GeoVer

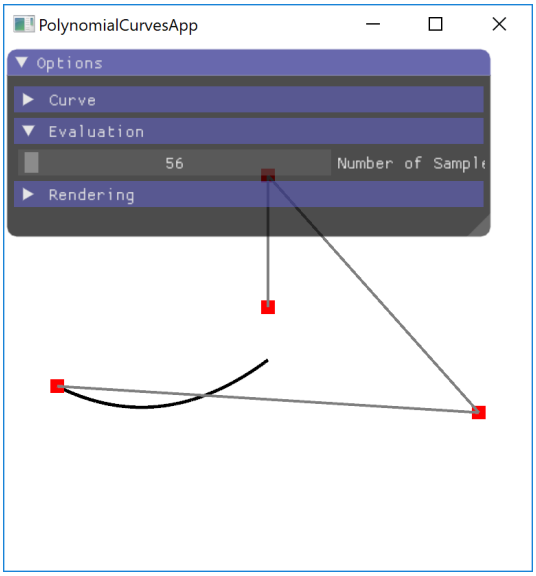
Praktikum P1

In diesem Praktikum werden wir verschiedene polynomielle Kurventypen behandeln. Als Basisklasse dient die Klasse `PolynomialCurveSegment`. Davon abgeleitet werden `MonomialCurveSegment`, `LagrangeCurveSegment` und `BezierCurveSegment`.

Aufgabe 1 Monom- und Lagrangekurven Segment

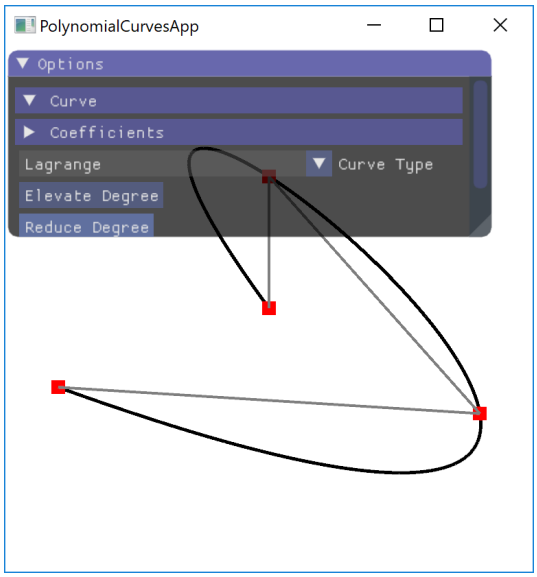
(a) Implementieren Sie die Methode `MonomialCurveSegment::evaluate`. Diese soll eine Kurve in der Monombasis auswerten. Benutzen Sie dabei das Horner Schema! Die Koeffizienten können Sie unter anderem mittels der Methode `getCoefficient(int i)` aus der Vaterklasse auslesen.

(b) Implementieren Sie nun die Methode `PolynomialCurveSegment::sample`. Diese soll die Kurve an `nSamplePoints` uniform im Parametergebiet auswerten. Rufen Sie zum Auswerten die virtuelle Methode `evaluate` auf. Sie sollten folgende Ausgabe erhalten:

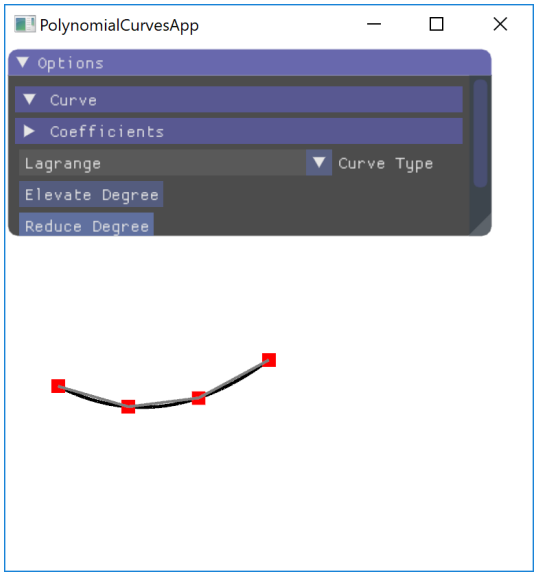


Sie können die Koeffizienten interaktiv verändern im dem Sie die roten Vierecke mit der Maus verschieben. Die Anzahl die Samplepunkte können Sie ebenfalls über das UI verändern, in dem Sie unter „Evaluation“ den Slider „Number of Samples“ verschieben.

