Variational Methods for Discrete Surface Parameterization. Applications and Implementation.

vorgelegt von
Dipl.-Math. techn. Stefan Sechelmann

von der Fakultät II - Mathematik und Naturwissenschaften
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
– Dr. rer. nat. –

Promotionsausschuss

Vorsitzender:	NN
Gutachter/Berichter:	Prof. Dr. Alexander I. Bobenko
	NN

Tag der wissenschaftlichen Aussprache: NN

Berlin, den 07. Januar 2013

Contents

List of Figures

Introduction

Part I

Uniformization of discrete Riemann surfaces

Part II

Variational Methods for Discrete Surface Parameterization

Part III

Software Packages

Chapter 1

Introduction

In this chapter words printed in SMALLCAPITALS are names of software packages, words printed in `TeleType` are names of JAVA classes, methods, or fields.

1.1 Mathematical Software Development

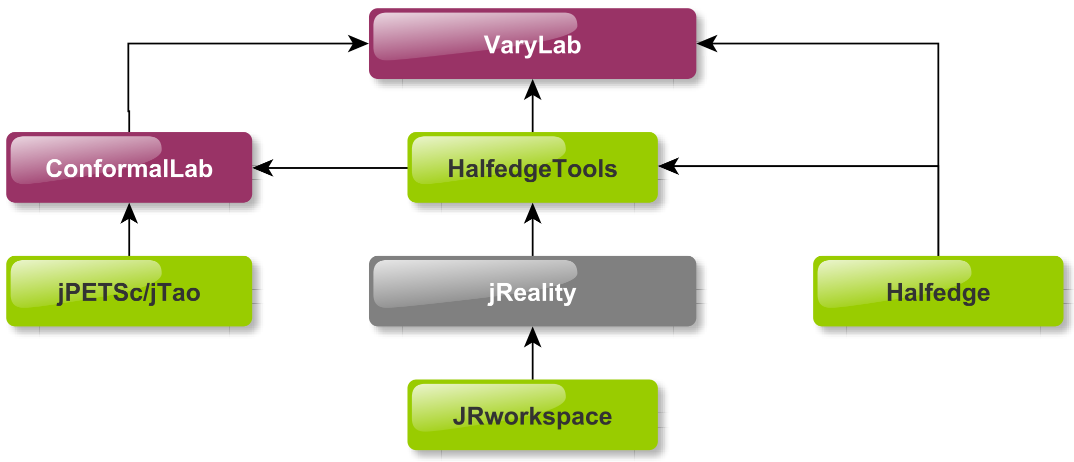

Figure 1.1: Software architecture and dependencies of the DDG Framework. JTEM library packages (green), mathematical software packages (red).

In the field of Discrete Differential Geometry (DDG) there is a special need for experiments conducted with the help of computer software. Especially if the methods of DDG are applied to problems in computer graphics, geometry processing, or architecture, algorithms have to be implemented and convincing examples have to be presented. Additionally a suitable visualization of the results has to be included in a state-of-the-art publication.

There is a growing knowledge of software development in the mathematical community. This is party due to the curricula of universities which started to include programming courses for undergraduate students with a visualization emphasis, e.g., [15, 12]. This knowledge enables students to extend their abilities of creating visualizations and mathematical software, where former generations of students solely used the visualization abilities of standard computer algebra packages like Mathematica or MatLab.

The audience of this chapter is two-fold. On the one hand it is students creating their visualizations of surfaces and develop algorithms. On the other hand it is researchers in the field of discrete differential geometry who need a stable data structure and programming infrastructure to get the job done.

This Chapter is the description and getting-started manual of a set of software packages (Berlin DDG Framework) written in the programming language Java. They are specifically designed for the creation of custom interactive software for experiments with algorithms and geometries treated within DDG. It is currently beeing used for teaching a mathematical visualization course at TU-Berlin [15] and for research projects within the geometry group.

Section 1.2 gives an overview of existing software packages that have a focus similar to the DDG Framework. Section 2 introduces the JRworkspace library of the Jtem project [2]. It is the foundation of any application created with the DDG Framework. It is also the user interface basis of Jreality, a mathematical visualization library that uses JRworkspace as plug-in and user interface tool [1]. Section 3 introduces the Halfedge and HalfedgeTools package. They implement half-edge data structure and various user interface tools and algorithms for interaction and editing. In Section 4 we describe the software ConformalLab. This package implements the methods of the publications [9, 27, 29, 10]. Section 5 introduces VaryLab, the software implementation of the methods described in the publications [20, 21, 29]. This package is also released to partners of the development group as VaryLab[Gridshells], VaryLab[Ultimate], or even online as VaryLab[Service][28].

Figure 1.1 shows the dependencies of the packages. Every application depends on JRworkspace which implements plug-in functionality. It is the basis of the Jreality plug-in system. HalfedgeTools is using Jreality for visualization and is build on top of the Jtem project Halfedge. ConformalLab and VaryLab use Jpetsc/Jtao to perform numerical optimization. Their algorithms are implemented as JRworkspace plug-ins.

The development of the described software is joint work with Thilo Rörig (HalfedgeTools, VaryLab), the Jreality members [1], Hannes Sommer (Jpetsc/Jtao) [32], Ulrich Pinkall and Paul Peters (JRworkspace), and Boris Springborn (Halfedge).

1.2 Related Work

JavaView, CGAL, ...

1.3 CD Content

Chapter 2

JRworkspace - Java API for modular applications

JRworkspace is part of the Jtem family of software projects [2]. It defines a simple API to create modular Java applications. This API consists of three basic classes (Listings 2.2, 2.3, and 2.4). The project contains a reference implementation that supports the creation of Java Swing applications using the JRworkspace API. This implementation is used in all applications described in this work.

