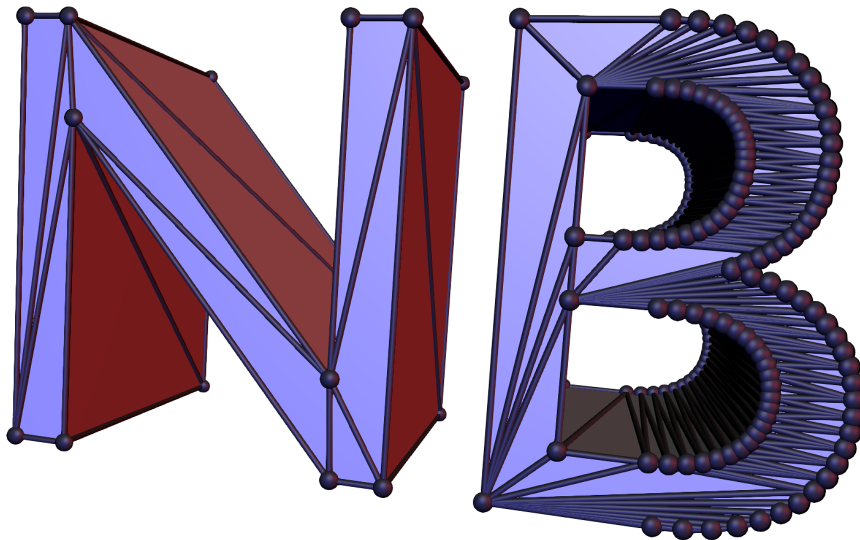
Universität Trier
Fachbereich Informatik

Masterarbeit

Nubuck. A system for visualization and animation of geometric algorithms.



Christian Jäger
Juni 2015

Universität Trier
Lehrstuhl für Datenstrukturen und Effiziente Algorithmen
Prof. Dr. Stefan Näher
Universität Trier
Fachbereich IV - Informatik

Masterarbeit
Nubuck. A system for visualization and animation of geometric algorithms.

Christian Jäger
Matrikelnummer: 984085
Juni 2015

Betreuer: Prof. Dr. Stefan Näher

Erstprüfer: Prof. Dr. Stefan Näher
Zweitprüfer: Prof. Dr. Stephan Diehl

ERKLÄRUNG ZUR BACHELORARBEIT / MASTERARBEIT

Hiermit erkläre ich, dass ich die Bachelorarbeit / Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe.

Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

Datum	Unterschrift

Contents

1. Introduction **6**

2. General Architecture **10**

3. Examples **12**
3.1. Example 1 13
3.2. Example 2 17

4. The programming interface **22**
4.1. Data structures 22
4.1.1. Polyhedrons 22
4.1.2. A mesh data structure 24
4.1.3. Rational numbers 28
4.2. Implementation of Example 1 29
4.2.1. Notation 29
4.2.2. Basic Operator Structure 30
4.2.3. Direct Input 39
4.2.4. Operator Panel 43
4.2.5. Animations 47

5. The rendering system **52**
5.1. Introduction to rendering 52
5.2. OpenGL 56
5.3. Nubuck renderer architecture 57
5.3.1. Class GLWidget 57
5.3.2. Class Renderer 58
5.3.3. Meshes 59
5.3.4. Effects 61
5.4. Nodes and edges 63
5.4.1. SM_LINES 63
5.4.2. SM_BILLBOARD_FAST 65
5.4.3. SM_BILLBOARD_NICE 68
5.4.4. SM_NICE 71
5.5. Transparency 73
5.6. Screenshots 78

6. Conclusion **81**
6.1. Future Work 81

A. Installation and Building **83**
A.1. Requirements 83
A.2. Building Nubuck 84
A.3. Running Nubuck 85
A.4. Changing the Nubuck sources 86
A.5. Building a standalone application 86
A.6. Building a plugin 86

1. Introduction

This thesis presents Nubuck, which is a new tool for visualizing and animating three-dimensional (3-D) geometric algorithms using the LEDA library. [1] A visualization tool like this helps understanding and presenting geometric algorithms in both development and education. The main contribution of this thesis is the description of a modern implementation that supports 3-D geometry and is readily available on today's machines.

The area of computer science that studies geometric problems is called computational geometry. Its two main branches are combinatorial computational geometry and numerical computational geometry. The study of problems that are stated in terms of discrete entities, such as points, lines, polygons and polyhedrons, is called combinatorial computational geometry, whereas parametric surfaces like curves are the subject of numerical computational geometry. The focus of this paper is combinatorial computational geometry. A comprehensive introduction can be found in O'Rourke [1998] or Berg et al. [2008]. Some of the classical problems in the field of computational geometry are:

- Convex Hull. Given a set P of points in the plane, find the smallest convex polygon $CH(P)$ that contains all the points of P.
- Delaunay Triangulation. Given a set P of points in the plane, find a subdivision of $CH(P)$ into triangles, such that no point of P is inside the circumcircle of any triangle.

Both problems generalize to higher dimensions.

Applications of computational geometry include computer graphics, computer aided design (CAD) and modeling, geographic information systems (GIS), robotics and motion planning.

There are several software libraries that provide the basic building blocks for writing geometric algorithms, such as LEDA and CGAL[2]. Both are C++ libraries that provide the necessary data types and composable implementations of most common geometric algorithms.

Of course, a natural and efficient way of understanding, debugging and presenting geometric algorithms is visualization. The most prominent part of visualization is rendering, which is generating the image on the screen. A good visualization, however, should also provide intuitive controls that allow easy exploration of the scene. A visualization can also use animations to clarify the relation between two objects, or to show the effect of

[1] Nubuck is a special kind of leather (german: Leder, pronounced like LEDA). The similarity to the phrase „new bug" is unfortunate.

[2] www.cgal.org

two consecutive steps of an algorithm. Moreover, animations are visually appealing.

Obviously, it is uneconomical to write the visualization part from scratch every time a geometric algorithm is implemented, so it is wise to use some kind of library or tool that does the heavy lifting. Moreover, visualization might not be an integral part of the product, because only the combinatorial results are of interest. Range queries on databases, for example, can be stated in geometric terms. Especially in this case, where visualization is only used in the course of development, it is preferable to minimize the effort put into it. Ideally, the visualization tool is compatible with the application's data structures. In addition, many tools provide a sandbox, so that inputs can be created easily.

Unfortunately, some of the visualization tools that have been used in the past are not readily available on modern operating systems. Also, some of these tools were implemented in a programming language that modern developers are generally not comfortable with, which makes them hard to maintain and extend. XYZ GeoBench [Schorn, 1991], for example, was implemented in an object oriented extension to Pascal on the Apple Macintosh. The Workbench for Computational Geometry [Epstein et al., 1994] was implemented in Smalltalk, also for the Apple Macintosh. GeoLab [de Rezende and Jacometti, 1993] was written in C++, although it runs only on SparcStations under Sun/OS. LiveCG [Kürten and Mulzer, 2014] is a more recent attempt to develop a visualization tool for modern systems in Java. GeoWin [Bäsken and Näher, 2001] is written in C++ and runs on modern systems. It's part of the LEDA library, but it's generic in the sense that it can also be interfaced with other geometry packages like CGAL. All of the programs mentioned so far focus mostly on the visualization of 2-D algorithms and there are only a few programs that allow the examination of 3-D polyhedrons. GASP-II [Shneerson and Tal, 1997] allows the presentation and exploration of 3-D algorithms in an distributed electronic classroom. LEDA provides the `d3_window` data type that supports 3-D visualization.

This paper presents the Nubuck system for visualization and animation of 3-D algorithms using the LEDA software library. Nubuck is intended as a replacement for LEDA's `d3_window` data type.

LEDA is a C++ library that provides efficient data types and algorithms for combinatorial and geometric computing. It offers building blocks for objects such as rational numbers, dictionaries, priority queues, graphs, polygons, and many more. LEDA was designed with ease-of-use in mind, and its features can be combined in an efficient manner, without requiring an intimate knowledge of the field of algorithms. LEDA is used in a variety of areas such as telecommunication, GIS, VLSI design, computational biology and computer-aided design. [3]

The LEDA project was started in the fall of 1988 by Kurt Mehlhorn and Stefan Näher at Saarland University in Saarbrücken. Stefan Näher became the head of the project and has been the main designer and implementer of LEDA. Licenses of LEDA are currently

[3] `http://www.algorithmic-solutions.com/leda/about/index.htm`

distributed by Algorithmic Solutions. [4]

LEDA's `d3_window` data type supports 3-D visualization of a single polyhedron. Figure Fig. 1.1a shows an application that computes the 3-D convex hull. The upper part of the window is called the panel section and contains all elements of the graphical user interface (GUI), such as buttons and sliders. The lower part is called the drawing section and shows the polyhedron. The polyhedron is lit properly by a single directional light, so the faces appear in different shades of gray. Various attributes of the nodes and edges, like color and size, can be changed to highlight aspects of the polyhedron. In addition, edges can be drawn with arrow heads or as beziér curves. The camera uses orthographic projection. It can be zoomed in and out by pressing the left and middle mouse button, respectively. The right mouse button toggles the rotation of the camera. When the camera is rotating, its rotation axis and angular velocity are determined by the position of the mouse pointer relative to the center of the screen.

The rendering is done in software, which means that `d3_window` does not utilize the dedicated graphics hardware found in almost all consumer machines today. A graphics processing unit (GPU) greatly increases the rendering performance and enables advanced rendering techniques that enhance the quality of the rendered image. The LEDA source code contains a data type called `d3_gl_window`, which is an effort to implement a hardware accelerated renderer. However, no official documentation or sample application is provided.

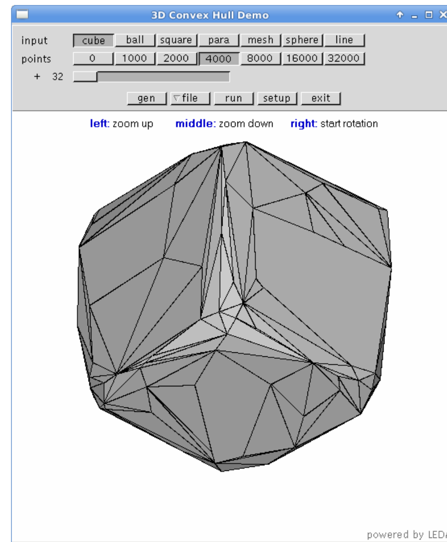
Nubuck seeks to be an improved replacement for `d3_window` by making the following contributions:

- Nubuck implements a hardware accelerated renderer using the OpenGL API. Modern shading techniques are employed to produce visually appealing renderings. High resolution images can be generated to be used in print. Features like perfect transparency and stylized hidden lines help to communicate the shape of complex geometry.
- Similar to many 2-D visualization tools like LEDA's GeoWin, Nubuck provides a sandbox environment for the construction of input scenes. Also, there is no restriction on the number of polyhedrons in the scene.
- Nubuck uses the Qt framework[5] to provide a modern looking graphical user interface (GUI). Custom widgets enable a pleasant and fluent user experience.

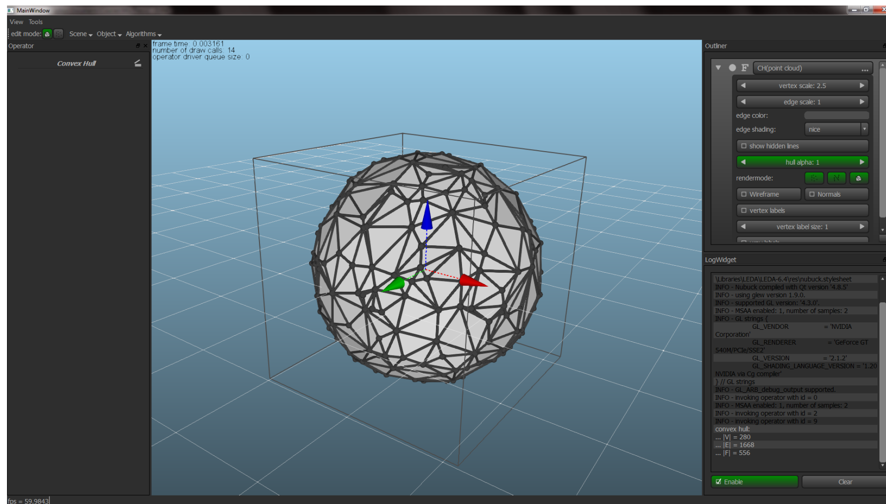
Figure 1.1b shows the 3-D convex hull operator of Nubuck.

The next section gives describes the general architecture of the system. In section 3, two examples are given that show how Nubuck can be used. Section 4 introduces the programming interface. In section 5, implementation details of Nubuck are explained with focus on the renderer. Section 6 concludes this paper. Details of the build system are the topic of appendix A.

[4] `http://www.algorithmic-solutions.com`
[5] `http://www.qt.io/developers/`



(a) LEDA's `d3_window`.



(b) Nubuck.

Figure 1.1.: A comparison between `d3_window` on the top and Nubuck on the bottom shows the differences in both the richness of the graphical user interface and the image quality.

2. General Architecture

Nubuck consists of two parts. We call the application itself the *sandbox*. The sandbox is the *interactive* part of Nubuck and it comprises all the facilities that are available at runtime, such as mesh generation, mesh manipulation, camera controls, algorithms, and so on. The programming application interface (API), on the other hand, is the set of functions and classes that a programmer uses to implement an algorithm that runs inside the sandbox.

First of all, we need to introduce some terminology. The *world* is the collection of all *entities*. Sometimes the word *scene* is used to refer to the visible part of the world. An entity is the generic term for anything that can be created and manipulated by the user. Currently, there are three entity types: *mesh*, *text*, and *transform gizmo*.

The *mesh* entity is the workhorse of Nubuck, as it represents general 3-D geometry and it is both the input and output of virtually all algorithms. We distinguish between a mesh entity and the *graph* it owns. A graph is an object of type `NB::Graph`, which is a subclass of `leda::GRAPH`. As it can be seen later on, a graph is an appropriate mesh data structure. It represents the geometry and topology of the mesh, and consists of vertices, edges, and faces. Attributes that vary among different vertices, edges, or faces — such as position, color, or size — are stored in the graph. A mesh entity owns a graph and places it in the world. Any attributes that pertain to the entire mesh — such as its name, rendering options, and visibility — are stored in the mesh entity, not the graph.

An *operator* creates, modifies, or deletes entities in the world. There are built-in operators for random point generation, translation and scaling, saving and loading, convex hull computation, and so on. At any time, there is exactly one *active* operator, which receives any *events* first.

Nubuck is *event-driven*, which is a common paradigm in GUI based applications. Any user inputs, like mouse clicks or key presses, generate an event that an operator reacts to accordingly.

In addition, an operator creates an *operator panel*, which is the GUI for that operator. An operator panel contains widgets (buttons, scrollbars, and so on), which generate additional events.

Nubuck does not distinguish between different types of operators; that is, all operators are derived from the same base class `Operator`. Conceptually, however, we can make out three classes of operators. *Generators* create inputs from scratch. There are generators to randomly create points inside a bounding domain, or to load a Wavefront `.obj` file. *Mesh operators* and *vertex operators* modify the currently selected mesh or vertices. There are

mesh operators that compute the Convex Hull and Delaunay Triangulation, and there is a vertex operator that merges two adjacent vertices. *Algorithms* are operators that implement a visualization of some sort, which typically consists of several steps, or provides some kind of interactive controls and animations. Put differently, algorithms not only show the result, but also the process of a computation.

By convention, the operators of the former two classes are designed to be *non-blocking*, which means that they take effect immediately and do not have an „apply" button. This enables a fluent workflow with a minimal number of clicks. A random polyhedron with 500 vertices, for example, can be created as follows: Invoke the random point generator. At this point, a random point set appears at the origin of the world. In the operator panel, change the bounding domain to „sphere" and set the size to 500. The random point set changes accordingly. Then, invoke the Convex Hull operator. The hull of the random point set is computed instantly.

Algorithms, on the other hand, are *blocking* by default; for example, you cannot invoke any other operator while an algorithm is running. However, the user can prematurely terminate an algorithm by clicking on the „pop operator" button in the operator panel.

3. Examples

After the elementary concepts of the Nubuck system have been introduced, we now give two examples that demonstrate its usefulness. In the following, the focus is on the interactive aspects of Nubuck and it is simply assumed that the two algorithms we use already exist, as the programming interface is the topic of later sections. In fact, a full implementation of the first example will be given in section 4.2.

First, we will see how Nubuck can be employed in classroom scenarios. In a typical undergraduate course in computational geometry, it is shown that there is a close relation among three of the most fundamental geometric structures, namely the 2-D Delaunay triangulation, the 2-D Voronoi Diagram, and the 3-D Convex Hull. Teaching a 3-D relation like this is inherently difficult using traditional devices like blackboards and static PowerPoint presentations. Thus the first example is an interactive visualization of these three geometric structures that provides free camera controls and allows easy manipulation of the input points. Using a program like this, students can easily observe how a change in one structure corresponds to changes in the other structures. What sets Nubuck apart from other visualization tools, like GeoWin and `d3_window`, and what makes this example possible in the first place, is its ability to show 2-D and 3-D structures side-by-side.

The second example demonstrates the value of Nubuck in the course of debugging geometric algorithms. More precisely, we study the execution of the algorithm of Preparata and Hong for the construction of the 3-D Convex Hull (see Preparata and Hong [1977]). It is a *divide-and-conquer* algorithm that essentially wraps two convex polyhedrons with a series of connected triangles in a way that is akin to the *gift wrapping* algorithm for the 2-D Convex Hull; it can be imagined that the two polyhedrons are wrapped by a rotating plane. Details of the algorithm will be given later on, but figure 3.4 already gives the general idea of the wrapping process. An important result is that, contrary to intuition, the new faces do not always trace a simple cycle on the original polyhedrons. So, naturally, the algorithm must be tested to work in this special case and it's necessary to reproduce the input polyhedrons that can be found in the literature; for example in O'Rourke [1998]. In the second example, we facilitate Nubuck's editing tools to construct such polyhedrons and save them for future use. This way, the user is spared from the manual specification of vertices and indices.

For the purpose of adequately presenting the examples it will be inevitable to talk about the theory of computational geometry — that is what Nubuck is there for, after all — but we try to be as succinct as possible.

3.1. Example 1

Arguably three of the most fundamental structures in the field of computational geometry are the *Convex Hull*, the *Voronoi Diagram*, and the *Delaunay Triangulation*.

In this example, we will describe a Nubuck visualization that shows these three geometric structures in the same scene to highlight certain connections among them. Also, the user can drag around input points and the structures are updated accordingly, such that the influence of one point is immediately apparent. The interactive nature of the visualization makes it a useful tool for teaching the subject to students.

It is mandatory to review some theoretical foundations that are relevant for the example. However, we do not intend to give a comprehensive discussion on any of these structures, as they are well covered in introductory textbooks such as Preparata and Shamos [1985], Berg et al. [2008], and O'Rourke [1998].

A set C of points in the plane is *convex* if it contains the line segment connecting each pair of points $p \in C$ and $q \in C$. The right hand side of figure 3.1 shows an example of a non-convex (concave) set, and a line segment xy that violates the convexity property. The red polygon on the left hand side of the figure is convex. Moreover, it is the Convex Hull of the 20 points it encloses.

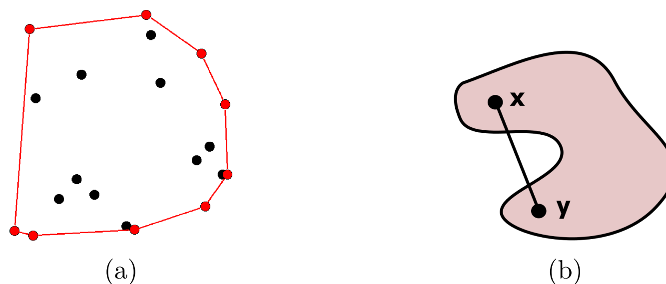


Figure 3.1.: (a) The set of points bounded by the red polygon is convex, and is the result of a Convex Hull computation. The image has been created using LEDA's GeoWin.[1] (b) An example for a non-convex (concave) point set. The line segment xy is not contained in the interior of the set and thus violates the convexity property.

Let S be a finite set of points in the plane. The *Convex Hull* of S, denoted by $CH(S)$, is the smallest (with respect to set inclusion) convex set that contains S. In the plane, the Convex Hull has the shape of a convex polygon, which can be easily represented by an ordered list of its vertices. The Convex Hull can also be explained by a physical analogy: if we stretch an elastic rubber band around all points in S and then let it go, the boundary of the Convex Hull is the shape assumed by the rubber band when contracted.

The Convex Hull problem generalizes to higher dimensions. In 3-D space, the result is

[1] `http://www.algorithmic-solutions.info/leda_guide/geo_algs/convex_hull.html`

a convex polyhedron, which is the natural extension of a polygon to 3-D. A rigorous treatment of polyhedrons is given in section 4.1.1, but for now it suffices to know that a polyhedron is the volume bounded by a finite set of polygonal faces.

Let P be a set of points in the plane. A *triangulation* of P is a decomposition of the convex hull of P into triangles, such that the set of vertices of all triangles coincides with P and two triangles share a common face, that is an edge or a vertex, or don't intersect at all. In general, a given point set can be triangulated in more than one way. Figure 3.2a shows an arbitrary triangulation of a point set.

A *Delaunay Triangulation* (DT) is a triangulation that satisfies the *sphere criterion*: the circumsphere of every simplex is empty, i.e., it does not contain any of the given points in its interior.

The triangulation in figure 3.2a is not Delaunay, as can be seen in figure 3.2b: the edge enclosed in the yellow quadrilateral violates the sphere criterion. The Delaunay Triangulation for the same point set is shown in figure 3.2c. The Delaunay Triangulation has several properties that make it well-suited for a lot of applications, where the most important one is that the Delaunay Triangulation maximizes the minimum angle. Put in another way we can say that the Delaunay Triangulation avoids thin triangles, which is not only visually appealing, but also important for practical purposes like numerical interpolation.

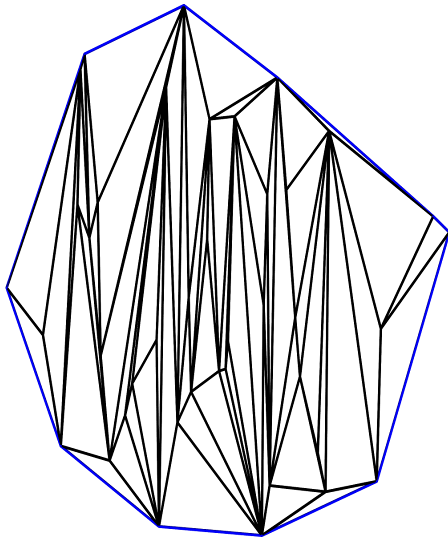
Let $S \subset E^2$ be a point set in the plane. For every point $x \in E^2$, there is some point in S that is closest (i.e., minimizes the distance) to x. Here, we use the Euclidean distance, which is important for the duality property we show later on. Let $close(x) \subset S$ be the set of points in S that are closest to x. If the closest point in S is not unique, then $1 < |close(x)|$. All points on the *bisector* of two points p, q, for example, are equidistant to p and q. The *Voronoi Diagram* is the set of all points x with $1 < |close(x)|$. [2] Figure 3.2d shows the Voronoi Diagram of 20 points (also called *sites*). The edges and vertices of the Voronoi Diagram partition the plane in a set of convex faces, called *Voronoi Regions*, and every Voronoi Region is the set of all points closest to some site.

Next we discuss the relationship between the Voronoi Diagram and the Delaunay Triangulation, resp. the Convex Hull and the Delaunay Triangulation.

