Computer Graphics

Lecture 10: geometric data structures and parametric curves

Ágoston Sipos
siposagoston@inf.elte.hu

Eötvös Loránd University
Faculty of Informatics

2025-2026. Spring semester

Table of contents

Storing geometry and topology

- Storing geometry
- Storing topology
  - Winged-edge data structure
  - Half-edge data structure

Representation of curves

- Linear interpolation
- Polynomial curves
- Hermite interpolation
- Bézier curves
- Subdivision curves

Direct storage of geometry

- Today's modern graphics APIs use point, segment, and triangle primitives
- In this section, we will look at how they can be stored efficiently
- Furthermore, how we can also extract neighborhood information

Storing geometry – brute force

- More generally, let our primitives be planar polygons, which are represented by listing their vertices (according to some fixed traversal order)
- Tasks related to polygons:
  - storing
  - transforming
  - neighborhood queries
- Brute force storage: for all faces, we store all its vertices once
- If there are only triangles, this is also called a *triangle soup*

Storing geometry – *"brute force"*

```
struct triangle {
    float x1, y1, z1;
    float x2, y2, z2;
    float x3, y3, z3;
};
```

"brute force" Storage Analysis

- *Storage:* if polygons have common vertices, then these are stored several times – unnecessarily $\rightarrow$ not very good
  - even worse if we have per-vertex attributes (e.g. normal vector, texture coordinates)
- *Transforming:* we perform the transformations on shared vertices multiple times $\rightarrow$ inefficient
- *Queries:* we have no idea who is whose neighbor, we can only get results by visiting all the vertices $\rightarrow$ disaster
- The only advantage is that it couldn't be stored more simply

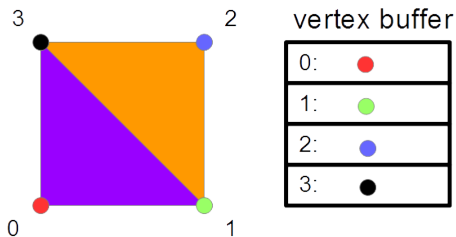
Index buffers

- The idea: store all vertices once, in a large shared array!
- Polygons should only refer to elements of the vertex array.
- This is the *index buffer*.
- All GPUs support it.

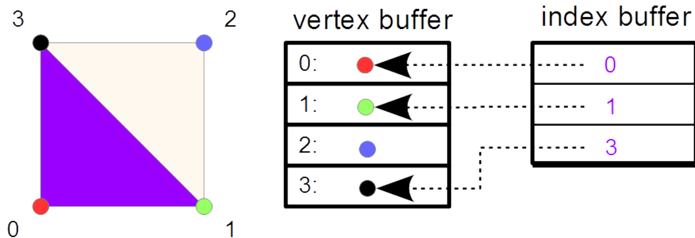
Index buffers

```
struct vec3 { float x,y,z; };

struct Vertex {
    vec3 pos;
    // + other attributes;
};

std::vector<Vertex> vertexBuffer;
std::vector<unsigned int> indexBuffer;
```

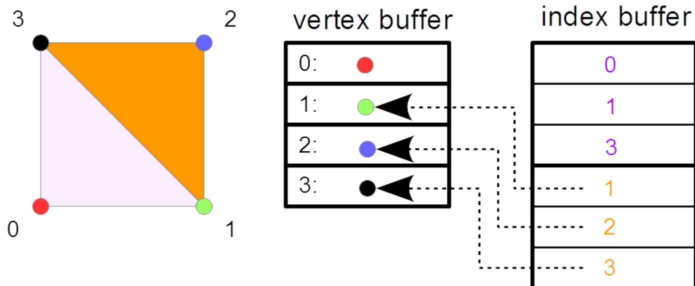
Index buffer



Index buffer



Index buffer



Example

- Take a grid consisting of $N \times N$ squares (each consisting of two triangles)! How much data do we need to store this?
- Size without *index buffer*: 6 vertex per square, $N \times N$ square: $6N^2$ *vertex*.
- Size with *index buffer*: $(N+1) \times (N+1) = N^2 + 2N + 1$ *vertex* ($+\ 6N^2$ integer, *index*)
- When is it worth it? In other words, how do $6N^2$ and $N^2 + 2N + 1$ relate to each other?

$$
\begin{aligned}
6N^2 >& N^2 + 2N + 1 \\
0 >& -5N^2 + 2N + 1 \\
N >& 1, \text{if } N \in \mathbb{Z}^+
\end{aligned}
$$

Example – continued

- If we have more than one square, it's already worth it!
- E.g. if $N = 10$
- Size without *index buffer*: we store and transform 600 vertices
- Size with *index buffer*: we store and transform 121 vertices

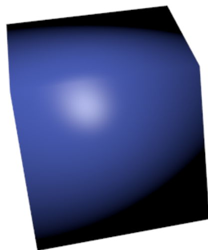
*Index buffer*s on the GPU

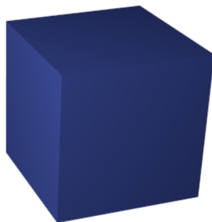
- Every graphics card, “which are not yet collected by museums” (quote from 2010) supports *index buffer*s.
- An array of vertices (*vertex buffer*) can contain not only positions, but also normal vectors, texture coordinates, and more.
- A reference to *vertex buffer* refers to all of these together.

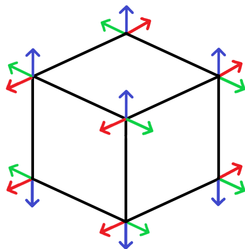
Cube problem

- Pay attention: the index buffer can only help if the same vertex is used for different triangles, but each triangle is an approximation of the *same surface*!
- If we assemble a cube from triangles, then a vertex is part of min. three, max. six triangles, but each vertex is a point of three different surfaces (the three faces that meet there).
- So, although it would be enough to register only 8 vertices with the *index buffer*, as soon as we try illuminate the cube and need surface normals, it will be a problem!

Cube problem







Cube problem

- The problem: although a corner represents a single point in space, it is actually a *surface point* of three different surfaces
- Since we also store surface properties in the vertices in addition to positions (normal), we cannot reduce spatial redundancy even with an index buffer in this case!
- The vertices must be specified separately for each side, the position is same, but with the help of three different normals according to the three sides: so a total of 3×8 vertices are placed in the *vertex buffer*

Index buffer storage analysis

- *Storage:* efficient.
- *Transformation:* efficient.
- *Neighborhood relations:* we know the common vertices, but we still have no idea.

Neighborhood relations

- Sometimes you need neighbors, e.g. when dividing surfaces, filtering out degenerate primitives, handling certain user inputs, etc.
- Vertices are known $\Rightarrow$ you can calculate who is whose neighbor!
- Any number of polygons can meet in a vertex $\Rightarrow$ a dynamic data structure is needed
- Better solution: *Winged-edge* data structure!

Winged-edge data structure

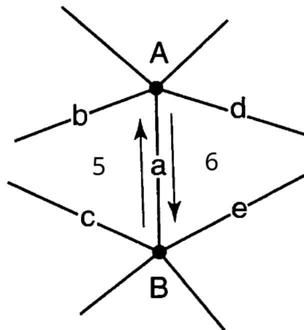
- The representation describing shapes with their boundaries (*B-rep*) (boundary representation) is one of the frequently used data structures to store the topology of manifold polyhedra
- During storage, it distinguishes *vertices*, *edges* and *faces*
- We store the surface in terms of edges
- Each edge has a fixed number of data
- E.g. with its help, we can quickly walk around the edges of a polygon, getting all the neighbors in the process

Winged-edge data structure

- Each face is bounded by an edge sequence – for each face, store a pointer to any edge belonging to its edge sequence
- We assign vertices to the edges (either it starts from it, or it is the goal) – let's assign one of them to the vertex

Single edge data

edge	vertex		face		toLeft		toRight	
	start	end	left	right	prev.	next	prev.	next
a	B	A	5	6	c	b	d	e



Single edge data

- An edge connects two vertices – we store them in the edge
- An edge can belong to at most two faces – one will be the left face, the other the right face, we store pointers (or indices) pointing to these
- On the above two face, the given edge is also part of an edge sequence (the series of edges that make up the given face) – in both edge sequence, we store the following and the preceding *for the given edge according to the traversal order of the given face* (!)