2.1 Plug-ins and the controller

In a JRworkspace application a feature is implemented as plug-in and the corresponding Java class extends the abstract class Plugin (Listing 2.2). The idea is that a plug-in can be installed by the controller calling its `install` method or uninstalled via the `uninstall` method. You can think of it as a feature added to your program. In particular there is no more than one instance of a plug-in class in a JRworkspace application.

A plug-in has a life-cycle during the runtime of the program which includes these basic steps:

instantiation	1	set default plug-in state
`restoreStates`	2	load plug-in state from `Controller`
`install`	3	calls `getPlugin` to obtain dependent plug-ins
-	4	program execution
`storeStates`	5	stores state values to the `Controller`
(`uninstall`	6	clean up)

Step 1 instantiates a plugin and initializes its default properties. In step 2 the controller calles the `restoreStates` method. Step 3 is the actual installation of the plug-in. During runtime of the application the plug-in can interact with possible user interface it created during installation or offer services to other plug-ins. Before program termination or before uninstall the `storeStates` method is called. The plug-in is supposed to store its state values by calling the `storeProperty` method of the controller. Inter-plug-in-communication is done via the `getPlugin` method of the controller. A plug-in should call `getPlugin` from within the `install` method to obtain the unique instance of a dependent plug-in. The `getPlugin` method always returns the same instance of a plug-in so its result can be stored by the `install` method for later reference, see for example Listing 2.1. Step 6 `uninstall` is only used with dynamic plug-ins that support this operation. An implementation of `Controller` may not support uninstallation of plug-ins.

We describe the basic API usage from a programmers point of view by giving an example plug-in in Listing 2.1 and the source code of the three basic API classes `Plugin`, `Controller`, and `PluginInfo` in Listings 2.2, 2.3, and 2.4.

```java
public class MyPlugin extends Plugin {
    private DependentPlugin dependency = null;
    private double doubleState = 0.0;

    public void helloPlugin() {
        String depName = dependency.getPluginInfo().name;
        System.out.println("I am a plug-in. I depend on " + depName);
    }
    @Override
    public void storeStates(Controller c) throws Exception {
        c.storeProperty(MyPlugin.class, "doubleState", doubleState);
    }
    @Override
    public void restoreStates(Controller c) throws Exception {
        doubleState = c.getProperty(MyPlugin.class, "doubleState", 1.0);
    }
    @Override
    public void install(Controller c) throws Exception {
        dependency = c.getPlugin(DependentPlugin.class);
    }
}
```

Listing 2.1: A simple plug-in class. It depends on a plug-in called `DependentPlugin` and has the property `doubleState`. It provides the method `helloPlugin()` that prints some message. In the `storeStates` method the value of `doubleState` is written to the controller. The class `MyPlugin` is used as context class. The name of this class is used as name space to avoid property name ambiguities. The value of `doubleState` is read from the controller in the `restoreStates` method using the same context class and property name as in `storeStates`. If there is no value with the given context and name the default value `1.0` is returned by the `getProperty` method.

```java
public abstract class Plugin {

    public PluginInfo getPluginInfo() {
        return PluginInfo.create(getClass());
    }

    public void install(Controller c) throws Exception{}
    public void uninstall(Controller c) throws Exception {}
    public void restoreStates(Controller c) throws Exception {}
    public void storeStates(Controller c) throws Exception {}

    @Override
    public String toString() {
        if (getPluginInfo().name == null) {
            return "No Name";
        } else {
            return getPluginInfo().name;
        }
    }

    @Override
```

```
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        } else {
            return getClass().equals(obj.getClass());
        }
    }
```

Listing 2.2: The `Plugin` base class (excerpt). Note that plug-ins are equal if their classes are. It is not supported to have multiple instances of the same plug-in class installed.

```
public interface Controller {

    public <T extends Plugin> T getPlugin(Class<T> clazz);
    public <T> List<T> getPlugins(Class<T> pClass);
    public Object storeProperty(Class<?> context, String key, Object property);
    public <T> T getProperty(Class<?> context, String key, T defaultValue);
    public <T> T deleteProperty(Class<?> context, String key);
    public boolean isActive(Plugin p);

}
```

Listing 2.3: The `Controller` interface. A plug-in can obtain other plug-in instances by calling `getPlugin` which returns a unique instance of the given plug-in class. The semantics of the `getPlugins` methods is different. It returns all plug-ins that are already known to the controller so no new dependencies are created by calling `getPlugins`. Property handling is done via the `xxProperty` methods. Note that any `Object` can be used as property value. This requires the controller to use generic serialization to store data. It is strongly discouraged to use other classes than official java API classes as stored values as deserialization may fail if class geometry changes.

```
public class PluginInfo {

    public String name = "unnamed";
    public String vendorName = "unknown";
    public String email = "unknown";
    public Icon icon = null;
    public URL documentationURL = null;
    public boolean isDynamic = true;

    public PluginInfo() {
    }

    public PluginInfo(String name) {
        this.name = name;
    }

    public PluginInfo(String name, String vendor) {
        this(name);
        this.vendorName = vendor;
    }

    public static PluginInfo create(Class<?> pluginClass) {
        PluginInfo pi;
        if (pluginClass == null) {
            pi = new PluginInfo();
        } else {
            pi = new PluginInfo(pluginClass.getSimpleName());
        }
        if (pluginClass != null &&  pluginClass.getPackage() != null) {
```

```
            pi.vendorName = pluginClass.getPackage().getImplementationVendor();
        }
        return pi;
    }

}
```

Listing 2.4: The plug-in meta data class (excerpt). Instances are returned by the `getPluginInfo` method of any plug-in. The value of the `name` field is a plaintext name that could be shown in a user interface as well are the `vendorName` and `email` information. An optional `icon` and a `documentationURL` can be given. The flag `isDynamic` is evaluated by controller implementations that support deinstallation of plug-ins. A dynamic plug-in can be installed or uninstalled during application runtime. A non-dynamic plug-in must be installed at startup and remains installed until program termination. The static `create` method returns a default `PluginInfo` instance for the given plug-in class.