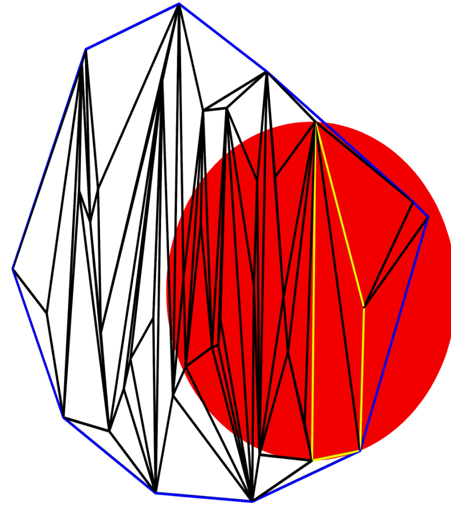
It can be shown that the Delaunay Triangulation is the straight-line embedding of the *dual graph* of the Voronoi Diagram. The dual graph D of a Voronoi Diagram has a node for every Voronoi Region and an edge between two nodes if the corresponding regions are adjacent [Berg et al., 2008]. A straight-line embedding of D draws every edge between two nodes $v, w \in D$ as a straight line segment $\overline{vw}$.

There is a close relation between the 2D Delaunay Triangulation and the 3D Convex Hull, such that the Delaunay Triangulation can be constructed from the Convex Hull. Let $\lambda : E^2 \to E^3$, $\lambda(x_1, x_2) = (x_1, x_2, x_1^2 + x_2^2)$ be the lifting map, which projects all points in the plane on a paraboloid. The Delaunay Triangulation of a point set $P \subset E^2$ is exactly the orthogonal projection of the lower faces of the Convex Hull of $\lambda(P)$ on the

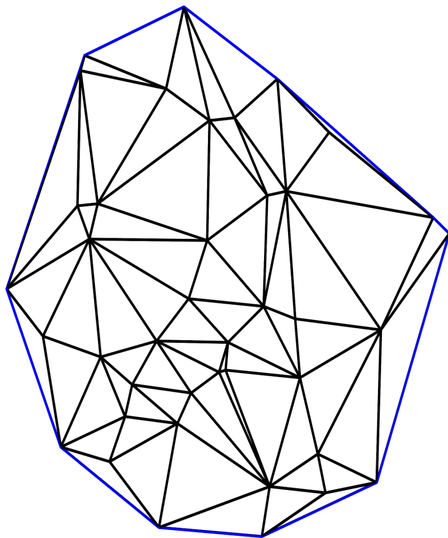
[2] `http://www.algorithmic-solutions.info/leda_guide/geo_algs/voronoi.html`



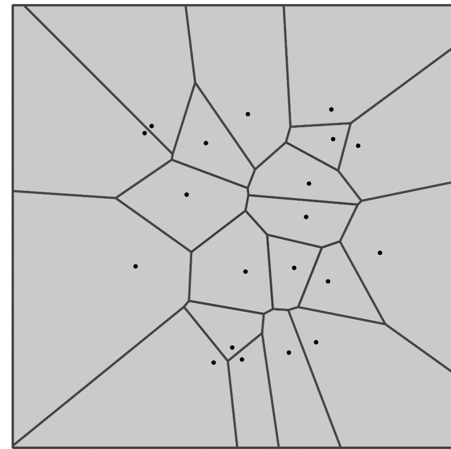
(a)
An arbitrary triangulation of a point set.



(b)
The edge enclosed in the yellow quadrilateral violates the sphere criterion.



(c)
The Delaunay Triangulation of the same point set.



(d)
The Voronoi Diagram of a point set. Note that the outer faces of the diagram can be unbounded, but we clipped the diagram against a large rectangle for drawing.

Figure 3.2.: Triangulations, Delaunay Triangulations, and Voronoi Diagram in 2-D space.

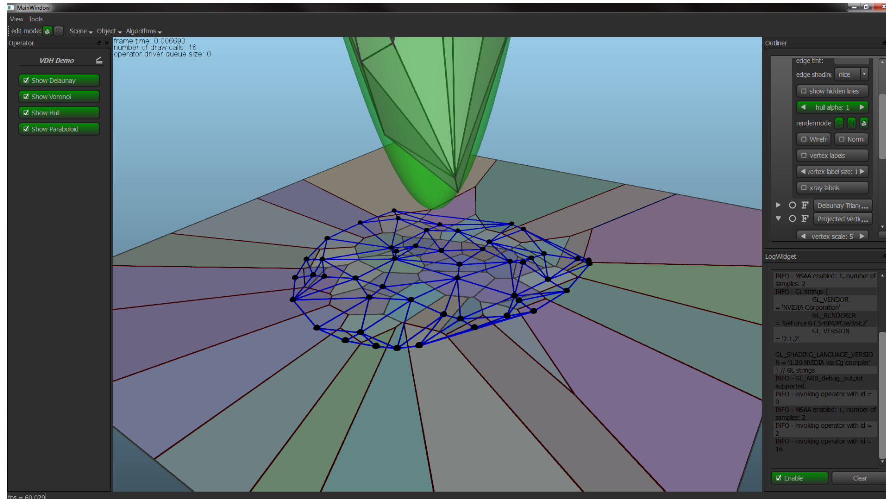


Figure 3.3.: The first example algorithm shows three elementary geometric structures and the relationships among them. Using Nubuck, it is possible to show the 2-D Voronoi Diagram and 2-D Delaunay Triangulation next to the 3-D Convex Hull.

xy-plane.

This concludes the theoretical foundations and we now begin the description of the Nubuck algorithm, which is also shown in figure 3.3. A point set is expected as input, so we generate one prior to launching the algorithm, e.g., by clicking on the scene menu button and selecting *Random Points*. In the operator panel, the domain can be set to *In disc* to create a point set in the xy-plane. It is recommended to keep the radius of the disc relatively small, because the z-coordinates of the projected points grow quadratically. Make sure the newly generated point set is selected. Then, invoke the operator by clicking on the button labeled *VDH demo* in the algorithms menu.

Initially, the algorithm shows only the Voronoi Diagram of the input points. As there is no 3-D mesh yet, you may want to press 7 on the Numpad to rotate the camera, so that it looks in $-z$ direction. The visibility of the other geometric structures can be toggled in the operator panel. Besides checkboxes for the Voronoi Diagram, the Delaunay Triangulation, and the Convex Hull, there is also a checkbox that shows the paraboloid on which the input points are projected. The color of the paraboloid is solid green, and for the Convex Hull inside to become visible, the transparency of the paraboloid has to be changed in the outliner window. In addition, the panel contains a spinbox that scales the Convex Hull and the paraboloid on the z-axis, because their large z-coordinates can render them unwieldy at times.

The camera may be freely rotated to watch the scene from different angles. In order to show how the 2-D Delaunay Triangulation can be obtained from the 3-D Convex Hull, we rotate the camera, so that it looks along the positive z-axis. The shortcut for this

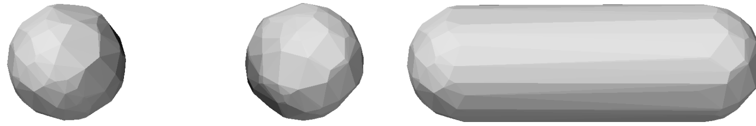


Figure 3.4.: Wrapping two convex polyhedrons.

is Shift + 7. The Voronoi Diagram is either hidden or transparent, so that it does not block the view. At first the two structures do not coincide at all, due to perspective foreshortening, but after setting the camera to orthographic projection by pressing 5 on the numpad, the edges of the two meshes align perfectly.

The input vertices can be selected by rightclicking them, and holding the Shift key while clicking adds to the selection. Selected vertices can be dragged around using a transform gizmo or by pressing the shortcut G. The input vertices are restricted to the xy-plane, so they have two degrees of freedom. As the input vertices are being moved around, all geometric structures are continuously updated and the dual nature of the VD and the DT is exposed.

3.2. Example 2

The second example shows how the process of developing and debugging a complex algorithm benefits from Nubuck's built-in editing facilities.

The algorithm that we look at is a *divide-and-conquer* algorithm for the 3-D Convex Hull and was originally proposed by Preparata and Hong in Preparata and Hong [1977]. The Convex Hull problem and its geometric form, the convex polyhedron, have been defined in the preceding section. Again, we refer the reader to section 4.1.1 for a rigorous treatment of polyhedrons. The algorithm of Preparata and Hong has a worst-case time complexity of $O(n \log n)$, which is optimal in 3-D space. However, the algorithm is difficult to implement and not frequently used in practice.

In accordance with the usual conventions of research work in computational geometry, we simplify the presentation of the algorithm by assuming that the input points are in *general position*; that is, we assume coordinates are pairwise distinct: Let $x_1(p), x_2(p), x_3(p)$ denote the coordinates of a point $p \in E^3$. Each pair p_1, p_2 of input points satisfies $x_i(p_1) \neq x_i(p_2)$, for $1 \leq i \leq 3$. As a corollary, there are no three collinear points and no four coplanar points. A real implementation must of course consider arbitrary point sets, which usually requires special case analysis. See for example Jäger [2012] for a concrete LEDA implementation of this algorithm.

From a bird's eye view, the algorithm works as follows: Sort the input points along the x-axis and divide them into a left half S_1 and a right half S_2, so that S_1 and S_2 are of equal size and $x_1(p_1) < x_1(p_2)\ \forall p_1 \in S_1,\ p_2 \in S_2$. Recursively compute the Convex Hulls $P_1 = CH(S_1)$ and $P_2 = CH(S_2)$. In the merge step, construct the Convex Hull $P = CH(P_1 \cup P_2)$ by wrapping the two convex polyhedrons P_1 and P_2 with a

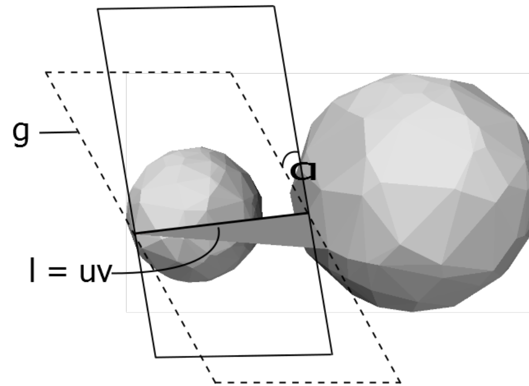


Figure 3.5.: The plane g is rotated around the pivot line segment l until it touches either P_1 or P_2.

„cylindrical” band of faces.

We commence the wrapping by finding a plane g supporting P_1 and P_2 from below, such that all points are contained in one half space of g, and g contains only the line segment $l = uv$ connecting two vertices $u \in P_1,\ v \in P_2$. In one iteration of the wrapping, the plane g is rotated around the line segment l until it touches another point of either P_1 or P_2. Without loss of generality, let g hit P_2 first, in point w. It can be shown that w must be a vertex of P_2 that is adjacent to v, and we call v the *winning vertex*. Then the triangular face uvw is added to the Convex Hull P and the wrapping continues with the new pivot $l' = uw$. Figure 3.5 depicts the rotation of g around l. The wrapping procedure constructs a band of triangular faces around P_1 and P_2, and terminates when the band is closed.

The wrapping procedure can be seen as a variant of the *gift wrapping* algorithm, which generalizes the idea of wrapping faces to compute the Convex Hull in any dimension.

In order to achieve the claimed time complexity of $O(n \log n)$, the wrapping must be done in time $O(n)$. A naïve approach of finding the winning vertex is testing all vertices adjacent to u and v. Since one vertex can have a large neighborhood, it can be shown that this implies a quadric runtime. The algorithm of Preparata and Hong, however, realizes the merge step in expected linear time by taking advantage of a monotonicity property that lets the search for the winning vertex resume at the last considered edge. For an extensive theoretical discussion, see Edelsbrunner [1987].

All practical complications set aside, the actual wrapping is fairly intuitive. A physical analogy is to wrap the two convex polyhedrons with paper (hence the name gift wrapping).

Contrary to intuition, however, the newly created faces do not trace simple circles on P_1 and P_2: For each polyhedron, we call the set of edges that are shared by newly created faces *shadow edges*. The shadow edges form a circuit, and by the cylindrical nature of the wrapping procedure it is reasonable to expect that each edge is visited exactly once.

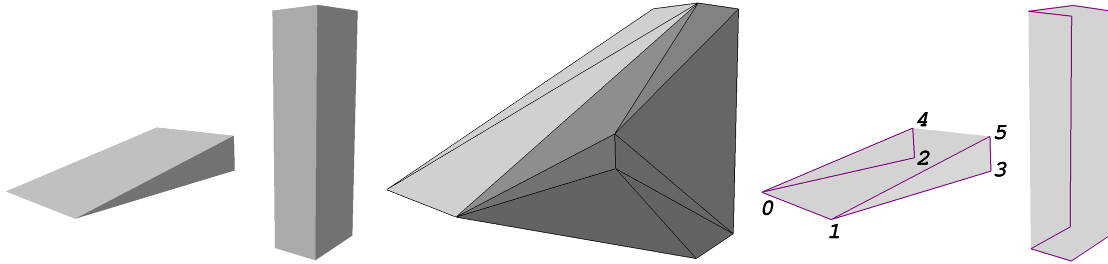


Figure 3.6.: The shadow edges (purple) on the left polyhedron do not form a simple circuit. The edge 01 is visited twice. (Example taken from O'Rourke [1998]).

Surprisingly, this is not necessarily true, and a proof is given in Edelsbrunner [1987]. In O'Rourke [1998], O'Rourke describes the implications for the removal of hidden faces, and also gives an example for a pair of polyhedrons (wedge and box) where the shadow edges form a non-simple circuit. See figure 3.6.

Examples such as this obviously pose as valuable test cases for an implementation of the algorithm, so it would be useful to recreate them in Nubuck.

One way of doing this is creating the test scene in code; for example, the algorithm panel could have buttons that create various scenes of interest. Alternatively, the scene creation code could make up one or more separate operators. Either way, however, involves coding which means high iteration times.

An easier way is to use Nubuck's built-in editing tools, which allow the basic manipulation of meshes. We introduce them by showing how to recreate the wedge shape depicted in figure 3.6.

We start by creating a simple box: click on the scene menu button and select *Platonic Solids* (there are exactly five regular, convex polyhedrons, which are called platonic solids). In the combobox in the operator panel choose the entry *hexahedron (cube)*. Select the cube mesh by rightclicking it.

First, we apply a non-uniform scale, so that the box approximately matches the shape of the wedge. With the mouse cursor in the renderview widget, press number key 2 to switch the transform gizmo mode from *translate* to *scale*. You will notice that the arrow heads are now spheres, indicating scale mode. Click and drag the different axes to scale the box.

Next, we merge the vertices of the frontmost face. With the vertices labeled as shown in figure 3.7, we merge the pairs $(1, 3)$ and $(5, 7)$. This way, we replace the face 1573 by a single edge. Merging two adjacent vertices u, v means adding edges uw for all $w \in adj(v) \setminus adj(u)$ and removing v, such that incident faces are preserved. Since we need to change individual vertices, as opposed to the entire box as before, we have to switch from *object mode* to *edit mode* first: Press TAB or the corresponding button in the toolbar. You will notice that the menu bar changed and now shows a menu button named

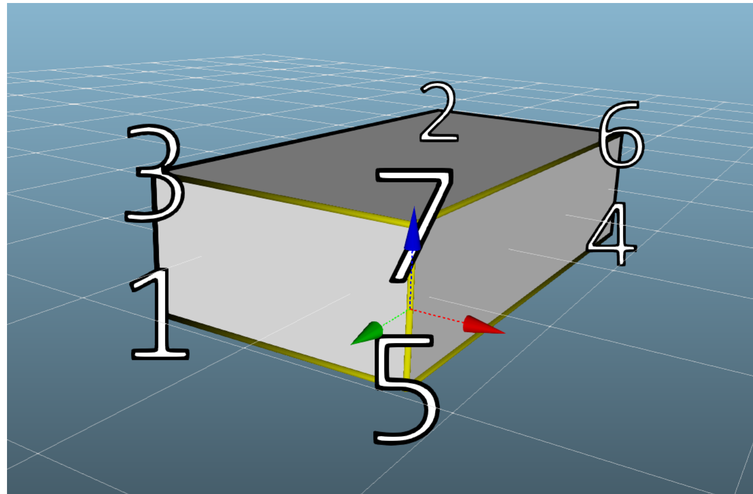


Figure 3.7.: Using mesh editing tools to create a wedge from a cube.

Vertex. Select a single vertex by rightclicking it, and add the other vertex to the selection by holding shift and rightclicking it. Then merge the two selected vertices by selecting *Merge* from the vertex menu or by pressing the shortcut M. The first selected vertex determines the position of the merged vertex. The second pair of vertices is merged analogously. The resulting edge must be subsequently translated in $+y$-direction.

The currently selected mesh can be saved to disk for future use by selecting *Save as .geom file* from the object menu. Conversely, there is an option in the scene menu for importing a mesh.

We have shown how meshes can be rapidly edited using built-in tools. Admittedly, the number of available tools is still very limited, but note that a (mesh editing) tool is simply an operator, too, so you can use the API presented in section 4 to extend Nubuck.

Once the input polyhedrons have been created, we select *Merge* from the algorithms menu. This algorithm isolates the wrapping procedure from the algorithm of Preparata and Hong and lets us carefully examine the situation. A click on *Step* advances the wrapping, and the subdivision is fine enough to observe the search for each winning vertex. The algorithm can be fast-forwarded by clicking *Next* or *Run*, each taking multiple steps at a time. Figure 3.8 shows an intermediate state of the wrapping. The red edge connecting the two polyhedrons is the pivot line around which the imaginary plane is creased. Emanating from both of its endpoints there are two more edges, red and yellow, that are used as cursors for the search for a winning vertex. The yellow edge is currently being investigated. The purple edges are the shadow edges. The wrapping faces are rendered semi-transparent, so that they do not obscure the view on the relevant edges. A detailed log output supplements the sensibly colored rendering. Clearly, a visualization like this is incredibly helpful in debugging a complicated algorithm.

Figure 3.8.: In the second example Nubuck is used to visualize a complicated 3-D algorithm. Sensible coloring, transparency, and step-by-step execution help to understand the current state of the algorithm.

4. The programming interface

In the preceding sections, two examples have been presented that proved the usefulness of Nubuck. The first example showed how the intricate relation among the Delaunay triangulation and Voronoi Diagram in 2-D, and the Convex Hull in 3-D can be communicated by interactively manipulating the scene. In the second example, we observed the runtime behavior of a 3-D algorithm for some pathological input that we were able to construct using the built-in operators. Both examples, however, focused mainly on the interactive aspects of Nubuck and assumed that the algorithms under discussion already exist. In this section, we view Nubuck from a programmer's perspective and take a look at the application programming interface (API), to see how to actually implement the algorithms. First, the necessary data structures will be discussed. Then the API will be presented by giving a full implementation of the first example. After showing some additional features of the API, we will iterate on some of the presented concepts in more detail.

4.1. Data structures

The mesh is the central entity in Nubuck and it's essential to precisely define what shapes it can represent. As we exclusively focus on combinatorial computational geometry, a mesh represents a *polyhedron*, which is the 3-D generalization of a 2-D polygon. A mesh cannot, for example, describe parametric surfaces, such as spline surfaces.

4.1.1. Polyhedrons

A polyhedron (or 3-polytope) is the generalization of a 2-D polygon to 3-D. The boundary of a polyhedron is a finite collection of planar, polygonal *faces*. Faces are joined by (one-dimensional) line segments called *edges*, and edges meet in (zero-dimensional) corner points called *vertices*. Figure 4.1 shows a selection of polyhedrons. The leftmost polyhedron is called a *tetrahedron*. It is the simplest polyhedron and consists of only four vertices, six edges, and four faces.

The faces of a polyhedron satisfy the following properties, which will be explained in that order:[1]

P1 the faces intersect properly,

[1] We use the definition of polyhedrons from O'Rourke [1998]

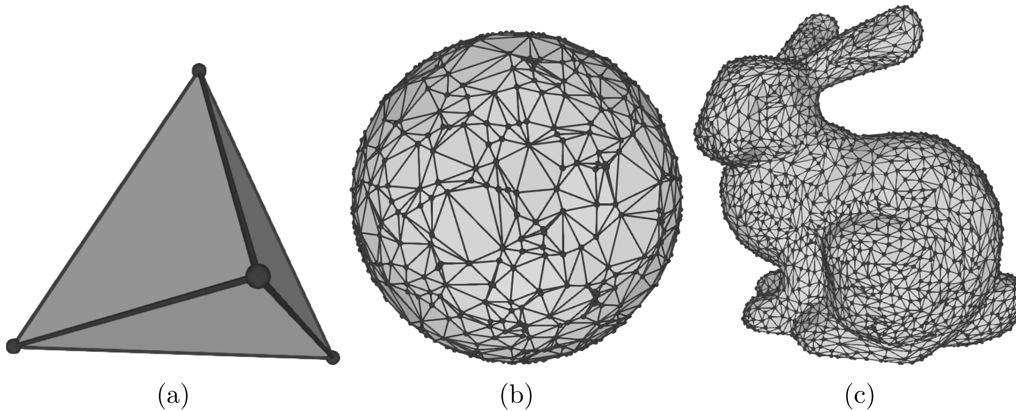


Figure 4.1.: Three example polyhedrons. (a) The tetrahedron (or simplex) is the simplest polyhedron in 3-D space and consists of only four vertices and faces, and six edges. (b) The Convex Hull of 1000 points randomly distributed on the surface of a sphere. (c) A version of the *Stanford bunny* that has been reduced to 2503 vertices and 14892 edges. Unlike the polyhedrons (a) and (b), the bunny is not a convex polyhedron.

P2 the neighborhood of every point is topologically an open disk, and

P3 the surface is connected and closed.

We adopt the convention from O'Rourke [1998] that faces are convex, so any non-convex faces are treated as if they were composed of coplanar convex faces, which simplifies **P1**.

P1. Each pair of faces of a polyhedron is either disjoint, or intersects such that they share a single vertex or two vertices and the edge connecting them.

P2. Simply put, this property enforces a proper local topology by excluding objects like the ones shown in figure 4.2. This can be expressed mathematically by requiring that the neighborhood of every point, i.e., an arbitrary small portion of its surrounding surface, is *homeomorphic* to a disk. Without going into technical details, we can imagine that the neighborhood of a point can be shaped into a disk by stretching and bending it, but without tearing.

P3. The surface of a polyhedron is connected, in the sense that there is a path on the surface between any two of its points. In particular, a polyhedron is not allowed to have isolated faces. Also, the surface of a polyedron is closed, which means that it encloses a bounded region of space. Note that our definition permits a polyhedron to have any number of holes (like the hole in a torus).

A polyhedron is said to be convex if it contains the line segment joining any pair of points in its interior or on its surface. The surface of a convex polyhedron does not intersect itself.

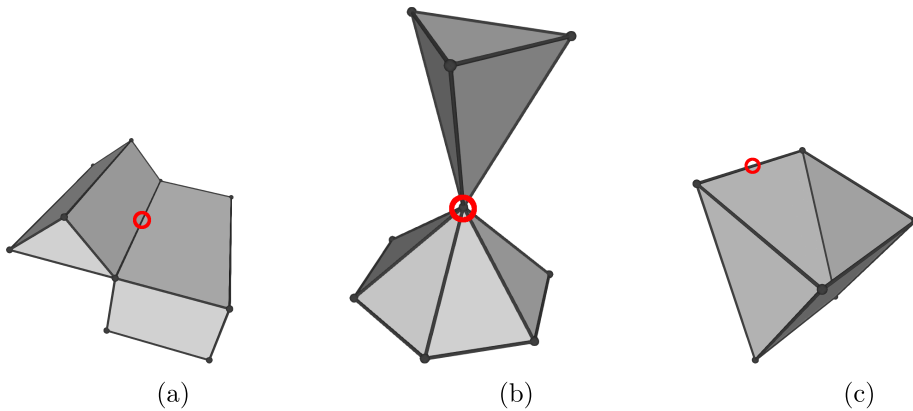


Figure 4.2.: Three meshes that are not polyhedral, because they violate property **P2**. All three examples can be found in O'Rourke [1998]. For each mesh, a red circle highlights a point whose local neighbourhood is not homeomorphic to a disk.

