TinySpline: A Small, yet Powerful Library for Interpolating, Transforming, and Querying NURBS, B-Splines, and Bézier Curves

Marcel Steinbeck
University of Bremen, Germany
marcel@informatik.uni-bremen.de

Rainer Koschke
University of Bremen, Germany
orcid.org/0000-0003-4094-3444

Abstract—**NURBS, B-Splines, and Bézier curves have a wide range of applications. One of them is software visualization, where these kinds of splines are often used to depict relations between objects in a visually appealing manner. For example, several visualization techniques make use of hierarchical edge bundles to reduce the visual clutter that can occur when drawing a large number of edges. Another example is the visualization of software in 3D, virtual reality, and augmented reality environments. In these environments edges can be drawn as splines in 3D space to overcome the natural limitations of the two-dimensional plane—e.g., the collision of edges with other objects. While Bézier curves are supported quite well by most UI frameworks and game engines, NURBS and B-Splines are not. Hence, spline-based visualizations are considerably more difficult to implement without in-depth knowledge in the area of splines.**

In this paper we present TinySpline, a general purpose library for NURBS, B-Splines, and Bézier curves that is well suited for implementing advanced edge visualization techniques—e.g., but not limited to, hierarchical edge bundles. The core of the library is written in ANSI C with a C++ wrapper for an object-oriented programming model. Based on the C++ wrapper, auto-generated bindings for C#, D, Go, Java, Lua, Octave, PHP, Python, R, and Ruby are provided, which enables TinySpline to be integrated into a large number of applications.

I. Introduction

Software visualization is the attempt to assist developers in understanding and analyzing complex software systems by mapping certain characteristics of software onto visual attributes—exploiting the human ability to recognize patterns in complex data. In many visualization approaches graphs with nodes and edges form the input data. For example, EvoStreets [1], [2] are well suited for visualizing the hierarchical structure of trees by depicting leaf nodes as three-dimensional blocks arranged on nested streets (branching at each level change of the hierarchy). In addition, edges (representing relations between leaf nodes) can be expressed by visually connecting related blocks with lines. To reduce the visual clutter that can occur when drawing a large number of edges, Holten introduced hierarchical edge bundles [3]. Instead of drawing edges as straight lines, Holten proposed to utilize the hierarchical structure of the input graph and to depict edges as B-Splines where the inner nodes of the node hierarchy are used to set up the control points of the rendered splines. Holten implemented and evaluated hierarchical edge bundles in different two-dimensional layouts, amongst others, radial and squarified Treemaps, rooted tree layouts, and balloon layouts. However, due to their genericness, edge bundles can also be used in three-dimensional layouts [4], [5]. Moreover, B-Splines are not only suitable for bundling edges but also in general for rendering edges in a visually appealing manner.

Most UI frameworks and game engines implement functions for drawing Bézier curves. B-Splines (and their generalization NURBS), however, are rarely supported—the only library with support for NURBS/B-Splines we know of is the GLU NURBS Interface [6]. Thereby, implementing advanced edge visualization techniques requires a deeper understanding in the area of splines.

Contributions. This paper presents *TinySpline*, a programming library for NURBS, B-Splines, and Bézier curves. Although TinySpline is designed as a general purpose library for these kinds of splines, it is well suited for being integrated into existing software visualization applications. With TinySpline, spline-based visualizations can be implemented without having to know all the intricacies of splines. Currently, the library is available for twelve different programming languages.

The remainder of this paper is structured as follows. Section II gives a short introduction into NURBS, B-Splines, and Bézier curves. Section III describes TinySpline's design and implementation. Section IV provides examples for solving common tasks—related to the visualization of splines—with TinySpline. Section V presents a showcase where we use TinySpline to draw hierarchical edge bundles in software visualization. Related works are presented in Section VI. Section VII, eventually, concludes.

