UNIVERSITÀ DEGLI STUDI DI FIRENZE
Scuola di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Magistrale in Informatica

B-SPLINE METHODS FOR THE DESIGN OF SMOOTH SPATIAL PATHS WITH OBSTACLE AVOIDANCE

METODI B-SPLINE PER IL DISEGNO DI PERCORSI REGOLARI IN AMBIENTI TRIDIMENSIONALI CONTENENTI OSTACOLI

Tesi di Laurea Magistrale in Informatica

Relatore: *Alessandra Sestini* Correlatore: *Carlotta Giannelli*

Candidato: STEFANO MARTINA

Luglio 2016, Anno accademico *2015/16*

Stefano Martina (stefano.martina@stud.unifi.it): *B-Spline methods for the design of smooth spatial paths with obstacle avoidance,* Corso di Laurea Magistrale in Informatica .

SUMMARY

summary

I need to insert a nice quotation here,
before printing this dissertation.

— Stefano Martina

THANKS

---

thanks

CONTENTS

1

INTRODUCTION

The design of *motion planning* strategies plays a fundamental role in different applications, from robotics to scientific visualization [13]. *Path planning* problem is narrower, it includes the identification of paths that do not intersect any obstacle [13]. Such paths are designed using B-spline curves, a reference standard in Computer-Aided Design (CAD) and Computer-Aided Geometric Design (CAGD) [17][18].

The considered topic is highly cross-disciplinary. In fact we designed this project with the help of an extended set of competencies acquired following the courses. We applied notions of *linear algebra* for the collision checks; *numerical analysis* for the design of the curves; *computational geometry*, *graph theory*, *probability* and *algorithm theory* for the design of the algorithms; and finally *theoretical computer science* for the analysis of the costs.

We focused on finding a trade-off between having a short curve, a smooth curve, and keeping the time complexity low. We explored different solutions with different collocations on the trade-off triangle and with different qualitative effects on the curve.

We developed a framework in Python using Visualization Tool Kit (VTK) for the graphic output. We used a roadmap method based on Voronoi Diagrams (VDs) for creating a graph (details in Section 5.1.1), that is the base structure for the project. Using such structure we developed three different solutions.

1. The first method benefits from the Convex Hull Property (CHP) of B-spline curves (Section 3.2.1). it applies a transformation on the graph such that every path in it can be used as a control polygon for an obstacle-free curve (Section 5.1.2). Therefore the algorithm selects the shortest path on the transformed graph and build the curve on it (Section 5.2.1).

2. The second method still benefits from the CHP, but it picks the shortest path directly on the base graph. If violations of the CHP emerge on it, then rectification measures are taken (crefsec:inter2).

3. The third method uses a probabilistic approach. Starting from the shortest path in the original graph, it performs a simulated annealing optimization (Section 3.4.3) that converge in a state were the desired trade-off between having a short curve, and low curvature and torsion peaks, is optimal (Section 5.6).

This document is structured as follows, we have three parts. The first (Part I) Is dedicated to the state of the art, we provide an survey of different topics and algorithms related to *motion planning*.

The second part (Part II) is committed to describing all the different parts of the algorithm. In detail Chapter 3 gives to the reader all the necessary notions to understand the following parts. Chapter 4 describes how the environment and the resulting curve are represented. Finally in Chapter 5 we describe how to obtain the basic structures (Section 5.1), how to avoid the obstacles using the three methods described before (Section 5.2), and how to improve the obtained curve simplifying the control polygon (Section 5.5), increasing the curve degree (Section 5.3) and changing the B-spline knot vector (Section 5.4).

The third part (Part III) describe the instruments used for implementing the algorithms (crefcha:codeStructure) and presents a series of tests with different scenes and configurations (Chapter 7). Furthermore we deduce conclusions from the execution of the tests (Chapter 8).

Part I

STATE OF THE ART

2

MOTION PLANNING

The problem of *motion planning* consists in determining a set of low level tasks, given an high level goal to be fulfilled [6]. For instance a classic motion planning problem is the *piano movers'* problem that involves the motion of a free flying rigid body in the 3-dimensional space from a start to a goal configuration by applying translations and rotations and by avoiding collisions with a set of obstacles [6][21]. Motion planning finds applications in different areas, like robotics, Unmanned Aerial Vehicles (UAVs) [14] and autonomous vehicles [23]. These are the most famous applications but it finds utilization also in other less common areas like motion of digital actors or molecule design [6].

Initially the term motion planning referred only to the translations and rotations of objects, ignoring the dynamics of them, but lately the research on this field started considering also the physical constraints of the object to be moved [21]. Usually the term *trajectory planning* is used to refer to the problem of taking the path produced by a motion planning algorithm and determine the time law for moving a robot on it by respecting its mechanical constraints [21].

For motion planning problems an important concept is the *state space*, that can have different dimensions, one for each degree of freedom of the robot. It can be a discrete or a continuous space [21]. We can call the space state $\mathbb{S}$ and, considering that there are obstacles or constraints on the scene, we can call $\mathbb{S}_{free} \subseteq \mathbb{S}$ the portion of the state space such that all its configurations are admissible. On this space state we have the two special states $\mathbf{s} \in \mathbb{S}$ and $\boldsymbol{e} \in \mathbb{S}$ respectively of the desired *start* and *end* configurations.

Another concept is the geometric design of the scene and the actor. The obstacles can be represented as convex polygons/polyhedrons, or also as more complex shapes [21].

Furthermore, it is important to define the possible admissible transformations of the body, if it is possible only to translate and rotate it or if

its motion is composed of rigid kinematic chains or trees or if it is even possible to have not rigid transformations (flexible materials) [21].

2.1 PROBLEM TYPES

In the literature many different problems related to motion planning have been introduced. In this section we present a short survey of the principal ones ordered by increasing complexity. Refer to [14] for the details.

POINT VEHICLE The body of the object to be moved is represented as a point in the space. Thus the state space $\mathbb{S}$ consists in the euclidean space $\mathbb{E}^2$ if we deal with land vehicles or $\mathbb{E}^3$ if we deal with aerial vehicles.

POINT VEHICLE WITH DIFFERENTIAL CONSTRAINTS This problem extends the point vehicle's problem by adding the constraints of the physical dynamic. For instance, constraints on acceleration, velocity, curvature, etc... when we want to model a real vehicle (whose shape is still approximated with a point).

JOGGER'S PROBLEM This kind of problems deals with the dynamic of a jogger that has a limited field of view. Consequently, in this case, we do not have a complete view of the scene and the path is updated as soon as the knowledge of the scene increases.

BUG'S PROBLEM This problem is an extreme case of the jogger's problem with a null field of view. Thus the scene updating can be done only when an obstacle is touched.

WEIGHTED REGIONS' PROBLEM This problem considers regions of the space as more desirable than others, rather than assuming completely obstructive obstacles as before. For instance this is the case of finding a path in an off-road behavior where the vehicle can move faster on certain terrains and slower over different configurations.

MOVER'S PROBLEM The vehicle is modeled as a rigid body, and so we need to add the dimensions for the spatial rotation of the body to the state space.

GENERAL VEHICLE WITH DIFFERENTIAL CONSTRAINTS This problem is a combination of the mover's problem and the point vehicle with

differential constraints: we add to the mover's problem also the physical constraints on the motion dynamic.

TIME VARYING ENVIRONMENTS These problems deal with moving obstacles.

MULTIPLE MOVERS This problem considers more than one vehicle. We then need to deal with different paths and with the problem of avoiding possible collisions between different movers. Actually we have to avoid collisions between the paths followed by different movers only if their collision point is reached by the movers at the same time.

2.2 ALGORITHM TYPES

We can divide the algorithms for motion planning in different types taking into account the specific problem they resolve. The algorithms belonging to a type can be further divided in different categories. For more details on the different algorithms see [14] and [6].

2.2.1 *Roadmap methods*

This kind of algorithms reduces the problem of motion planning to graph search algorithms. The state space is approximated with a certain graph in order to find a solution in terms of a polygonal chain.

Visibility graph

The visibility graph is one of the most known roadmap methods. The nodes of the graph correspond to the vertices of each polygonal obstacles in the considered scenario. The edges of the graph correspond to linear segments between pair of vertices that do not intersect any obstacle. The Dijkstra's algorithm is then usually considered to compute the *shortest path* between two vertices of the graph [9]. Note that the shortest path associated to the visibility graph in a planar configuration is the absolute shortest path from the start to the goal position with respect to the considered scenario, see e.g., [7]. While this method finds the optimal solution (with respect to a *distance* criterion) in the planar case, it does not properly scale in a 3-dimensional setting.

Edge sample visibility graph

The edge sample visibility graph is an extension of the visibility graph method to the 3-dimensional case. The main idea consists in distributing a discrete set of points along the edges of the obstacles by considering a certain density. The visibility graph and related shortest path of this configuration are then computed, but the corresponding solution is not optimal as in the planar case.

Voronoi roadmap

This method builds a graph that is kept equidistant to the obstacles, using VDs as base method for constructing it. We discuss VDs in detail in Section 3.3 and Voronoi roadmap method in Section 5.1.

Silhouette method

This method was developed by Canny [5]. It is not useful for practical uses but just for proving algorithmic bounds because it is proven to be complete in any dimension. It works sweeping the space with a line (plane in 3-dimensional space) perpendicular to the segment between $\mathbf{s}$ and $\boldsymbol{e}$ and building the shape of the obstacles when the sweeping line intersects them.

2.2.2 *Cell decomposition*

This method decomposes $\mathbb{S}_{free}$ in smaller convex polygons - i.e. trapezoids cylinders or balls - that are connected by a graph, then searches a solution in such graph. A cell decomposition method can be exact or approximate, the former kind operates occupying all $\mathbb{S}_{free}$ with the graph structure, the latter one can occupy also portions of $\mathbb{S} \setminus \mathbb{S}_{free}$ or all $\mathbb{S}$. Then the various polygons are labelled as obstacle-empty, inside obstacle or partially occupied by obstacles.

2.2.3 *Potential field methods*

This kind of methods operates assigning a potential field on every region of the space, the lowest potential is assigned to the goal point $\boldsymbol{e}$ and to the obstacles is assigned an high potential value. Then the path is calculated as a trajectory of a particle that reacts to those potentials, it is repelled by the obstacles and attracted by the end point.

2.2.4 *Probabilistic approaches*

This kind of methods uses probabilistic techniques for exploring the space of solutions and finding a good approximation of the optimal solution. In our project we provide also a mixed roadmap-probabilistic method, see Section 3.4 and Section 5.6 for further details.

2.2.5 *Rapidly-expanding Random Tree (RRT)*

This Method operates by doing a stochastic search, starting from the reference frame of the object to be moved and expanding a tree through the random sampling of the state space.

2.2.6 *Decoupled trajectory planning*

This kind of algorithms operates in a two-step way. First a discrete path through the state space is found, then the path is modified for adapting it to the dynamics constraints - i.e. the trajectory is constructed.

2.2.7 *Mathematical programming*

This method threats the trajectory planning problem as a numerical optimization problem, using methods like nonlinear programming to find the optimal solution.

2.3 PATH PLANNING

In our project we concentrate on a subsect of the motion planning problem, the *path planning* problem that consists [6] in determining a parametric curve

$$\mathbf{C} : [a, b] \subset \mathbb{R} \rightarrow \mathbb{S}$$

such that $\mathbf{C}(a) = \mathbf{s}$ coincides with the desired starting configuration, $\mathbf{C}(b) = \boldsymbol{e}$ the desired end configuration and the image of $\mathbf{C}$ is a subset of $\mathbb{S}_{free}$, in other words

$$\mathbf{C}(u) \in \mathbb{S}_{free} \quad \forall u \in [a, b].$$

In principle the space of the states $\mathbb{S}$ can be of any dimension, for instance if we focus on the piano movers' problem the state is composed

by 3 dimensions for the position and other 3 dimensions for the rotation of the object [21]. Also the curve $\mathbf{C}$ can be parameterized in any way.

In this project we concentrate on the problem of path planning where the state space is $\mathbb{S} = \mathbb{E}^3$ and the curve is parameterized in $[0, 1]$. Thus we find a curve from one point $\mathbf{s} \in \mathbb{E}^3$ to another point $\boldsymbol{e} \in \mathbb{E}^3$ avoiding obstacles. The object that we move is considered just as a point.

Part II

PROJECT

3

PREREQUISITES

3.1 SPLINES AND B-SPLINES

A *spline* is a piecewise polynomial function with prescribed regularity on its domain.

More formally we define a spline [8][11][27][26]

$$s : [a, b] \subset \mathbb{R} \to \mathbb{R}$$

as follows. We have a partition of that interval defined by the *breakpoints*

$$\tau = \{\tau_0, \ldots, \tau_\ell\}$$

such that $a = \tau_0 < \tau_1 < \cdots < \tau_{\ell-1} < \tau_\ell = b$ forming ℓ intervals

$$I_i = \begin{cases} [\tau_i, \tau_{i+1}) & \text{if } i = 0, \ldots, \ell - 2 \\ [\tau_i, \tau_{i+1}] & \text{if } i = \ell - 1 \end{cases}$$

is possible to define the following spaces:

PIECEWISE POLYNOMIAL FUNCTIONS SPACE $P_{m,\tau}$ is the space of the functions that are polynomials of maximum degree m in each interval I_i of the partition, formally:

$$P_{m,\tau} = \{f : [a, b] \to \mathbb{R} \mid \exists p_0 \ldots p_{\ell-1} \in \Pi_m \text{ such that } f(t) = p(t),\ \forall t \in I_i,\ i = 0 \ldots \ell - 1\}$$

where Π_m is the space of the polynomials of degree $\leqslant m$. The dimension of $P_{m,\tau}$ is

$$\dim(P_{m,\tau}) = \ell(m + 1)$$

because the dimension of Π_m is $m + 1$.

CLASSIC SPLINE SPACE $S_{m,\tau}$ is the space of the piecewise polynomial functions of degree m that have continuity C^{m-1} in the junctions of the intervals, formally:

$$S_{m,\tau} = P_{m,\tau} \cap C^{m-1}[a,b].$$

The dimension of this space is

$$\ell(m+1) - (\ell-1)\cdot m \;=\; \ell + m. \tag{3.1}$$

GENERALIZED SPLINE SPACE $S_{m,\tau,M}$ is the space of piecewise polynomial functions of degree m with a prescribed regularity at each breakpoint ranging from -1 to $m-1$. The regularity is prescribed by the multiplicity vector

$$M = \{m_1, \ldots, m_{\ell-1}\}, \quad m_i \in \mathbb{N}, \quad 1 \leqslant m_i \leqslant m+1$$

as follows,

$$\begin{aligned} S_{m,\tau,M} = \{ & f : [a,b] \to \mathbb{R} \;\mid\; \exists p_0 \ldots p_{\ell-1} \in \Pi_m \text{ such that} \\ & f(t) = p(t),\ \forall t \in I_i,\ i = 0 \ldots \ell-1 \text{ and} \\ & \qquad p_{i-1}^{(j)}(\tau_i) = p_i^{(j)}(\tau_i),\ j = 0, \ldots, m - m_i,\ i = 1, \ldots, \ell-1\}. \end{aligned}$$

The dimension of the space is equal to

$$\dim(S_{m,\tau,M}) = \ell(m+1) - \sum_{i=1}^{\ell-1}(m - m_i + 1) = m + \mu + 1 \qquad (\mu = \sum_{i=1}^{\ell-1} m_i)$$

and is true that

$$\Pi_m \subseteq S_{m,\tau} \subseteq S_{m,\tau,M} \subseteq P_{m,\tau},$$

in particular:

- if $m_i = 1$ for all $i = 1, \ldots, \ell-1$, then $S_{m,\tau,M} = S_{m,\tau}$;
- if $m_i = m+1$ for all $i = 1, \ldots, \ell-1$, then $S_{m,\tau,M} = P_{m,\tau}$.

3.1.1 *Truncated-powers basis for classic splines*

A truncated power $(t-\tau_i)_+^m$ is defined by

$$(t-\tau_i)_+^m = \begin{cases} 0, & \text{if} \quad t \leqslant \tau_i \\ (t-\tau_i)^m, & \text{otherwise.} \end{cases}$$

Is possible to demonstrate that the functions

$$g_i(t) = (t-\tau_i)_+^m) \ \in S_{m,\tau}, \quad i = 1,\dots,\ell-1$$

are linearly independents, and that

$$1, t, t^2, \dots, t^m, (t-\tau_1)_+^m, \dots, (t-\tau_{\ell-1})_+^m$$

form a basis for the classic spline space [8]. Then a generic element $s \in S_{m,\tau}$ can be expressed as follows,

$$s(t) = \sum_{i=0}^{m} c_i t^i + \sum_{j=1}^{\ell-1} d_j (t-\tau_j)_+^m \qquad \begin{matrix} c_i \in \mathbb{R},\ i = 0,\dots,m \\ d_j \in \mathbb{R},\ j = 1,\dots,\ell-1. \end{matrix} \tag{3.2}$$

3.1.2 *B-splines basis for classic splines*

B-splines are a specific basis wich can be alternatively used to represent any generalized spline [8][11][27][26]. In this paragraph however we consider only their definition to generate the classic spline space $S_{m,\tau}$. Furthermore in some textbook, for notational convenience, the *order*= $m+1$ is considered.

For defining the B-splines [8] we need to extend the partition vector $\tau = \{\tau_0, \cdots, \tau_\ell\}$ with m knots to the left and m to the right, so we define a new vector, usually called *extended knot* vector,

$$T = \{t_0, \dots, t_{m-1}, t_m, \dots, t_{n+1}, t_{n+2}, \dots, t_{n+m+1}\}$$

such that

$$t_0 \leqslant \cdots \leqslant t_{m-1} \leqslant \overset{\equiv\tau_0\equiv a}{t_m} < \cdots < \overset{\equiv\tau_\ell\equiv b}{t_{n+1}} \leqslant t_{n+2} \leqslant \cdots \leqslant t_{n+m+1}.$$

Since τ has $\ell+1$ elements, we can calculate the value of

$$n = \ell + m - 1.$$

Thus the dimension of $S_{m,\tau}$, for Eq. (3.1), is

$$dim(S_{m,\tau}) = \ell + m = n + 1$$

The $n+1$ basis $N_{i,m+1}(t)$ of the B-splines of degree m are defined, for $i = 0,\dots,n$, by the recursive formula:

$$N_{i,1}(t) = \begin{cases} 1, & \text{if} \quad t_i \leqslant t < t_{i+1} \\ 0, & \text{otherwise} \end{cases} \qquad i = 0,\dots,n+m$$

$$N_{i,r}(t) = \omega_{i,r-1}(t) \cdot N_{i,r-1}(t) \ + \ (1 - \omega_{i+1,r-1}(t)) \cdot N_{i+1,r-1}(t)$$

$$\text{i=0,...,n+m+1-3, r=2,...,m+1}$$

where

$$\omega_{i,r}(t) = \begin{cases} \frac{t-t_i}{t_{i+r}-t_i}, & \text{if } t_i \neq t_{i+r} \\ 0, & \text{otherwise.} \end{cases}$$

Then any function $s \in S_{m,\tau}$ can be expressed also as a linear combination of B-splines,

$$s(t) = \sum_{i=0}^{n} v_i N_{i,m+1}(t) \qquad , v_i \in \mathbb{R}, i = 0, \ldots, n. \tag{3.3}$$

3.1.3 *Spline curves*

A *spline curve* in the affine space $\mathbb{E}^d$ is the image of a parametric vector function $\mathbf{S} : [a, b] \to \mathbb{E}^d$ whose components are all splines belonging to a fixed spline space $S_{m,\tau}$. For $d = 3$

$$\mathbf{S}(u) = \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix}. \tag{3.4}$$

$\mathbf{S}(u)$ can be written as follows in the truncated-powers basis Eq. (3.2) replacing the coefficients c_i and d_i with points

$$\mathbf{S}(u) = \sum_{i=0}^{m} \mathbf{c}_i \cdot t^i + \sum_{j=1}^{\ell-1} \mathbf{d}_j \cdot (u - \tau_j)_+^m \qquad , \mathbf{c}_i \in \mathbb{E}^d,\ \mathbf{d}_j \in \mathbb{E}^d \quad i = 0, \ldots, m;\ j = 0, \ldots, \ell - 1 \tag{3.5}$$

where this representation is not practical because there isn't an intuitive correlation between the points $\mathbf{c}_i$, $\mathbf{d}_j$ and the curve itself. Moreover the determination of an interpolant to asset of argued points in $\mathbb{E}^d$ is not a well conditioned problem if this form is adopted [8]. To overcome those drawbacks the *B-splines basis* is adopted (Section 3.1.2).

We can apply control vertices to a spline expressed with the B-spline basis as in Eq. (3.3) replacing the coefficients v_i with points, in this case $\mathbf{S}(u)$ is represented as follows

$$\mathbf{S}(u) = \sum_{i=0}^{n} \boldsymbol{v}_i \cdot N_{i,m+1}(u) \qquad , \boldsymbol{v}_i \in \mathbb{E}^d,\ i = 0, \ldots, n. \tag{3.6}$$

The representation of Eq. (3.6) is more convenient than the previous one (Eq. (3.5)) because the curve $\mathbf{S}(u)$ roughly follows the shape given by the

points v_i. Those points are called *control vertices* because they are used to control the curve shape. Conformally the polygon they define is called *control polygon*.

3.2 B-SPLINES CURVES PROPERTIES

In this section we describe some properties of B-spline curves that we use for the development of the project.

3.2.1 *Convex Hull Property (CHP)*

The Convex Hull Property (CHP) states that a B-spline curve $\mathbf{S}(u)$ of order m, defined by the control polygon $v_0, v_1, \ldots, v_n$, is contained inside the union of the convex hulls composed of $m+1$ vertices of the control polygon [11]. If we call $\mathbf{Conv}(w_0, w_1, \ldots, w_j)$ the convex hull of the vertices $w_0, w_1, \ldots, w_j$ then we have

$$\begin{aligned} C_0 &= \mathbf{Conv}(v_0, v_1, \ldots, v_m) \\ C_1 &= \mathbf{Conv}(v_1, v_2, \ldots, v_{m+1}) \\ &\ldots \\ C_{n-m} &= \mathbf{Conv}(v_{n-m}, v_{n-m+1}, \ldots, v_n) \end{aligned}$$

and the area where $\mathbf{S}$ is contained is

$$C = C_0 \cup C_1 \cup \cdots \cup C_{n-m}$$

or in other words must be true

$$\mathbf{S}(u) \cap C = \mathbf{S}(u) \qquad \forall u \in [a, b]$$

whatever is the partition vector.

In Fig. 1 an example of control polygon is visible, together with the region C (in cyan) where an associated quadratic B-spline curve is located.

Note that the CHP holds also in 3-dimensional space - i.e. a quadratic B-spline in 3-dimensional space is contained inside a flat surface composed by the union of triangles. From degree 3 the area where $\mathbf{S}$ is contained is not anymore plane because it is composed of union of solid polyhedrons.

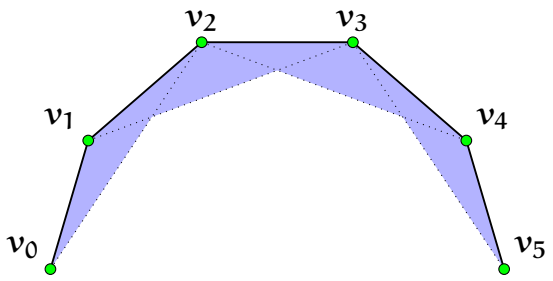


Figure 1.: Convex hull containing B-spline of degree 2

3.2.2 *Aligned vertices*

As in some parts of this project control polygons with sequences of aligned vertices have been adopted, in this section their specific effect on the curve shape is analyzed.

We can have the following situations:

m ALIGNED CONTROL VERTICES If m control vertices $v_i, \ldots, v_{i+m-1}$ of the control polygon are on the same line then the curve $\mathbf{S}$ touches the segment joining those vertices.

$m+1$ ALIGNED CONTROL VERTICES If $m+1$ control vertices $v_i, \ldots, v_{i+m}$ of the control polygon are on the same line then a polynomial arc of the curve $\mathbf{S}$ lays on the segment joining those vertices.

3.2.3 *Smoothness*

A function is said smooth of class C^d if it is possible to calculate the d-th derivative of it and if such derivative is continue. A function f that is not continue is said to be of class C^{-1}, a function that is continue until derivative d is said to be of class C^d, a function that is always continue for every derivative is said to be of class C^∞.

A B-spline curve of degree m with n control vertices is composed by $n-m$ polynomial segments, one for each interval

$$[t_i, t_{i+1}] \qquad i = m, \ldots, n+1$$

this mean that $\mathbf{S}(u)$ is C^∞ for

$$u \in (t_i, t_{i+1}) \qquad i = m, \ldots, n+1.$$

Note that, if we use generalized B-spline curves, an interval $[t_i, t_{i+1}]$ can also degenerate in just a point if we have a knot multiplicity > 1, in such case there isn't a polynomial segment. On every breakpoint τ_i with $i = 1, \ldots, n+m$ we have that the curve has smoothness[1] C^{m-1}.

In our project we don't use generalized B-spline curves, so globally a curve of degree m have smoothness

$$C^{m-1}.$$

3.2.4 *End point interpolation*

In general a B-spline curve with control vertices

$$\boldsymbol{v}_0, \ldots, \boldsymbol{v}_n$$

and extended knot vector

$$T = \{t_0, \ldots, t_{m-1}, t_m, \ldots, t_{n+1}, t_{n+2}, \ldots, t_{n+m+1}\}$$

does not necessarily interpolate any control vertex v_i, neither the first and the last one. But we are interested in using B-spline for representing paths from one point to another. So it should be a nice feature to have that the curve defined in the domain $[a, b]$ is shaped such that

$$\begin{cases} \mathbf{S}(u) = \boldsymbol{v}_0 & \text{for } u = a \\ \mathbf{S}(u) = \boldsymbol{v}_n & \text{for } u = b. \end{cases} \tag{3.7}$$

We can obtain [8] the conditions of Eq. (3.7) if we impose on the extended partition vector T:

$$t_0 = \cdots = t_{m-1} = \overset{\equiv a}{t_m} < \cdots < \overset{\equiv b}{t_{n+1}} = t_{n+2} = \cdots = t_{n+m+1}$$

in other words we want

$$T = \{\overbrace{a, \ldots, a}^{m}, t_{m+1}, \ldots, t_n, \overbrace{b, \ldots, b}^{m}\}$$

1 If we use generalized B-spline curves, it has smoothness C^{m-r} where r is the multiplicity of the knot [11].

3.2.5 *Curvature and torsion*

Since we are interested in comparing different curves, we need to recognize if a certain curve is a *good* or a *bad* one. One factor that characterizes a certain curve can be its smoothness (Section 3.2.3) - i.e. a C^3 curve is better than a C^2 curve - but this isn't enough for comparing curves. Usually *curvature* and *torsion* are used for this aim [10][27]. Both are scalar quantities defined on sufficiently smooth parametric curves for each value of the parameter, and they do not depend on the selected parametrization.

For a generic parametric curve[2] $\mathbf{S}(u)$ defined for $u \in [a, b]$ given the notation $\wedge$ for the vector product and for Eq. (3.4)

$$\dot{\mathbf{S}}(u) = \frac{d}{du}\mathbf{S}(u) = \begin{bmatrix} \frac{d}{du}\mathbf{x}(u) \\ \frac{d}{du}\mathbf{y}(u) \\ \frac{d}{du}\mathbf{z}(u) \end{bmatrix}.$$

we define the curvature $\kappa(u)$ and, in points with non vanishing curvature, the torsion $\tau(u)$ as

$$= \begin{cases} \kappa(u) = \dfrac{\left\|\dot{\mathbf{S}}(u) \wedge \ddot{\mathbf{S}}(u)\right\|_2}{\left\|\dot{\mathbf{S}}(u)\right\|_2^3} & (3.8) \\ \tau(u) = \dfrac{\det\left[\dot{\mathbf{S}}(u), \ddot{\mathbf{S}}(u), \dddot{\mathbf{S}}(u)\right]}{\left\|\dot{\mathbf{S}}(u) \wedge \ddot{\mathbf{S}}(u)\right\|_2} = \dfrac{\left(\dot{\mathbf{S}}(u) \wedge \ddot{\mathbf{S}}(u)\right) \cdot \dddot{\mathbf{S}}(u)}{\left\|\dot{\mathbf{S}}(u) \wedge \ddot{\mathbf{S}}(u)\right\|_2} & (3.9) \end{cases}$$

Eq. (3.8) and Eq. (3.9) describe completely the behavior of $\mathbf{S}(u)$ locally for each value of u. Curvature and torsion have also a geometric interpretation: for each value $\tilde{u}$ of the parameter u, the inverse $\frac{1}{\kappa(\tilde{u})}$ of the curvature is the radius of curvature of $\mathbf{S}$ at $\mathbf{S}(\tilde{u})$ - i.e. the radius of the osculating circle tangent in that point to the curve which belongs to the plane where the curve is bending and that is situated on the inner side of its turn - $\tau(\tilde{u})$ indicate (if $\kappa(\tilde{u}) \neq 0$) how sharply the plane where the curve lies is rotating.

The value of $\kappa(u)$ can be only non negative, while $\tau(u)$ is a signed quantity.

Two curves of same smoothness can be compared using the plots of curvature and torsion, in general curves that have lower peaks of $\kappa(u)$ and $\tau(u)$ are better than curves with higher peaks.

2 Thus also a B-spline curve.

3.2.6 *Arc length*

Sometimes we are interested in evaluating the length of a generic parametric curve[3] $\mathbf{S}(u)$ defined for $u \in [a, b]$. We can obtain such length, called *arc length*, calculating the integral

$$\int_a^b \left\| \dot{\mathbf{S}}(u) \right\|_2 du.$$

We can approximate this value using a discrete tabulation of the curve $\mathbf{S}(u)$ and some integrating method like the *trapezoidal rule* [24][30].

3.3 VORONOI DIAGRAMS

In this section we introduce Voronoi Diagrams (VDs), an important structure used in the project. VDs [7] provide a method for creating a partition of the space using distances from a set of input points called *sites*. Formally we have a set

$$S = \{\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_n\} \subset \mathbb{E}^d$$

of n sites in the euclidean space of dimension d, and we build a set of n Voronoi *cells*[4]

$$Vor(S) = \{V(\mathbf{s}_0), \ldots, V(\mathbf{s}_n)\} \subset 2^{\mathbb{E}^d}$$

such that

$$V(\mathbf{s}_i) = \{\mathbf{p} \in \mathbb{E}^d \ : \ \|\mathbf{p} - \mathbf{s}_i\|_2 < \left\|\mathbf{p} - \mathbf{s}_j\right\|_2 \ \forall \mathbf{s}_j \neq \mathbf{s}_i\}$$

is the set of the points in $\mathbb{E}^d$ closer to $\mathbf{s}_i$ than to any other site.

Figure 2 is an example of the VD built on some random sites, on the figure the dashed lines are edges that go to infinite.

The most important algorithm for calculating VDs is the *Fortune's sweeping line* algorithm that builds the diagram in $O(n \log n)$ and it is optimal. The algorithm involves building $Vor(S)$ incrementally while sweeping the space, see Fig. 3. Every time that the sweeping line finds a site the algorithm creates a parabola using the site as focus and the sweeping line as directrix. Such parabolas, or better the arcs between each intersection of them, constitute the *beach line*. A parabola disappears from the scene

3 See Footnote 2.

4 $2^{\mathbb{E}^d}$ is the power set of $\mathbb{E}^d$, the set of all the subsets of $\mathbb{E}^d$.

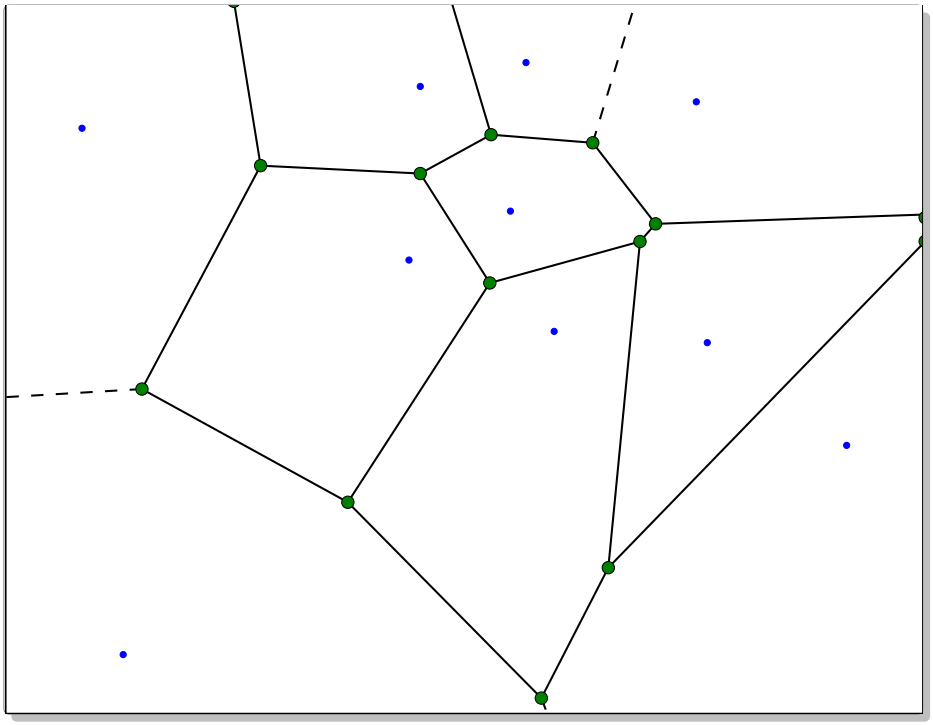


Figure 2.: Example of a VD, dashed lines are infinite edges.

when the associated arc vanish. The evolution of the intersection points on the beach line constitutes the edges of the VD, and each point where an arc of the beach line disappears constitutes a vertex of the VD. Refer to [7] and [12] for details about the Fortune's algorithm.

One property of VDs is that $V(\mathbf{s}_i)$ can be a closed or an open area - i.e. the edges of the cells can be infinite - it is important to keep this in mind if we want to interpret $Vor(S)$ as a graph. On that case the graph will have edges that go to infinite. We call such graph $G(Vor(S))$.

Another property is that if we have $d+1$ sites $\mathbf{s}'_0, \ldots, \mathbf{s}'_d$ that lay on the surface of a $(d-1)$-sphere[5] that doesn't have any other site on the interior, then the center point of the $(d-1)$-sphere is the vertex shared only between the $d+1$ cells $V(\mathbf{s}'_0), \ldots, V(\mathbf{s}'_d)$ [7]. This is not true for less than $d+1$ sites on a $(d-1)$-sphere because they are not enough to define it univocally, but is possible to have $n > d+1$ sites on a $(d-1)$-sphere, on that case the center of such $(d-1)$-sphere is the shared vertex of the cells corresponding to the n sites. This is important for reasoning

5 A circumference in 2-dimensional space, a sphere in 3-dimensional space, an hyper-sphere in n-dimensional space with $n \geqslant 3$.

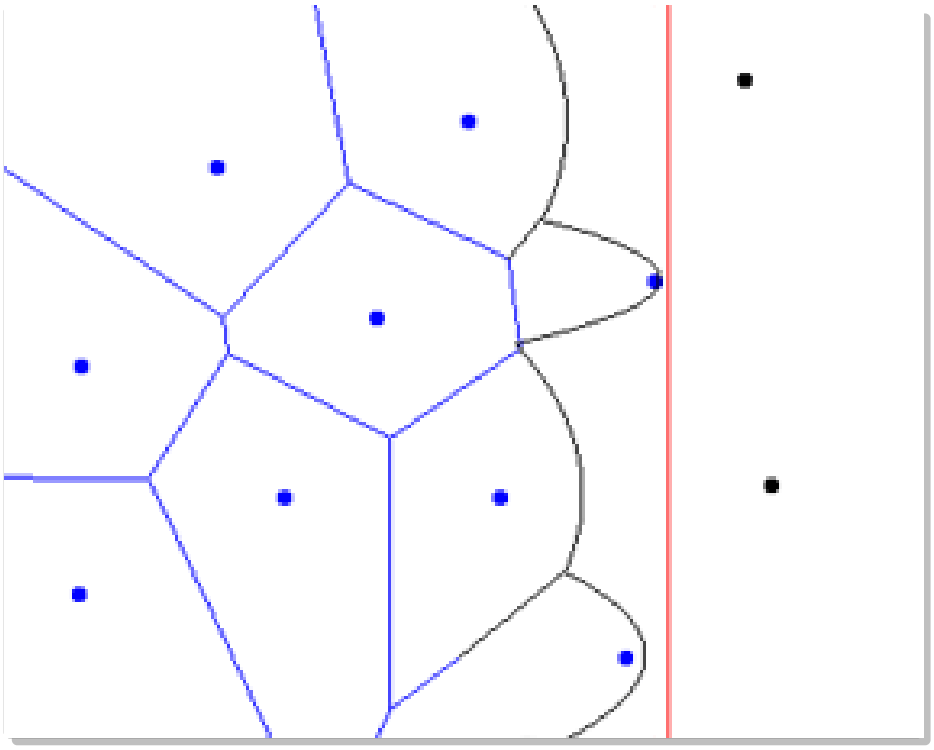


Figure 3.: Fortune's algorithm execution

about the topography of $G(Vor(S))$ because if we allow more than $d+1$ sites on a $(d-1)$-sphere then the maximum degree $\Delta(G(Vor(S)))$ of the graph can be arbitrarily big (until to the number of vertices). However the fact that we work with coordinates in $\mathbb{E}^d$ legitimizes the restriction[6] of not allowing more than $d+1$ sites on an $(d-1)$-sphere, limiting $\Delta(G(Vor(S)))$ to $d+1$.

3.4 STATISTICAL METHODS

In this section we make a short introduction to Monte Carlo Method (MCM) [22][29][2] and Simulated Annealing (SA) [19][16], two statistical methods for calculating unknown quantities and finding minimum of functions. Also we introduce Lagrangian Relaxation (LR) [32] a method for transforming a constrained optimization problem in an unconstrained optimization problem increasing the state space dimension.

3.4.1 *Notes on probabilities*

Probable Error (PE)

For a random variable X normally distributed with

- mean μ;
- variance σ;

for

$$r = 0.6745\sigma$$

we have that

$$\mathbf{P}(|X-\mu| < r) = \mathbf{P}(|X-\mu| > r) = 0.5$$

So values of X that deviate from μ less or more than r have the same probability, and r identifies the most PE in a normal distribution.

6 We can also relax this restriction and in case create multiple nodes connected by zero-distance edges on the graph.

Probability central limit theorem (PCLT)

Consider N independent and *identically-distributed* random variables $X_1, X_2, \ldots, X_N$, with same mean and same variance

$$\begin{aligned} \mathbf{E}[X_1] = \mathbf{E}[X_2] = \cdots = \mathbf{E}[X_N] &= m \\ \mathbf{Var}(X_1) = \mathbf{Var}(X_2) = \cdots = \mathbf{Var}(X_N) &= b^2. \end{aligned}$$

Consider the sum of those random variables:

$$Y = X_1 + X_2 + \cdots + X_N$$

we have that

$$\begin{aligned} \mathbf{E}[Y] &= \mathbf{E}[X_1 + X_2 + \cdots + X_N] = Nm \\ \mathbf{Var}(Y) &= \mathbf{Var}(X_1 + X_2 + \cdots + X_N) = Nb^2. \end{aligned}$$

Consider now a normally distributed random variable Z with parameters:

$$\begin{aligned} \mu &= Nm \\ \sigma &= b\sqrt{N} \end{aligned}$$

with Probability Density Function (PDF) $p_Z(x)$.

The *PCLT* affirms that for N big enough, and for every interval (x_1, x_2):

$$\mathbf{P}(x_1 < Y < x_2) \approx \int_{x_1}^{x_2} p_Z(x)dx. \tag{3.10}$$

So the sum of an elevate number of identically-distributed random variables is a random variable with normal distribution with mean Nm and variance Nb^2 even if $X_1, X_2, \ldots, X_N$ aren't normally distributed.

3.4.2 *Monte Carlo Method (MCM)*

Suppose that you need to calculate an unknown quantity m, you need to find a random variable X such that:

$$\mathbf{E}[X] = m.$$

If you have such distribution with variance:

$$\mathbf{Var}(X) = b^2$$

it is possible to formalize the following passages.

Consider N random variables $X_1, X_2, \ldots, X_N$ that have distribution identical to the distribution of X. For the PCLT Eq. (3.10) we have that, for N big enough

$$Y = X_1 + X_2 + \cdots + X_N$$

is normally distributed with parameters

$$\begin{aligned} \mu &= Nm \\ \sigma &= b\sqrt{N}. \end{aligned}$$

For the *three sigma rule* [25] we have that:

$$\mathbf{P}\left(\mu - 3\sigma < Y < \mu + 3\sigma\right) \approx 0.997$$

that is

$$\mathbf{P}\left(Nm - 3b\sqrt{N} < Y < Nm + 3b\sqrt{N}\right) \approx 0.997$$

dividing by N:

$$\mathbf{P}\left(m - \frac{3b}{\sqrt{N}} < \frac{Y}{N} < m + \frac{3b}{\sqrt{N}}\right) \approx 0.997$$

that is

$$\mathbf{P}\left(\left|\frac{Y}{N} - m\right| < \frac{3b}{\sqrt{N}}\right) \approx 0.997$$

and so:

$$\mathbf{P}\left(\left|\frac{1}{N}\sum_{i=1}^{N} X_i - m\right| < \frac{3b}{\sqrt{N}}\right) \approx 0.997. \tag{3.11}$$

Equation (3.11) asserts that, if you extract a sample for each random variable X_i, the arithmetic mean of those values is approximately equal to m. Moreover the error of such approximation is equal to $3b/\sqrt{N}$, that tend to 0 increasing N. It is also possible to further reduce the uncertainty ($1 - 0.997 = 0.003$) by increasing the number k of sigma used for the approximation and evaluating the error $kb/\sqrt{N}$.

In practice, since the random variables X_i have the same distribution of X, it is sufficient to extract N samples from X for reaching to the same conclusions.

The Monte Carlo Method (MCM) is constituted by the following procedure, to be adapted according to the problems:

1. find the distribution X having desired quantity m as mean value and b^2 as variance;
2. extract N samples from X, with N big enough to have an error small as desired;
3. the arithmetic mean of those N samples is the approximation of the desired value m.

Basically we transformed the problem from *calculate* m to *find the distribution* X, or anyway the N samples distributed accordingly to X.

If we want to characterize more in detail the error committed taking N samples, we can recur to PE. If we set $k = 0.6745$ then we have that

$$\mathbf{P}\left(\left|\frac{1}{N}\sum_{i=1}^{N} X_i - m\right| < \frac{0.6745 \cdot b}{\sqrt{N}}\right) \approx 0.5$$

and so

$$r_N = \frac{0.6745 \cdot b}{\sqrt{N}}$$

indicates how much the value $\frac{1}{N}\sum_{i=1}^{N} X_i$ deviates from the desired value m. Such value characterize the absolute error

$$\left|\frac{1}{N}\sum_{i=1}^{N} X_i - m\right|$$

committed taking N samples.

MCM is useful to simulate events that have an high degree of uncertainty in the inputs or an high degree of liberty in the state. For instance integrating numerically a function with many dimensions or Simulated Annealing (SA) (Section 3.4.3).

3.4.3 *Simulated Annealing (SA)*

The SA is a method used to find the global maximum or minimum of a function. It is inspired by a method used in metallurgy that consists in heating and then cooling slowly a material for augmenting the dimensions of the crystals and improving the chemico-physical properties. The function to be optimized can be defined in a multiple-dimensional space.

Statistical thermodynamic

For describing the base principles of statistical thermodynamic we consider an example. In a one-dimensional lattice every point is a particle with a value of spin that can be *up* or *down*. If the lattice has N points then the system can be in 2^N different configurations, where each one of those configurations corresponds to a value of energy, for instance:

$$E = B(n_+ - n_-)$$

where B is some constant, n_+ is the number of particles with spin *up* and n_- is the number of particles with spin *down*.

The probability $B(\sigma)$ of finding the system in a certain configuration σ is given by the distribution of *Boltzmann-Gibbs*:

$$P(\sigma) = Ce^{-E_\sigma/T} \tag{3.12}$$

where E_σ is the energy of the configuration, T is the temperature[7] and C is a normalization constant.

The average energy of the system is then:

$$\begin{aligned} \bar{E} &= \frac{\sum_\sigma E_\sigma P(\sigma)}{\sum_\sigma P(\sigma)} \\ &= \frac{\sum_\sigma E_\sigma e^{-E_\sigma/T}}{\sum_\sigma e^{-E_\sigma/T}}. \end{aligned}$$

The computation of the value of $\bar{E}$ can be difficult with an high number of states, but it is possible to create a MCM simulating the random fluctuation between the states such that the distribution given by Eq. (3.12) is respected. Starting from an arbitrary initial configuration, after a certain number of *Monte Carlo trials*, the method converges to the equilibrium status $\bar{E}$ and it continues to fluctuate around it. SA is a method of this kind.

Simulated Annealing (SA) algorithm

SA operates on a system starting from a certain initial state s_0, then it executes a series of iterations where a neighbour of the state is evaluated and, with a certain distribution of probability, the system is moved in the new state or not.

7 The real Boltzmann-Gibbs distribution is $P(\sigma) = Ce^{-E_\sigma/kT}$ where k is the *Boltzmann constant* and T is the thermodynamic temperature, but for the example the temperature is a parameter not correlated to the physical world, so it is possible to ignore k.

A possible algorithm for a SA method is Algorithm 1. s_0 is the initial state; $temp$ is the function that assigns a temperature based on the current iteration number such that for low k the returned temperature is high and for high k the returned temperature is low; $neighbour$ is the function that returns a random neighbour of the current state; $uniform$ returns an uniformly-randomly chosen number in $[0, 1]$; P_a is the distribution of accepting probability, that depends on the energy of the current state, on the energy of the neighbour, and on the current temperature. In case of acceptation the neighbour becomes the current state and the process continues.

Algorithm 1 Simulated Annealing (SA)

```
1:  function ANNEAL(s_0)
2:      s ← s_0
3:      for k ← 0, kMax do
4:          T ← temp(k / kMax)
5:          sNew ← neighbour(s)
6:          if uniform(0, 1) < P_a(E(s), E(sNew), T) then
7:              s ← sNew
8:          end if
9:      end for
10:     return s
11: end function
```

The relation with the statistical thermodynamic is that P_a is chosen such that Eq. (3.12) holds[8], moreover $temp$ returns decreasing values of temperature with the succession of iterations. This explains the comparison with the metallurgy annealing.

Initially P_a was chosen such that

$$P_a(E(s), E(sNew), T) = \begin{cases} 1, & \text{if } E(sNew) < E(s) \\ e^{-(E(sNew)-E(s))/T}, & \text{otherwise} \end{cases}$$

but this isn't strictly necessary for developing a SA method.

8 A similar distribution is enough.

3.4.4 *Lagrangian Relaxation (LR)*

A general constrained discrete optimization problem can be expressed in the form:

$$\begin{aligned} &\underset{x}{\text{minimize}} && f(x) \\ &\text{subject to} && g(x) = 0. \end{aligned} \tag{3.13}$$

where $x \in X$ is the state of the system in a discrete space X, $f(x)$ is the function to minimize, and $g(x) = 0$ is the constraint. The functions can also be in a multidimensional discrete space, in that case the x is a vector $\mathbf{x} = (x_1, \ldots, x_n)$ of variables.

For solving this class of problems a *Lagrange relaxation* method can be used [3], it augments the variable space X by a *Lagrange multiplier* space Λ of dimension equal to the number of constraints - one in the Problem 3.13.

The *generalized discrete Lagrangian function* corresponding to the Problem 3.13 is:

$$L_d(x, \lambda) = f(x) + \lambda H(g(x)) \tag{3.14}$$

where λ is a variable in Λ, and if the dimension of Λ is more than one λ must be transposed in Eq. (3.14). $H(x)$ is a non negative function with the property that $H(0) = 0$ and aimed to transform g in a non negative function. For instance, it can be $H(g(x)) = |g(x)|$ or $H(g(x)) = g^2(x)$.

Under the previous assumptions the set of *local minima* in Problem 3.13 - that respect the constraints - coincides with the set of *discrete saddle point* in the augmented space. A point (x^*, λ^*) is a discrete saddle point if:

$$L_d(x^*, \lambda) \leqslant L_d(x^*, \lambda^*) \leqslant L_d(x, \lambda^*)$$

for all $x \in \mathcal{N}(x^*)$ and for all $\lambda \in \Lambda$, where $\mathcal{N}(x^*)$ is the set of all neighbours of x^*.

For resolving the optimization Problem 3.13 it is necessary to calculate all the discrete saddle points (x^*, λ^*) on the surface represented by Eq. (3.14) using some optimization method (i.e. simulated annealing). Then it is chosen the one that minimize $f(x^*)$, x^* is then the desired minimum.

3.5 INTERSECTIONS IN SPACE

We work in a spatial environment with polyhedral obstacles. Thus, in order to define admissible paths, first of all we need routines performing the following three basic geometric tasks:

1. establish if a point is in or out of a convex polyhedron;
2. check if a segment intersects a triangle;
3. establish whether two triangles intersect.

In the project we need to deal with three kinds of collision detection methods in 3-dimension euclidean space:

3.5.1 *Point inside convex polyhedron in 3-d space*

For testing if a point $\mathbf{p}$ is inside a convex polyhedron V with vertices $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ we use a method that rely on convex hulls [7][28].

Algorithm 2 Check if point $\mathbf{p}$ is inside convex polyhedron V

```
1:  function isPointInPolyhedron(p, V)
2:      inside ← True
3:      C ← convexHullVertices([p, v_1, v_2, ..., v_n])
4:      for all c ∈ C do
5:          if c = p then
6:              inside ← False
7:              break
8:          end if
9:      end for
10:     return inside
11: end function
```

Algorithm 2 performs the first task. It first computes the vertices of the convex hull of all the vertices of V plus the point $\mathbf{p}$ and then checks if $\mathbf{p}$ is one of them or not. If $\mathbf{p}$ is on the convex hull that means that $\mathbf{p}$ is external[9] to V because we have extended the convex hull formed by the vertices of V. Otherwise this means that $\mathbf{p}$ is inside V.

The cost of this algorithm is

$$\mathcal{O}(n \log n)$$

9 Or $\mathbf{p}$ coincides with a vertex of V.

where n is the number of vertices of V, because the cost for constructing the convex hull is [7] $\mathcal{O}(n \log n)$, and then we have another negligible term $\mathcal{O}(n)$ for the cycle on Line 4.

3.5.2 *Segment-triangle in 3-d space*

We need to deal with the intersection between a segment $S = \overline{\mathbf{a}_2\mathbf{b}_2}$ and a triangle $T = \triangle \mathbf{a}_1\mathbf{b}_1\mathbf{c}_1$. S and T can be in one of the following cases also summarized in Table 1:

case 1 S and T do not intersect and the plane containing T is not in the sheaf of planes generated by the line containing S;

case 2 S and T do not intersect and the plane containing T is in the sheaf of planes generated by the line containing S;

case 3 S and T intersect only at one point and the plane containing T is not in the sheaf of planes generated by the line containing S;

case 4 S and T intersect in one or infinite points and the plane containing T is in the sheaf of planes generated by the line containing S.

The discriminants among the cases are two, presence of intersection and coplanarity. Table 1.

	not coplanar	coplanar
not intersect	**case 1**	**case 2**
intersect	**case 3**	**case 4**

Table 1.: Relations between S and T

In Fig. 4 a **case 3** situation is shown where there is intersection in only one point $\mathbf{x}$. For establishing whether S and T intersect, we need to solve four equation in four unknowns [28] where we look for a point $\mathbf{x}$ being a convex linear combination of $\mathbf{a}_2$ and $\mathbf{b}_2$ and at the same time a convex linear combination of $\mathbf{a}_1$, $\mathbf{b}_1$ and $\mathbf{c}_1$. In other words when there is a collision, then there is a solution for the unknowns α, β, γ, δ, ζ of the system

$$\begin{cases} \alpha\mathbf{a}_2 + \beta\mathbf{b}_2 = \gamma\mathbf{a}_1 + \delta\mathbf{b}_1 + \zeta\mathbf{c}_1 \\ \alpha + \beta = 1 \\ \gamma + \delta + \zeta = 1 \end{cases} \tag{3.15}$$

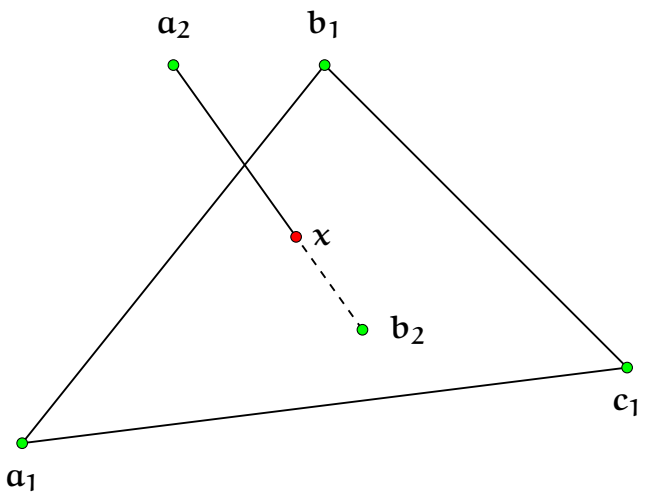


Figure 4.: Example intersection between a segment $\overline{a_2 b_2}$ and a triangle $\triangle a_1 b_1 c_1$.

with the further conditions

$$\begin{cases} \alpha \geqslant 0 \\ \beta \geqslant 0 \\ \gamma \geqslant 0 \\ \delta \geqslant 0 \\ \zeta \geqslant 0. \end{cases} \tag{3.16}$$

Note that the first equation of System 3.15 has vectorial coefficients a_2, b_2, a_1, b_1, c_1, so the system is of five unknowns in five equations. If System 3.15 has just one solution then we are on **case 1** or **case 3**, respectively when System 3.16 is fulfilled or not. If it has infinite solution then we are on **case 2** or **case 4**, again depending on the fulfillment of (3,16) or not. Finally, if it has not solution then S or T are degenerated.

We are interested in finding only **case 3** collisions because for simplicity we consider the special case of a segment that lays on the surface of

a triangle as not colliding with it, and for coherence we restrict the conditions of System 3.16 to

$$\begin{cases} \alpha > 0 \\ \beta > 0 \\ \gamma > 0 \\ \delta > 0 \\ \zeta > 0. \end{cases} \tag{3.17}$$

System 3.15 can be simplified in the three equations

$$\left\{ \alpha \mathbf{a}_2 + (1-\alpha)\mathbf{b}_2 = \gamma \mathbf{a}_1 + \delta \mathbf{b}_1 + (1-(\gamma+\delta))\mathbf{c}_1 \right. \tag{3.18}$$

in the unknowns α, γ and δ with the relative conditions

$$\begin{cases} \alpha > 0 \\ \alpha < 1 \\ \gamma > 0 \\ \delta > 0 \\ \gamma + \delta < 1. \end{cases} \tag{3.19}$$

Algorithm 3 Find intersection between segment S and triangle T

```
1: function INTERSECT(S, T)
2:     intersect ← False
3:     coordinates ← ∅
4:     if (α, γ, δ) ← solve(System 3.18) then
5:         if satisfy(System 3.19) then
6:             intersect ← True
7:             coordinates ← (γ, δ, 1 − (γ + δ))
8:         end if
9:     end if
10:    return (intersect, coordinates)
11: end function
```

Thus, Algorithm 3 which performs Task 2 basically consists in solving System 3.18 with the parameters $\mathbf{a}_2$, $\mathbf{b}_2$, $\mathbf{a}_1$, $\mathbf{b}_1$ and $\mathbf{c}_1$ from S and T; and

then in checking if the solution is admissible. The condition of Line 4 is True if System 3.18 has solution and if that is unique.

We have also the positive secondary effect that from the solution (α, γ, δ) of System 3.18 we can extract the barycentric coordinates $(\gamma, \delta, 1 - (\gamma + \delta))$ of the intersection point $\mathbf{x}$ on the system of the vertices $\mathbf{a_1}$, $\mathbf{b_1}$, $\mathbf{c_1}$ of T.

3.5.3 *Triangle-triangle in 3-d space*

We are interested in detecting collisions between two triangles $T_1 = \triangle \mathbf{a_1 b_1 c_1}$ and $T_2 = \triangle \mathbf{a_2 b_2 c_2}$ in 3-dimensional space. First of all consider the coplanarity relation between the two triangles, we have the cases:

case 1 T_1 and T_2 are contained by the same plane;

case 2 T_1 and T_2 are contained by different planes.

To simplify the problem we decided - analogously to the case of intersection between segment and triangle - that when we are on **case 1** we consider T_1 and T_2 not intersecting in any case, also if from a geometrical point of view they share points. After this premise we can assert that the possible relation between T_1 and T_2 can be exclusively one of the types [28]:

type 0 T_1 and T_2 do not intersect;

type 1 two edges of T_1 intersect the plane section delimited by T_2, or vice versa;

type 2 one edge of T_1 intersects the plane section delimited by T_2 and one edge of T_2 intersects the plane section delimited by T_1.

On Fig. 5 and Fig. 6 we can see two examples of **type 1** and **type 2**, respectively. For estabiliscing if T_1 and T_2 intersect we need to check if every edge of T_1 intersects T_2 and every edge of T_2 intersects T_1. If we found at least one edge that intersects with one triangle then T_1 and T_2 intersect. Algorithm 4 executes such check, the function *intersect* on Line 3 and Line 8 is the intersection check between a segment and a triangle done by Algorithm 3.

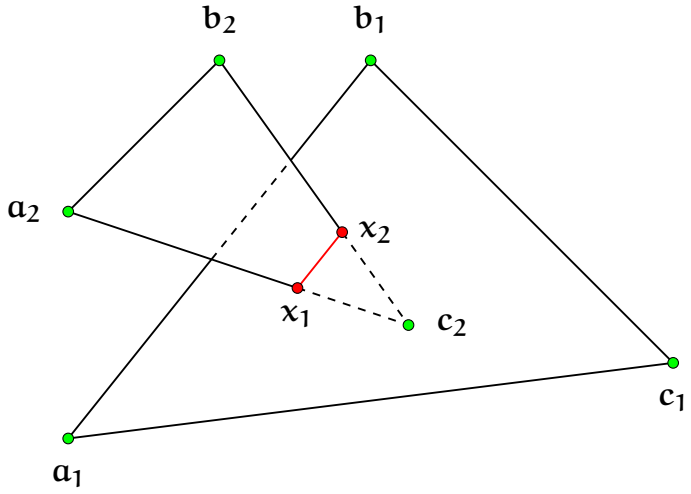


Figure 5.: Example of **type 1** intersection between a triangle $T_1 = \triangle a_1 b_1 c_1$ and another triangle $T_2 = \triangle a_2 b_2 c_2$.

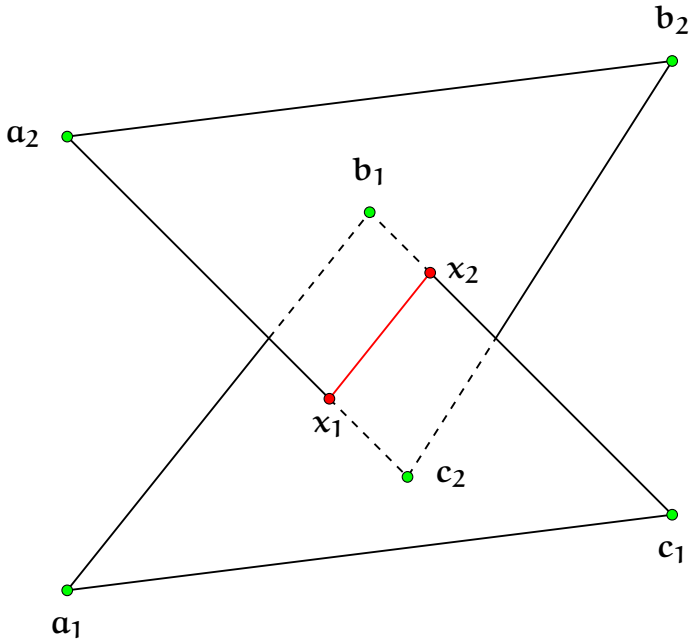


Figure 6.: Example of **type 2** intersection between a triangle $T_1 = \triangle a_1 b_1 c_1$ and another triangle $T_2 = \triangle a_2 b_2 c_2$.

Algorithm 4 Find intersection between triangle T_1 and triangle T_2

```
1:  function INTERSECT(T_1 = (a_1, b_1, c_1), T_2 = (a_2, b_2, c_2))
2:      for all S ∈ {a_1b_1, b_1c_1, c_1a_1} do
3:          if intersect(S, T_2) then
4:              return True
5:          end if
6:      end for
7:      for all S ∈ {a_2b_2, b_2c_2, c_2a_2} do
8:          if intersect(S, T_1) then
9:              return True
10:         end if
11:     end for
12:     return False
13: end function
```

4

SCENE REPRESENTATION

The problem of scene description basically consists in fixing a representation of the obstacles and of the path, besides in establishing the structures adopted for their storage.

4.1 BASIC ELEMENTS AND PATH

First of all, since we are interested in spatial path planning, all the point coordinates are in $\mathbb{E}^3$. Also we concentrate on B-splines because we want a standard representation for the output of the algorithm, the path between a start point s and an end point e. Actually B-splines curves are the standard adopted in CAD and CAGD systems [17][18].

Now the structures which uniquely identify a B-spline curve are three: its degree m, the associated control polygon and the extended knot vector

Regarding the degree of the curve, we give to the users the possibility of choosing among quadratics ($m = 2$), cubics ($m = 3$) and quartics ($m = 4$). The users choose also the starting and ending points s and e respectively, associated with the parameter values $t_0 = \cdots = t_m$ and $t_{n+1} = \cdots = t_{n+m+1}$.

The number of vertices and the other vertices themselves come from the algorithm, and they depend on the position of s and e and on the obstacles, see Section 5.1 for details.

The knots are generated automatically using one of two methods described in Section 5.4.

So for the curve we memorize only the control vertices P and the degree m. As usual in any computer graphic system, when we want its plotting, we tabulate $\mathbf{S}$ for a certain number[1] of values of t and then we draw the polygonal chain that connects them.

1 Enough for having a smooth look.

4.2 BASIC OBSTACLE REPRESENTATION

Apart the curve, in the scene we need to represent the obstacles. We call Obs the set of all obstacle in scene. We choose to represent each obstacle $Ob \in Obs$ as a set of triangular faces that we call Obstacle Triangular Faces (OTFs), each one containing three vertices. Resuming we have

$$\begin{aligned} Obs &= \{Ob_0, \ldots, Ob_{\#Obs}\} \\ Ob_i &= \{Otf_{i,0}, \ldots, Otf_{i,\#Otf_i}\} && i = 0, \ldots, \#Obs \\ Otf_{i,j} &= \{\mathbf{p}_{i,j,0}, \mathbf{p}_{i,j,1}, \mathbf{p}_{i,j,2}\} && i = 0, \ldots, \#Obs; \quad j = 0, \ldots, \#Otf_i \end{aligned}$$

where $\#Obs$ is the number of obstacles in the scene and $\#Otf_i$ is the number of OTFs in obstacle Ob_i.

We choose this specific configuration because in this way all the intersections that can occur are between triangle and triangle or triangle and segment and they can be easily calculated. This implies that, if an obstacle is a polyhedron more complex than just a tetrahedron, preliminarily its faces have to be triangulated.

We provide the methods explained in Section 4.3 for abstracting the creation of OTFs.

Using this solution also means that we can potentially insert in the scene open polyhedrons[2] or intersecting shapes, as we don't have any restriction on the position of the points $\mathbf{p}_{i,j,k}$.

4.3 COMPLEX OBSTACLES

In order to simplify the scene construction, four methods for easily building obstacles have been realized:

- one for building tetrahedrons;
- one for building parallelepipeds[3];
- one more general for building convex hulls;
- a special method used for building a bucket-shaped obstacle that we use in the tests.

Algorithm 5 takes the four vertices of a tetrahedron to and adds to Obs a new obstacle that have all the faces of the unique tetrahedron that can be built with the four points.

2 For instance a tetrahedron without one face

3 aligned with the axis

Algorithm 5 Abstract construction of tetrahedron

```
1: procedure buildTetrahedron(Obs, a, b, c, d)
2:     Obs ← Obs ∪ { {a, b, c}, {a, b, d}, {b, c, d}, {c, a, d} }
3: end procedure
```

Algorithm 6 Abstract construction of convex hull polyhedron

```
1: procedure buildConvexHullPolyhedron(Obs, p_0, ..., p_n)
2:     Ob ← ∅
3:     facets ← convexHull({p_0, ..., p_n})
4:     for all f ∈ facets do
5:         simplices ← triangularize(f)
6:         for all {s_0, s_1, s_2} ∈ simplices do
7:             Ob ← Ob ∪ {s_0, s_1, s_2}
8:         end for
9:     end for
10:    Obs ← Obs ∪ Ob
11: end procedure
```

Algorithm 6 is more complex, first we need to build the convex hull of the input points (see [7] and [24] for details on the convex hull algorithm), then we obtain a set of *facets* that have to be triangulated (see [7] and [24] for details on the triangularization algorithms). Finally we add each triangle as a new OTF of the obstacle.

4.4 BOUNDING BOX

We decided also to give to the user the possibility of adding a bounding box around the scene. It is built as an obstacle, using OTFs, in fact we provide a method that take two points $\mathbf{a}$ and $\mathbf{b}$ and build the parallelpiped with extremes those points and with all the faces triangularized like in Fig. 7.

In all the project, regarding the intersections, the OTFs of the bounding box are considered exactly like the OTFs of the obstacles. The only differences are that the bounding box is not visible when the scene is plotted, and a point inside the bounding box is not considered inside obstacle.

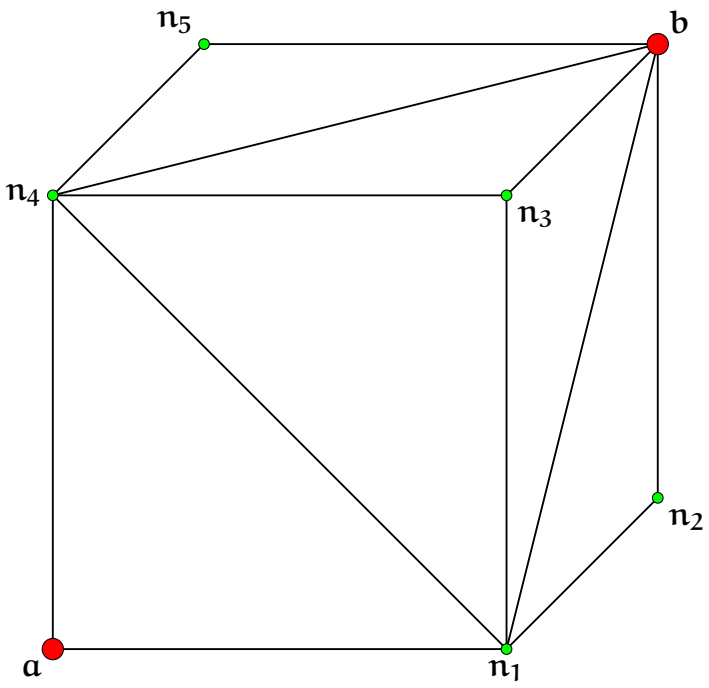


Figure 7.: Bounding box with extremes $\mathbf{a}$ and $\mathbf{b}$.

5

ALGORITHMS

In this chapter we analyze step-by-step the algorithms that implement the different parts of the program. We do this with the help of the test scene in Fig. 8.

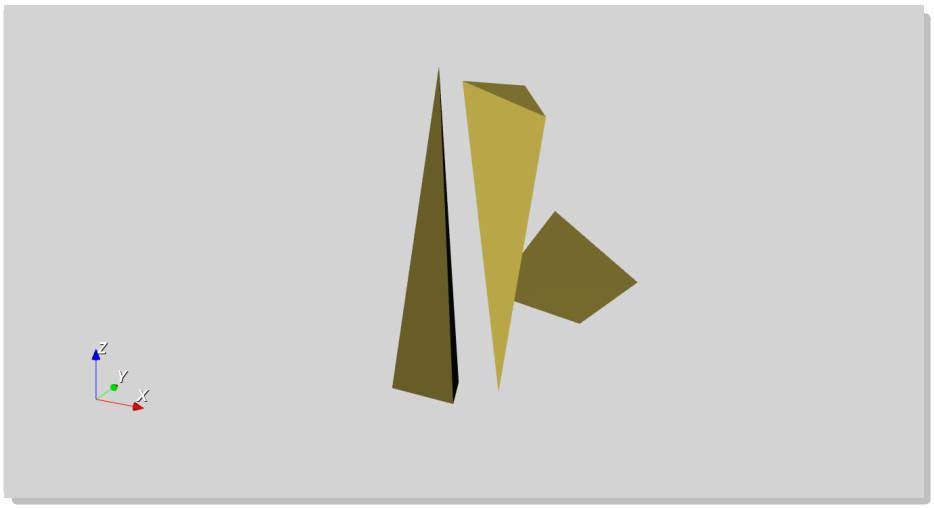


Figure 8.: Initial scene.

The general idea is to use an open B-spline curve of a certain degree interpolating the chosen starting and ending points, whose control polygon is a suitable modification of a polygonal chain extracted from a graph obtained with a VD method. In Section 5.1 we explain in detail how to build such polygonal chain. Before using the chain as a control polygon for the B-spline, it is refined and adjusted - as explained in detail in Section 5.2 and Section 5.3 - in order to ensure that the associated B-spline curve has no obstacle collision. Furthermore in Section 5.4 w implemented a method for an optional adaptive arrangement of the breakpoints of the B-spline. Finally Section 5.5 is devoted to an optional post processing of the path.

5.1 POLYGONAL CHAIN

In the first phase the objective is to extract a suitable polygonal chain from the scene, such that the extremes coincide with the start point $\mathbf{s}$ and the end point $\boldsymbol{e}$. In particular we are interested in short length chains. We calculate the shortest path in a graph that is obtained by using an adaptation to three dimensions of a well known bidimensional method [4][15][31] that use VDs as base.

We choose a Voronoi method because it builds a structure roughly equidistant from obstacles, and so we have low probability of collisions between the curve and the obstacles.

5.1.1 *Base Graph*

First we start distributing points on the OTFs and on an invisible bounding box, as in Fig. 9. The sites are distributed using a recursive method, for

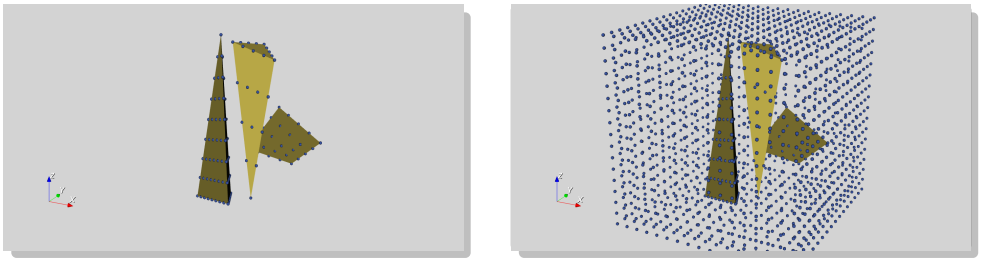


Figure 9.: Scene with Voronoi sites (distributed only on the obstacles surfaces on the left, and on obstacles and bounding box on the right).

each triangle of the scene we add three points - one for each vertex, if not already added before - and then we calculate the area of the triangle. If the area is bigger than a threshold we decompose the triangle in four triangles adding three more vertices on the midpoints of the edges of the original triangle as in Fig. 10, and repeating the process recursively for each new triangle.

We construct the VD using the Fortune's algorithm [12] on those points as input sites, and we build a graph

$$G = (V, E)$$

using the vertices of the Voronoi cells as graph nodes in V, and the edges of the cells[1] as graph edges in E. Furthermore we make G denser adding

1 Rejecting potential infinite edges.

Figure 10.: Decomposition of an OTF.

for every cell's face all the diagonals as edges (we connect every vertex to every other vertex in a face).

After we prune such graph deleting every edge that intersects an OTF using the methods explained in Section 3.5. The edge-pruning process considers a margin around the OTFs during the collision checks.

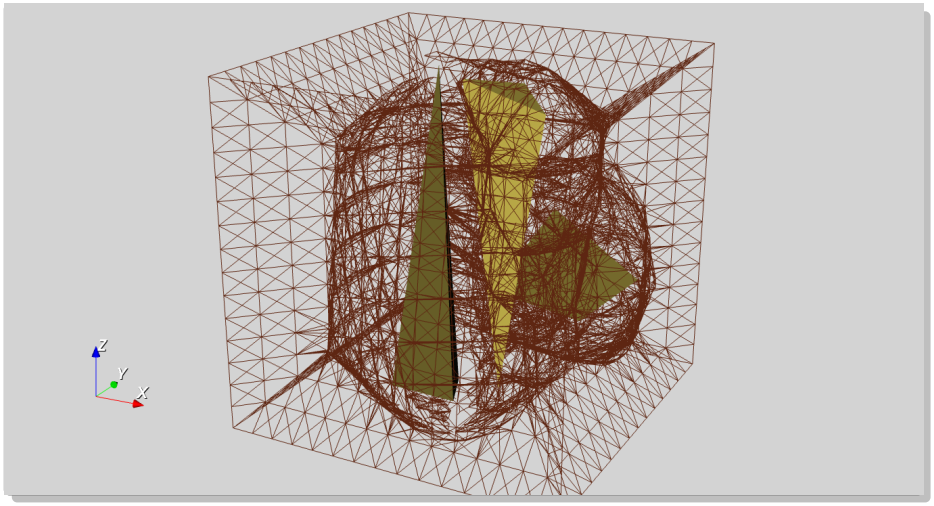


Figure 11.: Scene with pruned graph.

The result, visible in Fig. 11, is a graph that embraces the obstacles like a cobweb where the possible paths on it are roughly equidistant from the obstacles.

In the bidimensional case the equivalent method implies distributing the sites in the edges of the polygonal obstacles and then pruning the graph when an edge of the graph intersect an edge of the obstacle as in Fig. 12. We decided to extend the method in 3 dimensions distributing points in the whole OTF surface, an alternative would be distributing points only along the edges of the obstacles.

On the obtained graph G we attach the desired start and end points $\mathbf{s}$ and $\boldsymbol{e}$, and we can obtain a path between the two points using an

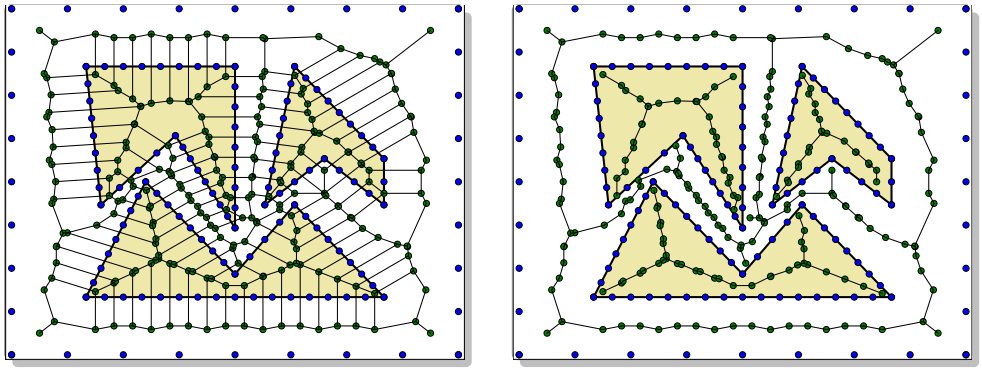


Figure 12.: Voronoi graph in 2D before (left) and after (right) pruning.

algorithm like Dijkstra [9][20]. For attaching $\mathbf{s}$ and $\boldsymbol{e}$ we designed two alternative methods:

- the first method finds the vertex $\boldsymbol{v_n} \in V_{vis} \subseteq V$ such that $\mathtt{dist}(\mathbf{s}, \boldsymbol{v_n}) \leqslant \mathtt{dist}(\mathbf{s}, \boldsymbol{v_i}), \forall \boldsymbol{v_i} \in V_{vis}$, where

  $$V_{vis} = \{\boldsymbol{v} \in V \ : \ \overline{\mathbf{s}\boldsymbol{v_i}} \text{ do not intersects any obstacle}\},$$

  then adds $\mathbf{s}$ to V and the edge $(\mathbf{s}, \boldsymbol{v_n})$ to E; similarly for $\boldsymbol{e}$;

- the second method adds $\mathbf{s}$ to V and all the edges $(\mathbf{s}, \boldsymbol{v_i})$ $\forall \boldsymbol{v_i} \in V_{vis}$; similarly for $\boldsymbol{e}$.

Before using that path as a control polygon we need to take into account the degree of the B-spline and the position of the obstacles, the details are in Section 5.2 and Section 5.3.

Complexity considerations

Fortune's algorithm runs in time $\mathcal{O}(|Is| \log |Is|)$ [7] where Is is the set of input sites. If we impose a maximum area A for the obstacles [2] then $|Is| = \mathcal{O}(|Ob|)$ where Ob is the set of obstacles, because on the worst case we have that $|Is| = C \cdot A \cdot |Ob|$ for some constant C that depends of the chosen density of sites per area.

In conclusion the time cost for the creation of the graph is

$$\mathcal{O}(|Ob| \log |Ob|) \tag{5.1}$$

2 Inserting the obstacles in a progressive order, the area of the i-th obstacle cannot be a function $f(i)$ of the number of the obstacles

and the number of the vertices in the graph is

$$|V| = \mathcal{O}(|Is|) = \mathcal{O}(|Ob|) \tag{5.2}$$

because the number of vertices in the resulting graph is the same order of number of input sites.

If we make the hypothesis of having maximum degree k in G - i.e. each vertex in V is connected at most to other k vertices - then we have that

$$|E| = \mathcal{O}(k|V|) = \mathcal{O}(k|Ob|). \tag{5.3}$$

In the worst case $k = |V|$ and $|E| = \mathcal{O}(|V|^2)$ but for VDs in plane there is a property that if we have n input sites that lay on a circumference, without any other site inside the circumference, then the center of the circumference is a vertex shared by n cells (Section 3.3 for details). The same property is extensible in space with spheres. We can make the assumption that no more than three sites can lay on a circumference so no vertex can have more than three neighbours, or the same with four vertices in sphere, and this assumption is plausible because we use floating point numbers for the coordinates of the vertices of the obstacles, and is quite improbable that more than four points lay on a sphere. With this assumption k is a constant, and Eq. (5.3) become

$$|E| = \mathcal{O}(|V|) = \mathcal{O}(|Ob|).$$

For pruning the graph of every edge that intersects obstacles we need to solve a system of three unknowns in thre equations for every edge and every OTF[3], so we have a cost of

$$\mathcal{O}(|E| \cdot |Ob|) = \mathcal{O}(k|Ob|^2) \tag{5.4}$$

and if we make the assumption of k constant

$$\mathcal{O}(|Ob|^2)$$

5.1.2 *Triple's graph*

Before calculating the shortest path on the chosen graph with Dijkstra [9][20], we transform it in a graph containing all the triples of three adjacent vertices in the original graph. This because we want to filter the

3 See Section 3.5.2.

triples for collisions as described in Section 5.2.1. We call the transformed graph

$$G_t = (V_t, E_t)$$

where we have triples of vertices of G in V_t.

The original graph G is not directed and it is weighted with the distance from vertex to vertex. The transformed graph G_t is directed and weighted, where if in G the nodes $\mathbf{a}$ and $\mathbf{b}$ are neighbouring, and $\mathbf{b}$ and $\mathbf{c}$ are neighbouring, then G_t has the two nodes $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ and $(\mathbf{c}, \mathbf{b}, \mathbf{a})$. In G_t a node $(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1)$ is a predecessor of $(\mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2)$ if $\mathbf{b}_1 = \mathbf{a}_2$ and $\mathbf{c}_1 = \mathbf{b}_2$, and the weight of the arc from $(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1)$ to $(\mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2)$ in G_t is equal to the weight of the arc from $\mathbf{a}_1$ to $\mathbf{b}_1 (= \mathbf{a}_2)$ in G.

The steps necessary for creating G_t are summarized in Algorithm 7. The input G is the base graph which has vertices V and edges E, $N_G(\mathbf{a})$ is the set of neighbours in G of the vertex $\mathbf{a}$, and the output is G_t.

