interactive networked virtual reality system

—

Free Form Deformation Manual

Marlene Hochrieser, Christoph Anthes and Roland Landertshamer

November 3, 2010

Abstract

The *free-form deformation* is a popular modeling method for solid geometries: A lattice is superimposed over a model. Then the user deforms the lattice by selecting and displacing points of the lattice. The model form changes in dependence of the lattice deformation by calculating parametric curves.
Such a deformation tool is provided as a module for the *inVRs* framework and a static library for *OpenSG*. Methods for parameterized deformation, like bending or twisting an object, are implemented as well. The *Deformation* module works along with the *inVRs Network* module for synchronizing multiuser applications.
This document describes how to use the *Deformation* module with the *inVRs* framework and the FFD library with an *OpenSG* application.

Contents

Chapter 1

Introduction

One of the most important approaches for object deformation in 3D was presented in the computer graphics journal by Alan H. Barr in 1984 [Bar84]: Conventional axis based operations like translation or rotation are extended to hierarchical deformation operations like twisting or bending. His publication introduces rules for global deformations, which explicitly modify the global space coordinates of a point, and local deformations, which modify the tangent space of a model, inclusive rules for the normal vector transformation.
Thomas W. Sederberg and Scott R. Parry [SP86] developed the free-form deformation (FFD) technique which deforms the space (defined by a lattice superimposed over a model) of an object and the object in dependence (by Bézier interpolation). This method has been advanced by S. Coquillart [Coq90] in order to allow a non-parallelpiped, even user-defined lattice. Coquillart's achievement is known as extended free-form deformation (EFFD).
Yu-Kuang Chang and Alyn P. Rockwood [cha94] proposed a different approach for the FFD: They warp an object along a Bézier curve by using the de Casteljau algorithm [Cas59].
The provided FFD package for *OpenSG* and *inVRs* is based on [SP86]. Each library provides functions for parametric lattice deformation: Bend, twist and taper, which are implemented according to [Bar84].

1.1 Overview

This manual gives an introduction to the technical fundamentals of the FFD technique and explains how to use the provided package. The manual should also familiarize the user with architectural considerations and give an overview on the context of the implemented classes.
The usage of the *OpenSG* library and the *inVRs* module is shown in a sample application for each variant. All relevant functions of the API are listed and explained in this document as well.
The following list gives an overview about the functionality of the FFD API:

- Creation of a lattice for any part of the scenegraph by using its AABB or a customized axis aligned box
- Execution of the bend, twist, taper deformation functions on the lattice
- Deformation of the lattice by applying any customized 4x4 matrix
- Selecting and displacing single or multiple lattice cell points
- FFD calculation with Bézier interpolation using any user-defined polynomial degree.
- Reverting deformation actions

Furthermore, the following visualization and helper methods are implemented:

- (De-)Selection of single or multiple lattice cell points

- Changing color for lattice lines, points (different colors for user defined and internal cell divisions), and selected points
- Switching between the visualization of internal or user-defined cell divisions
- Visualization of the lattice deformation on the lattice solely or on both, the lattice and the model

inVRs specific functionality:

- Synchronization for multiple clients (which can connect to an ongoing session)
- Individualized visualization (e.g. of lattice colors, selections, execution preference)

1.2 Outline

This manual presents the following topics:

- Chapter 2 - Theoretical Background
  In this chapter the fundamentals of the free form deformation and the used global deformation methods are presented.
- Chapter 3 - Using FFDlib and FFDinVRsLib
  This chapter explains the architecture of the libraries. A documentation of the API and an introduction to the usage of the libraries are presented as well.
- Chapter 4 - Outlook
  The final chapter summarizes the main aspects of the manual and envisions ideas about future enhancement.

Chapter 2

Theoretical Background

Free-form deformation is based on parametric curves: A lattice is superimposed over a model. The points of the lattice define the control points of the parametric curve. After displacing lattice points model vertices are interpolated in dependence.
The implemented procedure of the FFD can be described as the following steps:

- Superimpose a partitioned lattice over a model
- Copy all affected model vertices
- Assign each vertex the lattice cell index it resides in
- Deform the lattice by displacing cell corners
- In order to apply lattice deformation to the model, transform model coordinates into cell relative coordinates
- Do the interpolation calculation on the relative model coordinates
- Transform interpolated coordinates into lattice absolute coordinates
- Overwrite model vertices
- Reset the lattice corner positions by recalculating the new AABB
- For further deformations restart with the third step and reassign the cell indices to the model vertices

2.1 FFD

In the first step (lattice construction), the dimensions and the form of the model's axis aligned bounding box (AABB) are applied to the lattice. The user can specify the niceness, but also the scope of the deformation, by setting the number of subdivisions for the lattice. Internally, each partition is subdivided into lattice cells as illustrated in Figure 2.1.
These internal subdivisions define the degree of the Bernstein polynomial and build the proper control points for the Bézier curve as explained later.
After the lattice has been built, when accessing model vertices the lattice coordinate system has to be considered as well. The lattice local model coordinates are denoted as (s, t, u) coordinates, their position in the lattice coordinate system is defined as [SP86]:

$$X = X_0 + sS + tT + uU \tag{2.1}$$

where X_0 is the origin of the lattice, X is the current model vertex and (S, T, U) are the axis of the lattice coordinate system.

