A Versatile Framework for Research Code Development in Geometry Processing

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Jan Möbius
aus Viersen, Deutschland

Berichter: Universitätsprofessor Dr. Leif Kobbelt
Universitätsprofessor Dr. Torsten Wolfgang Kuhlen

Tag der mündlichen Prüfung: 02.02.2017

Abstract

Geometry processing and rendering are highly active research areas covering a large variety of subtopics. Some of these are the processing of data from laser scanners, modeling, animation, quad-meshing, rendering, and 3D printing. Today, a lot of research is performed on these topics requiring software to implement and test new algorithms or to visualize their results. As most of the software in geometry processing provides similar algorithmic structures and works on common data types like meshes or points, a common framework would greatly reduce the amount of redundant code and increase the development speed of new algorithms. In this thesis, we develop a versatile software framework for geometry processing and rendering. We study the common workflows and use cases in several geometry processing pipelines. Besides we take a look at existing software frameworks and collect requirements typical user groups put on these systems. From this analysis we derive a set of software specifications for a general geometry processing framework which can be used in a large variety of use cases. We describe how these software specifications have been implemented in our open source framework called OpenFlipper and provide a software solution for all components of such a framework. These components include the build system, abstractions to operating system and hardware, and a highly flexible plug-in architecture to achieve a modular structure. Furthermore, we provide plug-ins for geometric data types, user interface creation and management, as well as software interfaces for various aspects of the framework, including software automation, rendering and many more. Additionally, we show how software testing via continuous integration can assist by significantly improving the code quality and furthermore allows to constantly deliver ready-to-use software packages. We also cover the project management

and infrastructure which is required to successfully create and maintain such a large and fast-growing framework. In the end we present several use cases to show how OpenFlipper performed in practical courses and how it was used to simplify the development of research applications.

Contents

1 Introduction

Computer graphics and geometry processing are large application fields covering a vast variety of different subtopics. The results and tools developed in this research area are used in various fields of application. Among these are for example medical care (e.g. analyzing data from imaging systems), automotive industry (designing cars in CAD programs), gaming industry, and film industry. All these application areas offer a lot of interesting research topics.

Looking at a general geometry processing workflow, processing usually starts with retrieving data from various different input sources such as laser scanners, photogrammetry, Computer Tomography (CT), Magnetic Resonance Imaging (MRI), procedural generation or manual modeling via CAD software.

The acquired raw data can be of various types such as points, splats, triangle or polygonal meshes, skeletons, volumes and many others. This input data typically needs to be processed to conform to possible target requirements such as precision, numerical constraints or visual quality. Furthermore, it might be necessary to convert between the data types for different fields of application or algorithm constraints.

After processing, the final data can be used for simulations or gets visualized on a screen. The visualization can suit different purposes. For example, on mechanical parts specific features such as curvature might be highlighted for easier analysis by an engineer. He can then modify the part to suit e.g. production requirements or customer demands. Furthermore, the visualization can be used for entertainment such as animations in movies or computer games which gained increasing popularity in the last years.

To simplify development and research in all of these application areas, a versatile framework has been designed and developed in this thesis. To cope

with the large number of possible use cases, the framework is highly modular and can be easily adapted to suit the needs of researchers, developers and end-users. Almost all parts can be replaced to allow focused research on a single subject while exploiting the benefit of existing components, significantly reducing the coding effort and therefore speeding up development.

This thesis is divided into six chapters.

- Analysis: The first chapter will focus on analyzing workflows, applications and users in the field of geometry processing and computer graphics software. We use several typical workflows to find common building blocks which have to be covered by the framework. Furthermore, the user requirements are analyzed to provide tools for different personal scopes of application for the framework. Additionally, we will take a look at existing frameworks.

- Software Requirements: The second chapter will focus on extracting requirements and possible design strategies for the framework that can be identified from the information gained by evaluating the analysis in first chapter.

- Implementation: This chapter presents how the given software requirements are implemented in the OpenFlipper framework. It covers the overall structure of the framework, basic build system implementation, rendering infrastructure and many other aspects of the software. Some parts of this chapter are published in [59] and [60].

- Testing: An important task for such a complex framework is quality assurance. Users and Developers strongly benefit from a framework that is well tested and provides functions to easily identify bugs or errors in the code. This chapter will focus on the implementation of software testing in the framework from simple unit tests over smoke tests to full integration tests. Some parts of this chapter have been published in [60].

- Project Management: Next to the software requirements and the implementational details the project management is also an integral part of the development. In this chapter, we will cover the source code management, package creation and software development strategies.
- Case Studies: The last chapter will show some results and how the framework performed in several use cases. Some of the use cases have been published in [59]

Parts of this thesis have been published in [59] and [60].

2 Analysis

In this chapter we analyze the software requirements for geometry processing software. The chapter is divided into three sections.

Section 2.1 studies several common geometry processing workflows to derive software and framework requirements. The case studies range from acquiring data (e.g. laser scanning), over processing (surface modeling, quad meshing and simulation), to output generation (qnimations, 3D printing and rendering). Furthermore the involved data types of the workflows are extracted.

Section 2.2 deals with the different user groups of a geometry processing framework and which demands they put on the software. We identified six basic groups of software users for such a framework.

Section 2.3 will deal with existing software in this area and how it is developed.

2.1 Analysis of Geometry Processing Workflows and Common Use Cases

This section provides various examples of geometry processing algorithms. The majority of 3D data is either acquired by laser scanners or manually created by CAD Software. When using the raw input data from these inputs, the models usually have to be repaired and modified for better processing. Afterwards the models might get modified to match certain output requirements (e.g. for 3D printers, simulations, or efficient rendering). As a last step, the models get exported to the target format, which might be 3D meshes, point clouds or machine code for controlling 3D printers.

In the following sections we analyze the prerequisites of the different algorithms with respect to a geometry processing system to identify the semantic components which are common for these workflows. Furthermore, we collect required data types, interaction schemes and usage patterns which provide the basis to define the structure and components of the final framework.

2.1.1 Surface Data Acquisition via Laser Scanning

A common application in geometry processing is the acquisition of new data from real world objects. There exist several ways to create new geometric models from real objects. Next to re-modeling objects by hand, e.g. in a CAD/CAM software, the data could also be generated by transferring real life structures into a virtual representation by technical devices. This could be done by scanning them with infrared depth cameras, time of flight cameras or by using a laser scanner such as e.g. the Minolta laser scanner shown in Figure 2.1.

This scanner has a laser emitter at he bottom. The emitted laser point is moved across the scanned object in scan lines via a rotating mirror as depicted in Figure
reffig:laserscanning. A camera or detector is mounted in a fixed position at the top, relative to the laser emitter and records the positions and for modern scanners also the time of flight of the laser between the object and back to the scanner. Due to the fixed alignment between the laser emitter and the camera, it is possible to calculate the distance (yielding the depth) from the scanner to the objects surface. For each laser direction a depth is recorded and stored. As the scanner traverses a simple grid pattern with the laser, the resulting depth measurement points can be easily connected to a polygonal surface mesh which is the output of the device.

As the surface is usually not completely visible from one scanner position, objects often have to be scanned several times from different viewpoints, resulting in a set of unconnected polygonal meshes where each mesh is defined in its own local coordinate system. An additional processing step is required to combine these separate meshes into one complete representation.

Figure 2.1: Minolta VI-900 laser scanner. Top opening contains the camera. Bottom opening encloses a laser emitter.

To get color information, the scanner has an additional camera that takes a color image for each scan used for texturing. As this camera has a fixed position relative to the laser emitter and detector as well, it is easy to calculate the color for each point of the scan from the images.

To get a fully connected model of the real world object, the separate scans have to be transformed from their local coordinate system into a common global coordinate system. This involves several steps:

1. Cleanup scans (Remove outliers, remove possible background noise)
2. Register scans (Align scans in 3D)
3. Merge scans to one model (Create one connected surface mesh)

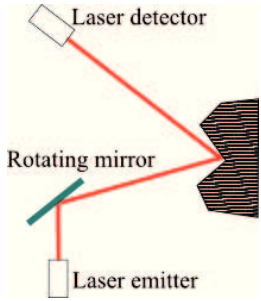


Figure 2.2: Laser scanner construction: The emitted laser is moved via a rotating mirror across the object on the right. The laser reflection is received by a laser detector. Based on the receiving angle, the distance from the scanner to the objects surface can be computed.

4. Map color camera images onto surface to get a colored object
5. Fill holes (Close holes which have not been covered by the scanner as they were not visible from any of the scan positions)
6. Smoothing (Remove scanner noise from the surface)
7. Remesh (Retriangulate to match a given target complexity)
8. Final cleanup (Remove unwanted artifacts or parts of the scanned object)

For a geometry processing framework, there are several components that are required for such a scenario. First of all, the input scans have to be read into the internal data representation. Most of the current scanners support the Stereo Litography File format (STL). Therefore, serialization and a reader for this format are required. Of course a writer for the final result is required as well. As there are several file formats, the readers and writers should be available as separate components to allow for easy extension of the framework to provide a large set of formats to the users and possibly increase the acceptance of the framework.

Several of the processing steps can be automated and use standard geometry processing algorithms. The cleanup may try to detect the biggest components and remove the rest of the scan to remove possible outliers. The connection, hole filling, smoothing and remeshing steps may run at least semi-automatic. The only step which usually requires a larger amount of user interaction is the scan registration. In this step, the user has to click on two meshes to mark correspondences between them. The system then uses these correspondences to compute an initial alignment of the two meshes and then refines it e.g. using an ICP [9] algorithm.

This kind of user interaction is a repeating task in geometry processing algorithms. The process of clicking on an object and retrieving the object (and possibly the primitive e.g. a single triangle of the object) id and position is called picking. It is required for all 3D user interactions inside a graphical user interface. Implementing it can be done in several ways, but as it usually requires to render the whole scene in a special way, it can be very time-consuming. As the user should not wait a long time for a reaction of the software during each interaction step, the picking must implemented fast, reliably and efficient. We describe our implementation in Section 4.6.

Next to picking, a framework has to implement a way to deal with several objects at the same time. In this case several meshes (one per scan) have to be handled at the same time and to avoid cluttering of the view, it has to be possible to disable their rendering on demand.

The cleanup stages might also involve some user interaction. Most of this interaction requires the user to select different parts and primitives of a mesh. To simplify this kind of interaction, a set of selection metaphors should be provided by the framework e.g. selecting the biggest component of a mesh or a Surface Lasso selection, where all primitives inside a polygon that is drawn by the user on a surface get selected. A more detailed description of the required selection metaphors is given in Section 4.7.

In the presented processing pipeline several steps are involved. Most of them have a set of parameters, that may have to be altered to yield a high quality result. Therefore the user of the framework might try to test different values

and select the parameter set with the best results. This requires some kind of backup system to allow to automatically backup the intermediate results and allow the user to switch back and forth until the result is what the user wanted to achieve. This leads to the requirement that there have to be undo and redo operations available.

After processing of the input data through this pipeline, the result should usually be a watertight surface mesh, which can be used as an input for a 3D printer. Most of the printers implement the required preprocessing (e.g. slicing) of the input data in their own software. Nevertheless, it would be only an extension of the pipeline to do this processing inside the framework as well. We will describe that pipeline extension in Section 2.1.4.

2.1.2 Point Cloud Acquisition via Laser Scanning

The second example uses a long range laser scanner to acquire data instead of the short range scanner in the previous section. One example for such a long range laser scanner is the Riegl VZ-400 scanner shown in Figure 2.3.

Its basic functionality is the same like that of the Minolta laser scanner described in Section 2.1.1. It uses a laser emitter and detector to measure distances to objects. Due to the long range of the scanner, the optical resolution of the usually small integrated cameras would not be sufficient to acquire accurate color information. Therefore, a high resolution DSLR camera with a high quality lens (Nikon D800 with 14mm ultra-wide lens) is mounted on top of the scanner to acquire the pictures required to generate a colored model.

Furthermore, the output of the scanner is no longer a surface patch but a series of colored points in 3D. This requires a different type of data inside the framework. This new data type has to efficiently handle a set of unconnected points, with a set of textures which are used to color the points of each scan position. Additionally these points are usually rendered as so called splats or if a higher visual quality is required, as textured splats which requires new rendering algorithms and a different rendering strategy than with polygonal meshes. Figure 2.4 shows several splat rendering techniques.

Figure 2.3: Riegl VZ-400 long-range laser scanner with Nikon D800 with 14mm ultra-wide lens mounted on top.

Most of the selection metaphors of the previous case can be reused for points as well, although the picking is different. In the previous case the picking could be implemented by rendering the surface consisting of points, edges, and triangles. For point clouds, only points can be selected. This means that the picking depends on the type of available primitives in the current scene objects. The framework should therefore have a picking implementation that is associated with the object types.

The laser scanner data of a long range scanner might be used for different applications. One is to extract a closed 3D surface of these scans. This would require the following processing pipeline:

1. Cleanup scans (Remove outliers, remove possible background noise)

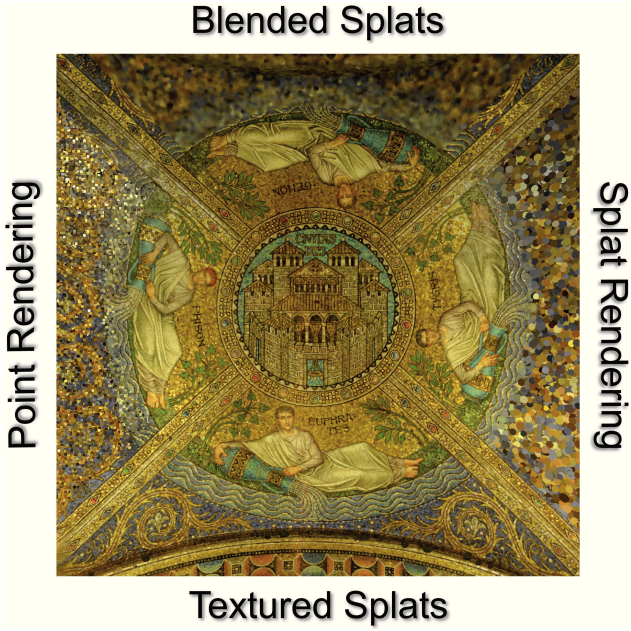


Figure 2.4: Splat rendering techniques. Left: Points rendered with color. Right: Splats rendered with color. Top: Splats rendered with color and blended into each other. Bottom: Splats rendered with textures taken from images.

2. Register scans (Align scans in 3D)

3. Surface Extraction (E.g. Khadzan Poisson reconstruction [48])

4. Mesh-post processing (defect removal, remeshing, smoothing, etc.)

The pipeline is similar to the one with the short range laser scanner, but the steps up to the surface extraction are performed on point clouds instead of surface patches. Some algorithms might work on several types of input data. A geometry processing framework should have an interface that can be used

by the algorithms to detect the type of input data and react accordingly. One example for this would be the ICP [9] algorithm, which can be implemented to use only points as its input. For triangle or polygonal meshes only a small interface change is required for the algorithm to retrieve the points from the mesh and pass them through the point based algorithm. The point clouds can be directly used by the ICP.

The rest of the surface reconstruction pipeline is identical to the short range scanner pipeline. The mesh post processing algorithms are identical (up to some parameters) and can be reused.

2.1.3 Processing Meshes from CAD/CAM Software

The following workflow is a simple example of what is required to process input from CAD/CAM Software. This kind of software allows users to easily model objects in 3D with a simple user interface. However, depending on the users experience in creating 3D models, the output is usually of average or bad quality regarding mesh processing. Such software tends to allow the designer to create unconnected objects with self-intersecting components. These artifacts usually do not pose an issue on renderings as they are invisible. But for mesh processing algorithms defects like holes, self-intersections, triangles with bad shapes and other inconsistencies cause severe problems to process the data. To use such models in 3D printers or in simulation software, they have to be fixed and optimized. This requires several algorithms and operations which already have been described in the previous workflows. These include capabilities to load the model and perform selections to guide certain algorithms, the mesh processing algorithms themselves and an routine to store the output. The mesh processing might involve remeshing, smoothing operations or even Boolean operations on single or multiple meshes similar to [23]. These operations can be used to remove defects of the mesh and get a triangulation that is appropriate for the final use case of the output surface model.

2.1.4 3D Printing

Another use case which has become quite common is to create 3D models by either scanning or modeling them and then printing a real world 3D model of it using a 3D printer. The input for 3D printing should be a watertight 3D mesh of decent quality generated by one of the previous workflows to simplify processing the data for the 3D printer.

To print a model, a 3D printer needs machine instructions, controlling movement of the print head and build plate, adjusting temperatures and fans, and controlling the material flow. Usually 3D printers come with their own software which gets a 3D mesh as input. This input is required to be of fairly good quality to ensure that the algorithm detects which parts of the model are inside or outside. If the mesh has holes or inconsistencies the software might create wrong tool paths from the data. The software then generates the machine code from that data. For instance the Ultimaker 2 (shown in Figure 2.5) comes with a free software named Cura [2] shown in Figure 2.6.

Since the printer software takes only a model file as input, all information of the previous modeling and processing steps is lost, as the printer software does not know about these steps when the preprocessed mesh is opened from a file. Therefore, if the machine code generation can be integrated into the framework as well, the system would have the advantage to access previous information on modeling steps to improve the machine code and allow the user to directly see the impact of previous processing steps on the result. Figure 2.7 shows the steps of a 3D printing pipeline.

Looking at the steps required to convert a 3D mesh into machine code for a 3D printer, we can derive additional requirements for a software framework for processing data for 3D printing.

A 3D printer prints a model slice by slice. Therefore, the model will be cut into layers with the thickness of the printers vertical resolution. This slicing algorithm intersects the input mesh with a set of planes. For each plane the raw output will be a set of polygonal lines at the intersection of the surface of the model with the plane. The generated polygonal lines will be one part of

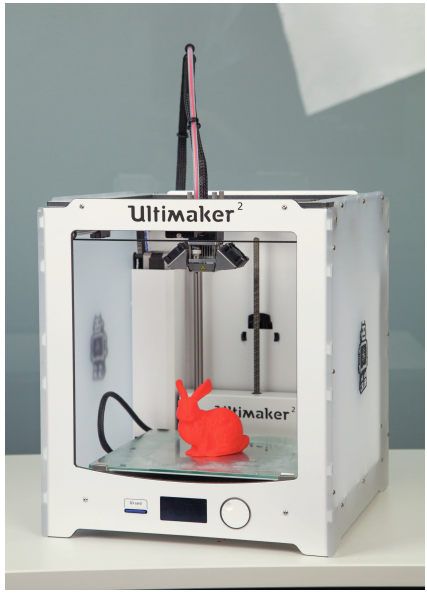


Figure 2.5: Ultimaker 2 3D printer with a Stanford Bunny printed in red.

the tool path for the print head. Next to these possibly unconnected tool paths to print the surface, additional movements are required to move the print head between multiple poly line segments or to another layer as well as code for the material feeder. The print head movement should of course be minimized to reduce printing time but maintain a predefined quality.

To do these operations, a framework would require the data types for polygonal lines and planes. Furthermore the machine code needs to be visualized to allow the user to identify problematic regions (strong overhangs, mesh defects) and solve them by adjusting parameters of the algorithm or modify the input mesh. In Section 7.2.1 we show the results of a 3D printing practical course which we realized using the framework.

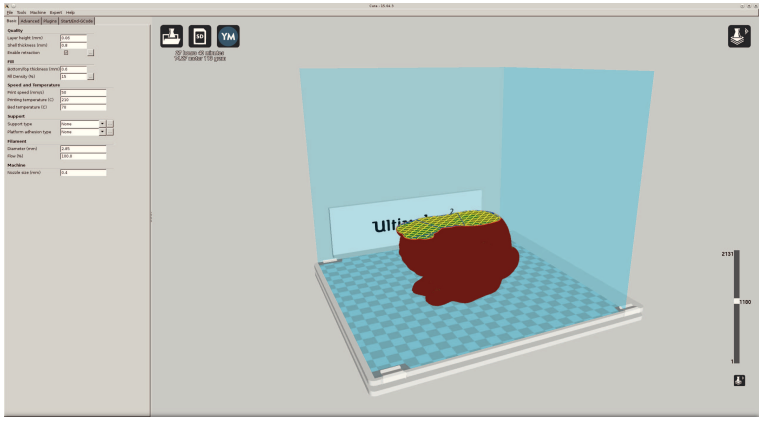


Figure 2.6: Screenshot of the 3D printing software Cura, shipped with the Ultimaker 2.

Figure 2.7: 3D printing pipeline: Original model, scanned model with holes, remeshed model in wireframe and shaded, sliced model, printing, printed model.

2.1.5 Surface Modeling

After a model has been acquired by a Laser scanner or manually created via CAD/CAM software, it might require additional modifications. This means for example that the user might want to select regions of the surface and deform them in various ways.

The simplest form of modeling would be to select a number of vertices or faces and then move them by a user defined amount. As before, this requires that the framework provides selection metaphors and input metaphors to define the movement.

A more sophisticated modeling metaphor is Laplacian modeling. An overview of such modeling techniques can be found in the excellent survey [18]. In this metaphor the user usually defines three kinds of regions. One is the area that is kept fix. The second region will be the part of the model that is deformed and the third is a handle inside the deformable region which will be moved by the user providing the direction and amount of deformation. After the user moved the handle, the deformable area is deformed by solving a large system of equations which distributes the deformation across the deformable area while using the fixed and handle regions as boundary constraints. An example of the editing procedure is shown in Figure 2.8. The blue area is the deformable region while the green part marks the handle which is kept rigid and moved by the user with the depicted manipulator. The gray component is kept fix.

The code for solving large mathematical problems which are quite common in geometry processing, can be very complex. To achieve high performance and robust algorithms, the code has to be of high quality and optimized to several processors and operating systems. Therefore, for solving such equation systems, usually additional highly optimized mathematical program libraries have to be used which should be easily usable from within the framework.