(c) Implementieren Sie `LagrangeCurveSegment::evaluate` um eine Lagrangekurve auszuwerten. Der i-te Koeffizient soll dabei an der Stelle $t_i = \frac{i}{n}$ interpoliert werden, wobei n der Grad ist.
In der UI können Sie den Kurventypen unter „Coefficients“, „Curve Type“ von „Monomial“ auf „Lagrange“ stellen. Sie sollten folgendes Ergebnis erhalten:



(d) Ihnen ist aufgefallen, dass sich durch Umschalten des Kurventyps der Kurvenverlauf ändert und die Kontrollpunkte gleichbleiben. Wir wollen genau das umgekehrte Verhalten: Der Kurvenverlauf soll gleichbleiben und die Koeffizienten sollen sich anpassen. Implementieren Sie dazu `MonomialCurveSegment::toLagrangeCurveSegment`. Nach dem Umschalten des Kurventypes von Monom auf Lagrange, sollten Sie folgendes Ergebnis bekommen:



Hinweis: Sie können dabei die Tatsache ausnutzen, dass für eine Lagrangekurve $\vec{l}(t_i) = \vec{p}_i$ gilt.

(e) Nun wollen wir eine Lagrangekurve wieder in eine Monomkurve konvertieren. Die Lagrangebasisfunktionen $L_i(t)$ lassen sich mittels der Matrix $\boldsymbol{V}$ in die Monombasis wie folgt konvertieren:

$$\begin{bmatrix} M_0(t) \\ M_1(t) \\ \vdots \\ M_n(t) \end{bmatrix} = \boldsymbol{V}^{\top} \cdot \begin{bmatrix} L_0(t) \\ L_1(t) \\ \vdots \\ L_n(t) \end{bmatrix},$$

wobei mit $v_{i,j} = t_i^j$ und $t_i = \frac{i}{n}, 0 \leq i, j \leq n.$ Implementieren Sie dazu `LagrangeCurveSegment::lagrangeBasisFunctionsToMonomialBasisfunctions`.

Wir verwenden für große Matrizen die Bibliothek Eigen (siehe https://gitlab.com/libeigen/eigen). Mittels `v(i, j)` können Sie das Element in der i-ten Zeile und der j-ten Spalte lesen und modifizieren, wobei i und j bei 0 beginnen.

Nutzen Sie `LagrangeCurveSegment::lagrangeBasisFunctionsToMonomialBasisfunctions()` um `LagrangeCurveSegment::toMonomialCurveSegment()` zu implementieren. Diese Methode soll Lagrangekurve in eine Monomkurve konvertieren.

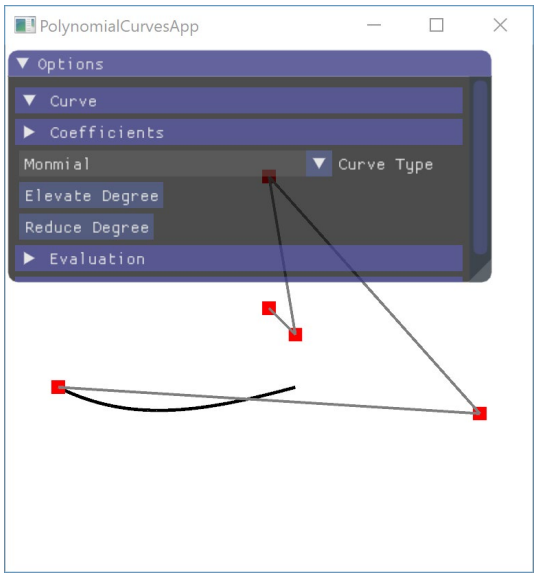
Hinweise:

- Um die Koeffizienten zu konvertieren brauchen Sie nicht die $\boldsymbol{V}$ sondern $\boldsymbol{V}^{-1}$.
- $\boldsymbol{V}^{-1}$ wird ähnlich wie die $\boldsymbol{C}$ auf Folie 40 01PolynomialCurves.pdf verwendet.
- Matrizen in Eigen können mittels der `inverse` Methode invertiert werden.
- Vermeiden Sie, wenn Sie Eigen-Typen deklarieren, das Schlüsselwort `auto`.

(f) Damit Nutzende mehr Freiheitsgrade zum Modellieren hat, wollen wir der Kurve einen neuen Koeffizienten hinzufügen. Dabei darf sich der Verlauf der Kurve nicht ändern.

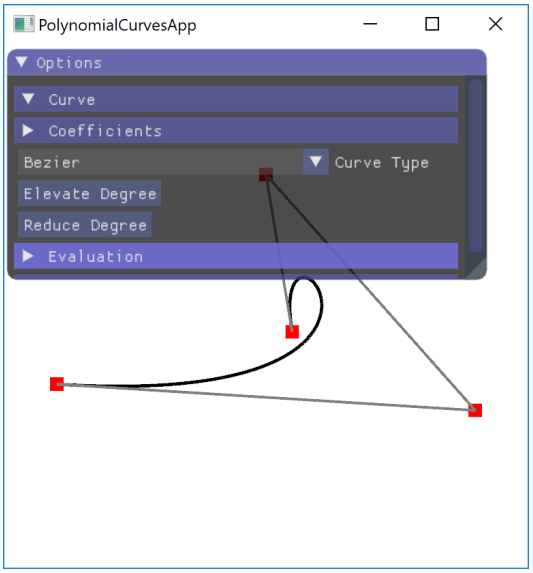
Implementieren Sie dazu `LagrangeCurveSegment::elevateDegree` und `MonomialCurveSegment::elevateDegree` und ändern Sie die Koeffizienten so, dass sie einen Koeffizienten mehr haben, der Kurvenverlauf sich aber nicht ändert!

