CGC Summary

Smonman

February 1, 2024

Contents

1 Graphical Data Processing

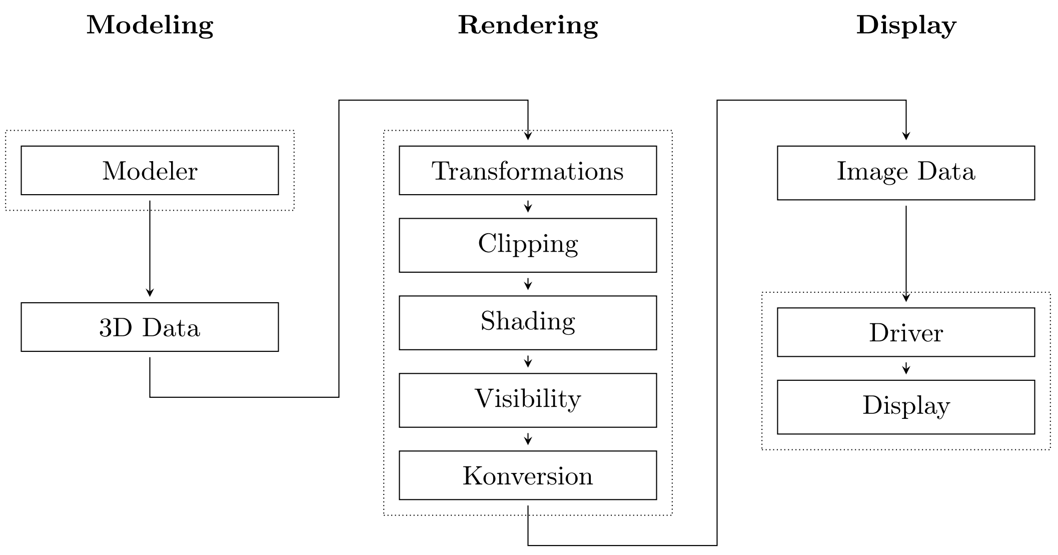


Figure 1: Graphical data processing

1.1 Modeling

- mesh creation
- dataset creation
  - volumetric data
  - particles
  - procedural modeling
  - fractals
- implicit object creation

1.2 Rendering

1.3 Display

- data in the image buffer

1.4 Scene Graph

- often found in more abstract graphics engines
- hierarchical structure
- describes structure of the scene
- easy to design a manipulate complex scenes

1.5 GUI Programming

- Qt
- GTK
- wxWidget
- WPF (Windows only)
- Java GUI Toolkit
- low level system I/O utilities

  - SDL
  - FreeGLUT
  - GLFW

1.6 Render Engines

- often used for high fidelity images
- RenderMan for Pixar

1.7 Game Engines

- realtime rendering
- usually supports additional functionality like
  - collision detection
- Unity3D
- Unreal Engine
- CryEngine
- Source 2 Engine
- Ogre
- ...

2 Transformations

Scaling	$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$
Rotation	$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$
X-Mirroring	$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$
Translation	Is not a linear transformation and needs homogeneous coordinates.

2.1 Homogeneous Coordinates

- Instead of $\begin{pmatrix} x \\ y \end{pmatrix}$ use $\begin{pmatrix} x_h \\ y_h \\ h \end{pmatrix}$ with $\begin{matrix} x = \frac{x_h}{h} \\ y = \frac{y_h}{h} \end{matrix}$.
- Very often $h = 1$.

With homogeneous coordinates, *all* transformations can be formulated in matrix form!

2.2 Homogeneous Transformations

Translation $$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad P' = T(t_x, t_y) \cdot P$$

Rotation $$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad P' = R(\theta) \cdot P$$

Scaling $$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad P' = S(s_x, s_y) \cdot P$$

The inverse of a rotation matrix can be achieved by inverting the angle:

$$R_x^{-1}(\theta) = R_x(-\theta) = R_x^T(\theta)$$

2.3 Composite Transformations

n transformations are applied after each other on a point P, these transformations are represented by matrices $M_1, M_2, \dots, M_n$.

For example:

$$\begin{aligned} P' &= M_1 \cdot P \\ P'' &= M_2 \cdot P \\ &\dots \\ P^{(n)} &= M_n \cdot P^{n-1} \end{aligned}$$

Shorter:

$$P^{(n)} = (M_n \cdot \dots (M_2 \cdot (M_1 \cdot P)) \dots)$$

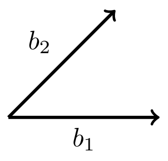
Matrix multiplications are *associative* but **not** *commutative*. Therefore it holds

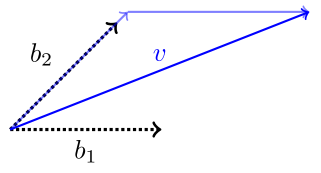
$$P^{(n)} = (M_n \cdot \dots (M_2 \cdot (M_1 \cdot P)) \dots) = (M_n \cdot \dots \cdot M_2 \cdot M_1) \cdot P$$

2.4 Linear Algebra

Let $B = \{ b_1, b_2 \}$ with b_1 and b_2 beeing linarly independent $\implies x \cdot b_1 + y \cdot b_2 = 0$. Then any vector v can be represented as the linear combination $v = x \cdot b_1 + y \cdot b_2$. x and y are called *coordinates*.

- the linearly independent vectors are called the *basis vectors*
- the amount of linearly independent vectors is called the *dimension*
- the factors are called *coordinates*
- coordinates are always dependent on the corresponding basis





The vector v can then be specified relative to a basis B like this: $\varphi_B(v) = \begin{pmatrix} 1.2 \\ 1.1 \end{pmatrix}$.

By convention the *unit basis* (Einheitsbasis) $E = \{ e_1, e_2 \}$ creates a reference vector space with:

$$e_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$e_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Every other vectors can be represented in reference to this unit basis. For example the basis vectors b_1 and b_2 from the example above might be:

$$\varphi_E(b_1) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$
$$\varphi_E(b_2) = \begin{pmatrix} 0.707 \\ 0.707 \end{pmatrix}$$

How can then the vector v, from the example above, be represented to the unit basis E?

$$\begin{aligned} \varphi_E(v) &= \varphi_E(x \cdot b_1 + y \cdot b_2) \\ &= \varphi_E(x \cdot b_1) + \varphi_E(y \cdot b_2) \\ &= x \cdot \varphi_E(b_1) + y \cdot \varphi_E(b_2) \end{aligned}$$

2.4.1 Transformations between Coordinate Systems

$$\begin{aligned} T_{CB} \cdot \varphi_C(v) &= \varphi_B(v) \\ \varphi_C(v) &= T_{CB}^{-1} \cdot \varphi_B(v) \\ &= T_{BC} \cdot \varphi_B(v) \end{aligned}$$

2.4.2 Orthogonality

The *dot product* (Skalaprodukt, Innere Produkt) is defined as $\langle v_1, v_2 \rangle := v_1^T \cdot v_2$. A more visual description might look as follows:

$$\left\langle \begin{pmatrix} a \\ b \end{pmatrix}, \begin{pmatrix} c \\ d \end{pmatrix} \right\rangle = (a, b) \cdot \begin{pmatrix} c \\ d \end{pmatrix}$$

Two vectors v_1 and v_2 are orthogonal if $\langle v_1, v_2 \rangle = 0$.

In an orthogonal coordinate system, the transformation matrix, which is used to transform a vector between two coordinate systems, has a special property: The inverse is equivalent to the transposition.

2.4.3 Row vs. Column Vectors

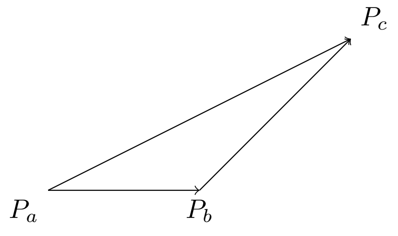
Row vectors can be interpreted as plane equations. In an orthogonal coordinate system a matrix multiplication with a vector can be interpreted as inserting the point into the plane equation. This yields the distance to the planes.

2.4.4 Affine Space

An *affine space* is defined as: $\langle A, V, \text{op} \rangle$, with A being a set of points, V being a vector space, and op being an operation $A \times V \mapsto A$, e.g. addition (+).

There are two axioms that need to hold for affine spaces:

1. $$P_a, P_b \in A \implies \exists v \in V : P_a + v = P_b$$
2. For three points a, b, c it holds:
$$(b - a) + (c - b) = (c - a)$$



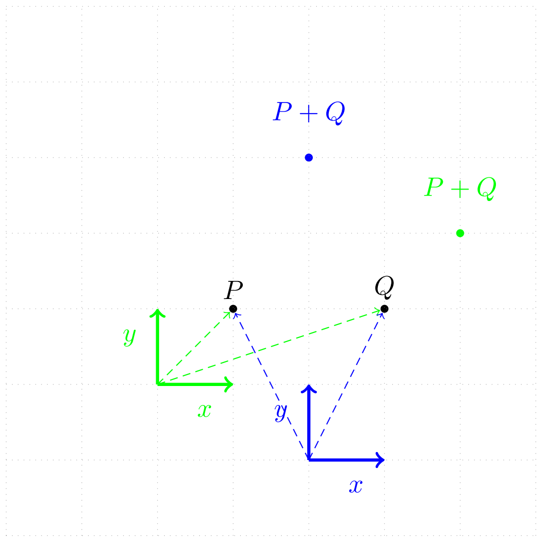
It can be said, that affine spaces are spaces where points exist. In contrast to linear spaces, where only directions exists.

Two vectors can define a linear space, but two vectors and a point define an affine space.

2.4.5 Affine Combination

The operation op is not well-defined for points: $a + b = ?, \quad a, b \in A$. This is because points only exist in relation to an origin. The coordinates of points can be calculated by subtracting the point from the origin. The result is a vector pointing from the origin to the point (Ortsvektor).

But a linear combination of points e.g. $a + t \cdot (b - a)$ with t being a scalar, is well-defined. Essentially this describes the operation where one point is chosen as an origin, e.g. a and a direction is chosen, with t being the distance along this direction. The same principle applies to higher dimension, only with more directions. In $\mathbb{R}^3$ this is a plane.



This is called an *affine combination*:

$$a + t \cdot (b - a)$$

$$\begin{aligned} a + t \cdot (b - a) &= (1 - t) \cdot a + t \cdot b \\ &= \alpha \cdot a + \beta \cdot b, \quad \alpha + \beta = 1 \end{aligned}$$

- the form $(1 - t) \cdot a + t \cdot b$ is also called *interpolation*
- the form $\alpha \cdot a + \beta \cdot b$ describes the affine combination as a linear combination. But the factors α and β must add up to 1.

2.4.6 Convex Combination

Is a special case of the affine combination. An affine combination is called a *convex combination* if the following holds:

$$\alpha, \beta > 0$$

which also implies

$$0 \leq t \leq 1$$

This means, that the result will be *between* a and b.

A convex combination in $\mathbb{R}$ are the *barycentric coordinates* between the points a and b.