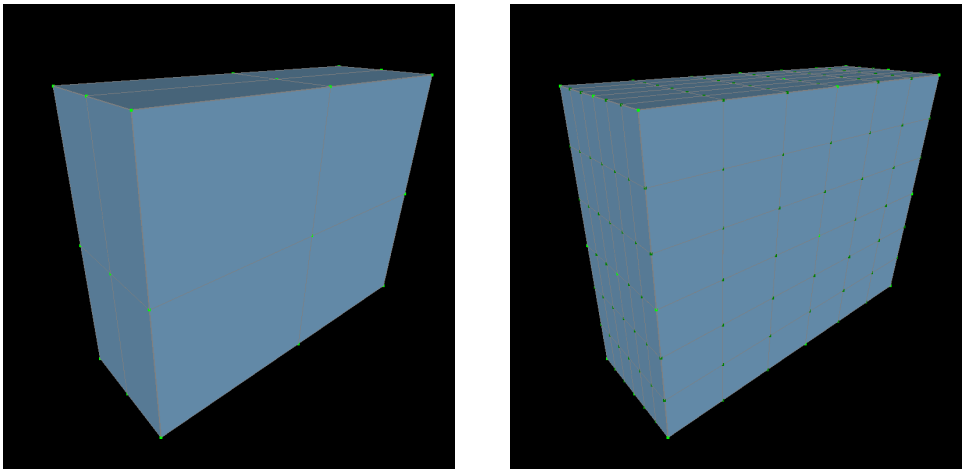

Figure 2.1: Lattice with user-defined and internally managed subdivisions

As described by Sederberg and Parry [SP86] (s, t, u) coordinates can be found using the following equations:

$$s = \frac{TXU(X - X_0)}{TXU \cdot S}, t = \frac{SXU(X - X_0)}{SXU \cdot T}, u = \frac{SXT(X - X_0)}{SXT \cdot U} \tag{2.2}$$

In the next step mapping the *model vertex to the cell it resides in* has to be done. This is necessary because the cell index for a model vertex may change during deformation. The mapping functions are defined by:

$$s_i = \lfloor \frac{(X_x - X_{0_x})}{l_x} \rfloor, t_i = \lfloor \frac{(X_y - X_{0_y})}{l_y} \rfloor, u_i = \lfloor \frac{(X_z - X_{0_z})}{l_z} \rfloor \tag{2.3}$$

(s_i, t_i, u_i) are the cell indices, l_x, l_y, l_z are defined by the length of the lattice in each dimension, X is the model vertex, and X_0 is the lattice origin.
Then, to deform an object, the positions of the internal lattice cell corners get changed by either selecting and moving a single corner or by applying a parameterized deformation.
The calculation of the interpolated position of a model vertex is based on the trivariate tensor product Bernstein polynomial in this implementation:

$$X_{ffd} = \sum_{i=0}^{l} \binom{l}{i} (1-s)^{l-i} s^i (\sum_{j=0}^{m} \binom{m}{j} (1-t)^{m-j} t^j (\sum_{k=0}^{n} \binom{n}{k} (1-u)^{n-k} u^k P_{ijk})) \tag{2.4}$$

The summation iterates over all control points P_{ijk}. *l, m, n* denote the degree of the polynomial in each dimension, which is implemented as a single value for all three dimensions with the same extend. The default value is *three*, but can be defined by the user during lattice creation. Using a higher degree will yield more curve control points, but also increase computational costs.
The base functions for a cubic Bézier curve with degree *three* are defined by four (*degree + 1*) cubic Bernstein polynomials [WP00], also shown in Figure 2.2:

$$B1(t) = (1-t)^3; \tag{2.5}$$

$$B2(t) = 3t(1-t)^2; \tag{2.6}$$

$$B3(t) = 3t^2(1-t); \tag{2.7}$$

$$B4(t) = t^3; \tag{2.8}$$

$$B1(t) + B2(t) + B3(t) + B4(t) = 1; \tag{2.9}$$

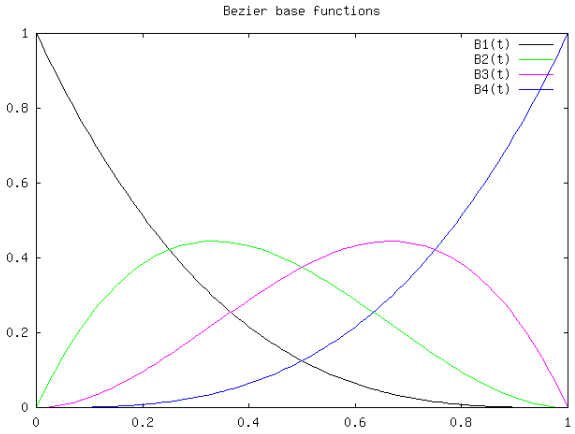

Figure 2.2: Bézier base functions

Each control point in each dimension defines one base curve B. A relative model vertex coordinate describes the weight t for the interpolation calculation in each dimension.
In the final step, the result of the polynomial function is applied to the (s, t, u) coordinate, which is the new model vertex position after transforming it back to an absolute value:

$$X_x = C_{i_x} + sl_x, X_y = C_{i_y} + tl_y, X_z = C_{i_z} + ul_z \tag{2.10}$$

C_i is the coordinate of the lattice cell origin a model vertex resided in before deformation. l is the length of the lattice in the concerning dimension
If the lattice is reset to the new AABB, the algorithm restarts with the third step: Creating the lattice cell index map for all model vertices.

2.2 Deformation functions

As mentioned in the introduction, some functions for lattice deformation are implemented according to [Bar84]: Bend, twist, and taper, which extent the basic transformation functions rotation, translation, and scale.
In the following the mathematical foundations for the operations bend, twist, and taper are explained. The equations have been taken from [Bar84] and [Par08].

2.2.1 Bending

The bending function deforms the object into a curved shape. One side of the object is being stretched while the other side is being compressed. The bending operation is implemented as function with three parameters:

- *Bending rate*: Defines the bending angle in *rad*.
- *Center*: Determines the center of the bend. The value has to be in range from 0 to 1 ($0 \rightarrow$ *bottom*, $0.5 \rightarrow$ *center*, $1 \rightarrow$ *top* of the bending axis). Figure 2.3 shows the effect of changes to the center value.
- *Axis*: Bending is performed along this axis.