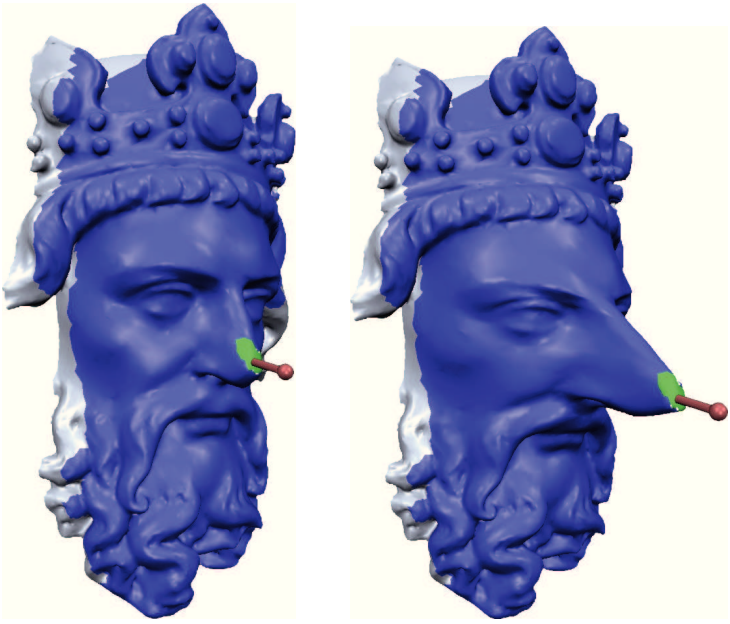


Figure 2.8: Left: Original Model. Right: Deformed Model. The gray area is kept fix while the green area is rigidly moved by the user via the manipulator. The blue area is deformed by the system.

2.1.6 Simulation

One advantage of optimized geometric models is that they can be directly used to simulate and optimize physical processes. The workflow in such cases is usually that a model is created and then tested with respect to physical parameters or just visualized. For several simulation algorithms e.g. based on finite element methods, the model requires a high surface quality (equilateral triangles, triangles of equal size,...). So the model might have to be preprocessed to optimize the quality like in previous workflows.

The optimized input model can then be used in a simulation. One possible aim for a simulation is to optimize the shape of an object with respect to aerodynamic or fluidal resistance. Therefore, a simulation step is performed and afterwards the object is deformed based on the simulation results, to yield a better performance with respect to predefined parameters. The resulting model is then fed back into the simulation. This process is iterated until the results match the requirements of the user.

For a large set of these simulation and update steps, the modifications can be automated. The framework should therefore provide interfaces to run algorithms automated, without user interaction or if user input is required, the system should be able to ask the user for additional input and use that for further processing.

An example application for simulation is described in Section 7.3.

2.1.7 Animation

A different geometry processing workflow is used to generate animated 3D meshes for computer games or video games. To animate meshes, two inputs are required.

The first is a surface mesh which is usually equipped with textures and surface maps. This mesh can be a scanned body, e.g. of a human or a manually modeled mesh of a fictional character. The meshes used for animation are most of the time quadrilateral meshes, as these provide a better and more intuitive structure.

Secondly, a skeleton inside the surface mesh is required. The skeleton is used to reduce the complexity of the animation, as only the joints of the skeleton need to be moved and not every single vertex of the surface mesh, resulting in a significant reduction of degrees of freedom for the animation.

The process of connecting a skeleton to a surface is called skinning. There are several algorithms to define the corresponding vertex weights which define how much each vertex of the surface mesh is influenced by the movement of each bone of the skeleton. The simplest version is to take each vertex and transform

it with the transformation matrix of every bone in the skeleton. This results in a set of possible positions after deformation. These positions are then blended linearly with the weights which the vertex has for every bone, resulting in the final position linearly blended from all possible positions.

If a vertex would be connected to only one bone (weight for one bone is 1 while all other weights are 0), this means that it is fully transformed with the transformation of that bone and not influenced by the others. A more detailed description of these linear blend skinning can be found in [55]. Although linear blend skinning is simple and efficient, it has the drawback, that it can yield self intersections of the animated mesh. There are several other algorithms developed to improve the final mesh quality, for example the Dual Quaternion [47]. An example rendering of a skeleton with an attached skin is shown in Figure 2.9

Figure 2.9: Screenshot of a 3D surface mesh of an Armadillo with an attached skeleton inside it. The surface can be deformed by moving the skeleton in its interior.

Beside creating skeletons by hand there are also algorithms to extract the skeletons automatically. One example for such an algorithm is the skeleton extraction by mesh contraction [7]. The surface mesh is shrunk by applying several smoothing iterations, until it collapses to the skeleton. After some automatic cleanup, removing noise and unnecessary joints, a skeleton is generated that can be used for the skinning process.

Most of these algorithms are only used for generating an initial solution while the final skinning is optimized manually to achieve best visual results.

For the animation itself, animation data needs to be captured. This can be achieved either by motion tracking systems e.g. ARTrack or by Microsoft Kinect. The tracking data can then be used to move the joints of the skeleton while the rendering system then has to move the surface mesh according to the skeleton. The full workflow for the animation is:

1. Acquire surface mesh
2. Optimize surface mesh for the animation (remeshing, possibly quad meshing)
3. Extract or hand-model skeleton
4. Skinning (connect skeleton and surface mesh)
5. Acquire motion
6. Play motion on skeleton and render result

This workflow requires a skeleton data type inside the framework that can be visualized, edited, and animated. The animation data needs to be managed and stored by the framework.

The animation of the surface mesh is usually done by a special vertex shader deforming the surface while rendering. This is possible as the linear blending consists of a set of matrix operations which can be done very efficiently inside a shader. Furthermore, it reduces the amount of computation required on the CPU for the animation and allows for real time animations using the GPU.

Setting up additional shaders requires a modified rendering code for this kind of application. Furthermore the framework needs a system that can playback animations interactively to allow the user to analyze the results and edit them in real time. Obviously the workflow has several possibilities to change the applied algorithms e.g. the skeleton extraction, skeleton editing, different skinning weights and techniques or the rendering of the animation. This demand for flexibility must be taken into account by a geometry processing framework.

2.1.8 Quad Meshing

Currently the most widespread representation for surfaces in geometry processing are triangle meshes. Today's graphic cards can directly work on triangular input data and provide very efficient algorithms implemented directly in hardware for rendering of this kind of primitives. However there has been a lot of attention on the problem of converting unstructured triangle meshes into quadrilateral meshes. Quad meshes have several advantages over triangle meshes.

First of all, they can be aligned to the surface more intuitively. On a surface we have two principal curvature directions. A quad can be be perfectly aligned to these with its two directions, while a triangle will always have the third edge which will be misaligned if two edges of the triangle are aligned.

Secondly, subdivision surfaces use quad-dominant control meshes. Therefore, it is easy to model an object at a coarser level and later use or render a high resolution subdivision surface.

In simulations, quad meshes are usually preferred over triangular meshes, as they provide a lower approximation error and better numerical stability.

Beside these properties several other advantage of quad or quad-dominant meshes promote the development of algorithms converting surfaces represented as triangle meshes into quad meshes. A comprehensive state of the art report on quad meshing can be found in [10] and [13].

An example for a Quad Meshing pipeline would require the following steps:

1. Compute principal curvatures

2. Compute orientation field (E.g. principal curvature directions can be used as a basis)
3. Compute sizing field
4. Compute a global parametrization under the constraints of the first two steps
5. Extract Quad Mesh

The input in this workflow is a triangular mesh which can be loaded from a file. The first two steps involve a computation of several properties on the input data. These computed values are then passed to the third step, where a system of equations needs to be solved. Using the computed parametrization, a quad mesh can be extracted which is then the output of the algorithm.

2.1.9 Geometry processing pipelines

Looking at the previous workflows, it becomes obvious that all of them can be viewed as geometry processing pipelines, where one or more input objects are combined or processed by several algorithms producing one or more outputs. This is very similar to video or image processing software where the input is processed by several filters producing a new output. Each geometry processing algorithm can be interpreted as such a filter with multiple inputs and outputs. The inputs can be meshes, skeletons or other data to be processed (like the principal curvature or sizing fields of the quad meshing) but also user interaction defining areas of interest or parameters for an algorithm. Video processing software implements the filters as blocks in graphical user interfaces, which can be connected to algorithm flows. This would also be an easy to use interface for a geometry processing framework.

The general workflow for mesh processing is to start with a mesh and apply one filter after the other (e.g. remesher, smoother or decimater) or in parallel with different parameters to optimize a mesh for a predefined use case. After finding the perfect setup of the pipeline, the whole processing pipeline can be

applied to a set of inputs without further user interaction. Even if additional user input is required, the pipeline can run non-interactive parts unsupervised and wait for interaction if required, yielding semi-automatic geometry processing. The so called batch processing where we apply the same processing chain to multiple inputs should be an integral part of a framework to support such use cases.

2.2 Analysis based on User Requirements

In this section we analyze the software requirements based on the distinct user groups of geometry processing software. From existing software we see that there are several user groups involved.

In a commercial setting, we have the software developers who write and of course use the software at the beginning. Usually, there is an additional layer of software testers who test the software and give feedback to the developers to optimize and polish the software. And finally the end users who buy the software to use it in their company or privately. In this setting we can therefore identify the developers, software testers, and private and commercial end users as possible user groups.

The university or research setup is different from the previous commercial point of view. The largest user group are the researchers who develop and evaluate new algorithms. The algorithms are implemented, tested and compared by researchers to existing algorithms. The final goal is to publish the research results on conferences and later on push the algorithms to a commercial usage. Furthermore students can use the software framework to implement algorithms in their courses and in turn get practical experience. The results are then reviewed by teachers. For this setup the possible user groups are researchers, teachers and students.

In the following sections, we will analyze the software requirements based on the identified user groups.

2.2.1 Developers

The first group which uses and works on a software project are the developers. To ensure that a framework gets used to create new applications, it must be easy and most importantly intuitive for the developers to write the first code based on it. This requires a clean structure of interfaces, a flexible and mostly automatic build system and a well written documentation for all components of a framework.

Furthermore it is important that it contains a flexible set of building blocks that can be reused to minimize code redundancy and speed up development. Repeating tasks such as finding required libraries or configuring for different platforms should be automated as well as to allow the use of the software on a large set of platforms and operating systems. To avoid licensing issues the framework should use a free license. A common argument against using an existing framework is a too restrictive, expensive or just too complex licensing.

2.2.2 Software Testers

After the developers created a software, it should be tested for quality assurance and possibly adapting the software for additional input data which does not work in the first place. The basic algorithm tests can be mostly automated, whereas the user interface often has to be tested manually to discover possible workflow improvements. Especially this part should be carried out by someone other than the developers themselves, as they know the software very well and therefore can't tell, if the interface is really intuitive to an end-user. For software testing, the framework should contain a tool set to simplify the testing process (especially the part that can run fully automatic such as smoke and unit tests) and also mechanisms to identify the source of errors for easier debugging. This can significantly improve the usability of the framework and keep up a high software quality, increasing the acceptance of a framework.

2.2.3 Researchers

The main goal of a researcher is to develop new ideas and algorithms which should be ideally published in the end. This process involves developing a new algorithm, implementing it in software for evaluation and possibly compare it to existing algorithms. This implementation should be done with minimal effort to allow focusing on the research topic and keep the count of possible error sources low. Therefore, from a research point of view, repeating tasks and operations of existing algorithms have to be re-used to avoid distraction by redundant and unnecessary implementation work. Furthermore, reused code has been executed and tested several times, resulting in a higher code quality and therefore a lower probability of errors.

In the geometry processing setting, these repetitive tasks range from input and output of data (e.g. 3D models or polygonal lines), over selection, filtering and modification of data, to the final rendering of results and data export. As there are different workflows (as seen in Section 2.1), the software framework must be very flexible, which can only be achieved in a highly modularized system, where each module can be easily replaced.

Researchers tend to write code of lower quality when approaching deadlines. After an algorithm is published, the implementation has to be reviewed and possibly rewritten. But the more code has been used from existing parts of the framework, the less effort is necessary to get the algorithm into an optimal state.

After cleanup the code can be used by other researchers or even directly by end users, if the proposed algorithm should be used commercially. The original code can be kept as a reference implementation to document the development process or to write tests comparing the result of the cleaned up code to the original version.

An additional requirement is of course that researchers have to analyze and compare their algorithms to existing approaches to justify their work. This means that the algorithms should run in the same ecosystem (software frame-

work, hardware, ...) to avoid unknown side effects due to implementation details which might distort the results, e.g. cause wrong timings.

Furthermore it would be convenient if two algorithms are implemented in the same framework to run them one after the other and directly compare their results inside the same application. For example two algorithms extracting a surface from a point cloud (e.g. Poisson reconstruction [48]) generating two surfaces which can be compared in the software with different metrics reaching from simple vertex or face counts to distance metrics like the Hausdorff [50] distance.

2.2.4 Teaching

From a teaching perspective we can differentiate between teaching somebody to use the framework itself or to teach and learn algorithms that can be used within the framework.

For teaching somebody to use a framework it is more convenient, if it can be understood step by step. This means that it is not necessary to know everything about the framework at once to use it but start with small high level components and going down to the low level components step by step. This significantly reduces the training period required to use the framework and therefore improves the acceptance of the software.

For teaching algorithms, a software framework requires to be very flexible as well. The framework must be adapted to the topic of the current lecture or practical. We will pick up two use cases to show the requirements. For all use cases the software should provide easy compilation.

It is essential for a framework to be usable on as many platforms as possible, giving the students freedom to choose a platform to work on. This requires some amount of abstraction of the operating system which can be usually achieved using existing build systems and user interface libraries which are available for all target platforms.

Furthermore, the build process should be as fast as possible. This adds the requirement on the framework that a package can be created, which only

includes the components necessary for the current lecture. This ensures that no unnecessary code will be shipped to and compiled by the students.

One example use case is the implementation of geometry processing methods such as mesh decimation [52] or subdivision (e.g. [51]) algorithms. This setup does not require the students to deal with implementing loading meshes, selection or rendering. The required code can be shipped in a package and the students only have to work on the mesh structure itself, implementing only the algorithm. Therefore, there is no overhead of learning how to render geometry or implement selection of primitives which can then be part of further exercises providing easy step by step learning.

Another lecture example deals with the rendering of geometry. The framework should only be equipped with everything but the rendering components which can then be implemented in exercises by the students.

From a teachers point of view, especially when correcting student exercises, it is important that the code of the students is easy to understand. This usually requires restricting the amount and style of code the students can write by certain constraints, such as predefined variables or small code snippets which have to be used. A significant amount of these constraints are provided by the framework, as the inputs and outputs are well defined.

Furthermore, the exercises should be handed out one by one. These requirements are also best fulfilled by a plug-in based system. Each exercise can be handed out as a separate plug-in. They contain only the relevant code for the current exercise and provide a code skeleton which will be used by the students. After each exercise has been corrected, the solution can be provided as the final plug-in. This also allows for building successive exercises on top of already solved ones without having to rely on the original student solutions. To give some hints about the results of the exercise, it should also be possible to provide a binary version of the exercise. The students can then compare the results of their own implementation with the results of the binary version without prematurely publishing the solution.

2.2.5 Commercial

Commercial development is very different to the previous use cases. For a company it is usually very important to keep the code private. Competitors on the market should not be able to get any insight into the companies algorithms and therefore their know-how. Furthermore, some kind of copy protection or licensing system is also desired to control the deployment of the software or its components. Looking at today's applications, it also gets increasingly important, to publish the software base for free while the actual algorithms or components are sold separately e.g. via in-app purchases.

To reduce coding overhead it is an advantage to use a highly modular system as well. Each component can go through its own testing and quality assurance process. After each component has been tested through unit tests, integration tests can be performed to make sure that all components work together and no side effects occur. Furthermore, in a production software setting it is important to add regression tests to ensure that bugs are actually fixed and were not reintroduced during a product life cycle.

For the final product, this ensures that each part works correctly. Of course the final system still needs to be tested by a human to ensure that the software is working as expected as not all software faults and design errors can be detected automatically. Like explained before, this holds especially for the user interface which is complicated to test automatically. But automating most of the testing process still significantly reduces the costs for this process to a minimum.

The commercial setting therefore usually focuses on efficiency (reducing development and testing costs), quality assurance and a wide range of potential customers.

2.2.6 End-Users

From the end users point of view the most important aspect is that the software is usable for their purpose. The software should work as expected and should not contain any bugs.

Of course it is a significant plus, if the user can stick to his preferred platform and operating system and if a license is required, it should be easy to acquire. So the framework should provide the same look and feel on all supported platforms.

For a good user experience it is also required that the interface is intuitive (and possibly even adaptive) and has a steep learning curve. In a framework this can be achieved by enabling the core of the framework to control and sort the user interface. Common GUI elements and workflows shared by all plug-ins ensure that the interaction metaphors are common to all components of the software which make them easier to be understood by the users.

It is also a plus, if the user can adapt the framework to his needs. For example, if a user rarely uses some components, he should be able to hide them or even group components together for different workflows.

2.3 Analysis of Existing Software

In this section, we will analyze some existing software projects in the field of geometry processing. We will start with the software and utilities used in our department when we started the development of the framework. Then we will take a look at a framework called Meshlab [28] that has been developed parallel to our framework. Furthermore, we will analyze graphite which is a research toolkit for 3D modeling. At the end of this section we will take a brief look at commercial software.

2.3.1 Department Software

At our department there existed various different software packages and tools for geometry processing. Most of them were stand alone programs focusing on a small subset of geometry processing functions usually emerging from a research topic of a research assistant. The simplest set of programs were command line tools e.g. for subdivision, decimation, or mesh smoothing.

Beside these command line tools there were tools with graphical user interfaces which also focused on some aspects of geometry processing e.g. rendering.

Furthermore, there existed one larger application called Flipper which was a commercial application, bundling some of these algorithms in one user interface.

The following subsection describes a few example applications and how these projects would benefit from a common framework.

2.3.2 Standalone test applications

The first group of applications have been the standalone applications. These applications focused usually on the research of one algorithm.

One example is a subdivider [51] utility which takes a triangle mesh as its input and outputs a subdivided mesh with a higher polygon count. This setup usually requires the tool to load the input mesh from a file, do the actual processing on an internal data structure and then write the result back to a specific file format.

A second example is a mesh smoother. The algorithm relocates the vertices of a mesh to get a smoother surface and outputs that surface to a file.

Some of these applications implement their own file readers and writers, although they worked on the same internal data structures (OpenMesh [17]), wasting time on reimplementing the input and output code.

The same applies to standalone applications which have a graphical user interface. E.g. the rendering code or mesh selection usually have been reimplemented and not shared between the applications. Even if code has been shared between applications, it was modified to suit the needs of the current developer resulting in very similar implementations of the same operation. This in turn caused code redundancy and a possibly confusing situation as the different code versions have different capabilities. This not only has the drawback of wasted developer resources but also reduces the amount of testing applied to such a code basis and therefore a less than optimal code quality.

Another major drawback of such standalone applications is the lack of possible interaction between them. If a mesh should be first smoothed and then subdivided, the mesh has to be loaded, smoothed, then written to a file which is loaded by the next application. So there is always an error prone and slow load

save process between the applications where usually no additional information other then the mesh can be handed over to the next application. Additional information which was available before the smoothing operation might have been useful to guide the second operation, but is lost for the subdivision process.

2.3.3 Flipper

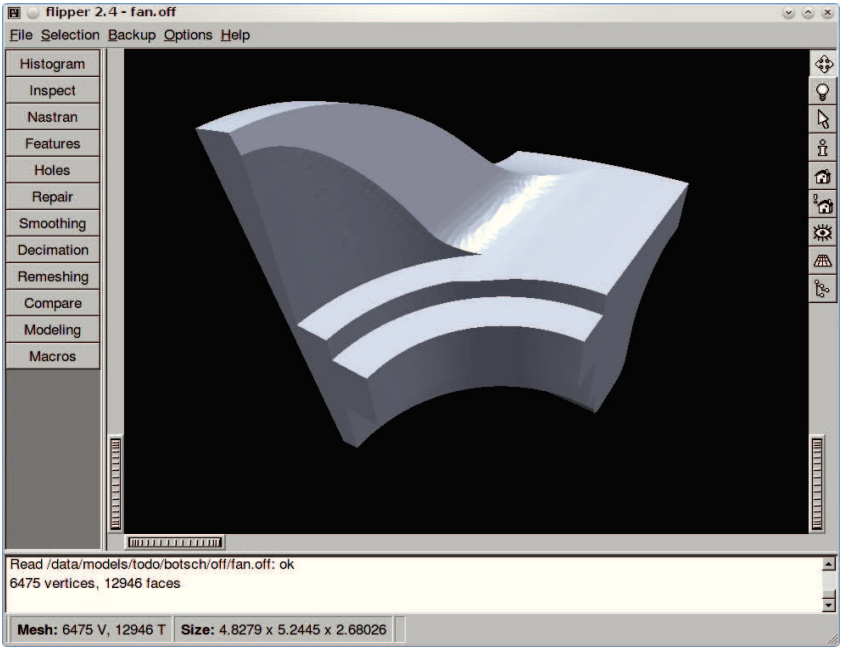


Figure 2.10: Screenshot of an application called Flipper. Flipper was one of the predecessors of the OpenFlipper framework.

Another application which acted as a basic framework for geometry processing was called Flipper. It provided a rudimentary plug-in system and integrated several algorithms into one application. The plug-in system was usable for algo-

rithms working directly on polygonal meshes and extended some aspects of the software. Nevertheless, a significant part of the framework was implemented in the core application itself. The rendering, input and output, most of the user interface and several other components where implemented directly in the core application and not in a plug-in which made it complicated or impossible to replace or enhance them.

Another drawback was that the software has been closed source. Therefore, it was impossible for external developers to modify the core of the application and therefore very important parts of the system. Only the interfaces of the software were available to write plug-ins. That made development complicated and a commercial usage for external developers practically impossible. But to acquire a significant number of developers and users, it is important to get a large community working on and with the software.

One example for the development of a plug-in for Flipper which proved to be problematic was a diploma thesis about geometry detail transfer [57]. Flipper was restricted to handle only one polygonal mesh at a time while, as described in Section 2.1, various geometry processing work-flows require more than one input and possibly different types of input at a time. The task of the thesis was to combine meshes requiring to position two meshes next to each other which was only achievable in this context by using non-public interfaces. Furthermore, the user interface could not be adapted to provide a nice workflow to the user. The functionality had to be implemented in an unclean way which caused instabilities in the beginning. A large amount of work was required to investigate the source of the problems and to solve them.

2.3.4 Meshlab

Meshlab [28] is an open source mesh processing framework mostly developed at the Visual computing lab of the *Istituto di Scienza e Tecnologie dell'Informazione* in Italy. It is available for the operating systems Windows, MacOS, and Linux.

