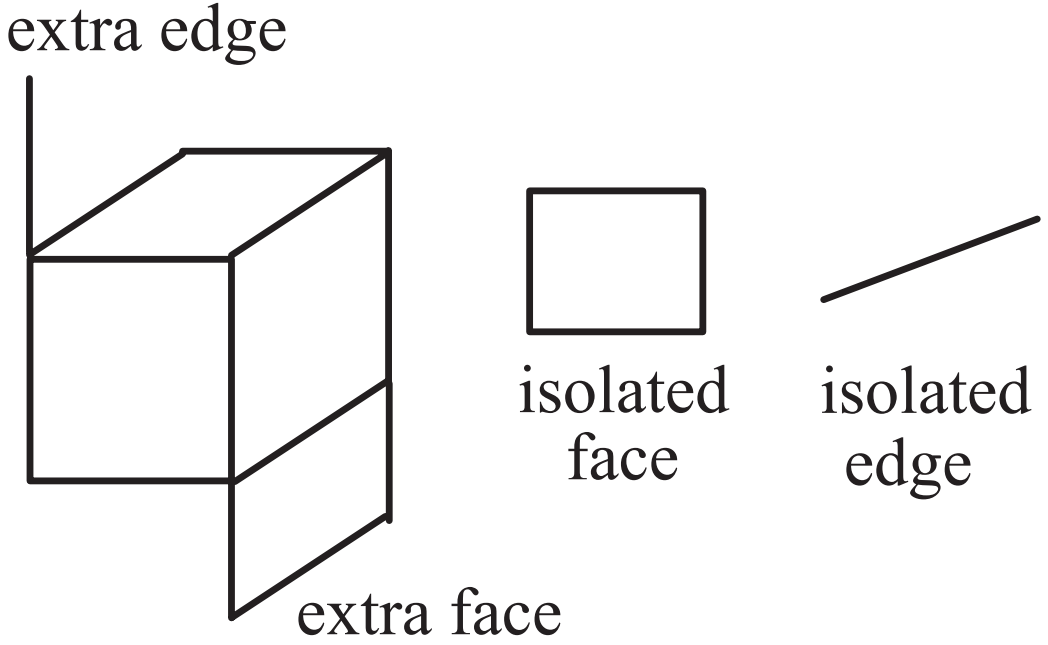
3D object modelling

Limitation to proper 3D structures:

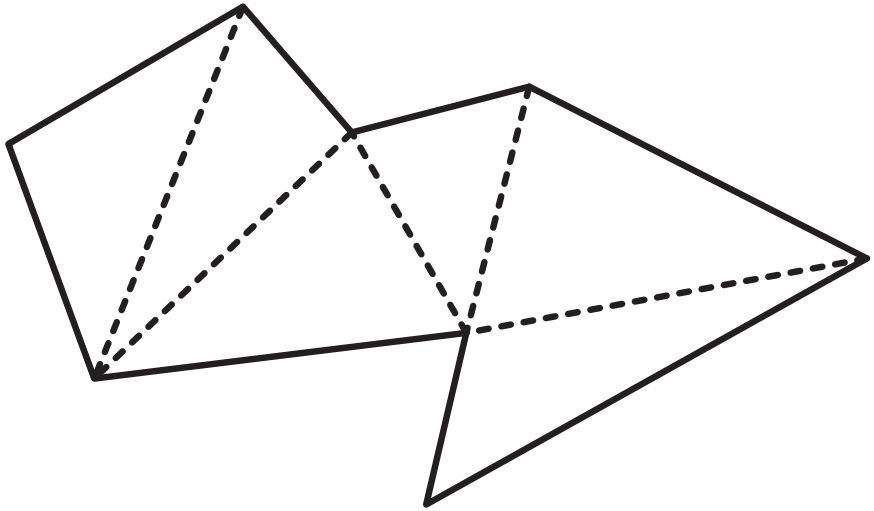
- no isolated points, edges and faces,
- no dangling or extra edges and faces.



Polygons as surface elements

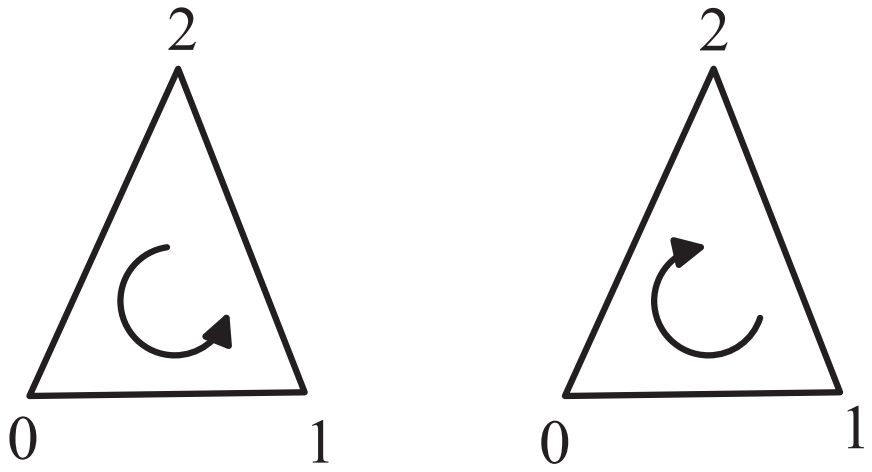
Modelling surfaces of 3D objects: flexible shapes for representing round and bent shapes.

Displaying surfaces: Approximation by polygons, usually triangles (apply triangulation if necessary).



Polygons as surface elements

Orientation of polygons: Vertices listed in anticlockwise order to indicate the outside of the surface.

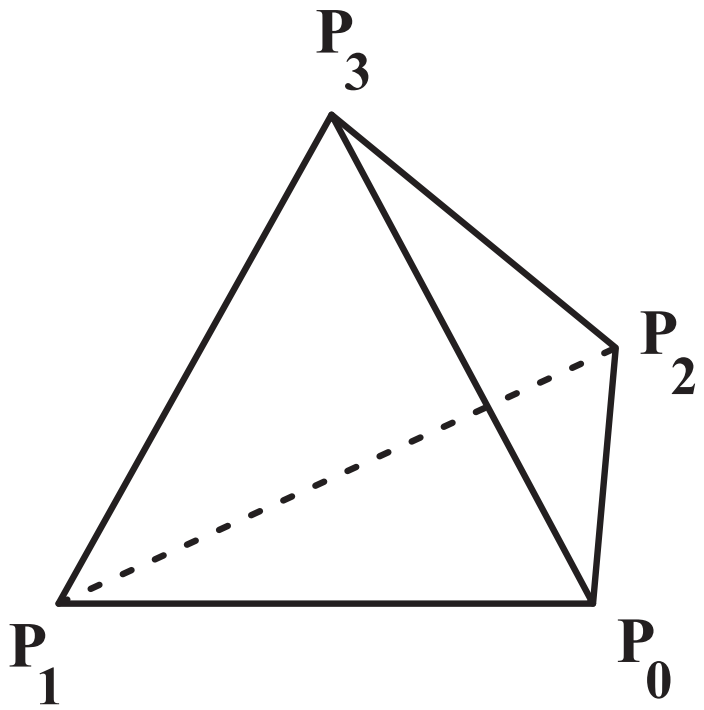


- 0,1,2: Visible for the viewer.
- 0,2,1: Invisible for the viewer (seeing the polygon from the back)

Example: Tetrahedron

Faces of the tetrahedron:

- $\mathbf{P}_0, \mathbf{P}_3, \mathbf{P}_1,$
- $\mathbf{P}_0, \mathbf{P}_2, \mathbf{P}_3,$
- $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2$
- $\mathbf{P}_1, \mathbf{P}_3, \mathbf{P}_2$



Topological notions

Let $M \subset \mathbb{R}^p$.

- A subset $U \subset \mathbb{R}^p$ is called a **neighbourhood** of the point $x_0 \in \mathbb{R}^p$ if there exists $\varepsilon > 0$ such that

$$\{x \in \mathbb{R}^P \mid \| x - x_0 \| < \varepsilon\} \subseteq U.$$

- A point $x \in M$ is called an **inner point** of M if there is a neighbourhood U of x such that $U \subseteq M$ holds.
- The set

$$\text{in}(M) = \{x \in M \mid x \text{ is an inner point of } M\}$$

of all inner points of M is called the **interior** or **kernel** of M.

Topological notions

- A point x is called a **boundary point** of M if every neighbourhood of x has nonempty intersections with M as well as with the complement of M.
- The set

  $$\text{bound}(M) = \{x \in M \mid x \text{ is a boundary point of } M\}$$

  of all boundary points of M is called the **boundary** of M.
- $\text{in}(M) = M \backslash \text{bound}(M)$.
- M is called an **open set** if M coincides with its interior, i.e., if $\text{in}(M) = M$ holds.

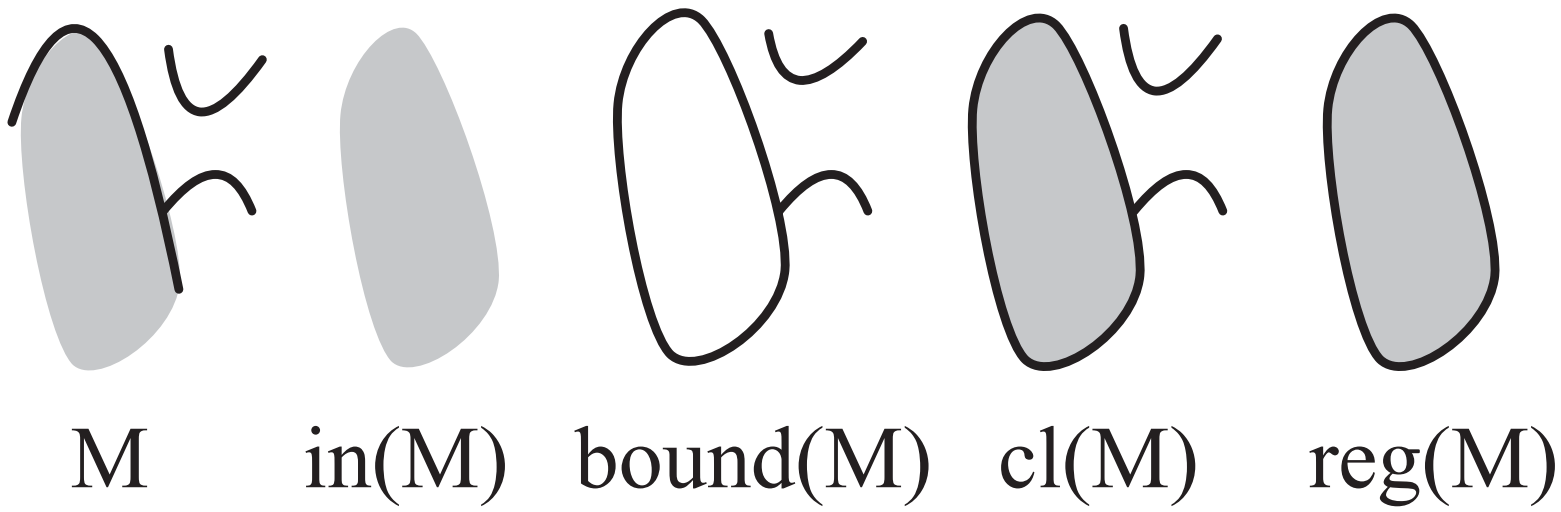
Topological notions

- The union of a set M with its boundary

$$\mathrm{cl}(M) = M \cup \mathrm{bound}(M)$$

  is the **closure** of M.
- M is called **closed** if the closure of M is M itself, i.e., if $\mathrm{cl}(M) = M$ holds.
- The **regularisation** $\mathrm{reg}(M) = \mathrm{cl}(\mathrm{in}(M))$ of a set M will cut off isolated as well as dangling edges and faces.
- M is called **regular** if $\mathrm{reg}(M) = M$ holds, i.e., if the set coincides with its regularisation.