Other tables

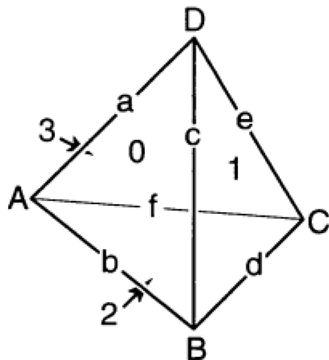
Vertex table

- vertex ID
- an edge starting from the vertex

Face table

- face ID
- an edge of the face

Example: tetrahedron



edge	*vertex 1*	*vertex 2*	*face left*	*face right*	*pred left*	*succ left*	*pred right*	*succ right*
a	A	D	3	0	f	e	c	b
b	A	B	0	2	a	c	d	f
c	B	D	0	1	b	a	e	d
d	B	C	1	2	c	e	f	b
e	C	D	1	3	d	c	a	f
f	C	A	3	2	e	a	b	d

vertex	*edge*
A	a
B	d
C	d
D	e

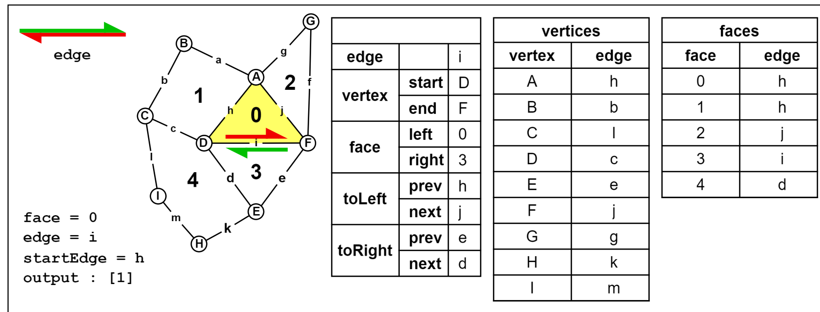
face	*edge*
0	a
1	c
2	d
3	a

Shirley, Fundamentals of Computer Graphics

E.g.: Enumeration of all neighboring faces of a face

```
def allNeighbours(face, edges, vertices, faces):
  startEdge = faces[face]
  edge = startEdge
  output = []
  do:
    if edges[edge].faceLeft == face:
      output.append(edges[edge].faceRight)
      edge = edges[edge].succLeft
    else:
      output.append(edges[edge].faceLeft)
      edge = edges[edge].succRight
  while edge != startEdge
```

E.g.: Enumeration of all neighboring faces of a face

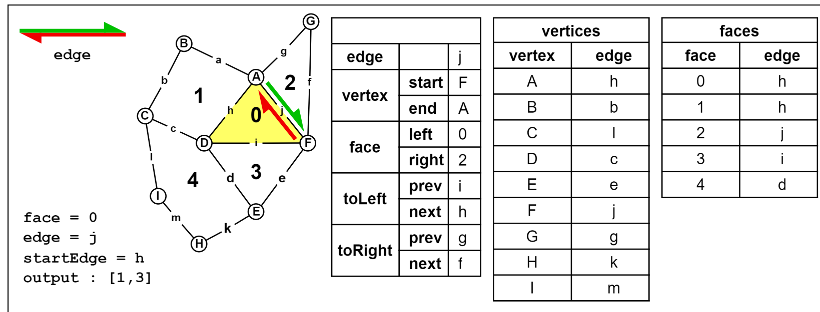


edge		i
vertex	start	D
	end	F
face	left	0
	right	3
toLeft	prev	h
	next	j
toRight	prev	e
	next	d

vertices	
vertex	edge
A	h
B	b
C	l
D	c
E	e
F	j
G	g
H	k
I	m

faces	
face	edge
0	h
1	h
2	j
3	i
4	d

E.g.: Enumeration of all neighboring faces of a face

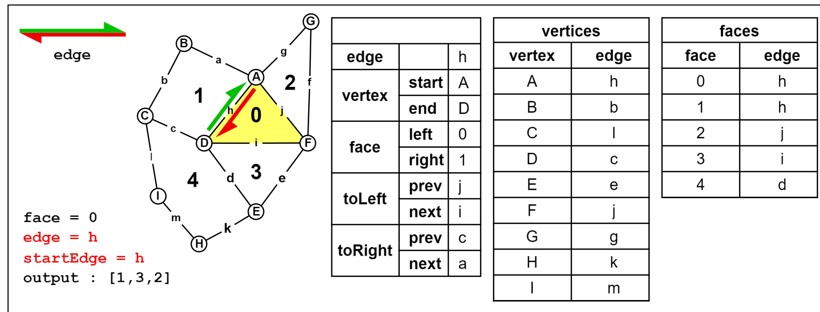


edge		j
vertex	start	F
	end	A
face	left	0
	right	2
toLeft	prev	i
	next	h
toRight	prev	g
	next	f

vertices	
vertex	edge
A	h
B	b
C	l
D	c
E	e
F	j
G	g
H	k
I	m

faces	
face	edge
0	h
1	h
2	j
3	i
4	d

E.g.: Enumeration of all neighboring faces of a face



edge		h
vertex	start	A
	end	D
face	left	0
	right	1
toLeft	prev	j
	next	i
toRight	prev	c
	next	a

vertices	
vertex	edge
A	h
B	b
C	l
D	c
E	e
F	j
G	g
H	k
I	m

faces	
face	edge
0	h
1	h
2	j
3	i
4	d

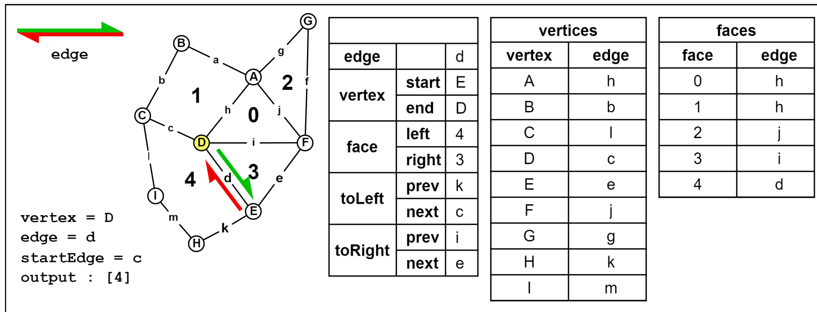
E.g.: Enumeration of all neighboring faces of a face

- That is: let's start from the representative edge of the given face (which we store for the face)
- If the left face of the current edge is the starting face: we write out the right face and we move to the next edge of the left face
- Otherwise the right face is the starting face, we write out the left face and move to the next edge of the right face
- The iteration should end as soon as we get back to the representative edge of the given face

E.g.: Enumeration of all faces containing the vertex

```
def allFaces(vertex, edges, vertices, faces):
  startEdge = vertices[vertex]
  edge = startEdge
  output = []
  do:
    if edges[edge].vertStart == vertex:
      output.append(edges[edge].faceLeft)
      edge = edges[edge].predLeft
    else:
      output.append(edges[edge].faceRight)
      edge = edges[edge].predRight
  while edge != startEdge
```

