

Essential Mathematics

for computational design

Rajaa Issa

Robert McNeel & Associates



Essential Mathematics for Computational Design, second edition by Robert McNeel & Associates is licensed under a [Creative Commons Attribution-Share Alike 3.0 United States License](https://creativecommons.org/licenses/by-sa/3.0/us/).

Preface

Essential Mathematics for Computational Design introduces design professionals to foundation mathematical concepts that are necessary for effective development of computational methods for 3D modeling and computer graphics. This is not meant to be a complete and comprehensive resource, but rather an overview of the basic and most commonly used concepts.

The material is directed towards designers who have little or no background in mathematics beyond high school. All concepts are explained visually using Grasshopper® (GH), the generative modeling environment for Rhinoceros® (Rhino). For more information, go to www.rhino3d.com and www.grasshopper3d.com.

The content is divided into three parts. The first discusses vector math including vector representation, vector operation, and line and plane equations. The second part reviews matrix operations and transformations. The third part includes a general review of parametric curves with special focus on NURBS curves and the concepts of continuity and curvature. It also quickly reviews NURBS surfaces and polysurfaces.

I would like to acknowledge the excellent and thorough technical review of Dr. Greg Arden of Robert McNeel and Associates. His valuable comments were instrumental to produce this second edition. I would also like to acknowledge Ms. Margaret Becker of Robert McNeel and Associates for reviewing the technical writing and formatting the document. Finally, I'd like to point out that the material in this book is based partly on a workshop I held at the University of Texas at Arlington for the Tex-Fab event February, 2010.

Rajaa Issa

Robert McNeel & Associates

Table of Contents

1	Vector Mathematics	1
	Vector representation.....	1
	Vector operations.....	3
	Vector equation of line	13
	Vector equation of a plane	14
2	Matrices and Transformations	16
	Introduction.....	16
	Matrix multiplication	16
	Affine transformations	17
3	Parametric Curves and Surfaces	22
	Introduction.....	22
	Cubic polynomial curves	22
	Geometric continuity	25
	Curvature.....	26
	Algorithms for evaluating parametric curves	28
	NURBS curves.....	31
	Characteristics of NURBS curves.....	33
	NURBS surfaces.....	36
	Characteristics of NURBS surfaces.....	37
	Polysurfaces	39
	References	42

1 Vector Mathematics

Vector representation

Vectors indicate a quantity that has "direction" and "magnitude" such as velocity or force. Vectors in 2D coordinate systems are represented with two real numbers in the form:

$$\mathbf{v} = \langle a_1, a_2 \rangle$$

Similarly, in 3-D coordinate system, vectors are represented by three real numbers and would look like:

$$\mathbf{v} = \langle a_1, a_2, a_3 \rangle$$

We will use a lower case bold letters to represent vectors. Also vector components are enclosed by angle brackets. Points will use upper case letters. Points' coordinates will always be enclosed by round brackets.

Using a coordinate system and any set of anchor points in that system, we can represent or visualize these vectors using a line-segment representation. We usually put an arrowhead to show the direction of vectors.

For example, if we have a vector that has a direction parallel to the x-axis of a given 3-D coordinate system and a magnitude equal to 5.18 units, then we can write the vector as follows:

$$\mathbf{v} = \langle 5.18, 0, 0 \rangle$$

To represent that vector, we need an anchor point in the coordinate system. For example, all of the red line segments in the following figure are equal representations of the same vector.

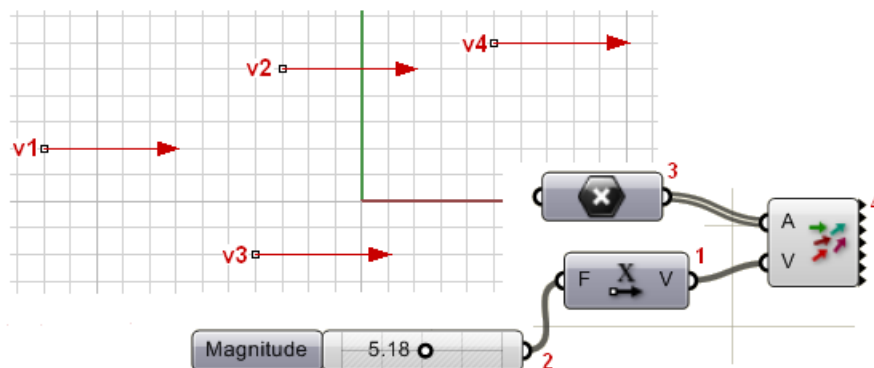


Figure (1): Vector representation in the 3D coordinate system. **1:** Grasshopper unit x-axis component. **2:** Grasshopper number slider component. **3:** Grasshopper point components that is set to reference multiple points in Rhino (in this case referencing **v1**, **v2**, **v3** and **v4**). **4:** Grasshopper vector display component

Given a 3-D vector $\mathbf{v} = \langle a_1, a_2, a_3 \rangle$, all vector components a_1, a_2, a_3 are real numbers. Also ALL line segments from a point $A(x,y,z)$ to point $B(x+a_1, y+a_2, z+a_3)$ are EQUIVALENT representation of vector \mathbf{v} .

So, how do we define the end points of a line segment that represent a given vector?

Let us define an anchor point (P_0) using Grasshopper "x,y,z point" component:

$$P_0 = (1,2,3)$$

And a vector using Grasshopper xyz vector component that takes as an input three real numbers:

$$\mathbf{v} = \langle 2, 2, 2 \rangle$$

The tip point (P1) of the vector is calculated by adding the corresponding components from anchor point and vector \mathbf{v} :

$$P1 = (1+2, 2+2, 3+2) = (3, 4, 5)$$

The following definition displays this vector using the Grasshopper vector display component, and marks the end of the displayed vector that expectantly coincides with point P1:

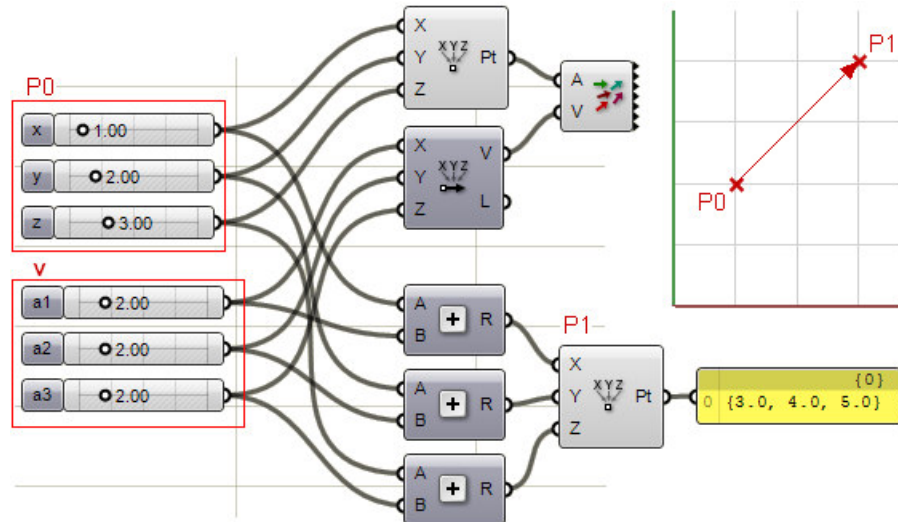


Figure (2): Relationship between a vector, vector anchor point and the point coinciding with vector tip location

Position vector

There is one special vector representation that uses the origin P0 (0,0,0) as the vector anchor point. The position vector $\mathbf{v} = \langle a1, a2, a3 \rangle$ is represented with a line segment between two points P0 and P1 so that:

$$P0 = (0, 0, 0)$$

$$P1 = (a1, a2, a3)$$

A position vector for a given vector $\mathbf{v} = \langle a1, a2, a3 \rangle$ is a special line segment representation from the origin_point(0,0,0) to point (a1, a2, a3).

It is very important not to confuse vectors with points that have equivalent components. They are two very different concepts. In the following Grasshopper definition, point P1 coordinates are equal to vector components.

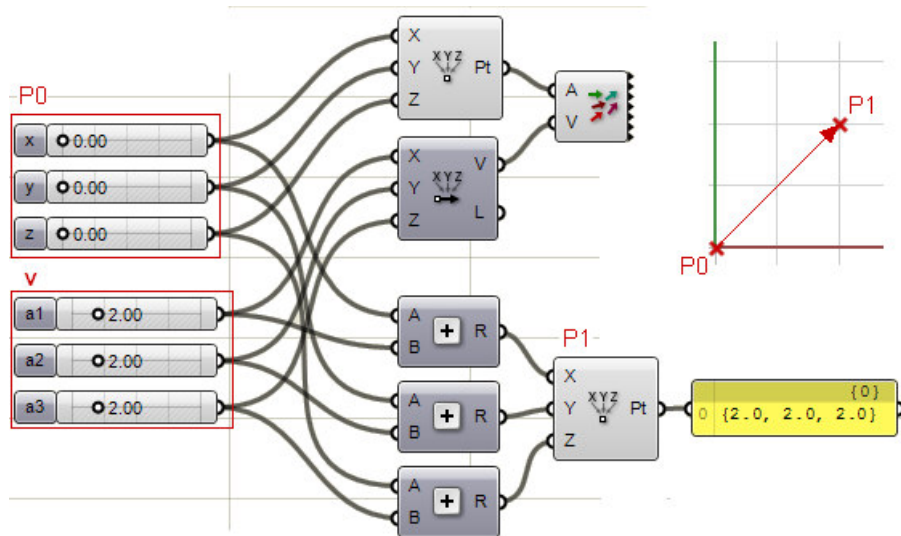


Figure (3): Position vector

Vector operations

Vector addition

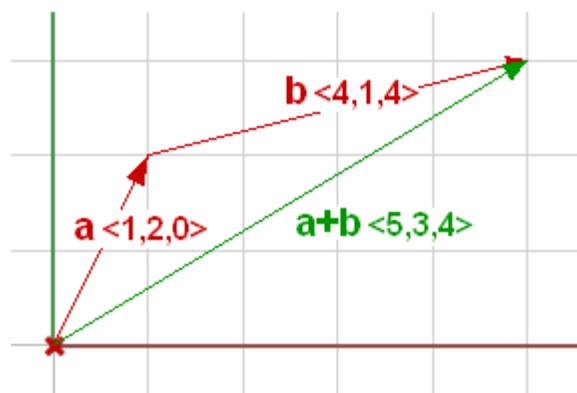
We add vectors by adding corresponding components. That is, if we have two vectors, **a** and **b**, the sum **a+b** is a vector that is calculated as follows:

$$\mathbf{a} = \langle a_1, a_2, a_3 \rangle$$

$$\mathbf{b} = \langle b_1, b_2, b_3 \rangle$$

$$\mathbf{a} + \mathbf{b} = \langle a_1 + b_1, a_2 + b_2, a_3 + b_3 \rangle$$

For example, if we have **a** <1, 2, 0> and **b** <4, 1, 4> the sum **a+b** <5, 3, 4> is shown in the following:



The following Grasshopper definition shows how to create the **a+b** vector by adding corresponding components of the two input vectors **a** and **b**.

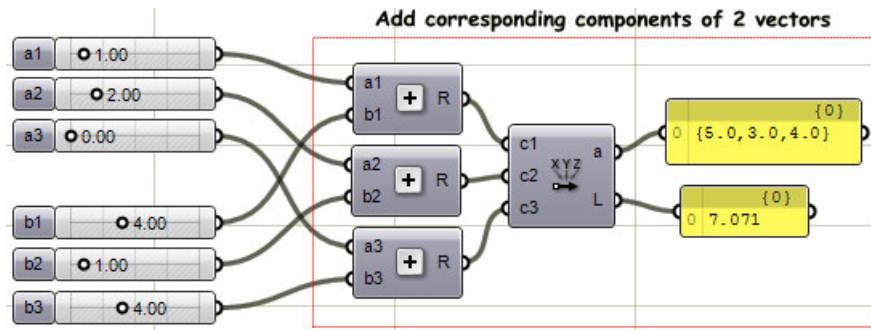


Figure (4): Adding vectors through adding their corresponding components

The resulting vector is the same as that resulting from using Grasshopper's built-in addition component:

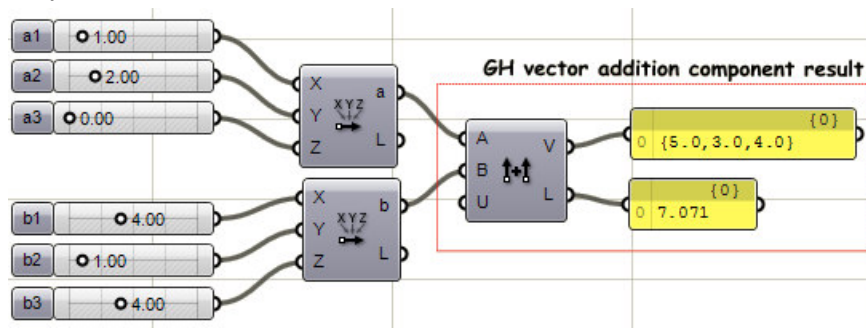


Figure (5): Adding vectors using GH vector addition component

Add two vectors by adding their corresponding components.

Vector addition is also useful for finding the average direction of multiple vectors. In this case, we usually use same-length vectors. Here is an example that shows the difference between using same-length vectors and different-length vectors on the resulting vector addition:

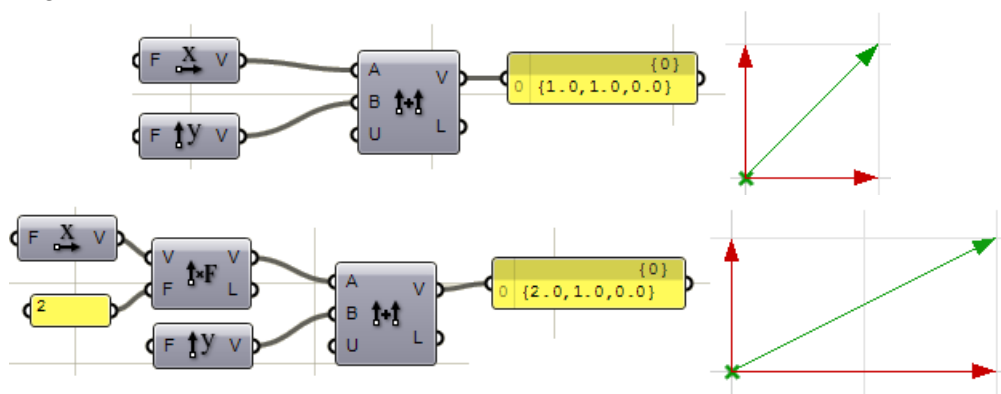


Figure (6): Adding vectors to find average direction

Input vectors are not likely to be same length. In order to find average direction, you need to use the "unit vector" of input vectors. As we will see later, a unit vector is a vector of that has a magnitude equal to one. Here is an example that solves adding vectors of different lengths to find average directions in Grasshopper.

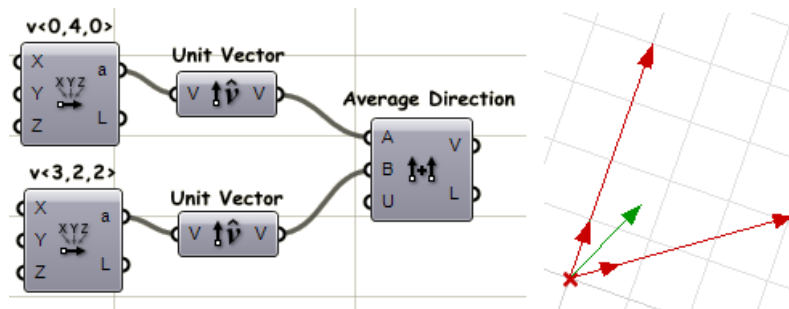


Figure (7): Use unit vectors to find average direction of two or more vectors

Vector length

We will use $|a|$ to notate the length of a given vector " a ". The **magnitude** or **length** of a vector $a = <a_1, a_2, a_3>$ is calculated by

$$|a| = \sqrt{a_1^2 + a_2^2 + a_3^2}$$

Here is an example of calculating vector magnitude using Grasshopper function component:

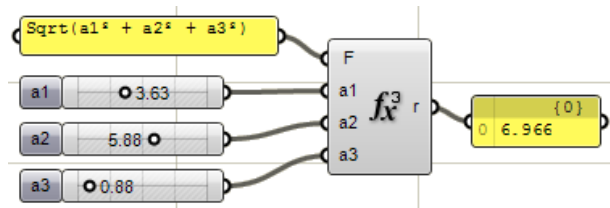


Figure (8): Calculate vector length

Note that Grasshopper vector component has an output "L" that represents the vector magnitude. Using the same vector in the above example, you'll notice that length is equal.

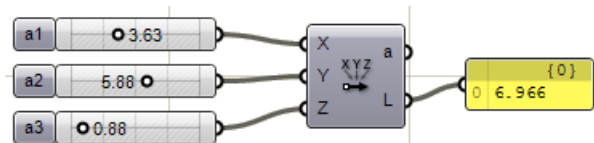


Figure (9): Vector length as a parameter is GH vector component

Vector scalar operation

Given vector $a = <a_1, a_2, a_3>$, and factor $t = \text{some real number}$,

$$a * t = <a_1 * t, a_2 * t, a_3 * t>$$

Here is the equation implemented in Grasshopper:

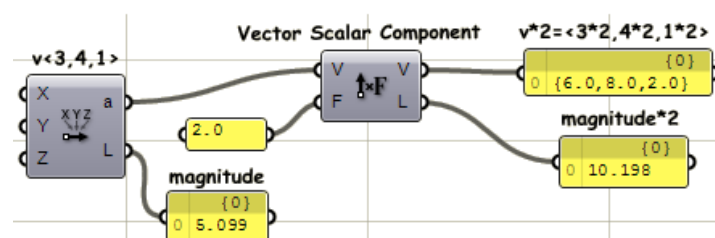


Figure (10): Vector scalar operation

Unit vector

A unit vector is a vector with a magnitude equal to one unit. Unit vectors are commonly used to compare directions of vectors.

A vector is called a unit vector when its length or magnitude is equal to one unit.