Sie können die Graderhöhung in der UI testen indem Sie den Knopf „Elevate“ drücken. Sie sollten dann folgendes Ergebnis bekommen:



Aufgabe 2 Bézierkurven

(a) Implementieren Sie `BezierCurveSegment::evaluate` unter Verwendung des Horner-Bézier Schemas um eine Bézierkurve auszuwerten. Das Ergebnis sollte so aussehen:



Hinweis: Für die Berechnung der Binomialkoeffizienten steht `BezierCurveSegment::binomial` zur Verfügung.

(b) Auch hier ist es wieder so, dass sich bei der Konvertierung von Monom- bzw. Lagrangekurven zu Bézierkurven der Kurvenverlauf ändert, die Kontrollpunkte aber gleich bleiben. Wir wollen aber genau das umgekehrte Verhalten.

Um Monombasisfunktionen $M_i(t)$ in Bernsteinbasisfunktionen $B_i(t)$ zu konvertieren, kann man die Matrix $\boldsymbol{M}$ verwenden:

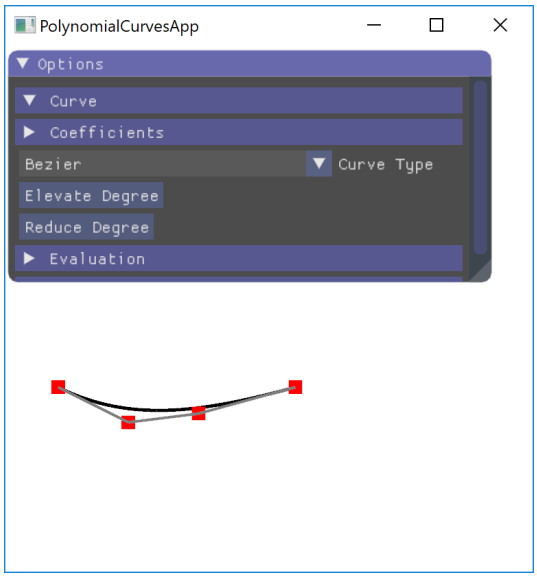
$$\begin{bmatrix} B_0(t) \\ B_1(t) \\ \vdots \\ B_n(t) \end{bmatrix} = \underbrace{\begin{bmatrix} m_{0,0} & m_{0,1} & \dots & m_{0,n} \\ \vdots & & \ddots & \vdots \\ m_{n,0} & m_{n,1} & \dots & m_{n,n} \end{bmatrix}}_{\boldsymbol{M}} \cdot \begin{bmatrix} M_0(t) \\ M_1(t) \\ \vdots \\ M_n(t) \end{bmatrix},$$

mit $m_{i,j} = (-1)^{j-i} \binom{n}{j} \binom{j}{i}$ wobei

$$\binom{i}{j} = \begin{cases} 0, & i \geq j \\ \dfrac{i!}{j!\,(i-j)!}, & \text{sonst.} \end{cases}$$

Geben Sie die Matrix $\boldsymbol{M}$ in `MonomialCurveSegment::monomialBasisFunctionsToBezierBasisFunctions` zurück.