The transformation of the graph is useful only for the obstacle avoidance algorithm of Section 5.2.1, theoretically is possible to bypass such transformation for the algorithm described in Section 5.2.2.

Complexity considerations

If we suppose a maximum degree k for each vertex in the graph G - i.e. each vertex in V can have at most k edges insisting on it, then the number of vertices in the transformed graph G_t is

$$|V_t| \leqslant |V| \cdot k \cdot (k-1) = \mathcal{O}(k^2|V|) \tag{5.5}$$

because for each vertex $\boldsymbol{v}$ in G we need to consider all the neighbours of $\boldsymbol{v}$ and the neighbours of the neighbours of $\boldsymbol{v}$ (excluded $\boldsymbol{v}$).

For how we defined the triples neighbour rule in G_t we have that each triple is a predecessor of at most $k-1$ other triples. For instance $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ in V_t is the predecessor of all the triples $(\mathbf{b}, \mathbf{c}, *)$ where $*$ can be one of the k neighbours of $\mathbf{c}$ in V excluded $\mathbf{b}$. So the number of edges in G_t is

$$|E_t| \leqslant |V_t| \cdot (k-1) = \mathcal{O}(k|V_t|) = \mathcal{O}(k^3|V|) \tag{5.6}$$

Also the time cost for the creation of G_t is

$$\mathcal{O}(k^2|E|) = \mathcal{O}(k^3|Ob|) \tag{5.7}$$

because Algorithm 7 for creating the transformed graph scans all the edges *e* on Line 3 and for each iteration the biggest cost is due to the two *for* on Line 20 and Line 21. Each one scans triples created from at most k neighbours of the vertices at the extremes of *e*.

Algorithm 7 Create triples graph G_t

```
1: function createTriplesGraph(G)
2:     V_t ← E_t ← ∅
3:     for all (a, b) ∈ E do
4:         leftOut ← leftIn ← rightOut ← rightIn ← ∅
5:         for all v ∈ N_G(a) \ {b} do
6:             leftOut ← leftOut ∪ {(v, a, b)}
7:             leftIn ← leftIn ∪ {(b, a, v)}
8:             V_t ← V_t ∪ {(v, a, b), (b, a, v)}
9:         end for
10:        for all v ∈ N_G(b) \ {a} do
11:            rightOut ← rightOut ∪ {(v, b, a)}
12:            rightIn ← rightIn ∪ {(a, b, v)}
13:            V_t ← V_t ∪ {(v, b, a), (a, b, v)}
14:        end for
15:        for all o ∈ leftOut do
16:            for all i ∈ rightIn do
17:                E_t ← E_t ∪ (o, i)
18:            end for
19:        end for
20:        for all o ∈ rightOut do
21:            for all i ∈ leftIn do
22:                E_t ← E_t ∪ (o, i)
23:            end for
24:        end for
25:    end for
26:    G_t ← (V_t, E_t)
27:    return G_t
28: end function
```

5.2 OBSTACLE AVOIDANCE

Before using the polynomial chain extracted as explained in Section 5.1 as a control polygon for the B-spline, we need to deal with a problem: every possible path in the graph G is free from collisions by construction - in fact we prune the graph of every edge that intersects an obstacle - but this does not guarantee that the associated curve will not cross any obstacle. This concept is exemplified in Fig. 13.

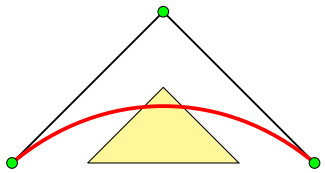


Figure 13.: Schematic of B-spline that intersects an obstacle in the plane.

In this chapter we make the hypothesis of using quadratic B-splines[4], in Section 5.3 it is explained how is possible to use curves with higher degree. With this assumption we can exploit the CHPs explained in Section 3.2, and assert that the resulting curve is contained inside the union of all the triangles of three consecutive control vertices of the control polygon. Using that property we can resolve the problem of the collision, maintaining free from collision with OTFs all the triangles associated with the control polygon. Note that the CHP of quadratic B-splines is valid also in space, so the convex hull is still composed of triangles, like the faces of the obstacles. This simplifies all the checks for collisions because they are all between triangles in space and we can use the methods described in Section 3.5.

We designed two different algorithms for dealing with the collision problem. The first solution described in Section 5.2.1 implements a modified version of Dijkstra's algorithm that finds the shortest path from start to end in the graph such that all the triangles formed by three consecutive points in such path are free from collisions. The second solution described in Section 5.2.2 use the classical Dijkstra's algorithm to find the shortest path from s to e in the graph G, checking later for collisions in the triangles formed of three consecutive points in such path. When a collision is found some actions are taken for dealing with that.

4 B-splines curves with degree 2 or equivalently order 3.

5.2.1 *First solution: Dijkstra's algorithm in* G_t

The first solution of the problem exploits the graph G_t obtained as explained in Section 5.1.2. Before applying Dijkstra's algorithm to G_t all the triples are filtered checking if the triangle composed of the vertices of the triple intersect an OTF. If a triple intersect an obstacle then it is removed from the graph and so a path cannot pass from such vertices in that order.

Note that if a triple $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ is removed from V_t - and consequently also the triple $(\mathbf{c}, \mathbf{b}, \mathbf{a})$ - not necessarily this excludes the three vertices $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ from being part of the final polynomial chain. For instance in Fig. 14 we have a graph G with vertices $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}$ and an obstacles that intersects triples on the transformed graph[5] G_t. The triple $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ and $(\mathbf{c}, \mathbf{b}, \mathbf{a})$ will be removed from G_t because the corresponding triangle intersects the obstacle, and the path $\mathbf{d} \to \mathbf{a} \to \mathbf{b} \to \mathbf{c} \to \mathbf{e}$ cannot be admissible. This doesn't preclude the nodes $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ to be part of the final admissible path $\mathbf{d} \to \mathbf{a} \to \mathbf{b} \to \mathbf{e} \to \mathbf{c} \to \mathbf{f}$.

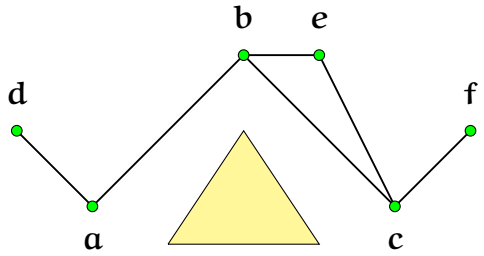


Figure 14.: Example of triples.

On the cleaned transformed graph it is possible to find the shortest path on the triples

$$P_t = (\mathbf{a}_0, \mathbf{b}_0, \mathbf{c}_0), (\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1), \ldots, (\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i), \ldots, (\mathbf{a}_n, \mathbf{b}_n, \mathbf{c}_n)$$

using an algorithm like Dijkstra. Then the shortest path P in G is constructed by taking the central vertex $\mathbf{b}_i$ of every triple $(\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i)$ of P_t, plus the extremes $\mathbf{a}_0$ and $\mathbf{c}_n$ of the first and last triples, obtaining

$$P = \mathbf{a}_0, \mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_i, \ldots, \mathbf{b}_{n-1}, \mathbf{b}_n, \mathbf{c}_n.$$

5 In the plane, this graph cannot be obtained using the procedure based on VDs explained in Section 3.3, but a similar situation is plausible considering Voronoi cells in space.

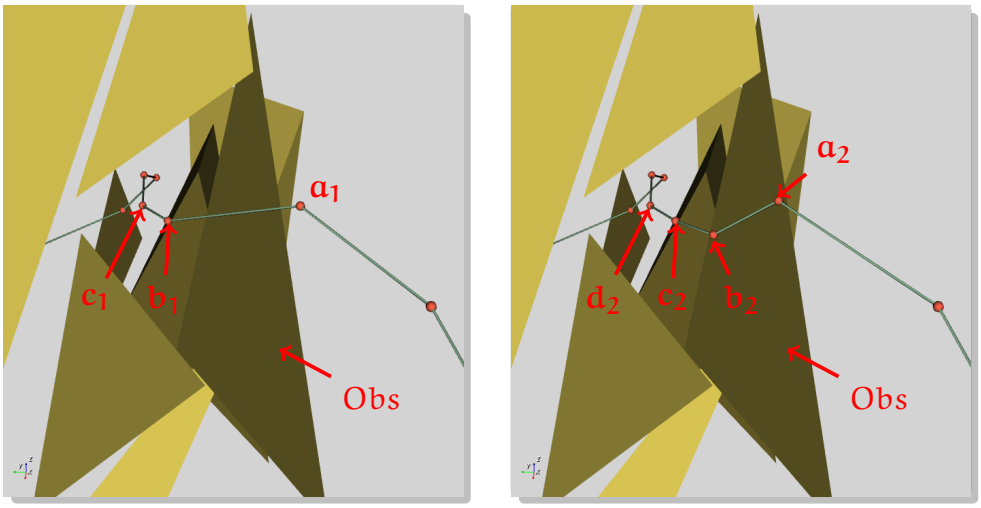


Figure 15.: Effects of application of solution one.

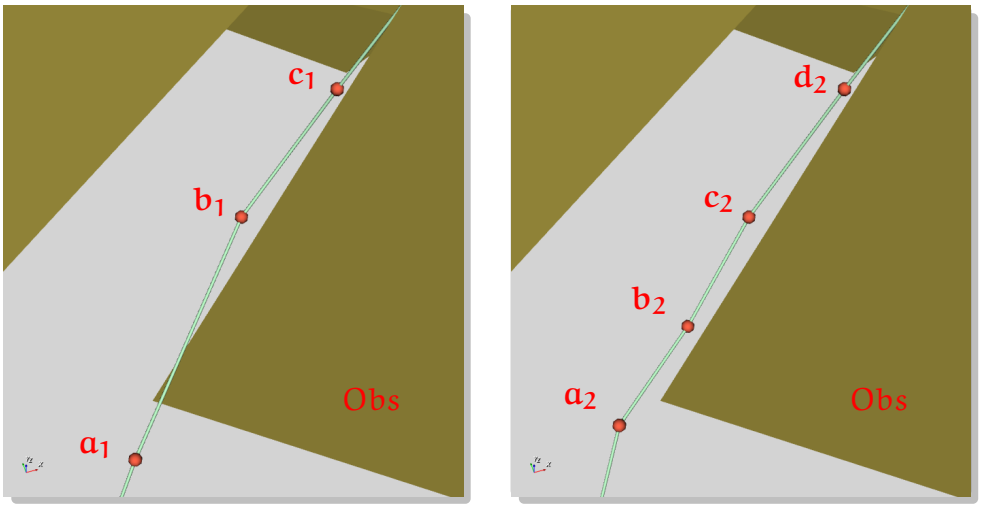


Figure 16.: Effects of application of solution one, other view point.

In Fig. 15 and Fig. 16 the effect of the application of the first solution is shown. The triangle formed by the vertices $\mathbf{a}_1$, $\mathbf{b}_1$, $\mathbf{c}_1$ on the left picture of Fig. 15 is colliding with the obstacle Obs in the back. On the right picture there is the path $\mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2, \mathbf{d}_2$ obtained applying the solution, where no triangles in the path collide with obstacles. In Fig. 16 another point of view of pictures shown in Fig. 15 is visible.

Complexity considerations

For each triple and each OTF we need to solve three 3×3 linear systems for the collision check[6], so in total the cost is

$$\mathcal{O}(|V_t| \cdot |Ob|)$$

and for Eq. (5.2) and Eq. (5.5) this is equal to

$$\mathcal{O}(|Ob|^2 k^2). \tag{5.8}$$

The cost of applying Dijkstra's algorithm[7] in G_t is [1][21]

$$\begin{aligned} \mathcal{O}(|E_t| + |V_t| \log |V_t|) &= \mathcal{O}(k^3|V| + k^2|V| \log(k^2|V|) \\ &= \mathcal{O}(k^3|Ob| + k^2|Ob| \log(k^2|Ob|). \end{aligned} \tag{5.9}$$

Such cost has two special cases:

- if G is a *clique* - i.e. each node in V is connected to every other node [1] - then $k = |V| - 1$ and the cost is

  $$\mathcal{O}(|V|^4);$$

- if k is constant - i.e. doesn't grow with $|V|$ - the cost is

  $$\mathcal{O}(|V| \log |V|).$$

The latter case is the more plausible if we assume the hypothesis that no more than four input sites in space can be on the same sphere, in fact in that case every Voronoi cell cannot have a vertex with more than four edges connected to it (see Section 3.3 for details).

If we sum all the costs we obtain:

$$\mathcal{O}(k^2|Ob|^2 + k^3|Ob|) \tag{5.10}$$

6 See Section 3.5.3.

7 In the worst case where no triples are removed in the cleaning phase.

where all the other terms are absorbed in those two. If we have k constant as we said before then we have an overall cost of

$$\mathcal{O}(|Ob|^2) \tag{5.11}$$

that derives from the collision-check controls.

We can improve this result if we divide the algorithm in two parts:

1. first we can construct the graph once with cost $\mathcal{O}(|Ob|^2)$;
2. then we can use the graph in different situations[8] with cost $\mathcal{O}(|Ob| \log |Ob|)$, only for the routing.

Description	Cost	Reference
Creation of G	$\mathcal{O}(\|Ob\| \log \|Ob\|)$	Eq. (5.1)
Pruning of G	$\mathcal{O}(k\|Ob\|^2)$	Eq. (5.4)
Creation of G_t	$\mathcal{O}(k^3\|Ob\|)$	Eq. (5.7)
Pruning of G_t	$\mathcal{O}(\|Ob\|^2k^2)$	Eq. (5.8)
Routing in G_t	$\mathcal{O}(k^3\|Ob\| + k^2\|Ob\| \log(k^2\|Ob\|)$	Eq. (5.9)
Total	$\mathcal{O}(k^2\|Ob\|^2 + k^3\|Ob\|)$	Eq. (5.10)
Total (k constant)	$\mathcal{O}(\|Ob\|^2)$	Eq. (5.11)

Table 2.: Resume of the costs for solution one

On Table 2 we resume all the terms that contributes to the total costs, and the total cost itself.

5.2.2 *Second solution: Dijkstra's algorithm in* G

The First solution is interesting from an algorithmic point of view, but it is not very practical because it throws all the triples that intersect an obstacle. We managed to develop a solution that uses another approach: obtain the shortest path from the Voronoi's graph G directly using Dijkstra's algorithm, without removing any triple. On that path - that we call P - we check every triple of consecutive vertices, and if it collides with an OTF then we take countermeasures (see Section 3.5.3 for the procedure implemented to identify collisions between two triangles). For instance if the path is composed from the vertices:

$$P = (v_0, v_1, \ldots, v_n)$$

8 With specific starting and ending points

then we check every one of the triangles

$$\begin{aligned} T_0 &= \triangle v_0 v_1 v_2 \\ T_1 &= \triangle v_1 v_2 v_3 \\ &\dots \\ T_i &= \triangle v_i v_{i+1} v_{i+2} \\ &\dots \\ T_{n-3} &= \triangle v_{n-3} v_{n-2} v_{n-1} \\ T_{n-2} &= \triangle v_{n-2} v_{n-1} v_n \end{aligned}$$

for intersections with OTFs, where $\triangle v_i v_j v_k$ denotes the triangle with vertices the points v_i, v_j and v_k.

Consider also that G is pruned from all the edges that intersect any obstacle, so none of the edges of the triangles T_i can intersect an OTF. The only possibility is that edges[9] of OTF intersect a triangle T_i. So for each T_i we have a (possibly empty) set of points of intersection between it and the edges of each OTF - we call that set O.

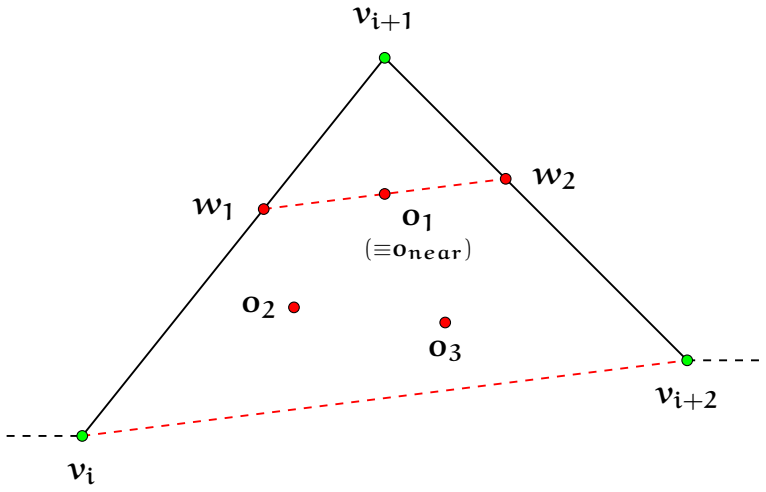


Figure 17.: $T_i (= \triangle v_i v_{i+1} v_{i+2})$ and the points o_1, o_2, o_3 of intersection between it and the edges of some OTFs.

In Fig. 17 we have an example of the triangle

$$T_i = \triangle v_i v_{i+1} v_{i+2}$$

9 At most two edges for each OTF if we ignore special cases.

that is intersected by obstacles in the points

$$O = \{\mathbf{o}_1, \mathbf{o}_2, \mathbf{o}_3\}.$$

where each one of the points in O is expressed in barycentric coordinates of the vertices $\mathbf{v}_i$, $\mathbf{v}_{i+1}$ and $\mathbf{v}_{i+2}$ of the triangle:

$$\begin{aligned}
\mathbf{o}_1 &= \alpha_1\mathbf{v}_i + \beta_1\mathbf{v}_{i+1} + \gamma_1\mathbf{v}_{i+2} \\
\mathbf{o}_2 &= \alpha_2\mathbf{v}_i + \beta_2\mathbf{v}_{i+1} + \gamma_2\mathbf{v}_{i+2} \\
\mathbf{o}_3 &= \alpha_3\mathbf{v}_i + \beta_3\mathbf{v}_{i+1} + \gamma_3\mathbf{v}_{i+2}
\end{aligned}$$

where $\alpha_i + \beta_i + \gamma_i = 1$ for $i = 1, 2, 3$.

We want to deal with collisions adding vertices in the control polygon, such that consecutive triangles are free from obstacles. We obtain this adding two new control vertices:

- $\mathbf{w}_1$ between $\mathbf{v}_i$ and $\mathbf{v}_{i+1}$;
- $\mathbf{w}_2$ between $\mathbf{v}_{i+1}$ and $\mathbf{v}_{i+2}$.

We add those points in a manner such that the segment $\overline{\mathbf{w}_1\mathbf{w}_2}$ is parallel to the segment $\overline{\mathbf{v}_i\mathbf{v}_{i+2}}$ and such that $\overline{\mathbf{w}_1\mathbf{w}_2}$ passes just above the obstacle point $\mathbf{o}_{near}$ that is the nearest to $\mathbf{v}_{i+1}$ ($\mathbf{o}_1$ in Fig. 17). The degenerate triangles $\triangle\mathbf{v}_i\mathbf{w}_1\mathbf{v}_{i+1}$ and $\triangle\mathbf{v}_{i+1}\mathbf{w}_2\mathbf{v}_{i+2}$, and the not degenerate triangle $\triangle\mathbf{w}_1\mathbf{v}_{i+1}\mathbf{w}_2$ replace the original triangle T_i and they do not collide with obstacles for construction.

When we check for collisions between a segment and a triangle we resolve a system of three unknowns in three equations and from the solutions we extract the barycentric coordinates of the point of collision. When we have all the coordinates of the points in O we can obtain $\mathbf{o}_{near}$ picking the one with the biggest β and then, using the corresponding β_{near}, we can obtain

$$\begin{aligned}
W_1 &= \beta_{near}\mathbf{v}_{i+1} + (1 - \beta_{near})\mathbf{v}_i \\
W_2 &= \beta_{near}\mathbf{v}_{i+1} + (1 - \beta_{near})\mathbf{v}_{i+2}.
\end{aligned}$$

In Fig. 18 and Fig. 19 we can see the effects of the application of this solution to a piece of the curve. The original pieces of control polygon are on the left pictures, the triangle composed of those vertices collide with the obstacle on the back. The two new vertices $\mathbf{w}_1$ and $\mathbf{w}_2$ are added to avoid the collision.

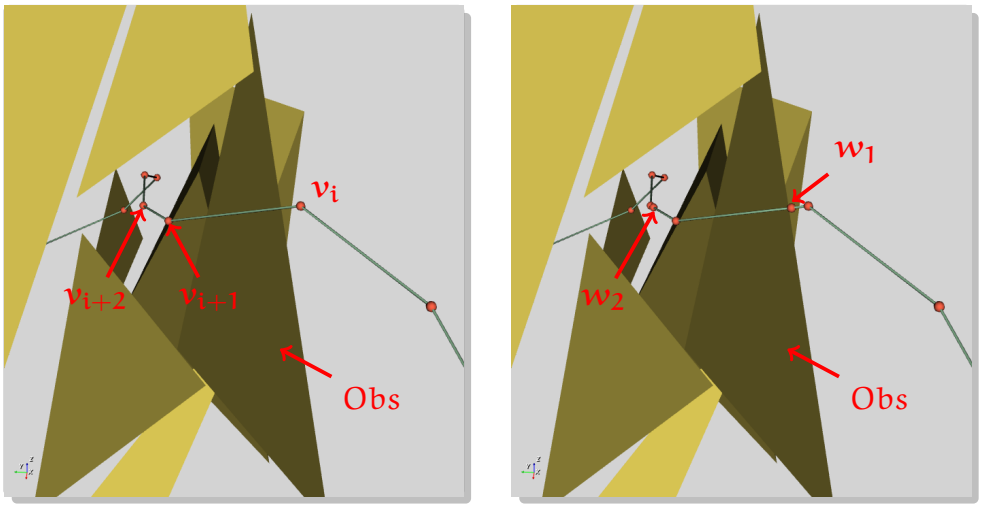


Figure 18.: Effects of application of solution two.

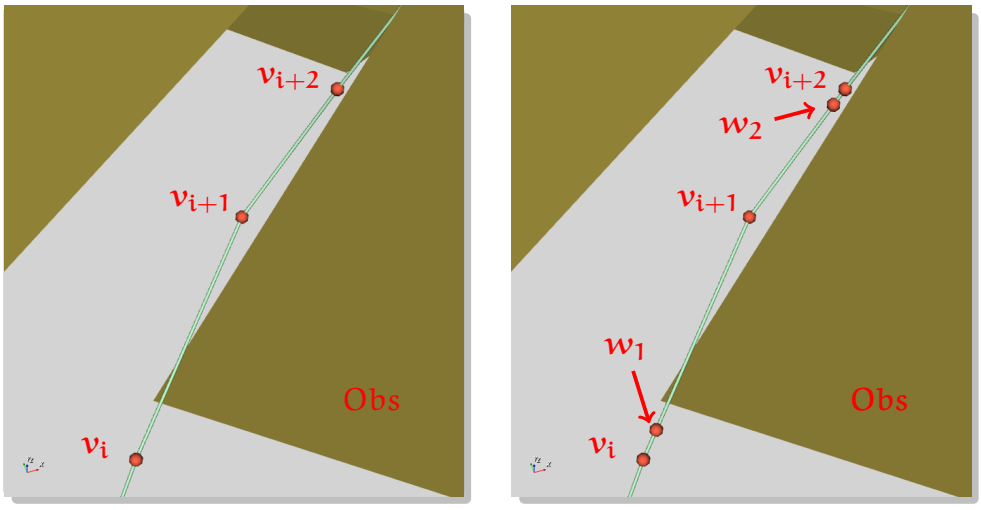


Figure 19.: Effects of application of solution two, other view point.

Complexity considerations

For this solution we have still the costs of Eq. (5.1) and Eq. (5.4) for the creation and pruning of the graph G. In addition we need to apply Dijkstra's algorithm in G for obtaining P with a cost [1][21]

$$\mathcal{O}(|E| + |V| \log |V|).$$

For Eq. (5.2) Eq. (5.3) that cost is equal to

$$\mathcal{O}(k|Ob| + |Ob| \log |Ob|) \tag{5.12}$$

and if we make the assumption of k constant we have a cost

$$\mathcal{O}(|Ob| \log |Ob|).$$

For checking and removing the collisions in the path we need to consider every face of obstacle in Ob for every triangle in P. The cost for doing this is[10] $\mathcal{O}(|P| \cdot |Ob|)$ where $|P|$ means the number of vertices in P. In the worst case $|P| = \mathcal{O}(|V|) = \mathcal{O}(|Ob|)$ so we have a cost

$$\mathcal{O}(|P| \cdot |Ob|) = \mathcal{O}(|Ob|^2) \tag{5.13}$$

Summing up all the costs we have

$$\mathcal{O}(k|Ob|^2) \tag{5.14}$$

and if we consider k constant

$$\mathcal{O}(|Ob|^2). \tag{5.15}$$

On Table 3 we resumed all the terms that contributes to the total costs, and the total cost itself.

The cost is comparable with the one of the first solution. Furthermore, also in this case we can divide the algorithm in two parts:

1. first we can construct only once G with cost $\mathcal{O}(|Ob|^2)$;
2. then we can use it for different situations with cost $\mathcal{O}(|Ob| \log |Ob| + |P| \cdot |Ob|)$.

10 If #OTFs $= \mathcal{O}(|Ob|)$ - i.e. the number of OTFs don't grow faster than the number of obstacles.

Description	Cost	Reference
Creation of G	$\mathcal{O}(\lvert Ob\rvert \log \lvert Ob\rvert)$	Eq. (5.1)
Pruning of G	$\mathcal{O}(k\lvert Ob\rvert^2)$	Eq. (5.4)
Routing in G	$\mathcal{O}(k\lvert Ob\rvert + \lvert Ob\rvert \log \lvert Ob\rvert)$	Eq. (5.12)
Clean path	$\mathcal{O}(\lvert P\rvert \cdot \lvert Ob\rvert) = \mathcal{O}(\lvert Ob\rvert^2)$	Eq. (5.13)
Total	$\mathcal{O}(k\lvert Ob\rvert^2)$	Eq. (5.14)
Total (k costant)	$\mathcal{O}(\lvert Ob\rvert^2)$	Eq. (5.15)

Table 3.: Resume of the costs for solution two

5.3 DEGREE INCREASE

Until now we have assumed of dealing only with quadratic B-splines - i.e. of degree m - because in that case, for the CHP (Section 3.2.1), we need to check intersections only between two triangles (one belonging to P and the other to OTFs). If we want to use higher degree curves we can modify the previous algorithms for dealing with polyhedral convex hulls, but this implies a complexity increasing.

We are interested in increasing the degree for achieving smooth curves with continue curvature and torsion. We adopted a compromise: we adapt the path obtained from the previous algorithms adding vertices and forcing the curve to remain in the same convex hull. However this approach have the drawback that we cannot achieve a good torsion[11] because the curve changes plane in an inflection point of the curvature.

We modify

$$P = (v_0, \dots, v_n)$$

adding a certain number of aligned new vertices $(w_0, w_1, \dots)$ between each pair (v_i, v_{i+1}) of vertices in P for $i = 0, \dots, n-1$. The number of w_j between each pair (v_i, v_{i+1}) depends on the desired grade of the curve. In fact we need $m-2$ new vertices between each (v_i, v_{i+1}) for B-spline curves of degree m. Thus the final modified path for a B-spline curve of degree $m \geqslant 3$ is

$$\tilde{P} = (v_0, w_0, \dots, w_{m-3}, v_1, \dots, v_i, w_{i(m-2)}, \dots, w_{(i+1)(m-2)-1}, v_{i+1}, \dots, v_n).$$

This strategy is used in the project only for lifting the degree from 2 to 3 or 4.

11 We can improve this with the post process.

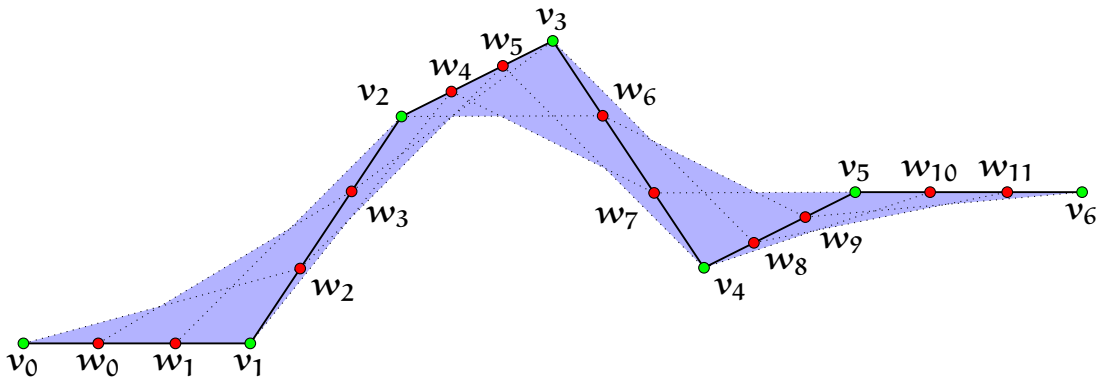


Figure 20.: Increase the degree m from 2 to 4.

In Fig. 20 is visible an example of path

$$P = (v_0, v_1, v_2, v_3, v_4, v_5, v_6).$$

In green we have the vertices of P, in red the added vertices and the cyan area is the convex hull of the final curve.

We want to adapt P to quartic B-spline curves, so we need to add two new vertices between each pair of vertices (v_i, v_{i+1}) for $i = 0, \ldots, 6$. Those new vertices are

$$(w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8, w_9, w_{10}, w_{11}).$$

Note that, with this algorithm, when we increase the degree from 2 to $m \geqslant 3$ we have that the convex hull containing a B-spline curve of degree m in $\tilde{P}$ is a subset of the convex hull containing a B-spline curve of degree 2 in P. This is because the polyhedrons of consecutive $m+1$ vertices in $\tilde{P}$ collapses in triangles contained inside the triangles of consecutive vertices in P. For instance in Fig. 20 the convex hull of the first 5 vertices v_0, w_0, w_1, v_1, w_2 of $\tilde{P}$ coincides with the triangle $\triangle v_0 v_1 w_2$ that is contained inside the triangle $\triangle v_0 v_1 v_2$ of the first 3 vertices of P.

One effect of the application of this method is that a curve of degree m in $\tilde{P}$ will touch every segment of the original control polygon P. This is because adding $m-2$ aligned vertices between each pair (v_i, v_{i+1}) will result in m aligned vertices on each original segment (Section 3.2.2).

5.4 KNOTS SELECTION

In the previous sections we never discussed the criterion adopted for fixing the extended knot vector T associated to the B-spline curve.

$$T = \{t_0, \dots, t_{m-1}, t_m, \dots, t_{n+1}, t_{n+2}, \dots, t_{n+m+1}\}$$

In this section we discuss two methods that we implemented.

First of all, we want that the curve interpolates the chosen start and end points, that correspond to the extremes $\boldsymbol{v}_0$ and $\boldsymbol{v}_n$ of the extracted path P. We saw in Section 3.2.4 that we can archive such interpolation if we impose

$$\begin{aligned} t_0 &= t_1 = \cdots = t_m = a \\ t_{n+1} &= t_{n+2} = \cdots = t_{n+m+1} = b \end{aligned} \tag{5.16}$$

where a and b are the extremes of the parametric domain of the curve.

The constraint of Eq. (5.16) is a mandatory choice, so we cannot change it. Regarding the parametric domain, we chose it to be $[0, 1]$ because changing the extremes do not change the behavior of the curve, only changing the ratios of the distances between the knots is effective [11]. We still need to chose how to select the inner $n - m$ knots $t_{m+1}, \dots, t_n$, and we developed two different ways of doing it:

method 1 Use a uniform partition where $t_i - t_{i-1} = c$ for $i = m+1, \dots, n+1$ for c constant;

method 2 Use an adaptive partition, where we try to make dense knots in correspondence of points on the curve where we have dense control vertices.

method 1 is the easiest way of choosing a knot vector and is common as first choice in textbooks [11][10], but it has the disadvantage of ignoring the geometry of the curve [11]. The steps to do for accomplish **method 1** are quite straightforward: we need to pick the nodes

$$\frac{i}{n-m+1}$$

for $i = 1, \dots, n-m$. Thus we concentrate on **method 2**.

We started from the idea that if we have a control polygon with uniformly-spaced vertices - i.e. $\|\boldsymbol{v}_1 - \boldsymbol{v}_0\|_2 = \|\boldsymbol{v}_2 - \boldsymbol{v}_1\|_2 = \cdots = \|\boldsymbol{v}_n - \boldsymbol{v}_{n-1}\|_2$ - then we agree on a uniform partition of the knots

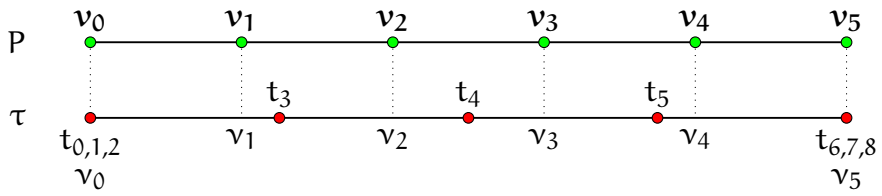


Figure 21.: Optimal case for a quadratic curve (we want uniform partition).

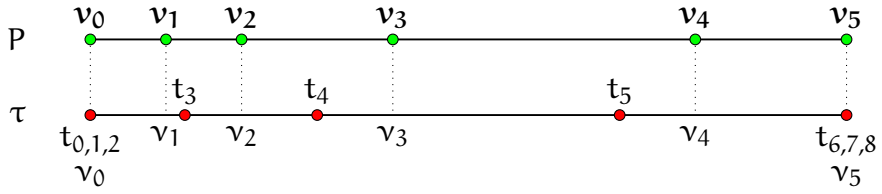


Figure 22.: General case for a quadratic(same distances between t_i and enclosing ν_j, ν_{j+1} as Fig. 21).

$(t_{m+1} - t_m = t_{m+2} - t_{m+1} = \cdots = t_{n+1} - t_n)$. In Fig. 21 there is an example of a quadratic B-spline curve with uniformly-spaced control polygon. The above segment is a *rectified* visualization of the control polygon with six control vertices $\boldsymbol{v}_0, \ldots, \boldsymbol{v}_5$. The segment below represents the partition of the domain from a (on the left) to b (on the right), with the projections $\nu_0, \ldots, \nu_5$ of the control vertices, scaled in length to the parametric domain axis[12], and the knots $t_0, \ldots, t_8$ on it.

Starting from this situation, if we have a generic control polygon with segments of different length as in Fig. 22, then we want that each t_i keep the same distance, in ratio, between the surrounding ν_j and ν_{j+1}, respect to the optimal case. For instance in Fig. 21 $\frac{t_3-\nu_1}{\nu_2-\nu_1} = \frac{1}{4}$ and $\frac{\nu_2-t_3}{\nu_2-\nu_1} = \frac{3}{4}$, this means that in Fig. 22 the same values must be preserved.

The problem now is how to calculate the values of t_i in the general case. We consider only the inner part τ of the partition vector, included the extremes

$$\tau_i = t_{i+m} \qquad i = 0, \ldots, n - m + 1$$

where $\tau_0 = a = 0$ and $\tau_{n-m+1} = b = 1$. In Fig. 21 and Fig. 22 $\tau = (t_2, t_3, t_4, t_5, t_6)$. Now we calculate the positions of all τ_i respect to the ν_j

12 $\boldsymbol{v}_0$ is projected to a, $\boldsymbol{v}_5$ is projected to b, and the ratios between the distances between vertices are preserved.

in the *optimal* case. We can achieve that using as unit the uniform distance $v_j - v_{j-1}$ for calculating the positions of τ_i. Purposely, we calculate

$$\tau_i^v = \frac{n}{n-m+1} \cdot i \qquad i = 0, \dots, n-m+1 \tag{5.17}$$

obtaining the numbers τ_i^v whose integer part $\lfloor \tau_i^v \rfloor$ represents the index j of the v_j that is to the left of τ_i, and the decimal part $(\tau_i^v - \lfloor \tau_i^v \rfloor)$ represents the distance from it: $\frac{\tau_i - v_j}{v_{j+1} - v_j}$.

Now we calculate the projections v_i in the *generic* case. We start calculating the incremental distances between the vertices

$$\begin{cases} d_0 = 0 \\ d_i = d_{i-1} + \|\boldsymbol{v}_i - \boldsymbol{v}_{i-1}\|_2 \qquad i = 1, \dots, n \end{cases}$$

and, remembering that the parametric domain is $[0, 1]$, we have

$$v_i = \frac{d_i}{d_n} \qquad i = 0, \dots, n. \tag{5.18}$$

Now, using the positions in Eq. (5.17) on the projection in Eq. (5.18), we obtain the values

$$\tau_i = v_{\lfloor \tau_i^v \rfloor} + (\tau_i^v - \lfloor \tau_i^v \rfloor)(v_{\lfloor \tau_i^v \rfloor + 1} - v_{\lfloor \tau_i^v \rfloor}) \qquad i = 0, \dots, n-m+1.$$

Finally, adding the duplicated knots, we obtain

$$t_i = \tau_{\min(n-m+1,\ \max(0,\ i-m))} \qquad i = 0, \dots, n+m+1.$$

5.5 POST PROCESSING

The purpose of the post processing phase is to try to simplify the path $P = (\boldsymbol{v}_0, \dots, \boldsymbol{v}_n)$ obtained in the previous phase removing useless vertices, trying to achieve a smoother path.

For obtaining this we realized Algorithm 8 that iterates through all the vertices, except the extremes, and for each v_i checks if it can be removed without consequences. With consequences we mean that removing v_i would cause that a triangle in P intersect one of the OTFs.

For clarifying the concept consider Fig. 23 that is a simplification in 2-dimensional space of the real case in 3-dimensional space. The path to process is

$$P = (\dots, \boldsymbol{v}_{i-2}, \boldsymbol{v}_{i-1}, \boldsymbol{v}_i, \boldsymbol{v}_{i+1}, \boldsymbol{v}_{i+2}, \dots)$$

Algorithm 8 Post processing algorithm on path P.

```
1: procedure postProcess(P)
2:     for i ← 1, n − 1 do
3:         if i = 1 or not intersectOTF(△v_{i−2}v_{i−1}v_{i+1}) then
4:             if i = n − 1 or not intersectOTF(△v_{i−1}v_{i+1}v_{i+2}) then
5:                 P ← P \ {v_i}
6:             end if
7:         end if
8:     end for
9: end procedure
```

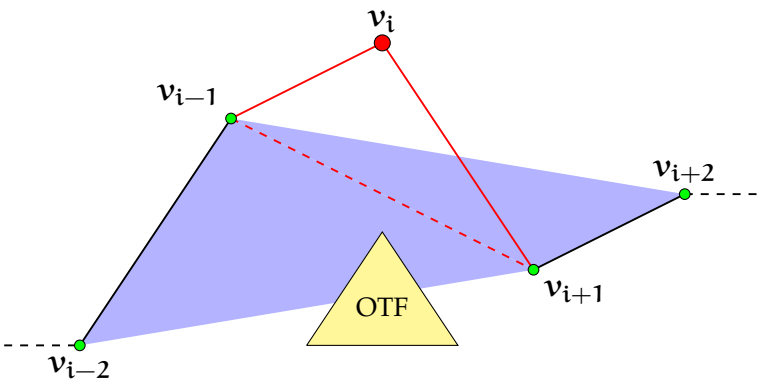


Figure 23.: Example of post process check removing v_i.

and we are considering removing v_i obtaining a modified path

$$\tilde{P} = (\dots, v_{i-2}, v_{i-1}, v_{i+1}, v_{i+2}, \dots).$$

Before doing this we need to check if any triangle in $\tilde{P}$ intersects any OTF. In detail we need to check only the two triangles $\triangle v_{i-2}v_{i-1}v_{i+1}$ and $\triangle v_{i-1}v_{i+1}v_{i+2}$ because the other triangles in $\tilde{P}$ where already present in P. For instance the obstacle in the figure do not intersects anyone of the triangles in P, but it intersects $\triangle v_{i-2}v_{i-1}v_{i+1}$ in $\tilde{P}$.

Complexity considerations

For every vertex of P we need to check if two triangles intersect with an OTF, so we have a complexity of

$$\mathcal{O}(|P| \cdot |Ob|) = \mathcal{O}(|Ob|^2)$$

where Ob is the set of obstacles and $|P|$ is the number of vertices in P.

5.6 THIRD SOLUTION: SIMULATED ANNEALING

The solutions proposed in Section 5.2.1 and Section 5.2.2 have two problems in common:

- both reject configurations in a prudent way considering only the control polygon;
- and both don't optimize neither length nor other quantities.

Anyway such solutions have also the benefit that:

- they produce paths that are obstacle-free from construction;
- the application of the post processing allows often a reduction of the curve length.

In this section we describe a third approach considered in the project, an approach based on a probabilistic computation.

We can see the problem of finding the shortest path as a constrained optimization problem, where a certain configuration of the control vertices (and consequently the B-spline) is the state of the system, and we want to minimize both the length of the control polygon (and consequently the B-spline[13]) and the peak in curvature and torsion of the B-spline, under

13 We give to the user also the possibility of selecting the arc length as quantity to minimize.

the constraint that the B-spline must not intersect the obstacles. We are interested in optimizing both length and curvature and torsion peaks because we want a path that is short but also a path that is fair.

5.6.1 *Lagrangian Relaxation (LR) applied to the project*

We can apply the concept explained in Section 3.4.4 to the project.

The variable space X is composed of all possible configuration of the path, or in other words is the vector $P = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n)$ of all n ordered vertices $\boldsymbol{v}_i = (x_i, y_i, z_i)$ of the path. The Problem 3.13 can be formulated as follows,

$$\begin{aligned} &\underset{P}{\text{minimize}} \quad \alpha \cdot \mathrm{maxCurv}(P) + \beta \cdot \mathrm{maxTors}(P) + \gamma \cdot \mathrm{normLen}(P) \\ &\text{subject to} \quad \left| \mathrm{bspline}(P) \cap \bigcup_{i \in I} \mathrm{obstacle}_i \right| = 0. \end{aligned}$$

where $\mathrm{maxCurv}(P)$ is the peak of curvature of the B-spline constructed using P as control polygon, $\mathrm{maxTors}(P)$ is the peak in absolute value of the torsion and $\mathrm{normLen}(P)$ is the length of the control polygon P normalized as percent of the length of the initial status[14] . α, β and γ are fixed coefficients used for giving different weights in the optimization process to the curvature peak, torsion peak and length. The normalization of length is necessary for decoupling the weight of the length from the length of path.

Curvature and torsion, and thus also the relative peaks, are obtained in a discrete form. The B-spline curve is tabulated in a number of points that depends on the length of P by a constant, then for each point the curvature and torsion values are calculated.

Regarding the constraint, $\mathrm{bspline}(P)$ is the set of points of the *B-spline*, using P as control polygon, and $\mathrm{obstacle}_i$ is the area of the i^{th} of m obstacles, and $I = \{1, \ldots, m\}$.

We need to build now the Lagrangian function corresponding to Eq. (3.14), for conciseness we define the function

$$\mathrm{gain}(P) = \alpha \cdot \mathrm{maxCurv}(P) + \beta \cdot \mathrm{maxTors}(P) + \gamma \cdot \mathrm{normLen}(P) \quad (5.19)$$

the constraint function is not negative, and is calculated as the ratio

$$\mathrm{constraint}(P) = \frac{|\{\mathbf{p} \in \mathrm{spline}(P) \ : \exists i \text{ s.t. } \mathbf{p} \in \mathrm{obstacle}_i\}|}{|\{\mathbf{p} \in \mathrm{spline}(P)\}|}. \quad (5.20)$$

14 If the user chooses to minimize the arc length, then $\mathrm{normLen}(P)$ becomes the length of the B-spline curve.

The points $\mathbf{p}$ of the spline are calculated in a discrete form, like curvature and torsion. Thus the constraint depends on the tabulation of the curve and it is also possible to have borderline cases where the calculated constraint doesn't reflect the real situation[15].

The function in Eq. (5.20) is not negative, thus the Lagrangian function corresponding to Eq. (3.14) is

$$L_d(P, \lambda) = gain(P) + \lambda \cdot constraint(P). \tag{5.21}$$

5.6.2 *Annealing phase*

The target of the simulated annealing phase is to find the minimum saddle points in the curve represented by the Eq. (5.21).

Algorithm 9 Annealing

```
1:  procedure ANNEALING(x)
2:      λ ← initialLambda
3:      T ← initialTemperature
4:      while not terminationCondition() do
5:          for all number of trials do
6:              changeLambda ← True with changeLambdaProb
7:              if changeLambda then
8:                  λ′ ← neighbour(λ)
9:                  λ ← λ′ with probability e^−([energy(x,λ)−energy(x,λ′)]^+/T)
10:             else
11:                 x′ ← neighbour(x)
12:                 x ← x′ with probability e^−([energy(x′,λ)−energy(x,λ)]^+/T)
13:             end if
14:         end for
15:         T ← T · warmingRatio
16:     end while
17: end procedure
```

The Algorithm 9 is the annealing process, the input is the initial status of the system $\mathbf{x}$ - i.e. the initial configuration of the control polygon.

1. λ and the temperature are initialized on Line 2 and Line 2 respectively;

15 For instance if we have very thin obstacles, a curve can pass through it having few points (or even none) of it inside them.

2. the *while* on Line 4 is the main loop and the terminating condition is given by a minimum temperature or a minimum variation of energy between two iterations;

3. the *for* at Line 5 repeats the annealing move for a certain number of trials, on each iteration the algorithm probabilistically tries to make a move of the state of the system;
   - first on Line 6 it makes the choice if moving in the Lagrangian space or in the space of the path;
   - after that, based on the previous choice, the algorithm tries probabilistically to move the system in a neighbouring state: in the Lagrangian space at Line 8 or in the path space at Line 11;

4. finally at the end of every trial set, at Line 15, the temperature T is cooled by a certain factor.

The termination condition in Line 4 is triggered by a minimum variation of energy $\Delta energy$ between two consecutive iterations of the cycle. Also the termination is triggered when a minimum temperature is reached, this is for impose a limit on the number of cycles.

The choice of the neighbour is made probabilistically in the meaning that if the energy increases in the Lagrangian space or decreases in the path space, then the probability of choosing the new state is 1. If the energy decreases in the Lagrangian space or increases in the path space, then the new state is accepted with a probability that is[16]:

$$\exp(-\frac{\Delta energy}{T}).$$

The $neighbour$ function chooses a neighbour of the state and it is defined depending on the input:

- a neighbour of λ is a value that is equal to λ plus a perturbation in a range $[-maxLambdaPert, maxLambdaPert]$;
- a neighbour of the path is obtained picking randomly one of the vertices v_i, except the extremes v_0 and v_n, then choosing an angle in $[0, 2\pi]$ and a distance uniformly in a specific range, and finally moving v_i by the chosen values.

16 Note that $[x]^+ = \max(0, x)$.

The $energy$ function is equivalent to L_d in the Eq. (5.21):

$$energy(x, \lambda) = gain(P) + \lambda \cdot constraint(P). \tag{5.22}$$

The annealing process finds a saddle point by probabilistically increasing the energy, when λ is moved, and decreasing the energy when the points are moved.

Complexity considerations

For this solution we have still the costs of Eq. (5.1) and Eq. (5.4) for the creation and pruning of the graph G. In addition we need to apply Dijkstra's algorithm in G for obtaining the initial path P with the cost of Eq. (5.12).

Regarding the annealing phase, for each *step* (an iteration of Line 4 in Algorithm 9) we have a fixed number of *trials* (the iterations of Line 5). For each trial we need to calculate the value of the energy, Eq. (5.22), that is the sum of the gain and the constraint.

For the gain, Eq. (5.19), we need to calculate the values of curvature and torsion for every tabulated point. Furthermore there is also a cost[17] of $\mathcal{O}(|P|)$ for calculating the length of the control polygon. The cost for calculating the gain is thus

$$\mathcal{O}(|Sp| + |P|)$$

where Sp is the set of the tabulated points of the curve. The number of points in Sp depends on the length of the control polygon $len(P)$. Thus we have a cost of $\mathcal{O}(len(P) + |P|)$, but in the worst case $|P| = \mathcal{O}(|V|) = \mathcal{O}(|Ob|)$, thus the cost is

$$\mathcal{O}(len(P) + |P|) = \mathcal{O}(len(P) + |Ob|). \tag{5.23}$$

Regarding the constraint, Eq. (5.20), we need to calculate if every point of the curve is inside an obstacle. This means a cost of

$$\mathcal{O}(len(P)|Ob|). \tag{5.24}$$

The total cost for the calculation of the annealing phase is thus

$$\mathcal{O}(\#steps \cdot \#trials \cdot (len(P)|Ob|)),$$

17 Only if the user do not choose to minimize the arc length.

but the number of steps and trials are bounded by constants[18]. Thus the cost becomes

$$\mathcal{O}(len(P)|Ob|). \tag{5.25}$$

The total cost for the solution is

$$\mathcal{O}(k|Ob|^2 + len(P)|Ob|). \tag{5.26}$$

Likewise the previous solutions, if we have that k is a constant then the total cost become

$$\mathcal{O}(|Ob|^2 + len(P)|Ob|). \tag{5.27}$$

Description	Cost	Reference
Creation of G	$\mathcal{O}(\|Ob\| \log \|Ob\|)$	Eq. (5.1)
Pruning of G	$\mathcal{O}(k\|Ob\|^2)$	Eq. (5.4)
Routing in G	$\mathcal{O}(k\|Ob\| + \|Ob\| \log \|Ob\|)$	Eq. (5.12)
Gain	$\mathcal{O}(len(P) + \|P\|) = \mathcal{O}(len(P) + \|Ob\|)$	Eq. (5.23)
Constraint	$\mathcal{O}(len(P)\|Ob\|)$	Eq. (5.24)
Annealing	$\mathcal{O}(len(P)\|Ob\|)$	Eq. (5.25)
Total	$\mathcal{O}(k\|Ob\|^2 + len(P)\|Ob\|)$	Eq. (5.26)
Total (k costant)	$\mathcal{O}(\|Ob\|^2 + len(P)\|Ob\|)$	Eq. (5.27)

Table 4.: Resume of the costs for solution three

In Table 4 we resumed all the costs. It is difficult to quantitatively compare the cost of this solution with the previous ones. This is due to the presence of the factor $len(P)$, that depends on the geometry of the scene. Anyway we can asserts that this solution is more complex than the previous two by a term $\mathcal{O}(len(P)|Ob|)$.

Furthermore also in this solution we can divide the algorithm in two parts:

1. first we can construct only once G with cost $\mathcal{O}(|Ob|^2)$;
2. then we can use it for different situations with cost $|Ob| \log |Ob| + len(P)|Ob|$.

18 Although such constants can be very high.

Part III

EVALUATION

6

CODE STRUCTURE

We designed the code with an Object Oriented Programming (OOP) methodology in Python 3 (`https://www.python.org/`). A versatile language with a strong appeal on scientific community, easy to learn and with an increasing active community of developers behind. We relied on SciPy (`https://www.scipy.org/`) and NumPy (`http://www.numpy.org/`) libraries for taking care of different numerical methods. Furthermore we used NetworkX library (`http://networkx.github.io/`) to represent graphs and to route in them. Regarding the graphic output we used VTK (`http://www.vtk.org/`) bindings in Python.

The main class is `Voronizator`, it maintains the status of the scene and provides all the methods for the interface with users.

- `addBoundingBox` is used for adding a bounding box at specified coordinates to the scene.
- `addPolyhedron` is used for adding a new obstacle to the scene, it required a `Polyhedron` object as argument.
- `setPolyhedronsSites` add the sites for the VD to the scene.
- The method `makeVoroGraph` is used for creating the graphs G and G_t using the algorithms described in Section 5.1.1 and Section 5.1.2.
- `setAdaptivePartition` and `setBsplineDegree` are used for selecting between uniform or adaptive knot partition, and for choosing the desired degree of the curve.
- `extractXmlTree` and `importXmlTree` are used for saving and restoring the scene in XML format.
- All the other methods `plot*` are used for drawing the different elements of the scene. They require a plotter as argument.

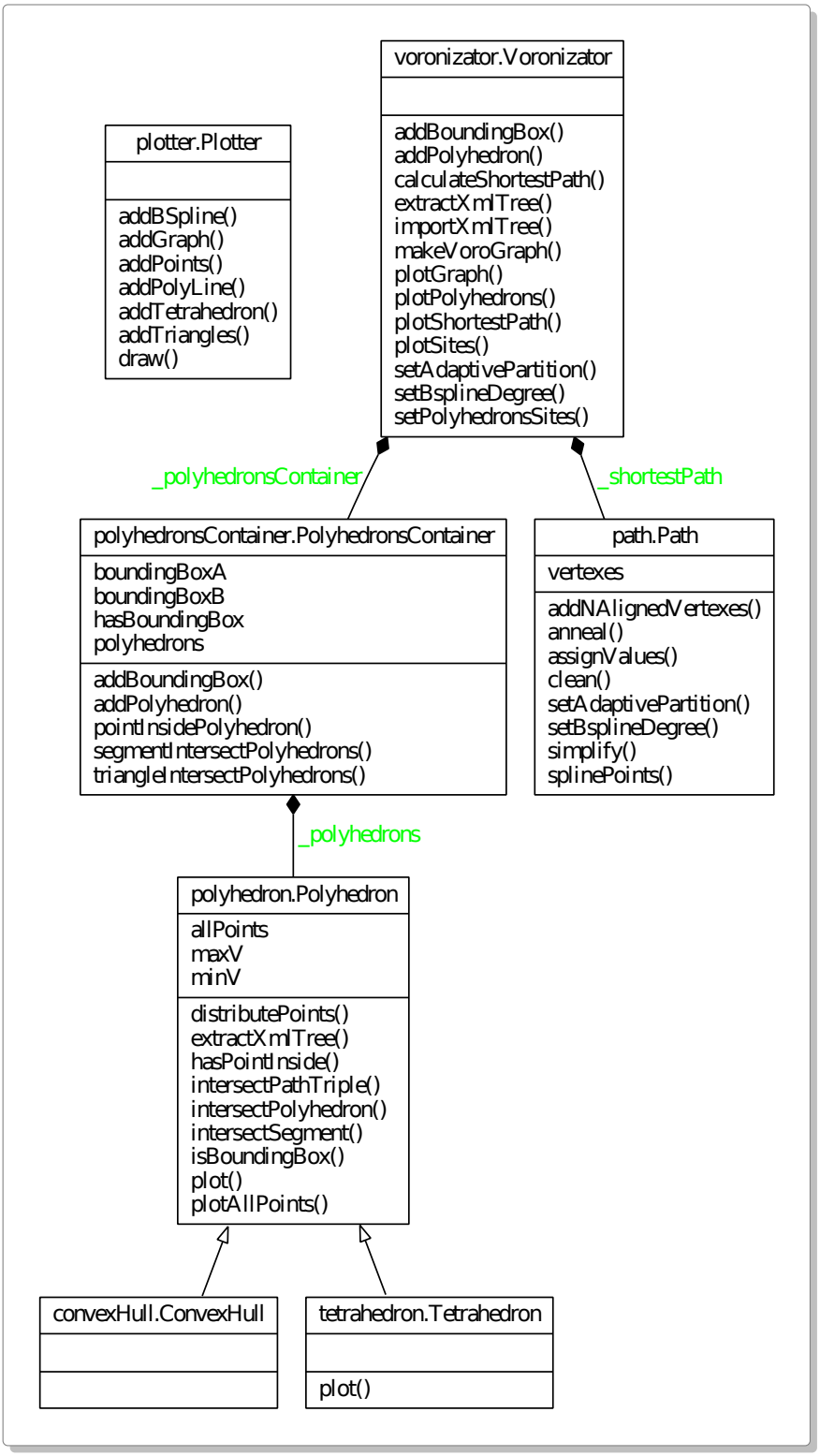


Figure 24.: Excerpt UML of the project

The class `Plotter` provides the interface for drawing all the necessary elements using VTK.

The class `Polyhedron` represents a single obstacle (that can also be one of the two subclasses `ConvexHull` and `Tetrahedron`). it provides all the necessary methods for performing geometry checks of point inclusion and intersection with segments, triangles, and other obstacles.

The class `Path` represents the control polygon of the curve. It provides the methods `addNAlignedVertexes` and `simplify` that perform respectively the degree increase (Section 5.3) and the post processing (Section 5.5). The method `clean` is necessary for the second solution described in Section 5.2.2. Furthermore this class provides also the functionality for optimizing the curve using the SA (Section 5.6) with the method `anneal`.

Furthermore we provide scripts for the creation of random scenes and for the execution of the different methods.

7

TESTING

For evaluating the project we made the tests on Table 7, Table 8, Table 9,Table 10. Where

- *Scene* refers to which scene of Table 5 was used;
- $\mathbf{s} \rightarrow \boldsymbol{e}$ indicates the starting and ending points
- *Deg.* is the degree of the B-spline curve;
- *Met.* is the method used, where
  - Method A is Dijkstra in G';
  - Method B is Dijkstra in G;
  - Method C is Simulated Annealing;
- *P. p.* indicate if the post processing is used (✓) or not (✗);
- *Part.* indicate if is used the uniform knot partition (**U**) or the adaptive one (**A**)
- *Config.* is used only for method C and indicates the used annealing configuration from Table 6.

The Table 5 gives some data about the scenes.

- *B.b. A* and *B.b. B* are the extremes of the bounding box.
- *Min. obs. rad.* and *Max. obs. rad.* are respectively the minimum radius and the maximum radius of the obstacles in the scene. For scenes 1, 1b, and 2 the obstacles are not-intersecting tetrahedrons generated randomly with the specific parameters. The scene 3 is a bucket-shaped obstacle with center in $[0.5, 0.5, 0.5]$, with width 0.2, height 0.4 and thickness 0.2.

- # *obs.* is the number of obstacles in the scene.
- *Max. empty area* is the maximum empty area for the distribution of the Voronoi sites as explained in Section 5.1.1.

Scene	B.b. A	B.b. B	Min. obs. rad.	Max. obs. rad.	# obs.	Max. empty area
1	$[-0.1, -0.1, -0.1]$	$[1.1, 1.1, 1.1]$	0.2	0.4	10	0.1
1b	$[-0.1, -0.1, -0.1]$	$[1.1, 1.1, 1.1]$	0.2	0.4	10	0.01
2	$[-0.1, -0.1, -0.1]$	$[1.1, 1.1, 1.1]$	0.2	0.4	100	0.1
3	$[0, 0, 0]$	$[1, 1, 1]$	-	-	1	0.1

Table 5.: Testing scenes.

Config.	T_0	Trials	warm.	min T	min ΔE	λ pert	V pert fact	λ_0	λP	Len type	Ratios
1	10	10	0.7	1e−7	1e−6	1000	10	0	5e−2	arc	$[0.1, 0.1, 0.8]$
2	10	10	0.7	1e−7	1e−6	1000	10	0	5e−2	poly	$[0.1, 0.1, 0.8]$
2b	10	100	0.7	1e−7	1e−6	1000	100	0	5e−2	poly	$[0.1, 0.1, 0.8]$
3	10	10	0.7	1e−7	1e−6	1000	10	0	5e−2	arc	$[0.3, 0.3, 0.4]$
3b	10	10	0.9	1e−5	1e−6	1000	100	0	5e−2	arc	$[0.3, 0.3, 0.4]$

Table 6.: Annealing configurations.

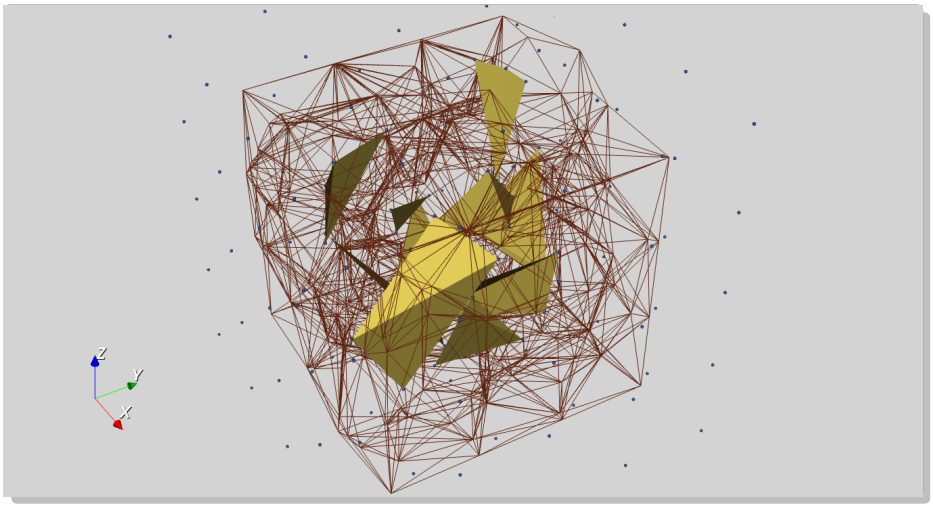


Figure 25.: Scene 1.

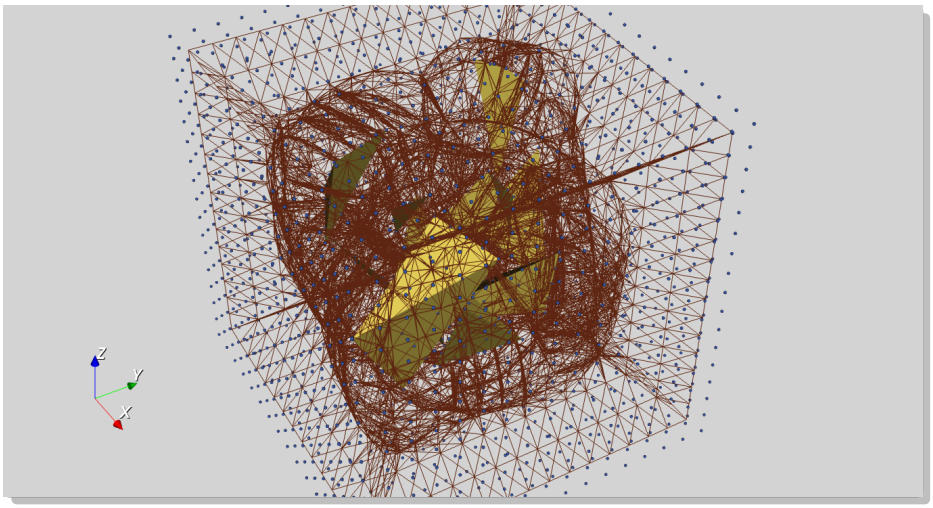


Figure 26.: Scene 1b.

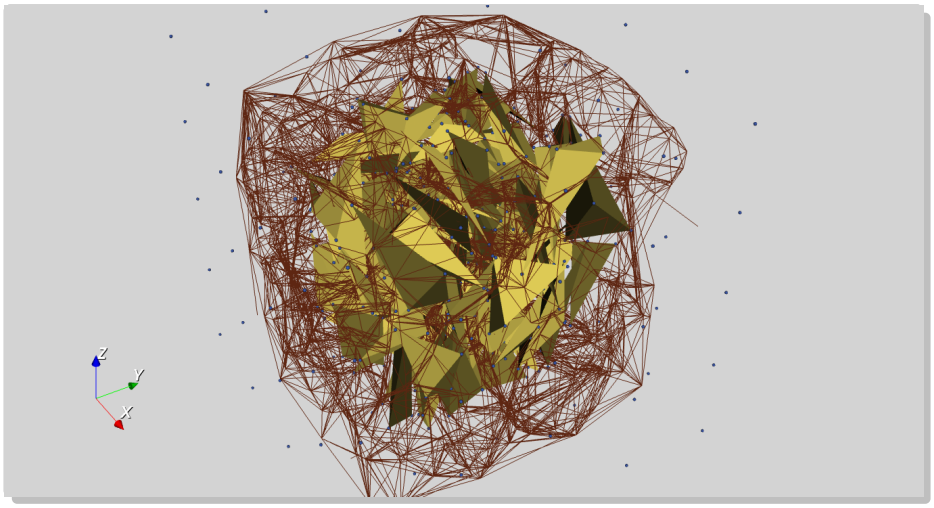


Figure 27.: Scene 2.

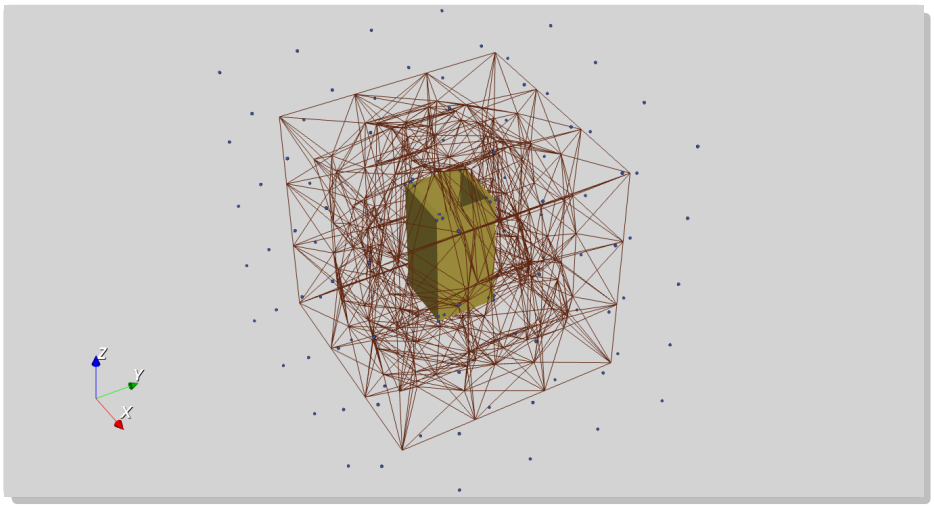


Figure 28.: Scene 3.

#	Scene	$s \rightarrow e$	Deg.	Met.	P. p.	Part.	Config.	figure
1	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	A	✗	**U**	-	Fig. 29
2	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	A	✓	**U**	-	Fig. 30
3	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	A	✗	**A**	-	Fig. 31
4	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	A	✓	**A**	-	Fig. 32
5	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	A	✗	**U**	-	Fig. 33
6	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	A	✓	**U**	-	Fig. 34
7	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	A	✗	**A**	-	Fig. 35
8	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	A	✓	**A**	-	Fig. 36
9	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	A	✗	**U**	-	Fig. 37
10	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	A	✓	**U**	-	Fig. 38
11	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	A	✗	**A**	-	Fig. 39
12	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	A	✓	**A**	-	Fig. 40
13	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✗	**U**	-	Fig. 41
14	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✓	**U**	-	Fig. 42
15	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✗	**A**	-	Fig. 43
16	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✓	**A**	-	Fig. 44
17	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✗	**U**	-	Fig. 45
18	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✓	**U**	-	Fig. 46
19	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✗	**A**	-	Fig. 47
20	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✓	**A**	-	Fig. 48
21	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✗	**U**	-	Fig. 49
22	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✓	**U**	-	Fig. 50
23	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✗	**A**	-	Fig. 51
24	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✓	**A**	-	Fig. 52

Table 7.: Resume of the tests.

#	Scene	$s \rightarrow e$	Deg.	Met.	P. p.	Part.	Config.	figure
25	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✗	**U**	-	Fig. 53
26	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✓	**U**	-	Fig. 54
27	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✗	**A**	-	Fig. 55
28	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✓	**A**	-	Fig. 56
29	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✗	**U**	-	Fig. 57
30	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✓	**U**	-	Fig. 58
31	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✗	**A**	-	Fig. 59
32	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✓	**A**	-	Fig. 60
33	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✗	**U**	-	Fig. 61
34	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✓	**U**	-	Fig. 62
35	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✗	**A**	-	Fig. 63
36	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✓	**A**	-	Fig. 64
37	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	B	✗	**U**	-	Fig. 65
38	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	B	✓	**U**	-	Fig. 66
39	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	B	✗	**A**	-	Fig. 67
40	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	B	✓	**A**	-	Fig. 68
41	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	B	✗	**U**	-	Fig. 69
42	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	B	✓	**U**	-	Fig. 70
43	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	B	✗	**A**	-	Fig. 71
44	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	B	✓	**A**	-	Fig. 72
45	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	B	✗	**U**	-	Fig. 73
46	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	B	✓	**U**	-	Fig. 74
47	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	B	✗	**A**	-	Fig. 75
48	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	B	✓	**A**	-	Fig. 76

Table 8.: Resume of the tests (continue).

#	Scene	$s \rightarrow e$	Deg.	Met.	P. p.	Part.	Config.	figure
49	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	A	✗	**U**	-	Fig. 77
50	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	A	✓	**U**	-	Fig. 78
51	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	A	✗	**A**	-	Fig. 79
52	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	A	✓	**A**	-	Fig. 80
53	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	A	✗	**U**	-	Fig. 81
54	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	A	✓	**U**	-	Fig. 82
55	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	A	✗	**A**	-	Fig. 83
56	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	A	✓	**A**	-	Fig. 84
57	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	A	✗	**U**	-	Fig. 85
58	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	A	✓	**U**	-	Fig. 86
59	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	A	✗	**A**	-	Fig. 87
60	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	A	✓	**A**	-	Fig. 88
61	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	B	✗	**U**	-	Fig. 89
62	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	B	✓	**U**	-	Fig. 90
63	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	B	✗	**A**	-	Fig. 91
64	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	B	✓	**A**	-	Fig. 92
65	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	B	✗	**U**	-	Fig. 93
66	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	B	✓	**U**	-	Fig. 94
67	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	B	✗	**A**	-	Fig. 95
68	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	B	✓	**A**	-	Fig. 96
69	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	B	✗	**U**	-	Fig. 97
70	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	B	✓	**U**	-	Fig. 98
71	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	B	✗	**A**	-	Fig. 99
72	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	B	✓	**A**	-	Fig. 100

Table 9.: Resume of the tests (continue).

#	Scene	$s \rightarrow e$	Deg.	Met.	P. p.	Part.	Config.	figure
73	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	C	-	**U**	1	Fig. 101
74	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	C	-	**U**	1	Fig. 102
75	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	C	-	**U**	1	Fig. 103
76	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	C	-	**U**	2	Fig. 104
77	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	C	-	**U**	2	Fig. 105
78	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	C	-	**U**	2	Fig. 106
79	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	C	-	**U**	3	Fig. 107
80	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	C	-	**U**	3	Fig. 108
81	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	C	-	**U**	3	Fig. 109
82	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	C	-	**U**	1	Fig. 110
83	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	C	-	**U**	1	Fig. 111
84	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	C	-	**U**	1	Fig. 112
85	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	C	-	**U**	2	Fig. 113
86	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	C	-	**U**	2	Fig. 114
87	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	C	-	**U**	2	Fig. 115
88	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	C	-	**U**	3	Fig. 116
89	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	C	-	**U**	3	Fig. 117
90	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	C	-	**U**	3	Fig. 118
91	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	C	-	**U**	1	Fig. 119
92	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	C	-	**U**	1	Fig. 120
93	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	C	-	**U**	1	Fig. 121
94	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	C	-	**U**	2b	Fig. 122
95	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	C	-	**U**	2b	Fig. 123
96	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	C	-	**U**	2b	Fig. 124
97	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	C	-	**U**	3b	Fig. 125
98	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	C	-	**U**	3b	Fig. 126
99	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	C	-	**U**	3b	Fig. 127

Table 10.: Resume of the tests (continue).

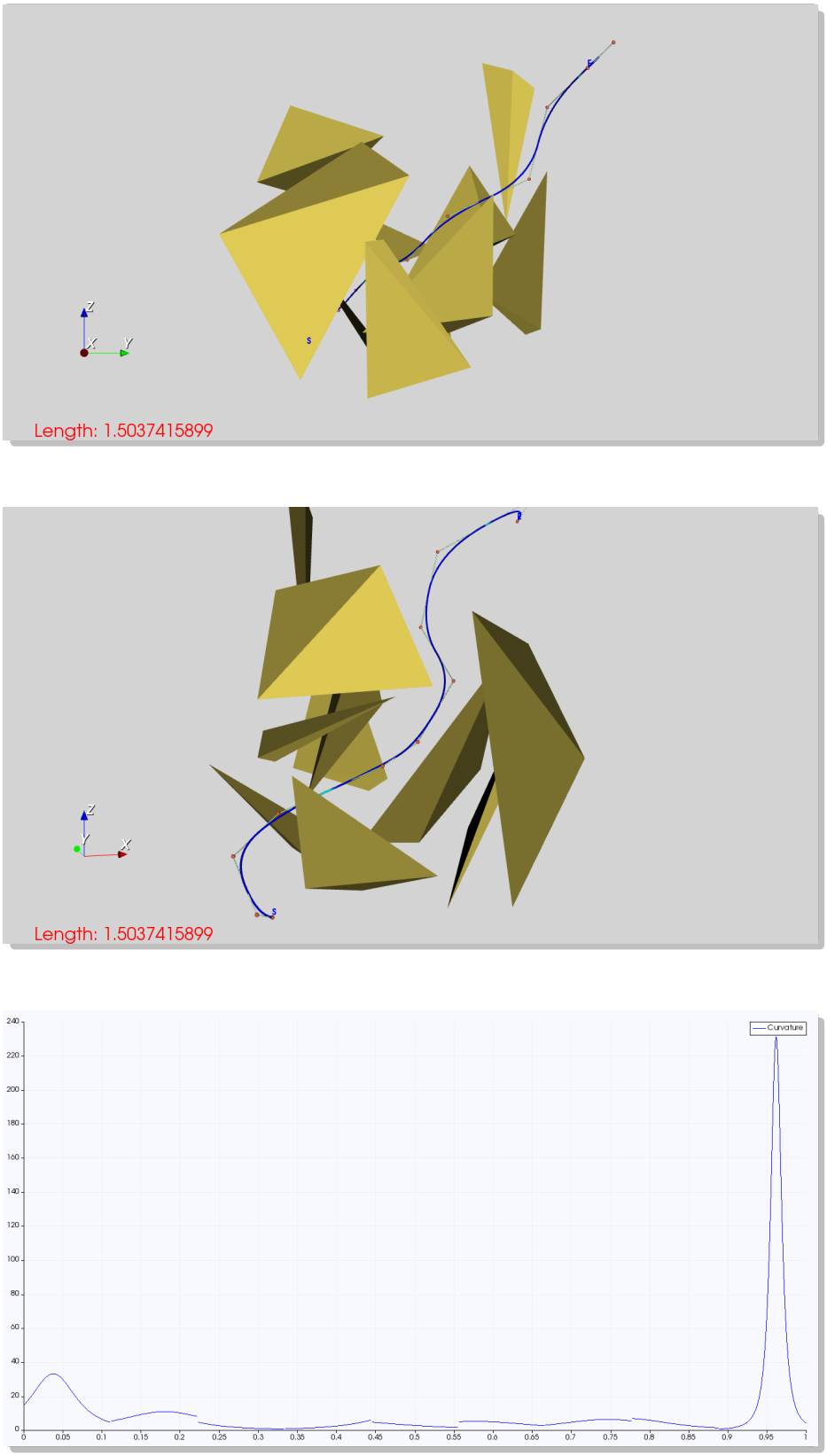


Figure 29.: Test 1; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. A; Post proc. ✗; Part. **U**.

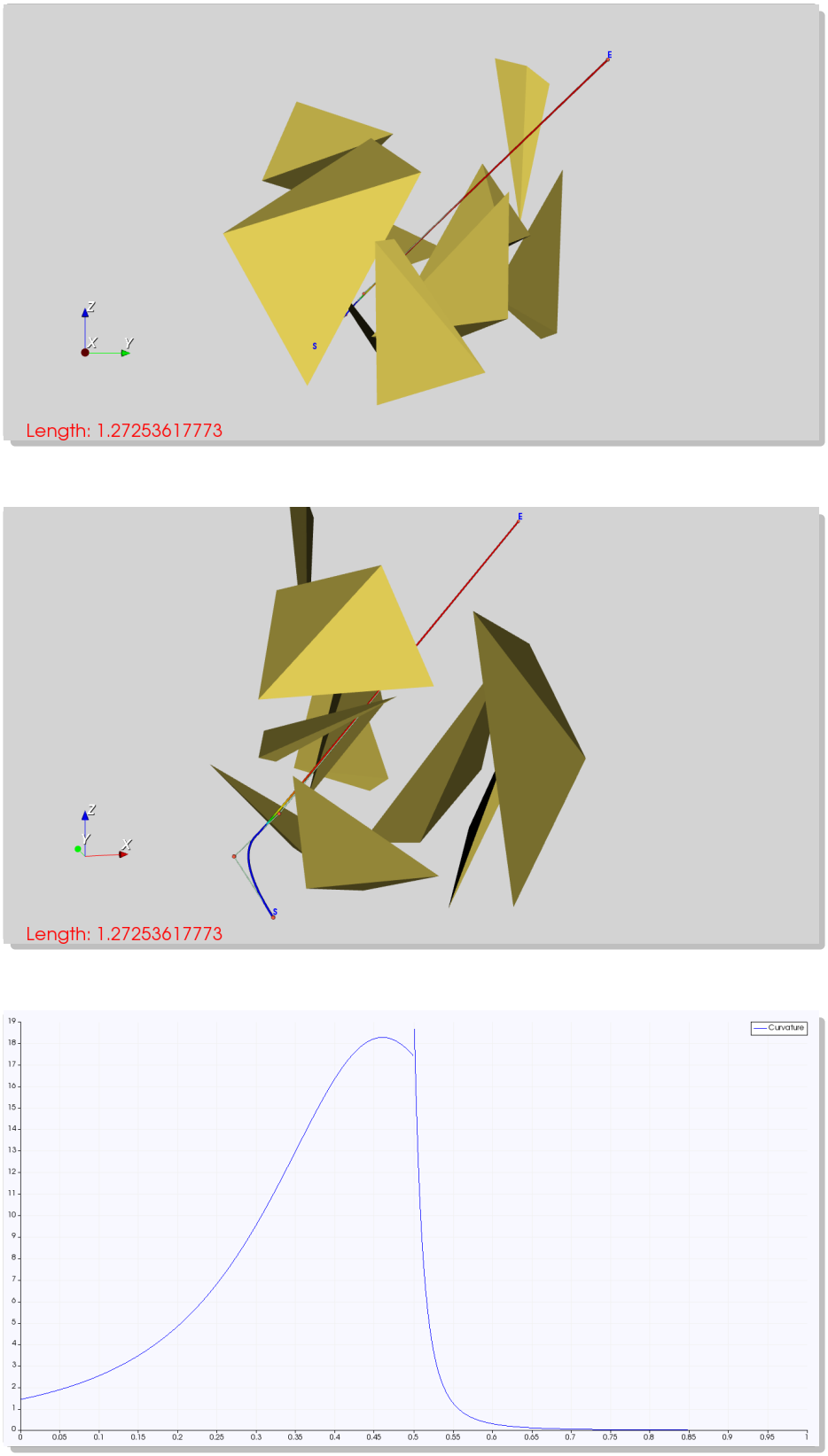
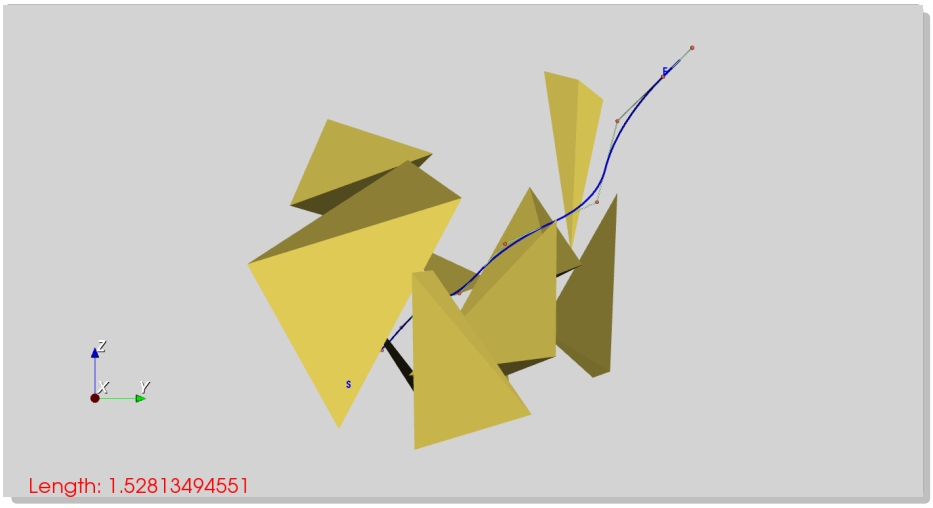
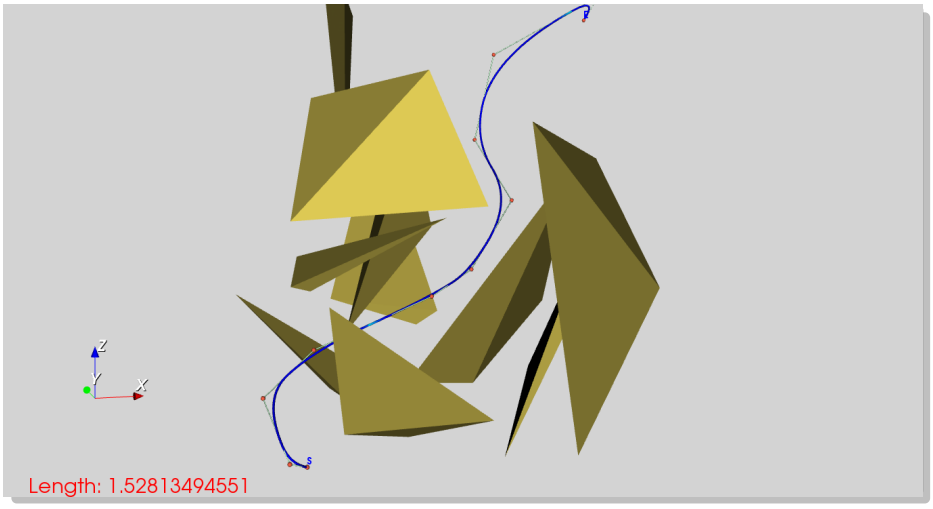


Figure 30.: Test 2; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. A; Post proc. ✓; Part. **U**.





