh is the fastest data structure followed by CGAL Surface Mesh which is 1.3 (resp. 1.8) times slower. Considering the memory footprint criterion, CGAL Surface Mesh has the less memory usage.

Additionally, Figures 1 and 2 indicate that CGAL Polyhedral Surface has significantly lower performances than the other data structures. It is important to note that this is mainly because we use associative property maps with CGAL Polyhedral Surface, whereas we use vector property maps with other data structures. Vector property maps are much faster than associative property maps and have a huge impact on the compression-decompression algorithm speed. We cannot use index-based vector property maps with CGAL Polyhedral Surface because its vertex identifier is not an index, unlike the other data structures.

[†] Memory usage is obtained via the Linux command "grep VmPeak /proc/$PID/status"

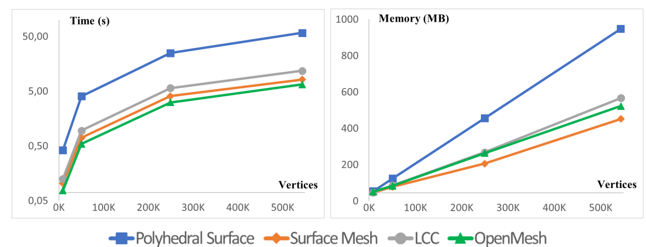

Figure 1: *Comparison of data structures (timing in log scale and memory in linear scale) for the compression of 4 different meshes.*

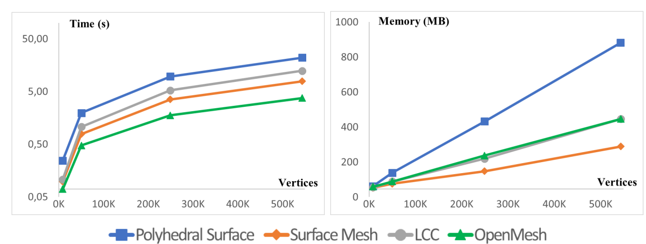

Figure 2: *Comparison of data structures (timing in log scale and memory in linear scale) for the decompression of 4 different meshes.*

	Polyhedral Surf.	Surface Mesh	LCC	OpenMesh
Comp. time (s)	0.42	0.10	0.13	**0.08**
Comp. memory (MB)	53.7	**45.6**	47.5	51.3
Decomp. time (s)	0.24	0.10	0.11	**0.07**
Decomp. memory (MB)	62.3	**52.5**	55.2	59.5

Table 1: *Comparison of data structures for the compression and decompression of the bimba model presented in Figure 3.*

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

	Polyhedral Surf.	Surface Mesh	LCC	OpenMesh
Comp. time (s)	4.02	0.70	0.95	**0.54**
Comp. memory (MB)	123.2	**77.4**	86.0	82.6
Decomp. time (s)	1.96	0.78	1.07	**0.47**
Decomp. memory (MB)	138.3	**75.5**	88.1	88.5

Table 2: *Comparison of data structures for the compression and decompression of the monkey model presented in Figure 4.*

	Polyhedral Surf.	Surface Mesh	LCC	OpenMesh
Comp. time (s)	24.80	4.06	5.68	**3.12**
Comp. memory (MB)	455.6	**205.6**	267.7	262.6
Decomp. time (s)	9.61	3.54	5.24	**1.76**
Decomp. memory (MB)	432.4	**147.0**	219.5	237.1

Table 3: *Comparison of data structures for the compression and decompression of the neptune model presented in Figure 5.*

	Polyhedral Surf.	Surface Mesh	LCC	OpenMesh
Comp. time (s)	57.87	8.14	11.74	**6.62**
Comp. memory (MB)	947.0	**451.7**	566.5	522.8
Decomp. time (s)	21.96	7.82	12.29	**3.77**
Decomp. memory (MB)	881.3	**288.7**	445.6	446.2

Table 4: *Comparison of data structures for the compression and decompression of the happy model presented in Figure 6.*