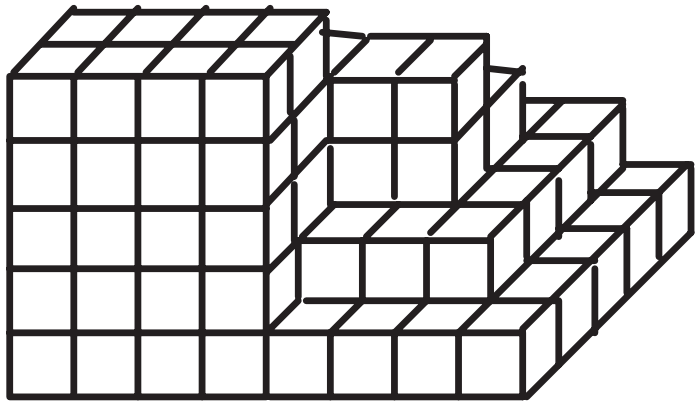
3D object modelling



Regularisation will cut off isolated as well as dangling edges and faces.

Voxel model

- Partitioning the three-dimensional space into a grid of of small, equisized cubes, called **voxels**.
- A three-dimensional object is defined by those voxels that are located inside the object.



Voxel model

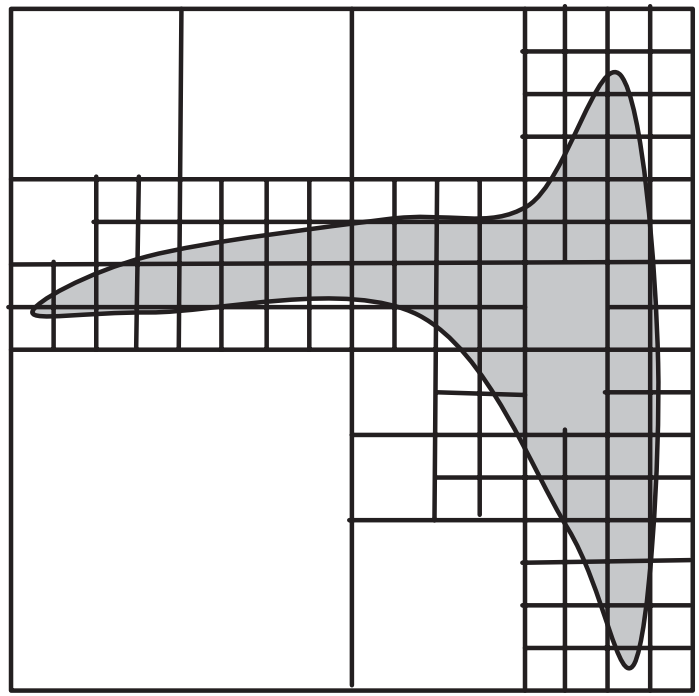
- A voxel belongs to a 3D object if its centre lies inside the 3D object.
- The quality of approximation of a 3D object by voxels depends on the size of the voxels.
  - Small voxels $\rightarrow$ Good approximation, but high computational and memory costs.
  - Larger voxels $\rightarrow$ Rough approximation, but low computational and memory costs.
- Store the 3D object as
  - a 3D binary matrix or
  - list of voxels.

Octrees

- Fit the 3D object into a sufficiently large cube or box.
- Partition the cube into eight smaller cubes.
- Smaller cubes that lie completely inside or completely outside the object are marked with *in* and *off*, respectively. No need for further refinement.
- The other cubes are marked with *on*, indicating that the cube intersects the surface of the object. The cubes are further refined in the same way.

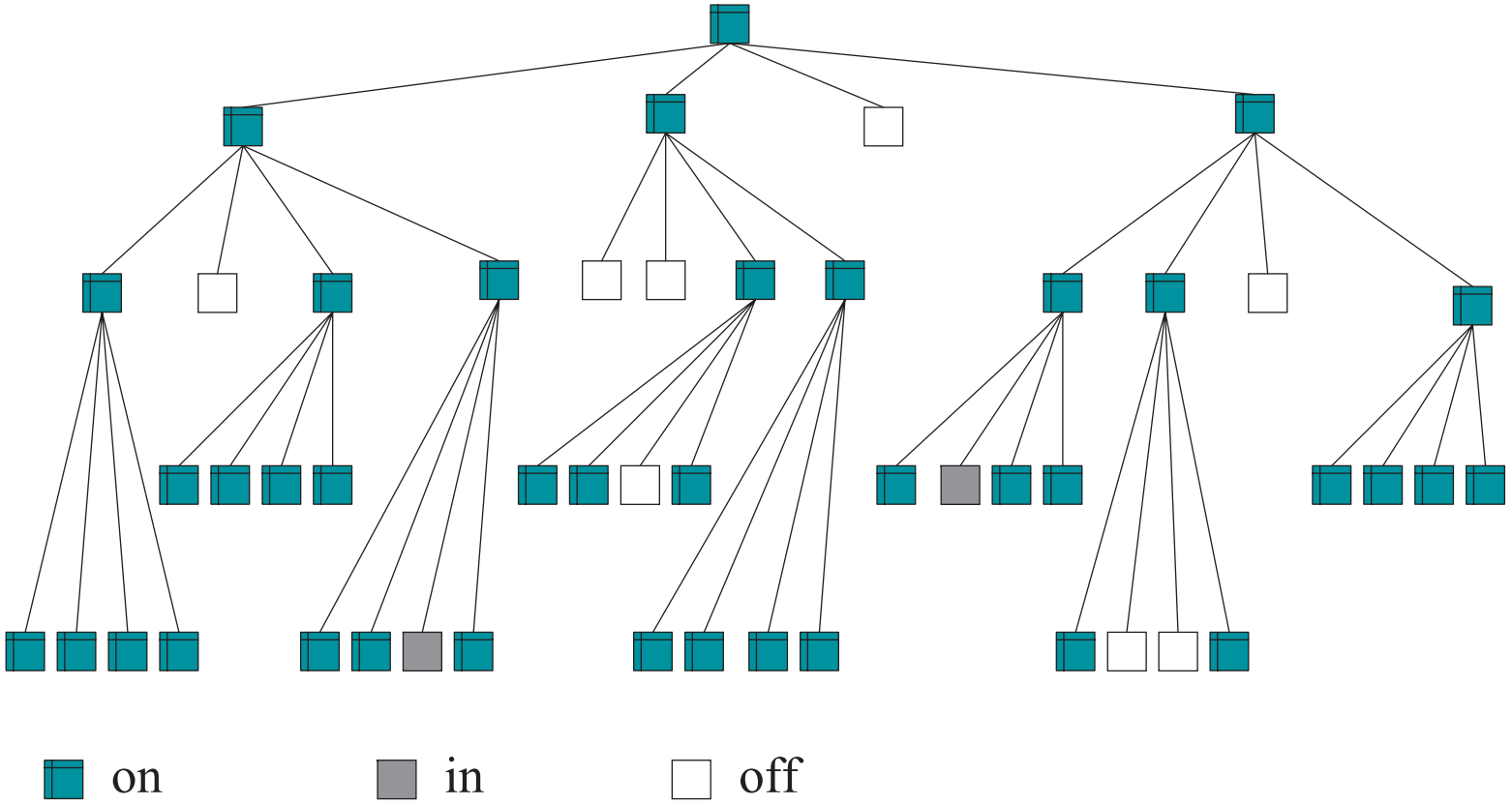
Octrees

2D counterpart: Quadtrees.



Octrees

Corresponding quadtree up to level 4:



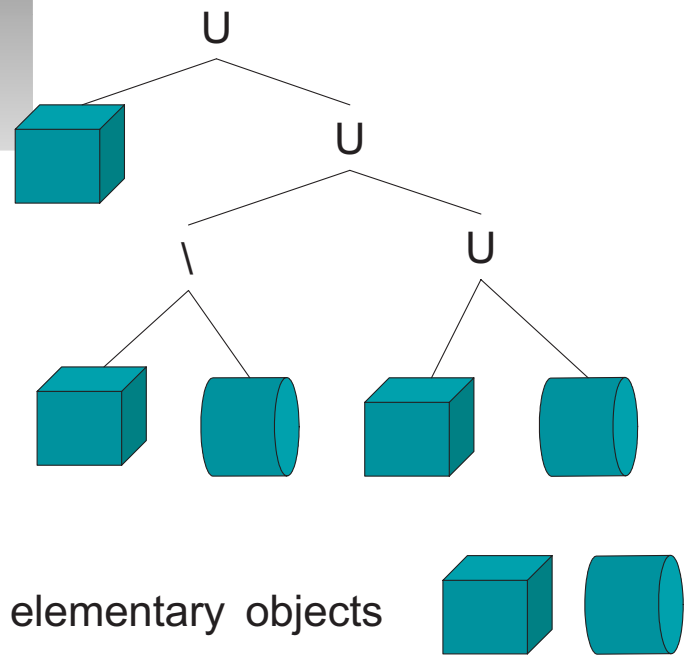
CSG scheme

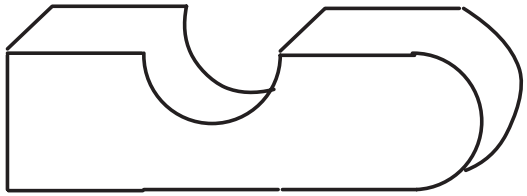
CSG: Constructive solid geometry

Similar to the use of the class `Area` in Java 2D:

- Collection of elementary geometric objects.(like box, sphere, cylinder, cone,. . .).
- Transformations and regularised set-theoretic operations can be applied to these objects to construct more complex objects.