Bending an object along the Y-axis:

$$bendingregion = z_{min}; z_{max}$$
$$bendingcenter = y_0; z_{min}$$

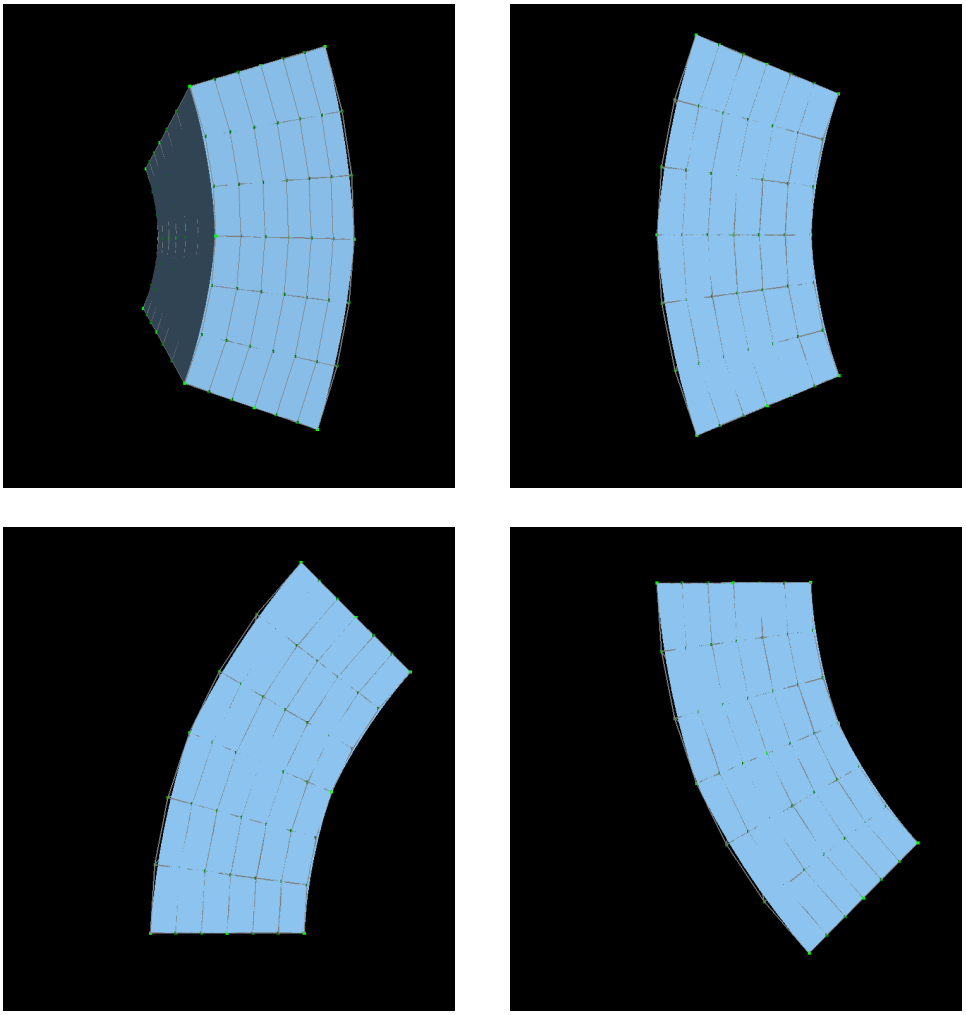

Figure 2.3: Box with bending deformation. The bending rate is 45 °C in the negative (upper left) / positive (upper right) direction. The bending center is 0.5 (upper), 0 (lower left), 1 (lower right)

$$\theta = \begin{cases} z - z_{max} & z < z_{max} \\ z_{max} - z_{min} & else \end{cases} \tag{2.11}$$

$$x' = x \tag{2.12}$$

$$y' = \begin{cases} y_0 - sin(\theta)(z - \frac{1}{k}) & y_{min} \leq y \leq y_{max} \\ y_0 - sin(\theta)(z - \frac{1}{k}) + cos(\theta)(y - y_{min}) & y \leq_{min} \\ y_0 - sin(\theta)(z - \frac{1}{k}) + cos(\theta)(y - y_{min}) & y \geq y_{max} \end{cases} \tag{2.13}$$

$$z' = \begin{cases} cos(\theta)(z - \frac{1}{k}) + \frac{1}{k} & y_{min} \leq y \leq y_{max} \\ cos(\theta)(z - \frac{1}{k}) + \frac{1}{k} + sin(\theta)(y - y_{min}) & y \leq_{min} \\ cos(\theta)(z - \frac{1}{k}) + \frac{1}{k} + sin(\theta)(y - y_{min}) & y \geq y_{max} \end{cases} \tag{2.14}$$

Minimum and maximum of the bending region are set to the lattice minimum and maximum. The bending center is defined by (y_0, z_{min}) for bending along the Y-axis.

2.2.2 Tapering

Taper is the rejuvenation of an object along a choosen axis. The *taper* function scales the cross section surface depending on the tapering axis value of the current coordinate. Applying the taper function on a box will result in a pyramid, see Figure 2.4.

- *Tapering factor*: Determines the scaling of the basis at the bottom of the choosen axis ($1 \rightarrow$ no scale, $< 1 \rightarrow$ scale down, $> 1 \rightarrow$ scale up).
- *Axis*: Tapering occurs along this axis. For a negative axis value, the taper is performed along the axis in the negative direction.