Vector properties

There are eight properties of vectors. If **a**, **b** and **c** are vectors and **s** and **t** are scalar, then:

1. $\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}$
2. $\mathbf{a} + 0 = \mathbf{a}$
3. $s(\mathbf{a} + \mathbf{b}) = s\mathbf{a} + s\mathbf{b}$
4. $st(\mathbf{a}) = s(t\mathbf{a})$
5. $\mathbf{a} + (\mathbf{b} + \mathbf{c}) = (\mathbf{a} + \mathbf{b}) + \mathbf{c}$
6. $\mathbf{a} + (-\mathbf{a}) = 0$
7. $(s + t)\mathbf{a} = s\mathbf{a} + t\mathbf{a}$
8. $1 * \mathbf{a} = \mathbf{a}$

Vector dot product

The dot product of two vectors is defined as follows:

Given:

vector **a** = $\langle a_1, a_2, a_3 \rangle$

vector **b** = $\langle b_1, b_2, b_3 \rangle$

$\mathbf{a} \cdot \mathbf{b} = a_1 * b_1 + a_2 * b_2 + a_3 * b_3$

In the following illustrations, we will show that Grasshopper vector dot product component yields the same result as the above **a.b** equation:

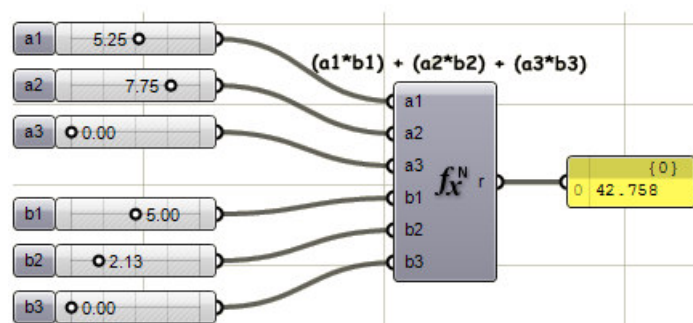


Figure (11): The dot product of two vectors as a sum of multiplying corresponding components

Grasshopper has a built-in vector dot product component as shown in the following illustration:

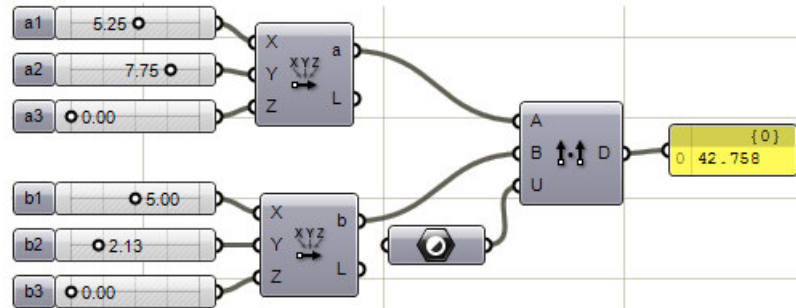


Figure (12): The dot product of two vectors using GH vector dot product component

When calculating the dot product of two unit vectors, the result is always between -1 and +1.

The dot product of a vector with itself is that vector's length to the power of two:

$$\mathbf{a} \cdot \mathbf{a} = |\mathbf{a}|^2$$

Proof:

If vector $\mathbf{a} = \langle a_1, a_2, a_3 \rangle$ then from the definition of dot product of two vectors:

$$\mathbf{a} \cdot \mathbf{a} = a_1 \cdot a_1 + a_2 \cdot a_2 + a_3 \cdot a_3$$

or

$$\mathbf{a} \cdot \mathbf{a} = a_1^2 + a_2^2 + a_3^2$$

Since we know that:

$$|\mathbf{a}| = \sqrt{a_1^2 + a_2^2 + a_3^2}$$

Therefore,

$$\mathbf{a} \cdot \mathbf{a} = |\mathbf{a}|^2$$

Here is a Grasshopper sample to demonstrate this property comparing result using dot product component with multiplying the vector length by itself:

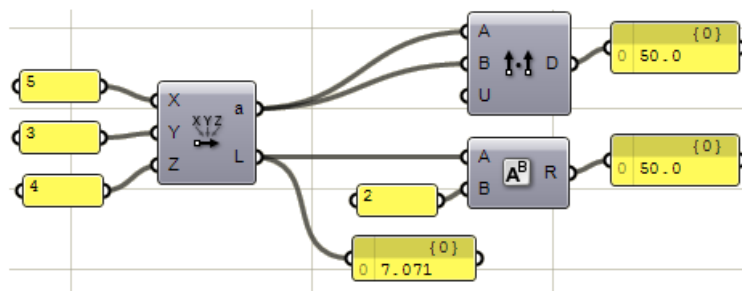


Figure (13): The dot product of a vector with itself

Dot product and angle between vectors

One important theorem for vector dot product is:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos(\theta), \text{ or}$$

$$\cos(\theta) = \mathbf{a} \cdot \mathbf{b} / (|\mathbf{a}| |\mathbf{b}|),$$

where " θ " is the angle included between the position vectors.

And if vectors \mathbf{a} and \mathbf{b} are unit vectors, we can simply say:

$$\cos(\theta) = \mathbf{a} \cdot \mathbf{b}$$

The dot product of two unit vectors equals the cosine of the angle between them

Proof:

From the law of cosines on triangle ABC

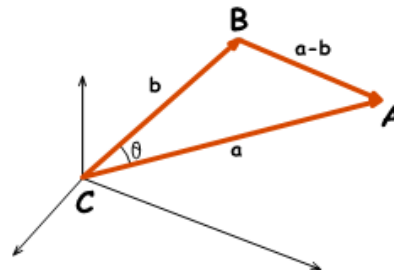
$$|AB|^2 = |CA|^2 + |CB|^2 - 2|CA||CB|\cos(\theta)$$

or:

$$|\mathbf{a}-\mathbf{b}|^2 = |\mathbf{a}|^2 + |\mathbf{b}|^2 - 2|\mathbf{a}||\mathbf{b}|\cos(\theta) \quad \text{--- (1)}$$

$|AB|^2$ is the same as $|\mathbf{a}-\mathbf{b}|^2$, so we can say:

$$\begin{aligned} |\mathbf{a}-\mathbf{b}|^2 &= (\mathbf{a}-\mathbf{b}) \cdot (\mathbf{a}-\mathbf{b}) \\ &= \mathbf{a} \cdot \mathbf{a} - \mathbf{a} \cdot \mathbf{b} - \mathbf{b} \cdot \mathbf{a} + \mathbf{b} \cdot \mathbf{b} \\ &= |\mathbf{a}|^2 - 2\mathbf{a} \cdot \mathbf{b} + |\mathbf{b}|^2 \quad \text{--- (2)} \end{aligned}$$



from (1) & (2)

$$|\mathbf{a}|^2 - 2\mathbf{a} \cdot \mathbf{b} + |\mathbf{b}|^2 = |\mathbf{a}|^2 + |\mathbf{b}|^2 - 2|\mathbf{a}||\mathbf{b}|\cos(\theta)$$

then:

$$2\mathbf{a} \cdot \mathbf{b} = 2|\mathbf{a}||\mathbf{b}|\cos(\theta)$$

or:

$$\cos(\theta) = \mathbf{a} \cdot \mathbf{b} / (|\mathbf{a}||\mathbf{b}|)$$

Vectors \mathbf{a} and \mathbf{b} are orthogonal if, and only if, $\mathbf{a} \cdot \mathbf{b} = 0$.

But what is the dot product of two unit vectors if they are parallel?

In the most practical way, you can think of the dot product of two vectors to be the projection length of one vector on the other.

Here is a demonstration of this concept using Grasshopper. In the first figure, we calculate the dot product of the x-axis unit vector with an input vector “ \mathbf{v} ”. In the second figure, we project the end point of the position vector “ \mathbf{v} ” onto a line along the x-axis and calculate the distance from origin to that projection point. You’ll notice that the dot product and projection length are equal.

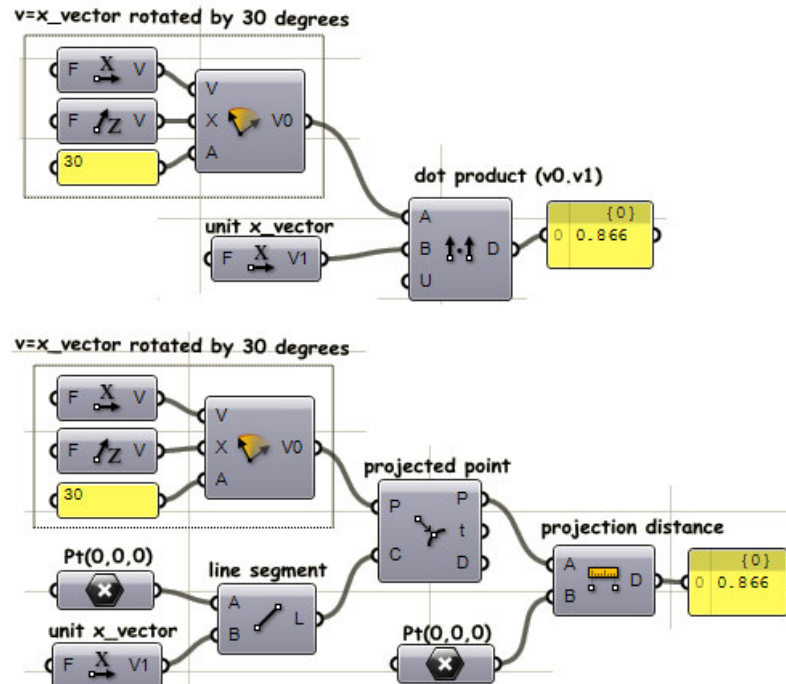


Figure (14): The dot product and angle between vectors

Dot product properties

If \mathbf{a} , \mathbf{b} and \mathbf{c} are vectors and s is scalar then:

1. $\mathbf{a} \cdot \mathbf{a} = |\mathbf{a}|^2$
2. $\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}$
3. $0 \cdot \mathbf{a} = 0$
4. $\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$
5. $(s\mathbf{a}) \cdot \mathbf{b} = s(\mathbf{a} \cdot \mathbf{b}) = \mathbf{a} \cdot (s\mathbf{b})$

