FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Interactive Shape-Aware Deformation of 3D Furniture Models

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Lea Aichner
Matrikelnummer 1226600

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Univ.Ass. Dipl.-Mediensys.wiss. Dr.techn. Przemyslaw Musialski

Wien, 8. September 2016

Lea Aichner	Michael Wimmer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Interactive Shape-Aware Deformation of 3D Furniture Models

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Lea Aichner
Registration Number 1226600

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Univ.Ass. Dipl.-Mediensys.wiss. Dr.techn. Przemyslaw Musialski

Vienna, 8th September, 2016

Lea Aichner

Michael Wimmer

Erklärung zur Verfassung der Arbeit

Lea Aichner
Bäuerlegasse 34/22, 1200 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8. September 2016

Lea Aichner

Abstract

Resizing of 3D models can be very useful when creating new models or when reusing old ones. However, naive resizing can create serious visual artifacts which destroy the characteristics of an object. In this thesis an algorithm that protects the features of 3D models during resizing is introduced. It is specialized for furniture models because it should be applied to a furniture configurator. We observed that the distortion that occurs during scaling is not distributed uniformly across the object. Our algorithm automatically detects the vulnerable parts of a model and then stretches only the non-vulnerable ones. Furthermore, the algorithm takes into account that when scaling a mesh in a specific direction, the texture has to be adapted as well in order to prevent representation errors.

Contents

CHAPTER 1

Introduction

Starting from the late 1990s, digital geometry has emerged as a new type of digital media. Discrete digital models such as meshes are widely used in many applications, for example in entertainment, design, or engineering. They allow for a great flexibility regarding modifications and adjustment. Generating such models is still time-consuming and demands a lot of experience. Because of this, there is an emerging trend towards the reuse of existing models, parts, or designs.

In practice, models often have the same structure but a different scaling. A natural idea in such scenarios is to reuse the existing models by reshaping them, for example through resizing. The simplest approach to resize a geometry is to apply a global scale to the mesh along a specific direction. However, this will lead to unwanted distortions of significant features as shown in Figure 1.1 (b) at the pendulum of the clock.

In order to reuse existing digital models, in some situations stretching is not enough. Interactive space deformation is for example a powerful approach for editing raster images, vector graphics, geometric models, and animated characters. The user acquires the possibility to change every part of an object at run-time without destroying its content information. An example of this approach is shown in Figure 1.2. The user can manipulate the crocodile using specific handles maintaining the content information.

1.1 Definition of Structure

Before introducing different structure aware shape deformation techniques, it is important to understand what a structure actually is. Shape structure can be defined as follows: “Shape structure is about the arrangement and relations between shape parts” [MWZ+13].

The Oxford dictionary defines structure in a similar way.

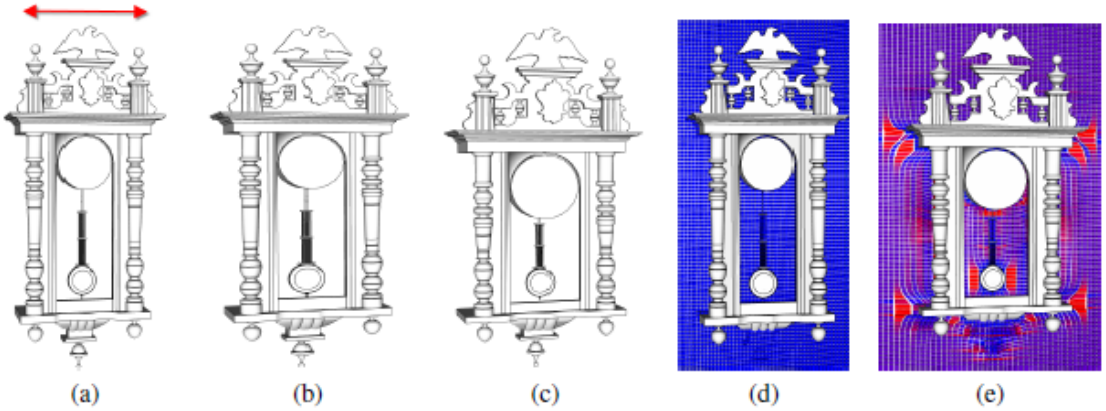


Figure 1.1: Resizing a clock model. Standard non-uniform scale distorts the shape of parts of the model, e.g. the dial (b). Non-homogeneous Resizing resizes the clock in a more natural manner protecting its shape (c). Images (d) and (e) show part of the protective grid before and after resizing. (Image taken from Kraevoy et al.[KSSCO08].)

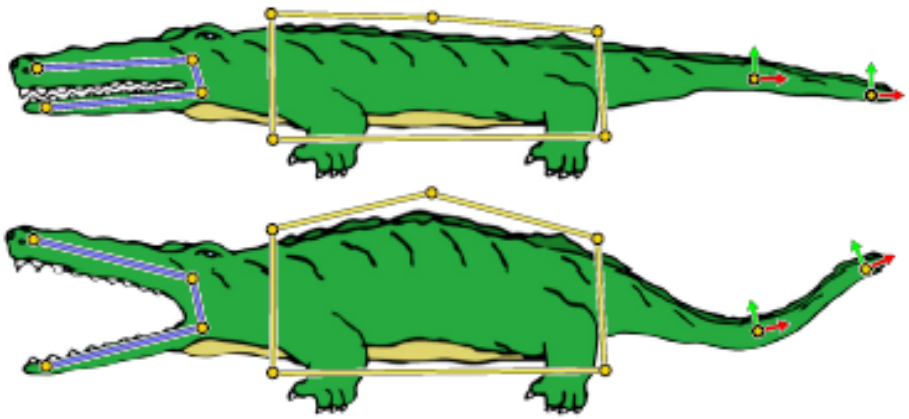


Figure 1.2: Bounded Biharmonic Weights for Real-Time Deformation support points, bones and cages as deformation handles. Bones can be used to control rigid parts, cages to enlarge areas and points to transform flexible parts. (Image taken from Jacobson et al. [JBPS11].)

> The arrangement of and relations between the parts or elements of something complex.

According to the given definitions, structures of real-world objects exhibit great variability and complexity. Even objects that at first sight seem to have a very simple structure such as tables or chairs obey a multitude of complex relations. These relations emerge

from various practical considerations, which can be categorized as:

- Semantic consideration: For example, table-tops are horizontal, a wall is vertical.
- Functional consideration: For example, the legs of a chair support the seat and keep the chair stable.
- Fabrication or economic considerations: For example, identical object parts are easier and faster to produce by reusing machining or modeling setups.

These considerations lead to characteristic object structures as shown in Figure 1.3.

For a designer or an artist of 3D models, it is important to stick to the previous mentioned considerations. Violating these characteristics leads to implausible and unnatural results. Therefore it is very important to facilitate the creation of 3D models through the use of methods that automatically recognize structural properties and invariants of a shape and assist the user in creating structurally plausible shapes efficiently.

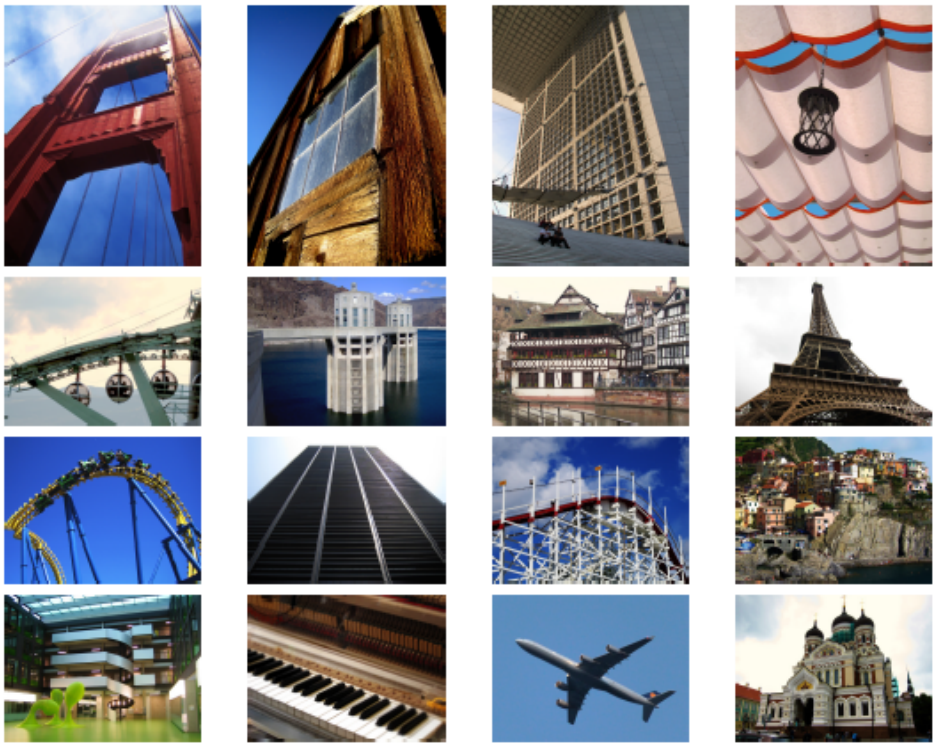


Figure 1.3: Structure in man-made objects arises from a multitude of factors, such as physical, aesthetical, or economical constraints. (Image taken from Mitra et al. [MWZ+13].)

Most deformation algorithms focus on man-made objects because they are characterized by simple structural invariants, such as symmetry, coplanarity, orthogonality, or regular arrangements.

1.2 Goal Definition and Contribution

Resizing by simply applying a global scale can lead to a distortion of various parts of the 3D model. Clearly, this visual distortion depends on the magnitude of the scale and is not distributed uniformly across the surface. The visual artifacts are located in specific, vulnerable regions on the surface. Other regions remain visually correct regardless of the magnitude of the scaling. This observation shows that it is important to resize a mesh non-homogeneously, protecting some parts, while stretching others excessively.

Our goal is to implement an algorithm that enables stretching a model without destroying its characteristic features. Therefore, it takes into account the structure of the 3D model and avoids visual artifacts when scaling at runtime. The possibilities to extract such high-level information in digital models are still limited. Instead, our algorithm uses low-level analysis of the geometry to automatically detect vulnerable parts given predefined scaling axes.

Similar to the Slippage Analysis of Kraevoy et al. ([KSSCO08]), our algorithm uses the projection of the triangle normal onto the scaling axis to decide whether a part of an element can be scaled or not. However, our method computes so-called *Scale Rectangles* (Figure 3.4) that indicate the beginning and the end of the non-vulnerable parts of a geometry. Furthermore, the user can influence the size and the number of the computed Scale Rectangles at runtime. When scaling the model, our algorithm only stretches the parts of the geometry that are specified by the Scale Rectangles. Thereby, all characteristic features are preserved.

In contrast to already existing context aware scaling algorithms, our work discusses the adjustment of the texture after scaling as well (Figure 4.2). Even if the algorithm scales only the regions that are indifferent to the scale, the texture has to be adapted on the whole 3D model. First, the correct mapping of the texture coordinates has to be computed. The mapping specifies which distance in world coordinates corresponds to a distance of 1 in the texture coordinates. Afterwards, we identify all vertices whose texture coordinates have to be adjusted. Furthermore, our algorithm has to identify if the u or v coordinate has to be adapted and if an enlargement of the world coordinate leads to an enlargement or a reduction of the texture coordinate.

Our approach is specialized for parts of furniture models (Figure 4.2) because it is supposed to be applied to a furniture configurator. The models are characterized by rather simple geometries which have a regular shape along a specific scale direction. However, with some modifications it can be applied to complex models as well (Figure 4.5). In order to facilitate the integration into existing Unity projects, the algorithm is implemented using the game engine Unity.

1.3 Structure of the Thesis

This thesis is structured as follows:

The second chapter, Related Work, explains what deformation of 3D models actually is, and introduces the two different approaches of surface-based deformation and space deformation. Furthermore, it describes the state of the art of Shape-Aware deformation techniques. The focus lies primarily on Free-Form deformations, Non-homogeneous Resizing of Complex Models and on Bounded Biharmonic Weights for Real-Time Deformation. The third chapter, Structure Aware Resizing Algorithm, discusses first the practical relevance of our algorithm. Then the analysis and processing phases of the algorithm are described in detail. The fourth chapter, Results and Evaluation, contains the results of evaluation achieved by our implementation. The last chapter, Conclusion and Future Work, includes a discussion of possible improvements and extensions to our application and summarizes the thesis.

CHAPTER 2

Related Work

2.1 Deformation

The deformation of a 3D triangle mesh can be achieved in different ways. One can distinguish between two classes of shape deformations: *surface-based deformations* and *space deformations* ([BKP+10]). The deformation of a surface S into the desired surface S' is mathematically described by a displacement function d. The displacement function associates to each point $p \in S$ a displacement vector $d(p)$ that maps the given surface S to S':

$$S' := p + d(p) \quad \| \quad p \in S. \tag{2.1}$$

When working with a discrete triangle mesh, the displacement function d is piecewise linear, such that it is fully defined by the displacement vectors $d_i = d(p_i)$ with $p_i \in S$.

A user can now define a set of handle points $p_i \in H \subset S$ to get the ability to control the deformation by prescribing displacements $\overline{d_i}$. Furthermore, he can define constraints for certain parts $F \subset S$ to stay fixed during the transformation (Figure 2.1):

$$d(p_i) = \overline{d_i}\,, \quad \forall p_i \in H, \tag{2.2}$$

$$d(p_i) = 0, \quad \forall p_i \in F, \tag{2.3}$$

The surface-based deformation differs from the space deformation in the calculation of the displacement vectors d_i.