Tapering an object along the positive Z-axis:

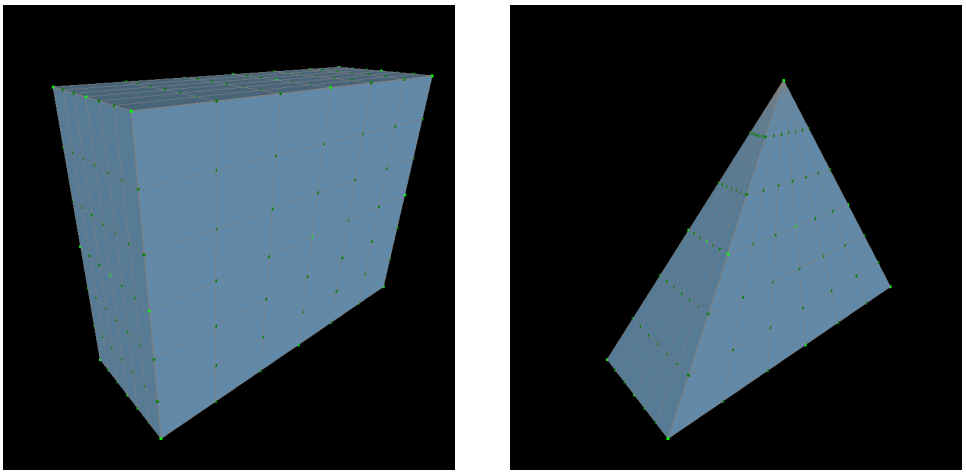

Figure 2.4: Box with taper deformation and no scaling of the basis

$$f(z) = \frac{max(z) - z}{max(z) - min(z)} \tag{2.15}$$

$$x' = f(z)x \tag{2.16}$$

$$y' = f(z)y \tag{2.17}$$

$$z' = z \tag{2.18}$$

2.2.3 Twisting

Twisting an object can be imagined as a rotation of the object in slices from bottom to top, while each slice is rotated a bit more than the last one. The result of a twist deformation is shown in Figure 2.5.
Axis and twist factor are the two parameters of this function:

- *Twisting factor*: Angle of the twist in *rad*. 2π defines a full twist.
- *Axis*: Twist occurs along this axis.

Twisting along the Z-Axis:

$$w = twistfactor \tag{2.19}$$

$$x' = xcos(wz) - ysin(wz) \tag{2.20}$$

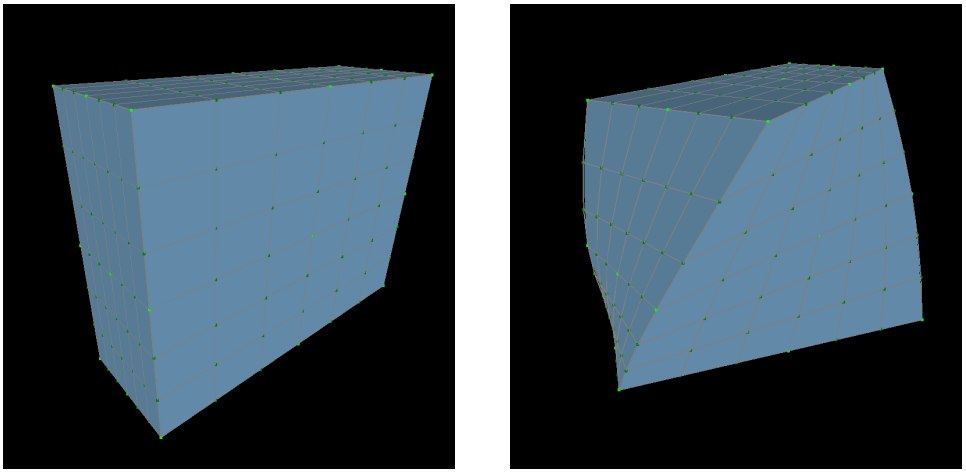

Figure 2.5: Box with twist deformation along the Y-axis

$$y' = xsin(wz) + ycos(wz) \tag{2.21}$$

$$z' = z \tag{2.22}$$

Before applying any deformation function to an object, the object needs to be centered: The object center is set to the origin. Otherwise the deformation would not be symmetric.
Bend, twist, and taper have been implemented as predefined functions for the lattice deformation. A fourth function takes a deformation matrix as parameter and transforms each lattice cell corner by the matrix in Barr's global deformation manner. A second parameter, *center*, determines whether the model should be translated to the origin or not.
In addition deforming the lattice manually by displacing single or multiple cell corners is possible as illustrated in Figure 2.6.

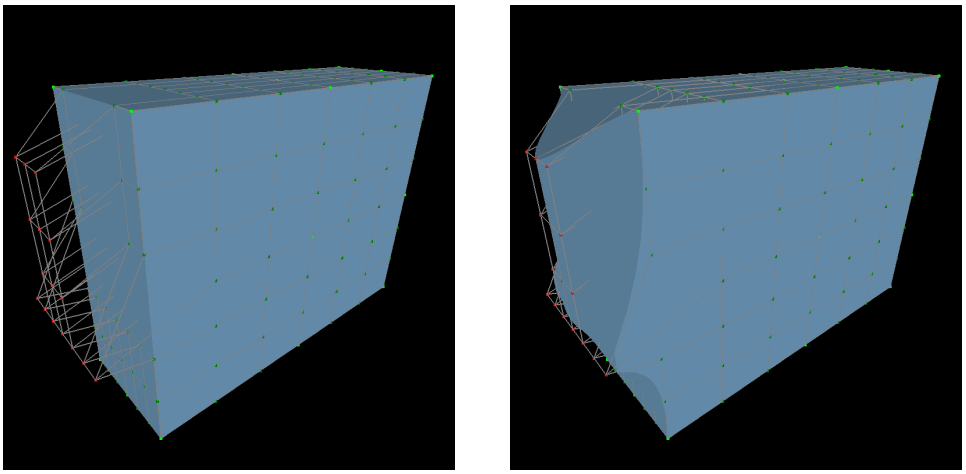

Figure 2.6: Deformating by manual lattice cell point manipulation

2.3 Summary

This chapter has given an overview to the basics of the FFD. The provided libraries implement a cubic Bézier function, which is based on Bernstein polynomials, for interpolation purposes, but

any other polynomial bases could be used as well.
Theoretical background to the deformation functions bend, twist, and taper have been introduced.
The next chapter explains how to use the libraries and write own applications using the FFD.

Chapter 3

Using the FFD libraries

The FFD implementation is available as static library for *OpenSG* and additionally as a module for the *inVRs* framework. Both provide methods for managing lattices, executing deformation operations, and customizing the visualization.
After a brief description of the architecture and functionality the API regarding *OpenSG* and *inVRs* is explained.