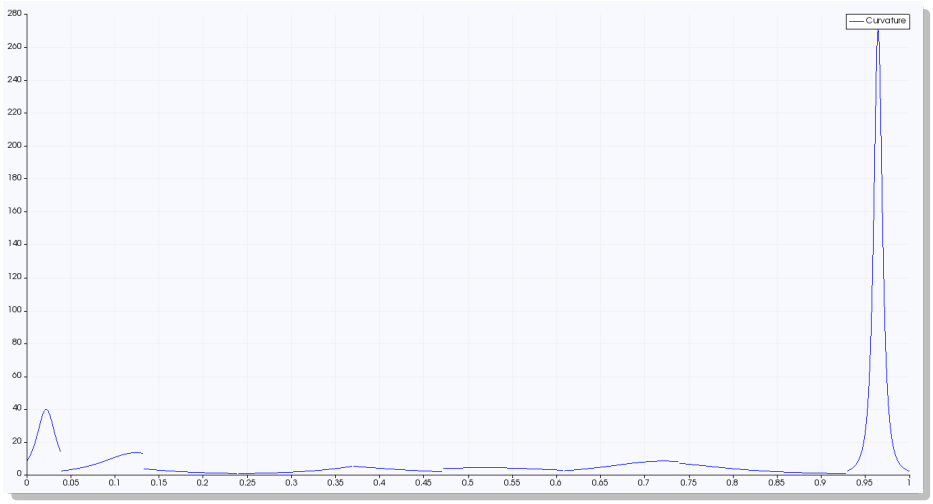


Figure 31.: Test 3; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. A; Post proc. ✗; Part. **A**.

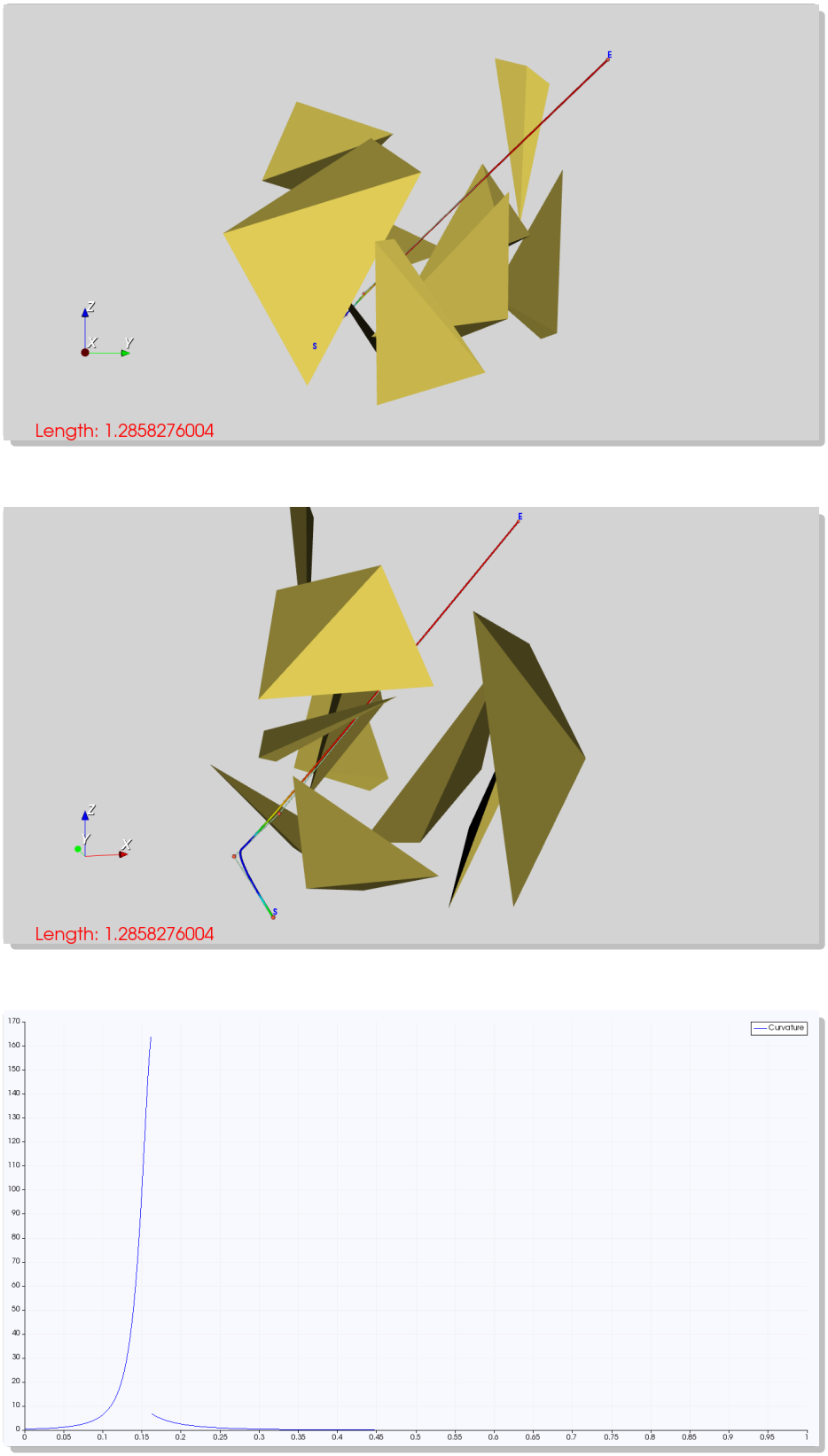
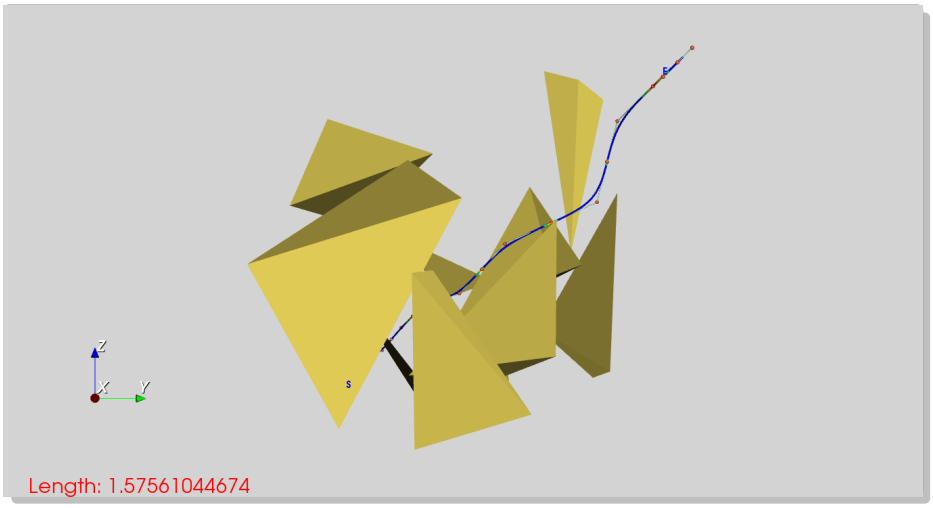
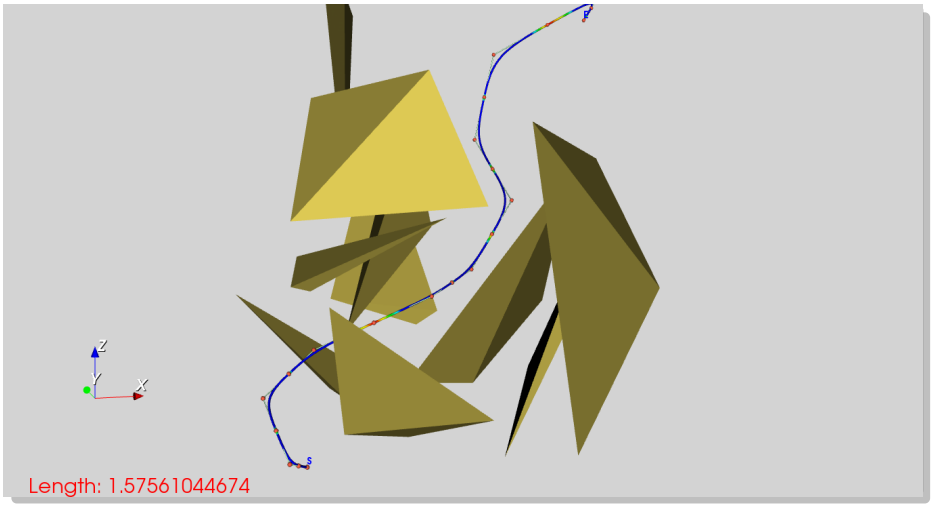


Figure 32.: Test 4; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. A; Post proc. ✓; Part. **A**.





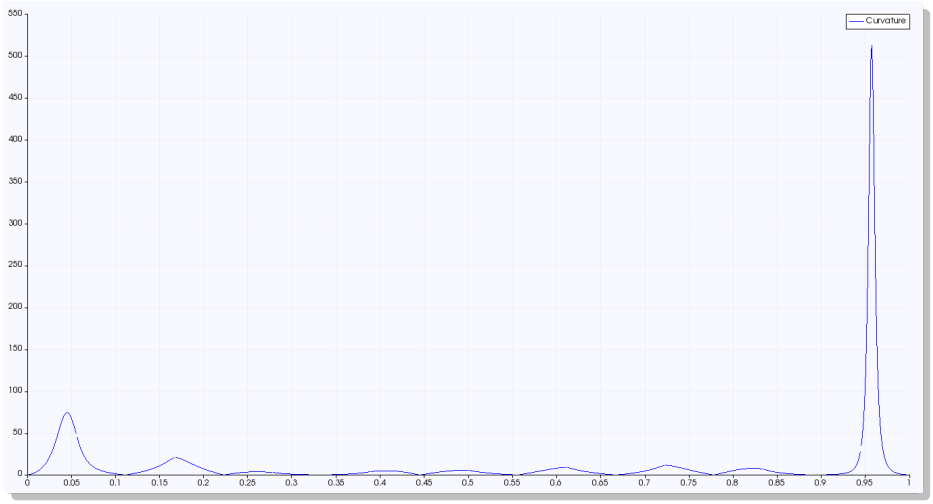


Figure 33.: Test 5; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. A; Post proc. ✗; Part. **U**.

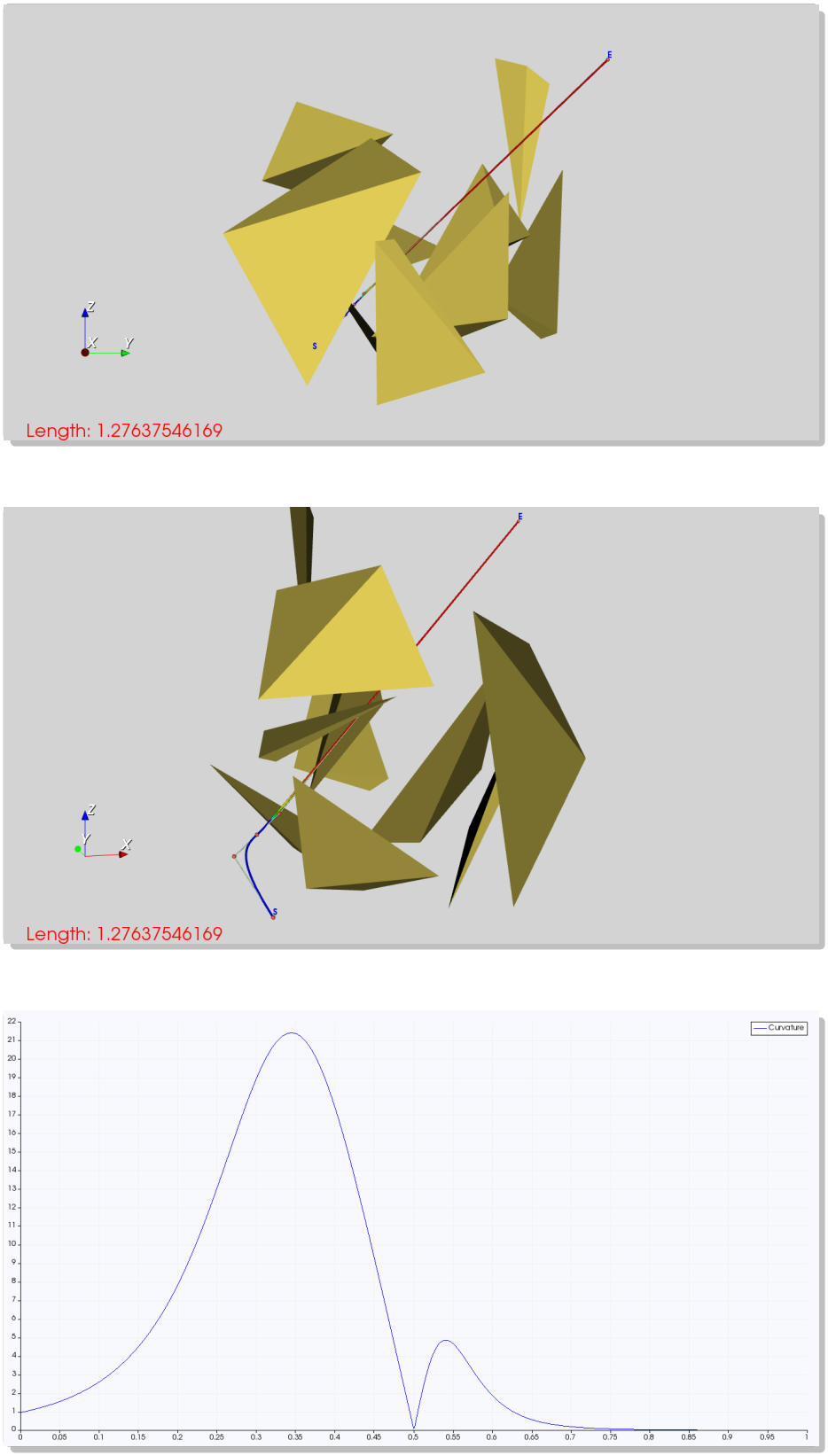


Figure 34.: Test 6; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. A; Post proc. ✓; Part. **U**.

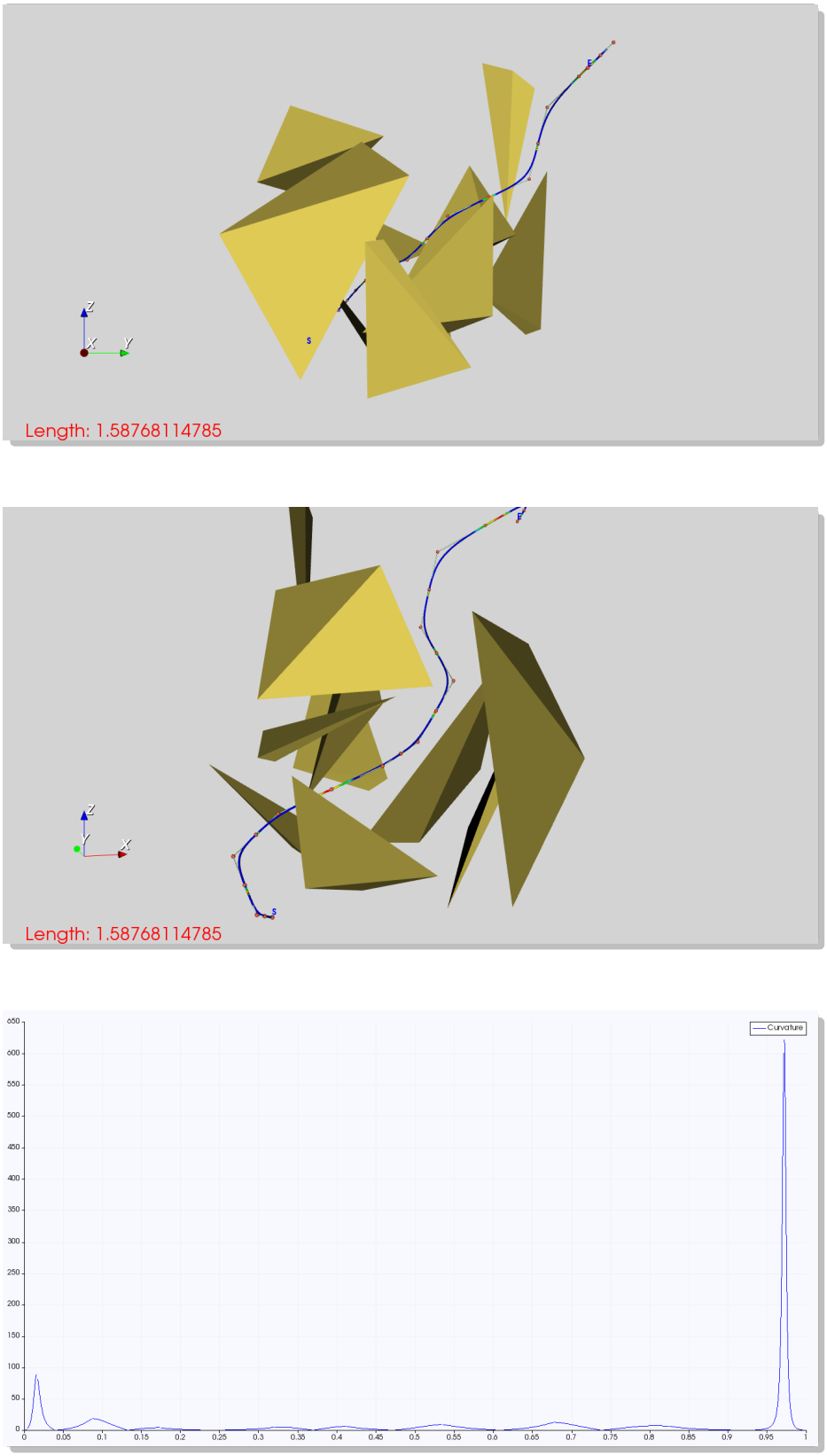


Figure 35.: Test 7; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. A; Post proc. ✗; Part. **A**.

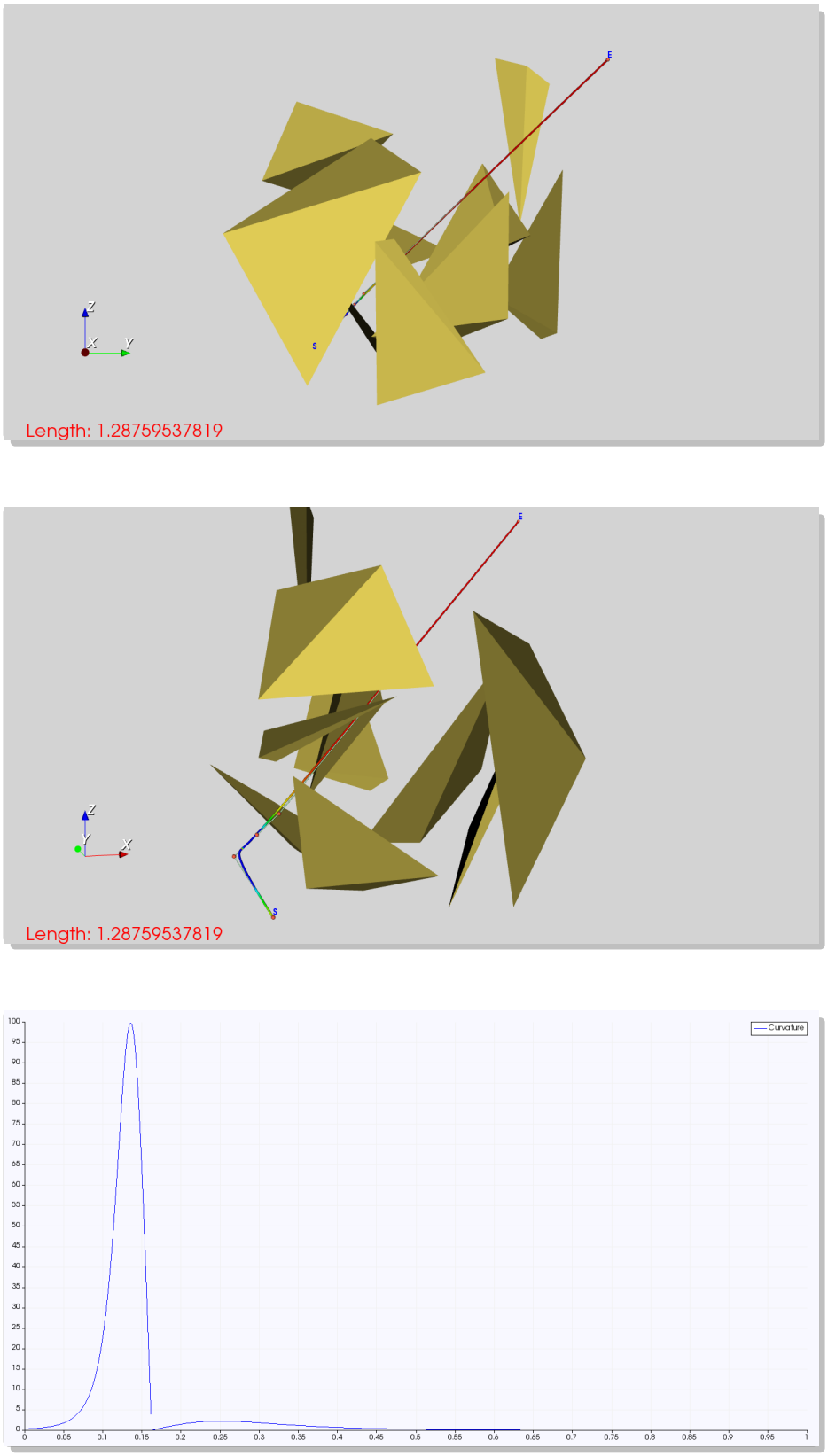


Figure 36.: Test 8; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. A; Post proc. ✓; Part. **A**.

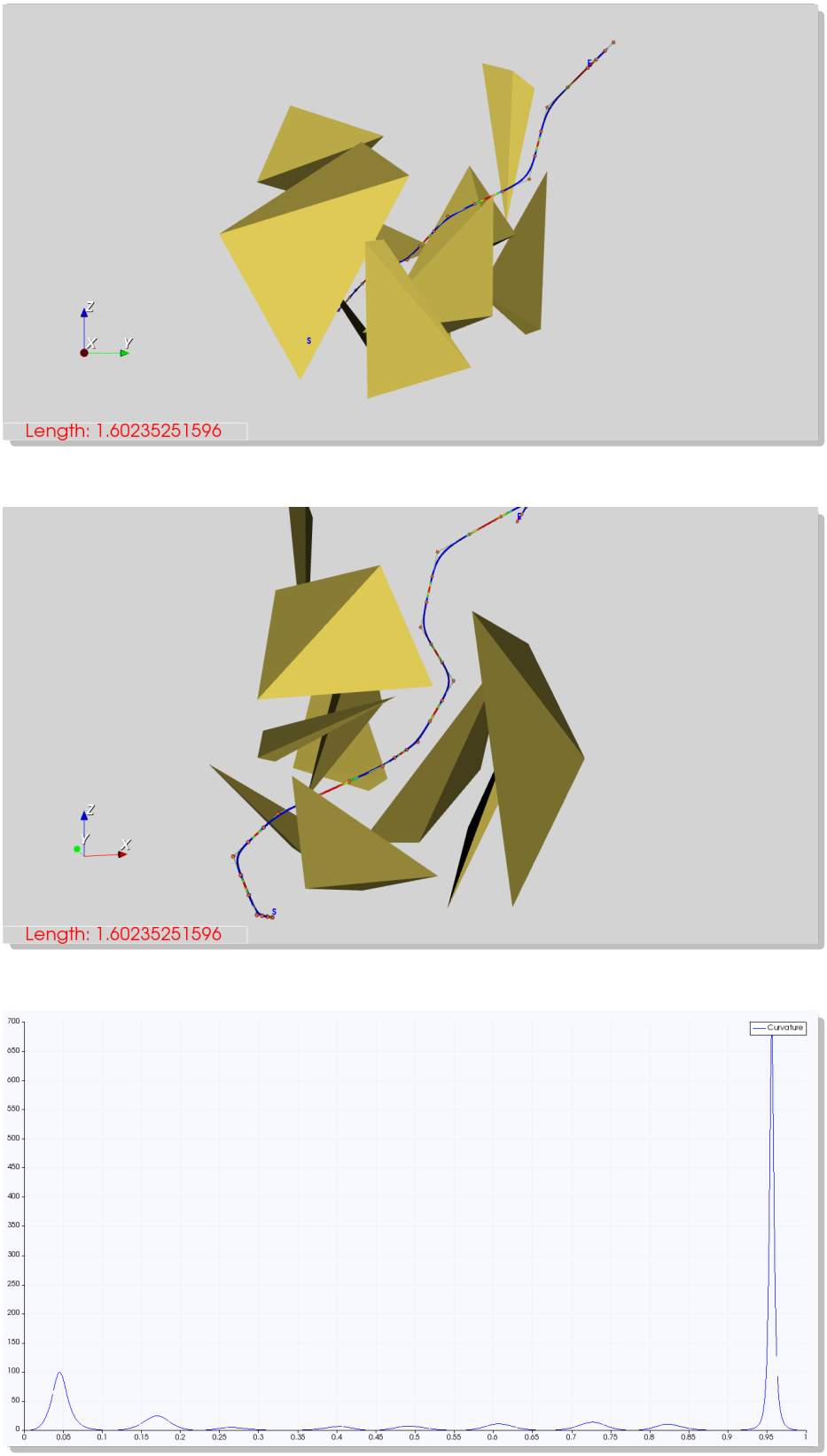


Figure 37.: Test 9; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. A; Post proc. ✗; Part. **U**.

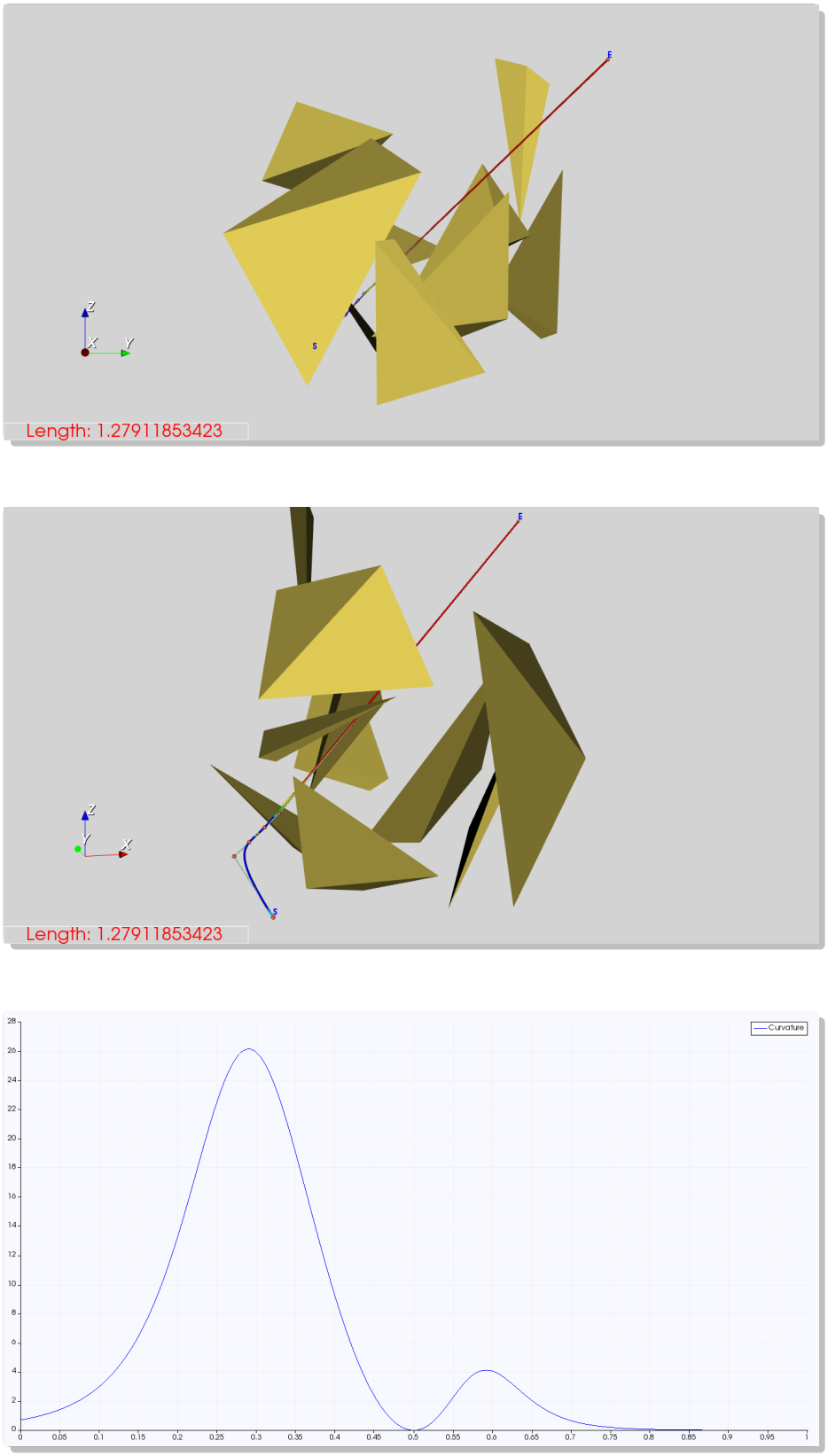
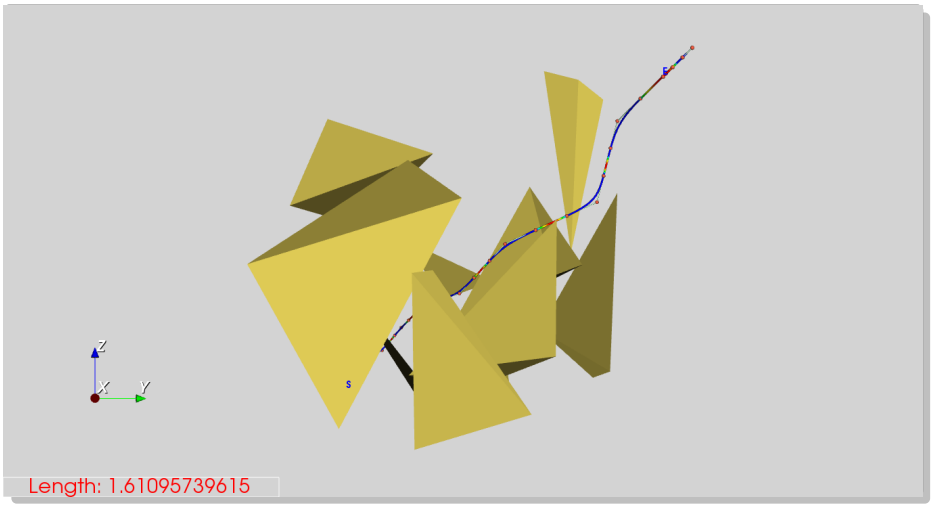
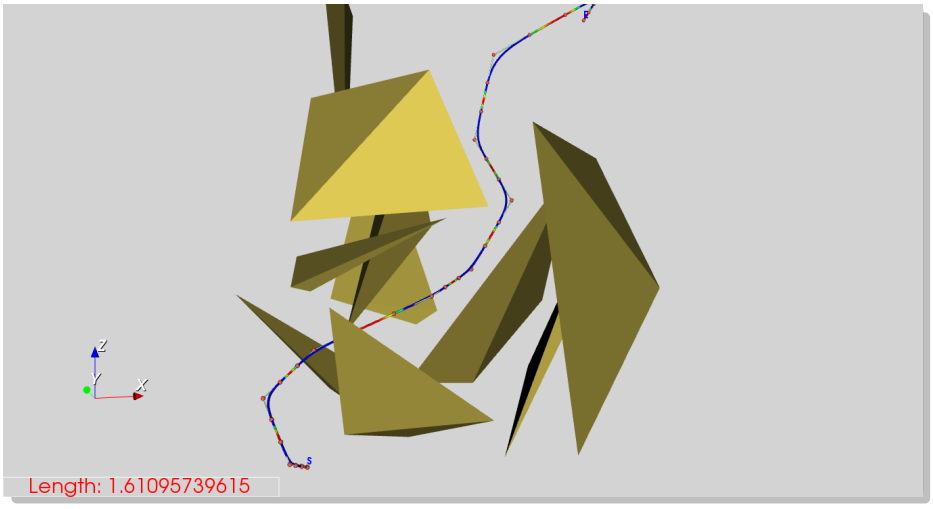


Figure 38.: Test 10; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. A; Post proc. ✓; Part. **U**.





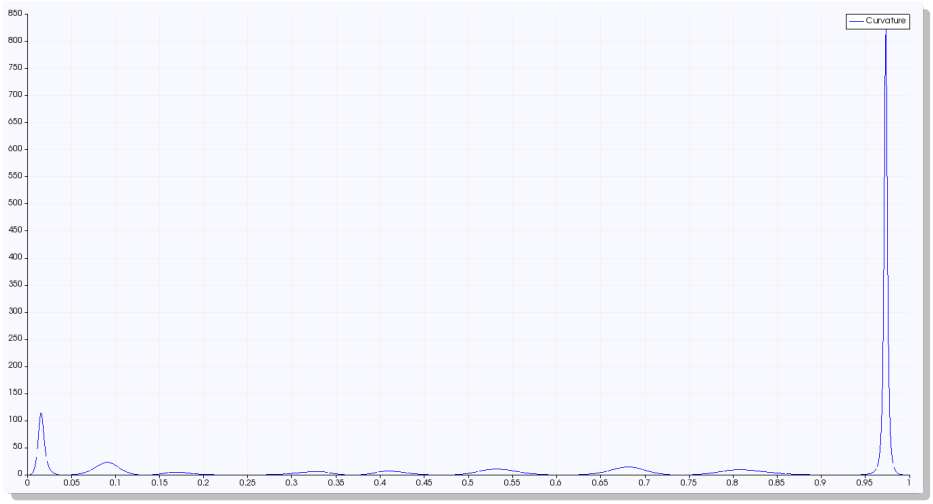


Figure 39.: Test 11; Scene 1; **s** → **e** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. A; Post proc. ✗; Part. **A**.

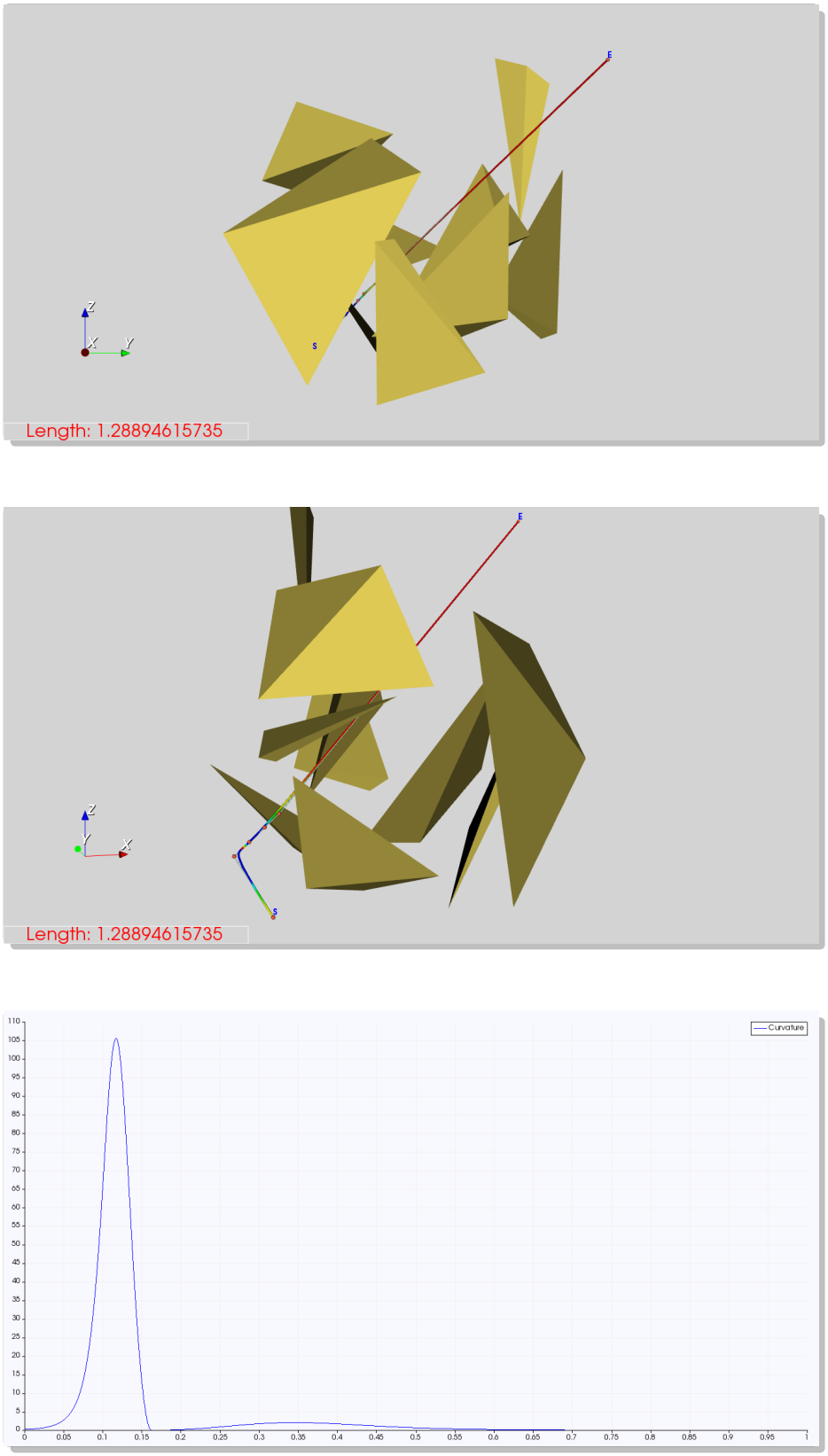


Figure 40.: Test 12; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. A; Post proc. ✓; Part. **A**.

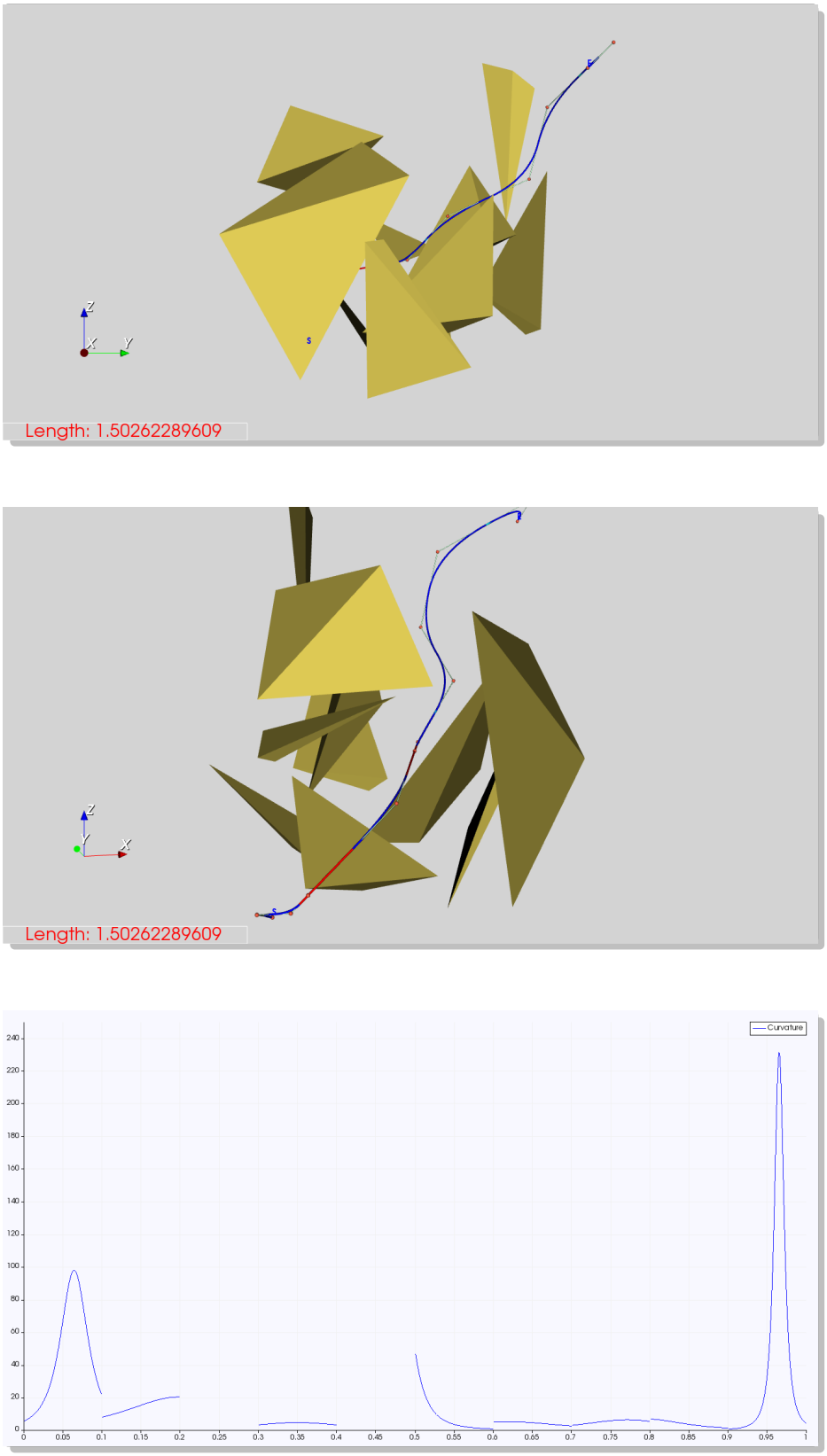


Figure 41.: Test 13; Scene 1; **s** → **e** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. B; Post proc. ✗; Part. **U**.

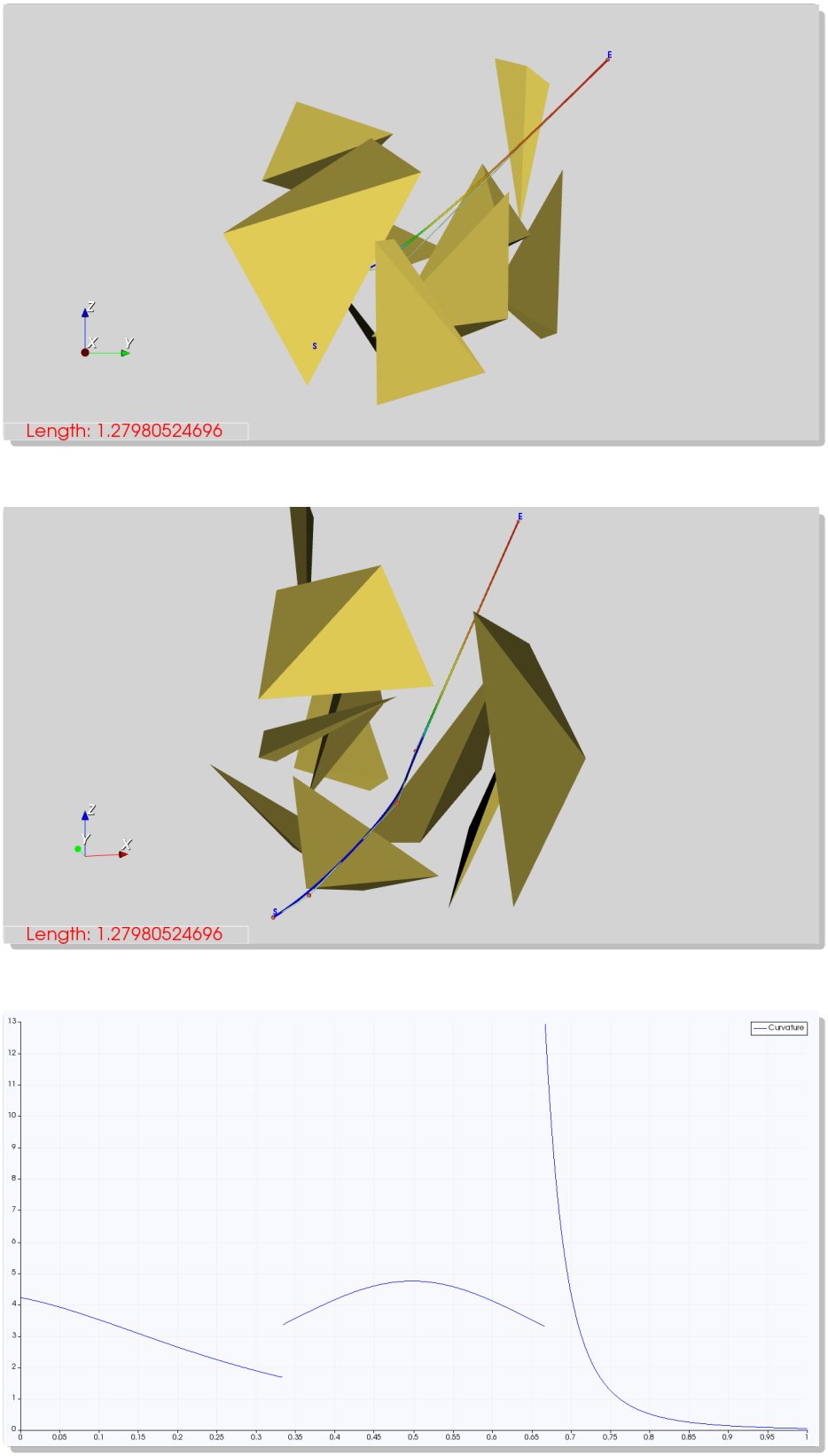


Figure 42.: Test 14; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. B; Post proc. ✓; Part. **U**.

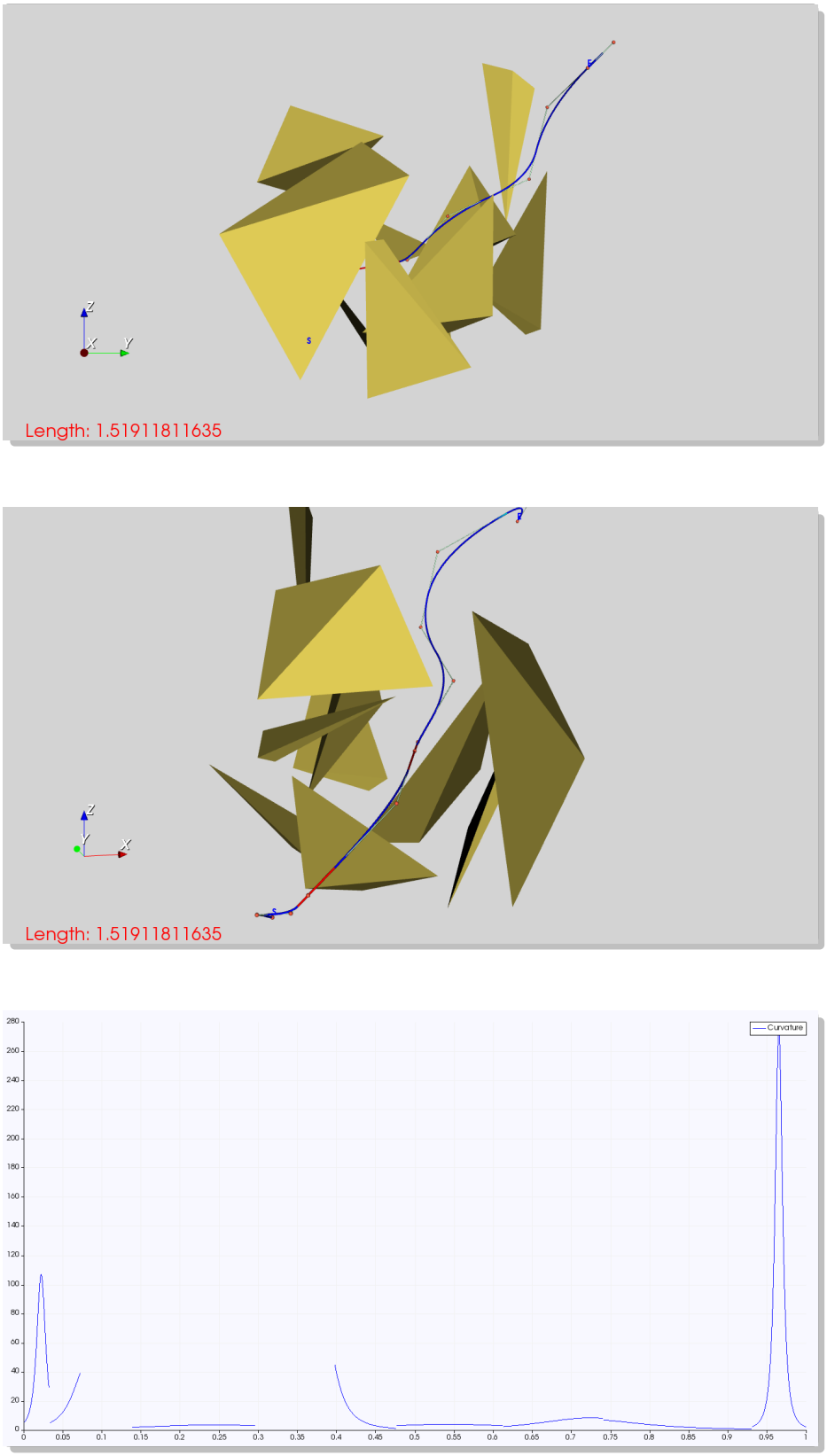


Figure 43.: Test 15; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. B; Post proc. ✗; Part. **A**.

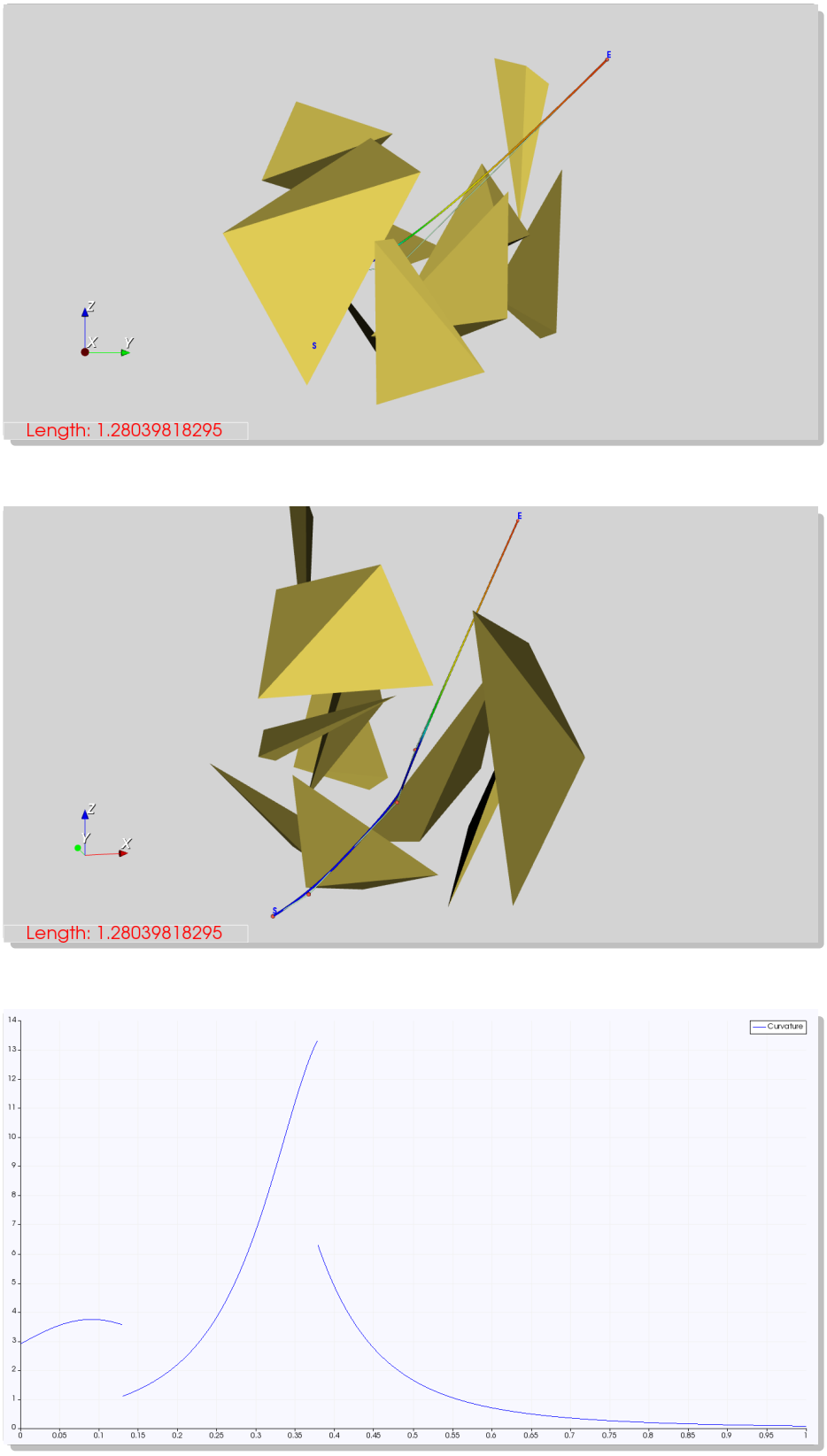
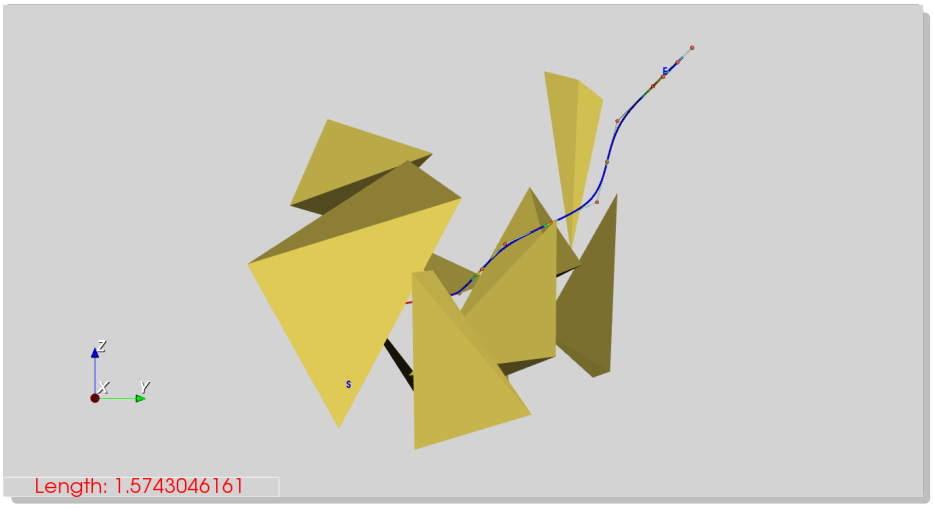
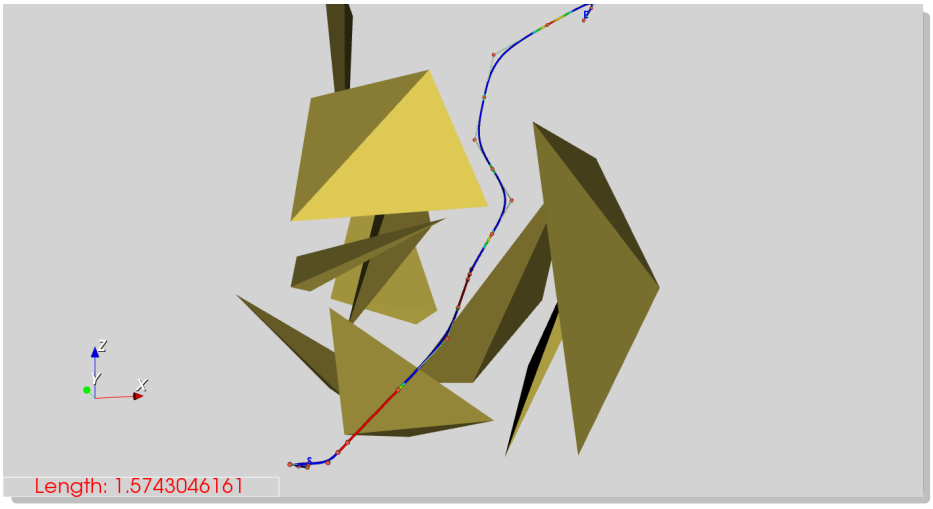


Figure 44.: Test 16; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. B; Post proc. ✓; Part. **A**.





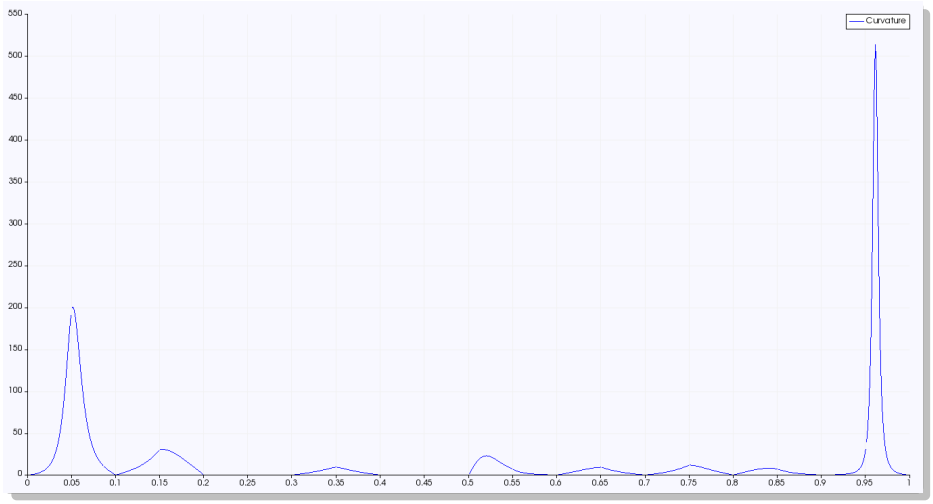


Figure 45.: Test 17; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. B; Post proc. ✗; Part. **U**.

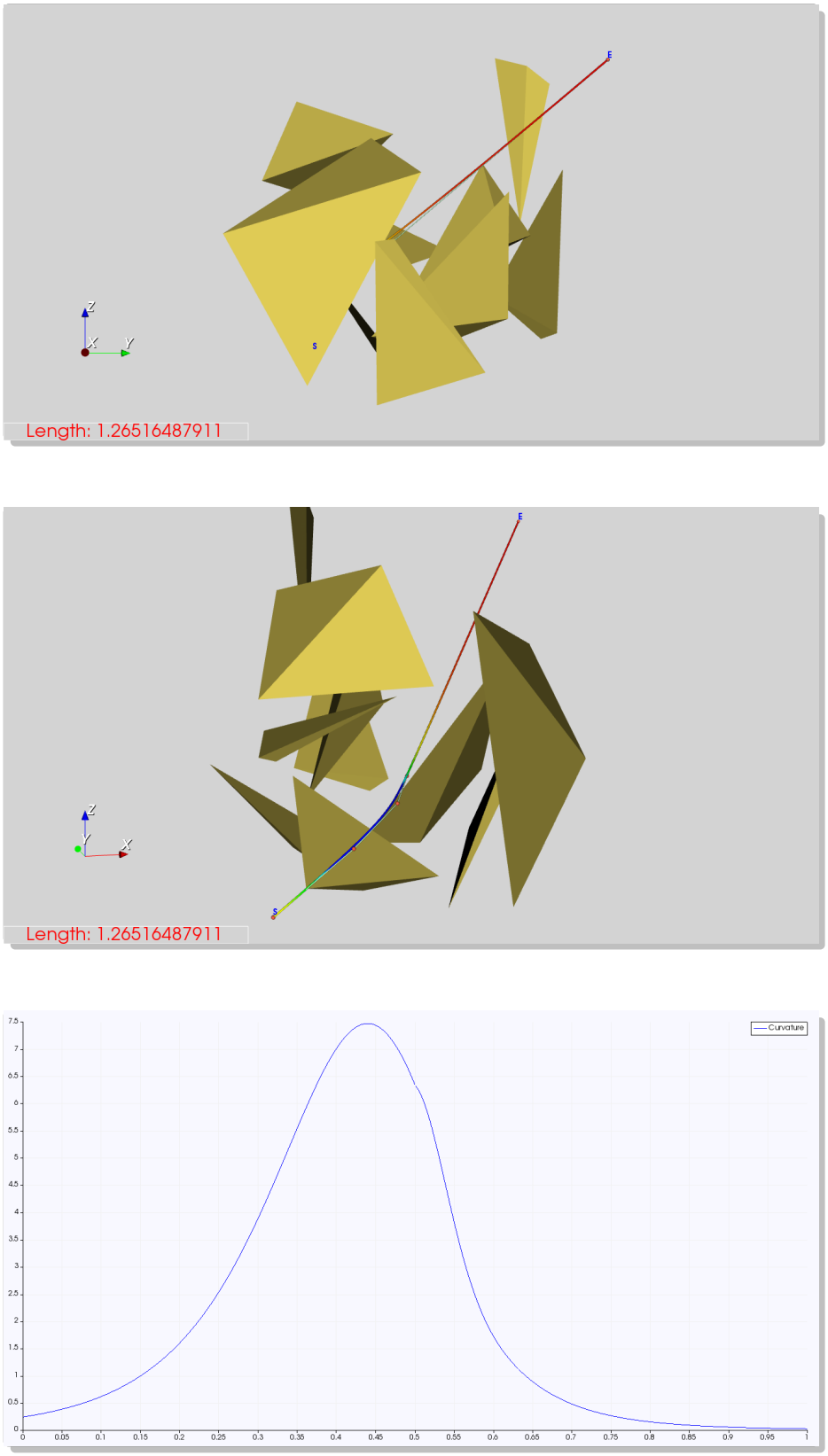


Figure 46.: Test 18; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. B; Post proc. ✓; Part. **U**.

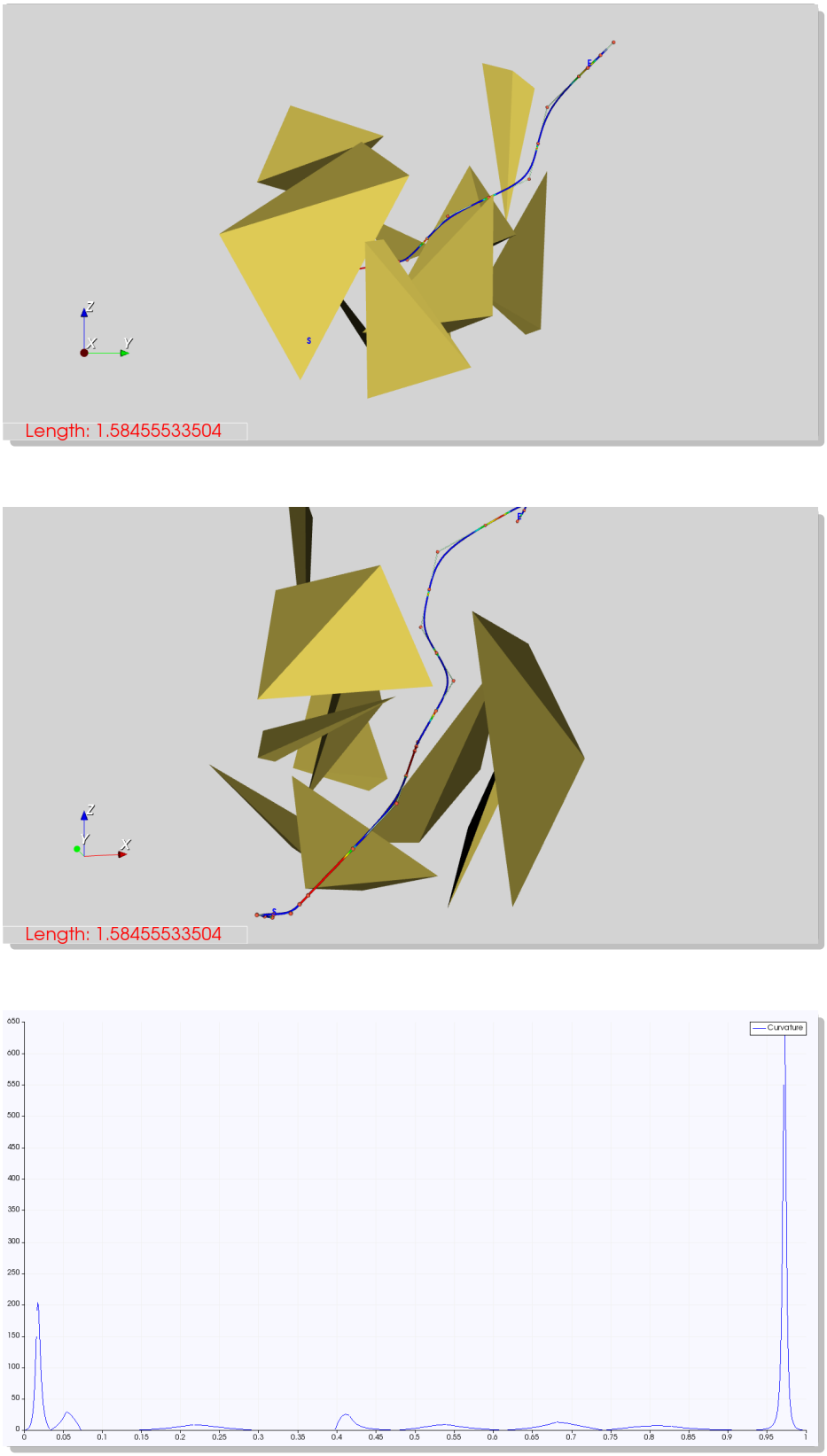


Figure 47.: Test 19; Scene 1; **s** → **e** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. B; Post proc. ✗; Part. **A**.

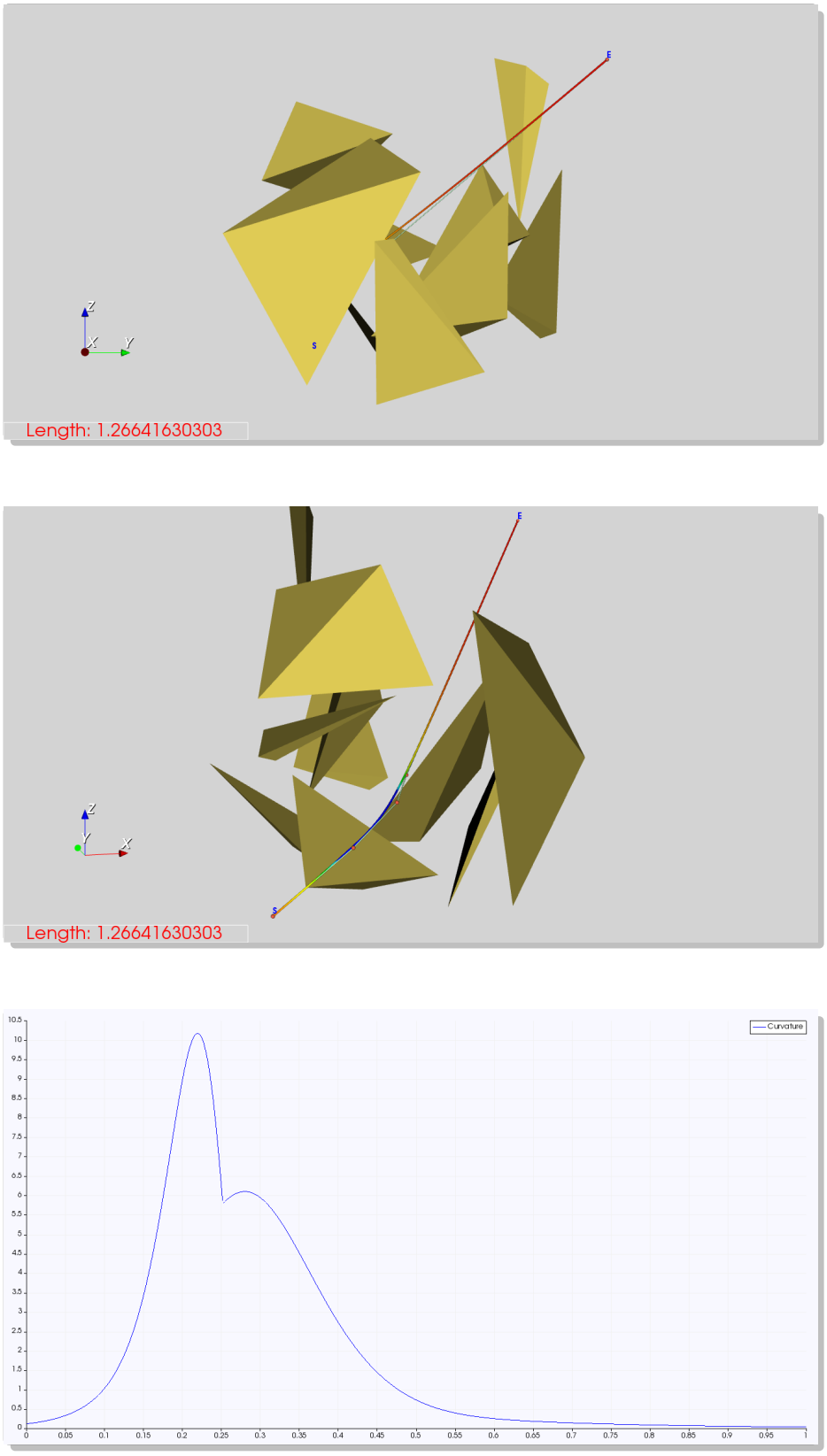


Figure 48.: Test 20; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. B; Post proc. ✓; Part. **A**.

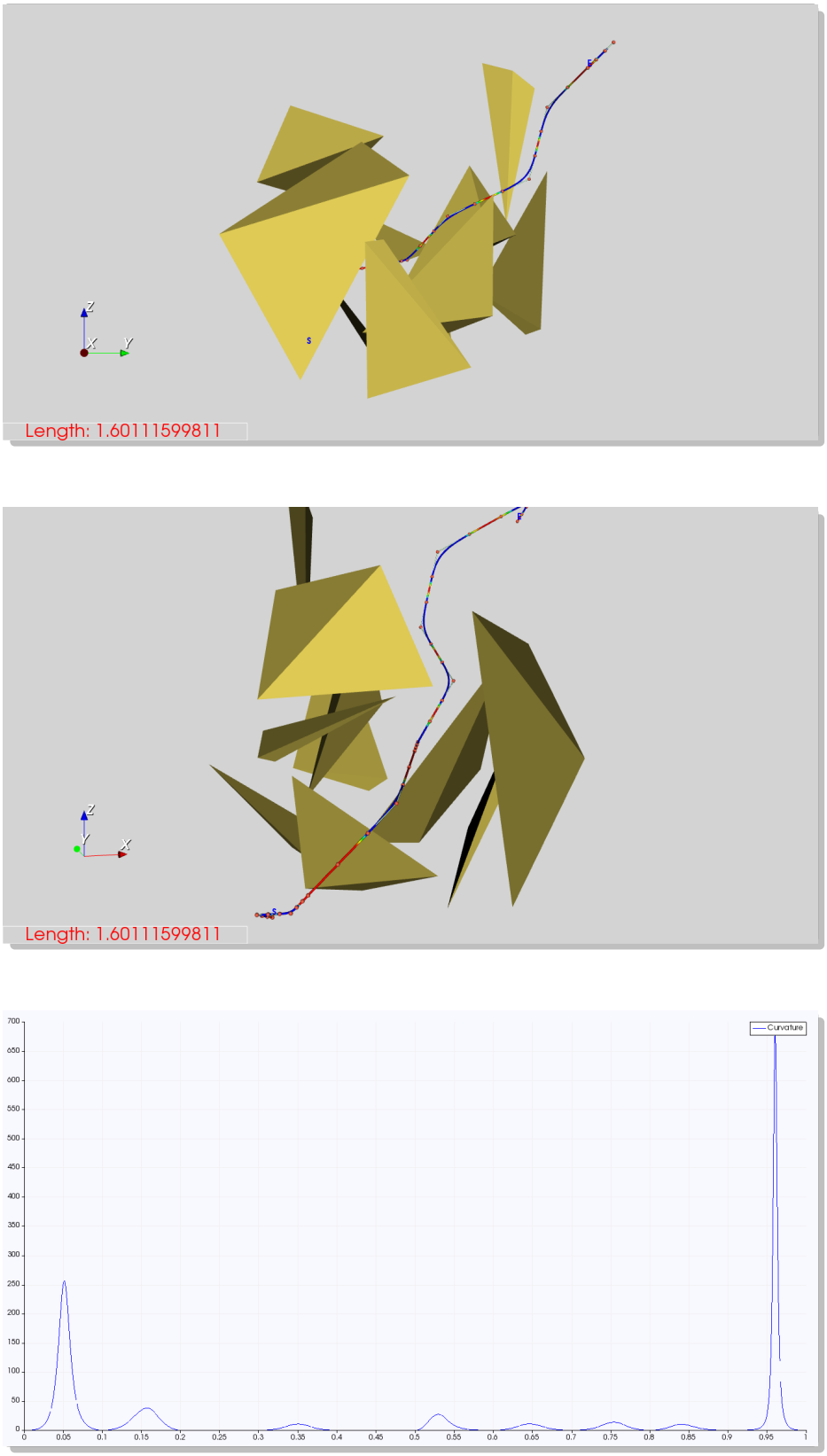


Figure 49.: Test 21; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. B; Post proc. ✗; Part. **U**.

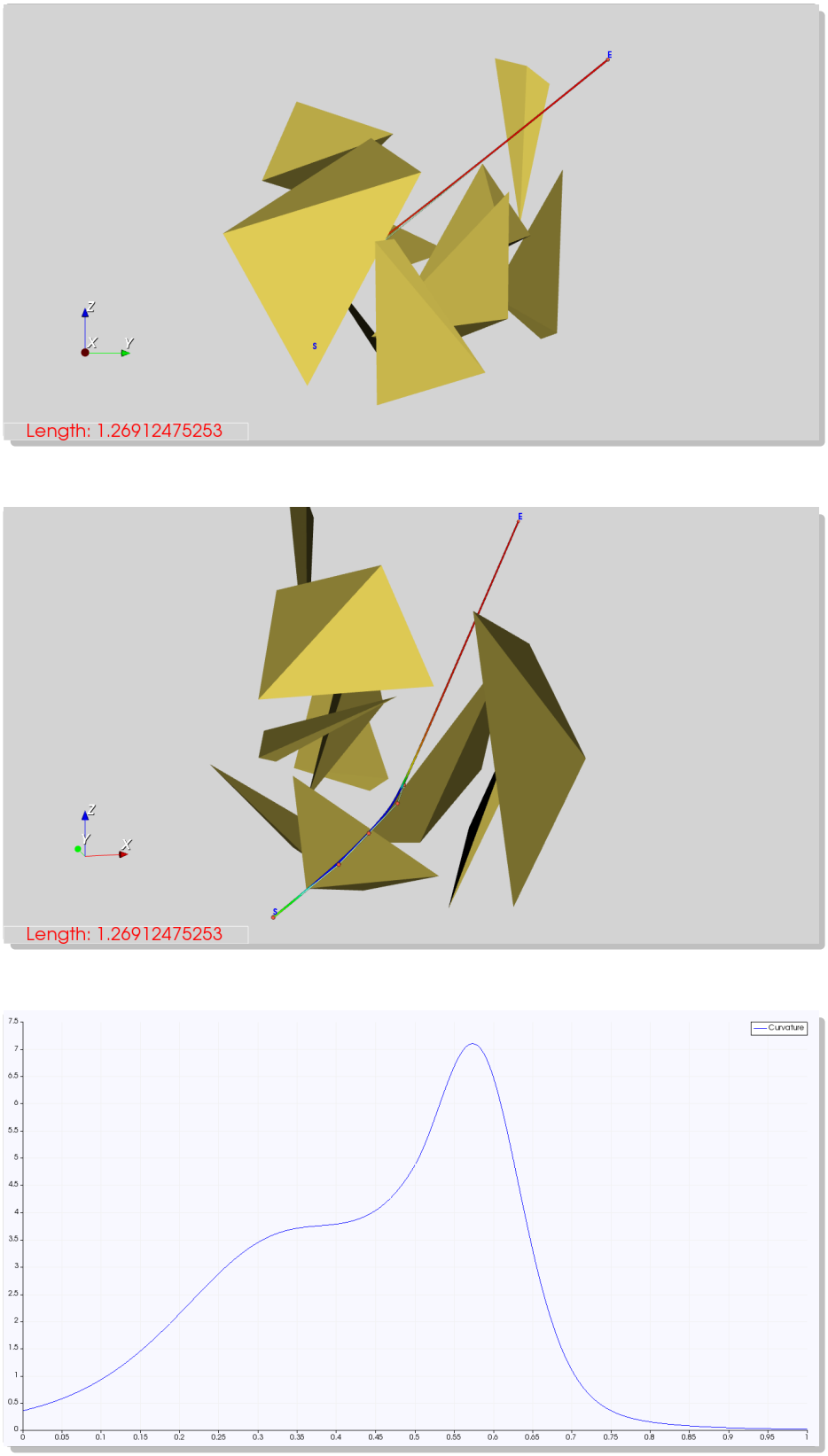


Figure 50.: Test 22; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. B; Post proc. ✓; Part. **U**.

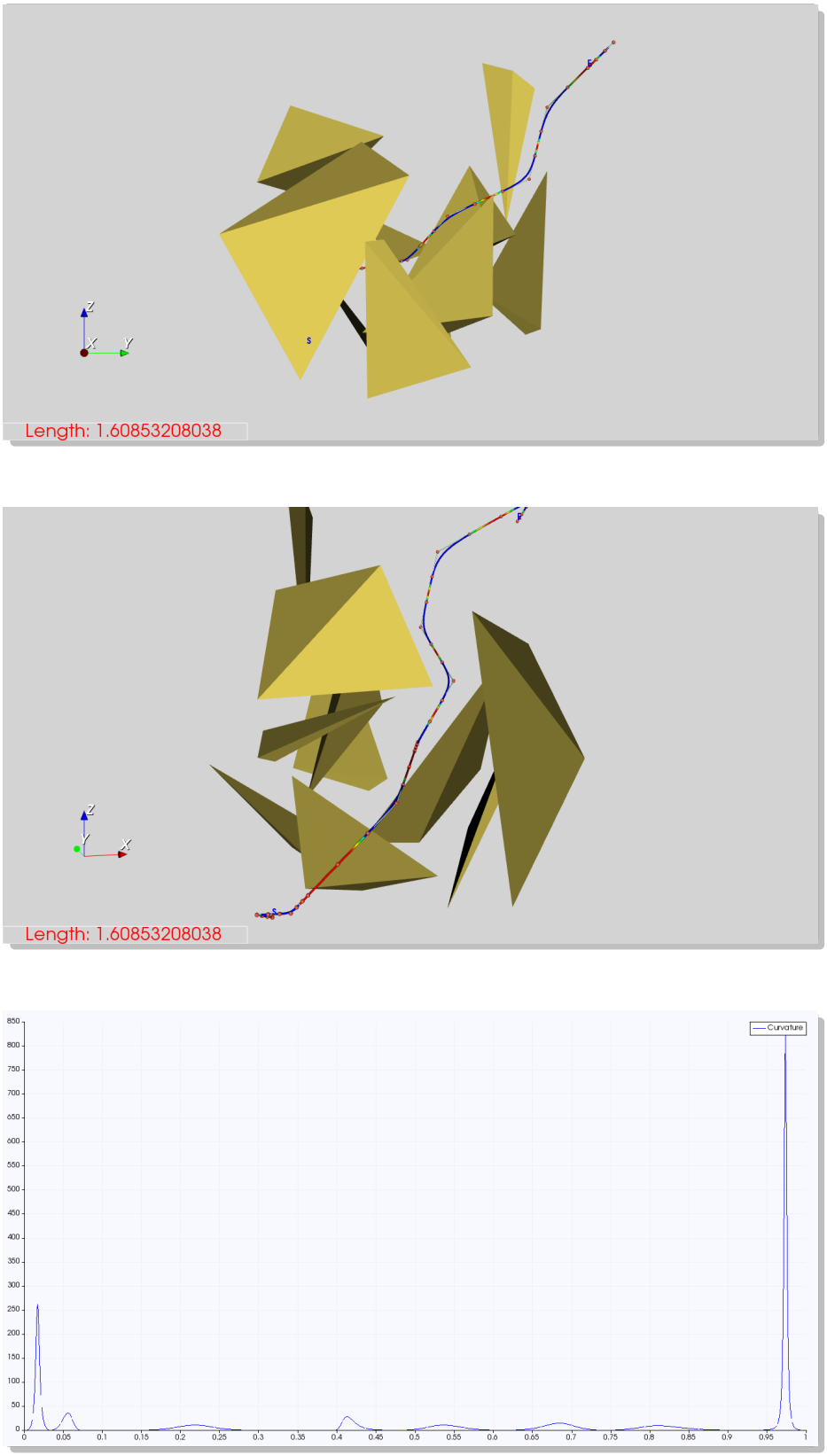


Figure 51.: Test 23; Scene 1; **s** → **e** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. B; Post proc. ✗; Part. **A**.