In the next section we describe an implementation of this API.

2.2 Reference implementation - `SimpleController`

This section describes a reference implementation of the JRworkspace plug-in API. It was started as a user interface framework for Jreality [1]. It implements the `Controller` interface in a class called `SimpleController`. This name is historic and did not change as the features evolved from simple into quite complex. `SimpleController` implements a Java Swing® framework for the creation of complex modular applications based on the JRworkspace API. It defines various plug-in flavors that define user interface features. The implementation does not support dynamic plug-ins.

In the remainder of this section I describe the basic and most interesting features of this implementation. For a complete API reference see the documentation on the Jtem website [2].

Perspective Flavor

A plug-in implementing the interface `PerspectiveFlavor` provides the base for a program's user interface. It implements the method `getCenterComponent` that returns a AWT `Conponent` that is placed in the main frame of the application. The main program window itself is created and managed by the controller. A reference implementation of this flavor is the `SideContainerPerspective`. It layouts its content using a `BorderLayout` and places slots in the north, south, east, and west of the main window. These slots can contain `ShrinkPanels` that can be moved between slots by drag-and-drop. A `ShrinkPanel` behaves like a `JPanel` and has a title bar that resizes the panel when the user clicks with the mouse.

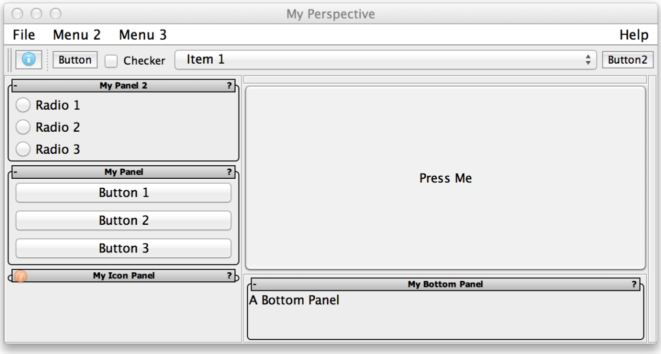

Figure 2.1: The `SideContainerPerspective` implementation uses slots to layout panels at the side of the main window. The left slot contains three `ShrinkPanels`, the top and right slots are empty. The menu bar and tool bar are created by respective plug-in flavors.

Menu Flavor

A plug-in implementing the `MenuFlavor` interface provides JAVA SWING® menu components that are placed at the top of the main window. A reference implementation of this flavor is the plug-in `MenuAggregator` that manages menu entries by contexts and menu paths. Its API provides four methods to add and remove menus, menu items, and separators. A typical method signature is

```
public void addMenu(Class<?> ctx, double priority, JMenu m, String... path)
```

where the plug-in stores the menu item with the given context class. This context is used to bulk-remove menus from a menu aggregator. Menus are sorted ascending by their priority. The menu item appears at the end of the given menu path. See Listing 2.5 and Figure 2.2.

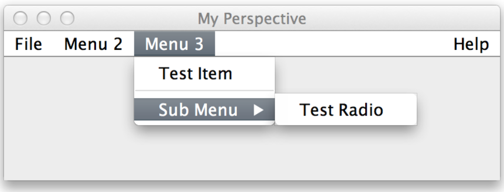

Figure 2.2: The menu created by Listing 2.5

```
    @Override
    public void install(Controller c) throws Exception {
        super.install(c);
        addMenu(MyMenuBar.class, 0.0, new JMenu("File"));
        addMenu(MyMenuBar.class, 1.0, new JMenu("Menu 2"));
        addMenuItem(MyMenuBar.class, 0.0, new QuitAction(), "File");
        addMenuItem(MyMenuBar.class, 0.0, new JCheckBoxMenuItem("Test Checker"), "
            Menu 2");
        addMenuItem(MyMenuBar.class, 0.0, new JRadioButtonMenuItem("Test Radio"), "
            Menu 3", "Sub Menu");
        addMenuItem(MyMenuBar.class, 0.0, new JMenuItem("Test Item"), "Menu 3");
        addMenuSeparator(MyMenuBar.class, 1.0, "Menu 3");
```

```
    }
```

Listing 2.5: Usage of the `MenuFlavor` interface and the `MenuAggregator` implementation.

Tool Bar Flavor

Plug-ins implementing this flavor interface create a Java Swing® tool bar at the top of the main window. There can be more than one plug-in implementing this interface to create multiple tool bars. The API method signatures a similar to the signatures of the menu aggregator flavor. As a tool bar does not have a hierarchy, there is no path parameter. The signature of a API method is, e.g.,

```
public void addAction(Class<?> context, double priority, Action a).
```

The tool bar aggregator implementation can handle `Actions`, `Components`, and tool bar separators. See Listing 2.6 and Figure 2.4.

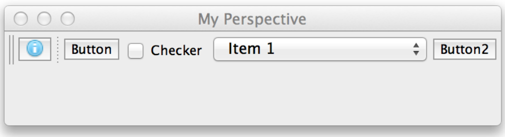

Figure 2.3: The menu bar created by Listing 2.6

```
    @Override
    public void install(Controller c) throws Exception {
        addAction(MyToolBar.class, 0.0, new MyAction());
        addTool(MyToolBar.class, 2.0, new JButton("Button"));
        addSeparator(MyToolBar.class, 1.0);
        addTool(MyToolBar.class, 3.0, new JCheckBox("Checker"));
        addTool(MyToolBar.class, 4.0, new JComboBox(testItems));
        addTool(MyToolBar.class, 5.0, new JButton("Button2"));
        super.install(c);
    }
```

Listing 2.6: Usage of the `ToolFlavor` interface and the `ToolBarAggregator` implementation.