It's main purpose is to provide functionality for processing triangular meshes which are for example generated by Laser scanners. It supports a large set of

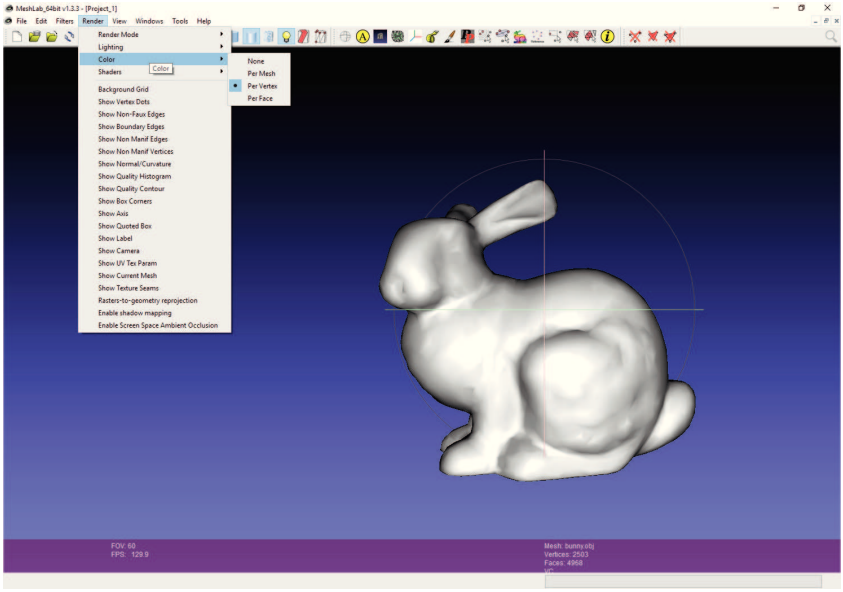


Figure 2.11: Screenshot of the Meshlab [28] application.

input and output file types. The mesh processing is provided via filters e.g for mesh repair, remeshing, and inspection. These filters can be extended via additional plug-ins which can implement new functionality.

A rendering infrastructure is available as well, which supports separate renderers as well as an interface to import code from Typhoon Lab's Shader Designer. The project is currently restricted to supporting meshes (polygonal and triangular) and points.

Meshlab is based on the Qt framework (Version 4) and uses the QMake build system to generate its make files. As the data types are directly integrated into the core, there is no easy way to add new types.

The code of the framework is licensed under the GNU General Public License version 2 [36].

2.3.5 Graphite

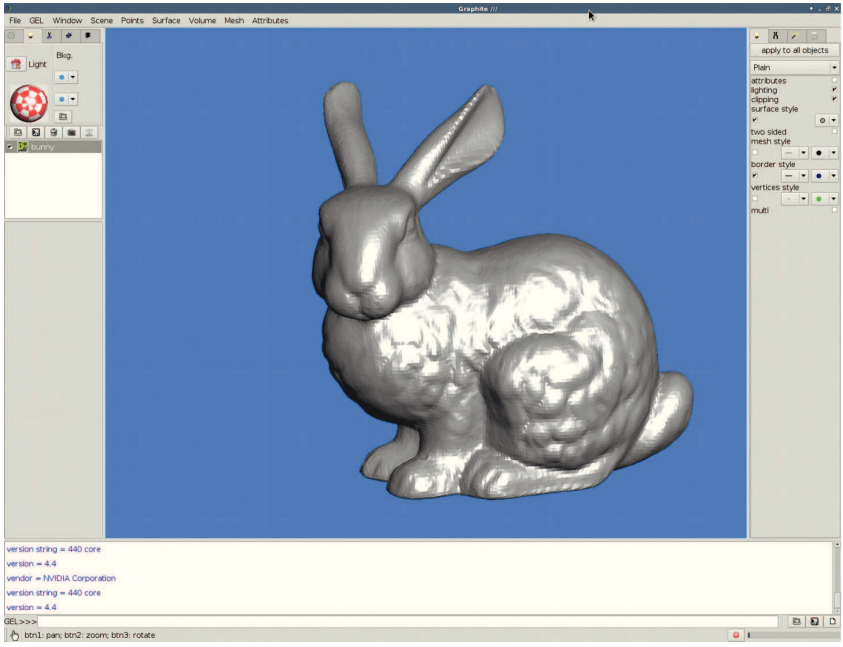


Figure 2.12: Graphite

Graphite [6] is also a research platform for 3d modeling and geometry processing based on the Geogram library [62] developed by the ALICE team at INRIA (*Institut national de recherche en informatique et en automatique*). It supports a variety of data types like point sets, surface and hexahedral volume meshes (via the Vorpaline library [63]). Like Meshlab it is also based on Qt (version 5) but uses cmake as its Makefile generator.

The system has basic rendering capabilities but is not meant to deal with research on new rendering technologies.

A major problem is the unclear license situation. Geogram is published under the 3-clause BSD License. However graphite is developed under the GPLv2 as stated in the source code.

2.3.6 Large Commercial Applications

Purely commercial applications generally have the drawback of being closed source. Plug-ins or extensions can be programmed via public interfaces. However several core functionalities which often cover renderers or other important components are fixed. It is also not possible to extend the core system directly as the source code is not available.

Furthermore, one usually has to buy licenses from the authors and rely on their distribution system which makes usage or extensions expensive, significantly reducing the number of possible users. For a research or teaching environment, this is generally not useful and prevents usage of such commercial applications.

3 Extraction of Software Requirements

In the first chapter, we analyzed several geometry processing workflows, user requirements and existing geometry processing software. In this chapter we will derive the software requirements from this information that have to be met by a general geometry processing framework.

3.1 Platform and Build System

Obviously to reach a large user base, the framework should be mostly independent from the underlying hardware platform and the operating system it is running on. This independence has to be achieved on several software component levels.

The first level is the decision of the programming language. The software requires very fast rendering and processing of data and should be available on as many machine types as possible. While java is available on many platforms, it does not provide the highest possible performance. C++ is available on almost every platform. Furthermore, a large set of libraries is available covering rendering, user interface and mathematical solvers. Besides it is easy to integrate libraries which are written in Fortran which is required for some high performance solver libraries. For example binary objects created by the Intel Fortran compiler can be linked to C++ programs.

The second component is the build system. Looking at different platforms, there is a large set of possible C++ compilers (Visual Studio, Intel, clang, gcc) and make tools (NMake, GNU Make, ninja). Furthermore, the library system of the platforms is completely heterogeneous. MacOS uses the Dynamic Libraries (dylib) format, Linux shared objects (so) and Windows dynamic link libraries

(dll). Besides architectural differences the build system needs to deal with the availability or lack of different system features or libraries, switching on and off specific components of the framework depending on them. The goal of the build system should be to hide these differences as much as possible from the user while providing useful information when components are missing or user interaction is required.

The build system also has to generate the project files automatically. For example on Windows we usually need Visual Studio solution files, while on Linux project files for Eclipse can be generated or plain *Makefiles*. This abstraction allows users to use their usual IDEs for development.

The different workflows showed that there is a large set of use cases of the software but not all components of the framework might be useful at any time. So the build system should provide the user with the possibility to build a framework that only contains the components which are useful for the current application yielding a slim and clear setup.

The last choice is the operating system abstraction layer. Several components of the hardware or software have to be addressed differently depending on the operating system used. There exist several libraries that provide abstraction of hardware access and/or user interface creation routines. The choice of these libraries is very important as it restricts where the framework can be compiled and run.

3.2 Flexibility

The vast number of possible input data, use cases, and workflows of geometry processing applications requires a highly modular system where all components can be removed, recombined or replaced. To get such a modular system, all semantic components should be implemented as plug-ins that use well defined interfaces while the core of the framework organizes the interconnections and ensures a common look and feel.

This modularity also ensures that the inputs and outputs of algorithms are well defined and can lead to small building blocks. These small building blocks

enable software testers to focus on single software components and it is easier to reuse them. The modularity also allows researchers to implement several algorithms for the same operation and compare them to analyze their results.

3.3 Data Types and Data Structures

Looking at the different workflows of geometry processing algorithms, we can identify a standard set of data types that should be supported by a geometry processing framework. Basic input from laser scanners are point sets. For supporting surface extraction, the software also needs to handle triangular and polygonal meshes. The Animation workflow also requires skeletons along with animation data which is attachable to surface meshes. Several 3D simulation techniques also require volumetric representations like hexahedral meshes. As there are possibly many other data types in geometry processing and new types emerge, the data types may not be part of the core system but should be a modular part of the framework as well.

In order to provide a good starting point for users, the basic data types, their rendering, and selection have to be available in the framework. Furthermore, existing algorithms should be easily usable inside the framework and have direct access to the underlying data types.

Most of these existing algorithms heavily rely on the data structures they are implemented on. An additional abstraction layer would require the algorithms to be adapted and, if the algorithm should be used outside the framework, this layer must be removed again. To allow easy porting of algorithms between the framework and other software using the same data structures, no additional layer should be introduced and the algorithms should be implemented in a way that they do not interact with the framework directly but with well defined parameters on their data structures. This ensures that the algorithm implementation is always clean and does not abuse parts of the framework which could lead to incompatibilities.

Obviously not every field of application requires all data types and they might confuse users or developers and extend the compile time. The framework has

to provide easy to use mechanisms such that types can be added or removed from the framework on demand.

3.4 Basic Interface Groups

The most important part of a framework is the definition of interfaces. The interfaces between the core application and the plug-ins and directly between plug-ins define the data flow in the framework and how plug-ins can modify the framework. This section defines the interface groups which can be found in the previous work-flows.

Looking at the analysis in the previous section, we identified the following major interface blocks:

- File Operations
- Object and data handling
- Plug-in communication and automation
- Selection
- Rendering
- User Interface modification and extension
- Input metaphor and device handling
- License Management

3.4.1 File Operations

Obviously each geometry processing algorithm needs data as input which is usually read from files and the result will also be written back to a file. To support a large variety of file formats, the file readers and writers have to be plug-ins such that more file formats can be added to the framework at will. The file plug-ins need an interface to tell the core of the application which data

types and file types they support, such that the system can detect which plug-in is best suited to load or save a specific file. Furthermore, it has to organize the user interface, if several plug-ins are available which support the same data type.

3.4.2 Object and data handling

Supporting multiple data types requires an additional plug-in interface which defines how information about the types is passed around the framework and provide a way to register them to the core. This information can then be used in all other plug-ins of the framework. Furthermore, there needs to be an interface which controls how backup information for each type is be stored and provides user functions to trigger backup operations. Undo and redo are an integral part of a geometry processing framework and therefore need to be mapped via an interface as well. As not every plug-in will support each data type (e.g. for a remesher it does not make sense to work on point clouds), there must be an interface for the plug-ins to control, which data types are supported and to pass data between plug-ins.

3.4.3 Plug-in communication and automation

To allow for easy reuse of code in the framework an interface for calling plug-in code from other plug-ins is required. This way, plug-ins can be implemented for specific tasks and provide services to the rest of the framework.

Such an interface would of course imply that a scripting system can be implemented inside the framework. All functions provided by other plug-ins can be used by a special scripting system, which translates script code into calls to plug-ins or other framework functions.

This also covers the batch processing requirement which was raised in the simulation workflow 2.1.6. In this case, the framework should execute with (if user input is required) or without user interface and run a given script automatically. The non-interactive processing is also a requirement for automated

testing, as the software can run the tests unsupervised, collect the results and compare them to the target values.

3.4.4 Selection

Algorithms can work on whole objects or only parts of them. E.g. for a smoother a restriction to only a small component of a mesh might be useful to keep details on the rest of the surface intact. The selection of the area to work on can be either done automatically or by user interaction. To provide a unified user experience for all selection metaphors and data types an interface has to be available in the framework, separating the picking operation with mouse interaction from the actual selection of the object or its components in the data structure. This ensures a high flexibility when creating new object types and selection metaphors. Usually the picking operation is done by a special rendering pass of the whole scene where the id of all visible primitives in the scene is encoded in the final image. On a mouse click, the information at the clicked position can be read from the image and converted into an object id or possibly a primitive id of the identified object in the scene at this position. A core requirement for an interactive system is that this look up must be performed in real time, otherwise the delay between mouse interaction and system feedback might be annoying to the user. This means that the picking infrastructure in the framework needs to be highly efficient and of course accurate.

3.4.5 Rendering

One of the most important parts of a geometry processing framework beside the processing components is the rendering system. The input and also the processed geometry needs to be efficiently visualized to be conveniently examined by the user. There are several rendering techniques which can also be implemented, like global illumination, direct rendering, or deferred rendering. Furthermore, it should be easy to develop new visualization techniques inside the framework.

Furthermore, there are many different platforms and versions which have to be supported by the rendering system. Thus, an interface has to define on which platform a specific renderer is supported and enable it on demand. Additionally the input for all renderers is required to be well defined to offer an abstraction layer to separate the data types from the rendering code. The implementation details for the rendering plug-ins are given in Section 4.5.

The output of the rendering plug-ins are one or more images inside the rendering buffers which might contain additional information such as depth values and normals. For additional processing or analysis of these images, a second rendering interface should provide functions for image filtering. We call the plug-ins of this step post processors.

3.4.6 User Interface modification and extension

From the user requirements of the first section it is apparent that not only the processing and rendering is important but also the graphical user interface. Thus, a framework requires methods to extend and modify the user interface, adding toolbars and boxes, and controlling the input devices. However, although plug-ins might add additional user interface elements, the core of the framework should organize them. Otherwise the resulting system will be very confusing, especially with a large set of plug-ins.

3.4.7 Input metaphor and device handling

In geometry processing several additional input devices are available on the market next to the standard interfaces like keyboard and mouse. These input devices reach from tablets to 3D input devices like a space navigator. They usually offer more axes then the two provided by a mouse. They are more intuitive and should be easily integrateble into the framework.

3.4.8 License Management

The last interface is mostly for commercial applications or for research code that should not be distributed except to e.g. reviewers. It provides a licensing system allowing to enable plug-ins to work only if a license for the current workstation is available. This is of course only useful for binary distributions of the framework when the source code is not publicly available.

3.5 Object Inspection and Information

For efficient algorithm development and testing, the framework needs advanced tools to analyze not only end-results but also intermediate steps or meta data produced. The tools should be usable to visualize different kinds of data from simple Boolean values up to vector data (e.g. showing normals). This could of course be used to compare the quality of algorithms, e.g. by calculating the distance of two meshes at each vertex and visualizing it by a texture. This way the user can easily identify regions where the algorithms differ most.

Besides showing information on a surface simple information like primitive counts should be available to the user.

3.6 Rendering Infrastructure

One fundamental part of a geometry processing framework is the rendering infrastructure. The input, intermediate results, and the output of the algorithms need to be inspected and evaluated by humans. Therefore, the data and possibly additional meta data needs to be visualized on the screen. Following the prerequisite of the framework being executable on many platforms, the number of possible rendering infrastructures is limited. The most commonly used rendering architectures on PC based hardware are currently OpenGL [43] and Direct3D [75]. Direct3D is only available on Windows as part of DirectX, so the obvious choice is OpenGL.

However it is not wise to restrict a framework to only one rendering infrastructure. New rendering interfaces will always have to be developed and integrated. One example is the new Vulkan [44] which is currently developed by the Khronos group as an OpenGL successor.

Therefore, a common interface should be defined providing an abstraction layer between the content to be rendered and the rendering itself. The common data all renderers access are the primitives that will be displayed on the screen along with additional information like color or normals. Furthermore, details might be required such as the choice if additional shading should be applied or if the data should be handled by additional shaders.

This abstract representation can then be handed over to the actual renderer, processing the data for the targeted rendering infrastructure. If this abstraction layer does not match all requirements of the target infrastructure, it should still be possible for a rendering plug-in to iterate over the objects by itself and collect the required data, creating its own abstraction layer.

3.7 Interaction

Obviously a geometry processing framework needs to provide methods to allow users to interact with the scene to control algorithms and to modify objects. The simplest interaction would be to click onto an object in the scene in order to select a point or any other primitive. This picking is the most common interaction between a user and the system, and must be performed in real time with a very low lag. A more sophisticated type of interaction are selection metaphors. For the surface modeling of Section 2.1.5 areas have to be selected to define deformable regions and handles. For this usually lasso selections were used, where the user can draw a polygon which will span a volume in the scene, selecting all primitives inside it. There are many more metaphors like painting selections on the mesh or selecting flat areas. Therefore, the framework should provide extensions to implement different selection metaphors on all data types.

These basic functions should be provided by the framework, as they can be difficult to implement when required at high frame rates.

3.8 External libraries

A large number of geometry processing algorithms require to solve linear equation systems. For instance the Quad Meshing workflow in Section 2.1.8 requires to solve an equation system in order to compute a global parametrization. Also the modeling in Section 2.1.5 requires to solve a Laplace system to compute a smooth deformation. Therefore, the framework needs to be able to handle solvers for various types of equation systems. However, as there are very good solvers (commercial and non-commercial) available, we do not want to directly integrate them but use them via external libraries. The framework should provide build system hooks to be able to easily integrate additional solvers and other libraries at will.

3.9 Judicial Subjects

A close look must be taken on legal issues of a software framework. For projects where the source code will be published, an open source license such as GPL [37] could be used. However the GPL is a so called copyleft (or viral) license. This means that also derivative works based on the original framework or code interacting with it would have to be published under the GPL license. This would render the framework unusable for commercial applications as the company would have to give away their know how. Therefore, a more permissive license has to be chosen to support closed source components within the framework. This could be for example the LGPL [38] (enforcing source code availability of the framework, but not necessarily of the plug-in) or even BSD-3 [69] providing almost full freedom of using the software.

Many companies still consider the LGPL license as problematic or too restrictive. The user must be given the possibility to replace the LGPL part in the final application with his own modified version. This means that the interface has to be defined in such a way that the binary can work with different versions of the linked code (usually the LGPL code is given as a shared or static library). This puts additional constraints on the software and can

prevent companies from using the system. Still this kind of license makes sure that modifications to the LGPL code will always be published, enforcing more code to be publicly available.

For open source development, it is still important to allow the use of open source libraries from within the framework. With the LGPL license this is not a problem. But for the BSD license the 3-clause version should be chosen. The 4-clause version contains an advertising clause, which is considered incompatible with the GPL license, preventing their combination. The 3-clause BSD does not contain that clause anymore, rendering it compatible with the GPL. The drawback of the BSD license is that the code does not have to be published at all.

It's up to the framework developer to choose the appropriate license. The project was started under the LGPL license. Although studies like [29] suggest that projects using copyleft licenses get a larger community then other projects, we switched to the 3-clause BSD license due to demands of several users who wanted to use the software in commercial setups. After switching from LGPL to BSD, a significant increase of code reflow from companies and other users back to the project could be recognized, as the framework got a larger and more professional user base. This might be related to the fact that geometry processing applications do not have a large private user base yet. This might change with the distribution of 3D printers in the future.

3.10 User Interface

The great functionality provided by a framework always comes with the risk of a complex and unintuitive user interface. To prevent the framework from producing cluttered user interfaces, the architecture needs to organize the visible information presented to the user. Otherwise, the requirement of an easy to use graphical interface can not be met.

Each of the plug-ins should be able to add user interface elements by registering them to the core. However, which elements are actually presented to the user should be controlled by the framework itself. Decisions of what is shown

can be made by checking which types of data objects (like polygonal lines or meshes) are currently visible and hide all user interface elements that can't work on these. This will provide a significantly cleaner user interface while offering full functionality.

Furthermore, tasks should be introduced for the plug-ins. E.g. a task for data acquisition from a scanner does not need to show controls for 3D print generation. This would significantly reduce the number of shown user interface elements.

Different user groups like developers, researchers and end-users have different standards for the interface. While for the former two groups the algorithms usually provide a lot of parameters, only the important ones should be handed over to the end-user. It might also be useful to have different interfaces to the same algorithm e.g. if parameters are already known from previous steps, these can be hidden from the interface, providing a more streamlined interface. So an additional requirement for the framework should be the possibility to separate the algorithm from the user interface. End-users can be presented with the stripped down interfaces, while developers use a full featured one. As the code behind the interfaces can be the same, only a small amount of work is required to generate it.

4 Implementation

In the first two chapters, we analyzed software in the field of geometry processing, its users and defined several software requirements, that have to be fulfilled by a geometry processing framework. In this chapter we focus on how these constraints have been implemented in our framework OpenFlipper. Some parts of this chapter are published in [59] and [60].

4.1 Build System

As depicted in Section 3.1 we want to address a large user base. Therefore, we support the three major platforms Windows, Linux and MacOS. This requires a cross-platform build system. One possibility is to provide pre-generated build scripts and ship them along with the framework, but this would require to modify the build scripts for every new plug-in and architecture which is error prone and adds unnecessary work load.

The alternative is to use an automatic build script generation tool to generate the files on demand at the users workstation. The possibly biggest build script generation tools providing the first abstraction layer of the build system are currently qmake (QT [30]), automake [40], and CMake [49].

Automake is a tool with its origin in the unix world. It usually requires a shell abstraction like e.g. cygwin to be run, rendering it too complex to be installed on Windows.

QMake, which comes with the Qt framework [30] is more flexible and available on all three platforms. However, it is not flexible enough and has a lack of documentation of many features it provides.

The last tool CMake [49] is available on all platforms, has a large user base and a well written documentation. So currently CMake is the best choice for the build script generation of a framework. Additionally, Qt also recently started to introduce CMake as an additional build system next to QMake.

Choosing CMake as the build system also allows to generate build scripts for a large number of 'make' systems, compilers and IDEs. For Windows, project files for all Visual Studio versions can be generated. We support different make systems like NMake (Visual Studio), GNU Make, or Ninja. And via CMake it is possible to generate project files for several IDEs like Eclipse or XCode.

4.2 Operating System Abstraction

The requirement to run on several operating systems and architectures requires an abstraction layer between the system and the framework. This could of course be implemented directly inside the framework, but there are several existing user interface framework libraries which are well tested and have a wide distribution. They provide not only pre- defined user interface elements but also abstractions such as e.g. for different file system access or network layer abstraction. Due to their large number of users, they are well tested and of good quality. So to reduce coding effort one of these frameworks should be used.

The two currently most popular gui frameworks are GTK [68] and Qt [30]. Qt is a more general framework while GTK puts its focus on the user interface elements. Qt also contains several additional classes for the abstraction of other tasks. Furthermore, the Qt framework supports a larger set of Operating systems and rendering backends. Therefore, we chose Qt as the framework for implementing the user interface in a platform independent way.