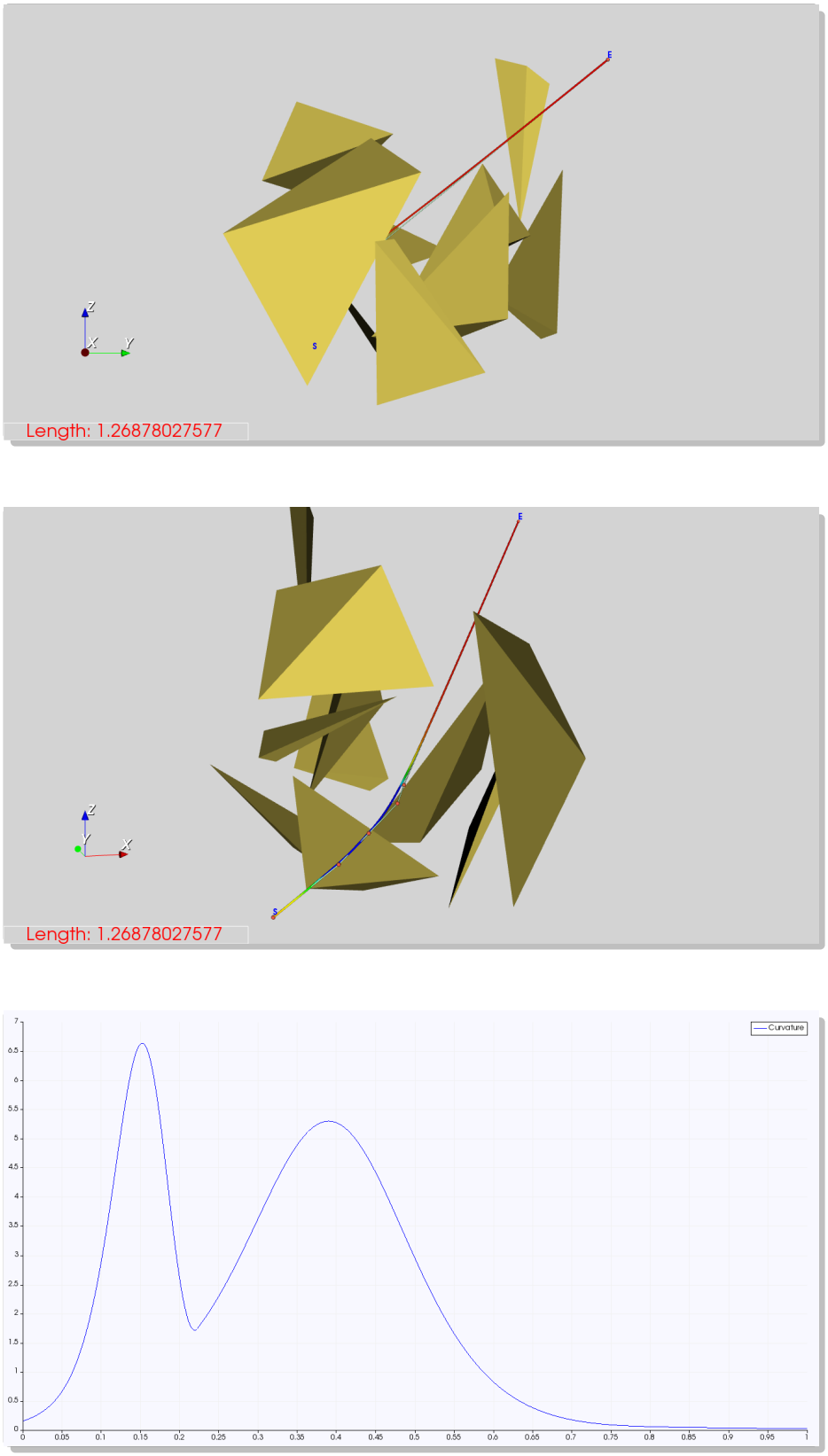


Figure 52.: Test 24; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. B; Post proc. ✓; Part. **A**.

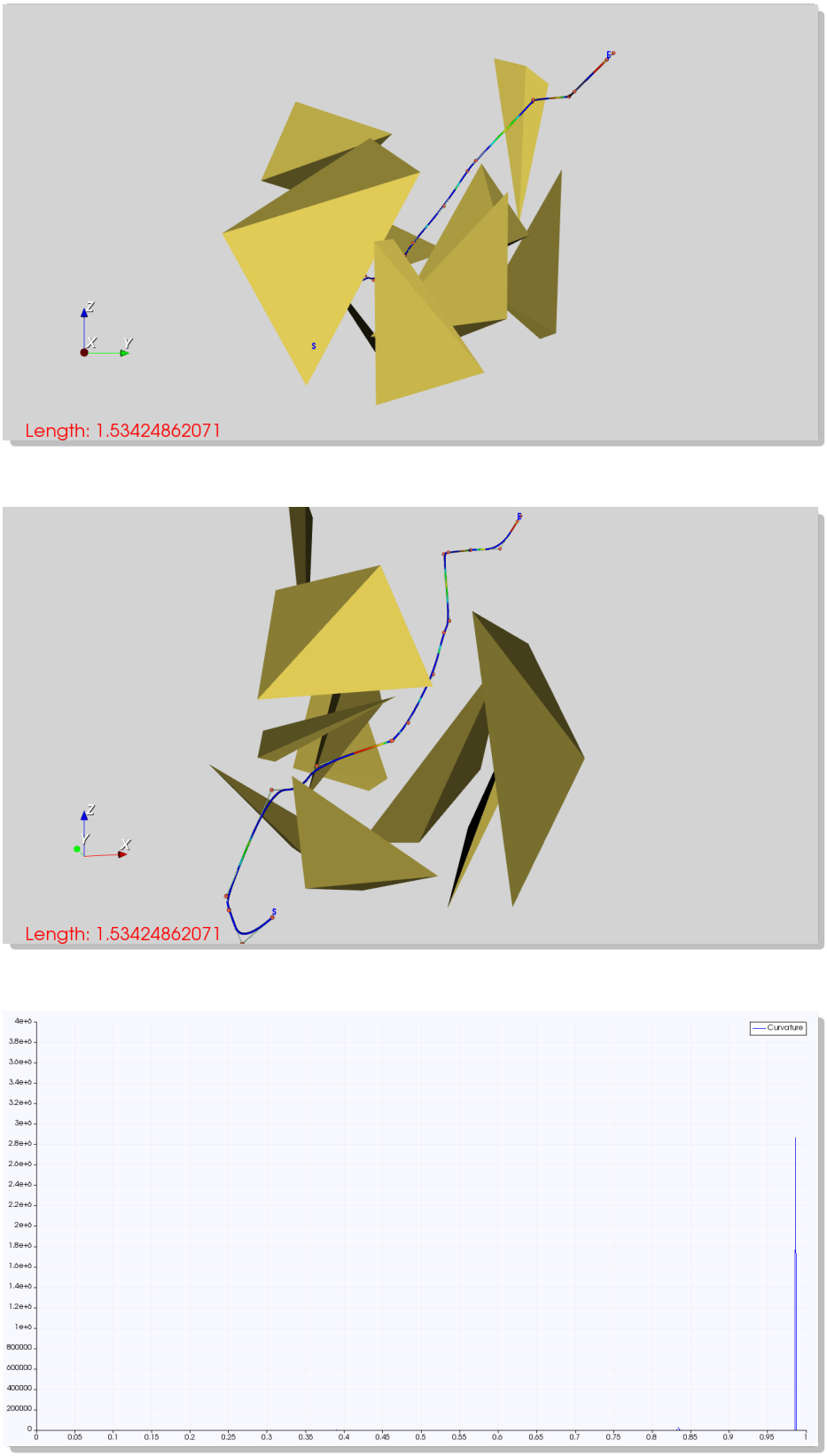


Figure 53.: Test 25; Scene 1b; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. B; Post proc. ✗; Part. **U**.

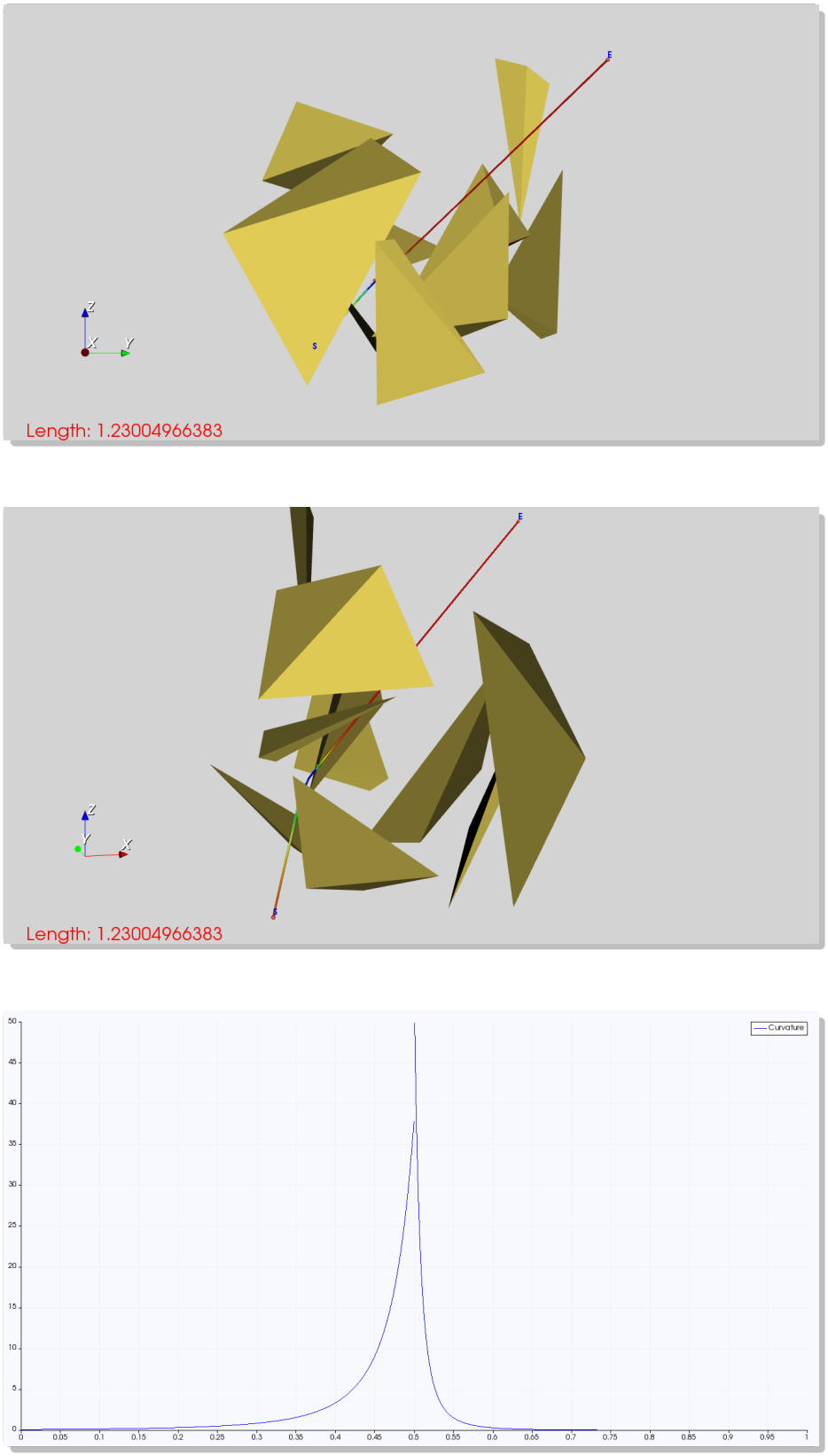


Figure 54.: Test 26; Scene 1b; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. B; Post proc. ✓; Part. **U**.

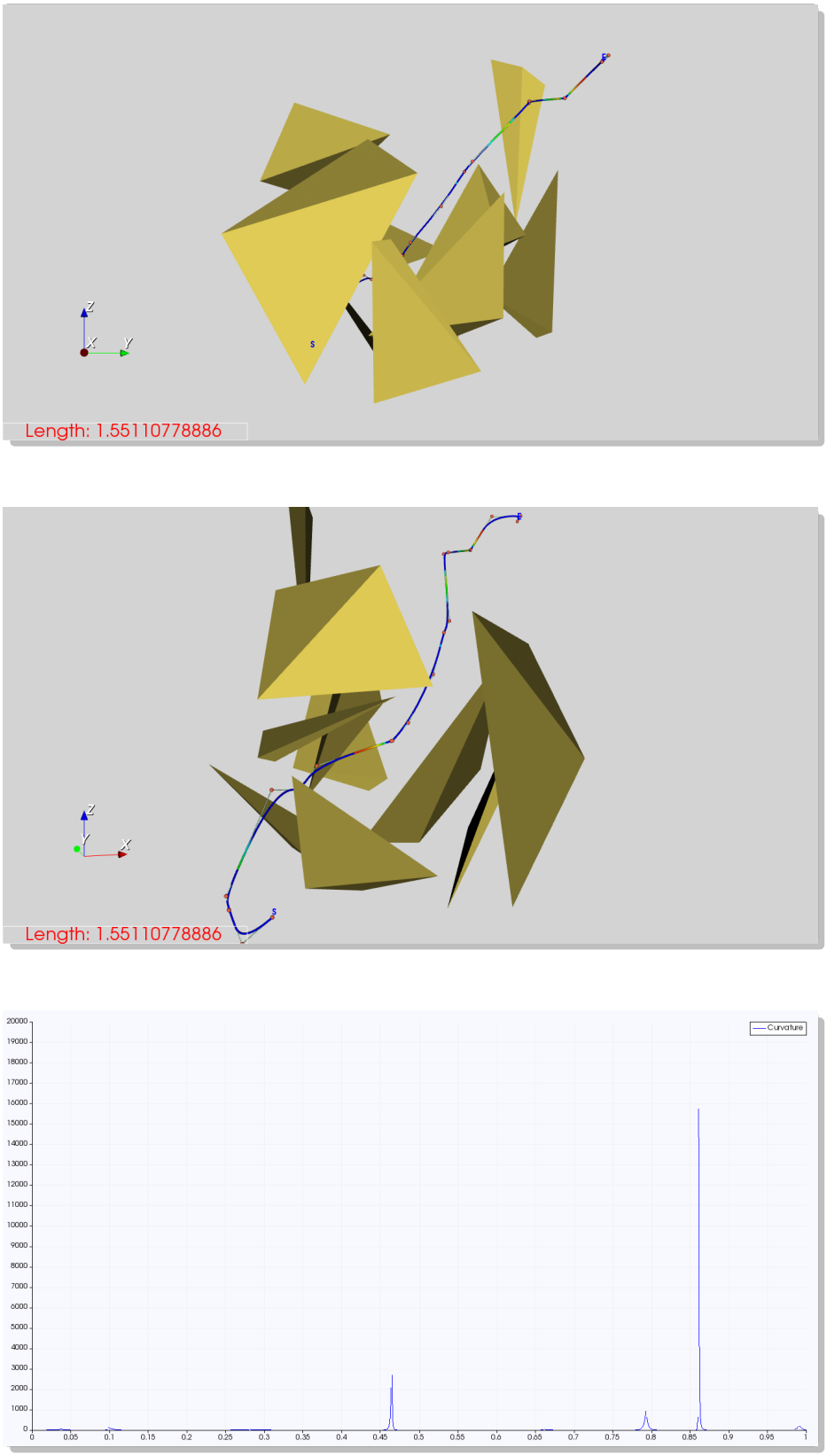


Figure 55.: Test 27; Scene 1b; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. B; Post proc. ✗; Part. **A**.

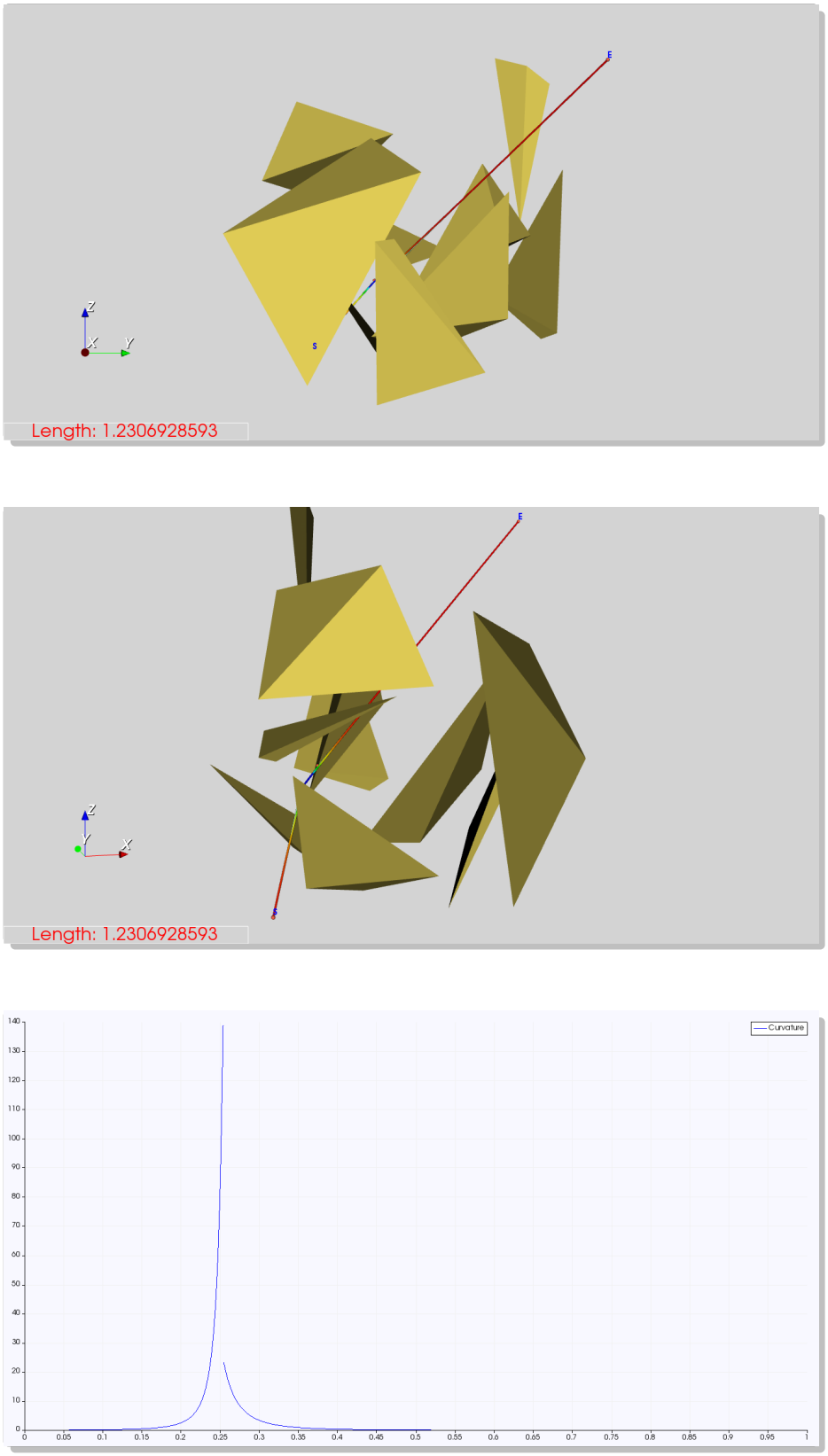
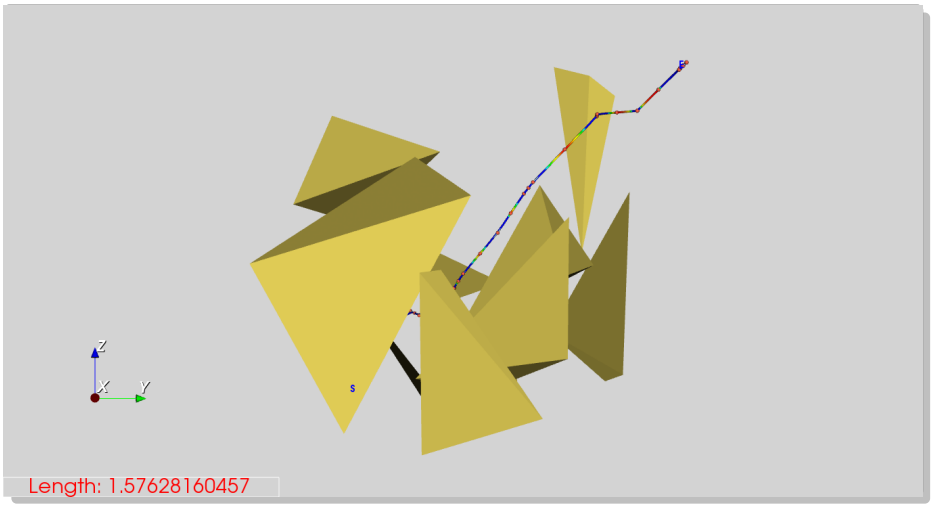
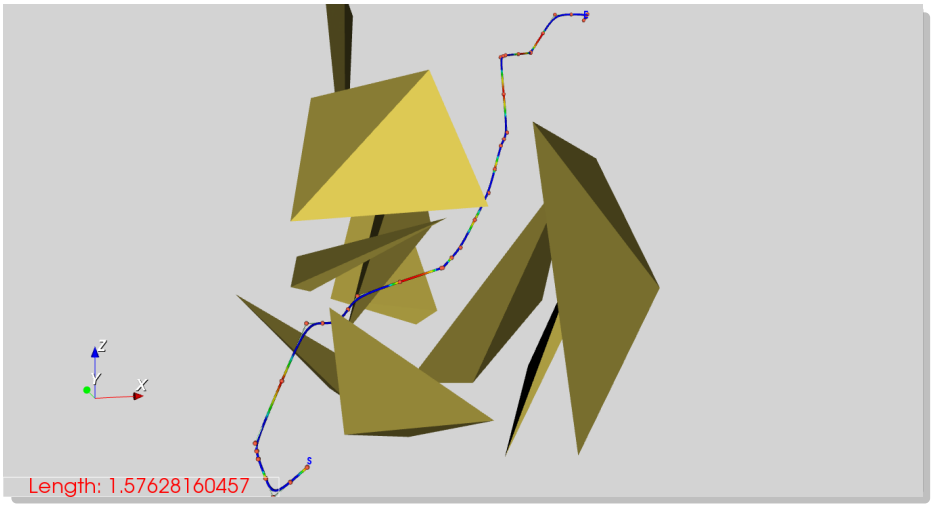


Figure 56.: Test 28; Scene 1b; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. B; Post proc. ✓; Part. **A**.





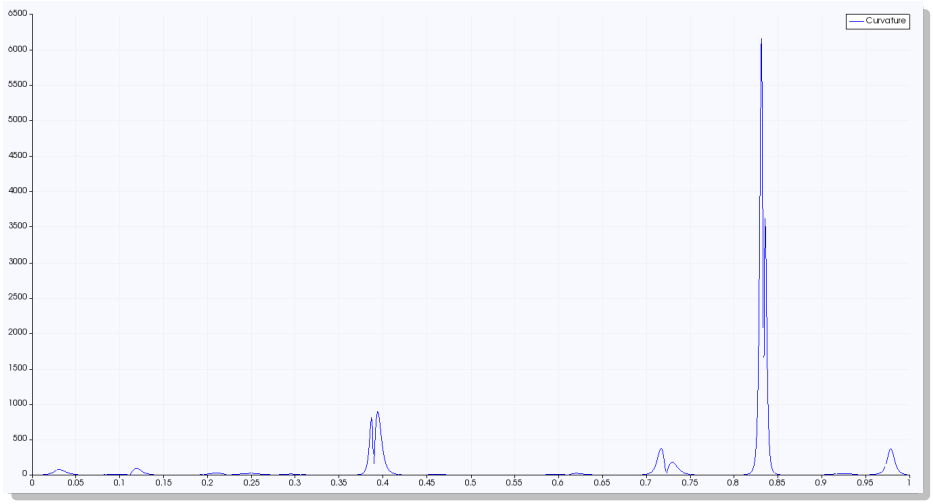


Figure 57.: Test 29; Scene 1b; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. B; Post proc. ✗; Part. **U**.

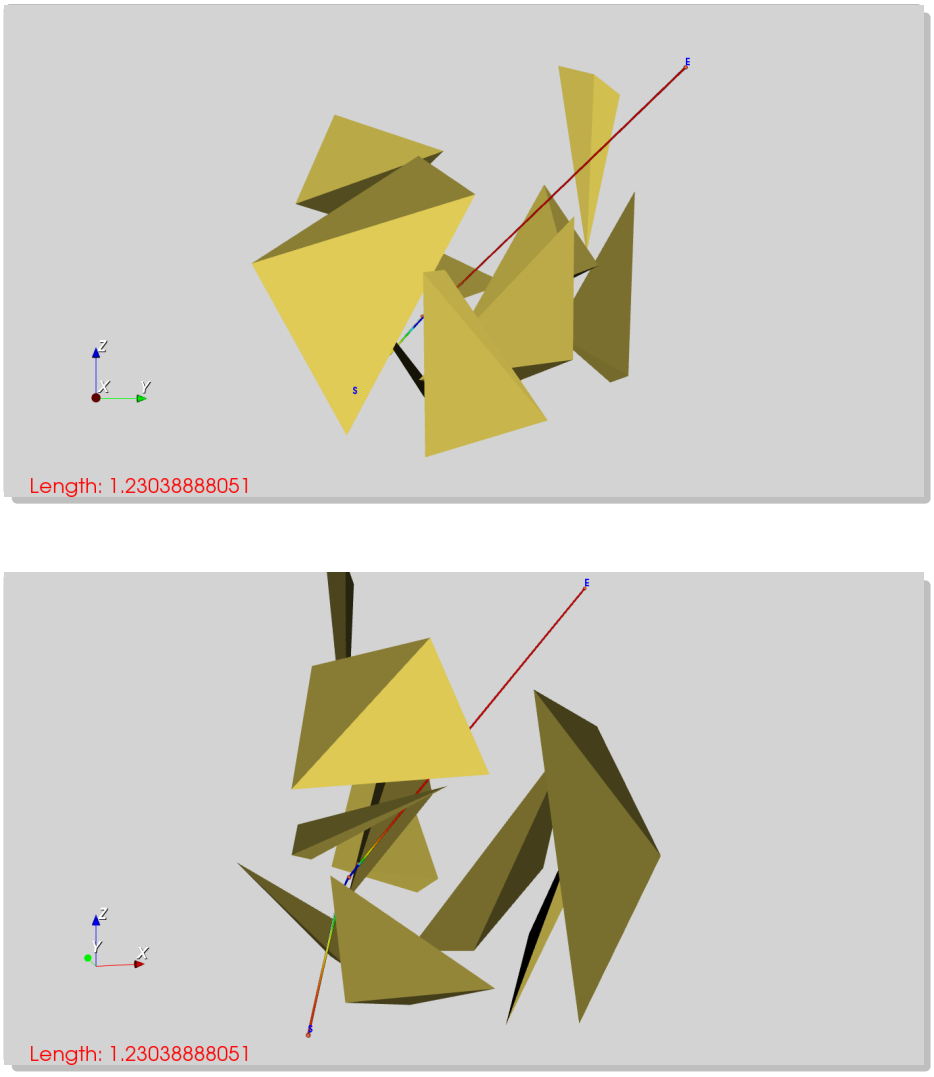
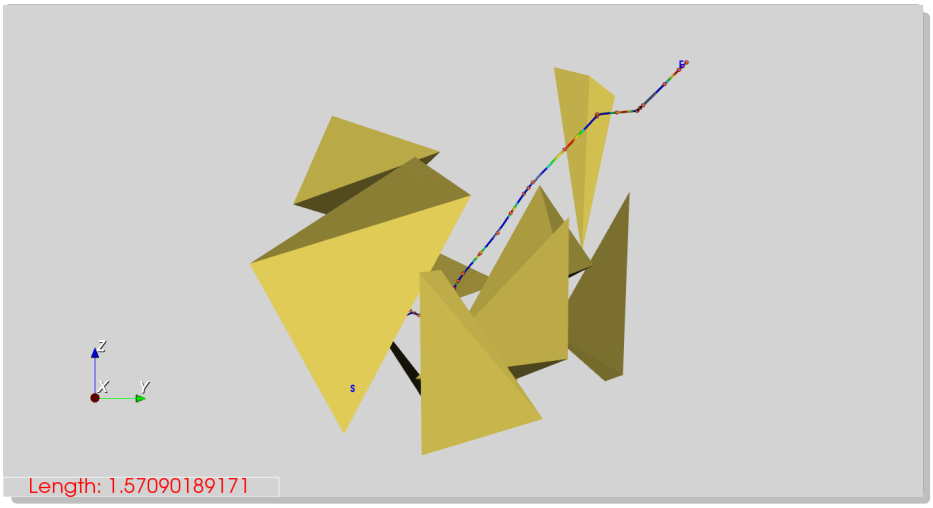
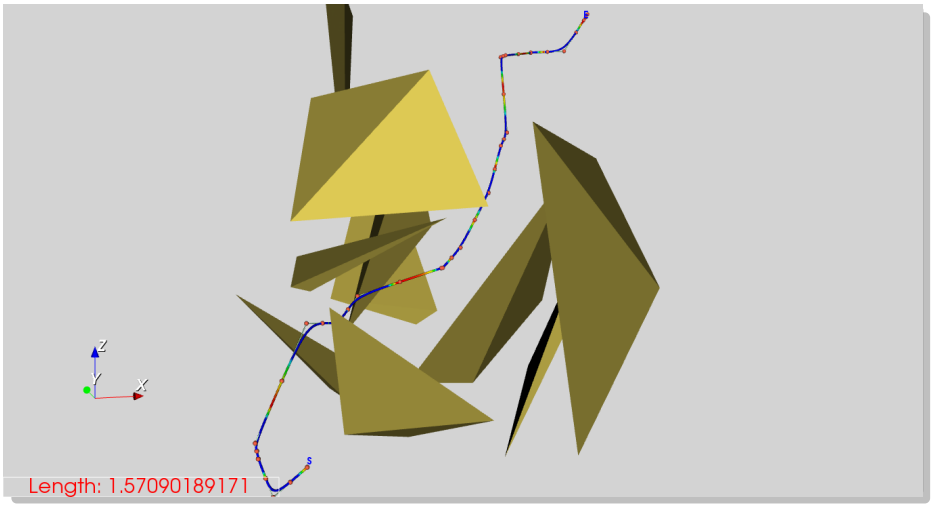


Figure 58.: Test 30; Scene 1b; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. B; Post proc. ✓; Part. **U**.





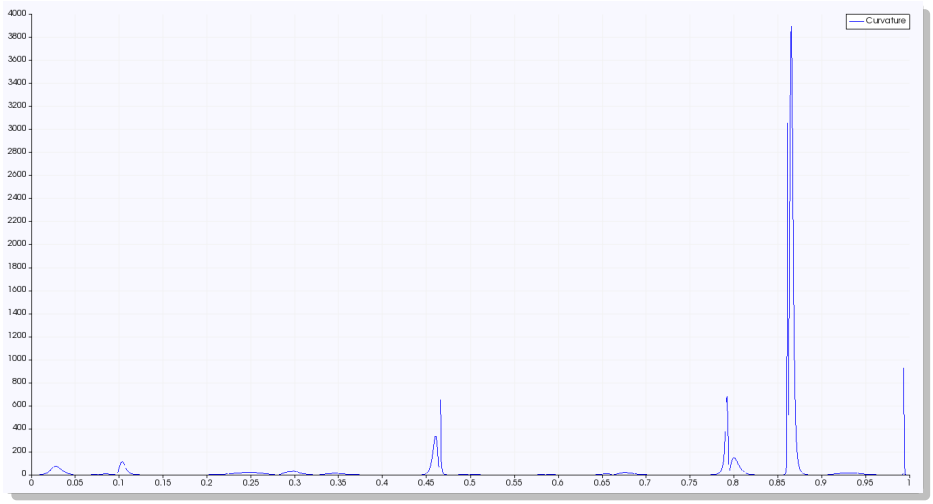


Figure 59.: Test 31; Scene 1b; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. B; Post proc. ✗; Part. **A**.

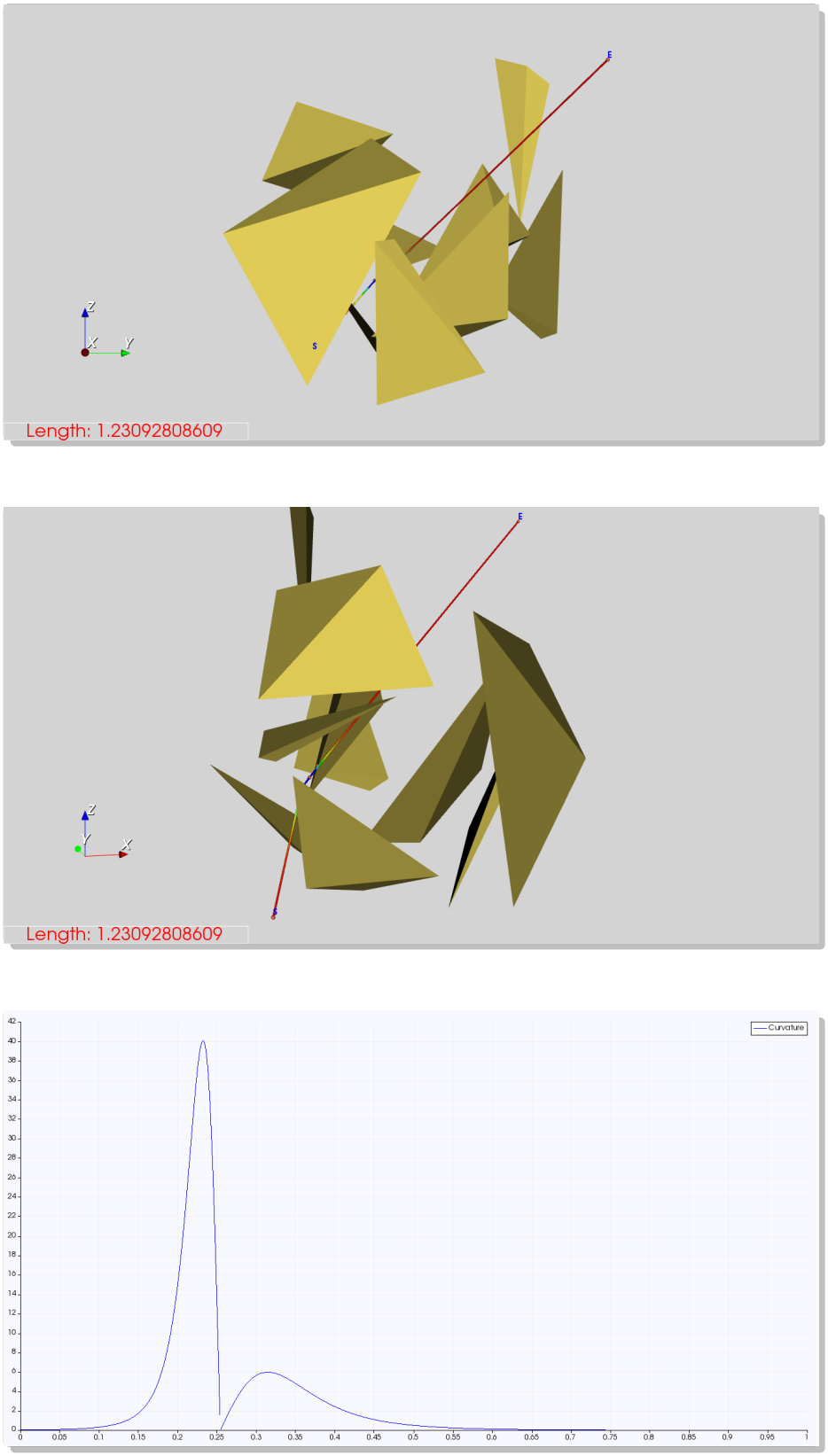
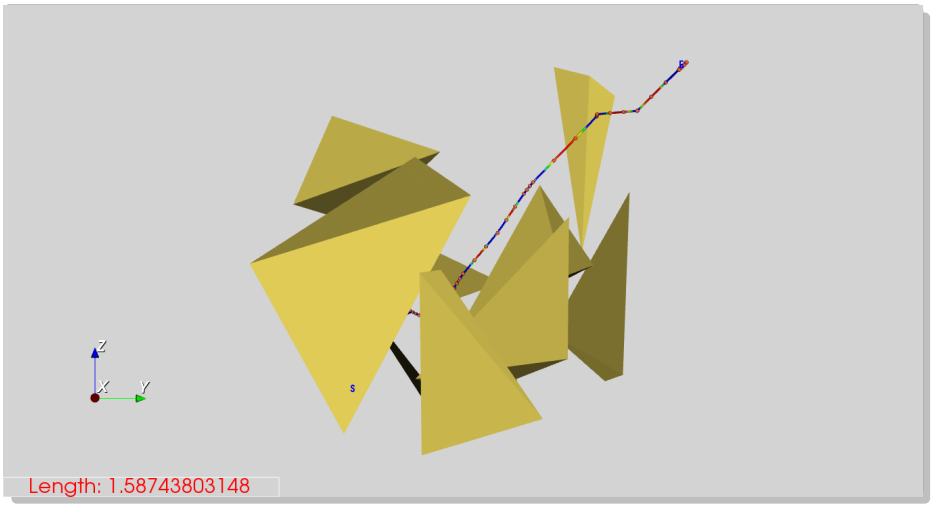
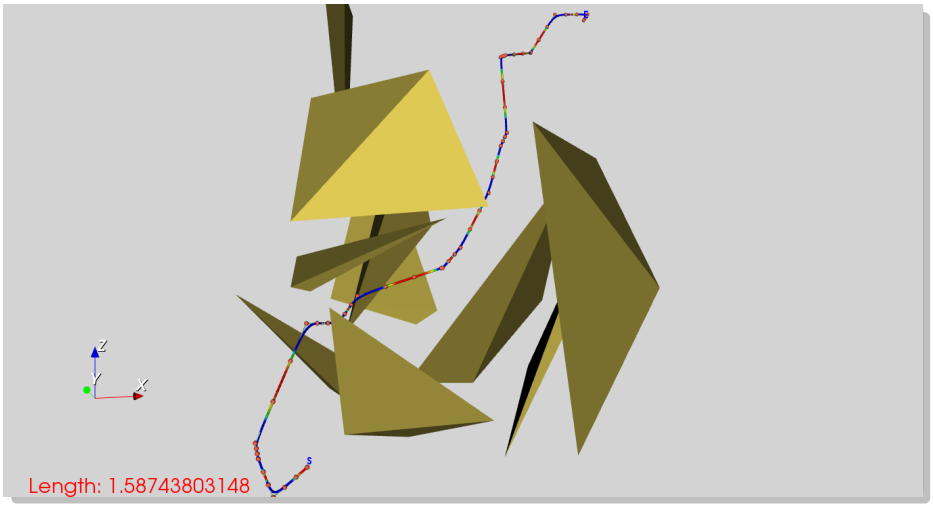


Figure 60.: Test 32; Scene 1b; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. B; Post proc. ✓; Part. **A**.





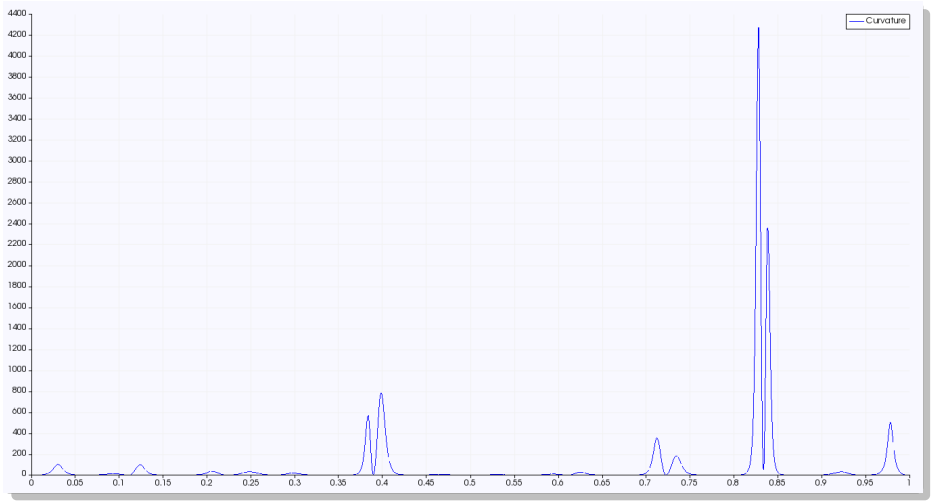


Figure 61.: Test 33; Scene 1b; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. B; Post proc. ✗; Part. **U**.

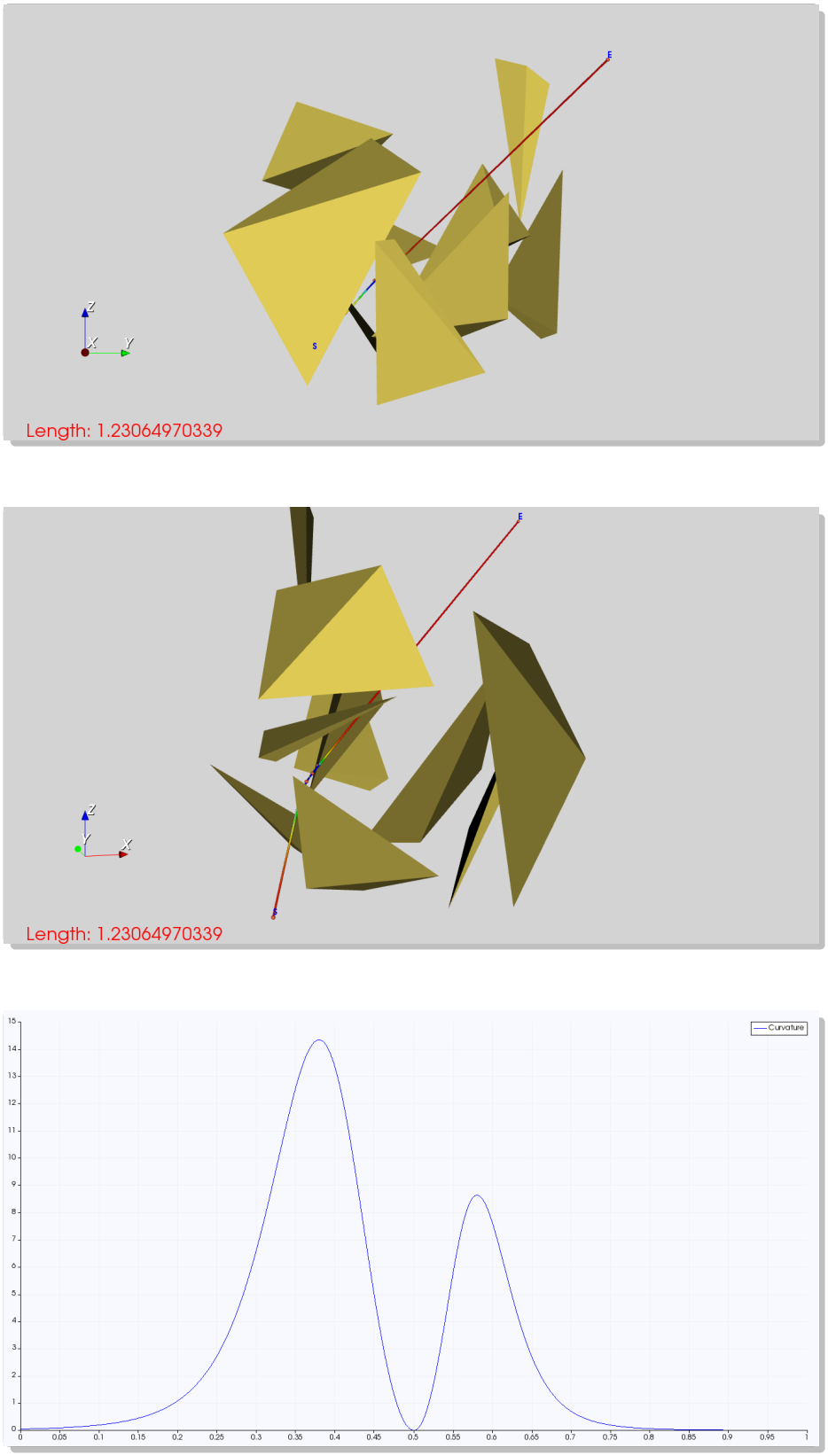
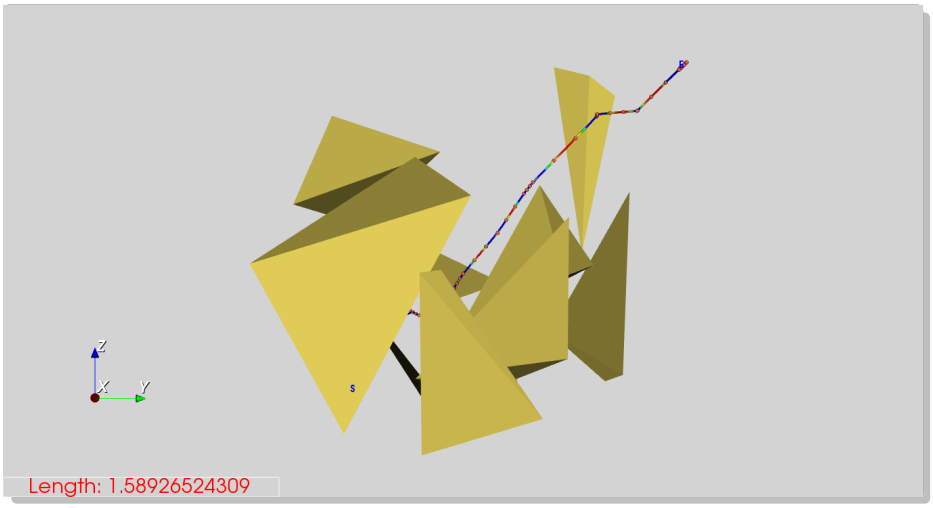
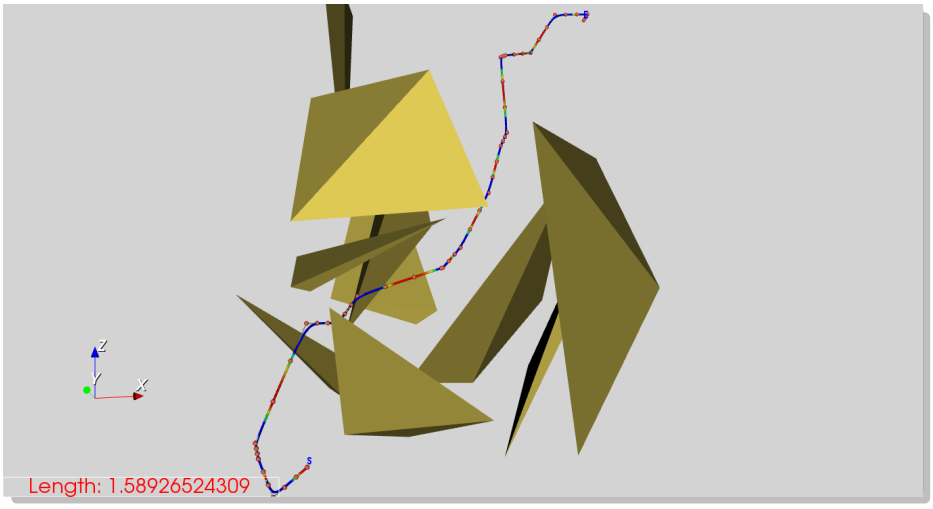


Figure 62.: Test 34; Scene 1b; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. B; Post proc. ✓; Part. **U**.





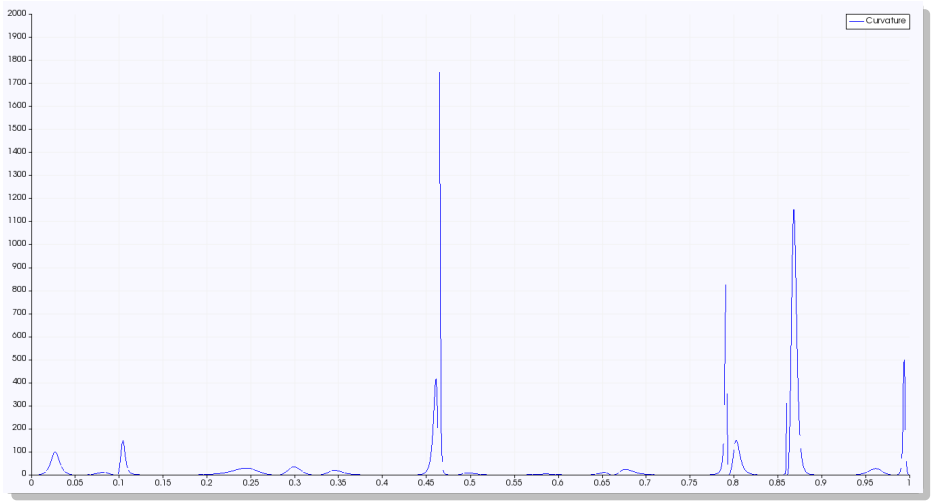


Figure 63.: Test 35; Scene 1b; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. B; Post proc. ✗; Part. **A**.

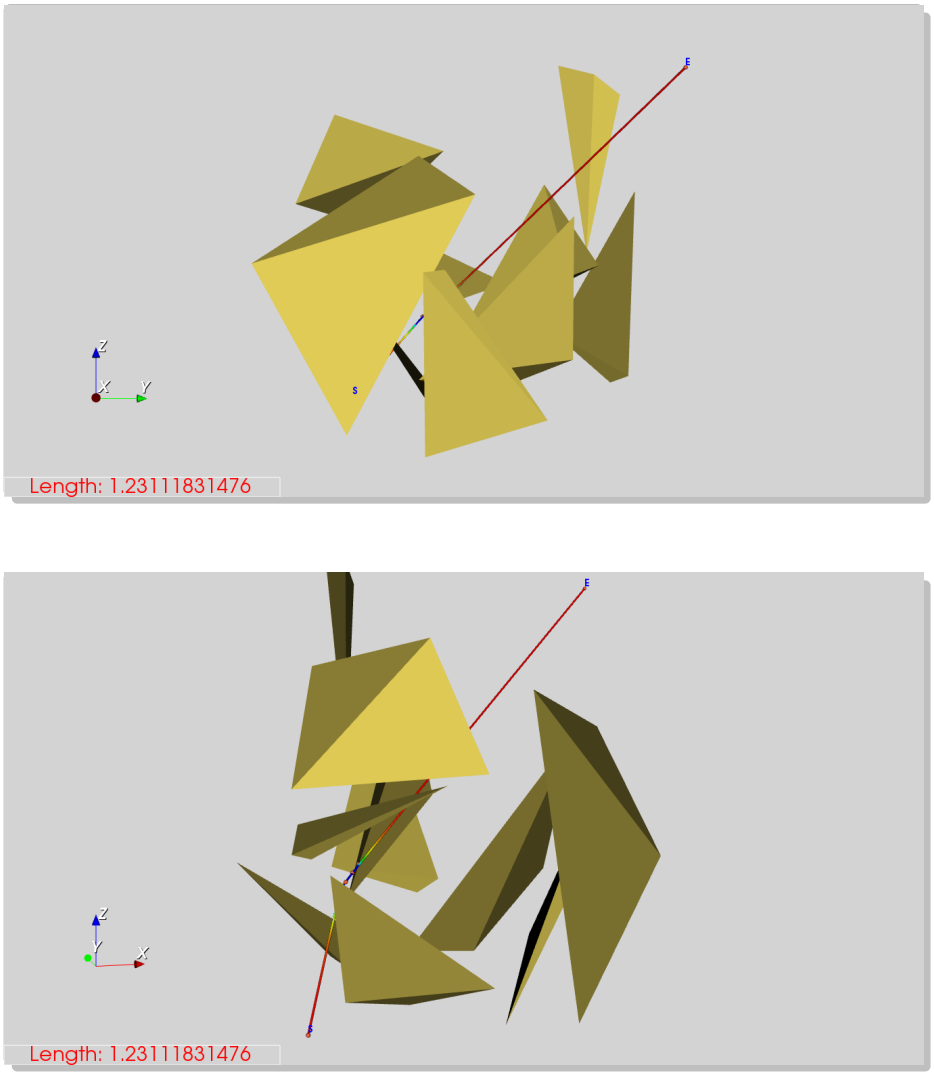
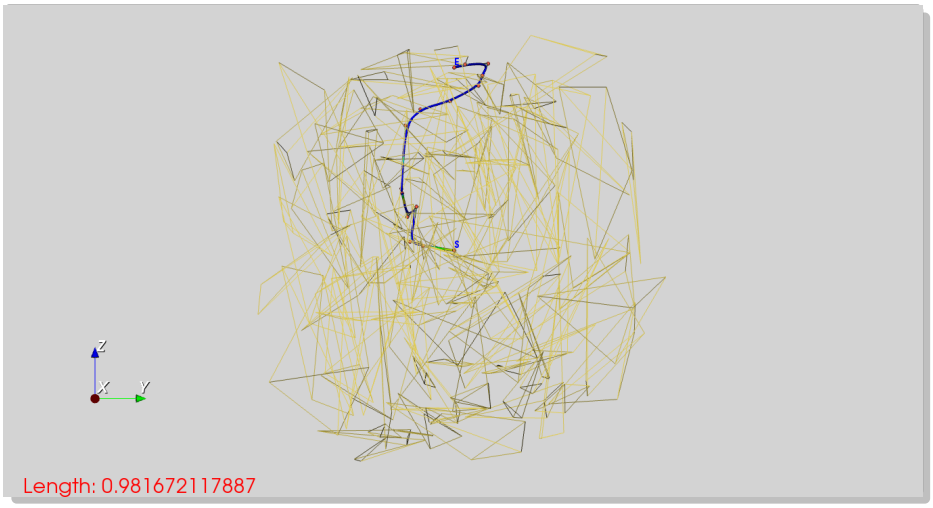
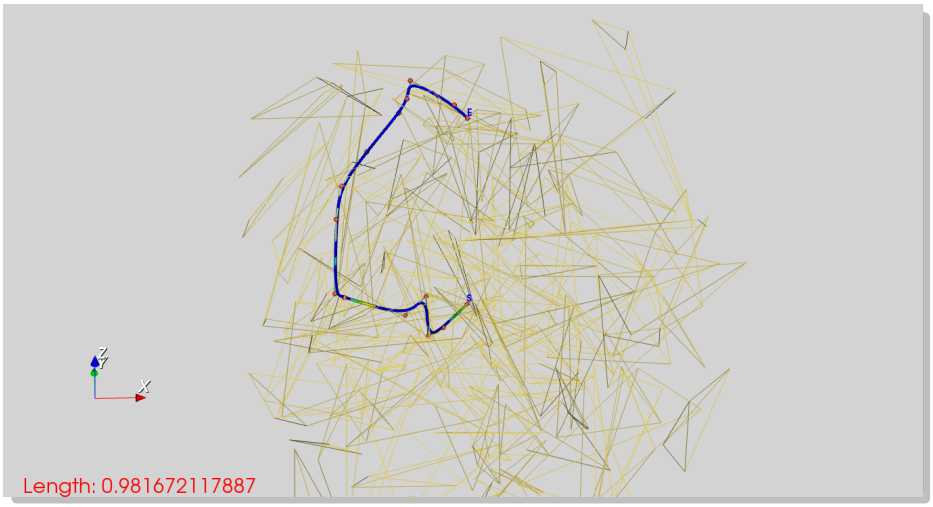


Figure 64.: Test 36; Scene 1b; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. B; Post proc. ✓; Part. **A**.





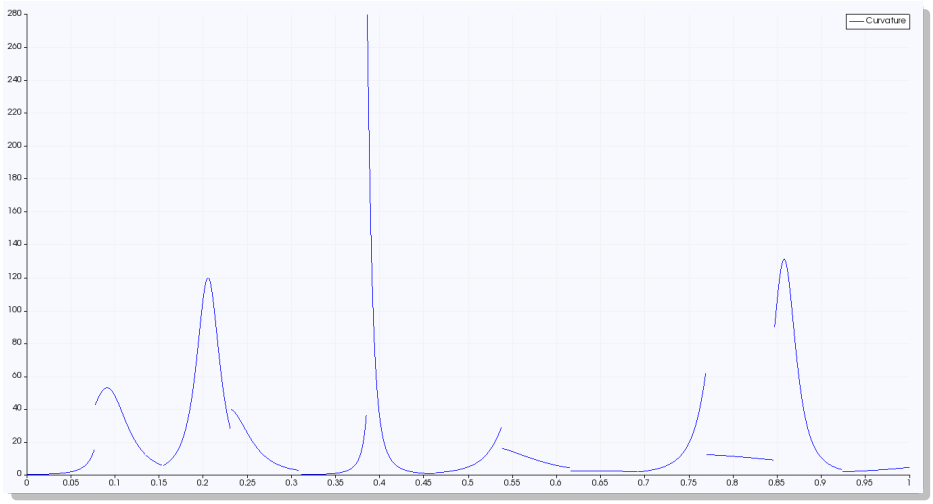


Figure 65.: Test 37; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 2; Meth. B; Post proc. ✗; Part. **U**.

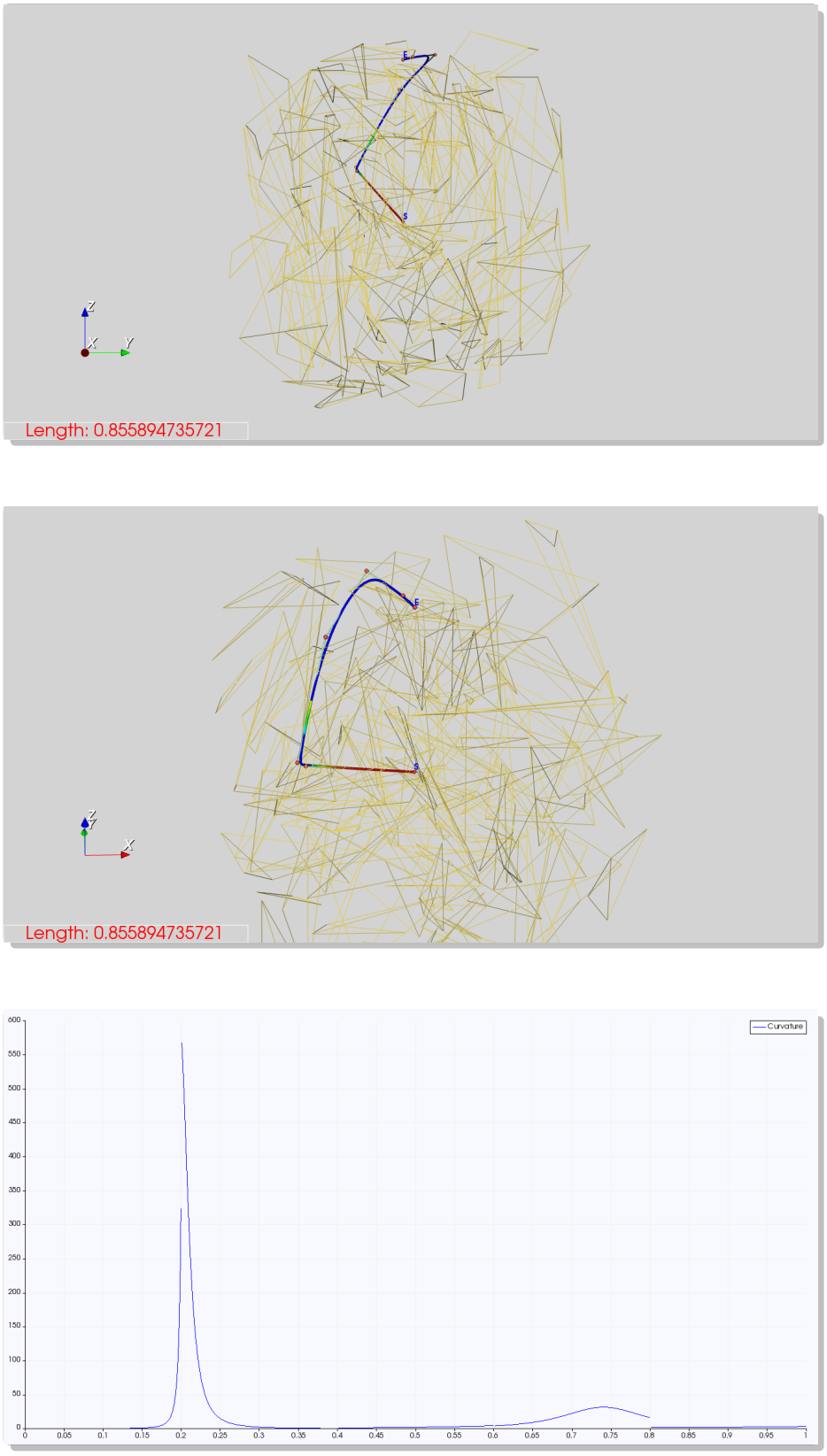


Figure 66.: Test 38; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 2; Meth. B; Post proc. ✓; Part. **U**.

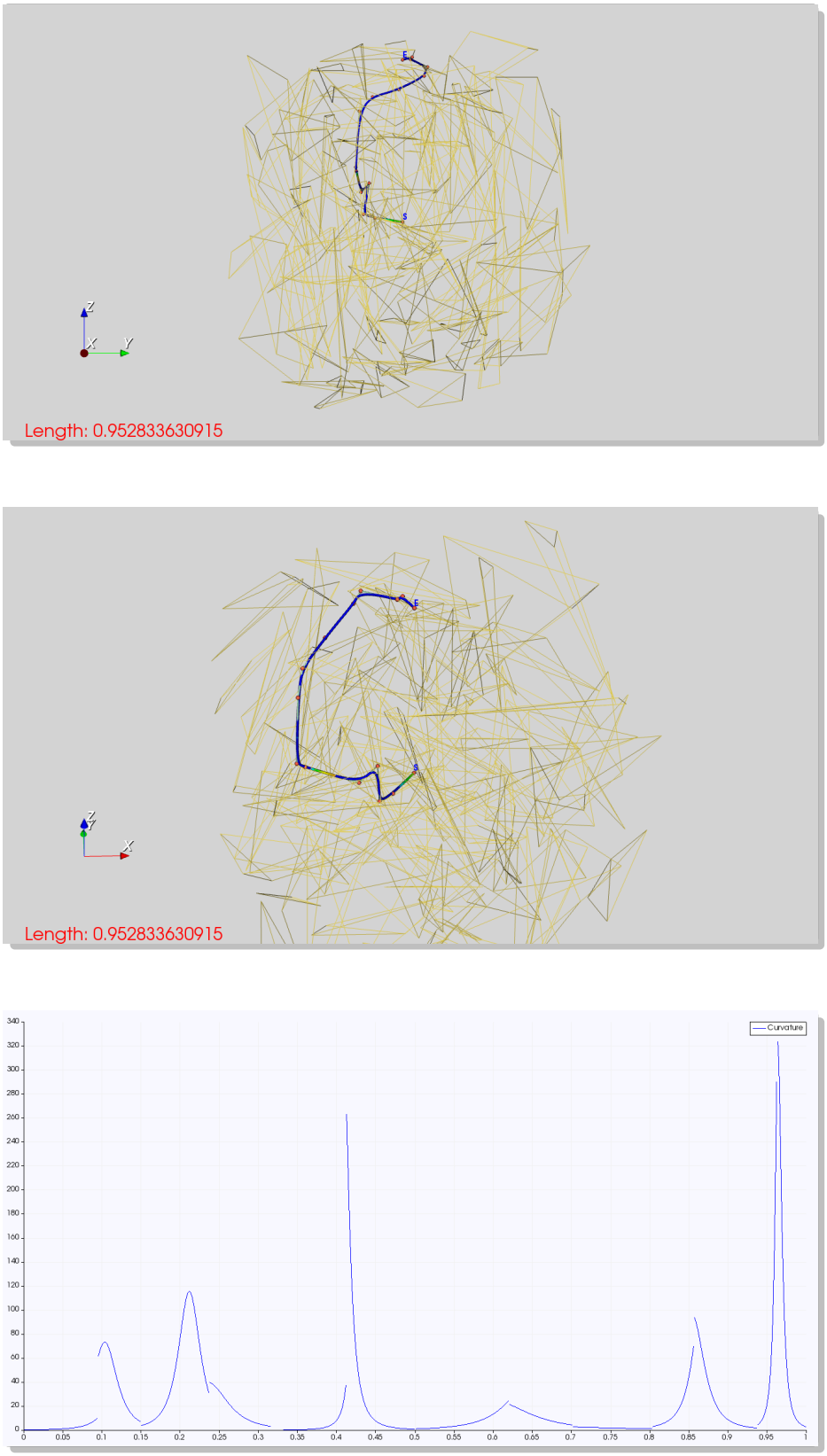


Figure 67.: Test 39; Scene 2; $\mathbf{s} \rightarrow \boldsymbol{e}$ [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 2; Meth. B; Post proc. ✗; Part. **A**.

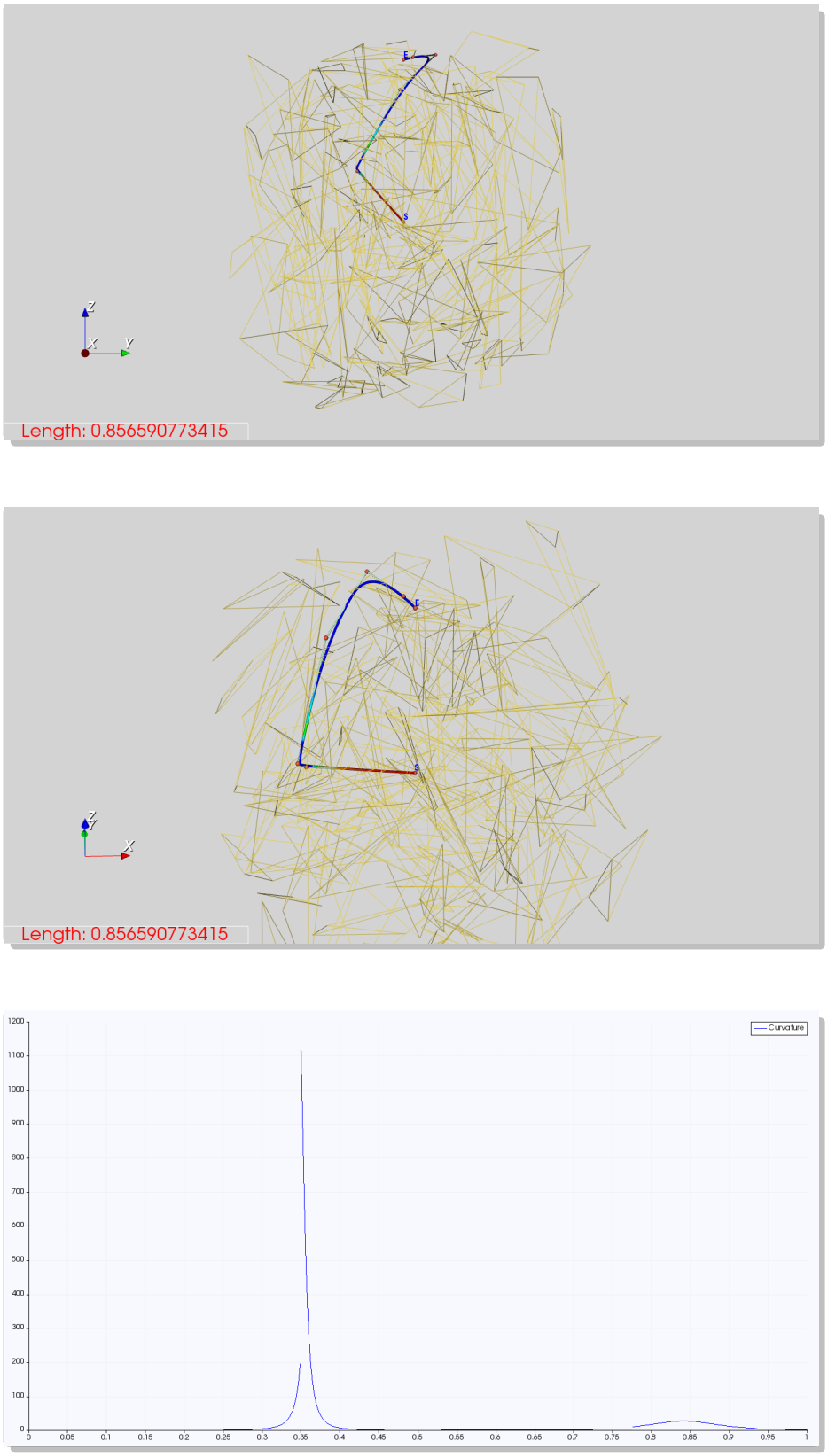
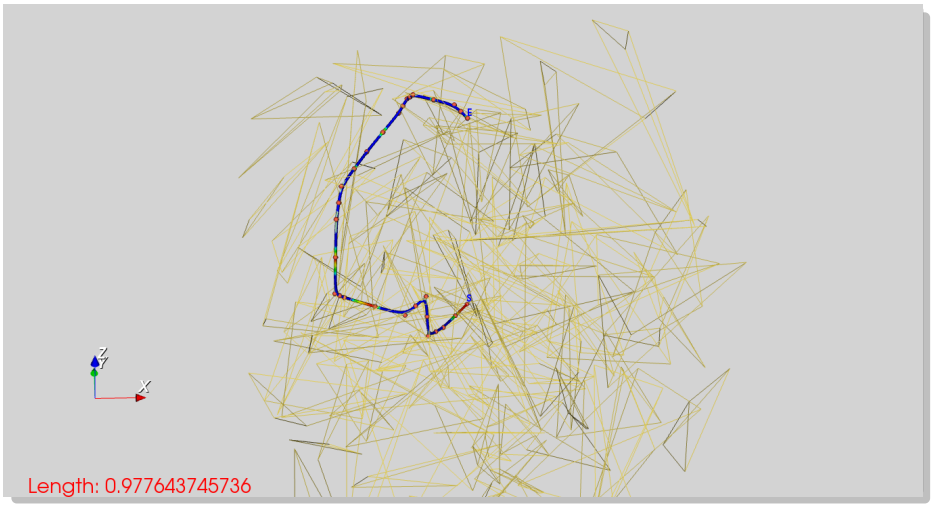


Figure 68.: Test 40; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 2; Meth. B; Post proc. ✓; Part. **A**.



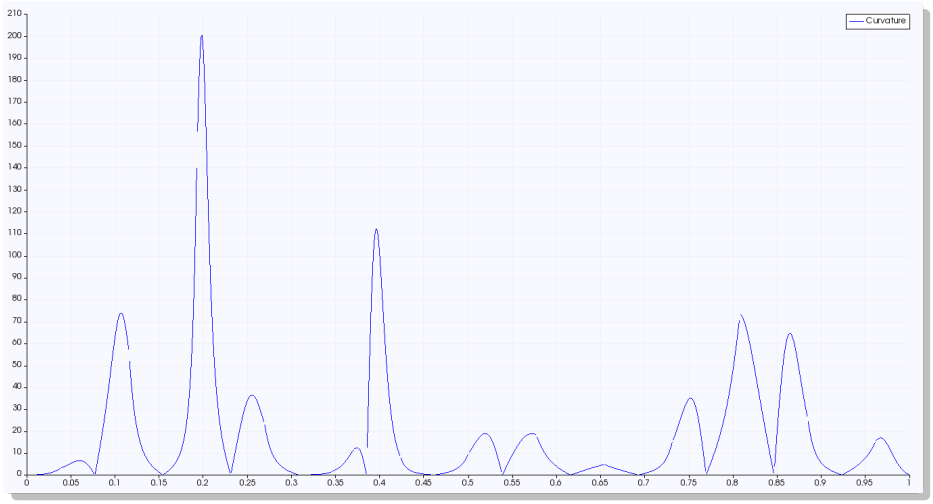


Figure 69.: Test 41; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 3; Meth. B; Post proc. ✗; Part. **U**.

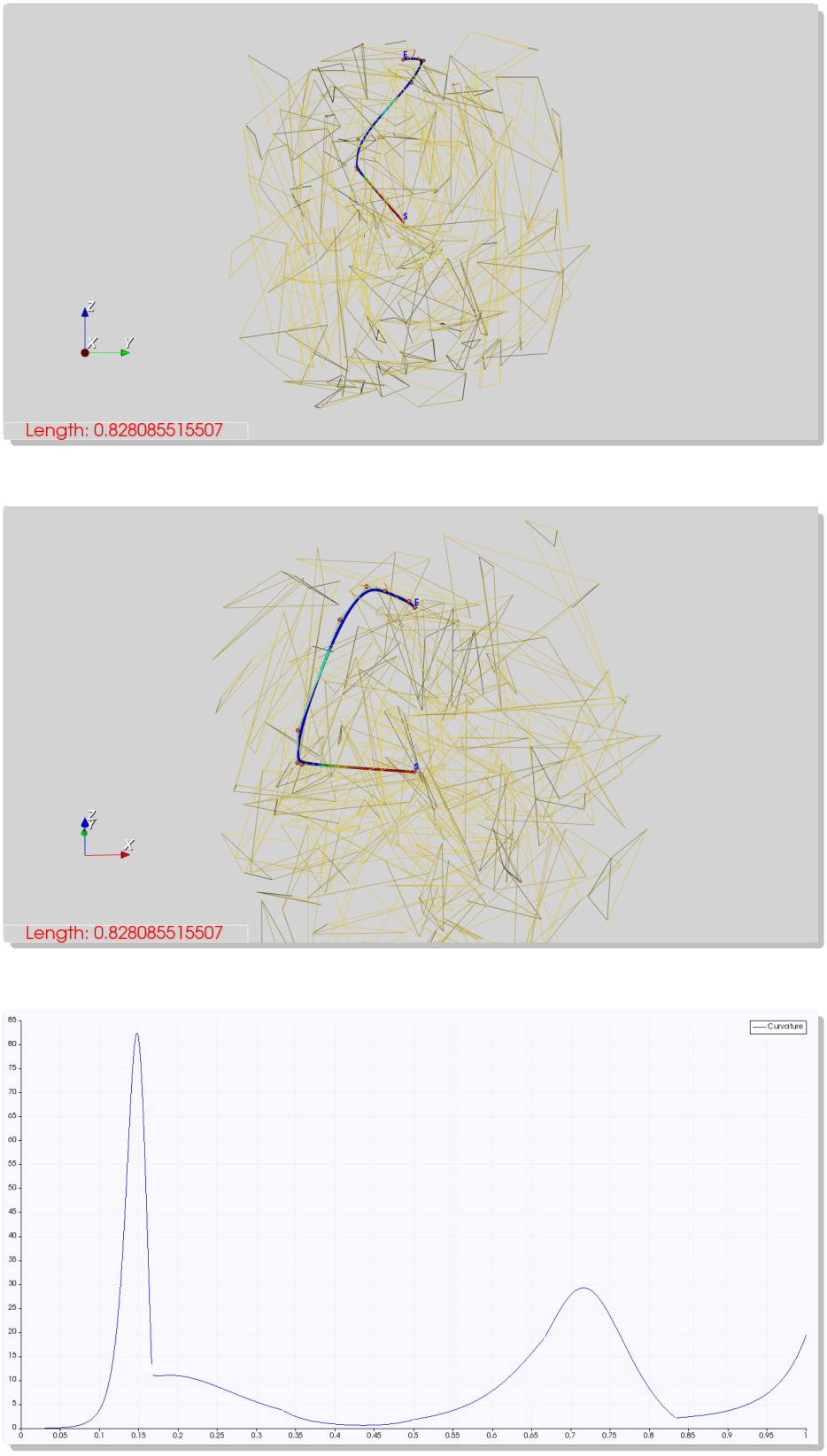


Figure 70.: Test 42; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 3; Meth. B; Post proc. ✓; Part. **U**.

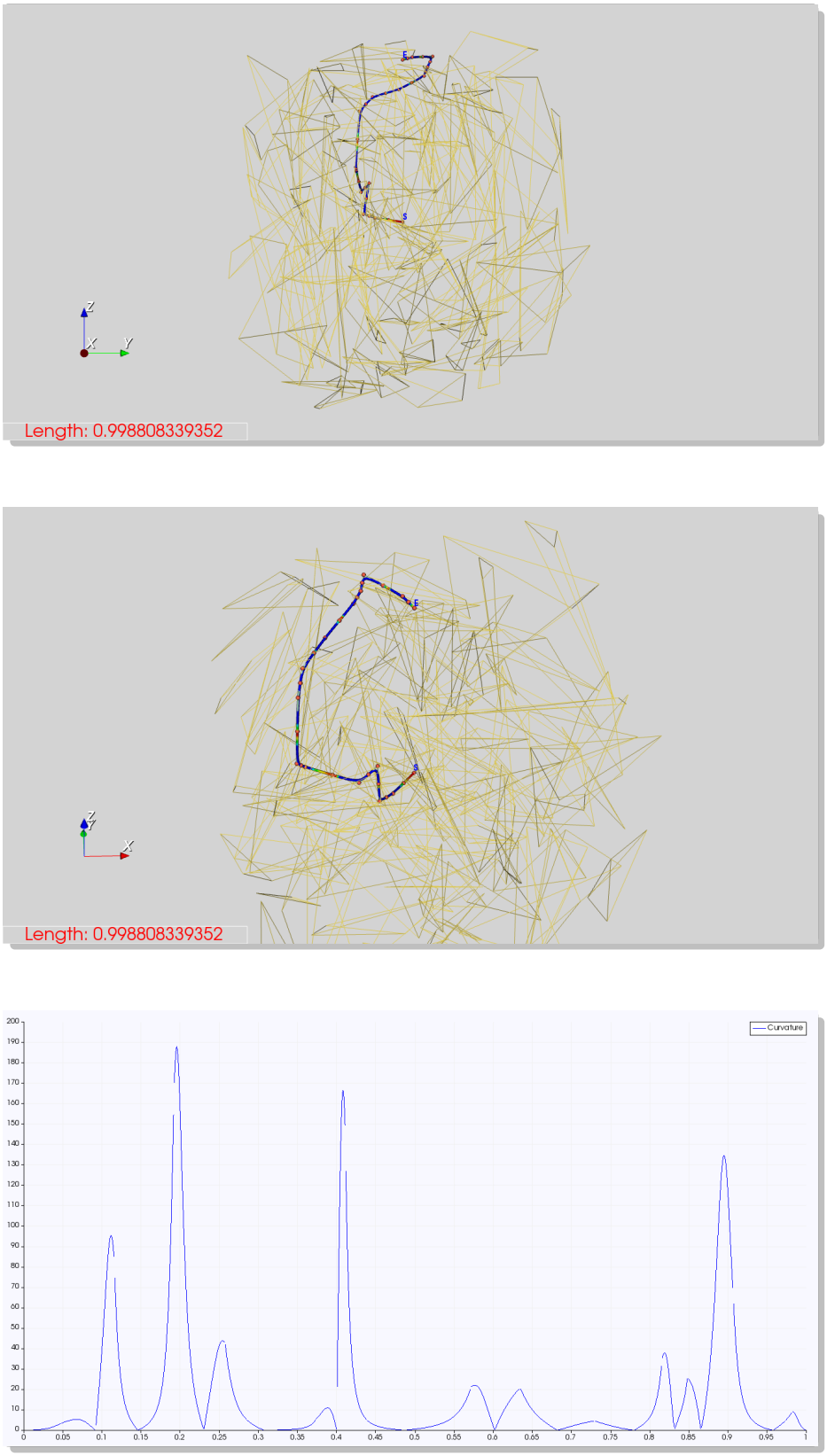


Figure 71.: Test 43; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 3; Meth. B; Post proc. ✗; Part. **A**.