The API of `SimpleController`

A plug-in implementation is independent of the concrete implementation of the `Controller`. To create an application with the `SimpleController` we need to register plug-ins we want to use and then invoke the startup sequence. A typical `main` method is, e.g.,

```
    public static void main(String[] args) throws Exception {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        SimpleController c = new SimpleController("My Application");
        c.setManageLookAndFeel(false);
        c.setPropertiesMode(StaticPropertiesFile);
        c.setStaticPropertiesFile(new File("MyApp.xml"));
        c.registerPlugin(MyPerspective.class);
        c.registerPlugin(MyMenuBar.class);
        c.registerPlugin(MyToolBar.class);
        c.registerPlugin(MyShrinkPanel.class);
        c.registerPlugin(MyShrinkPanel2.class);
        c.registerPlugin(MyShrinkPanel3.class);
        c.registerPlugin(MyShrinkPanel4.class);
```

```
        c.startup();
    }
```

Listing 2.7: `Main` method of a program created with the `SimpleController` implementation. The result is shown in Figure 2.1.

We set the look and feel to be the system look and feel, in this case the Mac OS style. Then we create a `SimpleController` and set the `magageLookAndFeel` property to `false`. Plug-in properties are saved to a file called `MyApp.xml`. There are two property modes defined by the `SimpleController`, `StaticPropertiesFile` and `UserPropertiesFile`. In static mode there is only one file location. In user mode the predefined location can be altered by the user of the program. This user decision is then stored as a JAVA preference. In this example we use the static properties.

2.3 JRWORKSPACE and JREALITY

The user interface of JREALITY [1] is based on the JRWORKSPACE API and reference implementation.

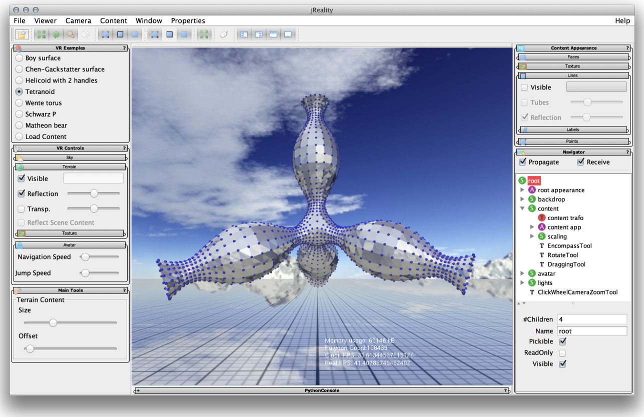

Figure 2.4: The JREALITY user interface. A `SideContainerPerspective` and `ShrinkPanels` is used. The tool bar and menu bar is created by the aggregators described in Section 2.2. This application uses a set of predefined user interface features and virtual reality components. In a custom application the developer usually registeres only a subset of these featurs.

A custom JREALITY application is based on the `JRViewer` class. This class uses `SimpleController` to manage plug-in registration and start-up.

Chapter 3

The JTEM libraries HALFEDGE and HALFEDGETOOLS

This chapter describes the implementation of a half-edge data structure and a set of tools contained in the package HALFEDGETOOLS. Both packages are part of the JTEM project [2]. This is joint work with Boris Springborn (HALFEDGE), Thilo Rörig, Felix Knöppel, and Kristoffer Josefsson (HALFEDGETOOLS), and other contributers to the JTEM project.

This particular implementation of a half-edge data structure was inspired by an implementation contained in the CGAL library [3].

3.1 The HALFEDGE data structure

This section is taken from the documentation of the package.

Cell decompositions of surfaces

Half-edge data structures are used primarily to represent cell decompositions of oriented surfaces. We say "primarily" because half-edge data structures can be used to represent somewhat more general combinatorial structures, such as, for example, a checker board surface with white squares removed.

Here, surface means two-dimensional manifold, possibly with boundary; and a cell decomposition of a surface is a graph embedded in the surface such that the complement of the graph is (topologically) a disjoint union of open disks. The term map on a surface means the same. Thus, a cell decomposition decomposes a surface into vertices, edges, and faces.

Regular and strongly regular

A cell decomposition of a surface is called regular, if it has no loops (edges with the same vertex on both ends), and if the boundary of a face contains an edge or vertex at most once. It is called strongly regular if two edges have at most one vertex in common, and if two faces have at most one edge or one vertex in common. A strongly regular cell decomposition is usually called a mesh.

This half-edge data structure implementation

This half-edge data structure implementation consists of different types of objects representing vertices, half-edges, and faces. The term half-edge can and should be thought of as synonymous with oriented edge or directed edge.

Every half-edge object holds references to:

- its oppositely oriented companion edge
- the next edge in the boundary of the face on its left hand side
- the previous edge in the boundary of the face on its left hand side
- the face on its left hand side
- the vertex it points to.

The face and vertex objects hold back references to a half-edge referencing them. Finally, there is the class `de.jtem.halfedge.HalfEdgeDataStructure` representing a whole half-edge data structure. It acts as a container for (and sort of factory of) its vertices, edges, and faces.

Use of generics

Typically, one wants to equip vertices, edges, and faces with additional properties or functionality. For example, vertices may have coordinates associated with them, edges may have weights, and faces may have colors.

Our half-edge data structure facilitates this by using generic classes as abstract base classes for vertex, edge, and face types: The classes `de.jtem.halfedge.Vertex`, `de.jtem.halfedge.Edge`, `de.jtem.halfedge.Face` are all parameterized with the associated vertex, edge, and face types.