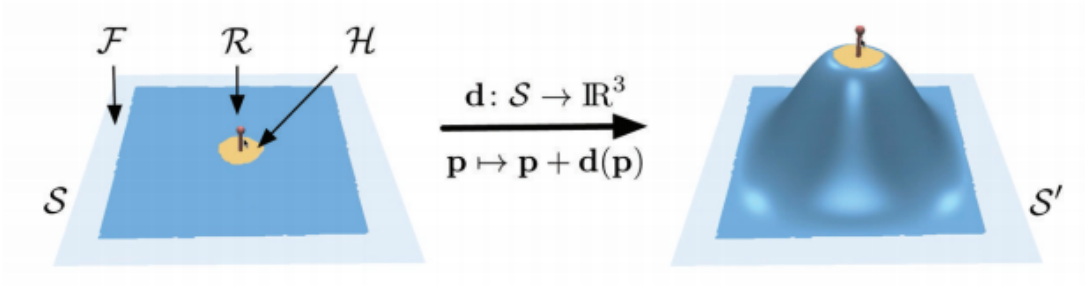


Figure 2.1: A given surface S is deformed into S' by a displacement function $d(p)$. By moving a handle region H (yellow), the user controls the deformation. The region F (gray) stays the same. The unconstrained deformation region R (blue) should deform in an intuitive way. (Image taken from Botsch et al. [BKP+10].)

2.1.1 Surface-Based Deformations

The displacement function $d : S \to \Re^3$ lives on the original surface S and is found by computations on the triangle mesh. These methods offer the ability to address every vertex individually, which leads to a high degree of control.

On the downside, the efficiency and robustness of the computations are strongly affected by the mesh complexity and the triangle quality of the original surface S.

An example for surface based deformation uses transformation propagation in order to deform the object. It works by calculating a smooth scalar field with a dimension of 1 at a specified handle and a dimension of 0 outside a predefined support region. An example can be seen in Figure 2.2.

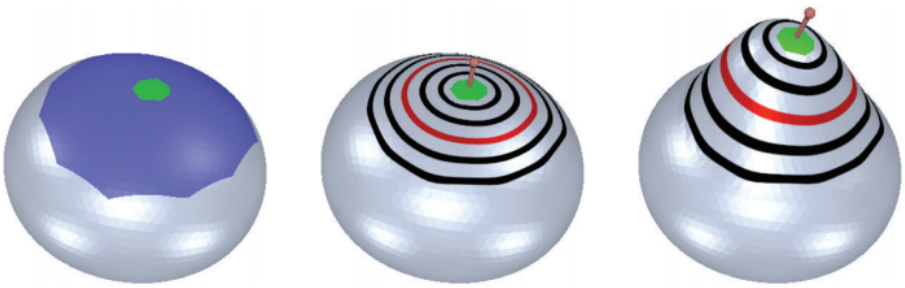


Figure 2.2: After specifying the blue support region and the green handle region (left), a smooth scalar field is constructed with a dimension of 1 at the handle region and a dimension of 0 outside the support region. In the center and right image the isolines of the scalar field are shown. (Image taken from Botsch et al. [BKP+10].)

2.1.2 Space Deformations

Space deformations compute a displacement function $d : \Re^3 \rightarrow \Re^3$ that deforms the surrounding three dimensional space around an object and through this implicitly deforms the surface S too (Figure 2.3). The advantage of this approach is that no computation on the triangle mesh S is needed. Therefore, such methods are less affected by the complexity and triangle quality of S.

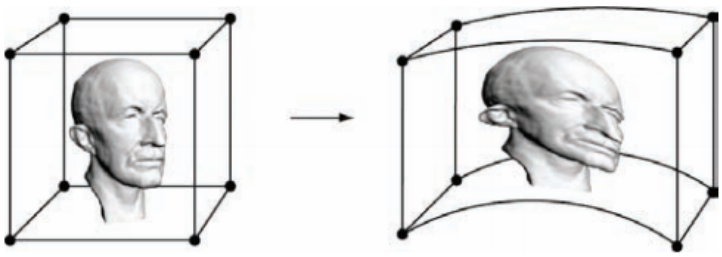


Figure 2.3: Space deformations deform the whole embedding space and thus implicitly deform the object. (Image taken from Botsch et al. [BKP+10].)

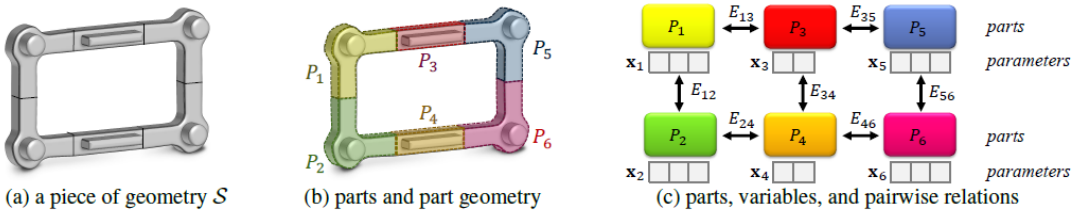
2.2 Structure Aware Shape Processing

In this section we present some algorithms that are used to edit existing shapes. All algorithms consist of an analysis phase and a processing phase. During the analysis phase the method extracts structural information from the 3D mesh. The processing phase then uses this information in order to deform the input in a content preserving manner. In other words, the algorithm removes degrees of freedom in comparison to general, unconstrained shape modification, therefore making it easier to perform plausible changes.

2.2.1 Free-Form Deformation

As aready explained above, the free-form deformation is a space based deformation technique. Simple free-form deformations allow low-level mesh operations and provide a high degree of flexibility. To avoid destroying the whole structure of the object however, several methods have been proposed to perform higher-level deformations. The free-form deformations implemented by Sederberg et al. [SP86] and Coquillart [Coq90] use a low-dimensional, band-limited, volumetric basis to impose smooth, low-frequency deformations to the geometry.

A shape S can be seen as a collection of parts and their parameters. A part of the shape is a logical entity of semantic significance that controls a portion of the geometry, which we call part geometry. Furthermore, each part holds a set of parameters that affects the shape of the part. Figure 2.4 shows an example of this concept.



(a) a piece of geometry $\mathcal{S}$ (b) parts and part geometry (c) parts, variables, and pairwise relations

Figure 2.4: The figure shows a shape that is divided into separate parts (a). Each part controls a portion of geometry (b). The parts (c) have parameters that control the deformation and relations to other parts. (Image taken from Mitra et al. [MWZ+13].)

The goal of the algorithm is to preserve parts with high-frequency details, while the low-frequency parts should be deformed. Therefore, a 3D control grid is specified which surrounds the 3D geometry. In the free-form deformation approach implemented by Sederberg et al. [SP86] and Coquillart et al. [Coq90], the vertices $x \in \Re^3$ are deformed by a function $f : \Re^3 \to \Re^3$, which is composed of low-frequency basis functions $b_i : \Re^3 \to \Re^3$, so that:

$$f(x) = \sum_{i=1}^{n} w_i b_i(x) \,. \tag{2.4}$$

The single parts of the geometry are represented by the scalar basis functions b_i. All vertices that are within the support of the part b_i define one part geometry. The parameter of the part b_i is the coefficient w_i, which is a 3D vector. For every knot in the grid there exists a basis function with the corresponding coefficient.

The user can now adjust all coefficients manually and can thereby deform the object. Structure is only implicitly imposed by using just a few, low-frequency basis functions. An expample of the deformation of an object using this approach of freeform deformations can be seen in Figure 2.5.

2.2.2 Structure-Aware Deformation

The drawback of free-form deformations is that they only have a local and non-adaptive way of preserving structure. They don't take into account the content of the shape or global relations.

Because of this a lot of algorithms have emerged that deal with structure-aware deformation. These algorithms are based either on the concept of local adaptivity or on the concept of non-local relations. In the following section we present the algorithm of Kraevoy et al. [KSSCO08] that considers local but adaptive deformation (Figure 1.1).

Global relations are used by Zheng et al. [ZFCO+11] (Figure 2.6). Semantic parts that belong together are represented by object-aligned bounding boxes or shape components

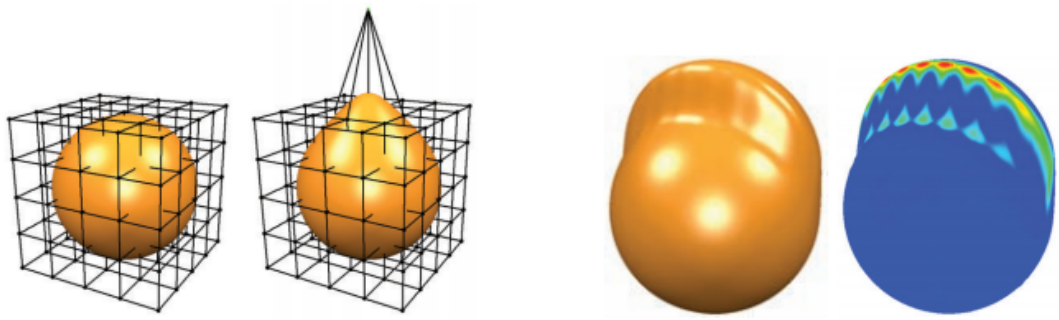


Figure 2.5: In the free-form deformation approach, a 3D control grid is used to specify the part geometries with its basis functions (left). The regular placement of grid basis functions can lead to alias artifacts in the deformed surface (right). (Image taken from Botsch [Bot05].)

obtained from segmentation. When deforming an object, Euclidean invariants (symmetries) are used to propagate edits to affect all symmetric elements similarly. A similar idea is used by iWires [GSMCO09] (Figure 2.7). This approach is discussed in detail below.

Figure 2.6: The controllers (shown in cyan) are made up of either a cuboid or a generalized cylinder. They serve as high-level deformation handles for sructure-aware deformation of man-made shapes. (Image taken from Mitra et al. [MWZ+13].)

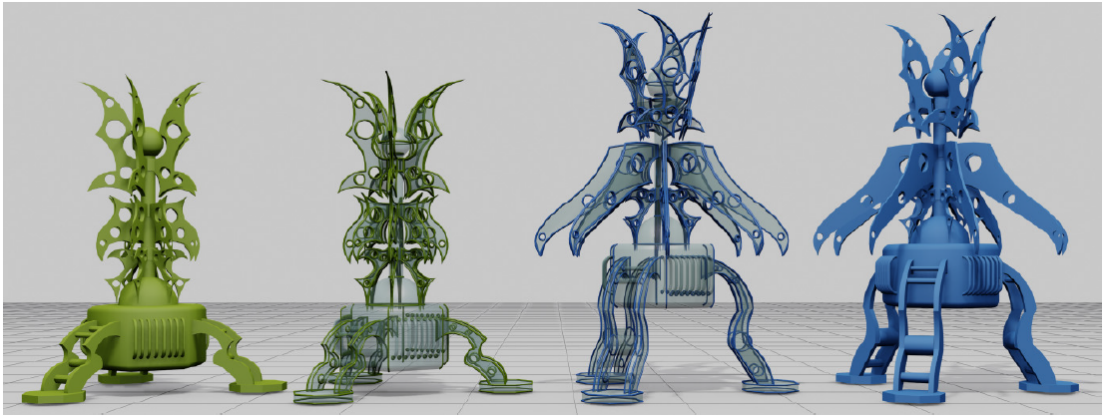


Figure 2.7: A complex model (left) that consists of 108 different components is analyzed and 250 intelligent wires (in green) are extracted. Editing a few wires leads to a new wire configuration (in blue). The result is shown on the right side. (Image taken from Gal et al. [GSMCO09].)

2.2.2.1 Non-Homogeneous Resizing

The algorithm of Kraevoy et al. [KSSCO08] allows to stretch 3D objects along several directions, while protecting the model features and structures during resizing. To avoid distortion in particular parts of the object it is important to scale the object non-homogeneously. For instance, while the proportions between the dial and the pendulum of the clock in Figure 1.1 (c) change, the model still appears visually correct.

Information about semantic parts or constraints of a digital model could be used to resize the model in a correct manner. Unfortunately, the possibility of extracting high-level information from digital models is still limited and is an important part of reverse engineering [ARSF07, AFS06, BMV01]. The analysis of mesh vulnerability used by Kraevoy et al. [KSSCO08] tries to achieve similar effects as reverse engineering but in a simpler way.

In this section we will present the algorithm of Kraevoy et al. [KSSCO08], which uses low-level analysis of the models to automatically detect the vulnerability given a pre-defined resizing axis.

Method Overview

When stretching the clock in Figure 1.1, some parts are more vulnerable than others. First, the algorithm automatically detects all vulnerable regions in the model and records this information in a protective grid defined around the object, the so called vulnerability map. The digital model is then scaled non-homogeneously by a space-deformation technique with respect to the vulnerability map.

The vulnerability of the surface depends on the resizing direction and is independent of the actual amount of stretch. Therefore, the method first estimates the degree of

vulnerability on the mesh given the resizing axes, and then the algorithm uses this information inside a linear resizing formulation.

Estimating Vulnerability

The computation of the vulnerability according to a specific scale axis is based on estimating the effect a non-uniform scale may have on the model. It is based on two components, *slippage* and *normal curvature*.