II. NURBS, B-Splines, and Bézier curves

In the following, a brief introduction into NURBS, B-Splines, and Bézier curves is given. We limit ourselves to the essential concepts that are necessary to create and visualize these kinds of splines with TinySpline—the introduction is primarily aimed for developers. For a more extensive introduction—in particular, into the mathematics behind splines—we refer the reader to existing literature [7].

a) Bézier curves: From among NURBS, B-Splines, and Bézier curves, Bézier curves are the simplest variant of splines. Bézier curves have two properties: i) a degree and ii) a set of control points. The degree and the number of control points of a Bézier curve are related to each other: A Bézier curve of degree p has exactly $m = p+1$ control points (in the following, we also refer to the order of splines which is $p+1$). Hence, the higher the degree (order) of a Bézier curve, the more control points it has. The lowest possible degree is 0. In this case, the corresponding curve has only one control point and forms a point. Bézier curves have proven useful because the shape of a Bézier curve can be adjusted quite intuitively by modifying its control points. While Bézier curves start and end at their first and last control point respectively, they generally do not pass through their internal control points (it is possible to create curves that pass through some or even all of their control points though). Instead, inner control points serve as points of attraction. Figure 1 shows an example Bézier curve of degree three (i.e., it has four control points).

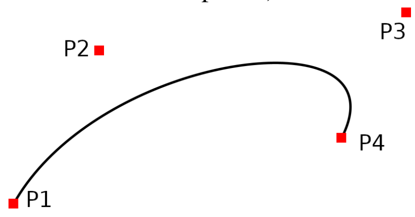

Fig. 1: Cubic Bézier curve with four control points $P1$, $P2$, $P3$, and $P4$. The curve starts at $P1$ and ends at $P4$ (it actually passes these points). The internal control points $P2$ and $P3$ serve as points of attraction (as if the spline is connected to them by a rubber band).

Most UI frameworks with support for Bézier curves provide functions for drawing quadratic ($p = 2, m = 3$,; also known as parabola) and cubic ($p = 3, m = 4$) curves. A sequence of connected (i.e., continuous) Bézier curves is also referred to as Bézier spline (a piecewise Bézier curve). If a Bézier spline is C^{p-1} continuous, it forms a B-Spline.

b) B-Splines: B-Splines (basis splines) are generalized Bézier curves. In addition to degree and control points, B-Splines have a third property: iii) the knot vector (a sequence of non-decreasing numbers—knots—where the multiplicity of each knot—i.e., how often it occurs in the sequence—must be less than or equal to the order of the spline). Based on a given set of control points, the knot vector can be used to (slightly) change the shape of the spline—yet, the relationship between the modifications of the knot vector and the resulting change in shape is less intuitive than with the control points. In the context of rendering edges as splines in software visualization (which is the main focus of this paper), the knot vector of B-Splines is of less importance and, thus, is not further discussed in this paper—TinySpline, by default, automatically creates the desired sequence of knots anyway.

B-Splines can be decomposed into C^{p-1} continuous Bézier splines of the same shape. This is in particular useful when drawing B-Splines in UI frameworks that support only Bézier curves. That is, the B-Spline to be drawn is decomposed into a sequence of Bézier curves which are then drawn one after another. An example of a decomposed B-Spline is shown in Figure 2.

c) NURBS: While B-Splines are generalized Bézier curves, NURBS (Non-uniform rational B-spline) are generalized B-Splines—transitively, NURBS are also generalized Bézier curves. Although being flexible enough in most scenarios (e.g., hierarchical edge bundling), there are also shapes that B-Splines cannot represent exactly—e.g., circles. The essential extension of NURBS is that control points are given a weight—weighted control points. NURBS whose control points all have weight 1 are basically B-Splines. Section IV-A shows how weighted control points are encoded in TinySpline. Nowadays, NURBS are commonly used in computer-aided design, yet, they may also be useful in software visualization.