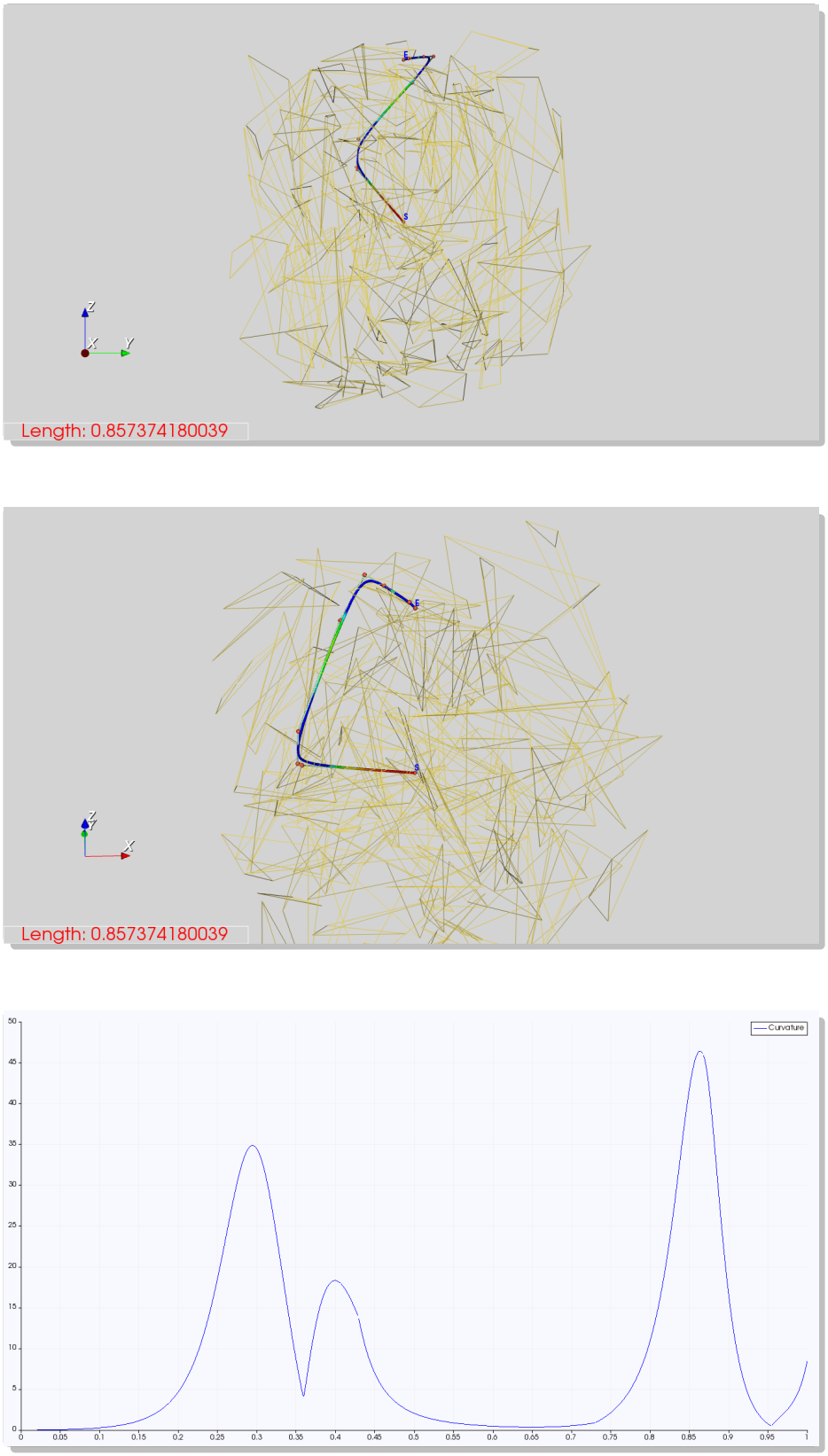


Figure 72.: Test 44; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 3; Meth. B; Post proc. ✓; Part. **A**.

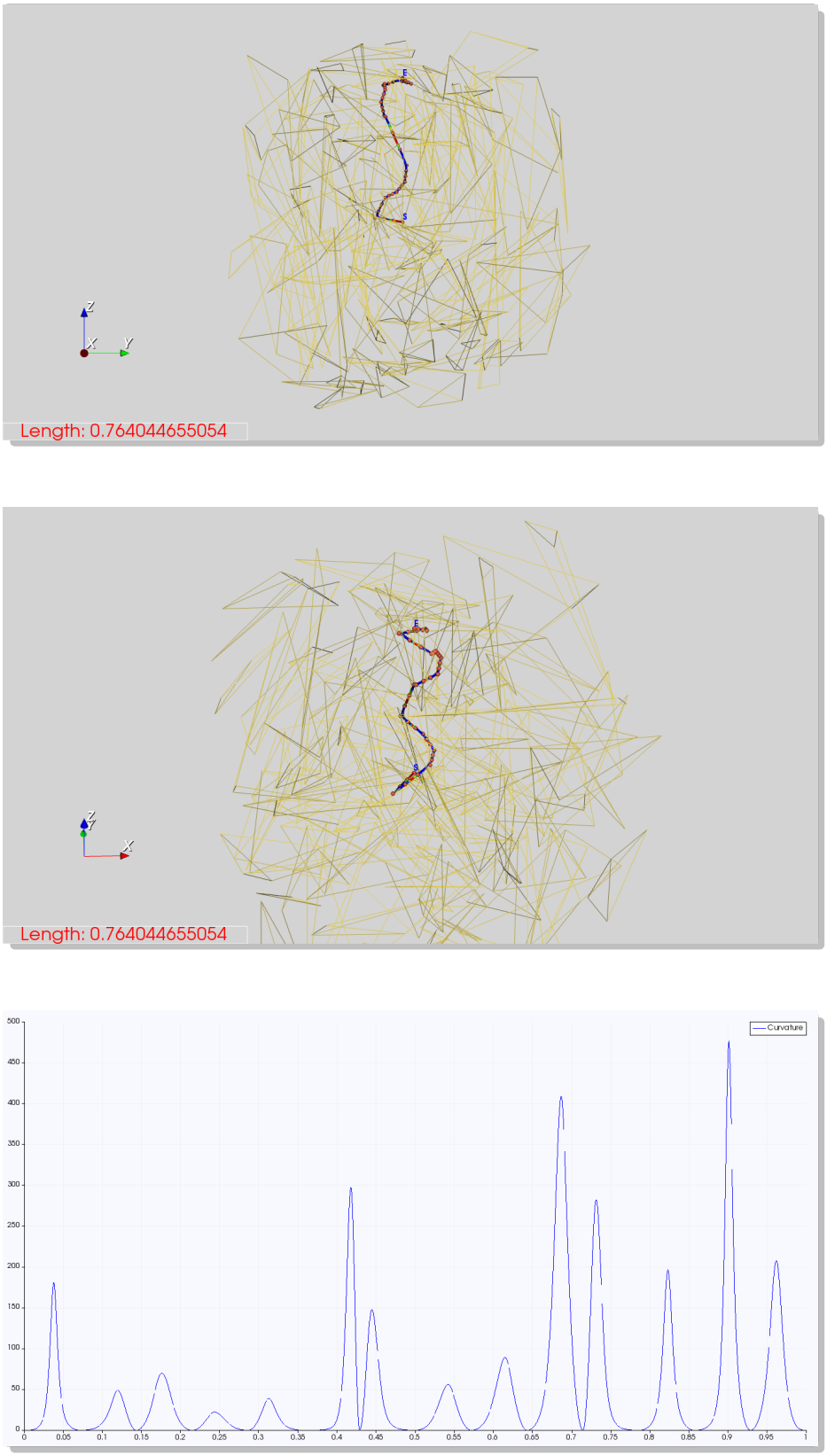


Figure 73.: Test 45; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 4; Meth. B; Post proc. ✗; Part. **U**.

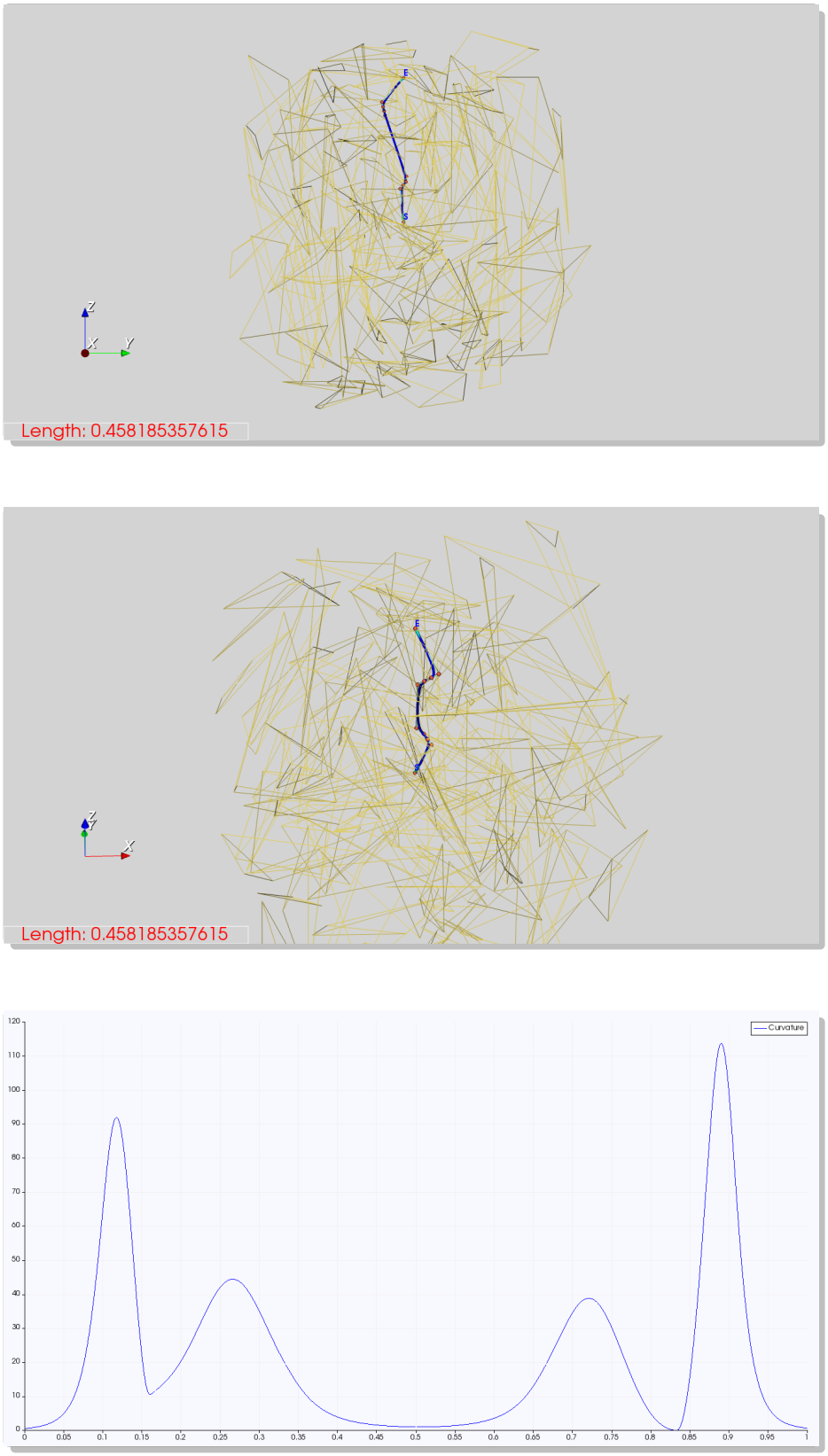


Figure 74.: Test 46; Scene 2; $\mathbf{s} \rightarrow \boldsymbol{e}$ [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 4; Meth. B; Post proc. ✓; Part. **U**.

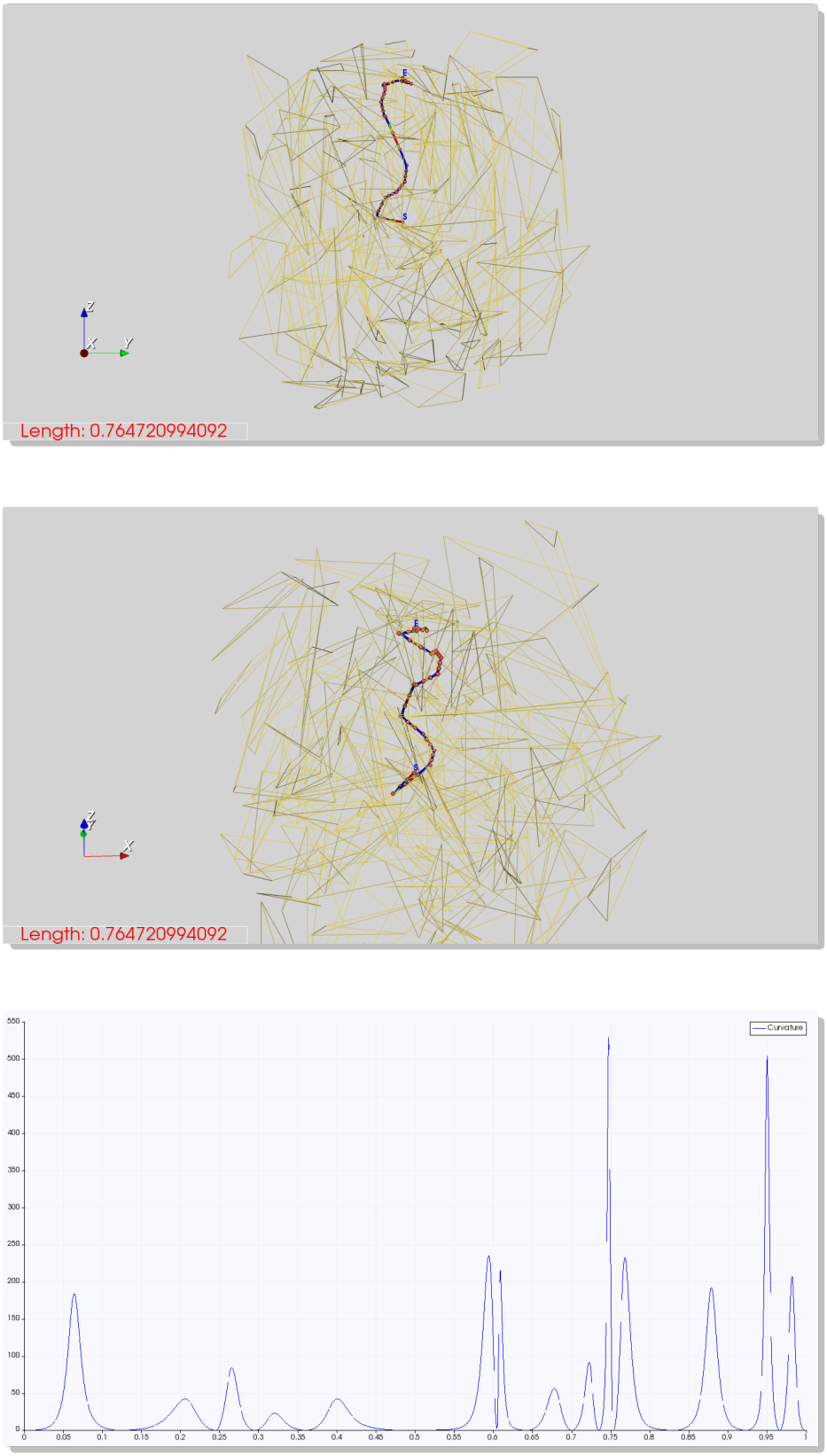
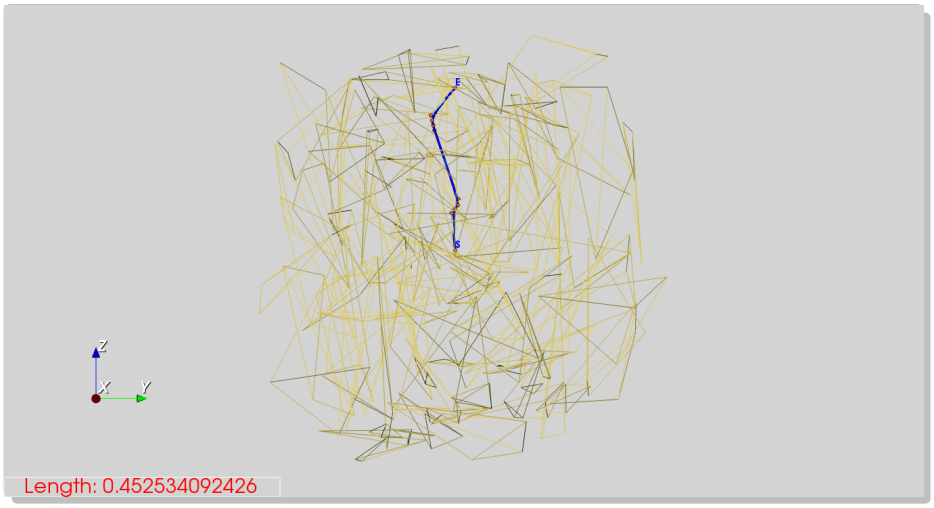
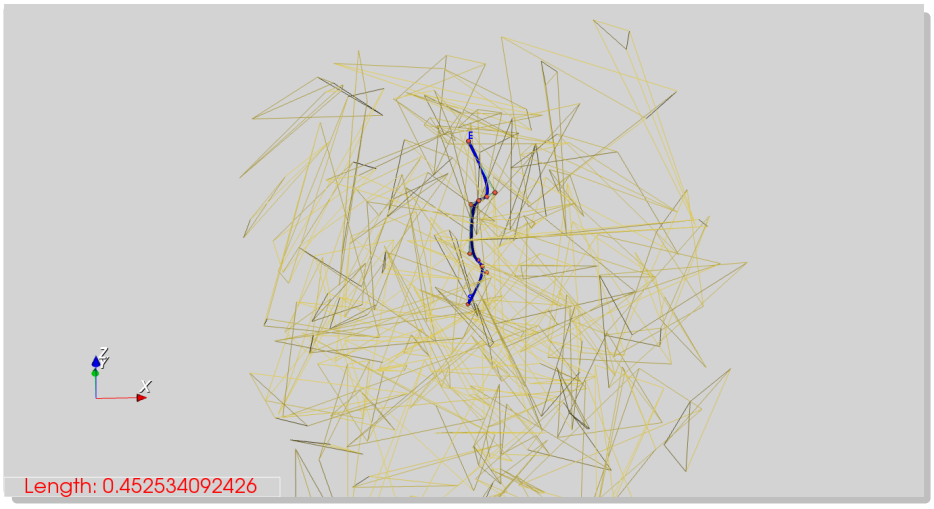


Figure 75.: Test 47; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 4; Meth. B; Post proc. ✗; Part. **A**.





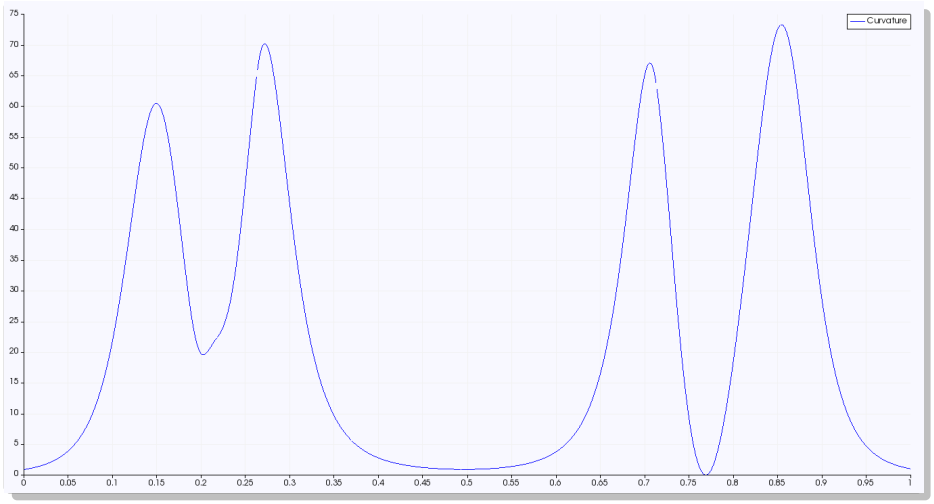
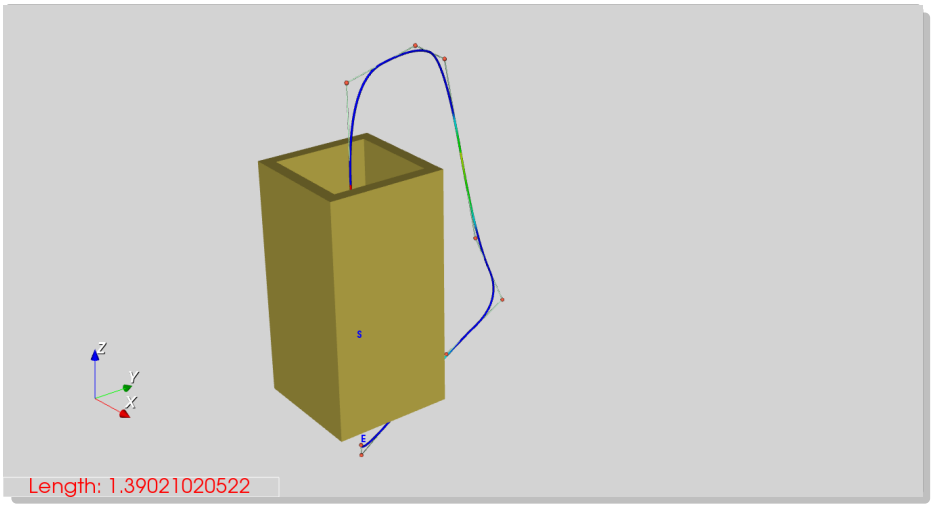
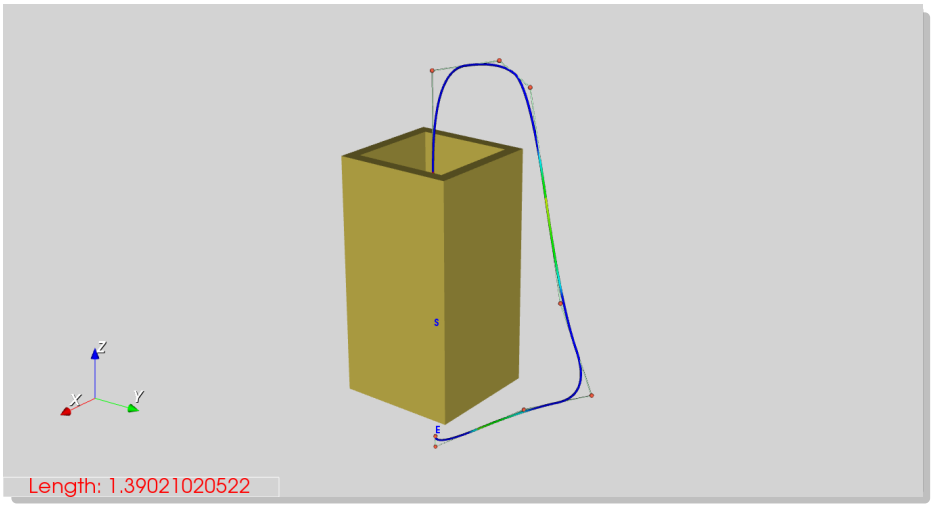


Figure 76.: Test 48; Scene 2; $\mathbf{s} \rightarrow \boldsymbol{e}$ [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 4; Meth. B; Post proc. ✓; Part. **A**.





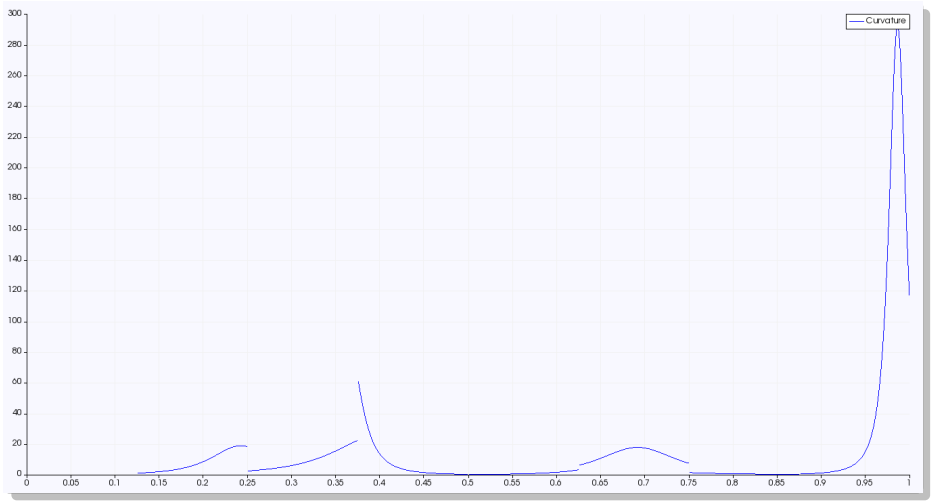


Figure 77.: Test 49; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 2; Meth. A; Post proc. ✗; Part. **U**.

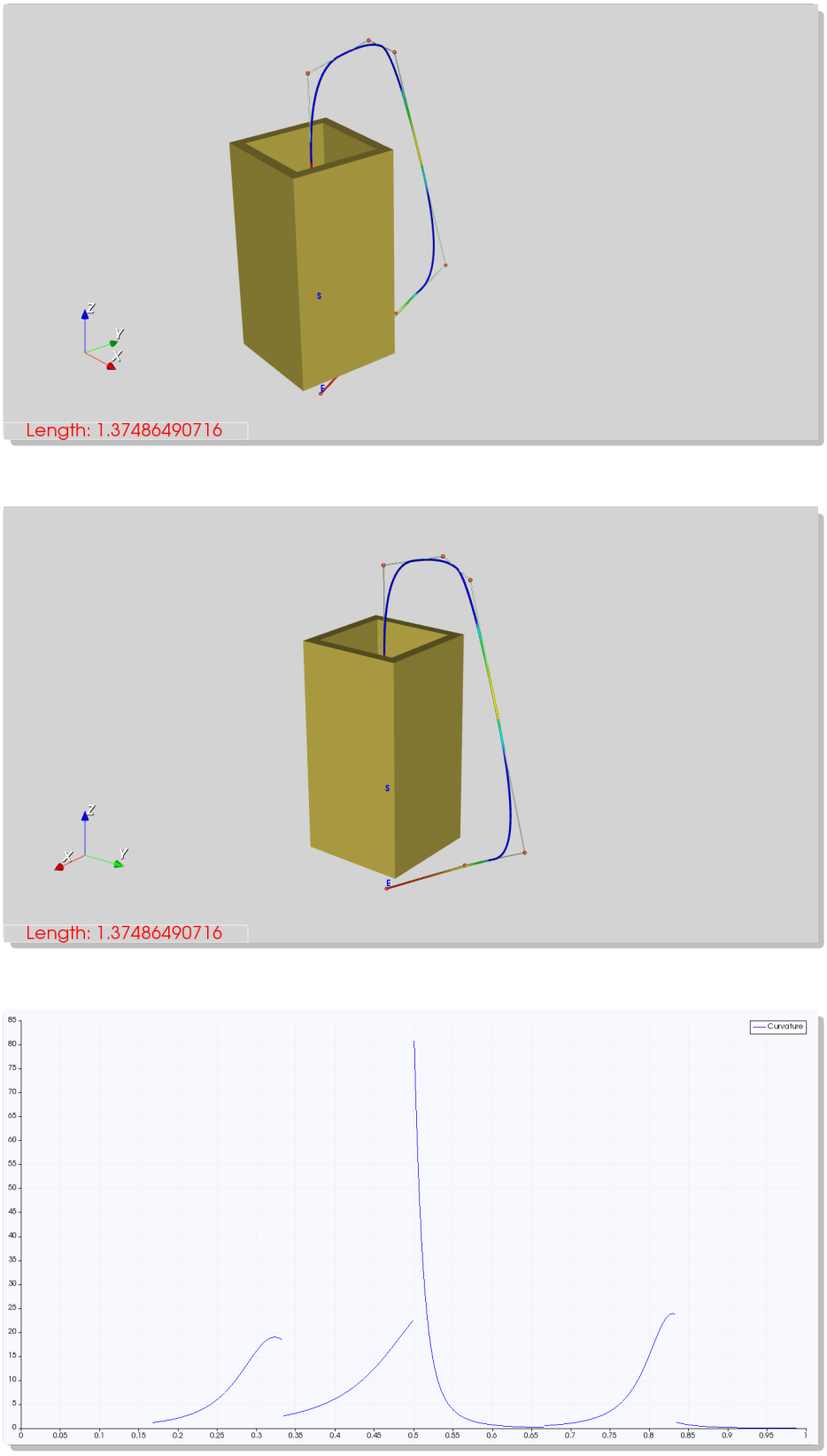
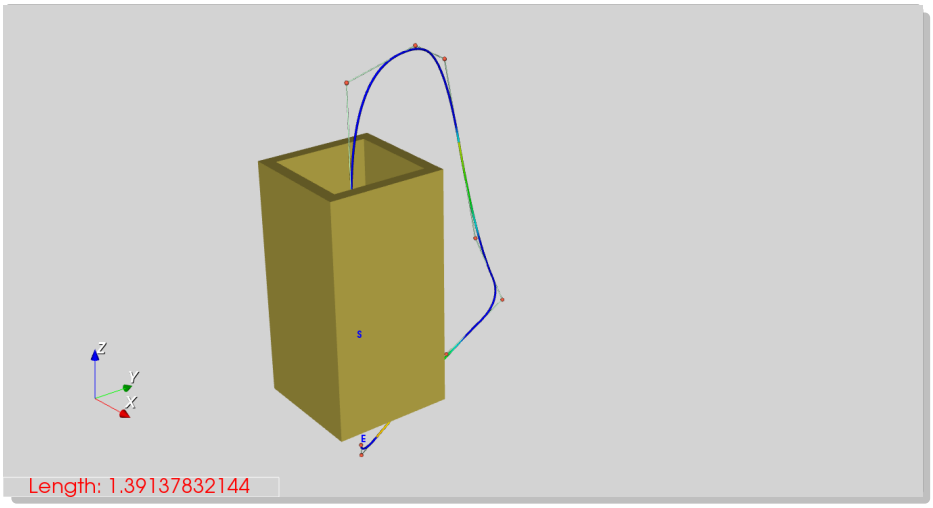
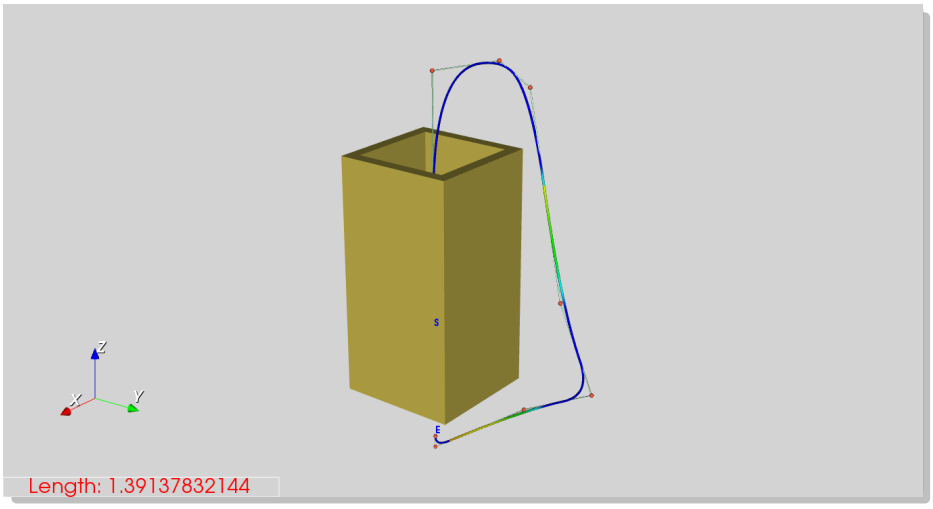


Figure 78.: Test 50; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 2; Meth. A; Post proc. ✓; Part. **U**.





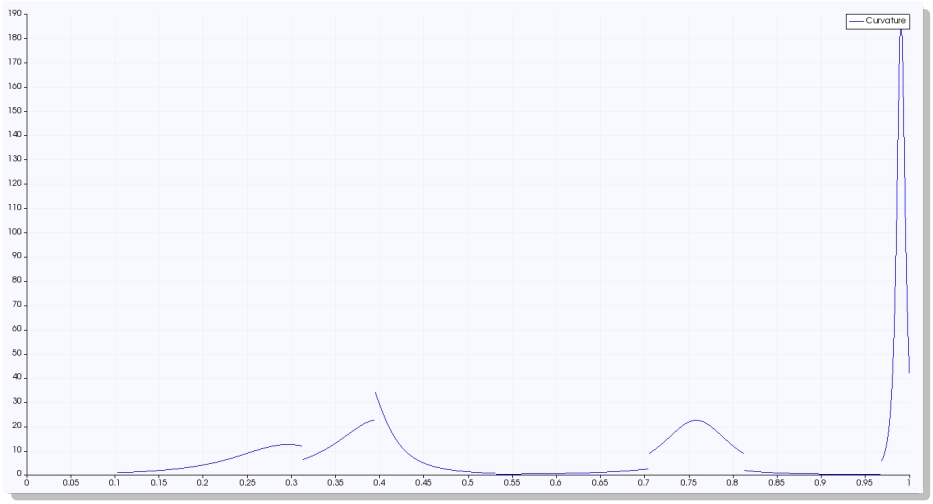


Figure 79.: Test 51; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 2; Meth. A; Post proc. ✗; Part. **A**.

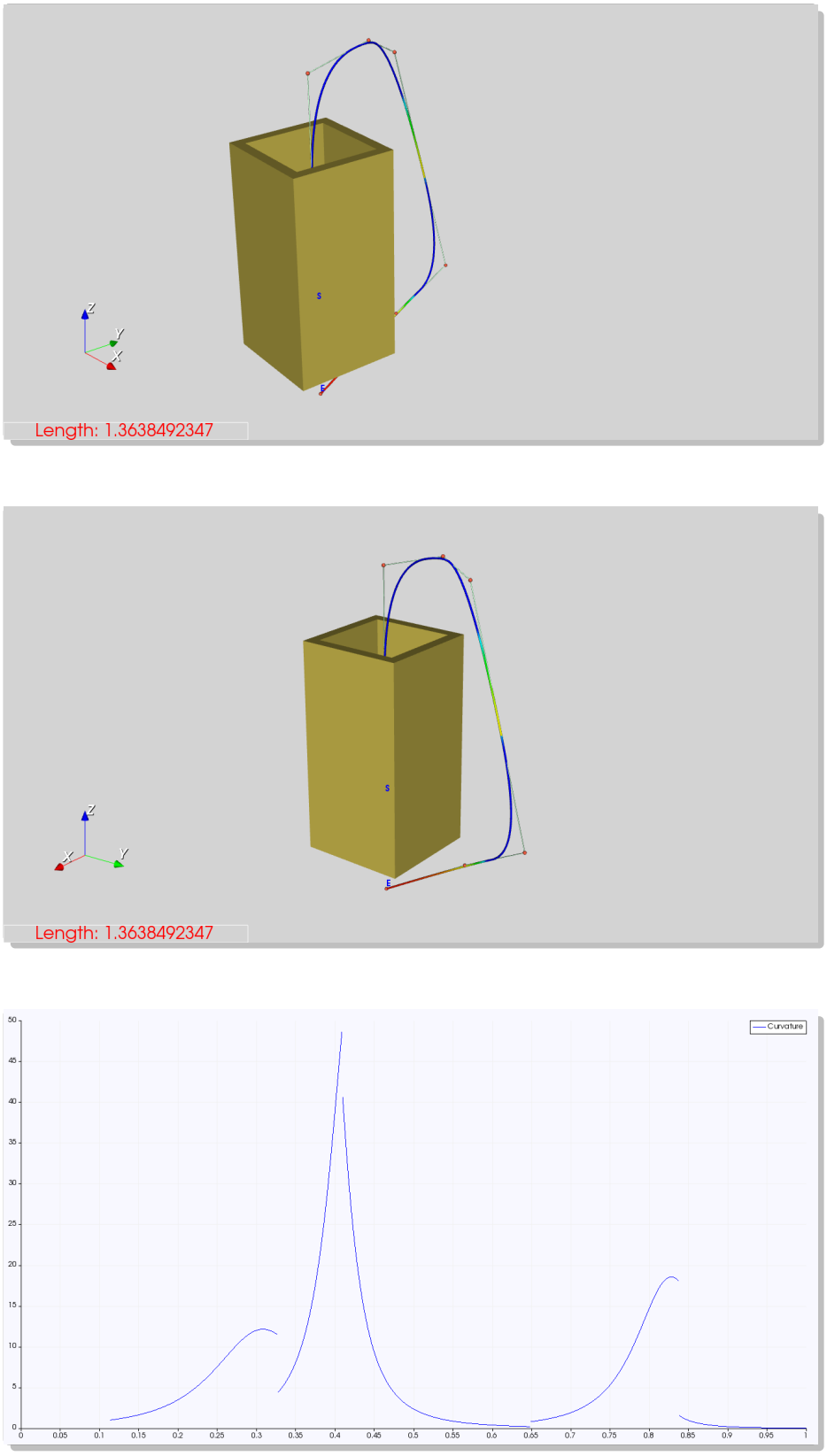
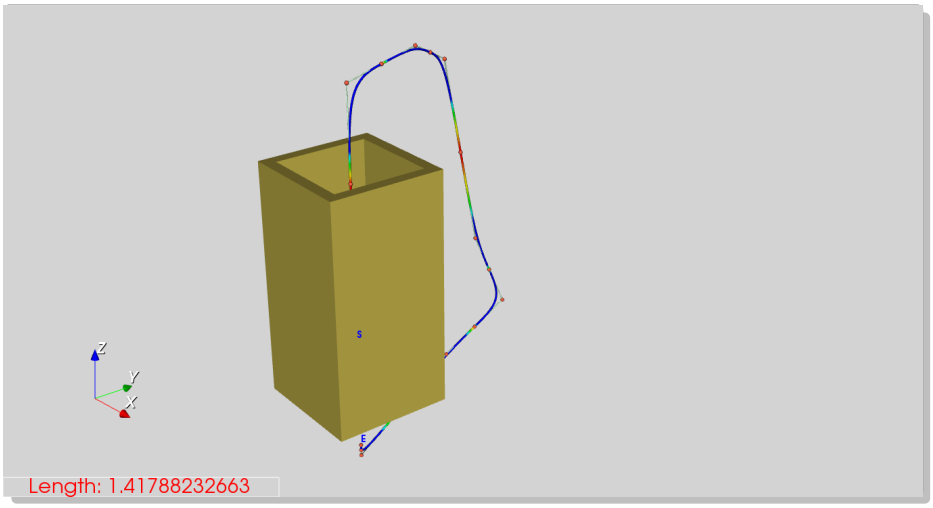
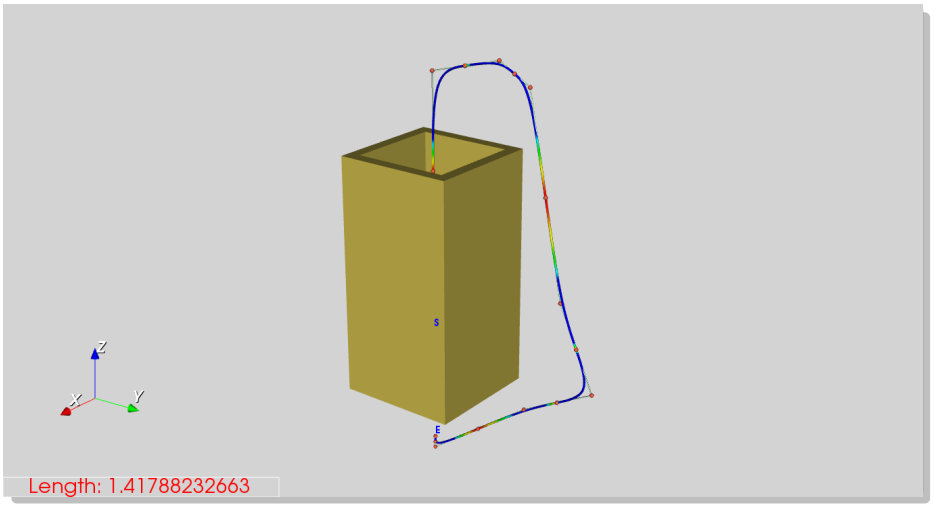


Figure 80.: Test 52; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 2; Meth. A; Post proc. ✓; Part. **A**.





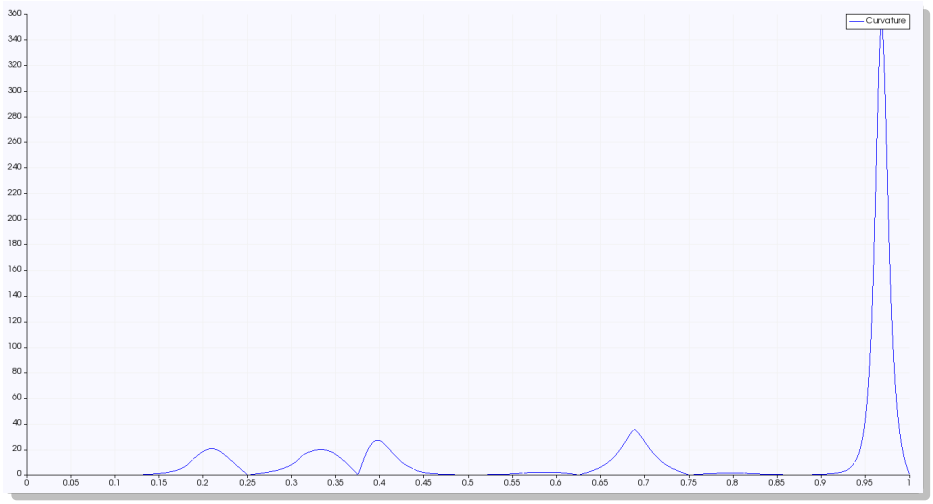


Figure 81.: Test 53; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 3; Meth. A; Post proc. ✗; Part. **U**.

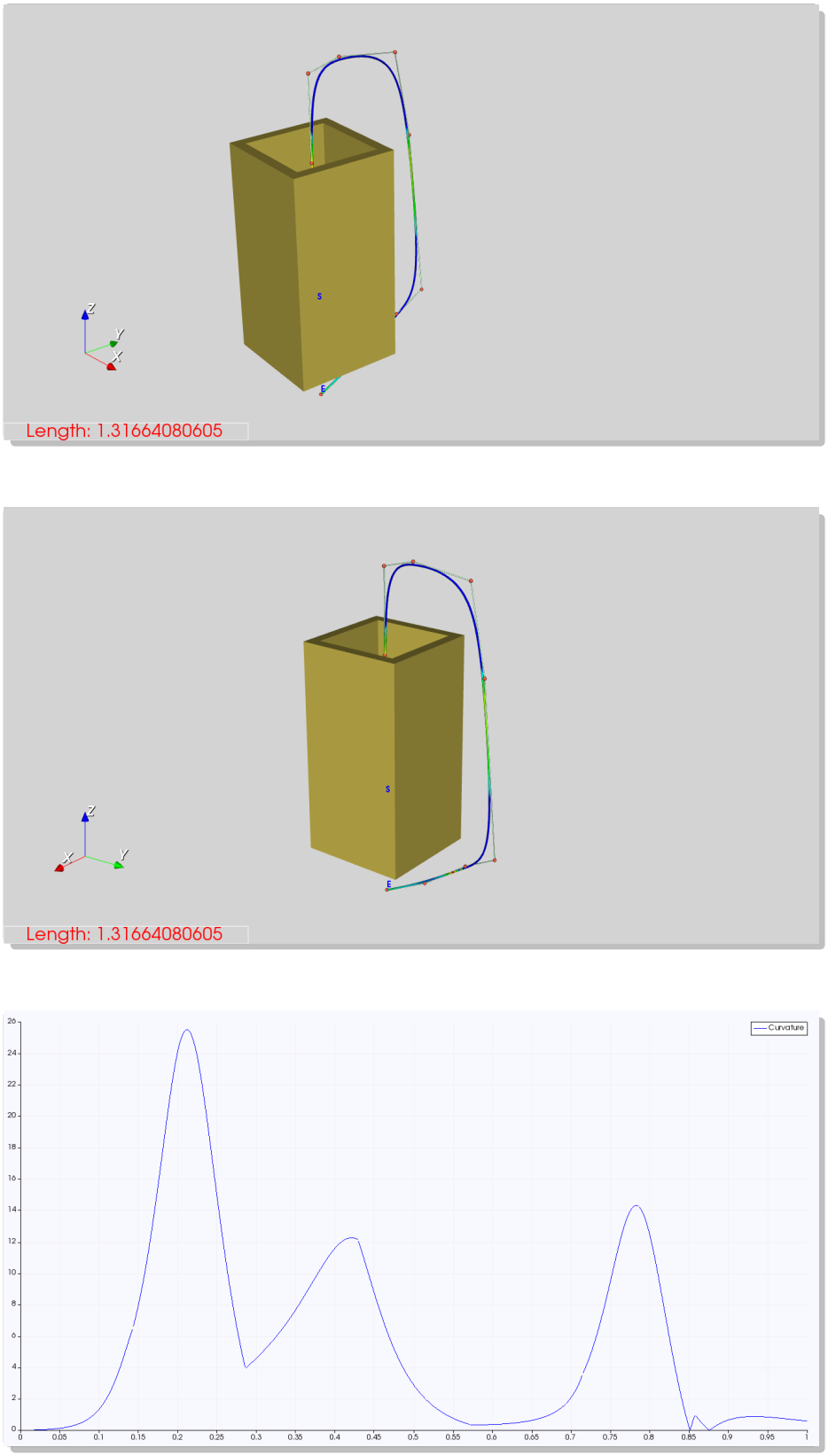
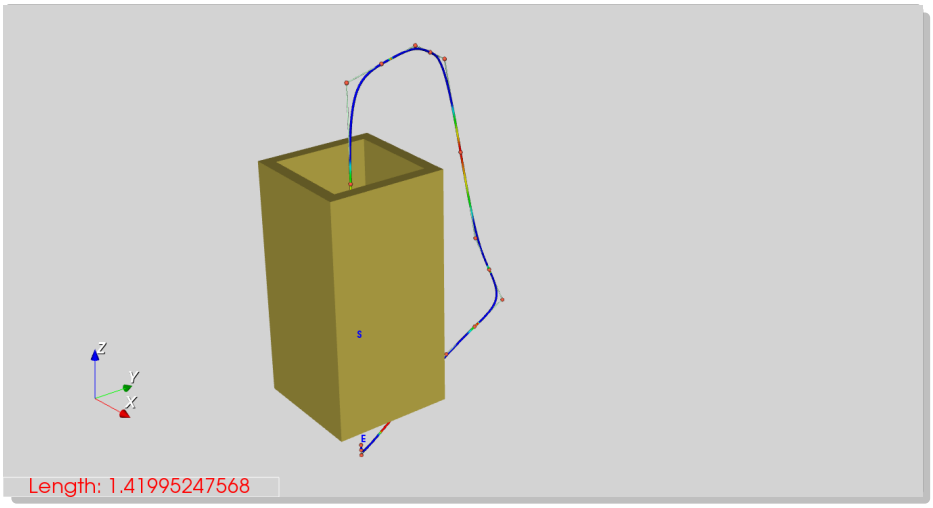
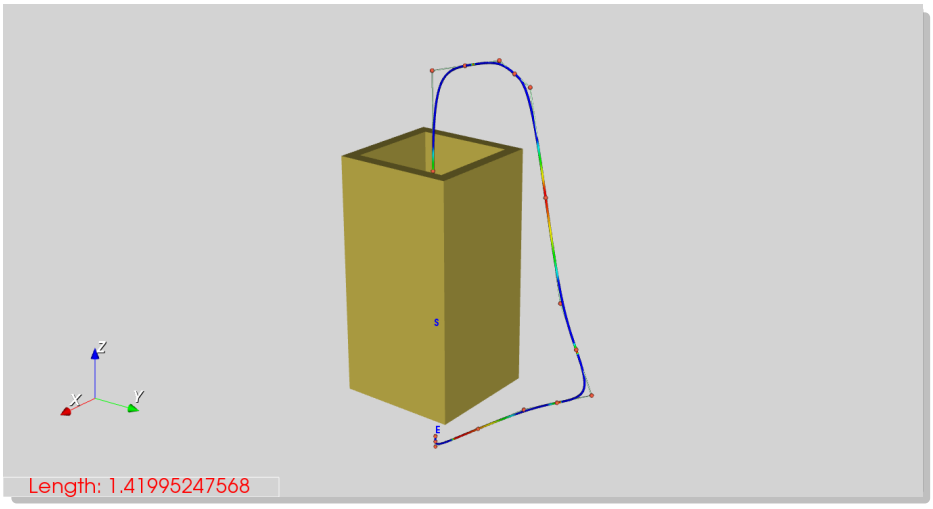


Figure 82.: Test 54; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 3; Meth. A; Post proc. ✓; Part. **U**.





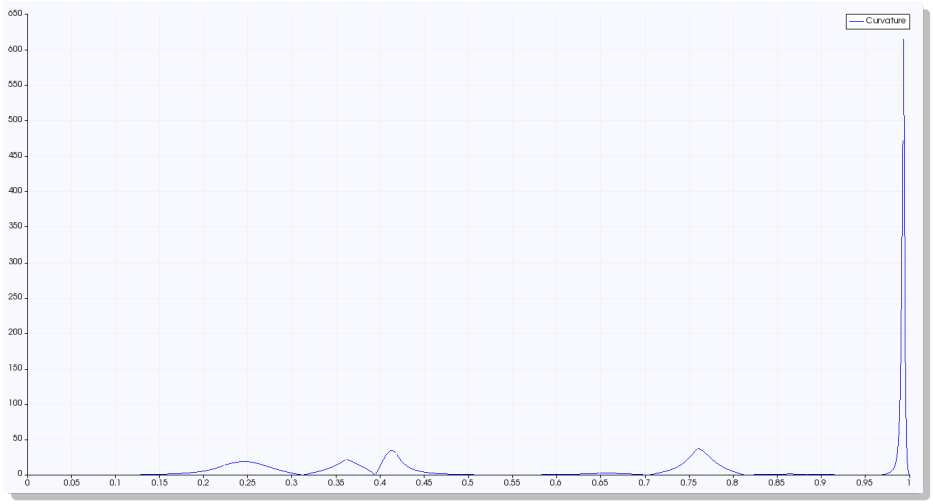


Figure 83.: Test 55; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 3; Meth. A; Post proc. ✗; Part. **A**.

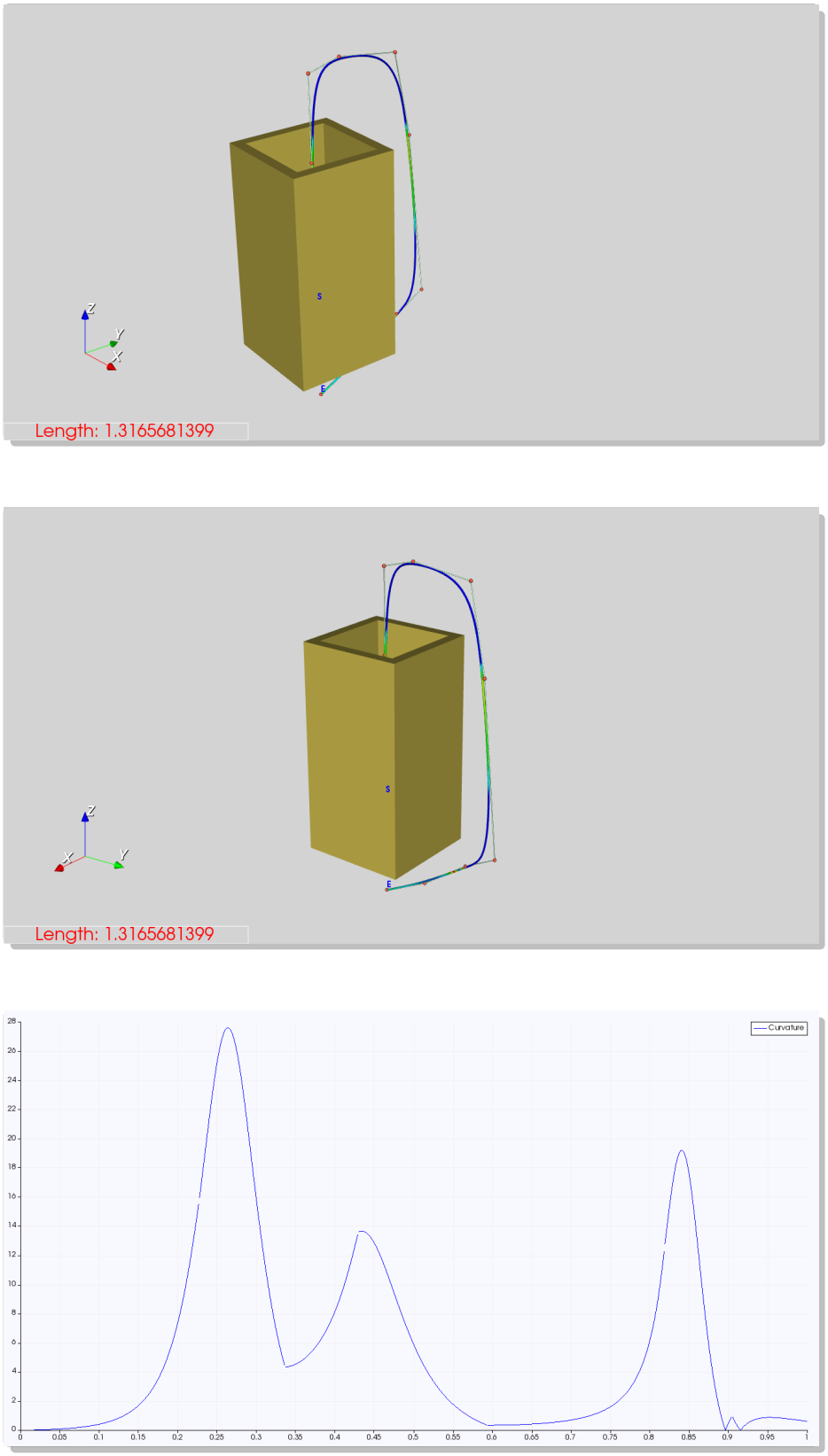
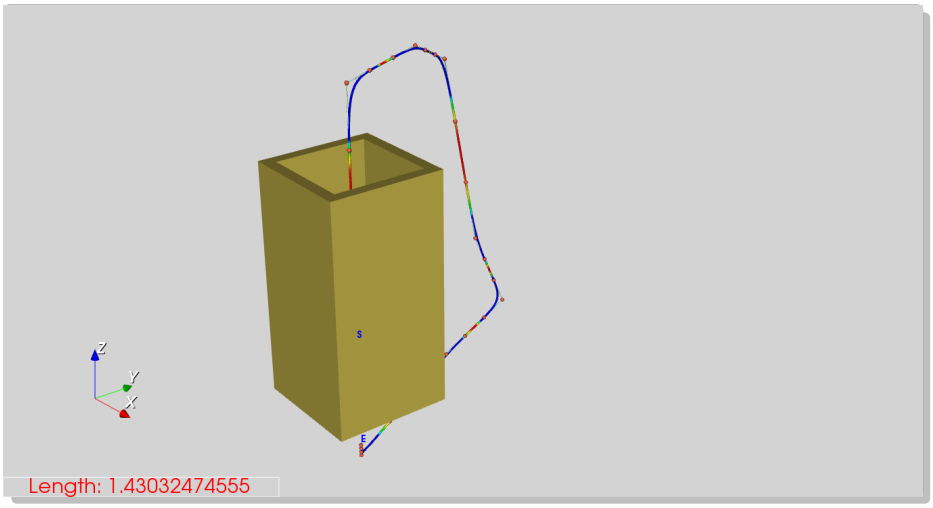
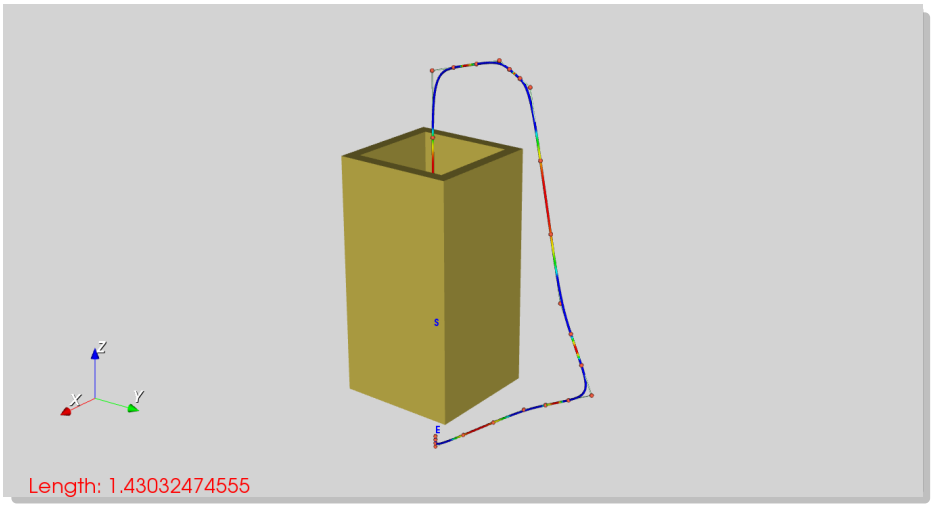


Figure 84.: Test 56; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 3; Meth. A; Post proc. ✓; Part. **A**.





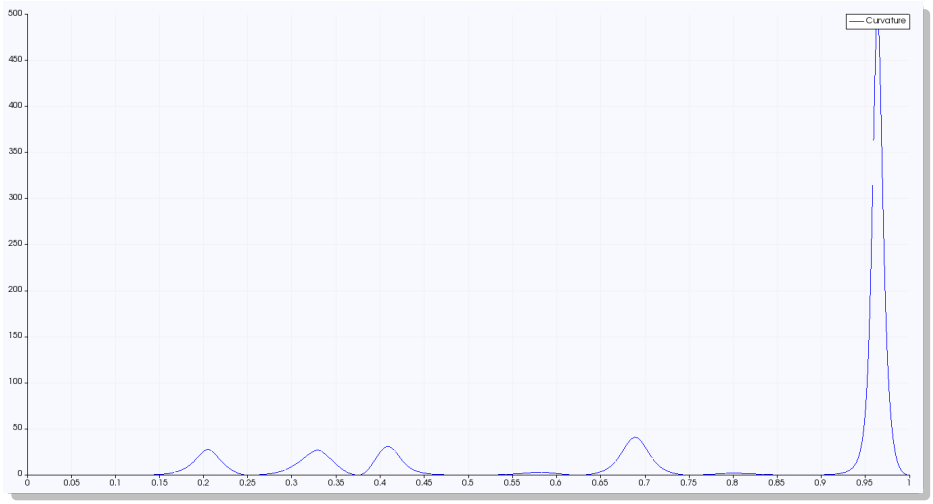


Figure 85.: Test 57; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 4; Meth. A; Post proc. ✗; Part. **U**.

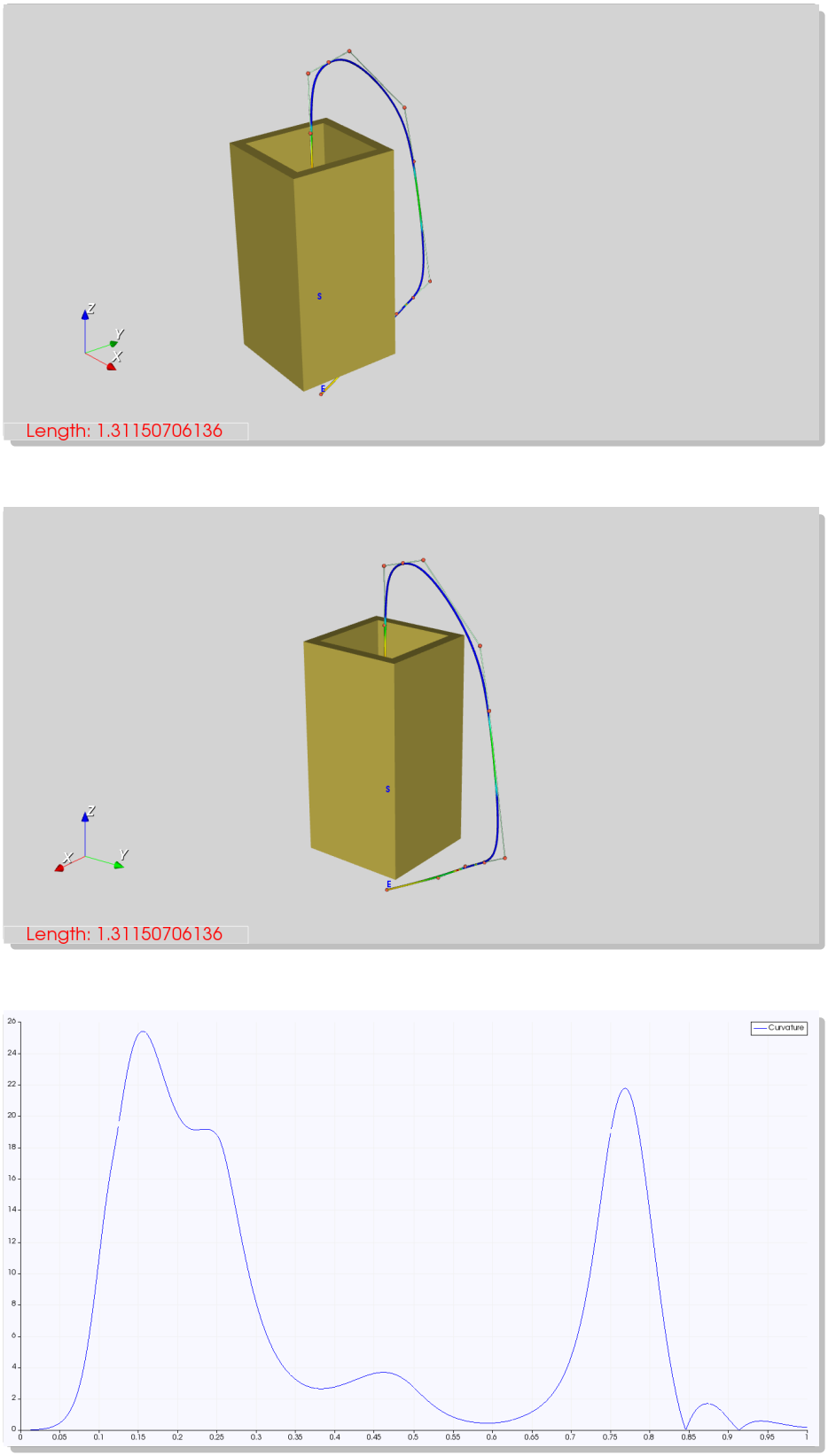
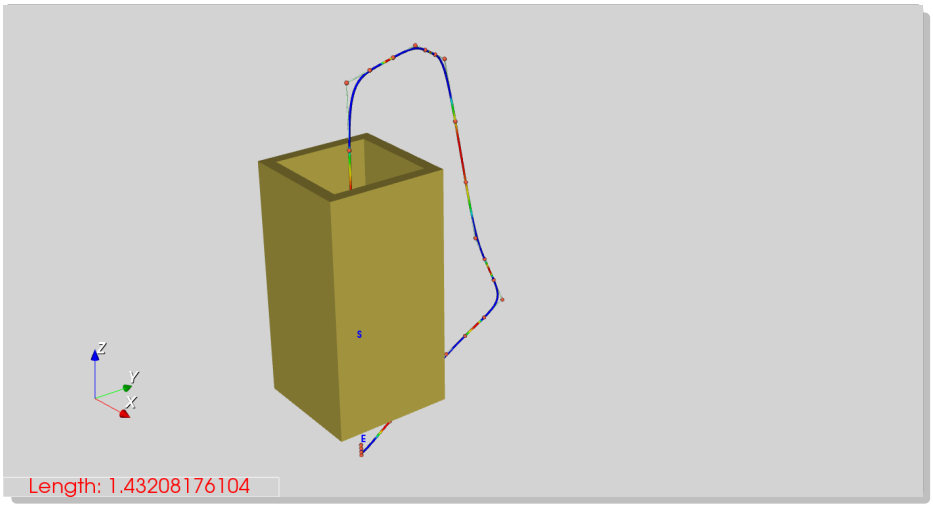
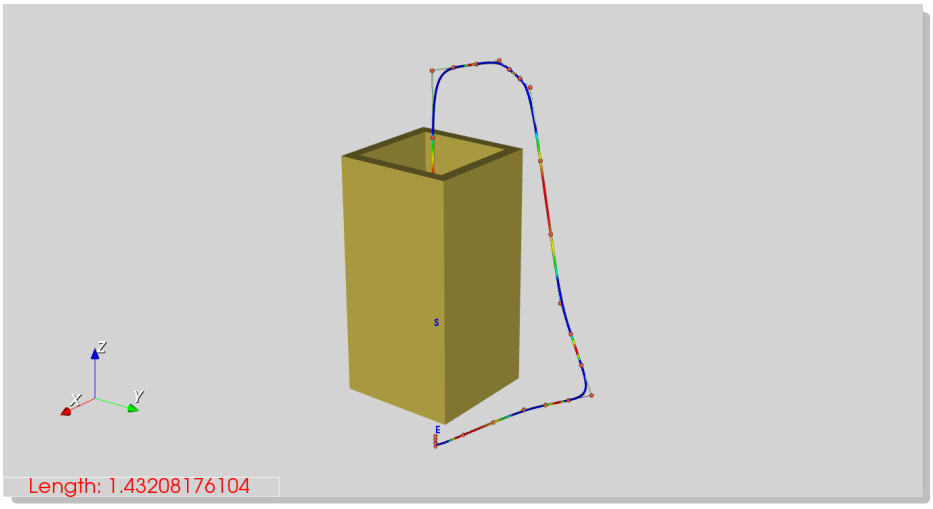


Figure 86.: Test 58; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 4; Meth. A; Post proc. ✓; Part. **U**.





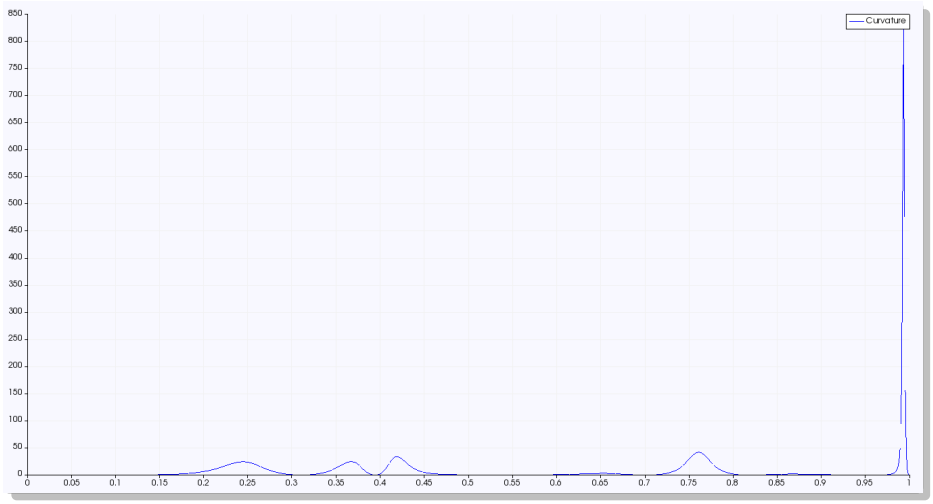


Figure 87.: Test 59; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 4; Meth. A; Post proc. ✗; Part. **A**.

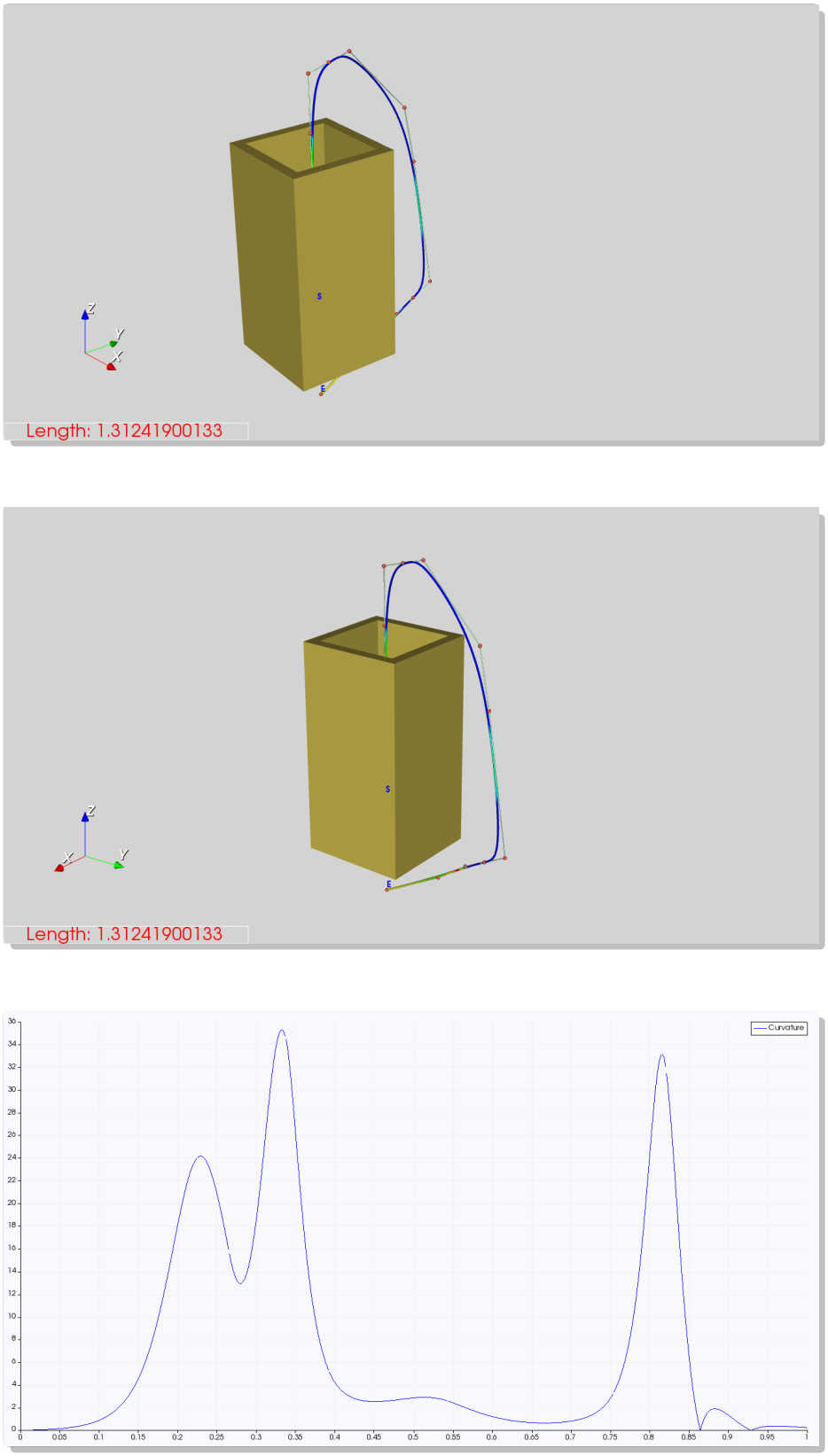
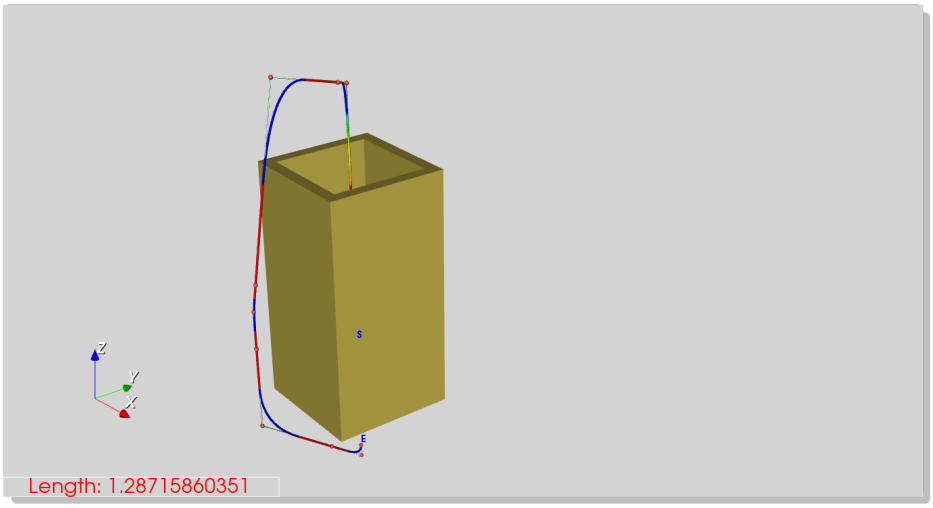
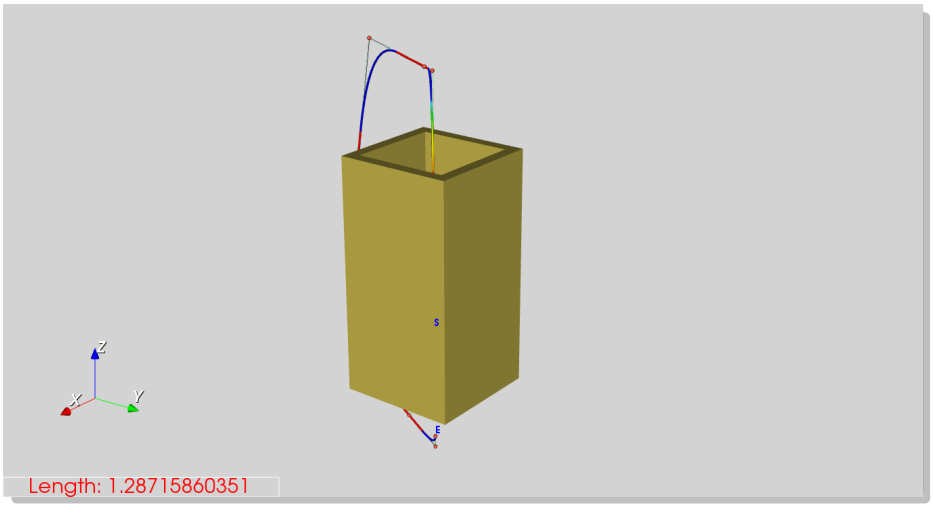


Figure 88.: Test 60; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 4; Meth. A; Post proc. ✓; Part. **A**.





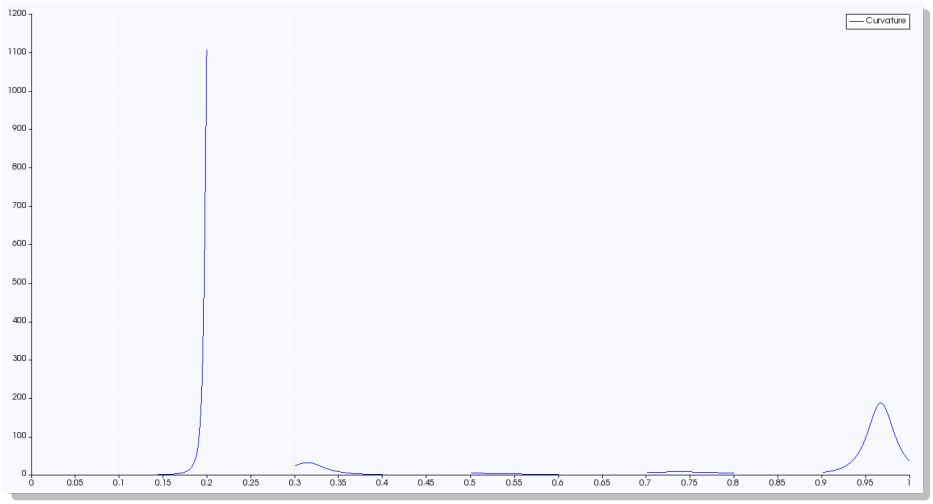
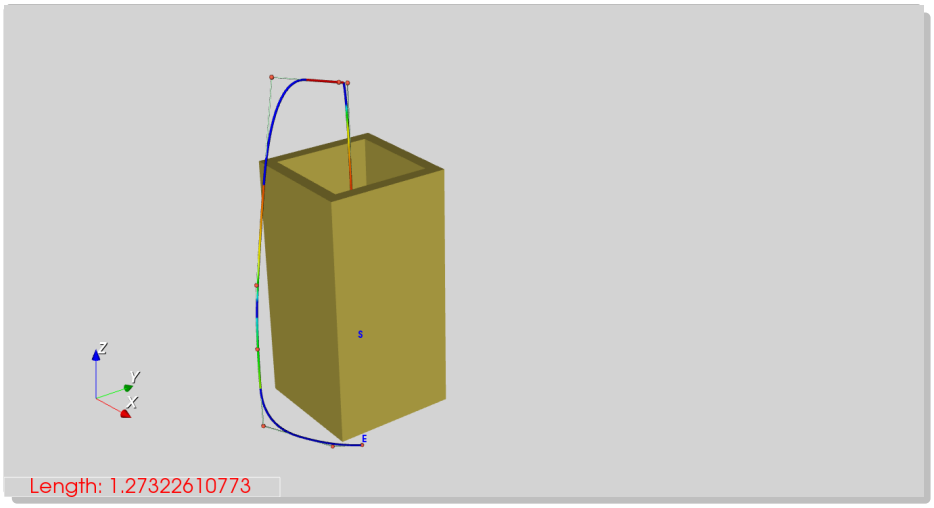
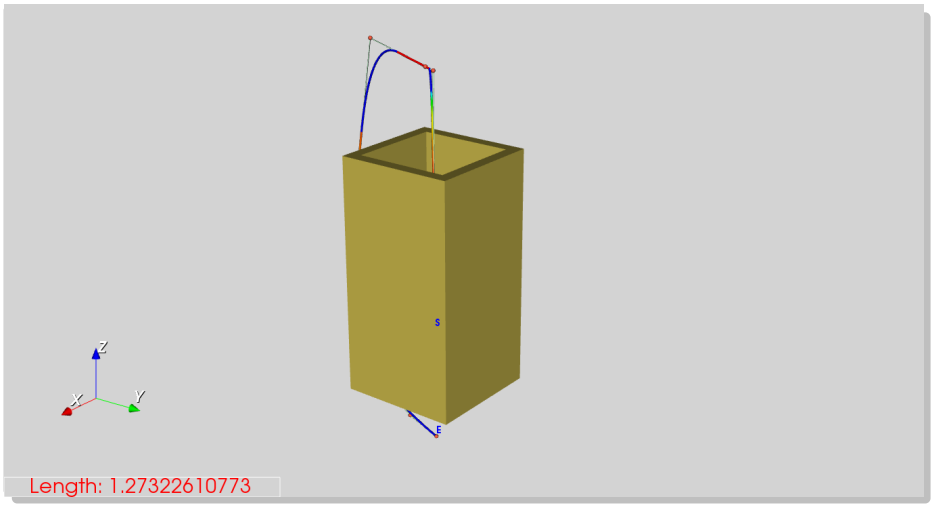


Figure 89.: Test 61; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 2; Meth. B; Post proc. ✗; Part. **U**.