The *slippage analysis* [GG04] specifies how persistent a surface is to a given transformation or transformation type. It measures if a local region on a surface remains on the surface after the transformation is applied. This is the case when the surface normal of the local region is perpendicular to the scale axis. Slippage can be measured around mesh vertices, by projecting the normal in each vertex's umbrella onto the scaling axis and then summing up the projection lengths. Unfortunately, man-made models, such as the fuel-tank in Figure 2.8, often contain very large triangles. This leads to an unreliable vertex-based slippage estimation. Because of this, the algorithm computes the slippage on mesh faces and projects the face normal onto the scale axis.

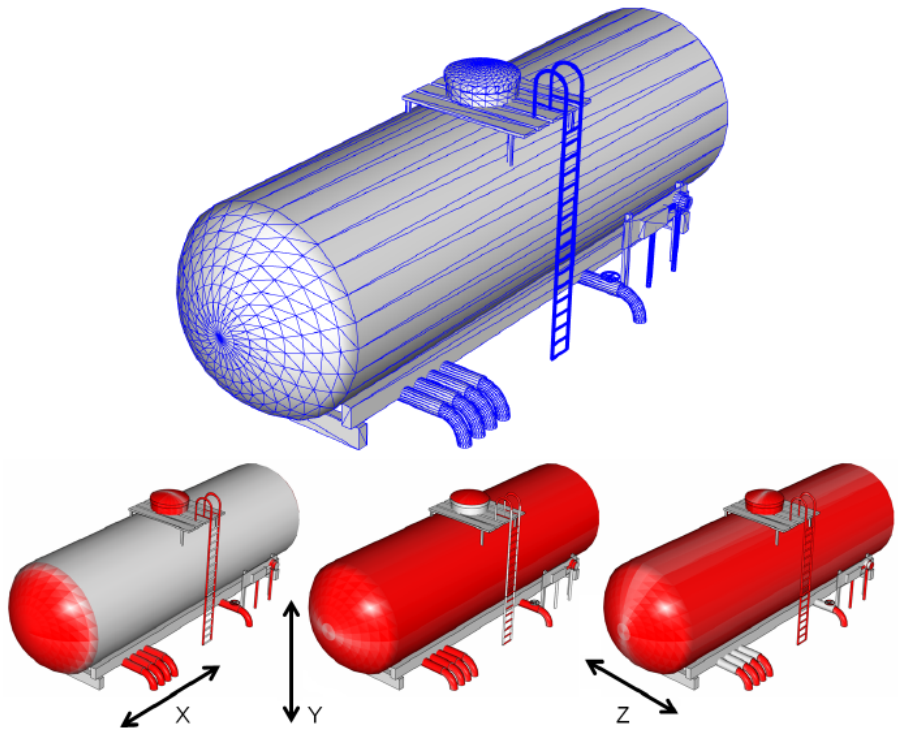


Figure 2.8: The model of a fuel tank. On the bottom the face vulnerability values in the three major directions are highlighted in red. (Image taken from Kraevoy et al.[KSSCO08].)

Slippage analysis is not sufficiently discriminative most times. For example, consider a cone aligned with the scaling axis and a sphere. When stretching the cone its overall shape is preserved. In contrast, when stretching the sphere the whole shape is deformed. As we can see, the slippage metric would classify both surfaces as vulnerable, without clearly distinguishing between the different degrees of vulnerability.

The normal curvature of the surface in the scaling direction predicts the amount of surface bending subject to the scale. The method measures normal curvature at mesh vertices, projecting the scale axis to the surface tangent plane.

Given a scaling axis u, the per-face vulnerability metric θ^u combines slippage with normal curvature,

$$\theta^u = s^u(\epsilon_\kappa + \kappa^u), \tag{2.5}$$

where s^u measures the per-face slippage for the axis u, and κ^u measures the per-face normal curvature. In order to prevent zero curvature, the algorithm adds ϵ_κ to the curvature ($\epsilon_\kappa = 1e^{-4}$). The red parts of the fuel tank in Figure 2.8 show the vulnerability according to the three scaling axes x, y and z.

Transfer to Grid

As we can see in Figure 2.9, the method embeds the digital model of the fuel tank in a protective volumetric grid. Thereby, space-deformation can be used to resize the model. This allows to apply resizing to complex man-made models while preserving spatial relations among parts that are not necessarily geodetically close.

The next step is to convert the surface vulnerability values to grid cell values. This is achieved by simply computing the maximum vulnerability of the mesh faces that intersect the grit cell.

The Resizing Operator

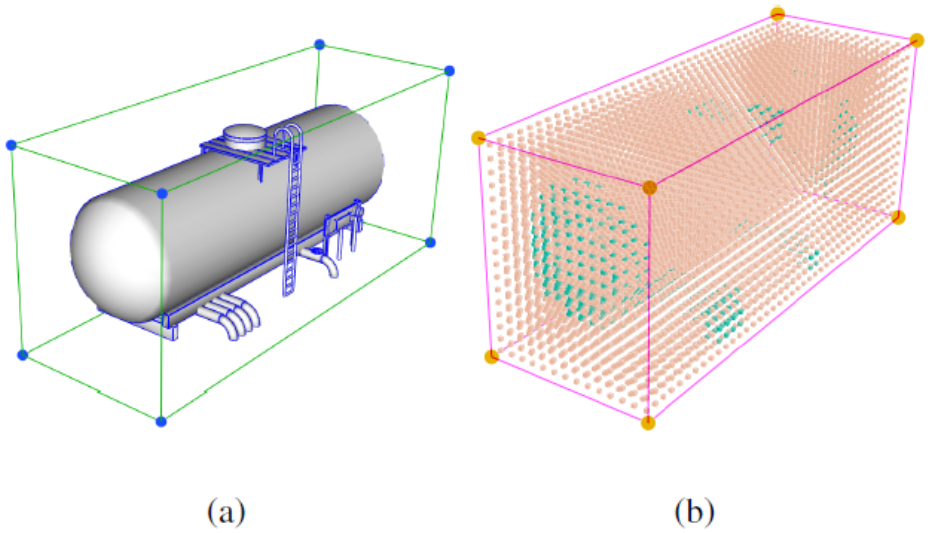
After the vulnerability analysis, the algorithm computes the scale gradients for each cell and for each direction. The aggregation of the scale gradients of each cell results in the global resizing transformation.

Based on the gradients, the method computes full per-cell transformations enforcing compatibility between adjacent cells. Therefore, the algorithm combines all cells together and derives the locations of the grid vertices. If adjacent cells apply the same scale to their shared faces, it is enough to scale each cell as specified. However, as cell scales might not be fully compatible, the algorithm uses a least-square formulation to obtain the coordinates for all the grid vertices at once, while keeping the per-cell scales as much as possible.

Finally, in each cell the transformation is carried back to the object using interpolation.

2.2.2.2 The iWires System Models

The iWIRES framework of Gal et al. [GSMCO09] uses global relations to achieve structure-aware deformation. It is particularly designed for man-made objects, such as furniture,



(a) (b)

Figure 2.9: The protective volumetric grid of the fuel tank generated from the algorithm of Kraevoy et al.[KSSCO08].

mechanical parts or electronic devices. Most parts of these models are characterized by smooth or flat faces and the shape is defined by a small number of features which carry special characteristics and geometric meaning. Conserving the properties of these features allows preserving the character of the whole model.

The method is based on the researches of Singh et al. [SF98] and Orzan et al. [OBB+13], which show that an entire shape can be defined by a small set of curves. Furthermore, Singh et al. [SF98] defined the name *wires* to denote the curves that are key structural features capturing the shape.

The goal of the algorithm is not rigorous reverse engineering of an input shape ([BMV01] and [ARSF07]) or structure detection ([PMW+08]), but mainly light-weight analysis and easy interaction. The user can handle a multitude of shapes, making the technique attractive to a wider user audience.

The method consists of an analyzing phase and an edit phase. The result of the analysis are intelligent wires that are used for a constrained deformation.

Analysis

First, the method analyzes the input shape to extract representative curves or 1D wires, their geometric characteristics and the relationship between different wires. This information is applied to the wires as additional attributes, making them "intelligent".

In order to get the wires, the algorithm has to extract feature curves. Doing that in general models is challenging, but for man-made models, sharp crease lines are good

candidates [OBS04]. Single wires can be combined to form a group that also hold geometric characteristics and information about relations.

An example of the wires can be seen in the first picture of Figure 2.10. In this Lego-Stone example, the algorithm defines eight circles and twelve line segments to form the wire collection. The eight planar circular curves and the six rectangles are identified as individual wires. Wires that lay on the same plane or on parallel planes form groups, as can be seen in the other images in Figure 2.10. Using these groups mutual relations can be easily recorded.

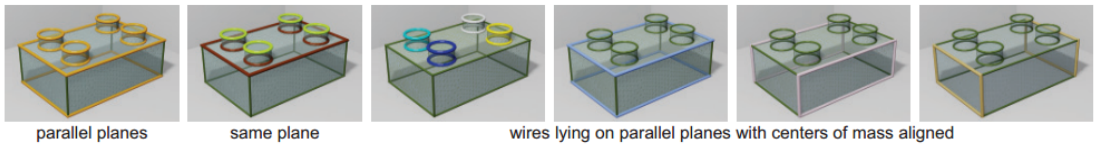


Figure 2.10: Analyzing the Lego model results in different groups of wires that are not necessarily mutually exclusive. Each group is displayed in a different color [GSMCO09].

Deformation Propagation

The user has the possibility to indicate modeling constraints by manipulating deformation handles or by sketching. The algorithm depicts the closest wire to the handle, the so called "seed", and optimizes the deformation to enforce the individual characteristics of the "seed". This edit is then propagated to other wires which are mutually related to the seed wire.

In order to perform a deformation, the algorithm propagates the deformation to the closest wire and enforces its individual characteristics. Thereby, it is important to maintain the group characteristics and group relations. Once all wires have been considered, they serve as modeling constraints for a differential surface deformation. An example of the whole process can be seen in Figure 2.11.

Wire Characteristics and Mutual Relations

The following properties specify the individual wire characteristics:

- Planar or non-planar.
- "Atomic" type: the entire wire can be approximated by a (part of a) circle, a straight line, (a part of) an ellipse or a polynomial curve of a bounded degree.
- Compound wire: When the error of the approximation by a single element is large, the algorithm segments the wire into sub-wires by dividing it at salient internal angles. This time, the sub-wires have to be classified as an "atomic" type.

In order to analyze the relationships between wires, the algorithm forms groups of wires that share certain common characteristics. It is important to mention that groups do

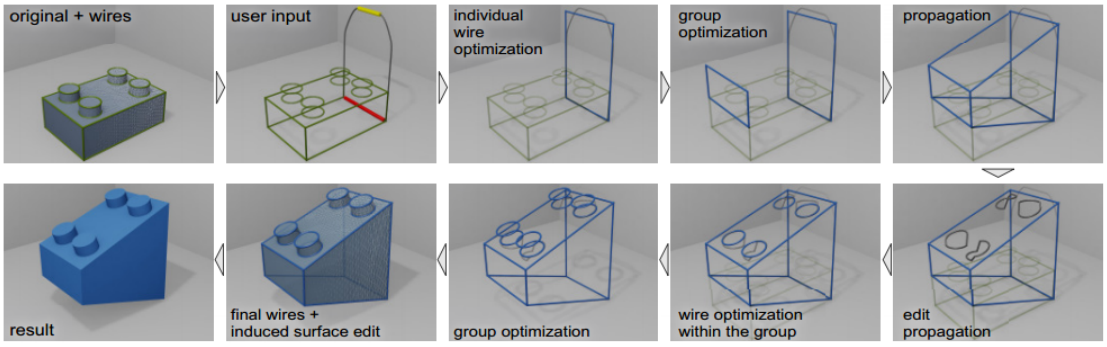


Figure 2.11: The user can deform the Lego model using the yellow and red handles. The edit is then mapped to one of the wires (in gray). The algorithm optimizes the individual wires as shown in blue and propagates the changes to other wires. Individual and group characteristics and relations are preserved [GSMCO09].

not have to be mutually exclusive. The grouping is performed based on two general criteria:

- Wires share common properties such as lying on the same plane or lying on parallel planes. An other common property is proximity. A wire may belong to a group if its Euclidean distance from the group is very small.

- Symmetry: An important criterion for grouping is symmetry. In order to detect the global symmetry between each pair of wires, the method of Mitra et al. [MGP06] is used.

2.2.3 Interactive Space Deformation

Object deformation with linear blending is a powerful approach for editing raster images, vector graphics, geometric models, and animated characters. Real-time performance is very important for interactive design, where tasks require exploration, or for interactive animation, where deformations need to be computed repeatedly. Because of their speed, linear blending methods like the methods of Schaefer et al. [SMW06] or Kavan et al. [KCŽO08] are often used in interactive space deformation: each point on the object is transformed by a linear combination of a small number of affine transformations.

Jacobson et al. [JBPS11] specifies an approach for real-time deformation of arbitrary 2D and 3D shapes by supplying weights for a linear blending scheme that produces smooth and intuitive deformation for handles of arbitrary topology and is described in this section.