4.1.2. A mesh data structure

A polygonal mesh, or polyhedral boundary representation (B-rep), is a collection of vertices, edges and faces that represent the boundary of a polyhedron. Each vertex has a position. Other attributes, like color, may be stored for each type of object, as well. There exist different mesh data structures and the use of one depends on the set of operations that have to be performed efficiently.

A simple and compact mesh data structure stores the vertices implicitly and each vertex is referred to by its index. Therefore, arrays can be used as lookup tables to store the vertex attributes. The faces are stored in a list and each face is a list of its vertices, such that the vertices are traversed in counter-clockwise order when viewed from outside the mesh. Similarly, an edge stores its two vertices. The edges, however, are usually not needed when this data structure is used. This data structure is sometimes called face-vertex mesh representation.

The face-vertex mesh representation is common in rendering applications and it is used by the two major hardware graphics APIs, Direct3D and OpenGL. Because graphics hardware is tuned to maximize the throughput of triangles, these APIs only support faces with three vertices. Any polygonal face of a mesh has to be triangulated before rendering. Older versions of the OpenGL API, which are still supported by modern operating systems and drivers, do provide the primitive types `GL_QUAD` and `GL_POLYGON`. However, these are deprecated in recent versions of the API. Because each face has the same number of vertices, the face list can be flattened to a sequence of indices. With OpenGL, the user specifies the vertex attribute arrays, a sequence of indices and a primitive type. The primitive type determines what geometric objects are constructed from the indices. For triangles, the primitive type is one of `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, or `GL_TRIANGLE_FAN`. The primitive type `GL_TRIANGLES` draws a series of triangles using

the indices i_0, i_1, i_2, then i_3, i_4, i_5, and so on. The other two primitive types can reduce the average number of indices to form a triangle from three to one, if a series of triangles share vertices.

When faces are processed individually, the face-vertex mesh representation offers good performance, because arrays have zero memory overhead and good cache utilization. However, the mesh is stored as a collection of disconnected triangles and adjacency information is not readily available. In order to find the two faces that share an edge, for example, the entire face list has to be traversed. This makes the face-vertex mesh representation generally ill-suited for geometric algorithms.

A data structure for geometric algorithms should support efficient modifications and efficient queries of the following form. Traverse all vertices or edges of a face in correct order. List all faces sharing a vertex. List the two faces sharing an edge. The time complexity of these basic operations should be optimal, i.e., proportional to the output size.

One idea is to augment the face-vertex mesh representation so that a face contains pointers to its neighboring faces and to its vertices. That might be a good idea for triangulated meshes, because a triangle would always need six pointers. In general, however, a face of a polyhedron can have an arbitrary number of adjacent faces. Consequently, a face would need a variable amount of memory, which impedes cache-efficient memory management. Therefore, mesh data structures for geometric algorithms focus on edges, instead of faces, because an edge is always shared by two faces.

One of the first mesh data structures developed is the winged-edge data structure (Baumgart, 1975). It's still relevant today, because of the applications that use it. The free and open-source subdivision modeler Wings 3D, for example, is named after this data structure. Typically, an implementation of the winged-edge data structure stores its elements explicitly. That means that the elements are stored as linked lists of objects and each object contains all the attributes of that element. Addition and removal of elements is therefore straightforward. Each vertex stores a pointer to one of its edges and each face points to one of its boundary edges. For each face there exists a cyclic sequence of its boundary edges, called face cycle, such that the edges appear in counter-clockwise order, when viewed from outside the mesh. In the face cycle of a particular face, an edge e has a successor and a predecessor. Also, because each edge is shared by two faces, an edge has two successors and two predecessors in total. We call these four adjacent edges the *wings* (of e). In the winged-edge data structure, an edge stores eight pointers. Two pointers to its vertices, two pointers to its adjacent faces, and four pointers to its wings.

The winged-edge data structure supports all relevant operations in optimal time. The next listing shows how to traverse the boundary edges of a particular face.

```
/*                                                           1
let e be of type WE_Edge and 0 <= i < 2.                     2
e.p[i] = predecessor of e with respect to f[i],              3
e.s[i] = successor of e with respect to f[i]                 4
```

```
*/
struct WE_Vertex { WE_Edge* e; };
struct WE_Face { WE_Edge* e; };

struct WE_Edge {
  WE_Vertex* v[2];
  WE_Face*  f[2];
  WE_Edge*  p[2];
  WE_Edge*  s[2];
};

void WE_traverse_boundary_edges_CCW(WE_Face* f) {
  WE_Edge* e0 = f->e;
  WE_Edge* e = e0;
  do {
    visit(e);

    if(f == e->f[0]) e = e->s[0];    // (A)
    else e = e->s[1];
  } while(e0 != e);
}
```

Although this traversal function is theoretically optimal, there is a practical issue: the condition in (A) causes branching, which in turn leads to branch mispredictions and pipeline bubbles. The branch predictors of typical CPUs do not mitigate this particular performance problem, because there is no regular branching pattern.

The *half-edge data structure* remedies this problem by splitting an edge into a pair of *half-edges* of opposite direction. A half-edge belongs to only one face. Again, we use the convention that faces must be oriented counter-clockwise. Each half-edge stores five pointers. A pointer to its source vertex (the one it emanates from), a pointer to its face, pointers to its successor and predecessor edges, and a pointer to its *reversal* edge, which is the edge in opposite direction. Note that the other vertex and face are stored in the reversal edge. Using the half-edge data structure, we can now traverse the boundary edges of a face without checking the orientation of an edge:

```
struct HE_Vertex { HE_Edge* e; };
struct HE_Face { HE_Edge* e; };

struct HE_Edge {
  HE_Vertex* v;
  HE_Face*  f;
  HE_Edge*  p;
  HE_Edge*  s;
  HE_Edge*  r; // reversal
```

```
};                                                         10
                                                           11
void HE_traverse_boundary_edges_CCW(HE_Face* f) {          12
  HE_Edge* e0 = f->e;                                      13
  HE_Edge* e = e0;                                         14
  do {                                                     15
    visit(e);                                              16
    e = e->s;                                              17
  } while(e0 != e);                                        18
}                                                          19
```

As it turns out, the half-edge data structure looks pretty much like a directed graph. In fact, the LEDA library does not provide a special polyhedron data type, but uses its graph data type `leda::graph` instead. LEDA explicitly stores the reversal $rev(e)$ of an edge, which can be accessed in $O(1)$ with `leda::graph::reversal`. Thus, for all practical purposes, we view the LEDA graph as an implementation of the half edge data structure.

There is one caveat, though. The LEDA graph is a general purpose graph data structure and does not implement the special concept of a successor and predecessor edge, as required by the half edge data structure. Instead, we impose a counter clockwise ordering of the edges around a vertex, which allows us to efficiently obtain these edges.

LEDA uses the *adjacency list* representation of a graph $G = (V, E)$; that is, each vertex $v \in V$ stores a doubly linked, cyclic list $adj_edges(v) = \{e = (v, w) \in E\}$ of all edges adjacent to v. When adding an edge to a `leda::graph`, it's possible to specify its position relative to another edge. Therefore, a counter clockwise ordering can be realized by maintaining the order of $adj_edges(v)$ accordingly.

To be precise: The edge $e_s = (u, w)$ is the successor of $e = (u, v)$ in counter clockwise order (when viewed from outside the polyhedron), if e_s is the successor of e in $adj_edges(u)$. In this case we simply call e_s the successor of e (in u) and write $e_s = succ(e)$. Likewise, we define the predecessor $pred(e)$ of e (in u). The successor and predecessor of an edge `e` of graph `leda::graph G` can be accessed in $O(1)$ using the methods `G.cyclic_adj_pred(e)` and `G.cyclic_adj_succ(e)`, respectively.

Next, we show how the ordering can be used to access the next edge in a face cycle. Let $face_succ : E \to E$, $face_succ(e) = pred(rev(e))$. For $e' = face_succ(e)$ the following properties hold: $src(e') = dst(e)$ and $left(e') = left(e)$, where $left(e)$ denotes the face left of the directed edge e. Using $face_succ$, all edges of a face can be traversed in counter clockwise order. A detailed description of $face_succ$ can be found in Mehlhorn and Näher [1999].

Nubuck does not use the LEDA graph directly, but provides its own data type `NB::Graph`. It is derived from `leda::GRAPH<leda::d3_rat_point, int>`, which in turn is derived from `leda::graph`. LEDA's parameterized `leda::GRAPH<TV, TE>` is a graph whose

nodes and edges store additional data of type `TV` and `TE`, respectively. [2] Naturally, we store the position vectors of the vertices. The decision for the rational vector type `leda::d3_rat_point` is justified in the next section. In order to be compatible with existing LEDA algorithms, we adhere to the convention `TE = int`. Additional attributes, like color and size, are stored in private maps of type `leda::node_map` and `leda::edge_map`. The `NB::Graph` type has a „dirty" flag, which is set and cleared by the system, and indicates that the owning mesh needs rebuilding.

As opposed to `leda::d3_window`, which only traverses face cycles, Nubuck stores per-face data, so the faces have to be explicitly computed. Some methods of `NB::Graph` update the faces implicitly. If a client directly manipulates the nodes and edges of a mesh, however, the faces must be updated with a call to `NB::Graph::compute_faces()`. The same applies when a `NB::Graph` is passed to a method that is oblivious of faces and takes as argument an object of type `leda::GRAPH<leda::d3_rat_point, int>`, like `leda::CONVEX_HULL`. We choose not to compute faces automatically, so that clients can deliberately leave faces in a partially constructed state.

When recreating simple polyhedrons that are easily given by a list of vertices and face indices, like cubes, it is often tedious to manually add the edges in the correct order, so a function `leda::make_from_indices` is provided that converts the face-vertex representation to a graph.

The `NB::Graph` data structure can also be exploited to represent degenerated meshes, i.e., meshes that violate one or more of the properties **P{1, 2, 3}**. A single face `f` can be hidden with `NB::Graph::set_visible(f, b)`, where the second argument is a boolean. Furthermore, a *masked* edge is skipped when traversing its face cycle (`NB::Graph::set_masked(e)`). A polygon can be represented by a polyhedron with zero volume, although in this case the backface must be hidden to avoid rendering artifacts caused by coincident faces.

4.1.3. Rational numbers

Dealing with limited number precision is an intrinsic difficulty of implementing geometric algorithms. Most current machines implement the IEEE-754 standard for single precision (32bit) and double precision (64bit) floating point numbers. Since they have a fixed number of bits, floating point numbers are subject to rounding errors, which can influence the correctness of an algorithm. A frequently used *geometric predicate* in 2-D, for example, is `orient(a, b, c)`, which tests whether the point `c` lies left of, right of, or on the directed line through `a` and `b`. In Berg et al. [2008] it is shown how rounding errors in the evaluation of this predicate lead to the violation of geometric invariants and the corruption of a convex hull algorithm. To make matters worse, the result of a floating point computation might vary across different machines, due to different compilers and CPU features (x87, SSE). In some circumstances integer numbers suffice for the coordinates of the input points. Also, many geometric predicates, like `orient`, can

[2] `http://www.algorithmic-solutions.info/leda_manual/bgraph.html`

be cleverly expressed as the computation of a determinant, which reduces to a number of additions, subtractions, and multiplications — but does not need division. In these cases arithmetics can be carried out with only integers, which do not suffer from precision errors. However, integers are still vulnerable to overflows and silent „wrapping”, which might go unnoticed in an implementation. In general, using built-in number types of fixed size, it is difficult to write *robust* algorithms, i.e., algorithms that are correct across all machines and inputs. Alternatively, exact arithmetic can be achieved using rational numbers of arbitrary precision. Arbitrary precision numbers, also called *bignums*, usually store their digits in arrays of variable length on the heap, which makes them orders of magnitude slower than built-in floating point numbers. Also, it can be shown that, simply put, two random numbers are most likely coprime, so cancellation is no panacea for producing smaller numbers. [3]

For the purpose of Nubuck it is entirely reasonable to accept the performance penalty of exact arithmetic in favor of eliminating precision problems altogether. Hence, the type of the vertex positions of a `NB::Graph` is `leda::d3_rat_point`.

4.2. Implementation of Example 1

Now that we understand the central mesh data structure, we turn our attention to the implementation of an operator. We will give the full implementation of example 1 from section 3.1, thereby presenting the main features of the Nubuck API.

4.2.1. Notation

For this presentation, we adopt the notations of *literate programming*, which is a programming paradigm initially proposed by Donald Knuth. A practitioner of literate programming writes her program as a bilingual *web* file, which contains both the documentation and the source code. A tool extracts the documentation and source code from the web in a process called *weaving* and *tangling* respectively. The original tool written by Knuth is called *WEB* and uses TEX as its documentation language and Pascal as its source code language. However, the concrete language choices are unimportant; the point is that a web file combines documentation written in a natural language (or markup language) with the source code, so that they stay consistent. Now, in contrast to documentation generators like *Doxygen* or *JavaDoc*, literate programming emphasizes the *logical structure* of a program. A literate program is composed of fragments, which are named sections of code. Fragments can reference other fragments. The fragments can be reordered in a logical manner, so that it's easy for a human reader to understand. Also, literate programming encourages comprehensive documentation of high-level concepts that might not be trivially evident from the code itself. The tangler extracts all fragments and resolves all references, i.e., subsitutes fragments, so that a

[3] http://en.wikipedia.org/wiki/Greatest_common_divisor

syntactically correct program is handed to the compiler. Notable large-scale applications written with literal programming are TEX and the pbrt raytracing engine (Pharr and Humphreys [2010]). Also, LEDA uses literate programming to produce manual pages, implementation reports, and the book Mehlhorn and Näher [1999].

Nubuck is not implemented with literate programming, but we adopt the notion of fragments to structure the following code presentation. A fragment definition looks like this:

fragment compute primes ≡

```
⟨include header files⟩
⟨global variables⟩
int main() {
  ⟨compute primes⟩
  ⟨print primes⟩
  return 0;
}
```

We use angle-brackets to reference other fragments, which can be defined later:

fragment include header files ≡

```
#include <stdio.h> // we need printf()
```

The definition of a fragment does not have to list the entire source code. Once its name is defined, code can be added to the end of a fragment using the following syntax (note the +=):

fragment include header files +=

```
#include <math.h> // we also need sqrt()
```

4.2.2. Basic Operator Structure

Now we begin with the implementation of example 1. The following listing shows the overall structure of the application:

fragment vdh_example.cpp ≡

```
#include <Nubuck\nb_common.h>

class VDH_Panel : public NB::OperatorPanel {   // (A)
private:
  ⟨private panel variables⟩
public:
  VDH_Panel();
};
```

```
                                                              9
(B) class VDH_Operator : public NB::Operator {               10
    private:                                                 11
      ⟨private operator variables⟩                           12
      ⟨private operator methods⟩                             13
    public:                                                  14
      void Register(Invoker& invoker) override;              15
      bool Invoke() override;                                16
      void Finish() override;                                17
    };                                                       18
                                                             19
    int main(int argc, char* argv) {                         20
(C)   NB::Init(argc, argv);                                  21
      NB::SetOperator(new VDH_Operator);                     22
      NB::SetPanel(new VDH_Panel);                           23
      return NB::Run();                                      24
    }                                                        25
```

Each application must provide exactly one operator that derives from `NB::Operator` (B), and at most one operator panel that derives from `NB::OperatorPanel` (A). If no panel is provided, the system creates a default panel that only displays the name of the operator.

Note that an operator and its panel are conceptually separated, because they run in different threads. This way, parts of the system stay responsive, even while the operator is busy. As a consequence, though, you must not store pointers between these two objects, or share data in any form, as this might lead to race conditions. Later on, we will learn how the operator and the panel can talk to each other using the *event system*, which is Nubuck's implementation of the *message passing* concept.

All of the work will be done in the operator and panel. In the function `main`, which is the entrance point of the application, we just initialize the library and launch the sandbox with our operator (C). The calls to `NB::SetOperator` and `NB::SetPanel` may be in any order relative to each other, but they must be enclosed by calls to `NB::Init` and `NB::Run`, as shown. Nubuck imposes the restriction of one operator per application, so if either of the functions `NB::SetOperator` or `NB::SetPanel` is called more than once, the previous argument will be discarded. The function `NB::Run` launches the sandbox and returns an exit code, when its main window is closed.

An operator must implement the abstract methods `Register`, `Invoke`, and `Finish`. Note that we chose to use the Microsoft specific `override` keyword, because it makes it obvious, which methods are part of the base class.

The `Register` method is the first method called on any operator and it typically creates the menu item. Its implementation is as follows.

fragment vdh_example.cpp +=

```
void VDH_Operator::Register(Invoker& invoker) {
  NB::AddMenuItem(NB::AlgorithmMenu(), "VDH Demo",  ↩
   invoker);
}
```

The method body consists of just one line, which is pretty much self-evident. Recall that Nubuck does not distinguish between different types of operators, so we are free to add the menu item to any menu. The four available menus are `NB::SceneMenu`, `NB::ObjectMenu`, `NB::VertexMenu`, and `NB::AlgorithmMenu`. In our case, of course, we decide that the operator should be listed in the algorithms menu.

For the time being, the argument of type `Invoker` is being treated as a black-box and just passed along. We will come back to it later, when we talk about custom Qt code. In short, other widgets, like buttons, can signal the *invoker* to call the operator, just like menu items.

The method `Finish` gets called whenever the operator is being asked to terminate, so it contains cleanup code. Note that the currently active operator may be preempted at any time. For now, the `Finish` method does nothing, but we add to it as we go:

fragment vdh_example.cpp +=

```
void VDH_Operator::Finish() {
  ⟨finish body⟩
}
```

The method `Invoke` is more interesting. As the name suggests, it gets called whenever the user selects the operator's menu item. It starts as follows:

fragment vdh_example.cpp +=

```
bool VDH_Operator::Invoke() {
  NB::Mesh inputMesh = FirstSelectedMesh();
  if(!inputMesh) {
    NB::LogPrintf("no input mesh selected.\n");
    return false;    // (A)
  }

  ⟨invoke body⟩

  return true;
}
```

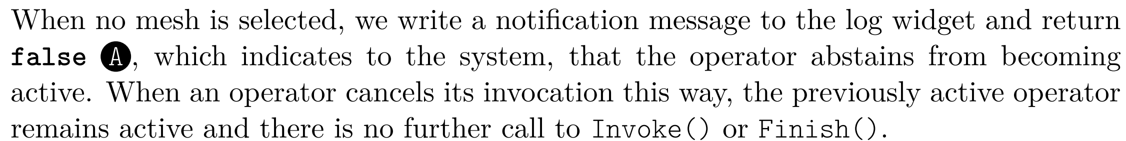
At this point, it's necessary to review the semantics of our algorithm. We use the first selected mesh entity as input. The vertices of the input mesh are the vertices of the

Delaunay Triangulation, and define the Voronoi regions. If no mesh is selected, we want to notify the user. Also, we project all input vertices on the *xy*-plane.

Recall that the user can shift-click to select multiple meshes. The function `NB::Mesh NB::FirstSelectedMesh()` returns the first selected mesh, or `NULL`, if there is none. The next selected mesh can be obtained by passing a selected mesh to `NB::Mesh NB::NextSelectedMesh(NB::Mesh)`. Again, a value of `NULL` is returned, if no such mesh exists. This way, we can iterate over all selected meshes.

The world in Nubuck is heterogeneous, and so is the selection. That means, the user can select multiple entities of different type, which sometimes can be useful. The *transform* operator, for example, translates and scales a mixed selection of mesh and text entities. The two functions `NB::Enitity NB::FirstSelectedEntity()` and `NB::Entity NB::NextSelectedEntity(NB::Entity)`, which return the generic base type `NB::Entity`, can be used to uniformly iterate over the entire selection.

The concrete type of an entity can be queried with `NB::GetType(NB::Entity)`, which is one of `ET_MESH`, `ET_TEXT`, or `ET_TRANSFORM_GIZMO`. There is also a family of functions that perform downcasts, e.g., `NB::CastToMesh(NB::Entity)`.



When no mesh is selected, we write a notification message to the log widget and return **`false`** (A), which indicates to the system, that the operator abstains from becoming active. When an operator cancels its invocation this way, the previously active operator remains active and there is no further call to `Invoke()` or `Finish()`.

Once it has been verified that the operator has been called properly, we proceed by copying the input mesh.

Operators like this, which leave the original inputs intact, are called *non-destructive* and have the advantage that any inputs that produce interesting results are preserved by default. Otherwise, the user would have to copy inputs precautionally.

The copy of the input mesh is stored in a member variable of the operator. Since we need to copy only the vertices, actually, we name the variable accordingly:

fragment private operator variables ≡

```
    NB::Mesh _verticesMesh;
```

By convention, identifiers of private member variables have a leading underscore. The body of `Invoke` continues as follows.

fragment invoke body ≡

```
NB::HideMesh(inputMesh);

// create mesh for projected vertices
_verticesMesh = NB::CreateMesh();
NB::SetMeshName(_verticesMesh, "Projected Vertices");
NB::SetMeshRenderMode(_verticesMesh, NB::RM_VERTICES);

```

```
// copy input mesh, project all vertices onto XY plane                8
const NB::Graph& inputGraph = NB::GetGraph(inputMesh);                 9
NB::Graph verticesGraph;                                               10
leda::node v, w;                                                       11
forall_nodes(v, inputGraph) {                                          12
  w = verticesGraph.new_node();                                        13
  verticesGraph.set_position(w, ProjectXY(inputGraph. ↩               14
   position_of(v)));
}                                                                      15
NB::SetGraph(_verticesMesh, verticesGraph);                            16
```

First, we hide the input mesh, so that it's out of the way when the user moves the other vertices around. A hidden mesh is not rendered, but still appears in the outliner widget.

Next we create the vertices mesh. We give the newly created mesh a descriptive name using the function `NB::SetMeshName`, so that it can be easily identified in the outliner. Also, we have to set the rendermode, which determines what elements of the mesh are rendered. The integer argument passed to `NB::SetRenderMode(NB::Mesh, int)` is a bitwise `OR` of the following flags: `RM_VERTICES`, `RM_EDGES`, `RM_FACES`, or `RM_ALL`. The default rendermode is `0`, which means that the mesh is invisible. Unlike a hidden mesh, an invisible mesh is still selectable. The rendermode of a mesh can be changed at any time in the outliner.

After that, we copy the input mesh. More precisely, we copy the vertices of the graph of the input mesh. Recall that every mesh owns a graph, and that the graph stores the geometry and topology of the mesh. We access the graph of a mesh using the function `NB::Graph& NB::GetGraph(NB::Mesh)`. Conversely, we set the graph using the function `NB::SetGraph(NB::Mesh, const NB::Graph&)`. Note that `NB::SetGraph` makes a copy of the graph, so you might want to consider manipulating the reference returned by `NB::GetGraph` instead. `NB::Graph` derives from `leda::GRAPH`, so the actual copy loop uses only familiar LEDA code and is straightforward. As mentioned earlier, the position of a vertex is stored in the graph, and can be changed and read with `NB::Graph::set_position` and `NB::Graph::position_of`, respectively.

The function `ProjectXY` belongs to the operator implementation and it is not part of the Nubuck API. As the name suggests, it simply discards the z-coordinate of it's argument, and we omit its implementation for the sake of brevity.