Figure 3: *bimba*

Figure 4: *monkey*

Figure 5: *neptune*

Figure 6: *happy*

1.3. Complex processing pipelines: full source code

```
#include "FEVV/DataStructures/DataStructures_pcl_point_cloud.h"
#include "FEVV/Wrappings/Graph_traits_pcl_point_cloud.h"
#include "FEVV/Wrappings/Geometry_traits_pcl_point_cloud.h"
#include "FEVV/Wrappings/Graph_properties_pcl_point_cloud.h"
#include "FEVV/Wrappings/properties_pcl_point_cloud.h"

#include "FEVV/Filters/PCL/pcl_point_cloud_reader.hpp"
#include "FEVV/Filters/PCL/pcl_point_cloud_writer.hpp"

#include <pcl/search/impl/search.hpp>
#include <pcl/features/normal_3d.h>
#include <pcl/kdtree/kdtree_flann.h>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/poisson_surface_reconstruction.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/mst_orient_normals.h>

#include "FEVV/Filters/Generic/Manifold/"
          "Compression_Valence/compression_valence.h"

#include "FEVV/DataStructures/DataStructures_cgal_polyhedron_3.h"
#include "FEVV/Wrappings/Geometry_traits_cgal_polyhedron_3.h"
#include "FEVV/Wrappings/properties_polyhedron_3.h"

#include <iostream>

int main(int argc, char *argv[])
{
// parse arguments
if(argc != 2)
  return EXIT_FAILURE;
std::string input_file = argv[1];

// load point cloud with FEVV generic reader
FEVV::PCLPointCloud  pc;
FEVV::PMapsContainer pmaps_bag; // FEVV bag of property maps
FEVV::Filters::read_mesh(input_file, pc, pmaps_bag);

// compute normals with PCL
typedef  FEVV::PCLEnrichedPoint  PointType;
typedef  FEVV::PCLEnrichedPoint  NormalType;

pcl::search::KdTree< PointType >::Ptr kdtree(
    new pcl::search::KdTree< PointType >);
pcl::NormalEstimation< PointType, NormalType > normal_estimator;
normal_estimator.setInputCloud(pc.makeShared());
normal_estimator.setSearchMethod(kdtree);
normal_estimator.setKSearch(18);
normal_estimator.compute(pc);

// copy PCL data structure into CGAL-compliant data structure
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_3 Point;
typedef Kernel::Vector_3 Vector;
typedef std::pair<Point, Vector> Pwn;
std::vector<Pwn> points;
for(auto p : pc)
  points.push_back(Pwn(Point(p.x, p.y, p.z),
                      Vector(p.normal_x, p.normal_y, p.normal_z)));

// ensure normal orientation is consistent with CGAL
CGAL::mst_orient_normals(
  points,
  18 /*nb_neighbors*/,
  CGAL::parameters::point_map(
                      CGAL::First_of_pair_property_map<Pwn>())
          .normal_map(CGAL::Second_of_pair_property_map<Pwn>()));

// reconstruct mesh with CGAL (Poisson method)
double average_spacing =
  CGAL::compute_average_spacing<CGAL::Sequential_tag>(
  points,
  6,
  CGAL::parameters::point_map(
                      CGAL::First_of_pair_property_map<Pwn>()));
CGAL::Polyhedron_3<Kernel> reconstructed_mesh;
```

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

```
CGAL::poisson_surface_reconstruction_delaunay(
      points.begin(), points.end(),
      CGAL::First_of_pair_property_map<Pwn>(),
      CGAL::Second_of_pair_property_map<Pwn>(),
      reconstructed_mesh, average_spacing);

// compress mesh with FEVV Compression Valence filter
bool with_adaptative_quantization = false;
bool with_compression = true;
int quantiz_bits = 10;
int max_vertices = 100;
using VertexColorMap =
    typename FEVV::PMap_traits< FEVV::vertex_color_t,
               CGAL::Polyhedron_3<Kernel> >::pmap_type;
VertexColorMap *v_cm_ptr = nullptr;

// retrive point property map and apply FEVV filter
auto pm = get(boost::vertex_point, reconstructed_mesh);
FEVV::Filters::compression_valence(reconstructed_mesh,
      &pm, v_cm_ptr, "", "cloud_to_mesh.compressed.p3d",
      with_compression, with_adaptative_quantization,
      max_vertices, quantiz_bits);

return 0;
}
```

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.