Example

To create a half-edge data structure with vertices that have 2D coordinates, proceed as follows.

- Step 1. Define appropriate subclasses of `de.jtem.halfedge.Vertex`, `de.jtem.halfedge.Edge`, and `de.jtem.halfedge.Face`, for example:

```
public class MyVertex extends Vertex<MyVertex, MyEdge, MyFace> {
    public Point2D p;
}
public class MyEdge extends Edge<MyVertex, MyEdge, MyFace> { }
public class MyFace extends Face<MyVertex, MyEdge, MyFace> { }
```

  Of course you might make the property `p` of `MyEdge` private and provide getter and setter methods, etc. Note that you always have to subclass `de.jtem.halfedge.Vertex`, `de.jtem.halfedge.Edge`, and `de.jtem.halfedge.Face`, even if you do not define any additional functionality or properties.

- Step 2. Instantiate a `de.jtem.halfedge.HalfEdgeDataStructure`:

```
HalfEdgeDataStructure heds = new HalfEdgeDataStructure(MyVertex.class, MyEdge.
    class, MyFace.class);
```

  The parameters of the constructor serve as run time type tokens. Alternatively you can create a subclass of `de.jtem.halfedge.HalfEdgeDataStructure` and create an instance of this:

```
public class MyHDS extends HalfEdgeDataStructure<MyVertex, MyEdge, MyFace> {
    public MyHDS() {
        super(MyVertex.class, MyEdge.class, MyFace.class);
    }
}
...
MyHDS mds = new MyHDS();
```

- Step 3. Instantiate vertices, edges, and faces using the `addNewVertex`, `addNewEdge`, and `addNewFace` methods, like this:

```
MyVertex v = heds.addNewVertex();
MyEdge e = heds.addNewEdge();
MyFace f = heds.addNewFace();
```

3.2 Data, Algorithms and Tools

A set of algorithms and tools is implemented in the JTEM project HALFEDGETOOLS [2].

Many algorithms in the library are purely combinatorial. This means there is no extra data involved during algorithm execution. Thus such an algorithm is generic by definition. The method signature could look like this:

```
public static <
    V extends Vertex<V, E, F>,
    E extends Edge<V, E, F>,
    F extends Face<V, E, F>,
    HDS extends HalfEdgeDataStructure<V, E, F>
> void triangulate(HDS hds){
    ...
}
```

This method works on any half-edge data structure that is either an instance of `de.jtem.halfedge.HalfEdgeDataStructure` or an instance of a sub-class. This method signature makes the algorithm code itself look very clean. For instance iterating over all vertices amounts to:

```
for (V v : hds.getVertices()) {
    E e = v.getIncomingEdge();
    ...
}
```

On the other hand when designing a generic algorithm that needs certain data associated with nodes we have basically two options. Option 1. requires the generic node classes to implement the required interfaces:

```
public static <
    V extends Vertex<V, E, F> & HasCoordinate3D,
    E extends Edge<V, E, F>,
    F extends Face<V, E, F>,
    HDS extends HalfEdgeDataStructure<V, E, F>
> void convexHull(HDS hds){
    ...
    Coordinate3D x = v.getCoordinate3D();
}
```

This forces the Vertex implementations that use this algorithm to implement an interface called `HasCoordinate3D`. It leads to explicit and clean code of the algorithm. A drawback of this is that an existing implementation that should use this algorithm has to be adapted to implement the possibly many interfaces required by the algorithm. This is not a feasable solution when is comes to a modular application where algorithms come as plug-ins without the chance to change the data structure.

AdapterSet and Adapters

The second option uses the concept of adapters and is implemented in the package `de.jtem.halfedgetools.adapter`. An adapter defines a map from nodes to a data type supported by the adapter. We first show how this concept works when designing algorithms and then describe the implementation of the required adapters.

In this next example we calculate the discrete Dirichlet energy of a double valued function with double valued weights on edges.

```
1  public static <
2      V extends Vertex<V, E, F>,
3      E extends Edge<V, E, F>,
4      F extends Face<V, E, F>,
5      HDS extends HalfEdgeDataStructure<V, E, F>
6  > double computeDirichlet(HDS hds, AdapterSet a){
7      double energy = 0.0;
8      for (E e : hds.getPositiveEdges()) {
9          V s = e.getStartVertex();
10         V t = e.getTargetVertex();
11         double fStart = a.get(FunctionValue.class, s, Double.class);
12         double fTarget = a.get(FunctionValue.class, t, Double.class);
13         double w = a.getDefault(Weight.class, e, 1.0);
14         double d = fStart - fTarget;
15         energy += w * d * d;
16     }
17     return energy;
18 }
```

Listing 3.1: Algorithm that uses data from the `AdapterSet`. The `get` method of the `AdapterSet` takes the data class type to find a matching adapter (Line 11 and 12). The corresponding `getDefault` method takes a default value that is returned if no matching adapter is found (Line 13). No data type class is needed in the case.

This method requires the `AdapterSet` to contain adapters that provide `FunctionValue` data on vertices (Line 11 and 12) and `Weight` data on half-edges (Line 13).