Vector cross product

The cross product of two 3d-vectors produces a third 3d-vector that is orthogonal to both input vectors. Given:

$$\mathbf{a} = \langle a_1, a_2, a_3 \rangle$$

$$\mathbf{b} = \langle b_1, b_2, b_3 \rangle$$

The cross product $\mathbf{a} \times \mathbf{b}$ is defined using determinants. Here is a quick illustration of how to calculate a determinant. We use the standard basis vectors $\mathbf{i} = \langle 1, 0, 0 \rangle$, $\mathbf{j} = \langle 0, 1, 0 \rangle$ and $\mathbf{k} = \langle 0, 0, 1 \rangle$

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \begin{vmatrix} \mathbf{i} & \mathbf{j} \\ a_1 & a_2 \\ b_1 & b_2 \end{vmatrix} - \begin{vmatrix} \mathbf{i} & \mathbf{k} \\ a_1 & a_3 \\ b_1 & b_3 \end{vmatrix} + \begin{vmatrix} \mathbf{j} & \mathbf{k} \\ a_2 & a_3 \\ b_2 & b_3 \end{vmatrix}$$

$$\mathbf{a} \times \mathbf{b} = \mathbf{i}(a_2b_3 - a_3b_2) + \mathbf{j}(a_3b_1 - a_1b_3) + \mathbf{k}(a_1b_2 - a_2b_1)$$

$$\mathbf{a} \times \mathbf{b} = \langle a_2b_3 - a_3b_2, a_3b_1 - a_1b_3, a_1b_2 - a_2b_1 \rangle$$

This is the Grasshopper definition for evaluating the cross product using these expressions and comparing it with the vector cross product built-in component. They both yield same result.

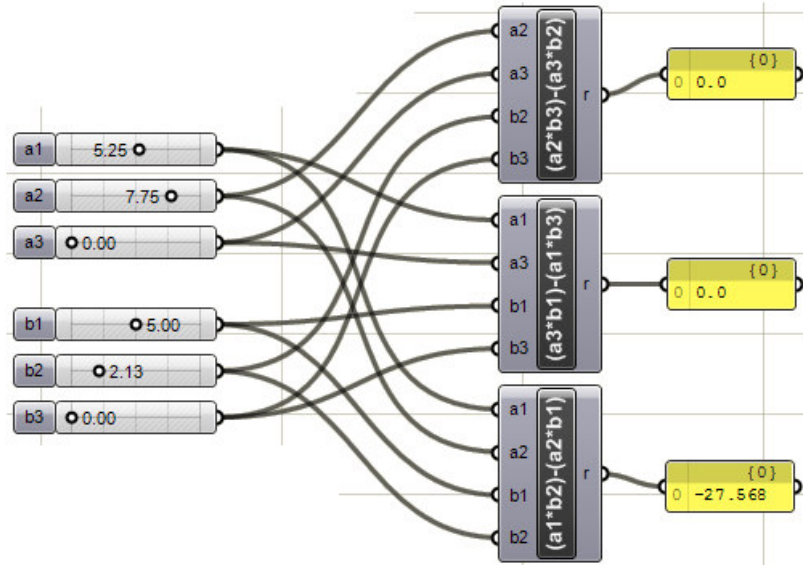


Figure (15): Calculating the cross product of two vectors

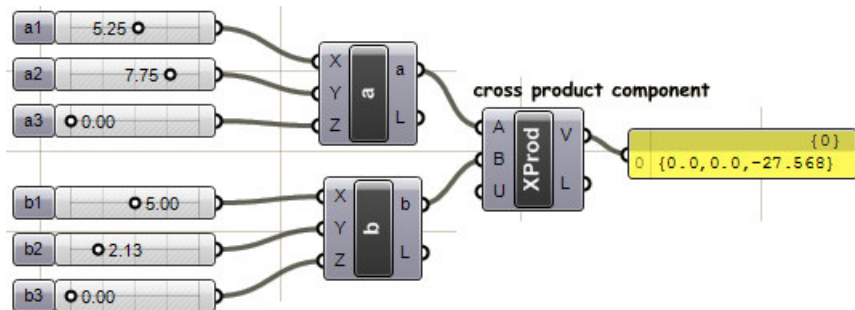


Figure (16): Calculating the cross product of two vectors using GH cross product component

The vector $\mathbf{a} \times \mathbf{b}$ is orthogonal to both \mathbf{a} and \mathbf{b}

Theorem

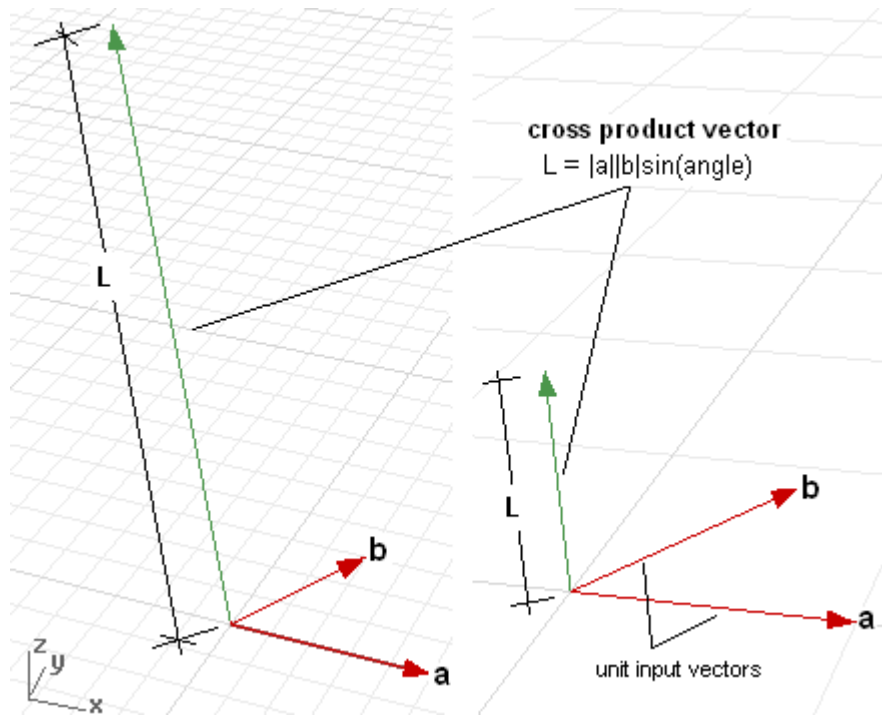
For any pair of 3d vectors \mathbf{a} and \mathbf{b}

$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}|\sin(\theta),$$

Where " θ " is the angle included between the position vectors of \mathbf{a} and \mathbf{b} .

Or if \mathbf{a} and \mathbf{b} are unit vectors, then the length of their cross product equals the sine of the angle between them. In other words:

$$|\mathbf{a} \times \mathbf{b}| = \sin(\theta)$$



This is an example to calculate the length of the cross product of two vectors using GH built-in cross product component and compare it to the calculation using the equation mentioned above. As expected, they both yield same result.

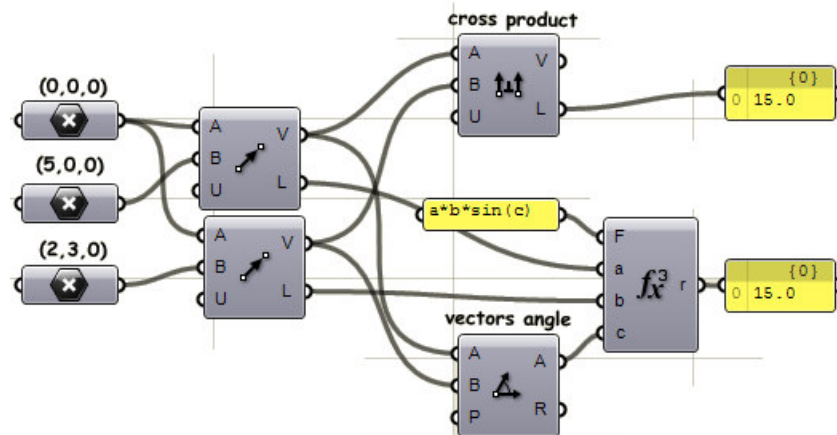


Figure (17): Calculating the length of cross product vector using a function and also using GH built-in component for vectors cross product

In determining the cross product, the order of operands is important. For example:

$$\begin{aligned} \mathbf{a} &= \langle 1, 0, 0 \rangle \\ \mathbf{b} &= \langle 0, 1, 0 \rangle \\ \mathbf{a} \times \mathbf{b} &= \langle 0, 0, 1 \rangle \\ \mathbf{b} \times \mathbf{a} &= \langle 0, 0, -1 \rangle \end{aligned}$$

In Rhino's right-handed system, the direction of $\mathbf{a} \times \mathbf{b}$ is given by the right-hand rule (where \mathbf{a} = index finger, \mathbf{b} = middle finger, and **result** = thumb).