Next, we create the meshes for all the other geometric structures that are computed from the input points, namely the Delaunay Triangulation, the Voronoi Diagram and the 3-D Convex Hull. We call these meshes the *result meshes*. The paraboloid is an ordinary mesh, too, so we add four meshes, in total. To minimize the number of code fragments, we use this opportunity to introduce all of the remaining operator variables.

fragment operator private members +=

```
  NB::Mesh _delaunayMesh;                                              1
```

```
    NB::Mesh _voronoiMesh;                                    2
    NB::Mesh _hullMesh;                                       3
    NB::Mesh _paraboloidMesh;                                 4
                                                              5
    leda::node_array<R::Color> _siteColors;                   6
```

The method Invoke continues as follows:

fragment invoke body +=

```
    // create mesh for projected delaunay triangulation          1
    _delaunayMesh = NB::CreateMesh();                            2
    NB::SetMeshName(_delaunayMesh, "Delaunay ↩                   3
     Triangulation");
    NB::SetMeshRenderMode(_delaunayMesh, NB::RM_NODES | ↩        4
     NB::RM_EDGES);
    NB::HideMesh(_delaunayMesh);                                 5
                                                                 6
    // create mesh for projected voronoi diagram                 7
    _voronoiMesh = NB::CreateMesh();                             8
    NB::SetMeshName(_voronoiMesh, "Voronoi Diagram");            9
    NB::SetMeshRenderMode(_voronoiMesh, NB::RM_ALL);             10
    NB::HideMesh(_voronoiMesh);                                  11
                                                                 12
    // create mesh for 3d convex hull                            13
    _hullMesh = NB::CreateMesh();                                14
    NB::SetMeshName(_hullMesh, "Convex Hull");                   15
    NB::SetMeshRenderMode(_hullMesh, renderAll);                 16
(A) NB::SetMeshPosition(_hullMesh, M::Vector3(0.0f, 0.0f, ↩      17
      2.0f));
    NB::HideMesh(_hullMesh);                                     18
                                                                 19
    // create paraboloid mesh                                    20
    _paraboloidMesh = NB::CreateMesh();                          21
    NB::SetMeshRenderMode(_paraboloidMesh, NB::RM_FACES);        22
    NB::SetMeshName(_paraboloidMesh, "Paraboloid");              23
(B) NB::SetMeshPosition(_paraboloidMesh, M::Vector3(0.0f, ↩      24
      0.0f, 2.0f));
    NB::HideMesh(_paraboloidMesh);                               25
```

Note that the newly created meshes are hidden, but we will add GUI controls that allow the user to toggle the visibility of the meshes.

In (A) and (B), we move the convex hull and the paraboloid slightly away from the xy-plane, to reduce clutter near the origin, where the z-coordinates vanish. The function

`NB::SetMeshPosition` sets the *world space* position of a mesh, and does not affect the vertex positions, which live in *local space*.

We illustrate the difference with an example. Let's consider a unit cube placed at the origin, with vertex coordinates in $\{-1, 1\}$. The performance of geometric algorithms generally increases, when the computations involve only small numbers. Remember that Nubuck uses rational numbers to store vertex positions, so we are talking about *bignum* arithmetic. Now assume the cube is moved away from the origin, because we want to, say, arrange it with a couple of other entities in the world. If this translation is applied to the vertices directly, a geometric algorithm would suffer from the distorted numbers, with respect to its performance. Nubuck therefore stores the vertex positions *relative* to the mesh's position.

Some geometric algorithms, however, operate on two or more meshes. Merge algorithms, for example, might expect a left and a right mesh as input. In this case, the vertex positions of the meshes must be expressed in the same coordinate system. The function `NB::ApplyTransformation(NB::Mesh)` converts the vertex positions to world space and sets the mesh's position to the origin.

At the moment, the world is populated by empty meshes, so we continue by constructing the actual graphs.

To keep the presentation short, we leave out the paraboloid graph, as its creation covers no additional information. It's sufficient to say that the paraboloid graph is a regular plane grid, which vertices have been lifted according to $z = x^2 + y^2$. The paraboloid is *degenerated*, in the sense that is has *zero volume*. Although they are topologically sound, degenerated meshes can cause rendering artifacts and we come back to this issue later, when we talk about the Voronoi Diagram.

Before building the result graphs, i.e., the Convex Hull, Delaunay Triangulation, and Voronoi Diagram, we have to do some more initialization. Every time the user moves the input vertices with the mouse, we update the result meshes in a *brute-force* fashion, by simply recomputing everything from scratch. The Voronoi cells, however, must retain their color between updates. Because every Voronoi cell corresponds to an input vertex, we can store their colors in the node array `_siteColors`.

fragment invoke body +=

```
_siteColors.init(verticesGraph);                1
leda::node v;                                   2
forall_nodes(v, verticesGraph) {                3
  _siteColors[v] = RandomColor();               4
}                                               5
```

We just initialize the node array and randomly generate a color for each vertex. We try to generate somewhat harmonic colors by mixing random, light colors (rgb in [0.5, 1]) with a constant, predefined color.

Finally, a call to the operator method `Update()` creates the graphs of the result meshes:

fragment invoke body +=

```
  Update();
```

Its declaration and implementation follow.

fragment private operator methods ≡

```
  void Update();
```

fragment vdh_example.cpp +=

```
void VDH_Operator::Update() {
  NB::Graph& verticesGraph = NB::GetGraph(_verticesMesh ↩
   );

  ConvexHull3D(verticesGraph, NB::GetGraph(_hullMesh));

  Delaunay2D(verticesGraph, NB::GetGraph(_delaunayMesh) ↩
   );

  Voronoi2D(verticesGraph, NB::GetGraph(_voronoiMesh), ↩
   _siteColors);
}
```

Simply enough, `Update()` just calls some other functions, so let's have a look at the function `ConvexHull3D`:

fragment vdh_example.cpp +=

```
static void ConvexHull3D(const NB::Graph& verticesGraph ↩
   , NB::Graph& hullGraph) {
  leda::list<leda::d3_rat_point> L;

  leda::node v;
  forall_nodes(v, verticesGraph) {
    L.push(ProjectOnParaboloid(verticesGraph. ↩
   position_of(v)));                              // (A)
  }

  hullGraph.clear();
  leda::CONVEX_HULL(L, hullGraph);
  hullGraph.compute_faces();                      // (B)
}
```

First, we create a list of points from the projections of the input vertices on the paraboloid. The function `ProjectOnParaboloid`, (A), takes as parameter a point (x, y, z) and returns

the lifted point $(x, y, x^2 + y^2)$. Then we simply use the appropriate LEDA function to compute the Convex Hull. Note that in Nubuck, in order to see the faces, we have to compute them explicitly (B). That is different from `leda::d3_window`, which uses only the face cycles of a graph.

The implementation of the function `Delaunay2D` is similar, although some complexity is added by the fact that we have to convert from 3-D to 2-D, and back.

fragment vdh_example.cpp +=

```
typedef leda::rat_point       point2_t;                           // (A)
typedef leda::GRAPH<point2_t, int> graph2_t;

void Delaunay2D(const NB::Graph& in, NB::Graph& out) {
  leda::list<point2_t> L(ToPointList2(in));                       // (B)
  graph2_t G2;
  leda::DELAUNAY_TRIANG(L, G2);

  out.clear();
  FromProjection(G2, out);                                        // (C)
}
```

LEDA's `leda::DELAUNAY_TRIANG` can be used to compute the Delaunay Triangulation in the plane, but it's a 2-D algorithm, so, using the typedef names (A), we have to convert the input vertex positions from `NB::Point3` to `point2_t` (B), and the result from `graph2_t` to `NB::Graph` (C). To no surprise, the function `FromProjection` creates a copy of the 2-D graph, and assigns z-coordinates of 0. Care is taken to preserve the edge ordering around the vertices. Note that we do not compute the faces of the Delaunay Triangulation, as we do not draw them, anyway.

Finally, we attend to the Voronoi Diagram. Its construction is a little more complicated than the previous two structures, so, in favor of a shorter presentation, we omit the implementation of the function `Voronoi2D`, and give a brief description instead.

The function `Voronoi2D` takes three arguments. The first argument is the input of type `NB::Graph`, which vertices are the *sites* of the Voronoi Diagram. The second argument is a node array, which assigns each input vertex - and therefore each site and its face - a color. The third argument is a reference to the output graph.

Again, we create a list of 2-D points from the input vertices. Then, we use LEDA's `leda::VORONOI` function to compute a plane map of the 2-D Voronoi Diagram. Note that we cannot pass the result to `FromProjection` directly, for two reason.

Firstly, the vertex data is not, as you might expect, of type `point2_t`, but of type `leda::rat_circle`. Each proper vertex `v` of the Voronoi Diagram `VD` is labeled with a circle `VD[v] = leda::rat_circle(a, b, c)`, where `a, b, c` are three distinct *sites* (input vertices), with the same distance to `v`. Recall that such points exist for every Voronoi vertex. The vertex `v` lies in the center of its circle, which is returned by

`VD[v].center()`.

Secondly, the Voronoi Diagram is *unbounded*, and for every unbounded edge, the result graph contains a vertex at *infinity*. The circle associated with a vertex placed at infinity is degenerated and has no meaningful center.

Therefore, we creates the *bounded* (or *boxed*) Voronoi Diagram. A bounded Voronoi Diagram is the planar graph obtained by intersecting a Voronoi Diagram with a sufficiently large rectangle (i.e., the rectangle contains all sites).

Analogously to `Delaunay2D`, we use the function `FromProjection` to turn the 2-D bounded Voronoi Diagram into a 3-D graph of type `NB::Graph`. We are interested in rendering the faces of the Voronoi Diagram, so we have call `NB::Graph::compute_faces()`, as well.

Then we assign the face colors, by looking up the color of their sites in the node array. The color of a face can be set with `G.set_color(leda::face f, const R::Color& c)`.

This concludes the part of the example that deals with mesh creation. We have shown how to create meshes and graphs, how they relate to each other, and how to change their various attributes.

4.2.3. Direct Input

Now, we will have a look at input handling. We distinguish between two forms of user input: Direct input comprises key presses and mouse clicks in the renderview. GUI input, on the other hand, is generated by widgets in the operator panel, such as buttons and scrollbars.

We start by using direct input handling to enable the user to interactively move around the input vertices. As the user is dragging the vertices, the result meshes are updated continuously.

Aside from some exceptions, like the 2-D Convex Hull, geometric structures become more complicated as the number of input points increases, and often the contribution of one individual input point is not obvious. However, if one point moves, while all other points are fixed, then the local changes in the structures make the influence of that particular point immediately apparent. Interactivity is therefore a useful feature of visualization.

First, we must be able to detect mouse inputs. To this end, we override the operator method `OnMouse`:

fragment public operator methods +=

```
void OnMouse(const EV::MouseEvent& event) override;    1
```

When the mouse pointer is inside the renderview widget, then any movement or click generates a mouse event. With the exception of left clicks, each mouse event is propagated to the currently active operator and can be processed in the `OnMouse` method.

The left mouse button is reserved for the system and such events do not propagate. The camera, for example, is controlled with the left mouse button and stays responsive, even when the operator is blocking with some computation. This also allows the examination of the scene in the erroneous case of an infinite loop. In practice, the missing left click is hardly a limitation and can easily be replaced by a modified right click, e.g., using the shift key.

The argument of `OnMouse` is a structure that contains the type of the event, the pressed button and modifier keys, and the coordinates of the mouse pointer in window coordinates. The event type is one of `MOUSE_UP`, `MOUSE_DOWN`, `MOUSE_MOVE`, and `MOUSE_WHEEL`.

Once we have detected a right click, we have to determine which vertex, if any, is positioned under the mouse pointer. This problem is called *picking* and involves the following operations: Find a ray, in world space, emanating from the camera (or eye) position and intersecting the viewing plane, such that its projection coincides with the mouse pointer coordinates. Test for intersection between the ray and every vertex. The most appropriate bounding volume for a vertex is, in our case, a sphere.

Also, a transform gizmo must be provided for the user, the dragging state must be managed, and so on. (Recall that a transform gizmo is a set of pairwise perpendicular arrows, and it's used to translate something in the world.)

Picking requires some linear algebra, which the user can, in theory, implement on her own, and a transform gizmo can be created with the function `NB::CreateTransformGizmo()`. An easier way, however, is to use one of Nubuck's *editors*: An editor is an object that allows the transformation of an entity or a group of entities in a unified fashion. In our case, we use the previously defined member variable `_vertexEditor` of type `NB::VertexEditor`. A *vertex editor* allows the user to select one or more vertices of a mesh, and to translate and scale them. The transform operator uses the same type of editor, so all the familiar controls apply, e.g., press G to grab.

A vertex editor manages its own selection, which is independent of the global selection that can be queried with `NB::FirstSelected*`. It also has its own transform gizmo.

A vertex editor must be attached to a particular mesh. In Nubuck terms, the editor is *open* on that mesh and we call that mesh the *subject* of the editor. Multiple editors can be open on a mesh, but editors of the same type conflict over the position of objects, so make sure that at most one vertex editor is open on each mesh at the same time.

We initialize the vertex editor at the end of the `Invoke` function, after the meshes have been created:

fragment invoke body +=

```
_vertexEditor.SetAxisFlags(NB::AF_X | NB::AF_Y);    1
_vertexEditor.Open(_verticesMesh);                  2
```

A vertex editor has three degrees of freedom by default, but the input vertices are supposed to have z-coordinates of 0. So, before it is opened, we call `SetAxisFlags(int)`

to restrict the actions of the editor to the xy-plane.

An editor is no special concept, but merely a convenience class that encapsulates picking and transformation. That means that the Nubuck system is not aware of editors and no mouse events are propagated to editors automatically. Therefore, we have to forward mouse events manually:

fragment vdh_example.cpp +=

```
void VDH_Operator::OnMouse(const EV::MouseEvent& event) ↩
    {
  if(_vertexEditor.HandleMouseEvent(event)) {
    Update();
    event.Accept();   // (A)
  }
}
```

The method `NB::VertexEditor::HandleMouseEvent` takes as parameter a mouse event and returns true, if the editor *accepts* this event. A mouse event gets accepted, if the editor interprets it as meaningful input and changes its state accordingly. For example: A click on a vertex selects that vertex, so the corresponding `MOUSE_DOWN` event gets accepted. If the user clicks on the transform gizmo, it initiates the dragging of the selected vertices, and such events get accepted, too. Also, any mouse movements get accepted while the vertices are being dragged. To put it differently, clicks that miss the vertices and the transform gizmo get not accepted by the vertex editor.

Events accepted by the editor should not be assigned additional functions by the operator, as this would confuse the user.

In case the method `NB::VertexEditor::HandleMouseEvent` returns true, the vertex positions of the editor's subject may have changed, so we call `Update()` to recompute the result meshes.

In (A), we accept the mouse event. Note that this is independent of the event's acceptance by the editor. By calling the method `EV::Event::Accept()`, we tell the system that we consumed the event and it should not be propagated any further.

Analogously, we could enable the editor hot keys by forwarding key events:

fragment vdh_example.cpp +=

```
void VDH_Operator::OnKey(const EV::KeyEvent& event) {
  if(_vertexEditor.HandleKeyEvent(event)) {
    Update();
    event.Accept();
  }
}
```

A key event contains its type, which is `KEY_UP` or `KEY_DOWN`, its key code, pressed modifier keys, and a flag, which indicates whether this key event comes from an auto-repeating

key. A key is initially pressed, if it's not auto-repeating. As a further example, we demonstrate how a key press could be used to toggle the visibility of a mesh:

fragment toggle visibility ≡

```
void VDH_Operator::OnKey(const EV::KeyEvent& event) {
  if('d' == event.keyCode && !event.autoRepeat) {
    bool isVisible = NB::IsMeshVisible(_delaunayMesh);
    NB::SetMeshVisible(_delaunayMesh, !isVisible);
    event.Accept();
  }
}
```

In our exemplary operator we recompute all geometric structures from scratch, every time the input changes. Using a simple brute-force approach, we can focus the discussion on the important material, namely the introduction of the Nubuck API. The operator at hand has been tested on a Intel Core i5-2410M, 2.30 GHz CPU, which is fast enough for the scene being updated without noticeable stutter for reasonably small numbers of input points. In a real-world application, however, you might want to use more sophisticated data structures, which perform *incremental* updates on the geometric structures. Incremental updates exploit the fact that only a small part of the geometric structure actually changes when most input points remain fixed.

In the following, assume that we have an object `incVoronoi` of type `IncVoronoi`, which has a reference to the graphs of the input mesh and the Voronoi mesh (variables `_verticesMesh` and `_voronoiMesh`). Furthermore, the type `IncVoronoi` has a method `Update(leda::node v)`, which takes as argument a node of the input graph that changed its position, and updates the Voronoi graph in an incremental fashion. To use this method, we must query which vertex has been moved by the editor:

fragment incremental update ≡

```
void VDH_Operator::OnMouse(const EV::MouseEvent& event)
{
  NB::VertexEditor& edt = _vertexEditor;
  if(edt.HandleMouseEvent(event)) {
(A)  if(NB::VertexEditor::DRAGGING == edt.Reason()) {
(B)    leda::node v = edt.FirstSelectedVertex();
      incVoronoi.Update(v);
    }
    event.Accept();
  }
}
```

First of all, in (A), we ask why the editor accepted the mouse event. The possible return values of `NB::VertexEditor::Reason()` include `SELECT`, `BEGIN_DRAGGING`,

END_DRAGGING, and DRAGGING. The latter indicates that a vertex has changed its position, in which case we can obtain the first selected vertex (B). Recall that a vertex editor manages its own selection and you can iterate over all selected vertices using the methods FirstSelectedVertex() and NextSelectedVertex(leda::node).

As the method IncVoronoi::Update(leda::node v) cannot handle more than one changed vertex at a time, we set the maximal size of the editor selection to conform to that limitation: _vertexEditor.SetSelectionSize(1).

4.2.4. Operator Panel

In this section, we will concentrate on the creation of the graphical user interface (GUI) for the operator. We will add buttons that toggle the visibility of the meshes, so that the user can focus on certain aspects of the scene. The paraboloid and the Convex Hull can become unwieldy at times, because their z-coordinates grow quadratically, so another button will be added that scales these two meshes on the z-axis.

In conformance with most GUI toolkits, we use the term *widget* to denote any clickable user interface control, such as buttons, checkboxes, comboboxes, and so on.

Widgets must be organized in *layouts*: Layouts set the size and position of widgets, and rearrange them automatically when the available amount of space changes (e.g., when the window is resized), so that no absolute positions have to be specified. There exist different layouts, such as box-layouts and grid-layouts, and elaborate interfaces can be achieved by nesting layouts.

Recall that the entire user interface of the operator is constrained to the operator panel, which is a separate object and appears as a single frame in the sandbox. Hence, in this section, we will extend the class VDH_Panel, which has been ignored until now. The class VDH_Panel, which derives from NB::OperatorPanel, has been declared in the first fragment, next to the class VDH_Operator.

We start by adding member variables for the buttons. In addition, each button must be assigned a unique integral identifier. As the concrete values of these identifiers do not matter, as long as they are distinct, we can use a simple enumeration to do this. We group the buttons together in an array, instead of having a different variable for each button, so that we can write the following listings more succinctly. For readers who are unfamiliar with C++ enumerations, we made the enumerator values explicit, although it's redundant.

fragment private panel variables ≡

```
enum {                          1
  ID_DELAUNAY = 0,              2
  ID_VORONOI = 1,               3
  ID_CONVEX_HULL = 2,           4
  ID_PARABOLOID = 3,            5
                                6
```

```
  ID_SCALE = 4                                  7
};                                              8
                                                9
NB::Button _toggleButtons[4];                   10
NB::SpinBox _scale;                             11
```

Then, the widgets are created in the constructor of the panel:

fragment vdh_example.cpp +=

```
void VDH_Panel::VDH_Panel() {                                   1
  // create buttons                                             2
  const char* buttonCaptions[] = {                              3
    "Toggle Delaunay Triangulation",                            4
    "Toggle Voronoi Diagram",                                   5
    "Toggle Convex Hull",                                       6
    "Toggle Paraboloid"                                         7
  };                                                            8
  for(int i = 0; i < 4; ++i) {                                  9
    _toggleButtons[i] = NB::CreateButton(                       10
      i, buttonCaptions[i]);                                    11
  }                                                             12
                                                                13
  // create spinbox                                             14
  _scale = NB::CreateSpinBox(ID_SCALE, "z-scale");              15
  NB::ShowSpinBoxProgressBar(_scale, true);                     16
  NB::SetSpinBoxMaximum(_scale, 1000);                          17
  NB::SetSpinBoxMinimum(_scale, 1);                             18
  NB::SetSpinBoxSingleStep(_scale, 10);                         19
  NB::SetSpinBoxValue(_scale, 1000);                            20
                                                                21
  ⟨create layout⟩                                               22
}                                                               23
```

The buttons and the spinbox are created with `NB::CreateButton()` and `NB::CreateSpinBox()`, respectively. Both functions take as arguments the unique identifier and the text of the widget they create. A spinbox has some additional properties, including its value. All properties of a spinbox can be changed with the `NB::SetSpinBox*` family of functions. The single step of a spinbox specifies the increment of its value when the user clicks on the little arrows on the left and right hand side of the widget. If a spinbox has both a minimum and a maximum value, we call that spinbox *bounded*. A bounded spinbox can display a progress bar, which shows the fraction $(value - minValue)/(maxValue - minValue)$.

After the widgets have been created, they must be put in a layout:

fragment create layout +=

```
// create layout                                          1
NB::BoxLayout vbox = NB::CreateVerticalBoxLayout();       2
                                                          3
// add widgets to layout                                  4
for(int i = 0; < 4; ++i) {                                5
  NB::AddWidgetToBox(vbox, NB::CastToWidget( ↩           6
    _toggleButtons[i]));
}                                                         7
NB::AddWidgetToBox(vbox, NB::CastToWidget(_scale));       8
                                                          9
// set layout as active layout of panel                   10
SetLayout(vbox);                                          11
```

First, we create a vertical box layout, which lines up its widgets vertically, such that the result is a single column of widgets of uniform width. Alternatively, we could use the other subclass of box layout to create a row of widgets. Then, we add the buttons and the spinbox to the newly created layout. The function `NB::AddWidgetToBox` takes as arguments the box layout and an object of type `NB::Widget`. In the same way that all entities derive from `NB::Entity`, the class `NB::Widget` is the superclass of all widgets, including `NB::Button` and `NB::SpinBox`. Therefore, there is only a single function to add arbitrary widgets to the layout. However, the flip side is having to call `NB::CastToWidget` first.

At this point, the panel contains orderly laid-out, albeit useless, buttons. In the following, we will concentrate on reacting to GUI events, such as button presses and spinbox changes.

Whenever a user interacts with a widget, that widget *emits* (or generates) an *event*. A simple button, for example, emits events of type „button clicked". Likewise, there are events that represent state changes of togglebuttons, value changes of spinboxes and comboboxes, and so on.

The notion of events is shared by most GUI toolkits, but there are different approaches to integrate the execution of client code, i.e., the code that *handles* a particular GUI event. Older toolkits, like the Win32 API or SDL, require the application code to contain a *message loop* that explicitly queries an event queue for new events. Inside the message loop, there is usually a large switch-statement that calls the appropriate function based on the event's type. Moreover, events are typically not type safe in these systems, and it is the burden of the client code to retrieve the relevant arguments from a large, aggregate structure, or to perform a cast to the correct type.

More recent toolkits use *callbacks* to directly bind a piece of code to an event. [4] In the C language, callbacks can be implemented with function pointers. Object-oriented implementations commonly use the *observer* design pattern (Gamma et al. [1995]) (sometimes

[4] `http://en.wikipedia.org/wiki/Callback_%28computer_programming%29`

called listener pattern) to pass callback objects instead.

The Qt toolkit introduced a *signal–slot* mechanism for the communication between objects: [5] A signal is emitted when an event occurs, and slots are methods that can be *connected* to signals. The system is type-safe, and objects communicating with signals and slots are *loosely coupled*, i.e., the signaling object is unaware of any connected slots. Qt's signal–slot mechanism has a very clean syntax, but requires a *meta-object compiler* (moc) to parse the sources and generate additional C++ source files. The idea of signals and slots has since been implemented by other libraries, as well. [6]

Although Nubuck uses the Qt library internally, its signal–slot mechanism is not reflected in the Nubuck API. Instead, a custom event system is used for the communication between the operator and the operator panel, for the following reasons: Firstly, Nubuck's event system is used to properly synchronize the operator and its panel, which run in different threads. Secondly, the GUI of a panel can be build without linking against the Qt framework. Lastly, Nubuck's event system uses only C++ language features, and does not need an additional meta-compiler.