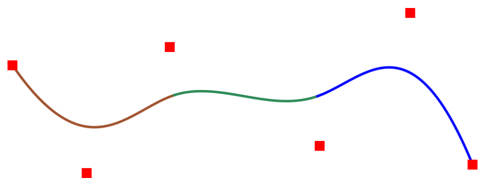

Fig. 2: Cubic B-Spline with six control points (red squares) decomposed into a C^{p-1} continuous Bézier spline of three Bézier curves (brown, green, and blue).

III. Design and Implementation

TinySpline is publicly available at GitHub [8] and licensed under the terms of the MIT license. The library was designed with the following goals in mind:

G1 Provide a common API for NURBS, B-Splines, and Bézier curves.
G2 Splines may have any degree and dimensionality (2D, 3D, etc.).
G3 Treat splines as objects. Protect the internal state of these objects from unwanted changes (rendering splines in an invalid state).
G4 In case of errors, provide meaningful error messages.
G5 Make as few assumptions as possible about how splines are integrated into the client code (e.g., how splines are drawn). That is, TinySpline is intended as a general purpose library for NURBS, B-Splines, and Bézier curves.
G6 The library should have as few third party dependencies as possible.
G7 The library should be available for several (desirably as many as possible) programming languages, so that it can be used by a larger amount of applications.

The core of the library is written in ANSI C (also referred to as C89), a C standard that is understood by almost all compiler suites (e.g., GCC, Clang, and MSVC). In addition, a C++ wrapper (C++98) is provided. The wrapper encapsulates the structs of the C interface in classes and maps functions to methods—providing an object-oriented programming model (*G3*). Errors reported by the C interface (e.g., when trying to set up a spline with an invalid number of control poitns; cf. Sections II and IV-A) are mapped to C++ exceptions (e.g., *std::runtime_error*) with corresponding error message (*G4*). Based on the C++ wrapper, bindings for additional programming languages are auto-generated using SWIG [9]. At the moment of writing, C#, D, Go, Java, Lua, Octave, PHP, Python, R, and Ruby are supported (*G7*).

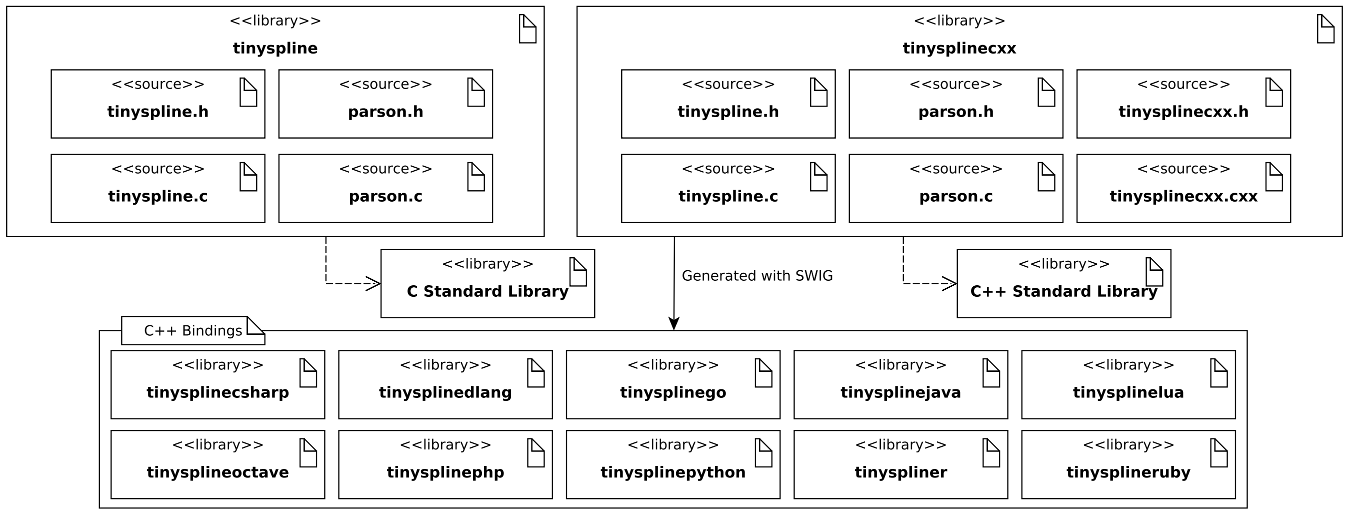

Fig. 3: TinySpline artifacts. The artifact *tinyspline* denotes the C interface. The artifact *tinysplinecxx* denotes the C++ interface.