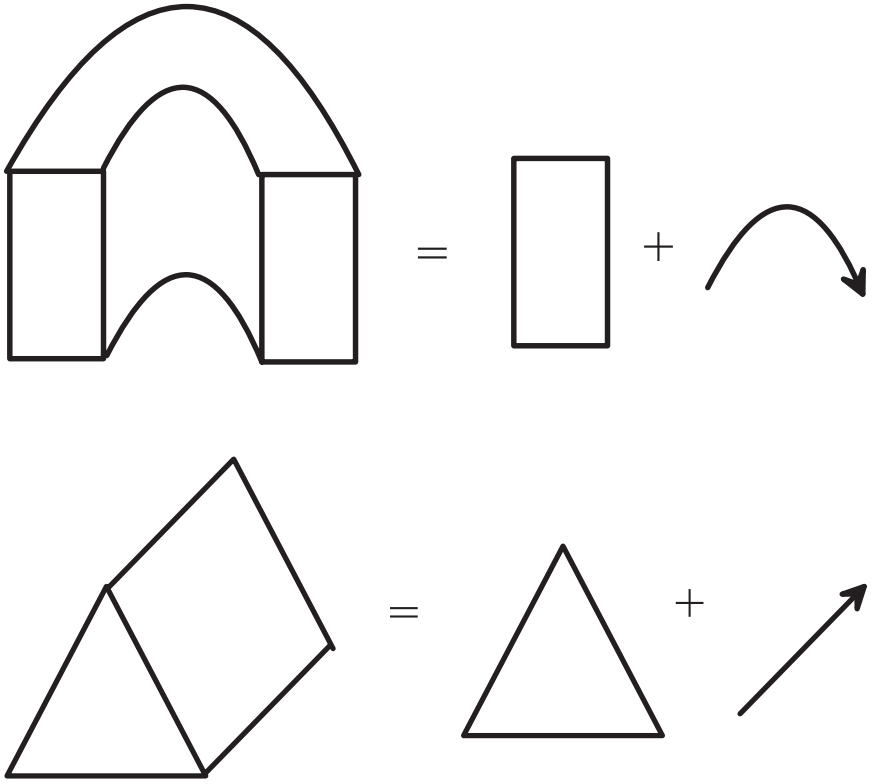
CSG scheme



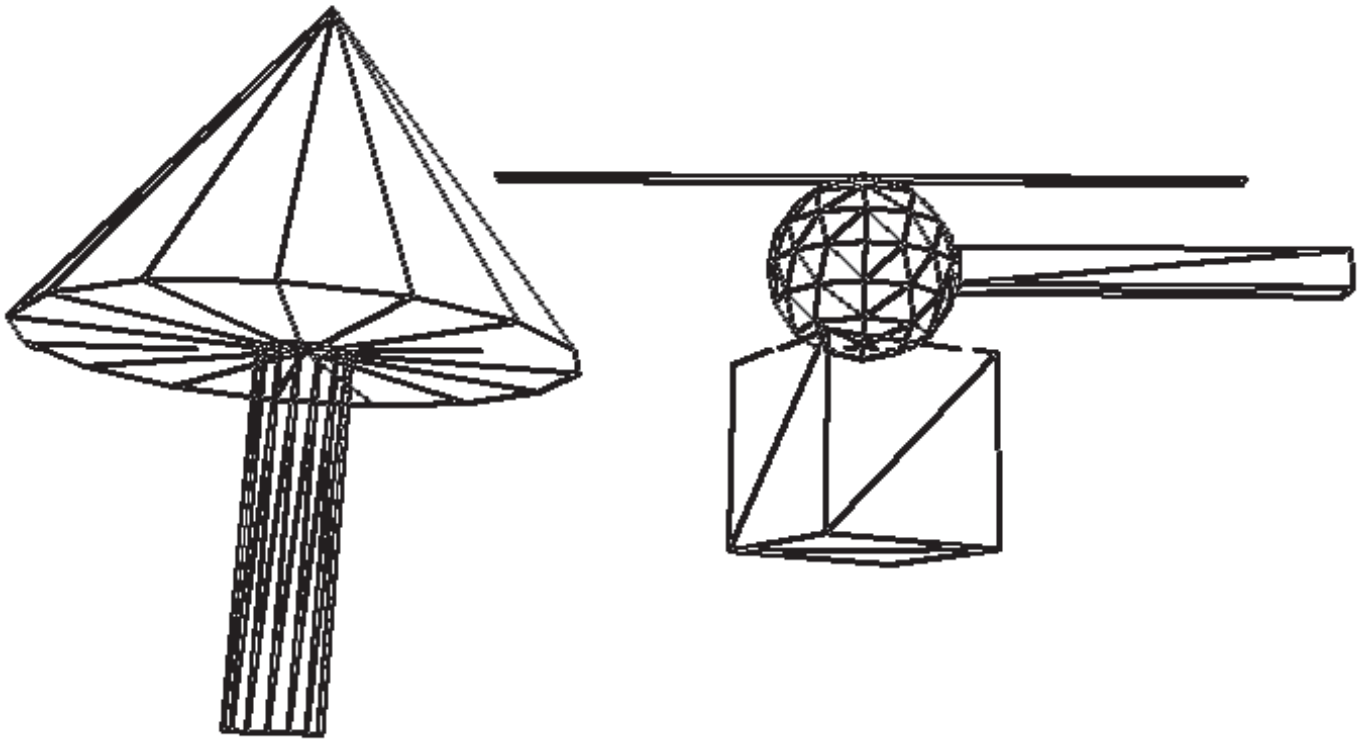


Sweep representation

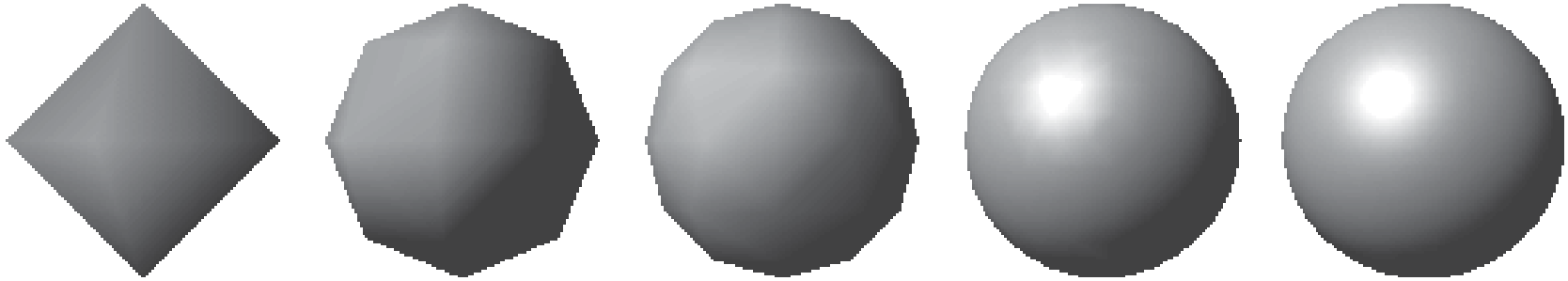
A three-dimensional object is generated from a two-dimensional shape that is moved along a trajectory.



Approximation by polygons



Approximation by polygons



Representation of a sphere with different tesselations

Tesselation in Java 3D

Displaying objects as wire frame models in Java 3D:

```
PolygonAttributes polygAttr =
                  new PolygonAttributes();

polygAttr.setPolygonMode(
      PolygonAttributes.POLYGON_LINE);

myApp.setPolygonAttributes(polygAttr);
```

Tesselation in Java 3D

Specifying the resolution for elementary geometric objects:

```
Sphere s = new Sphere(r,
                      Sphere.GENERATE_NORMALS,
                      res,sphereApp);
```

Approximation of the sphere's surface by `res` triangles at the circumference.

Tesselation in Java 3D

```
Cylinder c = new Cylinder(r,h,
                 Cylinder.GENERATE_NORMALS,
                 xres,yres,app);

Cone c = new Cone(r,h,
                      Cone.GENERATE_NORMALS,
                      xres,yres,app);
```

For the approximation of the surfaces, `xres` triangles are used around the circumference and `yres` triangles along the height.

Java 3D: GeometryArrays

A `GeometryArray` defines a 3D object (i.e. its surface) and mainly consists of

- points (vertices),
- faces (or polygons, usually triangles or quadrangles) defined via the specified vertices,
- information about colour or texture,
- normal vector containing information about the true structure of the surface.

Java 3D: GeometryArrays

A simple way to generate a `GeometryArray`:

- Partition the surface to be modelled into triangles.
- Define an array containing the vertices of the triangles.

```
Point3f[] vertices =
        {
          new Point3f(...),
          ...
          new Point3f(...)
        };
```

Java 3D: GeometryArrays

- Definition of the triangles.

```
int triangles[] = {
                    0,2,1,
                    1,4,7,
                    ...
                  };
```

The surface consists of the triangles given by the points (elements)

- 0, 2 and 1,
- 1, 4 and 7 in the `vertices` array.
- `...`

Java 3D: GeometryArrays

- First generate a `GeometryInfo` object:

```
GeometryInfo gi = new GeometryInfo(
        GeometryInfo.TRIANGLE_ARRAY);

    gi.setCoordinates(vertices);

    gi.setCoordinateIndices(triangles);

    NormalGenerator ng =
         new NormalGenerator();

    ng.generateNormals(gi);
```

Java 3D: GeometryArrays

- Generate an `GeometryArray` from the `GeometryInfo` object and turn this into a `Shape3D`.

```
GeometryArray ge = gi.getGeometryArray();

Shape3D inducedShape =
        new Shape3D(ge,desiredAppearance);
```

- This `Shape3D` object can be assigned to transformation groups in the same way as elementary geometric objects.

(see `GeomArrayExample.java`)

Loading a scene

Java 3D allows to import scenes in various graphics formats.

Example: **Wavefront Object** format (`.obj`)

This file format contains similar information as `GeometryArray`s.

Apart from importing some required packages, a scene can be loaded in the following way.