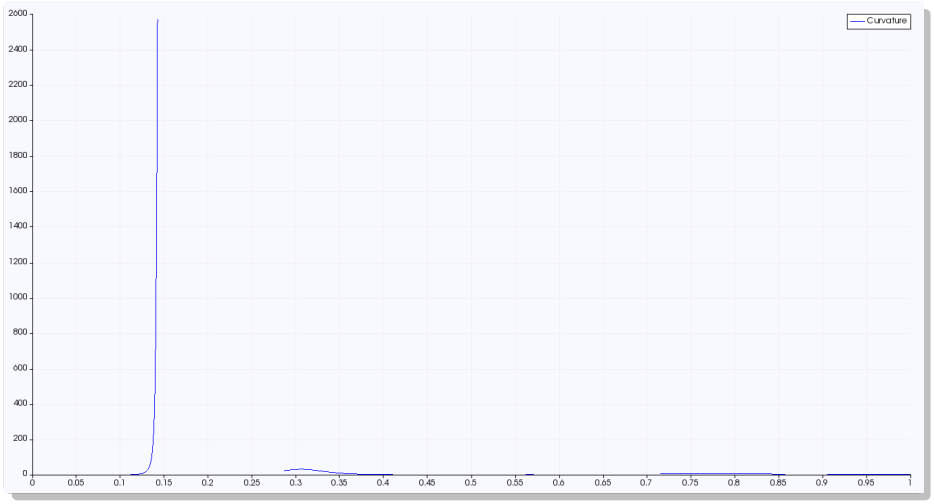


Figure 90.: Test 62; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 2; Meth. B; Post proc. ✓; Part. **U**.

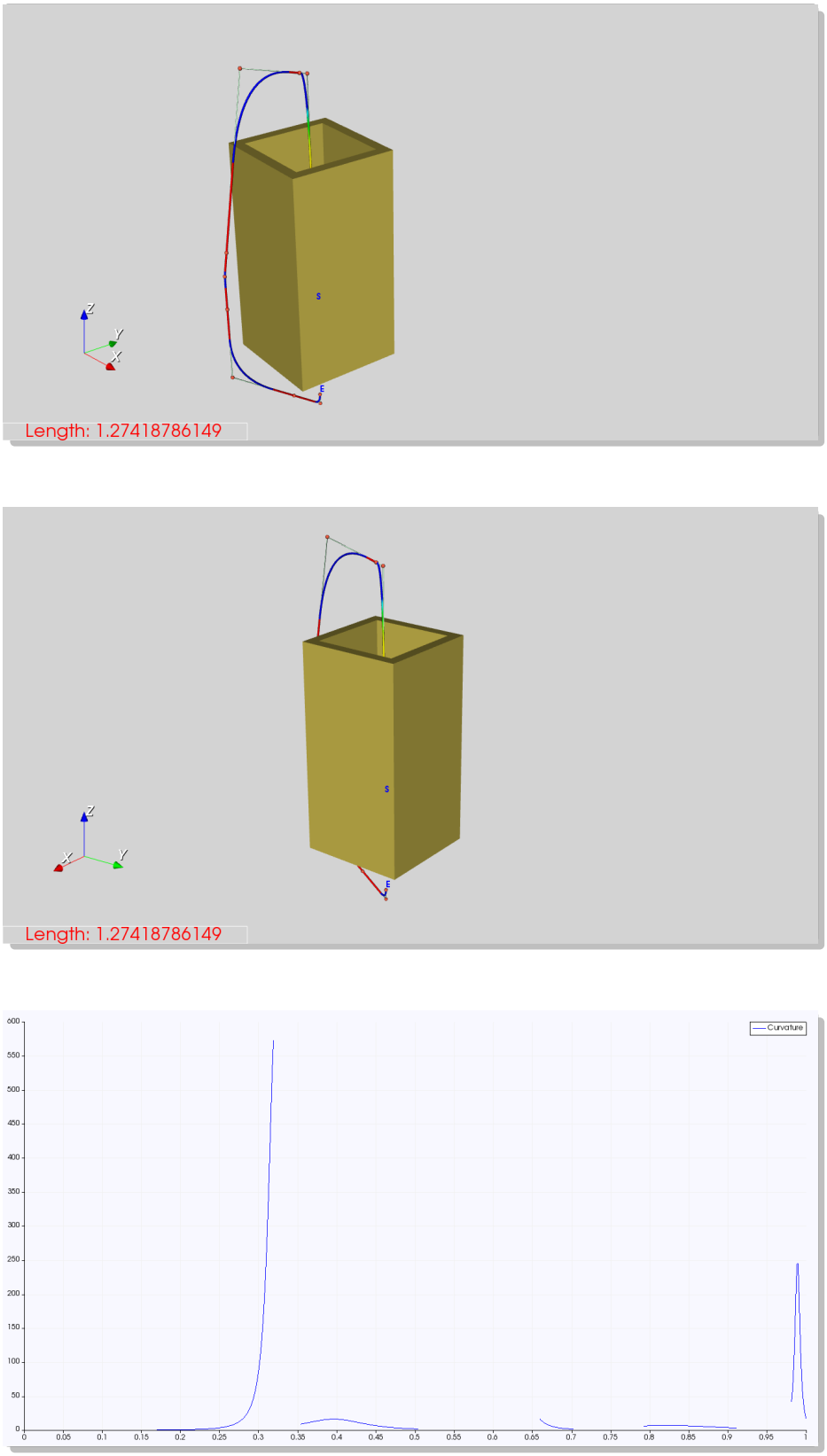


Figure 91.: Test 63; Scene 3; **s** → **e** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 2; Meth. B; Post proc. ✗; Part. **A**.

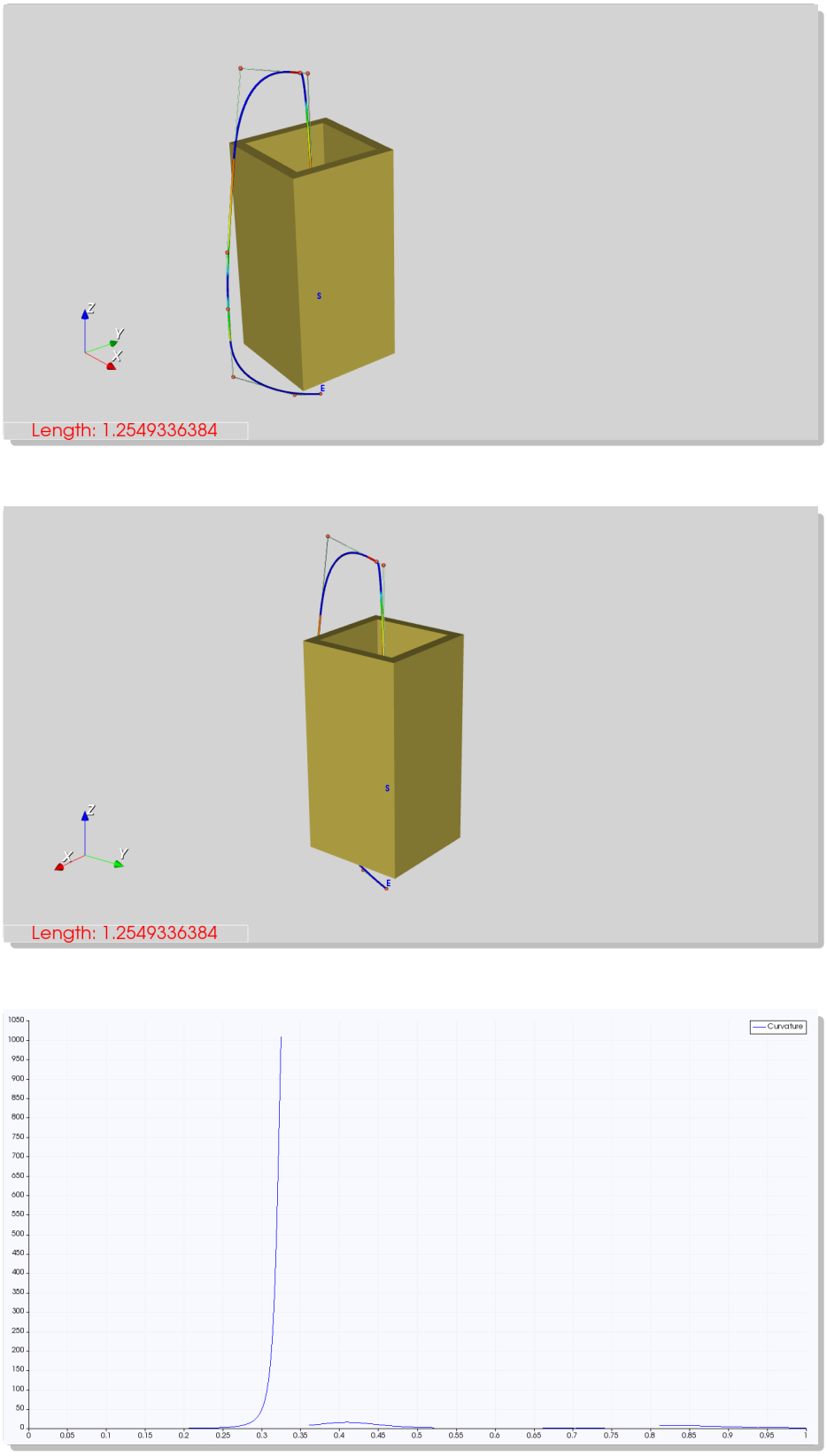


Figure 92.: Test 64; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 2; Meth. B; Post proc. ✓; Part. **A**.

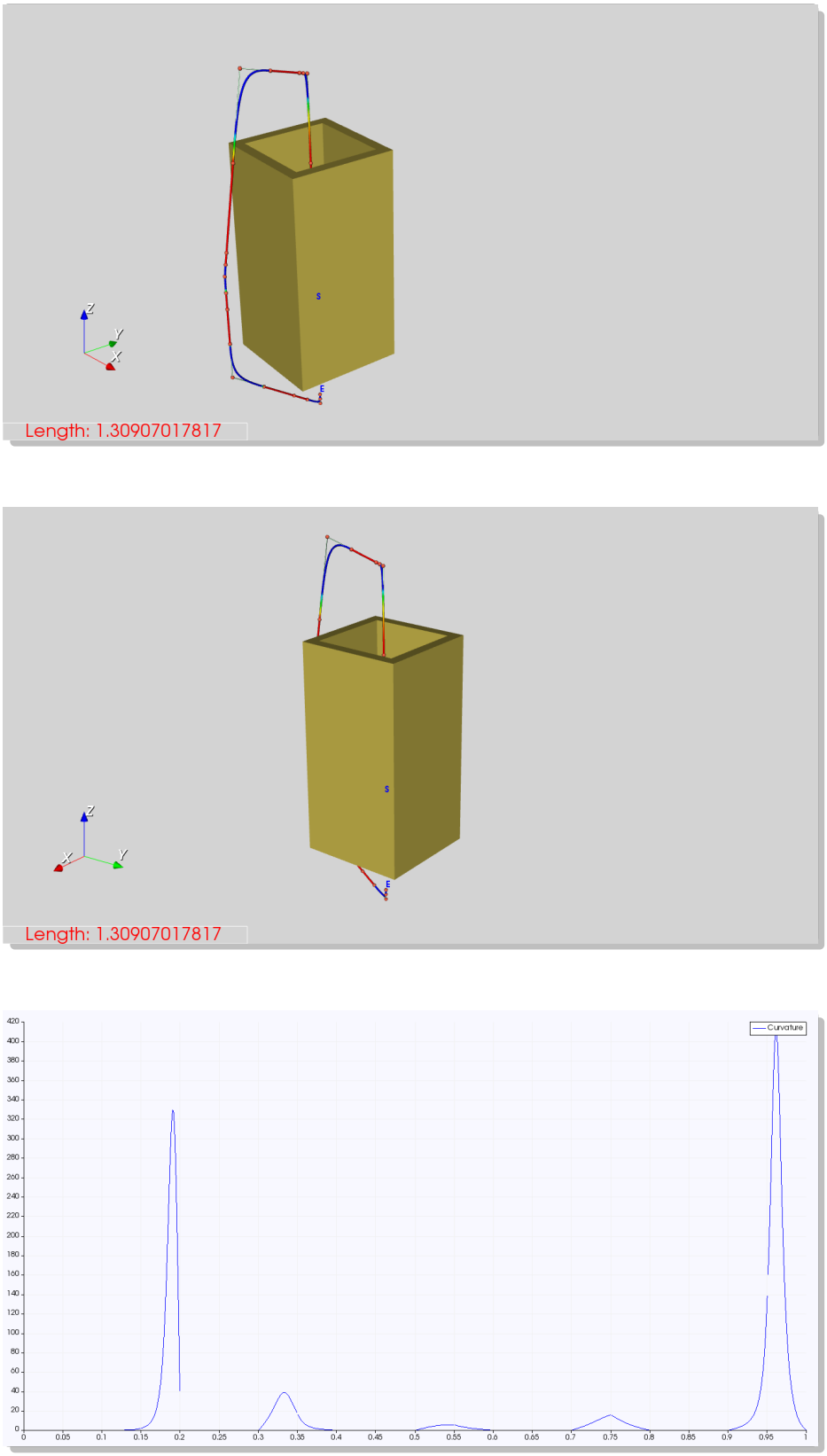


Figure 93.: Test 65; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 3; Meth. B; Post proc. ✗; Part. **U**.

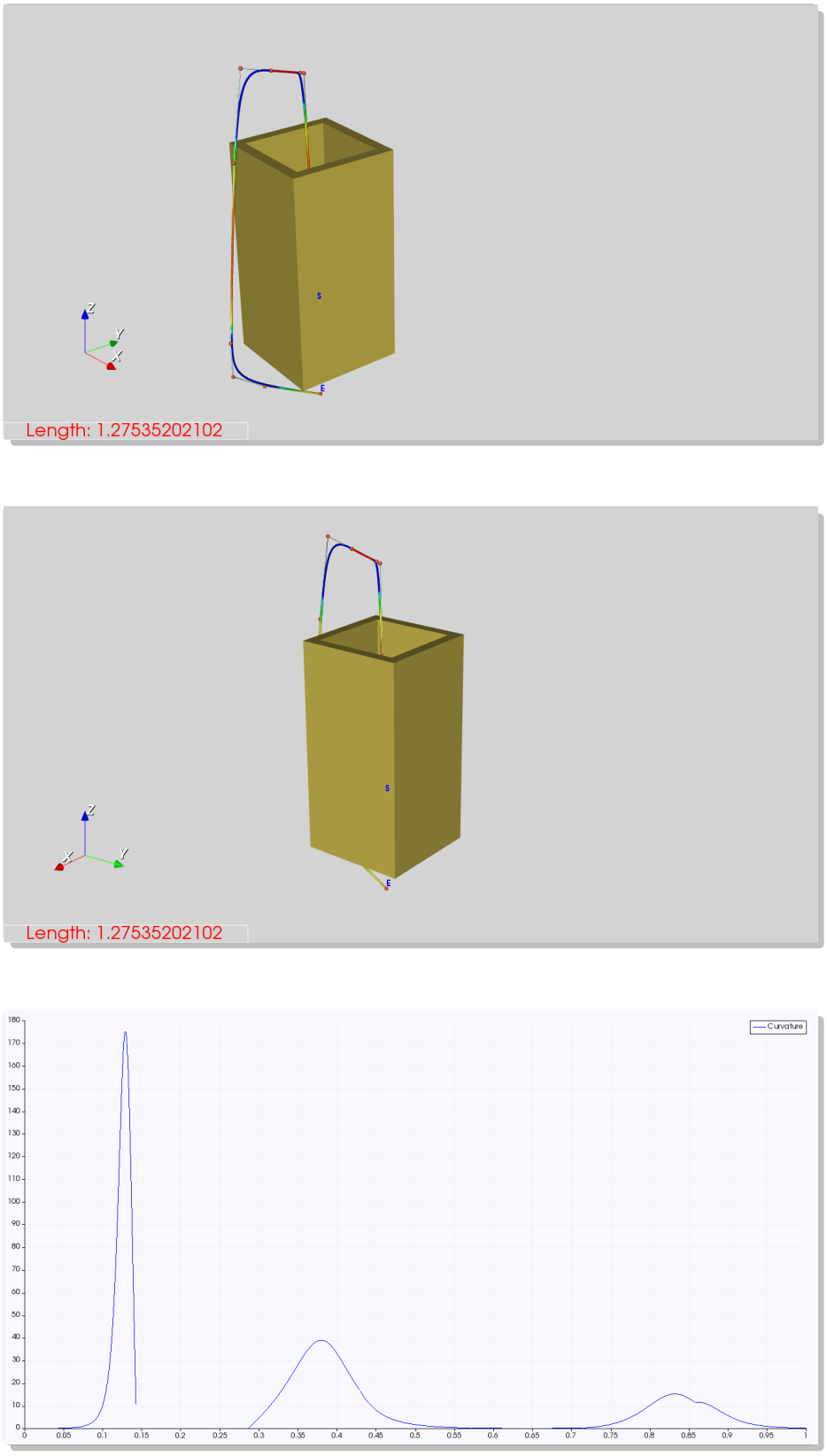
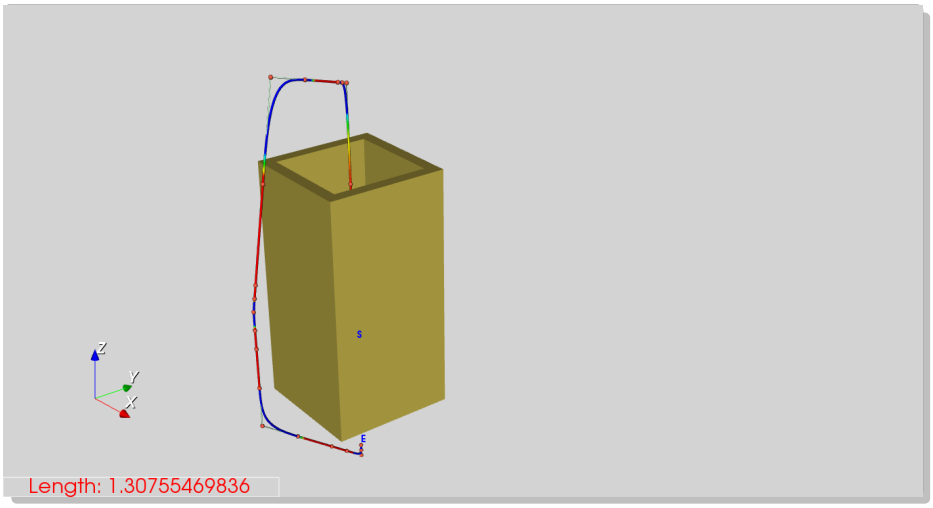
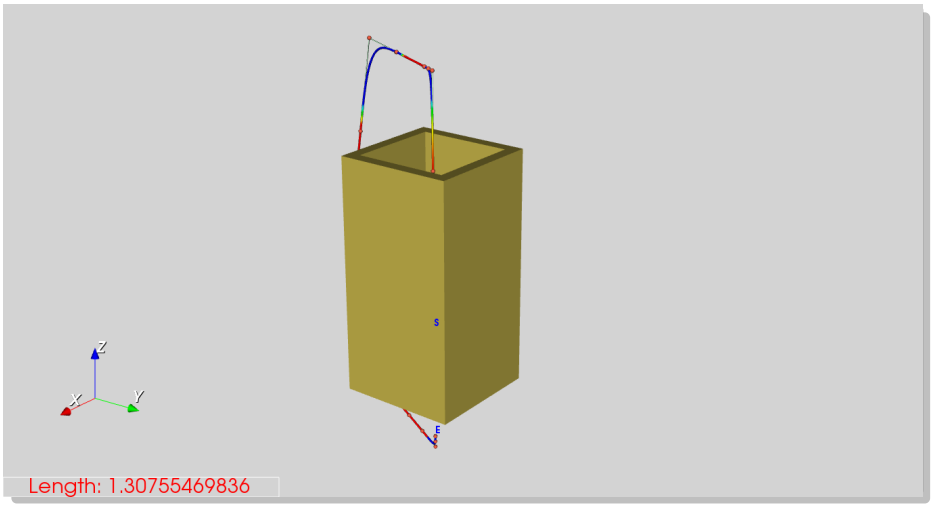


Figure 94.: Test 66; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 3; Meth. B; Post proc. ✓; Part. **U**.





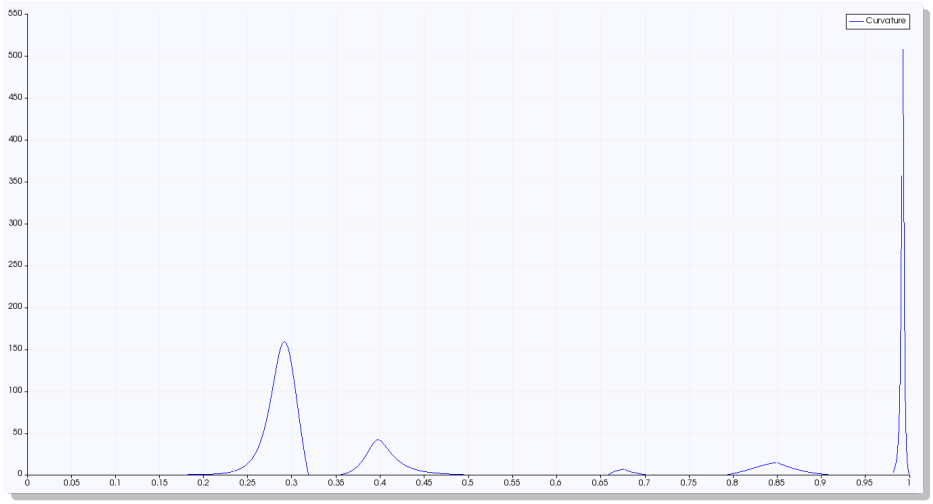


Figure 95.: Test 67; Scene 3; **s** → **e** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 3; Meth. B; Post proc. ✗; Part. **A**.

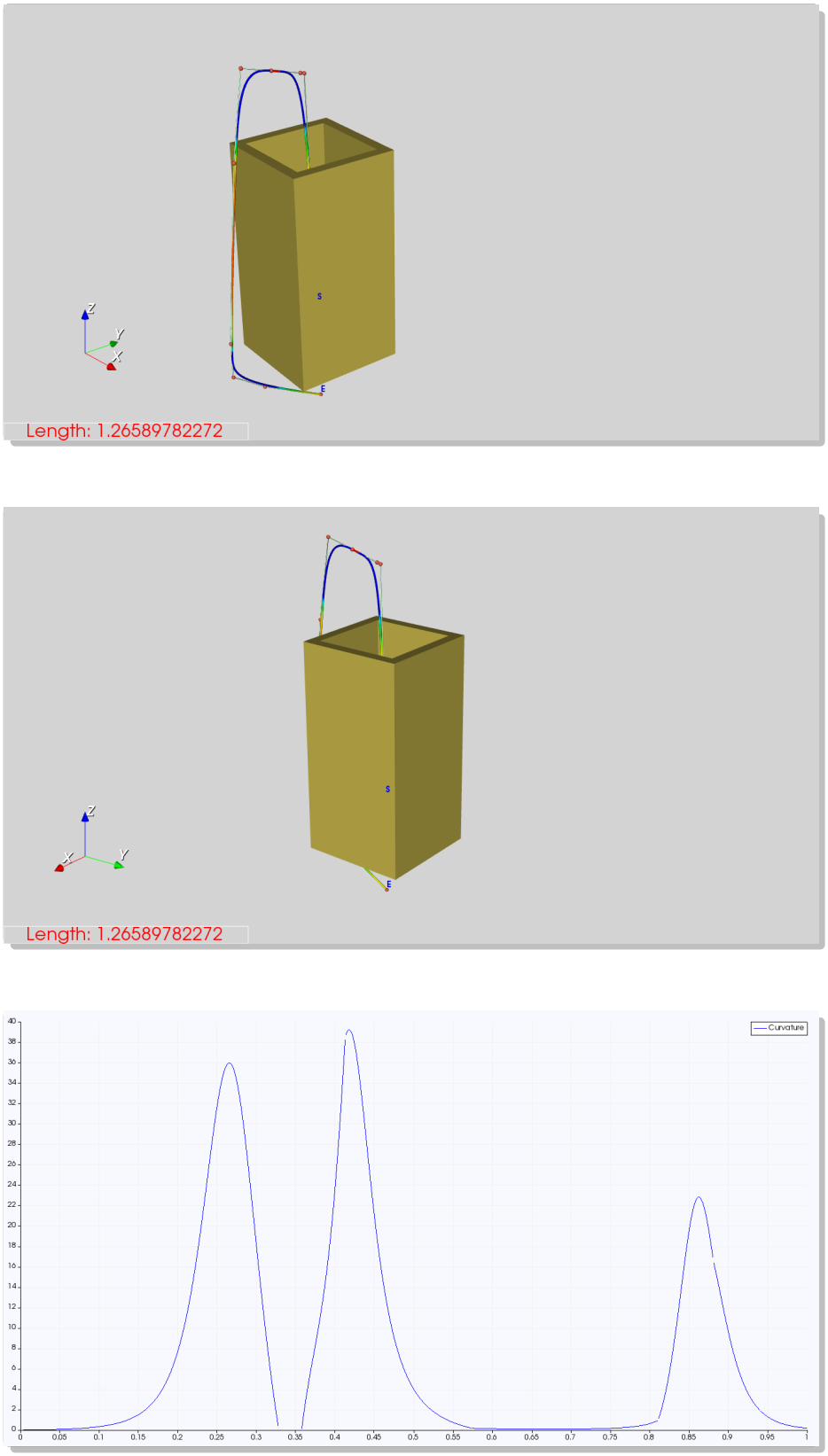
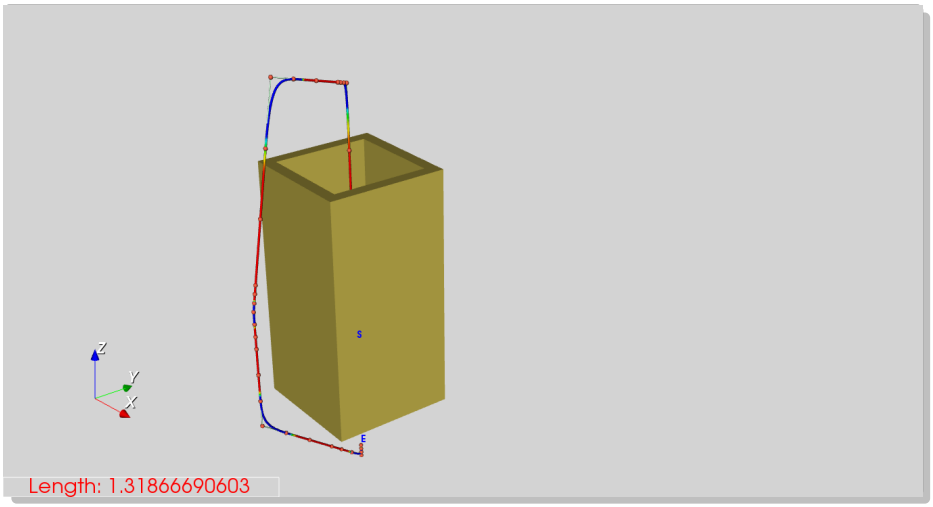
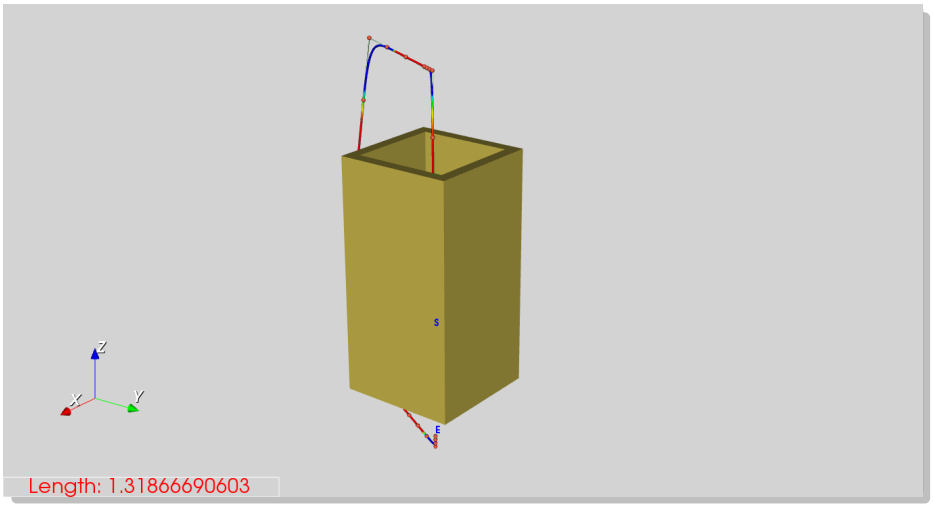


Figure 96.: Test 68; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 3; Meth. B; Post proc. ✓; Part. **A**.





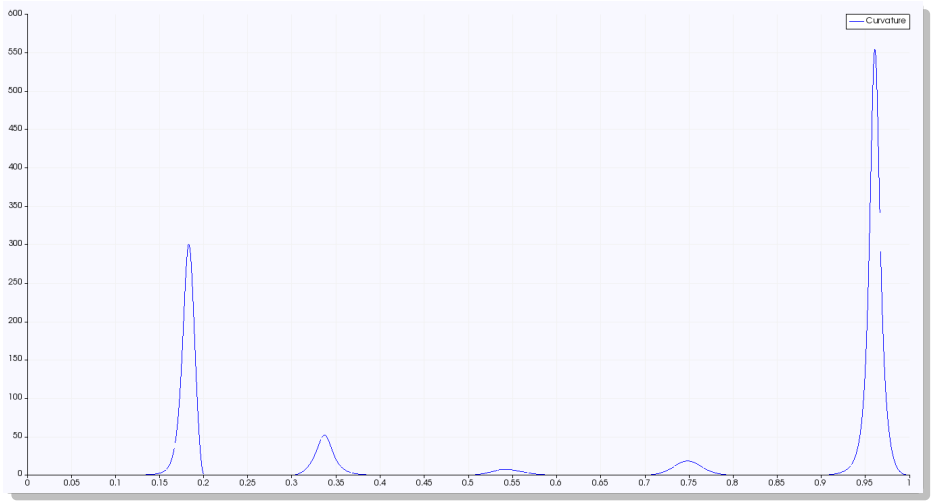


Figure 97.: Test 69; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 4; Meth. B; Post proc. ✗; Part. **U**.

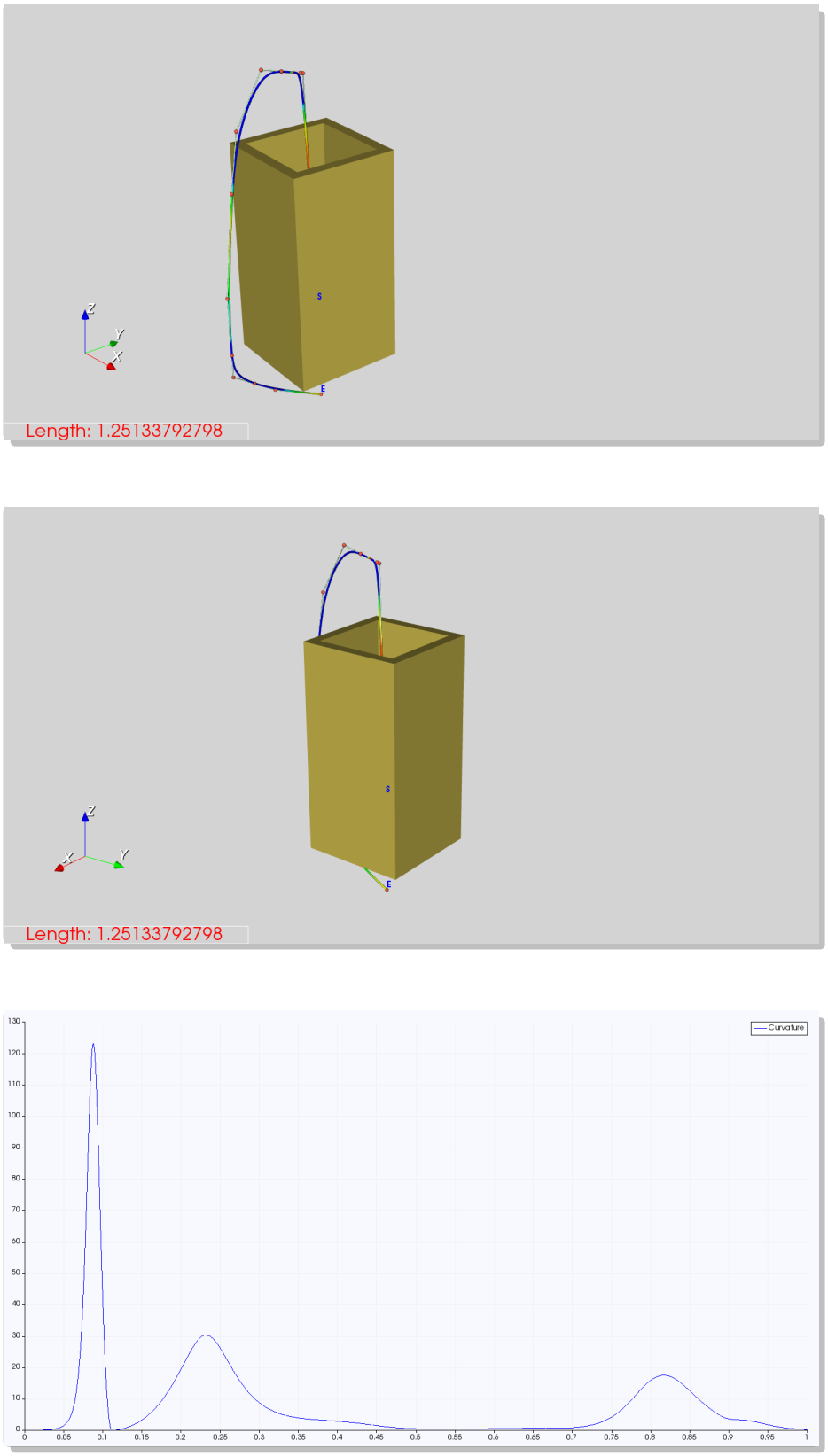
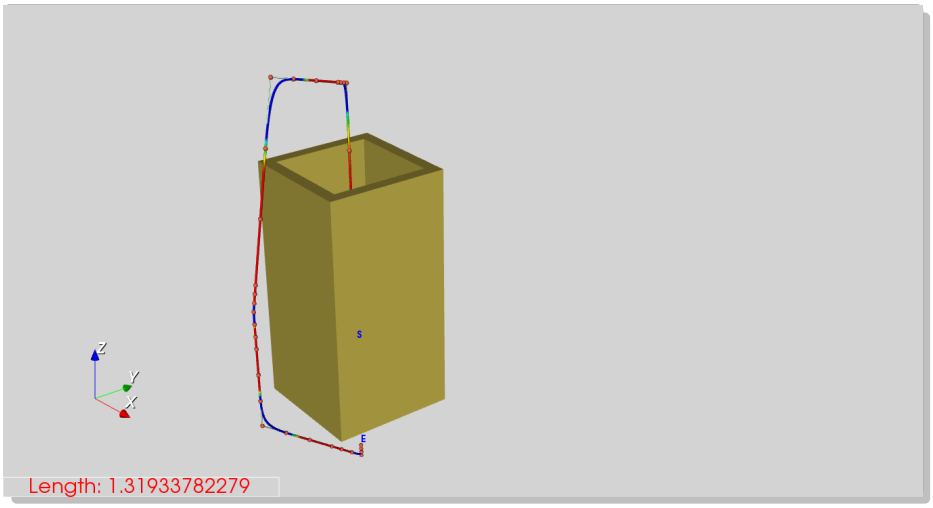
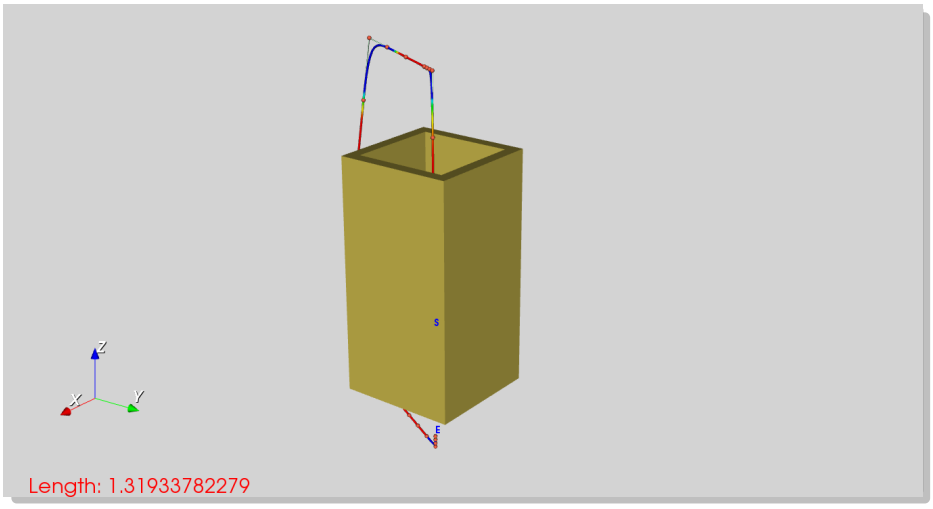


Figure 98.: Test 70; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 4; Meth. B; Post proc. ✓; Part. **U**.





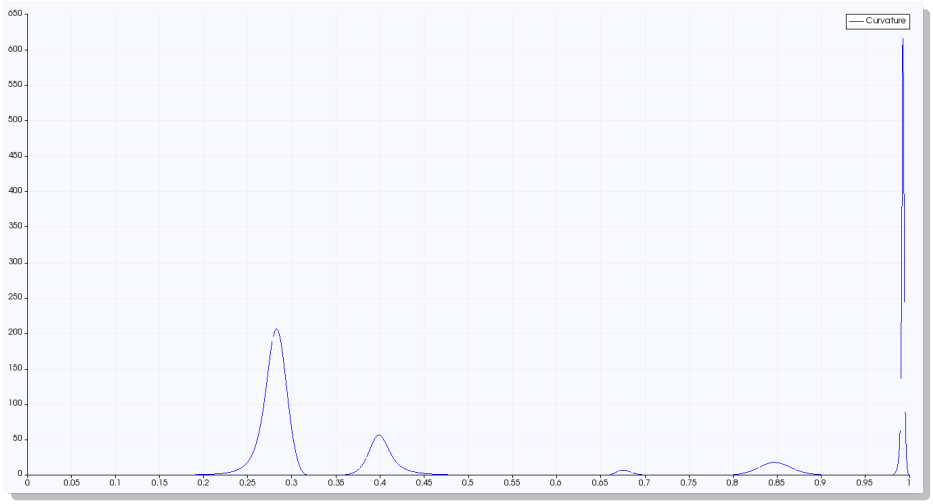


Figure 99.: Test 71; Scene 3; **s** → **e** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 4; Meth. B; Post proc. ✗; Part. **A**.

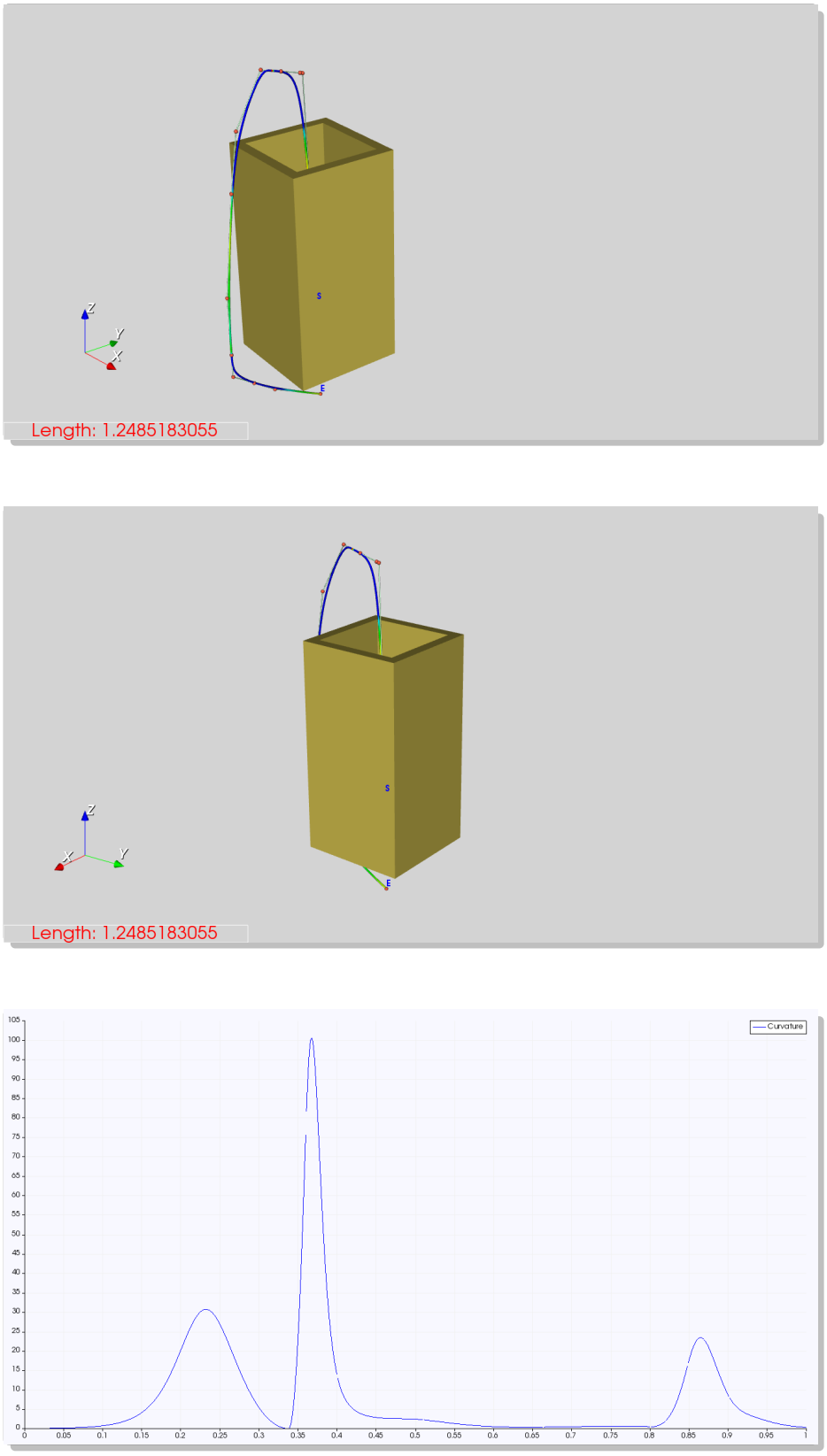
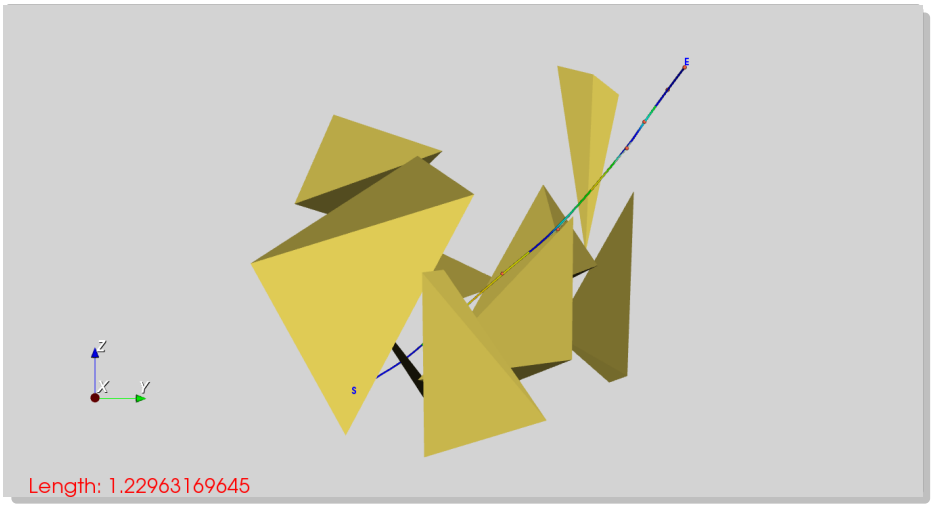
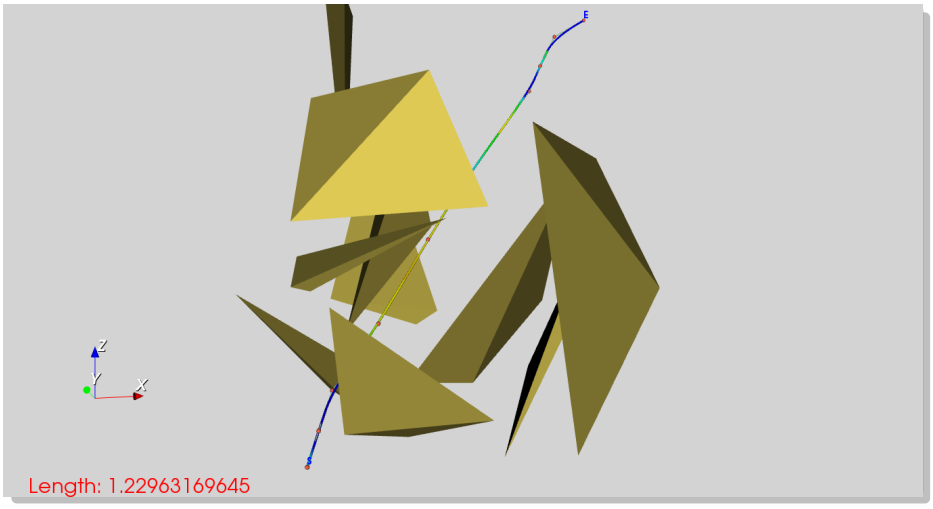


Figure 100.: Test 72; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 4; Meth. B; Post proc. ✓; Part. **A**.





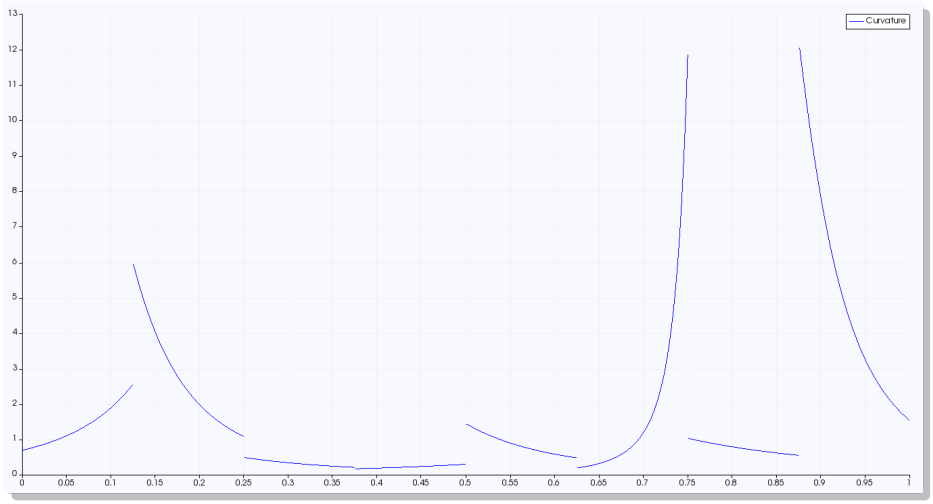


Figure 101.: Test 73; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. C; Part. **U**; Config. 1.

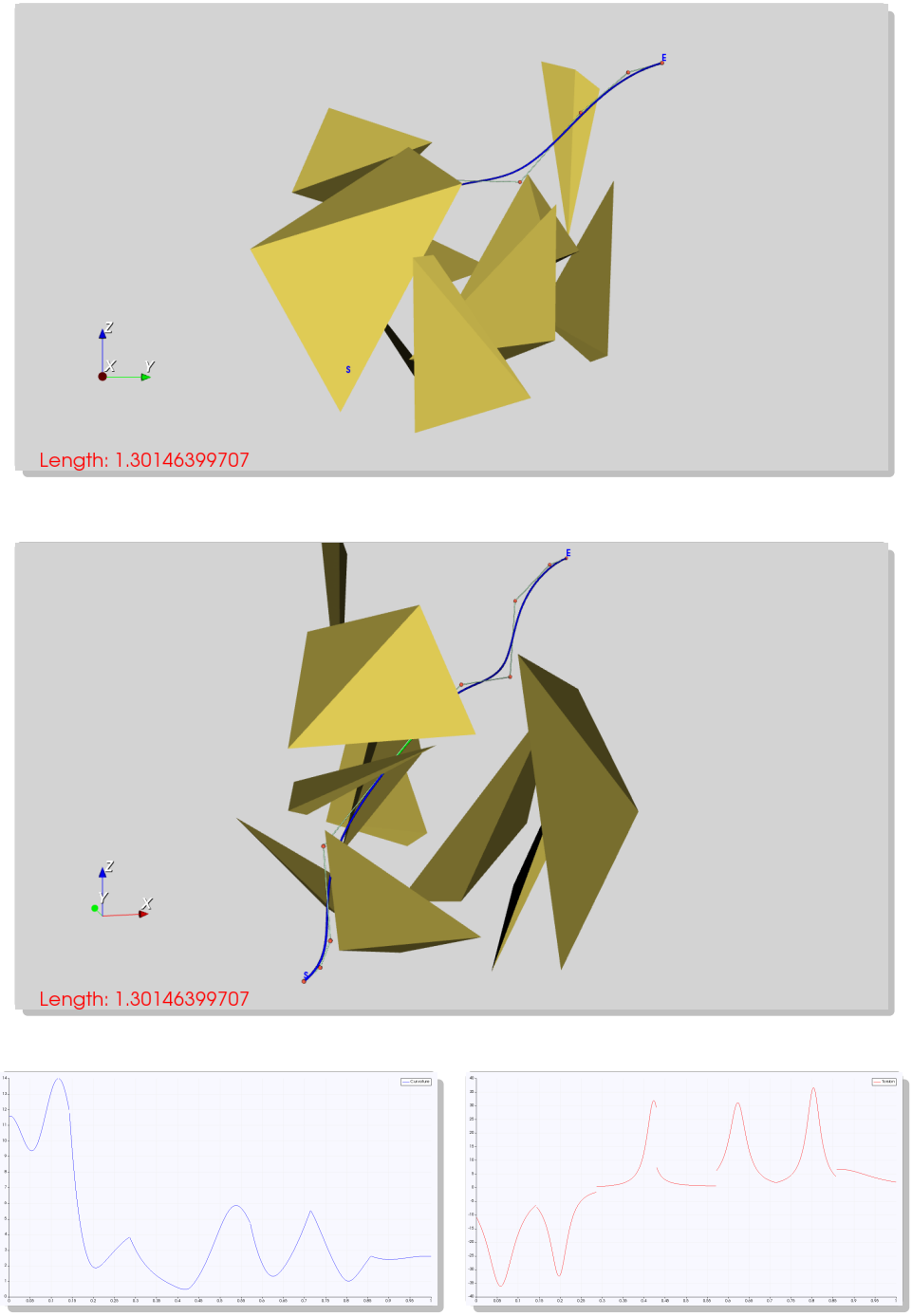


Figure 102.: Test 74; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. C; Part. **U**; Config. 1.

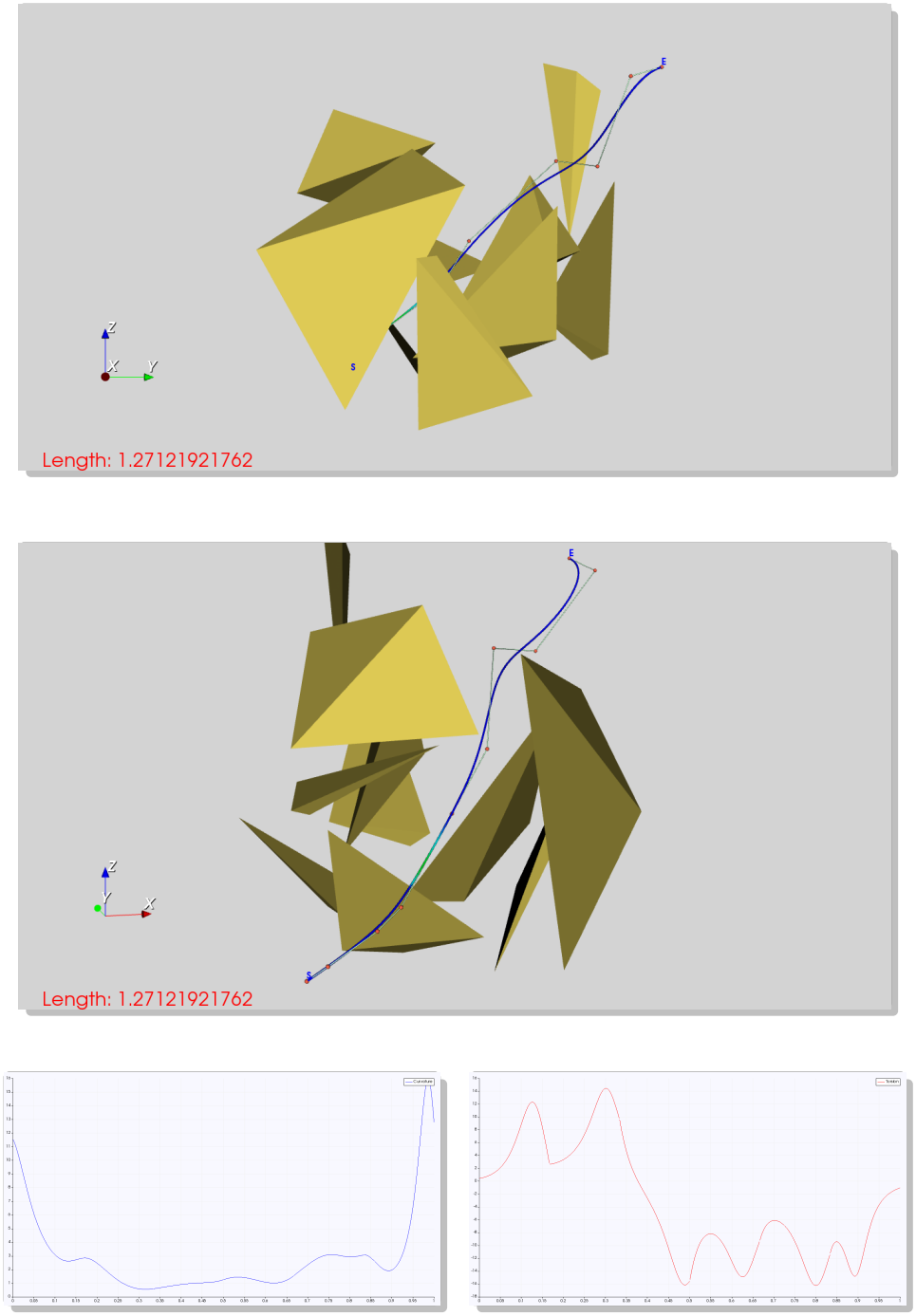


Figure 103.: Test 75; Scene 1; **s** → **e** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. C; Part. **U**; Config. 1.

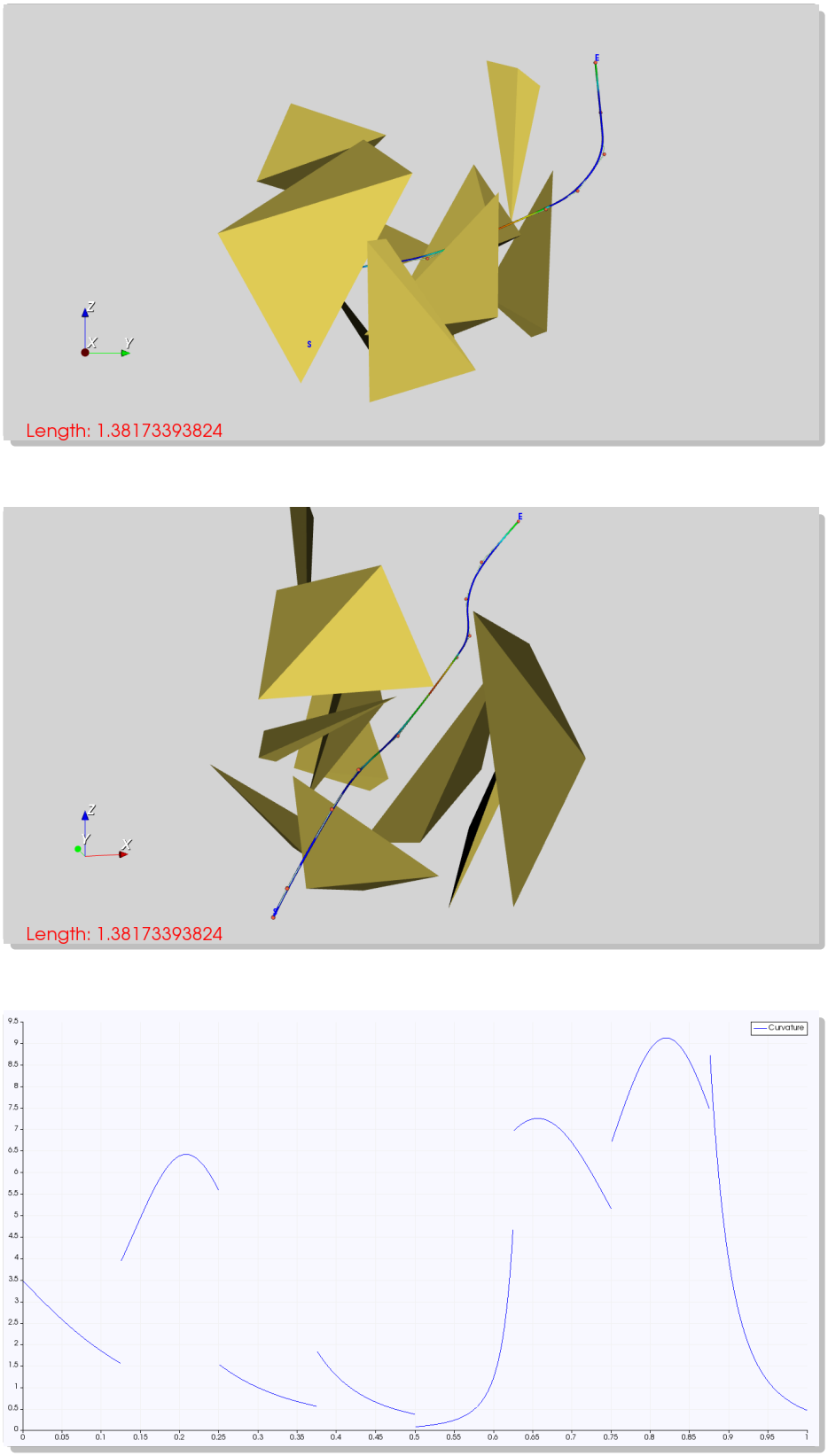


Figure 104.: Test 76; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. C; Part. **U**; Config. 2.

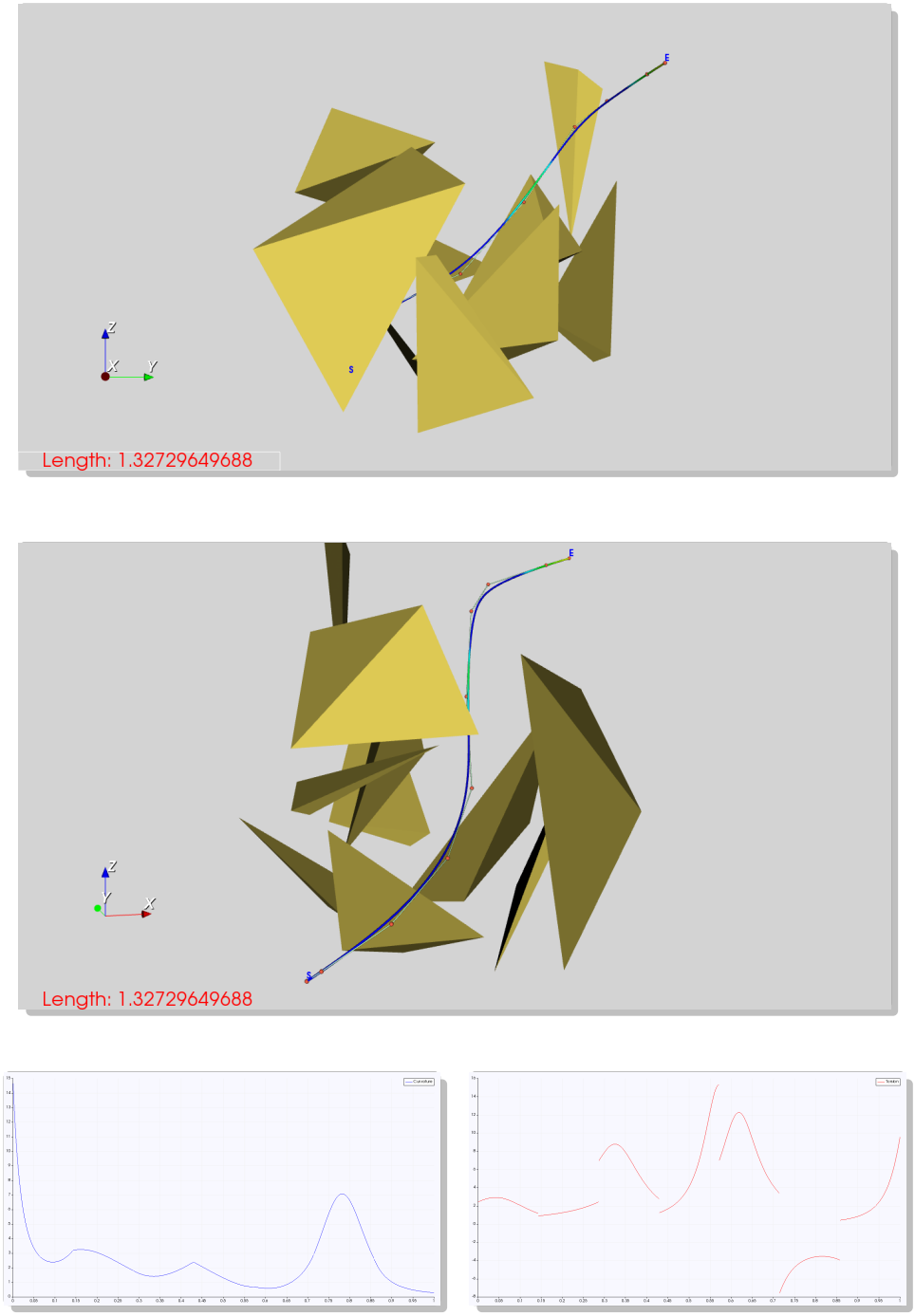


Figure 105.: Test 77; Scene 1; **s** → **e** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. C; Part. **U**; Config. 2.

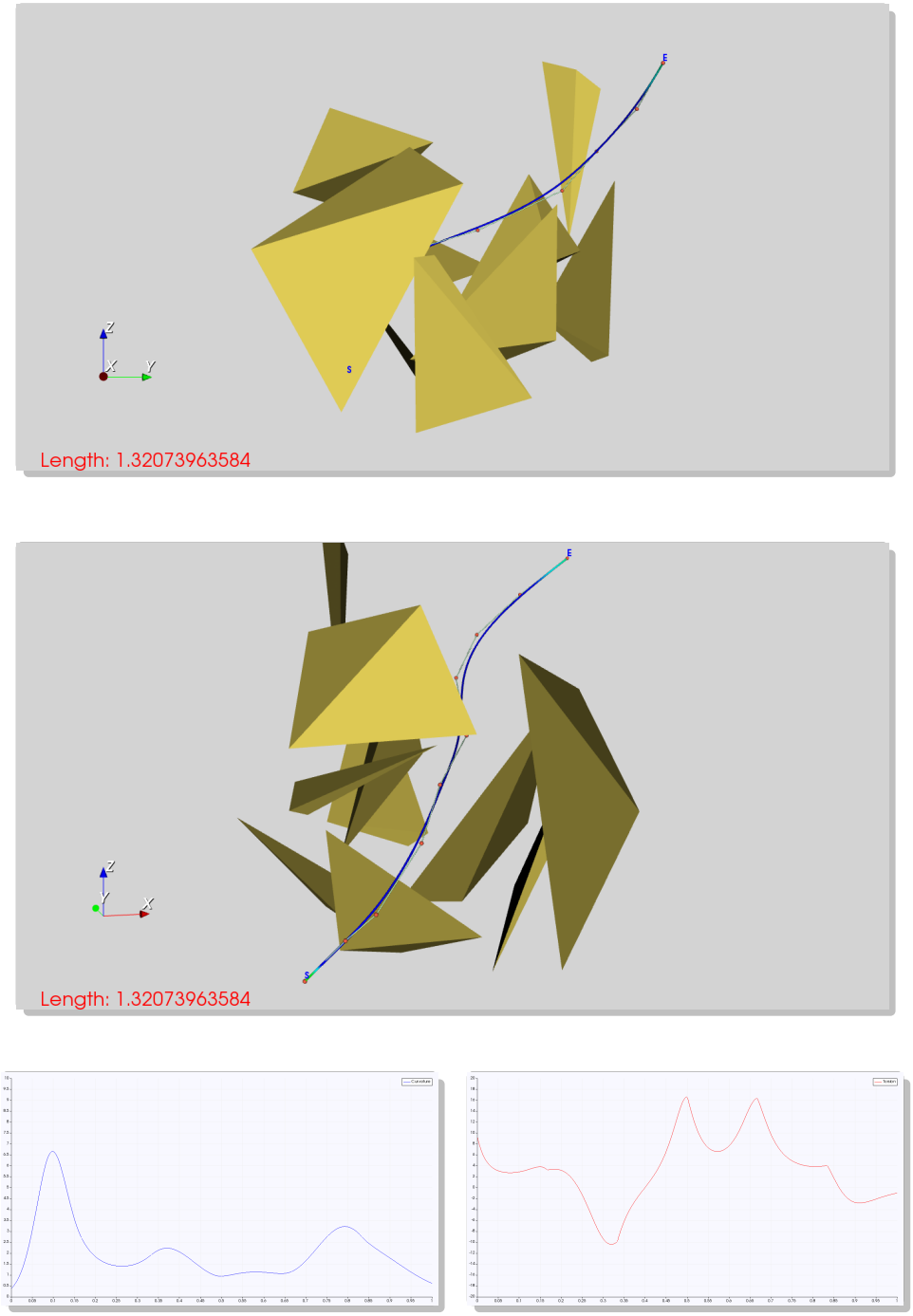


Figure 106.: Test 78; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. C; Part. **U**; Config. 2.

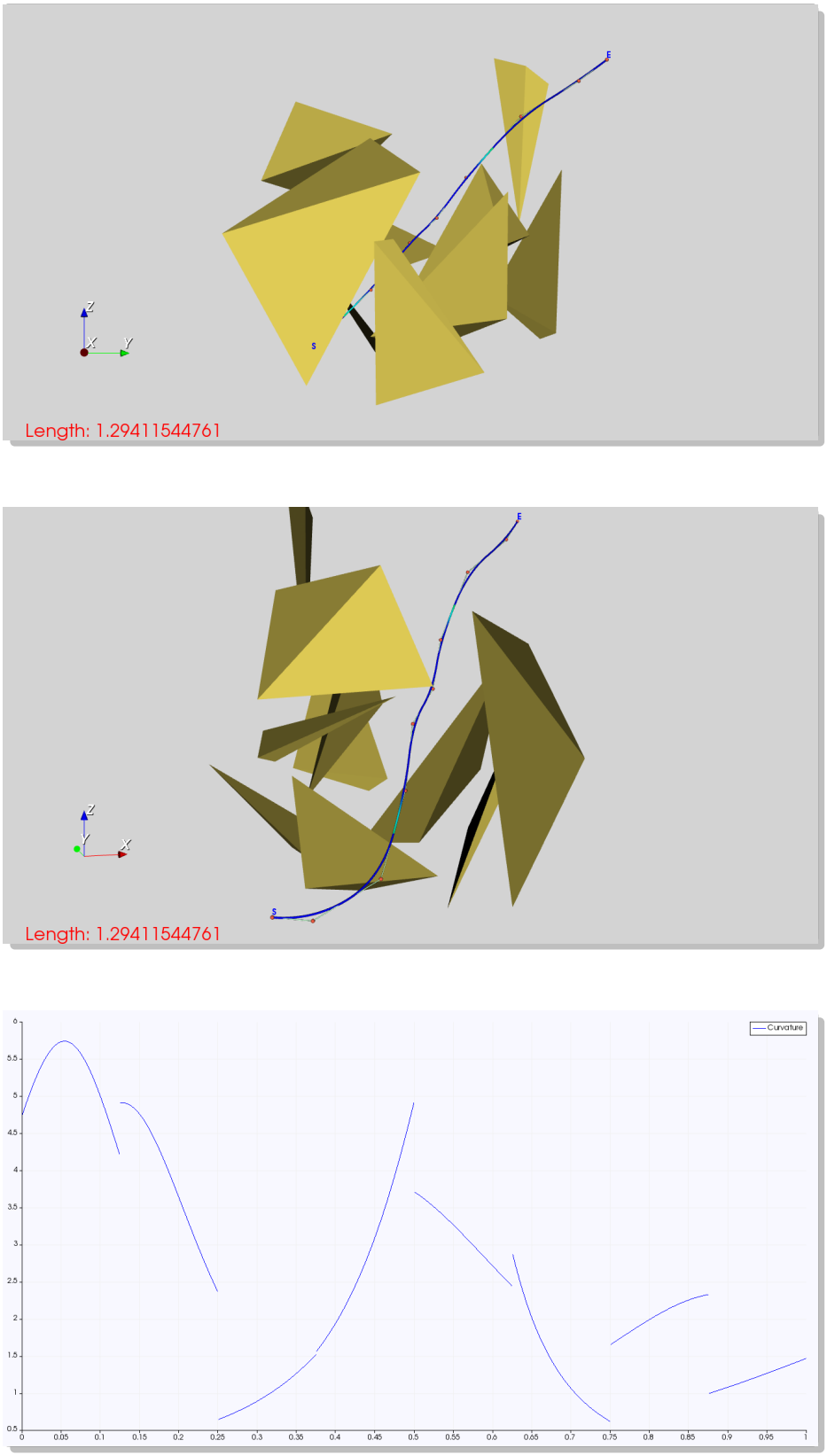


Figure 107.: Test 79; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 2; Meth. C; Part. **U**; Config. 3.

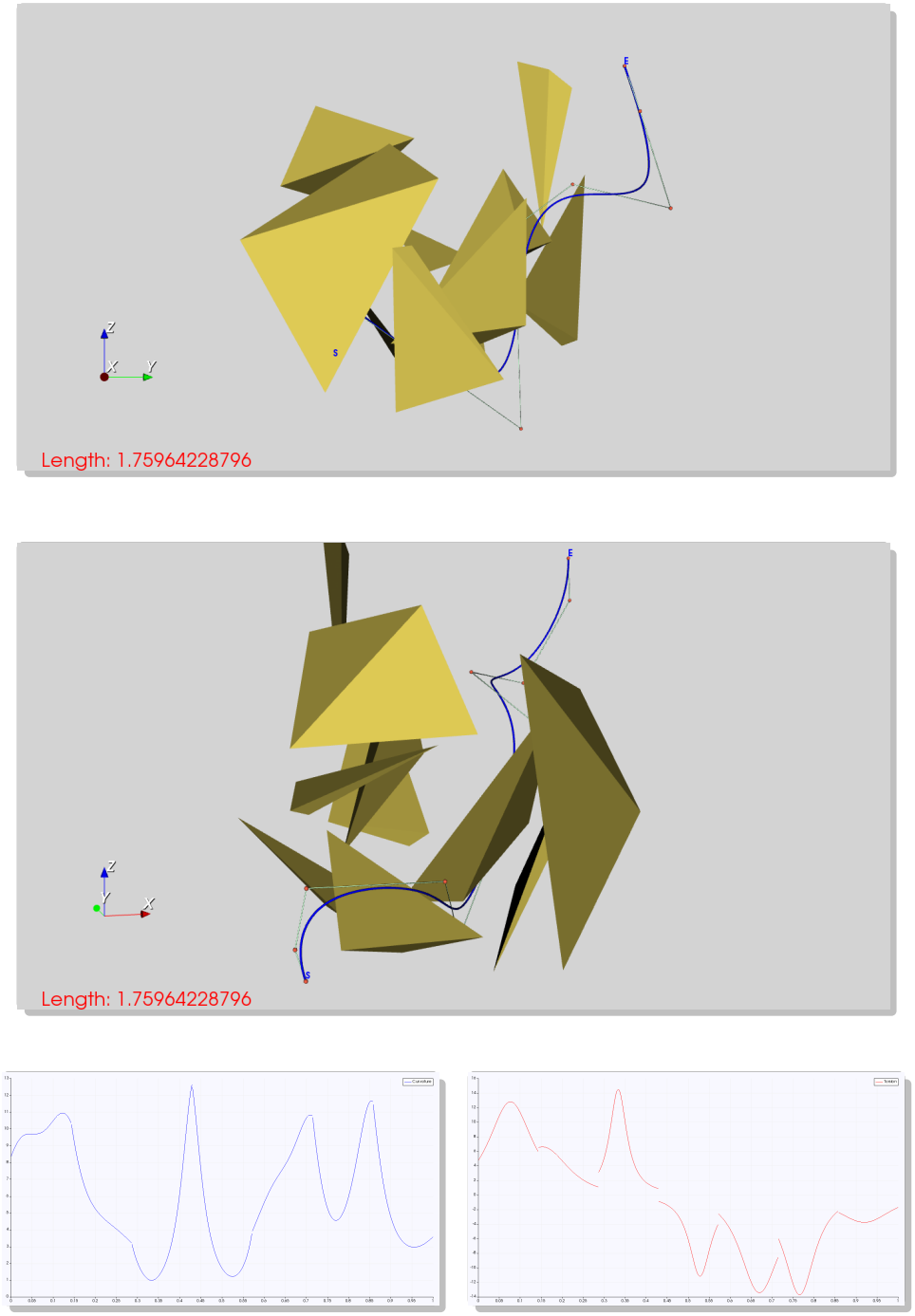


Figure 108.: Test 80; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 3; Meth. C; Part. **U**; Config. 3.

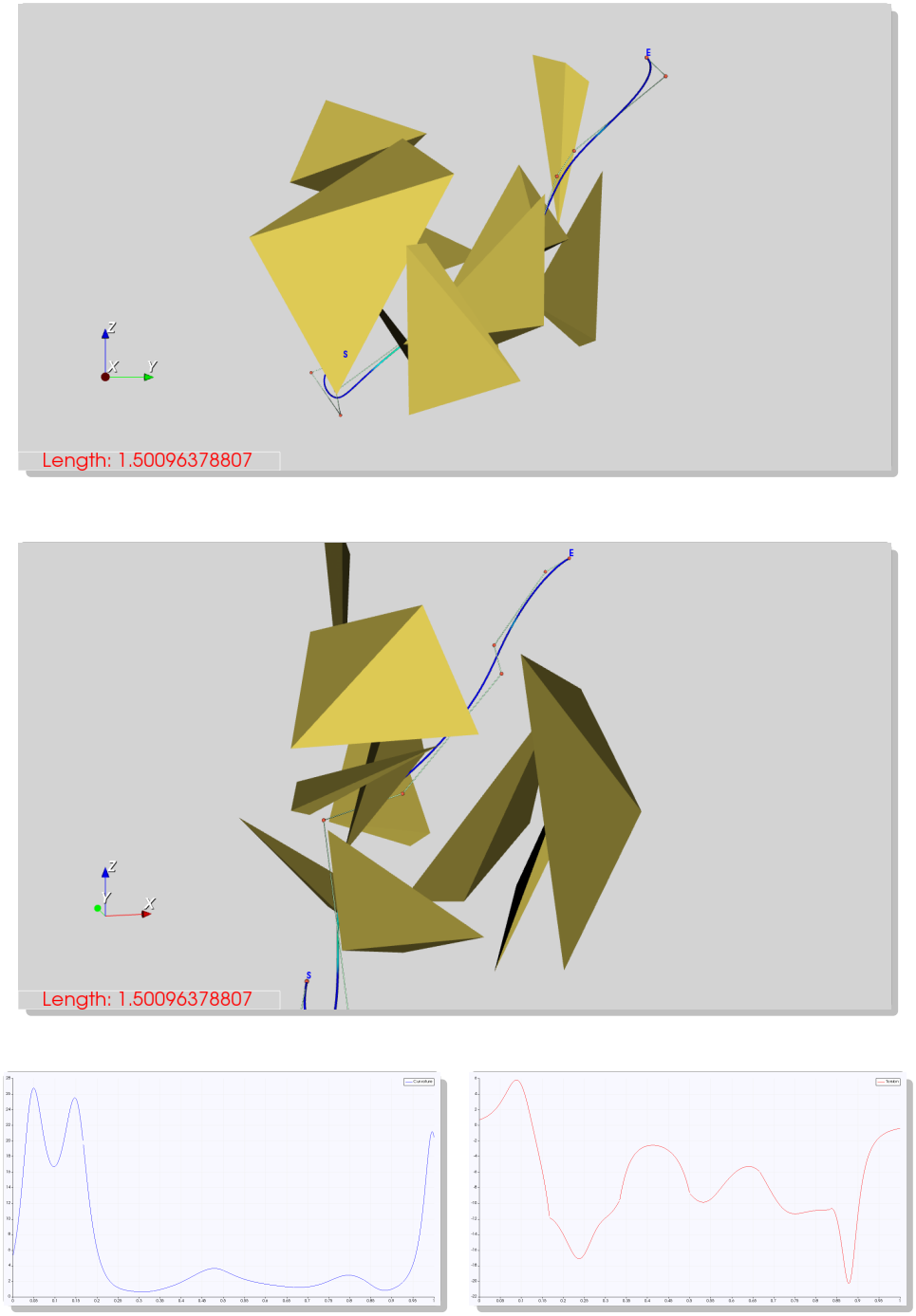


Figure 109.: Test 81; Scene 1; **s** → ***e*** [0.2,0.2,0.2]→[0.9,0.9,0.9]; Deg. 4; Meth. C; Part. **U**; Config. 3.

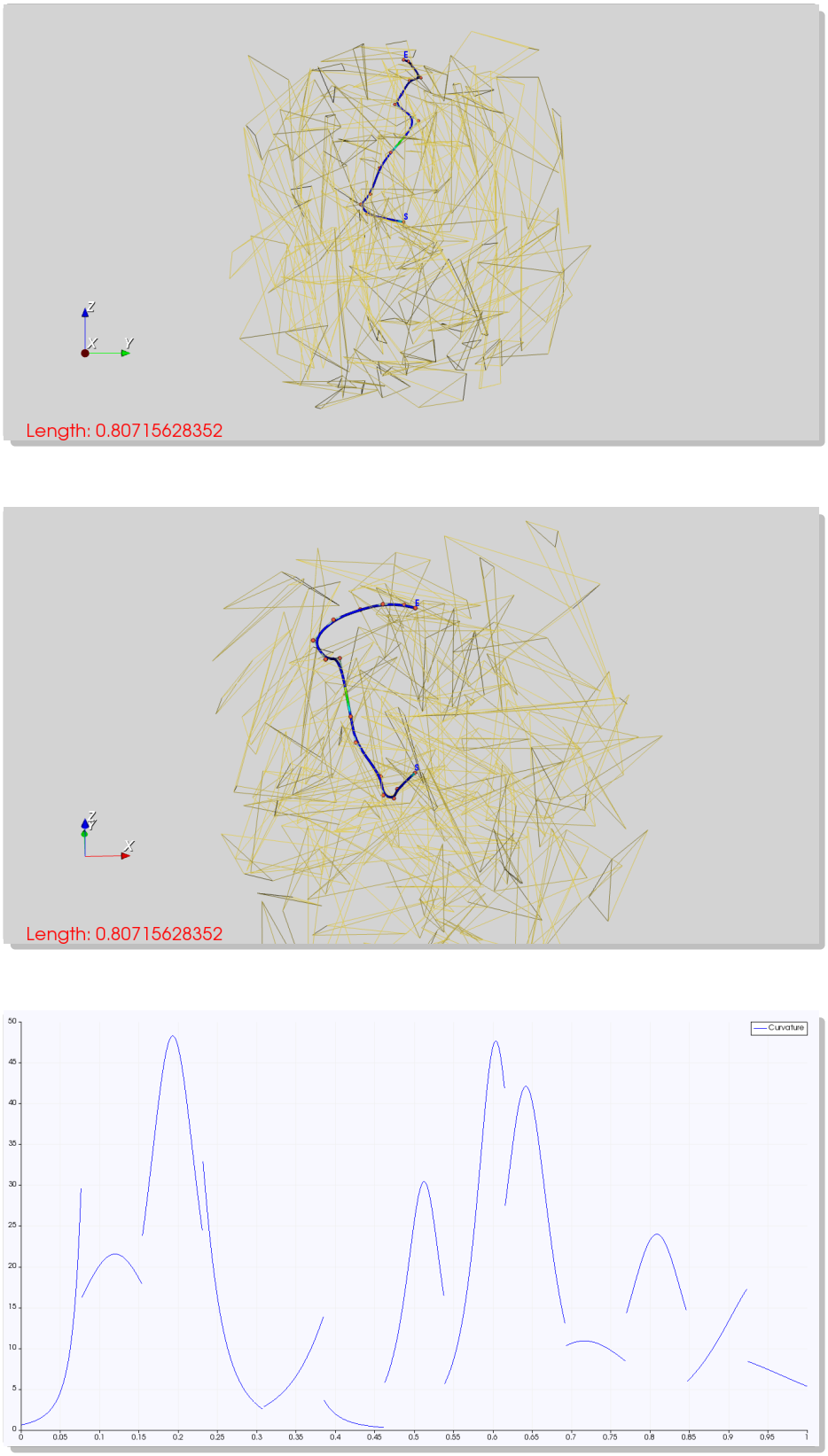


Figure 110.: Test 82; Scene 2; $\mathbf{s} \rightarrow \boldsymbol{e}$ [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 2; Meth. C; Part. **U**; Config. 1.

