FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Texturing of 3D Objects using Simple Physics and Equilateral Triangle Patches

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Matthias Glinzner
Matrikelnummer 0726342

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Dr.techn. Ivan Viola
Mitwirkung: Mathieu Le Muzic, MSc

Wien, 27. Juli 2016

Matthias Glinzner

Ivan Viola

Texturing of 3D Objects using Simple Physics and Equilateral Triangle Patches

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Matthias Glinzner
Registration Number 0726342

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dipl.-Ing. Dr.techn. Ivan Viola
Assistance: Mathieu Le Muzic, MSc

Vienna, 27th July, 2016

Matthias Glinzner	Ivan Viola

Erklärung zur Verfassung der Arbeit

Matthias Glinzner
Krottenbachstraße 247/4/1

Acknowledgements

Thanks to Ivan Viola for support and valuable input. Thanks to Mathieu Le Muzic for the idea and his support as well.

Kurzfassung

Die Visualisierung von Zellen im allgemeinen und von Zellmembranen im besonderen bildet die Grundlage für die hier vorliegende Arbeit. Diese Phospholipidmembranen grafisch darzustellen ist das Ziel der vorgestellten Methoden. Besonderes Augenmerk wurde dabei der nahtlosen Texturierung einer Oberfläche im dreidimensionalen Raum gewidmet, um durch Verwendung entsprechender Texturkacheln die Speichernutzung gering zu halten.

Der entwickelte Algorithmus erstellt zunächst ein Texturmesh, das der Oberflächenstruktur eines vom Nutzer bereitgestellten Input-Meshes treu ist. Dieses weist die Eigenschaft auf, eine Triangulierung, bestehend aus gleichseitigen Dreiecken, zu besitzen. Dies wird erreicht, indem die Punkte, bevor sie trianguliert werden, durch Simulation von abstoßenden Kräften zwischeneinander, auf der Oberfläche des Inputmeshes repositioniert werden. Besagte Eigenschaft erlaubt im Anschluss eine triviale Zuordnung von ebenfalls dreieckigen Texturkacheln. Dadurch ist die nahtlose Texturierung der Oberfläche hergestellt.

Neben Details der Implementierung werden exemplarische Ergebnisse ebenso dargestellt wie eine Performance-Analyse; diese zeigt Vor- und Nachteile, besonders in der Laufzeit, an. Des weiteren wird ein kurzer Überblick über verwandte Themen und frühere Arbeiten gegeben.

Das verwendete Framework hierbei ist Unity 3D.

Abstract

Visualizing cells, in particular cell membranes, is the inspiration for this work. The goal of the presented methods is the efficient visualization of phospholipid membranes. A prominent role hereby plays the concept of seamlessly texturing a surface in three-dimensional space. By using suitable texture patches, memory consumption can be kept low.

The developed algorithm first creates a texture mesh that stays faithful to the surface structure of a user-provided input-mesh. This texture mesh consists of equilateral triangles. The triangulation is achieved by first simulating repulsion between the vertices making up the texture mesh. This way they are moved around on the surface of the input-mesh until they are uniformly distributed. Mapping texture onto equilateral triangles becomes trivial if triangular texture patches are assumed as well. Thus, seamless texturing is achieved.

The implementation is described in detail, followed by the demonstration of results. Also, an exemplary performance-analysis is given, highlighting benefits and shortcomings of the algorithm, especially concerning runtime. Additionally, a short overview of related and prior work is given.

The used framework is Unity 3D.

Contents

CHAPTER 1

Introduction

Trademarks used in this thesis generally belong to respective owners.

Cells are the basic structural units of living organisms. The various parts that make up a cell are separated from each other by membranes assembled from phospholipids. Modelling this phospholipid membrane is the motivation for this thesis. If the membrane is represented by a mesh of arbitrary complexity, the question is how to populate this mesh with phospholipids while using memory efficiently and achieve fast rendering performance.

A phospholipid membrane has some unique characteristics. First, since membranes have distinct shapes, the utilized algorithm mostly does not have to deal with sharp angles. Also, their surface does not selfintersect. Secondly, while the general outline of a membrane is known, its surface oftentimes has a degree of randomization to it. Incorporating the described properties into one concept lead to work presented in this thesis.

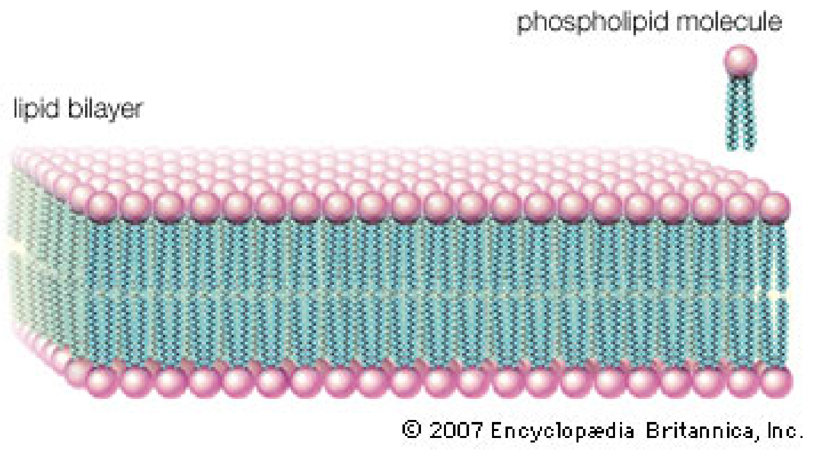


Figure 1.1: An example of a lipid membrane. Source: `https://www.britannica.com/science/lipid-bilayer26.07.2016`

In order to faithfully model a phospholipid membrane in computer graphics, seamless texturing of the mesh representing the membrane is a necessity. To conserve memory, one or more repeating texture patches are applied to the surface of the mesh. Moreover, these patches should be applied randomly to avoid visual aritfacts.

Applying texture to an arbitrary surface in three dimensional Euclidean space can be a complicated procedure; Depending on the given object, resources have to be devoted to correctly map the texture onto the surface while at the same time avoiding visible distortions and seams. Additionally, if the object's shape is not predetermined, the texture coordinates have to be computed at runtime which poses a problem on its own. Random texturing adds the task of choosing the right texture samples to this list. Since the human eye is especially proficient in detecting repeating patterns, care has to be taken to not only create a truly random distribution of texture patches but also to make sure that neighbouring patches fit well with each other.

To facilitate the texturing process, a special texture mesh can be used. Its main feature are uniformly distributed vertices, this way making assignment of texture coordinates easier.

Concentrating on the core problem of randomly texturing an arbitrary surface in three dimensional Euclidean space, the goal of this thesis is to create a simple yet versatile script with which it is possible to process molecule-like 3D objects. This processing consists of applying texture and displaying the textured object on screen. To this end, the following approach will be taken: A texture mesh is created, representing the surface of the input mesh. Vertices of the texture mesh are uniformly distributed, resulting in an equilateral triangulation. This triangulation will be achieved via a physics based approach. Then, texture patches are applied at random to the texture mesh. These patches are equilateral triangles as well, making mapping trivial. Finally the output is displayed.

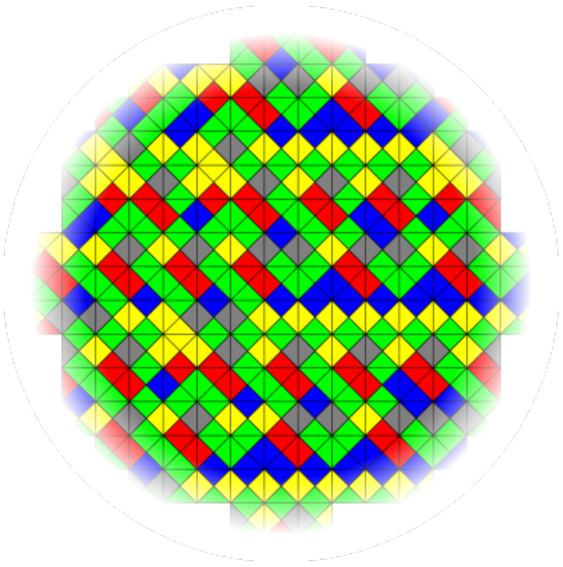


Figure 1.2: An example for a random texturing pattern (Wang tiles). Source: `https://en.wikipedia.org/wiki/Wang_tile26.07.2016`

CHAPTER 2

Related Work

This work is mainly about texturing, with mesh manipulation playing an important role as well. Both these aspects have their own scientific fields dedicated to them. Since this particular implementation will be physics based, particle physics simulations are a valuable source of information and will also be mentioned in this chapter.

2.1 Texturing

Texturing was first introduced by Edwin Catmull in his thesis "A Subdivision Algorithm for Computer Display of Curved Surfaces" [Cat74]. Here, it is described as photographs being "...'mapped' onto patches (of curved surfaces) thus providing a means for putting texture on computer generated pictures." The notion of mapping texture onto a surface is an integral part of the texturing process. Another, equally important step is choosing or generating the texture. Finally the whole procedure can be preprocessed or takes place at runtime.

Creating the texture itself can be done either by hand or automatically. The former is time-consuming but exact, the latter fast but care must be taken to avoid visible artifacts. A number of ways exist to procedurally generate textures. Simple methods include using sine-functions or Perlin Noise to generate textures from scratch. A more complex idea is the principle of reaction-diffusion: Two substances interact with each other and themselves. The reacting agents can be represented by mathematical equations; by simulating such equations over a period of time textures can also be created [Tur91]. These resemble the shapes and patterns in the fur of leopards or giraffes. An alternative method works with a user-input. From an input image a number of texture patches that follow specific constraints are created.

Correct mapping of texture patches onto an object's surface is an equally important step. The simplest methods used would be linear and trilinear projection as well as 3D-textures.

The first method projects a two-dimensional texture into 3D-space along one coordinate axis, the second uses three textures, one for each axis. Both these techniques are rather coarse and hardly suited for complex surfaces. A 3D-texture on the other hand foregoes the projection completely but is very memory-intensive. More sophisticated methods include cube mapping and polycube mapping. Finally chart segmentation breaks down a surface into small parts called charts that share a beneficial characteristic (like equal face normals), which are then stored in a texture atlas for easy access [Win11]. A special type of maps are jump maps: For each pixel they store a set of similar pixels in the sample image, weighted by similarity. During texture synthesis exchanging one pixel with one of its jump-neighbours is allowed [ZG03].

There are many ways of combining these methods that offer different benefits and yield different results. Turk employs the aforementioned technique of reaction-diffusion to generate texture patches [Tur91]. It is combined with a mapping scheme that relies on an even distribution of the vertices that make up the mesh of the 3D-model the texture is applied to. Since this distribution cannot be guaranteed for an arbitrary mesh, a second mesh with the desired quality has to be created. First a number of new vertices on the surface of the existing mesh is created. These are then repositioned using a relaxation technique and become the center of newly formed Voronoi-regions. In a last step said regions fulfil the role of cells in a reaction-diffusion process, this way creating a seamless texture. A different way to work with similar textures is to regard them as progressively-variant [ZZV+03]. This means that although a pattern exhibits an overall continuous variety it is stationary in small point neighbourhoods. Utilizing a texton-mask the user is able to mark said stationary features. These can then be reproduced on a larger scale without repetition using field distortion. In this case, seams are masked by feature-based warping and blending. Other techniques build upon the principle of working with a separate texture mesh. A direct side effect of uniform point distribution on a surface is the possibility to get a triangulation consisting of equilateral triangles. Mapping these to appropriate texture patches makes a seamless coverage possible as well [NC99]. Moreover, if the patches obey certain rules regarding edge connectivity, a pseudo-random tiling can be achieved. Nieser et al. describe a similar idea but instead of triangles, hexagons are used to cover a mesh [NPPZ10]. This shape, as well as a triangle, is well suited in this case since it is one of the basic geometric shapes that can cover a plane without holes when used as tiling. The previous two examples work well with triangulated meshes—an alternative exists for quad-meshes, replacing triangular patches with rectangles [SYXA+11].

Arranging rectangular texture patches usually utilizes Wang-tiling. Ensuring the necessary edge characteristics of these squares, as few as eight texture patches suffice to non-periodically cover a plane [CSHD03]. Transferring this problem into three dimensional space, Fu et al. combine the idea of polycube mapping with Wang-tiles [FL05]. First a complex object is mapped to a quad-based representation where the size of a quad is proportional to the local curvature of the surface. In a second step, every quad is covered with square texture patches. Since these follow the edge layout of Wang-tiles, seamless coverage is possible.

Another set of methods avoids irregularities by examining edges individually. One such method would be to select texture patches of different sizes and shapes from a sample image, ensuring edge connectivity for each patch. The surface is then hierarchically structured, creating a subdivision for every element in a lower hierarchical level. For example, the lowest level contains the whole surface, whereas higher levels consist of smaller and smaller parts of the respective surface one level below them. To texture this surface, each tier of the hierarchy is tested for distortions. If these are sufficiently low, a texture patch of appropriate size is applied. Otherwise the next smaller subdivision is examined in a coarse-to-fine approach [SCA02]. A variant of the above is employed by Wei et al.: Here each mesh vertex is parametrized locally using a rectangular neighbourhood [WL01]. The sample image is then searched for similar neighbourhoods which are mapped onto the surface accordingly.

Textures can also be quilted together from patches [MK03]. This consists of two phases: Preprocessing and synthesis. During the first phase each texture pixel is labelled by its neighbourhood. In the second phase triangular patches the size of mesh triangles are cut out and stitched together. The first step is necessary to make sure the edges of the patches correspond to each other. The whole process starts with a triangle on the surface, chosen at random, and is then continued outward. As an optional step edge blending can be performed.