Next, we use the event system to toggle the visibility of the Delaunay Triangulation, whenever the user clicks the corresponding button.

An *event definition* is a distinct object of the parameterized type `ConcreteEventDef<T>`, which binds a unique *event identifier* (or type) to a *parameter type* `T`. Every parameter type is derived from `EV::Event`. The same parameter type may be used by multiple event definitions, but the parameter type of an event is fixed. An event definition can be thought of as a blueprint for events: At runtime, instances of its parameter type are created that have the matching event identifier.

When the user clicks on the „Toggle Delaunay Triangulation" checkbox, for example, an event of type `ev_checkBoxToggled` is emitted. Its parameter type is `EV::Arg<bool>`, whose boolean attribute `value` says whether the button is checked or not.

In the operator, the method `AddEventHandler` can be called to associate a particular event with an *event handler*. An event handler is a method whose signature matches the parameter type of the event it receives. The same event handler may be used for different events that have the same parameter type.

Since its parameter type is `EV::Arg<bool>`, an event handler for `ev_checkBoxToggled` can be added to the class `VDH_Operator` like this:

```
void VDH_Operator::Event_ToggleDelaunay(const EV::Arg<
    bool>& event) {
  const bool isChecked = event.value;
  NB::SetMeshVisibility(_delaunayMesh, isChecked);
}

VDH_Operator::VDH_Operator() {
```

[5] http://doc.qt.io/qt-4.8/signalsandslots.html
[6] http://en.wikipedia.org/wiki/Signals_and_slots

```
    AddEventHandler(ev_checkBoxToggled, this, & ↩
     VDH_Operator::Event_ToggleDelaunay, ID_DELAUNAY);   // (A)
}
```

A widget tags every event it emits with its unique integral identifier, and event handlers can filter for events from a particular widget. In (A), the last argument to `AddEventHandler` tells the system that we only want to listen for events from the widget with identifier `ID_DELAUNAY`, which is the checkbox that toggles the Delaunay Triangulation.

If the last argument is omitted, the event handler is executed for every event of the specified type, disregarding its origin. The identifier of the sending widget, however, is stored in the event's field `id1` (`id0` is the identifier of the event definition), which can be used to identify the sender if only a single event handler is used:

```
void VDH_Operator::Event_OnCheckBox(const EV::Arg<bool ↩
   >& event) {
  const bool isChecked = event.value;
  switch(event.id1 /* widget ID */) {
    case ID_DELAUNAY:
      NB::SetMeshVisibility(_delaunayMesh, isChecked);
      break;
    ...
  }
}

VDH_Operator::VDH_Operator() {
  AddEventHandler(ev_checkBoxToggled, this, & ↩
   VDH_Operator::Event_OnCheckBox);
}
```

Parameter types may be arbitrary structures that store more than argument.

4.2.5. Animations

Animations are generally useful to show the transition from one state to another; for example, if vertices travel over a brief period of time, it's easy to appreciate how their new and old positions are related. Instantaneous changes, on the other hand, might confuse a viewer if many vertices are involved. Also, animated attributes, such as pulsing colors, can be used to attract the attention of the viewer to a certain object.

Since the ongoing example operator does not lend itself to being significantly improved by animations, we discuss them separately.

At first glance, Nubuck's way of starting animations may seem unnecessarily verbose and complicated, so we begin by addressing the limitations that we tried to avoid of the much simpler interface of LEDA's GraphWin.

LEDA's GraphWin displays graphs. Its methods `set_position(node v, point p)` and `set_position(node_array<point> dst)` set the position of a single node and all nodes, respectively. The translation of nodes is animated, but animations are subject to the following constraints:

- All animations have the exact same duration. The duration can be changed with the member function `set_animation_steps`, but it is constant for all animations. Regarding the second function, which moves several (or all) vertices at a time, this means that vertices traveling a greater distance have a higher velocity. That might be irritating when we think about all vertices sharing the same physical properties. Instead, vertices should have a constant acceleration, or at least a constant velocity, which means the animation must wait for the last vertex to arrive.
- The animation does a simple *lerp* (linear interpolation) between the source and destination positions. It is not possible to change the blending operator. Again, this might be irritating because lerping is an artificial form of movement.
- At any given time, at most one property is animated. It is not possible, for example, to change the position and color of a vertex simultaneously.

The Nubuck function `A::SetVertexPosition` is a convenience function which mimics the behavior of `set_position` and eases the transition from LEDA's API. The next listing shows its implementation in terms of Nubuck's animation system:

```
void SetVertexPosition(                          1
  const NB::Mesh subject,                        2
  const leda::node vertex,                       3
  const leda::d3_rat_point& position,            4
  const float duration)                          5
{                                                6
  BlendVertexPositionAnimation anim;             7
  anim.Init(subject, vertex, position);          8
  anim.SetDuration(duration);                    9
  anim.PlayFor(duration);                        10
  NB::WaitForAnimations();                       11
}                                                12
```

The first thing to point out is that an animation is represented by an object. Here, the type `A::BlendVertexPositionAnimation` is used, which translates one more vertices of the same mesh over some period of time. Each animation type derives from the base class `A::Animation`. When instantiated, an animation object automatically adds itself to a list of active animations. Likewise, it automatically removes itself from that list upon destruction.

Typically, an `Init`-method must be called to setup the essential parameters of an animation. Here, we pass the mesh, the vertex we like to animate, and its target position. Next, we set the duration of the animation. The type of movement can be set with the

method SetType, which can be one of the constants TYPE_FIXED_T, TYPE_FIXED_V, and TYPE_FIXED_A. The first type, which is the default, moves the vertex to its target in a fixed amount of time. The other two types move the vertex with a constant velocity or acceleration.

An animation is initially suspended and can be played by calling one of the following three methods inherited by A::Animation: PlayFor, PlayUntilIsDone, and PlayUntil.

The first method, PlayFor(**float** duration), plays the animation for a constant duration. Note that the interpolation factors are derived from the argument of SetDuration, but the actual animation time is the argument of PlayFor. Typically, the same duration is passed to both methods, but a shorter or longer duration for PlayFor can be used to intermit or stall the animation, respectively.

The method PlayUntilIsDone() plays an animation until it signals its own completion. In our case, the two methods PlayFor and PlayUntilIsDone can be used interchangeably, because the movement type is TYPE_FIXED_T and the vertices are guaranteed to reach their target positions on time. Otherwise, however, the animation needs to wait for the last vertex to arrive, which makes it necessary to use PlayUntilIsDone. Beware that some periodic animations never signal completion and run indefinitely when started with PlayUntilIsDone.

The other method, PlayUntil(eventFilter_t filter), plays an animation until a specific event occurs. It takes as argument a function pointer of type eventFilter_t, which is a **typedef** for **bool** (*eventFilter_t)(**const** EV::Event& event). The animation stops when the filter returns **true**. This method can be used, for example, to highlight a vertex at the end of one step of an algorithm until the user clicks on a „next" button or presses enter.

Now, an important point to realize is that playing animations run only when the operator is idling. Typically, the operator waits for all playing animations to finish by calling the function NB::WaitForAnimations(). It is a blocking function call, i.e., the operator thread waits at this point before it executes the next statement. An operator may make an arbitrary number of calls to NB::WaitForAnimations().

In summary, animations are used as follows: Create and initialize a set of animation objects, (2) play them, and (3) wait for all of them to finish.

Since each animation is an object, they can be easily composed and the number of possible combinations does not pose a problem. For example, it is trivial to let a vertex change its color while it's moving:

```
BlendVertexPositionAnimation pos;              1
pos.Init(subject, vertex, position);           2
pos.SetDuration(duration);                     3
pos.PlayFor(duration);                         4
                                               5
BlendVertexColorAnimation col;                 6
col.Init(subject, vertex, color);              7
```

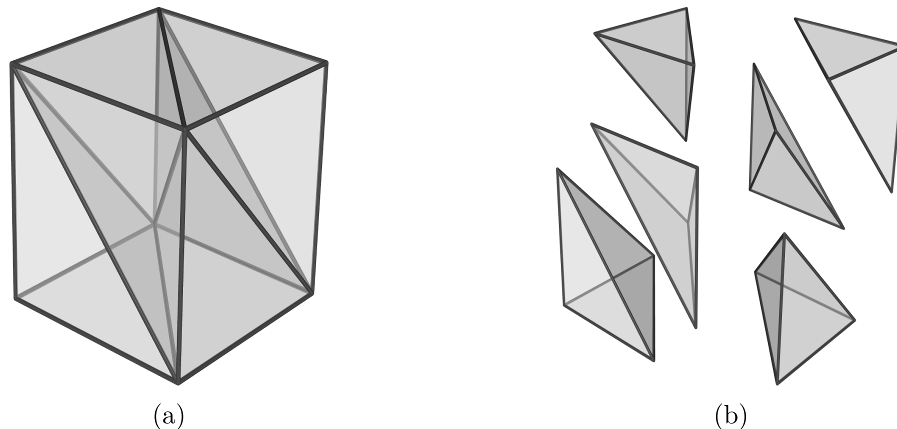


Figure 4.3.: Tetrahedralization (3-D triangulation) of a cube. The tetrahedrons (or simplices) are moved apart, such that it looks like the cube explodes.

```
pos.SetDuration(duration);           8
pos.PlayFor(duration);               9
                                     10
NB::WaitForAnimations();             11
```

Custom animations can be defined by subclassing `A::Animation`. As a practical example, we show the class `MoveSimplicesAnimation` that animates the explosion of the 3-D Delaunay Triangulation (DT). Nubuck comes with a visualization of an algorithm for the construction of the 3-D DT. Between steps, the tightly packed simplices (or tetrahedrons) of the DT are moved apart to give a better view. The simplices are moved in a way that resembles an explosion, because they are pushed away from the origin of the DT. Figure 4.3 gives the idea. The algorithm organizes the simplices in a specialized triangulation data structure named *(simplex) complex*, from which a `NB::Graph` is generated. The complex already has a method `set_scale(double s)` that moves the simplices, so the easiest way is to let an animation vary the argument `s` over time. Listing 4.1 shows the implementation of the custom animation `MoveSimplicesAnimation`.

Every subclass of `A::Animation` must override the method `Animate()`, which advances the animation one frame. Here, its implementation is simple enough. We compute an interpolated scaling value `s` and pass it to the appropriate method of the complex `C` (A). In this particular case, we have to explicitly flag the mesh for rebuilding in the next line. Note that the interpolated value depends on the elapsed animation time, and is therefore framerate-independent. In other words, the animation runs at the same pace on any machine. The elapsed time is accumulated in the variable `time`, which is increased in every frame. The method `A::Animation::GetSecsPassed()` returns the number of seconds passed since the last frame (B). The method `Animate` returns **`true`**, if the animation has completed. In our case, the animation simply ends after some set duration.

Listing 4.1: A custom animation that moves the simplices of a simplex complex.

```
struct MoveSimplicesAnimation : A::Animation {
public:
  Globals&        g;
  double          v0, v1;
  float           time;
  float           duration;

  MoveSimplicesAnimation(Globals& g, double from, ↩
   double to, float duration);
protected:
  bool Animate() override {
    double l = M::Min(1.0f, time / duration);
    double s = (1.0f - l) * v0 + l * v1;
    g.C.set_scale(s);                              // A
    NB::GetGraph(g.trGeom).force_rebuild();
    time += GetSecsPassed();                       // B
    return duration <= time;
  }
};
```

5. The rendering system

In this chapter we discuss details of the Nubuck renderer, which is the most prominent part of the visualization system. The renderer generates a 2-D image from a list of rendering jobs. All entities are submitted to the renderer to produce an image that is then displayed by the render-view widget. First, we give a brief introduction to rendering and the OpenGL API. Then we give an overview of the general Nubuck rendering architecture. The rest of the section dissects the various rendering techniques used by Nubuck.

5.1. Introduction to rendering

Rendering is the process of generating a 2-D image from a 3-D scene, as seen by a virtual camera or eye.

In animated movies or games, the goal is typically to achieve photorealism. The most important aspect of photorealism, besides high quality geometry and animations, is realistic lighting. Therefore, a photorealistic renderer attempts to solve the *rendering equation*, which is the standard lighting model in computer graphics. Simply put, the rendering equation gives the radiance leaving at any point by integrating the incoming radiance over all directions in a hemisphere. Radiance is a measure of radiometry for the power of light per unit area. Solving the rendering equation analytically is generally impossible, so a rendering algorithm has to approximate the solution. The rendering equation incorporates a class of functions called *bidirectional reflectance distribution functions* (BRDF), which model how light interacts with a given material.

The physical basis for computer graphics is geometrical optics, which describes light propagation in terms of rays of photons. Certain optical phenomena, like polarization, which are explained by the wave model, are generally ignored in computer graphics. When a photon hits matter, it gets either absorbed or scattered. In our case, we are only interested in the interaction of light and our model surfaces. We assume the space between objects to be void of any matter and ignore atmospheric effects, like fog. Absorption happens inside matter and reduces the amount of light by converting it into other kinds of energy, like heat. Scattering changes the direction of light. Light can be scattered into the surface (refraction, or transmission), or out of the surface (reflection).

When a ray of light bounces off at a surface, the direction of the incident ray and the surface normal determine the direction of the reflected ray, according to the law of reflection. We call this a *specular* reflection. When a ray of light gets refracted, scattered

inside the surface, and then re-emitted out of the surface, it's a *diffuse* reflection. The direction of diffuse reflection is assumed to be distributed uniformly.

A BRDF, which is evaluated in the rendering equation, models the light-matter interaction for some kind of material. Typically, specular and diffuse reflection are computed independently. A similar function can be used to model the transmission of a material. A BRDF assumes that the outgoing radiance of a point can be computed given only the incoming radiance at that point. This is a simplification of the real-world, where light can enter a surface at some point and leave it at another point, because of subsurface-scattering. A *bidirectional surface scattering distribution function* (BSSRDF) is a more general function that describes the light transport between any two rays that hit a surface.

However, not all renderers simulate reality. Non-photorealistic renderers produce stylized images for artistic purposes or to highlight certain aspects. CAD software, for example, often shows wireframes to show the primitives that make up an object. Software used in manufacturing often shows the silhouettes, boundaries and creases of an object to better communicate its shape.

We distinguish between *offline* and *realtime* renderers. Offline renderers may spend an arbitrary amount of time and thousands of CPU cores to produce a single image for movies or print magazines. Games, on the other hand, must run on the single GPU of a typical desktop machine and have to run at 30 to 60 frames per second (fps) to feel responsive and fluid. Rendering at 60 fps means that each frame takes about 16ms.

In general, rendering algorithms fall into two categories: raytracing and rasterization.

Classical (Whitted-style) raytracing computes the color of a pixel in the image plane, by casting a ray through that pixel into the scene. If the ray hits an object, the surface color of that object at the point of intersection determines the color of the pixel. The surface color is computed by casting additional rays to light sources and other objects (shadow rays and reflection rays). Multiple rays can be used to mitigate aliasing issues. Casting rays is a natural way of simulating the light propagation of photons. Other lighting phenomena, like refraction, can be added by casting additional rays. By sampling random rays, raytracing can be turned into a Monte Carlo [1] algorithm and it can be shown that this is a good integrator for the rendering equation.

Raytracing spawns a tree of rays for each pixel and each ray involves ray-object collision testing for the entire scene. To accelerate collision testing, a spatial subdivision data structure, like an octree or a *k*d-tree, should be used. Because Monte Carlo raytracing is a stochastic process, it may take a while for the image to converge. Techniques like importance sampling or bidirectional tracing may be used to improve the rate of convergence. There are efforts to develop realtime raytracers (e.g., the Nvidia Optix), however, a pure raytracing solution is prohibitively slow for realtime applications, at least on current hardware.

GPUs and most realtime software renderers use an algorithm called rasterization. Ras-

[1] `http://en.wikipedia.org/wiki/Monte_Carlo_algorithm`

terization is like the raytracing algorithm, but the inner and outer loop are swapped. For each primitive, determine for each pixel on the image plane if it's occupied by the projection of that primitive.

Rasterization can be done very efficiently. In the past, the *scaline rasterization* algorithm has been used, which is in a way related to the line-segment-intersection problem from computational geometry. Modern GPUs use a hierarchically approach to rasterization.

Because rasterization does not use raycasts, realtime renderers use various tricks to simulate lighting and shadows, which are not necessarily motivated by physics, but produce adequate results. One particularly hard problem to solve with rasterization is indirect illumination. In fact, many applications use direct diffuse lighting only.

Realtime renderers implement the *graphics pipeline*. The graphics pipeline, which is shown in figure Fig. 5.1, describes the sequence of operations that are performed on a primitive in order to draw it on the screen. Each primitive is processed by the pipeline individually.

An application that implements the graphics pipeline is called a software renderer. A software renderer can be executed by any CPU. Contemporary CPUs have up to eight cores and each core clocks around 2.5 Ghz. Rasterization is inherently parallel, so renderers scale well with the number of available cores. Therefore the question emerges: why would someone spend an additional $300 on a graphical co-processor?

A typical program consists of a single stream of instructions and spends most of it's time accessing slow memory and does little actual computation. Examples are sorting, graph traversal, updating a data base, editing text and so on. Also, programs exhibit patterns of *locality of reference*, in space and time. Spatial locality suggests that storage locations are referenced, which are near each other. Think of loops and sequential array accesses, for example. Temporal locality means that a recently referenced location is likely to be referenced again in the near future. Because a program is a single stream of instructions, CPUs are designed to minimize *latency*. To this end, modern CPUs do branch prediction and out-of-order execution. Slow memory and locality of reference motivate the use of cache hierarchies to hide memory latency.

In rendering, on the other hand, a single stream of instructions is executed for many primitives. This form of parallelization is called *data parallelism*. Rendering involves lots of arithmetics. Also, a higher primitive *throughput* is more important than the latency of individual primitives. Therefore, GPUs are optimized for data-parallel throughput computation. A contemporary GPU consists of multiple (around 16) *streaming multiprocessors* (SM). Each SM has its own instruction set, scheduler, dispatcher, register file and cache, and up to 32 ALU cores for computation. In contrast to CPUs, GPUs can hide latency with computation: each SM schedules up to one thousand threads and whenever one thread blocks, e.g., because it waits for data, there are typically several other threads ready to compute. A GPU thread is very lightweight and context switching is fast, because all thread registers are available at all times. Consequently, a GPU dedicates more transistors to computation. In addition, a GPU possesses many

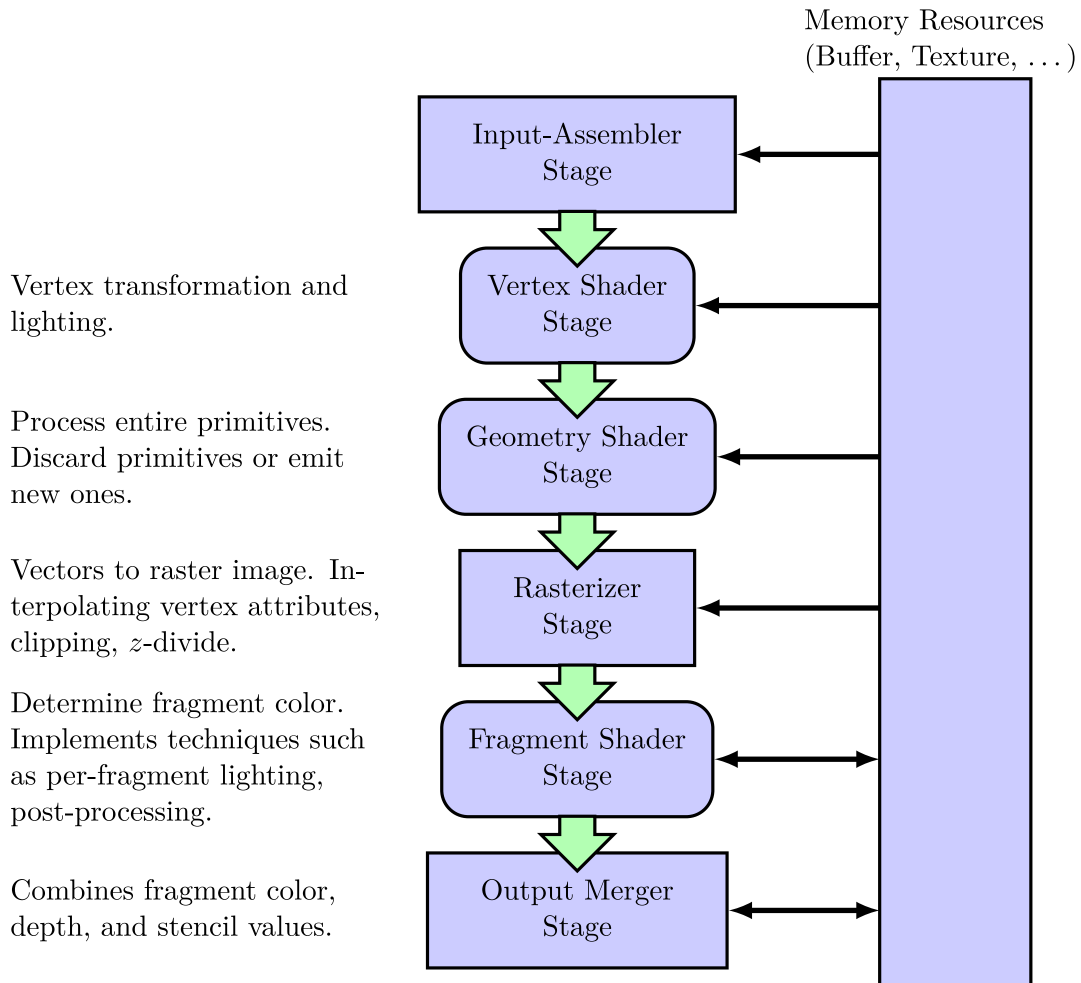


Figure 5.1.: Simplified version of diagram shown at `https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx` depicting the data flow through the rendering pipeline. Programmable stages are drawn with rounded corners.

fixed-function units to speed up tasks like primitive setup and rasterization.

For desktop computers, there exist only two relevant APIs that can be used to interface graphics hardware, namely Direct3D and OpenGL. Direct3D is available exclusively on Windows, whereas OpenGL is also supported by Linux and MacOS. Nubuck uses the OpenGL API.

5.2. OpenGL

OpenGL is a software library that interfaces the GPU in a vendor-independent way. It is a very low-level API and comprises only around 250 commands to draw simple primitives like points, lines, and triangles. More complex geometry is approximated by a set of triangles. Even round shapes, like spheres, can be approximated this way, although a large number of small triangles may be needed to give a convincingly smooth appearance. The OpenGL API specification is published by the Khronos Group. The creation and management of an OpenGL context is left to the underlying windowing system and is therefore platform-dependent. New versions are generally backwards compatible, but deprecated functions can be removed using an OpenGL context with an appropriate compatibility mode. GPU vendors may add functionality to the core API by means of *extensions*. Widely adopted extensions are usually included in new versions of the core API.

In OpenGL, there are three types of primitives: points, lines, and triangles. A set of triangles is called a *mesh*. We already discussed the face-vertex data structure, which OpenGL uses to represent meshes, in section 4.1, so we only summarize it briefly. In the face-vertex data structure, the vertices are stored implicitly, and triangular faces are given by a list of vertex indices. A *primitive type* specifies how consecutive indices form a face. Arrays can be used to store additional vertex attributes. In contrast to our polyhedron data structure, OpenGL does not need the triangles of a mesh to be connected.

In OpenGL (and Direct3D, for that matter), the rendering of a mesh works as follows. First, we set the *state*. Then, we issue a *draw call*, which renders the mesh according to the current state.