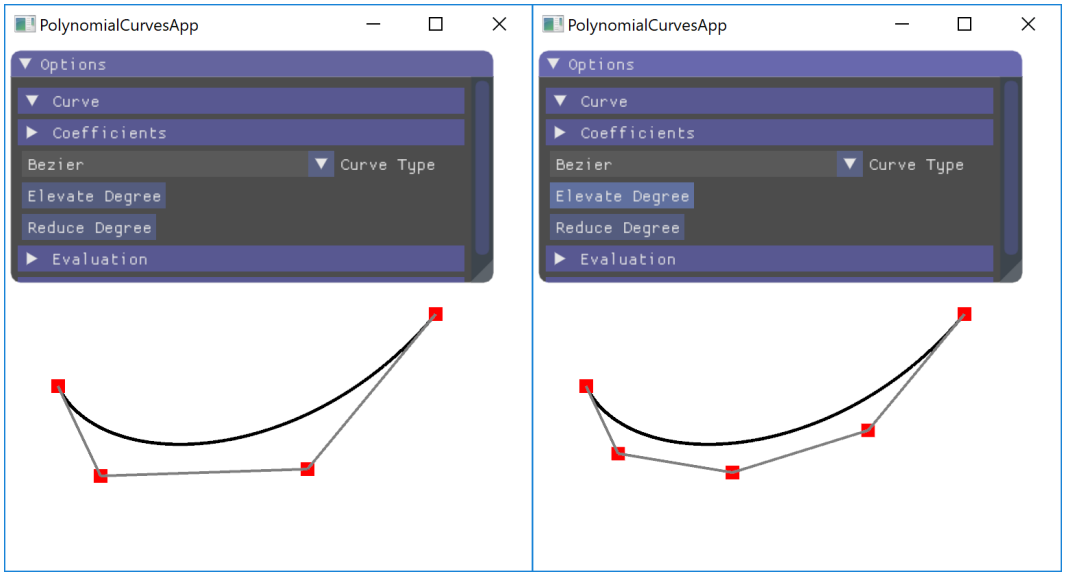
Nutzen Sie nun $\boldsymbol{M}$ um `MonomialCurveSegment::toBezierCurveSegment` und `BezierCurveSegment::toMonomialCurveSegment()` zu implementieren. Ein mögliches Ergebnis könnte so aussehen:



(c) Nutzen Sie nun die bereits vorhandenen Methoden um `BezierCurveSegment::toLagrangeCurveSegment`, `LagrangeCurveSegment::toBezierCurveSegment` zu implementieren.

Aufgabe 3 Algorithmen

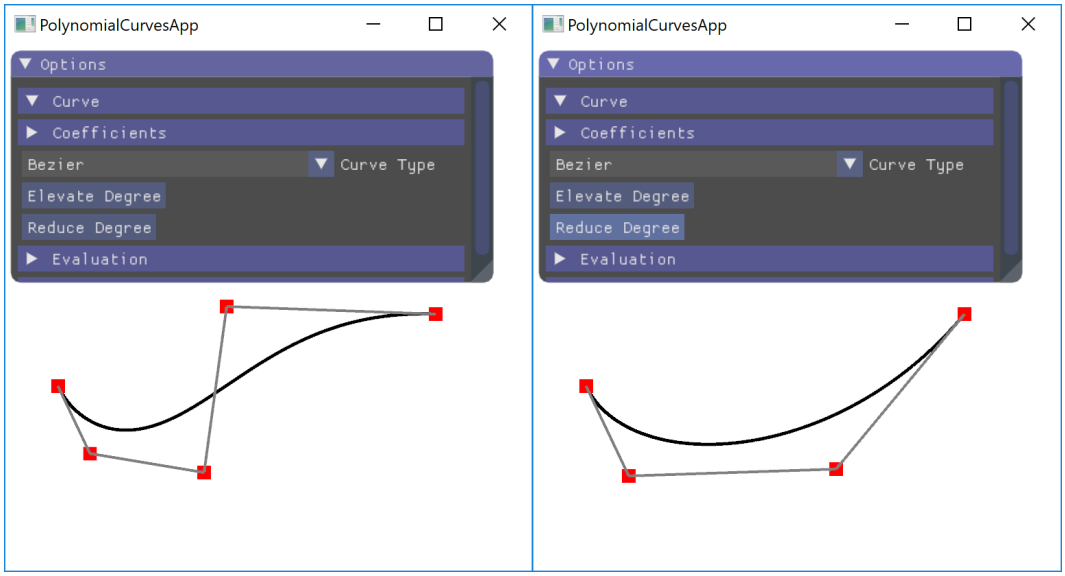
(a) Damit die Nutzerin bzw. der Nutzer mehr Freiheitsgrade zum Modellieren hat, wollen wir der Bézierkurve einen neuen Koeffizienten hinzufügen. Dabei darf sich der Verlauf der Kurve nicht ändern. Implementieren Sie dazu `BezierCurveSegment::elevateDegree()`.
Ein mögliches Ergebnis könnte so aussehen:



(b) Um den Grad von Bézierkurven zu reduzieren, implementieren Sie `BezierCurveSegment::reduceDegree`. Implementieren und verwenden Sie dazu die Methode `BezierCurveSegment::degreeElevationMatrix` welche eine Graderhöhungsmatrix bereitstellen soll (die Matrix können, müssen Sie aber nicht, auch bei 3a verwenden).

Hinweis: Die Bibliothek Eigen bietet die Methoden `inverse` und `transpose` auf Matrizen an und überlädt Operatoren „sinnvoll“. So ist z.B. die Matrixmultiplikation ist über den `*` Operator verfügbar.

Ein mögliches Ergebnis könnte so aussehen:



(c) Nutzen Sie `BezierCurveSegment::reduceDegree()` um `MonomialCurveSegment::reduceDegree()` und `LagrangeCurveSegment::reduceDegree()` zu implementieren.