A technique of dealing with visual artifacts that was mentioned before is texture blending. This method blends overlapping textures together, hiding the seam that they would otherwise create. A straightforward example for this approach are lapped textures [PFH00]. Assuming that a local parametrization exists even when a global one does not, texture patches are repeated across a surface, overlapping each other. Alpha blending is used to reduce artifacts in the overlapping regions and a tangential vector field controls alignment and scale as per user input. For patch-placement, a random point on the surface is chosen and the patch is grown from there. Once the distortion becomes excessive, another patch is placed.

The above-mentioned chart segmentation is used by Ying et al., as well as Kolar et al. [YHBZ01], [KCD15]. Here the object's surface is covered by an atlas of overlapping charts. As with other methods, texture patches are created from a sample image and edge connectivity is ensured by checking for similarities between them.

Expanding on the general idea of mapping a surface into a different domain is the concept of shell mapping [PBFJ05]. Here a bijective function is utilized that maps each surface point to a point on a shell map which consists of 3D-objects as well, essentially replacing a two-dimensional texture with a mesh. This is especially useful in contexts where, to represent a real-world object with satisfactory graphical fidelity, finely detailed surface structures have to be rendered as well.

Finally an example for a molecular visualization context is described in [WSB14]. This work introduces a framework for representing cells on a mesoscopic as well as on a molecular scale. To avoid the problems usually associated with variable resolutions, objects are not textured on their whole surface but only on a user specified region instead. This region can be changed at runtime, thus making it possible for the whole surface to be inspected while maintaining a distortion-free texture mapping.

2.2 Mesh manipulation

As mentioned in the previous section the mesh plays an important role in the texturing process. Following is a short description of basic properties of meshes as well as common operations performed on them. A mesh, or wire-frame model, is a representation of an object in three-dimensional space consisting of vertices and edges connecting the vertices. All these connections taken together form the surface of the mesh. A surface in general can be viewed as the two-dimensional boundary of a 3D-solid [BKP$^+$10]. It can be described using parametric or implicit representations. Examples for the former include spline and subdivision surfaces as well as triangle meshes. Subdivision surfaces require a coarse control mesh and a set of rules describing how points can be inserted into this mesh to create a finer, more detailed output mesh [Cas12]. The latter—also called volumetric representation—is achieved by deciding for each point of the embedding space if it lies inside, outside of or on the object. Conversions exist between representations, for example the marching cube method [LC87].

This algorithm works on the principle of partitioning space into a three-dimensional grid, with every grid cell being a cube. Each cube's corners are then marked as either inside or outside a given mesh. Depending on the configuration of each cube a triangular mesh is then constructed by processing the grid cells one by one [NY06]. Since it's introduction, this algorithm has been extended in various ways, for example to process higher-dimensional data sets or time-varying data. Efficiency has also been improved by incorporating the concept of octrees. Different output types exist nowadays as well, including quads and spline meshes.

To store mesh data, different structures can be employed [BKP$^+$10]. A simple method is a face-based data structure. Here, individual polygonal faces, defined by their vertex positions, are stored. The trade-off hereby is the loss of connectivity information. To retain this kind of information, several edge-based data structures exist. They are in turn refined by the concepts of halfedge- and directed-edge-based concepts. These allow for finer control over a surface while increasing memory consumption. These boundary representation data structures are able to represent vertices and edges, as well as surfaces, explicitly, while also storing information about their components' respective relationships [ASB13].

When building a mesh, the local and global structure is of importance. The local structure defines the properties of mesh elements, whereas the global structure describes their interaction. Mesh elements can be discerned by their type and shape as well as their density. Prominent types are triangles and quadrangles, while being isotropic or anisotropic determines an element's shape. Finally, mesh elements can either be uniformly or nonuniformly distributed. Global structure mainly depends on vertex neighbourhood—vertices are called regular if they have a certain number of neighbours (depending on their type and placement). Once these characteristics are decided upon, edges can be constructed between vertices. An important role hereby play Voronoi-diagrams and their dual structure of Delaunay-triangulation, more specifically restricted Delaunay-triangulation [DH92], [KS16]. Notable because of its simplicity is the Bowyer-Watson-algorithm that generates such a triangulation by adding one point at a time and then checking if Voronoi conditions are met [Reb93]. Delaunay-triangulation can also be used to refine or optimize already existing meshes. Here certain favourable attributes of triangles, that are created this way, are exploited, namely their angle size and their area [She02].

Improving upon triangulated meshes, especially encoding-wise, are triangle strips. Instead of storing each face individually they are grouped together, exploiting spatial coherence, thus reducing redundancy [VdFG99]. As with other methods, triangle strip creation can be either done as a conversion step from a triangle mesh or dynamically as an addition to a triangulation process [ESEK+00].

If the desired shapes are quadrangles, the spawning algorithm falls into one of four classes: Triangle to quad conversion, patch based, parametrization based or Voronoi based [BLP+13]. One such conversion uses integer-grid maps, a class of piece-wise linear maps that map a grid of integer isolines non-degenerately into a quad mesh [BCE+13]. Although a conversion, the technique falls into the patch based domain; here, instead of mapping triangles onto square patches, the original surface is mapped directly.

As described by Bommes et al., quadrangle based meshes are actually preferable to triangle meshes in a number of scenarios [BLP+13]. Polygon modelling, for example, benefits from this representation since quadrangles more easily follow the lines of an artist's pen stroke. Also—for similar reasons—texture mapping onto quad meshes is quite intuitive. The downside of quadrangles compared to triangles is their more complex orientation in space and the resulting defining values.

It is also possible to combine the two type paradigms using a multiresolution atlas structure [MVS14]. This way especially dense-polygon meshes become easier to render. Meshes can also be used to describe something other than surfaces: in a biological context for example, they are used to simulate blood flow [MCG+12]. In this case a three-dimensional triangulation using tetrahedrons is implemented.

Finished models often times consume a lot of memory. A number of simplification methods exist to deal with this problem—vertex clustering, incremental decimation and shape approximation, to name just a few [BKP+10]. They all work under the common principle of reducing memory consumption by sacrificing object detail. A simple example for incremental decimation works as follows: One by one the vertices are examined and checked if they can be safely removed. This means that by removing the vertex the topology remains unchanged. If a vertex is removed, the resulting hole is then filled with a new triangulation [SZL92]. Sometimes it is not necessary to reduce vertex numbers but instead to reposition them while maintaining a faithful representation of an object's surface. Turk proposes such a method to achieve uniform mesh element distribution [Tur92]. A number of new vertices are introduced into a mesh and then repositioned, using a relaxation procedure. The old vertices are then decimated one by one, creating a uniform point distribution.

Finally, techniques exist to repair meshes damaged either by data loss or sampling errors. Examples of such errors include holes, singular vertices, overlaps and inconsistent face orientation. Algorithms for model repair can be classified as either surface-oriented or volumetric [BKP$^+$10]. The former operate directly on the input data whereas the latter convert the input mesh into volumetric space from where an output is created [ACK13].

The techniques described up to this point all have in common that they view an object's surface as a 2D-manifold embedded in 3D-space, essentially reducing the triangulation problem in space to a problem in a plane. An alternative approach is to consider point clouds as a surface representation. Algorithms that triangulate such a point cloud fall into one of three categories: Sculpting-based approaches, contour-tracing approaches and region-growing approaches [LTW04]. In sculpting-based approaches, the three-dimensional Delaunay-triangulation is constructed first, resulting in a solid object made up of tetrahedrons. This object is then decimated, removing vertices until only the surface remains. Contour-tracing methods on the other hand use a signed distance function to approximate the model. Region-growing techniques start with a seed triangle patch that is then grown outwards by adding edges. At termination, the hull is completely triangulated.

2.3 Physics, Math and Tools

In this section a few especially important concepts for this thesis are introduced. They range from physics calculations to open-source libraries. As described previously, repositioning of vertices is sometimes necessary. One possible solution for finding new positions is based on dynamic physics simulations. Here, each vertex is viewed as a physical object able to interact with other objects in the same domain.

When considering such simulations, two of the most important models are particle systems and rigid-body systems [Cou13]. Rigid-body computations usually take into account the mass and shape of an object while at the same time assume them staying unchanged throughout the simulation. Since rigid-bodies, in contrast to particles, are three-dimensional, it is also possible to compute intersections and collisions between objects. Since rigid-bodies are inherently unable to inter-penetrate, the resulting physics calculations have to take this into account. Once collision is detected, forces between colliding objects have to be calculated that prevent inter-penetration and thus simulate the actual motion of these objects according to their shape and mass [Bar89] or friction [Bar94].

To detect a collision in the first place, various detection algorithms exist. Simple implementations rely on testing vertex-triangle-penetration or utilizing objects' faces' orientation by testing if a given vertex lies behind (inside) another triangle [MW88]. Since most simulations deal with more than two potentially colliding bodies, more sophisticated algorithms usually start with reducing the number of intersection tests that have to be performed. Possibilities to achieve this include spatial subdivision or space decomposition in general, sort-and-prune-methods or event-driven approaches. A way to speed up collision testing further is by using bounding volumes. These range from spheres over axis aligned bounding boxes to partitioning into voxelised containers [LAM01].

Once collision is detected between two rigid-bodies, a collision response has to follow. This is usually application of force. A very simple way is to simulate a spring between the two touching points. This spring can be of varying elasticity, this way controlling the force exerted equally in both directions, pushing the objects apart [MW88]. An alternative is to provide analytical solutions. These solutions conserve momentum during a collision, resulting in a new angular and linear velocity for all involved objects.

Particle systems are especially useful when individual points' properties are not as important as the behaviour of the object they make up, for example fluids and gases. In contrast to rigid bodies, these objects are not represented by a well defined surface but instead are clouds of particles defining their volume. These particle systems are not necessarily static; often times new particles are introduced into the system and old ones removed at time intervals. This gives particle clouds an intrinsic non-deterministic character [Ree83]. Although generally associated with liquid and gas-like object simulation, particles can also be used to represent solid bodies. This technique can be used to facilitate state changes within the model, for example freezing water [Ton91]. The forces simulated between particles can range from simple point-mass systems to complex implementations of computational fluid mechanics or spring connections [Cou13].

The basic principle behind such physics simulations is as follows: At first, for every actor in the simulated system, all of the influencing forces are determined and computed. Then every one of them is repositioned accordingly. Repeating these steps, the whole system's state at a given time is always the result of a previous state, this way simulating physics forces [SFM12]. To summarize, the position of an object in such a system is the result of a time integration method used to update velocities according to accelerations. The velocity determines the final position, whereas acceleration is derived from internal and external forces using computations based on Newton's second law of motion [BMO$^+$12].

An alternative to these force-based methods exists in the form of position-based simulation methods. Instead of evolving positions as described through numerical integration of accelerations and velocities, position-based approaches compute positions directly [BMO$^+$12].

Another important aspect when dealing with object interactions and physics are metrics. A prominent role in three-dimensional space plays the Euclidean distance, measuring the direct distance between two points. A classic algorithm computing the Euclidean distance between two convex objects is the Gilbert-Johnson-Keerthi distance algorithm

[Lin09]. As a side effect, the objects' respective points closest to each other are also determined. Geometrical objects are described using supporting mapping functions and in a simplification step one object's distance to the origin is computed using the Minkowski difference. Finally the actual distance between the two objects is obtained, relying on simplices contained in the distant object.

But, dealing with 2D-surfaces embedded into 3D, measuring distance on the surface instead can produce useful data. Geodesic paths are exactly that—a distance measure on a surface. Examples for their application include mesh parametrization, mesh segmentation or the definition of surface vector fields [SSK+05]. Although far from trivial, many algorithms exist that solve geodesic distance exactly by employing window propagation techniques or sequence trees [CHK13]. Sometimes though, an approximation is sufficient—one such approach, using wavefront-propagation, is described by Tang et al. [TWZZ07]. Other possibilities include graph approximation and fast-marching methods, propagating distances across triangle faces as well as edges of a mesh [CHK13].

This final section is devoted to presenting a representative set of tools and libraries that can be used to facilitate calculations necessary to perform some of the already described tasks. The first entry, "MeshLab", is an open source mesh processing system [CCC+08]. Its interface is modelled as a mesh viewer, allowing for a variety of tasks to be carried out. These include selection and deletion of whole meshes or portions, smoothing and colouring. Many different mesh data formats are supported as well as point clouds. Aside from that, it is possible to repair and remesh loaded models as well as measuring distances. Lastly MeshLab can also be used as a range map processing tool.

Second on this list is the "Computational Geometry Algorithm Library" or "CGAL", a C++ library of data structures and algorithms. The project was started 1995 and is supported to this day [FP09]. It consists of a kernel, a basic and a support library. The kernel contains simple geometric objects and operations thereupon, whereas the basic library offers more complex data structures. These include convex hulls, triangulations and polygons.

The last entry is the "Point Cloud Library", a library for point cloud processing, written in C++ [RC11]. It supports data filtering (downsampling, outlier removal, projections), feature extraction (surface normals, boundary point estimation, curvatures), cluster extraction, surface meshing and convex hulls. Since PCL has its own visualization libraries it is also possible to render point clouds as well as changing visual properties.

CHAPTER 3

Approach

The goal of this work is the seamless texturing of an arbitrary three-dimensional object with semi-random texture patches. Semi-random means that the whole surface gets textured by choosing from a fixed set of texture patches at random. Figure 3.1 shows two possible results side by side.