2.5 Affine Transformations

Affine transformations are linear transformations combined with a translation. An affine transformation can be written like this:

$$\begin{aligned} x' &= l_{xx}x + l_{xy}y + t_x \\ y' &= l_{yx}x + l_{yy}y + t_y \end{aligned}$$

where l denotes the linear transformation:

$$l = \begin{pmatrix} l_{xx} & l_{xy} \\ l_{yx} & l_{yy} \end{pmatrix}$$

and t the translation:

$$t = \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

2.5.1 Properties of Affine Transformations

- affine transformations are *collinear* $\implies$ points on a line stay on a line
- parallel lines stay parallel
- ratios of distances along a line are preserved
- finite points stay finite points
- any affine transformation is a combination of
  - translation
  - rotation
  - scaling
  - reflection
  - shear
- if only translation, rotation and reflections are used then the following properties are preserved
  - angle
  - length

2.6 Transformation Classes

There exist different transformation classes:

- *Rigid* (Euclidean)
  - has a degree of freedom of 6 (translation + rotation)
  - transformations in this class are *isometric*, which means that distances do not change
- *Similarity*
  - has a degree of freedom of 7 (rigid + scale)
  - transformations in this class preserve angles and ratios, but not distances
- *Linear*
  - has a degree of freedom of 9
  - with a transformation matrix like

$$\begin{pmatrix} r_1 & r_4 & r_7 \\ r_2 & r_5 & r_8 \\ r_3 & r_6 & r_9 \end{pmatrix}$$

- *Affine*
  - has a degree of freedom of 12
  - like the linear transformation plus translation
  - with a transformation matrix like

$$\begin{pmatrix} r_1 & r_4 & r_7 & t_x \\ r_2 & r_5 & r_8 & t_y \\ r_3 & r_6 & r_9 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- *Projective*
  - has a degree of freedom of 15
  - with a transformation matrix like

$$\begin{pmatrix} r_1 & r_4 & r_7 & t_x \\ r_2 & r_5 & r_8 & t_y \\ r_3 & r_6 & r_9 & t_z \\ p_1 & p_2 & p_3 & 1 \end{pmatrix}$$

Class		Affine Space (4 × 4)		Linear Space (3 × 3)	
Name	DoF	Transformation	DoF	Transformation	DoF
Rigid (Euclidean)	6	Translation	3	-	-
		Rotation	3	Orthogonal	3
Similarity	7	Uniform Scale	1	Uniform Scale	1
		Affine Transformation	12	Linear Transformation	9
		Projective Transformation	15	-	-

Table 1: Transformation classes summary

2.7 Constructing Coordinate Systems

- *incrementally* over time
  - for moving in a game
  - e.g. M as the world matrix, T as a translation and R_y as a rotation around the y-axis

$$M' = M \cdot T$$
$$M'' = M' \cdot R_y$$

- *hierarchical*
  - e.g. move shoe model based on the foot position
  - in hierarchical structures, like bones of an object, where every bone has a separate coordinate system
  - chained matrices, that express different local coordinate systems

$$T_1 \cdot R_1 \cdot T_2 \cdot R_2$$

- *specify basis vectors*
  - e.g. used in `LookAt`

2.8 Summary

2.8.1 Linear Transformations

- rotation
- scaling

2.8.2 Affine Transformations

- translation
- rotation
- scaling
- shear
- preserve:
  - collinearity
  - parallel lines
  - ratios along lines
  - finite points

2.8.3 Transformation Classes

- rigid (6)
- similarity (7)
- linear (9)

- affine (12)
- projective (15)

2.8.4 Constructing Coordinate Systems

- incrementally
- hierarchical
- specifying basis vectors

3 Viewing

3.1 Projection Transformation (Orthographic)

Transformation from an orthographic view volume in camera space into a unit cube in clip space. We have these assumptions:

- scene in box $(L, R) \times (B, T) \times (F, N)$ with

  L ... left
  R ... right
  B ... bottom
  T ... top
  F ... far
  N ... near

- orthographic camera looking in $-z$ direction

We want to transform these points accordingly:

$$\begin{pmatrix} L \\ B \\ F \end{pmatrix} \to \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix}$$
$$\begin{pmatrix} R \\ T \\ N \end{pmatrix} \to \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

This can be done with this matrix:

$$M_{\text{orth}} = \begin{pmatrix} \frac{2}{R-L} & 0 & 0 & -\frac{R+L}{R-L} \\ 0 & \frac{2}{T-B} & 0 & -\frac{T+B}{T-B} \\ 0 & 0 & \frac{2}{N-F} & -\frac{N+F}{N-F} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3.2 Camera

A camera is defined by the following properties:

- position
- direction
- orientation around the direction
- zoom

The *view matrix* is then constructed from the first three properties: (i) position, (ii) direction and (iii) orientation.

The *projection matrix* is constructed from the zoom.

3.2.1 Camera Transformation

To create the camera coordinate system 3 things are needed:

e . . . eye position
g . . . gaze direction
t . . . view-up vector

The basis vectors of the camera's coordinate system w, u and v can then be computed in the following steps:

$$\begin{aligned} w &= -\frac{g}{|g|} \\ u &= \frac{t \times w}{\|r \times w\|} \\ v &= w \times u \end{aligned}$$

This process is used in the `LookAt` function.

The camera matrix M_{cam} can then be computed as follows:

$$M_{\text{cam}} = \begin{pmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_w \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The red section is describing the rotational part of the matrix R. The $\mathbf{e}$ is sort of a translation vector. To get a point P from camera space into world space the following equation can be used:

$$P_w = \mathbf{e} + R \cdot P_v$$

Conversely, to get a point P from world space into camera space this is needed:

$$\begin{aligned} P_w &= \mathbf{e} + R \cdot P_v \\ P_w - \mathbf{e} &= R \cdot P_v \\ P_v &= R^{-1} \cdot (P_w - \mathbf{e}) \\ &= R^{-1} \cdot P_w - R^{-1} \cdot \mathbf{e} \end{aligned}$$

The two matrices above, to the right of the equal sign, can now be explained. They represent the term $R^{-1} \cdot (P_w - \mathbf{e})$, where the first matrix represents the R^{-1} part. Specifically the blue section of the matrix is equivalent to R^T. The term $P_w - \mathbf{e}$ represents a translation of $-\mathbf{e}$. The whole second matrix is representing a translation of $-\mathbf{e}$.

It is important to note, that $R^T \equiv R^{-1}$ *iff* the basis vectors u, v and w are orthogonal.

3.2.2 Camera + Projection + Viewport

$$\begin{pmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ z \\ 1 \end{pmatrix} = M_{\text{viewport}} \cdot M_{\text{orth}} \cdot M_{\text{cam}} \cdot P_w$$

3.3 Perspective Projection

- *parallel projection*
  - preserves affine transformations
  - relative proportions
  - parallel features
- *perspective projection*
  - center of projection
  - realistic views

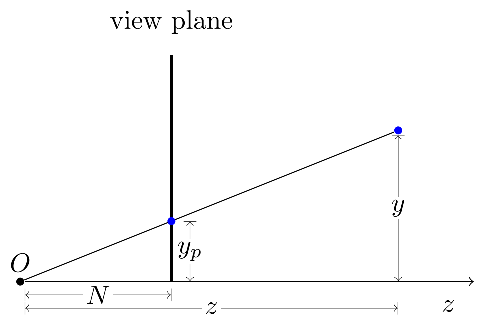


Figure 3: Perspective Projection

  - lines parallel to the viewing plane are preserved
  - lines parallel to basis vectors have a vanishing point

The projection matrix P is defined as:

$$P = \begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & N+F & -F \cdot N \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

The red section is needed to preserve the z value for the *Z-Buffer*. If the perspective projection were just a projection the z value would be lost, as all the points would be projected onto the viewing plane, which has a constant distance to the eye point, therefore also fixating all the z values. This is because the camera looks into the negative z axis. This can be demonstrated using this matrix:

$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & N & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \cdot N \\ y \cdot N \\ z \cdot N \\ z \end{pmatrix}$$

but after *homogenization* this happens:

$$\begin{pmatrix} x \cdot N \\ y \cdot N \\ z \cdot N \\ z \end{pmatrix} \div z = \begin{pmatrix} \frac{x \cdot N}{z} \\ \frac{y \cdot N}{z} \\ \frac{z \cdot N}{z} \\ \frac{z}{z} \end{pmatrix} = \begin{pmatrix} \frac{x \cdot N}{z} \\ \frac{y \cdot N}{z} \\ N \\ 1 \end{pmatrix}$$

Instead of the z values being projected onto the viewing plane, they should be mapped between the near and far plane. The terms in the red section do that. It is important to note, that the new z value, that ends up in the Z-Buffer is not the actual z coordinate but rather a term of the form $a + \frac{b}{z}$. Because of this, the z value does not map linearly between the near and far plane. In the distance, close z values map to the same Z-Buffer-value. If that happens, *z-fighting* might occur.

The final perspective projection matrix then amounts to $M_{\text{per}} = M_{\text{orth}} \cdot P$:

$$M_{\text{per}} = \begin{pmatrix} \frac{2 \cdot N}{R-L} & 0 & \frac{L+R}{L-R} & 0 \\ 0 & \frac{2 \cdot N}{T-B} & \frac{B+T}{B-T} & 0 \\ 0 & 0 & \frac{F+N}{N-F} & \frac{2 \cdot F \cdot N}{F-N} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

3.4 Summary

3.4.1 Projections

- parallel projection preserve
- perspective projection

4 Data Structures

4.1 3D Data Structures

4.1.1 Point Cloud

Objects are represented only as a set of points, without any correlation between these points.

An exact representation is possible if there are ≤ 1 points per pixel.

Transformations

- multiply the points in the point list with linear transformation matrices

Combinations

- objects can be combined by appending the point lists to each other

Rendering

- project and draw the points onto the image plane

Advantages

- fast rendering
- exact representation possible
- exact rendering possible
- fast transformations

Disadvantages

- many points (exact representation of curved objects)
- high memory consumption
- limited combination operations

Examples

- Surfels (Surface Elements)
- QSplat

4.1.2 Wire-Frame Model

The object is simplified to 3D lines. No face information included.

Transformations

- multiply the points in the point list with linear transformation matrices

Combinations

- objects can be combined by appending the point and edge lists to each other

Rendering

- projection of all points onto image plane and drawing of edges between

Advantages

- quick rendering
- easy and quick transformations
- generation of models via digitization

Disadvantages

- inexact (no surfaces, no occlusion)
- restricted combination possibilities
- curves are approximated by straight lines

4.1.3 Boundary Representation (B-Rep)

A *Boundary Representation* is a class of data structures. It is the most common one, and only specifies the boundary, the surface, of the object.

B-Reps have to be *water tight*. This is an important property that many algorithms require.

Additionally, a B-Rep is said to be a *manifold*, iff every edge has at most two faces.

Transformations

- all points are transformed like the wire-frame model
- additionally, surface equations or normal vectors can be transformed

Combinations

- see section 4.4

Rendering

- hidden surface or hidden line algorithms can be used because the surfaces of the objects are known, and the visibility can be calculated (backface-culling)

Advantages

- general representation
- generation of models via digitization
- transformations are easy and fast

Disadvantages

- high memory requirement
- combinations are relatively costly
- curved objects must be approximated

Examples

- Vertex, Edge and Face List
  - Disadvantages
    - it is not easy to find out for a given vertex, what faces contain this vertex
- Winged Edge Data Structure
  - alternative for normal hierarchical B-Rep
  - central element is the edge
- Half Edge Data Structure
  - same as the Winged Edge Data Structure but only save one face per edge
  - allows the representation of manifold borders

4.2 Partitioning of Object Surfaces

- necessary to approximate curved surfaces
- surfaces that *can* be parameterized:
  - partitioning of parameter space, one patch for every 2D parameter interval

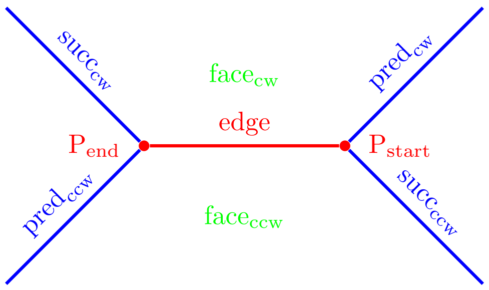


Figure 4: Winged Edge

  - free form surfaces
  - quadrics
  - superquadrics
- surfaces that *cannot* be parameterized:
  - use tessellation (subdivision)
  - implicit surfaces
  - *bent* polygons

4.3 Tessellation

- divide polygon in smaller polygons until the approximation is exact enough
- normal vector criterion as termination condition:

$$N_1 \cdot N_2 \leq 1 - \epsilon$$

  If the normal vector is *too different*, subdivide

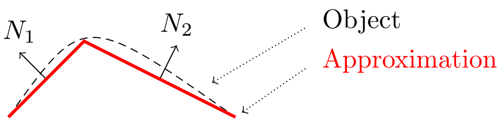


Figure 5: Tessellation

4.4 Combinations of B-Reps

1. Split the polygons of object A at the intersections with the polygons of object B
2. Split the polygons of object B at the intersections with the polygons of object A
3. Classify all polygons of A as (i) in B, (ii) outside B or (iii) on the surface of B
4. Classify all polygons of B in the same way
5. Remove the redundant polygons of A and B according to the operator and combine the remaining polygons of A and B

- every polygon has a *box enclosure* $\rightarrow$ simple test if polygons can intersect
- use only *convex* polygons and produce only convex polygons as results $\rightarrow$ simple intersection tests

4.4.1 Point to Polygon Test

How to find out, if a point is inside the polygon or outside?