E.g.: Enumeration of all faces containing the vertex

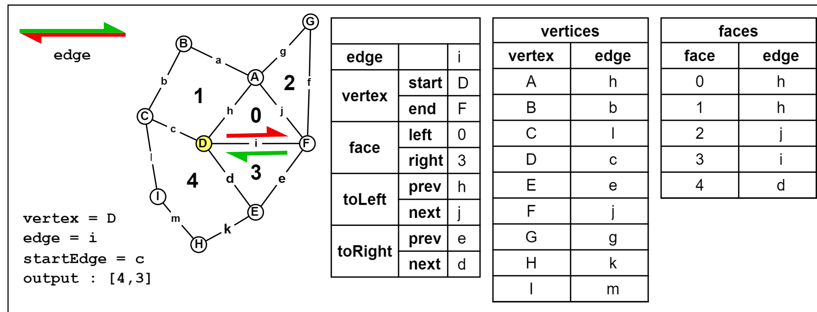


edge		d
vertex	**start**	E
	end	D
face	**left**	4
	right	3
toLeft	**prev**	k
	next	c
toRight	**prev**	i
	next	e

vertices	
vertex	**edge**
A	h
B	b
C	l
D	c
E	e
F	j
G	g
H	k
I	m

faces	
face	**edge**
0	h
1	h
2	j
3	i
4	d

E.g.: Enumeration of all faces containing the vertex

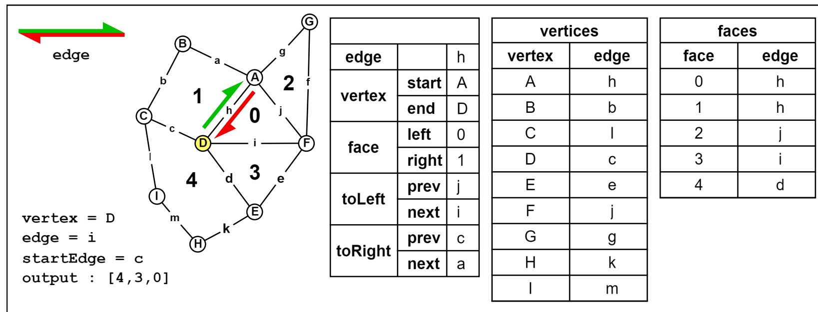


edge		i
vertex	start	D
	end	F
face	left	0
	right	3
toLeft	prev	h
	next	j
toRight	prev	e
	next	d

vertices	
vertex	edge
A	h
B	b
C	l
D	c
E	e
F	j
G	g
H	k
I	m

faces	
face	edge
0	h
1	h
2	j
3	i
4	d

E.g.: Enumeration of all faces containing the vertex



edge		h
vertex	start	A
	end	D
face	left	0
	right	1
toLeft	prev	j
	next	i
toRight	prev	c
	next	a

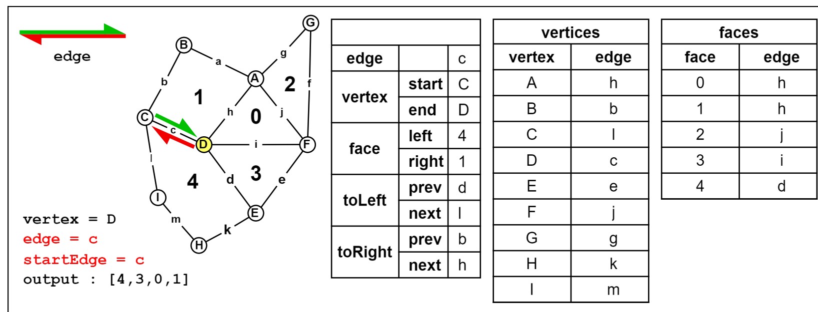
vertices

vertex	edge
A	h
B	b
C	l
D	c
E	e
F	j
G	g
H	k
I	m

faces

face	edge
0	h
1	h
2	j
3	i
4	d

E.g.: Enumeration of all faces containing the vertex



edge		c
vertex	start	C
	end	D
face	left	4
	right	1
toLeft	prev	d
	next	l
toRight	prev	b
	next	h

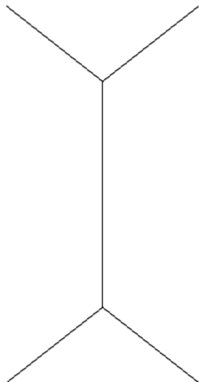
vertices	
vertex	edge
A	h
B	b
C	l
D	c
E	e
F	j
G	g
H	k
I	m

faces	
face	edge
0	h
1	h
2	j
3	i
4	d

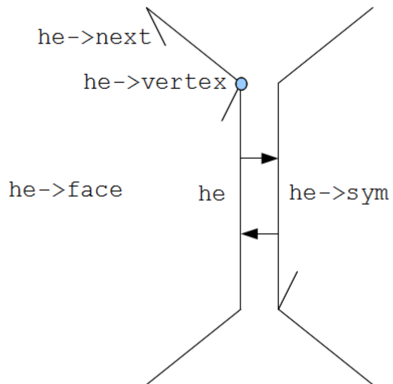
Half-edge data structure

- Separate the edge of the winged-edge into two half-edges
- $\rightarrow$ essentially, we work with the projection of the edges onto the faces
- Only one face can belong to the half-edge + we must store the sibling half-edge (the projection of the given edge onto the other face)
- The central element of the representation is the half-edge

Half-edge



Half-edge



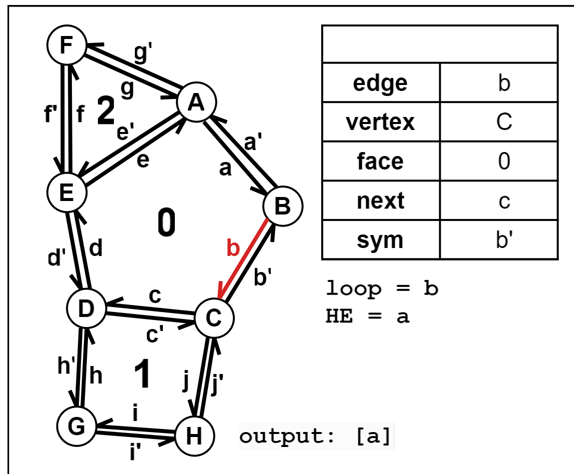
Half-edge data structure

- Strictly speaking, a half-edge has exactly one vertex, edge and face (but in practice it can be useful to store more than this)
- The followings are stored in a half-edge: the half-edge's target (vertex), the half-edge's sibling (sym), the half-edge's face (face) and the next in the half-edge series surrounding the face (next)
- For the faces we store an arbitrary bounding half-edge
- For the vertices an incoming half-edge
- HE→sym→sym = HE, HE→next→sym→vertex = HE→vertex etc.