3.1 Architecture

The FFD calculation, which uses the functions of the `Bezier` namespace for interpolation, is implemented in the `Lattice` class. `Bezier` and `Lattice` are independent from the scene graph in use.
Lattice operations, e.g. inserting or bending a lattice, are called actions. Each action is implemented as a separate class and inherits from `DAction` or `DeformAction` (which again is derived from `DAction`):
All operations which actually deform the lattice inherit from `DeformAction`, general lattice operations are derived directly from `DAction`. All lattices and actions are managed in the class `OpenSGLatticeActionDeque`.
Lattice deforming actions are:

- `BendDeformAction`
- `TwistDeformAction`
- `TaperDeformAction`
- `SetPointDeformAction`
- `GlobalDeformAction`

General lattice operations are:

- `InsertLatticeDAction`
- `ExecuteDAction`
- `RemoveLatticeDAction`

In the following sections details about the architecture are explained for *OpenSG* and *inVRs* in particular. Since the *inVRs* module is built upon the *OpenSG* implementation, both sections are relevant for the *inVRs* specific part.

3.1.1 FFD library for OpenSG

If a user inserts a lattice, a new *OpenSG* `Geometry` node for lattice visualization is placed above the subscene. Figure 3.1 shows a schematic view of the scenegraph before and after pasting the lattice.

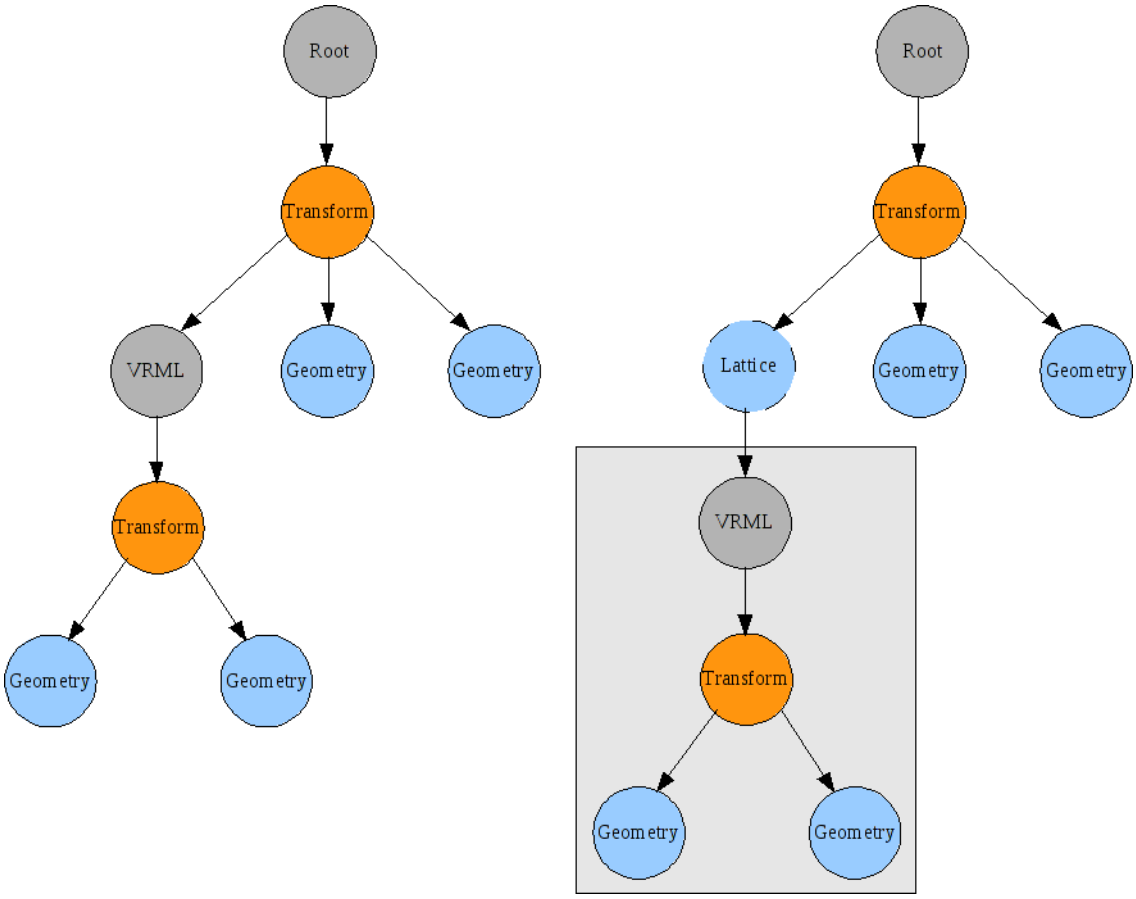

Figure 3.1: The *VRML* node represents the *OpenSG* subscene which is going to be deformed. The lattice is placed above the *VRML* node and affects all children (right).

The visualization of the lattice is implemented in the `OpenSGLatticeGeometry` class. The lattice `Geometry` is built on primitive types (GL_POINTS, GL_LINES) with single indexing. In order to manipulate a `Geometry` in the scene, all model vertex coordinates are copied by the `OpenSGModelPointManager`. When lattice deformations are applied to the model, all model vertex coordinates are overwritten with the manipulated copy. The `OpenSGModelPointManager` ensures that all coordinates are transformed into the correct coordinate system while reading and writing data.
The main class, most relevant for the user, is the `OpenSGDeformActionManager` class. This static class provides all necessary methods to insert a lattice, deform models, remove the lattice, and alter the visualization parameters.

3.1.2 FFD Deformation module for InVRs

inVRs object transformation works on entire entities (here: *OpenSG* nodes). In order to gain access to model vertices with *inVRs*, working on the level of *OpenSG* nodes is necessary. Therefore, the implementation regarding deformations is a wrapper to map *inVRs* entities to *OpenSG* nodes.

