

1. What does de Casteljau algorithm do? Implement the algorithm in C and test the results using arbitrary control points.

곡선 위의 임의의 위치를 표현하는 식이 control points를 잇는 선의 연속된 내분 과정으로 바뀌었다. 이 과정에서 하나의 곡선을 표현하기 위한 control points의 집합이 유일하지 않음을 알 수 있다. 따라서 곡선의 형태를 바꾸지 않고 곡선을 둘로 나눌 수 있게 되었다.

[illegible]

```

54         strm << "(" << obj.m_x << ", "
55             << obj.m_y << ", "
56             << obj.m_z << ")";
57     };
58
59 /**
60  * Function:    Destructor
61  */
62     ~Vertex() {};
63 };
64
65 /* Binomial Coefficient */
66 int N_Choose_K (int degree_n,
67                 int index_k);
68
69 /* Curve Algorithms */
70 Vertex Bezier( const Vertex* arr_control_points,
71                int degree_n,
72                double parameter_t);
73 Vertex DeCasteljau( const Vertex* arr_control_points,
74                     int degree_n,
75                     double parameter_t);
76
77 /**
78  * Main
79  */
80 int main(int argc, char* argv[]) {
81     /* Initialize control points as constant variable */
82     const Vertex P0(0.0, 0.0, 0.0);
83     const Vertex P1(0.0, 3.0, 0.0);
84     const Vertex P2(1.0, 3.0, 0.0);
85     const Vertex P3(1.0, 0.0, 0.0);
86     /* Insert control points to array */
87     const Vertex arr_control_points_list[DEGREE+1] = {P0, P1, P2, P3};
88
89     std::cout << "Degree of curve: " << DEGREE
90               << std::endl;
91     std::cout << "Control points: " << P0 << ", "
92               << P1 << ", "
93               << P2 << ", "
94               << P3
95               << std::endl;
96     std::cout << "Parameter: " << PARAMETER
97               << std::endl;
98
99     /* Calculate vertex on Bezier curve */
100    Vertex curve = Bezier(arr_control_points_list, DEGREE, PARAMETER);
101    std::cout << "Bezier curve: " << curve << std::endl;
102    /* Calculate vertex by using de Casteljau algorithm */
103    curve = DeCasteljau(arr_control_points_list, DEGREE, PARAMETER);
104    std::cout << "de Casteljau: " << curve << std::endl;
105
106    return 0;
107 } /* main */
108
109 /**
110  * Function:    Bezier
111  * Purpose:     Calculation of coordinates on Bezier curve
112  *              w.r.t. the value of t.
113  * In args:     arr_control_points (list of control points)
114  *              degree_n           (degree of curve)
115  *              parameter_t        (the value of parameter)
116  * Out arg:     trace_on_curve     (vertex on Bezier curve)
117  */
118 Vertex Bezier( const Vertex* arr_control_points,