Loading a scene

```
ObjectFile f = new ObjectFile(
                        ObjectFile.RESIZE);

Scene s = null;

try
{
  s = f.load("file.obj");
}
catch (Exception e)
{
  System.out.println("Error ...");
}

BranchGroup theScene = s.getSceneGroup();
```

Loading a scene

The names of the objects contained in this scene are obtained in the following way.

```
Hashtable sObs = s.getNamedObjects();

Enumeration enum = sObs.keys();

String name;

while (enum.hasMoreElements())
{
  name = (String) enum.nextElement()
  System.out.println("Name: "+name);
}
```

Loading a scene

An object in a scene can be accessed by its name and its colour can be changed.

```
Shape3D sceneObject =
        (Shape3D) sObs.get("objName");

sceneObject.setAppearance(
                    desiredAppearance);
```

Loading a scene

Using only a part of a loaded scene.

```
Shape3D part = (Shape3D)
            namedObjects.get("partName");
Shape3D extractedPart = (Shape3D)
                part.cloneTree();
extractedPart.setAppearance(app);
tg.addChild(extractedPart);
```

(see `Load3DExample.java` and `Extract3DExample.java`)

Surfaces as functions

another way to define a surface: function in two variables in implicit form:

$$F(x, y, z) = 0$$

Disadvantages:

- Difficult to find an equation that models the desired surface.
- The equation needs to be solved for rendering

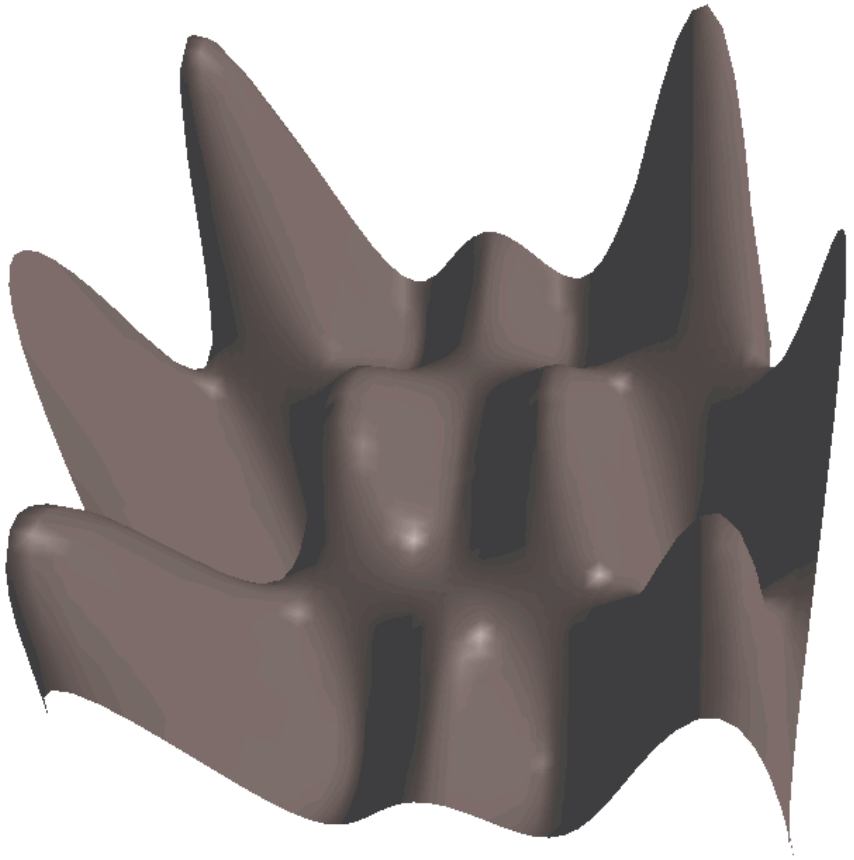
Surfaces as functions

simpler solution: Description of a surface as a function in explicit form

$$z = f(x, y).$$

Disadvantage: Closed surfaces must be defined piecewise.

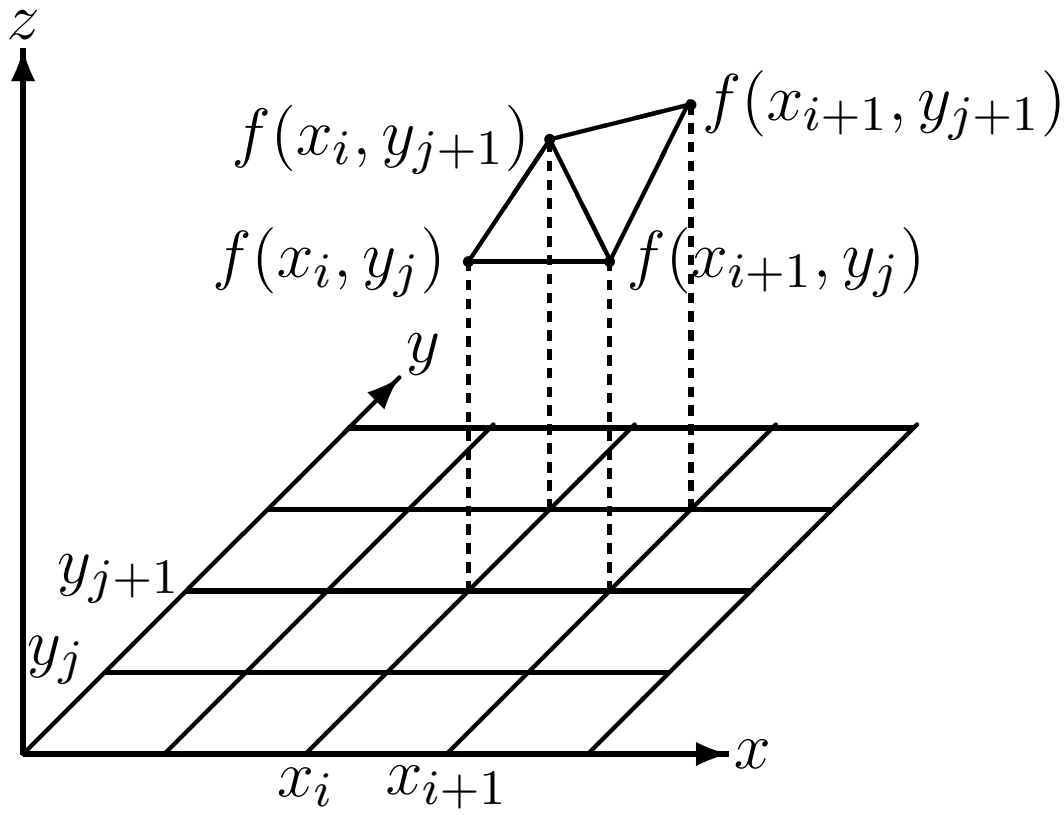
Surfaces as functions



$$z = f(x,y) = x\sin(7x)\cos(4y) \qquad (-1 \leq x,y \leq 1)$$

Surfaces as functions

Determining the triangles for the representation of the function:



Surfaces as functions

Choosing the resolution (tesselation) for the triangles:

- high resolution: good approximation of the function, but high computational costs.
- low resolution: bad approximation of the function, but low computational costs.

Surfaces as functions

possible solution:

variable resolution

In each square the approximation error is computed. Each square is recursively divided into smaller squares until the approximation error is small enough or a minimum size of the squares is reached.

Surfaces as functions

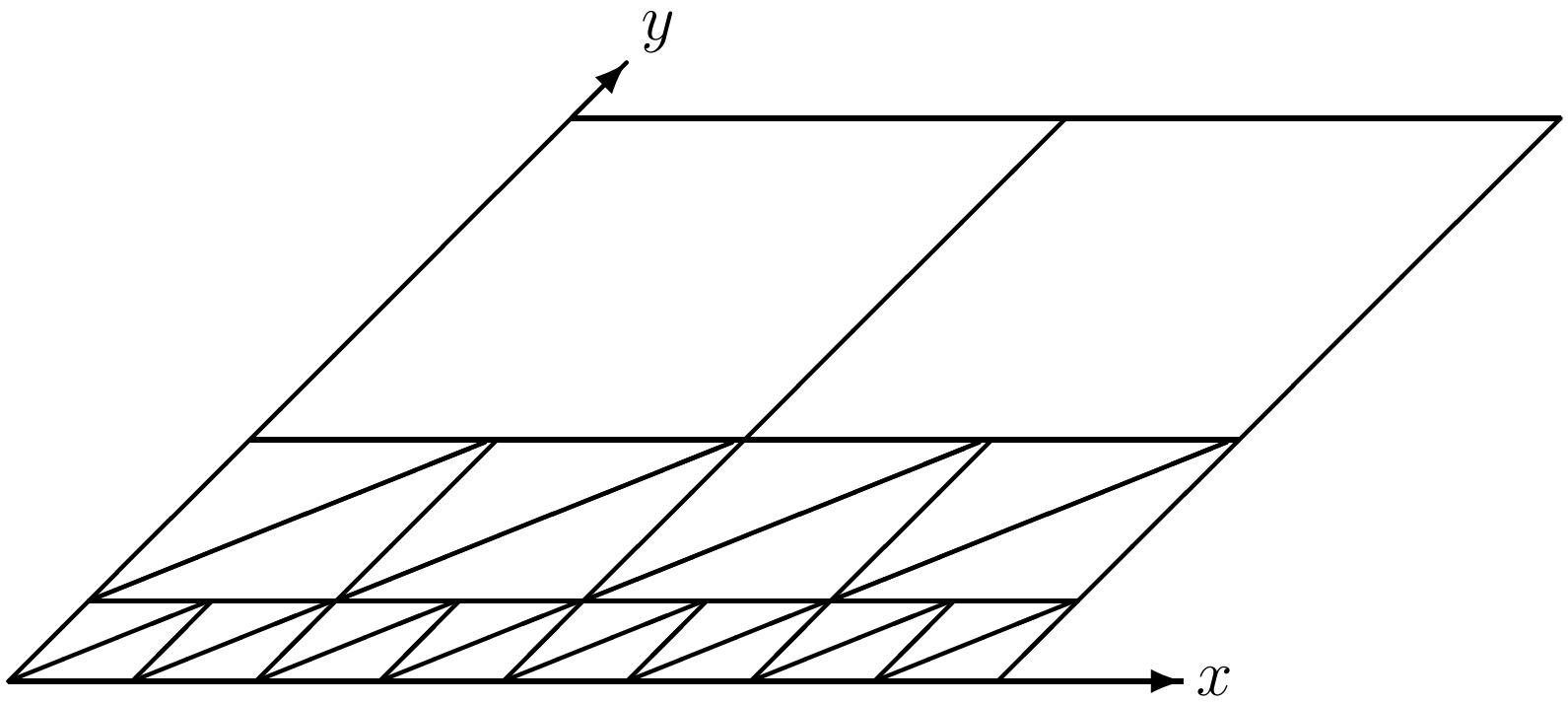
Landscapes consist of height and texture information.