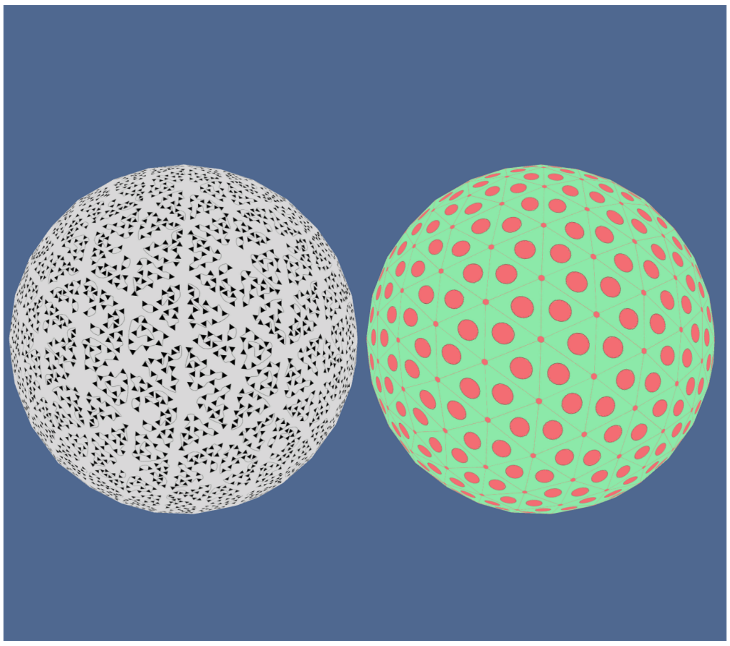


Figure 3.1: Two objects textured semi-randomly. The object on the left had its texture patches chosen at random from four different input patches. The other object only has one patch applied to its surface. Notice the visible triangulation.

To achieve the stated goal, a physics-based approach has been adopted, exchanging the input mesh with a newly generated output mesh. The input mesh is called the geometry mesh (since it retains all the information about the object's shape and position) and the output mesh is called the texture mesh, as the texture will be applied to it (Figure 3.2).

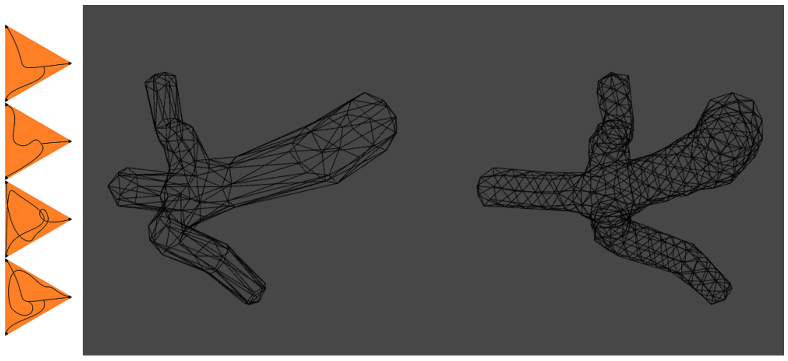


Figure 3.2: From left to right: Triangle patches, texture mesh, geometry mesh.

The texture meshes' main characteristic will be the fact that it is made up of equilateral triangles, serving as trivial mapping points for the equally triangular texture patches. Turk shows that this can be done without losing information about the surfaces shape [Tur92]. Choosing to create a second mesh instead of repositioning the vertices of the input mesh has the advantage that the resolution of the texture patches can be taken into account, resulting in a more fine or coarse texturing (Figure 3.3).

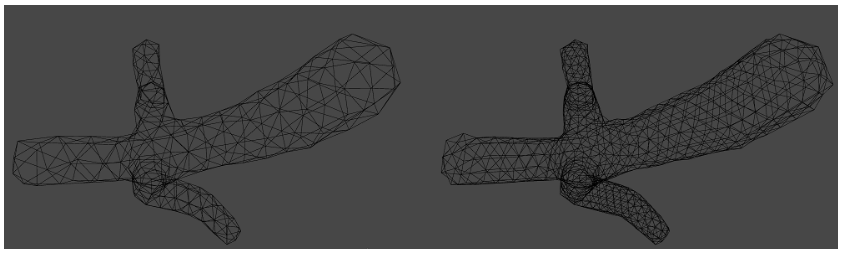


Figure 3.3: Examples for a coarse (left) and fine (right) texture mesh.

To set the vertex positions for the texture mesh, a simple system of repulsive forces is implemented. It pushes a user specified number of new vertices across the surface of the geometry mesh until they are regularly distributed, thus forming a mesh of equilateral triangles. The number of new vertices at the same time determines the resolution of the texture mesh. After their positions are calculated, these points are triangulated, resulting in the texture mesh. Here, every triangle corresponds to a single texture patch, chosen at random from the set of input patches. To recapitulate: A texture mesh, consisting of equilateral triangles and reproducing the geometry meshes' surface, is created using repulsive physics forces on a user specified number of newly introduced texture vertices. Due to its regularity, the vertices can easily be mapped to texture space, resulting in a fully textured and trivially seamless surface.

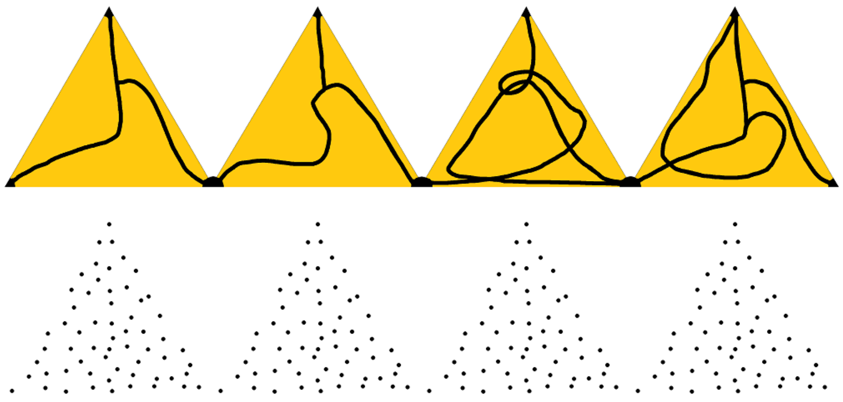


Figure 3.4: Examples for different input textures.

The algorithm is divided into three phases and a setup step. Figure 3.5 shows a representation of the pipeline. During setup, data from the input mesh is extracted and preliminary parameters are calculated according to user input. In Phase 1, new vertices are created and—using a simple statistical analysis of triangle size—positioned on the geometry mesh. They are subsequently moved by repulsion. During the second phase, the texture vertices are triangulated, creating the texture mesh. Lastly, in the third and final phase, texture coordinates are assigned and texture patches are applied onto the surface.

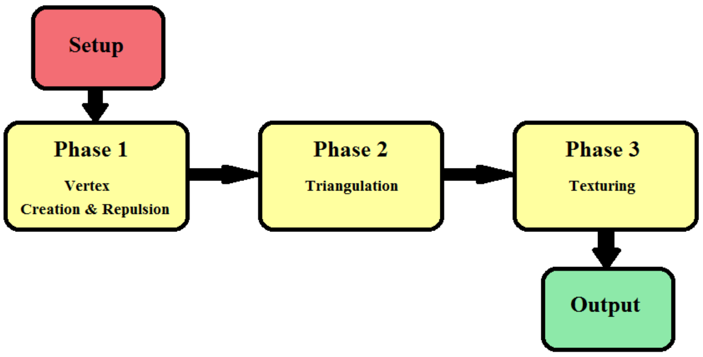
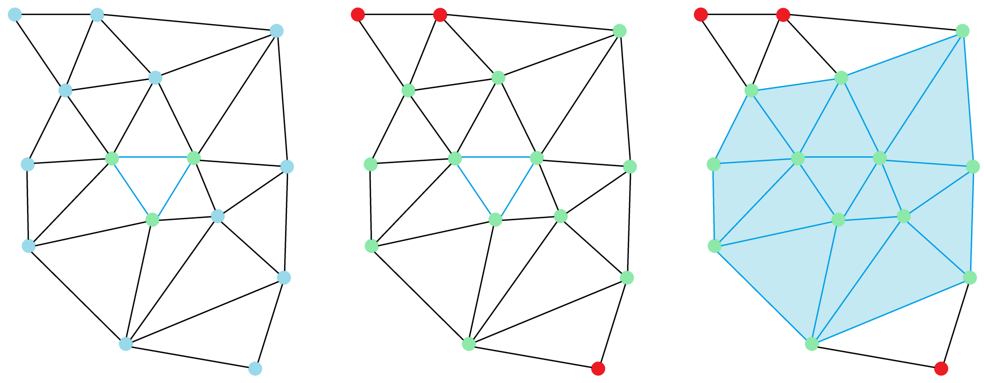


Figure 3.5: The texturing pipeline.

3.1 Setup

During setup, the radius of influence for each vertex is calculated as well as data from the input mesh is extracted. This means two lists, one for triangles and one for vertices, are filled. The radius of influence determines the distance inside which each vertex exerts repulsing force onto other vertices. Additionally, triangle area and neighbourhood are calculated. A triangle's area is needed to place texture vertices in phase one, its neighbourhood during vertex repositioning to find the origin of repulsing forces. To obtain the triangle neighbourhood, a greedy algorithm is employed: for every one of the three triangle vertices the distance to each of their vertex neighbours is measured. If this distance is smaller or equal to the radius of influence times two, the vertex is added to the neighbourhood. This specific value is chosen because it creates a sufficiently large neighbourhood, making sure that potential neighbours are not missed at the cost of memory usage. This is repeated for the second-level neighbours as well, until no more additions to the list can be found. Then for every vertex in this list, each triangle that contains it, is added to the triangle neighbourhood.



(a) A triangle is selected from the list of triangles.

(b) Every vertex within range is determined.

(c) Triangles are added to the neighbourhood.

Figure 3.6: The three steps of creating a triangle's neighbourhood.

3.2 Phase One

In phase one, the vertices that will later make up the texture mesh are introduced into the geometry mesh. This is done by assigning a position inside one of the geometry meshes' triangles to each new vertex. This triangle is called the vertex' parent triangle. It is determined at random for each newly created vertex individually by choosing from available triangles that make up the geometry mesh. To account for meshes with non-uniform sized triangles, the chance to become a vertex parent is directly proportional to triangle area. This also acts as an optimization step towards achieving a uniform distribution of texture vertices. Important to note is that these new points are not connected to each other or any triangle other than by data reference. At this point they act as simple particles for the following physics calculations.

As mentioned, once the vertex creation is finished, the vertex repositioning starts. This serves the purpose of uniformly distributing the texture vertices on the surface of the geometry mesh. Throughout these calculations the geometry mesh only acts as a frame of reference; Its vertices are not considered in the described repulsion process, only the newly introduced points are of interest.

In each loop of code for each texture vertex a check for neighbouring texture vertices is performed. This adjacency is dependent on the radius of influence r that was calculated during setup. Once all the neighbours of one vertex are found, repulsion is simulated by simply adding the vector between the point and a neighbour to the current position $\mathbf{pos_c}$ of the vertex with a negative sign after normalizing it. Then this direction $\mathbf{d}$ is scaled by distance: The farther away a neighbouring point $\mathbf{pos_o}$ is, the weaker the repulsion. Additional scaling is done by multiplying with a fixed force constant f with a value of 0.5. This generally damps a vertex' movement and avoids erratic behaviour. The value of 0.5 was adjusted by experiment.

$$\mathbf{pos_{new}} = \mathbf{pos_c} + (r - ||\mathbf{pos_c} - \mathbf{pos_o}||) * f * \mathbf{d} \tag{3.1}$$

Since the vertices inside the neighbourhood cannot, by definition, be farther away than r, the term $(r - ||\mathbf{pos_c} - \mathbf{pos_o}||)$ has a range of $[0, r]$.

Performing this change of position for each neighbour, a new position is calculated for each texture vertex. Since this position does not have to lie inside a triangle of the geometry mesh, as a final step the vertex is snapped back onto the geometry's surface. First, the closest triangle is determined. Then, the closest point inside this triangle is calculated and the vertex' final position is set to that point.

Turk uses a value of $k = 40$ for determining how often this process is repeated [Tur91]. Through testing, we found this value to produce satisfying results and therefore have adopted it.

During development of this phase a number of alternative approaches were investigated and discarded. An early concept made use of rigid body physics. Simulating points as rigid bodies would allow each vertex to interact with other vertices as described above. But the caveat is that this functionality is not optimized for large amounts of rigid bodies, resulting in notable performance loss. When changing the vertices' position it can happen that they are pushed off a triangle edge. In this case a mechanism is needed to correctly reposition them on the surface again. To this end, a "gravitational" constant g was initially introduced. This constant would add a force towards the surface to the new position of each vertex.

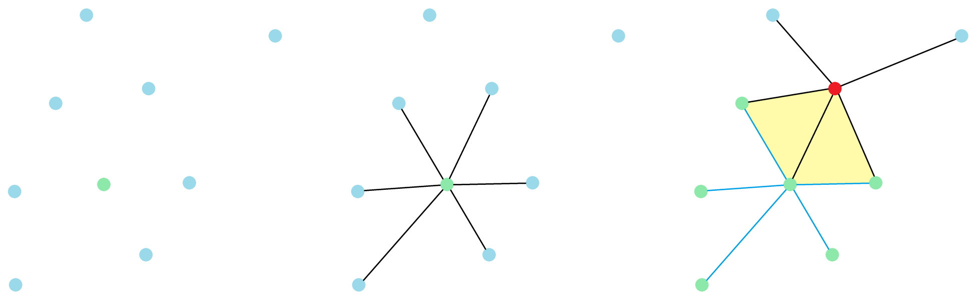
$$\mathbf{pos_{final}} = \mathbf{pos_{new}} + g \tag{3.2}$$

In certain configurations, however, the vertices would hover above the surface. This was due to strong repulsive forces from neighbouring vertices already positioned directly below them. In the final version, the idea of adding such a constant to the repulsion process was switched out for the more direct snapping to the surface. This was done to ensure that the texture vertices stay true to the original surfaces' form.

The whole algorithm depends on distance calculations between vertices. The used measure is Euclidean distance, simply calculating a straight line in three dimensional space between two points and returning this line's length. But an alternative exists in geodesic distance. Here, the distance between two points is the length of the shortest path on a surface they are both part of. For example, the distance between two cities on a road map is actually geodesic, since both cities are part of the earth's surface. Using this measure, vertices could be positioned more precisely. But, finally, the notion of geodesic distance calculation got cut from the final algorithm. Although the overall precision of the repositioning step would have been increased, the tested implementations (one an exact calculation, the other an approximation) either introduced too many numerical errors or slowed down the algorithm too much.