Figure 3 shows all library artifacts provided by TinySpline as UML diagram. The files *tinyspline.h* and *tinyspline.c* contain the source code of the C core. Likewise, *tinysplinecxx.h* and *tinysplinecxx.cxx* contain the source code of the C++ wrapper. TinySpline includes a local copy of the ANSI C library Parson [10] (*parson.h* and *parson.c*). It is used privately (i.e., Parson is not exposed by TinySpline's API) for serialization and deserialization of splines to and from JSON. This is in particular useful when sharing splines between programming languages (e.g., when splines are created in Python but visualized in Java). Besides Parson, the C and C++ library depend on the C standard library and the C++ standard library respectively. Beyond that, they have no further dependencies (*G6*). The bindings generated from the C++ library using SWIG, however, may have additional (as a matter of principle unavoidable) dependencies—e.g., the runtime library of the target language. For the sake of overview, these dependencies are not outlined explicitly in Figure 3. Instead, we refer the reader to the build instructions of TinySpline that can be found in the Git repository [8].

IV. Usage

In this section, we demonstrate how to create, interpolate, and draw (i.e., how to prepare the data) splines with TinySpline—the essentials for using splines in software visualization. The code snippets listed in this section use TinySpline's Python interface. However, they can be adapted to the other interfaces without much effort.

A. Creating Splines

Splines are created by calling the constructor of class *BSpline*. The simplest call only passes the number of desired control points:

Listing 1: Create a spline with seven control points.

```
spline = BSpline(7)
```

This creates a two-dimensional, cubic B-Spline with seven control points. Instead of using the default values for dimension and degree, the desired values can be set explicitly:

Listing 2: Create a three-dimensional, quintic spline with ten control points.

```
spline = BSpline(10, 3, 5)
```

Because Bézier curves are a special case of general B-Splines ($m = p + 1$) they can simply be represented as B-Spline objects. For example:

Listing 3: Create a four-dimensional, quadratic Bézier curve with three control points.

```
beizer = BSpline(3, 4, 2)
```

In order to understand how NURBS are created, it should first be shown how control points are managed in TinySpline.

As stated by *G2*, splines may have any dimensionality. That is, control points may consist of any number of components (for technical reasons, the lower limit is one; nildimensional splines have no operational purpose anyway). Thus, there is no explicit representation for control points. Instead, they are encoded as one-dimensional list. For example:

Listing 4: Setting up control points of a two-dimensional, cubic spline with four control points.

```
spline = BSpline(4, 2, 3)
ctrlps = [p0x, p0y, p1x, p1y, p2x, p2y, p3x, p3y]
spline.control_points = ctrlps
```

As already mentioned in Section II, NURBS, unlike Bézier curves and B-Spline, have weighted control points—each control point is assigned a dedicated weight. Weighted control points can be encoded as homogeneous coordinates of B-Splines. That is, i) each control point has an additional component, $w \neq 0$, and ii) the remaining components are weighted with w. For example:

Listing 5: Setting up control points of a three-dimensional, quardratic NURBS with three weighted control points.

```
nurbs = BSpline(3, 3, 2)
nurbs.control_points = [
   p0x*w0, p0y*w0, w0,
   p1x*w1, p1y*w1, w1,
   p2x*w2, p2y*w2, w2]
```

Representing NURBS as B-Spline objects with homogeneous coordinates has two advantages. Firstly, NURBS, as with Bézier curves, do not need their own representation (*G1*). Secondly, homogeneous coordinates do not need special case treatment. Accordingly, all functions provided by TinySpline—e.g., spline decomposition—can be used seamlessly for NURBS, B-Splines, and Bézier curves (*G1*).