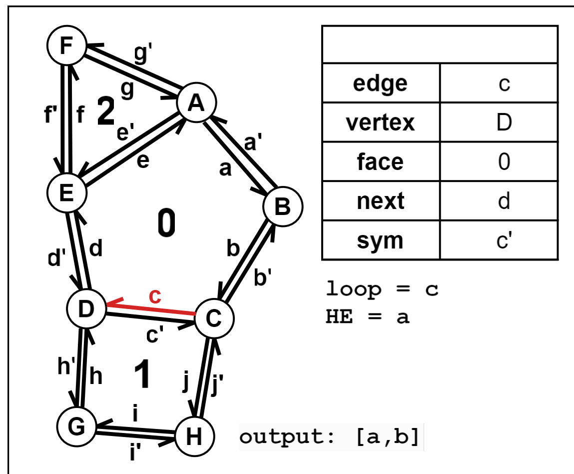
E.g.: Traversing edges of the face

```
def faceLoop(HE):
  loop = HE
  output = []
  do:
    output.append(loop)
    loop = loop->next
  while loop != HE
```

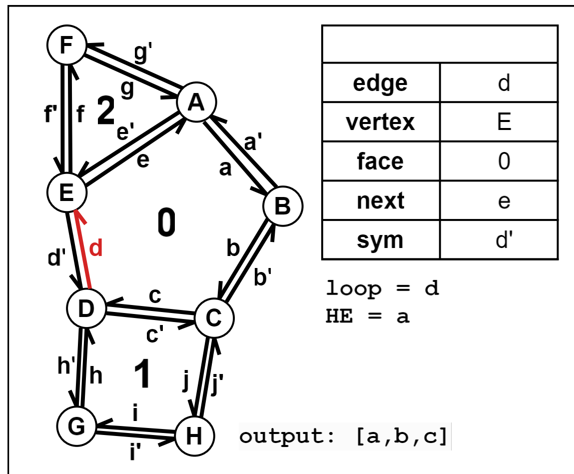
E.g.: Traversing edges of the face



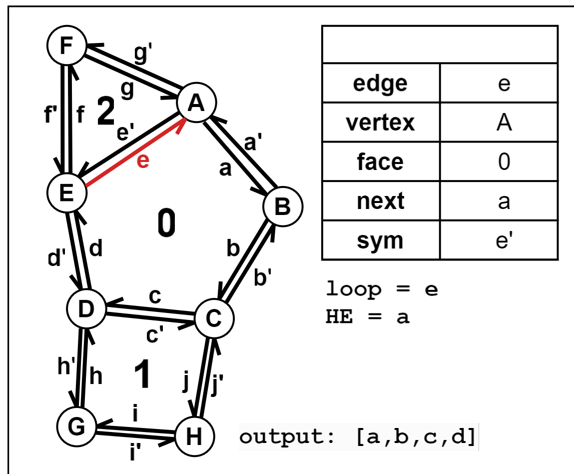
E.g.: Traversing edges of the face



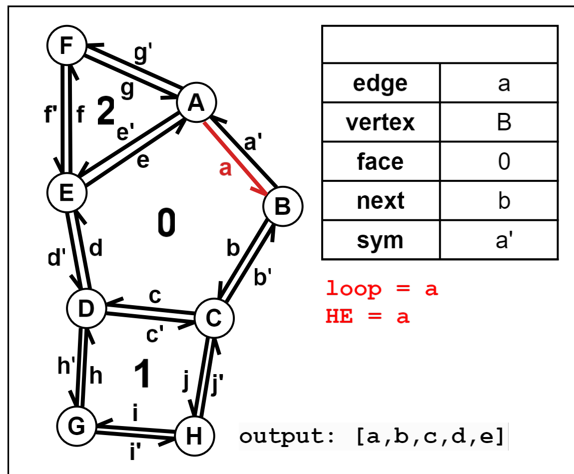
E.g.: Traversing edges of the face



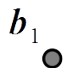
E.g.: Traversing edges of the face

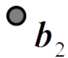


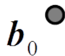
E.g.: Traversing edges of the face



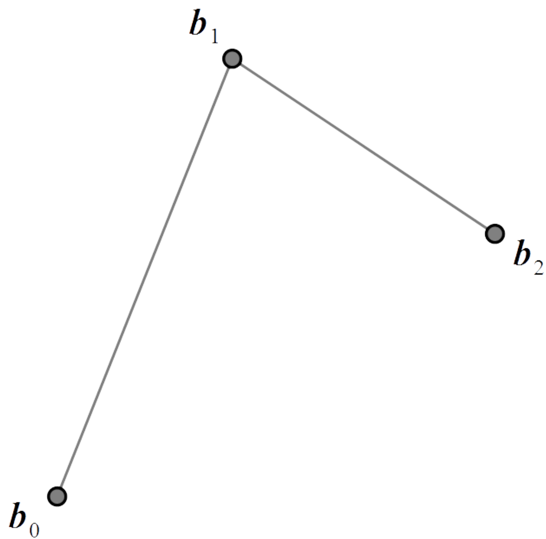
Representation



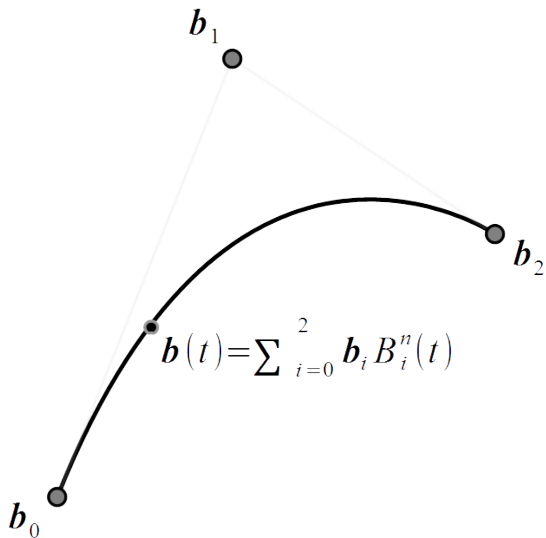




Representation



Representation



Representing curves

- We have already seen three ways of representing curves:
  - explicit: $y = f(x)$
  - parametric: $\mathbf{p}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}, t \in \mathbb{R}$
  - implicit: $f(x, y) = 0, \ x, y \in \mathbb{R}$

Representing curves

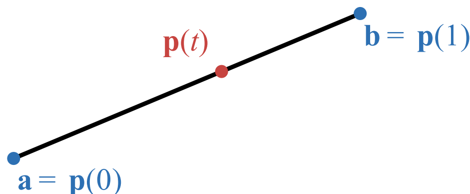
- Now we examine what data must be stored in order to represent an arbitrary (so-called *free-form*) curve
- Let's look at this using the parametric form

Linear interpolation

- Let two points be, $\mathbf{a}, \mathbf{b} \in \mathbb{E}^3$. The parametric equation of the line passing through the two points:

$$\mathbf{p}(t) = (1 - t)\mathbf{a} + t\mathbf{b},$$

  where $t \in \mathbb{R}$.
- If $t \in [0, 1]$, then we get the straight segment connecting $\mathbf{a}, \mathbf{b}$ points.



Linear interpolation

- A straight segment is clearly identified by the two endpoints of the segment, **a** and **b**
- This is the simplest "curve" between the two points
- How does this become a "beautiful" curve?
- "beautiful" $\approx$ something that is naturally changing

Beauty – parametric continuity

- C^0: there are no holes in the curve/surface, it does not break anywhere
- C^1: the derivative also changes continuously (BUT: the derivative *depends on the parameterization*!)
- C^2: the second derivatives of the curve/surface also change continuously

Beauty – derivatives

- Remember, the parametric curve has the form $\mathbf{p}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$
- So the form of the derivatives $\mathbf{p}'(t) = \begin{bmatrix} x'(t) \\ y'(t) \end{bmatrix}$,

  $\mathbf{p}''(t) = \begin{bmatrix} x''(t) \\ y''(t) \end{bmatrix}$ etc.
- Example: what is the derivative of the curve $\mathbf{p}(t) = \begin{bmatrix} t^2 + t \\ t^3 \end{bmatrix}$

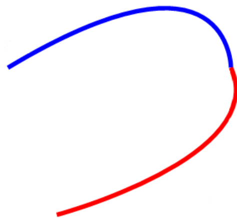
  at $t = 0$ and $t = 1$ points?
- Example: what will be the first and second derivative curves (hodograph) of the curve $\mathbf{p}(t) = (1 - t)\mathbf{a} + t\mathbf{b}$?