4.3 Plug-in Architecture

The OpenFlipper framework is semantically divided into two parts: The core application and the set of plug-ins. The core creates a Qt [30] and OpenGL

context as well as the application window and some basic GUI elements. OpenFlipper's GUI is composed of the user interface elements provided by Qt. But, as we support picking, every element in the scene can act as an interaction element (like the coordinate system). Furthermore, OpenFlipper's core provides a very basic rendering system used as a fallback solution on systems with outdated graphics cards or application setups that do not incorporate any rendering plug-in (see Section 4.5 for further information on that topic). Apart from that, OpenFlipper's core does not contain any further advanced functionality but manages the interaction and communication between plug-ins and organizes the user interface.

Practically all functional units are added individually in terms of dynamically linked plug-in libraries. OpenFlipper's API provides a variety of plug-in interfaces from which plug-ins may inherit in order to access a set of specialized functions. These functions are used for the communication between plug-ins and the core application as well as between different plug-ins. As OpenFlipper is an event-driven architecture (as most applications using mouse interaction are), all communication is accomplished by making extensive use of Qt's event system, i.e. signals and slots that are processed with the help of event queues. There exist different types of events, synchronous and asynchronous ones that can be used in order to provide a powerful way of interoperability between the different parts of OpenFlipper even in multi-thread environments.

4.3.1 Plug-in concept

We decided to use the Qt framework [30] as the basis for our geometry processing framework. Qt comes with a powerful signal/slot mechanism used to connect functions inside the framework.

Usually callbacks are used to trigger a function from within another function. These callbacks are pointers and usually there is no type checking available when passing parameters. Furthermore, the callback must be triggered by the calling function which then has to know which function will be called.

In Qt a widget or class can define signals. These signals can be triggered (emitted) from any function inside the class. This class does not need to know about the called function and how many functions are connected, providing a great flexibility. Each of these signals can be connected to slots of the same class or from other classes. There are two important types of connections: direct connections and queued connections. The difference between them is how the control flow is handed over to the called function.

When a direct connection is established between a signal and a slot, then, if the signal is triggered inside a function, execution directly continues in the called slot. If multiple slots are connected to it, they are executed one after the other. If all slots finished, execution is continued in the calling function.

A Queued connection will be handled via Qt's event queue. The function emitting the signal continues until it finishes. The slots connected to the signal will be put into a queue which will be processed after the function has finished. Therefore, it is possible to emit several signals in a function without having to wait for their execution. The queue ensures that the slots will be executed in the order they are triggered by the signals.

Another nice side effect of the signal/slot infrastructure of Qt is the possibility to define interfaces between the plug-ins of the processing framework that can be loaded and implemented on demand. A plug-in can partially implement an interface.

When loading a plug-in, the core will detect what part of each interface is available in the plug-in and will only connect these parts. This ensures, that only the part relevant for the plug-in needs to be implemented, which reduces code complexity and furthermore, it reduces the number of function calls by limiting them to relevant plug-ins. An example for this is that a plug-in adding a file importer does not need to be informed about object or view updates after a file has been loaded.

4.3.2 Plug-in communication

As almost all functionality in OpenFlipper is provided via plug-ins, the core of the system needs to organize the communication between them. The first task of the framework is to provide an infrastructure for the startup of the application. For an event driven system like OpenFlipper it is important to use a fixed initialization order to make sure that the dependencies of the plug-ins are available when they are required. This does not only include the startup of the user interface and the core but also initialization of the plug-ins themselves. The startup phase of the framework is shown in Figure 4.1

To get a consistent plug-in initialization the system first starts the user interface with rendering contexts and log output. For the plug-ins, the base interface contains two initialization slots which will be called on startup.

The first (initializePlugin) is called for every plug-in. In this stage, the plug-ins are not allowed to rely on functions of other plug-ins or to interact with everything except the core (adding ui elements,etc.).

The second slot (pluginsInitialized) is called by the core for all plug-ins, if the first stage is finished. Now the plug-ins can be sure that all other plug-ins are loaded and executed their basic initialization. Therefore, initialization requiring inter plug-in communication can be performed in this second stage.

After initialization, the plug-ins can load additional configuration data which can be used to store their former state or configuration settings. Now the system is ready to accept external events like user input. The usual event flow is that the user triggers a command (e.g. by pressing a button or clicking into the scene) which in turn triggers a function in a plug-in. If the plug-in then modifies any data, the core and all other plug-ins need to be informed about the change. The framework distinguishes between object and scene/view updates.

Figure 4.2 shows the control flow of object updates. The plug-in modifies an object and then sends a signal to the core which object has been changed and possibly the kind of change (which can be used to reduce the complexity of a screen update e.g. if only colors changed). This information is then passed from the core to all plug-ins which can react on this change. Afterwards the

core can trigger a redraw of the scene incorporating all changes made during this update call. This reduces the number of rendering updates required, as they are collected first and then performed only once.

Scene updates and redraws can be triggered by plug-ins via object updates, direct scene updates or by the core. Figure 4.3 shows the possible control flow of such an update. After a change to the scene, the core informs all plug-ins that the view or objects have been modified. Afterwards, the view management inside the core will redraw the scene and call the slotSceneDrawn on all plug-ins.

Redrawing the scene is an expensive task. It possibly requires the rendering buffers to be updated and uploaded to the GPU. A system is usually considered to be interactive if a frame rate above 30 frames per second is achieved. Therefore, this costly update is only required to run at that rate. So to ensure that not every object update triggers a redraw, the core view-management limits the update rate to 30fps (adjustable by the user). If more updates arrive, the data is collected and after the redraw time has been reached, the rendering will be updated. Note that only the visualization will be held back. The inter plug-in communication and internal data structures will be updated just in time to have consistent data throughout the framework at any time.

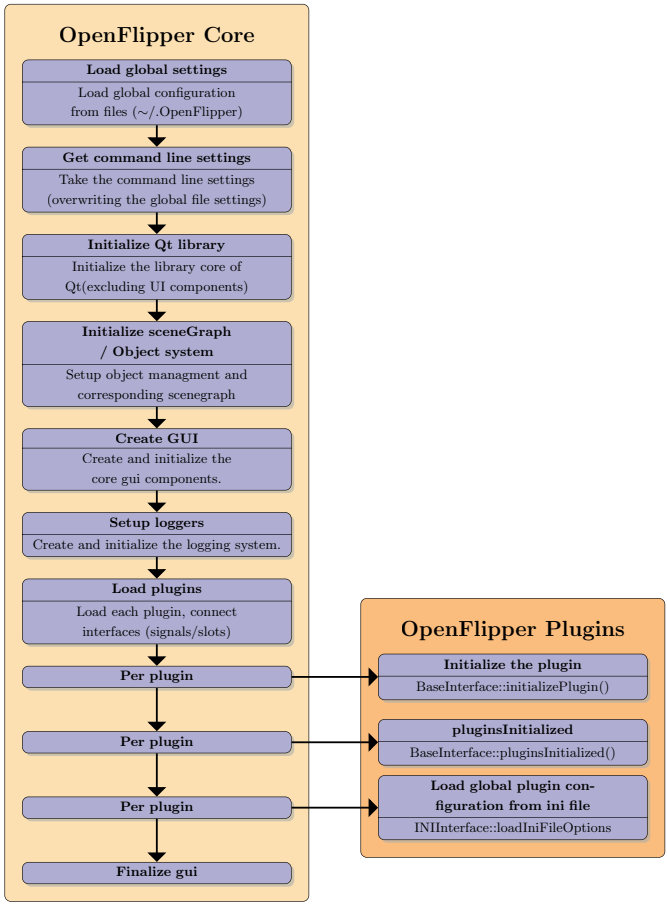


Figure 4.1: OpenFlipper startup diagram. This figure shows the order in which the core components of OpenFlipper and the corresponding plug-ins are initialized.

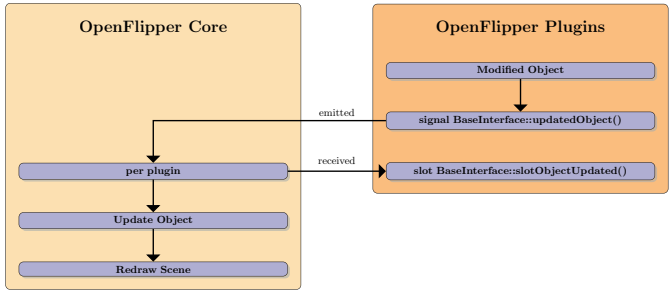


Figure 4.2: Object update notification control flow between the application core and the plug-ins.

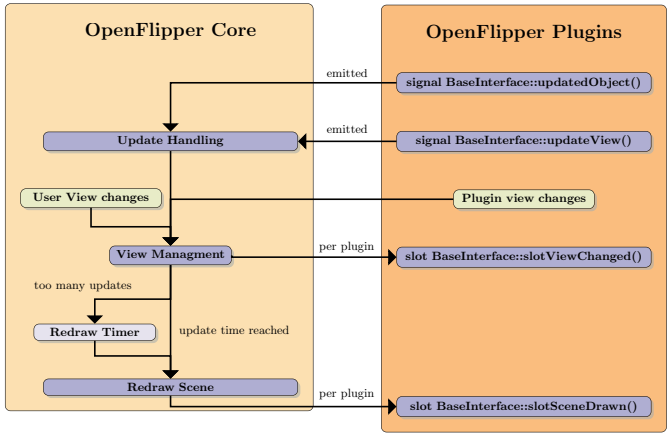


Figure 4.3: Scene update notification control flow in the OpenFlipper framework.

4.3.3 Interfaces

In Section 3.4 we identified several interfaces and interface groups that have to be available in the framework. The most important interface for all plug-ins is the BaseInterface. This interface is mandatory to all plug-ins as it provides the functions for loading and managing every plug-in. The object and scene updates as well as the initialization are handled via this interface. Every other interface can be used on demand. It is also possible to implement only parts of the interface if required. Table 4.1 shows a list of all currently available interfaces and their purpose in OpenFlipper.

User Interface Control	
Context Menu	Add entries to the scene context menu
Logging	Log output with different log levels
Menu	Create Menu entries in the main menu bar
Options Widget	Add entries and widgets to the options menu
Status Bar	Show notifications in the status bar at the bottom
Toolbox	Add additional toolbox widgets at the right side
Toolbar	Add toolbars and toolbar icons
View Modes	Categorize UI components and show only relevant parts
About	Add tabs to the about widget (e.g. show licenses used)
Object and Data handling	
File	File input and output
Load Save	Trigger load and save operations across plug-ins
Meta data	Attach meta data to objects (stored e.g. to screenshots)
Selection	Organize and provide selection metaphors
Type	Add and manage new object types in the framework
Information	Provide statistics about an object
Backup	Handle backup and restore operations
Input device handling	
Picking	Identification of clicked objects in the scene
Mouse	Handle mouse events
Keyboard	Keyboard interaction
Rendering	
Renderer	Implement rendering functionality
Post processor	Filter images generated by renderers
Texture	Control and add textures
Control flow	
RPC	Call functions across plugins via scripting
Scripting	Provide a scripting language for the framework
Plug-in connection	Directly connect plugin functions
Process	Run functions in separate threads, job management
License management	
License	Per plugin license management

Table 4.1: Interfaces available in OpenFlipper

4.4 Data types

The data types in OpenFlipper can be removed or added at will. The core itself does not contain type specific information. A special plug-in group is used to register new types to the core. These plug-ins provide the basic functionality to handle the data structures of the types, allocate their memory and provide some additional information.

The two most important functions of the type interface used for these plug-ins are the slots registerType, which will add the type name and icon information to the core, and the addEmpty function, which will create a new instance of the type and return its id to the core.

All access to an object is handled via the unique id that gets assigned at creation. Each data type also gets a name which is passed through OpenFlipper as a string. If a plug-in wants to work on an object, it can request it via its id. The core then returns the object as a BaseObject. The BaseObject is the base class, all objects in OpenFlipper are derived from. It is used to manage the data internally, without the core having to know details about the type. The BaseObject class can be dynamically cast to the actual object class to use it. Figure 4.4 shows the class Hierarchy. Some types in the hierarchy are invisible and used for management and sorting purposes, while the BaseObjectData class is used for visible objects in the scene.

This abstraction layer ensures, that plug-ins can be implemented to work on a subset of available types. We also provide automatic build system flags to disable or restrict plug-ins if all or some of the supported types are missing.

For easy object access, it is also possible to iterate over all objects currently in the system by the ObjectIterator. This iterator can be restricted to specific types and/or selected objects. The operator will return the objects as their BaseObject class, from which all other object types are derived.

A new ObjectType is implemented by providing a new object class derived from the BaseObject class. This new object class overwrites some basic functions specific to the new object type, like the rendering nodes for the scenegraph, and an encapsulation of the data structure. For example the MeshOb-

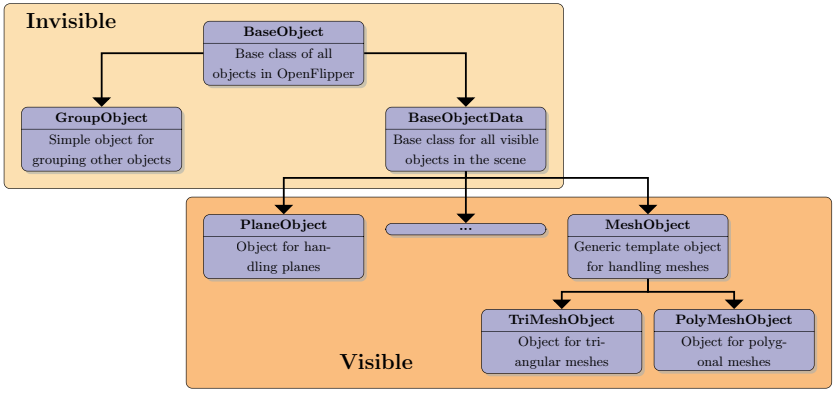


Figure 4.4: Object Hierarchy in OpenFlipper

ject type in OpenFlipper encapsulates the OpenMesh data structure. Plug-ins can retrieve the underlying data structure and directly modify it. After the modification the plug-ins will emit the updatedObject signal as described in Section 4.3.2 to announce the change.

To get a easy to use system, we tried to implement a large set of possible types in OpenFlipper. Table 4.2 shows the currently available types. Object management, rendering and selections are already implemented such that developers can directly start working on algorithms using these types.

Triangle and polygonal meshes are implemented using the OpenMesh [17] data structure. Volumetric meshes are using OpenVolumeMesh [53] and currently support Hexahedral and general Polyhedral meshes. The other data types use other simple data structures we implemented.

As proposed in the requirements of Section 3.3, we provide direct access to the original data structures used by the objects and separated the algorithmic

parts from the object types. This ensures easy migration of algorithms outside the framework to other software using the same or compatible data structures.

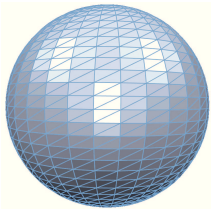
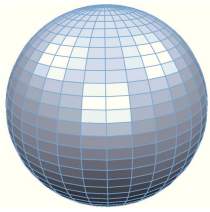
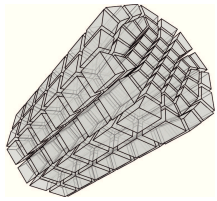
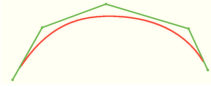
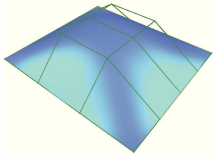
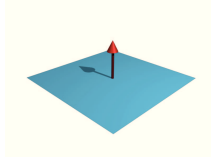
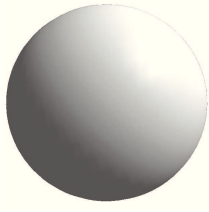
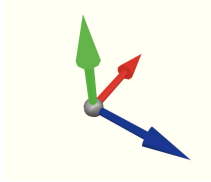
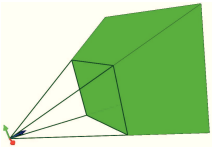
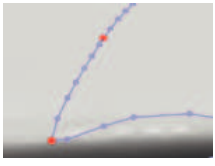
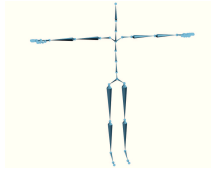
		
Triangle Meshes	Polygonal Meshes	Volumetric Meshes
		
Spline Curve	Spline Surface	Plane
		
Splats	Sphere	Coordinate System
		
Camera	Polygonal Lines	Skeleton

Table 4.2: Supported Data Types

4.5 Rendering

Undoubtedly, one of the most important parts of a virtual reality framework is the rendering back-end. Like the other components in OpenFlipper, the rendering functionality is suspended to plug-ins as well.

The core of OpenFlipper represents the scene in a hierarchical scene graph data structure. Each object in the scene has a set of corresponding nodes in the scene graph which take care of transforming and rendering the object as well as auxiliary information such as selections. Additionally, the scene graph contains nodes to control different OpenGL states, e.g. the current material and textures attached to the object. Due to the scene graphs hierarchical structure, the states of a node are also applied to all of its attached child nodes.

The scenegraph structure can be separated into two major sections. The global scenegraph components handled by the core and the scenegraph sub trees used for the objects.

The core section provides several nodes for organizing the rendering. Most of the nodes in this part of the scenegraph don't contain any rendering code but only set status information which is later used to influence the rendering of the scene. One example are the light nodes, which configure light sources that are used for the whole scene.

There is also one section of nodes, which is used to render OpenFlippers OpenGL user interface components such as the coordinate system. A clipping node has been added to configure the clipping planes.

The programming interface allows plug-ins to add additional nodes if necessary at pre-defined positions in the Hierarchy. Figure 4.5 shows the core section of the scene graph and the possible modification points for plug-ins.

Next to the global scene graph hierarchy, each Object which is visible in the scene has an associated subtree as depicted in Figure 4.6.

It contains a separator node, without any functionality but used to group the nodes internally and to act as a connection point where nodes can be added without any transformation. Below the separator node, a translation node is added which controls the orientation and position of the object in the scene.

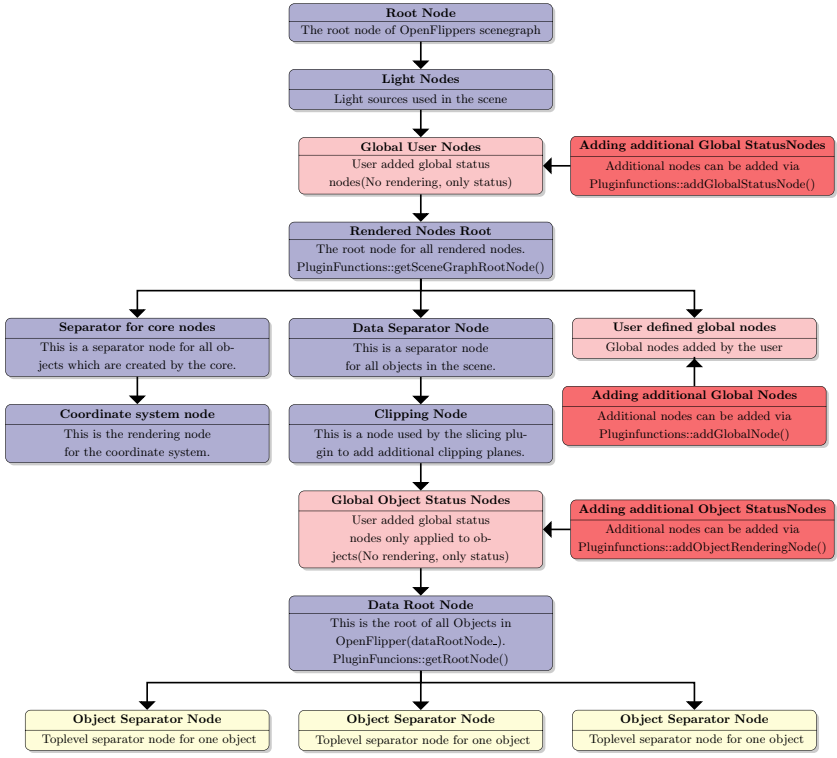


Figure 4.5: OpenFlippers core scenegraph structure. This scenegraph is parsed starting at the root node, when the scene gets rendered.

The bounding box node stores information about the bounding box of the objects below it and visualizes it on demand. The stencilRefNode and the material node are used to influence the objects visualization by providing basic

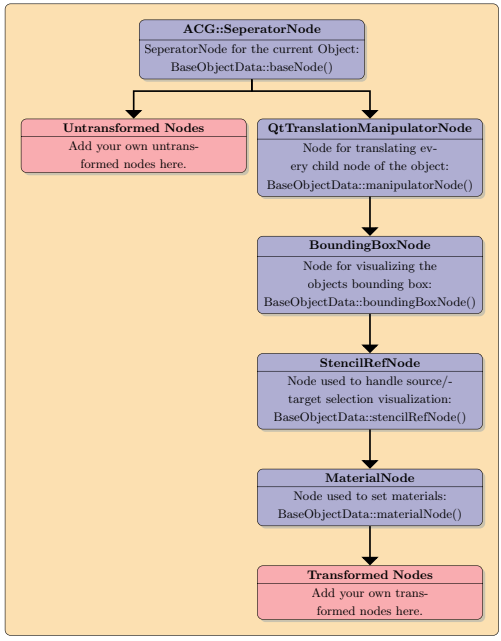


Figure 4.6: Subtree for each visible Object in OpenFlippers scene graph.

settings such as colors or specularity. Below these nodes the actual rendering nodes for each object can be added.

The structure is created when objects are loaded in the scene or when they are modified. E.g. if meshes are loaded, they are added to the scene graph as a node, which optimizes them for rendering (Cache optimization [67]). If they get modified, only the required parts of this optimized data structure are recomputed (topology,geometry,selection).

Rendering this scene graph structure is done via OpenFlippers renderering plug-ins. In order to keep the core as simple as possible, only a very simple

fallback renderer is integrated, that allows basic rendering, in case no external renderer plug-in is available. In the publicly available version of OpenFlipper, several rendering plug-ins are contained that support various rendering algorithms and strategies. They have access to the OpenGL context of the viewer and the scene graph. Prior to loading a renderer plug-in OpenFlipper checks whether the system's OpenGL version is sufficient in order to run the plug-in (each renderer plug-ins can use individual OpenGL driver revisions). If the currently installed OpenGL version is insufficient, the core will refuse to load the plug-in in order to avoid unexpected behavior or crashes due to unsupported hardware. The plug-ins are completely independent from each other, such that it is easy to develop and test new rendering algorithms without interfering with existing code.

Furthermore, OpenFlipper allows to split the screen into separate parts, each of which can be processed by a different renderer (Figure 4.7). This allows for directly comparing the results of the active renderers and using them to highlight different aspects of the objects (e.g. rendering an object using proper material and lighting simulations in one part of the screen, while the other part visualizes the object as a wire frame).