3.3 Phase Two

Phase two is closely connected to phase one. As explained above, during phase one the texture vertices are repositioned, forming equilateral triangles. In phase two they are triangulated, thus creating the texture mesh that is textured in the final phase. The goal of this phase is to create a new mesh consisting of equilateral triangles that represents the original geometry. Triangulation itself is done using a straightforward greedy algorithm. It assumes that each vertex' neighbourhood only contains vertices that are to be connected by an edge in the texture mesh. To ensure this condition is met, each vertex neighbourhood has to be checked for integrity. This is done by constructing hypothetical edges between each vertex and its neighbours, creating a star-like structure.

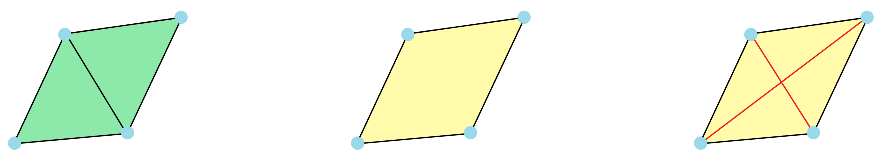


(a) A single vertex is selected from the list of vertices. (b) Hypothetical edges are constructed. (c) Each neighbour's neighbours are examined.

Figure 3.7: A vertex' neighbourhood is checked for irregularities.

By checking for each neighbour's neighbours, three cases can be observed for the space between edges:

Case 1: The connected vertices form a triangle (Figure 3.8a). This is the desired case and does not require further handling.
Case 2: The connected vertices form a square (Figure 3.8b). In this case all four of the concerned vertices have their individual radii of influence slightly increased, resulting in the potential inclusion of another vertex in the neighbourhood, creating a diagonal for the square.
Case 3: The connected vertices form a square with intersecting diagonals (Figure 3.8c). This case is resolved by slightly decreasing the radius of influence for each involved vertex, thus removing a connection from the neighbourhood.



(a) An example for case 1. (b) An example for case 2. (c) An example for case 3.

Figure 3.8: The three cases of edge neighbourhood.

The adjustment of radii is done until equilibrium is achieved. Then the triangulation is initialized: Starting with the first texture vertex in the list, its neighbourhood is converted into triangles. Depending on user input, each vertex is assigned a random color or a random texture coordinate. Once the triangulation and conversion step is done, the final data structures consist of an array of vertex positions as well as color or texture coordinates, an array of normals and an array of vertex indices representing triangles.

As with phase one, for phase two several alternative approaches have been investigated. A first concept actually combined the two phases, starting with a seed vertex that would spawn a set of neighbours already connected to each other. The aforementioned physics simulations were then used to determine the final position before spawning the next set of points. The problem with this approach was handling the last few triangles when the object was almost completely covered, oftentimes resulting in heavy distortions. The reason for these distortions was the different number of repulsing neighbours for each vertex. While some vertices would only be repulsed by one or two neighbours, others would accumulate stronger forces because of a neighbourhood size of four or five. These differences in spacing resulted in irregular triangles.

Neyret et al. describe a technique called "Mutual Tessellation" [NC99]. It works by first triangulating the old geometry mesh by iteratively including the new texture vertices. Subsequently, all geometry vertices are removed from this mutual tessellation, resulting in a mesh consisting only of texture vertices. The problem hereby is, that the part of their algorithm that creates a new triangulation for the holes that remain once a geometry vertex is removed is not trivial. It would have been beyond the scope of this work to include all the possible cases and test for correctness.

3.4 Phase 3

During this final phase either the user-provided texture patches are applied to the mesh or the vertices are coloured. Colouring is straightforward: Each vertex is assigned a random number between one and three. This number corresponds with red, green or blue respectively. If texture is to be applied, the user has to provide texture patches. Since every vertex in the texture mesh corresponds to one corner of a texture patch, assigning UV-coordinates is the same as assigning colours. The only addition is that on top of a random texture coordinate, the specific patch is also chosen at random from the four patches provided.

This means that the texture patches, aside from them being equilateral, also have to have corresponding edges no matter the configuration. This means that each possible pairing of touching edges between neighbouring texture patches has to be seamless. If the desired texture effect calls for a more complex pattern the algorithm has to be changed accordingly, adding functionality to control texture assignment instead of random chance.

Since the mapping to equilateral triangles is now easily done, a number of additions is possible. Having a more complex texture by creating dependencies between patches was already mentioned above. In addition to that it is now feasible to replace the texture by 3D-objects, thus creating a more detailed surface that can then in turn be textured. This corresponds with our initial goal of visualizing a phospholipid membrane. Instead of an image texture, we can place the respective phospholipid geometry on an object's surface. This geometry has to cover an equilateral triangle so seamless coverage of the object is possible while maintaining a small memory footprint. Integrating presented texturing concept into molecular visualization is the scope of our future work.

Figure 3.9 shows a sphere that was textured with four texture patches, distributed at random. Close examination reveals the corners of the triangles that make up the texture mesh since they were used as simple points of connectivity to create the texture. More importantly, though, is the fact that no neighbouring regions look exactly the same.

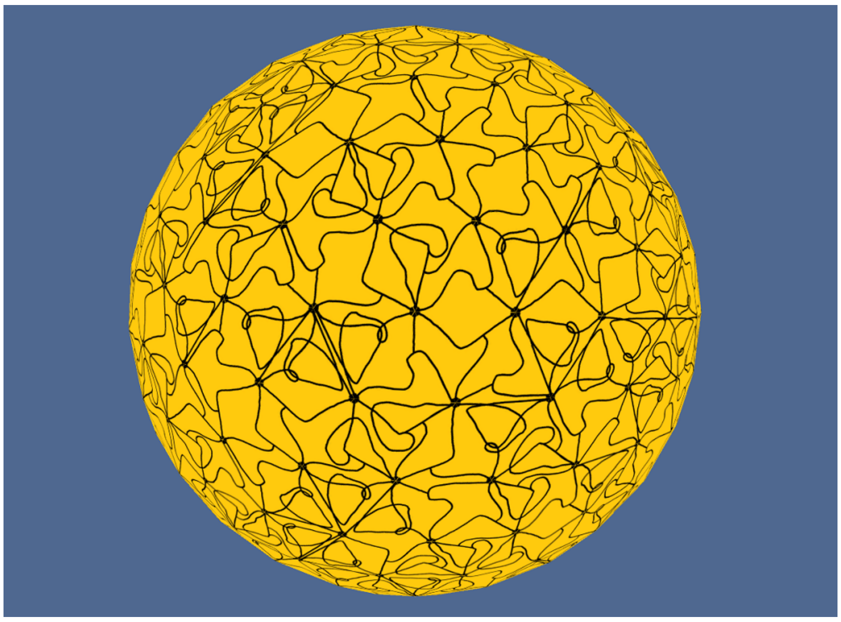


Figure 3.9: Example of a completely textured sphere. Four texture patches were used to create this surface.

CHAPTER 4

Implementation

The tool used for visualizing meshes and textures is the Unity 3D-engine. It also provides a framework for algorithms to be realized as scripts. The scripting language is C#.

4.1 Data structures and algorithms

To aid in the texturing process and to compensate for some of Unity's shortcomings on this sector, a few classes representing specific data structures are introduced. Although the Unity engine provides sufficient support for linear algebra, the meshes are stored with a face-based data scheme. In concrete terms, every mesh consists of a list of vertices with corresponding normals, texture coordinates and colors, each stored in a separate list of equal cardinality. The mapping between these lists is bijective. Connectivity information comes from another list containing vertex list indices: three subsequent entries link to the members of one triangle.

This results in some problems. Since entries in the triangle list are unordered, neither triangle nor vertex neighbours can be extracted. Also, since normal vectors are stored on a per-vertex basis, two neighbouring triangles (i.e. triangles sharing two vertices) with different surface normals require separate entries for the same vertex in the vertex list. Although these have the same position, their normals are different, making it rather obtuse how many vertices actually make up the mesh.

As a remedy, three classes are used.
1. Vertex: This class saves a position and all the vertex list indices of the geometry mesh with that same position, this way making normal, uv and color information obtainable by looking them up in the respective list. Additionally, neighbouring vertices and triangles are referenced. The class methods are used to store vectors of influencing forces and to reposition the vertex according to these forces. Parameters for repositioning are also set via variables: these include force, drag and repulsion radius. In the case of a newly

introduced vertex the triangle containing it called the parent triangle is also saved.
2. Edge: This basically acts as a helper class representing an edge; it references the two vertices the edge connects.
3. Triangle: This class saves the three vertices making up a triangle as well as the surface normal. Moreover, the triangle area and the index of the triangle in the original list is stored as well as the texture vertices contained within the triangle. This—combined with a vertex- as well as a triangle neighbourhood—is done to facilitate neighbourhood-searches. These neighbourhoods are represented as lists. Methods exist to find the opposing edge of an input vertex as well as a comparator to make sorting by area possible.

Algorithm 4.1: Vertex neighbourhood creation

```
input  : Vertex $v$, Float $radius$
output : List $l$ of neighbouring vertices
1  Queue $q$ = new Queue;
2  $q \leftarrow v$;
3  while $q \neg empty$ do
4      Vertex $current \leftarrow q$.pop;
5      for $n \in current.neighbours$ do // 1-ring neighbours are known
6          if $||n - v|| < radius$ then
7              $l \leftarrow n$;
8              $q \leftarrow n$;
9          end
10     end
11 end
```

Algorithm 4.2: Triangle neighbourhood creation

```
input  : List $lVert$ of neighbouring vertices
output : List $lTri$ of neighbouring triangles
1  for $n \in lVert$ do
2      for $Triangle\ t \in n.triangles$ do // Triangles containing a vertex
           are known
3          $lTir \leftarrow t$;
4      end
5  end
```

The input consists of a triangulated mesh, the desired number of texture vertices, a set of texture patches and two shaders. Since our technology should be well suited for structural biological models, some constraints apply: The mesh is expected to be without holes or irregularities. Edges cannot be shared by more than two or less than one triangle. Although not being a hard constraint, it is advisable for the input mesh not to have too many sharp angles or corners as they can result in minor distortions. Furthermore, corners generally do not correspond to the molecular context. The number of texture vertices directly controls the coarseness of the texture mesh; choosing too few will also result in distortions. Since the notion of "too few texture vertices" greatly depends on the input mesh, no reliable heuristic can be given, thus leaving this control in the users hands. The texture has to contain four equilateral triangles arranged horizontally. The shaders are optional input and only necessary if instead of texture the user wants color applied to the mesh.

Repositioning and triangulation of vertices is implemented as follows:

Algorithm 4.3: Vertex repositioning

```
input  : Vertex $v$
Result: Repositioning of $v$
for $n \in v.neighbours$ do
    $v$.position $\leftarrow$ $v$.position $+(radius - n$.position$) * force * n$.direction;
end
```

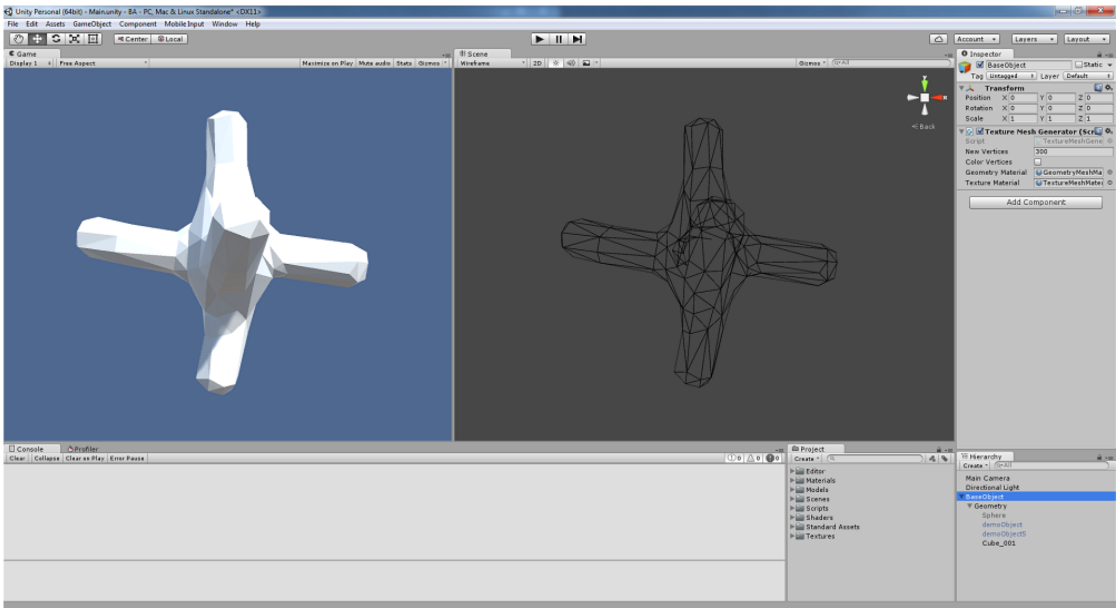
Algorithm 4.4: Triangulation

```
input  : List $verts$ of texture vertices
Result: Triangulation of $verts$
for $v \in verts$ do
    for $n \in v.neighbours$ do
        for $m \in v.neighbours$ do
            if $n \in m.neighbours$ then
                Add new Triangle$(v, n, m)$ to Triangulation;
            end
        end
    end
end
```

4.2 User interface

This section describes the user interface and how to operate it. A basic understanding of Unity's functionality is assumed. The main scene consists of three GameObjects: "Main Camera", "Directional Light" and "BaseObject". Those can be seen in Figure 4.1 in the hierarchy-window in the bottom right corner.