Vectors **a** and **b** are parallel if, and only if, $\mathbf{a} \times \mathbf{b} = 0$

Cross product properties

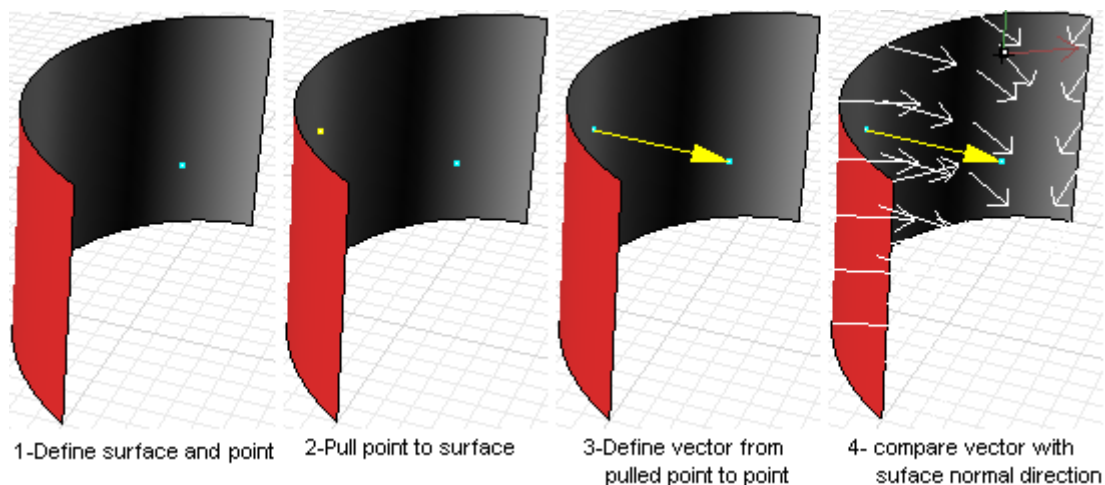
If **a**, **b**, and **c** are vectors and **s** is scalar then:

1. $\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$
2. $(s\mathbf{a}) \times \mathbf{b} = s(\mathbf{a} \times \mathbf{b}) = \mathbf{a} \times (s\mathbf{b})$
3. $\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$
4. $(\mathbf{a} + \mathbf{b}) \times \mathbf{c} = \mathbf{a} \times \mathbf{c} + \mathbf{b} \times \mathbf{c}$
5. $\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c}$
6. $\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \cdot \mathbf{c})\mathbf{b} - (\mathbf{a} \cdot \mathbf{b})\mathbf{c}$

Example

All the concepts we reviewed so far have direct application to solving geometry problems encountered when modeling. For example, given a point and a surface, how can we determine whether the point is facing the front or back of that surface?

Here are the steps to solve the problem:



Here is the Grasshopper solution following the same steps. Note that in this case the dot product is greater than 0 which means the point is facing the front side of the surface. If the dot product were less than 0 then the point would be on the back.

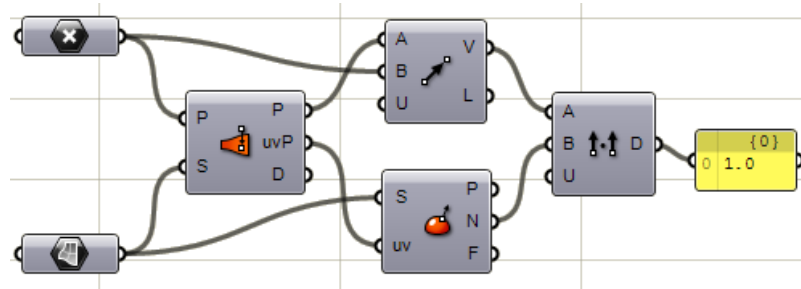
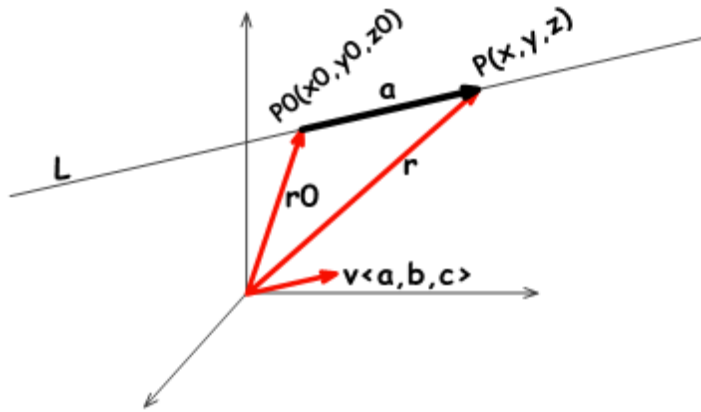


Figure (18): Find point location relative to surface front and back faces

Vector equation of line

The vector line equation is used in 3-D modeling applications and computer graphics. Here is a description of that equation and how it might be used.



In the figure:

L = line

\mathbf{v} = line direction vector

$P0$ = line position point

$\mathbf{r} = \mathbf{r0} + \mathbf{a}$ --- (1)

$\mathbf{a} = t * \mathbf{v}$ --- (2)

Therefore from 1 and 2:

$\mathbf{r} = \mathbf{r0} + t * \mathbf{v}$ --- (3)

However, we can write (3) as follows:

$\langle x, y, z \rangle = \langle x0, y0, z0 \rangle + \langle ta, tb, tc \rangle$

$\langle x, y, z \rangle = \langle x0 + ta, y0 + tb, z0 + tc \rangle$

Therefore:

$x = x0 + ta$

$y = y0 + tb$

$z = z0 + tc$

Which is the same as:

$P = P0 + t\mathbf{v}$

This is a Grasshopper definition to get any point on a line:

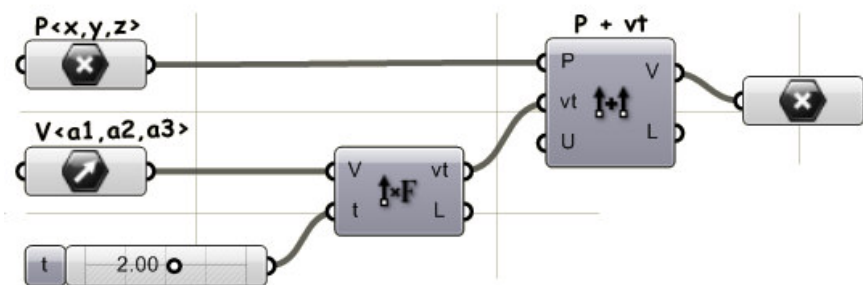
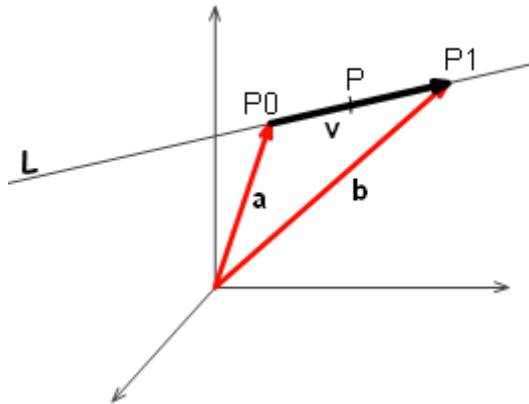


Figure (19): Find points on a line

Example

In the following figure, given points P0 and P1, find mid point P.



Notice that:

a is the position vector for point P0

b is the position vector for point P1

v is the vector going from P0 to P1

From vector addition property:

$$\mathbf{a} + \mathbf{v} = \mathbf{b}, \text{ or}$$

$$\mathbf{v} = \mathbf{b} - \mathbf{a}$$

However, the line equation is: $\mathbf{P} = \mathbf{P}_0 + t\mathbf{v}$, and since $t=0.5$ and $\mathbf{v}=\mathbf{b}-\mathbf{a}$ (from the above), then we can say:

$$\mathbf{P} = \mathbf{P}_0 + 0.5(\mathbf{b}-\mathbf{a})$$

Use the above equation to create a Grasshopper definition:

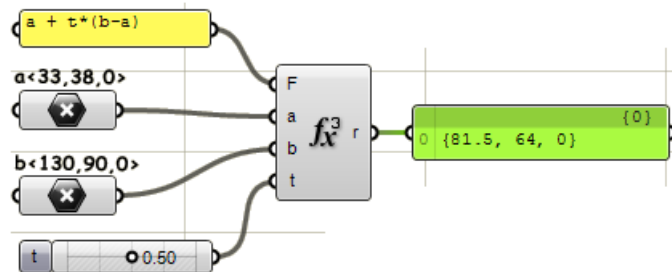


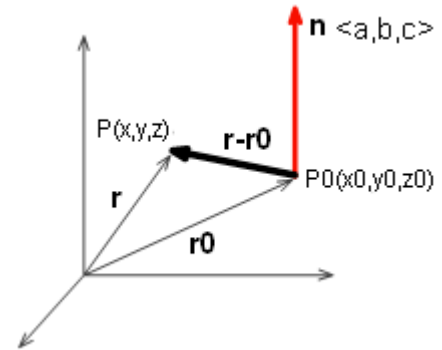
Figure (20): Find mid point between two input points

In general, you can find any point between P0 and P1 by changing the t value between 0 and 1.

Vector equation of a plane

In the figure above:

$P_0(x_0, y_0, z_0)$ = a given point on the plane
 $\mathbf{r}_0 \langle x_0, y_0, z_0 \rangle$ = position vector to P_0
 $\mathbf{n} \langle a, b, c \rangle$ = normal vector of the plane
 $P(x, y, z)$ = arbitrary point on the plane
 $\mathbf{r} \langle x, y, z \rangle$ = position vector to P



We know that the dot product of two orthogonal vectors equals 0, therefore:

$$\mathbf{n} \cdot (\mathbf{r} - \mathbf{r}_0) = 0$$

Or we can say:

$$\langle a, b, c \rangle \cdot \langle x - x_0, y - y_0, z - z_0 \rangle = 0$$

Solving the dot product gives the scalar equation of the plane:

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0$$

Example

How can we find the plane that goes through three points using the origin point and plane normal?

In order to find a plane, we need an origin and a plane normal. We have an origin point, which can be any of our three points, so how do we find the normal?