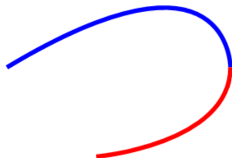
Parametric continuity

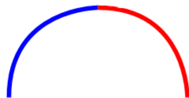
Let two parametric curves be, $\mathbf{r}(t), \mathbf{s}(t) : [0, 1] \to \mathbb{E}^3$, which have a common point, namely e.g. $\mathbf{r}(1) = \mathbf{s}(0) = \mathbf{p}$. Then the two curves in $\mathbf{p}$

- $C^0 \Leftrightarrow \mathbf{r}(1) = \mathbf{s}(0)$
- $C^1 \Leftrightarrow C^0 \wedge \mathbf{r}'(1) = \mathbf{s}'(0)$
- $C^2 \Leftrightarrow C^1 \wedge \mathbf{r}''(1) = \mathbf{s}''(0)$
- $C^n \Leftrightarrow C^{n-1} \wedge \mathbf{r}^{(n)}(1) = \mathbf{s}^{(n)}(0)$

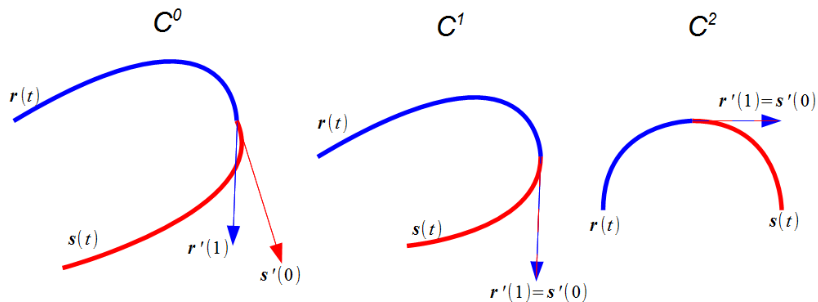
Parametric continuity



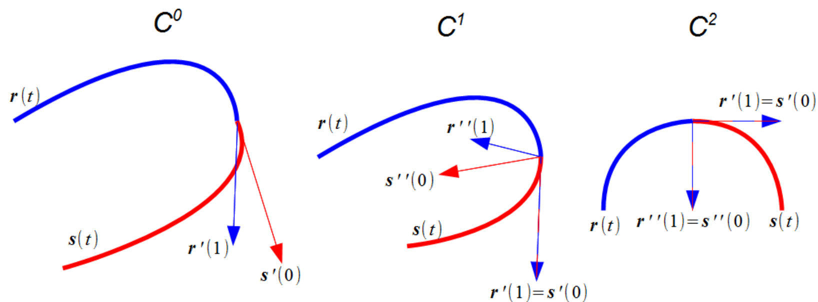




Parametric continuity



Parametric continuity

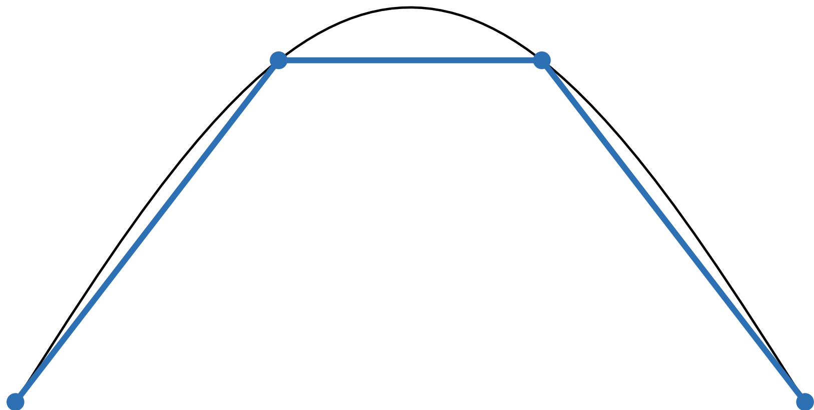


Detour – geometric continuity

- To our eyes, not only what is parametrically continuous will be continuous
- The following won't be completely rigorous mathematical definitions
- For geometric continuity, we make continuity constraints independent of parameterization:
  - G^0: there are no holes in the curve/surface, it does not break anywhere
  - G^1: for connections, if the derivatives do not match, but $\exists\alpha > 0$ such that $\mathbf{m}_i = \alpha\mathbf{m}_{i+1}$
  - G^2: the curvature of the curve/surface is also continuous in the connection

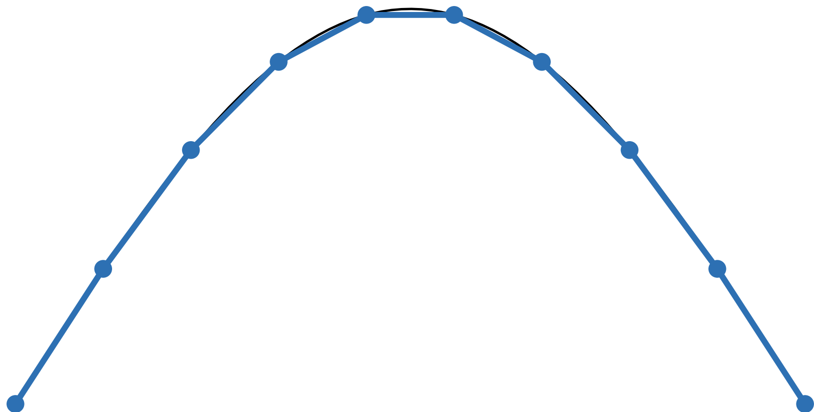
Segmented line

If we want to represent a "smooth" curve, we can approximate it with segments (tessellation)



Segmented line

If we want to represent a "smooth" curve, we can approximate it with segments (tessellation)



Segmented line

- Let $\mathbf{p}_i \in \mathbb{E}^3, i = 0, ..., n$ points and connect every $\mathbf{p}_i, \mathbf{p}_{i+1}, i = 0, ..., n-1$ point pair with a segment!
- Let $t_0 \le t_1 \le ... \le t_n \in \mathbb{R}$ be parameters assigned to points $\mathbf{p}_i$
- Then the resulting segmented line can be written with a single parameter: for the current $t \in [t_0, t_n]$ value if $t \in [t_i, t_{i+1}]$ is true, then the corresponding point

$$\frac{t_{i+1} - t}{t_{i+1} - t_i}\mathbf{p}_i + \frac{t - t_i}{t_{i+1} - t_i}\mathbf{p}_{i+1}$$

- This is an interpolation, i.e. the represented curve passes through all elements of the set of points forming the representation

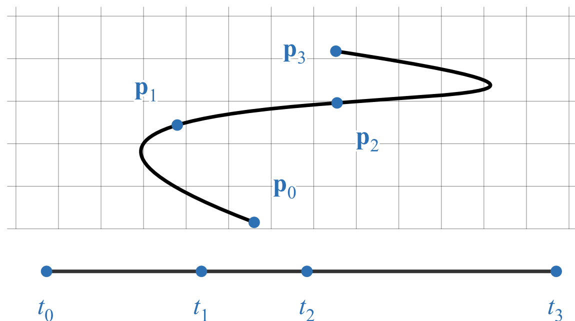
Polynomial curves

- If we want **guaranteed** (!) continuity of degree higher than C^0, we can try using polynomials
- An degree n polynomial can be fitted to points $\mathbf{p}_0, \dots, \mathbf{p}_n$
- In the well known power function $\mathbf{p}(t) = \sum_{i=0}^{n} \mathbf{a}_i t^i$, $t \in \mathbb{R}$ it has the form (e.g.: $\mathbf{p}(t) = \begin{bmatrix} 1 \\ 2 \end{bmatrix} t^2 + \begin{bmatrix} 0 \\ 1 \end{bmatrix} t + \begin{bmatrix} 1 \\ 1 \end{bmatrix}$)
- But what would be the geometric interpretation of the coefficients $\mathbf{a}_i \in \mathbb{R}^k$? What are they depicting?
- $\rightarrow$ let's look for another base, where the elements that make up the representation have a more tangible meaning
- But before that, let's solve the task!