The main class, `InVRsDeformActionManager`, provides all functionality for altering the visualization, inserting, deforming, and removing a lattice.
Concerning the network, the module has to take care about persistent synchronization of all actions. If the *Deformation* module is used in a multi-user application, actions need to be sent to all connected users. Therefore, each action is extended to an *inVRs* `Event`, which implements encoding and decoding of the action parameters. For each action an event is created, encoded, sent to all connected users, decoded, and executed locally.
If a new user connects to an ongoing session, sending a synchronization event, containing all actions and parameters for all entities, to the new user is triggered. On the recently joined users side, the package is decoded and all actions are applied to the entities in the scene. Encoding and decoding for this purpose is implemented in the class `SyncDActionsEvent`..

3.2 Basic API

The FFD libraries are easy to use, only an instance of the `OpenSGDeformActionManager` or `inVRsDeformActionManager` is needed in order to use the full functionality of the package. The following list summarizes the main steps:

- Get a static instance of `OpenSGDeformActionManager` or `inVRsDeformActionManager` (singleton) and call all further methods on this instance.
- Insert a lattice for any scene node.
- Deform the lattice with bend, twist, taper, deform, or setPoint.
- Execute the deformation on the model by running execute or revert it by removing the lattice.
- Optionally set visualization settings (instant execution, lattice divisions, colors, wireframe/-points/shaded model) as explained later.
- Clean up if the FFD module is no longer used.

Functions of the API are explained for *OpenSG* in the next section. Since *inVRs* function signatures are very similar, only the differences are described.

3.2.1 *OpenSG* related API

OpenSGDeformActionManager All deformation methods are implemented in the class with the same name `OpenSGDeformActionManager`.

- `static OpenSGDeformActionManager* GetInstance()`

  Create and get a singleton instance of the static OpenSGDeformActionManager.

- `static void Destroy()`

  Destroy the instance of OpenSGDeformActionManager and clean up allocated memory of all classes in use.

- ```
  bool insertLattice(NodePtr subScene, int height, int width, int length,
          size_t dim = 3, bool isMasked = false, gmtl::AABoxf aabb =
          gmtl::AABoxf(gmtl::Vec3f(0, 0, 0), gmtl::Vec3f(0, 0, 0)),
          float epsilon = 0.0001f)
  ```