The camera and the light are there to provide basic lighting and the possibility to view the textured object once it is finished. The "BaseObject" has the script containing the functions for texturing attached to it and a child object, "Geometry". This "Geometry"-



BaseObject Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Texture Mesh Generator (Scr
Script TextureMeshGene
New Vertices 300
Color Vertices ☐
Geometry Material GeometryMeshMa
Texture Material TextureMeshMate

Main Camera
Directional Light
BaseObject
Geometry
Sphere
demoObject
demoObject5
Cube_001

Figure 4.1: The main Unity window. On the left side the textured object is displayed once the calculations are finished. The center shows the geometry mesh. On the right are the input-fields for the script and in the bottom right is the hierarchy-window, where new objects have to be placed.

object in turn should be the parent of the mesh that is to be textured. This mesh is user-provided. It is activated by dragging it onto the "Geometry"-object.

The script attached to "BaseObject" handles the rest of the necessary user input: The variable "New Vertices" expects an integer value, the checkbox "Color Vertices" is used to switch between random colouring and texturing and the two materials "Geometry Material" and "Texture Material" need to be set to provide shader functionality (since in Unity, shaders are attached to materials). Figure 4.1 shows a possible configuration: 300 new texture vertices will be created and the "Color Vertices"-option is switched off. Both materials have been set. By clicking "Play" (at the top of the screen), the calculations are started.

All of the above-mentioned inputs underlie certain constraints. As mentioned before the algorithm is developed with cell membranes as the main visualization goal in mind. Consequently, if provided a sharp-angled object as input, visual distortions are likely to occur. Since the number of new vertices directly corresponds to texture mesh resolution, it is advised to choose this quantity carefully. Also, if less than three vertices are created, triangulation cannot be completed. A value that worked well during development was the number of vertices in the geometry mesh. Starting with this amount of new vertices the user can then de- or increase that number according to the output.

CHAPTER 5

Demonstration

This chapter will contain examples of possible output. Special care has to be taken when choosing shaders: If the texture mesh is to be coloured at random (by checking "Color Vertices"), an appropriate shader has to be attached to the "Texture Material". This is mentioned because none of Unity's built-in shaders provide vertex color support. Selecting a shader is done via the drop-down menu of a material (Figure 5.1).

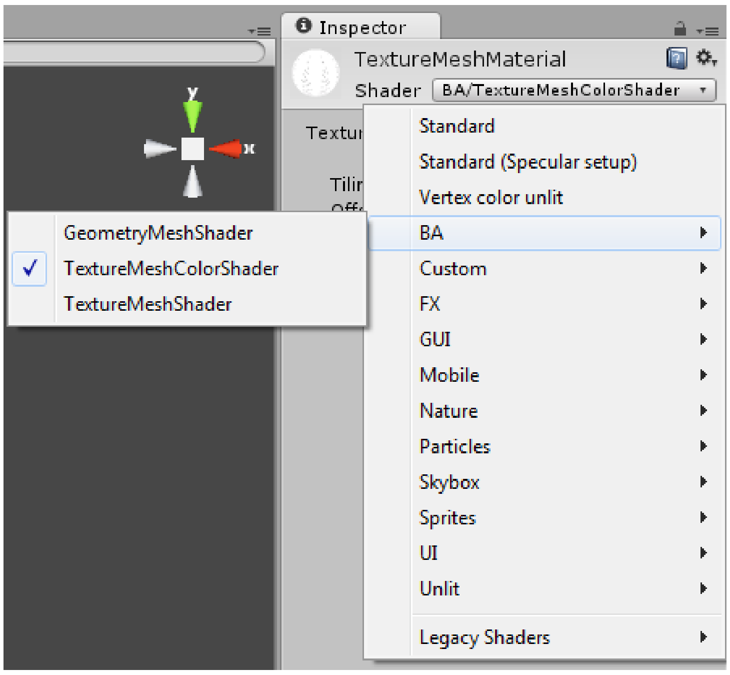
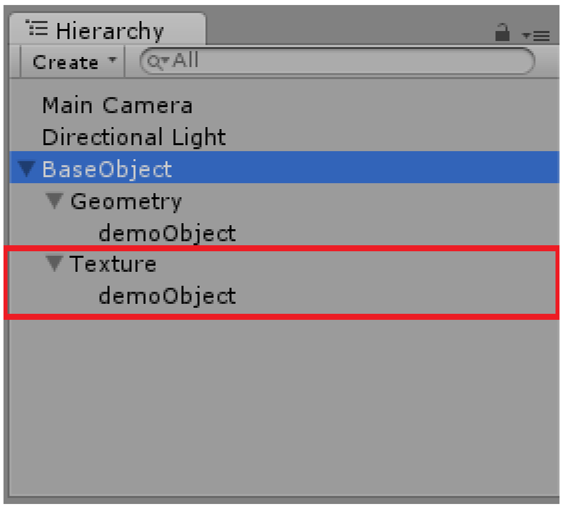


Figure 5.1: Different shaders can be selected by using the drop-down-menu of an assigned material.

Once input is taken care of, the script can be started by pressing "Play". The "Game" view will then show the texture mesh. In this view, camera controls work by holding down the left mousebutton and moving the mouse. The scrollwheel can be used to zoom in and out. For additional manipulations, the texture mesh can also be found in the "Hierarchy"-window as a child of the newly created "Texture" GameObject (Figure 5.2).



Main Camera
Directional Light
BaseObject
Geometry
demoObject
Texture
demoObject

Figure 5.2: Once it is created, the texture mesh can be selected as a child of BaseObject.

Figure 5.3 shows a basic example for a coloured texture mesh. The colour values are chosen at random from either red, green or blue.

A more complex example can be seen in Figure 5.4: Here, four texture patches are applied to the mesh. Because of the patches' motifs, no triangle edges are visible. This is achieved by avoiding placement of foreground-objects (in this example: islands) near the edges.

In Figure 5.5, the randomness of the distribution of texture patches can be observed. Each patch has one to four dots on a white background, similar to a die. The figure shows which patch number is assigned to different parts of the object's surface.

Finally, Figure 5.6 shows a comparison between a coarse and a fine texture mesh.

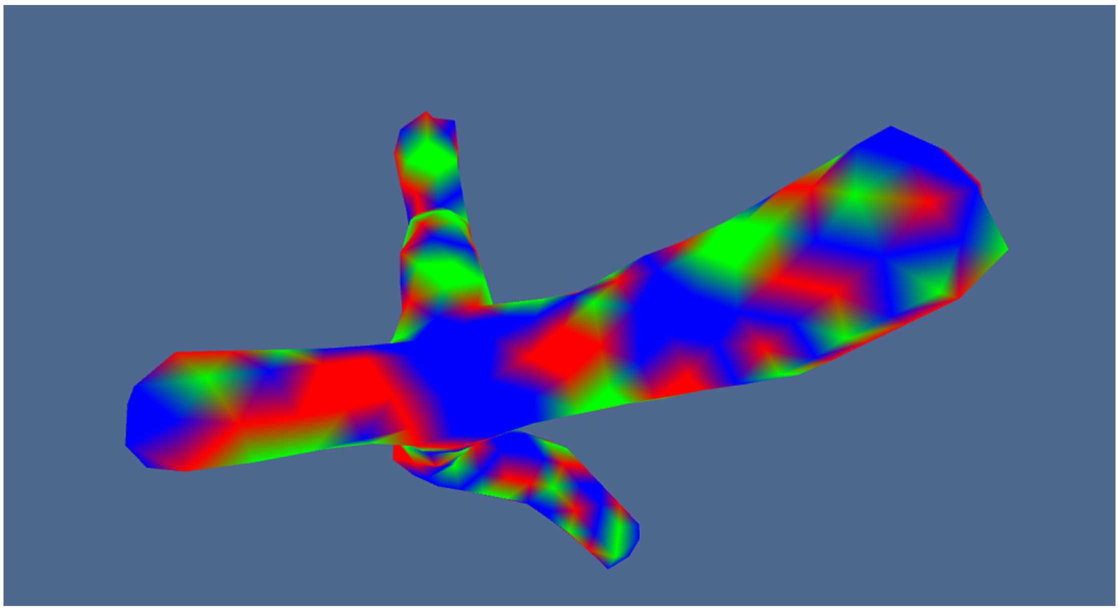


Figure 5.3: An object with randomly assigned colours for each vertex. Values are interpolated in the fragment shader.

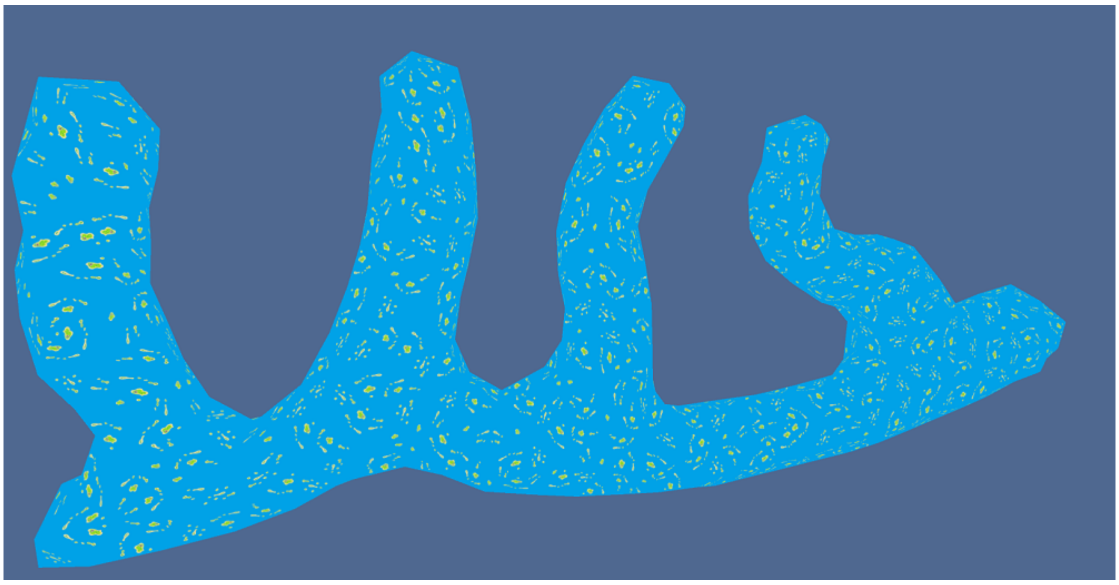


Figure 5.4: An object with texture applied at random, chosen from four input patches. No triangle edges are visible.

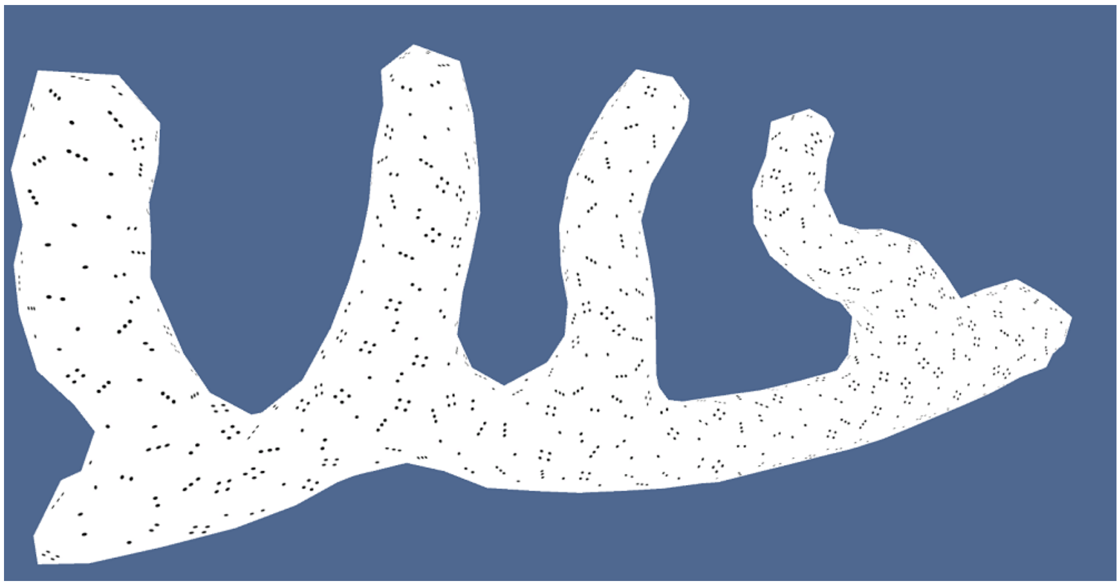


Figure 5.5: An object with texture applied at random, chosen from four input patches. The texture either contains one, two, three or four dots. This is an example of how the texture patches are distributed at random.