To simplify the implementation of rendering code and to support legacy graphics cards, we provide two different rendering interfaces in OpenFlipper: the classical and the advanced rendering interface. Renderer plug-ins can be derived from either of these interfaces depending on the degree of desired compatibility. The following sections provide a more elaborate description of the basic rendering interfaces.

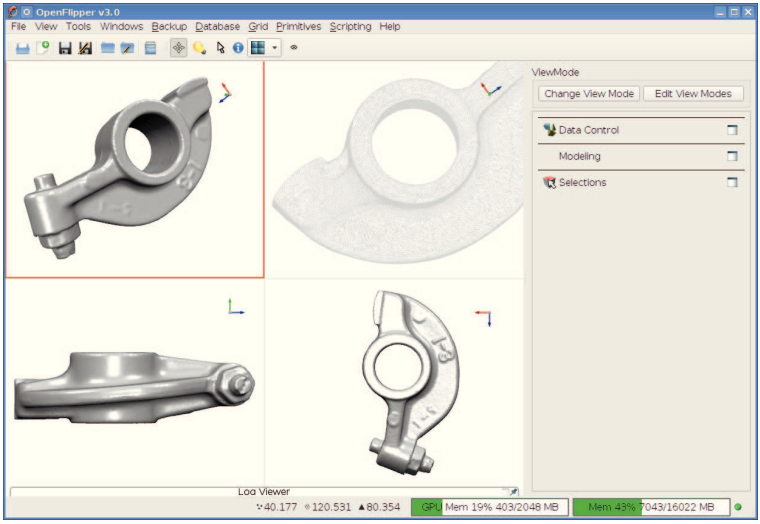


Figure 4.7: Screenshot of the multi view interface of OpenFlipper.

4.5.1 Classical Rendering Interface

The classical interface does not support high-performance rendering. It is rather intended to provide means for visualization on legacy systems. For this mode, each node of the scene graph has to provide a draw function that takes care of rendering the object represented by that node. The renderer plug-in itself does not need to know anything about the object to be rendered, as the corresponding OpenGL code is encapsulated in the node. Therefore, it is easy to create new objects and nodes as they only need to implement the draw function and no changes to the external renderer are required.

The drawback of placing the actual rendering code in the nodes is that, if a different visualization is wanted, one has to replace or extend this code. To allow different styles of rendering, the draw function in the nodes gets an additional draw mode parameter. These draw modes can be used to switch

or combine the visualization (e.g. wire frame or smooth shading). But still all rendering functions reside in the nodes.

When the scene is rendered, the scene graph is traversed by the active renderer plug-in. For each node an enter function is called setting the required OpenGL states. Then the corresponding draw function of the node is called. Afterwards, a leave function resets the OpenGL states to the original ones.

The limitation of this mode is that no optimization can be performed across the objects (e.g. no sorting of objects based on depth, shaders, primitives, materials or shared rendering buffers). Furthermore, the OpenGL states changed between draw calls cannot be controlled by the renderer, which is incompatible with global rendering techniques like Dual Depth Peeling [8]. Moreover, this approach is not seamlessly compatible with shader programming in general. For example, it is unclear when to set the uniforms, as the drawing nodes are independent from the texture, shader, and material nodes, but the uniforms required might depend on all of them. To overcome these problems and to add more flexibility when programming new shaders an advanced rendering interface has been added.

4.5.2 Advanced Rendering Interface

With the introduction of the advanced rendering interfaces the actual rendering code moves from the scene graph nodes to dedicated rendering plug-ins. This implies that render plug-ins would have to manage the different visualization modes of the individual object types. As a result, new object types would require modifications made to *all* available renderers. To avoid this restriction, a function which returns so called *render objects* is provided by the scene graph nodes. The idea of these objects is that graphic cards only support a fixed set of primitives, i.e. triangles, lines, etc. Therefore, it would be sufficient for the scene graph nodes to generate the required primitive buffers and provide them in a unified data structure. The render objects then contain pointers to OpenGL buffers which include the data to be rendered. Furthermore, they provide information about how the data is organized in the buffer (normals,

colors, etc.), which kind of primitive (triangle, point, line, etc.) should be rendered and the material, texture, or other states which have to be applied during the rendering process of an object. Therefore, one node has to provide more than one render object, e.g. if more than one texture or state is used in the object.

These render objects enable a unified draw procedure as the renderer itself can traverse the scene graph, collect the required render objects from the nodes, and start an optimization phase when all data is available. In this optimization the renderer can sort the render objects such that the number of state switches (shaders, textures,...) is minimized. Afterwards, the scene is drawn in that optimized order.

To simplify the creation of new renderer plug-ins, this process is implemented in a renderer base class. It consists of several steps:

1. The scene graph is traversed to collect the render objects.

2. The render objects are sorted and some initial OpenGL states are set.

3. Each render object is processed in newly determined order by binding its buffers, setting the required uniforms for the shaders, e.g. matrices, lighting parameters, etc., and performing the actual draw call.

Note that this setup allows full control of all state changes from within the renderer which allows for the implementation of more sophisticated, modern rendering techniques such as deferred shading. When all objects are drawn, all OpenGL states are reset to their defaults in order to prevent interference with other components of the application.

Due to these predefined functions a simple standard renderer is only a few lines of code calling the different stages. Still it is a small amount of additional code that is required to replace parts of the pipeline (e.g. replace the sorting algorithm) in order to create more advanced renderers. This flexible architecture allows for implementing highly modular renderers customized to meet the requirements of specific hardware configurations, e.g. surround-screen

projection systems such as the CAVE [31], and different applications, e.g. non-photorealistic visualizations. Figure 4.8 shows the data flow of the advanced rendering pipeline.

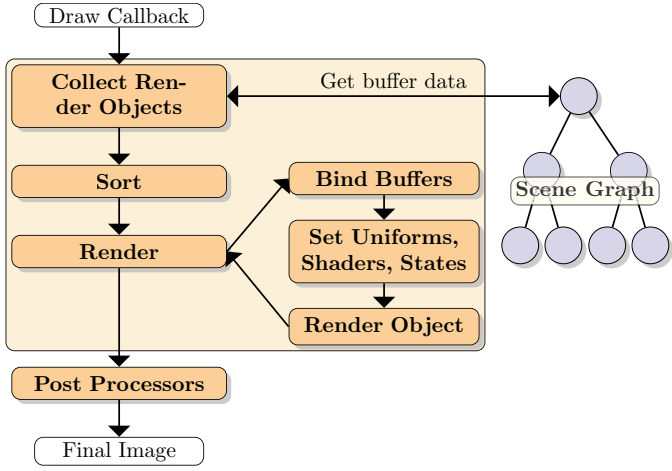


Figure 4.8: Data flow of the advanced rendering pipeline.

To simplify the construction of shaders, OpenFlipper provides a shader generator that uses template shader code files which can be customized by the renderer plug-ins. These shader template files contain markers which are replaced by custom code blocks at run time via so called shader modifiers. Basic variables like the current view/projection matrices or materials are automatically added to these template files. Therefore, all required uniform variables are passed to the shader by the rendering system so that the developer does not have to be concerned with their setup. Of course, it is still possible to write entirely customized shaders and to use them in the pipeline. To avoid unnecessary switching and compilation of shaders, a shader cache manages the efficient handling of the shaders, i.e. if a shader is used twice, it will only be compiled and linked once and reused for multiple render objects to avoid overhead.

This interface allows for creating advanced renderer plug-ins like Dual Depth Peeling [8] that require to create additional render targets and shaders to compute transparency in the scene without having to render the objects back to front. Therefore, the plug-in needs full control over the shaders and buffers while rendering several passes of the same scene with different shader setups. Our implementation follows closely the one described in [8]. Figure 4.9 shows a result of the renderer.

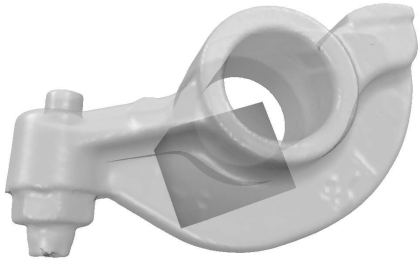


Figure 4.9: Rendering with depth peeling. Cube behind a semi translucent rocker arm model.

4.5.3 Post-Processors

To enable flexible rendering effects, OpenFlipper provides an additional rendering stage, the post-processing. This stage is run on the output of the renderers, i.e. frame buffers, depth buffers, etc. Post-processor plug-ins usually perform image processing algorithms executed on a rendered image but could equally be used to adapt the image to different output devices. Post-processor plug-ins have access to the OpenGL context and can therefore use all available buffers as their input. There are typically two different scenarios that require post-processing:

- Executing image based algorithms which analyze or enhance the images such as the detection of sharp features/corners that may then be accentuated in the final image.
- Reprocessing the images to be displayed on different output media.

One example for the latter is to split the image into segments and stream them to multiple displays (like a video wall consisting of an array of monitors). In this case, the post processor takes care of the splitting operation, compresses the image into a format compatible with the target platform and sends them, e.g. via network, to the display devices. Therefore, the renderers do not need to know anything about the final processing step, except for possible adaptations of the frame buffer's resolution when rendering for high-resolution targets. Some example post processor results are shown in Figure 4.10.

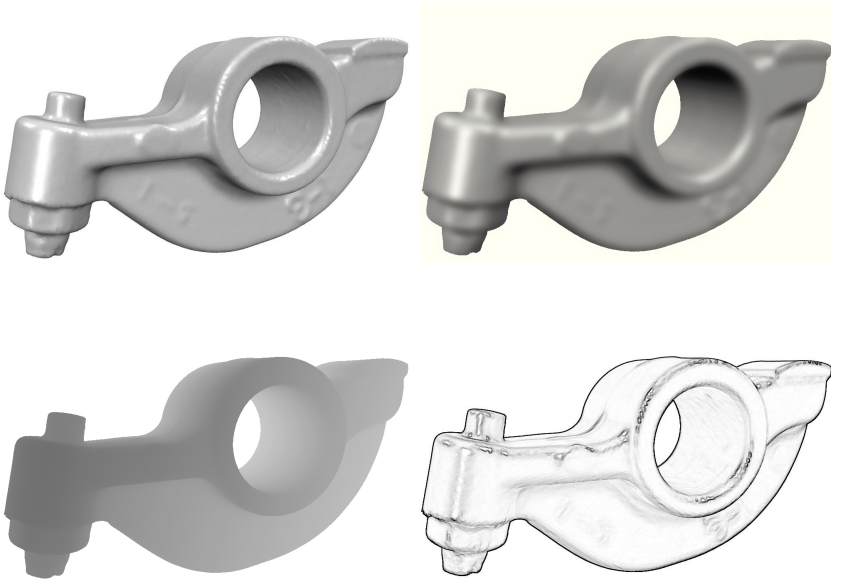


Figure 4.10: Top left: input from rendering plugins. Top right: Bilateral blur filter. Bottom left: depth component filter. Bottom right: Sobel filter

4.5.4 Stereo Support

Another configuration in need of post-processing steps is to generate output used on stereoscopic displays. Therefore, the available rendering plug-ins support a number of techniques for stereoscopy. Currently OpenFlipper supports three different modes:

- OpenGL stereo: In this mode, one image for each eye is rendered. Depending on the available hardware these images can either be displayed at the same time using e.g. polarization filters (passive stereo) or in an alternating way via shutter glasses (active stereo).

- Anaglyph stereo: As many devices do not support direct stereo rendering, OpenFlipper provides means for anaglyphic stereoscopy (see Figure 4.11 left).

- Auto stereoscopic displays: This is a special mode for some auto stereoscopic displays. They take as input a color image and a depth image and compute an (approximated) 3D view. This additional mode is produced by a simple post processor plug-in which combines the color and depth buffer in one final image (Figure 4.11 right).

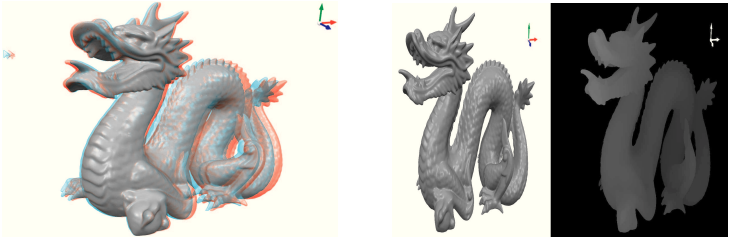


Figure 4.11: Left: Anaglyph stereo output. Right: Combined output of the color and depth image for auto stereoscopic displays.

One common problem of the interaction in virtual worlds when viewing geometry is the rendering of the mouse pointer. The level of immersion heavily depends on how plausible the rendering of different scene components appears for the human perception. If the scene is rendered and perceived in 3D while e.g. a mouse pointer's positions are restricted to 2D, the user perceives this as highly irritating and the sensation of depth is severely affected. OpenFlipper already includes a pointer infrastructure to render visual pointers at the correct depth. The correct depth is computed automatically such that the pointer is rendered at the same depth as the object behind it. The renderer plug-ins can also get the pointer information and replace the representation with a customized one.

4.6 Picking

Selections are an important component of a geometry processing framework. To provide an interactive selection system the user has to be able to select objects or part of them in realtime. There are different implementations available for picking. The two most important ones are the OpenGL based picking using OpenGL functions or color picking. Both versions are implemented in OpenFlipper but the OpenGL version is only available as a fallback for the color picking implementation.

For color picking, an additional render pass is required, where every primitive which is in the scene (triangles, edges,...) is assigned a color based on their id in the original data structure combined with the object id used in OpenFlipper. This number is encoded in the Red, Green, Blue and Alpha component of each primitive. Then the scene is rendered with these colors (with deactivated blending and shading to prevent color modifications) and stored in an off screen picking buffer. We provide a special rendering plug-in which can be used to show the picking buffer to simplify implementation and debugging of the picking for new data types. Figure 4.12 shows an example picking buffer of the Bahkauv model. Figure 4.13 shows a closeup of the picking buffer.

One can see that every primitive has a slightly different color. The faces are rendered first and afterwards edges and then points. If a user clicks into the screen, a look up is made at the position in the picking buffer. The color at the clicked pixel can then be translated to the object which has been clicked on and the exact primitive that has been hit.

There are several advantages of the color picking approach. First of all, the buffers for rendering the color image for visualization and the picking are very similar. The geometry is identical and can be used for both rendering passes. Only the colors have to be computed and stored into an additional buffer. With a recent OpenGL version, even the additional color buffer can be omitted. The fragment shaders get an additional primitive id since OpenGL Shading Language 1.5. This means that the primitives are numbered in the order they were processed. This number can be used in the shader to compute

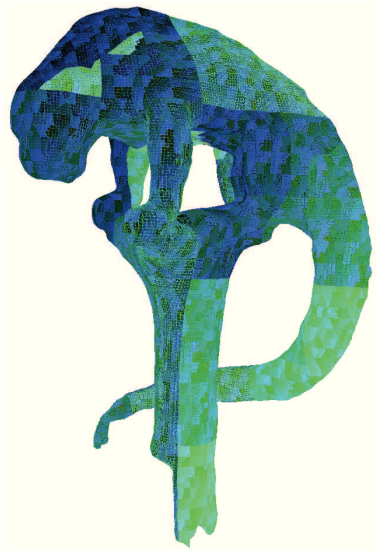


Figure 4.12: Picking buffer rendered by OpenFlipper. The picking buffer is used to identify a primitive, the user clicked on during interaction. To differentiate the primitives, they are rendered with different colors representing an unique id.

the color directly. Therefore, no additional buffer is required and the calculation in a shader on the GPU is much more performant than on the CPU.

Another advantage of the color picking is that the picking buffer can be reused. While no objects in the scene are changed and the view is not changed, the picking buffer does not need to be recomputed. OpenFlipper stores the buffer and reuses it if possible. To detect when the buffer gets invalid, the picking subsystem tracks the plug-in communication for object or view updates. If such a signal is detected the buffer is marked invalid. It will not be recomputed directly as the picking is only required when the user actually

Figure 4.13: Picking buffer closeup with enabled face and edge picking.

clicks into the scene. Therefore, the buffer will only be marked as invalid and recomputed on the fly when it is used again, saving computational time.

Besides this two additional picking tools have been implemented. The first one is a picking that reduces the problem that the user has to click exactly on a primitive to select it. On a picking operation, OpenFlipper will try to read the picking buffer at the exact position, the user selected with the mouse. If this pixel is empty, the next ring of pixels around the click will be checked and then a third ring. This ensures that even if the user slightly missed the primitive, it will be selected. As the pixels only need to be looked up from the buffer without additional computations, the extension comes at almost no additional cost.

The second improvement is a high precision picking. The 3D point the user clicked on can be directly retrieved from the picking buffer. It contains a depth value and therefore we can unproject the clicked point into 3D coordinates.

The drawback from using the depth buffer is its limited resolution. This causes the unprojected 3D point to be off the surface by an unpredictable distance.

The high precision picking in OpenFlipper requires the object type to implement a ray primitive intersection function that is called by the picking system. When the user clicks into the scene, first the hit object and the primitive is derived from the color in the picking buffer. Then the corresponding object is retrieved and the high precision picking function is called with the primitive id and a viewing or picking ray constructed from the current eye position and the clicked pixel. This ray can then be used by the object to intersect the ray with the primitive using exact coordinates retrieved from the data structure. The computed intersection point is then returned to the picking system. The advantage of this algorithm is that the intersection is performed at high precision on the data structure itself and not on the lower precision of the picking buffer. Another advantage is, that the data structure can also handle wrong intersections. For example if very small triangles are close to each other, the numerical inaccuracy of the depth buffer can result in a wrong primitive id. If the picking is then given to the high precision picking function of the object, it will detect that miss. It can even try to intersect the picking ray with neighboring triangles and then return the correct primitive that was hit.

4.7 Selection Metaphors

The selection of individual entities or groups of objects is a fundamental metaphor widely used in visualization and geometry processing applications. Selections are used to determine regions of interest e.g. to be subject to further editing and/or processing of algorithms. The presented framework supports handling objects of different kinds, such as polygonal meshes, polynomial curves and surfaces (B-splines), volumetric meshes, and many more (see Table 4.2). Some selection metaphors can be transferred trivially to different kinds of objects, e.g. the selection of vertices of a polygonal mesh and the selection of control points of a B-spline curve. However, this does not apply to all metaphors in general. In many cases, each of these object types consists of characteristic enti-

ties that need special handling when it comes to selections. For instance, when selecting a point on a B-spline curve, one might want to specify whether one is interested in selecting the actual point on the curve (thus in the curve's embedding space) or rather determine the corresponding pre-image in the curve's parameter space. In practice, both metaphors require two different selection modes.

From a software-architectural point of view, we solved this issue by splitting up OpenFlippers selection unit into a hierarchical tree of functionally differing selection layers. At the core is the base selection plug-in that implements—independent from specific object types—a set of elementary metaphors that are commonly shared among most object types, see Section 4.7.1 for details. On a higher level, there is a set of object specific selection plug-ins. These plug-ins implement the individual functionalities tailored to the specific object types. Apart from informing the application about the supported object dependent entity types, i.e. vertices, edges, etc., they also manage which of the basic selection metaphors, provided by the selection base plug-in, should be accessible for each entity type. The actual selection is also implemented in these plug-ins. Furthermore, one may add individual, object specific selection metaphors in these plug-ins.

The two different layers are described in more detail in Sections 4.7.1 and 4.7.2. Figure 4.14 depicts the underlying hierarchy of the mentioned selection layers.

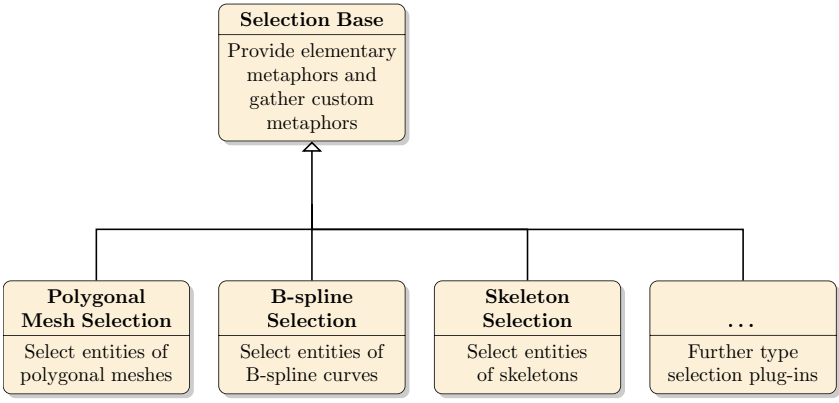


Figure 4.14: Hierarchy of selection layers in OpenFlipper

4.7.1 Basic Selection Layer

This layer is independent from specific object types. It provides basic selection metaphors that are commonly shared across multiple object types. Furthermore, it keeps track of all available primitive types as well as custom selection metaphors provided by the object specific selection plug-ins. The set of basic metaphors currently comprises the following operations: Toggle, Surface and Volume Lasso, Sphere, Flood Fill, Boundary, Connected Component.

4.7.2 Object Specific Selection Layer

This layer contains a set of object specific selection plug-ins—one for each object type. During the initialization stage these plug-ins inform the selection base plug-in about the individual entities enabled for selection (e.g., in the case of 2D polygonal meshes, vertices, edges, and faces). In a subsequent step they inform the base selection plug-in about which basic selection metaphor should be enabled for which entity. Additionally, further custom selection metaphors can be added optionally.

Then, while the user interacts with OpenFlipper, whenever a primitive as well as a metaphor is activated for selection, all mouse events are intercepted by the selection base and propagated through all object specific selection plug-ins.

4.7.3 Selection Data Flow

The object specific selection plug-ins provide information about all available custom selection metaphors, i.e. metaphors not provided by the selection base plug-in. It also provides a mapping of each primitive type to the available metaphors that indicates which metaphor should be enabled for use with a particular entity type. All available primitive types and associated metaphors will then appear in OpenFlippers GUI as buttons on a tool bar:

If the user activates a primitive type as well as a metaphor and clicks into the scene, the base plug-in will intercept the event triggered by the input device, determine the currently activated primitive type as well as the selection metaphor and passes this information on to all object specific selection plug-ins. The object specific plug-ins perform the actual picking and, where necessary, the algorithms used for the currently active selection metaphor. They directly modify the states of the respective objects in the scene. After the selection operation is done, they trigger a scene update in order to display the selections. Figure 4.15 schematically shows the underlying call sequence of a selection operation.