- A ray is traced in the direction of the normal vector of the polygon to be classified:
  - ray hits no polygon of $B \implies$ *outside* B
  - first polygon of B hit from front $\implies$ *outside* B

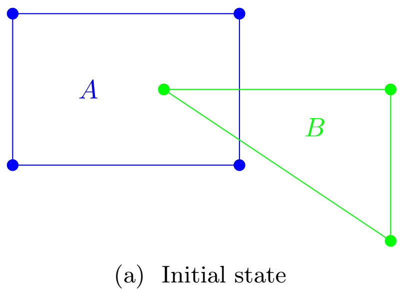


Figure 6: Combinations of B-Reps

- first polygon of B hit from back $\Longrightarrow$ *in* B

- Improvement
  - points of A, which lie on the surface of B, are marked as border points during the dividing process (and vice versa) $\rightarrow$ only a few polygons have to be classified with the complex method
  - the classification can only change at border points

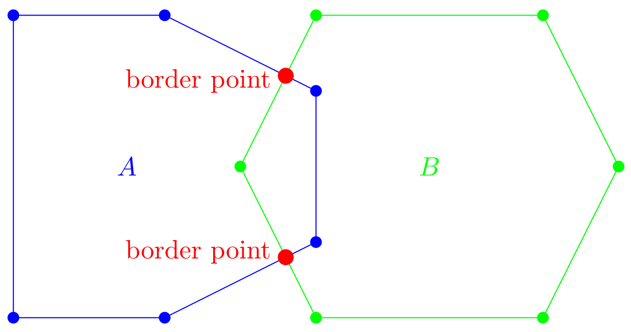


Figure 7: 2D representation of the Point to Polygon Test improvement

What Polygons to Remove? Polygons can then be removed according to tables:

operation	in B	outside B	on B (*coplanar*)	
			N equal	**N** different
$A \cup B$	yes	no	no	yes
$A \cap B$	no	yes	no	yes
$A - B$	yes	no	yes	no

Table 2: Table for removing polygons of A

operation	in A	outside A	on A (*coplanar*)	
			N equal	**N** different
$A \cup B$	yes	no	yes	yes
$A \cap B$	no	yes	yes	yes
$A - B$	no	yes	yes	yes

Table 3: Table for removing polygons of B

Requirements

- no open objects (everything must be *watertight*)
- only convex polygons
- no double points

- add links in the vertex list between neighboring points with equal classification

4.4.2 Binary Space Partitioning Tree (BSP)

- special B-Rep for quick rendering with visibility
- especially for static scenes
- is used for space partitioning in ray tracing

The BSP tree partitions the space into two spaces using a *carrier polygon* (Trägerpolygon), which defines a base plane to cut along. The polygons are then classified to be either *in front* of the base plane, or *behind* the base plane.

The tree is then constructed based on this classification:

- *left subtree* of a node contains only polygons that are *in front* of the base plane
- *right subtree* of a node contains only polygons that are *behind* the base plane

Polygons that lie in both halves are divided by the base plane into two parts, and classified.

Transformations

- points, plane equations and normal vectors have to be transformed

Combinations

- either perform combination with B-Rep and then generate a new BSP tree, or
- combine BSP trees directly (faster)
- see section 4.4.2

Rendering

- BSP trees are very good for fast rendering
- see section 4.4.2

Advantages

- fast rendering
- fast transformation
- combinations faster than for B-Reps
- general representation, can represent everything that a B-Rep can also represent
- generation of models via digitization
- tree structure provides fast search

Disadvantages

- curved objects must be approximated
- only convex polygons
- high memory cost

Generation of a BSP Tree

1. find the polygon whose plane *intersects the fewest* other polygons and cut these in two
2. divide the polygon list in *two sets*:
    - in front of the base plane
    - behind the base plane
3. The polygon found in step one is the *root* of the BSP tree, the left and the right subtrees can be generated *recursively*

- for *convex* objects the BSP Tree is a linear (linked) list

BSP Trees as Solids

- *left empty* trees represent *outside* spaces
- *right empty* trees represent *inside* volumes

Painters algorithm

```
if eye is in front of $a$ then
begin
draw all polygons of $A-$;
draw $a$;
draw all polygons of $A+$
end else begin
draw all polygons of $A+$;
draw $a$;
draw all polygons of $A-$;
end
```

Combinations of BSP Trees

- algorithm for $A \operatorname{op} B = C$
- $A \cup B$ homogeneous (in or out) → simple rules
- else
  1. divide the root polygon a of A at object B in a_{in} and a_{out}
  2. root node c of C:
  $$c = \begin{cases} a_{\text{in}}, & \text{op} = \cap \text{ (and)} \\ a_{\text{out}}, & \text{otherwise} \end{cases}$$
  3. divide B at plane of a into B_{in} and B_{out}
  4. recursively evaluation of the subtrees:
  $$C_{\text{left}} = A_{\text{out}} \operatorname{op} B_{\text{out}}$$
  $$C_{\text{right}} = A_{\text{in}} \operatorname{op} B_{\text{in}}$$

Simple BSP Node Combination Rules These rules apply when one of the two nodes are homogeneous. The values *in* and *out* are also called *full* and *empty.*

Operator	A	B	$A \operatorname{op} B$
$\cup$	inhomogeneous	in	in
	inhomogeneous	out	A
	in	inhomogeneous	in
	out	inhomogeneous	B
$\cap$	inhomogeneous	in	A
	inhomogeneous	out	out
	in	inhomogeneous	B
	out	inhomogeneous	out
$-$	inhomogeneous	in	out
	inhomogeneous	out	A
	in	inhomogeneous	$-B$
	out	inhomogeneous	out

Table 4: Node combination rules

4.4.3 kD Tree

- special case of the BSP tree
- only axis-aligned paritioning planes → specified by one value
- partitioning direction specified either implicitly (pre-defined order) or explicitly
- objects may lie in multiple sections of the tree at once

4.4.4 Octree

- used to represent solid volumetric objects
- each node is subdivided in 8 subspaces
- each subspaces is either *empty*, *full* or further divided
- the subdivision stops when an object can be represented accurately enough

Transformations

- hard to implement
- rotations of 90° easy

Combinations

- can easily be done by logical operations
- both octrees must be adapted to each other to have the same depth in each subspace

Rendering

- the octree is rendered depending on the view direction starting with the subspace farthest away from the viewer

Advantages

- combinations are easy
- spartial search is fast due to the tree structure
- rendering is fast

Disadvantages

- high storage consumption for approximated objects
- transformations are not trivial
- general objects cannot be represented exactly

4.4.5 Extended Octree

- additional node type
  - *face nodes*: contain a surface
  - *ege nodes*: contain an edge
  - *vertex node*: contain a corner point

Generation

1. generate B-Rep
2. divide point and surface list at the subdivision planes into 8 sets
3. For each octant:
   - point and surface list empty $\implies$ *full* or *empty*
   - only one vertex $\implies$ *vertex* node
   - only one surface $\implies$ *face* node

    - only two surfaces $\implies$ *edge* node
    - else: subdivide recursively

Octrees as Spatial Directory

- octree as search structure for objects in other representations
- octree of low depth is sufficient

4.4.6 Constructive Solid Geometry Tree (CSG)

- a CSG tree consists of
  - simpe primitives
  - transformations
  - logical operations
- useful to describe complex objects with a small number of primitives

Transformations

- an object is transformed by adding the transformation to the transformation of each primitive

Combinations

- two objects are simply combined by adding them as children in a new tree

Rendering

- needs to be converted into a B-Rep or it is rendered with raytracing

Advantages

- minimal storage consumption
- combinations are simple
- transformations are simple
- objects can be represented exactly
- tree structure (fast search)

Disadvantages

- cannot be rendered directly
- slow rendering
- model generation cannot be done through digitization of real objects

4.4.7 Bintree

- 3D Tree
- subdivision order $xyzxyz\dots$
- choose separation plane for optimized (irregular) subdivision
- fewer nodes than octree

4.4.8 Grid

- regular subdivision
- directly addresses cells
- simple neighborhood finding $O(1)$
- Problem
  - too few / many cells

  - → hierarchical grid

4.5 Summary

Name	Examples	Transformations	Combinations	Rendering	Advantages	Disadvantages
Point Cloud	• Surface Elements (Surfels) • QSplat	• multiply the points in the point list with linear transformation matrices	• objecs can be combined by appending the point lists to each other	• project and draw the points onto an image plane	• fast rendering • fast transformations • exact representation possible • exact rendering possible	• many points (exact representation of curved objects) • high memory consumption • limited combination possibilities
Wire Frame Model	-	• multiply the points in the point list with linear transformation matrices	• objects can be combined by appending the point and edge list to each other	• projection of all points onto image plane • drawing of edges inbetween	• fast rendering • fast transformations • easy transformations • generation of models via digitization	• inexact • no surfaces • no occlusion • limited combination possibilities • curves are approximated
Boundary Representation (B-Rep)	• vertex, edge and face list • Winged Edge • Half Winged Edge	• multiply the points in the point list with linear transformation matrices • transform surface equations • transform normal vectors	• see 4.4	• occlusion possible	• fast transformations • easy transformations • general representation • generation of models via digitization	• high memory consumption • combinations are costly • curves are approximated
Binary Space Partitioning Tree (BSP)	-	• points have to be transformed • plane equations have to be transformed • normal vectors have to be transformed	• perform combination with B-Rep and generate new tree • combine BSP directly (see 4.4.2)	• fast render (see 4.4.2)	• fast rendering • fast transformation • faster combination than B-Reps • general representation • generation of models via digitization • tree structure • fast search	• high memory consumption • curves are approximated • only convex polygones

Name	Examples	Transformations	Combinations	Rendering	Advantages	Disadvantages
Octree	-	• only rotations of 90° easy	• easy logical operations • both octrees must have same depth at each subspace	• from farthest to nearest substpace	• fast rendering • easy combinations • tree structure • fast search	• high memory consumption • hard transformations • no exact representation
Constructive Solid Geometry Tree (CSG)	-	• transformation to each primitive	• add as children to tree	• convert to B-Rep • raytracing	• low memory consumption • easy combinations • easy transformations • exact representation • tree structure • fast search	• slow rendering • cannot be rendered directly • no generation of models via digitization

Table 5: Summary of datastructures

5 Modeling

5.1 Curves

There are three mathematical representations for curves:

- *explicit*
  - represented with a function
  - every point in the domain maps to exactly one point in the target
  - e.g. cannot represent a circle
- *implicit*
  - represented with an equation
  - e.g. circle with $r^2 = x^2 + y^2$
  - loose one degree of freedom, e.g. for the circle we have three variables, but only two degrees of freedom to satisfy the equation
- *parametric*
  - multiple functions with parameters (k amount)

Because we loose one degree of freedom in the implicit form it holds: If you want to represent a k dimensional sub-space (Unterraum) in an n dimensional space, you either need: (i) $n - k$ number of equations in implicit form, or (ii) a parametric representation with k parameters.

5.1.1 Parametric Curves

Let c be a curve

$$c(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}$$

where $t \in \{ a, b \} \subset \mathbb{R}$. The functions $x(t)$, $y(t)$ and $z(t)$ are differentiable in t.

- the tangent vector $c_t(t)$ is the first derivate of the function $c(t)$:

$$c_t(t) = \frac{d}{dt}c(t) = \begin{pmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \\ \frac{dz}{dt} \end{pmatrix}$$

  The tangent vector is dependent on the parameterization of the curve. Different parameterizations might yield different lengths of tangent vectors.
- a point p on a curve at t is called *regular*, if the tangent $c_t(t) \neq 0$
- a curve c is called *regular* iff there exists a parameterization, such that the derivate is non-null for every t. That means if every point on the curve is regular
- the arc length t' is defined as follows:

$$t' = \int_0 |c_t(t)| dt$$

5.1.2 Parametric Surfaces

Let s be a surface:

$$\mathbf{s}(u, v) = \begin{pmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{pmatrix}$$

where $(u, v) \in \{ a, b \} \times \{ c, d \} \subset \mathbb{R}^2$. The functions $x(u, v)$, $y(u, v)$ and $z(u, v)$ are differentiable in u and v.

It is clear to see, that for a 2D object, a surface, two parameters are needed. In this case they are callen u and v. To display this surface, a 3D space is needed, therefore three functions are needed: $x(u, v)$, $y(u, v)$ and $z(u, v)$.