The height information corresponds to sampling a function $z = f(x, y)$ where z is the height of the landscape at the point (x, y).

Problem: Ein narrow grid with height information leads to a large number of triangles to be rendered, even for those parts of the landscape that are far away from the viewer.

Level of Detail (LOD)

Resolution depending on the distance of the viewer:



Text

Classes for text in 3D: `Text2D` and `Text3D`
Creating a `Shape` object (similar to `Text2D`):

```
Font3D f3d = new Font3D(f,
                        new FontExtrusion());
Text3D t3d = new Text3D(f3d,s,p);
Shape3D textShape = new Shape3D(t3d,app);
```

Billboard Behaviour

For text as legend it is important that the text is always oriented to the viewer. This is called **Billboard Behaviour**.

```
Billboard bb = new Billboard(tgBBGroup,
                Billboard.ROTATE_ABOUT_POINT,
                                        p);
bb.setSchedulingBounds(bounds);
tgBBGroup.addChild(bb);
tgBBGroup.setCapability(
   TransformGroup.ALLOW_TRANSFORM_WRITE);
```

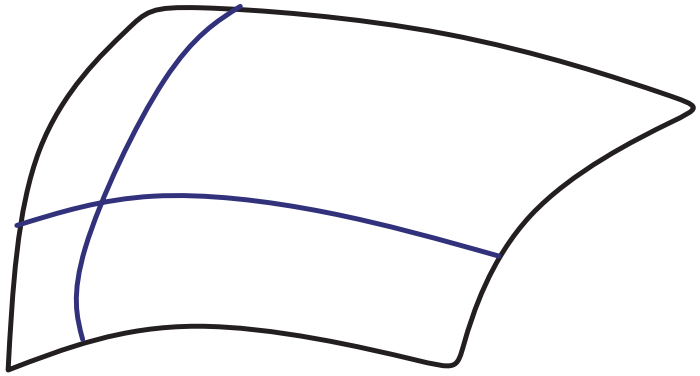
Surface modelling

- Round or bent surfaces are usually approximated by a sufficiently large number of small triangles.
- For modelling surfaces, it is more common to use freeform surfaces, especially for CAD applications.
- From a freeform surface model, different approximations by triangles can be generated, depending on the desired precision or on the distance of the viewer to the object (LOD: level of detail).

Surface modelling

- Normal vectors to surfaces are needed for shading effects (and also for elimination of invisible surfaces).
- The normal vectors for the triangles are taken as the normal vectors to the planar polygons that the triangles represent, but as normal vectors to the original bent surface.
- Methods for computing normal vectors will be introduced later on.

Parametric curves and surfaces



Parametric curves can be considered as a part of parametric surfaces.

Parametric curves

Parametric surfaces are defined on the basis of parametric curves (similar to `QuadCurve2D` or `CubicCurve2D` in Java 2D).

Parametric curves are defined via points (in 3D space), so-called control points.

Interpolation refers to curves that pass through all control points.

Approximation only requires that the curve gets close to the control points, but it does not have to pass through them.

Modelling curves

When curves (or also surfaces) are defined via a set of parameters (f.e. points in 3D space), the following properties will make modelling and adjusting such curves easier.

Controllability: The influence of the parameters on the shape of the curve can be understood in an intuitive way. When the shape of a curve has to be changed, it should be clear for the user which parameters he should modify in which way in order to achieve the desired change.

Modelling curves

Locality principle: It must be possible to carry out local changes on the curve. Modifying one control point should only change the curve in the neighbourhood of this control point and not alter the curve completely.

Smoothness: The curve should satisfy certain smoothness properties. It should not only be continuous without jumps, it should have no sharp bends. The latter property requires the curve to be differentiable. In some cases it is even necessary that higher derivatives exist. It also desirable that the curve is of bounded variation. This means it should stay somehow close to its control points.

Interpolation with polynomials

Given $(n+1)$ control points, there is always an interpolation polynomial of degree n or less that passes exactly through the control points.

Disadvantages:

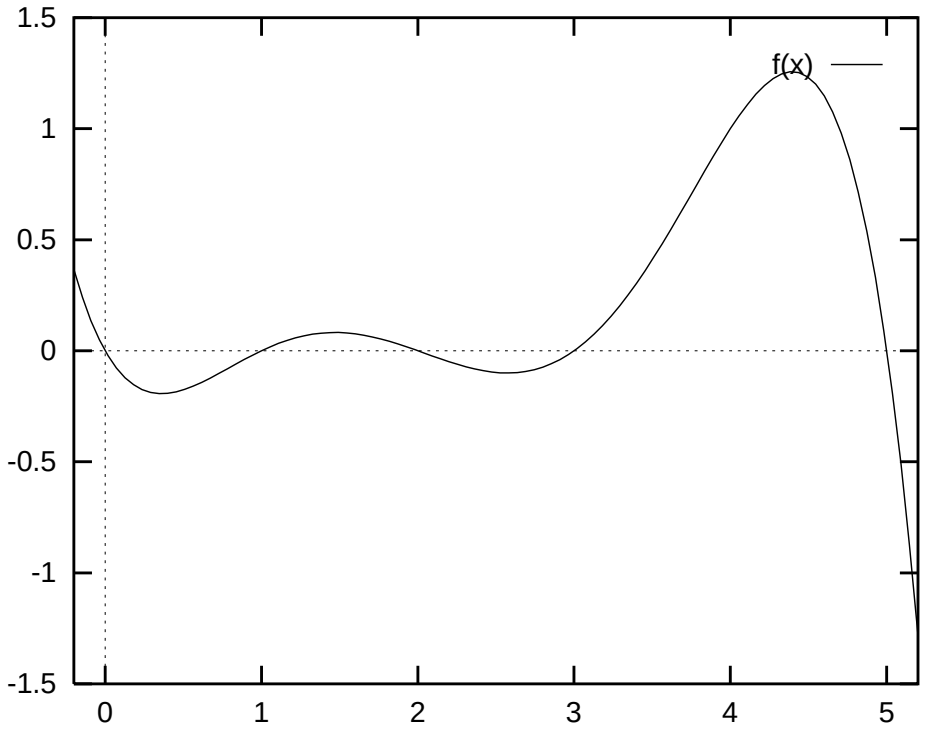
- For a larger number of points, the polynomial will have a high degree leading to high computational costs.
- Interpolation polynomials do not satisfy the locality principle.

Interpolation with polynomials

Disadvantages:

- Clipping for such polynomials is also not easy, since a polynomial interpolating a given set of control points can deviate arbitrarily from the region around the control points.
- Polynomials of high degree tend to oscillate between the control points.

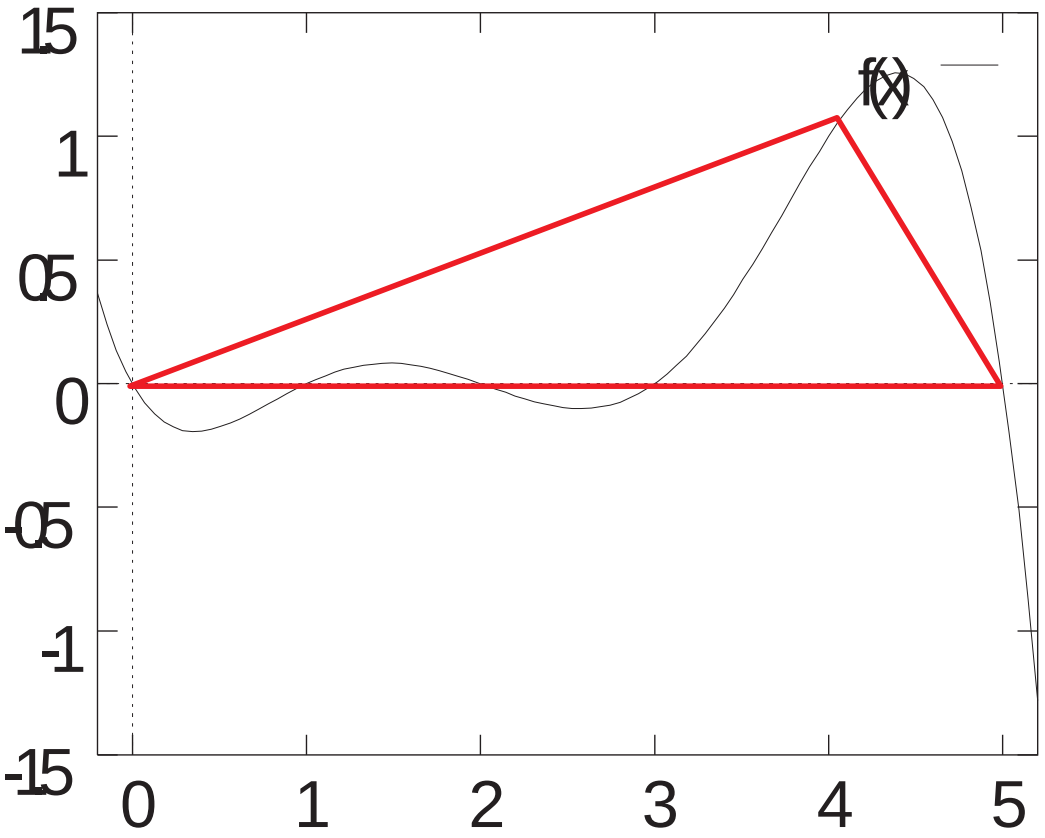
Interpolation with polynomials



Interpolation polynomial of degree 5 passing through the control points (0,0), (1,0), (2,0), (3,0), (4,1), (5,0):

$$f(x) = \frac{1}{24} \cdot \left(-x^5 + 11x^4 - 41x^3 + 61x^2 + 30x\right)$$

Interpolation with polynomials



The interpolation polynomial does not stay within the convex hull of the control points.