2.2.3.1 Handles

In order to perform linear blending, the user first has to specify a number of handles. Afterwards, the deformation system binds the object to these handles (bind time). The user can then manipulate these handles and the system deforms the shape accordingly (pose time). The handles can be points, bones, or cages. Points are quickly placed on the shape and are easy to manipulate. Using points, local deformation properties like position, rotation, and scaling can be smoothly propagated onto nearby areas of the object. Bones can be used in order to make some directions stiffer than others. If a region between two points appears too supple, bones can transform it into a rigid limb. Cages allow the user to influence a specific portion of the object at once. This makes it easier to control bulging and thinning in regions of interest. An example of the different handle types can be seen in Figure 1.2.

2.2.3.2 Bounded Biharmonic Weights

Given a shape S and a number of (disjoint) control handles $H_j \subset \Omega; j = 1, ..., m$, where $\Omega \subset \Re^2$ or $\Re^3$ denotes the volumetric domain enclosed by S. The user defines an affine transformation T_j for each handle H_j, and all points $p \in \Omega$ are computed by their weighted combinations:

$$p' = \sum_{j=1}^{m} w_j(p)\, T_j p, \tag{2.6}$$

where $w_j : \Omega \to \Re$ is the weight function associated with handle H_j. An example of the bounded biharmonic weights is shown in Figure 2.12.

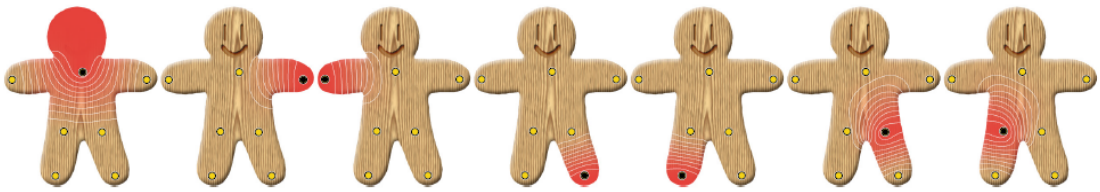


Figure 2.12: The bounded biharmonic weights are shown in red for each handle. The handles have maximum influence in their immediate environment [JBPS11].

The weights are computed once at bind time and have the following desired properties:

- Smoothness: Smoothness is important to avoid visible artifacts in 2D textured shapes and to place handles directly on 3D shapes.
- Non-negativity: Weights cannot be negative, because regions with negative weights move in the opposite direction to the prescribed transformation. This leads to unnatural behavior.

- Shape-awareness: An intuitive correspondence between the handles and the domain Ω is important.
- Partition of unity: If a transformation T is applied to all handles, the entire object will be transformed by T.
- Locality and sparsity: Each handle should influence a shape feature in its vicinity. Furthermore, each point in Ω should only be controlled by a few closest handles.
- No local maxima: Each weight function w_j should have its only global maximum on H_j.

CHAPTER 3

Structure Aware Resizing Algorithm

3.1 Practical Relevance

Our structure aware resizing algorithm will be used in an interactive 3D furniture configurator. Through the use of a 2D furniture configurator, the user obtains the ability to create individual furniture such as the sofa shown in Figure 3.1. The user can choose among different elements on the right side and arrange them as he likes. Invalid combinations and collisions are detected by the application. Furthermore, the user can configure a lot of properties of the furniture, for example the texture, the seat height, or the width of the arm rests.

For many people it is very difficult to visualize the sofa mentally if they only see its 2D composition. In order to improve the imagination, it is important to provide a 3D representation of the self constructed piece of furniture. Because of the fact that the 2D configurator gives the user the ability to modify every single component of the furniture, including armrests or feet, a 3D representation of these single components is needed. Only at run-time, the application arranges the single meshes into one piece of furniture.

Unfortunately, this leads to a big amount of data and redundant information. For example, if the user can choose among ten different forms of arm rests and four different arm rest widths, the application has to save forty different 3D meshes. It would obviously be better to save only one mesh for every form and adjust its width at run-time. In the following sections, a content preserving algorithm that aims to solve this problem is presented. Furthermore, this algorithm takes into account that, when scaling a mesh in a specific direction, the uv coordinates have to be adapted as well in order to prevent representation errors.

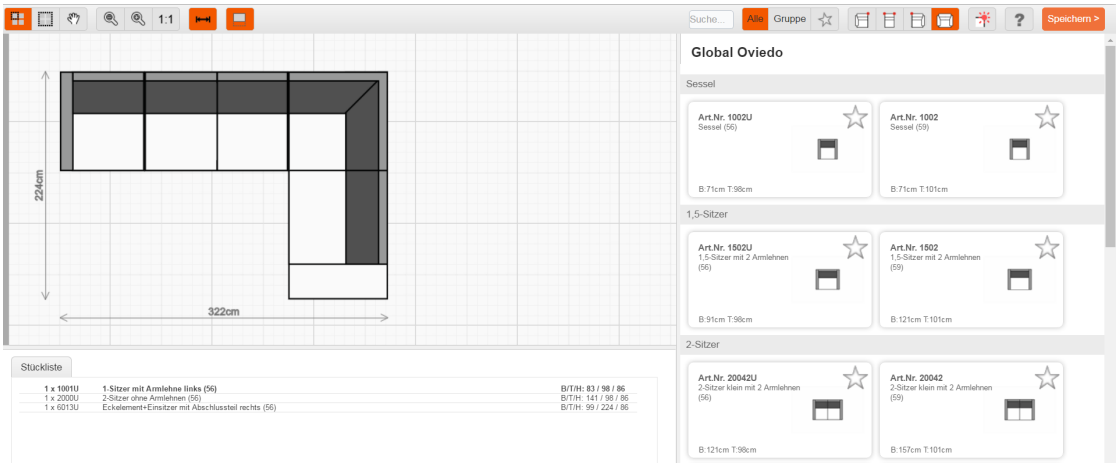


Figure 3.1: A screen-shot of the 2D furniture configurator. The user can select the components of the sofa on the right side and arrange them as he likes. On the grid the currently planned sofa is displayed. The single elements can be moved, rotated, or deleted.

3.2 Program Architecture

The application is implemented in Unity, a cross-platform game engine developed by Unity Technologies. It is mainly used to develop video games for PC, consoles, mobile devices and websites. All parts of the program are implemented in the C# programming language.

Figure 3.2 shows the application in the Unity environment. On the top left it can be seen the *Hierarchy* that contains the 3D meshes, also called *Game Objects*, which are processed by the algorithm. When clicking on a Game Object in the Hierarchy, the *Inspector* appears on the right side of the window. It shows all scripts associated with the Game Object. In our case, the *Scale Rectangle Script* is important since it holds the main implementation of the algorithm.

Furthermore, Unity distinguishes between a *Game View* and a *Scene View*. The Scene View can be used to change the position, rotation or scaling of a Game Object and makes it easy to navigate through the scene. The Game View is active when the program is running. One can interact with the elements and navigate through the scene only on the basis of the implementation.

The whole application can be divided into four sections:

1. The visualization of the Game Objects: A selected element has to be highlighted and a user interface is needed to manipulate the element.

2. The processing of the user input: The user needs the ability to navigate through the scene, to select elements, to rotate, to translate and of course to scale them.

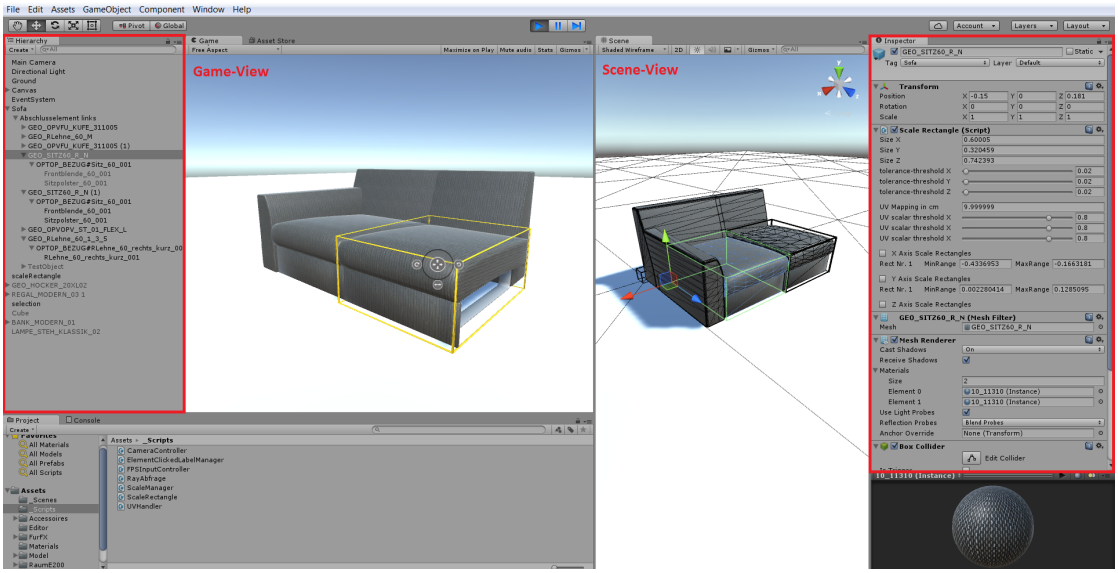


Figure 3.2: A screen-shot of the Unity project. Highlighted are the Hierarchy on the left side with the Game Objects, the Game View and the Scene View and the Inspector that holds the Scale Rectangle Script on the right side.

3. The *Analysis Phase* of the algorithm: In a pre-processing step the areas of the model that can be scaled along a specific axis are computed. Furthermore, the uv coordinates are analyzed in order to simplify their manipulation.

4. The *Processing Phase* of the algorithm: This step is executed at run-time. The world coordinates and texture coordinates are updated according to the results of the analysis phase.

Furthermore, the Unity application consists of the following six scripts:

- CameraController: It handles the navigation through the scene.
- ElementClickedLabelManager: It handles the menu in the Game View.
- RayAbfrage: This script is used to calculate which element is selected and computes its bounding box.
- ScaleManager: The script is used when the user scales the element.
- ScaleRectangle: Performs the Analyzing Phase and the Processing phase when scaling an element.
- UVHandler: Performs the Analyzing Phase and the Processing Phase of the texture coordinates.

3.3 Requirements and Visualization

A Game Object that is visualized in the Game View can consist of multiple meshes and multiple materials. In order to achieve a correct visualization of the model, the root node of one geometry has to be marked with a special tag so that the application knows which Game Object in the Hierarchy holds the transformation information of the whole object.

Figure 3.2 for example shows multiple 3D models that are arranged into one sofa element. Every single geometry like the arm rest or the back rest can be selected and stretched, as long as its root element is marked with the tag “Sofa” in the inspector. The application knows now that all children of the element are part of its geometry.

In order to support different materials on one geometry, every material has to be represented by one submesh. If one mesh consists of multiple materials the algorithm is not able to display the mesh in a correct way.

Furthermore, the root element of one mesh has to contain the so-called Scale Rectangle Script. The algorithm then considers all submeshes by grouping them into a single mesh.

In order to distinguish a selected element at run-time from the others, it has to be highlighted. Therefore, it is surrounded by its bounding box. If the geometry is rotated, the bounding box is rotated too.

Another important part of the visualization is the user interface that aids the user to manipulate the Game Object at run-time. When the user double clicks on an element, a menu appears.

3.4 Processing User Input

When clicking on an element, the program casts a *ray* from the mouse position on the screen into the scene. As shown in the code snipped 3.1, the ray collides with all elements in the scene that hold a Box Collider Script. This script is automatically added to all Game Objects that have a Scale Rectangle Script. If the ray collides with a Game Object that holds a “Sofa” tag, or if an ancestor has the tag, it is marked as selected and its Bounding Box is rendered. Furthermore, a menu is displayed that helps the user to manipulate the object. If the ray does not collide with any element in the scene, nothing is selected and all bounding boxes are removed.

```
1 RaycastHit hit;
2 Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
3 if (Physics.Raycast (ray, out hit)) {
4     Transform sofaTransform = getAncestorWithTag (hit.transform, "Sofa");
5     if (sofaTransform != null) {
6         panel.SetActive (true);
7         updateBoundingBox(sofaTransform.gameObject);
```

```
8       } else {
9           removeSelectedTag ();
10          setVisibility (false);
11      }
12      doubleClick = false;
13  } else {
14      removeSelectedTag ();
15      setVisibility (false);
16  }
```

Code 3.1: The code snippet shows the routine to select an element in the scene through a Raycast

Using the menu (Figure 3.3), the user can rotate the Object around the y-axis or move the Game Object on the ground. In order to move it, a raycast is thrown from the movement button into the scene. The Game Object is set to the position where the ray collides with the ground.

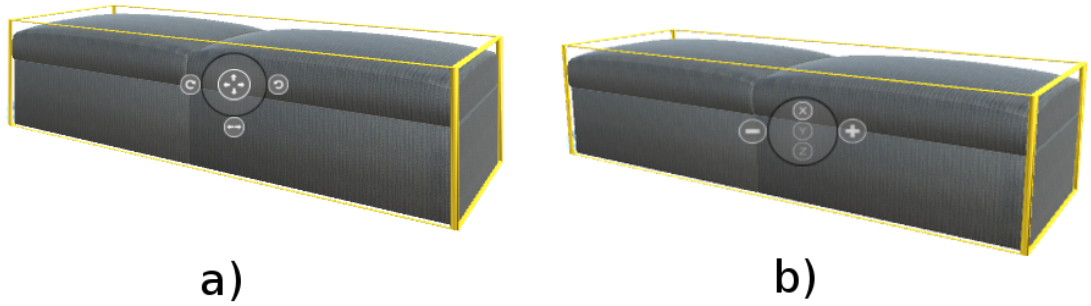


Figure 3.3: The Figure (a) shows the UI menu for rotating and translating the element. When clicking on the button on the bottom the scaling menu appears, shown in Figure (b). The bounding boxes are rendered in yellow.