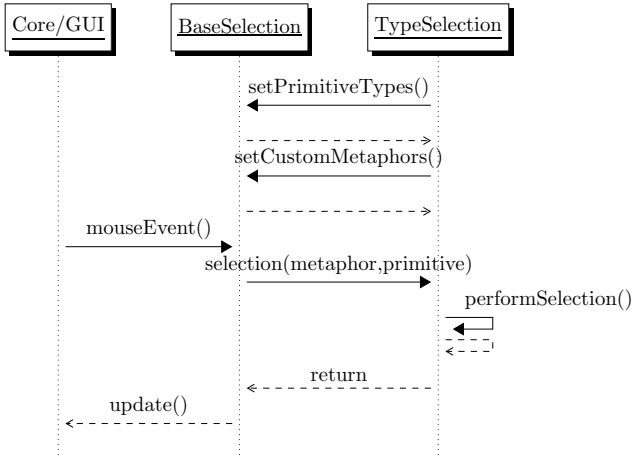


Figure 4.15: Data flow of a selection operation. All available primitive types and associated selection metaphors are registered in the initialization stage. Selection events triggered by input devices are then passed from the core to the base selection plug-in that triggers the actual selection operation in the object specific plug-ins.

4.8 Scripting

OpenFlipper provides a variety of modules to ease the development of an interactive application. At a later stage of software development extensive testing of algorithms is required. OpenFlipper provides a scripting environment integrated into the framework to automate such processes in a batch mode. Qt already ships with an excellent scripting system that is used as the basis for OpenFlippers scripting environment. As the language follows the ECMA-262 standard [5] which is also used for JavaScript, the syntax is familiar to a large number of developers.

The scripting system of OpenFlipper can be divided into two major parts. The first one is the general *text based scripting* system for experienced developers while the second one is a high level *Visual Scripting Interface* built on top of the general scripting.

4.8.1 Scripting Interface

OpenFlipper includes a text based scripting editor and interpreter. The editor collects and shows all available functions exported by the plug-ins, a description of their functionality and parameters. Scripts are visualized with syntax highlighting and can be directly executed without any compilation. Each function provided by a plug-in can be made available to the scripting system.

Basic types like vectors or matrices are known to the system and can be used or manipulated directly. Like JavaScript, the language provides loops, conditionals, input/output and many other standard operators. As scripts are evaluated at runtime, all existing algorithms in OpenFlipper can be used and controlled via the scripting system without having to recompile code. This is especially useful when testing and evaluating algorithms or trying to find optimal parameters for a set of algorithms.

The scripting system is also capable of modifying and extending the user interface. Qt includes the Qt Designer tool which can be used to generate user interfaces and toolboxes. The user interface specification files generated by this graphical designer tool can be loaded at run time and connected to all existing algorithms via the scripting language. Consequently no change to a plug-in is necessary for creating a new interface, a simple script is sufficient. The quadrilateral remeshing application described in Section 7.1.1 uses this option.

4.8.2 Visual Scripting Interface

Scripting is a powerful tool to combine simple algorithmic blocks to more complex algorithms. However, a programming or scripting language is usually too

complex for end users. To support less technically-experienced users in generating scripts, we implemented the more abstract Visual Scripting Interface [46]. The Visual Scripting Interface is build directly on top of OpenFlippers text based scripting system. The visual script editor is a data flow based block editor inspired by the block or filter based processing in audio or video processing applications [64]. The algorithms in OpenFlipper are represented as blocks with separate inputs and outputs. A simple example is an isotropic remesher. Input and output for this algorithm are surface triangle meshes. Additionally there can be other input parameters like, e.g., average edge length for the remeshed output. Figure 4.16 shows a simple example for such a visual script. This script consists of only three blocks. The first block computes the average edge length of an input mesh. Afterwards the computed length is passed to a math block and divided by a user specified number. The result is then passed to the isotropic remesher that uses the input value as the target edge length for its output mesh. The execution order for the different blocks can get fairly complicated, so the user has to define an order in which the algorithms are called. This is visualized by the data flow connections. For blocks that don't change objects (e.g., math blocks) the execution order is computed automatically.

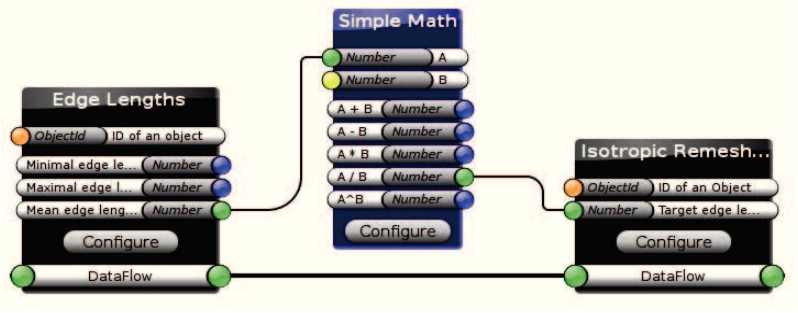


Figure 4.16: Remeshing algorithm in the Visual Scripting Editor

Many algorithms require user interaction to select an object to work on. The visual scripting system provides several interactive blocks, allowing to select objects or asking for user input.

From the implementational point of view every block in the editor is associated with an xml file containing in- and output specifications as well as small code snippets which represent the blocks in the final script. Every visual script is therefore parsed, the blocks from the xml files and all variables are connected to a documented OpenFlipper script which optionally can be viewed and modified by the user in the text based script editor. We have observed that these generated scripts provide an excellent foundation to learn the OpenFlipper scripting language. Users can read the code and use it as a starting point for creating more complex algorithms. The scripts are documented and changes made can be directly executed to get new results.

As the scripting blocks are defined and composed from simple xml files, the visual scripting system is not restricted to OpenFlippers language. Therefore, the editor and its components can be used to create script code for arbitrary languages.

4.9 User Interface Management and Extension

A drawback of the possibility that every plug-in is being able to modify the user interface is that the GUI can easily get overloaded. To avoid this, we moved the management of the interface components into the core of the framework. Every toolbar, toolbox or menu entry has to be registered via the core. The plug-ins can e.g. create a toolbox and then register this via the corresponding toolbox interface. This way, the core has a complete list of all added entries and can manage them in a meaningful way.

We introduced so called view modes into the framework. Either a plug-in or the user can create a view mode consisting of a set of menus, toolbars, toolboxes, context menus and status bars. If the view mode is activated via the interface (Figure 4.17) only the specified elements will be presented to the user. Figure 4.18 shows the view modes All and Modeling next to each other.

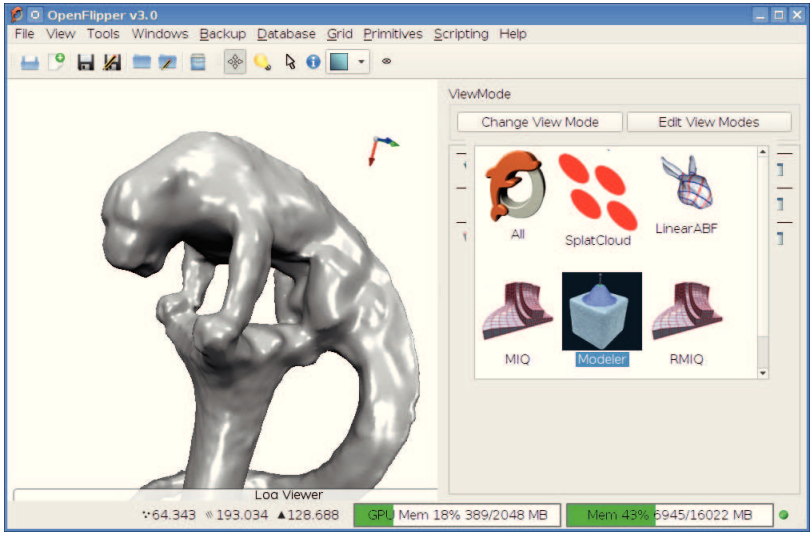


Figure 4.17: View mode selection

Note that next to the number of toolboxes on the right, also the toolbar icons at the top got reduced to the relevant ones.

Beside the possibility of adding gui interface elements via C++ code from plug-ins, we also integrated a scriptable solution to extend the user interface. Qt comes with a tool called Qt Designer to create user interfaces. OpenFlipper can use the scripting interface to load the files exported by Qt Designer and then integrate them into the user interface at runtime. The buttons can be connected to the plug-in via scripting functions such that the user can create a new user interface on top of the existing one. Of course these scripts can be shipped with an OpenFlipper application and executed at application startup or at runtime, completely hiding the original developer interface.

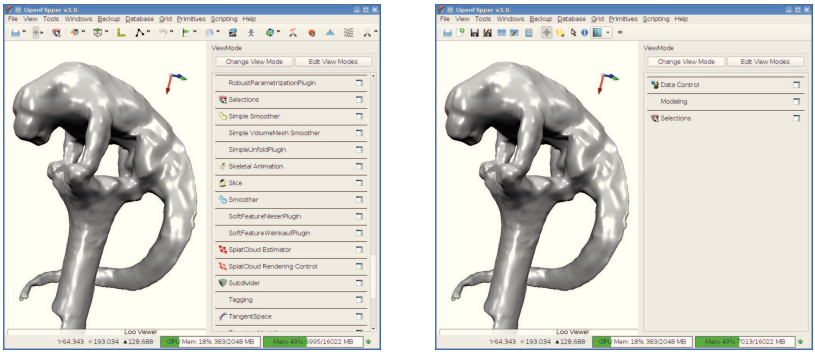


Figure 4.18: Left: View Mode All. Right: View mode Modeling.

4.10 License Management

To provide functions for commercial usage or to prevent plug-ins from uncontrolled distribution, we integrated a license management system into the framework. It supports a per plug-in license that will be bound to a specific hardware.

When a plug-in with enabled licensing is first loaded by the core, it detects that and starts an authorization procedure. The core calls an authentication function inside the plug-in. The plug-in will then make sure that the core and the plug-in binaries have not been modified by checking their hashes and writing them to a file. Beside the hashes of the binaries, the plug-in will be bound to the machine it is running at. This is established by retrieving information about CPU, Network Mac addresses and Windows Id depending on the operating it is currently running on. Each information is separately hashed and then stored to a license request file. The hashing ensures privacy such that no usable information about the machine is leaked to the party providing the license.

The license request file must be send to the producers of the plug-in. They will add an expiration date for the license along with a signature to the file

and send it back to the user who has to install it to a license folder in the framework.

If the plug-in is now loaded with the license file present, the binaries are checked against the hash values in the file. If they are ok, the plug-in will compare the machine information to the license. If all is correct, and the date in the license file shows that it has not expired and has a valid signature, the plug-in is loaded and usable by the core. Otherwise, the plug-in will deactivate itself and disconnect from the core until a valid license is provided. The advantage of the license system being implemented inside the plug-in is that modifications to the core to crack the license will be directly detected and the plug-in can refuse to work.

For the developer of a licensed plug-in the framework automatically generates a tool to create signatures for license requests. The request file is loaded into the tool and checks it for validity. If any of the hashes of the binaries or of the system are altered, it will be detected and no license can be generated. If everything is valid, the developer can adjust the expiration date of the license and let the tool generate the license file for the end-user.

5 Testing

While developing applications, considerable effort has to be put in the identification and resolution of software bugs. Especially in highly interactive systems, composed of various plug-ins, this can be time-consuming as the interaction between plug-ins may have unintended side effects. Additionally, these problems can become worse if the development team is distributed over several locations and projects. To overcome this, we set up an automated testing system with several stages of quality assurance: unit testing, smoke testing and continuous integration. Section 5.1 describes the integrated testing inside the framework, Section 5.2 the infrastructure configured to run the tests.

Some parts of this chapter are published in [60].

5.1 Testing Framework

The development pipeline of OpenFlipper is equipped with an integrated framework for testing many components at various implementational levels of the system.

On the lowest level of tests, a series of *unit tests* is performed for numerous low level functions independent of the core. This may be the creation of spatial trees from polygonal meshes, sorting algorithms or simply a random number generator. The functions which are tested are required to run without any user interface or interaction. This level of testing uses the C++ testing framework [42] developed by Google.

In the second level of testing, *smoke tests* make sure the main application is able to start with different combinations of plug-ins. At an early stage, these

tests make sure that no memory corruptions or interferences between plug-in functions are encountered during the start-up process.

The tests are composed of two consecutive parts. First off, OpenFlipper is run in batch mode, i.e. without user interface, for the purpose of checking whether the core and the plug-ins start up correctly without graphical user interface elements. They would return an error if plug-ins cannot be loaded due to linking errors (missing symbols) or if plug-ins conflict. Furthermore, the plug-ins are initialized during this test, so if any dependency of a plug-in is not met, the test will fail as well.

If this first start up succeeds, OpenFlipper is run with the user interface to verify that the graphical part of the application also works correctly and whether the plug-ins can expose their user interface components to the core.

At the highest level of the testing framework are the *integration tests*. They check the correctness of algorithms and interaction between plug-ins or even a whole workflow encoded as scripts. Again, OpenFlipper can run in batch mode with or without a user interface to check the basic components of algorithms or, in graphical user interface mode, to check the GUI and rendering results. For example, the cache optimizer class provides a smart way of caching the entities of polygonal meshes for efficient rendering. The tests on this unit can be run in batch mode to check whether the optimization algorithm works with different parameters, but nothing is actually rendered. In a next step, the algorithm is run again, but this time the results are rendered, collected and compared against a ground truth data set consisting of snapshots from previous application runs provided by a developer. As the user interface can also be included in the snapshots and is modifiable by the scripting, we can include it into the analysis. This way it is possible to narrow the potential error sources and see, if the underlying algorithm is broken or something goes wrong during the rendering.

As OpenFlipper can also take snapshots via the scripting interface, it is furthermore possible to check if rendered content suffers from regressions. To perform these checks, manually generated snapshots of the expected results are taken as ground truth and compared to the images resulting during the test

runs. These comparisons can be parametrized in different ways. The image is not necessarily required to be exactly the same as the ground truth. Therefore, it is reasonable to compare the images based on a threshold with different criteria (color difference per pixel, brightness, histogram of the image, etc.). If the difference exceeds the threshold, the developer is obliged to check the result. If the divergent result is still acceptable, it can be added to a pool of reference images used for subsequent tests. In some cases it is even reasonable to replace the initial ground truth image. This procedure allows for easy recognition of changes in the renderings and therefore detects regressions during the development of the application.

5.2 Infrastructure

To take advantage of this integrated testing environment, an automated infrastructure is required to run the tests and save time of the software testers. We use the continuous integration system *Jenkins* [4]. All check-ins into the code repository are automatically analyzed on all supported platforms in various ways. This ensures that the code compiles and executes correctly on all targeted operating systems and no regressions are introduced with new code revisions.

Furthermore, the code is checked at different levels to keep it as clean as possible. The lowest level is the static code analysis (we use *Cppcheck* [1] for this). The code is analyzed with respect to possible semantic and syntactic errors, compiler warnings, and issues concerning code style. Afterwards, the code is compiled on the different platforms and compilers (MSVC, GCC, Clang, Windows, Linux, MacOS). The automatic testing process is schematically depicted in Figure 5.1.

A list of all located issues is sent to the developer who caused them. This significantly improves the code quality and portability. Due to the automatic testing, we can support a rolling release schedule as the repository is kept clean and of high quality. Additionally, the continuous integration server automatically creates setup bundles of all builds (if the corresponding builds succeed

and pass all the tests). These can be directly used to install the full application on test systems or to deliver new versions directly to customers as soon as the code is updated.

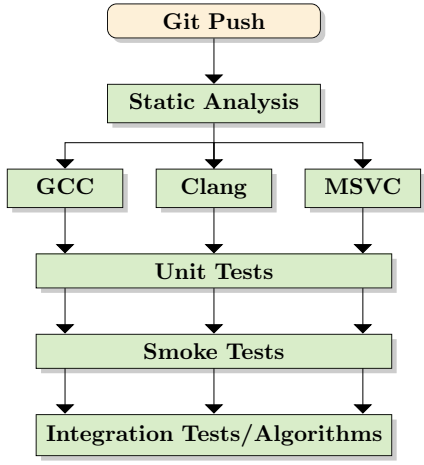


Figure 5.1: A schematic overview of the automated testing pipeline. After a push of the source code to the repository, it gets statically analyzed. Afterwards, the application is build on different compilers like GCC or Clang. The resulting binaries are executed and tested separately at their individual platforms.

As OpenFlipper is modular, it is convenient to create different packages for each project. These packages usually contain a different set of plug-ins. Plug-ins created in the publicly available part of the framework are simply linked to the individual packages such that updates are automatically propagated.

6 Project Management

The management of such a large framework with various components can get quite complicated. However, the modular infrastructure allows for sharing the responsibility among different work groups.

The most important group are the core developers. They maintain the core of the framework along with the interfaces, as well as the most fundamental set of plug-ins in the framework. They have to ensure that the interface specification of the core stays constant over a long time. This helps other developers as they can rely on a stable interface. However, the core developers also have to apply interface changes to the framework if a possible breakage helps to improve the interface or will reduce the number of bugs.

Other groups can manage their own set of plug-ins that are related to their current project. All of the separate projects are first tested in their own setup to make sure that they are kept in a good state. All of these projects are managed in the source code management system.

6.1 Source Code Management

The source code management is a central part of each software project. It is required to store the program code and to keep track of the changes made to the system and to manage different software releases. It stores all changes ever made to the program and who applied them. If errors are detected, one can go back through the history (e.g. by bisection) and identify which change (commit) introduced them and revert these changes. Furthermore, the revision control system is responsible for merging the work of different developers while preventing collisions when changes where made to the same code blocks.

For the actual source code management several toolkits exist like Subversion [65] or Git [3]. Subversion uses a centralized server where all code and change sets are collected. Git uses a distributed infrastructure, where each user has a full copy of the code in his repository.

For software projects which might also contain closed source components, the toolkit has to support a user management to assign different access permissions to developers working on the framework. While this is easy to achieve with Subversion (each directory and file in the repository can have its own access control list), a git setup is much more complicated as the standard git servers don't contain a detailed access control. Here usually all developers get access to the repository and no file or directory specific access restrictions can be applied. Therefore, we chose Gitlab [45] as our git server. The Gitlab server allows the creation of groups and several projects with different access permissions on top of git repositories. The core and all basic plug-ins of the framework belong to one master group with full read/write access. All other plug-ins can be managed in project-specific groups where the developer access can be managed independently or on an individual plug-in basis.

The continuous integration services can get special read access to the git repositories required for each project. When a user pushes his code to one of the projects, Gitlab will trigger automatic builds and tests on all architectures (either via the integrated continuous integration service of Gitlab or via Jenkins). The results will be collected in the Gitlab interface and the user is informed.

Setup binaries or files which might be useful after the automated builds, are collected as so called build artifacts. They are kept for a specified period while the rest of the build process gets deleted. A developer can directly download the compiled ready to use program for all supported platforms from this archive.

6.2 Code updates

For modifications to the code and especially to the core we suggest to use merge requests. This means that every user can modify the core and then ask the core

developer group to integrate these changes into the mainline repository. If such a merge request is made, Gitlab will first check if there are any source code collisions between the modifications made by the user and possible changes that have already been applied to the target repository. If there are no collisions, the code will be virtually integrated and the result is then build again and checked on all architectures. The results are recorded in the merge request.

If all tests run fine, the core developers have to do a code review. They can then decide if they accept the proposed changes or reject them for example if it would interfere with other projects or if the code quality is not sufficient. If this happens, the core team should mediate between the projects or request improved code.

This procedure can and should be applied to each sub project as well, defining one or more developers as masters who could approve a merge request. A specific release branch should be maintained, containing only the changes that are ready for a software release. New features and changes should be kept in separate branches and only be merged when they are ready and pass all quality and build tests.

6.3 Packages

For each software project in the framework only subsets of all available plug-ins and object types are required. To simplify sub-project management in OpenFlipper, we introduced so called packages. These packages are a collection of plug-ins, libraries and object types that belong to a separate project. Using a package, a set of plug-ins can be collected and distributed that are specific to the defined use case. The content of a package can also be a link to the relevant code (Subversion calls them externals while Git refers to them as submodules). Each plug-in should be managed in only one repository. Duplicating the plug-in in each package would result in redundant code which can lead to several different versions of the same plug-in and prevent other users from benefiting from new implementations.

6.4 Software Development Model

Software development in a framework for large projects involves management of various developer teams, software testers and users. For this kind of software project, an Agile development process [39] seems to be the best approach. This means that a software is developed iteratively in self-organizing teams providing continuous improvements and early delivery of results. Due to the high modularity of the framework, working software versions can be delivered automatically to the users. The developers can focus on their plug-ins and projects, resulting in a high motivation and faster progress. The continuous integration and testing established for the framework can significantly improve the overall quality and help to always deliver working software versions just in time.

6.5 Release Management

An important question for such a framework is the release management. There are several questions regarding e.g. the release dates or the classification of changes into major and minor revisions. For our framework we sticked to a rolling release for the core components of the system with point releases on major interface changes. However, we kept the number of breaking interface changes as low as possible. Plugins can define their own schedule as they only have to be compatible with the core.

For research environments, the rolling releases through our Git repository allow everybody to always get the latest improvements to the software, while the automated testing infrastructure ensures that errors or bugs in the code are detected before an actual merge. This provides a high development speed while keeping a good code quality.

6.6 Compatibility Management

To keep a framework useable, it always needs to be compatible with the libraries and tools it depends on. For OpenFlipper these are the build system Cmake, OpenGL, Glew and most importantly Qt. Major release updates for Qt and the other tools usually break compatibility with parts of the framework. To ensure a good user experience it is important to support the latest versions of these libraries while keeping backward compatibility to older versions. Especially for commercial customers or companies, version updates for the dependencies are made less frequent.

This also holds for compilers. For example OpenFlipper tries to maintain compatibility with old Visual Studio compilers which do not support a C++ specification higher than C++98 as these compilers are still used in a large variety of companies. However this requires a large amount of extra work to maintain the compatibility code and also puts additional load on the auto build systems as they have to build and test the software on several compiler versions per platform. On the other hand this method allows to reach a large group of users with one framework.

The old versions are supported as long as possible. Major releases however have to drop support for very old compilers in order to keep a manageable code base. This tradeoff holds for old dependencies as well.

7 Case Studies and Usage

During its development and after publication, OpenFlipper has been used in a variety of projects. These projects covered all workflows and user groups described in Section 2. It was used for research projects like the Quad meshing of Section 7.1.1 which were made commercially available directly inside the OpenFlipper project, effectively bridging the gap between research and end-user code.