- the tangent plane is defined as:

$$\mathbf{t}_0(l, m) = \mathbf{s}(u_0, v_0) + l \cdot \mathbf{s}_u(u_0, v_0) + m \cdot \mathbf{s}_v(u_0, v_0)$$

$\mathbf{s}_u$ and $\mathbf{s}_v$ are partial derivatives of the surface. They have the form:

$$\mathbf{s}_i = \begin{pmatrix} \frac{\partial x}{\partial i} \\ \frac{\partial y}{\partial i} \\ \frac{\partial z}{\partial i} \end{pmatrix}$$

The tangent plane is described by a point on the surface, $\mathbf{s}(u_0, v_0)$, and a linear combinations of the tangent vectors, one for each parameter.

- the normal vector $\mathbf{n}$ is defined as follows:

$$\mathbf{n}(u_0, v_0) = \mathbf{s}_u(u_0, v_0) \times \mathbf{s}_v(u_0, v_0)$$

- a point p is *regular* in s iff the normal vector $\mathbf{n}$ at that point is not equal to the zero vector:

$$\mathbf{s}(u_0, v_0) \equiv \mathbf{n}(\mathbf{u_0}, \mathbf{v_0}) \neq \mathbf{0}$$

- a surface s is *regular* iff s can be parameterized in a way, such that all surface points are regular

5.2 Bézier Curves

5.2.1 Casteljau Algorithm

Given $n+1$ points $\mathbf{b}_0, \dots, \mathbf{b}_n \in \mathbb{E}^3$ and an arbitrary $t \in \mathbb{R}$. Set $\mathbf{b}_i^r := (1-t) \cdot \mathbf{b}_i^{r-1} + t \cdot \mathbf{b}_{i+1}^{r-1}$ with $\mathbf{b}_i^0 := \mathbf{b}_i$.

Then $\mathbf{b}_0^n$ is a curve point on the corresponding Bézier curve of degree n.

The points $\mathbf{b}_0^{n-1}$ and $\mathbf{b}_1^{n-1}$ determine the tangent line of the curve at point $\mathbf{b}_0^n$.

$$\begin{tikzcd}
\mathbf{b}_0^0 \arrow[dr, "1-t"] & & & \\
& \mathbf{b}_0^1 & & \\
\mathbf{b}_1^0 \arrow[ur, "t"] & & \mathbf{b}_0^2 & \\
& \mathbf{b}_1^1 & & \mathbf{b}_0^3 \\
\mathbf{b}_2^0 & & \mathbf{b}_1^2 & \\
& \mathbf{b}_2^1 & & \\
\mathbf{b}_3^0 & & &
\end{tikzcd}$$

Figure 8: De Casteljau Scheme

5.2.2 Bernstein Polynomials

From the Casteljau Algorithm, the Bernstein polynomials can be derived. The Bernstein polynomials are:

$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i$$

The Bézier curve can then be represented with the Bernstein polynomials, where b_i are the *control points* $b_i, i = 0, \dots, n$.

$$b(t) = \sum_{i=0}^{n} b_i B_i^n(t)$$

5.2.3 Properties of Bézier Curves

- affine invariance
  - affine transformations can be applied to the control points or to the points on the curve with the same outcome
- convex hull property

    - regardless of the number of control points the curve will never cross the convex hull spanned by the controll points
- endpoint interpolation
    - the first and last point will be interpolated. That means the curve starts in the first point and end in the last one.
- linear precision
    - if the control points lie on a line, the curve will also be a line
- variation diminishing property
    - any line passing through the curve will have at most as many intersections as intersections between the line and the convex hull

Disadvantages

- only pseudo local control
- high calculation cost on high degree

5.2.4 Degree Elevation

Create the same Bézier curve, but with one more control point. The degree elevation is cutting the corners of the original control polygon. After multiple iterations the control polygon approaches the actual curve.

5.2.5 Subdivision

Reduce the degree of a Bézier curve. This can be done by splitting the curve, according to auxillary points created by the Casteljau algorithm. This can be used to approximate a curve.

5.2.6 Evaluation

- the Casteljau algorithm is numerically stable, but inefficient for evaluation
- a *Horner Scheme* like evaluation is more efficient:

$$b(t) = \left(\dots\left(\left(\binom{n}{0}s\mathbf{b_0} + \binom{n}{1}t\mathbf{b}_1\right)\cdot s + \binom{n}{2}t^2\mathbf{b}_2\right)\cdot s + \dots\right)\cdot s + \binom{n}{n}t^n\mathbf{b}_n$$

  with $s = 1 - t$.
- repreated subdivision gives a good approximation of the curve

5.3 B-Spline Curves

- B-Spline curves of order k are piecewise polynomial curves of k pieces with degree $k-1$
- have a degree independent of the number of control points
- allow local control over the shape of the curve
- is a generic curve that is equivalent to Bézier curves when $n = k-1$

Given $n+1$ control points $\mathbf{d}_i \in \mathbb{E}^3, \quad i = 0, \dots, n$, and a knot vector $U = (u_0, \dots, u_{n+k})$. If a knot u_i appears j times in the the know vector U with $j > 1$, it is said that the know u_i has a *multiplicity* of j.

The curve is then defined as follows:

$$\mathbf{s}(u) = \sum_{i=0}^{n} \mathbf{d}_i N_i^k(u), \quad u \in [u_0, u_{n+k})$$

with the B-Spline basis functions $N_i^k(u)$ of order k.

5.3.1 Basis Functions

The order k is the number of control points that have an influence over a specific point on the curve. The basis function $N_i^k(u)$ is defined as follows:

- $k = 1$:
$$N_i^1(u) = \begin{cases} 1, & u \in [u_i, u_{i+1}) \\ 0, & \text{otherwise} \end{cases}$$
- $k > 1$:
$$n_i^k(u) = \frac{u - u_i}{u_{i+k-1} - u_i} N_i^{k-1}(u) + \frac{u_{i+k} - u}{u_{i+k} - u_{i+1}} N_{i+1}^{k-1}(u)$$

5.3.2 Properties of Basis Functions

- partition of unity: $\sum_{i=0}^{n} N_i^k(u) \equiv 1$
- positivity: $N_i^k(u) \leq 0$
- local support: $N_i^k(u) = 0$ if $u \notin [u_i, u_{i+k})$

5.3.3 Properties of B-Spline Curves

- affine invariance
- strong convex hull property
  - the curve always lies within the convex hull of it interval
- variation diminishing property
- local support
- knot points of multiplicity k are coincident with one of the control points
- a B-Spline curve of order k which has only two knots of multiplicity k is a Bézier curve $(n = k - 1)$
- if the multiplicity of the start end end points is k, the start and end points are interpolated

5.3.4 De Boor algorithm

Given $n + 1$ control points $\mathrm{d}_0, \ldots, \mathrm{d}_n \in \mathbb{E}^3$, a knot vector $U = (u_0, \ldots, u_{n+k})$ and an arbitrary $t \in [u_0, u_{n+k})$.

Set $\mathrm{d}_i^r := (1 - \alpha_i^r)\mathrm{d}_{i-1}^{r-1} + \alpha_i^r \mathrm{d}_i^{r-1}$ with $\mathrm{d}_i^0 := \mathrm{d}_i$ and $\alpha_i^r := \frac{t - u_i}{u_{i+k-r} - u_i}$.

Then d_m^{k-1} is the point for $\mathrm{x}(t_0), t_0 \in [t_m, t_{m+1}]$ on the corresponding B-Spline curve.

The points d_{m-1}^{k-2} and d_m^{k-2} determine the tangent line of the curve at a point b_m^{k-1}.

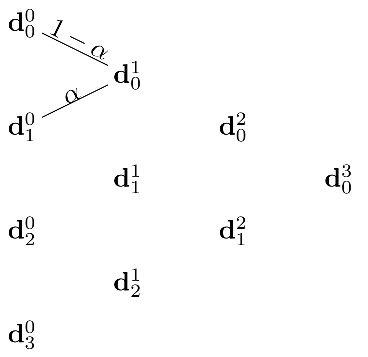


Figure 9: The de Boor scheme

5.3.5 Direct Evaluation of B-Spline Curves

- find the knot span $[u_i, u_{i+1})$ in which the parameter value t lies
- compute all non-zero basis functions
- multiply the values of the non-zero basis functions with the corresponding control points

5.3.6 Open B-Spline Curves

Start and end points are interpolated.

$$u_0 = \cdots = u_k < u_{k+1} < \cdots < u_n = \cdots = u_{n+k}$$

5.3.7 Closed B-Spline Curves

The parameter interval is cyclic. The *last* k knots are equivalent to the first k knots.

$$\mathbf{d}_{n+1} := \mathbf{d}_0, \dots, \mathbf{d}_{n+k} := \mathbf{d}_{k-1}$$

and

$$U = (u_0, \dots, u_{n+k}, \dots, u_{n+2k-2})$$

5.3.8 Uniform B-Spline Curves

The parameter interval is uniform.

$$U = (u_0, u_0 + d, \dots, u_0 + (n+k)d)$$

5.3.9 Algorithms for B-Spline Curves

- knot insertions
  - to increase flexibility
  - to compute derivatives
  - to split curves (subdivision)
  - to evaluate the curve (de Boor)
  - to approxiate the curve
- degree elevation
  - to adapt curve degrees

5.4 Rational Curves

$$\mathbf{c}(u) = \frac{1}{w(u)} \cdot \begin{pmatrix} x(u) \\ y(u) \\ z(u) \end{pmatrix}, \quad u \in I \subset \mathbb{R}$$

The homogeneous representation has the homogeneous value $w(u)$ as the first element, by definition:

$$\mathbf{c}_H(u) = \begin{pmatrix} w(u) \\ x(u) \\ y(u) \\ z(u) \end{pmatrix}, \quad u \in I \subset \mathbb{R}$$

All conic shapes are rational curves.

5.4.1 Rational Bézier Curves

A rational Bézier curve is defined as:

$$\mathbf{b}(u) = \frac{\sum_{i=0}^{n} w_i \mathbf{b}_i B_i^n(u)}{\sum_{i=0}^{n} w_i B_i^n(u)}, \quad u \in I \subset \mathbb{R}$$

The $w_i > 0, i = 0, \dots, n$ are called *weights.*

Homogeneous representation:

$$\mathbf{b}_H(u) = \sum_{i=0}^{n} \mathbf{b}_{H_i} B_i^n(n), \quad u \in I \subset \mathbb{R}$$

with the homogeneous Bézier points

$$\mathbf{b}_{H_i} = \begin{pmatrix} w_i \\ w_i \mathbf{b}_i \end{pmatrix}$$

Properties

- the same properties like polynomial curve
- projective invariance
  - projecting the control points first, and calculating the curve second yields the same result as projecting the points on the curve
- the weights are an additional design parameter

Algorithms

- all algorithms of polynomial Bézier curves can be applied without any change to the homogeneous representation of rational Bézier curves

5.4.2 NURBS

Are called *Non-Uniform Rational B-Splines*, and are rational B-Spline curves.

A NURBS curve with respect to the control points $d_i, i = 0, \dots, n$ and the knot vector $U = (u_0, \dots, u_{n+k})$ is defined as:

$$\mathbf{n}(u) = \frac{\sum_{i=0}^{n} w_i \mathbf{d}_i N_i^k(u)}{\sum_{i=0}^{n} w_i N_i^k(u)}, \quad u \in [u_0, u_{n+l}) \subset \mathbb{R}$$

The $w_i > 0, i = 0, \dots, n$ are called weights.

Homogeneous representation:

$$\mathbf{n}_H(u) = \sum_{i=0}^{n} \mathbf{n}_{H_i} N_i^k(u), \quad u \in [u_0, u_{n+l}) \subset \mathbb{R}$$

with the homogeneous B-Spline points

$$\mathbf{n}_{H_i} = \begin{pmatrix} w_i \\ w_i \mathbf{n}_i \end{pmatrix}$$

Properties

- the same properties like polynomial curve
- projective invariance
- changing the weights w_i affects only the interval $[u_i, u_{i+k})$

Algorithms

- all algorithms of B-Spline curves can be applied without any change to the homogeneous representation of the NURBS curves.