OpenGL can be thought of as a state machine. The state controls blending, culling, depth-buffer operations, and so on. The current state remains in effect until it gets changed. The state also includes *buffer* bindings and the currently bound *program*.

Buffers are GPU resources that contain vertex and index data, pixel data, texture images, uniform data, and so on. Buffers are stored in device memory, i.e., the memory of the graphics hardware. In general, the GPU memory is physically separated from the CPU main memory, and GPU and CPU share data over a PCI bus. In laptops or cell phones, however, GPU and CPU share the same memory.

A program, in the context of the OpenGL API, represents the shader stages of the

rendering pipeline. It contains the fully compiled and linked shaders.

A shader is executed on some stage of the rendering pipeline. In OpenGL, shaders are written in a high-level OpenGL Shading Language (GLSL). OpenGL defines the following five shader stages: vertex shaders, tessellation control and evaluation shaders, geometry shaders, fragment shaders, and compute shaders. The Nubuck renderer uses only vertex, fragment, and geometry shaders.

Every draw call specifies the primitive type, and vertex and index data. Note that every draw call is carried out immediately. That is different from more high-level APIs, where you describe the entire scene first.

At some point, a frame is drawn to the screen or window, but it is also possible to use *framebuffer* objects for offscreen rendering. Framebuffers are used to draw to textures, which in turn can be sampled as input in subsequent renderings to other framebuffers.

As opposed to old, fixed-function OpenGL code that uses deprecated functions, such as `glBegin` and `glEnd`, any modern, shader-based OpenGL code involves shader loading, program creation, and vertex arrays. Therefore it's no longer feasible to give a minimal, working OpenGL example, because it would easily exceed 20 lines of code, and we refer to Wright et al. [2007] instead.

5.3. Nubuck renderer architecture

In the following, we present the general architecture of the Nubuck renderer and the most important classes that form an abstraction of low-level OpenGL commands. The implementation of the renderer can be found in `%LEDA_DIR%\src\Nubuck\renderer`.

First of all, it's important to decide which OpenGL version to target, because it determines the number of features at our disposal. Nubuck should run on a Intel Core i7-620M mobile CPU, which has an integrated, first-generation Intel Graphics HD GPU. This GPU supports OpenGL version 2.1 and GLSL version 1.20, but also supports a large subset of OpenGL 3.2 core extensions, including shaders, framebuffer objects, and uniform buffer objects.

5.3.1. Class GLWidget

Before we start, we must create an OpenGL *rendering context*, which enables us to display OpenGL graphics inside a widget. For that, the Qt library provides the class `QGLWidget`. Subclasses of `QGLWidget` can override the methods `initializeGL`, `resizeGL`, and `paintGL`. Nubuck, however, uses its own `UI::GLWidget` class, which mimics the interface of `QGLWidget`, and has the following two advantages. First, it's guaranteed that all instances of `UI::GLWidget` share the same rendering context, so they trivially share resources. Therefore, the same renderer can be used to draw to multiple widgets, which is important, because there are some widgets, other than the renderview, which display

OpenGL graphics. The `UI::DirLight` widget, for example, appears in the lighting configuration panel and draws a lit sphere that indicates the direction of a light source. The second advantage of `UI::GLWidget` is that a *debugging context* can be requested, which displays valuable debugging messages and performance hints. However, a debugging context is not supported by all platforms and drivers. The rendering context is created using Window's WGL API, so at this point `UI::GLWidget` is not portable.

5.3.2. Class Renderer

The renderer is encapsulated in the class `R::Renderer`. There exists a single instance of this class, called `theRenderer`, which is used for all rendering. Note that an OpenGL rendering context is local to a particular thread. In Nubuck, the rendering thread is different from the operator driver thread, so the renderer is not exposed to the operators. A single frame can be drawn as follows:

```
theRenderer.BeginFrame();
theRenderer.Render(myRenderList0);
theRenderer.Render(myRenderList1);
...
theRenderer.EndFrame();
```

All rendering commands must be bracketed by calls to `BeginFrame` and `EndFrame`. Pairs of `BeginFrame`–`EndFrame` must not be nested. The method `Render` takes as arguments a *render list*, which specifies a transformation matrix and the directions of the three fixed light sources in the scene. It also contains a list of *mesh jobs* to be rendered. A mesh job represents all the state and buffer data that is needed to draw a mesh. In the render list of the render view widget, for example, there is at least one mesh job for each mesh entity in the scene. Depending on the desired effect, more than one mesh job may be created for each mesh entity, because the same mesh has to be drawn multiple times.

```
struct MeshJob {
  unsigned     layer;
  std::string   fx;
  tfmeshPtr_t   tfmesh;
  GLenum      primType;
  Material     material;

  // handled by renderer
  ...
};
```

The variable `layer` determines the layer a mesh job is put in. The rendering algorithm processes a list of *layers* in a certain order. A layer can be thought of as a bucket of meshes. Some layers have special properties. For instance, the triangles of all the meshes

in the layer `GEOMETRY_0_TRANSPARENT_SORTED` are rendered in a back-to-front order. The layer `GEOMETRY_0_TRANSPARENT_DEPTH_PEELING_N` is processed several times in a row, and the number of loops can be set in the rendering configuration.

The value of `primType` is usually zero. If it's not, it overrides the primitive type specified by `tfmesh`.

The other variables specify the *effect*, *(transformed) mesh*, and *material*, which together describe the entire state that is needed to draw a mesh.

5.3.3. Meshes

Admittedly, the term „mesh" is exceedingly overloaded. We distinguish between *geometric meshes* and *render meshes*. A geometric mesh has type `NB::Graph` and is the result of a geometric algorithm. A render mesh, on the other hand, has type `R::Mesh` and is drawn to the screen by the renderer. The class `R::Mesh` represents the buffer resources of the vertices and indices of a set of triangles. For each geometric mesh G a render mesh $R(G)$ is created that contains the faces of G. We also call $R(G)$ the *face mesh* (of G). There are separate render meshes for the vertices and edges of G, which will be discussed in section 5.4.

A mesh is created from a *mesh description*, which contains simple arrays of the vertex and index data. The creation of a mesh involves the object `meshMgr`, which is the single instance of the factory class `R::MeshManager`:

```
R::Mesh::Desc myMeshDesc;                                    1
myMeshDesc.vertices = { ... };                               2
myMeshDesc.numVertices = 12;                                 3
myMeshDesc.indices = { ... };                                4
myMeshDesc.numIndices = 89;                                  5
myMeshDesc.primType = GL_TRIANGLE_FAN;                       6
                                                             7
R::meshPtr_t myMesh = R::meshMgr.Create(myMeshDesc);         8
```

A vertex has type `R::Mesh::Vertex`, which simply is an aggregation of its attributes. Different effects, however, require different sets of vertex attributes: A depth-only pass needs only vertex positions, whereas a lighting pass also needs normal and color information. In Nubuck, rendering imposter edges requires the maximal number of vertex attributes, as an additional 3×4-matrix and texture coordinates are needed. The type `R::Mesh::Vertex` is large enough to contain all sets of attributes, which wastes some storage space. Alternatively, the memory footprint could be reduced by using multiple vertex formats, or by scattering the attributes among multiple vertex buffers. A single, *fat* vertex format, however, simplifies the code and also allows multiple meshes to share the same buffer, which we will talk about in a moment.

For efficient rendering, the vertex attributes and indices must be stored in GPU buffers. A buffer that stores vertex attributes is called a vertex buffer, and a buffer that stores

indices is called an element buffer. Typically, a pair of vertex and element buffer is created for each mesh. As we will see later on, when we discuss transparency, it's sometimes necessary to draw the combined set of triangles of a number of different meshes in a back-to-front order, which means that those meshes must share the same vertex buffer. Therefore, all vertex attributes of all meshes are stored in a single, large vertex buffer, called the *giant buffer*. The giant buffer virtualizes the GPU memory. A mesh can allocate and free a buffer handle in the same way that heap memory can be allocated and freed using **new** and **delete**. The giant buffer does rudimentary memory management and coalesces adjacent free buffer storage. The giant buffer starts small and grows similar to a `std::vector`. It may also purge a mesh from the buffer, if it hasn't been used in a while. A mesh is said to be *cached*, if it resides in the giant buffer. At the beginning of each frame, all meshes in the render list are cached.

A mesh accepts any primitive type, but triangle fans and triangle strips are cut into triangles, so that a list of triangles can be provided, if they need to be sorted.

The vertex positions of a mesh are specified in *local* (or *object*) space. Before making a draw call, a transformation matrix is set, that transforms the vertices of a mesh from local space to world space. When dealing with transparency, however, the triangles of the meshes must be expressed in the same space, so that they can be rendered in a single draw call. A *transformed mesh* of type `R::TFMesh` stores a reference to a `R::Mesh`, as well as a transformation matrix. In order to avoid confusion, we also call a mesh of type `R::Mesh` an untransformed mesh. If it's determined that the triangles of a transformed mesh need to be sorted, the transformed mesh makes a copy of the original vertices and transforms them to world space. A transformed mesh is created using a factory method, too:

```
R::meshPtr_t myTFMesh = R::meshMgr.Create(myMesh);
```

Multiple transformed meshes can reference the same untransformed mesh. If no sorted transparency is required, then no vertices are copied. Note that a mesh job has a handle to a transformed mesh, not to an untransformed one.

Every mesh is affected by exactly three directional light sources, whose diffuse color can be configured in the rendering options widget. The direction of each light is specified in eye space, and a black diffuse color is equivalent to disabling a light source.

By default, the lighting model uses only Lambert's term for perfectly matte diffuse reflectance. At runtime, however,, the configuration variable **`r_lightingModel`** can be set in the console window to add a BRDF model for the specular term. A value of 1 corresponds to the *Blinn-Phong* BRDF model, which lets surfaces look like plastic. A value of 2, on the other hand, produces rather metallic surfaces by using the *Cook-Torrance* BRDF model. Properties of both models, such as *shininess* and *fresnel factor*, can be tweaked with additional configuration variables. An overview of BRDF models can be found in Akenine-Moller et al. [2002].

5.3.4. Effects

An *effect* represents the shader state and other pipeline state for a mesh job. Nubuck implements a system that is somewhat similar to the effect system of Direct3D 9. A simple effect language has been devised that makes it possible to load effects from human-readable `.nfx`-files. Listing 5.1 shows an excerpt from such a file.

The declaration of every effect starts with `fx IDENT { }`, where `IDENT` denotes an identifier that adheres to the naming rules of the C language. As effects are referenced by name (string `meshJob.effect`), the specification of one is mandatory. At this time, a `.nfx`-file is restricted to exactly one `fx`-block.

An optional sorting key (A) can be used to define a certain rendering order of effects on the same rendering layer.

Inside the `fx`-block, an effect contains one or more *passes*. The mesh of a mesh job is drawn once for each pass of its effect, in the order the passes appear in the `.nfx`-file. The effect shown in listing 5.1, for example, draws the backfaces of a mesh in the first pass, before it draws the frontfaces in a second pass. On every rendering layer, adjacent mesh jobs sharing the same effect are merged, such that each pass draws the combined list of meshes.

The syntax of a pass is

```
pass IDENT {                                  1
  state { ... }                               2
  vs = STRING;                                3
  fs = STRING;                                4
  [gs = STRING;]                              5
}                                             6
```

The state controls culling, blending, alpha testing, rasterization, depth-, stencil-, and color buffer operations, and so on. The Nubuck effect language offers a simple nesting syntax to set variables of the same category. The following two states are identical, but the code on the right hand side avoids the repetition of qualifiers.

fully qualified

```
state {
 color.maskEnabled.red = GL_FALSE;
 color.maskEnabled.green = GL_FALSE;
 color.maskEnabled.blue = GL_FALSE;
}
```

group syntax

```
state {
 color.maskEnabled {
  red = GL_FALSE;
  green = GL_FALSE;
  blue = GL_FALSE;
 }
}
```

However, fully qualified names are preferred if the effect sets only a single variable of a specific category.

Listing 5.1: An example of an effect file.

```
fx LitDirectionalTransparent {
 sortkey = 1;

 pass Backfaces {
  state {
   depth.maskEnabled = GL_FALSE;
   culling.hw {
    enabled  = GL_TRUE;
    cullFace = GL_FRONT;
   }
   blend {
    enabled  = GL_TRUE;
    srcFactor = GL_SRC_ALPHA;
    dstFactor = GL_ONE_MINUS_SRC_ALPHA;
   }
  }

  vs = "#include <Shaders\mesh.vert>";

  fs =
  """
  const bool LIGHTING_ENABLED       = true;
  const bool LIGHTING_TWOSIDED_ENABLED  = true;
  const bool PREMULT_ALPHA        = false;
  const bool PERFORM_DEPTH_TEST      = false;
  const bool PERFORM_DEPTH_PEEL      = false;

  #include <Shaders\mesh.frag>
  """;
 }

 pass Frontfaces {
   // ...
 }
}
```

The effect variables `{v, f, g}s` are assigned the shader sources. Every effect is required to provide both a vertex and fragment shader (variable `vs` resp. `fs`), but the use of a geometry shader (variable `gs`) is optional.

A shader source can be either a single-line string (delimited by `"..."`), or a multi-line string (delimited by `"""..."""`). Nubuck implements an `#include`-mechanism (B), so most shader sources reside in separate files and are shared among multiple effects. Some shaders can be configured by setting constant variables prior to including (C).

Shader sources are for the most part pure GLSL code, but there a few Nubuck-specific language extensions. One of them is the `#include`-directive, which has already been mentioned. Additionally, there are two new keywords. The `material` storage-qualifier gets replaced by the `uniform` keyword, but also flags its variable as a *material parameter*. The `attribute(n)` syntax is used as a substitute for the `layout(location = n)` layout qualifier syntax, which is not present on all platforms supported by Nubuck. Internally, the argument `n` is passed to `glBindAttribLocation`.

Next to a mesh and an effect, a mesh job also defines a *material*. A material is a key–value data structure that binds values to uniforms. A uniform variable flagged as material parameter must be set by the material, or an error is generated.

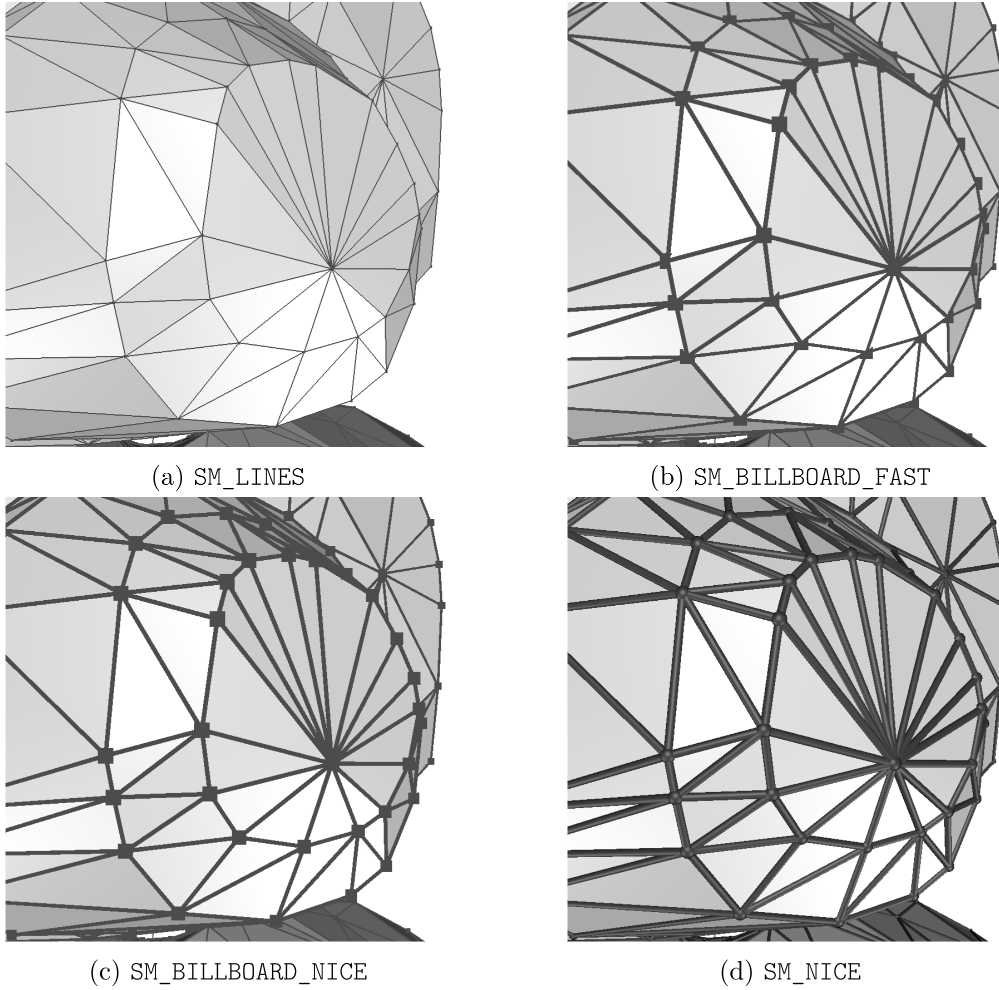
5.4. Nodes and edges

The Nubuck API function `NB::SetMeshShadingMode` sets the shading mode of a mesh, which determines how the nodes and edges are rendered. The shading mode can also be set in the outliner window. There are four shading modes — `SM_LINES`, `SM_NICE`, `SM_BILLBOARD_FAST`, and `SM_BILLBOARD_NICE` — which differ in rendering performance and appearance. Figure 5.2 shows a comparison of all four modes.

First, we briefly review the terminology used in this section. A geometric mesh has type `NB::Graph` and is the result of a geometric algorithm. A render mesh, on the other hand, has type `R::Mesh` and gets drawn to the screen by the renderer. For each geometric mesh G, a render mesh R is created for its faces. The mesh R is also called the face mesh (of G). Now, additional render meshes will be created for the nodes and edges of G. In the following, we sometimes call a vertex of G a *node* (because G really is a graph type), to differentiate it from the vertices of a render mesh.

5.4.1. SM_LINES

When the shading mode is equal to `SM_LINES`, the nodes and edges are rendered using the OpenGL primitive types `GL_POINTS` and `GL_LINES`, respectively. This is arguably the most popular method to draw the edges of a meshes, because it's fast and easy to implement. Note that in our case drawing the edges is different from a generic wireframe rendering, because multiple triangles of a face mesh can make up a single face of the respective `NB::Graph`. Thus, we build an additional point mesh and



(a) SM_LINES

(b) SM_BILLBOARD_FAST

(c) SM_BILLBOARD_NICE

(d) SM_NICE

Figure 5.2.: The four different shading modes for nodes and edges.

line mesh, instead of drawing the face mesh a second time with polygon mode set to `GL_LINES`. One problem with using line primitives is that lines and triangles are not rasterized in the same way, which means that the depth values generated for a line might differ from the depth values of a triangle edge for the same pair of vertices. Therefore, a line might have discontinuities, because some of its fragments do not pass the depth test. This visually unpleasant effect is called *stitching* and can be mitigated in various ways. The most common and fastest method is to slightly push the triangles away from the camera by adding a depth offset. Here we use the OpenGL command `glPolygonOffset(GLfloat factor, GLfloat units)`, which adds an offset of $m * factor + r * units$ that is linear in the slope m of the triangle. The offset is set once and does not change during the lifetime of the application. Values of $factor$ and *units* that work for most meshes have been found by experimentation. Adding a depth offset is no panacea, however, and stitching might occur nonetheless. Another drawback of `SM_LINES` is that the width of line primitives is part of the state (set by `glLineWidth(GLfloat width)`), so all lines of a draw call have uniform width. Often, however, it's useful to highlight certain edges of a mesh by drawing them wider. Using line primitives, this would require multiple draw calls and batching of the edges by width. Also, the maximal line width is implementation-dependent and for many implementations there exist a width threshold, above which the quality of lines drastically degrades. Moreover, the line width is specified in window coordinates, which has two implications. First, lines do not get smaller as they move away from the camera, which means that meshes look more dense in the distance. Second, for a fixed line width, lines are thinner at higher resolutions or greater window sizes. As we will see later on, this also becomes an issue when making screenshots that have a higher resolution than the window. For these reason, different edge size are ignored when in shading mode `SM_LINES`, and all lines have uniform width instead. The other three shading modes, however, support different edge sizes. The option „stylized hidden lines" in the outliner widget enables the highlighting of hidden edges by drawing them with a stippled (dashed) pattern. Figure 5.3 shows the hidden lines of a dodecahedron. The implementation is fairly easy: The hidden lines effect disables writes to the depth buffer and sets the depth comparison function to `GL_GREATER`. This way, lines are drawn only when they are behind the face mesh. The stippling effect is achieved by passing appropriate parameters to `glLineStipple`.

5.4.2. SM_BILLBOARD_FAST

The next shading mode, `SM_BILLBOARD_FAST`, does not use the primitive types `GL_POINTS` and `GL_LINES`, but renders the nodes and edges as *billboards* instead. A billboard is a quad, formed by a pair of two triangles, that always faces the camera. A billboard faces the camera when it is perpendicular to a view ray to its center. Billboards are generally an improvement over line primitives, but they also introduce new problems.

We begin with the edge billboards mesh. It contains a billboard for each edge of the geometric mesh. By using triangle fans instead of triangles, we can save one index per

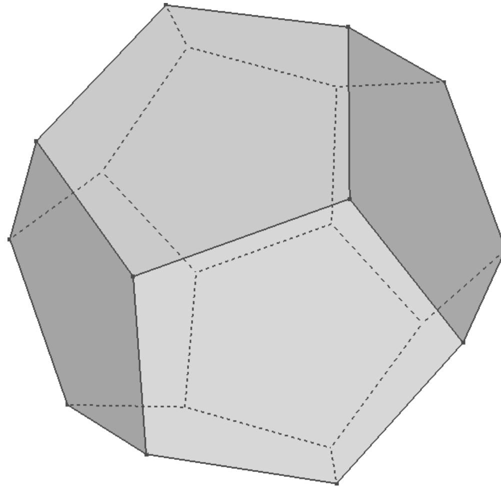


Figure 5.3.: Showing the hidden edges of a dodecahedron.

billboard (we still need a restart index), and a billboard comprises four vertices and five indices. As we will see later, a geometry shader can be used to construct a billboard from a single vertex, which reduces vertex buffer sizes. A billboard always faces the camera, so its vertices must be updated whenever the camera changes. Instead of updating the vertex buffer, however, which involves CPU arithmetics and CPU-to-GPU memory copies, we offload this work to the vertex shader. Let p_0, p_1 be the positions of the endpoints of an edge e. For each billboard, the vertex buffer contains four vertices (one for each corner of the billboard), but the position of each of these vertices is equal to p_0. Also, each vertex is assigned its billboard coordinate: We define a billboard coordinate system, where the lower left hand vertex has coordinates $(-0.5, 0)$, and the upper right hand vertex has coordinates $(0.5, 1)$. These local coordinates are stored in the 2-D texture coordinates vector of a vertex. The vector $p_1 - p_0$ is called the *axis* of e, and it's stored in the normal vector (billboard edges are unlit, so the normal vector is not used otherwise). Finally, each vertex stores the radius (or width) of an edge. Billboard edges are opaque, so we repurpose the alpha component of the vertex color for this. So, the vertex buffer stores only degenerated billboards, i.e., the vertices of each billboard coincide. Using the vertex attributes, however, the vertex shader can inflate a real billboard that faces the camera. The relevant files are `edge_line_billboard.nfx` and `edge_line_billboard.vert`.