Bézier curves

i-th **Bernstein polynomial** of degree n ($i \in \{0, \ldots, n\}$):

$$B_i^{(n)}(t) = \binom{n}{i} \cdot (1-t)^{n-i} \cdot t^i \qquad (t \in [0,1])$$

Properties:

$$B_i^{(n)}(t) \in [0,1] \qquad \text{for all } t \in [0,1],$$

$$\sum_{i=0}^{n} B_i^{(n)}(t) = 1 \qquad \text{for all } t \in [0,1].$$

Bézier curves

Given $(n+1)$ control points $\mathbf{b}_0, \ldots, \mathbf{b}_n \in \mathbb{R}^p$ to define a curve in $\mathbb{R}^p$ (for computer graphics $p = 2$ or $p = 3$):

$$\mathbf{x}(t) \;=\; \sum_{i=0}^{n} \mathbf{b}_i \cdot B_i^{(n)}(t) \qquad (t \in [0,1])$$

is called **Bézier curve** of degree n.

$\mathbf{b}_0, \ldots, \mathbf{b}_n$ are called **Bézier** or **control points.**

Bézier curves

- $\mathbf{x}(0) = \mathbf{b}_0$ and $\mathbf{x}(1) = \mathbf{b}_n$ always hold.

  The Bézier curve interpolates the first and the last point.

  In general, the curve does not pass through the other control points.

- Furthermore:

$$\begin{aligned} \dot{\mathbf{x}}(0) &= n \cdot (\mathbf{b}_1 - \mathbf{b}_0) \\ \dot{\mathbf{x}}(1) &= n \cdot (\mathbf{b}_n - \mathbf{b}_{n-1}) \end{aligned}$$

  The tangent vector in the first point $\mathbf{b}_0$ points in the direction of the point $\mathbf{b}_1$ and the tangent vector in the last point $\mathbf{b}_n$ points in the direction of the point $\mathbf{b}_{n-1}$.

Bézier curves

- Due to the properties of the Bernstein polynomials, the Bézier curve is everywhere a convex combination of the control points.

  The Bézier curve stays within the convex hull of the control points.

- Bézier curves are invariant under affine transformations.

  When an affine transformation is applied to the control points, the resulting Bézier curve with respect to the new control points coincides with the transformed Bézier curve.

Bézier curves

- Bézier curves are symmetric w.r.t. the control points, i.e. the control points $\mathbf{b}_0, \ldots, \mathbf{b}_n$ and $\mathbf{b}_n, \ldots, \mathbf{b}_0$ lead to the same curve. The curve is only passed through in the reverse direction.
- Using a convex combination of control points of two sets of control points, the resulting Bézier curve is the convex combination of the two corresponding Bézier curves.
  - $\tilde{\mathbf{b}}_0, \ldots, \tilde{\mathbf{b}}_n$ defines the Bézier curve $\tilde{\mathbf{x}}(t)$.
  - $\hat{\mathbf{b}}_0, \ldots, \hat{\mathbf{b}}_n$ defines the Bézier curve $\hat{\mathbf{x}}(t)$.
  - $\alpha\tilde{\mathbf{b}}_0 + \beta\hat{\mathbf{b}}_0, \ldots, \alpha\tilde{\mathbf{b}}_n + \beta\hat{\mathbf{b}}_n$ defines the Bézier curve $\mathbf{x}(t) = \alpha\tilde{\mathbf{x}}(t) + \beta\hat{\mathbf{x}}(t)$, given $\alpha + \beta = 1$, $\alpha, \beta \geq 0$.

Bézier curves

- When all control points lie on a line or a parabola, then the resulting Bézier curve will be the corresponding line or parabola.
- Bézier curves also preserve certain geometrical shape properties like monotonicity or convexity of the control points.

Despite the nice properties of Bézier curves, they are not suited for larger sets of control points since this would lead to polynomials of high degree.

B-splines

B-splines are composed of a number of Bézier curves of lower polynomial degree – usually degree three or four.

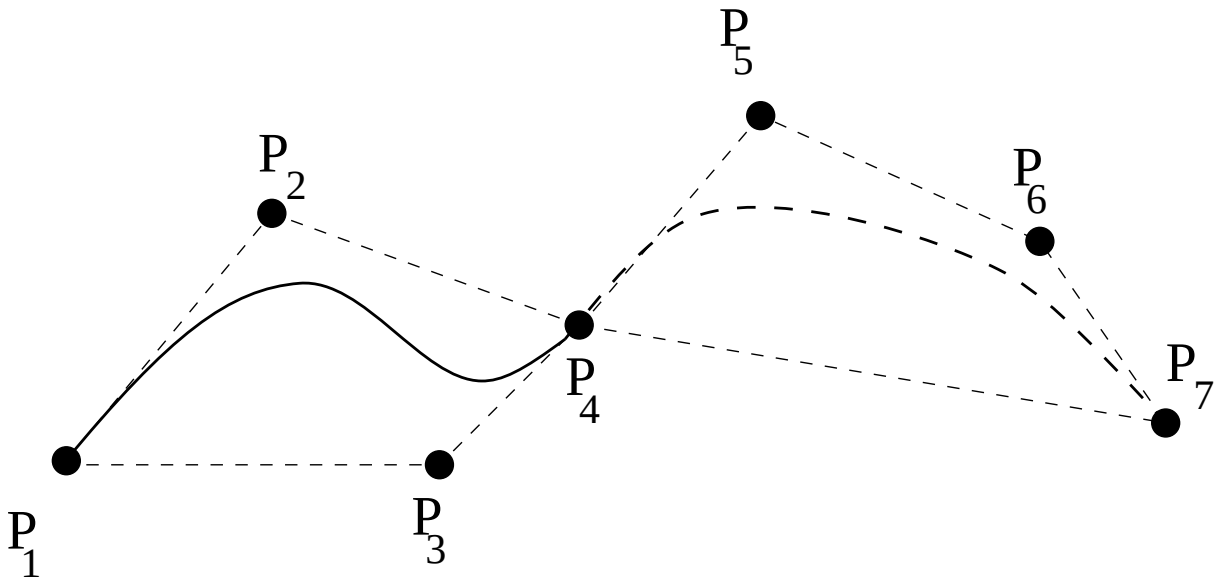
- For a sequence of n control points (for instance $n = 4$) a Bézier curve is computed.
- The last control point of the sequence is used as the starting point of the next sequence for the next Bézier curve.

B-splines

B-splines interpolate those control points where the single Bézier curves are glued together.

- These junctions are also called **knots**.
- The other control points are called **inner Bézier points**.

B-splines



B-spline with knots P_1, P_4, P_7 and inner Bézier points P_2, P_3, P_5, P_6

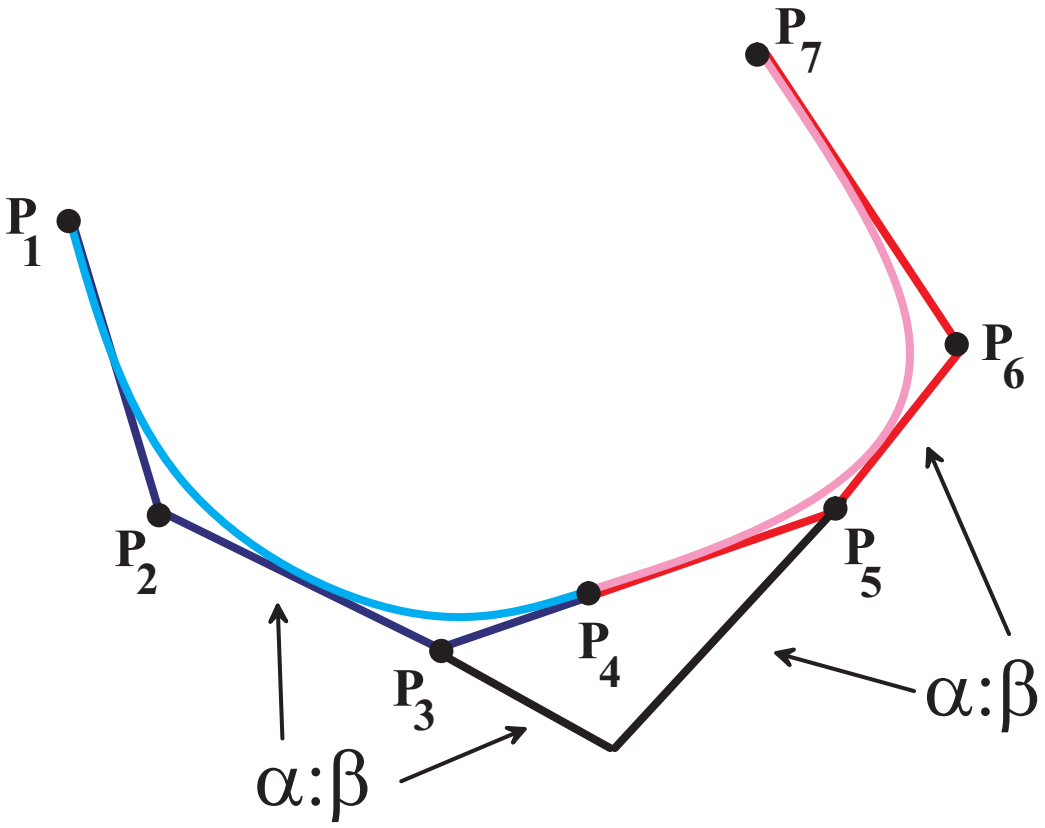
In order to avoid sharp bends at junctions between the Bézier curves, each knot and its two neighbouring inner Bézier points should be collinear.

B-splines

By choosing the inner Bézier points properly, a B-spline of degree n can be differentiated $(n-1)$ times.

Cubic B-splines are based on polynomials of degree three and can therefore be twice differential when the inner Bézier points are chosen correctly.

B-splines



B-splines

- stay within the convex hull of the control points,
- are invariant under affine transformations,
- interpolate the first and last control point,
- are symmetric in the control points and
- satisfy the locality principle.

B-splines

Perspective projection of a parametric curve in homogeneous coordinates:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{z_0} & 1 \end{pmatrix} \cdot \begin{pmatrix} P_x(t) \\ P_y(t) \\ P_z(t) \\ 1 \end{pmatrix} = \begin{pmatrix} P_x(t) \\ P_y(t) \\ 0 \\ \frac{P_z(t)}{z_0} + 1 \end{pmatrix}$$

B-splines

In Cartesian coordinates:

$$\begin{pmatrix} \frac{P_x(t)}{\frac{P_z(t)}{z_0}+1} \\ \\ \frac{P_x(t)}{\frac{P_z(t)}{z_0}+1} \\ \\ 0 \end{pmatrix}$$

Result: Rational function in t.

NURBS

Perspective projection of polynomials results in rational functions.

B-splines are not invariant under arbitrary projections.