Polynomial curves

- Let the points be $\mathbf{p}_0, ..., \mathbf{p}_n \in \mathbb{E}^3$ and the parameters $t_0 < t_1 < .. < t_n \in \mathbb{R}$
- Let's find the degree n $\mathbf{p}(t) = \sum_{i=0}^{n} \mathbf{a}_i t^i, \; t \in \mathbb{R}$ parametric curve, which interpolates the points at the prescribed parameters, i.e. to which

$$\mathbf{p}(t_i) = \mathbf{p}_i, \quad i = 0, 1, .., n$$



Polynomial curves

- We need to solve

$$\underbrace{\begin{bmatrix} 1 & t_0 & t_0^2 & .. & t_0^n \\ 1 & t_1 & t_1^2 & .. & t_1^n \\ .. & .. & .. & .. & .. \\ 1 & t_n & t_n^2 & .. & t_n^n \end{bmatrix}}_{V} \cdot \underbrace{\begin{bmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ .. \\ \mathbf{a}_n \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ .. \\ \mathbf{p}_n \end{bmatrix}}_{\mathbf{b}}$$

  system of linear equations, for the $\mathbf{a}_0, .., \mathbf{a}_n \in \mathbb{R}^3$ unknown coefficients
- If $det(V) \neq 0$, then there is a solution
- But: V is a Vandermonde matrix $\rightarrow$ determinant is not zero (since there is no $i \neq j$, such that $t_i = t_j$)
- Therefore the coefficients are $\mathbf{a} = V^{-1}\mathbf{b}$

Polynomial curves – parabola example

- So if, for example we want to interpolate the points $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ at $t = 0, 1, 2$ with a parabola, then we need to solve

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \end{bmatrix}$$

  equation

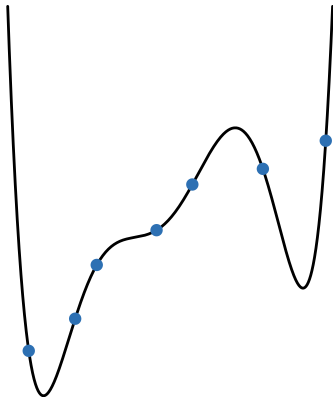
- That is, the coefficients we are searching for

$$\begin{bmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{3}{2} & 2 & -\frac{1}{2} \\ \frac{1}{2} & -1 & \frac{1}{2} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \end{bmatrix}$$

- And the parabola $\mathbf{p}(t) = \mathbf{a}_2 \cdot t^2 + \mathbf{a}_1 \cdot t + \mathbf{a}_0$

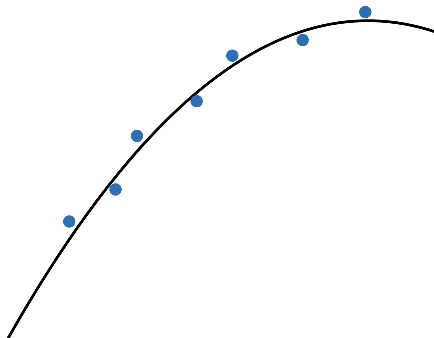
Interpolation or approximation?

Interpolation: the curve must pass through the control points



Interpolation or approximation?

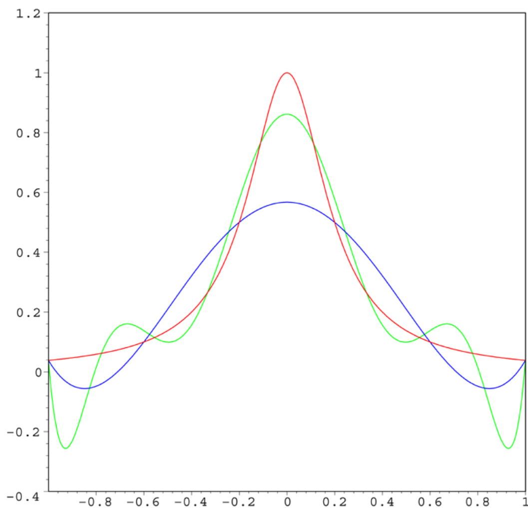
Approximation: the curve should only approximate the control points



A case of many control points

- With the number of control point the degree increases too $\rightarrow$ after a while we may have to settle for an approximation, but it won't be always good either
- High degree polynomials can strongly "ripple"
- Runge's phenomenon: during the approximation of the function $f(x) = \frac{1}{1+25x^2}$ (red) with polynomials of the fifth degree (blue) and ninth degree (green)

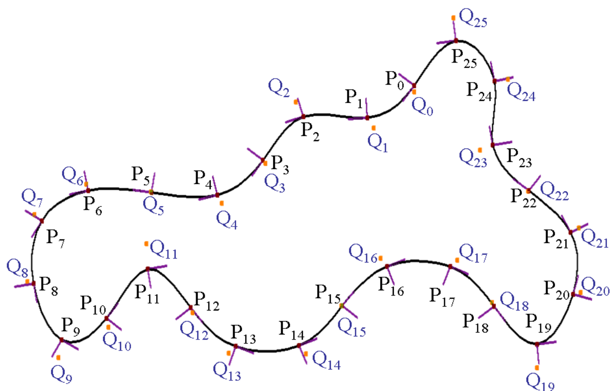
A case of many control points



Splines

- Do not try to interpolate or approximate our data points (control points) with a single polynomial
- Let's use a *complex curve*, which consists of lower degree segments
- In this way, we can eliminate the oscillations from the high degree polynomials and modifying one control point won't affect the entire curve, but we must pay attention to the continuous connection of the polynomial segments.
- And to much more, but...
- Details: Num.methods - BSc.; Geometric Modelling, Surface and Body Modeling - MSc.

Splines



Hermite interpolation

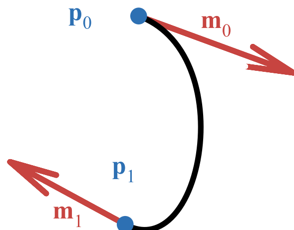
- To store our curve, at certain points of the curve we record the position, speed, acceleration vectors, etc. (i.e. let $\mathbf{x}(t_i), \mathbf{x}'(t_i), \mathbf{x}''(t_i), \ldots$ point and derivative data, $i = 0, 1, \ldots$)
- Let's find a base in which, by entering the above data as *coordinates*, the resulting curve will interpolate the entered data.