When clicking on the button on the bottom of the menu, the scaling menu appears. The user can first choose the scaling direction and by clicking on the plus or minus button the geometry is scaled up or down respectively.

The user can navigate through the scene by using the arrow keys or the mouse. With the left mouse button the user can change the view direction. The right mouse button can be used to navigate through the scene according to the current view direction.

3.5 Analysis Phase

Like most of the content preserving scale algorithms described in Related Work, our algorithm is divided into an analysis and a processing phase. In the analysis phase the

algorithm has to determine the parts of a mesh that can be scaled without destroying its content features.

The adjustment of the world coordinates, however, is not enough for the practical use. Scaling a mesh leads to a distortion of the texture, so that the *uv coordinates* have to be adapted too. In order to solve this problem, the analysis phase is divided once more into an analysis phase of the world coordinates and an analysis phase of the uv coordinates.

3.5.1 Analysis World Coordinates

Similar to the algorithm of Kraevoy et al. [KSSCO08], our algorithm uses the projection of the triangle normal onto the scaling axis to decide whether a part of an element can be scaled or not.

After the aggregation of all submeshes into a single mesh, the algorithm runs though all triangles. One triangle consists of three vertices v_1, v_2, and v_3. In order to compute the surface normal of the triangle, the algorithm calculates the cross product of its edges (Equation 3.1). Afterwards, the dot product of the normal n and the unit vector of every scale axis x, y and z is computed (Equation 3.2). This results in a triplet of scalars γ_x, γ_y and γ_z that indicate how vulnerable the triangle is to a scaling along each direction.

$$s_1 = v_2 - v_1; \quad s_2 = v_3 - v_1; \quad n = s_1 \times s_2; \quad n = n \,/\, |n|; \tag{3.1}$$

$$\gamma_x = n \cdot \vec{e_x}; \quad \gamma_y = n \cdot \vec{e_y}; \quad \gamma_z = n \cdot \vec{e_z}; \tag{3.2}$$

The surface normal n is perpendicular to the scale axis if the absolute value of the dot product is zero. In that case, this part of the geometry can be scaled without destroying content features. Obviously, that implicates that all triangles with a dot product unequal to zero shall not be adjusted. Due to imprecise vertex values, the dot product is seldom exactly zero, even if the surface is parallel to the scale axis. Therefore, the application uses a *tolerance-threshold* ϵ (e.g. $\epsilon = 0.02$). If the absolute value of the dot product is bigger than ϵ, this part of the geometry must not be changed.

For the sake of simplicity, the algorithm first computes all parts of the mesh that should not be modified. In order to specify these parts, the algorithm uses so called *ranges*. One range is defined by only two values indicating the start and the end of the range. If for example the scalar of the normal and the x axis is bigger than the tolerance-threshold ($\gamma_x > \epsilon$), the minimum $\min(v_{1x}\ v_{2x}\ v_{3x})$ and the maximum $\max(v_{1x}\ v_{2x}\ v_{3x})$ of the vertices generate a new range r_x. An array `nonScaleRangeX` saves all calculated ranges for the x axis. Before adding a new range to the `nonScaleRangeX` array, the algorithm has to determine if one of the already existing ranges overlaps with r_x. If this is the case, the existing ranges have to be updated. Otherwise, the new range r_x is added to the `nonScaleRangeX` array.

After running through all the triangles, the program can calculate all ranges that have to be scaled based on the non-scalable ranges and saves them in the arrays `scaleRectanglesX`, `scaleRectanglesY` and `scaleRectanglesZ`. In the following chapters of this thesis

we will use the term Scale Rectangle rather than range, since we are working with 3D models and a range is more associated with a 1D object. Some examples of the Scale Rectangles are shown in Figure 3.4.

The next step is now to calculate the ratio of every single Scale Rectangle. If a Scale Rectangle is bigger than another, it is obvious that the bigger one has to be stretched more than the smaller one in order to maintain the correct proportions of the model.

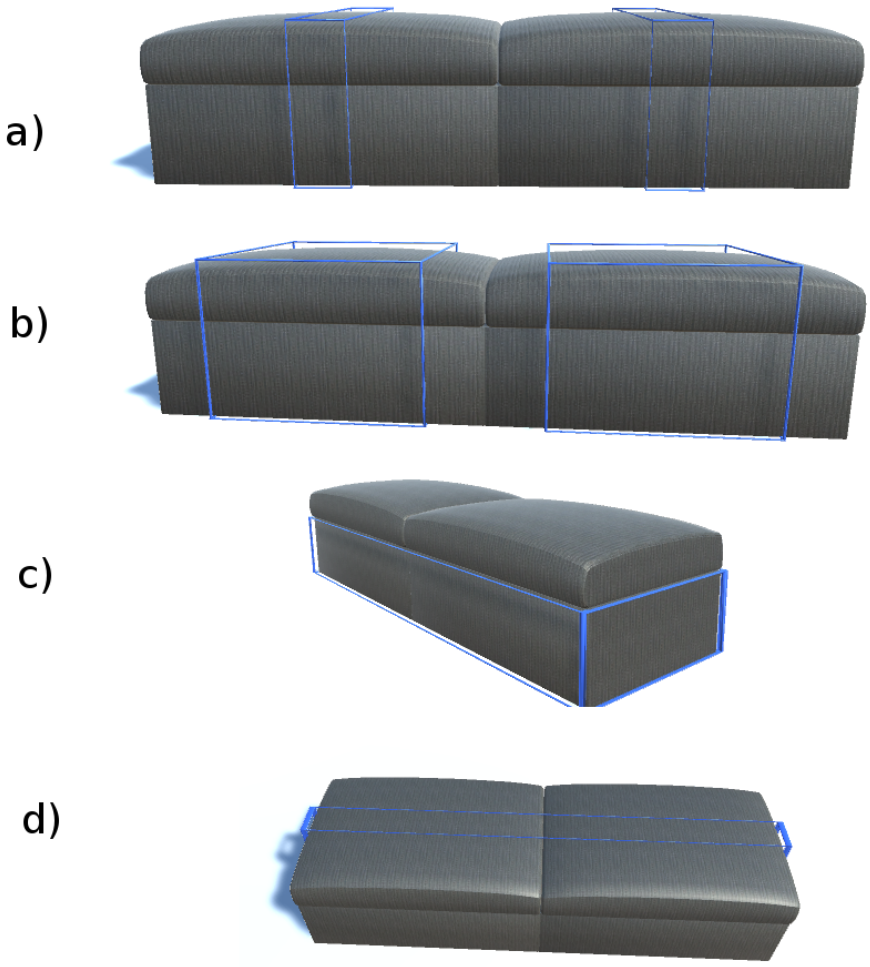


Figure 3.4: The figure displays the Scale Rectangles for the different scale axis. (a) The scale Rectangles for the scale axis x with the default threshold $\epsilon = 0.02$. (b) The Scale Rectangle for the scale axis x with the threshold $\epsilon = 0.055$. (c) The Scale Rectangle for the scale axis y. (d) The Scale Rectangle for the scale axis z.

3.5.2 Analysis UV Coordinates

The analysis phase of the uv Coordinates is more complex than the analysis phase of the world coordinates. Therefore, we divide it into 3 steps.

UV Scale Factor Calculation

A 3D model is scaled by applying a certain value to a part of the vertices of a mesh. Adding the same value to the uv coordinates of the modified vertices does not lead to a correct result in most cases. Therefore, the algorithm has to find out the mapping of the texture coordinates if it is not known by default. The mapping specifies which distance in world coordinates corresponds to the distance of 1 in the texture coordinates.

The precondition for a correct calculation of the mapping is that the used textures have the same mapping in the u and v direction because our algorithm takes into account only the y coordinates of the geometry. A distance of 1 in the u-coordinates corresponds to the same distance in the world coordinates as a distance of 1 in the v-coordinates.

The calculation of the mapping is performed while running though all mesh triangles during the analysis phase of the world coordinates. The algorithm first searches the two vertices v_1 and v_2 with the largest distance in the y coordinates. Afterwards, it compares this distance with the v values in the texture coordinates of v_1 and v_2. The algorithm uses the largest distance in order to maximize the accuracy of the algorithm.

Identify the Vertices Whose UV Coordinates Need to Be Adjusted.

Let's assume that $V \subset \Re^3$ consists of all vertices of a 3D model. When scaling the model in the x direction, V_x holds all vertices whose world coordinates have to be adjusted. As we already explained above, the uv coordinates of the V_x have to be adapted as well. As shown in the Figure 3.5, not all texture coordinates in V_x have to be modified. The texture coordinates of perpendicular parts to the scale axis must not be adapted, in the picture highlighted with a red color. All triangles of the model whose uv coordinates need to be adjusted are identified by T_x.

When running through the mesh triangles, the program saves the index of all vertices whose normal is smaller than a certain threshold λ (λ is by default 0.8). These triangles are not perpendicular to the scale axis and therefore need to be adjusted. The threshold λ is used to make the algorithm more robust against inaccuracies of the model and can be changed by the user if needed.

Identify Scale Direction of Texture Coordinates and Correlation

At this point of the analysis phase, the algorithm knows the triangles $T_i,\ i \in x, y, z$, whose texture coordinates have to be adjusted, as well as the amount of the scaling of the texture coordinates. When changing the x coordinate of a vertex v, the question is whether to modify the u or the v coordinate of the vertex. Therefore, the algorithm has to determine the so called *UV Scale Axis* for every vertex in T_i.

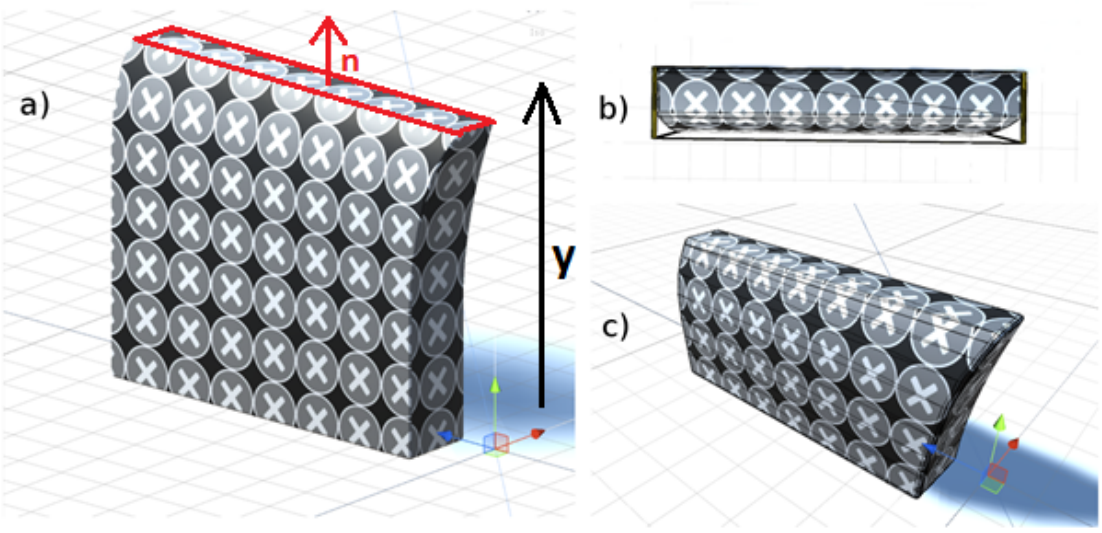


Figure 3.5: The figure shows the model of the arm rest with a test texture. When scaling the model in the positive (a) and negative (c) y direction, the textures are adjusted correctly. (b) shows the original model from above. The uv coordinates in Figure (a) and highlighted in red are not modified.

Furthermore, an enlargement of a world coordinate does not lead automatically to an enlargement of the texture coordinate. The application needs to describe the correlation between world coordinates and texture coordinates for every single vertex.

The algorithm calculates the UV Scale Axis for every triangle in T_x, T_y and T_z using the previously computed UV Scale Factor. In order to compute the UV Scale Axis, the algorithm runs though all triangles T, whose texture coordinates have to be adjusted. Assuming the scale direction is the x-axis, then the algorithm first computes the distance between all x coordinates of the three vertices that compose the triangle. As shown in the code snippet 3.2, the distance is then multiplied by the UV Scale Factor. The result `uvDistShouldBe` is the distance in uv coordinates that the vertices should have.

In the next step the correct distance for both the u coordinates and the v coordinates are computed based on the original uv coordinates of the mesh (`uvCorrectDistX` and `uvCorrectDistY`).