5.5 Tensor-Product Surfaces

Given a curve

$$\mathbf{f}(u) = \sum_{i=0}^{n} \mathbf{c}_i F_i(u), \quad u \in I \subset \mathbb{R}$$

moving the control points yields

$$\mathbf{c}_i(v) = \sum_{j=0}^{m} \mathbf{a}_{ij} G_j(v), \quad v \in J \subset \mathbb{R}$$

Combining both yields a tensor-product surface:

$$s(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{a}_{ij} F_i(u) G_j(v), \quad (u, v) \in I \times J \subset \mathbb{R}^2$$

5.5.1 Tensor Product

- let V, W be vector spaces, then $V \otimes W$ is a *tensor product*, and its elements are *tensors*
- V, W are one-dimensional curve spaces (Bézier, B-Spline)
- the product of curves is a bilinear function in the coefficients (control points)

5.5.2 Tensor-Product Bézier Surfaces

A tensor-product Bézier surface is given by:

$$\mathbf{b}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} \mathbf{b}_{ij} B_i^n(u) B_j^m(v), \quad (u,v) \in I \times J \subset \mathbb{R}^2$$

The Bézier points $\mathbf{b}_{ij}$ form the control net of the surface.

Properties

- analogue to that of Bézier curves

Algorithms

- apply algorithms for curves in two steps:
  1. apply to $\mathbf{b}_i(v) = \sum_{j=0}^{m} \mathbf{b}_{ij} B_j^m(v), \quad i = 0, \ldots, n$
  2. apply to $\mathbf{b}(u,v) = \sum_{i=0}^{n} \mathbf{b}_i(v) B_i^n(u)$

5.6 Tensor-Product B-Spline Surfaces

A tensor-product B-Spline surface with respect to the knot vectors

$$U = (u_0, \ldots, u_{n+k}), \\ V = (v_0, \ldots, v_{m+l})$$

is given by

$$\mathbf{d}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} \mathbf{d}_{ij} N_i^k(u) N_j^l(v), \quad (u,v) \in [u_0, u_{n+k}) \times [v_0, v_{m+l}) \subset \mathbb{R}^2$$

The control points $\mathbf{d}_{ij}$ form the control net of the surface.

Properties

- Analogous to the description for Bézier tensor-product surfaces.

Algorithms

- Analogous to the description for Bézier tensor-product surfaces.

5.7 Bézier Triangles

A triangular Bézier patch is defined by

$$b(u,v,w) = \sum_{\substack{i+j+k=n \\ i,j,k>0}} \mathbf{b}_{ijk} B_{ijk}^n(u,v,w)$$

The u, v and w are *barycentric coordinates* of the triangular parameter domain.

Generalised Bernstein polynomials:

$$B_{ijk}^n(u,v,w) = \underbrace{\frac{n!}{i!j!k!}}_{\text{Multinomial Coefficient}} u^i v^j w^k$$

Properties

- the same as in the univariate case

Algorithms

- de Casteljau:
$$b^l_{ijk} = ub^{l-1}_{i+1jk} + vb^{l-1}_{ij+1k} + wb^{l-1}_{ijk+1}$$
- subdivision
- degree elevation

5.8 Subdivision Surfaces

- polygon-mesh surfaces generated from a base mesh through an iterative process that smoothes the mesh while increasing its density
- represented as functions defined on a parametric domain with values in $\mathbb{R}^3$
- allow to use the initial control mesh as the domain
- developed for the purpose of CG and animation

The Idea

- in each iteration
  1. refine the initial control mesh
  2. increase the number of vertices and faces
- the mesh vertices converge to a limit surface

Classification

- type of *refinement rule*
  - vertex insertion
  - corner cutting
- type of *generated mesh*
  - triangular
  - quadrilateral
- approximating vs. interpolating

5.8.1 Loop's Scheme

- for triangular meshes
- new edge vertices as a weighted average of the adjacent vertices
- new vertices as another weighted average

5.8.2 Catmull-Clark Scheme

- for quad meshes
- new face vertice
- new edge vertices
- new vertex as a weigted sum of face and edge vertices

5.9 Other Types of Modeling

- not everything can be represented as meshes
- topological change over time
- other modeling ideas

5.10 Particle Systems

- modeling of objects changing over time
  - flowing
  - billowing
  - splattering
  - expanding
- modeling of natural phenomena
  - rain
  - snow
  - clouds
  - explosions
  - fireworks
  - smoke
  - fire
  - sprays
  - waterfalls
  - lumps of grass
- certain number of particles is rendered
- particle parameters change over time
  - location
  - speed
  - appearance
- particles die
- particle shapes may be
  - spheres
  - boxes
  - splats (billboards)
  - arbitrary models
- size and shape may vary over time
- motion may be controlled by external forces e.g. gravity, wind
- particles interfere with other particles

5.10.1 Grass Clumps

- particles can be used to model other objects
- e.g. grass clumps can be modelled by firing particles upward, and tracing their path
- lifetime can be encoded by color

5.10.2 Fluid Simulations

- Euler
  - simulate on a regular grid
- Lagrange
  - simulate particles (Smooth Particle Hydrodynamics)

5.11 Implicit Modeling

- surface or object are defined through functions not meshes
- no fixed shape and topology
- modeling of
  - molecular structures
  - water droplets
  - melting objects
  - muscle shapes
- shape and topology change
  - in motion
  - in proximity to other objects
- no seams
- oriented surface
  - well defined inside and outside
- differentiable
- closed
- continuous

An implicit equation $f(x, y) = -(x^2 + y^2) = T$ might define a 2D curve. T is called the threshold. For every equation of that form, with a T on the right side, the T is often called *level*, and the resulting surfaces are called *level sets*.

Level Sets

- $\mathcal{R}^2 \mapsto \mathcal{R}$ level curve (*iso contour*, contour line)
- $\mathcal{R}^3 \mapsto \mathcal{R}$ level surface (*iso surface*)
- $\mathcal{R}_n \mapsto \mathcal{R}$ level hypersurface

In an $n + 1$ dimensional space, an n dimensional hypersurface can be modelled.

The surface of an implicit model is defined as the set of points that fulfill the implicit equation.

5.11.1 Blobby Objects

Sum of Gaussian density functions centered at the k control points $X_k = (x_k, y_k, z_k)$.

$$f(x, y, z) = \sum_k b_k e^{-a_k r_k^2} - T = 0$$

where

$$r_k^2 = (x - x_k)^2 + (y - y_k)^2 + (z - z_k)^2$$

T is a specified threshold, and a_k and b_k adjust the *blobbiness* of control point k. b changes the *height* of the function, and a the *width* of the function.

The problem with the Gaussian density function is, that it never reaches 0.

Metaballs model uses density functiosn that drop off to 0 at a finite intervall. *Quadratic* functions are used. This means that not every control point k has an influence on all the other points. Because of that, less space needs to be computed.

Soft Object model uses the same approach but with a different density-distribution characteristic.

Other Options

- inhomogeneous density
- distance functions (*distance transform*), like interpolated distance

Rendering

- Marching Cubes
  - evaluate function on a regular grid
  - find cells with different signs
- Ray Tracing
  - cast ray
  - evaulate function in intervals along the ray
  - on a sign change, the surfance will be in the last interval

5.12 Procedural Modeling

- high geometric complexity
- low memory footprint
- object does no exists as geometry but as a set of rules

5.12.1 Superquadrics

- *quadrics* are conic sections (Kegelschnitte). Have the form $x^2 + y^2 + z^2$ according to the dimensions.
- generalization of quadric representation
  - especially the power has a parameter s now
- additional parameters
- increased flexibility for adjusting object shapes
- one additional parameter for curves and two parameters for surfaces

Superellipse Exponent of x and y terms of a standard ellipse are allowed to be variable:

$$\left(\frac{x}{r_x}\right)^{\frac{2}{s}} + \left(\frac{y}{r_y}\right)^{\frac{2}{s}} = 1$$

Changing s changes the shape of the ellipse:

- $s = 1 \implies$ circle
- $s = 2 \implies$ diamond

Superellipsoid Exponent of x, y and z terms of a standard ellipsoid are allowed to be variable:

$$\left(\left(\frac{x}{r_x}\right)^{\frac{2}{s_2}} + \left(\frac{y}{r_y}\right)^{\frac{2}{s_2}}\right)^{\frac{s_2}{s_1}} + \left(\frac{z}{r_z}\right)^{\frac{2}{s_1}} = 1$$

5.12.2 L-Systems

- use formal languages to construct geometry
- *turtle graphics*

5.12.3 Shape Grammar

- developed for architects, especially for facades
- a language that can describe 2D planes
- context sensitiv

5.12.4 Split Grammar

- like Shape Grammar but simpler
- no context sensitivity
- symbols are replaced

5.12.5 Generative Modeling Language

- like PostScript for geometry (stack oriented)
- changing parameters yields very different models
- complex geometry with rules and parameters

5.12.6 Sweeps

- modeling of objects with symmetries
  - translational
  - rotational
- represented by
  - 2D shape
  - sweep-path

Translational Sweeps

- translate a curve along a direction
- e.g. create cylinder from path and circle

Rotational Sweeps

- rotate a curve around an axis
- e.g. create a torus from an axis and circle

General Sweeps

- seep a curve along a path
- e.g. create tube from path and circle
- path can be animated over time

Advantages

- generates shapes that are hard to do otherwise

Disadvantages

- hard to render
- difficult modeling

5.12.7 Cellular Texture Generation

- a cellular particle system, that changes the geometry of the surface
  - cell state
    - position
    - orientation
    - shape
    - chemical concentrations (reaction-diffusion)
  - cell programms
    - go to surface
    - die if too far from surface
    - align
    - adhere to other cells
    - divide until surface is covered

    - extracellular environments
        * neighbor orientation
- can be used e.g. fur
- automatic LOD generation

5.12.8 Knitwear

- simulation of thin 3D structure with instanced volume elements
- generate knits as sweeps along paths
- can be rendered with ray tracing

5.12.9 Terrain Simulation

- fractals
- geographical data
- simulation (errosion)
- hybrids

5.12.10 Ecosystem Modeling

- terrain
- procedural plant models
- weight maps

Specification of Plant Populations

- Space-occupancy
    - explicit specification (counting plants)
    - procedural generation (cellular automata, reaction-diffusion)
- Individual based
    - explicit specification (survey, interactive specification, real world data)
    - procedural generation (point pattern generation model)

Plant Models

- complex models are necessary for realistic appearance
    - plant distribution by ecosystem simulation
    - reduce geometric complexity by approximate instancing
        * similar plants
        * groups of plants
        * plant organs
    - parameterized models of individual plants

Self Thinning

- some plants have a radius around them, where other plans get dominated
- no other plants grow there
- more realistic look

5.13 Structure-Deforming Transformations

- non-linear transformations
    - *tapering* non-linear scaling

  - *twist* non-linear rotation
  - *bend* non linear rotation

Tapering the scale factor is a function:

$$\mathbf{x}' = \begin{pmatrix} f_x(\mathbf{x}) & 0 & 0 \\ 0 & f_y(\mathbf{x}) & 0 \\ 0 & 0 & f_z(\mathbf{x}) \end{pmatrix} \cdot \mathbf{x}$$

Twist the angle of rotation is a function: (e.g. for rotation around the z-axis)

$$\mathbf{x}' = \begin{pmatrix} \cos f(\mathbf{x}) & -\sin f(\mathbf{x}) & 0 \\ \sin f(\mathbf{x}) & \cos f(\mathbf{x}) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \mathbf{x}$$

Bend

$$y = \begin{cases} -\sin(\theta) \cdot (z - r) + y_0, & y_{\min} < y < y_{\max} \\ -\sin(\theta) \cdot (z - r) + y_0 + \cos(\theta) \cdot (y - y_{\min}), & y < y_{\min} \\ -\sin(\theta) \cdot (z - r) + y_0 + \cos(\theta) \cdot (y - y_{\max}), & y > y - \max \end{cases}$$

$$z = \begin{cases} \cos(\theta) \cdot (z - r) + y_0 + r, & y_{\min} < y < y_{\max} \\ \cos(\theta) \cdot (z - r) + y_0 + r + \sin(\theta) \cdot (y - y_{\min}), & y < y_{\min} \\ \cos(\theta) \cdot (z - r) + y_0 + r + \sin(\theta) \cdot (y - y_{\mathrm{nax}}), & y > y_{\max} \end{cases}$$

5.14 Summary

5.14.1 Curves

- explicit
- implicit
- parametric

5.14.2 Subdivision Surface

- refinement rule
  - vertex insertion
  - corner cutting
- generated mesh
  - triangular
  - quadrilateral
- approximating vs. interpolating

5.14.3 Fluid Simulations

- Euler
- Lagrange

5.14.4 Procedural Modeling

- super quadratics
  - superellipses
  - superellipsoids
- L-System
- shape grammar

- split grammar
- generative modeling language
- sweeps
  - translational
  - rotational
  - general
- cellular texture generation
- knitwear
- terrain simulation
- ecosystem modeling

Name	Properties		Algorithms	Evaluation
	Curve	Basis Function		
Bézier Curve	• affine invariance • convex hull property • endpoint interpolation • linear precision • variation diminishing property	• global support	• degree elevation • subdivision	• de Casteljau • Horner scheme
B-Spline Curve	• affine invariance • *strong* convex hull property • linear precision • variation diminishing property	• local support • partition of unity • positivity	• degree elevation • knot insertion	• direct evaluation
Rational Bézier Curve	• same as Bézier curve • projective invariance • weights as design parameter	• global support	• same as Bézier curve for the homogeneous representation	-
NURBS	• same as B-Spline curve • projective invariance	• local support	• same as B-Spline curve for the homogeneous representation	-
Tensor-Product Bézier Surface	• same as Bézier curve	-	• same as Bézier curve • apply in two steps	-
Tensor-Product B-Spline surface	• same as Tensor-Product Bézier surface	-	• same as Tensor-Product Bézier surface	-
Bézier Triangles	-	• generalized Bernstein polynomial	• subdivision • degree elevation • de Casteljau	-

Table 6: Curves and surfaces summary

6 Sampling and Reconstruction

6.1 Image Data

One *scanline* of an image can be interpreted as a signal of light intensity. This works analogously for every channel of Red, Green and Blue.