```

```

119         int degree_n,
120         double parameter_t) {
121     Vertex trace_on_curve(0.0, 0.0, 0.0);
122
123     /* Check parameter's errors */
124     if (parameter_t < 0 || parameter_t > 1) {
125         std::cerr << "ERROR::FUNC::Bezier: "
126                 << "Variable parameter_t must be the value from 0 to 1."
127                 << std::endl;
128     }
129     /* Check degree's errors */
130     if (degree_n < 2) {
131         std::cerr << "ERROR::FUNC::Bezier: "
132                 << "Control points must be more than or equal to 3 for Bezier curve."
133                 << std::endl;
134     }
135     /* Calculate coordinates of Bezier curve */
136     else {
137         for (int k = 0; k <= degree_n; k++) {
138             /* Calculate Bernstein basis function */
139             double bernstein = N_Choose_K(degree_n, k)
140                               * std::pow(1 - parameter_t, degree_n - k)
141                               * std::pow(parameter_t, k);
142             /* Multiply control point and berstein basis */
143             trace_on_curve.m_x += arr_control_points[k].m_x * bernstein;
144             trace_on_curve.m_y += arr_control_points[k].m_y * bernstein;
145             trace_on_curve.m_z += arr_control_points[k].m_z * bernstein;
146         }
147     }
148
149     return trace_on_curve;
150 } /* Bezier */
151
152 /**
153  * Function:    N_Choose_K
154  * Purpose:     Calculation of binomial coefficient
155  * In args:     degree_n          (degree of binomial)
156  *              index_k           ('k'th picking value)
157  * Out arg:     result            (binomial coefficient)
158  */
159 int N_Choose_K (int degree_n,
160                int index_k) {
161     int result, denominator = 1, numerator = 1;
162
163     /* Check degree's errors */
164     if (degree_n < 0) {
165         std::cerr << "ERROR::FUNC::N_Choose_K: "
166                 << "Variable degree_n must be greater than 0"
167                 << std::endl;
168         result = 0;
169     }
170     /* Check restriction */
171     if (degree_n < index_k) {
172         std::cerr << "ERROR::FUNC::N_Choose_K: "
173                 << "Variable degree_n must be greater than index_k."
174                 << std::endl;
175         result = 0;
176     }
177     /* Calculate N choose K */
178     else {
179         if (degree_n == 0 || index_k == 0) {
180             result = 1;
181         }
182         else if (index_k == 1) {
183             result = degree_n;

```

```

184     }
185     /* Compare which of two values, N and K, is greater */
186     else if (degree_n > index_k<<1) {
187         /* n*(n-1)*...*(n-k+1) / k*(k-1)*...*1 */
188         for (int i = 1; i <= index_k; i++) {
189             denominator *= i;
190             numerator *= degree_n - (i-1);
191         }
192         result = numerator / denominator;
193     }
194     else {
195         /* n*(n-1)*...*(k+1) / (n-k)*(n-k-1)*...*1 */
196         for (int i = 1; i <= degree_n - index_k; i++) {
197             denominator *= i;
198             numerator *= degree_n - (i-1);
199         }
200         result = numerator / denominator;
201     }
202 }
203
204 return result;
205 } /* N_Choose_K */
206
207 /**
208  * Function:    DeCasteljau
209  * Purpose:     Calculation of de Casteljau algorithm
210  * In args:     arr_control_points (list of control points)
211  *              degree_n           (degree of curve)
212  *              parameter_t        (the value of parameter)
213  * Out arg:     trace_on_curve     (vertex on Bezier curve)
214  */
215 Vertex DeCasteljau( const Vertex* arr_control_points,
216                     int degree_n,
217                     double parameter_t) {
218     Vertex trace_on_curve(0.0, 0.0, 0.0);
219
220     /* Check parameter's errors */
221     if (parameter_t < 0 || parameter_t > 1) {
222         std::cerr << "ERROR::FUNC::DeCasteljau: "
223                   << "Variable parameter_t must be the value from 0 to 1."
224                   << std::endl;
225     }
226     /* Check degree's errors */
227     if (degree_n < 2) {
228         std::cerr << "ERROR::FUNC::DeCasteljau: "
229                   << "Control points must be more than or equal to 3 for Bezier curve."
230                   << std::endl;
231     }
232     /* Calculate coordinates of Bezier curve by using de Casteljau algorithm */
233     else {
234         /* Initialize list of points for curve fitting */
235         Vertex* arr_curve_fitting_points = new Vertex[degree_n + 1];
236         for (size_t i = 0; i <= degree_n; i++) {
237             arr_curve_fitting_points[i] = arr_control_points[i];
238         }
239         /* 'k'th curve fitting point at step 'r' =
240          *      (1 - t) * ('k-1'th curve fitting point at step 'r-1')
241          *      + t * ('k'th curve fitting point at step 'r-1')
242          */
243         for (int r = 1; r <= degree_n; r++) {
244             for (int k = 1; k <= degree_n - r + 1; k++) {
245                 arr_curve_fitting_points[k - 1].m_x
246                     = (1 - parameter_t) * arr_curve_fitting_points[k - 1].m_x
247                     + parameter_t * arr_curve_fitting_points[k].m_x;
248                 arr_curve_fitting_points[k - 1].m_y