The algorithm compares then the correct distance `uvCorrectDistX` and `uvCorrectDistY` with the calculated distance `uvDistShouldBe` through the method shown in the code snippet 3.3.

```
1
2 public void initUVScaleAxisX(Mesh mesh)
3 {
4 Vector3[] vertices = mesh.vertices;
```

```
5  vertexUVInformationListX = new VertexUVInformation[vertices.Length];
6  int [] triangles = mesh.triangles;
7  Vector2[] uvs = mesh.uv;
8
9  for (int i = 0; i < uvTriangleIndexToScaleX.Count; i++)
10 {
11     int index1 = triangles[uvTriangleIndexToScaleX[i]];
12     int index2 = triangles[(uvTriangleIndexToScaleX[i] + 1) % triangles.Length];
13     int index3 = triangles[(uvTriangleIndexToScaleX[i] + 2) % triangles.Length];
14
15     Vector3 vertex1 = vertices[index1];
16     Vector3 vertex2 = vertices[index2];
17     Vector3 vertex3 = vertices[index3];
18
19     float distVertex1_2 = vertex2.x - vertex1.x;
20     float distVertex1_3 = vertex3.x - vertex1.x;
21     float distVertex3_2 = vertex2.x - vertex3.x;
22
23     float uvDistShouldBe1_2 = distVertex1_2 * factorVertexToUv;
24     float uvDistShouldBe1_3 = distVertex1_3 * factorVertexToUv;
25     float uvDistShouldBe3_2 = distVertex3_2 * factorVertexToUv;
26
27     float uvCorrectDistX1_2 = uvs[index2].x - uvs[index1].x;
28     float uvCorrectDistY1_2 = uvs[index2].y - uvs[index1].y;
29     float uvCorrectDistX1_3 = uvs[index3].x - uvs[index1].x;
30     float uvCorrectDistY1_3 = uvs[index3].y - uvs[index1].y;
31     float uvCorrectDistX3_2 = uvs[index2].x - uvs[index3].x;
32     float uvCorrectDistY3_2 = uvs[index2].y - uvs[index3].y;
33
34     int xCount = 0;
35     int posCorrelationCount = 0;
36     int helpXCount = 0;
37     int helpPosCorrelationCount = 0;
38
39     getUVAxisWithSmallerErrorAndCorrelation(uvDistShouldBe1_2,
           uvCorrectDistX1_2, uvCorrectDistY1_2, out helpXCount, out
           helpPosCorrelationCount);
40     xCount += helpXCount;
41     posCorrelationCount += helpPosCorrelationCount;
42     getUVAxisWithSmallerErrorAndCorrelation(uvDistShouldBe1_3,
           uvCorrectDistX1_3, uvCorrectDistY1_3, out helpXCount, out
           helpPosCorrelationCount);
43     xCount += helpXCount;
```

```
44      posCorrelationCount += helpPosCorrelationCount;
45      getUVAxisWithSmallerErrorAndCorrelation(uvDistShouldBe3_2,
            uvCorrectDistX3_2, uvCorrectDistY3_2, out helpXCount, out
            helpPosCorrelationCount);
46      xCount += helpXCount;
47      posCorrelationCount += helpPosCorrelationCount;
48
49      float  maxDistTriangles = getBiggestValue(Mathf.Abs(uvDistShouldBe1_2),
            Mathf.Abs(uvDistShouldBe1_3), Mathf.Abs(uvDistShouldBe3_2));
50      vertexUVInformationListX =
            updateUVInformationList(vertexUVInformationListX, xCount,
            posCorrelationCount , index1, maxDistTriangles);
51      vertexUVInformationListX =
            updateUVInformationList(vertexUVInformationListX, xCount,
            posCorrelationCount , index2, maxDistTriangles);
52      vertexUVInformationListX =
            updateUVInformationList(vertexUVInformationListX, xCount,
            posCorrelationCount , index3, maxDistTriangles);
53      }
54  }
```

Code 3.2: The initialization of the uv coordinates for the x direction

```
1
2   private void getUVAxisWithSmallerErrorAndCorrelation(float uvDistShouldBe,
3   float  correctUvDistX, float correctUvDistY, out int uvXAxis, out int posCorrelation)
4   {
5       uvXAxis = (Mathf.Abs(Mathf.Abs(uvDistShouldBe) -
            Mathf.Abs(correctUvDistX)) < Mathf.Abs(Mathf.Abs(uvDistShouldBe) -
            Mathf.Abs(correctUvDistY))) ? 1 : 0;
6     if  (uvXAxis == 1) {
7         posCorrelation = (uvDistShouldBe < 0 && correctUvDistX < 0 ||
              uvDistShouldBe >= 0 && correctUvDistX>= 0 ) ? 1 : 0;
8     }
9      else
10    {
11        posCorrelation = (uvDistShouldBe < 0 && correctUvDistY < 0 ||
              uvDistShouldBe >= 0 && correctUvDistY >= 0)? 1:0;
12    }
13  }
```

Code 3.3: The method calculates the UV Scale Axis and the correlation. The function returns 1 if the x-axis o the texture has the smaller error. It returns 1 for the correlation if a smaller vertexValue means a smaller uv value.

The function `getUVAxisWithSmallerErrorAndCorrelation` takes as parameter the computed UV distance `uvDistShouldBe` and the correct distances `correctUvDistX` and `correctUvDistY`. First, the method compares the calculated distance with the correct distance by subtracting one from the other. The axis with the smaller difference yields the wanted axis.

The function is called for all three vertices of the triangle in order to make the algorithm more stable. Therefore, the function returns a variable `uvXAxis` that contains 1 if the UVs have to be modified in u-axis and 0 otherwise. The values are summed up and after evaluating all three vertices of the triangle, the texture coordinates are adjusted in u direction if the `uvXAxis` is bigger than 1. This additional computation is especially needed for triangles on edges of 3D models, where the distances are very small.

Furthermore, the method `getUVAxisWithSmallerErrorAndCorrelation` calculates the correlation between the texture coordinate and world coordinate of a vertex. It simply compares the sign of the calculated distance with the sign of the correct distance. If they are the same, the vertex has a positive correlation. If they are different, an enlargement of the world coordinate means a reduction of the texture coordinate. Especially when working with distances that are very small, the interpretation of the sign can be error-prone. Therefore, the algorithm computes the correlation for all 3 vertices, similar to the calculation of the UV Scale Axis.

The last step is now to update the information for the three vertices of the triangle (Code snippet 3.4). Obviously, the same vertex is processed multiple times because it is part of several triangles. In order to save the most significant value, our method saves the results calculated from the biggest triangle. At the end of the analysis phase the program holds a separate array for every scale axis `vertexUVInformationList` with the information on whether the texture coordinate should be scaled in u or v direction and whether an enlargement of the world coordinate leads to an enlargement or reduction of the texture coordinate.

```
1  private VertexUVInformation[] updateUVInformationList(VertexUVInformation[]
       vertexUVInformationList, int xCount, int posCorrelationCount, int indexVertex,
       float maxDistTriangle)
2  {
3      VertexUVInformation vertexUvInfo = new VertexUVInformation() ;
4      bool updateUv = false;
5      if (vertexUVInformationList[indexVertex] == null) {
6          updateUv = true;
7          vertexUVInformationList[indexVertex] = vertexUvInfo;
8      }
9      else {
10         vertexUvInfo = vertexUVInformationList[indexVertex];
11         if (vertexUvInfo.maxVertexDelta < maxDistTriangle)
12         {
```

```
13              updateUv = true;
14          }
15      }
16      if (updateUv)
17      {
18          vertexUvInfo.maxVertexDelta = maxDistTriangle;
19          vertexUvInfo.uvScaleAxis = (xCount >= 2) ? "X" : "Y";
20          vertexUvInfo.posCorrelation = (posCorrelationCount >= 2) ? true : false;
21          vertexUVInformationList[indexVertex] = vertexUvInfo;
22      }
23      return vertexUVInformationList;
24  }
```

Code 3.4: The method calculates the UV Scale Axis and the correlation

3.5.3 Results and Visualization of the Analysis Phase

The application allows the user to visualize the calculated Scale Rectangles and modify the thresholds through the Scale Rectangle Script in the Inspector (Figure 3.6). First, the size of the element is displayed. When scaling the model though the scale menu, the values in the Inspector are automatically adapted. Another possibility to scale the element is by simply writing the desired size in the provided field. The size is implemented in meters.

On the bottom of the Scale Rectangle Script the Scale Rectangles for every scale axis are shown. It is easy to see that an element can have multiple Scale Rectangles for one direction. However, in some situations no rectangle is computed at all. This can happen when the used threshold is too strong or too weak for the given 3D mesh. Therefore, the user can change the thresholds manually.

The threshold “tolerance-threshold” in the Script corresponds to the above mentioned buffer value ϵ. It specifies the maximum of the projection of the face normal on the scale axis. All triangles whose projection has a bigger value than ϵ must not be changed.

Furthermore, the threshold λ used for the analysis of the uv coordinates can be adjusted manually by changing the value “UV scalar threshold” in the Inspector. All triangles that are not perpendicular to the scale axis and therefore have a dot product that is smaller than λ need to be adjusted.

The Scale Rectangle Script displays the UV Scale Factor (mapping from uv To Vertex in cm) too. Normally, the user knows this factor because when creating new 3D meshes the same mapping is used. If the program calculates a wrong factor for any reason, the user can correct it by simply changing the value in the Inspector.

The application gives the user the possibility to view the calculated Scale Rectangles by simply clicking on the toggle field. This activates the Scale Rectangles for the different directions.

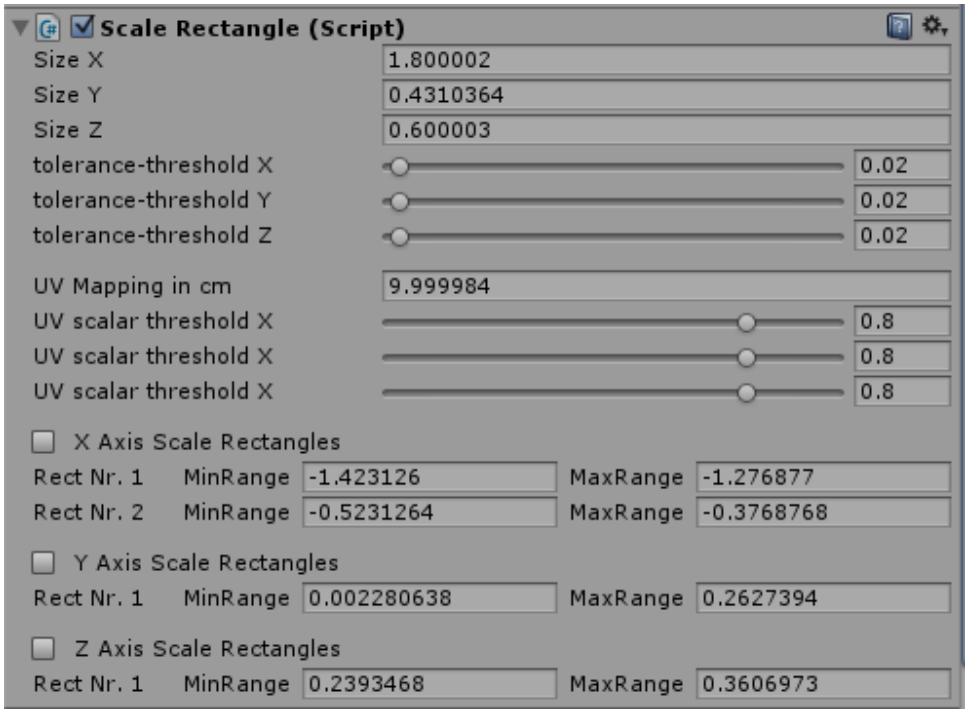


Figure 3.6: A screenshot of the Scale Rectangle Script in the Inspector.

3.6 Scaling Phase

After the analysis phase the 3D model has to be scaled at run-time. This scaling process is again divided into the adjustment of the world coordinates and the adjustment of the texture coordinates.

When scaling the element for example in the y direction, the algorithm runs though all previously computed Scale Rectangles as shown in code snippet 3.5. Depending on the ratio of the range, all coordinates of the vertices that are bigger than the minimum value of the range have to be adapted. If the 3D mesh consists of several Scale Rectangles for one direction, the algorithm starts with the smallest range. The positions of the following ranges have to be adapted too by summing up the already scaled `deltaPosition`.

```
1 public void scaleElementY(float deltaSize)
2 {
3     float newSizeY = this.maxYVertices - this.minYVertices + deltaSize;
4     if (newSizeY < this.minSize.y)
5     {
6         return;
7     }
8
9     Mesh mesh = GetComponent<MeshFilter>().mesh;
```

```
10      Vector3[] vertices = mesh.vertices;
11
12      float deltaPositionRanges = 0;
13      Vector2[] uvs = null;
14      foreach (Range range in this.scaleRectanglesY)
15      {
16          float deltaPosition = deltaSize * range.getRatio();
17          range.shiftRange(deltaPositionRanges);
18          deltaPositionRanges += deltaPosition;
19
20          for (int i = 0; i < vertices.Length; i++)
21          {
22              if (vertices[i].y > range.getMin())
23              {
24                  vertices[i].y = vertices[i].y + deltaPosition;
25              }
26          }
27          range.resize(deltaPosition);
28          uvs = uvHandler.updateUvsYDirection(mesh, deltaPosition, vertices);
29          mesh.vertices = vertices;
30          mesh.uv = uvs;
31      }
32      this.maxYVertices += deltaSize;
33      mesh.RecalculateBounds();
34      resetBoxCollider();
35
36      if (this.displayScaleRectX)
37              drawScaleRectangleX();
38      else if (this.displayScaleRectY)
39              drawScaleRectangleY();
40      else if (this.displayScaleRectZ)
41              drawScaleRectangleZ();
42  }
```

Code 3.5: Resizes the current Game Object along the y-axis by the *deltaSize*

```
1   public Vector2[] updateUvsYDirection(Mesh mesh, float deltaVertices, Vector3[]
        correctedVertices)
2   {
3       float deltaUV = deltaVertices * factorVertexToUv;
4     Vector3[] vertices = mesh.vertices;
5     Vector2[] uvs = mesh.uv;
6
7     for (int i = 0; i < vertices.Length; i++)
```

```
8   {
9       if (vertexUVInformationListY[i] != null)
10    {
11        if (correctedVertices[i].y != vertices[i].y)
12      {
13          VertexUVInformation vertexUvInfo = vertexUVInformationListY[i];
14        uvs[i] = updateUv(uvs[i], deltaUV, vertexUvInfo.posCorrelation,
              vertexUvInfo.uvScaleAxis);
15        }
16    }
17   }
18  return uvs;
19 }
```

Code 3.6: Adjust the uv coordinates of the mesh.