B. Spline Interpolation

Often the situation arises that a spline should not only be attracted by its inner control points, but actually pass through them. The process of calculating control points forming a spline that passes through a set of given points is called spline interpolation. It should be noted that there is no such thing as one single method for interpolating splines, but many different with different objectives. TinySpline provides an implementation that is well suited for drawing edges as splines in software visualization: natural cubic spline interpolation (for more details on the interpolation of splines, we refer the reader to existing literature [11]). A list of points can be interpolated as follows (the second parameter of *interpolate_cubic_natural* specifies the dimensionality of the points to interpolate and, hence, the resulting spline):

Listing 6: Natural cubic spline interpolation.

```
points = [p0x, p0y, ..., pnx, pny]
spline = BSpline.interpolate_cubic_natural(points, 2)
```

C. Drawing Splines

Most UI frameworks (and game engines) cannot render NURBS/B-Splines out-of-the-box but provide—at least basic—support for Bézier curves. In such case, splines can be drawn by decomposing them into Bézier splines (using the method *to_beziers*) and processing each of the Bézier curve contained therein one after another. The following code snippets shows a pattern (*curve* denotes the method provided by the UI framework in use for drawing Bézier curves):

Listing 7: Drawing a spline as a sequence of Bézier curves.

```
spline = BSpline(5, 2, 3)
spline.control_points = [...]
ctrlps = spline.to_beziers().control_points
offset = spline.order * spline.dimension
for n in range(int(len(ctrlps) / offset)):
   p0x = ctrlps[n * offset]
   p0y = ctrlps[n * offset + 1]
   p1x = ctrlps[n * offset + 2]
   p1y = ctrlps[n * offset + 3]
   p2x = ctrlps[n * offset + 4]
   p2y = ctrlps[n * offset + 5]
   p3x = ctrlps[n * offset + 6]
   p3y = ctrlps[n * offset + 7]
   curve(p0x, p0y, p1x, p1y, p2x, p2y, p3x, p3y)
```

If the UI framework in use does not support Bézier curves either, splines can also be decomposed into a sequence of connected lines—polyline—using the method *sample*. If no argument is passed to *sample*, TinySpline tries to estimate a suitable amount of lines. Otherwise, a fixed number of lines is created. Again, the following code snippets shows a pattern (*line* denotes the method provided by the UI framework in use for drawing lines):

Listing 8: Drawing a spline as a sequence of lines.

```
spline = BSpline(5, 2, 3)
spline.control_points = [...]
pts = spline.sample()
for p in range(int(len(pts) / spline.dimension - 1)):
   p0x = pts[p * spline.dimension]
   p0y = pts[p * spline.dimension + 1]
   p1x = pts[(p+1) * spline.dimension]
   p1y = pts[(p+1) * spline.dimension + 1]
   line(p0x, p0y, p1x, p1y)
```

D. Spline Straightening

Along with hierarchical edge bundles, Holten proposed spline straightening—diminishing the bundling strength of splines—so as to resolving bundling ambiguity. The straightening of splines is realized as a function applied to each spline controlled by a parameter $\beta \in [0, 1]$. If β is 0, the resulting spline becomes a straight line connecting its start and end control points (no bundling at all). If β is 1, the spline keeps its original shape (full bundling strength). With TinySpline, splines can be straighten using the method *tension*:

Listing 9: Straighten a spline with $\beta = 0.85$.

```
spline = BSpline(...)
straightened = spline.tension(0.85)
```

V. Showcase

Figure 4 shows a screenshot of our visualization tool *SEE* (Software Engineering Experience). SEE uses the Unity game engine for visualizing hierarchical graphs, supporting different node and edge layouts. The example shown in Figure 4 highlights cloning in the networking component of the Linux kernel using a balloon layout along with hierarchical edge bundles. The control points of the splines are computed based on the center of the hierarchically nested circles. Because Unity does not support Bézier curves out-of-the-box, we use TinySpline's line decomposition feature (cf. Listing 8) in order to render splines with Unity's LineRenderer component [12].