Signals are represented as a sum of sine waves.

6.2 Fourier Series

Different equations for the same thing:

1. equation

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos(n\omega x - \varphi_n))$$

2. equation

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos(n\omega x) + b_n \sin(n\omega x))$$

3. equation

$$f(x) = \sum_{n=-\infty}^{\infty} \left(c_n e^{in\omega x}\right)$$

Euler's identiy: $e^{ix} = \cos x + i \sin x$

6.3 Fourier Transform

- transforms a singal into the *frequency space*
- Fourier Series with an *infinite* periode length
- link between spatial and frequency domain
- yields complex functions for frequency domain
- extends to higher dimensions
- complex part is pase information, which is usually ingored
- the *length* of the complex result is often used
- alternatives
  - Harley transform
  - Wavelet transform

$$F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i \omega x} dx$$
$$f(x) = \int_{-\infty}^{\infty} F(\omega) e^{2\pi i \omega x} d\omega$$

- the Fourier transform creates a *spectrum* of frequencies
- the value at the zero point is the average frequency called the *Gleichstromanteil*

6.3.1 Descrete Fourier Transform

- for discrete signals (sets of samples)
- complexity
  - DFT: $O(n^2)$
  - FFT: $O(n \log n)$

$$F(\omega) = \frac{1}{n}\sum_{\omega=0}^{n-1} f(x)e^{\frac{-2\pi j\omega x}{n}}$$
$$f(x) = \sum_{\omega=0}^{n-1} F(\omega)e^{\frac{2\pi j\omega x}{n}}$$

6.4 Box and Tent

- the *box function* and *tent functions* are special functions
- the tent function is just a convolution of two box functions
- the Fourier transform of the box function $f(x)$ is $F(u) = \text{sinc}$
- the Fourier transform of the tent function $f(x)$ is $F(u) = \text{sinc}^2$

6.5 Sinc Function

$$\text{sinc}(x) = \begin{cases} \frac{\sin \pi x}{\pi x}, & x \neq 0 \\ 1 & \text{otherwise} \end{cases}$$

An undesireable property of the sinc function is, that it has negative values, and that it is not zero in a finite range.

6.6 Convolution

- operation on two functions
- one function needs to be flipped
- result: sliding weighted average of a function
- the second function provides the weights

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x-\tau)g(\tau)d\tau$$

- the convolution of two box functions yield the tent function
- the convolution of samples and the tent function yield the linear interpolated samples
- the convolution of samples and the box function yields nearest neighbors interpolated samples

6.6.1 Convolution Theorem

The spectrum of the convolution of two functions is equivalent to the product of the transforms of both input signals, and vice versa:

$$F(f_1 * f_2) \equiv F(f_1) \cdot F(f_2)$$

where F is the Fourier transform. This also means:

$$F^{-1}(f_1 * f_2) \equiv F^{-1}(f_1) \cdot F^{-1}(f_2)$$

Convolution in one space is equivalent to multiplication in another space.

6.6.2 Low-Pass

- in the *spatial domain*:
  - convolution with the sinc function
  - smooths the function
- in the *frequency domain*:
  - multiplication with box function
  - cutoff of high frequencies (value far away from the zero point)
- the sinc function corresponds to the box function and vice versa

6.7 Sampling

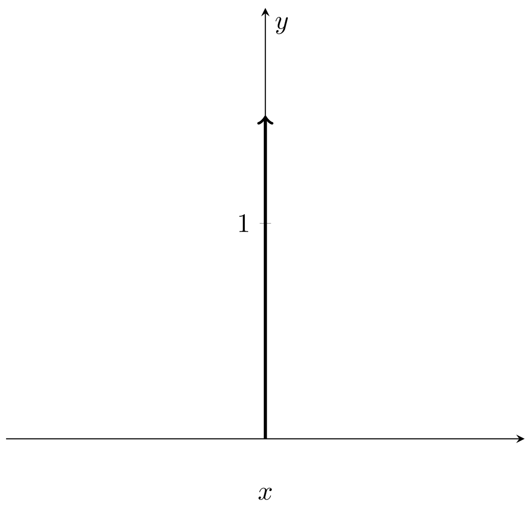
Sampling is the evaluation of the signal at a specific point. A central problem of sampling is the sampling frequency. How many samples should be taken?

The process of sampling is a multiplication of the signal with a *comb* function comb_T, where T is the sampling interval.

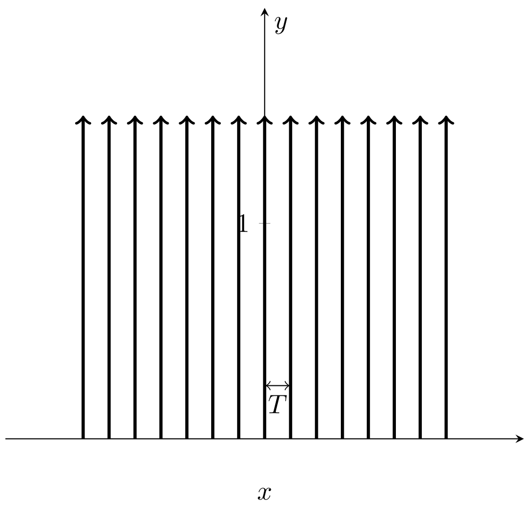
$$f_s(x) = f(x) \cdot \text{comb}_T(x)$$

The frequency response is convolved with a transformed comb function. The spacings of the comb function is inversely proportional to that in the frequency space.

$$F_s(\omega) = F(\omega) * \text{comb}_{\frac{1}{T}}(\omega)$$



(a) Dirac Impulse



(b) Comb Function

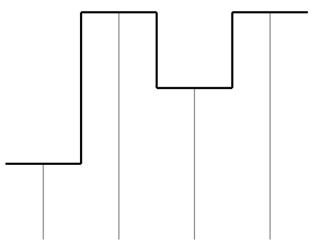
Sampling a signal in the spatial domain with the comb function comb_T yields the same result as convolving the spectrum (the function in the frequency space, after the Fourier transform), with the comb function $\text{comb}_{\frac{1}{T}}$. This creates many duplicates of the original spectrum, one for each comb strand.

If the original specrum has high frequency parts on the far side of the spectrum, the copying places them on top of other frequencies, possibly closer to the center point of the frequency space. The area where the low frequencies reside. This means high frequencies are appearing as low frequencies. The are using an *alias.* Thus this phenomenon is called *aliasing.*

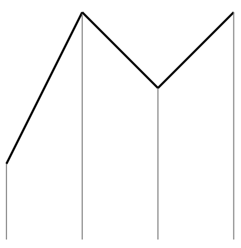
This means, always aliasing stems from *sampling the signal.* Every other error is bad reconstruction. That means doing linear interpolation as reconstruction might lead to a different outcome than the original signal, but that would not be aliasing.

6.8 Reconstruction

Is the reconstruction of the whole signal based on samples. This can be done in different ways, for example:



(a) Nearest Neighbor Reconstruction



(b) Linear Interpolation Reconstruction

A function is called *band-limited* if it contains no frequencies outside of the interval $[-u, u]$. u is called the *bandwidth* of the function. This can be thought of as the highest frequency that occurs.

The *Nyquist frequency* of a function is twice its bandwidth: $w = 2u$.

Sampling a function with the Nyquist frequency erradicates aliasing. This is because the spectra will not overlap, as it has a *width* of $2u$ which is, by definition, equal to the Nyquist frequency.

6.8.1 Sampling Theorem

A function $f(x)$ that is (i) band-limited and (ii) samped above the Nyquist frequency is completely determined by its samples.

6.8.2 Ideal Reconstruction

For an ideal reconstruction of the original signal, the copies of the spectrum in the frequency domain must not overlap, or else aliasing will occur.

This is achieved by sampling with a frequency higher than the Nyquist frequency. This results in the *shadow spectra* being far enough apart to not overlap.

Now, how can the original spectrum be recovered from all the replicas? Simply multiplying the spectrum with the box filter with the width of one bandwidth of the original signal.

Multiplying the spectrum with the box filter in the frequency space is equialent to a convolution of the sampled signal with the sinc function.

Problems In real live applications the infinite nature of the sinc function cannot always be expressed. This leads to an incorrect reconstruction. Cutting off the sinc function, to avoid this problem creates another. This is because cutting off the sinc function is equivalent to multiplying it with the box function. This in turn is equivalent to a convolution of the box filter with the sinc function. This adds high frequencies to the box function.

Used Reconstruction Filters

- nearest neighbor
- linear interpolation
- symmetric cubic filters
- windowed sinc
- Gaussian-Filter

6.8.3 Linear interpolation

Linear interpolation is equivalent to a convolution of the samples with a tent function. This in turn is equivalent to a multiplication of the sinc^2 function in the frequency space. This is not a perfect reconstruction, as some of the shadow spectra will be kept.

6.9 Band-Limiting a Signal

If the original signal has a frequency that is to high, and therefore it cannot adequately be sampled with twice its highest frequency, the Nyquist frequency, aliasing will occur.

But there is a workaround: the original freqency can be lowered. Before sampling the signal, it will put through a low-pass filter. That is equivalent with convoluting the original signal with the sinc function in the spatial domain, or multiplying the spectrum of the original signal with the box filter in the frequency domain.

Doing that before sampling the signal is called *band-limiting* the signal, and is also known as *anti-aliasing*.

This also means, that anti-aliasing can only be done on the original analog signal.

6.10 Errors

- *Aliasing*: due to overlap of original frequency spectrum with replicas (information loss)
- *Truncation Error*: due to use of a finite reconstruction filter instead of the infine sinc function.
- *Non*-sinc error: due to use of a reconstruction filter that has a shape different from the sinc filter.

7 Texturing

7.1 Texture

A *texture* is a function:

- *procedural*
  - fast to evaluate
  - limited in variety
- *sampled*
  - most common method
  - raster images are taken with a digital camera, scanned or synthesized

Textures can be defined:

- in 3D object space
  - 3D texturing
  - volume mapping
- on 2D object surface
  - texture mapping
- as 1D function
  - lookup table
  - e.g. sample the sin function and create a lookup table

7.2 Cube Mapping

Is used to encode the environment (skybox). To achieve this a 3D texture would be overkill:

- memory
- takes time to generate
- takes time to sample

The solution would be to take 6 planes, oriented to form a cube, where each side is displaying a texture. The texture is addressed via a 3D vector, a direction, not a position.

This is often use for the skybox, or to approximate reflections, where the direction of the reflected vector is used as a look up key in the cube map.

7.3 3D Texturing

- not directly applied for polygonal models
- representation of some 3D value field
- applications
  - volume rendering
  - media rendering (gas, liquid)
  - hybrid rendering

7.4 Parameterization

Parameterization is the process of mapping a texture to a surface.