```

```

249         = (1 - parameter_t) * arr_curve_fitting_points[k - 1].m_y
250           + parameter_t      * arr_curve_fitting_points[k].m_y;
251     arr_curve_fitting_points[k - 1].m_z
252     = (1 - parameter_t) * arr_curve_fitting_points[k - 1].m_z
253       + parameter_t      * arr_curve_fitting_points[k].m_z;
254     }
255 }
256
257 trace_on_curve = arr_curve_fitting_points[0];
258 delete[] arr_curve_fitting_points;
259 }
260
261 return trace_on_curve;
262 } /* DeCasteljau */

```

#### RESULTS:

Degree of curve: 3  
 Control points: (0, 0, 0), (0, 3, 0), (1, 3, 0), (1, 0, 0)  
 Parameter: 0.5  
 Bezier curve: (0.5, 2.25, 0)  
 de Casteljau: (0.5, 2.25, 0)

Process returned 0 (0x0)    execution time : 0.005 s  
 Press ENTER to continue.

## 2. In Bezier curves, explain the end tangent interpolation.

곡선의 양 끝점(첫 점과 끝 점)에서의 기울기는 control points가 만들어내는 볼록 다각형의 양 끝 점에서의 기울기와 반드시 같아야 한다는 것이다. 따라서 end point interpolation을 동시에 생각한다면, 곡선은 항상 양 끝점에서 폴리곤에 접한다.

$$\vec{r}(t) = \sum_{k=0}^n \vec{P}_k b_k^n(t)$$

$$\frac{d}{dt} \vec{r}(t) = \sum_{k=0}^{n-1} (\vec{P}_{k+1} - \vec{P}_k) n b_k^{n-1}(t)$$

subject to  $t = 0, t = 1$

$$\left. \frac{d}{dt} \vec{r}(t) \right|_{t=0} = n(\vec{P}_1 - \vec{P}_0)$$

$$\left. \frac{d}{dt} \vec{r}(t) \right|_{t=1} = n(\vec{P}_n - \vec{P}_{n-1})$$

위의 식을 통해서, 첫 점( $t = 0$ )과 끝 점( $t = 1$ )에서 곡선의 기울기는 폴리곤의 벡터 방향과 동일하다는 것을 증명하였다.

## 3. Explain the convex hull property.

convex hull property를 이해하기 위해서는 convex combination을 알고 있어야 한다.

먼저, 결합의 종류 중에 각 항의 계수가 양수이면서 계수의 합이 1인 결합을 convex combination이라 정의한다.

$$a_0\phi_0 + a_1\phi_1 + a_2\phi_2 + \dots + a_n\phi_n$$
$$\forall k : a_k \geq 0 \wedge \sum_{k=0}^n a_k = 1$$

만약  $\phi$ 를 어떤한 위치라고 가정하고 식의 의미를 살펴보면,

$\phi$ 들로 이루어지는 어떠한 공간이 있고,  $\phi$ 를 이용해 convex combination으로 표현되는 공간 상의 값은 각각의  $\phi$ 를 직선으로 잇는 경계와 그 내부를 표현하게 된다. 즉, convex combination으로 표현된 공간은  $\phi$ 들이 이루는 경계를 넘어서지 않는다.

그렇다면 Bernstein basis function ( $b_k^n(t)$ )과 control point( $\vec{P_k}$ )로 표현되는 Bezier curve는 convex combination이기 때문에 Bezier curve는 각각의 control points를 이어서 만들 수 있는 최소 볼록 다각형 안에 존재하게 된다.

이것이 Bezier curve가 갖는 convex hull property다.

## 4. Why do we need the rational form?

분수 형식이 아닌 Bezier 곡선으로는 원뿔 곡선을 표현할 수 없다. 2차원 원뿔 곡선을 표현하기 위해서는 2차원 Bezier 곡선을 평면에 투영하는 방식(perspective projection)을 사용하면 된다. 이를 통해 나타나는 식이 분수 형식이기 때문에, 우리는 Bezier 곡선으로 원뿔 곡선을 표현하기 위해서 분수 형식을 사용한다.