  Inserts a new lattice above the given *subScene* if the subScene node and none of its children or parents have a lattice inserted, a new lattice will be created and the given node will be managed by the OpenSGDeformActionManager. If the subScene node is manged already, this method verifies that the last action was removeLattice or the action deque is empty.
  ```

The parameters *height*, *width*, and *length* set the lattice divisions in each dimension. *dim* determines the polynomial degree. If *isMasked* is set to *true*, only model vertices of the subscene which are inside the given *aabb* are affected. The default value for *isMasked* is *false*. In this case the *aabb* is ignored and found internally, so that all model vertices of the subscene reside in the bounding box. *epsilon* is a threshold which extends the AABB in each dimension before calculating the relative model vertex coordinates. This is necessary because of possible floating point imprecisions. The default value should be sufficient, but needs to be increased if any assertions regarding cell indices occur.

- `void removeLattice(NodePtr subScene)`

  Removes the lattice from the given node if the lattice is exactly one hierachy above, as inserted by insertLattice.

- `void executeFfd(NodePtr subScene)`

  Executes the free-form deformation on all affected model vertices.

- `void bend(NodePtr subScene, const float bendFactor, const float center, const int axis)`

  Bends an object with bending angle in *rad* at *center* of bend along the bending *axis*. The *center* of bend has to be in range from 0 to 1 ($0 \rightarrow$ *bottom*, $0.5 \rightarrow$ *center*, $1 \rightarrow$ *top* of the bending axis). The bend operation is not invertible.

- `void twist(NodePtr subScene, const float twistFactor, const int axis)`

  Twist an object with the given twistFactor in *rad* along the given *axis*. A *twistFactor* of 2π defines a full twist. The twist operation is invertible.

- `void taper(NodePtr subScene, const float taperFactor, const int axis)`

  Taper an object with the given *taperFactor* along the given *axis*. The top of the lattice is set to the center. The basis of the lattice is scaled along the choosen axis, determined by the *taperFactor*. ($1 \rightarrow$ no scale, $< 1 \rightarrow$ scale down, $> 1 \rightarrow$ scale up). The taper operation is not invertible.

- `void deform(NodePtr subScene, const gmtl::Matrix44f dMatrix, const bool center = true)`

  Deforms the lattice with the customized matrix *dMatrix*. Actually each point of the lattice is multiplied with the given matrix, using the *GMTL* method *xform*. If *center* is set to *true*, the deformation is performed after setting the local center of model to the origin, which results in a symmetric deformation. The customized deformation is invertible.

- `void setPoint(NodePtr subScene, const gmtl::Vec3i index, const gmtl::Vec3f position)`

  Set the position of a single lattice cell point. The lattice cell point index refers to the internal subdivisions of a lattice.

- `void setPoints(NodePtr subScene, const vector<gmtl::Vec3i>& indices, const vector<gmtl::Vec3f>& positions)`

  Set the position of a vector of lattice cell points to different positions. Index and position vector have to be of the same length.

- `void setPoints(NodePtr subScene, const vector<gmtl::Vec3i>& indices, const gmtl::Vec3f addPosition);`

  Add the same vector to all lattice cell points.

- `gmtl::Vec3i getCellDivisions(NodePtr subScene)`

  Returns the number of the *lattice divisions * internal subdivisions (polynomial degree)*.

- `vector<gmtl::Vec3i> getSelection(NodePtr subScene)`

  Returns a copy of the current selection indices.

- `gmtl::AABoxf getAabb(NodePtr subScene)`

  Calculates and returns the AABB for the given *subScene*.

- `int getNumOfDeques() const`

  Returns the number of currently managed action deques. Managed action deques also include deques of nodes which do not have an active lattice at the moment.

- `OpenSGLatticeActionDeque* getDaDeque(NodePtr subScene)`

  Returns a pointer to the action deque for the given *subScene*.

- `bool undo(NodePtr subScene)`

  Reverts the last action if possible.

- `void selectLatticeCellPoints(const vector<gmtl::Vec3i>& selectPoints, NodePtr subScene, bool append = true)`

  Sets the lattice points with the index of *selectPoints* selected and changes their color to the selection color.

- `bool selectLatticeCellPoint(SimpleSceneManager* mgr, NodePtr subScene, int x, int y, bool append = true)`

  Tries to select a lattice point for the given mouse coordinates x and y by performing the *OpenSG* intersection test[1]. If the subScene gets hit, the nearest lattice cell point gets calculated and selected. *append* determines whether the selected point should replace or be added to the current selection. Returns true if selecting was possible.

- `void unselectLatticeCellPoints(NodePtr subScene)`

  Clears the selection and updates the visualization by reseting the color for the lattice corners.

- `bool unselectLatticeCellPoint(SimpleSceneManager* mgr, NodePtr subScene, int x, int y)`

  Tries to unselect a lattice point by performing the same intersection test as *selectLatticeCellPoints* does, but removes the lattice corner from the selection list if the hit succeeds. Returns true if unselecting was successful.

- `setShowAll(NodePtr subScene, bool showAll)`

  Toggle the visualization for internal subdivisions.

- `setInstantExecution(NodePtr subScene, bool doExecute)`

  Toggles the execution calculation to be either performed on the lattice solely or on both, the lattice and the model, by setting doExecute: If doExecute is true, the deformation will call executeFFD for each deformation action done by the user, but no execute actions will be inserted into the action deque until executeFFD is called explicitly.
  If instant execution is toggled *executeFFD* is called, previous deformations to the lattice are applied to the model, and the lattice is reset to the AABB. Then, if doExecute is set to true, the OpenSGModelPointManager is assigned to create a copy of all model points of the subScene, called a savepoint, which is necessary for the FFD calculation, especially if the deformation should not be applied to the model at the end.
  Setting doExecute to true and later back to false will not delete the created savepoint.

[1] http://www.oliver-abert.de/opensg/Traversal.html#TutorialTraversalActionsIntersect

- `dump()`

  Prints all actions of all current and past lattices, including nodes which do not have a lattice present.

- `dump(NodePtr subScene)`

  Prints all actions of the lattice of the *subScene*.

Actions are managed in the `OpenSGLatticeActionDeque` with regards to the order of their execution. Thus reverting actions is potentially possible:

- `InsertLatticeDAction`: Undo is supported.
- `RemoveLatticeDAction`: Undo is only supported if this remove has been done after an insert or execute action.
- `ExecuteDAction`: Undo is only supported if the last action was insert or execute.
- `DeformAction`: The inverse deform method of a `DeformAction` is executed if the flag *invertible* is set to *true*.

If reverting an action is successful, the last action in the deque is removed. For example, executing the actions insert and remove lattice one after the other, gives the same visual result as insert lattice and undo, but the state of deque is different. In the first variant, the deque holds an insert and a remove action, in the second variant, the insert action is deleted and none of both actions appear in the deque.
Visualization changes (e.g. altering the line color of the lattice) apply exactly to one lattice, so different settings for different lattices in the scene are possible. These changes are not managed as actions and are not affected by the undo action.

3.2.2 *inVRs* related API

InVRsDeformActionManager All deformation methods are implemented in the class of the same name `InVRsDeformActionManager`. The implementation for *inVRs* has almost the same signature as for *OpenSG*. The following explains the differences between the *OpenSG* and *inVRs*.

- `static InVRsDeformActionManager* GetInstance()`

  Get the singleton instance of the InVRsDeformActionManager.

- `void twist(Entity* entity, const float twistFactor, const int axis)`

  Sends an event to all connected users, causing them to execute the twist deformation with given parameters locally.

- `void twistLocal(Entity* entity, const float twistFactor, const int axis);`

  Each action has a method with the postfix *Local*. These methods can be used with *inVRs* applications excluding the network module. No event will be sent and the action is executed only locally.

- `setInstantExecution(Entity* entity, bool doExecute)`

  Toggles the instant execution calculation for a given entity. Instant execution is only performed on the local host and is not send to other participants. If instant execution is active, executeFFD, insertLattice, and removeLattice the deformations on the model is the same for all users. Between these actions only users having the doExecute flag set to true will see results on the model, others see the deformations on the lattice solely.

All deformation methods take an *inVRs* `Entity` pointer instead of a *OpenSG* `NodePtr` as first parameter. Also a local version of all deformation methods exist. Otherwise the signature of the *inVRs* and *OpenSG* methods of the *DeformationManagers* are the same.

3.3 Using *FFDlib* with OpenSG

A sample application for *OpenSG* is provided in `FFDSample/main.cpp`. This examples has a usage description, which is displayed after pressing the key *h* and after starting the application. The sample application allows users to play with the deformation tool in an interactive way. All relevant deformation functions are used in the sample.
The following example summarizes how to use the FFD library by listing a method which performs taper and bend to a certain node:

```
void bendAndTaper(NodePtr node)
{
  OpenSGDeformActionManager* osgdam = OpenSGDeformActionManager::GetInstance();
  if(osgdam != 0)
  {
    // insert a lattice with two subdivisions in each dimension above a certain
        node
    osgdam->insertLattice(node, 2, 2, 2);

    // taper the lattice along the x-axis and bend afterwards
    osgdam->taper(node, 1.0f, 1);
    osgdam->bend(node, gmtl::Math::deg2Rad(5.0f), 0.5f, 1);

    // apply lattice deformations to the node
    osgdam->executeFfd(node);

    // remove the lattice
    osgdam->removeLattice(node);
  }
}
```

Listing 3.1: Short FFD usage example for OpenSG applications

To free allocated memory call *Destroy* (before exiting the program at the latest) if the deformation library is no longer used:

```
void keyboard(unsigned char k, int x, int y)
{
  switch(k)
  {
    case 27:
    {
      OpenSGDeformActionManager::Destroy();
      osgExit();
      exit(0);
    }
    // ...
  }
}
```

Listing 3.2: Cleanup

3.4 Using *FFD* with *inVRs*

A sample application for *inVRs* is provided in `FFDinVRsSample/main.cpp`. This examples has a usage description, which is displayed after pressing the key *h* and after starting the application. The sample application allows users to play with the deformation tool in an interactive way. Also all relevant deformation functions are used in the sample.
In the following all necessary steps for using the Deformation module are shown in a few code examples.

Extend *initModules* in order to initialize the *Deformation* module. For a multiuser application also initialize the network module:

```
#include "InVRsDeformActionManager.h"
#include "Deformation.h"

Deformation* deformation;

void initModules(ModuleInterface* module)
{
  if (module->getName() == "Network")
    network = (NetworkInterface*)module;
  else if (module->getName() == "Deformation")
    deformation = (Deformation*)module;
}
```

Listing 3.3: Initialization of the Deformation module

Add the deformation event pipe handling in the display method:

```
void display(void)
{
  SystemCore::step();

  float currentTimeStamp;
  float dt;
  currentTimeStamp = timer.getTime();
  dt = currentTimeStamp - lastTimeStamp;

  // process the deformation pipe
  deformation->step(dt);

  lastTimeStamp = currentTimeStamp;

  mgr->redraw(); // redraw the window
} // display
```

The following is a very short example to describe how to use the *Deformation* module to deform an entity:

```
void bendAndTaper(Entity* entity)
{
  InVRsDeformActionManager* invrsDam = InVRsDeformActionManager::GetInstance();
  if(invrsDam != 0)
  {
    // insert a lattice with two subdivisions in each dimension above a certain
        entity
    invrsDam->insertLattice(entity, 2, 2, 2);

    // taper the lattice along the x-axis and bend afterwards
    invrsDam->taper(entity, 1.0f, 1);
    invrsDam->bend(entity, gmtl::Math::deg2Rad(5.0f), 0.5f, 1);

    // apply lattice deformations to the entity
    invrsDam->executeFfd(entity);

    // remove the lattice
    invrsDam->removeLattice(entity);
  }
}
```

Listing 3.4: Short FFD usage example for inVRs applications

Add a call to the static *Destroy* in the cleanup routine to free allocated memory:

```
void cleanup()
{
  InVRsDeformActionManager::Destroy();
  SystemCore::cleanup();
  osgExit();
}
```

Listing 3.5: Cleanup

In order to load the deformation module, the configuration and the *deformation.xml* must be present.

```
<!-- Paths for Module Datastructure -->
<path name="NetworkConfiguration" path="config/modules/network/" />
<path name="DeformationConfiguration" path="config/modules/deformation/" />
```

Listing 3.6: Changes to general.xml

```
<modules>
  <module name="Network" cfgPath="network.xml" />
  <module name="Deformation" cfgPath="deformation.xml"/>
</modules>
```

Listing 3.7: Changes to modules.xml

The *deformation.xml* does not include any configuration at the moment

3.5 Summary

This chapter has introduced the FFD architecture. The API calls have been explained and demonstrated by code examples.

Chapter 4

Outlook

4.1 Future Work

Implementing optimization for the action deque would be useful, mainly for synchronizing deformations. A set of rules must be defined, e.g. consecutive setPoint calls changing the same point can be merged to one call by accumulating the position changes.
Another optimization could be realized by updating only affected model vertex coordinates. This could be done better by using a mesh datastructure which allows accessing neighbors of a vertex easily.
For nicer results after interpolation, especially when deforming only a part of the model continuity control is desired.
Advanced selection tools like smooth selection or methods for selecting whole regions would be a nice feature. Also applying deformations weighted for single model vertices would enhance the tool in a useful way.
Adding more deformation actions could be achieved easily. Implementing different interpolation methods and providing an interface for switching the interpolation is an easy-to-realize feature as well.
Extending the modeling possibilities to a sophisticated modeling tool, also with direct vertex manipulation, would be the ultimate goal. Support for animated FFD would be of great value for the package.

Bibliography

[Bar84] Alan H. Barr. Global and Local Deformations of Solid Primitives. 18(3):21–30, July 1984.

[Cas59] P. De Casteljau. Outillages méthodes calcul. Technical report, A. Citroen, Paris, 1959.

[cha94] *A generalized de Casteljau approach to 3D free-form deformation*, New York, NY, USA, 1994. ACM.

[Coq90] Sabine Coquillart. Extended free-form deformation: a sculpturing tool for 3d geometric modeling. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 187–196, New York, NY, USA, 1990. ACM.

[Par08] Rick Parent. *Computer Animation Algorithms & Techniques*. Morgan Kaufmann, second edition edition, 2008.

[SP86] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. pages 151–160, 1986.

[WP00] Alan Watt and Fabio Policarpo. *3d Computer Games Technology: Real-Time Rendering and Software with Cdrom*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

List of Figures

Listings

Appendix

Installing *FFD*

Building *FFD* with CMake

Building *FFD* libraries and samples with Code::Blocks (LINUX)

With CMake a *Code::Blocks* project can be created via command line:

```
cd cmake
cmake .. -G "CodeBlocks - Unix Makefiles"
```

The project can be found in the *cmake* directory, the name is the project name in capital letters with the extension *.cbp* (e.g. *FFDOSG.cbp*).
Since *Code::Blocks* does not import the header files, select the project and choose *"Add files recursively.."* from the context menu (right mouse button) or the *Project* menu and choose the src folder.

To execute the sample applications from *Code::Blocks* the working directory for the execution needs to be changed: Activate the project and choose *"Properties"* from the context menu. In the tabulator *"Build targets"* highlight the sample target and change the path of *"Execution working dir"* to point to the sample directory (*FFDsample* for the *OpenSG* or *ffd_inVRs_samle* for the *inVRs* sample).
After compiling, building the target *install* copies the executables to the execution directory.