The projection of textures onto 3D objects often comes with distortions of that texture. Additionally, for non-convex objects, the mapping is not *bijective*.

7.4.1 1D Texture

- parameter can have arbitrary domain (along one axis, incident angle, etc.)

7.4.2 2D Texture

- projection of 2D data
  - orthogonal projection
  - cylindrical parameterization
  - spherical parameterization

7.5 Texture Mapping

- parameterization per vertex
  - apply (u, v) texture coordinates to every vertex
- at runtime *(bi)linearly interpolate* between these coordinates during rasterization

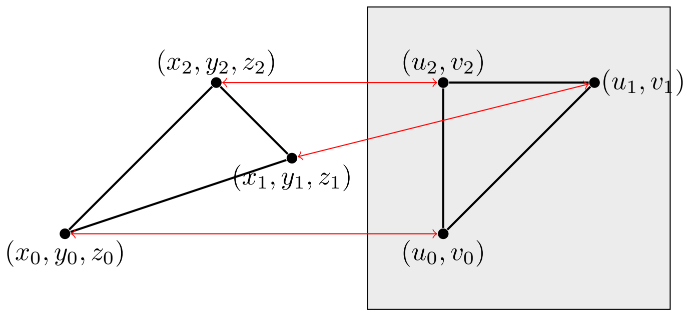


Figure 12: Texture Mapping

The interpolation of the texture on a triangle can be achieved using *barycentric coordinates*. These are weighted sums of coordinates of the corners of the triangle. The coefficients of the barycentric coordinates of the point in the object space, can also be used to compute the correct interpolated point in the texture space.

7.5.1 Perspective Correctness

The problem is, that affine texture mapping does not take the depth information into account.

$$u_\alpha = (1-\alpha)u_0 + \alpha u_1, \quad 0 \leq \alpha \leq 1$$

To map the texture correctly on the object, the texture coodinates also have to be transformed, according to the perspective projection. We need to homogenize the (u, v) coordinates.

$$u_\alpha = \frac{(1-\alpha)\frac{u_0}{z_0} + \alpha\frac{u_1}{z_1}}{(1-\alpha)\frac{1}{z_0} + \alpha\frac{1}{z_1}}$$

7.6 Aliasing

This problem arises, if one pixel in the image space covers multiple texels in texture space. Aliasing can also be caused by *undersamling*. That means, that texels in the texture space are skipped, and information that was not adjacent in texel space is adjacent in image space. This leads to a loss of information.

7.7 Anti-Aliasing

A good pixel value is the weighted mean of the area projected into texture space.

7.7.1 Direct Convolution

Calculation of the weighted mean at runtime called *direct convolution*. The texel "footprint" is approximated with other shapes, like

- rectangles
- quadrilaterals through the original corners

- ellipses

7.7.2 Prefiltering

Another option for anti-aliasing would be to prefilter the textures beforehand. This would be a calculation of weighted mean in a preprocessed texture.

There are different approaches to prefiltering:

Summed Area Table

- calculate a new texture S, where for each texel, the value is calculated as the sum of all the texels left and below the new texel in the original texture T:

  $$S(u_0, v_0) = \sum T(u, v), \quad u \leq u_0, v \leq v_0$$

  The new texture S can then be used to look up any rectangular area, with origin $(0, 0)$.
- But often, an arbitrary rectangular area of the texture is wanted, not just areas which are bound to the origin of the texture.

  This can be done by subtracting all rectangular parts that are not needed. With 4 lookups all areas can be created.
- Summed area tables do not work with large textures, without loosing numeric precision. The texel opposite to the origin hold the sum over all other texels in the texture, which, by nature of the textures size, might be a huge number.

Mip-Mapping MIP, meaning *multum in parvo* (lots in less). In mip-mapping the texture size is reduces by a factor of 2. This is also called *down sampling.* This is done by using a simple 4×4 pixel average. The final image is only one texel large. All different stages of the downsampling are saved in one file.

The total memory consumption of mip-maps are at most $\frac{4}{3}$ of the original texture memory consumption.

To decide what mip-mapping-level D_0 to use, the following algorithm is used. Start by drawing two diagonals d_1 and d_2 between the four corners of the projected pixel in the texel space. The mip-map level can be calculated as follows:

$$D := \text{ld}(\max(d_1, d_2))$$

Because ld maps to real numbers, we need to truncate the value to get the actual integer level:

$$D_0 := \text{trunc}(D)$$

Because of this truncation *mip banding* can occur. That are clear lines on geometry, typically roads, where the transition between different mip-map-levels can be seen.

After sampling the texture at the corret mip-map-level, we still need to reconstruct this signal. This is usually done with *trilinear interpolation.*

7.7.3 Bilinear Interpolation

Bilinear reconstruction for texture magnification (*oversampling*). That means too many pixels for one texel.

For a pixel that gets mapped onto $(u + \Delta u, v + \Delta v)$, the neighboring texels at (u, v), $(u + 1, v)$, $(u, v + 1)$ and $(u + 1, v + 1)$ get interpolated. First along the u axis, then the v axis. Because two interpolations happen, it is called *bilinear interpolation.*

$$\begin{aligned} T(u + \Delta u, v + \Delta v) = {} & \Delta u \cdot \Delta v \cdot T(u + 1, v + 1) \\ & + \Delta u \cdot (1 - \Delta v) \cdot T(u + 1, v) \\ & + (1 - \Delta u) \cdot \Delta v \cdot T(u, v + 1) \\ & + (1 - \Delta u) \cdot (1 - \Delta v) \cdot T(u, v) \end{aligned}$$

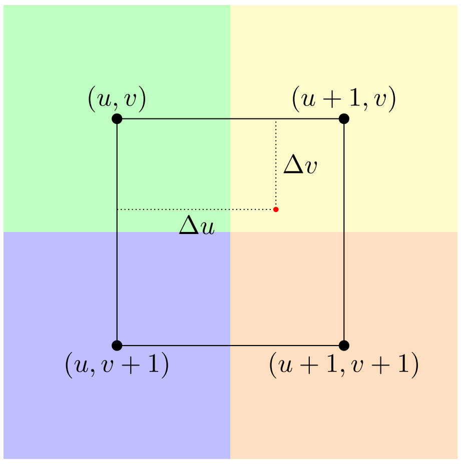


Figure 13: Bilinear Interpolation

7.7.4 Trilinear Interpolation

Trilinear interpolation interpolates between values of two adjacent mip-map levels. Because each of the mip-map-levels is interpolated, in total three interpolations happen, thus coining the phrase trilinear interpolation.

$$\begin{aligned} T_0 &:= \text{value from texture } D_0 \\ T_1 &:= \text{value from texture } D_1 = D_0 + 1 \\ \text{pixel value} &:= (D_1 - D) \cdot T_1 + (D - D_0) \cdot T_1 \end{aligned}$$

where D is the actual, non truncated mip-map-level. T_0 and T_1 are also bilinearly interpolated.

Trilinear interpolation gets rid of the unwanted *mip banding*, but needs double the texel lookpup.

7.8 Anisotropic Filtering

The problem with mip-mapping is, that only rectangular of textures areas can be properly sampled. But with perspective projection, a ground plane might have multiple rows of texels mapped onto only a few rows of pixes. That means a large rectangular area needs to be sampled, introducing colors from areas that are not wanted. The result is a bleeding effect.

Anisotropic filtering tries to combat that. In essence an non rectangular area for sampling is the goal.

7.8.1 Rip-Mapping

Rip-Mapping is similar to mip-mapping, only that it also saves the different levels in different aspect ratios. This is fast, but needs more memory.

7.8.2 Footprint Assembly

This is anisotropic filtering embedded in hardware.

- approximate pixel footprint by parallelogram
- sample along the parallelogram using mip-mapping

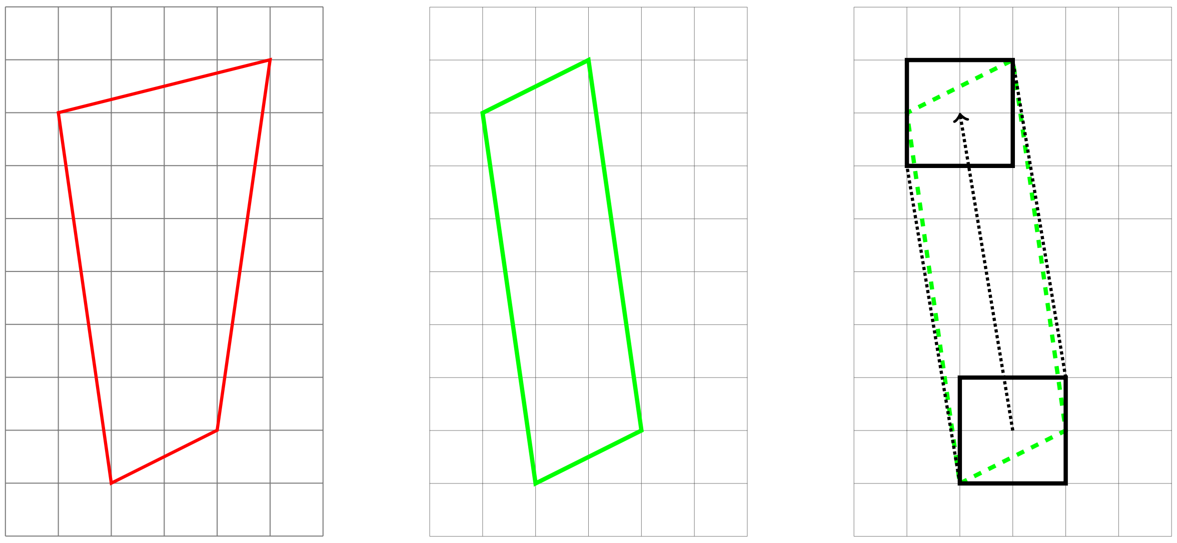

(a) Footprint in texture space (b) Approximation by parallelogram (c) Sampling using mip-mapping

Figure 14: Footprint Assembly

7.9 Bump Mapping

Use textures to encode height information. This can be done in two steps:

1. *preprocess*
   - compute normal vectors from height information
2. *runtime*
   - use computed normals for illumination

This illusion of depth breaks down on the silhouette. Also, bumps created by this method cannot cast shadows on other geometry.

7.10 Horizon Mapping

This problem, of not being able to cast shadows can be solved with horizon mapping. With horizon mapping local shadows can be created.

1. *preprocess*
   - compute n horizon values per texel
2. *runtime*
   - interpolate horizon values
   - shadow accordingly

7.11 Parallax Mapping

Parallax mapping tries to simulate the parallax by shifting the texture lookup position. Parallax mapping cannot model occlusions.

1. *preprocess*
   - compute *texture coordinate shifts*
2. *runtime*
   - alter texture lookup position

Without interpolation banding will occur.

7.12 Relief Mapping

- *runtime*

  - perform ray casting in the fragment shader
    1. calculate entry point A and exit point B
    2. march along the ray until an intersection with the height field is found. The stepping size is usually one texel
- can be sped up with binary search along the ray

7.13 Summary

7.13.1 Parameterization

- 1D
  - arbitrary domain
- 2D
  - orthogonal projection
  - cylindrical parameterization
  - spherical parameterization

7.13.2 Mappings

- Cube Mapping
- Texture Mapping
- Bump Mapping
- Horizon Mapping
- Parallax Mapping
- Relief Mapping

7.13.3 Anti-Aliasing

- direct convolution
- prefilitering
  - summed area table
  - mip-mapping

7.13.4 Anisotropic Filtering

- rip-mapping
- footprint assembly

8 Lightning

8.1 Bidirectional Reflectance Distribution Function (BRDF)

$$R(\theta_i, \phi_i, \theta_o, \phi_o)$$

- 4D function
- 2 angles for incoming direction (spherical coordinates)
- 2 angles for outgoing direction (spherical coordinates)

The incomming light is called *irradiance*, the outgoing *radiance*.

If the BRDF is constant it is equivalent with *Lambert Reflectance*.

8.2 Incoming Radiance

- amount of light received by a surface depends on the incoming angle

- this can be modelled with *Lambert's Cosine Law*

$$\cos\theta = \langle\, l_i, n \,\rangle$$

The light intensity behaves like the cosine of the incoming light with the surface normal. This can also be calculated with the dot product of the surfaces normal with the incoming light direction.