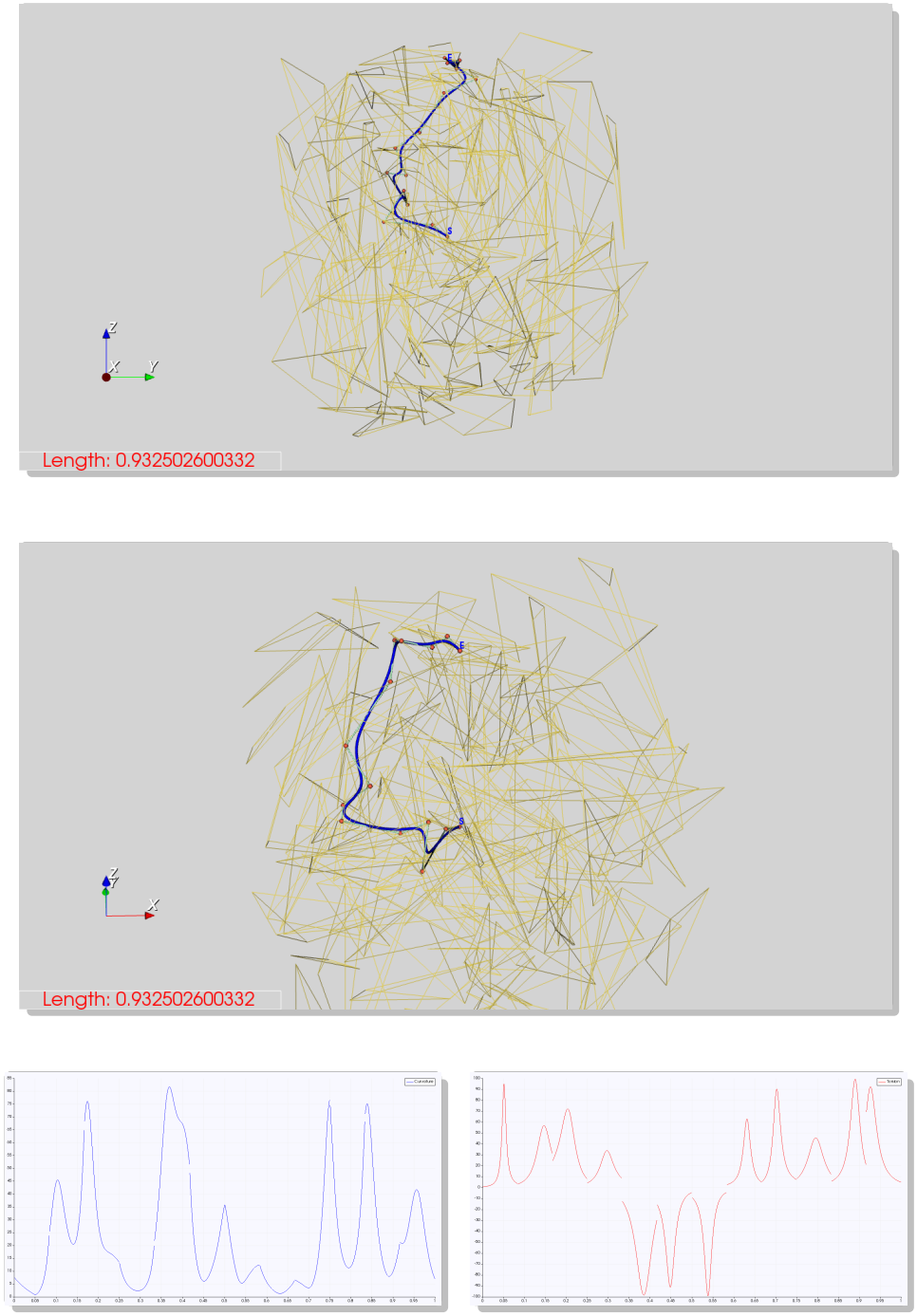


Figure 111.: Test 83; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 3; Meth. C; Part. **U**; Config. 1.

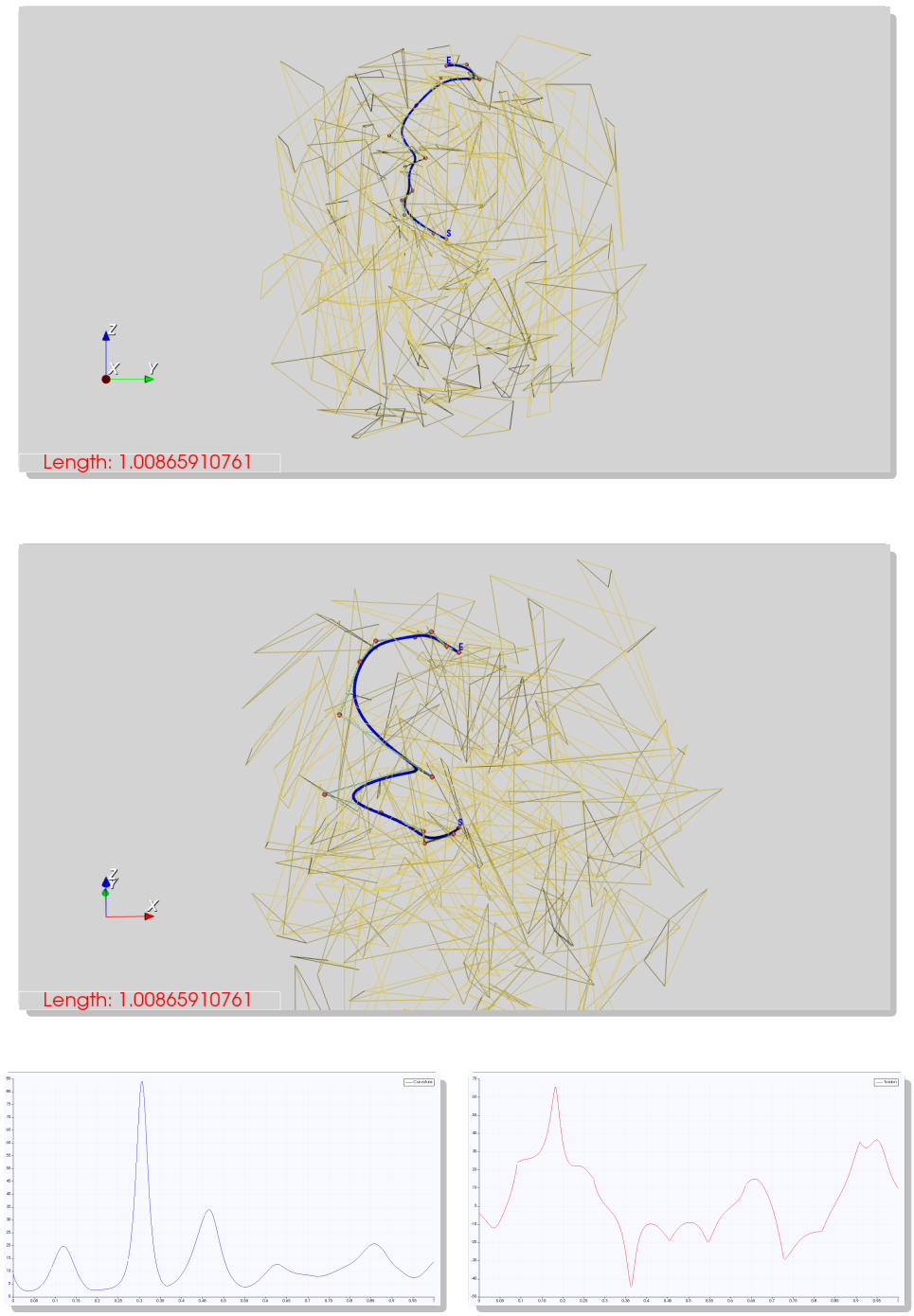


Figure 112.: Test 84; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 4; Meth. C; Part. **U**; Config. 1.

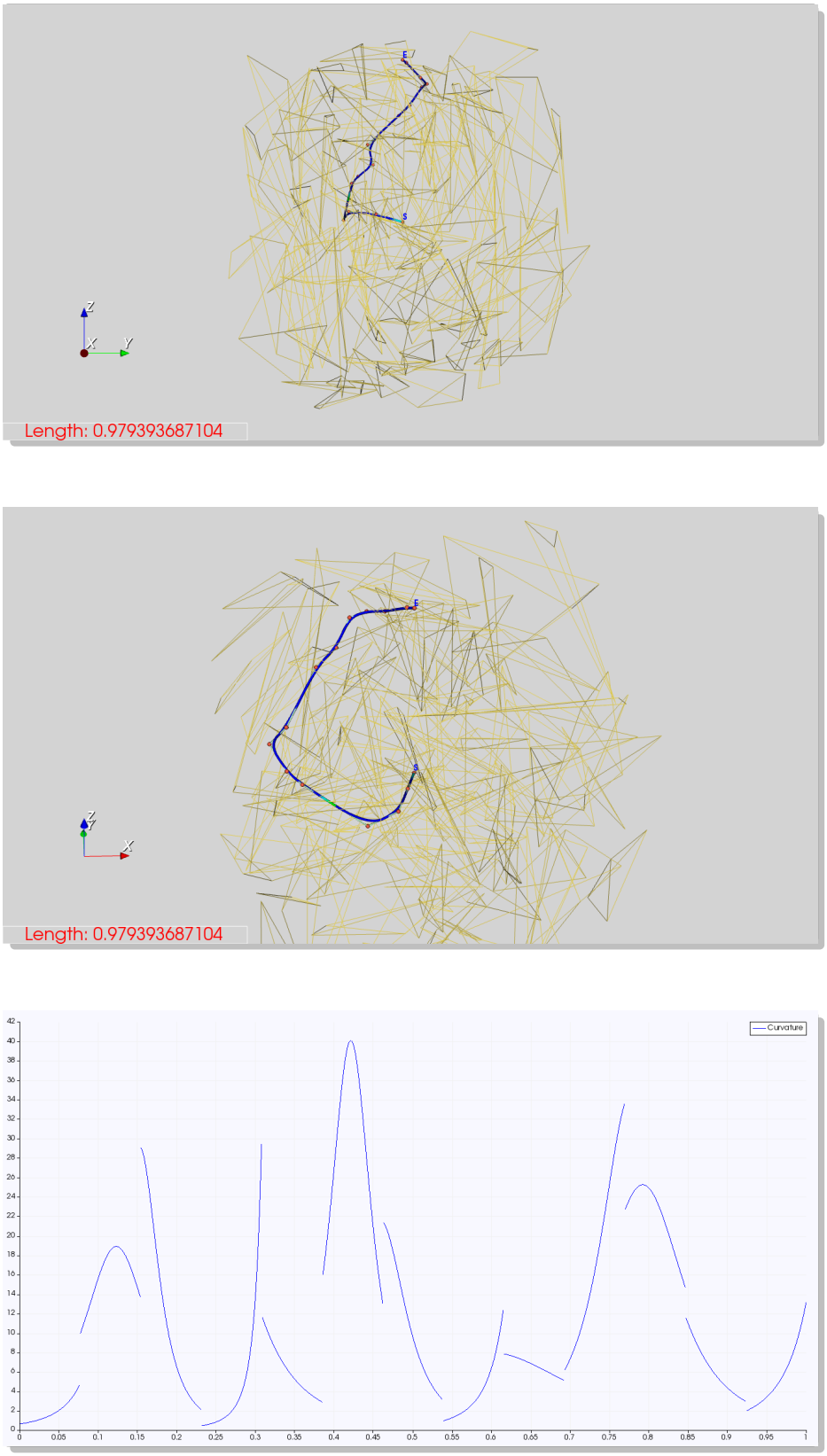


Figure 113.: Test 85; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 2; Meth. C; Part. **U**; Config. 2.

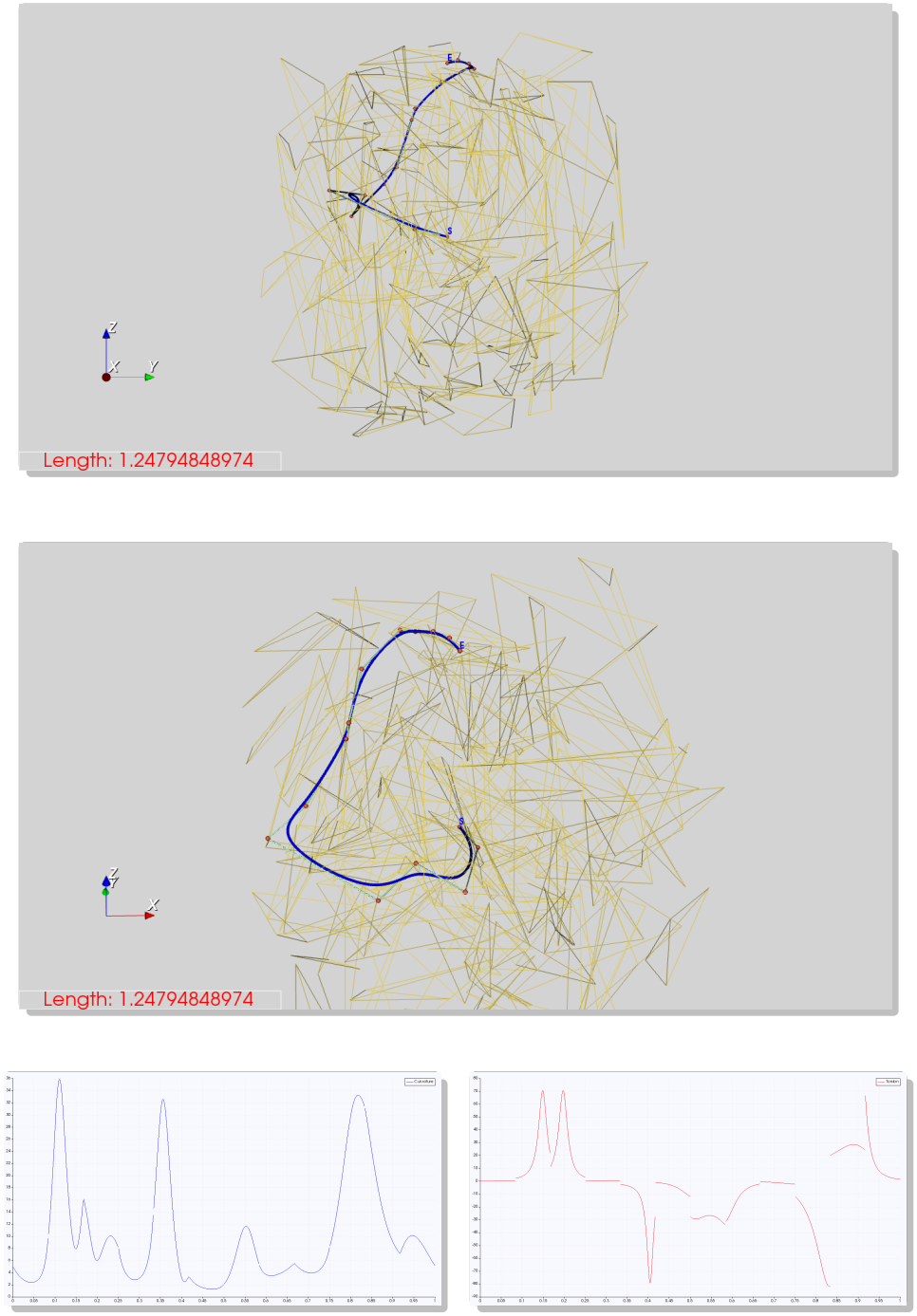


Figure 114.: Test 86; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 3; Meth. C; Part. **U**; Config. 2.

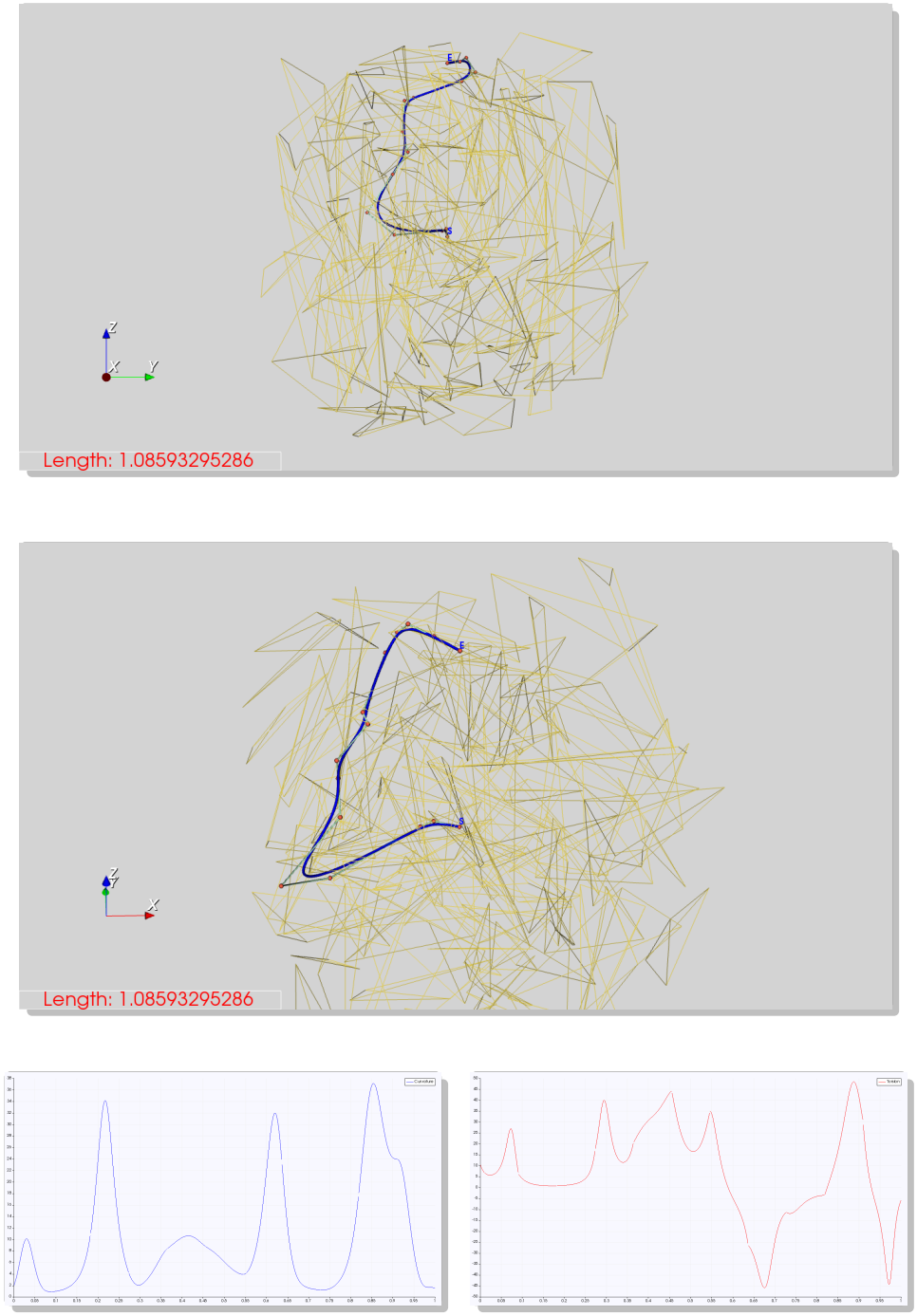


Figure 115.: Test 87; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 4; Meth. C; Part. **U**; Config. 2.

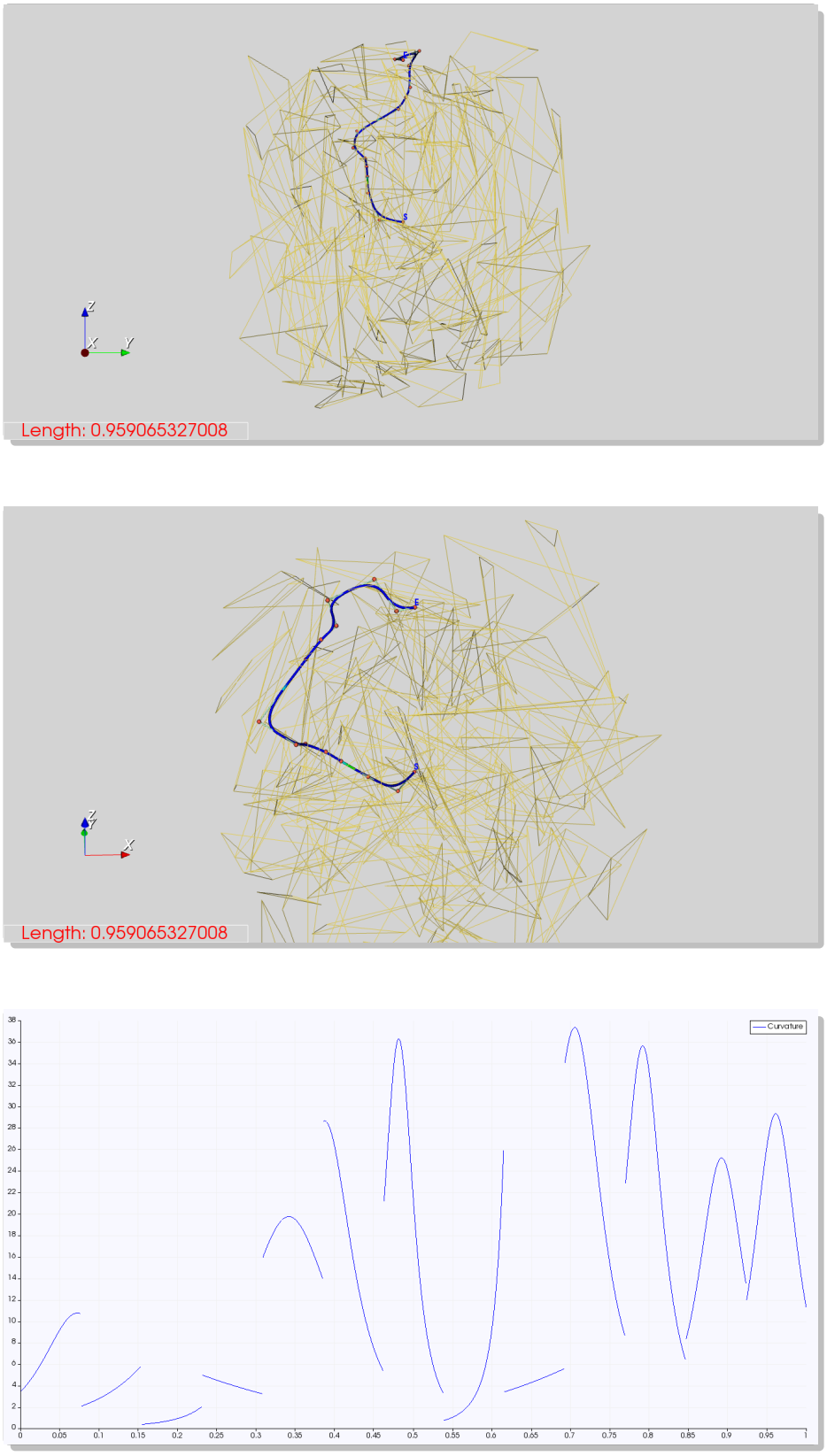


Figure 116.: Test 88; Scene 2; **s** → **e** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 2; Meth. C; Part. **U**; Config. 3.

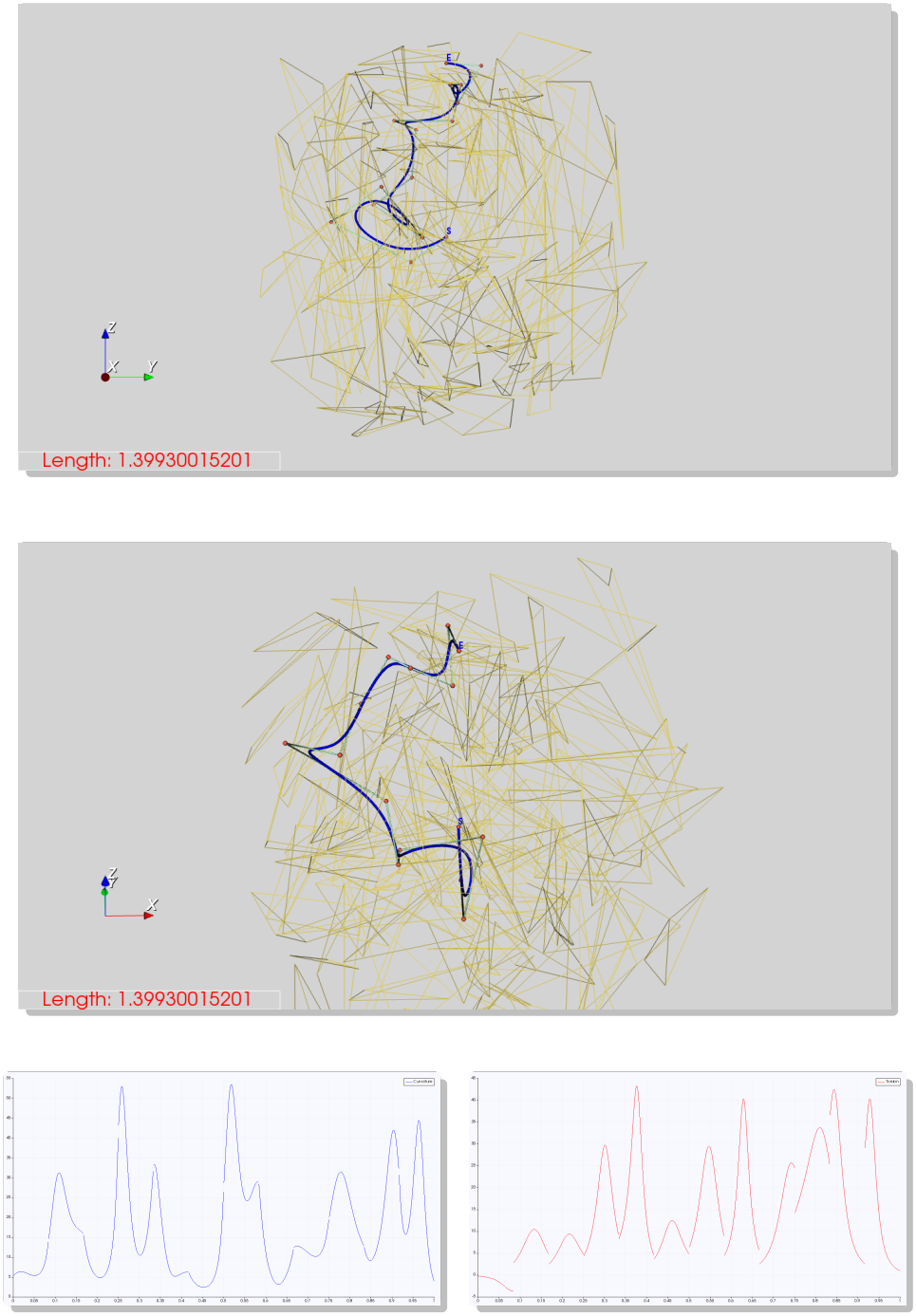


Figure 117.: Test 89; Scene 2; $\mathbf{s} \rightarrow \boldsymbol{e}$ [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 3; Meth. C; Part. **U**; Config. 3.

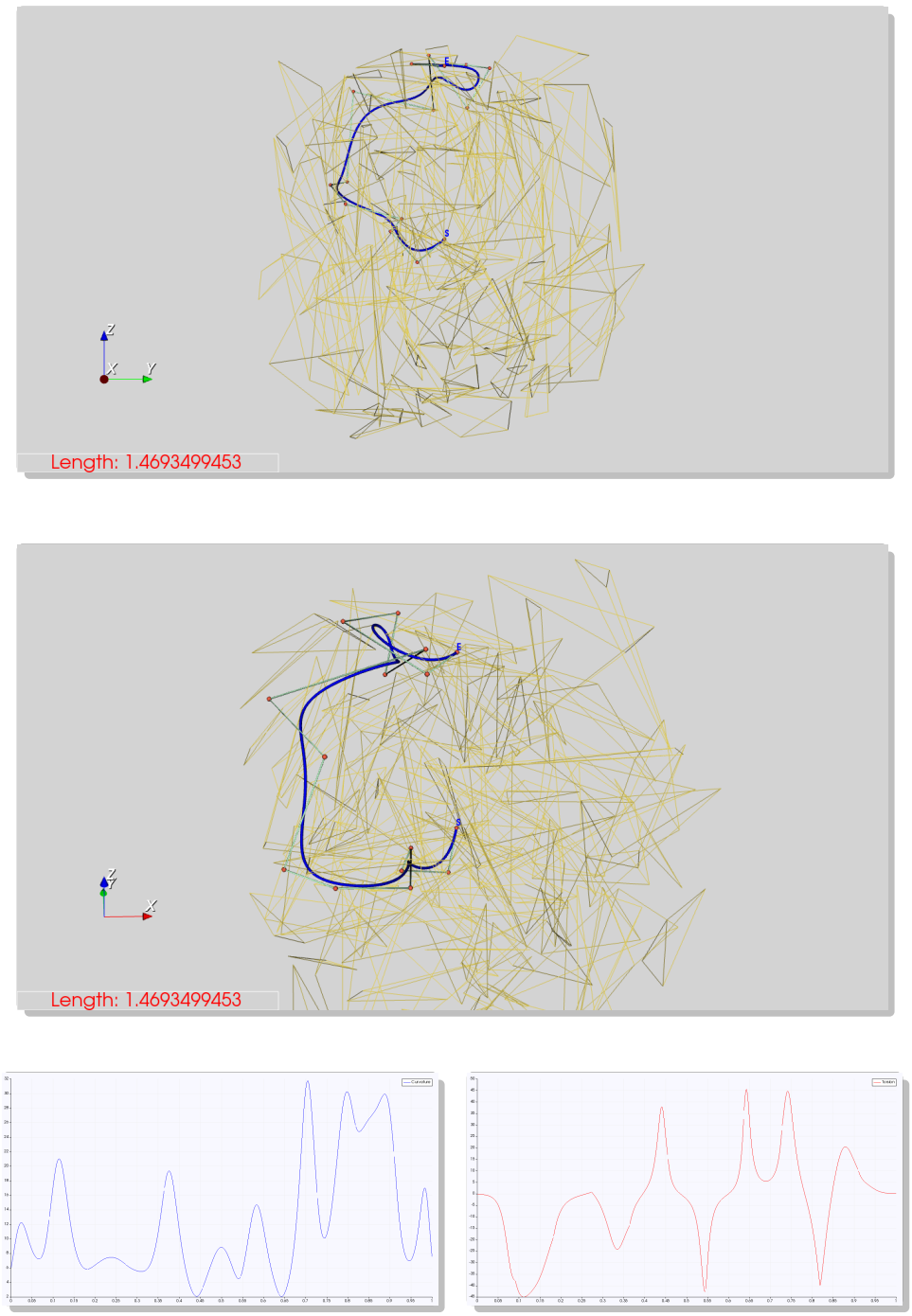


Figure 118.: Test 90; Scene 2; **s** → ***e*** [0.5,0.5,0.5]→[0.5,0.5,0.95]; Deg. 4; Meth. C; Part. **U**; Config. 3.

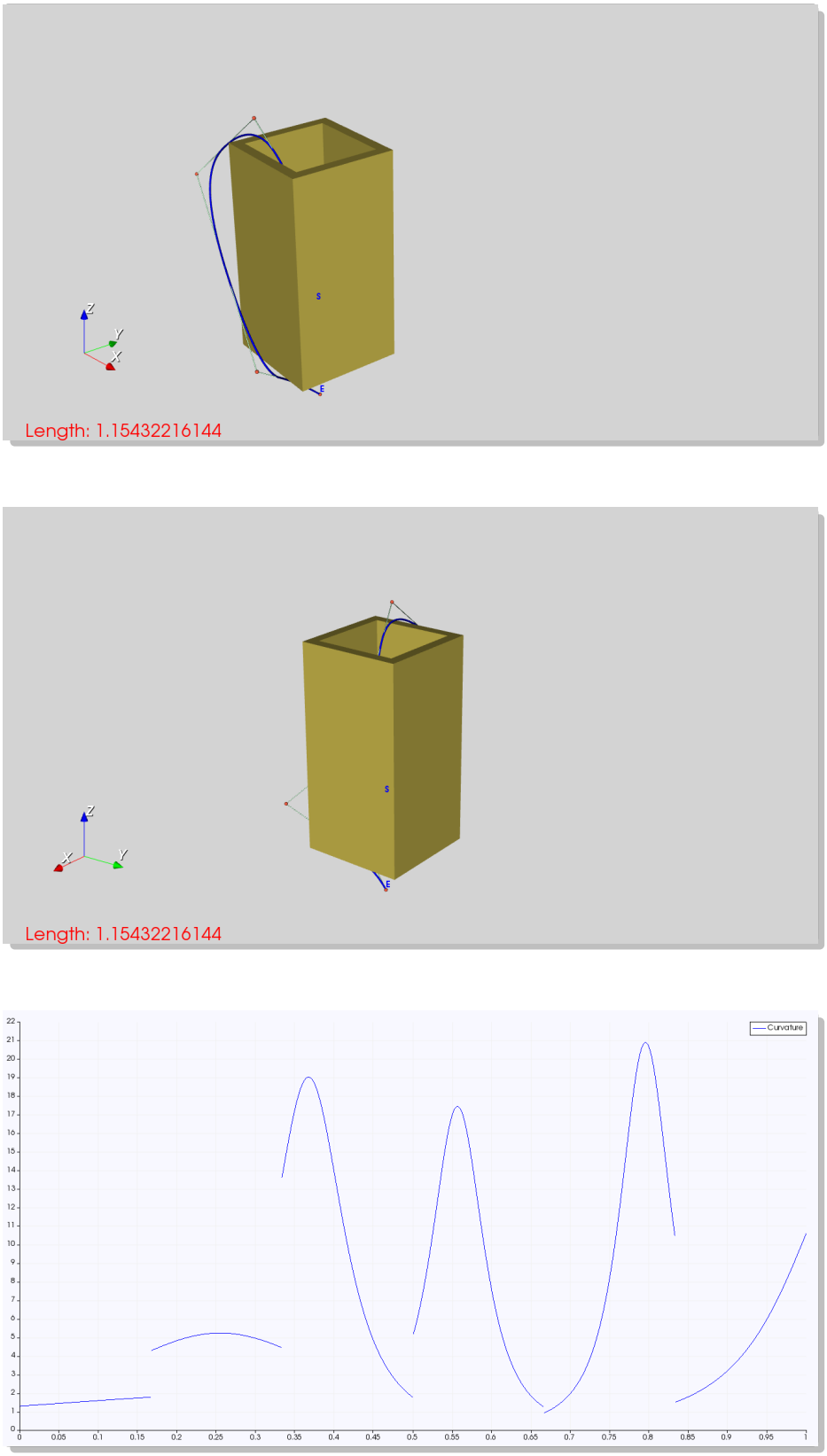


Figure 119.: Test 91; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 2; Meth. C; Part. **U**; Config. 1.

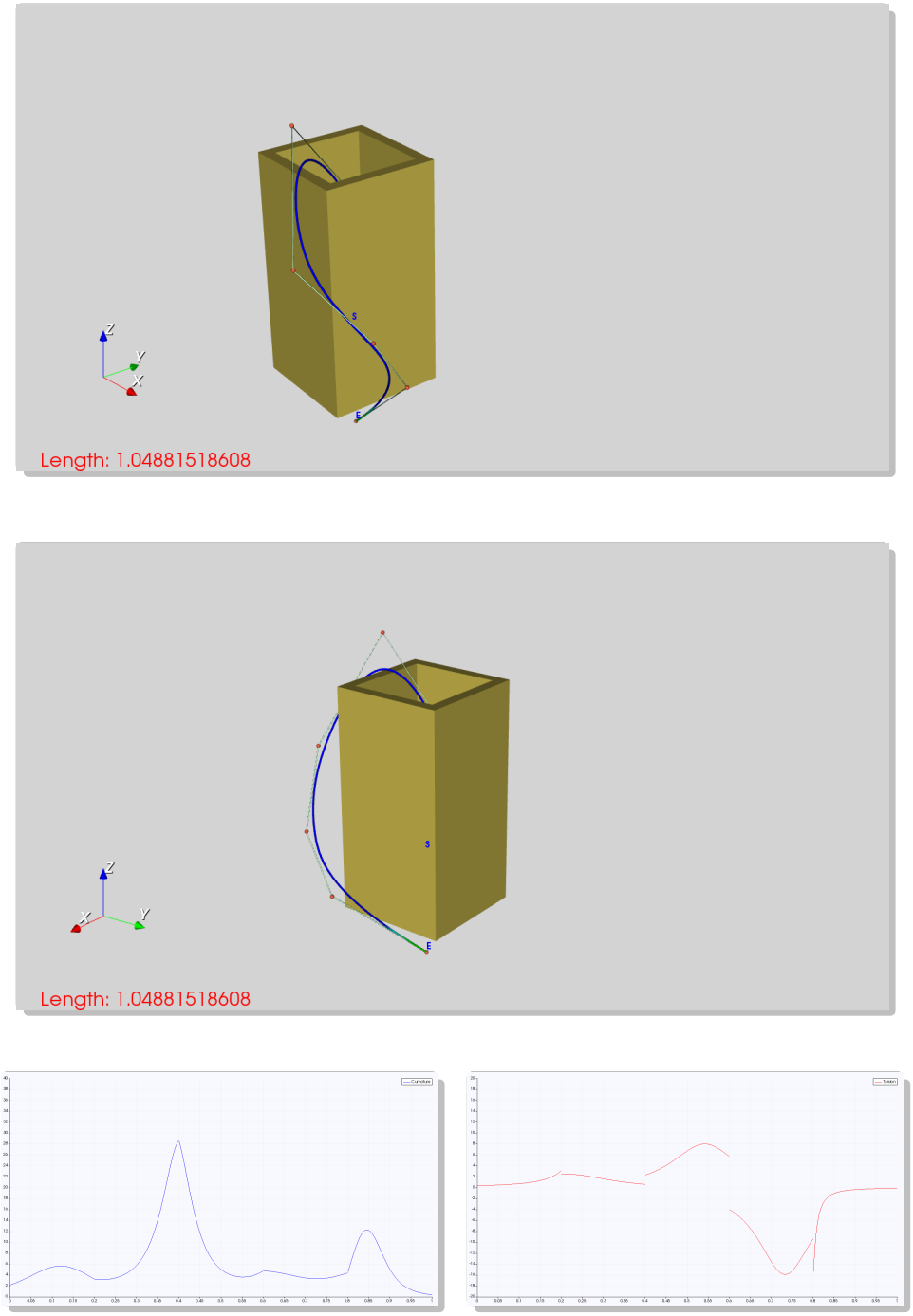


Figure 120.: Test 92; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 3; Meth. C; Part. **U**; Config. 1.

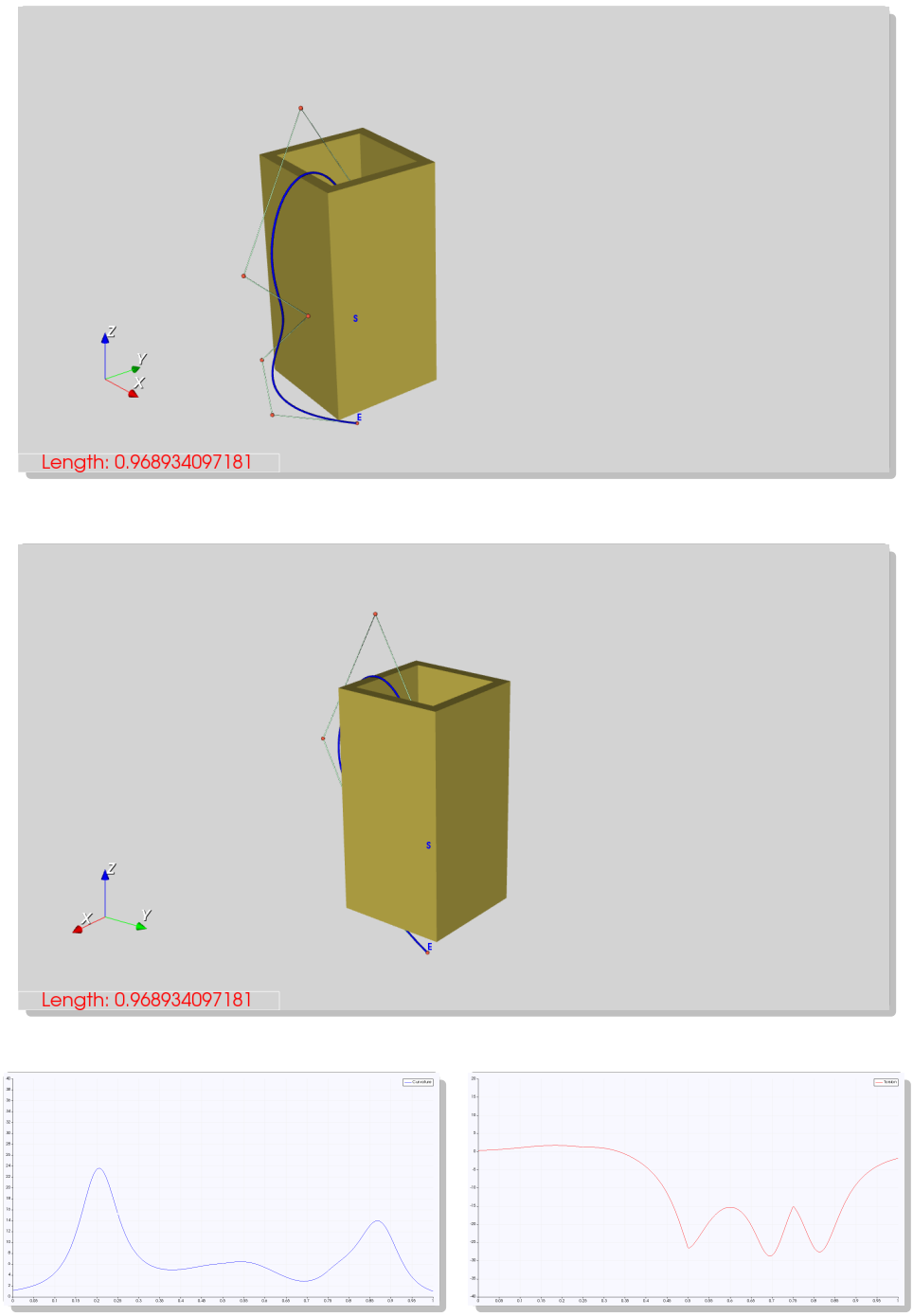


Figure 121.: Test 93; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 4; Meth. C; Part. **U**; Config. 1.

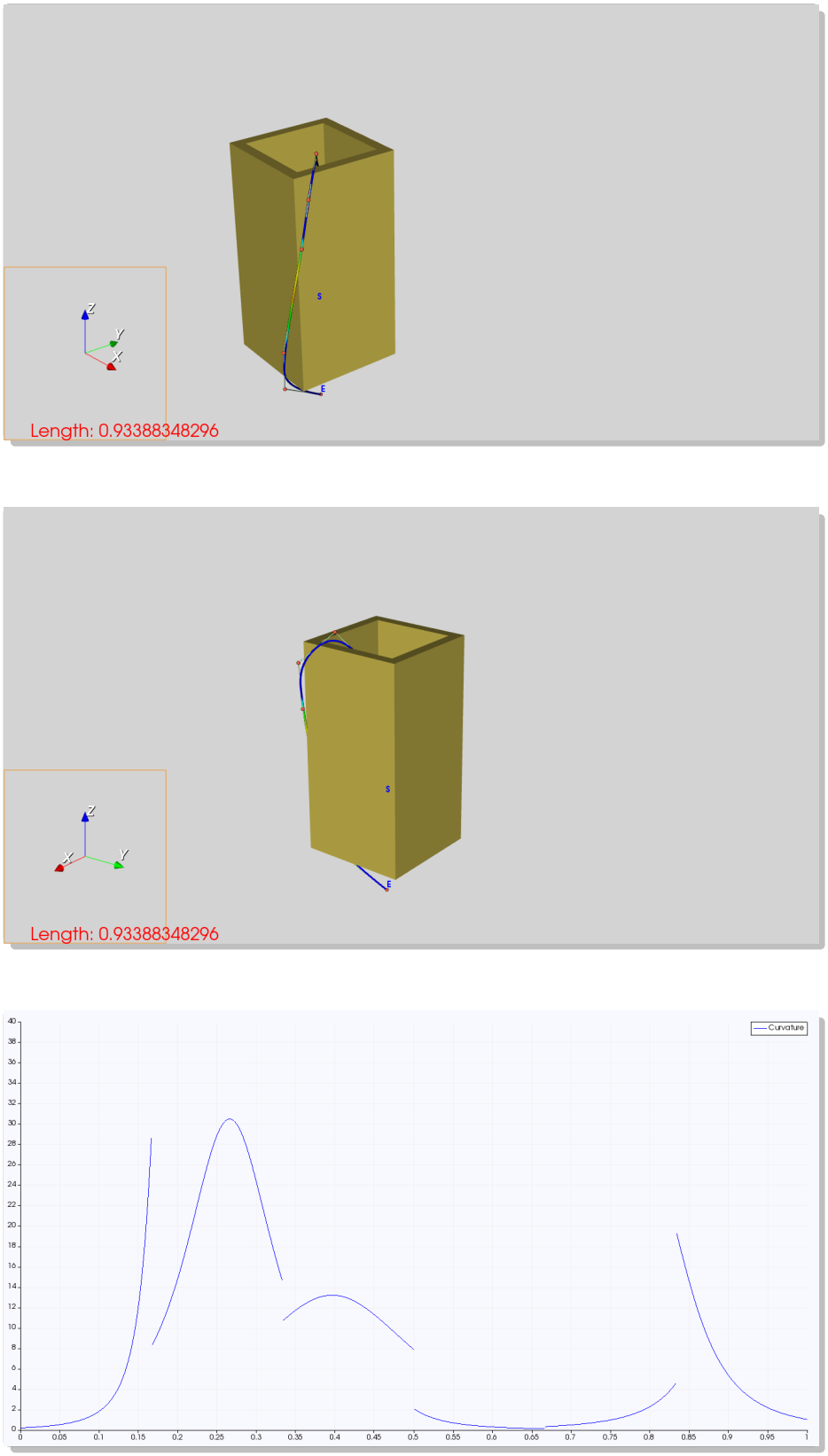


Figure 122.: Test 94; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 2; Meth. C; Part. **U**; Config. 2b.

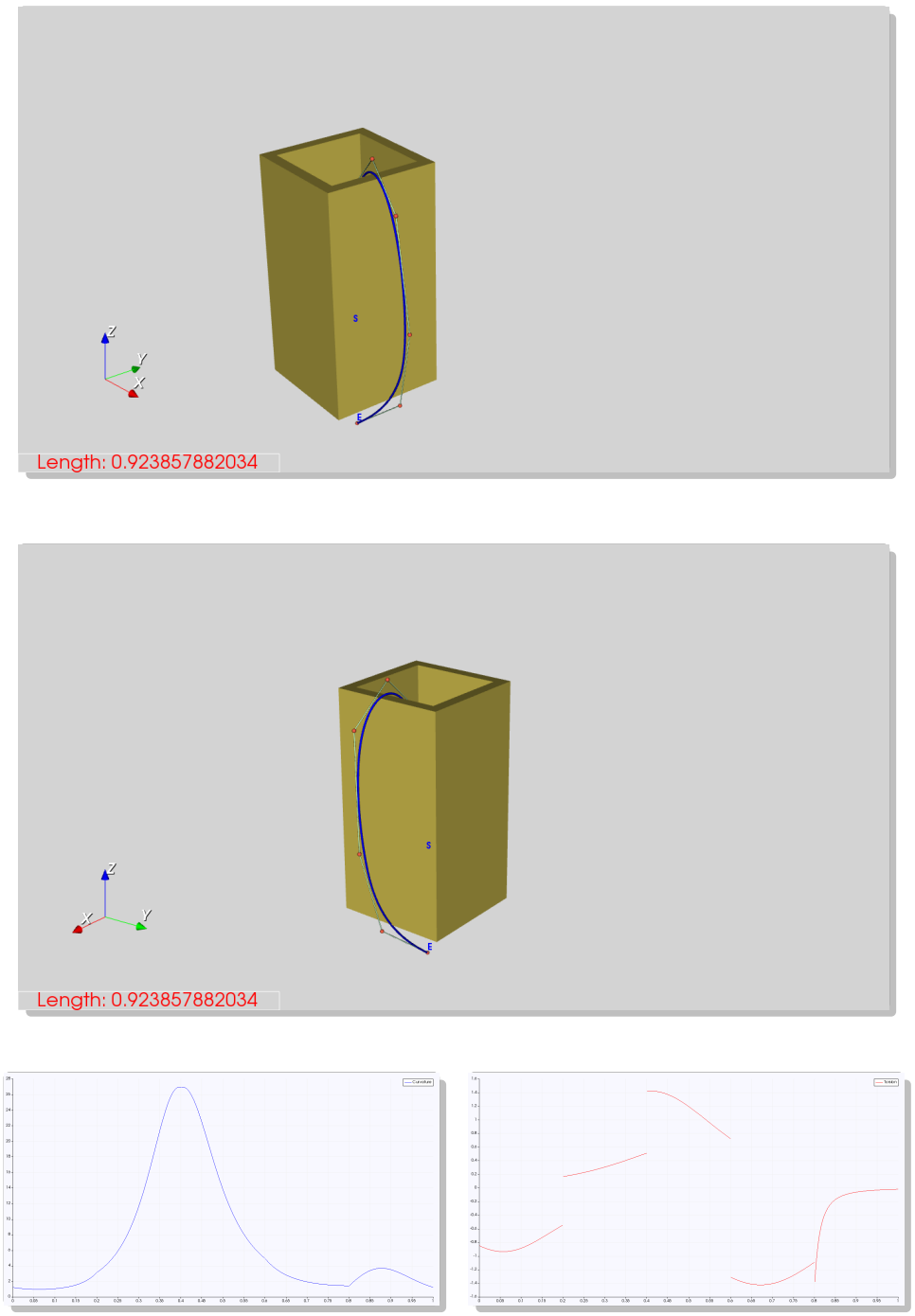


Figure 123.: Test 95; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 3; Meth. C; Part. **U**; Config. 2b.

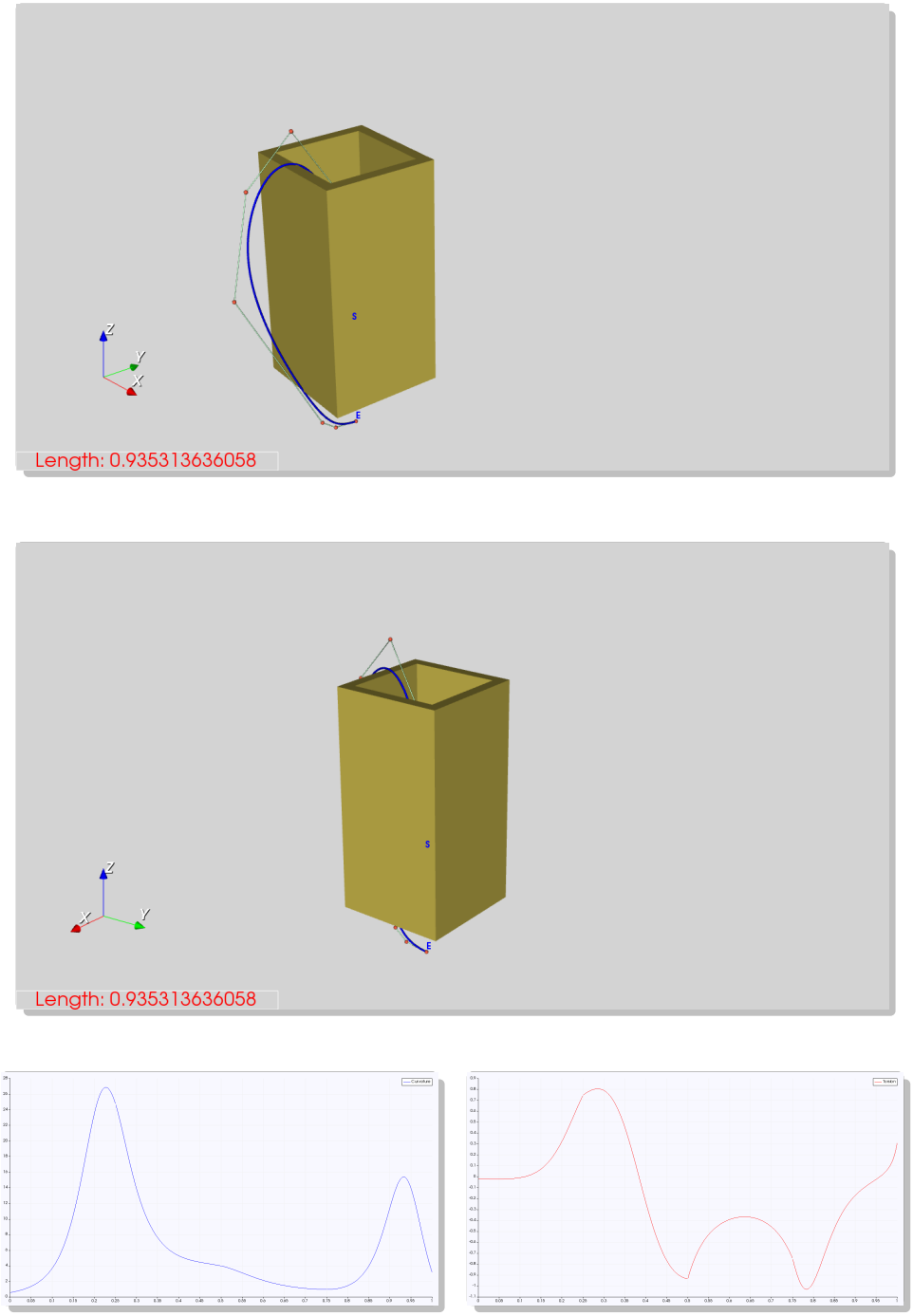


Figure 124.: Test 96; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 4; Meth. C; Part. **U**; Config. 2b.

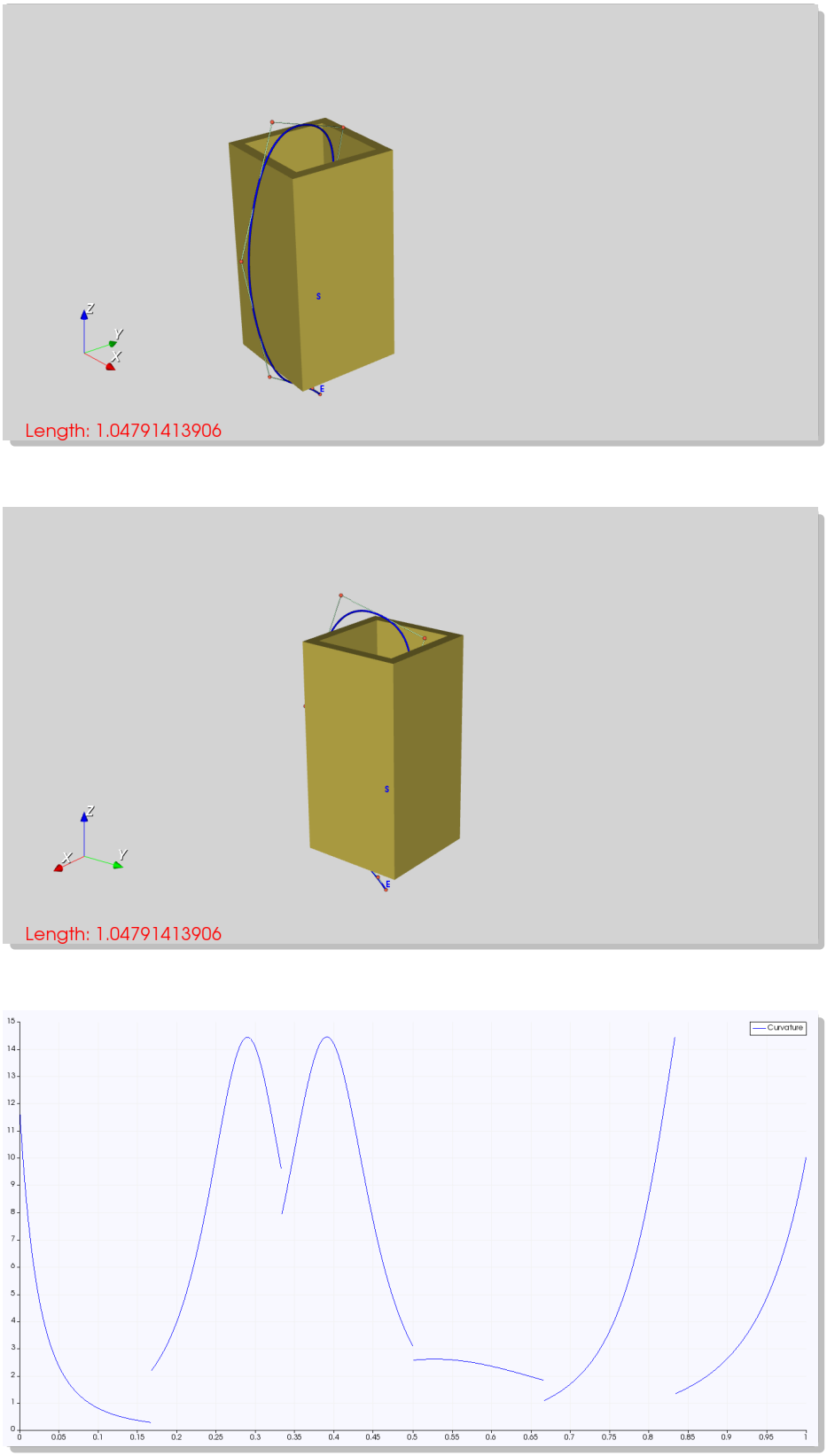


Figure 125.: Test 97; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 2; Meth. C; Part. **U**; Config. 3b.

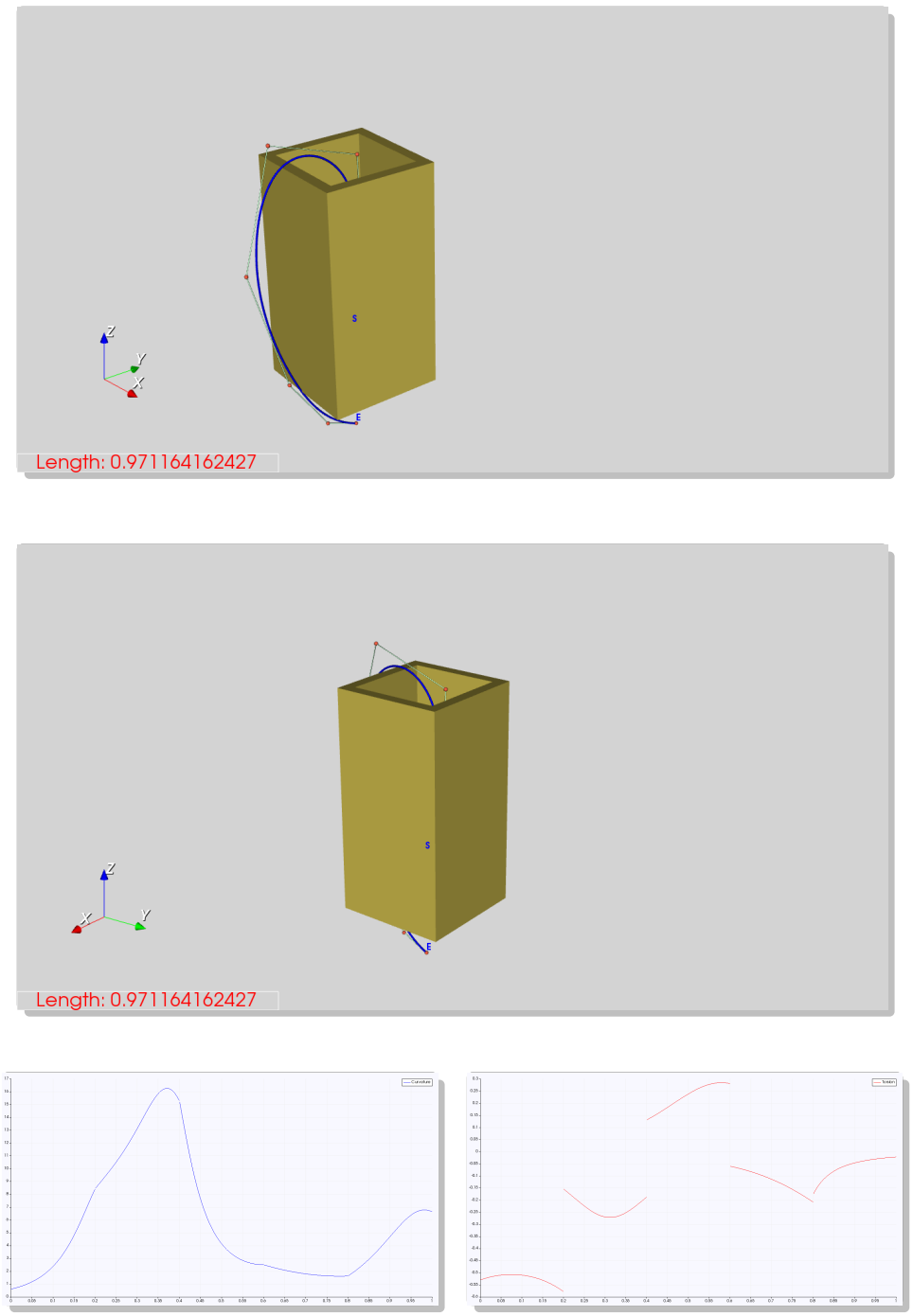


Figure 126.: Test 98; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 3; Meth. C; Part. **U**; Config. 3b.

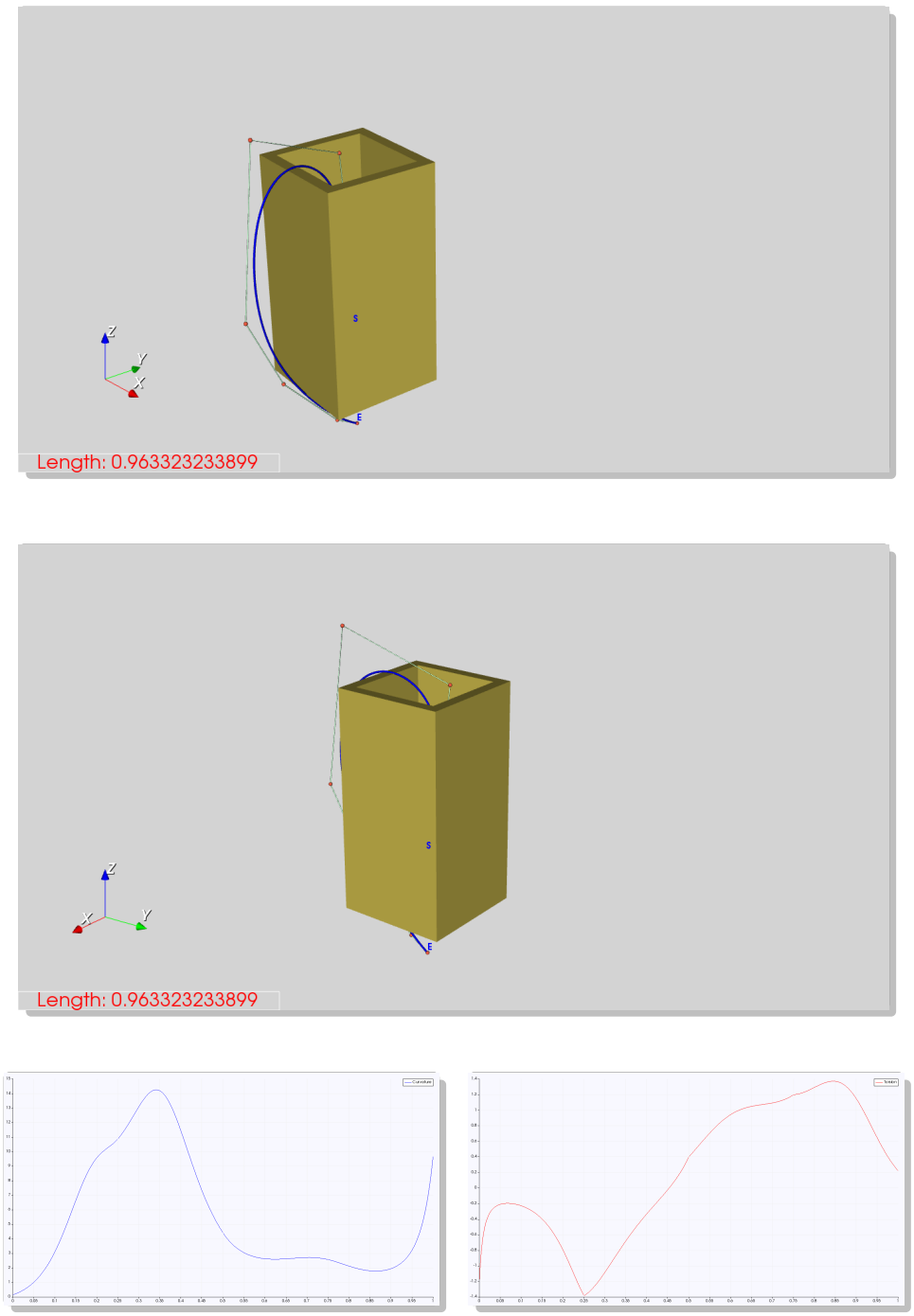


Figure 127.: Test 99; Scene 3; **s** → ***e*** [0.5,0.5,0.4]→[0.5,0.5,0.2]; Deg. 4; Meth. C; Part. **U**; Config. 3b.

8

CONCLUSIONS

We evince from the tests that

Part IV

APPENDIX

A

SOURCE CODE

A.1 CLASSES

A.1.1 *voronizator.py*

```
import numpy as np
import numpy.linalg
import scipy as sp
import scipy.spatial
import networkx as nx
import numpy.linalg
import polyhedron
import polyhedronsContainer
import path
import uuid
import xml.etree.cElementTree as ET

class Voronizator:
    _pruningMargin = 0.3

    def __init__(self, sites=np.array([]), bsplineDegree=4, adaptivePartition=False):
        self._shortestPath = path.Path(bsplineDegree, adaptivePartition)
        self._sites = sites
        self._graph = nx.Graph()
        self._tGraph = nx.DiGraph()
        self._startTriplet = None
        self._endTriplet = None
        self._polyhedronsContainer = polyhedronsContainer.PolyhedronsContainer()
        self._pathStart = np.array([])
        self._pathEnd = np.array([])
        self._startId = uuid.uuid4()
        self._endId = uuid.uuid4()
        self._bsplineDegree = bsplineDegree

    def setBsplineDegree(self, bsplineDegree):
        self._bsplineDegree = bsplineDegree
        self._shortestPath.setBsplineDegree(bsplineDegree)

    def setAdaptivePartition(self, adaptivePartition):
        self._shortestPath.setAdaptivePartition(adaptivePartition)

    def setCustomSites(self, sites):
        self._sites = sites

    def setRandomSites(self, number, seed=None):
        if seed != None:
            np.random.seed(0)
        self._sites = sp.rand(number,3)

    def addPolyhedron(self, polyhedron):
        self._polyhedronsContainer.addPolyhedron(polyhedron)

    def addBoundingBox(self, a, b, maxEmptyArea=1, invisible=True, verbose=False):
        if verbose:
            print('Add bounding box', flush=True)

        self._polyhedronsContainer.addBoundingBox(a,b,maxEmptyArea, invisible)
```

```
53
54      def setPolyhedronsSites(self, verbose=False):
55          if verbose:
56              print('Set sites for Voronoi', flush=True)
57
58          sites = []
59          for polyhedron in self._polyhedronsContainer.polyhedrons:
60              sites.extend(polyhedron.allPoints)
61
62          self._sites = np.array(sites)
63
64      def makeVoroGraph(self, prune=True, verbose=False, debug=False):
65          if verbose:
66              print('Calculate Voronoi cells', flush=True)
67          ids = {}
68          vor = sp.spatial.Voronoi(self._sites)
69
70          if verbose:
71              print('Make pruned Graph from cell edges ', end='', flush=True)
72              printDotBunch = 0
73          vorVer = vor.vertices
74          for ridge in vor.ridge_vertices:
75              if verbose:
76                  if printDotBunch == 0:
77                      print('.', end='', flush=True)
78                  printDotBunch = (printDotBunch+1)%10
79
80              for i in range(1, len(ridge)):
81                  for j in range(i):
82                      if (ridge[i] != -1) and (ridge[j] != -1):
83                          a = vorVer[ridge[i]]
84                          b = vorVer[ridge[j]]
85                          if (not prune) or (not self._polyhedronsContainer.segmentIntersectPolyhedrons(a,b, intersectionMargin
                                ↪ = self._pruningMargin)):
86                              if tuple(a) in ids.keys():
87                                  idA = ids[tuple(a)]
88                              else:
89                                  idA = uuid.uuid4()
90                                  self._graph.add_node(idA, coord=a)
91                                  ids[tuple(a)] = idA
92
93                              if tuple(b) in ids.keys():
94                                  idB = ids[tuple(b)]
95                              else:
96                                  idB = uuid.uuid4()
97                                  self._graph.add_node(idB, coord=b)
98                                  ids[tuple(b)] = idB
99
100                             self._graph.add_edge(idA, idB, weight=np.linalg.norm(a-b))
101
102         if verbose:
103             print('', flush=True)
104
105         self._createTripleGraph(verbose, debug)
106
107     def calculateShortestPath(self, start, end, attachMode='near', prune=True, useMethod='cleanPath', postSimplify=True,
            ↪ verbose=False, debug=False):
108         """
109         useMethod: cleanPath; trijkstra; annealing; none
110         """
111         if useMethod == 'trijkstra' or useMethod == 'cleanPath' or useMethod == 'annealing' or useMethod == 'none':
112             if verbose:
113                 print('Attach start and end points', flush=True)
114             if attachMode=='near':
115                 self._attachToGraphNear(start, end, prune)
116             elif attachMode=='all':
117                 self._attachToGraphAll(start, end, prune)
118             else:
119                 self._attachToGraphNear(start, end, prune)
120
121             self._attachSpecialStartEndTriples(verbose)
122
123             self._pathStart = start
124             self._pathEnd = end
125
126             if useMethod == 'trijkstra':
127                 self._removeCollidingTriples(verbose, debug)
128
129             triPath = self._dijkstra(verbose, debug)
130             path = self._extractPath(triPath, verbose)
131             self._shortestPath.assignValues(path, self._polyhedronsContainer)
132
```

```
133             if useMethod == 'cleanPath':
134                 self._shortestPath.clean(verbose, debug)
135             elif useMethod == 'annealing':
136                 self._shortestPath.anneal(verbose)
137 
138             #print(self._bsplineDegree)
139             if useMethod != 'annealing' and useMethod != 'none':
140                 if self._bsplineDegree == 3:
141                     self._shortestPath.addNAlignedVertexes(1, verbose, debug)
142                 if self._bsplineDegree == 4:
143                     self._shortestPath.addNAlignedVertexes(2, verbose, debug)
144 
145             if postSimplify:
146                 self._shortestPath.simplify(verbose, debug)
147 
148 
149     def plotSites(self, plotter, verbose=False):
150         if verbose:
151             print('Plot Sites', end='', flush=True)
152 
153         if self._sites.size > 0:
154             plotter.addPoints(self._sites, plotter.COLOR_SITES, thick=True)
155 
156     def plotPolyhedrons(self, plotter, verbose=False):
157         if verbose:
158             print('Plot Polyhedrons', end='', flush=True)
159 
160         for poly in self._polyhedronsContainer.polyhedrons:
161             poly.plot(plotter)
162             if verbose:
163                 print('.', end='', flush=True)
164 
165         if verbose:
166             print('', flush=True)
167 
168     def plotShortestPath(self, plotter, verbose=False):
169         if verbose:
170             print('Plot shortest path', flush=True)
171 
172         if self._shortestPath.vertexes.size > 0:
173             if self._polyhedronsContainer.hasBoundingBox:
174                 splineThickness = np.linalg.norm(np.array(self._polyhedronsContainer.boundingBoxB) - np.array(self.
                        ↪ _polyhedronsContainer.boundingBoxA)) / 1000.
175                 pointThickness = splineThickness * 2.
176                 lineThickness = splineThickness / 2.
177 
178                 plotter.addPolyLine(self._shortestPath.vertexes, plotter.COLOR_CONTROL_POLIG, thick=True, thickness=
                        ↪ lineThickness)
179                 plotter.addPoints(self._shortestPath.vertexes, plotter.COLOR_CONTROL_POINTS, thick=True, thickness=
                        ↪ pointThickness)
180                 plotter.addBSpline(self._shortestPath, self._bsplineDegree, plotter.COLOR_PATH, thick=True, thickness=
                        ↪ splineThickness)
181 
182             else:
183                 plotter.addPolyLine(self._shortestPath.vertexes, plotter.COLOR_CONTROL_POLIG, thick=True)
184                 plotter.addPoints(self._shortestPath.vertexes, plotter.COLOR_CONTROL_POINTS, thick=True)
185                 plotter.addBSpline(self._shortestPath, self._bsplineDegree, plotter.COLOR_PATH, thick=True)
186 
187     def plotGraph(self, plotter, verbose=False):
188         if verbose:
189             print('Plot shortest path', flush=True)
190 
191         plotter.addGraph(self._graph, plotter.COLOR_GRAPH)
192 
193     def extractXmlTree(self, root):
194         if self._polyhedronsContainer.hasBoundingBox:
195             xmlBoundingBox = ET.SubElement(root, 'boundingBox')
196             ET.SubElement(xmlBoundingBox, 'a', x=str(self._polyhedronsContainer.boundingBoxA[0]), y=str(self.
                    ↪ _polyhedronsContainer.boundingBoxA[1]), z=str(self._polyhedronsContainer.boundingBoxA[2]))
197             ET.SubElement(xmlBoundingBox, 'b', x=str(self._polyhedronsContainer.boundingBoxB[0]), y=str(self.
                    ↪ _polyhedronsContainer.boundingBoxB[1]), z=str(self._polyhedronsContainer.boundingBoxB[2]))
198 
199         xmlPolyhedrons = ET.SubElement(root, 'polyhedrons')
200         for polyhedron in self._polyhedronsContainer.polyhedrons:
201             xmlPolyhedron = polyhedron.extractXmlTree(xmlPolyhedrons)
202 
203     def importXmlTree(self, root, maxEmptyArea):
204         xmlBoundingBox = root.find('boundingBox')
205         if xmlBoundingBox:
206             xmlA = xmlBoundingBox.find('a')
207             xmlB = xmlBoundingBox.find('b')
208 
```

```
            self._polyhedronsContainer.hasBoundingBox = True
            self._polyhedronsContainer.boundingBoxA = [float(xmlA.attrib['x']), float(xmlA.attrib['y']), float(xmlA.attrib['z
                ↪ '])]
            self._polyhedronsContainer.boundingBoxB = [float(xmlB.attrib['x']), float(xmlB.attrib['y']), float(xmlB.attrib['z
                ↪ '])]

        xmlPolyhedrons = root.find('polyhedrons')
        if xmlPolyhedrons:
            for xmlPolyhedron in xmlPolyhedrons.iter('polyhedron'):
                invisible = False
                if 'invisible' in xmlPolyhedron.attrib.keys():
                    invisible = bool(eval(xmlPolyhedron.attrib['invisible']))

                boundingBox = False
                if 'boundingBox' in xmlPolyhedron.attrib.keys():
                    boundingBox = bool(eval(xmlPolyhedron.attrib['boundingBox']))

                faces = []
                for xmlFace in xmlPolyhedron.iter('face'):
                    vertexes = []
                    for xmlVertex in xmlFace.iter('vertex'):
                        vertexes.append([float(xmlVertex.attrib['x']), float(xmlVertex.attrib['y']), float(xmlVertex.attrib['
                            ↪ z'])])

                    faces.append(vertexes)

                newPolyhedron = polyhedron.Polyhedron(faces=np.array(faces), invisible=invisible, maxEmptyArea=maxEmptyArea,
                    ↪ boundingBox=boundingBox)
                self._polyhedronsContainer.addPolyhedron(newPolyhedron)

    def _attachToGraphNear(self, start, end, prune):
        firstS = True
        firstE = True
        minAttachS = None
        minAttachE = None
        minDistS = 0.
        minDistE = 0.
        for node,nodeAttr in self._graph.node.items():
            if (not prune) or (not self._polyhedronsContainer.segmentIntersectPolyhedrons(start,nodeAttr['coord'],
                ↪ intersectionMargin= self._pruningMargin)):
                if firstS:
                    minAttachS = node
                    minDistS = np.linalg.norm(start - nodeAttr['coord'])
                    firstS = False
                else:
                    currDist = np.linalg.norm(start - nodeAttr['coord'])
                    if currDist < minDistS:
                        minAttachS = node
                        minDistS = currDist

            if (not prune) or (not self._polyhedronsContainer.segmentIntersectPolyhedrons(end, nodeAttr['coord'],
                ↪ intersectionMargin= self._pruningMargin)):
                if firstE:
                    minAttachE = node
                    minDistE = np.linalg.norm(end - nodeAttr['coord'])
                    firstE = False
                else:
                    currDist = np.linalg.norm(end - nodeAttr['coord'])
                    if currDist < minDistE:
                        minAttachE = node
                        minDistE = currDist

        if minAttachS != None:
            self._addNodeToTGraph(self._startId, start, minAttachS, minDistS, rightDirection=True)
        if minAttachE != None:
            self._addNodeToTGraph(self._endId, end, minAttachE, minDistE, rightDirection=False)

    def _attachToGraphAll(self, start, end, prune):
        for node,nodeAttr in self._graph.node.items():
            if (not prune) or (not self._polyhedronsContainer.segmentIntersectPolyhedrons(start, nodeAttr['coord'],
                ↪ intersectionMargin= self._pruningMargin)):
                self._addNodeToTGraph(self._startId, start, node, np.linalg.norm(start - nodeAttr['coord']), rightDirection=
                    ↪ True)
            if (not prune) or (not self._polyhedronsContainer.segmentIntersectPolyhedrons(end, nodeAttr['coord'],
                ↪ intersectionMargin= self._pruningMargin)):
                self._addNodeToTGraph(self._endId, end, node, np.linalg.norm(end - nodeAttr['coord']), rightDirection=False)

    def _addNodeToTGraph(self, newId, coord, attachId, dist, rightDirection):
        self._graph.add_node(newId, coord=coord)
        self._graph.add_edge(newId, attachId, weight=dist)
        for otherId in filter(lambda node: node != newId, self._graph.neighbors(attachId)):
            newTriplet = uuid.uuid4()
```

```
282             if rightDirection:
283                 self._tGraph.add_node(newTriplet, triplet=[newId,attachId,otherId])
284                 self._tGraph.add_edges_from([(newTriplet, otherTriplet, {'weight':dist}) for otherTriplet in self._tGraph.
                        ↪ nodes() if self._tGraph.node[otherTriplet]['triplet'][0] == attachId  and self._tGraph.node[
                        ↪ otherTriplet]['triplet'][1] == otherId])
285 
286             else:
287                 self._tGraph.add_node(newTriplet, triplet=[otherId,attachId,newId])
288                 self._tGraph.add_edges_from([(otherTriplet, newTriplet, {'weight':dist}) for otherTriplet in self._tGraph.
                        ↪ nodes() if self._tGraph.node[otherTriplet]['triplet'][1] == otherId  and self._tGraph.node[
                        ↪ otherTriplet]['triplet'][2] == attachId])
289 
290     def _attachSpecialStartEndTriples(self, verbose):
291         #attach special starting and ending triplet
292         if verbose:
293             print('Create starting and ending triplets', flush=True)
294 
295         self._startTriplet = uuid.uuid4()
296         self._endTriplet = uuid.uuid4()
297         self._tGraph.add_node(self._startTriplet, triplet = [self._startId,self._startId,self._startId], hit = False)
298         self._tGraph.add_node(self._endTriplet, triplet = [self._endId,self._endId,self._endId], hit = False)
299         self._tGraph.add_edges_from([(self._startTriplet, n, {'weight':0.}) for n in self._tGraph.nodes() if self._tGraph.
                ↪ node[n]['triplet'][0] == self._startId])
300         self._tGraph.add_edges_from([(n, self._endTriplet, {'weight':0.}) for n in self._tGraph.nodes() if self._tGraph.node[
                ↪ n]['triplet'][2] == self._endId])
301 
302     def _createTripleGraph(self, verbose, debug):
303         #create triplets
304 
305         if debug:
306             triplets_file = open("triplets.txt","w")
307 
308         if verbose:
309             print('Create triplets ', end='', flush=True)
310             printDotBunch = 0
311 
312         tripletIdList = {}
313         def getUniqueId(triplet):
314             if tuple(triplet) in tripletIdList.keys():
315                 tripletId = tripletIdList[tuple(triplet)]
316             else:
317                 tripletId = uuid.uuid4()
318                 tripletIdList[tuple(triplet)] = tripletId
319                 self._tGraph.add_node(tripletId, triplet = triplet)
320             return tripletId
321 
322         for edge in self._graph.edges():
323             if verbose:
324                 if printDotBunch == 0:
325                     print('.', end='', flush=True)
326                 printDotBunch = (printDotBunch+1)%10
327 
328 
329             tripletsSxOutgoing = []
330             tripletsSxIngoing = []
331             tripletsDxOutgoing = []
332             tripletsDxIngoing = []
333 
334             for nodeSx in filter(lambda node: node != edge[1], self._graph.neighbors(edge[0])):
335                 tripletId = getUniqueId([nodeSx,edge[0],edge[1]])
336                 tripletsSxOutgoing.append(tripletId)
337                 if debug:
338                     triplets_file.write('SxO: {}\n'.format(self._tGraph.node[tripletId]['triplet']))
339 
340                 tripletId = getUniqueId([edge[1],edge[0],nodeSx])
341                 tripletsSxIngoing.append(tripletId)
342                 if debug:
343                     triplets_file.write('SxI: {}\n'.format(self._tGraph.node[tripletId]['triplet']))
344 
345             for nodeDx in filter(lambda node: node != edge[0], self._graph.neighbors(edge[1])):
346                 tripletId = getUniqueId([nodeDx,edge[1],edge[0]])
347                 tripletsDxOutgoing.append(tripletId)
348                 if debug:
349                     triplets_file.write('DxO: {}\n'.format(self._tGraph.node[tripletId]['triplet']))
350 
351                 tripletId = getUniqueId([edge[0],edge[1],nodeDx])
352                 tripletsDxIngoing.append(tripletId)
353                 if debug:
354                     triplets_file.write('DxI: {}\n'.format(self._tGraph.node[tripletId]['triplet']))
355 
356             for tripletSx in tripletsSxOutgoing:
357                 for tripletDx in tripletsDxIngoing:
```

```
                    self._tGraph.add_edge(tripletSx, tripletDx, {'weight':self._graph.edge[self._tGraph.node[tripletSx]['
                        ↪ triplet'][0]][self._tGraph.node[tripletDx]['triplet'][0]]['weight']})

            for tripletDx in tripletsDxOutgoing:
                for tripletSx in tripletsSxIngoing:
                    self._tGraph.add_edge(tripletDx, tripletSx, {'weight':self._graph.edge[self._tGraph.node[tripletDx]['
                        ↪ triplet'][0]][self._tGraph.node[tripletSx]['triplet'][0]]['weight']})

        if verbose:
            print('', flush=True)

        if debug:
            triplets_file.close()


    def _dijkstra(self, verbose, debug):
        try:
            if verbose:
                print('Dijkstra algorithm', flush=True)

            length,triPath=nx.bidirectional_dijkstra(self._tGraph, self._startTriplet, self._endTriplet)


        except (nx.NetworkXNoPath, nx.NetworkXError):
            print('ERROR: Impossible to find a path')
            triPath = []

        return triPath

    def _extractPath(self, triPath, verbose):
        if verbose:
            print('Extract path', flush=True)

        path = []
        for t in triPath:
            path.append(self._graph.node[self._tGraph.node[t]['triplet'][1]]['coord'])
        return np.array(path)


    def _removeCollidingTriples(self, verbose, debug):
        if verbose:
            print('Remove colliding triples', flush=True)
            printDotBunch = 0

        toRemove = []
        for triple in self._tGraph:
            if verbose:
                if printDotBunch == 0:
                    print('.', end='', flush=True)
                printDotBunch = (printDotBunch+1)%10

            a = self._graph.node[self._tGraph.node[triple]['triplet'][0]]['coord']
            b = self._graph.node[self._tGraph.node[triple]['triplet'][1]]['coord']
            c = self._graph.node[self._tGraph.node[triple]['triplet'][2]]['coord']
            intersect,intersectRes = self._polyhedronsContainer.triangleIntersectPolyhedrons(a, b, c)
            if intersect:
                toRemove.append(triple)

        if verbose:
            print ("", flush=True)

        for triple in toRemove:
            self._tGraph.remove_node(triple)
```