After adjusting the world coordinates for one Range, the uv coordinates have to be adapted too (Code Snippet 3.6). If the vertex has an entry in the `vertexUVInformationList` and if the world coordinates have changed, the uv coordinate of the vertex is adapted depending on the above computed correlation and UV Scale Factor.

3.7 Special Case X-Axis

Because of the fact that Unity saves the pivot point of a 3D mesh in the back bottom left corner of the mesh (the position with the biggest x value, the smallest y value and the smallest z value), scaling an element in the x-axis has to be executed somewhat differently than scaling it in the y and z direction. When increasing the size of an element, the mesh should not grow along the positive x-axis but along the negative axis. Thereby, the pivot position of the element can remain the same and does not have to be adjusted.

CHAPTER 4

Results and Evaluation

This chapter presents examples we achieved with our algorithm as well as basic performance tests. Most of the geometries we used for testing will be used in the 3D furniture generator too. However, in order to identify the flexibility of the algorithm we also tested it with more complex 3D models.

4.1 Test Setup

Our tests were performed on a system with an Intel i7-4770 quad-core processor and 16 GiB RAM. The used graphics card is a NVIDIA GeForce GTX 970. The tests were performed in Unity version 5.3.3f1 personal edition.

4.2 Scale Rectangles and Scaling

First, we analyzed the calculated Scale Rectangles with different thresholds and different 3D models. Then we tested whether all world coordinates were adjusted correctly after scaling the model in a specific direction. Using the wireframe mode in Unity allowed us to view only the mesh of the geometry and supported us in the analyzing phase. Furthermore, we tested if the texture coordinates were also scaled correctly by analyzing the adjusted texture.

In Figure 3.4 the Scale Rectangles for the three different scaling directions x, y and z are displayed. The images (a) and (b) display the Scale Rectangle for the x direction with different ϵ values. The smaller the value, the more accurate the Scale Rectangles. Furthermore, the figure shows that the algorithm is able to detect more than just one Scale Rectangle per scale axis.

Figure 4.1 shows the stool model before and after scaling it along the y direction. The blue box in Figure (b) displays the Scale Rectangle. Comparing the wireframes in Figure

(a) and (c) it can be seen that all vertices outside the Scale Rectangle are not modified at all.

Additional results of our algorithm are shown in the Figures 4.6, 4.7 and 4.8.

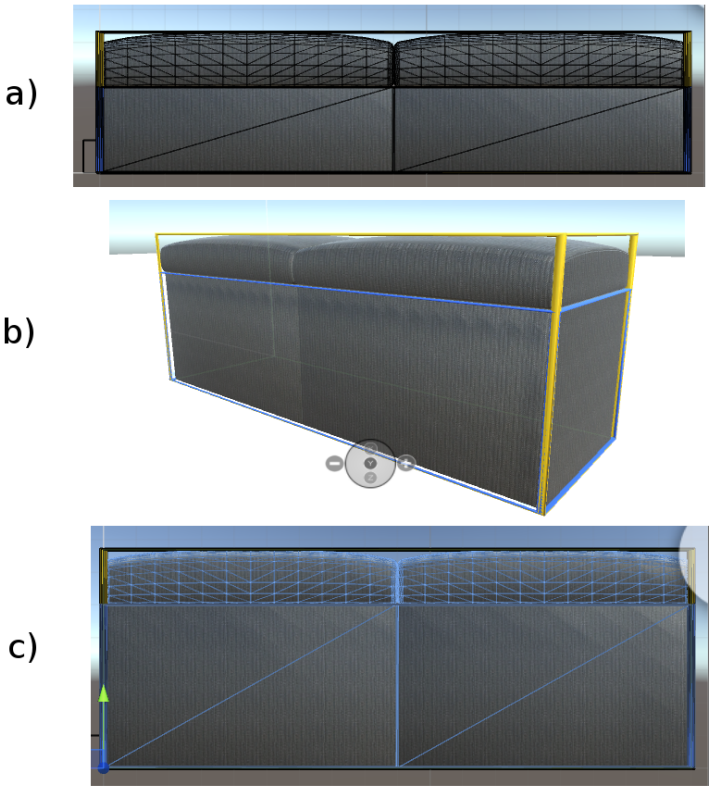


Figure 4.1: The figure shows the stool (a) before and in (b) and (c) after scaling it in the y direction. (b) displays the element in the Game View with the menu as it looks during the scaling process. Comparing the images (a) and (c) it can be seen how the vertices were modified.

4.3 Textures

In order to analyze the adjustment of the uv coordinates, we used a simple test texture, as shown in Figure 4.2. When enlarging the 3D model as well as when reducing its size, the texture coordinates were adjusted in a correct manner. The texture repeated itself without causing artifacts. Since the relation between world coordinates and uv coordinates were considered too, the texture was properly adjusted on the opposite side of the model as well.

As we already explained above, not all texture coordinates of the updated vertices have to be adjusted (Figure 3.5). Even if the vertices on top of the model have been shifted,

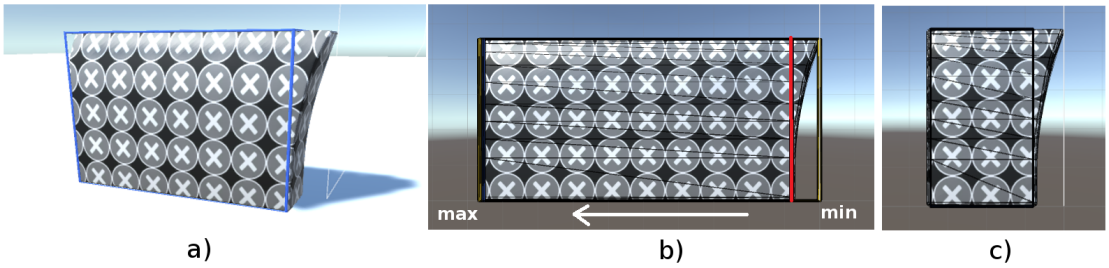


Figure 4.2: The 3D model of an arm rest with the test-texture. (a) Original model with the Scale Rectangle in z direction. It's size is 77.6 cm. (b) The armrest scaled to a size of 96 cm. The algorithm increased all vertices on the left side of the red line. (c) The armrest reduced to a size of 35 cm.

the texture coordinates remain the same. This is because the triangles are perpendicular to the scale direction.

4.4 Complex Models

Even if it is not important for the practical use, we tested the algorithm with more complex geometries. As shown in Figure 4.3, we applied the method to a bench. The left side shows the original geometry with the Scale Rectangles for the z-axis. After scaling it up, the characteristic features such as the stake in the middle are preserved. Furthermore, it can be seen that the algorithm can handle multiple textures on one geometry.

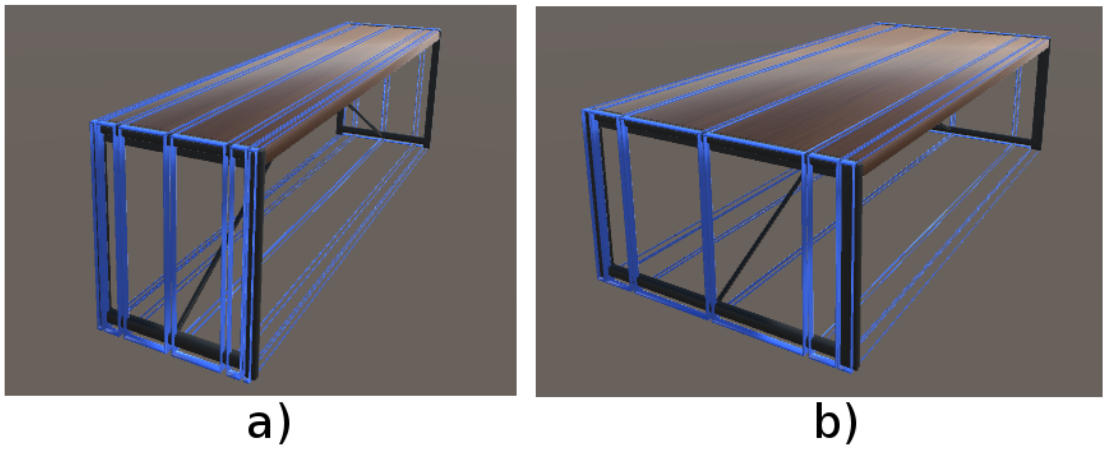


Figure 4.3: The figure shows the Scale Rectangles in z direction of a common bench (a) that is then scaled up (b). As can be seen the stake in the middle of the model has not changed.

It is important that the user has the ability to change the threshold ϵ manually. As shown in Figure 4.4 (a), the default threshold is not enough if the lampshade should be scaled as well. In this case, increasing the threshold leads to more Scale Rectangles, which has an impact on the performance of the algorithm at run-time. Furthermore, we noticed that the more complex a geometry is, the more Scale Rectangles are computed.

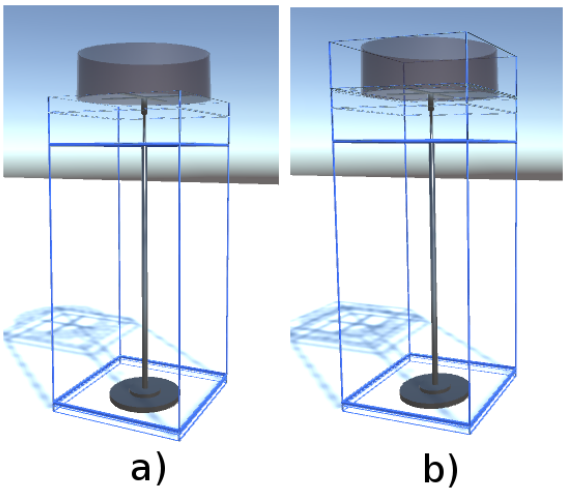


Figure 4.4: (a) The algorithm computes 6 Scale Rectangles when $\epsilon = 0.02$. (b) When $\epsilon = 0.02$ already 10 Scale Rectangles are computed.

The last geometry we used during the testing was a rather complex shelf (Figure 4.5). The parts of the shelf that contain no books were scaled extremely. By analyzing the section with the chemistry model, we noticed that only the foot of the model was scaled as well as the part of the books above it. The chemistry model itself remained the same. This geometry showed the limits of our algorithm. Even if it scaled the geometry correctly, the shelf was deformed in a rather unintuitive way.

4.5 Performance

In order to analyze the performance of the algorithm at run-time, we compared the FPS and CPU utilization of the algorithm when scaling two 3D models with different complexity.

The simple model of the armrest shown in Figure 4.2 has only one Scale Rectangle for each scale axis. Scaling the model was very fluid, the frame rate rose from initial 75 frames per second to more than 110 FPS. The CPU utilization rose from 9% to 14%.

The number of the Scale Rectangles has a strong impact on the performance of the application. The shelf for example has three Scale Rectangles for the z direction and one can already notice a little delay when scaling it continuously. When scaling it in the x

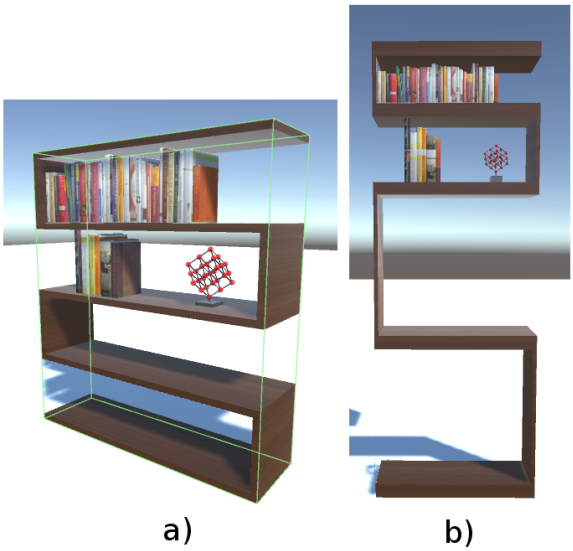


Figure 4.5: A complex geometry of a shelf. Even if our algorithm works correctly the shelf is not scaled in an intuitive way.

direction, the impact of the 49 Scale Rectangles is very strong. The frame rate shrinks to only 27 FPS and the CPU utilization rises to 18%.

Fortunately, these performance issues occur only when enlarging the model continuously. When scaling the model through the scaling menu in the Game View, the application calls the scaling function every frame. In the practical use, the scaling algorithm only has to be called when the user changes the width or height of a specific element. Therefore, the scaling function is only called once and the delay is nearly insignificant.