The classes `FunctionValue` and `Weight` are runtime annotation classes, e.g.,

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionValue {}
```

A adapter class annotated with this annotation serves as `FunctionaValue` data adapter when called for as in Line 11-13 of Listing 3.4. There are three basic classes that could serve as base class of an adapter. It is

- `Adapter` - The abstract base class of all adapters. Should never be subclassed directly.
- `AbstractAdapter` - A adapter class that knows about the supported data type and implements all getter and setter methods. Here only the needed methods can be overwritten. Node type checking is done manually thus allows for the creation of generic adapters.
- `AbstractTypedAdpter` - If you know which half edge node classes the adpter is supposed to work with this is the adapter base class you should use. Node type checking and casting is done in the super class.

An adapter implementation using the `AbstractAdapter` is for instance:

```
@FunctionValue
public class MyAbstractAdapter extends AbstractAdapter<Double> {
    private Map<Vertex<?, ?, ?>, Double> valueMap = null;

    public MyAbstractAdapter() {
        super(Double.class, true, false);
    }
```

```
9       @Override
10      public <
11          N extends Node<?, ?, ?>
12      > boolean canAccept(Class<N> nodeClass) {
13          return Vertex.class.isAssignableFrom(nodeClass);
14      }

16      @Override
17      public <
18          V extends Vertex<V, E, F>,
19          E extends Edge<V, E, F>,
20          F extends Face<V, E, F>
21      > Double getV(V v, AdapterSet aSet) {
22          return valueMap.get(v);
23      }

25      @Override
26      public <
27          V extends Vertex<V, E, F>,
28          E extends Edge<V, E, F>,
29          F extends Face<V, E, F>
30      > void setV(V v, Double value, AdapterSet aSet) {
31          valueMap.put(v, value);
32      }
33  }
```

Listing 3.2: Adapter implementation using the `AbstractAdapter` as base class and a map as storage concept for double values on generic vertices. It is annotated with a `FunctionValue` annotation to serve as provider for data in the `AdapterSet` (Line 1). The super class `AbstractAdapter` is parameterized with the data type of the implementation; In this case `Double` (Line 1). The super class constructor is invoked with the class object of this type and flags that tell the adapter if get and/or set operations are permitted (Line 6). The method `canAccept` decides whether the adapter can work with the given node class; In this case the adapter can accept any `Vertex` object (Line 12). Vertex getter and setter methods are generic methods (Line 16 to 32).

When writing the adapter for concrete node classes we have a more concise description:

```
@FunctionValue
public class MyTypedAdapter extends AbstractTypedAdapter<VV, VE, VF, Double> {
    public MyTypedAdapter() {
        super(VV.class, null, null, Double.class, true, true);
    }

    @Override
    public Double getVertexValue(VV v, AdapterSet aSet) {
        return v.value;
    }

    @Override
    public void setVertexValue(VV v, Double value, AdapterSet aSet) {
        v.value = value;
    }
}
```

Listing 3.3: An adapter using `AbstractTypedAdpter` as base class. Also annotated with the `FunctionValue` annotation. This super class is parameterized with a set of node class implementations and the adapter data type. Here the super constructor takes the node class objects or `null` if a node typep is not supported, the data type class object, and the getter/setter flags. The vertex getter and setter methods are not generic and the corresponding casting is done in the super class.

Using this concept of typed adapters the usage of an algorithm amounts to the implementation of required data adapters and the creation of a suitable `AdapterSet`.