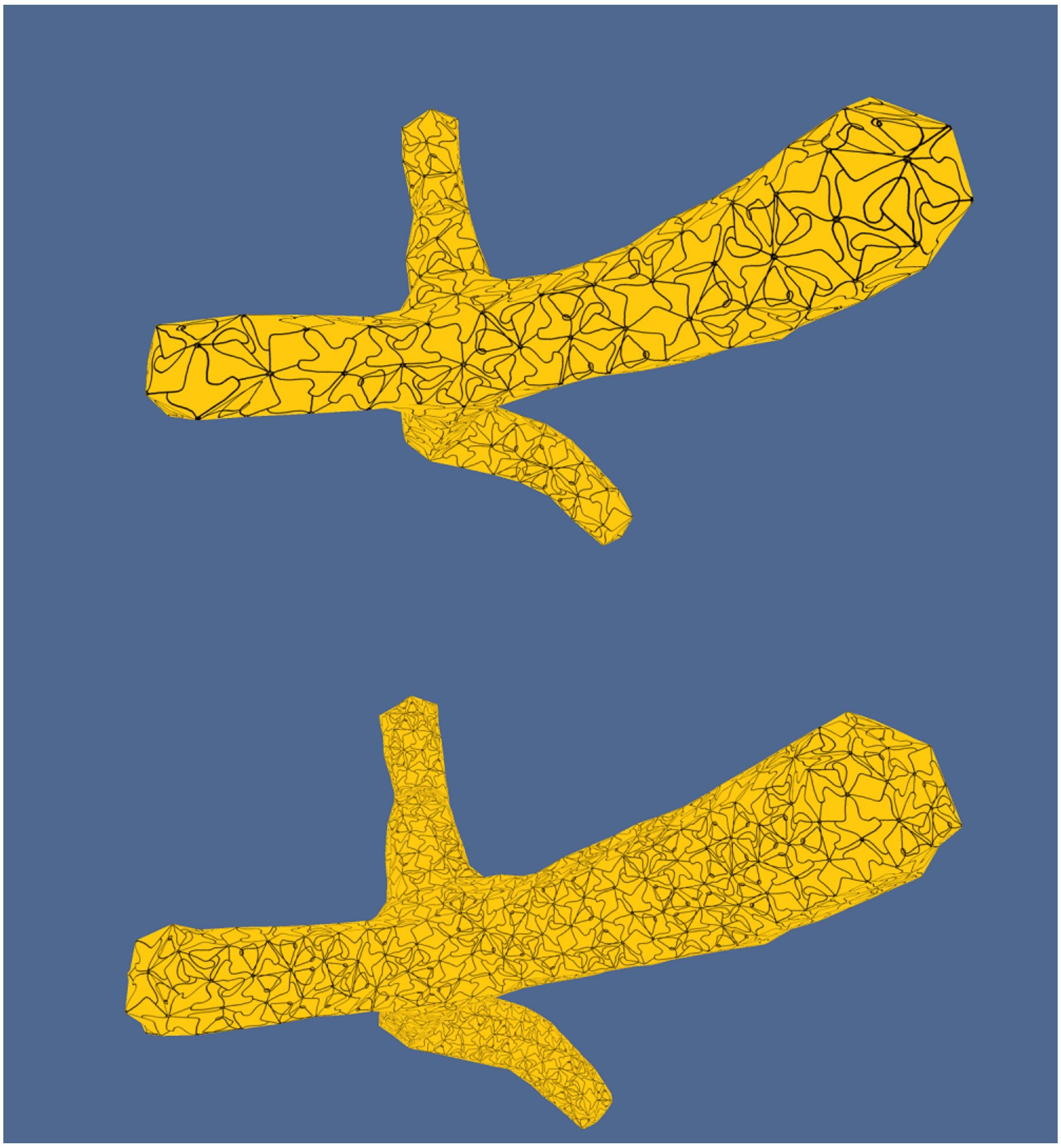


Figure 5.6: The same object with different texture mesh densities. On the top is a coarse texture mesh, on the bottom a fine texture mesh.

CHAPTER 6

Evaluation

Demonstrative cases from the previous chapter are now benchmarked for performance and their visual quality is discussed. Analysing the code, $O(n^2)$ presents itself as an expected runtime, depending on the number of new texture vertices. This is mainly due to the fact that neighbour-comparison, as mentioned earlier, has to check each possible neighbour for every vertex. But, as can be seen in the following benchmark results, the actual runtime is worse than that. The reason for this is the way triangulation is implemented. Sometimes an unstable equilibrium is created in a situation where a vertex either has an empty or a filled square as one neighbour. By changing the radius, the square switches from one false state to the other without reaching correct triangulation until the resulting physics rearrange the whole vertex neighbourhood, which can take rather long. An easy, albeit potentially imprecise solution to this problem would be to stop rebalancing after a set amount of loops and just create or remove one arbitrary diagonal for each detected hole. A more thorough approach calls for a restructured triangulation procedure that creates more reliable output.

The following tables are to showcase input variations. Four objects, shown in figure 6.1, were textured as well as coloured on three different computers, using different texture mesh densities. The machines have these specifications:

PC 1:
-Intel Core i5-6600K 3,50 GHz
-16 GB DDR4 RAM
-AMD Radeon HD 5800
-Win7 64 Bit

PC 2:
-Intel Core i5-4200U 1,60 GHz
-4 GB DDR3 RAM
-Intel HD Graphics 4400 onboard
-Win7 64 Bit

PC 3:
-Intel Core 2 Duo 2,33 GHz
-2 GB DDR3 RAM
-Nvidia GeForce 8800 GT
-Win7 64 Bit

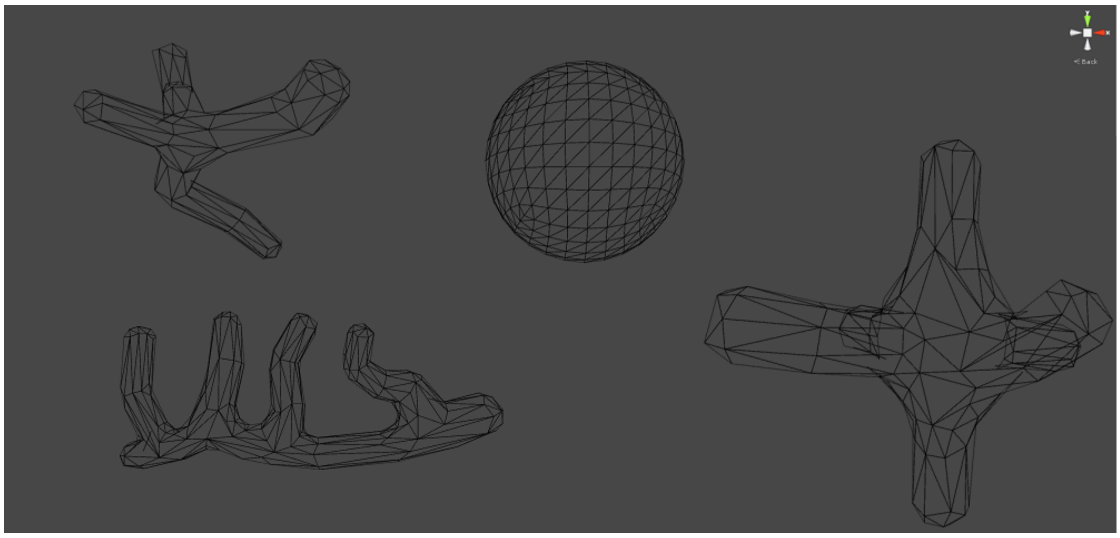


Figure 6.1: Sphere and objects 1 (top left), 2 (bottom left) and 3 (bottom right).

Each table shows benchmark results for RAM (divided into total memory used, memory reserved for textures and memory reserved for meshes), VRAM and execution time. The time-column is further divided into time needed for vertex repositioning, triangulating the texture mesh and total runtime. These values are listed along the x-axis. Along the y-axis, the three test-systems are listed, with different rows for coloured or textured output and vertex numbers. As expected, total memory usage corresponds to the number of texture vertices since repositioning takes a lot of RAM. In the case of PC2 and PC3, some entries in the memory column are almost identically high regardless of vertex density. The reason for this is improper flushing of RAM after code execution. For example, in Table 6.3, a coloured mesh with 250 vertices requires the same amount of memory as the textured mesh with 800 vertices in Table 6.2 for PC2.

Texture and Mesh memory stay relatively low regardless of texture vertices. This is a desired result, for it means that once the texture mesh is computed, it can be used for display with a small memory footprint. The same goes for VRAM which also stays constantly low.

As far as runtime is concerned, the time it takes to reposition the texture vertices shows the expected behaviour: The more vertices, the longer it takes to reposition them. But triangulation unfortunately shows erratic behaviour. In Table 6.4 for example, a texture mesh with 400 vertices takes longer to triangulate than a texture mesh with 800 vertices (rows 11 & 12). As already mentioned, this is due to the way triangulation is implemented.

			RAM			VRAM	Execution Time		
			Total	Texture	Mesh		Repositioning	Triangulation	Total
PC 1	coloured	250 Vertices	189,7 MB	13 MB	2,6 MB	30,7 MB	1683,505 ms	2487,901 ms	2493,373 ms
		1000 Vertices	0,52 GB	13 MB	2,6 MB	30,7 MB	5445,339 ms	11127,24 ms	11139,89 ms
		5000 Vertices	1,11 GB	13 MB	3,1 MB	38,5 MB	59505,01 ms	574925,6 ms	574987,9 ms
	textured	250 Vertices	0,65 GB	13,2 MB	2,6 MB	31,3 MB	1573,705 ms	2355,94 ms	2361,679 ms
		1000 Vertices	0,71 GB	13,2 MB	2,6 MB	31,3 MB	5768,306 ms	11404,41 ms	11423,04 ms
		5000 Vertices	1,11 GB	13,2 MB	2,6 MB	31,3 MB	57207,27 ms	570355,01 ms	570429,8 ms
PC 2	coloured	250 Vertices	121,1 MB	9,1 MB	3,2 MB	32,3 MB	3183,376 ms	5509,886 ms	5526,309 ms
		1000 Vertices	309,6 MB	9,1 MB	3,2 MB	32,3 MB	11803,36 ms	45364,07 ms	45403,78 ms
		5000 Vertices	0,63 GB	9,1 MB	3,2 MB	32,3 MB	115859,3 ms	1564378 ms	1564514 ms
	textured	250 Vertices	0,7 GB	16,1 MB	3,2 MB	33,0 MB	2916,545 ms	5051,496 ms	5064,292 ms
		1000 Vertices	0,69 GB	16,1 MB	3,2 MB	33,0 MB	16151,66 ms	57378,82 ms	57425,89 ms
		5000 Vertices	0,68 GB	10,4 MB	3,2 MB	33,0 MB	151260,5 ms	1754535 ms	1754711 ms
PC 3	coloured	250 Vertices	133,6 MB	10,5 MB	3,3 MB	22,0 MB	4928,497 ms	7390,876 ms	7401,269 ms
		1000 Vertices	311,0 MB	10,5 MB	3,3 MB	22,0 MB	13605,91 ms	50964,46 ms	50998,43 ms
		5000 Vertices	0,61 GB	10,5 MB	3,3 MB	22,0 MB	127161,2 ms	1826559 ms	1827006 ms
	textured	250 Vertices	0,62 GB	10,5 MB	3,3 MB	22,7 MB	3745,882 ms	6051,249 ms	6062,324 ms
		1000 Vertices	0,68 GB	10,5 MB	3,3 MB	22,7 MB	13407,05 ms	50935,86 ms	50972,65 ms
		5000 Vertices	0,61 GB	10,5 MB	3,3 MB	22,7 MB	126308,6 ms	1846259 ms	1846702 ms

Table 6.1: Benchmark results: Sphere

			RAM			VRAM	Execution Time		
			Total	Texture	Mesh		Repositioning	Triangulation	Total
PC 1	coloured	250 Vertices	0,65 GB	13,2 MB	2,6 MB	30,7 MB	1831,824 ms	71588,16 ms	71593,63 ms
		400 Vertices	262,2 MB	11,6 MB	2,6 MB	30,7 MB	2706,358 ms	26456,61 ms	26463,58 ms
		800 Vertices	311,4 MB	11,6 MB	2,6 MB	30,7 MB	6400,358 ms	61923,73 ms	61934,48 ms
	textured	250 Vertices	290,6 MB	13,0 MB	2,6 MB	31,3 MB	1788,846 ms	71545,41 ms	71551,85 ms
		400 Vertices	329,2 MB	13,0 MB	2,6 MB	31,3 MB	2751,559 ms	26347,67 ms	26355,36 ms
		800 Vertices	378,7 MB	13,0 MB	2,6 MB	31,3 MB	6313,446 ms	65208,42 ms	65220,35 ms
PC 2	coloured	250 Vertices	0,69 GB	16,1 MB	3,2 MB	32,3 MB	3330,815 ms	140587,8 ms	140599,8 ms
		400 Vertices	0,69 GB	16,1 MB	3,2 MB	32,3 MB	5430,884 ms	58543,9 ms	58559,99 ms
		800 Vertices	0,78 GB	16,1 MB	3,2 MB	32,3 MB	13193,18 ms	143408,5 ms	143426,8 ms
	textured	250 Vertices	0,75 GB	16,1 MB	3,2 MB	33,0 MB	3384,735 ms	150364,5 ms	150376,9 ms
		400 Vertices	0,69 GB	16,1 MB	3,2 MB	33,0 MB	5527,129 ms	53177,78 ms	53194,63 ms
		800 Vertices	0,69 GB	16,1 MB	3,3 MB	33,0 MB	13381,74 ms	135645,6 ms	135676,1 ms
PC 3	coloured	250 Vertices	0,66 GB	10,5 MB	3,3 MB	22,0 MB	4241,007 ms	136325,6 ms	136335,8 ms
		400 Vertices	0,61 GB	10,5 MB	3,3 MB	22,0 MB	6548,736 ms	56029,32 ms	56042,68 ms
		800 Vertices	0,61 GB	10,5 MB	3,3 MB	22,0 MB	14788,7 ms	139602,5 ms	139623,1 ms
	textured	250 Vertices	0,61 GB	10,5 MB	3,3 MB	22,7 MB	4228,289 ms	163946,8 ms	163957,8 ms
		400 Vertices	0,65 GB	10,5 MB	3,3 MB	22,7 MB	6895,375 ms	56541,52 ms	56555,82 ms
		800 Vertices	0,7 GB	10,5 MB	3,3 MB	22,7 MB	14844,23 ms	139335,7 ms	139359,1