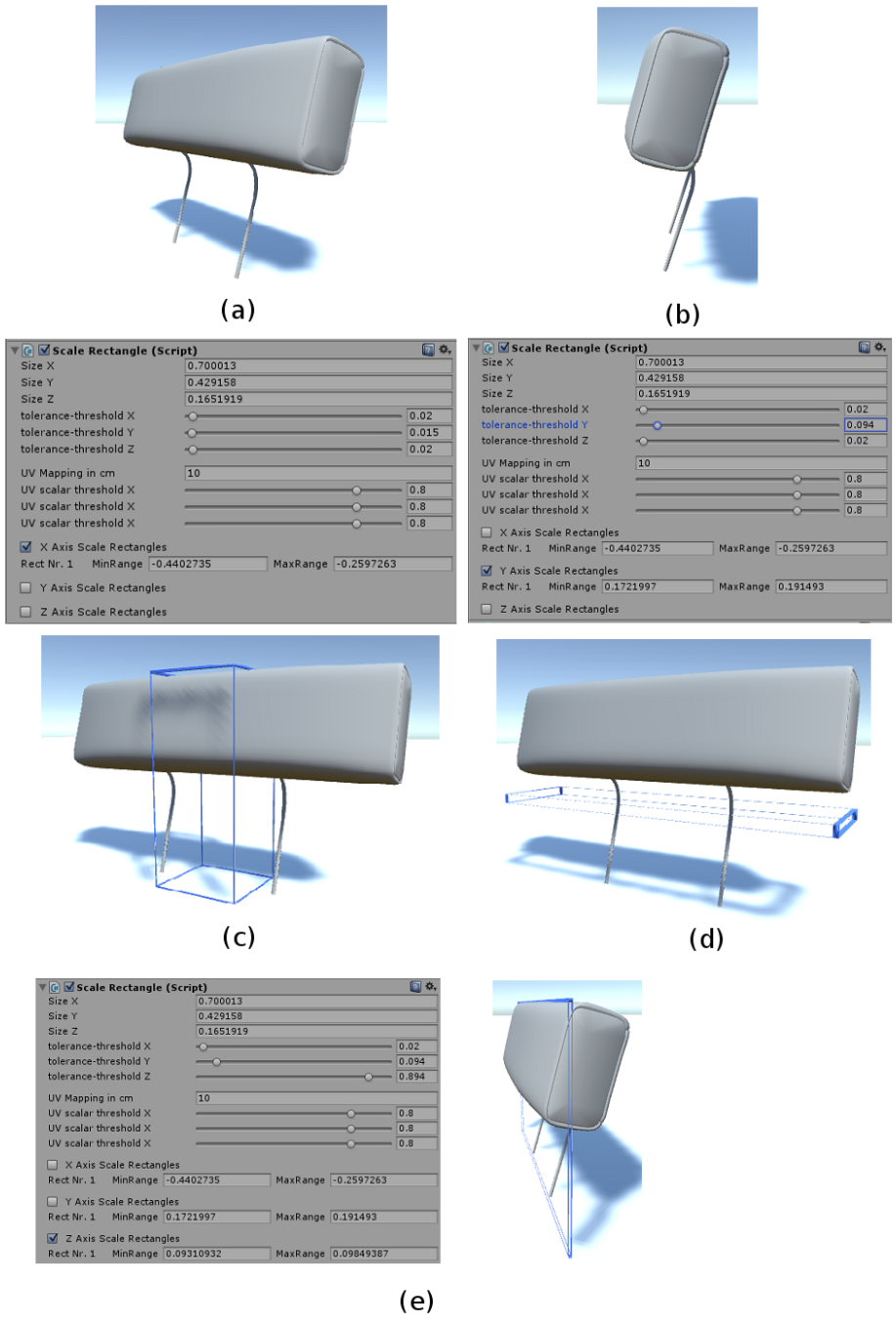


Figure 4.6: The figure shows a geometry that has no straight alignment with the x-axis, the y-axis or the z-axis. (a) and (b) show the geometry from two different positions. (c) displays the Scale Rectangle in x direction. It is the only direction where the algorithm finds a Scale Rectangle with the default threshold. For the other two scaling directions the user has to change the threshold manually as shown in (d) and (e).

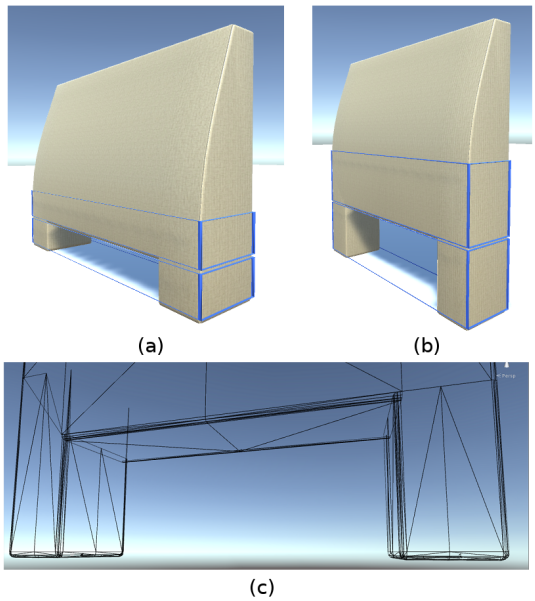


Figure 4.7: The geometry of another arm rest. Our algorithm finds two Scale Rectangles (a) when scaling along the x-axis (b). This is because of the fact that the edge of the geometry is composed of multiple vertices (c).

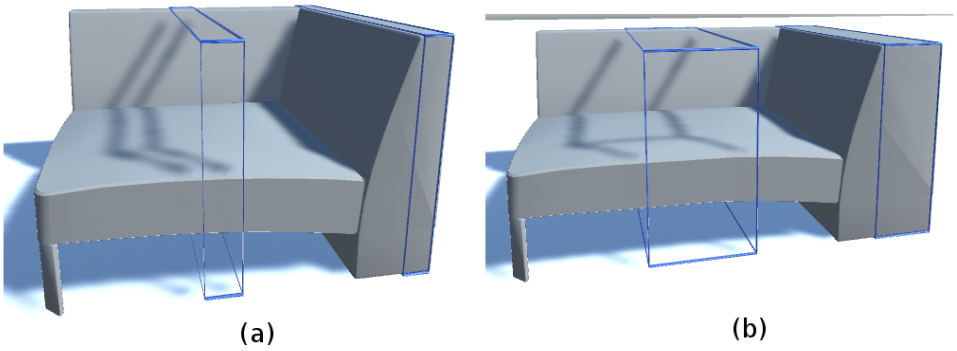


Figure 4.8: This piece of furniture is composed of several simple geometries. Nonetheless, the algorithm detects the correct Scale Rectangles.

CHAPTER 5

Conclusion and Future Work

Resizing of 3D models can be very useful when creating new models. However, naive resizing can create serious visual artifacts which destroy the characteristics of an object. This work presents a structure aware resizing algorithm that protects the model structures and takes the correct adaption of the texture into account as well. In order to facilitate the integration into existing projects, the algorithm was implemented using the game engine Unity. The application gives the user the ability to resize custom 3D models along predefined scaling directions at run time.

Our algorithm works well when working with rather simple 3D models that have a regular shape along a specific scale direction such as the arm rest, the stool, or the bench. Our method uses a threshold that indicates if the world coordinates of a mesh triangle must be changed and a threshold that specifies if a texture coordinate must be adjusted. Furthermore, the algorithm uses so-called Scale Rectangles to indicate all parts of the mesh that have to be scaled along a specific scale axis. For both thresholds a value can be defined that can be used over the entire geometry to compute the Scale Rectangles and to adjust the uv coordinates in a correct manner. Furthermore, only a small set of Scale Rectangles is computed. This leads to a good performance when scaling the model at runtime.

However, our method has a number of limitations. When scaling complex geometries as the shelf in Figure 4.5 or the lampshade in Figure 4.4 the algorithm computes unintuitive and unnatural results. This is because the algorithm computes too many Scale Rectangles. Complex geometries consist of many parts, where the mesh triangles are not parallel to the scaling axis and therefore vulnerable to the scale. However, some regions are parallel to the scaling axis and produce a Scale Rectangle even if these regions are very small. For every Scale Rectangle the algorithm repeats the resizing calculation and the adjustment of the uv coordinates. Therefore, a big number of Scale Rectangles quickly leads to a bad performance.

In future work, we would give the user more possibilities to influence the calculation of the Scale Rectangles. When scaling the piece of furniture in Figure 4.8, one might prefer to scale only the seat and ignore the arm rest. Because of this, it would be reasonable to allow the user to delete selected Scale Rectangles after their computation or define new ones manually. Figure 4.6 (e) shows that sometimes it is impossible to find a Scale Rectangle that makes sense for a specific scale direction. The possibility to define custom scaling directions would therefore be very helpful as well. Furthermore, we would like to optimize the performance when scaling a 3D model. There are times when Scale Rectangles are so small that they have no visual impact on the results. Because of this, it would be reasonable to only allow Scale Rectangles bigger than a specific threshold. Alternatively, the algorithm could merge multiple Scale Rectangles that are close together into one.

Analyzing the state of the art leads to the conclusion that there is no general solution for scaling a model without destroying its geometric features. Depending on the structure of the geometry (man-made or not) and the field of application (e.g. at run-time or for architectural models) different algorithms have to be chosen. Nevertheless, because of its numerous benefits, structure-aware shape processing will remain a topic of research well into the future.

Bibliography

[AFS06] Marco Attene, Bianca Falcidieno, and Michela Spagnuolo. Hierarchical mesh segmentation based on fitting primitives. *The Visual Computer*, 22(3):181–193, 2006.

[ARSF07] Marco Attene, Francesco Robbiano, Michela Spagnuolo, and Bianca Falcidieno. Semantic annotation of 3d surface meshes based on feature characterization. In *International Conference on Semantic and Digital Media Technologies*, pages 126–139. Springer, 2007.

[BKP+10] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. *Polygon mesh processing*. CRC press, 2010.

[BMV01] Pál Benkő, Ralph R Martin, and Tamás Várady. Algorithms for reverse engineering boundary representation models. *Computer-Aided Design*, 33(11):839–851, 2001.

[Bot05] Mario Botsch. *High quality surface generation and efficient multiresolution editing based on triangle meshes*. Citeseer, 2005.

[Coq90] Sabine Coquillart. *Extended free-form deformation: a sculpturing tool for 3D geometric modeling*, volume 24. ACM, 1990.

[GG04] Natasha Gelfand and Leonidas J Guibas. Shape segmentation using local slippage analysis. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 214–223. ACM, 2004.

[GSMCO09] Ran Gal, Olga Sorkine, Niloy J Mitra, and Daniel Cohen-Or. iwires: an analyze-and-edit approach to shape manipulation. In *ACM Transactions on Graphics (TOG)*, volume 28, page 33. ACM, 2009.

[JBPS11] Alec Jacobson, Ilya Baran, Jovan Popovic, and Olga Sorkine. Bounded biharmonic weights for real-time deformation. *ACM Trans. Graph.*, 30(4):78, 2011.

[KCŽO08] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Transactions on Graphics (TOG)*, 27(4):105, 2008.

[KSSCO08] Vladislav Kraevoy, Alla Sheffer, Ariel Shamir, and Daniel Cohen-Or. Non-homogeneous resizing of complex models. *ACM Transactions on Graphics (TOG)*, 27(5):111, 2008.

[MGP06] Niloy J Mitra, Leonidas J Guibas, and Mark Pauly. Partial and approximate symmetry detection for 3d geometry. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 560–568. ACM, 2006.

[MWZ+13] Niloy Mitra, Michael Wand, Hao Richard Zhang, Daniel Cohen-Or, Vladimir Kim, and Qi-Xing Huang. Structure-aware shape processing. In *SIGGRAPH Asia 2013 Courses*, page 1. ACM, 2013.

[OBB+13] Alexandrina Orzan, Adrien Bousseau, Pascal Barla, Holger Winnemöller, Joëlle Thollot, and David Salesin. Diffusion curves: a vector representation for smooth-shaded images. *Communications of the ACM*, 56(7):101–108, 2013.

[OBS04] Yutaka Ohtake, Alexander Belyaev, and Hans-Peter Seidel. Ridge-valley lines on meshes via implicit surface fitting. *ACM transactions on graphics (TOG)*, 23(3):609–612, 2004.

[PMW+08] Mark Pauly, Niloy J Mitra, Johannes Wallner, Helmut Pottmann, and Leonidas J Guibas. Discovering structural regularity in 3d geometry. In *ACM transactions on graphics (TOG)*, volume 27, page 43. ACM, 2008.

[SF98] Karan Singh and Eugene Fiume. Wires: a geometric deformation technique. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 405–414. ACM, 1998.

[SMW06] Scott Schaefer, Travis McPhail, and Joe Warren. Image deformation using moving least squares. In *ACM transactions on graphics (TOG)*, volume 25, pages 533–540. ACM, 2006.

[SP86] Thomas W Sederberg and Scott R Parry. Free-form deformation of solid geometric models. *ACM SIGGRAPH computer graphics*, 20(4):151–160, 1986.

[ZFCO+11] Youyi Zheng, Hongbo Fu, Daniel Cohen-Or, Oscar Kin-Chung Au, and Chiew-Lan Tai. Component-wise controllers for structure-preserving shape manipulation. In *Computer Graphics Forum*, volume 30, pages 563–572. Wiley Online Library, 2011.