Listing 5.2 shows the GLSL code of the vertex shader. First, we compute the axis of e, and the position of p_0 in eye space, by transforming them with the world (or model) matrix. Then, we create a set of three pairwise-perpendicular vectors that form the basis of the billboard coordinate system (A). The y-axis is the axis of edge e, and it is fixed, but we can rotate the billboard around its y-axis. We chose the x-axis, so that it is perpendicular to a view ray to p_0. Since this is eye space, the direction of a view ray is simply $-p_0$. With the axis vectors and the size of the billboard, we can create the matrix `M`, which rotates and scales vectors from billboard space to eye space. Using

Listing 5.2: Inflating an edge billboard in the vertex shader.

```
// ... uniforms, attributes, varying variables

void main() {
  vec3 axis = (uTransform * vec4(aNormal, 0.0)).xyz;

  vec3 p0 = (uTransform * aPosition).xyz;

  vec3 axisY = normalize(axis);                              // A
  vec3 axisX = normalize(cross(axisY, -p0));
  vec3 axisZ = normalize(cross(axisX, axisY));

  mat3 rotate = mat3(axisX, axisY, axisZ);

  float height = length(axis);
  float radius = aColor.a;

  mat3 scale = mat3(
    radius, 0.0, 0.0,
    0.0, height, 0.0,
    0.0, 0.0,} 1.0);
  mat3 M = rotate * scale;

  vec3 pos = p0 + M * vec3(aTexCoord0, 0.0);                 // B

  vColor = aColor.xyz;

  mat4 projection = uProjection;
  if(PROJECTION_OFFSET_ENABLED) {
    projection[2][2] *= (1.0 - PROJECTION_OFFSET);           // C
  }

  gl_Position = projection * vec4(pos, 1.0);
}
```

this matrix, it's easy to compute the eye space position of a vertex using its billboard space coordinates, which are stored in its texture coordinates (B). In C, we reduce stitching artifacts, by pulling the edge vertices towards the camera. As it turns out, we can slightly offset the z-coordinate by disturbing the projection matrix, such that the other coordinates are not affected. See Lengyel [2002].

The node billboards are created in a similar way, but we relax the orientation constraint: Instead of facing the camera, a node billboard is only parallel to the projection plane, i.e., the xy-plane in eye space. This simplifies the vertex shader, because no transformation matrix `M` is needed. Although this does make a difference for larger viewing angles, it is hardly noticeable in practice.

As the size of an edge or node is encoded in the vertex attributes, the size can vary among edges or nodes in a single draw call. This is an improvement over the fixed-size restriction of shading mode `SM_LINES`. Billboards, however, exhibit the problem of *self-intersection*: The node and edge billboards intersect the face mesh, which results in truncated quads. When the face mesh is rendered transparent, the intersecting part of a billboard appears brighter. This visually very distracting effect becomes more pronounced as the size of the nodes and edges increases. The shading mode `SM_BILLBOARDS_NICE` solves this problem, but it is computationally more expensive.

5.4.3. SM_BILLBOARD_NICE

The shading mode `SM_BILLBOARD_NICE` is similar to `SM_BILLBOARD_FAST`, but the billboards do not suffer from self-intersection. This is achieved by performing a special depth test in the fragment shader.

For each node n in the geometric mesh, the face mesh contains a group of corresponding vertices V; since the face mesh is flat-shaded, there is a vertex for each face that is incident on n. Conceptually, we can think of V as a single vertex, because all vertices in V have the same position. We call V the *spine* of n. The node billboard of n should be visible, if and only if the spine of n is visible.

Now, the idea is as follows. In the framebuffer, a vertex is visible, if it passes the depth test. Assume that `solidDepth` is a texture that contains the depth values of the face mesh. Analogously to the depth buffer, a texture containing depth values is called a *depth texture*. Furthermore, each billboard vertex stores the position of its spine s. Then, in the fragment shader, we can check the visibility of s by comparing its depth value with the value of `solidDepth` sampled at the screen space position of s. If the spine fails the depth test, we discard the fragment. Put in another way, the fragment shader reproduces the depth test performed by the fixed-function hardware to test the visibility of the spine. Note that we perform the depth test for the screen space position of the spine, which is, in general, different from the position of the fragment. The relevant files are `{node, edge}_billboard_gs.nfx` and `{node, edge}_billboard_gs.frag`.

Listing 5.3 shows the GLSL source of the node fragment shader. The depth test is per-

Listing 5.3: Performing a custom depth test to check the visibility of the spine.

```
layout(std140) uniform UniformsRenderTarget {
  int uWidth;
  int uHeight;
};

uniform sampler2D depthTex;
uniform sampler2D solidDepth;
uniform sampler2D peelDepth;

in BillboardData {
  vec3 spinePos_ss;
  vec3 color;
  vec2 texCoords;
} inData;                                                    // (A)

void main() {
  vec2 texCoords;
  texCoords.x = inData.spinePos_ss.x / uWidth;
  texCoords.y = inData.spinePos_ss.y / uHeight;

  if(1 == SHAPE && 1.0 < dot(inData.texCoords, inData.       // (B)
   texCoords)) {
    discard;
  }

  float sdepth = texture2D(solidDepth, texCoords).a;
  if(inData.spinePos_ss.z > sdepth) discard;                 // (C)

  if(PERFORM_DEPTH_PEEL) {                                   // (D)
    float pdepth = texture2D(peelDepth, texCoords).a;
    if(inData.spinePos_ss.z > pdepth) discard;

    float depth = texture2D(depthTex, texCoords).a;
    if(inData.spinePos_ss.z <= depth) discard;
  }

  gl_FragColor = vec4(inData.color, 1.0);
}
```

Listing 5.4: generating a billboard from a single vertex.

```
layout(points) in;
layout(triangle_strip) out;
layout(max_vertices = 4) out;

// ... uniforms

in VertexData {
  vec3  color;
  float  size;
} inData[];

out BillboardData {
  vec3 spinePos_ss;
  vec3 color;
  vec2 texCoords; // in [-1, 1]
} outData;

vec3 EyeToScreenSpace(vec4 v) {
  vec4 v_cs = uProjection * v;
  vec4 v_ns = v_cs / v_cs.w;
  vec3 v_ss = 0.5 * (vec3(1.0, 1.0, 1.0) + v_ns.xyz);
  v_ss.x *= uWidth;
  v_ss.y *= uHeight;
  return v_ss;
}

void main() {
  vec3 center = gl_in[0].gl_Position.xyz;

  float size = inData[0].size;

  vec3 spinePos_ss = EyeToScreenSpace(vec4(center, 1.0) ↩
   );

  gl_Position = uProjection * vec4(center + vec3(-size, ↩
    -size, 0.0), 1.0);
  outData.spinePos_ss = spinePos_ss;
  outData.color = inData[0].color;
  outData.texCoords = vec2(-1.0, -1.0);
  EmitVertex();

  // ... emit other three vertices
}
```

formed in **C**. The listing also shows how to perform a depth peel (**D**), which can be ignored for now, but will be important later, when we talk about transparency. Furthermore, in **B**, the billboard coordinates are used to discard certain fragments, so the billboard is rendered as a circle, instead of a square. In shading mode `SM_BILLBOARD_NICE`, a geometry shader is used to emit the four vertices of a billboard from a single input vertex. The geometry shader also fills in the structure `inData` to pass additional values to the fragment shader. The variable `inData.spinePos_ss` contains the screen space position of the spine, so `inData.spinPos_ss.z` is the depth value. Listing 5.4 shows the GLSL source of `node_billboard_gs.geom`. The screen space position of a point is computed in **A**. First, the point is transformed to clip space by applying the projection matrix. Dividing the point by its w-coordinate is called perspective divide and yields the point in normalized device space (NDC). Finally, a simple affine transformation gives the point in screen space. Note that this last step uses the hard-coded default values `glDepthRange`. Using a geometry shader, the size of the vertex buffer can be reduced by a factor of four, but on most hardware geometry shaders reduce the rendering performance, as well. However, we found that this does not pose a serious problem for our application.

Previous to rendering the billboards, the depth texture has to be created, which requires an additional rendering pass. To this end, we render the face meshes to a separated framebuffer, which depth attachment is the texture `solidDepth`. Consequently, the format of `solidDepth` is `GL_DEPTH_COMPOMENT`. Also, we set the texture parameters of depth textures, so that the fragment shader can read the depth value from the alpha component. The texture `solidDepth` can be sampled in subsequent renderings to other framebuffers. A framebuffer switch is costly, so the shading mode `SM_BILLBOARD_NICE` solves the self-intersection problem, but is computationally more expensive than `SM_BILLBOARD_FAST`.

5.4.4. SM_NICE

The last of the four shading modes in Nubuck is `SM_NICE`. As the name suggests, it is the visually most appealing shading mode, but costly in terms of performance. In this shading mode, the nodes and edges are drawn as fully lit spheres and cylinders, respectively. The spheres and cylinders use the same lighting model as the face meshes: they are illuminated by the three world lighting sources and also show specular highlights.

The most straight forward method to render spheres and cylinders is to model them as triangle meshes. A geometric mesh, however, may have hundreds of nodes. Also, a large number of triangles is needed for round silhouettes and nice specular highlights, although Gouraud Shading conceals this to some degree. There are two ways to draw the nodes as spheres. First, the sphere can be stored once and rendered at different positions in multiple draw calls. Each draw call, however, introduces some driver overhead, which can quickly accumulate to a significant amount. Second, a single vertex buffer containing different sphere instances can be rendered in a single draw call. However, the buffers would require a prohibitively large amount of memory. Using hardware *instancing*,

multiple instances of the same mesh can be drawn in a single draw call. An instanced draw call takes as additional argument a buffer that stores per-instance data, such as position. However, benchmarks have shown that an alternative approach is superior in terms of performance.

In shading mode `SM_NICE`, the spheres and cylinders are drawn as *impostors*, which are sophisticatedly shaded billboards. This method has been successfully used in *ball and stick* visualizations of large molecules (see Tarini et al. [2006]). In our case, a 3-D box is drawn for each edge, instead of a simple quad, but the idea remains the same: we render a simple bounding mesh and trick the eye into seeing a smooth object.

We begin with the node billboards. A node billboard is a quad that always faces the camera. As we have seen before, a billboard defines a local coordinate system. In this case, we define the lower left hand corner to have coordinates $(-1, -1)$, and the upper right hand corner to have coordinates $(1, 1)$. Let's assume the billboard bounds the sphere S that we really want to see. So, the radius r of S is equal to half the edge length of the billboard. From any viewing direction, only one half sphere is visible, so let's assume the billboard separates S in a visible half sphere and an invisible half sphere. Now, when drawing a fragment of the billboard, we want to shade it *as if* it was the fragment of the visible half sphere that lies in front of it. Shading a fragment requires the normal vector of a point, which is perpendicular to the surface at that point. The normal vector for a point p on the surface of a sphere is $p - c$, where c is the center of the sphere. The trick is to express the sphere in billboard coordinates. Let x, y be the billboard coordinates of some fragment, where $-1 \leq x, y \leq 1$. In billboard coordinates, S is normalized and has radius 1. If $||(x, y)|| < 1$, then there exist a point p on the surface of S with $xcoord(p) = x$ and $ycoord(p) = y$. As p lies on a sphere, its z-coordinate is equal to $\sqrt{1 - x^2 + y^2}$. In billboard coordinates, the center of S is 0, so the normal vector is given by $normalize(x, y, z)$. As before, we use the simplification that a node billboard is parallel to the viewing plane. Therefore, the billboard space z-axis coincides with the eye space z-axis and the normal vector does not need to be transformed any further. The GLSL source of `node_billboard.frag` is given in listing 5.5. Besides shading, we have to write the correct depth buffer values of a sphere, too, so in Ⓐ, the screen space z-coordinate of p is computed. Again, this involves projection, perspective divide, and depth range transformation. Correct depth values are crucial for proper intersections between nodes and other meshes.

In Tarini et al. [2006] an edge is represented by a quadrilateral imposter, too, but this technique could not be transfered to Nubuck, as it only works for orthographic projection. Instead, the edge cylinders are rendered by doing raycasts in the fragment shader. Cylinder edges define the maximal number of per-vertex attributes, as the edge's orientation must be stored for each vertex. Raycasting a cylinder is rather straight forward, so we omit the listing of the shader sources and refer the reader to the files `edge_billboard.vert` and `edge_billboard.frag`.

As imposters evaluate the model surfaces per-fragment, perfectly smooth spheres and cylinders can be achieved. Another advantage is that fewer fragments have to processed,

Listing 5.5: shading a sphere impostor.

```
// ... uniforms, attributes, and varying variables

void main() {
  float clip = dot(vTexCoord0, vTexCoord0);
  if(clip <= 1.0) {
    float n = sqrt(1.0 - clip);
    vec3 position = vPosition;
    position.z += vRadius * n;

    vec3 view = -normalize(position);
    vec3 normal = normalize(vec3(vTexCoord0, n));

    gl_FragColor = vec4(lighting(normal, view, vColor.rgb), 1.0);

    vec4 proj = uProjection * vec4(position, 1.0);
    gl_FragDepth = 0.5 * (1.0 + proj.z / proj.w);    // A
  }
  else discard;
}
```

as the screen size of an impostor decreases. The downside is that impostors are not subject to some anti-aliasing techniques, like MSAA, because they only work for real triangle geometry. FSAA, on the other hand, works perfectly fine and is integrated in many drivers.

5.5. Transparency

A transparent surface transmits light without scattering it. Transparency is a valuable tool for scene inspection, because a surface that blocks the view of the observer can be assigned a semi-transparent material, so that other objects behind it are visible. In the second introductory example (section 3.2), we have seen how the Convex Hull of two convex polyhedrons can be computed by gift wrapping. In the wrapping process, the two input polyhedrons become partly obscured by the newly created faces, such that for the viewer it's tough to tell why a given edge has been chosen by the algorithm to advance the wrapping. By drawing the new faces with a semi-transparent material, however, all relevant edges are visible, while it's still possible to see where the new faces are attached to the polyhedrons.

There are alternatives to transparency. For example, an interior mesh M can be made

visible in an *x-ray* rendering: After all other meshes have been drawn, the depth buffer is cleared, and M is drawn in a second pass. This way, M is drawn on top of all other meshes, but the relative depth of M is lost. Using true transparency, on the other hand, the color of an obscured fragment is a cue for the number of faces that lie in front of it. Another alternative to make interior meshes visible is to eliminate all triangles that block the view. This method is called clipping. Transparency, however, has the advantage that the shape of the surrounding geometry is preserved.

In Nubuck (and many other realtime graphics applications), a semi-transparent surface only attenuates the color of objects that lie behind it, and other real-world phenomena, like refraction, are ignored. Or, put differently: we do not care about general translucency.

One way to implement transparency is called *screen-door* transparency. The basic idea is to draw only every other pixel of a semi-transparent surface. For a surface that is half transparent, for example, a checkerboard pattern can be used, and just two diagonal pixels in a 2×2-pixel tile are drawn, so that the other two pixels show the color stored in the framebuffer. Ideally, the distance between the viewer and the monitor is so large that individual pixels are not discernible to the eye, and the screen-door pattern is not noticeable. *Dithering* applies the same idea to give the illusion of a greater color depth by mixing colors from a limited palette. Screen-door transparency can be implemented on a wide range of hardware, and has the advantage that transparent surfaces can be drawn in any order. However, this technique does not scale well, because the size of the screen-door pattern increases with the number of transparent surfaces. Also, it's hard to support different degrees of opacity. Stochastic transparency is a randomized sub-pixel version of screen-door transparency.

Usually, the color of a fragment that passes the depth test is written to the framebuffer, such that the previously stored value is overridden. When *blending* is enabled, it's possible to control how the color of the new fragment (the *source* color) is combined with the color stored in the framebuffer (the *destination* color). General transparency can be achieved with *alpha blending*: a color is a RGBA tuple, where the last component is called the *alpha channel* (or α-channel) and represents opacity. A value of $\alpha = 0$ means fully transparent and a value of $\alpha = 1$ means fully opaque. With alpha blending enabled, the source color c_s and destination color c_d are blended in the following way:

$$c_f = \alpha_s c_s + (1 - \alpha_s) c_d \tag{5.1}$$

If three fragments are alpha-blended with colors c_0, c_1, c_2, in that order, the final framebuffer color expands to

$$c_f = \alpha_2 c_2 + (1 - \alpha_2)(\alpha_1 c_1 + (1 - \alpha_1) c_0) \tag{5.2}$$

Clearly, alpha blending is noncommutative and semi-transparent fragments have to be drawn in a back-to-front order. Since frame buffer and depth buffer only store the

values of the nearest fragment, there is no way to sort all fragments with the same *xy*-coordinates by distance. It's easy to think of generalized *deep buffers* that store linked-lists of fragments, but implementations suffer from storage and synchronization problems, so deep buffers are not available on current generation graphics hardware.

Nubuck supports three transparency modes. The first two, `TM_BACKFACES_FRONTFACES` and `TM_SORTED_TRIANGLES`, approximate perfect transparency by drawing triangles in a certain order. The other mode, `TM_DEPTH_PEELING` implements perfect transparency using a multipass rendering approach.

The cheapest and fastest mode is `TM_BACKFACES_FRONTFACES`, but it produces good results only for convex polyhedrons. It's still relevant, because there are algorithms that work only with convex polyhedrons. Also, this mode is a fallback for general meshes, if the other two modes turn out to be too slow. In this case, all objects at least appear, although with noticeable errors in transparency.

The triangles of a mesh can be partitioned into front facing and back facing triangles. Because convex polyhedrons are not self-intersecting, perfect transparency can be achieved by first rendering all back faces and then rendering all front faces. Explicit sorting is not necessary, because hardware culling can be used to efficiently discard faces of a particular orientation. The relevant effect file, **`lit_directional_transparent.nfx`**, has already been shown in listing 5.1.

When rendering, meshes are sorted by the distance of their center to the camera, so this transparency mode can also be used for multiple convex polyhedrons, as long as they do not intersect each other.

Opaque and transparent meshes are put in different rendering layers, so that all opaque meshes are drawn with alpha-blending disabled, before any transparent meshes are drawn. This applies to the other two transparency modes, as well.

The mode `TM_BACKFACES_FRONTFACES` is computationally cheap, but it produces poor images for non-convex meshes or interpenetrating meshes. The second transparency mode, `TM_SORTED_TRIANGLES`, gives better results by sorting individual triangles. To that end, the combined set of all triangles of all transparent meshes in the scene must be sorted and rendered in one draw call. This is made possible by three implementation decisions presented earlier:

- all vertices of all meshes share the same vertex buffer (the giant buffer).
- meshes convert mesh descriptions with primitive type `GL_TRIANGLE_FAN` or `GL_TRIANGLE_STRIP` to `GL_TRIANGLES`.
- transformed meshes may create a copy of transformed vertices.

With `TM_SORTED_TRIANGLES` enabled, rendering works as follows. Gather the triangles of all transparent meshes (opaque meshes are still drawn in a previous pass). Sort the triangles and generate a list of sorted indices. Issue a draw call. Since we use a giant buffer, it is possible to get by with a single call to `glDrawElements`, but unfortunately we have to stream the indices from CPU to GPU memory in every frame.

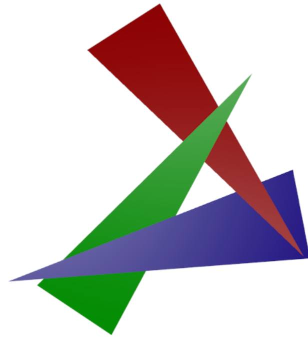


Figure 5.4.: An arrangement of three triangles for which no consistent „in-front-of" relation can be defined.

The idea is to sort the triangles back-to-front, and the first two comparison operators that come to mind are comparing the centroids of two triangles and comparing their nearest vertices (i.e., the vertices closest to the camera). Both operators seem to work equally well in practice, and Nubuck uses the former one. However, the sorted triangles approach is intrinsically flawed because some configurations cannot be consistently ordered in a „in-front-of" fashion. The canonical example is three interlocked triangles, as shown in figure 5.4. In this case, any ordering of the triangles produces transparency errors in some part of the image where triangles overlap. Moreover, as the camera is being rotated, the ordering of triangles can change and cause spontaneous color changes that might irritate a viewer. In general, however, `TM_SORTED_TRIANGLES` is a clear gain over `TM_BACKFACES_FRONTFACES`.

The third transparency mode, `TM_DEPTH_PEELING`, achieves perfect transparency with a rendering technique called *depth peeling*. The idea presented in Everitt [2001] was originally implemented using NVIDA OpenGL extensions for shadow mapping, but in Nubuck we took advantage of the availability of framebuffers on modern GPUs. Another presentation of this technique (and transparency in general) can be found in Akenine-Moller et al. [2002].

First of all, depth peeling transparency reverses the drawing order. The alpha blending equation 5.1 requires fragments being drawn in a back-to-front order and corresponds to a call to `glBlendFunc(GLenum sfactor, GLenum dfactor)` with arguments `sfactor = GL_SRC_ALPHA` and `dfactor = GL_ONE_MINUS_SRC_ALPHA`. We also call this the *over operator*. The following blending equation that corresponds to blending arguments `sfactor = GL_ONE_MINUS_DST_ALPHA` and `dfactor = GL_ONE` is called the *under operator*:

$$c_f = c_d + (1 - \alpha_d)c_s \tag{5.3}$$

As it turns out, the under operator produces results equal to the over operator, when

1. fragments are drawn in a front-to-back order, and
2. fragments are drawn with *premultiplied alpha*, and

3. the initial clear color is $(0, 0, 0, 0)$.

If a fragment shader draws a color $RGBA = (c, \alpha)$ with premultiplied alpha enabled, the actual value written to the framebuffer (i.e., the value assigned to `gl_FragColor`) is $(\alpha c, \alpha)$.

As an example for the under operator, we draw combine three colors c_2, c_1, c_0 in that order. Initially, the framebuffer is cleared to $(0, 0, 0, 0)$, so the value stored in the framebuffer after drawing c_2 is $(\alpha_2 c_2, \alpha_2)$ (fragments are drawn with premultiplied-alpha). Drawing c_1 gives

$$
\begin{aligned}
&(\alpha_2 c_2, \alpha_2) + (1 - \alpha_2)(\alpha_1 c_1, \alpha_1) \\
=&(\alpha_2 c_2 + (1 - \alpha_2)\alpha_1 c_1, (1 - \alpha_2)\alpha_1)
\end{aligned}
$$

Finally, drawing c_0 gives the same value for the final color as equation 5.2.

As the name suggests, depth peeling „peels away" layers of fragments in a front-to-back fashion. The principal observation is that the front-most fragments are captured by the depthbuffer. Thus, the first layer of fragments can be readily obtained by an ordinary rendering pass. The main idea is to read the depthbuffer written by the first pass to extract the second layer in the next pass, by performing a *dual depth test* as follows. The first depth test is an ordinary depth test that determines the fragments with minimal z-coordinates. The second depth test rejects fragments that lie on or in front of fragments of the first pass. Iteratively, the next layer is stripped away using the depthbuffer of the previous pass.

Nubuck's implementation of depth peeling uses framebuffer objects to render to *depth textures* that can be sampled in the fragment shaders. The relevant section of `mesh.frag` is given in the following listing:

```
// ... in mesh.frag                                          1
if(PERFORM_DEPTH_TEST || PERFORM_DEPTH_PEEL) {               2
  vec2 texCoords;                                            3
  texCoords.x = gl_FragCoord.x / uWidth;                     4
  texCoords.y = gl_FragCoord.y / uHeight;                    5
(A) if(PERFORM_DEPTH_TEST) {                                 6
    float sdepth = texture2D(solidDepth, texCoords).a;       7
    if(gl_FragCoord.z > sdepth) discard;                     8
  }                                                          9
(B) if(PERFORM_DEPTH_PEEL) {                                 10
    float depth = texture2D(depthTex, texCoords).a;          11
    if(gl_FragCoord.z <= depth) discard;                     12
  }                                                          13
}                                                            14
```

The second depth test that rejects fragments from the previous layer is in (B). The other depth test, (A), rejects fragments that are behind opaque meshes, which have been rendered in a previous rendering pass.