VI. Related Works

NURBS, B-Splines, and Bézier curves have a wide range of applications (software visualization, computer aided design, statistics, to name only a few). Accordingly, several libraries have been implemented (and made public) in the last years [13], [14], [15], [16], [17], [18], [19], [20]. Among them, *SPLINTER*, *NURBS-Python (geomdl)*, and *verb* are some of the strongest competitors to TinySpline.

SPLINTER is a C++ library whose main focus is on function approximation, regression, data smoothing, and data

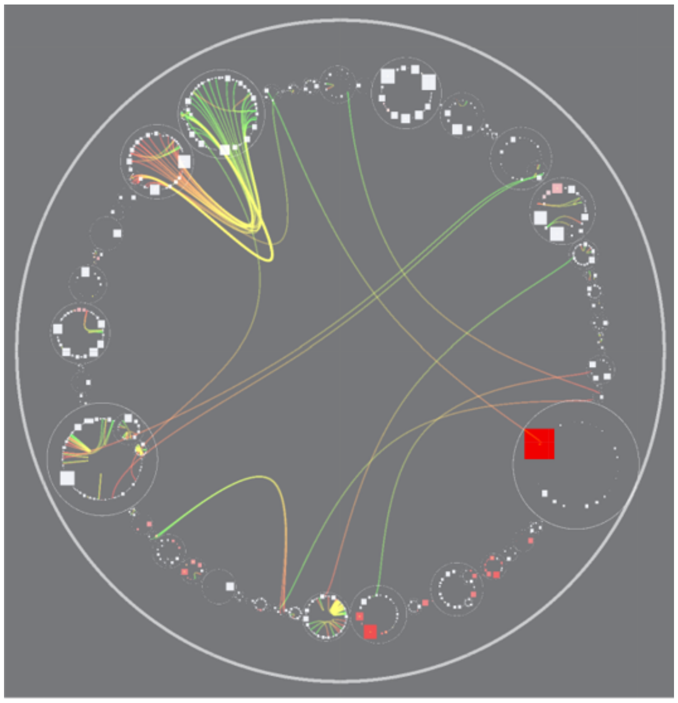

Fig. 4: Hierarchical edge bundles with TinySpline ($\beta = 0.85$).

reduction. In addition to C++, interfaces for C, Matlab, and Python are provided. SPLINTER, unlike TinySpline, explicitly supports P-Splines (penalized B-spline), a spline model that is well suited for smoothing data—points—rather than interpolating it.

Geomdl is a feature rich NURBS library written in Python without external dependencies. The library not only supports curves, but also surfaces and volumes (which is out of TinySpline's scope). Hence, it is well suited for geometric modeling. Similar to TinySpline, geomdl represents splines as objects in object-oriented programming.

Verb is a general purpose library for NURBS curves and surfaces. It is written in the Haxe programming language and, by Haxe's design, provides interfaces for several other languages—at the moment of writing, C++, C#, JavaScript, PHP, and Python are supported (the main target is JavaScript though). On the one hand, the interfaces generated by verb do not rely on native, operating-system-dependent libraries (as opposed to the interfaces of TinySpline), which simplifies the distribution of binary packages. On the other hand, SWIG (the tool that is used by TinySpline to generate bindings) supports many more target languages, allowing TinySpline to be integrated into a larger number of applications—e.g., applications written in D, Go, R, and Ruby.

The strengths of TinySpline compared to other libraries can be summarized as follows: i) support for splines of any degree and dimensionality (most other libraries support only two- and three-dimensional splines), ii) performance (due to lack of space we cannot provide numbers though), iii) features specific to software visualization (e.g., spline straightening), and iv) bindings for a larger set of programming languages.