```
public double calculate() {
    VHDS hds = new VHDS();
    AdapterSet a = new AdapterSet();
    a.add(new MyTypedAdapter());
    return computeDirichlet(hds, a);
}
```

Listing 3.4: Usage example of the algorithm presented in Listing 3.4. In this example an empty data structure is created and processed by the algorithm. The `AdapterSet` contains the `FunctionValue` annotated adapter to provide `double` values on vertices.

3.3 HalfEdgeTools and JReality

The HalfEdgeTools package contains utility classes for the visualization of half-edge data with JReality [1]. It is part of the JTem project [2].

Half-edge Interface

A central role plays the plug-in `de.jtem.halfedgetools.plugin.HalfedgeInterface`. This plug-in works as a converter between half-edge data structure and `IndexedFaceSet` data structure used internally by JReality.

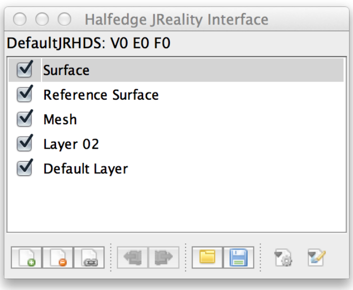

Figure 3.1: The user interface created by the `HalfedgeInterface` plug-in. It manages layers that contain different instances of half-edge data structures and the corresponding visualizations. Layers can be merged, `OBJ` files can be exported and imported and visualization options can be adjusted.

The API of the `HalfedgeInterface` supports `set` and `get` methods that convert to and from JReality. During conversion data is read from an `AdapterSet` that is managed by the half-edge interface. The plug-in supports various data on node types. We give a list of annotation types and their purposes. All conversion adapters work with `double[]` or `double` data types. All supported annotation types are in the package `de.jtem.halfedgetools.adapter.type`.

- **@Position** - Positions of vertices can have lengths 2, 3, or 4.

- **@Color** - Colors either on vertices, edges, or faces.
- **@Normal** - Normals usually for vertices or faces.
- **@TexturePosition** - Texture coordinates of length 2, 3, or 4.
- **@Label** - Text annotations that appear next to the node.
- **@Radius** - Radii of vertex sphere representations or edge cylinders when rendered as spheres or tubes.
- **@Size** - Size in pixels of vertex points or edge lines when rendered as points or lines.

Internally the conversion is done using the classes from the package `de.jtem.halfedgetools.jreality`. Conversion from and to a JREALITY `IndexedFaceSet` is implemented in `ConverterHeds2JR` and `ConverterJR2Heds`.

Visualization Interface

A second important plug-in is the `VisualizationInterface`. It defines an plug-in API and various implementations for data visualization with the half-edge data structure.

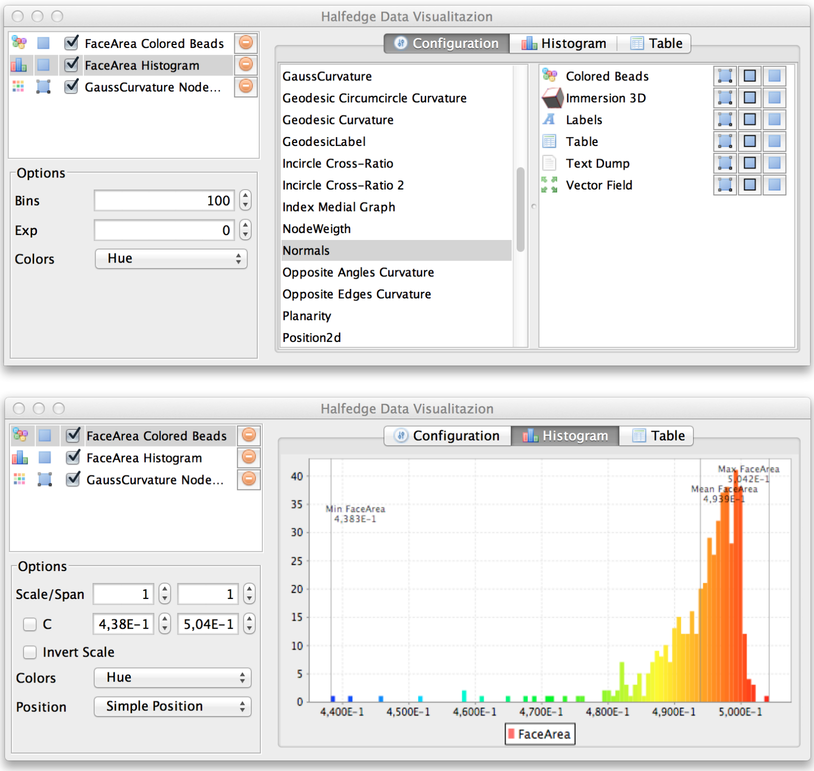

Figure 3.2: The half-edge visualization user interface. Selection of a visualization plug-in for a data adapter (top). A histogram view for scalar data on half-edge nodes (bottom).

Every `Adapter` that is managed by the `HalfedgeInterface` is available as data source. In addition to this every plug-in extending `de.jtem.halfedgetools.plugin.data.DataSourceProvider` is asked for data adapters. A list of these data sources is displayed in the user interface, see Figure 3.2 (top). A visualization is created by a corresponding `DataVisualizer` plug-in. For scalar data on vertices, edges, or faces we provide the Colored Beads Visualizer or simply colored nodes. There is a histogram that displays a density plot for scalar data on nodes, Figure 3.2 (bottom). For vector valued data there is a visualizer that creates arrows starting at half-edge nodes.

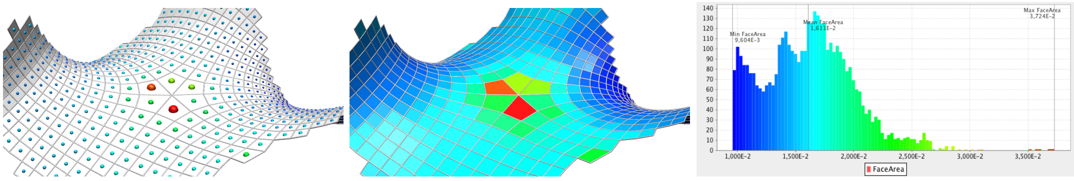

Figure 3.3: Different visualization plug-ins for scalar data on faces. Colored beads (left), node colors, and histogram view.

Chapter 4

ConformalLab - Conformal maps and uniformization

4.1 Embedded surfaces

4.2 Elliptic and hyperelliptic surfaces

4.3 Schottky data

4.4 Surfaces with boundary

Chapter 5

VaryLab - Variational methods for discrete surfaces

5.1 Functional plug-ins

5.2 Implemented functionals and options

5.3 Remeshing

Chapter 6

Non-linear optimization with JPETSC/JTAO

6.1 A java wrapper for PETSc/Tao

Chapter 7

U3D - 3D content in presentations and online publiciations

7.1 The JREALITY U3D export module

7.2 Discrete S-isothermic minimal surfaces

Bibliography

[1] JREALITY, http://www.jreality.de.

[2] JTEM, *Java Tools for Experimental Mathematics*, http://www.jtem.de.

[3] CGAL, *Computational Geometry Algorithms Library*, http://www.cgal.org.

[4] Pierre Alliez, David Cohen-Steiner, Olivier Devillers, Bruno Lévy, and Mathieu Desbrun, *Anisotropic polygonal remeshing*, ACM Trans. Graph. **22** (2003), no. 3, 485–493.

[5] Satish Balay, Jed Brown, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang, *PETSc Web page*, 2011, http://www.mcs.anl.gov/petsc.

[6] Steve Benson, Lois Curfman McInnes, Jorge Moré, Todd Munson, and Jason Sarich, *TAO user manual (revision 1.9)*, 2007, http://www.mcs.anl.gov/tao.

[7] Alexander I. Bobenko, Tim Hoffmann, and Boris Springborn, *Minimal surfaces from circle patterns: geometry from combinatorics*, Ann. of Math. (2) **164** (2006), no. 1, 231–264. MR 2233848 (2007b:53006)