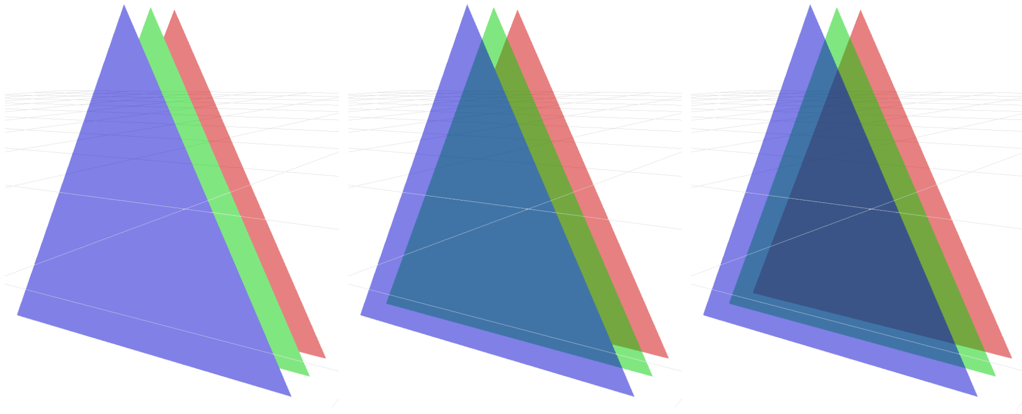


Figure 5.5.: Three differently colored triangles drawn with an (from left to right) increasingly number of depth peels. The initial peel shows only the frontmost fragments, and subsequent peels reveal additional layers.

An obvious disadvantage of depth peeling is the fact that each depth layer needs an additional rendering pass. Also, the number of depth layers needed for any given scene is generally unknown, so the total number of depth peels must be sufficiently large. On some hardware a counter exists that can be used to terminate the peel when no more fragments are generated.[2] As we draw the depth layers in a front-to-back order, however, we draw the most significant layers first. In practice, a depth layer count of $n = 5$ already gives an adequate approximation to perfect transparency. The number of depth peels can be set in the rendering options of Nubuck. Figure 5.5 shows the same scene with an increasingly number of depth peels.

A fixed number of framebuffers suffices for an arbitrary number of depth peels, because the framebuffers can be used in an alternating fashion (*ping-pong* buffers): Let A_0, A_1 be two depth buffers. Then, the ith peeling pass writes to $A_{i \mod 2}$, while reading from $A_{(i+1) \mod 2}$.

5.6. Screenshots

At some point a user might want to save a rendering as an image file; for example, to publish it in a thesis like this. Obviously, Windows' built-in screenshot tool can be used to make a picture of the entire application — including all GUI elements. In almost all cases, however, only the content of the renderview widget is of interest. To spare the user from laborious image cropping, Nubuck can make its own screenshots. By pressing the F12 key, the content of the framebuffer is saved as an uncompressed TGA file, named `screenshot.tga`, in the current working directory.

[2] `https://www.opengl.org/registry/specs/NV/occlusion_query.txt`

The Truvision TGA[3] file format has been chosen because of its very compact specification, which made it possible to implement a reader and writer from scratch. There are superior exchange formats, such as `.jpg` and `.png`, which produce smaller files, but they are so complex that they imply additional library dependencies. Also, it is worth noting that TGA files are lossless and not gamma corrected.

A screenshot made with this technique inherits its resolution from the framebuffer, which in turn matches the size of the renderview widget. This is unfortunate on devices with limited screen real estate, such as 13 inches laptops, where it might not be possible to scale the renderview widget to the desired resolution, so screenshots would always appear pixelated when magnified.

As a solution, Nubuck can make larger screenshots by temporarily resizing the framebuffer for one frame. In fact, there are multiple framebuffers and all of them need to be resized (recall that some effects like depth peeling render to more than one offscreen framebuffer). Resizing a framebuffer means destroying the old buffer resource and allocating a new one, which is a costly operation, but screenshots are taken very infrequently. The maximum sizes of textures and renderbuffers, which are bound to the color attachment points of a framebuffer, are implementation-dependent. On a Nvidia GeForce 286 GPU we were able to create screenshots with 2000×2000 pixels. The size of a screenshot is set with the configuration variables `screenshot_width` and `screenshot_height`, where negative values mean the current size of the renderview widget. If the variable `screenshot_keepAspect` is set then the width of the screenshot is determined by its height and the aspect ratio of the renderview widget.

Again, the maximum resolution of a screenshot is subject to the capabilities of a particular device. Another function of Nubuck creates *large* screenshots of arbitrary size on any device, albeit with some restrictions. The shortcut for large screenshots is F11 and the size can be set with the configuration variables `screenshot_largeWidth` and `screenshot_largeHeight`.

Nubuck achieves truly high resolutions by subdividing the final image into a number of smaller, non-overlapping rectangles (*tiles*), which are rendered one frame at a time. For each frame, the viewport and view frustum exactly match the location and dimension of a tile, so there are no visible seams between adjacent tiles. At the moment, it's only possible to create large screenshots with perspective projection. The idea of tiled images is also presented in Woo et al. [1999] and is loosely related to *tiled renderers*, which use tiles to exploit spatial coherence or must run on scarce hardware resources.[4] Here, this approach has implications for some rendering techniques.

By default, the renderview widget shows a linear background *gradient* that darkens the background color towards the bottom edge of the widget. Not only does the background gradient look very appealing, it also gives a fake sense of depth. The background gradient is a screen-space effect, which means that it always fills the entire viewport, i.e., background color at the top, and black at the bottom of the viewport. Since the view-

[3] `http://en.wikipedia.org/wiki/Truevision_TGA`
[4] `http://en.wikipedia.org/wiki/Tiled_rendering`

port is adjusted for each tile, however, generating a high resolution screenshot produces a grid of discontinuous gradients. Therefore, it's recommended to disable the background gradient before taking screenshots. In most situations the background color needs to be uniformly white anyway, so this does not appear overly restrictive.

Also, when the shading mode is set to `SM_BILLBOARDS_NICE`, edges are interrupted as they cross adjacent tiles. Recall that this shading mode performs a modified depth test in the fragment shader to test the visibility of the spine of an edge. The problem is that the corresponding point on the spine of some billboard fragment might not be visible. So, for the depth test to work properly, the framebuffer must be larger than the viewport. As the viewport matches a tile exactly, however, the discontinuities become apparent. Although it's clear that the problem can be solved by a larger framebuffer, care must be taken to adjust the view frustums accordingly, so that the perspectives of adjacent tiles match. The required math is not trivial, so this problem has been postponed for now.

Another concern is the edge width in shading mode `SM_LINES`. In this shading mode, the edges and vertices are drawn using OpenGL's built-in primitives, which have a fixed size in screen space. Consequently, the size of edges and vertices approaches zero, as the resolution of the screenshot increases.

Many 2-D visualization tools, including LEDA's GraphWin and GeoWin, can export scalable vector graphics (SVG), which are resolution independent. As 3-D polyhedrons consist of vertices and edges, it's theoretically possible to represent them with SVG, too. An export procedure might look like this: Project the vertices and edges. Perform hidden line removal. Express the visible vertices and lines as SVG primitives. The downside is that SVG is generally not suitable to capture the per-fragment shading of the faces, so an important cue for the shape of the mesh is lost. By the very nature of OpenGL, however, it is easier to produce raster images and no efforts have been made to produce SVG images.

Experiments have been conducted to offload the rendering of high resolution images to external applications. In particular, scenes have been exported to POV-Ray, which is a standalone raytracer that generates images from human-readable scene description files.[5] The description language includes solid primitives like spheres and cylinders, so it has been straightforward to imitate shading mode `SM_NICE`. The camera parameters have been set to match the renderview. Unfortunately, it's not possible to exactly reproduce the rendering of Nubuck in POV-Ray, for two reasons. First, Nubuck uses directional lights, which are not supported by POV-Ray. Second, it's hard to match the shading model. Although POV-Ray comes with a rich library of materials, it's not trivial to use a Blinn-Phong or Cook-Torrance model with specific parameters. For these shortcomings, in addition to the problem of rendering text and other primitives, this trail has not been followed any further.

[5] http://www.povray.org

6. Conclusion

In this thesis we presented Nubuck, a new tool for visualizing and animating geometric algorithms. In contrast to many other, similar visualization tools, Nubuck is implemented using contemporary libraries and is readily available on modern machines. Moreover, the widely used C++ programming language and the easy-to-use LEDA library make Nubuck accessible to a large audience of developers. In fact, existing LEDA algorithms require little modifications to work with Nubuck's API. Also, Nubuck is one of only a few tools that focus on the presentation of 3-D algorithms. In comparison to LEDA's `d3_window`, Nubuck offers a richer user interface, more intuitive controls, and a higher image quality.

We presented two examples of potential areas of application. The first example showed how Nubuck can be used in a classroom scenario to teach the relationships among some elementary 2-D and 3-D geometric structures. In the second example we motivated how the process of developing and debugging a complex 3-D algorithm benefits from Nubuck's capabilities.

We discussed the principal concepts of Nubuck and introduced its API, which largely adheres to the design principles of LEDA. The topics covered include mesh creation and manipulation, direct user input, GUI, and animation.

The implementation of the renderer, which is the most prominent part of the visualization, was described in detail. The hardware-accelerated renderer uses modern rendering techniques to deliver high-quality images with few artifacts.

6.1. Future Work

At this point, the number of built-in operators is still fairly limited, and we hope it will increase over time. Other than that, the code contains some features that we would like to iterate on, but did not describe, as they are still in their infancy. Figure 6.1, for example, shows a face highlighted by a *bézier curve*. The curves are animated, can show different decals, and work for faces with arbitrary numbers of vertices. Round highlights such as these generally make a more pleasant appearance than colored faces. Also, experiments have been conducted with automatic camera movements, i.e., the camera automatically rotates so that the vertex currently visited by an algorithm is in the center of the viewport. Reliable, automatic camera movements that work in all cases, however, are hard to realize. For further research in that direction, see for example Shneerson and Tal [1997].

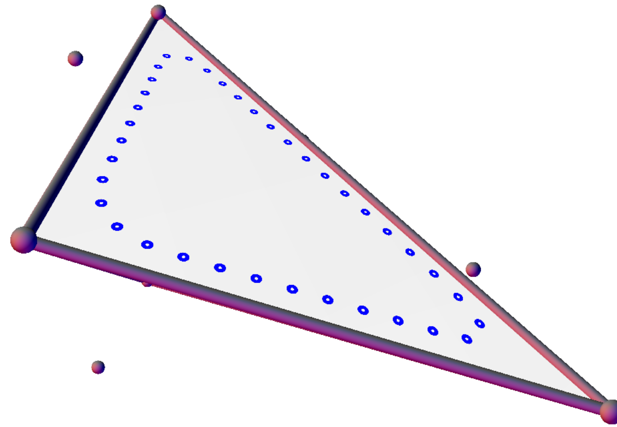


Figure 6.1.: An experimental feature highlights faces with animated bézier curves.

Finally, we believe Nubuck is a competitive visualization tool and would like to see it being actively used in the field of computational geometry in the future.

A. Installation and Building

In this section you will learn to compile programs that use Nubuck. You will see how to create standalone applications and plugins, and we will discuss the benefits of both approaches. We begin by installing and building the Nubuck library itself.

A.1. Requirements

Nubuck works only on Microsoft Windows operating systems. Although both libraries Nubuck depends on, LEDA and Qt, are cross-platform and provide wrappers for system specific functions, some parts of Nubuck use the Windows APIs directly. Examples are threads, locks, timers, and OpenGL context creation. It should be straightforward to port Nubuck to other platforms, but no effort has been made to do so thus far. Therefore, we assume that you run a Windows operating system.

In addition, you must use the compiler that ships with Microsoft Visual C++. Microsoft Visual C++, which is part of the Visual Studio product family, is a commercial integrated development environment (IDE), which features tools for editing and debugging C and C++ programs. We strongly encourage you to use this IDE for developing, but we will just need the compiler, which comes as a separate commandline tool. There exist various editions of Visual C++, varying in features and pricing, but a stripped-down version, called Visual C++ Express, can be downloaded for free.[1] We recommend Visual C++ 2010, but more recent versions should work as well. Note that Nubuck uses some Microsoft-specific language extensions and pragmas, like `__declspec` and `#pragma pack`, which may not be supported by other compilers.

We assume that you are familiar with the build process of the LEDA library. Nubuck expects LEDA to be compiled as a dynamic link library (DLL), which is the Microsoft term for shared library. Also, LEDA must be compiled with the preprocessor identifier `LEDA_MULTI_THREADED` defined. If that's not already the case for your LEDA installation, you have to invoke the configuration script again with the arguments `msc-mt shared`, or `msc-mtd shared`, depending on the runtime libraries you want. Then you have to rebuild LEDA. Note that x86, i.e., 32bit, is the only architecture supported by Nubuck. Nubuck has been tested only with the most recent version of LEDA, LEDA-6.4. In the following, `LEDA_DIR` denotes the root directory of LEDA on your machine. The path to the LEDA DLL, for example, is `%LEDA_DIR%/leda.dll`.

You have to install the Qt libraries, too. Nubuck requires Qt 4.x, version 4.6 or higher.

[1] https://www.visualstudio.com/en-us/products/visual-studio-express-vs.aspx

For a fresh install, use version 4.8.6, which is the most recent 4.x release. There exist an installer for Visual C++ 2010. [2] Qt 5 is already available, but it's not fully backwards-compatible and you will get compilation errors if you attempt to build Nubuck with this version. If you want to get really serious about Qt development, you may want to download the Visual C++ plugin and QtCreator, too, although it's not necessary for Nubuck.

In addition to the library dependencies, you will need two external tools to build Nubuck: Git and CMake.

Git is a revision control system, or source code management (SCM) system, which was originally developed by Linus Torvalds for Linux kernel development. Since then, it has been widely adopted and replaces other SCM tools, like CVS or Subversion. We will use Git to obtain the source code of Nubuck. Alternatively, you can download a zipped version of the sources and extract them manually. A Windows client of Git is available[3], which brings its own Bash emulation to use Git from the commandline.

CMake is a free and open-source build tool.[4] In contrast to most other build systems, CMake does not compile a project directly, but generates the input for the native build system on each platform. That is, classic Makefiles on Linux, or Visual Studio solutions on Windows. You can also generate Makefiles on Windows, if you want. Moreover, CMake provides a number of modules for finding libraries installed on your machine, so you don't have to configure include and library paths manually. Typically, these modules use information from environment variables, the registry, and common install locations to find a package. Nubuck uses the `FindQt` module. As mentioned above, we use the environment variable `LEDA_DIR` to locate LEDA. It would be nice, however, to have a `FindLEDA` module for CMake.

A.2. Building Nubuck

We are now going to show you how to build Nubuck. First, we obtain the Nubuck sources using Git. Open the Git Bash. Change into your LEDA directory and download the repository.

```
cd $LEDA_DIR
git clone https://github.com/plainoldcj/nubuck.git
```

You can close the Git Bash now.

Next, we use CMake to generate a Visual C++ solution. We build out-of-source, which means that we create a dedicated directory, called *build*, which will contain all project files and binaries. Open a command prompt. Change into the location

[2] `http://download.qt.io/archive/qt/4.8/4.8.6/qt-opensource-windows-x86-vs2010-4.8.6.exe`
[3] http://git-scm.com/downloads
[4] http://www.cmake.org/download/

%LEDA_DIR%\src\Nubuck. This is the directory that contains the CMake project file. Create the build directory and change into it. Then call CMake. We tell it to create a Visual C++ 2010 solution of the project located in the parent directory.

```
cd %LEDA_DIR%\src\Nubuck
mkdir build
cd build
cmake -G "Visual Studio 10" ..
```

CMake creates the Visual C++ solution build\nubuck.sln. Open it. In Visual Studio, on the left hand side, you see the *solution explorer*. Right-click on the solution and hit *Build*. That's it.

The Visual Studio solution contains several projects and each project creates a binary. The project named nubuck is the main library and creates the files nubuck.lib and nubuck.dll. Note that on Windows, the static library nubuck.lib is used as an *import library* for the shared library nubuck.dll. Projects having the prefix alg_* or gen_* are Nubuck plugins. Plugins are shared libraries, too. Their .dll-files are moved to the plugin directory %LEDA_DIR%\res\Operators in a post-build step. The project nubuck_standalone creates an executable. Basically, it's a dummy application that runs Nubuck without providing an operator. However, you can still use the build-in operators and plugins.

A.3. Running Nubuck

After the solution has been compiled successfully, you can try to run nubuck_standalone. Depending on your build configuration, you find it in build\Debug or build\Release.

First, however, you must make sure that the application is able to find all the DLLs it depends on, namely nubuck.dll, leda.dll and the Qt libraries. The easiest way to do this is to copy all files into the same folder. Visual Studio put nubuck.dll in the right directory already, and the LEDA DLL can be found in %LEDA_DIR%. Now, the components of Qt are spread over a number of different .dll-files, which are located in %QT_DIR%\bin. For each library, there is a release mode version and a debug mode version. The debug mode versions have the suffix „d". QtCore4d.dll, for example, is the debug mode version of QtCore4.dll. Make sure you copy the right files, depending on your build configuration. Nubuck depends on QtCore4.dll, QtGui4.dll, QtOpenGL4.dll, and QtSvg4.dll. You may want to copy the .pdb+-files as well, if you need to debug Qt code. Also, you must copy the directory %QT_DIR%\plugins\iconengines. Otherwise, some icons are displayed erroneous, or not all. Having done all that, nubuck_standalone.exe should run just fine.

A.4. Changing the Nubuck sources

If you make any changes to the existing Nubuck sources, all you have to do is hit *Build* again in Visual Studio. You do not have to call CMake again. If, on the other hand, you want to add or remove source files, then you must call CMake to update the solution file. Each directory in `%LEDA_DIR%\incl\Nubuck` and `%LEDA_DIR%\src\Nubuck` contains a file named `sources.cmake`, which adds a list of source files to the project. Nubuck's CMake project file is `%LEDA_DIR%\src\Nubuck\CMakeLists.txt`. It collects all files listed in subdirectories. After adding or removing a file, you have to update the `sources.cmake` files accordingly. Then you have to rerun CMake. CMake caches its arguments, so a second invocation can omit the generator: in directory `build`, type „`cmake ..`".

A.5. Building a standalone application

We call a program that provides its own entry point (`main` function) a standalone application. Analogously to the application implemented in section 3.1, every standalone application calls the functions `NB::Init` and `NB::Run` to initialize and execute the Nubuck sandbox. A couple of additional example applications can be found at `%LEDA_DIR%\src\NubuckStandaloneSamples`.

A standalone application consisting of a single source file `main.cpp` can be compiled from the Windows commandline as follows:

```
set CFLAGS= ^
 /I%LEDA_DIR%\incl ^
 /DLEDA_DLL /DLEDA_MULTI_THREAD ^
 /EHsc /MDd
set LFLAGS=/LIBPATH:"%LEDA_DIR%" leda.lib nubuck.lib

cl /nologo main.cpp %CFLAGS% /link %LFLAGS%
```

If the application contains custom Qt code then the Qt libraries must be added, as well. Again, for portability and maintenance we recommend the use of either CMake or the Qt plugin for Visual Studio for, but a minimal commandline example can be found in `%LEDA_DIR%\src\NubuckStandaloneStamples\convex_hull\build.bat`.

A.6. Building a plugin

Nubuck implements a simple *plugin* mechanism that allows the loading of operators at runtime. The advantage of operator plugins is that they can be easily distributed. When written as a plugin, an operator project compiles to a separate DLL file. The operator DLL file can be loaded either explicitly using a menu entry or at startup when it is

located in `%LEDA_DIR%\res\Operators`. When you write a new point generator that would also be beneficial for a coworker, for example, you can simply send the DLL file via E-mail, and the coworker can instantly add your operator to her sandbox, without compiling or linking any source files.

A plugin project compiles to a DLL file and does not provide a `main` function. Instead, it exports the following functions, which create the operator and operator panel:

```
NUBUCK_OPERATOR OP::OperatorPanel* CreateOperatorPanel ↩
    ();
NUBUCK_OPERATOR OP::Operator* CreateOperator();
```

Some of the algorithms that come with Nubuck are, in fact, operator plugins, and can be found in `%LEDA_DIR%\src\NubuckOperators`. All plugins located in this directory are part of the Visual Studio solution generated by CMake. The Nubuck build system makes it easy to add an operator plugin to the main project: Create a new subdirectory in `NubuckOperators`, which contains a file named `CMakeLists.txt` with the following content:

```
recurse()
add_sources(
  // ... list source files here
)

add_operator(${operatorName})
```

Bibliography

Tomas Akenine-Moller, Tomas Moller, and Eric Haines. *Real-Time Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2002. ISBN 1568811829.

Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008. ISBN 3540779736, 9783540779735.

Matthias Bäsken and Stefan Näher. Geowin - a generic tool for interactive visualization of geometric algorithms. In Stephan Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 88–100. Springer, 2001. ISBN 3-540-43323-6. URL `http://dblp.uni-trier.de/db/conf/dagstuhl/svis2001.html#BaskenN01`.

Pedro Jussieu de Rezende and Welson R. Jacometti. Geolab: An environment for development of algorithms in computational geometry. In *CCCG*, pages 175–180. University of Waterloo, 1993. URL `http://dblp.uni-trier.de/db/conf/cccg/cccg1993.html#RezendeJ93`.

Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag New York, Inc., New York, NY, USA, 1987. ISBN 0-387-13722-X.

Peter Epstein, J. Kavanagh, A. Knight, J. May, T. Nguyen, and Jörg-Rüdiger Sack. A workbench for computational geometry. *Algorithmica*, 11(4):404–428, 1994. URL `http://dblp.uni-trier.de/db/journals/algorithmica/algorithmica11.html#EpsteinKKMNS94`.

Cass Everitt. Interactive order-independent transparency, 2001.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

Christian Jäger. Der Algorithmus zur Berechnung der konvexen Hülle von Preparata und Hong. Bachelor thesis, University of Trier, August 2012. URL `http://cip.uni-trier.de/~jaeger/ph.pdf`.

Sebastian Kürten and Wolfgang Mulzer. Livecg: an interactive visualization environment for computational geometry. In *30th Annual Symposium on Computational Geometry, SOCG'14, Kyoto, Japan, June 08 - 11, 2014*, page 86, 2014. doi: 10.1145/2582112.2595645. URL `http://doi.acm.org/10.1145/2582112.2595645`.

Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*. Charles River Media, Inc., Rockland, MA, USA, 2002. ISBN 1-58450-037-9.

Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-56329-1.

Joseph O'Rourke. *Computational Geometry in C (2Nd Ed.)*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-64976-5.

Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010. ISBN 0123750792, 9780123750792.

F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20(2):87–93, February 1977. ISSN 0001-0782. doi: 10.1145/359423.359430. URL `http://doi.acm.org/10.1145/359423.359430`.

Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985. ISBN 0-387-96131-3.

Peter Schorn. Implementing the xyz geobench: A programming environment for geometric algorithms. In Hanspeter Bieri and Hartmut Noltemeier, editors, *Workshop on Computational Geometry*, volume 553 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 1991. ISBN 3-540-54891-2. URL `http://dblp.uni-trier.de/db/conf/cga/cga1991.html#Schorn91`.

Maria Shneerson and Ayellet Tal. Visualization of geometric algorithms in an electronic classroom. In *Proceedings of the 8th Conference on Visualization '97*, VIS '97, pages 455–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press. ISBN 1-58113-011-2. URL `http://dl.acm.org/citation.cfm?id=266989.267120`.

Marco Tarini, Paolo Cignoni, and Claudio Montani. Ambient occlusion and edge cueing for enhancing real time molecular visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1237–1244, September 2006. ISSN 1077-2626. doi: 10.1109/TVCG.2006.115. URL `http://dx.doi.org/10.1109/TVCG.2006.115`.

Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999. ISBN 0201604582.

Richard Wright, Benjamin Lipchak, and Nicholas Haemel. *Opengl Superbible: Comprehensive Tutorial and Reference, Fourth Edition*. Addison-Wesley Professional, fourth edition, 2007. ISBN 9780321498823.