Of course it has also been used for teaching. The teaching aspects are shown in Section 7.2. Section 7.4 provides a list of publications which used OpenFlipper during their development. Some parts of this chapter are published in [59].

7.1 Industrial Use Cases

A major design goal for OpenFlipper was to provide a toolkit allowing us to reduce the time and implementational overhead when converting research code to an end user application. This requires a solid base of working code that can be legally used in commercial and open source projects (LGPL or even BSD). Researchers are provided with a stable toolkit, enabling them to focus on implementation and testing of new algorithms while visualization, selection or analytic tools are readily available.

Additionally, the user interface in research and end user applications is usually quite different. As OpenFlipper allows us to create an additional interface on top of the existing functionality, only little effort needs to be spend on the abstraction of the interface while keeping the original interface available for expert users.

There are already several commercial projects using OpenFlipper as their platform. The projects provide continuous improvements and new features to OpenFlippers freely available parts and algorithms. A lot of basic algorithms are implemented for these projects which will also be published as open source components and are therefore usable and valuable for the community. In the following sections, we present two of these projects where OpenFlipper significantly reduced the coding effort during development.

7.1.1 Quadrilateral Remeshing

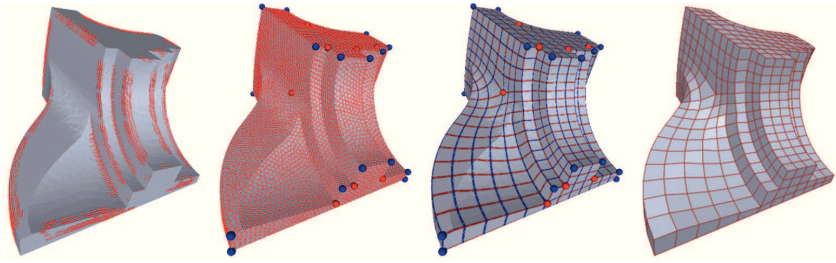


Figure 7.1: Quadmeshing algorithm (Picture from [15]).

Based on the OpenFlipper toolkit, a software for generating high quality quadrangular meshes from unstructured triangle meshes [15] has been developed in the context of a commercial project. The algorithm provides an automatic and a semi-automatic mode. The automatic mode does not require user interaction. In the semi-automatic mode, the user can control the algorithm's output by providing additional constraints for the output structures and therefore modify the resulting quadrangulation.

In the development process of this project the interaction metaphors already defined in OpenFlipper have been used to define these constraints. One of the constraint controls is OpenFlippers selection system. The user can select important edges and the algorithm uses the selection as a guidance for the final

edge directions. Additionally, OpenFlipper provides freely drawable polygonal lines on surfaces which are also used by the system to control the final output. Furthermore, the selection system is used to specify where singular vertices should be positioned. These singular vertices can be seen in Figure 7.1 as blue and red dots. All interaction metaphors, visualization, data types and input/output functions for this algorithm were already provided by the toolkit.

At its frontend the algorithm makes extensive use of multiple interfaces. The implementation consists of several parts which are implemented in independent plug-ins. The first plug-in computes the principal curvature directions on the input mesh. The second plug-in computes, based on the principal directions and possible user hints, a direction guiding field which is used to control the edge flow in the resulting quadrangular mesh. The third plug-in generates a parametrization and extracts a quadrangular mesh. The interfaces to all plug-ins are available to the professional user while a simple unified interface exists showing only the relevant steps and settings while hiding the remaining parameters (with empirically derived defaults) from the user. This additional interface is purely defined as an OpenFlipper script and can be loaded and even modified at run time. Figure 7.2 shows a comparison of the interfaces.

The mixed-integer quadrangulation solver used in [15] is also freely available as a separate library (CoMISo [14]). An example for the algorithm's output is shown in Figure 7.3.

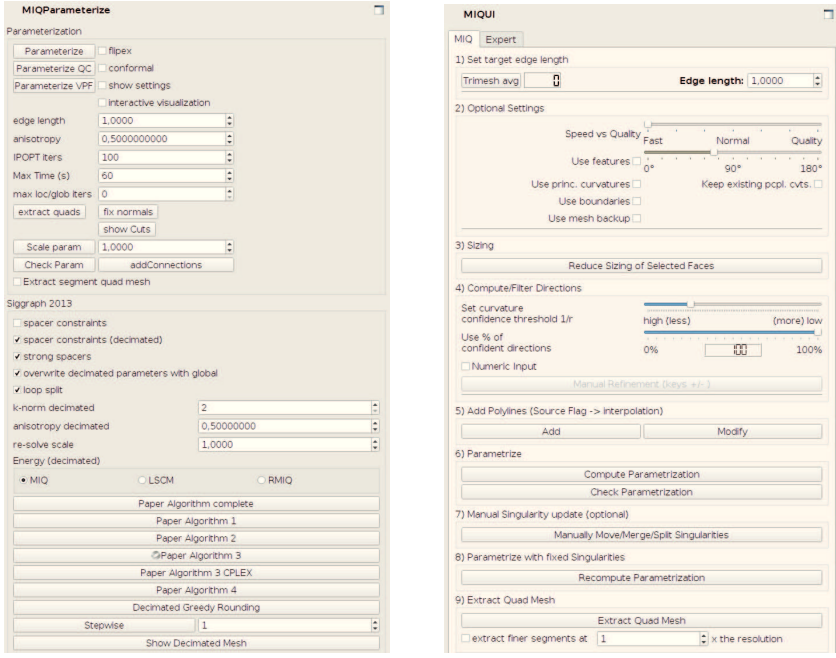


Figure 7.2: Left: Screenshot of the first page (of four pages) of the full developer interface, Right: Screenshot of the complete clean interface.

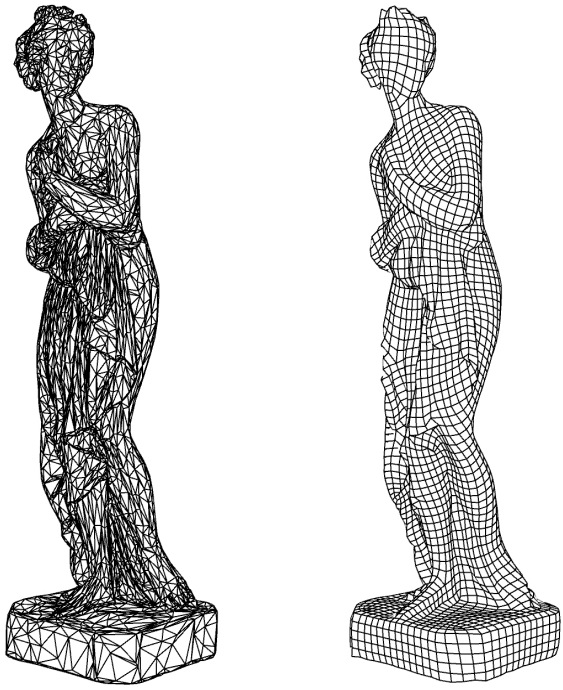


Figure 7.3: Model of Iphigenie as an unstructured triangle mesh and the result after the quadrangular remeshing algorithm.

7.1.2 Car Modeling

In this project [33], a semi-automatic approach to efficiently and robustly recover the characteristic feature curves of a given free-form surface has been developed where the input is not required to be a proper manifold. The technique supports a sketch-based interface implemented in OpenFlipper where the user must roughly sketch the feature location by drawing a stroke on the input mesh. For this type of interaction OpenFlippers picking system provides functions that return the 3D position of a mouse click in the scene. The system then snaps the initial sketch curve to the correct position based on a graph-cut optimization scheme that takes various surface properties into account. Additional positional constraints can be placed and modified manually which allows interactive feature curve editing. The feature curves can be used as handles for surface deformation, since they describe the main characteristics of an object. The system allows the user to manipulate a curve while the underlying surface adopts itself to the deformed feature.

During development of this project a lot of the existing functionality of OpenFlipper has been used and therefore significantly reduced the coding effort for this project. No rendering code was required as it was already available for the B-Spline and mesh data types used in this project. For these types the IO and file management already existed in the framework. Figure 7.4 shows an example for modeling a car using the final application.

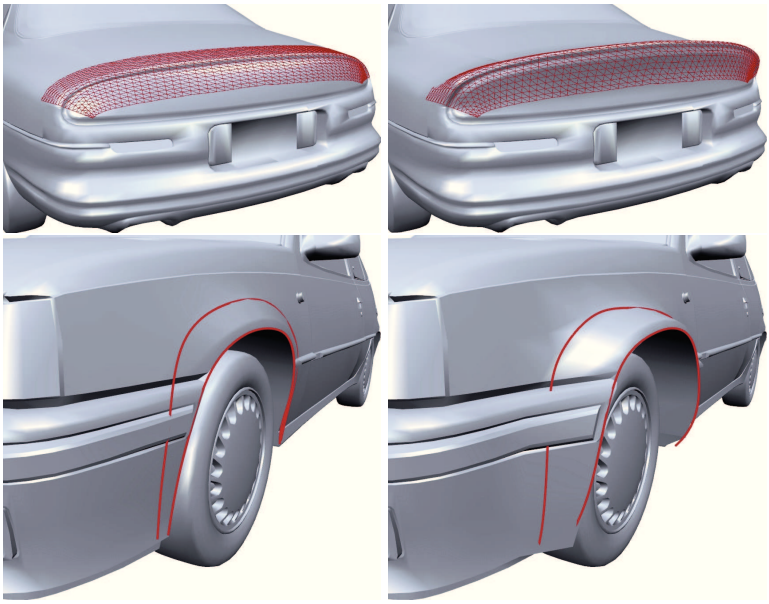


Figure 7.4: Car modelling implemented on top of OpenFlipper. Left: Original models, Right: Modified models.

7.2 Teaching

We use OpenFlipper for several classes. Some of them are for example Global Illumination and of course Geometry Processing. In both cases, OpenFlipper provides an abstraction layer which works on the most widespread operating systems. Students can stick to their platform while the exercise preconditions remain identical. File input and output and all basic functions are available. This means that for each exercise we can give the students a new plug-in containing a code skeleton which must be completed. For geometry processing this means that the students can directly work on a mesh without having to deal with rendering, selection or anything else. However they still have the

possibility to extend the plug-ins and learn more about the algorithms and the framework step by step to later on develop their own algorithms.

The following two subsections give some details about how the framework has been used in practicals or bachelor theses.

7.2.1 3D Printing Practical

Beside the classes, we also use OpenFlipper in practicals. One large practical realized with OpenFlipper was related to 3D printing. Three groups were assigned different tasks which are part of the code generation for a 3D printer.

One group implemented a slicer which cuts the object into slices of equal thickness which will be send as machine code to the printer to generate the output layer by layer. Next to this they also created a rendering plug-in used to visualize the generated machine code (which is called G-code), Figure 7.5 top.

The second group was responsible for support generation. As the 3D printer used in this practical creates the output layer by layer, problems arise when printing overhanging parts, as there might be nothing in the previous layer to print on. Therefore, these areas have to be braced by printing support structures from the bottom to the contact points. There are several possible support types. Tree-like support structures are depicted in Figure 7.5 bottom left. They have minimal contact points to the object surface as all support will leave marks on the object when removed which should be avoided.

The third group was responsible for generating code to create the so called infill. If an object is printed hollowly, it is not very stable and can be crushed easily. Therefore, an infill structure needs to be generated to provide stability. This infill always has the tradeoff between material and time spend on the infill versus stability and overall printing time. The generated quad and hexagonal patterns can be seen in Figure 7.5 bottom right.

The result of this practical was a fully functional 3d printing software. An object could be loaded as a surface mesh. The software then created support

structures and infill and sliced the object into printable layers. These slices are converted into machine instructions and can then be sent to a 3d printer.

Furthermore, a rendering plug-in has been developed to visualize arbitrary G-Code. This plug-in can now also be used to load G-Code generated by other software and visualize how a machine would move based on the given code.

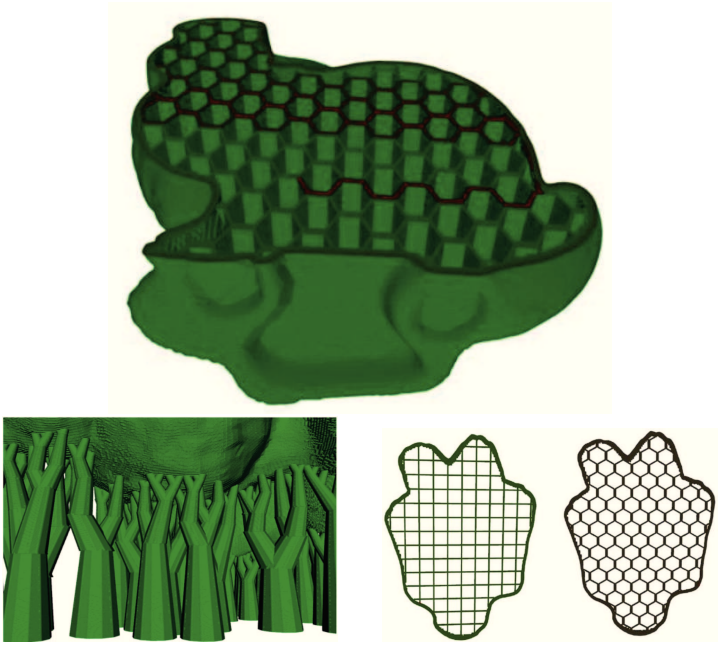


Figure 7.5: Top: Slicing and G-code visualization. Bottom left: Generated tree support. Bottom right: Quad and hexagonal infill.

Overall the practical was a success as the result was a useable software and also extensions to the framework have been created to handle and render more geometric data.

7.2.2 Yarn Generation

This project was a bachelor thesis [61] to develop a highly efficient renderer for fabrics. The goal was to reduce the amount of information which needs to be transmitted to and stored on the graphics card in order to save bandwidth, memory and computation time while preserving high rendering quality. At the same time the fabrics should be rendered at different levels of detail ranging from coarse overviews down to fine filament levels. To achieve these goals, the information send to the graphics card was reduced from a surface mesh per filament to a polygonal line with normals and a small set of additional parameters (e.g. texture information). From this simplified representation, the surface of a filament had to be reconstructed on the GPU using a geometry shader. To simplify the development of a reconstruction algorithm, the task has been split into two separate stages and therefore two different plugins in the framework.

The first stage was the development of the reconstruction algorithm creating a filament surface from a simple polygonal. This algorithm has been implemented at first in a separate plug-in which took polygonal lines as input and created standard surface meshes from them. These surface meshes have been visualized directly via the existing mesh renderers. Therefore, the Bachelor student was able to test and optimize the algorithm without having to deal with the more complex GPU implementation at the beginning which would have been much harder to debug.

The second stage of the project then transferred this implementation into a rendering plug-in which could directly render colored and textured filaments from a set of polygonal lines. An additional side effect was that the first stage of the thesis resulted in a plug-in which can be used to convert filaments from a polygonal line representation into a classical surface. This conventional representation can be used e.g. for simulations that require a surface structure instead of the simplified representation.

Figure 7.6 shows some renderings created with the plug-in.

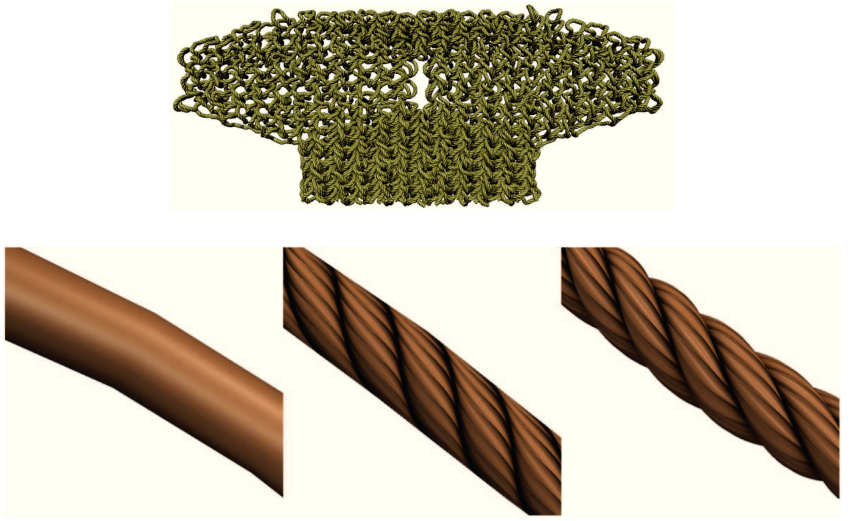


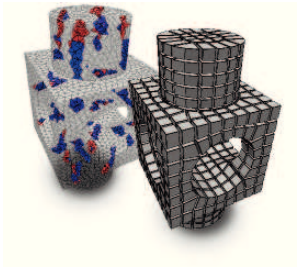
Figure 7.6: Top: Fabric patch. Bottom: Different level of detail renderings. Pictures taken from [61].

7.3 Computational Object Optimization

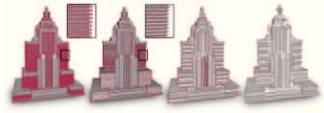
In this scenario OpenFlipper was used to optimize the surface of objects regarding constraints posed by fluid flows around or through them. Probst [66] developed a system to couple Computational Fluid Dynamics algorithms with geometric deformation methods and optimization systems. An optimization algorithm gets an object and possible deformations as input. The fluid flow around or through the object is analyzed using computational fluid dynamics. The results are measured by objective functions. Based on these results the object is deformed by the OpenFlipper framework (without user interaction in batch mode) to achieve better results. This deformed object is then passed back into the simulation. This process is repeated until an optimal solution is found.

7.4 Research Usage

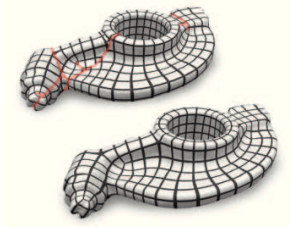
During its development the OpenFlipper framework has been used in a large number of research projects. Besides the examples given in Section 7.1.1 and 7.1.2, the following table gives a selection of research publications which have benefited from using the framework:



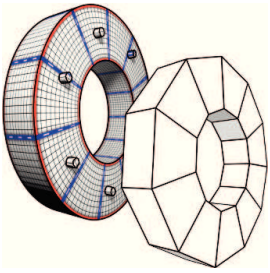
Max Lyon, David Bommes, Leif Kobbelt
HexEx: Robust Hexahedral Mesh Extraction [56]



Anne Gehre, Isaak Lim, Leif Kobbelt
Adapting Feature Curve Networks to a Prescribed Scale [41]

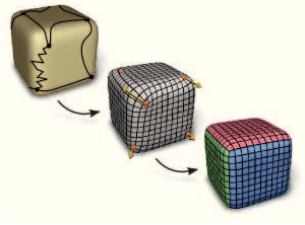


Marcel Campen, David Bommes, Leif Kobbelt
Quantized Global Parametrization [21]

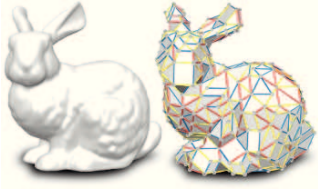


Hans-Christian Ebke, Marcel Campen, David Bommes, Leif Kobbelt
Level-of-Detail Quad Meshing [35]

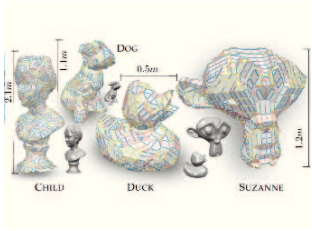
Marcel Campen, Leif Kobbelt
Dual Strip Weaving: Interactive Design of Quad Layouts using Elastica Strips [26]



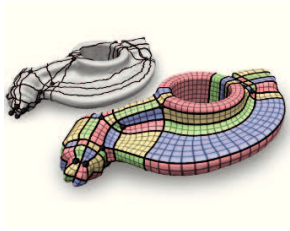
Marcel Campen, Leif Kobbelt
Quad Layout Embedding via Aligned Parameterization [27]



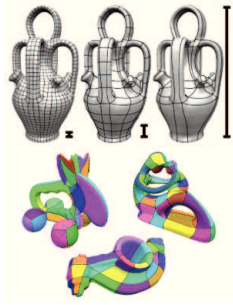
Henrik Zimmer, Florent Lafarge, Pierre Alliez, Leif Kobbelt
Zometool Shape Approximation [74]



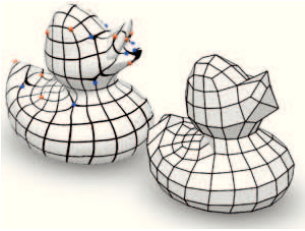
Henrik Zimmer, Leif Kobbelt
Zometool Rationalization of Freeform Surfaces [73]



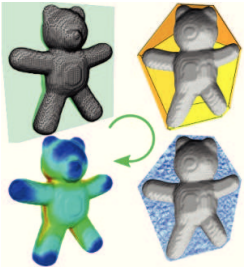
Marcel Campen
Quad Layouts Generation and Optimization of Conforming Quadrilateral Surface Partitions [19]



David Bommes, Marcel Campen, Hans-Christian Ebke, Pierre Alliez, Leif Kobbelt
Integer-Grid Maps for Reliable Quad Meshing [11]

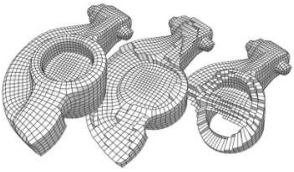


Hans-Christian Ebke, David Bommes, Marcel Campen, Leif Kobbelt
QEx: Robust Quad Mesh Extraction [34]



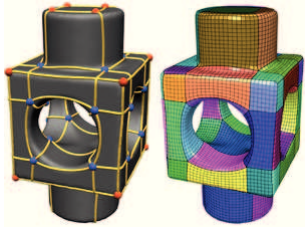
Henrik Zimmer and Marcel Campen and Leif Kobbelt
Efficient Computation of Shortest Path-Concavity for 3D Meshes [72]

Marcel Campen, Martin Heistermann, Leif Kobbelt Practical Anisotropic Geodesy [22]

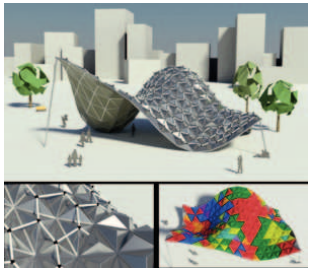


Michael Kremer, David Bommes, Isaak Lim, Leif Kobbelt
Advanced Automatic Hexahedral Mesh Generation from Surface Quad Meshes [54]