VII. Conclusion

We presented TinySpline, a library for NURBS, B-Splines, and Bézier curves. While TinySpline's core is implemented in ANSI C, interfaces for C++, C#, D, Go, Java, Lua, Octave, PHP, Python, R, and Ruby are also provided—allowing the integration of TinySpline into a large number of applications.

We demonstrated how TinySpline assists in the creation and rendering of splines in the context of software visualization (without the need to know all the intricacies of splines). Anyhow, TinySpline is designed as a general purpose library and, thus, can be used in other contexts, too. For example, the open-source applications *LibreCAD_3* and *KiCad* use TinySpline's spline decomposition feature for processing splines in computer aided design.

TinySpline is publicly available at GitHub and we encourage researchers and developers to request and add new features. We started the development of the library in 2014. Since then, we constantly improve our code base.

References

[1] F. Steinbrückner and C. Lewerentz, "Representing development history in software cities." ACM, 2010, pp. 193–202.

[2] F. Steinbrückner, "Consistent software cities: supporting comprehension of evolving software systems," Ph.D. dissertation, Brandenburgische Technische Universität Cottbus, 06 2013. [Online]. Available: https://opus4.kobv.de/opus4-btu/frontdoor/index/index/docId/1681

[3] D. H. R. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," vol. 12, no. 5, pp. 741–748, Sep. 2006.

[4] M. Steinbeck, R. Koschke, and M.-O. Rüdel, "Comparing the evostreet visualization technique in two- and three-dimensional environments—a controlled experiment," 2019, pp. 231–242.

[5] R. Koschke and M. Steinbeck, "Clustering paths with dynamic time warping," in *2020 Working Conference on Software Visualization (VISSOFT)*, 2020, pp. 89–99.

[6] "Opengl programming guide," last accessed: 2020/11/15. [Online]. Available: https://www.glprogramming.com/red/chapter12.html

[7] L. Piegl, "On nurbs: a survey," *IEEE Computer Graphics and Applications*, vol. 11, no. 1, pp. 55–71, 1991.

[8] "Tinyspline git repository," last accessed: 2020/11/15. [Online]. Available: https://github.com/msteinbeck/tinyspline

[9] D. Beazley, "Swig homepage," 2003, last accessed: 2020/11/15. [Online]. Available: http://www.swig.org

[10] "Parson git repository," last accessed: 2020/11/15. [Online]. Available: https://github.com/kgabis/parson

[11] L. Piegl and W. Tiller, *The NURBS Book*, 2nd ed. New York, NY, USA: Springer-Verlag, 1996.

[12] "Unity linerenderer component documentation," last accessed: 2020/11/15. [Online]. Available: hhttps://docs.unity3d.com/ScriptReference/LineRenderer.html

[13] B. Grimstad *et al.*, "SPLINTER: a library for multivariate function approximation with splines," http://github.com/bgrimstad/splinter, 2015, accessed: 2015-05-16.

[14] O. R. Bingol and A. Krishnamurthy, "NURBS-Python: An open-source object-oriented NURBS modeling framework in Python," *SoftwareX*, vol. 9, pp. 85–94, 2019.

[15] "verb homepage," last accessed: 2020/11/15. [Online]. Available: http://verbnurbs.com

[16] "libnurbs homepage," last accessed: 2020/11/15. [Online]. Available: http://libnurbs.sourceforge.net

[17] "Gsl documentation," last accessed: 2020/11/15. [Online]. Available: https://www.gnu.org/software/gsl/doc/html/bspline.html

[18] "C++ library for cubic spline interpolation," last accessed: 2020/11/15. [Online]. Available: https://kluge.in-chemnitz.de/opensource/spline

[19] "Splinelibrary git repository," last accessed: 2020/11/15. [Online]. Available: https://github.com/ejmahler/SplineLibrary

[20] "b-spline git repository," last accessed: 2020/11/15. [Online]. Available: https://github.com/thibauts/b-spline