Table 6.2: Benchmark results: Object 1

			RAM			VRAM	Execution Time		
			Total	Texture	Mesh		Repositioning	Triangulation	Total
PC 1	coloured	250 Vertices	297,3 MB	13,0 MB	2,6 MB	30,7 MB	4029,991 ms	18267,49 ms	18273,67 ms
		400 Vertices	319,0 MB	13,0 MB	2,6 MB	30,7 MB	4205,69 ms	16881,24 ms	16888,77 ms
		800 Vertices	354,1 MB	13,0 MB	2,6 MB	30,7 MB	5554,576 ms	41163,99 ms	41175,02 ms
	textured	250 Vertices	295,4 MB	13,0 MB	2,6 MB	31,3 MB	4002,161 ms	18221,09 ms	18227,43 ms
		400 Vertices	321,2 MB	13,0 MB	2,6 MB	31,3 MB	4088,836 ms	16816,96 ms	16824,83 ms
		800 Vertices	354,2 MB	13,0 MB	2,6 MB	31,3 MB	5495,5 ms	41125,15 ms	41137,05 ms
PC 2	coloured	250 Vertices	0,69 GB	16,1 MB	3,2 MB	32,3 MB	9813,864 ms	22956,93 ms	22969,34 ms
		400 Vertices	0,73 GB	16,1 MB	3,3 MB	32,3 MB	10232,3 ms	60504,98 ms	60516,87 ms
		800 Vertices	0,69 GB	16,1 MB	3,2 MB	32,3 MB	12707,5 ms	182208,4 ms	182234,6 ms
	textured	250 Vertices	0,72 GB	16,1 MB	3,2 MB	33,0 MB	9884,462 ms	21677,35 ms	21702,17 ms
		400 Vertices	0,69 GB	16,1 MB	3,2 MB	33,0 MB	10088,16 ms	54918,77 ms	54931,95 ms
		800 Vertices	0,69 GB	16,1 MB	3,2 MB	33,0 MB	13201,49 ms	139092,3 ms	139122,7 ms
PC 3	coloured	250 Vertices	0,61 GB	10,5 MB	3,3 MB	22,0 MB	11823,02 ms	25729,13 ms	25741,29 ms
		400 Vertices	0,61 GB	10,5 MB	3,3 MB	22,0 MB	12310,6 ms	65535,46 ms	65548,63 ms
		800 Vertices	0,61 GB	10,5 MB	3,3 MB	22,0 MB	15599,69 ms	151474,9 ms	151496 ms
	textured	250 Vertices	0,63 GB	10,5 MB	3,3 MB	22,7 MB	11863,59 ms	25695,69 ms	25721,99 ms
		400 Vertices	0,65 GB	10,5 MB	3,3 MB	22,7 MB	12333,46 ms	65319,29 ms	65347,97 ms
		800 Vertices	0,61 GB	10,5 MB	3,3 MB	22,7 MB	15676,98 ms	151730,6 ms	151753,8 ms

Table 6.3: Benchmark results: Object 2

But although, without optimization, the algorithm is not suited for big vertex numbers, oftentimes satisfying results can be achieved with vertex counts in the range between 10^2 and 10^3. In this range, texture mesh creation happens fast enough to be feasible in real time. However, if triangulation is regarded as a preprocessing step, higher vertex numbers are possible.

			RAM			VRAM	Execution Time		
			Total	Texture	Mesh		Repositioning	Triangulation	Total
PC 1	coloured	250 Vertices	439,6 MB	13,1 MB	2,6 MB	30,7 MB	22256,89 ms	28168,12 ms	28192,62 ms
		400 Vertices	368,4 MB	13,1 MB	2,6 MB	30,7 MB	22152,57 ms	36372,26 ms	36380,02 ms
		800 Vertices	357,8 MB	13,1 MB	2,6 MB	30,7 MB	21978,75 ms	48985,91 ms	48996,88 ms
	textured	250 Vertices	371,1 MB	13,1 MB	2,6 MB	31,3 MB	2236,17 ms	28215,7 ms	28222,21 ms
		400 Vertices	367,3 MB	13,1 MB	2,6 MB	31,3 MB	22960,54 ms	36231,82 ms	36239,78 ms
		800 Vertices	357,5 MB	13,1 MB	2,6 MB	31,3 MB	22079,1 ms	49056,01 ms	49067,82 ms
PC 2	coloured	250 Vertices	0,69 GB	16,1 MB	3,2 MB	32,3 MB	67437,02 ms	82084,99 ms	82098,55 ms
		400 Vertices	0,69 GB	16,1 MB	3,2 MB	32,3 MB	45401,6 ms	94068,98 ms	94087,04 ms
		800 Vertices	0,75 GB	16,1 MB	3,2 MB	32,3 MB	46161,54 ms	109780,6 ms	109810 ms
	textured	250 Vertices	0,7 GB	16,1 MB	3,2 MB	33,0 MB	47625,23 ms	66045,52 ms	66060,79 ms
		400 Vertices	0,77 GB	16,1 MB	3,2 MB	33,0 MB	48254 ms	90751,25 ms	90769,13 ms
		800 Vertices	0,76 GB	16,1 MB	3,2 MB	33,0 MB	47541,57 ms	86373,43 ms	86394,65 ms
PC 3	coloured	250 Vertices	0,67 GB	10,5 MB	3,3 MB	22,0 MB	80220,44 ms	95170,34 ms	95180,71 ms
		400 Vertices	0,61 GB	10,5 MB	3,3 MB	22,0 MB	58570,4 ms	98826,88 ms	98839,85 ms
		800 Vertices	0,61 GB	10,5 MB	3,3 MB	22,0 MB	58522,55 ms	105062,7 ms	105083,5 ms
	textured	250 Vertices	0,67 GB	10,5 MB	3,3 MB	22,7 MB	59161,63 ms	74207,41 ms	74218,48 ms
		400 Vertices	0,61 GB	10,5 MB	3,3 MB	22,7 MB	58781,5 ms	98903,3 ms	98952,61 ms
		800 Vertices	0,67 GB	10,5 MB	3,3 MB	22,7 MB	58657,45 ms	105108,2 ms	105131 ms

Table 6.4: Benchmark results: Object 3

Figure 6.2 shows a texture mesh with 200 vertices. As can be observed, the surface is well represented and shows no visible edges.

Figure 6.3 on the other hand is an example for an anomaly that can occur if the texture patches are not chosen carefully: Although the geometry is represented well by the texture mesh, the edges connecting the texture vertices are clearly visible. This is because the individual patches consist of a dense center, tracing the outline of the triangle, and a strip of empty background along each edge. As a result, no matter the configuration, whenever two edges meet, a thick white line is created.

Finally, figure 6.4 shows how a single patch with densely distributed points can sometimes achieve good results. The used mesh is the same as in figure 6.2. On the right is a magnification of the highlighted area: The structure of the texture mesh can be seen by carefully searching for concentric circles.

Surprisingly, even a single texture patch with randomly distributed points can be applied without noticeable artifacts. This increases the mapping possibilities as will be mentioned in the conclusion.

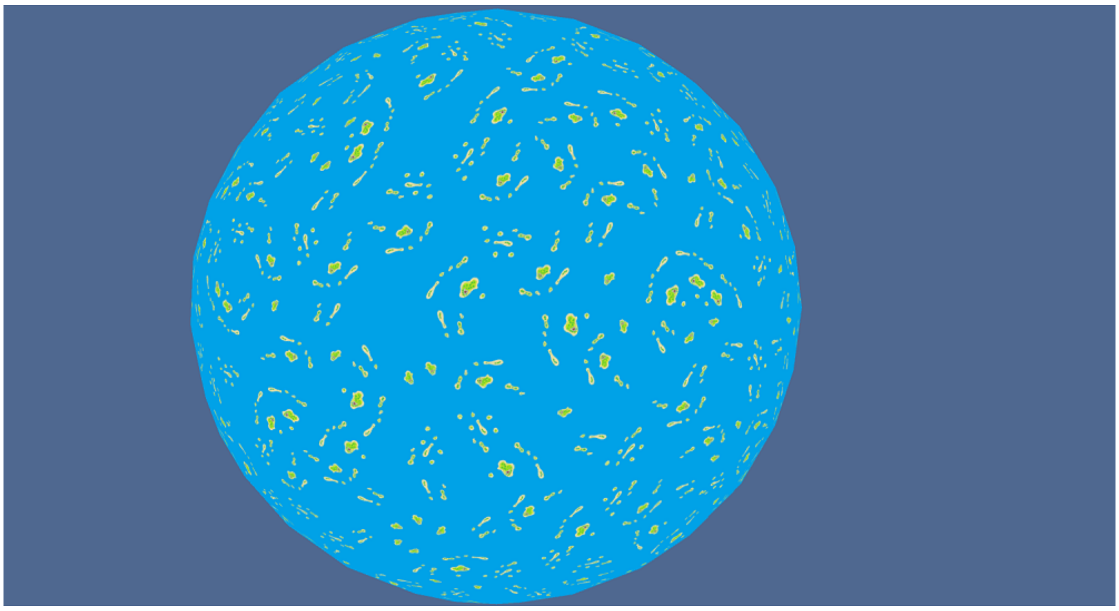


Figure 6.2: A sphere textured with four randomly assigned texture patches. Instances of mirroring can be detected but the overall distribution is without heavy repetition.

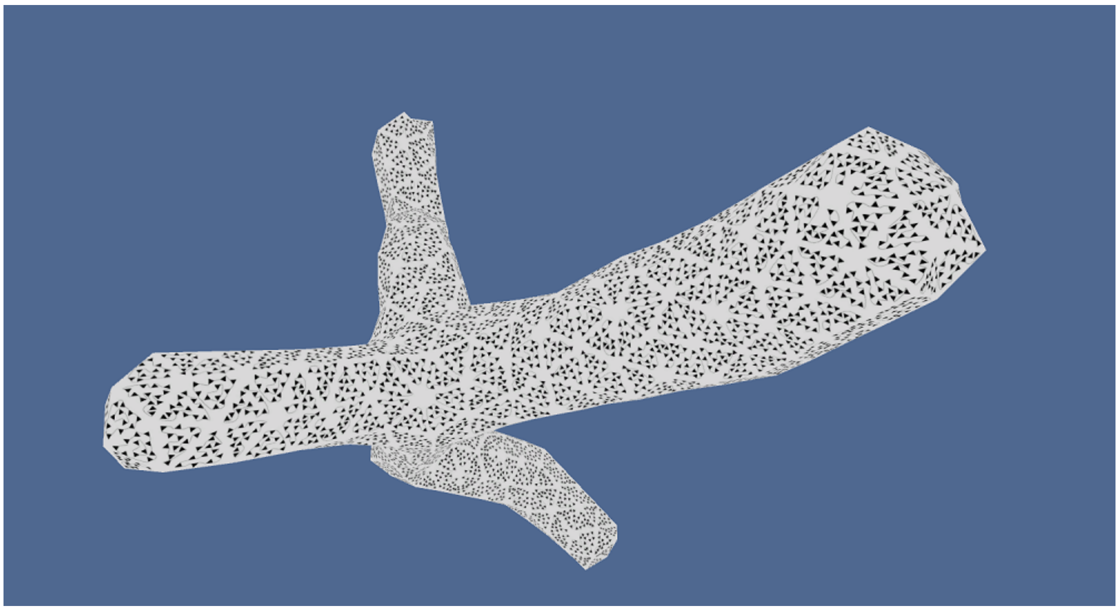


Figure 6.3: This object demonstrates a disadvantageous choice of texture patches: although four patches are used, their difference is not distinct enough. Furthermore, their visual structure creates visible edges.

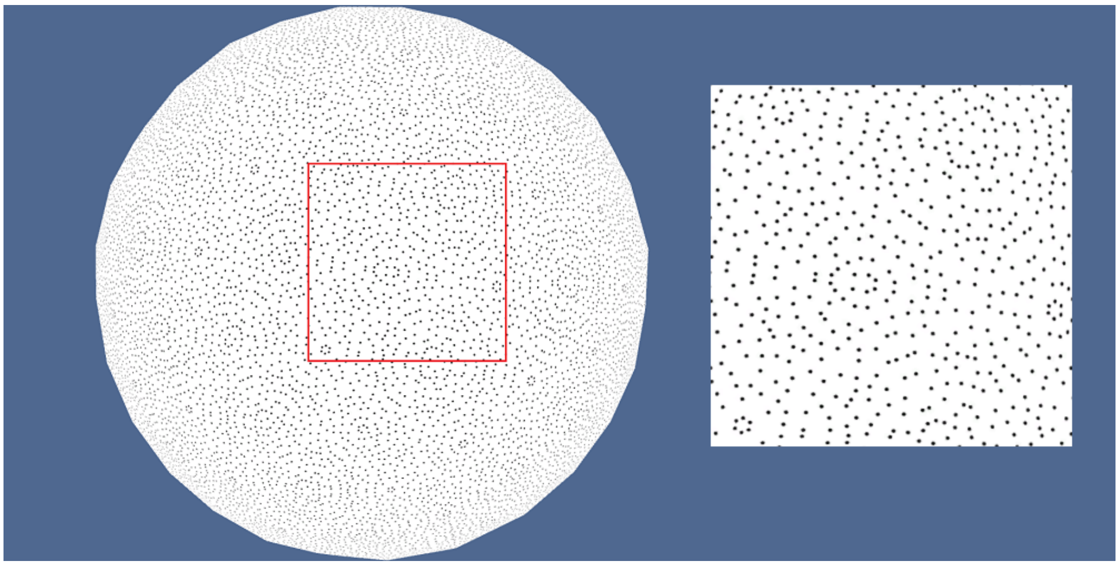


Figure 6.4: A sphere that was textured using only one patch with randomly distributed points. The magnification shows that underlying edges are very hard to see despite that.

CHAPTER 7

Conclusions

This work represents a small foray into texturing and visualization within the framework the Unity engine provides. It gave me the opportunity to familiarize myself with a powerful game engine as well as learn a new programming language in the form of C# and Cg/HLSL respectively. On top of that, various new concepts had to be incorporated into the program such as triangulation and particle calculations. The meshes used were created in blender, another technology I was not too well versed in and thus had to get to know.