A.1.2 *path.py*

```
import random
import math
import numpy as np
import scipy as sp
import scipy.interpolate

class Path:
    _initialTemperature = 10#1000
    _trials = 10#100
    _warmingRatio = 0.9#0.9
```

```
    _minTemperature=0.00001#0.00000001
    _minDeltaEnergy=0.000001
    _maxVlambdaPert = 1000.
    _maxVertexPertFactor = 100.
    _initialVlambda = 0.
    _changeVlambdaProbability = 0.05
    #====1
    #_useArcLen = True
    #_ratioCurvTorsLen = [0.1, 0.1, 0.8]
    #====2
    _useArcLen = False
    _ratioCurvTorsLen = [0.1, 0.1, 0.8]
    #====3
    #_useArcLen = True
    #_ratioCurvTorsLen = [0.3, 0.3, 0.4]

    def __init__(self, bsplineDegree, adaptivePartition):
        self._bsplineDegree = bsplineDegree
        self._adaptivePartition = adaptivePartition
        self._vertexes = np.array([])
        self._dimC = 0
        self._polyhedronsContainer = []
        self._vlambda = self._initialVlambda


    @property
    def vertexes(self):
        return self._vertexes

    def assignValues(self, path, polyhedronsContainer):
        self._vertexes = path
        self._dimC = self._vertexes.shape[1]

        self._polyhedronsContainer = polyhedronsContainer
        tau, u, spline, splineD1, splineD2, splineD3, curv, tors, arcLength, polLength = self._splinePoints(self._vertexes)
        self._maxVertexPert = polLength / self._maxVertexPertFactor


    def setBsplineDegree(self, bsplineDegree):
        self._bsplineDegree = bsplineDegree

    def setAdaptivePartition(self, adaptivePartition):
        self._adaptivePartition = adaptivePartition

    def clean(self, verbose, debug):
        if verbose:
            print('Clean path (avoid obstacles)', flush=True)

        newPath = []
        if len(self._vertexes) > 0:
            a = self._vertexes[0]
            newPath.append(self._vertexes[0])

        for i in range(1, len(self._vertexes)-1):
            v = self._vertexes[i]
            b = self._vertexes[i+1]

            intersect,intersectRes = self._polyhedronsContainer.triangleIntersectPolyhedrons(a, v, b)
            if intersect:
                alpha = intersectRes[1]

                a1 = (1.-alpha)*a + alpha*v
                b1 = alpha*v + (1.-alpha)*b

                newPath.append(a1)
                newPath.append(v)
                newPath.append(b1)

                a = b1
            else:
                newPath.append(v)

                a = v

        if len(self._vertexes) > 0:
            newPath.append(self._vertexes[len(self._vertexes)-1])

        self._vertexes = np.array(newPath)


    def anneal(self, verbose):
        if verbose:
```

```
            print('Anneal path', flush=True)

        tau, u, self._spline, splineD1, splineD2, splineD3, curv, tors, arcLength, polLength = self._splinePoints(self._vertexes)
        self._currentEnergy, self._maxCurvatureLength, self._currentConstraints = self._initializePathEnergy(self._vertexes, self._spline, splineD1, splineD2, self._vlambda)


        temperature = self._initialTemperature
        while True:
            initialEnergy = self._currentEnergy
            numMovedLambda = 0
            numMovedVertex = 0
            for i in range(self._trials):
                movedLambda,movedVertex = self._tryMove(temperature)
                if movedLambda:
                    numMovedLambda += 1
                if movedVertex:
                    numMovedVertex += 1
            deltaEnergy = abs(initialEnergy - self._currentEnergy)
            temperature = temperature * self._warmingRatio
            if verbose:
                print("T:{}; E:{}; DE:{}; L:{}; C:{}; ML:{}; MV:{}".format(temperature, self._currentEnergy, deltaEnergy, self._vlambda, self._currentConstraints, numMovedLambda, numMovedVertex), flush=True)
                #print(self._vertexes)

            if (temperature < self._minTemperature) or (numMovedVertex > 0 and (deltaEnergy < self._minDeltaEnergy) and self._currentConstraints == 0.):
                break


    def simplify(self, verbose, debug):
        if verbose:
            print('Simplify path (remove useless triples)', flush=True)
        if self._bsplineDegree == 2:
            self._simplify2()
        elif self._bsplineDegree == 3:
            self._simplify3()
        elif self._bsplineDegree == 4:
            self._simplify4()

    def _simplify2(self):
        simplifiedPath = []
        if len(self._vertexes) > 0:
            a = self._vertexes[0]
            simplifiedPath.append(self._vertexes[0])
        first = True
        for i in range(1,len(self._vertexes)-1):
            v = self._vertexes[i]
            b = self._vertexes[i+1]
            keepV = False

            intersectCurr,nihil = self._polyhedronsContainer.triangleIntersectPolyhedrons(a, v, b)

            if not intersectCurr:
                if first:
                    intersectPrec = False
                else:
                    #a1 = self._vertexes[i-2]
                    intersectPrec,nihil = self._polyhedronsContainer.triangleIntersectPolyhedrons(a1, a, b)

                if i == len(self._vertexes)-2:
                    intersectSucc = False
                else:
                    b1 = self._vertexes[i+2]
                    intersectSucc,nihil = self._polyhedronsContainer.triangleIntersectPolyhedrons(a, b, b1)

                if intersectPrec or intersectSucc:
                    keepV = True

            else:
                keepV = True

            if keepV:
                first = False
                simplifiedPath.append(v)
                a1 = a
                a = v

        if len(self._vertexes) > 0:
            simplifiedPath.append(self._vertexes[len(self._vertexes)-1])

```

```
171         self._vertexes = np.array(simplifiedPath)
172
173     def _simplify3(self):
174         simp = list(self._vertexes)
175         toEval = 0
176         while toEval < len(simp)-2:
177             toEval += 1
178             if toEval >= 3:
179                 if toEval < len(simp)-1:
180                     if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-3], simp[toEval-2], simp[
                            ↪ toEval-1], simp[toEval+1]]):
181                         continue
182
183             if toEval >= 2:
184                 if toEval < len(simp)-2:
185                     if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-2], simp[toEval-1], simp[
                            ↪ toEval+1], simp[toEval+2]]):
186                         continue
187
188             if toEval < len(simp)-3:
189                 if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-1], simp[toEval+1], simp[toEval
                        ↪ +2], simp[toEval+3]]):
190                     continue
191
192             del simp[toEval]
193             toEval -= 1
194
195         self._vertexes = np.array(simp)
196
197     def _simplify4(self):
198         simp = list(self._vertexes)
199         toEval = 0
200         while toEval < len(simp)-2:
201             toEval += 1
202             if toEval >= 4:
203                 if toEval < len(simp)-1:
204                     if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-4], simp[toEval-3], simp[
                            ↪ toEval-2], simp[toEval-1], simp[toEval+1]]):
205                         continue
206
207             if toEval >= 3:
208                 if toEval < len(simp)-2:
209                     if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-3], simp[toEval-2], simp[
                            ↪ toEval-1], simp[toEval+1], simp[toEval+2]]):
210                         continue
211
212             if toEval >= 2:
213                 if toEval < len(simp)-3:
214                     if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-2], simp[toEval-1], simp[
                            ↪ toEval+1], simp[toEval+2], simp[toEval+3]]):
215                         continue
216
217             if toEval < len(simp)-4:
218                 if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-1], simp[toEval+1], simp[toEval
                        ↪ +2], simp[toEval+3], simp[toEval+4]]):
219                     continue
220
221             del(simp[toEval])
222             toEval -= 1
223
224         self._vertexes = np.array(simp)
225
226     def addNAlignedVertexes(self, numVertexes, verbose, debug):
227         if verbose:
228             print('Increase degree', flush=True)
229
230         newPath = []
231         for i in range(1, len(self._vertexes)):
232             a = self._vertexes[i-1]
233             b = self._vertexes[i]
234             newPath.append(a)
235
236             if numVertexes == 1:
237                 n = 0.5 * a + 0.5 * b
238                 newPath.append(n)
239
240             elif numVertexes == 2:
241                 n1 = 0.33 * b + 0.67 * a
242                 n2 = 0.33 * a + 0.67 * b
243                 newPath.append(n1)
244                 newPath.append(n2)
245
```

```
        if len(self._vertexes) > 0:
            newPath.append(self._vertexes[len(self._vertexes)-1])

        self._vertexes = np.array(newPath)

    def splinePoints(self):
        return self._splinePoints(self._vertexes)

    def _splinePoints(self, vertexes):

        x = vertexes[:,0]
        y = vertexes[:,1]
        z = vertexes[:,2]

        polLen = self._calculatePolyLength(vertexes)

        tau,t = self._createKnotPartition(vertexes)

        #[knots, coeff, degree]
        tck = [t,[x,y,z], self._bsplineDegree]

        u=np.linspace(0,1,(max(polLen*5,1000)),endpoint=True)

        out = sp.interpolate.splev(u, tck)
        outD1 = sp.interpolate.splev(u, tck, 1)
        outD2 = sp.interpolate.splev(u, tck, 2)

        spline = np.stack(out).T
        splineD1 = np.stack(outD1).T
        splineD2 = np.stack(outD2).T

        if self._bsplineDegree >= 3:
            outD3 = sp.interpolate.splev(u, tck, 3)
            splineD3 = np.stack(outD3).T
        else:
            splineD3 = None

        curv = []
        tors = []
        arcLength = 0.
        for i in range(len(u)):
            d1Xd2 = np.cross(splineD1[i], splineD2[i])
            Nd1Xd2 = np.linalg.norm(d1Xd2)
            Nd1 = np.linalg.norm(splineD1[i])

            currCurv = 0.
            if Nd1 >0.: #>= 1.:
                currCurv = Nd1Xd2 / math.pow(Nd1,3)

            currTors = 0.
            if self._bsplineDegree >= 3 and Nd1Xd2 > 0.: #>= 1.:
                try:
                    currTors = np.dot(d1Xd2, splineD3[i]) / math.pow(Nd1Xd2, 2)
                except RuntimeWarning:
                    currTors = 0.

            curv.append(currCurv)
            tors.append(currTors)

            if i >= 1:
                dMin = min(prevNd1, Nd1)
                dMax = max(prevNd1, Nd1)
                arcLength += (u[i]-u[i-1]) * (dMin + ((dMax-dMin) / 2.))

            prevNd1 = Nd1

        return (tau, u, spline, splineD1, splineD2, splineD3, curv, tors, arcLength, polLen)

    def _createKnotPartition(self, controlPolygon):
        nv = len(controlPolygon)
        nn = nv - self._bsplineDegree + 1

        if not self._adaptivePartition:
            T = np.linspace(0,1,nv-self._bsplineDegree+1,endpoint=True)
        else:
            d = [0]
            for j in range(1, nv):
                d.append(d[j-1] + np.linalg.norm(controlPolygon[j] - controlPolygon[j-1]))
            t = []
            for i in range(nn-1):
                a = i * (nv-1) / (nn-1)
                ai = math.floor(a)
```

```
328                 ad = a - ai
329                 p = ad * controlPolygon[ai+1] + (1-ad) * controlPolygon[ai]
330                 l = d[ai] + np.linalg.norm(p - controlPolygon[ai])
331                 t.append(l / d[nv-1])
332 
333             t.append(1.)
334 
335             T = np.array(t)
336 
337         Text = np.append([0]*self._bsplineDegree, T)
338         Text = np.append(Text, [1]*self._bsplineDegree)
339 
340         return (T,Text)
341 
342     def _initializePathEnergy(self, vertexes, spline, splineD1, splineD2, vlambda):
343         tau, u, spline, splineD1, splineD2, splineD3, curv, tors, arcLength, polLength = self._splinePoints(vertexes)
344         if self._useArcLen:
345             length = arcLength
346         else:
347             length = polLength
348 
349         self._initialLength = length
350         maxCurvatureLength = self._calculateMaxCurvatureLength(length, curv, tors)
351         constraints = self._calculateConstraints(spline)
352         energy = maxCurvatureLength + vlambda * constraints
353 
354         return (energy, maxCurvatureLength, constraints)
355 
356     def _tryMove(self, temperature):
357         """
358         Move the path or lambda multipiers in a neighbouring state,
359         with a certain acceptance probability.
360         Pick a random vertex (except extremes), and move
361         it in a random direction (with a maximum perturbance).
362         Use a lagrangian relaxation because we need to evaluate
363         min(measure(path)) given the constraint that all quadrilaters
364         formed by 4 consecutive points in the path must be collision
365         free; where measure(path) is, depending of the choose method,
366         the length of the path or the mean
367         of the supplementary angles of each pair of edges of the path.
368         If neighbourMode=0 then move the node uniformly, if
369         neighbourMode=1 then move the node with gaussian probabilities
370         with mean in the perpendicular direction respect to the
371         previous-next nodes axis.
372         """
373 
374         movedLambda = False
375         movedVertex = False
376         moveVlambda = random.random() < self._changeVlambdaProbability
377         if moveVlambda:
378             newVlambda = self._vlambda
379             newVlambda = newVlambda + (random.uniform(-1.,1.) * self._maxVlambdaPert)
380 
381             newEnergy = self._calculatePathEnergyLambda(newVlambda)
382 
383             #attention, different formula from below
384             if (newEnergy > self._currentEnergy) or (math.exp(-(self._currentEnergy-newEnergy)/temperature) >= random.random
                    ↪ ()):
385                 self._vlambda = newVlambda
386                 self._currentEnergy = newEnergy
387                 movedLambda = True
388 
389         else:
390             newVertexes = np.copy(self._vertexes)
391             movedV = random.randint(1,len(self._vertexes) - 2) #don't change extremes
392 
393             moveC = random.randint(0,self._dimC - 1)
394             newVertexes[movedV][moveC] = newVertexes[movedV][moveC] + (random.uniform(-1.,1.) * self._maxVertexPert)
395 
396             newEnergy,newMaxCurvatureLength,newConstraints = self._calculatePathEnergyVertex(newVertexes)
397 
398             #attention, different formula from above
399             if (newEnergy < self._currentEnergy) or (math.exp(-(newEnergy-self._currentEnergy)/temperature) >= random.random
                    ↪ ()):
400                 self._vertexes = newVertexes
401                 self._currentEnergy = newEnergy
402                 self._currentMaxCurvatureLength = newMaxCurvatureLength
403                 self._currentConstraints = newConstraints
404                 movedVertex = True
405 
406         return (movedLambda, movedVertex)
407 
```

```
    def _calculatePathEnergyLambda(self, vlambda):
        """
        calculate the energy when lambda is moved.
        """
        return (self._currentEnergy - (self._vlambda * self._currentConstraints) + (vlambda * self._currentConstraints))

    def _calculatePathEnergyVertex(self, vertexes):
        """
        calculate the energy when a vertex is moved and returns it.
        """
        tau, u, spline, splineD1, splineD2, splineD3, curv, tors, arcLength, polLength = self._splinePoints(vertexes)
        if self._useArcLen:
            length = arcLength
        else:
            length = polLength

        constraints = self._calculateConstraints(spline)#this is bottleneck
        maxCurvatureLength = self._calculateMaxCurvatureLength(length, curv, tors)

        energy = maxCurvatureLength + self._vlambda * constraints

        return (energy, maxCurvatureLength, constraints)

    def _calculatePolyLength(self, vertexes):
        length = 0.
        for i in range(1, len(vertexes)):
            length += sp.spatial.distance.euclidean(vertexes[i-1], vertexes[i])
            #length += np.linalg.norm(np.subtract(vertexes[i], vertexes[i-1]))
        return length

    def _calculateMaxCurvatureLength(self, length, curv, tors):
        normLength = length/self._initialLength * 100 #for making the ratio indipendent of the initial length

        maxCurvature = 0.
        maxTorsion = 0.
        for i in range(0, len(curv)):
            currCurv = curv[i]
            currTors = abs(tors[i])
            if currCurv > maxCurvature:
                maxCurvature = currCurv
            if currTors > maxTorsion:
                maxTorsion = currTors

        return self._ratioCurvTorsLen[0]*maxCurvature + self._ratioCurvTorsLen[1]*maxTorsion + self._ratioCurvTorsLen[2]*
            ↪ normLength

    def _calculateConstraints(self, spline):
        """
        calculate the constraints function. Is the ratio of the points
        of the calculated spline that are inside obstacles respect the
        total number of points of the spline.
        """
        pointsInside = 0
        for p in spline:
            if self._polyhedronsContainer.pointInsidePolyhedron(p):
                pointsInside = pointsInside + 1

        constraints = pointsInside / len(spline)

        return constraints
```

A.1.3 *plotter.py*

```
import numpy as np
import scipy as sp
import scipy.interpolate
import scipy.spatial
import pickle
import vtk
import vtk.util.colors
import math
import warnings
warnings.filterwarnings("error")

class Plotter:
    COLOR_BG = vtk.util.colors.light_grey
```

```
    COLOR_BG_PLOT = vtk.util.colors.ghost_white
    #vtk.util.colors.ivory
    COLOR_OBSTACLE = vtk.util.colors.banana
    COLOR_SITES = vtk.util.colors.cobalt
    COLOR_PATH = vtk.util.colors.brick
    COLOR_CONTROL_POINTS = vtk.util.colors.tomato
    COLOR_CONTROL_POLIG = vtk.util.colors.mint
    COLOR_GRAPH = vtk.util.colors.sepia
    COLOR_PLOT_CURV = vtk.util.colors.blue
    COLOR_PLOT_TORS = vtk.util.colors.red
    COLOR_LABELS = vtk.util.colors.blue
    COLOR_LENGTH = vtk.util.colors.red

    _DEFAULT_LINE_THICKNESS = 0.0005
    _DEFAULT_POINT_THICKNESS = 0.01#0.002
    _DEFAULT_BSPLINE_THICKNESS = 0.001

    class KeyPressInteractorStyle(vtk.vtkInteractorStyleUnicam):
        _screenshotFile = "/tmp/screenshot.png"
        _cameraFile = "/tmp/cameraData.dat"
        _cameraFile2 = "/tmp/cameraData2.dat"
        def __init__(self, parent=None):
            self.AddObserver("KeyPressEvent", self._keyPressEvent)
            self.AddObserver("RightButtonPressEvent", self._mousePressEvent)
            #super(KeyPressInteractorStyle, self).__init__()

        def SetCamera(self, camera):
            self._camera = camera

        def SetRenderer(self, renderer):
            self._renderer = renderer

        def SetRenderWindow(self, renderWindow):
            self._renderWindow = renderWindow

        def _keyPressEvent(self, obj, event):
            if obj.GetInteractor().GetKeySym() == "l":
                print("Scene screenshot in "+self._screenshotFile)
                w2if = vtk.vtkWindowToImageFilter()
                w2if.SetInput(self._renderWindow)
                w2if.Update()

                writer = vtk.vtkPNGWriter()
                writer.SetFileName(self._screenshotFile)
                writer.SetInputData(w2if.GetOutput())
                writer.Write()

            elif obj.GetInteractor().GetKeySym() == "c":
                print("Save camera data in "+self._cameraFile)
                record = {}
                record['position'] = self._camera.GetPosition()
                record['focalPoint'] = self._camera.GetFocalPoint()
                record['viewAngle'] = self._camera.GetViewAngle()
                record['viewUp'] = self._camera.GetViewUp()
                record['clippingRange'] = self._camera.GetClippingRange()

                with open(self._cameraFile, 'wb') as f:
                    pickle.dump(record, f)

            elif obj.GetInteractor().GetKeySym() == "v":
                print("Restore camera data from "+self._cameraFile)

                with open(self._cameraFile, 'rb') as f:
                    record = pickle.load(f)

                    self._camera.SetPosition(record['position'])
                    self._camera.SetFocalPoint(record['focalPoint'])
                    self._camera.SetViewAngle(record['viewAngle'])
                    self._camera.SetViewUp(record['viewUp'])
                    self._camera.SetClippingRange(record['clippingRange'])

                    self._renderWindow.Render()

            elif obj.GetInteractor().GetKeySym() == "b":
                print("Restore camera data from "+self._cameraFile2)

                with open(self._cameraFile2, 'rb') as f:
                    record = pickle.load(f)

                    self._camera.SetPosition(record['position'])
                    self._camera.SetFocalPoint(record['focalPoint'])
                    self._camera.SetViewAngle(record['viewAngle'])
```

```
                    self._camera.SetViewUp(record['viewUp'])
                    self._camera.SetClippingRange(record['clippingRange'])

                    self._renderWindow.Render()


            self.OnKeyPress()

        def _mousePressEvent(self, obj, event):
            clickPos = obj.GetInteractor().GetEventPosition()
            picker =vtk.vtkPropPicker()
            picker.Pick(clickPos[0], clickPos[1], 0, self._renderer)
            pos = picker.GetPickPosition()
            print(pos)


    class KeyPressContextInteractorStyle(vtk.vtkContextInteractorStyle):
        _screenshotFile = "/tmp/screenshot.png"
        def __init__(self, parent=None):
            self.AddObserver("KeyPressEvent",self._keyPressEvent)

        def SetRenderWindow(self, renderWindow):
            self._renderWindow = renderWindow

        def _keyPressEvent(self, obj, event):
            if obj.GetInteractor().GetKeySym() == "l":
                print("Plot screenshot in "+self._screenshotFile)
                w2if = vtk.vtkWindowToImageFilter()
                w2if.SetInput(self._renderWindow)
                w2if.Update()

                writer = vtk.vtkPNGWriter()
                writer.SetFileName(self._screenshotFile)
                writer.SetInputData(w2if.GetOutput())
                writer.Write()



    def __init__(self):
        self._rendererScene = vtk.vtkRenderer()
        self._rendererScene.SetBackground(self.COLOR_BG)

        self._renderWindowScene = vtk.vtkRenderWindow()
        self._renderWindowScene.AddRenderer(self._rendererScene)
        self._renderWindowInteractor = vtk.vtkRenderWindowInteractor()
        self._renderWindowInteractor.SetRenderWindow(self._renderWindowScene)
        #self._interactorStyle = vtk.vtkInteractorStyleUnicam()
        self._interactorStyle = self.KeyPressInteractorStyle()
        self._interactorStyle.SetCamera(self._rendererScene.GetActiveCamera())
        self._interactorStyle.SetRenderer(self._rendererScene)
        self._interactorStyle.SetRenderWindow(self._renderWindowScene)

        self._contextViewPlotCurv = vtk.vtkContextView()
        self._contextViewPlotCurv.GetRenderer().SetBackground(self.COLOR_BG_PLOT)

        self._contextInteractorStyleCurv = self.KeyPressContextInteractorStyle()
        self._contextInteractorStyleCurv.SetRenderWindow(self._contextViewPlotCurv.GetRenderWindow())

        self._chartXYCurv = vtk.vtkChartXY()
        self._contextViewPlotCurv.GetScene().AddItem(self._chartXYCurv)
        self._chartXYCurv.SetShowLegend(True)
        self._chartXYCurv.GetAxis(vtk.vtkAxis.LEFT).SetTitle("")
        self._chartXYCurv.GetAxis(vtk.vtkAxis.BOTTOM).SetTitle("")

        self._contextViewPlotTors = vtk.vtkContextView()
        self._contextViewPlotTors.GetRenderer().SetBackground(self.COLOR_BG_PLOT)

        self._contextInteractorStyleTors = self.KeyPressContextInteractorStyle()
        self._contextInteractorStyleTors.SetRenderWindow(self._contextViewPlotTors.GetRenderWindow())

        self._chartXYTors = vtk.vtkChartXY()
        self._contextViewPlotTors.GetScene().AddItem(self._chartXYTors)
        self._chartXYTors.SetShowLegend(True)
        self._chartXYTors.GetAxis(vtk.vtkAxis.LEFT).SetTitle("")
        self._chartXYTors.GetAxis(vtk.vtkAxis.BOTTOM).SetTitle("")

        self._textActor = vtk.vtkTextActor()
        self._textActor.GetTextProperty().SetColor(self.COLOR_LENGTH)

        self._addedBSpline = False

    def draw(self):
```

```
178         self._renderWindowInteractor.Initialize()
179         self._renderWindowInteractor.SetInteractorStyle(self._interactorStyle)
180 
181         axes = vtk.vtkAxesActor()
182         widget = vtk.vtkOrientationMarkerWidget()
183         widget.SetOutlineColor(0.9300, 0.5700, 0.1300)
184         widget.SetOrientationMarker(axes)
185         widget.SetInteractor(self._renderWindowInteractor)
186         #widget.SetViewport(0.0, 0.0, 0.1, 0.1)
187         widget.SetViewport(0.0, 0.0, 0.2, 0.4)
188         widget.SetEnabled(True)
189         widget.InteractiveOn()
190 
191         textWidget = vtk.vtkTextWidget()
192 
193         textRepresentation = vtk.vtkTextRepresentation()
194         textRepresentation.GetPositionCoordinate().SetValue(.0,.0 )
195         textRepresentation.GetPosition2Coordinate().SetValue(.3,.04 )
196         textWidget.SetRepresentation(textRepresentation)
197 
198         textWidget.SetInteractor(self._renderWindowInteractor)
199         textWidget.SetTextActor(self._textActor)
200         textWidget.SelectableOff()
201         textWidget.On()
202 
203         self._rendererScene.ResetCamera()
204         camPos = self._rendererScene.GetActiveCamera().GetPosition()
205         self._rendererScene.GetActiveCamera().SetPosition((camPos[2],camPos[1],camPos[0]))
206         self._rendererScene.GetActiveCamera().SetViewUp((0.0,0.0,1.0))
207         self._rendererScene.GetActiveCamera().Zoom(1.4)
208 
209         self._renderWindowScene.Render()
210 
211         if self._addedBSpline:
212             self._contextViewPlotCurv.GetRenderWindow().SetMultiSamples(0)
213             self._contextViewPlotCurv.GetInteractor().Initialize()
214             self._contextViewPlotCurv.GetInteractor().SetInteractorStyle(self._contextInteractorStyleCurv)
215             #self._contextViewPlotCurv.GetInteractor().Start()
216 
217             self._contextViewPlotTors.GetRenderWindow().SetMultiSamples(0)
218             self._contextViewPlotTors.GetInteractor().Initialize()
219             self._contextViewPlotTors.GetInteractor().SetInteractorStyle(self._contextInteractorStyleTors)
220             self._contextViewPlotTors.GetInteractor().Start()
221         else:
222             self._renderWindowInteractor.Start()
223 
224 
225     def addTetrahedron(self, vertexes, color):
226         vtkPoints = vtk.vtkPoints()
227         vtkPoints.InsertNextPoint(vertexes[0][0], vertexes[0][1], vertexes[0][2])
228         vtkPoints.InsertNextPoint(vertexes[1][0], vertexes[1][1], vertexes[1][2])
229         vtkPoints.InsertNextPoint(vertexes[2][0], vertexes[2][1], vertexes[2][2])
230         vtkPoints.InsertNextPoint(vertexes[3][0], vertexes[3][1], vertexes[3][2])
231 
232         unstructuredGrid = vtk.vtkUnstructuredGrid()
233         unstructuredGrid.SetPoints(vtkPoints)
234         unstructuredGrid.InsertNextCell(vtk.VTK_TETRA, 4, range(4))
235 
236         mapper = vtk.vtkDataSetMapper()
237         mapper.SetInputData(unstructuredGrid)
238 
239         actor = vtk.vtkActor()
240         actor.SetMapper(mapper)
241         actor.GetProperty().SetColor(color)
242 
243         self._rendererScene.AddActor(actor)
244 
245     def addTriangles(self, triangles, color):
246         vtkPoints = vtk.vtkPoints()
247         idPoint = 0
248         allIdsTriangle = []
249 
250         for triangle in triangles:
251             idsTriangle = []
252 
253             for point in triangle:
254                 vtkPoints.InsertNextPoint(point[0], point[1], point[2])
255                 idsTriangle.append(idPoint)
256                 idPoint += 1
257 
258             allIdsTriangle.append(idsTriangle)
259 
```

```
        unstructuredGrid = vtk.vtkUnstructuredGrid()
        unstructuredGrid.SetPoints(vtkPoints)
        for idsTriangle in allIdsTriangle:
            unstructuredGrid.InsertNextCell(vtk.VTK_TRIANGLE, 3, idsTriangle)

        mapper = vtk.vtkDataSetMapper()
        mapper.SetInputData(unstructuredGrid)

        actor = vtk.vtkActor()
        actor.SetMapper(mapper)
        actor.GetProperty().SetColor(color)

        self._rendererScene.AddActor(actor)

    def addPolyLine(self, points, color, thick=False, thickness=_DEFAULT_LINE_THICKNESS):
        vtkPoints = vtk.vtkPoints()
        for point in points:
            vtkPoints.InsertNextPoint(point[0], point[1], point[2])

        if thick:
            cellArray = vtk.vtkCellArray()
            cellArray.InsertNextCell(len(points))
            for i in range(len(points)):
                cellArray.InsertCellPoint(i)

            polyData = vtk.vtkPolyData()
            polyData.SetPoints(vtkPoints)
            polyData.SetLines(cellArray)

            tubeFilter = vtk.vtkTubeFilter()
            tubeFilter.SetNumberOfSides(8)
            tubeFilter.SetInputData(polyData)
            tubeFilter.SetRadius(thickness)
            tubeFilter.Update()

            mapper = vtk.vtkPolyDataMapper()
            mapper.SetInputConnection(tubeFilter.GetOutputPort())

        else:
            unstructuredGrid = vtk.vtkUnstructuredGrid()
            unstructuredGrid.SetPoints(vtkPoints)
            for i in range(1, len(points)):
                unstructuredGrid.InsertNextCell(vtk.VTK_LINE, 2, [i-1, i])

            mapper = vtk.vtkDataSetMapper()
            mapper.SetInputData(unstructuredGrid)

        actor = vtk.vtkActor()
        actor.SetMapper(mapper)
        actor.GetProperty().SetColor(color)

        self._rendererScene.AddActor(actor)

    def addPoints(self, points, color, thick=False, thickness=_DEFAULT_POINT_THICKNESS):
        vtkPoints = vtk.vtkPoints()
        for point in points:
            vtkPoints.InsertNextPoint(point[0], point[1], point[2])

        pointsPolyData = vtk.vtkPolyData()
        pointsPolyData.SetPoints(vtkPoints)

        if thick:
            sphereSource = vtk.vtkSphereSource()
            sphereSource.SetRadius(thickness)

            glyph3D = vtk.vtkGlyph3D()
            glyph3D.SetSourceConnection(sphereSource.GetOutputPort())
            glyph3D.SetInputData(pointsPolyData)
            glyph3D.Update()

            mapper = vtk.vtkPolyDataMapper()
            mapper.SetInputConnection(glyph3D.GetOutputPort())
        else:
            vertexFilter = vtk.vtkVertexGlyphFilter()
            vertexFilter.SetInputData(pointsPolyData)
            vertexFilter.Update()

            mapper = vtk.vtkPolyDataMapper()
            mapper.SetInputData(vertexFilter.GetOutput())

        actor = vtk.vtkActor()
        actor.SetMapper(mapper)
```

```
342         actor.GetProperty().SetColor(color)
343 
344         self._rendererScene.AddActor(actor)
345 
346     def addBSpline(self, path, degree, color, thick=False, thickness=_DEFAULT_BSPLINE_THICKNESS):
347         self._addedBSpline = True
348 
349         tau, u, spline, splineD1, splineD2, splineD3, curv, tors, arcLength, polLength = path.splinePoints()
350 
351         self._textActor.SetInput("Length: "+str(arcLength))
352 
353         numIntervals = len(tau)-1
354 
355         curvPlotActor = vtk.vtkXYPlotActor()
356         curvPlotActor.SetTitle("Curvature")
357         curvPlotActor.SetXTitle("")
358         curvPlotActor.SetYTitle("")
359         curvPlotActor.SetXValuesToIndex()
360 
361         torsPlotActor = vtk.vtkXYPlotActor()
362         torsPlotActor.SetTitle("Torsion")
363         torsPlotActor.SetXTitle("")
364         torsPlotActor.SetYTitle("")
365         torsPlotActor.SetXValuesToIndex()
366 
367         uArrays = []
368         curvArrays = []
369         torsArrays = []
370         for i in range(numIntervals):
371             uArrays.append(vtk.vtkDoubleArray())
372             uArrays[i].SetName("t")
373 
374             curvArrays.append(vtk.vtkDoubleArray())
375             curvArrays[i].SetName("Curvature")
376 
377             torsArrays.append(vtk.vtkDoubleArray())
378             torsArrays[i].SetName("Torsion")
379 
380         curvTorsArray = vtk.vtkDoubleArray()
381 
382         #minCurv = minTors = minNd1Xd2 = float("inf")
383         #maxCurv = maxTors = float("-inf")
384 
385         for i in range(len(u)):
386             for j in range(numIntervals):
387                 if u[i] >= tau[j] and u[i] < tau[j+1]:
388                     break
389 
390             uArrays[j].InsertNextValue(u[i])
391             curvArrays[j].InsertNextValue(curv[i])
392             torsArrays[j].InsertNextValue(tors[i])
393 
394             curvTorsArray.InsertNextValue(curv[i])# + abs(tors[i]))
395 
396         #print("minCurv: {:e}; maxCurv: {:e}; minTors: {:e}; maxTors: {:e}; minNd1Xd2: {:e}".format(minCurv, maxCurv, minTors
                ↪ , maxTors, minNd1Xd2))
397 
398         for inter in range(numIntervals):
399             plotTable = vtk.vtkTable()
400             plotTable.AddColumn(uArrays[inter])
401             plotTable.AddColumn(curvArrays[inter])
402             plotTable.AddColumn(torsArrays[inter])
403 
404             points = self._chartXYCurv.AddPlot(vtk.vtkChart.LINE)
405             points.SetInputData(plotTable, 0, 1)
406             points.SetColor(self.COLOR_PLOT_CURV[0], self.COLOR_PLOT_CURV[1], self.COLOR_PLOT_CURV[2])
407             points.SetWidth(1.0)
408             if inter > 0:
409                 points.SetLegendVisibility(False)
410 
411             points = self._chartXYTors.AddPlot(vtk.vtkChart.LINE)
412             points.SetInputData(plotTable, 0, 2)
413             points.SetColor(self.COLOR_PLOT_TORS[0], self.COLOR_PLOT_TORS[1], self.COLOR_PLOT_TORS[2])
414             points.SetWidth(1.0)
415             if inter > 0:
416                 points.SetLegendVisibility(False)
417 
418 
419         vtkPoints = vtk.vtkPoints()
420         for point in spline:
421             vtkPoints.InsertNextPoint(point[0], point[1], point[2])
422 
```

```
        polyDataLabelP = vtk.vtkPolyData()
        polyDataLabelP.SetPoints(vtkPoints)

        labels = vtk.vtkStringArray()
        labels.SetNumberOfValues(len(spline))
        labels.SetName("labels")
        for i in range(len(spline)):
            if i == 0:
                labels.SetValue(i, "S")
            elif i == len(spline)-1:
                labels.SetValue(i, "E")
            else:
                labels.SetValue(i, "")

        polyDataLabelP.GetPointData().AddArray(labels)

        sizes = vtk.vtkIntArray()
        sizes.SetNumberOfValues(len(spline))
        sizes.SetName("sizes")
        for i in range(len(spline)):
            if i == 0 or i == len(spline)-1:
                sizes.SetValue(i, 10)
            else:
                sizes.SetValue(i,1)

        polyDataLabelP.GetPointData().AddArray(sizes)

        pointMapper = vtk.vtkPolyDataMapper()
        pointMapper.SetInputData(polyDataLabelP)

        pointActor = vtk.vtkActor()
        pointActor.SetMapper(pointMapper)

        pointSetToLabelHierarchyFilter = vtk.vtkPointSetToLabelHierarchy()
        pointSetToLabelHierarchyFilter.SetInputData(polyDataLabelP)
        pointSetToLabelHierarchyFilter.SetLabelArrayName("labels")
        pointSetToLabelHierarchyFilter.SetPriorityArrayName("sizes")
        pointSetToLabelHierarchyFilter.GetTextProperty().SetColor(self.COLOR_LABELS)
        pointSetToLabelHierarchyFilter.GetTextProperty().SetFontSize(15)
        pointSetToLabelHierarchyFilter.GetTextProperty().SetBold(True)
        pointSetToLabelHierarchyFilter.Update()

        labelMapper = vtk.vtkLabelPlacementMapper()
        labelMapper.SetInputConnection(pointSetToLabelHierarchyFilter.GetOutputPort())
        labelActor = vtk.vtkActor2D()
        labelActor.SetMapper(labelMapper)

        self._rendererScene.AddActor(labelActor)

        if thick:
            cellArray = vtk.vtkCellArray()
            cellArray.InsertNextCell(len(spline))
            for i in range(len(spline)):
                cellArray.InsertCellPoint(i)

            polyData = vtk.vtkPolyData()
            polyData.SetPoints(vtkPoints)
            polyData.SetLines(cellArray)

            polyData.GetPointData().SetScalars(curvTorsArray)

            tubeFilter = vtk.vtkTubeFilter()
            tubeFilter.SetNumberOfSides(8)
            tubeFilter.SetInputData(polyData)
            tubeFilter.SetRadius(thickness)
            tubeFilter.Update()

            mapper = vtk.vtkPolyDataMapper()
            mapper.SetInputConnection(tubeFilter.GetOutputPort())

        else:
            unstructuredGrid = vtk.vtkUnstructuredGrid()
            unstructuredGrid.SetPoints(vtkPoints)
            for i in range(1, len(spline)):
                unstructuredGrid.InsertNextCell(vtk.VTK_LINE, 2, [i-1, i])

            unstructuredGrid.GetPointData().SetScalars(curvArray)

            mapper = vtk.vtkDataSetMapper()
            mapper.SetInputData(unstructuredGrid)

        actor = vtk.vtkActor()
```

```
505         actor.SetMapper(mapper)
506         actor.GetProperty().SetColor(color)
507 
508         self._rendererScene.AddActor(actor)
509 
510 
511         #self.addPolyLine(list(zip(out[0], out[1], out[2])), color, thick, thickness)
512 
513     def addBSplineDEPRECATED(self, controlPolygon, degree, color, thick=False, thickness=_DEFAULT_BSPLINE_THICKNESS):
514         x = controlPolygon[:,0]
515         y = controlPolygon[:,1]
516         z = controlPolygon[:,2]
517 
518         polLen = 0.
519         for i in range(1, len(controlPolygon)):
520             polLen += sp.spatial.distance.euclidean(controlPolygon[i-1], controlPolygon[i])
521 
522         t = range(len(controlPolygon))
523         ipl_t = np.linspace(0.0, len(controlPolygon) - 1, max(polLen*100,100))
524 
525         x_tup = sp.interpolate.splrep(t, x, k = degree)
526         y_tup = sp.interpolate.splrep(t, y, k = degree)
527         z_tup = sp.interpolate.splrep(t, z, k = degree)
528 
529         x_list = list(x_tup)
530         xl = x.tolist()
531         x_list[1] = xl + [0.0, 0.0, 0.0, 0.0]
532 
533         y_list = list(y_tup)
534         yl = y.tolist()
535         y_list[1] = yl + [0.0, 0.0, 0.0, 0.0]
536 
537         z_list = list(z_tup)
538         zl = z.tolist()
539         z_list[1] = zl + [0.0, 0.0, 0.0, 0.0]
540 
541         x_i = sp.interpolate.splev(ipl_t, x_list)
542         y_i = sp.interpolate.splev(ipl_t, y_list)
543         z_i = sp.interpolate.splev(ipl_t, z_list)
544 
545         self.addPolyLine(list(zip(x_i, y_i, z_i)), color, thick, thickness)
546 
547     def addGraph(self, graph, color):
548         vtkPoints = vtk.vtkPoints()
549         vtkId = 0
550         graph2VtkId = {}
551 
552         for node in graph.nodes():
553             vtkPoints.InsertNextPoint(graph.node[node]['coord'][0], graph.node[node]['coord'][1], graph.node[node]['coord'
                    ↪ ][2])
554             graph2VtkId[node] = vtkId
555             vtkId += 1
556 
557         unstructuredGrid = vtk.vtkUnstructuredGrid()
558         unstructuredGrid.SetPoints(vtkPoints)
559 
560         for edge in graph.edges():
561             unstructuredGrid.InsertNextCell(vtk.VTK_LINE, 2, [graph2VtkId[edge[0]], graph2VtkId[edge[1]]])
562 
563         mapper = vtk.vtkDataSetMapper()
564         mapper.SetInputData(unstructuredGrid)
565 
566         actor = vtk.vtkActor()
567         actor.SetMapper(mapper)
568         actor.GetProperty().SetColor(color)
569 
570         self._rendererScene.AddActor(actor)
```

A.1.4 *polyhedronsContainer.py*

```
1 import numpy as np
2 import scipy as sp
3 import scipy.spatial
4 import polyhedron
5 import parallelepiped
6 
```

```
class PolyhedronsContainer:
    def __init__(self):
        self._polyhedrons = []
        self._hasBoundingBox = False
        self._boundingBoxA = None
        self._boundingBoxB = None

    @property
    def polyhedrons(self):
        return self._polyhedrons

    @property
    def hasBoundingBox(self):
        return self._hasBoundingBox

    @hasBoundingBox.setter
    def hasBoundingBox(self, value):
        self._hasBoundingBox = value

    @property
    def boundingBoxA(self):
        return self._boundingBoxA

    @boundingBoxA.setter
    def boundingBoxA(self, value):
        self._boundingBoxA = value

    @property
    def boundingBoxB(self):
        return self._boundingBoxB

    @boundingBoxB.setter
    def boundingBoxB(self, value):
        self._boundingBoxB = value

    def addPolyhedron(self, polyhedron):
        self._polyhedrons.append(polyhedron)

    def addBoundingBox(self, a, b, maxEmptyArea, invisible):
        self._hasBoundingBox = True
        self._boundingBoxA = a
        self._boundingBoxB = b

        self.addPolyhedron(parallelepiped.Parallelepiped(a=a, b=b, invisible=invisible, maxEmptyArea=maxEmptyArea,
            ↪ boundingBox=True))

    def pointInsidePolyhedron(self, p):
        inside = False
        if self._hasBoundingBox:
            if (p<self._boundingBoxA).any() or (p>self._boundingBoxB).any():
                inside = True

        if not inside:
            for polyhedron in self._polyhedrons:
                if (not polyhedron.isBoundingBox()) and polyhedron.hasPointInside(p):
                    inside = True
                    break

        return inside


    def segmentIntersectPolyhedrons(self, a, b, intersectionMargin = 0.):
        intersect = False
        if self._hasBoundingBox:
            if((a<self._boundingBoxA).any() or (a>self._boundingBoxB).any() or (b<self._boundingBoxA).any() or (b>self.
                ↪ _boundingBoxB).any()):
                intersect = True

        if not intersect:
            minS = np.array([min(a[0],b[0]),min(a[1],b[1]),min(a[2],b[2])])
            maxS = np.array([max(a[0],b[0]),max(a[1],b[1]),max(a[2],b[2])])

            for polyhedron in self._polyhedrons:
                if polyhedron.intersectSegment(a,b,minS,maxS, intersectionMargin=intersectionMargin)[0]:
                    intersect = True
                    break

        return intersect

    def triangleIntersectPolyhedrons(self, a, b, c):
        triangle = polyhedron.Polyhedron(faces=np.array([[a,b,c]]), distributePoints = False)
        intersect = False
```

```
        result = np.array([])
        for currPolyhedron in self._polyhedrons:
            currIntersect,currResult = currPolyhedron.intersectPathTriple(triangle)
            if currIntersect and (not intersect or (currResult[1] > result[1])):
                intersect = True
                result = currResult

        return (intersect, result)

    def convexHullIntersectsPolyhedrons(self, vertexes):
        convHull = sp.spatial.ConvexHull(vertexes, qhull_options="QJ Pp")
        for simplex in convHull.simplices:
            if self.triangleIntersectPolyhedrons(convHull.points[simplex[0]], convHull.points[simplex[1]], convHull.points[
                ↪ simplex[2]])[0]:
                return True

        return False
```

A.1.5 *polyhedron.py*

```
import numpy as np
import scipy as sp
import scipy.spatial
import math
import xml.etree.cElementTree as ET

class Polyhedron:
    def __init__(self, faces, invisible=False, distributePoints=True, maxEmptyArea=0.1, boundingBox=False):
        """
        can be composed only by combined triangles
        faces -> an np.array of triangular faces
        if invisible=True when plot will be called it will be useless
        """
        self._faces = faces
        self._invisible = invisible
        self._boundingBox = boundingBox

        self._minV = np.array([float('inf'),float('inf'),float('inf')])
        self._maxV = np.array([float('-inf'),float('-inf'),float('-inf')])
        for face in self._faces:
            for vertex in face:
                for i in range(len(vertex)):
                    if vertex[i] < self._minV[i]:
                        self._minV[i] = vertex[i]

                for i in range(len(vertex)):
                    if vertex[i] > self._maxV[i]:
                        self._maxV[i] = vertex[i]

        if distributePoints:
            self.distributePoints(maxEmptyArea)
        else:
            self._allPoints = np.array([])

    @property
    def allPoints(self):
        return self._allPoints

    @property
    def minV(self):
        return self._minV

    @property
    def maxV(self):
        return self._maxV

    def isBoundingBox(self):
        return self._boundingBox

    def _area(self, triangle):
        a = np.linalg.norm(triangle[1]-triangle[0])
        b = np.linalg.norm(triangle[2]-triangle[1])
        c = np.linalg.norm(triangle[0]-triangle[2])
        s =  (a+b+c) / 2.
        return math.sqrt(s * (s-a) * (s-b) *(s-c))

```

```
    _comb2 = lambda self,a,b: 0.5*a + 0.5*b
    #_comb3 = lambda self,a,b,c: 0.33*a + 0.33*b + 0.33*c

    def distributePoints(self, maxEmptyArea):
        allPoints = []
        triangles = []

        for face in self._faces:
            triangles.append(face)

        while triangles:
            triangle = triangles.pop(0)
            a = triangle[0]
            b = triangle[1]
            c = triangle[2]
            if not any((a == x).all() for x in allPoints):
                allPoints.append(a)
            if not any((b == x).all() for x in allPoints):
                allPoints.append(b)
            if not any((c == x).all() for x in allPoints):
                allPoints.append(c)
            if (self._area(triangle) > maxEmptyArea):
                ab = self._comb2(a,b)
                bc = self._comb2(b,c)
                ca = self._comb2(c,a)
                #abc = self._comb3(a,b,c)

                triangles.append(np.array([a,ab,ca]))
                triangles.append(np.array([ab,b,bc]))
                triangles.append(np.array([bc,c,ca]))
                triangles.append(np.array([ab,bc,ca]))
                #triangles.append(np.array([a,ab,abc]))
                #triangles.append(np.array([ab,b,abc]))
                #triangles.append(np.array([b,bc,abc]))
                #triangles.append(np.array([bc,c,abc]))
                #triangles.append(np.array([c,ca,abc]))
                #triangles.append(np.array([ca,a,abc]))

        self._allPoints = np.array(allPoints)

    def hasPointInside(self, p):
        """
        check if a point is inside the convex hull of obstacle vertexes
        """
        outside = True
        if (p>self._minV).all() and (p<self._maxV).all():
            vertexes = [p]
            for triangle in self._faces:
                vertexes.append(triangle[0])
                vertexes.append(triangle[1])
                vertexes.append(triangle[2])

            chull = sp.spatial.ConvexHull(np.array(vertexes))
            outside = False
            for vertex in chull.vertices:
                if (p == chull.points[vertex]).all():
                    outside = True
                    break
            # for simplex in chull.simplices:
            #     if (p == chull.points[simplex[0]]).all() or (p == chull.points[simplex[1]]).all() or (p == chull.points[
                    ↪ simplex[2]]).all():
            #         outside = True
            #         break

        return not outside

    def intersectSegment(self, a, b, minS=None, maxS=None, intersectionMargin=0.):
        if minS is None or maxS is None:
            minS = np.array([min(a[0],b[0]),min(a[1],b[1]),min(a[2],b[2])])
            maxS = np.array([max(a[0],b[0]),max(a[1],b[1]),max(a[2],b[2])])

        if not ((self._minV > maxS).any() or (self._maxV < minS).any()):
            for triangle in self._faces:
                #solve {
                #        a+k(b-a) = v*triangle[0] + w*triangle[1] + s*triangle[2]
                #        v+w+s = 1
                #      }
                # for variables k, v, w, s

                #simplified in
                #      a+k(b-a) = (1-w-s)*triangle[0] + w*triangle[1] + s*triangle[2]
                # for variables k, w, s
```

```
                diffba = b-a
                difft0t1 = triangle[0] - triangle[1]
                difft0t2 = triangle[0] - triangle[2]
                difft0a = triangle[0] - a

                A = np.array([
                    [diffba[0], difft0t1[0], difft0t2[0]],
                    [diffba[1], difft0t1[1], difft0t2[1]],
                    [diffba[2], difft0t1[2], difft0t2[2]]])
                B = np.array([difft0a[0], difft0a[1], difft0a[2]])

                try:
                    x = np.linalg.solve(A,B)
                    # check (with margins) if
                    #          0 < k < 1,
                    #          w > 0
                    #          s > 0
                    #          w+s < 1
                    if (x[0] >= 0. - intersectionMargin) and (x[0] <= 1. + intersectionMargin) and (x[1] >= 0. -
                        ↪ intersectionMargin) and (x[2] >= 0. - intersectionMargin) and (x[1]+x[2] <= 1. +
                        ↪ intersectionMargin):
                            return (True, x)
                except np.linalg.linalg.LinAlgError:
                    pass

        return (False,np.array([]))

    def intersectPolyhedron(self, polyhedron):
        """alert, not case of one polyhedron inside other"""
        if not ((self._minV > polyhedron.maxV).any() or (self._maxV < polyhedron.minV).any()):
            for otherFace in polyhedron._faces:
                for myFace in self._faces:
                    if (
                            self.intersectSegment(otherFace[0],otherFace[1])[0] or
                            self.intersectSegment(otherFace[1],otherFace[2])[0] or
                            self.intersectSegment(otherFace[2],otherFace[0])[0] or
                            polyhedron.intersectSegment(myFace[0], myFace[1])[0] or
                            polyhedron.intersectSegment(myFace[1], myFace[2])[0] or
                            polyhedron.intersectSegment(myFace[2], myFace[0])[0]):
                        return True
        return False

    def intersectPathTriple(self, triple):
        """alert, not case of one polyhedron inside other, and only
        check if the segments of self intersect the triple."""
        intersect = False
        result = np.array([])

        if not ((self._minV > triple.maxV).any() or (self._maxV < triple.minV).any()):
            for myFace in self._faces:
                intersect1, result1 = triple.intersectSegment(myFace[0], myFace[1])
                intersect2, result2 = triple.intersectSegment(myFace[1], myFace[2])
                intersect3, result3 = triple.intersectSegment(myFace[2], myFace[0])
                if intersect1:
                    intersect = True
                    result = result1
                if intersect2 and (not intersect or (result2[1] > result[1])):
                    intersect = True
                    result = result2
                if intersect3 and (not intersect or (result3[1] > result[1])):
                    intersect = True
                    result = result3

        return intersect,result


    def plotAllPoints(self, plotter):
        if self._allPoints.size > 0:
            plotter.addPoints(self._allPoints, plotter.COLOR_SITES)

    def plot(self, plotter):
        if self._invisible == False:
            plotter.addTriangles(self._faces, plotter.COLOR_OBSTACLE)

    def extractXmlTree(self, root):
        xmlPolyhedron = ET.SubElement(root, 'polyhedron', invisible=str(self._invisible), boundingBox=str(self._boundingBox))
        for face in self._faces:
            xmlFace = ET.SubElement(xmlPolyhedron, 'face')
            for vertex in face:
                xmlVertex = ET.SubElement(xmlFace, 'vertex', x=str(vertex[0]), y=str(vertex[1]), z=str(vertex[2]))
```

A.1.6 *compositePolyhedron.py*

```
import numpy as np
import polyhedron

class CompositePolyhedron(polyhedron.Polyhedron):
    def __init__(self, components):
        self._components = components

        self._boundingBox = False

        self._minV = np.array([float('inf'),float('inf'),float('inf')])
        self._maxV = np.array([float('-inf'),float('-inf'),float('-inf')])

        for component in self._components:
            for i in range(3):
                if component.minV[i] < self._minV[i]:
                    self._minV[i] = component.minV[i]

                if component.maxV[i] > self._maxV[i]:
                    self._maxV[i] = component.maxV[i]

    @property
    def allPoints(self):
        allPoints = []
        for component in self._components:
            allPoints.extend(list(component.allPoints))

        return np.array(allPoints)

    @property
    def minV(self):
        return self._minV

    @property
    def maxV(self):
        return self._maxV


    def distributePoints(self, maxEmptyArea):
        for component in self._components:
            component.distributePoints(maxEmptyArea)

    def hasPointInside(self, p):
        hasPI = False
        for component in self._components:
            if component.hasPointInside(p):
                hasPI = True
                break

        return hasPI

    def intersectSegment(self, a, b, minS=None, maxS=None, intersectionMargin=0.):
        intersect = (False,np.array([]))
        for component in self._components:
            current = component.intersectSegment(a,b,minS,maxS,intersectionMargin)
            if current[0]:
                intersect = current
                break

        return intersect

    def intersectPolyhedron(self, polyhedron):
        intersect = False
        for component in self._components:
            if component.intersectPolyhedron(polyhedron):
                intersect = True
                break

        return intersect

    def intersectPathTriple(self, triple):
        intersect = (False, np.array([]))
        for component in self._components:
            current = component.intersectPathTriple(triple)
            if current[0]:
                intersect = current
                break

        return intersect
```

```
79
80      def plotAllPoints(self, plotter):
81          for component in self._components:
82              component.plotAllPoints(plotter)
83
84      def plot(self, plotter):
85          for component in self._components:
86              component.plot(plotter)
87
88      def extractXmlTree(self, root):
89          for component in self._components:
90              component.extractXmlTree(root)
```

A.1.7 *tetrahedron.py*

```
1  import numpy as np
2  import polyhedron
3
4  class Tetrahedron(polyhedron.Polyhedron):
5      def __init__(self, a, b, c, d, invisible=False, distributePoints=True, maxEmptyArea=0.1):
6          super(Tetrahedron, self).__init__(np.array([[a,b,c],[a,b,d],[b,c,d],[c,a,d]]), invisible, distributePoints,
               ↪ maxEmptyArea)
7
8          self._vertexes = [a,b,c,d]
9
10     def plot(self, plotter):
11         if self._invisible == False:
12             plotter.addTetrahedron(self._vertexes, plotter.COLOR_OBSTACLE)
```

A.1.8 *parallelepiped.py*

```
1  import numpy as np
2  import polyhedron
3
4  class Parallelepiped(polyhedron.Polyhedron):
5      def __init__(self, a, b, invisible=False, distributePoints=True, maxEmptyArea=0.1, boundingBox=False):
6
7          c = [a[0], b[1], a[2]]
8          d = [b[0], a[1], a[2]]
9          e = [a[0], a[1], b[2]]
10         f = [b[0], b[1], a[2]]
11         g = [b[0], a[1], b[2]]
12         h = [a[0], b[1], b[2]]
13
14         super(Parallelepiped, self).__init__(faces=np.array([
15             [a,g,e],[a,d,g],[d,f,g],[f,b,g],[f,b,h],[f,h,c],
16             [h,a,e],[h,c,a],[e,h,g],[h,b,g],[a,d,f],[a,f,c]
17             ]), invisible=invisible, distributePoints=distributePoints, maxEmptyArea=maxEmptyArea, boundingBox=boundingBox)
```

A.1.9 *convexHull.py*

```
1  import numpy as np
2  import scipy as sp
3  import scipy.spatial
4  import polyhedron
5
6  class ConvexHull(polyhedron.Polyhedron):
7      def __init__(self, points, invisible=False, distributePoints=True, maxEmptyArea=0.1):
8          convHull = sp.spatial.ConvexHull(points)
9          faces = []
10         for simplex in convHull.simplices:
11             faces.append([convHull.points[simplex[0]], convHull.points[simplex[1]], convHull.points[simplex[2]]])
12
13         super(ConvexHull, self).__init__(np.array(faces), invisible, distributePoints, maxEmptyArea)
```

A.1.10 *bucket.py*

```
import numpy as np
import compositePolyhedron
import parallelepiped

class Bucket(compositePolyhedron.CompositePolyhedron):
    def __init__(self, center, width, height, thickness, invisible=False, distributePoints=True, maxEmptyArea=0.1,
        ↪ boundingBox=False):
        c = center
        l = width
        h = height
        d = thickness
        parallelepipeds = []

        parallelepipeds.append(parallelepiped.Parallelepiped(
            np.array([c[0]-(l/2), c[1]-(l/2), c[2]-(h/2)]),\
            np.array([c[0]+(l/2), c[1]+(l/2), c[2]-(h/2)+d]), invisible, distributePoints, maxEmptyArea, boundingBox))

        parallelepipeds.append(parallelepiped.Parallelepiped(
            np.array([c[0]-(l/2), c[1]+(l/2)-d, c[2]-(h/2)+d]),\
            np.array([c[0]+(l/2), c[1]+(l/2), c[2]+(h/2)]), invisible, distributePoints, maxEmptyArea, boundingBox))

        parallelepipeds.append(parallelepiped.Parallelepiped(
            np.array([c[0]-(l/2), c[1]-(l/2), c[2]-(h/2)+d]),\
            np.array([c[0]+(l/2), c[1]-(l/2)+d, c[2]+(h/2)]), invisible, distributePoints, maxEmptyArea, boundingBox))

        parallelepipeds.append(parallelepiped.Parallelepiped(
            np.array([c[0]-(l/2), c[1]-(l/2)+d, c[2]-(h/2)+d]),\
            np.array([c[0]-(l/2)+d, c[1]+(l/2)-d, c[2]+(h/2)]), invisible, distributePoints, maxEmptyArea, boundingBox))

        parallelepipeds.append(parallelepiped.Parallelepiped(
            np.array([c[0]+(l/2)-d, c[1]-(l/2)+d, c[2]-(h/2)+d]),\
            np.array([c[0]+(l/2), c[1]+(l/2)-d, c[2]+(h/2)]), invisible, distributePoints, maxEmptyArea, boundingBox))

        super(Bucket, self).__init__(parallelepipeds)
```

A.2 SCRIPTS

A.2.1 *makeRandomScene.py*

```
#!/bin/python

import sys
import numpy as np
import random
import math
import pickle
import voronizator
import tetrahedron

if len(sys.argv) >= 14 and len(sys.argv) <= 15:
    i = 1
    minX = float(sys.argv[i])
    i += 1
    minY = float(sys.argv[i])
    i += 1
    minZ = float(sys.argv[i])
    i += 1
    maxX = float(sys.argv[i])
    i += 1
    maxY = float(sys.argv[i])
    i += 1
    maxZ = float(sys.argv[i])
    i += 1
    bbMargin = float(sys.argv[i])
    i += 1
    fixedRadius = bool(eval(sys.argv[i]))
    i += 1
    if fixedRadius:
        radius = float(sys.argv[i])
```

```
        i += 1
    else:
        minRadius = float(sys.argv[i])
        i += 1
        maxRadius = float(sys.argv[i])
        i += 1
    avoidCollisions = bool(eval(sys.argv[i]))
    i += 1
    numObstacles = int(sys.argv[i])
    i += 1
    maxEmptyArea = float(sys.argv[i])
    i += 1
    fileName = sys.argv[i]

else:
    minX = float(input('Insert min scene X: '))
    minY = float(input('Insert min scene Y: '))
    minZ = float(input('Insert min scene Z: '))
    maxX = float(input('Insert max scene X: '))
    maxY = float(input('Insert max scene Y: '))
    maxZ = float(input('Insert max scene Z: '))
    bbMargin = float(input('Insert bounding box margin: '))
    fixedRadius = bool(eval(input('Do you want fixed obstacle radius? (True/False): ')))
    if fixedRadius:
        radius = float(input('Insert obstacle radius: '))
    else:
        minRadius = float(input('Insert min obstacle radius: '))
        maxRadius = float(input('Insert max obstacle radius: '))
    avoidCollisions = bool(eval(input('Do you want to avoid collisions between obstacles? (True/False): ')))
    numObstacles = int(input('Insert obstacles number: '))
    maxEmptyArea = float(input('Insert max empty area (for points distribution in obstacles): '))

    fileName = input('Insert file name: ')

voronoi = voronizator.Voronizator()
obstacles = []

for ob in range(numObstacles):
    print('Creating obstacle {} '.format(ob+1), end='', flush=True)
    ok = False
    while not ok:
        print('.', end='', flush=True)
        if not fixedRadius:
            radius = random.uniform(minRadius,maxRadius)
        center = np.array([random.uniform(minX+radius,maxX-radius), random.uniform(minY+radius,maxY-radius), random.uniform(
            ↪ minZ+radius,maxZ-radius)])
        points = []
        for pt in range(4):
            elev = random.uniform(-math.pi/2., math.pi/2.)
            azim = random.uniform(0., 2.*math.pi)
            points[:0] = [center+np.array([
                radius*math.cos(elev)*math.cos(azim),
                radius*math.cos(elev)*math.sin(azim),
                radius*math.sin(elev)])]

        newObstacle = tetrahedron.Tetrahedron(a = points[0], b = points[1], c = points[2], d = points[3], distributePoints =
            ↪ True, maxEmptyArea = maxEmptyArea)

        ok = True
        if avoidCollisions:
            for obstacle in obstacles:
                if newObstacle.intersectPolyhedron(obstacle):
                    ok = False
                    break

        if ok:
            voronoi.addPolyhedron(newObstacle)
            if avoidCollisions:
                obstacles[:0] = [newObstacle]

    print(' done', flush=True)

voronoi.addBoundingBox([minX-bbMargin, minY-bbMargin, minZ-bbMargin], [maxX+bbMargin, maxY+bbMargin, maxZ+bbMargin],
    ↪ maxEmptyArea, verbose=True)

voronoi.setPolyhedronsSites(verbose=True)
voronoi.makeVoroGraph(verbose=True)

print('Write file', flush=True)
record = {}
record['voronoi'] = voronoi

```

```
with open(fileName, 'wb') as f:
    pickle.dump(record, f)
```

A.2.2 *makeBucketScene.py*

```
#!/bin/python

import sys
import numpy as np
import random
import math
import pickle
import voronizator
import bucket

if len(sys.argv) == 9:
    i = 1
    minPoint = np.array(tuple(eval(sys.argv[i])),dtype=float)
    i += 1
    maxPoint = np.array(tuple(eval(sys.argv[i])),dtype=float)
    i += 1
    center = np.array(tuple(eval(sys.argv[i])),dtype=float)
    i += 1
    width = float(sys.argv[i])
    i += 1
    height = float(sys.argv[i])
    i += 1
    thickness = float(sys.argv[i])
    i += 1
    maxEmptyArea = float(sys.argv[i])
    i += 1
    fileName = sys.argv[i]

else:
    minPoint = np.array(tuple(eval(input('Insert min point (x,y,z): '))),dtype=float)
    maxPoint = np.array(tuple(eval(input('Insert max point (x,y,z): '))),dtype=float)
    center = np.array(tuple(eval(input('Insert bucket center point (x,y,z): '))),dtype=float)
    width = float(input('Insert bucket width: '))
    height = float(input('Insert bucket height: '))
    thickness = float(input('Insert bucket thickness: '))
    maxEmptyArea = float(input('Insert max empty area (for points distribution in obstacles): '))
    fileName = input('Insert file name: ')

voronoi = voronizator.Voronizator()

print('Create bucket', flush=True)
voronoi.addPolyhedron(bucket.Bucket(center, width, height, thickness, distributePoints=True, maxEmptyArea=maxEmptyArea))
voronoi.addBoundingBox(minPoint, maxPoint, maxEmptyArea, verbose=True)
voronoi.setPolyhedronsSites(verbose=True)
voronoi.makeVoroGraph(verbose=True)

print('Write file', flush=True)
record = {}
record['voronoi'] = voronoi

with open(fileName, 'wb') as f:
    pickle.dump(record, f)
```

A.2.3 *plotScene.py*

```
#!/bin/python

import sys
import pickle
import plotter

if len(sys.argv) >= 2:
    if len(sys.argv) == 4:
        i = 2
        plotSites = bool(eval(sys.argv[i]))
```

```
        i += 1
        plotGraph = bool(eval(sys.argv[i]))
    else:
        plotSites = bool(eval(input('Do you want to plot Voronoi sites? (True/False): ')))
        plotGraph = bool(eval(input('Do you want to plot graph? (True/False): ')))


    print('Load file', flush=True)
    with open(sys.argv[1], 'rb') as f:
        record = pickle.load(f)

    voronoi = record['voronoi']

    print('Build renderer, window and interactor', flush=True)
    plt = plotter.Plotter()

    voronoi.plotPolyhedrons(plt, verbose = True)
    if plotSites:
        voronoi.plotSites(plt, verbose = True)
    if plotGraph:
        voronoi.plotGraph(plt, verbose = True)

    print('Render', flush=True)
    plt.draw()

else:
    print('use: {} sceneFile [plotSites plotGraph]'.format(sys.argv[0]))
```

A.2.4 *executeInScene.py*

```
#!/bin/python

import sys
import numpy as np
import pickle
import plotter

if len(sys.argv) >= 2:
    if len(sys.argv) == 8:
        i = 2
        startPoint = np.array(tuple(eval(sys.argv[i])),dtype=float)
        i += 1
        endPoint = np.array(tuple(eval(sys.argv[i])),dtype=float)
        i += 1
        bsplineDegree = int(sys.argv[i])
        i += 1
        useMethod = str(sys.argv[i])
        i += 1
        postSimplify = bool(eval(sys.argv[i]))
        i += 1
        adaptivePartition = bool(eval(sys.argv[i]))
    else:
        startPoint = np.array(tuple(eval(input('Insert start point (x,y,z): '))),dtype=float)
        endPoint = np.array(tuple(eval(input('Insert end point (x,y,z): '))),dtype=float)
        bsplineDegree = int(input('Insert B-spline degree (2/3/4): '))
        useMethod = str(input('Wich method you want to use? (none/trijkstra/cleanPath/annealing): '))
        postSimplify = bool(eval(input('Do you want post processing? (True/False): ')))
        adaptivePartition = bool(eval(input('Do you want adaptive partition? (True/False): ')))

    print('Load file', flush=True)
    with open(sys.argv[1], 'rb') as f:
        record = pickle.load(f)

    voronoi = record['voronoi']
    voronoi.setBsplineDegree(bsplineDegree)
    voronoi.setAdaptivePartition(adaptivePartition)

    voronoi.calculateShortestPath(startPoint, endPoint, 'near', useMethod=useMethod, postSimplify=postSimplify, verbose=True,
        ↪  debug=False)

    print('Build renderer, window and interactor', flush=True)
    plt = plotter.Plotter()

    #voronoi.plotSites(plt, verbose = True)
    voronoi.plotPolyhedrons(plt, verbose = True)
    #voronoi.plotGraph(plt, verbose = True)
```

```
    voronoi.plotShortestPath(plt, verbose = True)

    print('Render', flush=True)
    plt.draw()

else:
    print('use: {} sceneFile [startPoint endPoint degree(2,4) useMethod postProcessing adaptivePartition]'.format(sys.argv
        ↪ [0]))
```

A.2.5 *scene2coord.py*

```
#!/bin/python

import sys
import pickle
import xml.etree.cElementTree as ET

if len(sys.argv) == 3:

    print('Load file', flush=True)
    with open(sys.argv[1], 'rb') as f:
        record = pickle.load(f)

    voronoi = record['voronoi']

    print('Create XML', flush=True)
    xmlRoot = ET.Element('scene')
    voronoi.extractXmlTree(xmlRoot)
    xmlTree = ET.ElementTree(xmlRoot)

    print('Write file', flush=True)
    xmlTree.write(sys.argv[2])

else:
    print('use: {} sceneFile coordinateFile'.format(sys.argv[0]))
```

A.2.6 *coord2scene.py*

```
#!/bin/python

import sys
import pickle
import xml.etree.cElementTree as ET
import voronizator

if len(sys.argv) == 4:
    xmlFileName = sys.argv[1]
    sceneFileName = sys.argv[2]
    maxEmptyArea = float(sys.argv[3])

    xmlRoot = ET.parse(xmlFileName).getroot()

    voronoi = voronizator.Voronizator()

    print('Import XML', flush=True)
    voronoi.importXmlTree(xmlRoot, maxEmptyArea)

    print('Set sites and make graph', flush=True)
    voronoi.setPolyhedronsSites(verbose=True)
    voronoi.makeVoroGraph(verbose=True)

    print('Write file', flush=True)
    record = {}
    record['voronoi'] = voronoi
    with open(sceneFileName, 'wb') as f:
        pickle.dump(record, f)

else:
    print('use: {} coordinateFile sceneFile maxEmptyArea'.format(sys.argv[0]))
```

BIBLIOGRAPHY

[1] Adrian Bondy, U. M. (2008). *Graph theory*. Graduate texts in mathematics 244. Springer, 3rd corrected printing. edition. (Cited on pages 53 and 58.)

[2] Barkema, M. and Newman, G. (1999). *Monte Carlo Methods in Statistical Physics*. Oxford Uinversity Press. (Cited on page 24.)

[3] Bertsekas, D. P. (1999). *Nonlinear programming*. Athena Scientific, 2nd edition. (Cited on page 30.)

[4] Bhattacharya, P. and Gavrilova, M. L. (2008). Roadmap-based path planning - using the voronoi diagram for a clearance-based shortest path. *IEEE Robot. Automat. Mag.*, 15(2):58–66. (Cited on page 44.)

[5] Canny, J. F. (1988). *Complexity of Robot Motion Planning*. ACM Doctoral Dissertation Award. The MIT Press. (Cited on page 8.)

[6] Choset H., e. a. (2005). *Principles of Robot Motion: Theory, Algorithms, and Implementations*. Intelligent Robotics and Autonomous Agents series. MIT. (Cited on pages 5, 7, and 9.)

[7] de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. (2008). *Computational Geometry Algorithms and Applications*. Springer-Verlag, Berlin Heidelberg, 3 edition. (Cited on pages 7, 21, 22, 31, 32, 41, and 46.)

[8] de Boor, C. (1978). *A Practical Guide to Splines*. Springer-Verlag. (Cited on pages 13, 15, 16, and 19.)

[9] Dijkstra, E. (1959). A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271. (Cited on pages 7, 46, and 47.)

[10] do Carmo, M. P. (1976). *Differential geometry of curves and surfaces.* Prentice Hall. (Cited on pages 20 and 61.)

[11] Farin, G. E. (1990). *Curves and surfaces for computer aided geometric design - a practical guide (2.ed.).* Computer science and scientific computing. Academic Press. (Cited on pages 13, 15, 17, 19, and 61.)

[12] Fortune, S. (1986). A sweepline algorithm for voronoi diagrams. In Aggarwal, A., editor, *Symposium on Computational Geometry*, pages 313–322. ACM. (Cited on pages 22 and 44.)

[13] Giannelli, C., Mugnaini, D., and Sestini, A. (2016). Path planning with obstacle avoidance by G^1 PH quintic splines. *Computer-Aided Design*, 75–76:47 – 60. (Cited on page 1.)

[14] Goerzen, C., Kong, Z., and Mettler, B. (2009). A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57(1):65–100. (Cited on pages 5, 6, and 7.)

[15] Ho, Y.-J. and Liu, J.-S. (2009). Collision-free curvature-bounded smooth path planning using composite bezier curve based on voronoi diagram. In *CIRA*, pages 463–468. IEEE. (Cited on page 44.)

[16] Ho, Y.-J. and Liu, J.-S. (2010). Simulated annealing based algorithm for smooth robot path planning with different kinematic constraints. In Shin, S. Y., Ossowski, S., Schumacher, M., Palakal, M. J., and Hung, C.-C., editors, *SAC*, pages 1277–1281. ACM. (Cited on page 24.)

[17] Hughes J.F., e. a. (2013). *Computer Graphics: Principles and Practice*. 3rd Edition. Addison-Wesley Professional, 3 edition. (Cited on pages 1 and 39.)

[18] James D. Foley, e. a. (1995). *Computer Graphics. Principles and Practice in C*. Addison-Wesley, 2 edition. (Cited on pages 1 and 39.)

[19] Kirkpatrick, S., Gelatt, C. D. J., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*. (Cited on page 24.)

[20] Knuth, D. E. (1977). A generalization of dijkstra's algorithm. *Information Processing Letters*, 6(1):1 – 5. (Cited on pages 46 and 47.)

[21] LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press. (Cited on pages 5, 6, 10, 53, and 58.)

[22] Metropolis, N. and Ulam, S. (1949). The monte carlo method. *J. Am. Stat. Assoc.*, 44:335. (Cited on page 24.)

[23] Paden, B., Cáp, M., Yong, S. Z., Yershov, D. S., and Frazzoli, E. (2016). A survey of motion planning and control techniques for self-driving urban vehicles. *CoRR*, abs/1604.07446. (Cited on page 5.)

[24] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. (1992). *Numerical Recipes*. Cambridge University Press. (Cited on pages 21 and 41.)

[25] Pukelsheim, F. (1994). The three sigma rule. *The American Statistician*, 48(2):88–91. (Cited on page 26.)

[26] Richard H. Bartels, John C. Beatty, B. A. B. (1995). *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling (The Morgan Kaufmann Series in Computer Graphics)*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann, 1 edition. (Cited on pages 13 and 15.)

[27] Salomon, D. (2006). *Curves and Surfaces for Computer Graphics*. Springer. (Cited on pages 13, 15, and 20.)

[28] Schneider, P. J. and Eberly, D. (2002). *Geometric Tools for Computer Graphics*. Elsevier Science Inc., New York, NY, USA. (Cited on pages 31, 32, and 35.)

[29] Sobol', I. M. (1994). *A primer for the Monte Carlo method*. CRC Press. (Cited on page 24.)

[30] Stoer, J. and Bulirsch, R. (1992). *Introduction to numerical analysis*. Springer-Verlag, New York. (Cited on page 21.)

[31] Šeda, M. and Pich, V. (2008). Robot motion planning using generalised voronoi diagrams. In *Proceedings of the 8th Conference on Signal Processing, Computational Geometry and Artificial Vision*, ISCGAV'08, pages 215–220, Stevens Point, Wisconsin, USA. World Scientific and Engineering Academy and Society (WSEAS). (Cited on page 44.)

[32] Wah, B. W. and Wang, T. (1999). Constrained simulated annealing with applications in nonlinear continuous constrained global optimization. In *ICTAI*, pages 381–. IEEE Computer Society. (Cited on page 24.)

ACRONYMS

CAD Computer-Aided Design

CAGD Computer-Aided Geometric Design

CHP Convex Hull Property

LR Lagrangian Relaxation

MCM Monte Carlo Method

OOP Object Oriented Programming

OTF Obstacle Triangular Face

PCLT Probability central limit theorem

PDF Probability Density Function

PE Probable Error

RRT Rapidly-expanding Random Tree

SA Simulated Annealing

UAV Unmanned Aerial Vehicle

UML Unified Modeling Language

VD Voronoi Diagram

VTK Visualization Tool Kit

XML eXtensible Markup Language

INDEX