Cubic Hermite interpolation

- Between two points, expressed for $t \in [0,1]$, we search for a parametric curve with the form:

$$\mathbf{p}(t) = H_0^0(t)\mathbf{p}_0 + H_0^1(t)\mathbf{m}_0 + H_1^0(t)\mathbf{p}_1 + H_1^1(t)\mathbf{m}_1$$

$$\mathbf{p}(0) = \mathbf{p}_0$$
$$\mathbf{p}(1) = \mathbf{p}_1$$
$$\mathbf{p}'(0) = \mathbf{m}_0$$
$$\mathbf{p}'(1) = \mathbf{m}_1$$



Cubic Hermite basis functions

$$\mathbf{p}(t) = H_0^0(t)\mathbf{p}_0 + H_0^1(t)\mathbf{m}_0 + H_1^0(t)\mathbf{p}_1 + H_1^1(t)\mathbf{m}_1$$

- Let us find the unknown basis functions $H_i^j(t)$ as a cubic integer polynomial, i.e let

$$H_i^j(t) = a_{ij}t^3 + b_{ij}t^2 + c_{ij}t + d_{ij}$$

- We want $\mathbf{p}(0) = \mathbf{p}_0$, $\mathbf{p}(1) = \mathbf{p}_1$, $\mathbf{p}'(0) = \mathbf{m}_0$, $\mathbf{p}'(1) = \mathbf{m}_1$ to be true
- Then to be solved for $a_{ij}, b_{ij}, c_{ij}, d_{ij},\ i, j = 0, 1$

$$\begin{array}{cccc}
H_0^0(0) = 1 & H_0^1(0) = 0 & H_1^0(0) = 0 & H_1^1(0) = 0 \\
(H_0^0)'(0) = 0 & (H_0^1)'(0) = 1 & (H_1^0)'(0) = 0 & (H_1^1)'(0) = 0 \\
H_0^0(1) = 0 & H_0^1(1) = 0 & H_1^0(1) = 1 & H_1^1(1) = 0 \\
(H_0^0)'(1) = 0 & (H_0^1)'(1) = 0 & (H_1^0)'(1) = 0 & (H_1^1)'(1) = 1
\end{array}$$

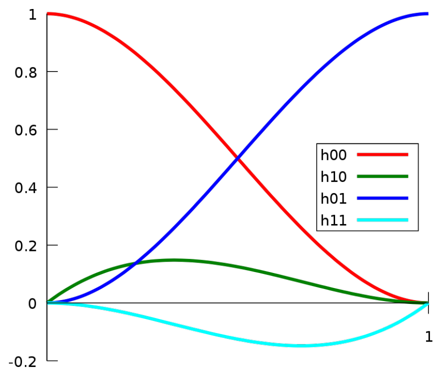
Cubic Hermite basis

- Then our cubic base will be:

$$
\begin{aligned}
H_0^0(t) &= 2t^3 - 3t^2 + 1 \\
H_0^1(t) &= t^3 - 2t^2 + t \\
H_1^0(t) &= -2t^3 + 3t^2 \\
H_1^1(t) &= t^3 - t^2
\end{aligned}
$$

- If we have $n+1$ incoming data, then by inserting one curve between each pair, after joining them we get a C^1 *spline* consisting of cubic segments

Cubic Hermite basis

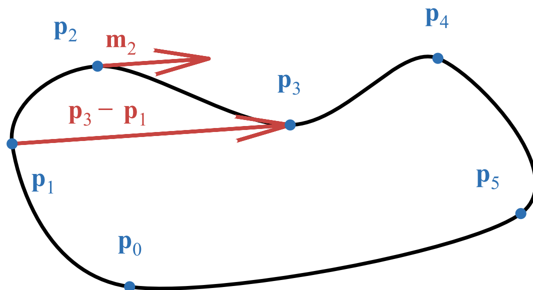


Catmull-Rom spline

- The derivative should not be given directly, but we calculate them from the neighboring points as follows:

$$\mathbf{m}_i = \frac{\mathbf{p}_{i+1} - \mathbf{p}_{i-1}}{t_{i+1} - t_{i-1}}$$

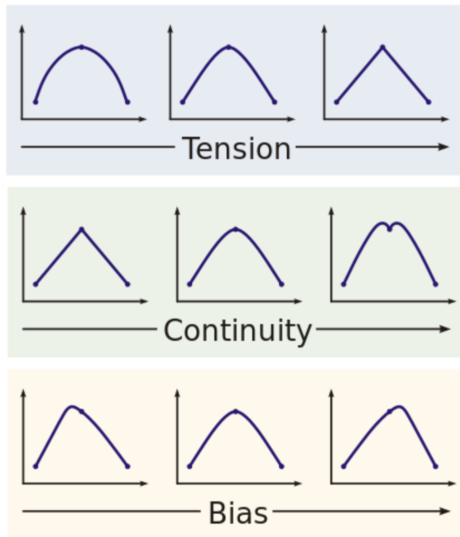
- Then, based on $\mathbf{p}_i$ and the calculated $\mathbf{m}_i$ we pairwise fit Hermite curves on the data



*Kochanek-Bartels spline

- As with the Catmull-Rom spline, we will calculate the tangent data, but three parameters have been added:
  - T: tension, $T \in [-1, 1]$
  - B: bias, $B \in [-1, 1]$
  - C: continuity, $C \in [-1, 1]$
- We get Catmull-Rom spline, if $T = B = C = 0$

*Kochanek-Bartels spline

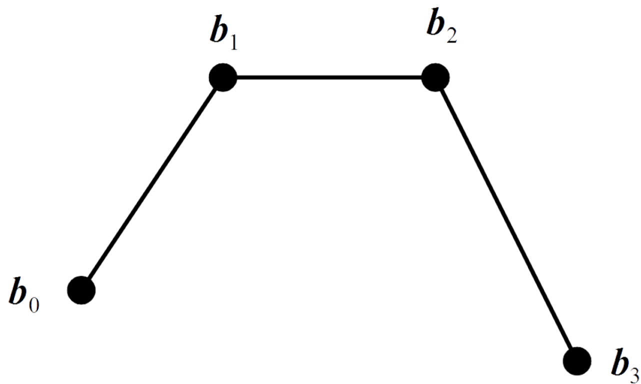


*Kochanek-Bartels spline

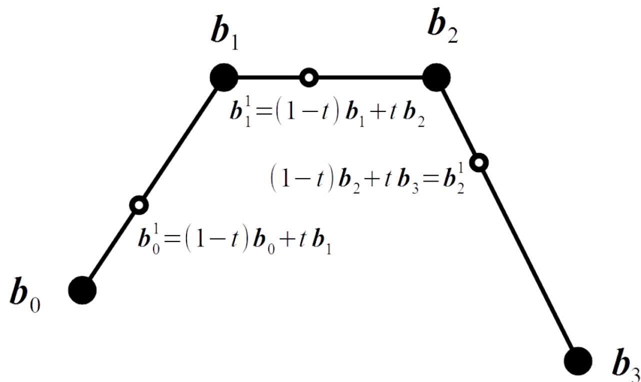
- Using the above, the derivatives of the two endpoints of the i-th segment should be

$$
\begin{aligned}
\mathbf{m}_i &= \frac{(1-T)(1+B)(1+C)}{2}(\mathbf{p}_i - \mathbf{p}_{i-1}) \\
&\quad + \frac{(1-T)(1-B)(1-C)}{2}(\mathbf{p}_{i+1} - \mathbf{p}_i) \\
\mathbf{m}_{i+1} &= \frac{(1-T)(1+B)(1-C)}{2}(\mathbf{p}_{i+1} - \mathbf{p}_i) \\
&\quad + \frac{(1-T)(1-B)(1+C)}{2}(\mathbf{p}_{i+2} - \mathbf{p}_{i+1})
\end{aligned}
$$

