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

Chapter 1

Discrete Differential Geometry - Software Packages

1.1 Introduction

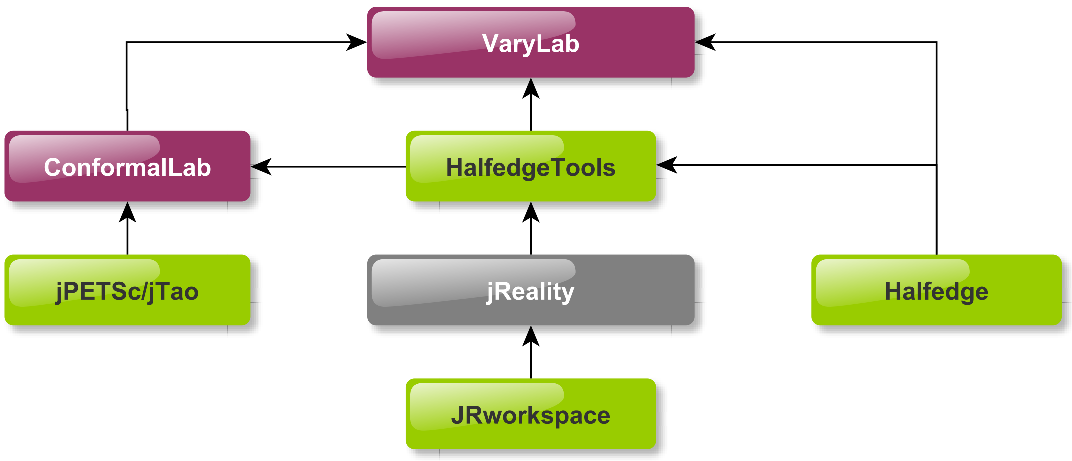

Figure 1.1: Software architecture and dependencies of the DDG Framework. JTEM library packages (green), mathematical software packages (red).

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 due to the curricula of universities which started to include programming courses for undergraduate students with a visualization emphasis, e.g., [4, 3]. It 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 [4] 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 1.3 introduces the JRWORKSPACE library of the JTEM project [8]. 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 [7]. Section 1.4 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 1.5 we describe the software CONFORMALLAB. This package implements the methods of the publications [1, 9, 11, 2]. Section 1.6 introduces VARYLAB, the software implementation of the methods described in the publications [5, 6, 11]. This package is also released to partners of the development group as VARYLAB[GRIDSHELLS], VARYLAB[ULTIMATE], or even online as VARYLAB[SERVICE][10].

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 [7], Hannes Sommer (JPETSC/JTAO) [12], Ulrich Pinkall and Paul Peters (JRWORKSPACE), and Boris Springborn (HALFEDGE).

1.2 Related Work

JavaView, CGAL, ...

1.3 JRWORKSPACE - Java API for modular applications

JRWORKSPACE is part of the JTEM family of software projects [8]. It defines a simple API to create modular Java applications. This API consists of three basic classes (Listings 1.2, 1.3, and 1.4). 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.

1.3.1 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 1.2). 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 1.1. 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 1.1 and the source code of the three basic API classes `Plugin`, `Controller`, and `PluginInfo` in Listings 1.2, 1.3, and 1.4.

```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 1.1: 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.

```
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 1.2: 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 1.3: 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.

```java
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 1.4: 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` isevaluated 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.

1.3.2 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 [7]. 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 defines user interface features.

1.3.3 Gui elements

1.3.4 JRworkspace and Jreality

1.3.5 Building a JRworkspace application

1.4 The Jtem libraries Halfedge and HalfedgeTools

1.4.1 The halfedge data structure and tools

1.4.2 Data model and algorithms

1.5 ConformalLab - Conformal maps and uniformization

1.5.1 Embedded surfaces

1.5.2 Elliptic and hyperelliptic surfaces

1.5.3 Schottky data

1.5.4 Surfaces with boundary

1.6 VaryLab - Variational methods for discrete surfaces

1.6.1 Functional plug-ins

1.6.2 Implemented functionals and options

1.6.3 Remeshing

1.7 Non-linear optimization with Jpetsc/Jtao

1.7.1 A java wrapper for PETSc/Tao

1.8 U3D - 3D content in presentations and online publiciations

1.8.1 The Jreality U3D export module

1.8.2 Discrete S-isothermic minimal surfaces

Bibliography

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

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

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

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

[5] 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.

[6] 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.

[7] JReality development team, JReality *website*, 2013, `http://www.jreality.de`.

[8] JTEM development team, JTEM *website*, 2013, `http://www.jtem.de`.

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

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

[11] 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.

[12] Hannes Sommer, *jPETScTao JNI library web page*, 2010, `http://jpetsctao.zwoggel.net/`.

Acknowledgements