NURBS (**non-uniform rational B-splines**) are generalisations of B-splines based on extensions of Bézier curves to rational functions in the following form.

$$\mathbf{x}(t) = \frac{\sum_{i=0}^{n} w_i \cdot \mathbf{b}_i \cdot B_i^{(n)}(t)}{\sum_{i=0}^{n} w_i \cdot B_i^{(n)}(t)}$$

Efficient polynomial evaluation

In order to draw a parametric curve (or surface), polynomials, usually of degree 3, have to be evaluated.

For drawing a cubic curve, the parametric curve is evaluated at equidistant values of the parameter t.

The corresponding points are computed and connected by line segments.

Efficient polynomial evaluation

Scheme of forward differences $\delta > 0$:

$$\Delta f(t) \;=\; f(t+\delta) - f(t)$$

i.e.

$$f(t+\delta) \;=\; f(t) + \Delta f(t)$$

or

$$f_{n+1} \;=\; f_n + \Delta f_n$$

For $f(t) = at^3 + bt^2 + ct + d$, this leads to

$$\Delta f(t) \;=\; 3at^2\delta + t(3a\delta^2 + 2b\delta) + a\delta^3 + b\delta^2 + c\delta.$$

Efficient polynomial evaluation

Applying a similar scheme of forward differences to the computation of $\Delta f(t)$ yields

$$\Delta^2 f(t) = \Delta(\Delta f(t)) = \Delta f(t+\delta) - \Delta f(t)$$

$$= 6a\delta^2 t + 6a\delta^3 + 2b\delta^2$$

$$\Delta f_n = \Delta f_{n-1} + \Delta^2 f_{n-1}$$

Efficient polynomial evaluation

Applying a similar scheme of forward differences to the computation of $\Delta^2 f(t)$ yields

$$\Delta^3 f(t) \;=\; \Delta^2 f(t+\delta) - \Delta^2 f(t) \;=\; 6a\delta^3$$

Initialisation: Set $t_0 = 0$.

$$\begin{aligned} f_0 &= d \\ \Delta f_0 &= a\delta^3 + b\delta^2 + c\delta \\ \Delta^2 f_0 &= 6a\delta^3 + 2b\delta^2 \\ \Delta^3 f_0 &= 6a\delta^3 \end{aligned}$$

Efficient polynomial evaluation

$t_0 = 0$		$t_0 + \delta$		$t_0 + 2\delta$		$t_0 + 3\delta$	$\ldots$
f_0	$\rightarrow$	$+$	$\rightarrow$	$+$	$\rightarrow$	$+$	$\ldots$
Δf_0	$\nearrow$ $\rightarrow$	$+$	$\nearrow$ $\rightarrow$	$+$	$\nearrow$ $\rightarrow$	$+$	$\ldots$
$\Delta^2 f_0$	$\nearrow$ $\rightarrow$	$+$	$\nearrow$ $\rightarrow$	$+$	$\nearrow$ $\rightarrow$	$+$	$\ldots$
$\Delta^3 f_0$	$\nearrow$ $\rightarrow$	$\Delta^3 f_0$	$\nearrow$ $\rightarrow$	$\Delta^3 f_0$	$\nearrow$ $\rightarrow$	$\Delta^3 f_0$	$\ldots$

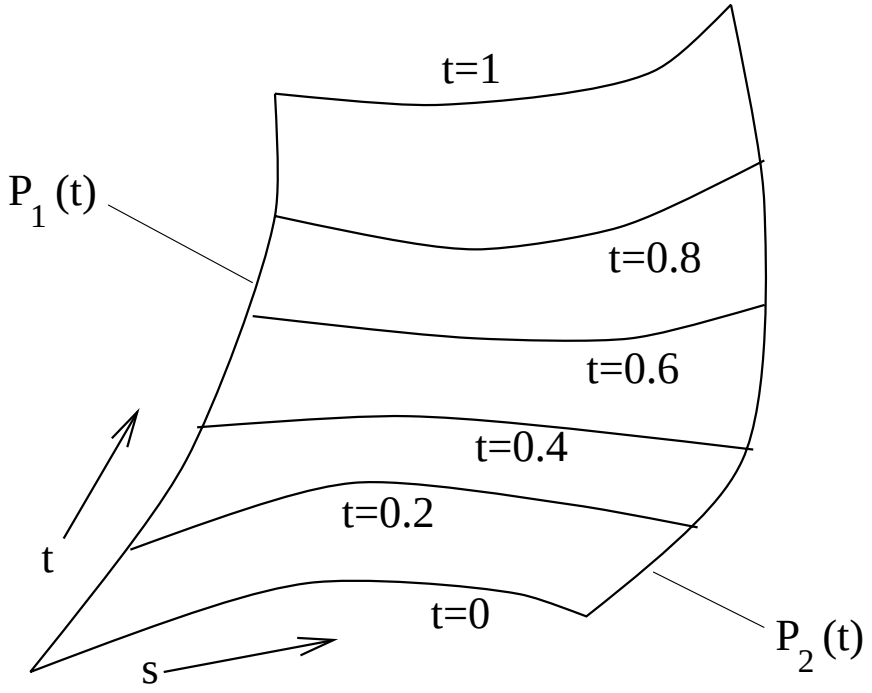
Scheme of forward differences for a polynomial of degree 3

Efficient polynomial evaluation

$t=0$		$t=1$		$t=2$		$t=3$		$t=4$	$\ldots$
3	$\rightarrow$	6	$\rightarrow$	15	$\rightarrow$	36	$\rightarrow$	75	$\ldots$
3	$\nearrow$ $\rightarrow$	9	$\nearrow$ $\rightarrow$	21	$\nearrow$ $\rightarrow$	39	$\nearrow$ $\rightarrow$	63	$\ldots$
6	$\nearrow$ $\rightarrow$	12	$\nearrow$ $\rightarrow$	18	$\nearrow$ $\rightarrow$	24	$\nearrow$ $\rightarrow$	30	$\ldots$
6	$\nearrow$ $\rightarrow$	6	$\nearrow$ $\rightarrow$	6	$\nearrow$ $\rightarrow$	6	$\nearrow$ $\rightarrow$	6	$\ldots$

Scheme of forward differences for the polynomial $f(t) = t^3 + 2t + 3$ with step width $\delta = 1$.

Freeform surfaces



Freeform surfaces as parametric curves in two parameters.

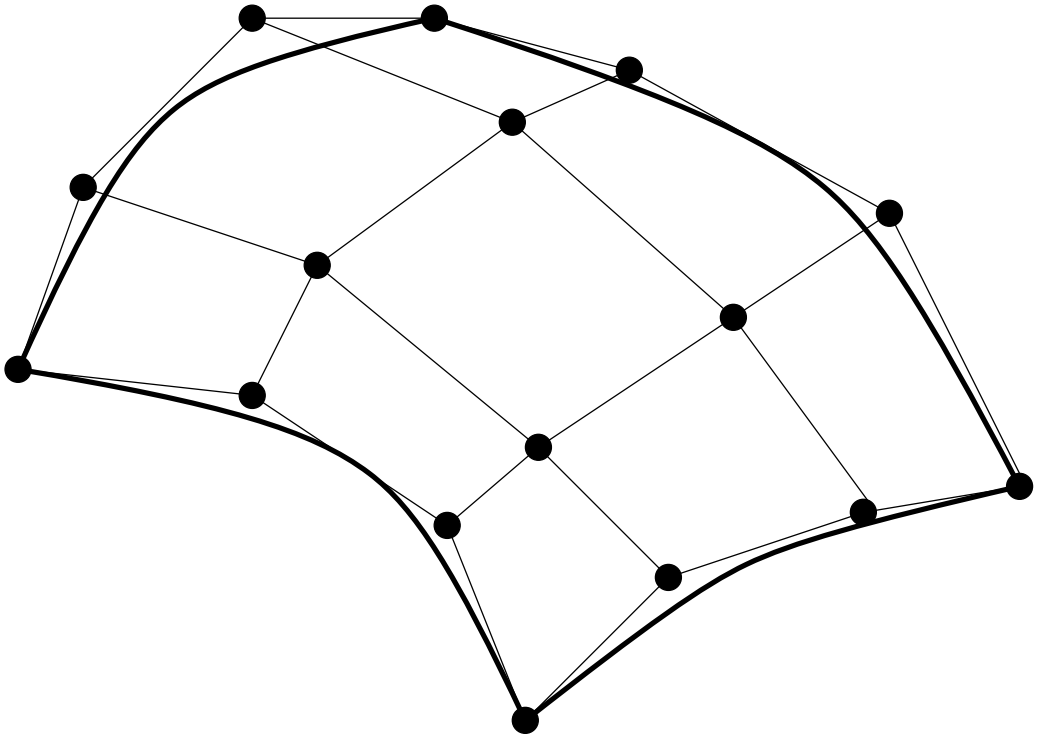
Freeform surfaces

Use of Bézier surfaces instead of Bézier curves.

$$\mathbf{x}(s,t) = \sum_{i=0}^{n}\sum_{j=0}^{m} \mathbf{b}_{ij} \cdot B_i^{(n)}(s) \cdot B_j^{(m)}(t)$$

- with $s, t \in [0,1]$ and
- $(m+1) \times (n+1)$ specified control points $\mathbf{b}_{ij}$.

Freeform surfaces



Freeform surfaces

- The four outer points $\mathbf{b}_{00}, \mathbf{b}_{0m}, \mathbf{b}_{n0}, \mathbf{b}_{nm}$ lie on the surface, the other points generally not.
- The surface stays within the convex hull of the control points.
- The curves with constant value $s = s_0$ are Bézier curves w.r.t. the control points

$$\mathbf{b}_j = \sum_{i=0}^{n} \mathbf{b}_{ij} \cdot B_j^{(n)}(s_0).$$

Analogously for curves with constant value $t = t_0$.

Freeform surfaces

Tesselations, as they are required in computer graphics, approximate surfaces with triangles.