The original goal of this work, the seamless texturing of an arbitrary 3D-object, was achieved. There were, however, unexpected results. First is the worse than quadratic performance. Second is the fact that quick results heavily depend on the user choosing a "good" amount of texture vertices. Both of these problems have their origin in the chosen implementation of triangulation and can be solved by further optimizing this implementation. However, performance optimization was not considered the main focus of this thesis and the long runtime can be considered a pre-processing step if mesh generation is static. In dynamic cases, though, the algorithm proves to be too slow.

This being said, an interesting question presenting itself at the end is this: Is Unity 3D the best tool to work with in this context? As stated earlier, Unity has certain benefits. These include portability, accessibility and a solid library of basic functionality. For example, the framework contains libraries for vector calculations, rigid body-physics and most importantly presentation in the form of lighting calculations and a streamlined texturing process. The editor is easy to operate and the provided documentation and tutorials cover most of the important topics. But despite all these benefits, Unity also suffers from certain shortcomings. As it turns out, trying to manipulate meshes on such a fine level (i.e. manipulating single vertices) quickly shows the engine's limits. Especially the used data structures are what hinders easy control of meshes. This is mostly due to the fact that in game development such functionality is seldom needed since most 3D-models are either used as-is or provided with an animation rig prior to import into

Unity. Another library that would have been useful is one containing functions from the field of computational geometry, such as geodesic distance or three dimensional surface manipulation. This is something even games would benefit from and some functionality will in fact be added in the next major update.

To summarize my experience with Unity, I think it is a great engine that gave me the opportunity to work from the ground up, implementing all the functionality I needed myself. But the alternative of having some functions already at my disposal would have possibly resulted in a more powerful algorithm overall. So for the future if I want to create a game I definitely will consider Unity again, but aside from that I'd probably use a more common, better equipped framework.

One such library was already mentioned: CGAL. An alternative would be NVIDIA's "Flex" framework. Both of them provide functions to easily calculate surface properties as well as reposition vertices. Instead of writing the whole script in Unity it would have been possible to create a pipeline that first repositions vertices in Flex and then triangulates them using CGAL-functionality before using Unity to texture and display the resulting mesh. Another alternative for triangulation would have been MeshLab.

These suggestions are mentioned because the algorithm in this work can definitely be improved. The main flaw is the triangulation procedure, which takes too long to finish. As mentioned during the evaluation this is due to the physics calculations continuing in this step. An obvious solution would be to implement either point cloud- or Delaunay-triangulation that has already been tested. Also, if Unity is used for point repositioning, a robust geodesic distance calculation would benefit the precision greatly, as well as overhauled or extended data structures. Main focus hereby should be accessible neighbourhood structures with fast lookup.

Possible improvements aside, the algorithm definitely represents a stepping stone towards more complex visualization. The assignment of texture coordinates can easily be extended to accommodate for texture patches with edge constraints. Right now the input patches have to fit together regardless of rotation. But by changing some of the code, it would be possible to choose from a larger number of patches with edge constraints, making only some of the other patches possible neighbours. This could be combined with automated texture generation from a sample image, making the task of texturing an object even easier for the user.

If physics calculations are reliable and fast, splitting objects would present another way to add to the versatility of the algorithm. When an object's animation calls for a surface that is divided, new texture vertices could be introduced into the new seam and repositioned at runtime to once again close the texture over the object's parts.

But even the fact that texturing using only a single patch proves to be valid can be used to further advance the algorithm. If a texture containing randomly distributed points is applied, each point on the texture can in turn be mapped to another, more complex texture or even geometry.

These examples are just some of the possibilities this work opens up.

Bibliography

[ACK13] Marco Attene, Marcel Campen, and Leif Kobbelt. Polygon mesh repairing: An application perspective. *ACM Computing Surveys*, 45(2):15:1–15:33, March 2013, doi:10.1145/2431211.2431214.

[ASB13] Maryam Asghari, Mojtaba T Sadat, and Pawel Boguslawski. An overview on boundary representation data structures for 3d models representation. In *International Symposium & Exhibition on Geoinformation (ISG)*, 2013.

[Bar89] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, pages 223–232, 1989, doi:10.1145/74333.74356.

[Bar94] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 23–34, 1994, doi:10.1145/192161.192168.

[BCE^{+}13] David Bommes, Marcel Campen, Hans-Christian Ebke, Pierre Alliez, and Leif Kobbelt. Integer-grid maps for reliable quad meshing. *ACM Transactions on Graphics*, 32(4):98:1–98:12, July 2013, doi:10.1145/2461912.2462014.

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

[BLP^{+}13] David Bommes, Bruno Lévy, Nico Pietroni, Enrico Puppo, Claudio Silva, Marco Tarini, and Denis Zorin. Quad-mesh generation and processing: A survey. *Computer Graphics Forum*, 32(6):51–76, September 2013, doi:10.1111/cgf.12014.

[BMO^{+}12] Jan Bender, Matthias Müller, Miguel A. Otaduy, Matthias Teschner, and Miles Macklin. A survey on position-based simulation methods in computer graphics. In *Computer Graphics Forum*, volume 33, pages 228–251, 2012, doi:10.1111/cgf.12346.

[Cas12] Thomas J. Cashman. Beyond catmull-clark? a survey of advances in subdivision surface methods. *Computer Graphics Forum*, 31(1):42–61, February 2012, doi:10.1111/j.1467-8659.2011.02083.x.

[Cat74] Edwin Catmull. A subdivision algorithm for computer display of curved surfaces. Technical report, DTIC Document, 1974.

[CCC$^+$08] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. Meshlab: an open-source mesh processing tool. In *Eurographics Italian Chapter Conference*, volume 2008, pages 129–136, 2008, doi:10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[CHK13] Marcel Campen, Martin Heistermann, and Leif Kobbelt. Practical anisotropic geodesy. In *Computer Graphics Forum*, pages 63–71, 2013, doi:10.1111/cgf.12173.

[Cou13] Murilo G Coutinho. *Dynamic simulations of multibody systems*. Springer Science & Business Media, 2013.

[CSHD03] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 287–294, 2003, doi:10.1145/1201775.882265.

[DH92] Ding-Zhu Du and Frank Hwang, editors. *Computing in Euclidean Geometry*. World Scientific Publishing Co., Inc., 1992.

[ESEK$^+$00] Jihad El-Sana, Francine Evans, Aravind Kalaiah, Amitabh Varshney, Steven Skiena, and Elvir Azanli. Efficiently computing and updating triangle strips for real-time rendering. *Computer-Aided Design*, 32(13):753–772, 2000, doi:10.1016/S0010-4485(00)00071-3.

[FL05] Chi-Wing Fu and Man-Kang Leung. Texture tiling on arbitrary topological surfaces using wang tiles. In *Proceedings of the 16th Eurographics Symposium on Rendering*, EGSR '05, pages 99–104, 2005, doi:10.2312/EGWR/EGSR05/099-104.

[FP09] Andreas Fabri and Sylvain Pion. Cgal: The computational geometry algorithms library. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 538–539, 2009, doi:10.1145/1653771.1653865.

[KCD15] Martin Kolář, Alan Chalmers, and Kurt Debattista. Repeatable texture sampling with interchangeable patches. *The Visual Computer*, 31(10):1–10, 2015, doi:10.1007/s00371-015-1161-4.

[KS16] Marc Khoury and Jonathan Richard Shewchuk. Fixed points of the restricted delaunay triangulation operator. In *32nd International Symposium on Computational Geometry (SoCG 2016)*, volume 51 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 47:1–47:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, doi:10.4230/LIPIcs.SoCG.2016.47.

[LAM01] Thomas Larsson and Tomas Akenine-Möller. Collision detection for continuously deforming bodies. In *Eurographics 2001 - Short Presentations*, pages 325–333, 2001, doi:10.2312/egs.20011005.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, 1987, doi:10.1145/37401.37422.

[Lin09] Patrick Lindemann. The Gilbert-Johnson-Keerthi distance algorithm. *Algorithms in Media Informatics*, 2009.

[LTW04] Hong-Wei Lin, Chiew-Lan Tai, and Guo-Jin Wang. A mesh reconstruction algorithm driven by an intrinsic property of a point cloud. *Computer-Aided Design*, 36(1):1–9, 2004, doi:10.1016/S0010-4485(03)00064-2.

[MCG$^+$12] Emilie Marchandise, Paolo Crosetto, Christophe Geuzaine, Jean-François Remacle, and Emilie Sauvage. *Quality open source mesh generation for cardiovascular flow simulations*, pages 395–414. 2012.

[MK03] Sebastian Magda and David Kriegman. Fast texture synthesis on arbitrary meshes. In *Proceedings of the 14th Eurographics Symposium on Rendering*, EGSR '03, pages 82–89, 2003, doi:10.2312/EGWR/EGWR03/082-089.

[MVS14] Andre Maximo, Luiz Velho, and Marcelo Siqueira. Adaptive multi-chart and multiresolution mesh representation. *Computers & Graphics*, 38:332–340, February 2014, doi:10.1016/j.cag.2013.11.013.

[MW88] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 289–298, 1988, doi:10.1145/54852.378528.

[NC99] Fabrice Neyret and Marie-Paule Cani. Pattern-based texturing revisited. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 235–242, 1999, doi:10.1145/311535.311561.

[NPPZ10] Matthias Nieser, Jonathan Palacios, Konrad Polthier, and Eugene Zhang. Hexagonal global parameterization of arbitrary surfaces. In

ACM SIGGRAPH ASIA 2010 Sketches, SA '10, pages 5:1–5:2, 2010, doi:10.1145/1899950.1899955.

[NY06] Timothy S. Newman and Hong Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30(5):854–879, 2006, doi:10.1016/j.cag.2006.07.021.

[PBFJ05] Serban D. Porumbescu, Brian Budge, Louis Feng, and Kenneth I. Joy. Shell maps. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 626–633, 2005, doi:10.1145/1186822.1073239.

[PFH00] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 465–470, 2000, doi:10.1145/344779.344987.

[RC11] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1–4, 2011.

[Reb93] Stefano Rebay. Efficient unstructured mesh generation by means of delaunay triangulation and bowyer-watson algorithm. *Journal of Computational Physics*, 106(1):125–138, May 1993, doi:10.1006/jcph.1993.1097.

[Ree83] William T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, April 1983, doi:10.1145/357318.357320.

[SCA02] Cyril Soler, Marie-Paule Cani, and Alexis Angelidis. Hierarchical pattern mapping. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 673–680, 2002, doi:10.1145/566570.566635.

[SFM12] Daniel Shiffman, Shannon Fry, and Zannah Marsh. *The nature of code*. D. Shiffman, 2012.

[She02] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry*, 22(1):21–74, May 2002, doi:10.1016/S0925-7721(01)00047-5.

[SSK$^+$05] Vitaly Surazhsky, Tatiana Surazhsky, Danil Kirsanov, Steven J. Gortler, and Hugues Hoppe. Fast exact and approximate geodesics on meshes. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 553–560, 2005, doi:10.1145/1186822.1073228.

[SYXA$^+$11] Shi-Yu, Qing Xing, Ergun Akleman, Jianer Chen, and Jonathan Gross. Pattern mapping with quad-pattern-coverable quad-meshes. In *ACM SIGGRAPH 2011 Talks*, SIGGRAPH '11, pages 26:1–26:1, 2011, doi:10.1145/2037826.2037860.

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 65–70, 1992, doi:10.1145/133994.134010.

[Ton91] David Tonnesen. Modeling liquids and solids using thermal particles. In *Graphics Interface '91*, pages 255–262, 1991.

[Tur91] Greg Turk. Generating textures on arbitrary surfaces using reaction-diffusion. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, pages 289–298, 1991, doi:10.1145/122718.122749.

[Tur92] Greg Turk. Re-tiling polygonal surfaces. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 55–64, 1992, doi:10.1145/133994.134008.

[TWZZ07] Jie Tang, Gang-Shan Wu, Fu-Yan Zhang, and Ming-Min Zhang. Fast approximate geodesic paths on triangle mesh. *International Journal of Automation and Computing*, 4(1):8–13, January 2007, doi:10.1007/s11633-007-0008-5.

[VdFG99] Luiz Velho, Henrique Luiz de Figueiredo, and Jonas Gomes. Hierarchical generalized triangle strips. *The Visual Computer*, 15(1):21–35, 1999, doi:10.1007/s003710050160.

[Win11] Olov Winberg. Examining automatic texture mapping of arbitrary terrains. Master's thesis, Mälardalen University, School of Innovation, Design and Engineering, 2011.

[WL01] Li-Yi Wei and Marc Levoy. Texture synthesis over arbitrary manifold surfaces. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 355–360, 2001, doi:10.1145/383259.383298.

[WSB14] Thomas Waltemate, Björn Sommer, and Mario Botsch. Membrane mapping: Combining mesoscopic and molecular cell visualization. In *Proceedings of the 14th Eurographics Workshop on Visual Computing for Biology and Medicine*, VCBM '14, pages 89–96, 2014, doi:10.2312/vcbm.20141187.

[YHBZ01] Lexing Ying, Aaron Hertzmann, Henning Biermann, and Denis Zorin. Texture and shape synthesis on surfaces. In *Proceedings of the 12th Eurographics Workshop on Rendering*, EGWR '01, pages 301–312, 2001, doi:10.2312/EGWR/EGWR01/301-312.

[ZG03] Steve Zelinka and Michael Garland. Interactive texture synthesis on surfaces using jump maps. In *Proceedings of the 14th Eurographics Symposium on Rendering*, EGSR '03, pages 90–96, 2003, doi:10.2312/EGWR/EGWR03/090-096.

[ZZV^{+}03] Jingdan Zhang, Kun Zhou, Luiz Velho, Baining Guo, and Heung-Yeung Shum. Synthesis of progressively-variant textures on arbitrary surfaces. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 295–302, 2003, doi:10.1145/1201775.882266.