We know that the cross product of two vectors is a third vector normal to both of them. This would be the plane normal. Hence, this is how we may solve the question using Grasshopper:

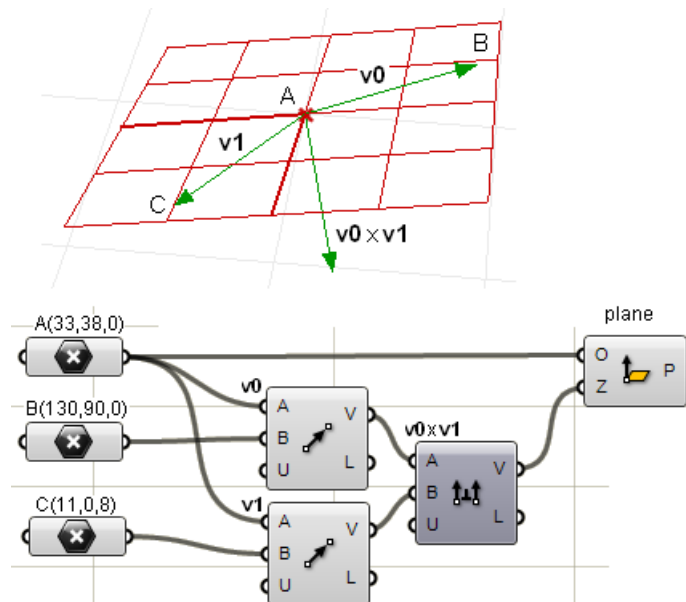


Figure (21): Find plane that goes through three points

2 Matrices and Transformations

Introduction

Although designers might not need to use matrix mathematics directly in computational design, knowledge of the basics is very useful for appreciating what is happening behind the scene. Transformation matrices are responsible for moving, rotating, projecting, and scaling objects. Matrices are also used for transformations between coordinate systems, for example from the 3-D world coordinate to the 2-D screen coordinate system.

We can define transformation as a function that takes a point (or a vector) and maps that point into another point (or vector). What is a matrix, and why do we need it for transformations?

A matrix is a rectangular array of numbers. A matrix dimension is m-by-n where:

m: number of rows

n: number of columns

So if we have a matrix **M** with two rows and three columns, we express the dimension of the matrix as follows:

$$\dim(\mathbf{M}) = [2,3]$$

Matrices have proven to be very a useful representation for transformations. Multiple transformations can be performed very quickly using this representation. The key is to find one format that can represent ALL transformations such as translation (move), rotation, and scale.

Matrix multiplication

Matrix multiplication is used to apply transformation to geometry. A series of transformation matrices is first multiplied to get a final transformation matrix that is in turn used for transforming geometry. Matrix multiplication is one of the frequently used matrix operations, so it is useful to elaborate on.

In order to multiply two matrices, they have to matching dimensions. In other words, the number of columns of the first matrix must equal the number of rows of the second matrix. The resulting matrix has size equal to the number of rows from the first matrix and the number of columns from the second matrix. For example, if we have two matrices, **M1** and **M2**, with dimension equal to [2x4] and [4x5] respectively, then there resulting multiplication matrix **M1.M2** would have a dimension equal to [2x5] as shown in the following:

$$\begin{bmatrix} + & + & + & + \\ 1 & 2 & 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} + & + & a & + & + \\ + & + & b & + & + \\ + & + & c & + & + \\ + & + & d & + & + \end{bmatrix} = \begin{bmatrix} + & + & + & + & + \\ + & + & R_{2,3} & + & + \end{bmatrix}$$

$\dim(\mathbf{M1}) = [2 \times 4] \quad \dim(\mathbf{M2}) = [4 \times 5] \quad \dim(\mathbf{M1.M2}) = [2 \times 5]$

$$R_{2,3} = 1 \cdot a + 2 \cdot b + 3 \cdot c + 4 \cdot d$$

Here are the general steps to multiply two matrices:

1. Make sure they match.
That is, given two matrices of size $\dim(\mathbf{M}_1)=[a \times b]$, $\dim(\mathbf{M}_2)=[c \times d]$, \mathbf{b} must be equal to \mathbf{c} .
2. Find the sum of multiplying corresponding items from the first row of the left matrix with the first column of the right matrix to get the item at index(1,1) of the resulting matrix.
3. Repeat step 2 to get all items of the resulting matrix.
For example the sum of multiplying third row of left matrix with second column of right matrix yields item at index (3,2) in the resulting matrix.

One special matrix is the identity matrix. The main property of this matrix is that if it is multiplied by any other matrix, it does not change its values as in the following:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*2+0*3+0*1+0*1 \\ 0*2+1*3+0*1+0*1 \\ 0*2+0*3+1*1+0*1 \\ 0*2+0*3+0*1+1*1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 1 \end{bmatrix}$$

Affine transformations

In this section, we will cover a special, but very common, type of transformation called an *affine transformation*. When applied to geometry affine transformations have the property of preserving parallel line relationships, but not length or angles. Translation (move), rotation, scale, and shear are affine transformations.

Translation (move) transformation

Moving a point from a starting position by certain vector is calculated as follows:

$$\mathbf{P}' = \mathbf{P} + \mathbf{V}$$

Suppose:

$\mathbf{P}(x,y,z)$ was a given point

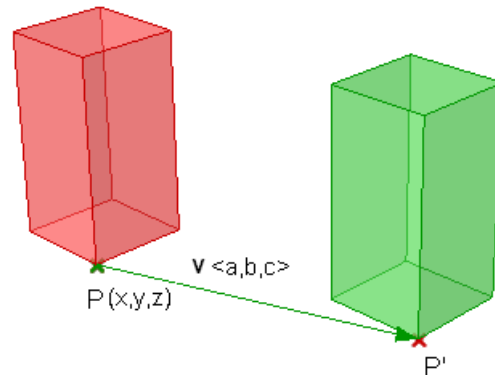
$\mathbf{v} \langle a,b,c \rangle$ was a translation vector

then:

$$\mathbf{P}'(x) = x + a$$

$$\mathbf{P}'(y) = y + b$$

$$\mathbf{P}'(z) = z + c$$



We represent a 3d-point as a 4x1 column matrix with a 1 in the last row. This “trick” allows us to represent translation, and in fact any affine transformation, by matrix multiplication.

The general format for a translation matrix is:

$$\begin{bmatrix} 1 & 0 & 0 & a1 \\ 0 & 1 & 0 & a2 \\ 0 & 0 & 1 & a3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For example, to move point $P(2,3,1)$ by vector $\mathbf{v}\langle 2,2,2\rangle$, the new point location is:

$$\mathbf{P}' = \mathbf{P} + \mathbf{v} = (2+2, 3+2, 1+2) = (4, 5, 3)$$

If we use the matrix form and multiply the translation matrix by the input point, then we get the new point location as in the following:

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*2+0*3+0*1+2*1 \\ 0*2+1*3+0*1+2*1 \\ 0*2+0*3+1*1+2*1 \\ 0*2+0*3+0*1+1*1 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ 3 \\ 1 \end{bmatrix}$$

Rotation transformation

This example shows how to calculate rotation around z-axis and origin point using trigonometry, and then deduce the general matrix format for the rotation.

Take a point on x,y plane $P(x,y)$ and rotate it by angle (b) . From the figure, we can say the following:

$$x = d \cos(a) \quad \text{---(1)}$$

$$y = d \sin(a) \quad \text{---(2)}$$

$$x' = d \cos(b+a) \quad \text{---(3)}$$

$$y' = d \sin(b+a) \quad \text{--- (4)}$$

Expanding 3 and 4 using trigonometric identities for the sine and cosine of the sum of angles:

$$x' = d \cos(a)\cos(b) - d \sin(a)\sin(b) \quad \text{---(5)}$$

$$y' = d \cos(a)\sin(b) + d \sin(a)\cos(b) \quad \text{---(6)}$$

Using Eq 1 and 2:

$$x' = x \cos(b) - y \sin(b)$$

$$y' = x \sin(b) + y \cos(b)$$

Using the homogenous coordinates, the rotation matrix around **z-axis** looks like:

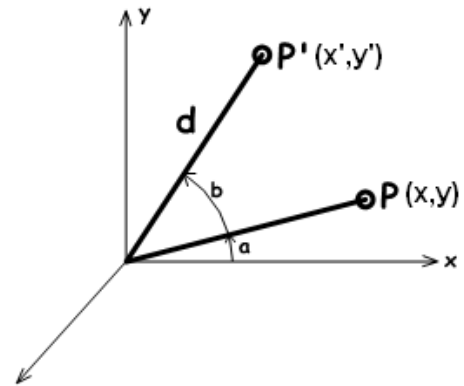
$$\begin{bmatrix} \cos(b) & -\sin(b) & 0 & 0 \\ \sin(b) & \cos(b) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rotation matrix around **x-axis** by angle **b**:

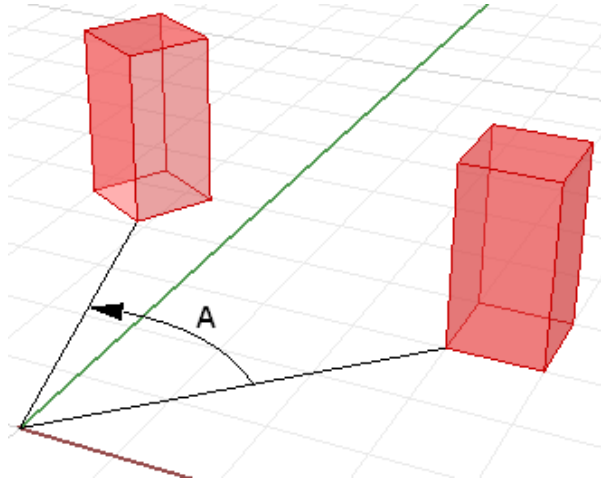
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(b) & -\sin(b) & 0 \\ 0 & \sin(b) & \cos(b) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rotation matrix around **y-axis** by angle **b**:

$$\begin{bmatrix} \cos(b) & 0 & \sin(b) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(b) & 0 & \cos(b) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



OpenNURBS™, the Rhino geometry library (<http://www.openNURBS.org>), contains a class called OnXform that handles transformations. It stores a transformation matrix and performs matrix operations. The following is an example of how to rotate an object and examines OnXform matrix values to compare to the general format of the rotation matrix. You can use the same principle to examine other transformations.