Bézier surfaces of degree $n = 3$ defined over a grid of triangles:

$$\mathbf{x}(t_1, t_2, t_3) = \sum_{i,j,k \geq 0 : i+j+k=n} \mathbf{b}_{ijk} \cdot B_{ijk}^{(n)}(t_1, t_2, t_3)$$

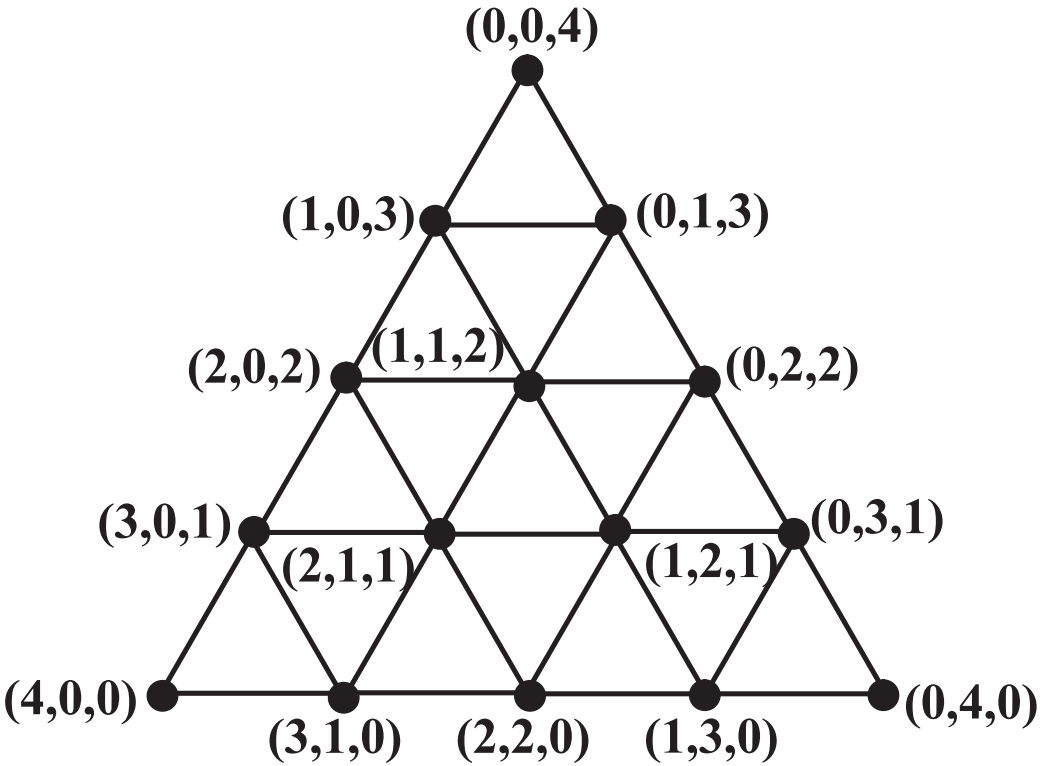
with the Bernstein polynomials

$$B_{ijk}^{(n)}(t_1, t_2, t_3) = \frac{n!}{i!j!k!} \cdot t_1^i \cdot t_2^j \cdot t_3^k$$

where $t_1 + t_2 + t_3 = 1$, $t_1, t_2, t_3 \geq 0$ and

$i + j + k = n$, $i, j, k \in \mathbb{N}$.

Triangular grid



Normal vectors for triangles

Equation of the plane E induced by a triangle:

$$Ax + By + Cz + D = 0.$$

The vector (A, B, C) is the (nonnormalised) normal vector to the plane:

Let $\mathbf{n} = (n_x, n_y, n_z)^\top$ (nonnormalised) normal vector.

Let $\mathbf{v} = (v_x, v_y, v_z)^\top$ be a point in the plane E.

Normal vectors for triangles

$(x, y, z)^\top \in E \Leftrightarrow$ The vector connecting $\mathbf{v}$ and $(x, y, z)^\top$ lies in the plane, i.e. this vector is also orthogonal to the normal vector.

$$0 = \mathbf{n}^\top \cdot \Big((x, y, z)^\top - \mathbf{v} \Big) = n_x \cdot x + n_y \cdot y + n_z \cdot z - \mathbf{n}^\top \cdot \mathbf{v}$$

Define $A = n_x$, $B = n_y$, $C = n_z$ and $D = \mathbf{n}^\top \cdot \mathbf{v}$.

Normal vectors for triangles

Given at least three (noncollinear) points $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ in a plane, the normal vector can be calculated by the cross product by

$$\mathbf{n} = (\mathbf{P}_2 - \mathbf{P}_1) \times (\mathbf{P}_3 - \mathbf{P}_1)$$

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \times \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} y_1 \cdot z_2 - y_2 \cdot z_1 \\ z_1 \cdot x_2 - z_2 \cdot x_1 \\ x_1 \cdot y_2 - x_2 \cdot y_1 \end{pmatrix}$$

The value D is obtained by inserting one of the points of the triangle: $D = \mathbf{n}^\top \cdot \mathbf{P}_1$

Normal vectors for surfaces

s-tangent vector:

$$\left(\frac{\partial}{\partial s}\mathbf{x}(s,t_0)\right)_{s=s_0} = \left(\frac{\partial}{\partial s}\sum_{i=0}^{n}\sum_{j=0}^{m}\mathbf{b}_{ij}\cdot B_i^{(n)}(s)\cdot B_j^{(m)}(t_0)\right)_{s=s_0}$$

$$= \sum_{j=0}^{m} B_j^{(m)}(t_0)\cdot\sum_{i=0}^{n}\mathbf{b}_{ij}\cdot\left(\frac{\partial B_i^{(n)}(s)}{\partial s}\right)_{s=s_0}$$

Normal vectors for surfaces

t-tangent vector:

$$\left(\frac{\partial}{\partial t}\mathbf{x}(s_0,t)\right)_{t=t_0} = \left(\frac{\partial}{\partial t}\sum_{i=0}^{n}\sum_{j=0}^{m}\mathbf{b}_{ij}\cdot B_i^{(n)}(s_0)\cdot B_j^{(m)}(t)\right)_{t=t_0}$$

$$= \sum_{i=0}^{n} B_i^{(n)}(s_0)\cdot\sum_{j=0}^{m}\mathbf{b}_{ij}\cdot\left(\frac{\partial B_j^{(m)}(t)}{\partial t}\right)_{t=t_0}$$

Normal vectors for surfaces

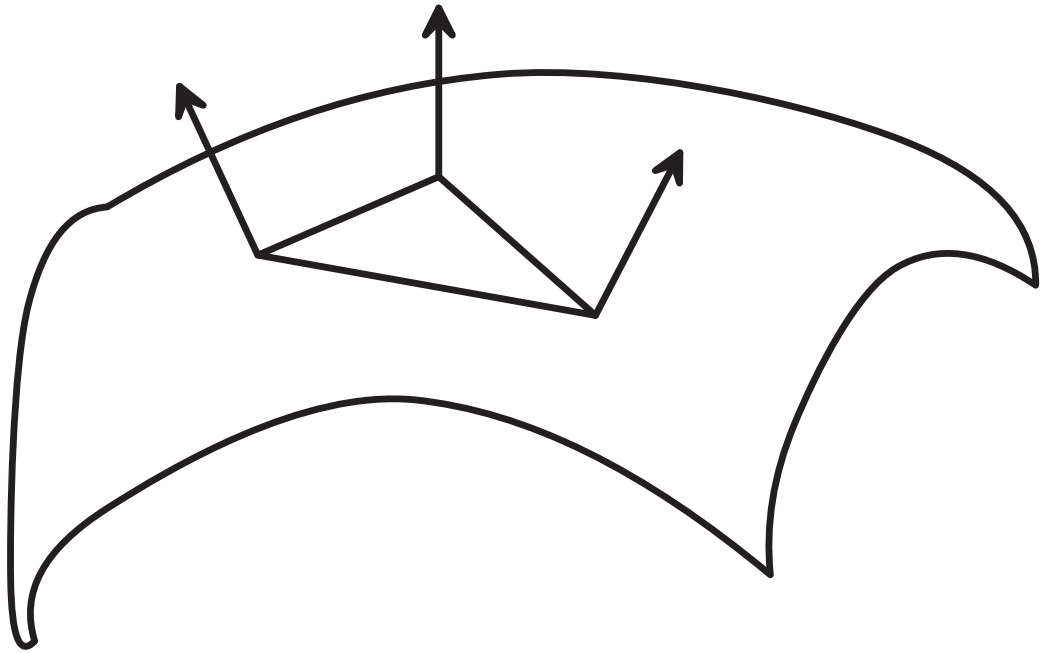
The two tangent vectors are parallel to the surface in the point (s_0, t_0) and induce the tangent plane in this point.

Normal vector to the surface at the point $\mathbf{x}(s_0, t_0)$:

$$\frac{\partial}{\partial s}\mathbf{x}(s,t) \times \frac{\partial}{\partial t}\mathbf{x}(s,t).$$

Varying normal vectors for triangles

The normal vectors for the three vertices of a triangle should be computed and stored as the normal vectors to the original surface in the corresponding points.



Normal vectors for surfaces

When modelling a surface in Java 3D with a `GeometryInfo` object, the normal vectors will be computed w.r.t. to the planar triangles.

By setting the crease angle, interpolation of normal vectors of triangles can be enforced.

```
NormalGenerator ng =
                    new NormalGenerator();

ng.setCreaseAngle(alpha);
```

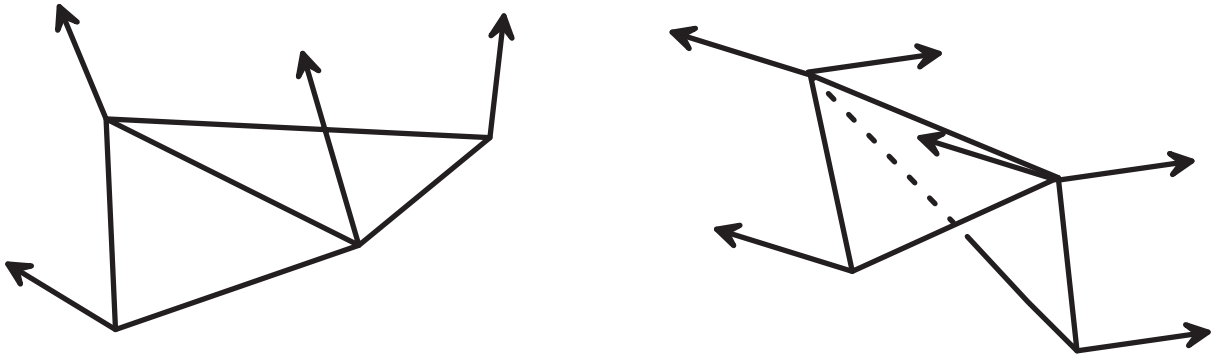
Normal vectors for surfaces

The angle `alpha` specifies how much neighbouring triangles may deviate from the ideal same plane in order to apply interpolation of normal vectors.

The default value for `alpha` is zero (no interpolation).

(see `NormalsForGeomArrays.java`)

Normal vectors for surfaces



Interpolated and noninterpolated normal vectors