8.3 Ideal Diffuse Reflectance

- assume surface reflects equally in all directions
- at a microscopic level, this means a very rough surface (like chalk)

8.3.1 Single Point Light Source

$$L_o = k_d(\mathbf{n} \cdot \mathbf{l})\frac{L_i}{r^2}$$

k_d ... diffuse coefficient (often the object's color)
$\mathbf{n}$ ... surface normal
$\mathbf{l}$ ... light direction
L_i ... light intensity
r ... distance to light source

Implementation specifics

- use dot product
- normalize vectors
- use $\max(\mathbf{n} \cdot \mathbf{l}, 0)$, because illumination cannot be negative

8.4 Ideal Specular Reflectance

- incoming ray is reflected in exactly one outgoing direction
- at a microscopic level, this means the surface elements are oriented in the same direction as the surface itself (mirror, polished metals)

8.4.1 Phong Lightning Model

$$\begin{aligned} L_o &= k_s(\cos\alpha)^q \frac{L_i}{r^2} \\ &= k_s(\mathbf{v} \cdot \mathbf{r})^q \frac{L_i}{r^2} \end{aligned}$$

k_s ... specular reflectance coefficient
q ... reflection exponent (sometimes called α)
$\mathbf{v}$ ... view direction
$\mathbf{r}$ ... reflectance direction

8.4.2 Blinn-Phong Lightning Model

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|}$$

$$L_0 = k_s(\mathbf{n} \cdot \mathbf{h})^q \frac{L_i}{r^2}$$

h . . . halfway vector between **l** and **v**
l . . . incoming light direction
v . . . view direction
n . . . surface normal

8.5 Ambient Illumination

$$L(\omega_r) = k_a L_a$$

- represents the reflection of all indirect illumination
- just a workaround to *global illumination*

8.6 Whole Lightning Model

The whole lightning model is the sum of all of its components:

- ambient illumination
- diffuse reflection
- specular reflection

Phong Illumination Model

$$L_0 = \underbrace{k_a L_a}_{\text{ambient}} + \left(\underbrace{k_d(\mathbf{n} \cdot \mathbf{l})}_{\text{diffuse}} + \underbrace{k_s(\mathbf{v} + \mathbf{r})^q}_{\text{specular}} \right) \frac{L_i}{r^2}$$

9 Physically-Based Shading

One important idea of physically based shading is the *conservation of energy*.

9.1 Law of Reflection / Snell's Law

This assumes a perfectly flat surface:

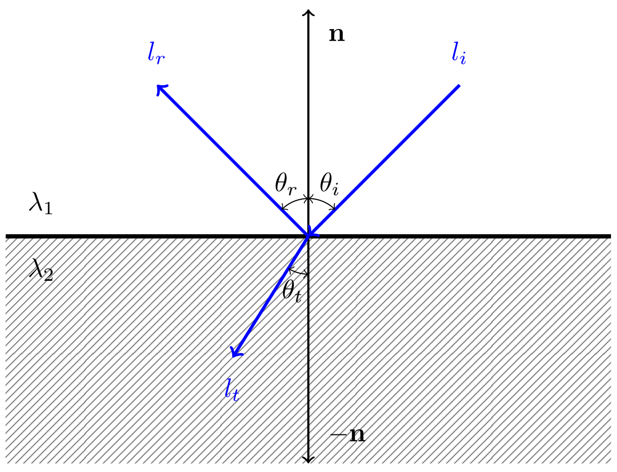


Figure 15: Snell's Law

- reflection

$$\theta_r = \theta_i$$

- transmission

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\lambda_1}{\lambda_2}$$

9.2 Fresnel's Law

Energy must be conserved:

$$l_i = l_r + l_t$$

with

$$\begin{aligned} l_r &= F(\theta_i) l_i \\ l_t &= (1 - F(\theta_i)) l_i \end{aligned}$$

where $F(\theta)$ is the *Fresnel reflectance.*

9.2.1 Fresnel Reflectance

The Fresnel reflectance is defined as

$$F(\theta) \approx F_0 + (1 - F_0)(1 - \cos\theta)^5$$

where F_0 is the *characteristic reflactance* of the material

$$F_0 = \left(\frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2}\right)^2$$

where λ_1 and λ_2 are the *refractive indices* (IOR).

Metals vs. Non-Metals

- non-metals have *monochrome reflections*
  - $F_0 = F_{0_{\text{red}}} = F_{0_{\text{green}}} = F_{0_{\text{blue}}}$
- metals can have tinted reflections
  - $F_{0_{\text{red}}} \neq F_{0_{\text{green}}} \neq F_{0_{\text{blue}}}$

Metallic-Roughness vs. Specular-Shininess Model

- Metallic-Roughness Model
  - F_0 is fixed ($\approx 4\,\%$)
- Specular-Shininess Model
  - F_0 is freely adjustable

9.3 Rendering Equation

The mirco scale of a surface can be simulated on the macro scale using statistical models. This includes modeling how much light is hit on a hemisphere at a cerain point.

$$l_o(\mathbf{v}) = l_e(\mathbf{v}) + \int_\Omega f(\mathbf{l}, \mathbf{v}) l_i(\mathbf{l})(\mathbf{n} \cdot \mathbf{l}) dA$$

$\mathbf{v}$	. . . view direction
$\mathbf{l}$	. . . light direction
$\mathbf{n}$	. . . surface normal vector
$l_o(\mathbf{v})$	. . . light outgoing in the view direction $\mathbf{v}$
$l_i(\mathbf{l})$	. . . light incoming from direction $\mathbf{l}$
$l_e(\mathbf{e})$	. . . light emitted in view direction $\mathbf{v}$
$f(\mathbf{l}, \mathbf{v})$	. . . fraction of incoming light from $\mathbf{l}$ reflected in direction $\mathbf{v}$

The function f is the *BRDF*. It is high, if light from $\mathbf{l}$ reflects strongly into $\mathbf{v}$ and vica versa.

9.3.1 Approximation

The function f can be approximated with a new constant c. If the BRDF is constant, it is called *Lambert Reflectance.* The directional light l_i can be approxiamted with the *Lambert Term* $\mathbf{n} \cdot \mathbf{l}_{\text{dir}}$.

$$\begin{aligned} l_o(\mathbf{v}) &= \int_\Omega f(\mathbf{l}, \mathbf{v}) l_i(\mathbf{l})(\mathbf{n} \cdot \mathbf{l}) dA \\ &\approx cL(\mathbf{n} \cdot \mathbf{l}_{\text{dir}}) \end{aligned}$$

9.4 Microfacet Model

- defines families of BRDFs for *specular* reflection
- derived from micro geometry
- statistical model of how the distribution of normal vectors

We assume a model which has a normal vector $\mathbf{n}$. The actual normal of the irregular underlying surface is called $\mathbf{m}$. These normales are modeled with a statstical distribution.

If the surface only has a gentle slope at some small area, the angle θ between $\mathbf{n}$ and $\mathbf{m}$ is small. If it has a steep slope, θ is large. That means the distribution of θ is proportional to the smoothness of the material:

- smooth surface $\implies$ small θ's are more common
- rough surface $\implies$ large θ's are more common

9.4.1 Normal Distribution

This distibution $D(\theta)$ is called the *Normal Distribution Function* (NDF).

There are many options for $D(\theta)$. One of the more famous option is the *Trowbridge-Reitz NDF* (GGX):

$$D_{\text{tr}}(\theta, \alpha) = \frac{\alpha^2}{\pi \left((\cos^2 \theta)(\alpha^2 - 1) + 1 \right)^2}$$

Which can be brought into a form that is more useful for shader computations:

$$D_{\text{tr}}(\mathbf{m}, \alpha) = \frac{\alpha^2}{\pi \left((\mathbf{n} \cdot \mathbf{m})^2 (\alpha^2 - 1) + 1 \right)^2}$$

where

α ... roughness

$\mathbf{n}$ ... normal of the model

$\mathbf{m}$ ... modelled distributed normal

9.4.2 Microfacets

Every little microfacet behaves like a perfect mirror. Light is reflected between the light direction $\mathbf{l}$ and the viewing direction $\mathbf{v}$. The halfvector $\mathbf{h}$ behaves like the normal of that microfacet.

Over a large mesh, only these *patches* where the halfvector $\mathbf{h}$ is equal to the underlying normal $\mathbf{h} = \mathbf{m}$ can contribute to the actual reflection of the large mesh.

Since the underlying normals $\mathbf{m}$ are statistically distributed, we can calculate for a specific $\mathbf{h}$ how many $\mathbf{m}$ there should be in relation to the whole mesh.

In practice, when the light hights a point p the distribution function D is sampled with the halfvector $\mathbf{h}$. Over the course of multiple hits, the actual normal will be approached.

9.4.3 Microfacet Occlusions

Not every facet $\mathbf{h} = \mathbf{m}$ has to contribute. The facet might be occluded by a bump on the surface. Either it is:

- *Shadowing* which means, the incoming light is blocked by geometry, or
- *Masking* which means, the outgoing light (reflection) is blocked by geometry.

The geometry function $G(\mathbf{l}, \mathbf{v}, \mathbf{n})$ is modeling a *Shadow-Masking* function function.

$$G_{\text{uncor}}(\mathbf{l}, \mathbf{v}, \mathbf{n}) = \underbrace{G_1(\mathbf{l}, \mathbf{n})}_{\text{shadowing}} \underbrace{G_1(\mathbf{v}, \mathbf{n})}_{\text{masking}}$$

The function G_1 is defined as:

$$G_1(\mathbf{x}, \mathbf{n}) = \frac{1}{1 + \Lambda(\mathbf{x} \cdot \mathbf{n})}$$

with Λ

$$\Lambda(r) = \frac{-1 + \sqrt{1 + \frac{\alpha^2(1-r^2)}{r^2}}}{2}$$

The shadow-masking function is usually derived from the distribution function D. It is important, that these two functions fit together. The Λ function can be changed based on the distribution function, the rest is usually not changed.

The uncorrelated version, where the actual *height* of the point on the surface does not matter is defined as follows:

$$\begin{aligned} G_{\text{uncor}}(\mathbf{l}, \mathbf{v}, \mathbf{n}) &= G_1(\mathbf{l}, \mathbf{n}) G_1(\mathbf{v}, \mathbf{n}) \\ &= \frac{1}{1 + \Lambda(\mathbf{l} \cdot \mathbf{n})} \cdot \frac{1}{1 + \Lambda(\mathbf{v} \cdot \mathbf{n})} \end{aligned}$$

The correlated version is defined as:

$$G_{\text{cor}}(\mathbf{l}, \mathbf{v}, \mathbf{n}) = \frac{1}{1 + \Lambda(\mathbf{l} \cdot \mathbf{n}) + \Lambda(\mathbf{v} \cdot \mathbf{n})}$$

9.4.4 Whole Microfacet Model

$$f_{\mu\text{facet}}(\mathbf{l}, \mathbf{v}) = \frac{\overbrace{F(\mathbf{l}, \mathbf{h})}^{\text{Fresnel's Law}} \; \overbrace{G(\mathbf{l}, \mathbf{v}, \mathbf{n})}^{\text{Shadow-Masking function}} \; \overbrace{D(\mathbf{h})}^{\text{NDF}}}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})}$$

Sometimes the term:

$$\frac{G(\mathbf{l}, \mathbf{b}, \mathbf{n})}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})}$$

is combined into a so called *visibility term*:

$$V(\mathbf{l}, \mathbf{v}, \mathbf{n})$$

yielding:

$$f_{\mu\text{facet}}(\mathbf{l}, \mathbf{v}) = F(\mathbf{l}, \mathbf{h}) V(\mathbf{l}, \mathbf{v}, \mathbf{n}) D(\mathbf{h})$$

9.4.5 Microfacet Rendering Equation

$$\begin{aligned} l_o(\mathbf{v}) &= l_e(\mathbf{v}) + \int_\Omega f_{\mu\text{facet}}(\mathbf{l}, \mathbf{v}) l_i(\mathbf{l})(\mathbf{n} \cdot \mathbf{l}) dA \\ &= l_e(\mathbf{v}) + \int_\Omega F(\mathbf{l}, \mathbf{h}) V(\mathbf{l}, \mathbf{v}, \mathbf{n}) D(\mathbf{h}) l_i(\mathbf{l})(\mathbf{n} \cdot \mathbf{l}) dA \end{aligned}$$