Here is a Grasshopper definition for rotating geometry and output matrix values to compare with the general matrix format:

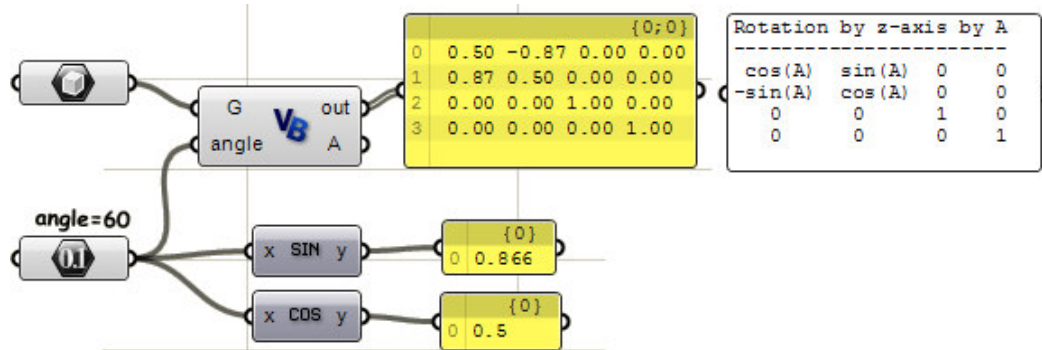


Figure (22): Rotate geometry and print the transformation matrix

Scale transformation

We know that:

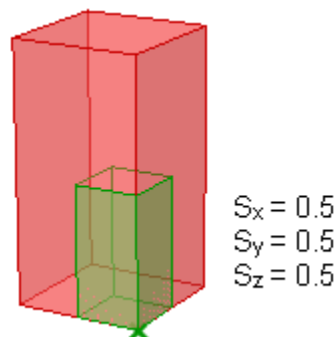
$$P' = \text{ScaleFactor}(S) * P$$

or:

$$P'.x = S_x * P.x$$

$$P'.y = S_y * P.y$$

$$P'.z = S_z * P.z$$



This is the matrix format for

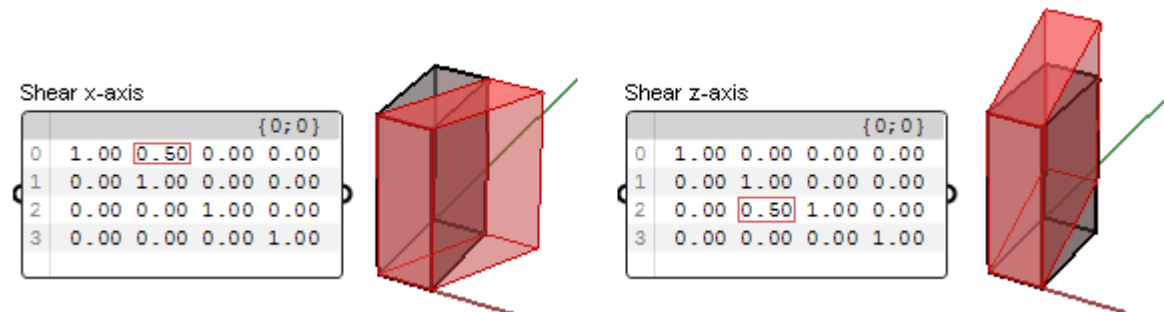
$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

scale transformation.

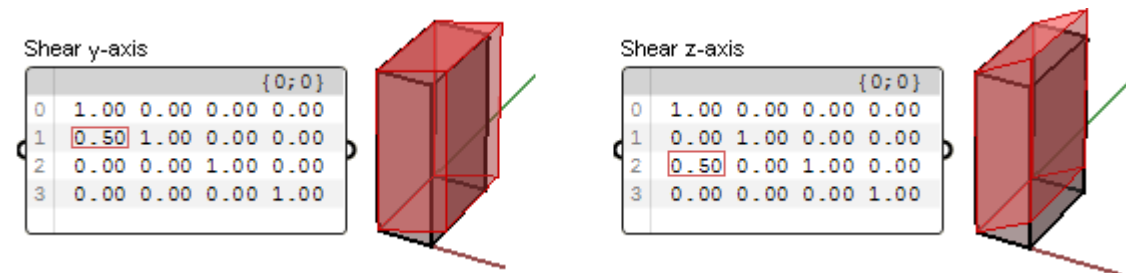
Shear transformation

Shear in 3-D is measured along a pair of axes relative to the third axis. For example, a shear along a z-axis will not change geometry along that axis, but will alter x and y.

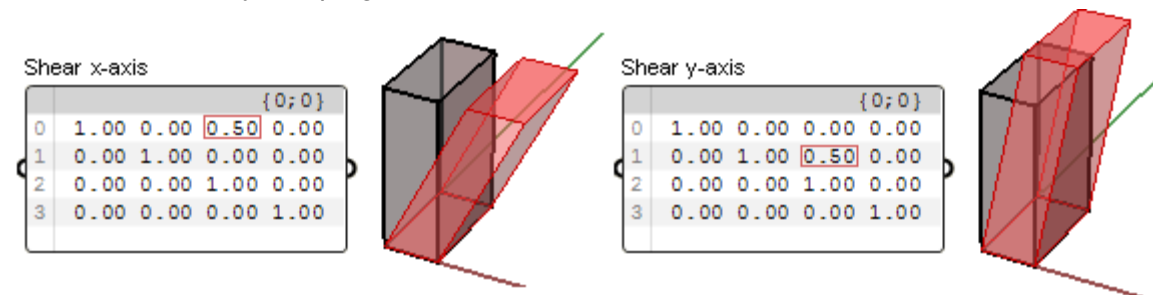
1- Shear in x and z, keeping the **y-coordinate** fixed:



2- Shear in y and z, keeping the **x-coordinate** fixed:



3- Shear in x and y, keeping the **z-coordinate** fixed:



Here is a GH definition to change different values in the transformation matrix:

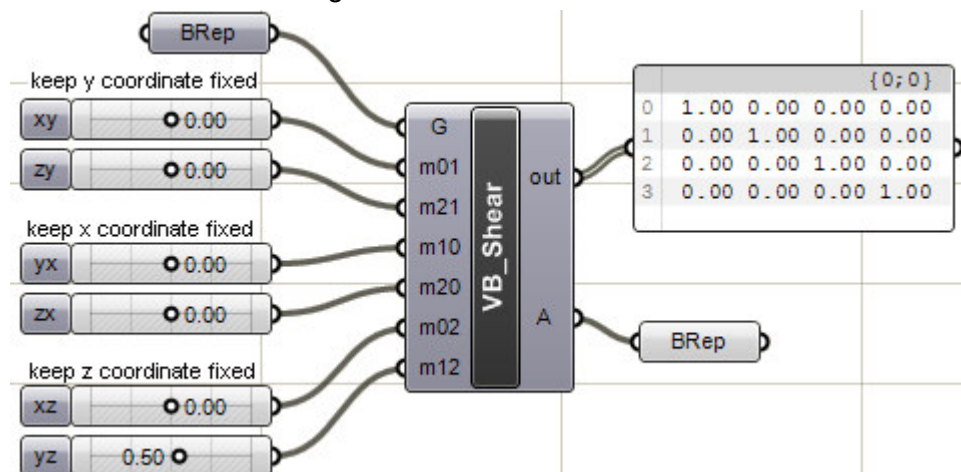
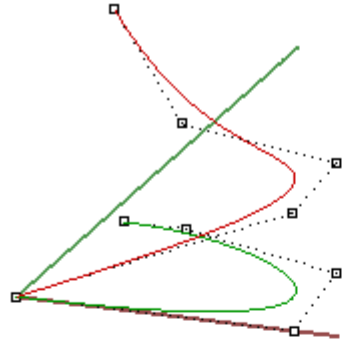


Figure (23): Shear Matrix

Planar Projection transformation

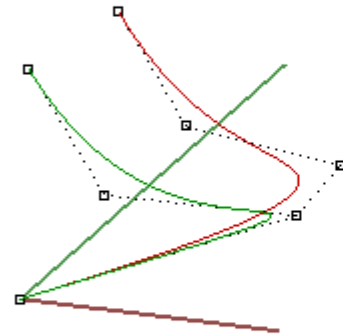
Intuitively, the projection point of a given 3D point $P(x,y,z)$ on the world xy -plane equals $P_{xy}(x,y,0)$. Similarly, a projection to xz -plane of point P is $P_{xz}(x,0,z)$. When projecting to yz -plane, $P_{yz} = (0,y,z)$. Those are called orthogonal projections¹.

So if we have a curve as an input and we apply the planar projection transformation, then we get a projected curve to that plane. The following shows an example of projected curve to xy -plane with the matrix format. Note that NURBS curves (explained in the next chapter) use control points to define curves. Projecting a curve amounts to projecting its control points.