[8] Alexander I. Bobenko and Ulrich Pinkall, *Discretization of surfaces and integrable systems*, Discrete integrable geometry and physics (Vienna, 1996), Oxford Lecture Ser. Math. Appl., vol. 16, Oxford Univ. Press, New York, 1999, pp. 3–58. MR 1676682 (2001j:37128)

[9] Alexander I. Bobenko, Ulrich Pinkall, and Boris Springborn, *Discrete conformal maps and ideal hyperbolic polyhedra*, Preprint; http://arxiv.org/abs/1005.2698, 2010.

[10] Alexander I. Bobenko, Stefan Sechelmann, and Boris Springborn, *Uniformization of discrete Riemann surfaces*, in preparation.

[11] Alexander I. Bobenko and Yuri B. Suris, *Discrete differential geometry – integrable structure*, Graduate Studies in Mathematics, vol. 98, American Mathematical Society, Providence, RI, 2008. MR 2467378 (2010f:37125)

[12] Caltch Discretization Center, *DDG lecture notes and assignments*, http://brickisland.net/cs177/.

[13] David Cohen-Steiner and Jean-Marie Morvan, *Restricted Delaunay triangulations and normal cycle*, Symposium on Computational Geometry, 2003, pp. 312–321.

[14] M. S. Floater and K. Hormann, *Surface parameterization: a tutorial and survey*, Advances in Multiresolution for Geometric Modelling (N. A. Dodgson, M. S. Floater, and M. A. Sabin, eds.), Mathematics and Visualization, Springer, Berlin, Heidelberg, 2005, pp. 157–186.

[15] Geometry Group@TU-Berlin, *Mathematical visualization undergraduate course*, `http://www3.math.tu-berlin.de/geometrie/Lehre/{WS08-SS13}/MathVis/`.

[16] Ren Guo, *Combinatorial yamabe flow on hyperbolic surfaces with boundary*, Commun. Contemp. Math. **13** (2011), no. 5, 827–842.

[17] Udo Hertrich-Jeromin, *Introduction to möbius differential geometry*, London Mathematical Society Lecture Note Series, Cambridge University Press, 2003.

[18] Jürgen Jost, *Compact riemann surfaces*, Universitext (En ligne), Springer-Verlag Berlin Heidelberg, 2007.

[19] Felix Kälberer, Matthias Nieser, and Konrad Polthier, *Quadcover - surface parameterization using branched coverings*, Comput. Graph. Forum **26** (2007), no. 3, 375–384.

[20] Elisa Lafuente Hernández, Christoph Gengnagel, Stefan Sechelmann, and Thilo Rörig, *On the materiality and structural behaviour of highly-elastic gridshell structures.*, Computational Design Modeling: Proceedings of the Design Modeling Symposium Berlin 2011 (C. Gengnagel, A. Kilian, N. Palz, and F. Scheurer, eds.), Springer, 2011, pp. 123–135.

[21] Elisa Lafuente Hernández, Stefan Sechelmann, Thilo Rörig, and Christoph Gengnagel, *Topology optimisation of regular and irregular elastic gridshells by means of a non-linear variational method*, Advances in Architectural Geometry 2012 (L. Hesselgren, S. Sharma, J. Wallner, N. Baldassini, P. Bompas, and J. Raynaud, eds.), Springer, 2012, pp. 147–160.

[22] Yang Liu, Helmut Pottmann, Johannes Wallner, Yong-Liang Yang, and Wenping Wang, *Geometric modeling with conical meshes and developable surfaces*, ACM Trans. Graph. **25** (2006), no. 3, 681–689.

[23] Feng Luo, *Combinatorial yamabe flow on surfaces.*, Commun. Contemp. Math. **6** (2004), no. 5, 765–780 (English).

[24] Helmut Pottmann, Yang Liu, Johannes Wallner, Alexander Bobenko, and Wenping Wang, *Geometry of multi-layer freeform structures for architecture*, ACM Trans. Graph. **26** (2007), no. 3.

[25] Helmut Pottmann, Alexander Schiftner, Pengbo Bo, Heinz Schmiedhofer, Wenping Wang, Niccolo Baldassini, and Johannes Wallner, *Freeform surfaces from single curved panels*, ACM Trans. Graph. **27** (2008), no. 3, #76, 1–10.

[26] Alexander Schiftner, Mathias Höbinger, Johannes Wallner, and Helmut Pottmann, *Packing circles and spheres on surfaces*, ACM Trans. Graphics **28** (2009), no. 5, #139,1–8, Proc. SIGGRAPH Asia.

[27] Stefan Sechelmann, *Uniformization of discrete Riemann surfaces*, Oberwolfach Reports, 2012.

[28] Stefan Sechelmann and Thilo Rörig, *VaryLab Web page*, 2013, `http://www.varylab.com`.

[29] Stefan Sechelmann, Thilo Rörig, and Alexander I. Bobenko, *Quasiisothermic mesh layout*, Advances in Architectural Geometry 2012 (L. Hesselgren, S. Sharma, J. Wallner, N. Baldassini, P. Bompas, and J. Raynaud, eds.), Springer, 2012, pp. 243–258.

[30] A. Sheffer and E. de Sturler, *Parameterization of Faceted Surfaces for Meshing using Angle-Based Flattening*, Engineering with Computers **17** (2001), 326–337.

[31] Alla Sheffer, Emil Praun, and Kenneth Rose, *Mesh parameterization methods and their applications*, Foundations and Trends in Computer Graphics and Vision **2** (2006), no. 2, 105–171.

[32] Hannes Sommer, JpetscTao, *JNI wrapper*, 2010, `http://jpetsctao.zwoggel.net/`.

[33] Boris Springborn, *Discrete conformal equivalence for triangle meshes*, Oberwolfach Reports, Oberwolfach Reports, vol. 6, 2009, pp. 104–106.

[34] Boris Springborn, Peter Schröder, and Ulrich Pinkall, *Conformal equivalence of triangle meshes*, ACM Trans. Graph. **27** (2008), no. 3, 77:1–77:11.

[35] Mirko Zadravec, Alexander Schiftner, and Johannes Wallner, *Designing quad-dominant meshes with planar faces*, Computer Graphics Forum **29** (2010), no. 5, 1671–1679, Proc. Symp. Geometry Processing.

Acknowledgements