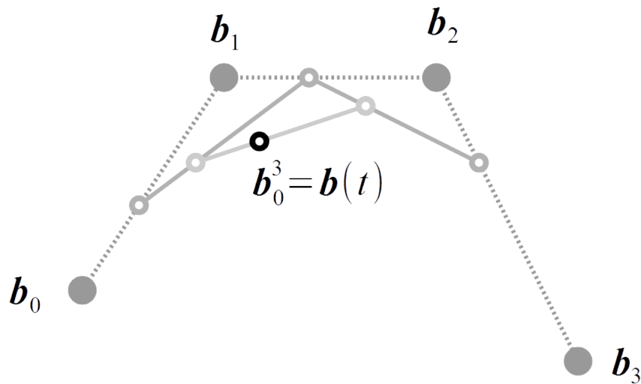
Bézier curve – de Casteljau's algorithm



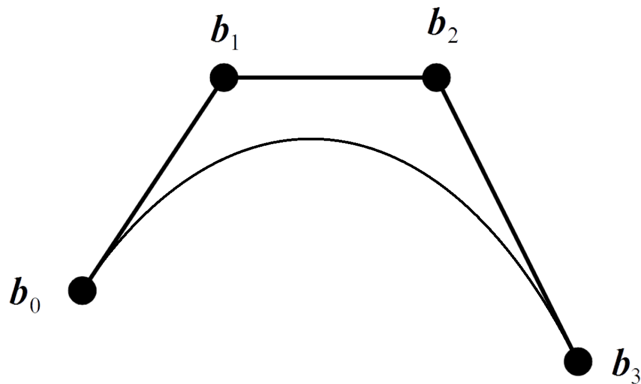
Bézier curve – de Casteljau's algorithm



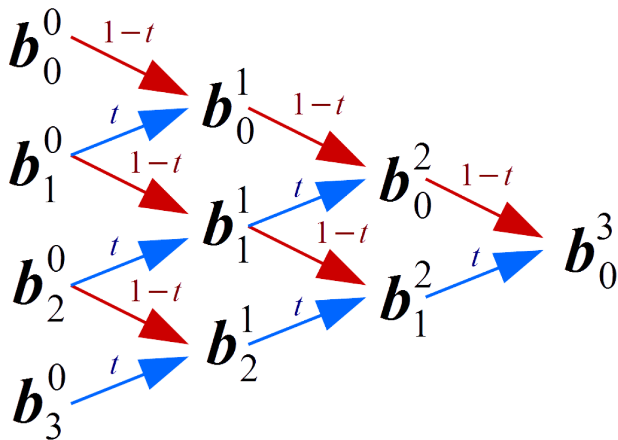
Bézier curve – de Casteljau's algorithm



Bézier curve – de Casteljau's algorithm



Bézier curve – de Casteljau's algorithm



Bézier curve – de Casteljau's algorithm

- [Demo]
- In general: a degree n Bézier curve has $n+1$ control points, denoted by $\mathbf{b}_i \in \mathbb{E}^3$, $i = 0, 1, \ldots, n$
- The de Casteljau algorithm is a recursive evaluation of the curve:

$$\mathbf{b}_i^{k+1} = (1-t)\mathbf{b}_i^k + t\mathbf{b}_{i+1}^k$$

where $\mathbf{b}_i^0 := \mathbf{b}_i, \quad k = 0, 1, \ldots, n-1, \quad i = 0, \ldots, n-k.$

- The point of the parametric curve corresponding to $t \in [0, 1]$

$$\mathbf{b}(t) := \mathbf{b}_0^n$$

Bézier curve – in Bernstein basis

- Let's use the *Bernstein basis*: $B_i^n(t) := \binom{n}{i} t^i (1-t)^{n-i}$
- The degree n Bézier curve defined by the $\mathbf{b}_0, \dots, \mathbf{b}_n \in \mathbb{R}^3$ control points is then

$$\mathbf{b}(t) = \sum_{i=0}^{n} B_i^n(t) \mathbf{b}_i,$$

  where $t \in [0, 1]$.
- Homework: $\sum_{i=0}^{n} B_i^n(t) = 1$ true, $\forall t \in [0, 1]$
- The curve "roughly" follows the shape of the control points's polygon, but it does not pass through all of them! This is an *approximation*
- More details: Geometric Modeling MSc, Analysis (Stone-Weierstrass approximation theorem)

Bézier curve

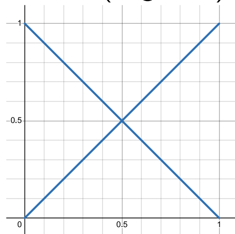
- Modifying a single control point affects the entire curve
- Bernstein basis for some n:
  - $B_0^1(t) = 1 - t, B_1^1(t) = t \rightarrow$ linear interpolation!
  - $B_0^2(t) = (1-t)^2, B_1^2(t) = 2t(1-t), B_2^2(t) = t^2$
  - $n = 3$: Homework
- In essence, we "blend" our control points by weighting them with the above functions, thus saying which control point plays an "important" role in determining the shape of the curve for a given parameter value $t \in [0, 1]$

Bézier curve properites

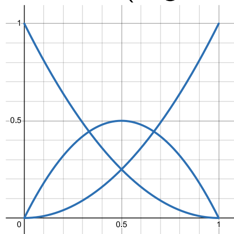
- The Bézier curve passes through the two end points (points $\mathbf{b}_0$ and $\mathbf{b}_n$)
- In the starting point, the tangent line of the curve is the line through the first two control points. Similar is true for the endpoint.
- Convex hull property: the Bézier curve runs inside the convex hull of its control points
- Affine invariance: the Bézier curve is *invariant to affine transformations*

Bernstein basis

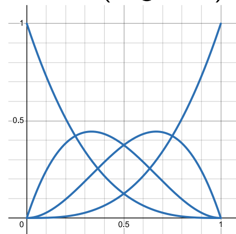
Linear (degree 1)



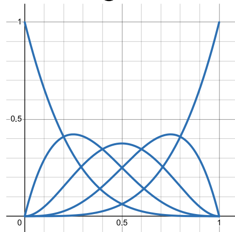
Quadratic (degree 2)



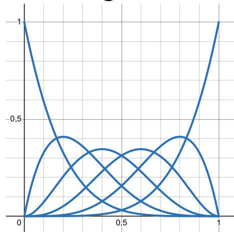
Cubic (degree 3)



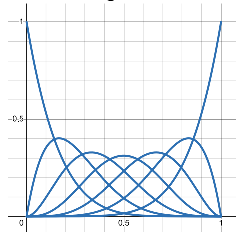
Degree 4



Degree 5



Degree 6



Subdivision curves

- Idea: so far, we immediately created a polynomial from our control points (essentially: from the control polygon defined by the control points)
- However during display we have to approximate it with segments $\rightarrow$ let's work on the control polygon itself!
- Subdivision, or schemes defined by recursive division, recursively "refine" our initial set of points (set of control points), giving an ever finer linear approximation (most of the time)

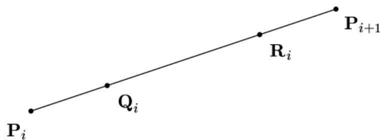
Subdivision curves

- The curve that we get from the initial set of points after "infinitely many" refinements is the limit curve
- High expressive power (for example, the Chaikin's corner cutting algorithm gives a quadratic B-spline curve), but in many cases it is possible to calculate curves more efficiently

Subdivision curves – Chaikin's corner cutting algorithm

- Let the current set of control points $\{\mathbf{p}_i \in \mathbb{R}^3\}_{i=0}^{n}$
- During iteration, our new control point set will be $\{\mathbf{q}_i, \mathbf{r}_i \in \mathbb{R}^3\}_{i=0}^{n-1}$, where

$$\mathbf{q}_i = \frac{3}{4}\mathbf{p}_i + \frac{1}{4}\mathbf{p}_{i+1}$$
$$\mathbf{r}_i = \frac{1}{4}\mathbf{p}_i + \frac{3}{4}\mathbf{p}_{i+1}$$



Subdivision curves – Chaikin's corner cutting algorithm