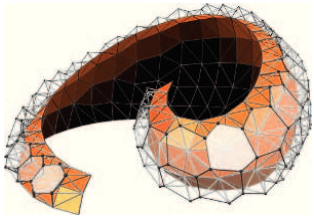
Ellen Dekkers, Leif Kobbelt
Geometry Seam Carving [32]



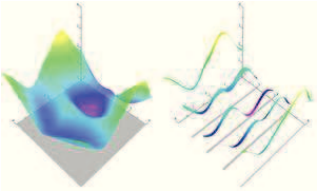
Marcel Campen, David Bommes, Leif Kobbelt
Dual Loops Meshing: Quality Quad Layouts on Manifolds [20]



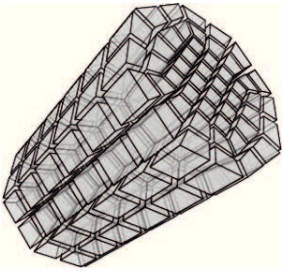
Henrik Zimmer, Marcel Campen, David Bommes, Leif Kobbelt
Rationalization of Triangle-Based Point-Folding Structures [70]



Henrik Zimmer, Marcel Campen, Ralf Herkrath, Leif Kobbelt
Variational Tangent Plane Intersection for Planar Polygonal Meshing [71]



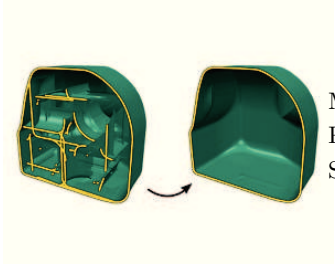
David Bommes, Henrik Zimmer, Leif Kobbelt
Practical Mixed-Integer Optimization for Geometry Processing [16]



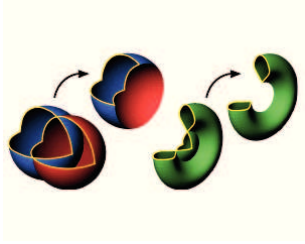
Michael Kremer, David Bommes, Leif Kobbelt
OpenVolumeMesh - A Versatile Index-Based Data Structure for 3D Polytopal Complexes [53]

David Bommes, Timm Lempfer, Leif Kobbelt
Global Structure Optimization of Quadrilateral Meshes [12]

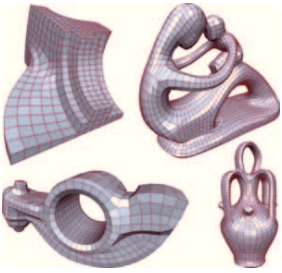
Marcel Campen, Leif Kobbelt
Walking On Broken Mesh: Defect-Tolerant Geodesic Distances and Parameterizations [25]



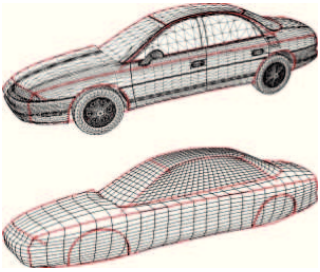
Marcel Campen, Leif Kobbelt
Polygonal Boundary Evaluation of Minkowski Sums and Swept Volumes [24]



Marcel Campen, Leif Kobbelt
Exact and Robust (Self-)Intersections for Polygonal Meshes [23]



David Bommes, Henrik Zimmer, Leif Kobbelt
Mixed-Integer Quadrangulation [15]



Ellen Dekkers, Leif Kobbelt, Richard Pawlicki, Randall C. Smith
A Sketching Interface for Feature Curve Recovery of Free-Form Surfaces [33]

8 Conclusion

In this thesis, we develop a versatile framework for geometry processing and rendering. We analyze the common workflows and requirements in existing geometry processing tasks as well as in existing software. Furthermore, we study how a software framework can be used for teaching. Based on this analysis we derive a set of software requirements for a general geometry processing framework.

In the implementation part we describe how these software requirements have been implemented in our OpenFlipper framework. We provide a software solution for all components of such a framework like the build system, abstractions from the operating system and hardware, and a highly flexible plug-in architecture to achieve a modular structure. Furthermore, we provide plug-ins for geometric data types, user interface creation and management, as well as software interfaces for various aspects of the framework, including batch processing, rendering, selection and many more.

Additionally, we show how software testing via continuous integration can help to significantly improve the code quality and allows us to constantly provide ready-to-use software packages. We also cover project management which is required to successfully create and maintain such a large and fast-growing framework.

We collect a lot of practical experience while creating and extending the framework. The most important aspect is that the interface has to be kept stable and backward compatible. By this, developers are guaranteed to always have a working version of their software. When it was unavoidable to change the interface to a new version, the framework has been deployed keeping the old interface version available. When launched, OpenFlipper will warn, that

a deprecated version of an interface has been used and ask the developer to update to the new version.

Furthermore, continuous integration proved to be a very efficient tool during development of algorithms in a research setting. The subsequent tests helped to uncover hidden bugs at an early development stage and prevent time consuming debugging when the code got larger.

For a high acceptance of such a framework, it shows to be inevitable to get new code into the framework as soon as possible because other researchers or users need new functionality just in time. With continuous integration and deployment we are able to provide a rolling release pattern reducing the time from new code to deployment on all architectures to only a few hours.

The plug-in architecture also proves to be useful for geometry processing. Many implementations reuse a large amount of existing code and algorithms creating new processing pipelines. Several tasks encountered during the development like creating a 3D printing pipeline were significantly speed up as a lot of geometry processing algorithms, input and output, selection and rendering were already available and could be used out of the box.

A large problem in the early phase of the project was the lack of documentation and examples. This was observed when comparing the framework with other projects at the EEFSW workshop [58]. We significantly improved the documentation and provided examples for every interface in the framework. To keep the documentation in a consistent state, all updates changing or adding interfaces and functionality were only allowed to enter a release, if they also contain corresponding documentation.

In summary, the framework helped a lot to streamline the research and development in our department. It is used in commercial projects to directly transfer research results to companies and provides a platform for a large number of publications. Furthermore, we used OpenFlipper in our practicals and lectures providing great results.

Moreover, a growing community of external users (companies, artists, private users) work with the framework to create their own products or use it

for geometry processing. They also support the framework development by submitting new algorithms or fixes back to the framework.

In conclusion, OpenFlipper proves to be a very useful framework for a large and diverse user group and will continue to provide a helpful research and publishing platform.

9 Outlook

OpenFlipper is a fast developing framework. There is always new code which has to be integrated into the system and new interfaces are added due to new experience collected in various projects or feedback by users.

From the technical point of view, the biggest challenge will be to support the new rendering system Vulkan [44]. It should provide better performance and a cleaner interface than the current OpenGL implementation. However, as we have the additional abstraction layer introduced to the data, we will be able to implement renderers for the new structure as additional plugins while keeping OpenGL compatibility via the existing plugins.

Furthermore, it would be great to integrate input and output plugins to directly control scanners or printers without the intermediate steps via files. For example Riegl has an interface library to directly control their scanner and retrieve the data produced by it on the fly. The user can directly see results and react to problems. The same holds for the Ultimaker2 which can be controlled via USB interface. The G-Code produced by a slicer could directly be send to the printer without having to use the indirection of writing to files the printer reads from

Beside these points, it always requires great effort to support the latest Qt version. Code paths can break or modules get deprecated or replaced. However, to keep such a large framework alive, these transitions have to be made as fast as possible while providing backward compatibility.

From a research point of view, it would be interesting to investigate an additional abstraction layer to the data types, such that the plug-ins retrieve e.g. points and faces from the data types in an identical way.

Also it would be great to investigate in more detail, how geometry processing and rendering algorithms can be parallelized. Current workstations offer more and more CPU cores which can and should be used to accelerate processing. However not all geometry processing algorithms are easily parallelizable, as the problems are usually strongly coupled and not local problems which could be distributed more easily.

Bibliography

[1] Cppcheck, A tool for static C/C++ code analysis. http://cppcheck.sourceforge.net.

[2] Cura, 3D Printer Software. https://ultimaker.com/en/products/cura-software.

[3] Git - A free and open source distributed version control system. www.git-scm.com.

[4] Jenkins, An extendable open source continuous integration server . http://jenkins-ci.org.

[5] Standard ECMA-262, ECMA Script Language Specification, 5th edition, December 2009.

[6] Project ALICE. Graphite is a research platform for computer graphics, 3D modeling and numerical geometry. http://alice.loria.fr/software/graphite/.

[7] Oscar Kin-Chung Au, Chiew-Lan Tai, Hung-Kuo Chu, Daniel Cohen-Or, and Tong-Yee Lee. Skeleton extraction by mesh contraction. In *ACM Transactions on Graphics (TOG)*, volume 27, page 44. ACM, 2008.

[8] L. Bavoil and K. Myers. Order Independent Transparency With Dual Depth Peeling. Technical report, NVIDIA Developer SDK 10, 2008.

[9] P. J. Besl and H. D. McKay. A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, February 1992.

[10] D. Bommes, B. Lévy, N. Pietroni, E. Puppo, C. Silva, M. Tarini, and D. Zorin. State of the Art in Quad Meshing. In *Eurographics STARS*, 2012.

[11] David Bommes, Marcel Campen, Hans-Christian Ebke, Pierre Alliez, and Leif Kobbelt. Integer-grid maps for reliable quad meshing. *ACM Transactions on Graphics (TOG)*, 32(4):98, 2013.

[12] David Bommes, Timm Lempfer, and Leif Kobbelt. Global structure optimization of quadrilateral meshes. In *Computer Graphics Forum*, volume 30, pages 375–384. Wiley Online Library, 2011.

[13] David Bommes, Bruno Lévy, Nico Pietroni, Enrico Puppo, Claudio Silva, Marco Tarini, and Denis Zorin. Quad-mesh generation and processing: A survey. In *Computer Graphics Forum*, volume 32, pages 51–76. Wiley Online Library, 2013.

[14] David Bommes, Henrik Zimmer, and Leif Kobbelt. CoMISo: Constrained Mixed-Integer Solver library.

[15] David Bommes, Henrik Zimmer, and Leif Kobbelt. Mixed-integer Quadrangulation. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, pages 77:1–77:10, New York, NY, USA, 2009. ACM.

[16] David Bommes, Henrik Zimmer, and Leif Kobbelt. Practical mixed-integer optimization for geometry processing. In *Curves and Surfaces*, pages 193–206. Springer, 2010.

[17] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. OpenMesh – a generic and efficient polygon mesh data structure. In *OpenSG Symposium*, 2002.

[18] Mario Botsch and Olga Sorkine. On linear variational surface deformation methods. *Visualization and Computer Graphics, IEEE Transactions on*, 14(1):213–230, 2008.

[19] Marcel Campen. *Quad Layouts–Generation and Optimization of Conforming Quadrilateral Surface Partitions*. PhD thesis, RWTH Aachen University, 2014.

[20] Marcel Campen, David Bommes, and Leif Kobbelt. Dual loops meshing: quality quad layouts on manifolds. *ACM Transactions on Graphics (TOG)*, 31(4):110, 2012.

[21] Marcel Campen, David Bommes, and Leif Kobbelt. Quantized global parametrization. *ACM Transactions on Graphics (TOG)*, 34(6):192, 2015.

[22] Marcel Campen, Martin Heistermann, and Leif Kobbelt. Practical anisotropic geodesy. In *Computer graphics forum*, volume 32, pages 63–71. Wiley Online Library, 2013.

[23] Marcel Campen and Leif Kobbelt. Exact and Robust (Self-) Intersections for Polygonal Meshes. In *Computer Graphics Forum (Proc. Eurographics)*, volume 29, pages 397–406, 2010.

[24] Marcel Campen and Leif Kobbelt. Polygonal boundary evaluation of minkowski sums and swept volumes. In *Computer Graphics Forum*, volume 29, pages 1613–1622. Wiley Online Library, 2010.

[25] Marcel Campen and Leif Kobbelt. Walking on broken mesh: Defect-tolerant geodesic distances and parameterizations. In *Computer Graphics Forum*, volume 30, pages 623–632. Wiley Online Library, 2011.

[26] Marcel Campen and Leif Kobbelt. Dual strip weaving: Interactive design of quad layouts using elastica strips. *ACM Transactions on Graphics (TOG)*, 33(6):183, 2014.

[27] Marcel Campen and Leif Kobbelt. Quad layout embedding via aligned parameterization. In *Computer Graphics Forum*, volume 33, pages 69–81. Wiley Online Library, 2014.

[28] Paolo Cignoni, Massimiliano Corsini, and Guido Ranzuglia. Meshlab: an open-source 3d mesh processing system. *Ercim news*, 73(45-46):6, 2008.

[29] Jorge Colazo and Yulin Fang. Impact of license choice on open source software development activity. *Journal of the American Society for Information Science and Technology*, 60(5):997–1011, 2009.

[30] Qt company. Qt cross-platform application and UI framework. http://www.qt.io.

[31] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality: the design and implementation of the CAVE. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 135–142, New York, NY, USA, 1993. ACM.

[32] Ellen Dekkers and Leif Kobbelt. Geometry seam carving. *Computer-Aided Design*, 46:120–128, 2014.

[33] Ellen Dekkers, Leif Kobbelt, Richard Pawlicki, and Randall C. Smith. A Sketching Interface for Feature Curve Recovery of Free-form Surfaces. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, pages 235–245, New York, NY, USA, 2009. ACM.

[34] Hans-Christian Ebke, David Bommes, Marcel Campen, and Leif Kobbelt. Qex: Robust quad mesh extraction. *ACM Trans. Graph.*, 32(6):168:1–168:10, November 2013.

[35] Hans-Christian Ebke, Marcel Campen, David Bommes, and Leif Kobbelt. Level-of-detail quad meshing. *ACM Transactions on Graphics*, 33(6), 2014.

[36] Free Software Foundation. GNU General Public License v2. http://www.gnu.org/licenses/gpl-2.0.

[37] Free Software Foundation. GNU General Public License v3. http://www.gnu.org/licenses/gpl-3.0.

[38] Free Software Foundation. GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html.

[39] Martin Fowler and Jim Highsmith. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[40] GNU Project Free Software Foundation (FSF). Automake - Tool for automatically generating Makefile.in files compliant with the GNU Coding Standards. https://www.gnu.org/software/automake/.

[41] Anne Gehre, Isaak Lim, and Leif Kobbelt. Adapting feature curve networks to a prescribed scale. *Proc. EUROGRAPHICS 2016*, 2016 forthcomming.

[42] Google. Google C++ Testing Framework. http://code.google.com/p/googletest.

[43] The Khronos Group. OpenGL. https://www.khronos.org/opengl/.

[44] The Khronos Group. Vulkan. https://www.khronos.org/vulkan.

[45] GitLab Inc. Gitlab - Code collaboration platform. www.gitlab.com.

[46] Dennis Kasprzyk. Diploma Thesis on Optimized User Interface for Geometry-Algorithms. Master's thesis, RWTH Aachen University, 2009.

[47] Ladislav Kavan, Steven Collins, Jiri Zara, and Carol O'Sullivan. Skinning with dual quaternions. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 39–46. ACM, 2007.

[48] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, volume 7, 2006.

[49] Kitware. CMake, an open-source, cross-platform family of tools designed to build, test and package software. http://cmake.org/.

[50] Reinhard Klein, Gunther Liebich, and Wolfgang Straßer. Mesh reduction with error control. In *Visualization'96. Proceedings.*, pages 311–318. IEEE, 1996.

[51] Leif Kobbelt. sqrt(3)-subdivision. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 103–112. ACM Press/Addison-Wesley Publishing Co., 2000.

[52] Leif Kobbelt, Swen Campagna, and Hans-Peter Seidel. A general framework for mesh decimation. In *Graphics interface*, volume 98, pages 43–50, 1998.

[53] Michael Kremer, David Bommes, and Leif Kobbelt. OpenVolumeMesh - A Versatile Index-Based Data Structure for 3D Polytopal Complexes. In Xiangmin Jiao and Jean-Christophe Weill, editors, *Proceedings of the 21st International Meshing Roundtable*, pages 531–548, Berlin, 2012. Springer-Verlag.

[54] Michael Kremer, David Bommes, Isaak Lim, and Leif Kobbelt. Advanced automatic hexahedral mesh generation from surface quad meshes. In *To appear in: Proceedings of the 22nd International Meshing Roundtable*, Berlin, 2013. Springer-Verlag.

[55] John P Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 165–172. ACM Press/Addison-Wesley Publishing Co., 2000.

[56] Max Lyon, David Bommes, and Leif Kobbelt. Hexex: Robust hexahedral mesh extraction. *Proc. EUROGRAPHICS 2016*, 2016 forthcomming.

[57] Jan Möbius. Geometry Detail Transfer based on Rotation-Invariant Mesh Coordinates. Diplomarbeit, RWTH-Aachen University, Germany, July 2006.

[58] Jan Möbius and Leif Kobbelt. Openflipper: An open source geometry processing and rendering framework. In Bernhard Rumpe and Horst Lichter, editors, *Entwicklung und Evolution von Forschungssoftware*, pages 31–36. Shaker, 2011.

[59] Jan Möbius and Leif Kobbelt. Openflipper: An open source geometry processing and rendering framework. In Jean-Daniel Boissonnat, Patrick Chenin, Albert Cohen, Christian Gout, Tom Lyche, Marie-Laurence Mazure, and Larry Schumaker, editors, *Curves and Surfaces*, volume 6920 of *Lecture Notes in Computer Science*, pages 488–500. Springer Berlin / Heidelberg, 2012.

[60] Jan Möbius, Michael Kremer, and Leif Kobbelt. Openflipper-a highly modular framework for processing and visualization of complex geometric models. In *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2013 6th Workshop on*, pages 25–32. IEEE, 2013.

[61] Jonas Nagy-Kuhlen. Realistic Real-time Fabric Rendering. Bachelor thesis, RWTH-Aachen University, Germany, September 2014.

[62] Institut national de recherche en informatique et en automatique. Geogram: A programming library of geometric algorithms. http://alice.loria.fr/software/geogram.

[63] Institut national de recherche en informatique et en automatique. Vorpaline: 3D mesh generation software. http://alice.loria.fr/index.php/erc-vorpaline.html.

[64] Darko Pavic, Volker Schönefeld, Lars Krecklau, Martin Habbecke, and Leif Kobbelt. 2D Video Editing for 3D Effects. In Oliver Deussen, Daniel A. Keim, and Dietmar Saupe, editors, *VMV*, pages 389–398. Aka GmbH, 2008.

[65] C Michael Pilato, Ben Collins-Sussman, and Brian W Fitzpatrick. *Version control with subversion.* "O'Reilly Media, Inc.", 2008.

[66] Markus Probst. *Robust Shape Optimization for Incompressible Flow of Shear-Thinning Fluids*. Phd thesis, RWTH-Aachen University, Germany, 2013.

[67] Pedro V. Sander, Diego Nehab, and Joshua Barczak. Fast Triangle Reordering for Vertex Locality and Reduced Overdraw. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3), August 2007.

[68] The GTK+ Team. GTK - Multi-platform toolkit for creating graphical user interfaces. http://www.gtk.org/.

[69] Berkeley University of California. The BSD 3-Clause License. https://opensource.org/licenses/BSD-3-Clause.

[70] Henrik Zimmer, Marcel Campen, David Bommes, and Leif Kobbelt. Rationalization of triangle-based point-folding structures. In *Computer Graphics Forum*, volume 31, pages 611–620. Wiley Online Library, 2012.

[71] Henrik Zimmer, Marcel Campen, Ralf Herkrath, and Leif Kobbelt. Variational tangent plane intersection for planar polygonal meshing. In *Advances in Architectural Geometry 2012*. Springer, 2012.

[72] Henrik Zimmer, Marcel Campen, and Leif Kobbelt. Efficient computation of shortest path-concavity for 3d meshes. June 2013.

[73] Henrik Zimmer and Leif Kobbelt. Zometool rationalization of freeform surfaces. *Visualization and Computer Graphics, IEEE Transactions on*, 20(10):1461–1473, 2014.

[74] Henrik Zimmer, Florent Lafarge, Pierre Alliez, and Leif Kobbelt. Zometool shape approximation. *Graphical Models*, 76(5):390–401, 2014.

[75] Jason Zink, Matt Pettineo, and Jack Hoxley. *Practical rendering and computation with Direct3D 11*. CRC Press, 2011.

Curriculum Vitae

Jan Möbius

E-Mail	jan.moebius@rwth-aachen.de
Date of Birth	09.07.1980
Place of Birth	Viersen, Germany
Citizenship	German

Academic Education

Apr. 2006 – Feb. 2017	Doctoral Student at RWTH Aachen University, Visual Computing Institute Degree: Dr. rer. nat. Supervisor: Prof. Dr. Leif Kobbelt
Oct. 2000 – Mar. 2006	Computer Science Studies at RWTH Aachen University Degree: Dipl. Inform. Supervisor: Prof. Dr. Leif Kobbelt

Publications

Jan Möbius, Leif Kobbelt: *OpenFlipper: An Open Source Geometry Processing and Rendering Framework*. In Bernhard Rumpe, Horst Lichter, editors, *Entwicklung und Evolution von Forschungssoftware*, pages 31-36, Rolduc, 2011. Shaker-Verlag

Jan Möbius, Leif Kobbelt: *OpenFlipper: An Open Source Geometry Processing and Rendering Framework*, Proceedings of the 7th International Conference on Curves and Surfaces 2012

Jan Möbius, Michael Kremer, Leif Kobbelt: *OpenFlipper - A Highly Modular Framework for Processing and Visualization of Complex Geometric Models*, 6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS) 2013

Statement of Originality

Several components, ideas and implementations of the presented software framework have been influenced by discussions and feedback provided by the members of the Computer Graphics Group led by Professor Dr. Leif Kobbelt. In what follows I will detail my contributions to articles that are relevant for this thesis and which previously have been published.

[58] As the main developer of the framework I designed its structure, developed the main components and implemented almost all modules of the presented system. I was responsible for implementing the presented application and for evaluating the results. I was the main author writing the report.

[59] As the main developer of the framework I designed its structure, developed the main components and implemented almost all modules of the presented system. I was responsible for implementing the presented application and for evaluating the results. I was the main author writing the paper.

[60] As the main developer of the framework I designed its structure, developed the main components and implemented most of the modules of the presented system. I was responsible for implementing the presented application and for evaluating the results. I was the main author writing the paper.