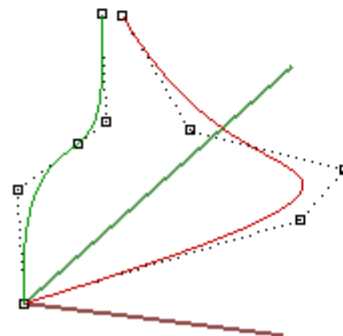
xy projection

0	1.00	0.00	0.00	0.00
1	0.00	1.00	0.00	0.00
2	0.00	0.00	0.00	0.00
3	0.00	0.00	0.00	1.00



xz projection

0	1.00	0.00	0.00	0.00
1	0.00	0.00	0.00	0.00
2	0.00	0.00	1.00	0.00
3	0.00	0.00	0.00	1.00



yz projection

0	0.00	0.00	0.00	0.00
1	0.00	1.00	0.00	0.00
2	0.00	0.00	1.00	0.00
3	0.00	0.00	0.00	1.00

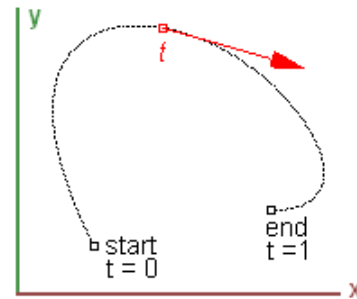
¹ For more details see: [http://en.wikipedia.org/wiki/Projection_\(linear_algebra\)](http://en.wikipedia.org/wiki/Projection_(linear_algebra))

3 Parametric Curves and Surfaces

Introduction

Parametric curves are very compact and an intuitive way to represent smooth curves. They also are very easy to modify compared to other representation formats. For example, polylines use first-degree piecewise approximation and therefore use a large number of points to store a curve that is somewhat smooth. In addition, curve manipulation is very tedious, especially if the smoothness of the curve needs to be maintained. The higher the accuracy of the curve, the heavier the curve storage grows and the more difficult to it is edit.

A parametric curve is a function of one independent parameter (usually denoted t)² over some domain (usually between 0 and 1). Consider for example the path traced out by a traveler. The domain is the time between the trip start and the trip end. The parametric representation gives the travelers location at any time in addition to the locations that he traveler passed through.



Let's take for example a circle. You probably remember the circle equation to be:

$$x^2 + y^2 = r^2$$

The parametric equation of the circle is defined using parameter " t " as follows:

$$x = r \cos(t)$$

$$y = r \sin(t)$$

Just to show that the two represent same curve, we can derive the original equation for the parametric one as follows:

$$x/r = \cos(t)$$

$$y/r = \sin(t)$$

And since:

$$\cos(t)^2 + \sin(t)^2 = 1 \text{ (Pythagorean identity)}$$

Then:

$$(x/r)^2 + (y/r)^2 = 1, \text{ or } x^2 + y^2 = r^2$$

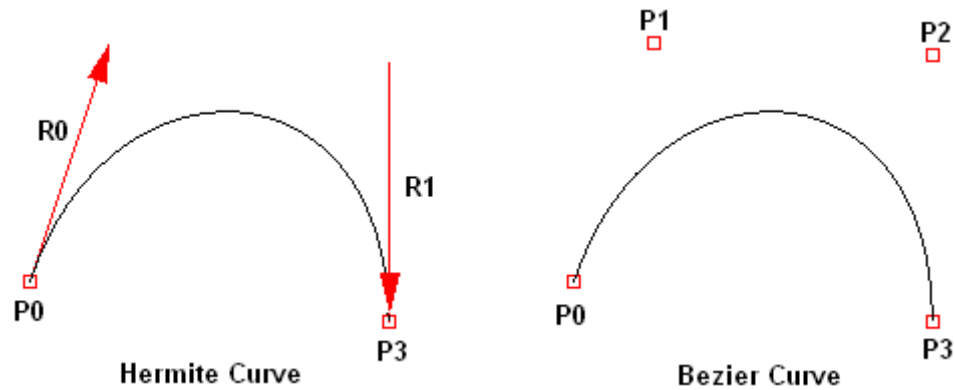
Cubic polynomial curves

Hermite³ and Bézier⁴ curves are two examples of cubic polynomial curves that are determined by four parameters. A Hermite curve is determined by two end points and two tangent vectors at these points, while a Bézier curve is defined by four points. While they differ mathematically, they share similar characteristics and limitations.

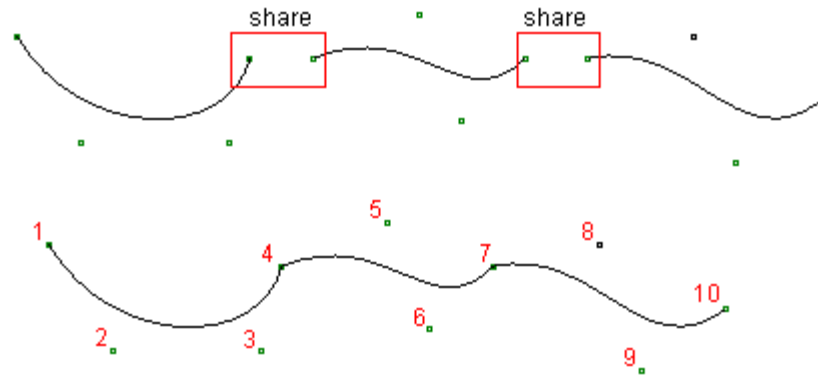
² For more details see: http://en.wikipedia.org/wiki/Parametric_equation

³ For more details see: http://en.wikipedia.org/wiki/Cubic_Hermite_spline

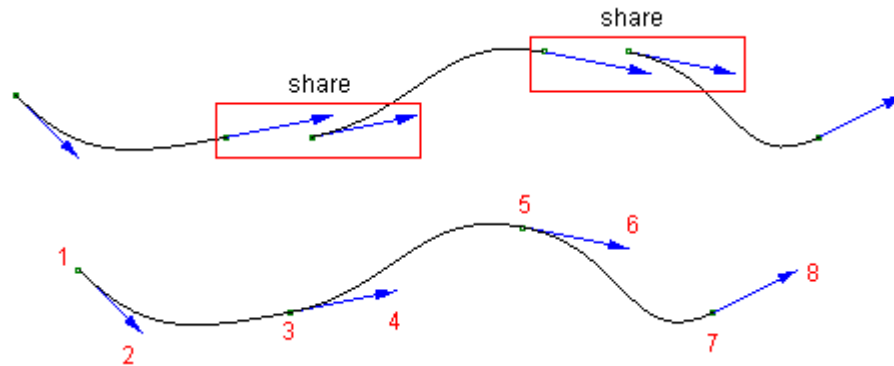
⁴ For more details, see: http://en.wikipedia.org/wiki/B%C3%A9zier_curve



In most cases, curves are made out of multiple segments. This requires making what is called piecewise cubic curves. Here is an illustration of a piecewise Bezier curve that uses 10 storage points to create a three-segment curve. Note that although the final curve is joined, it does not look smooth.

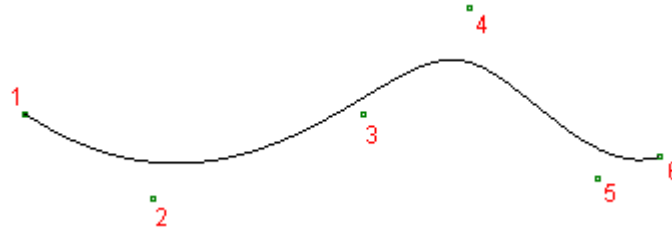


Although Hermite curves use same amount of parameters as that is of Bezier (4 to define a curve), we have additional information of the tangent curve that can be shared with the next piece to create smoother looking curve with less storage as in the following figure:



In order to maintain even smoother and more continuous curves, there is a powerful curve representation called Non Uniform Rational B-Spline⁵ (NURBS). Segments share more control points to achieve even smoother curve with less storage as in the following:

⁵ For more details, see: http://en.wikipedia.org/wiki/Non-uniform_rational_B-spline



NURBS curves and surfaces are the main mathematical representation used by Rhino to represent geometry. NURBS curves characteristics and components will be covered with some detail later in this chapter.

But how does the parametric equation of a cubic polynomial curve look like? You will likely not need to use these equations in your work, but I thought it would be useful to include the generic form so that you have a reference of it.

The parametric equation of a cubic polynomial curve segment:

$$\mathbf{Q}(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix}$$

Takes the form:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$

$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$

$$z(t) = a_z t^3 + b_z t^2 + c_z t + d_z$$

We can rewrite the above $\mathbf{Q}(t)$ equation to be::

$$\mathbf{Q}(t) = \mathbf{C} \cdot \mathbf{T}$$

Where \mathbf{T} is:

$$\mathbf{T} = \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

And \mathbf{C} is the matrix of coefficients:

$$\mathbf{C} = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix}$$

We can quickly verify that we can get the original form of the curve equation using matrix multiplication:

$$\mathbf{Q}(t) = \mathbf{C} \cdot \mathbf{T} = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix} \cdot \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} a_x t^3 + b_x t^2 + c_x t + d_x \\ a_y t^3 + b_y t^2 + c_y t + d_y \\ a_z t^3 + b_z t^2 + c_z t + d_z \end{bmatrix} = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix}$$

Geometric continuity

Continuity is an important concept in 3-D modeling. Continuity is important for achieving visual smoothness and for obtaining smooth light and airflow.

The following table shows various continuities and their definitions:

G0 (Position continuous)	Two curve segments joined together
G1 (Tangent continuous)	Direction of tangent at joint point is the same for both curve segments
G2 (Curvature Continuous)	Curvatures as well as tangents agree for both curve segments at the common endpoint
GN	The curves agree to higher order.

The following example compares curves continuities between curve “a” from one side and curves “b”, “c”, and “d” from the other. The GH script components calculate tangent vector of each curve at point P and the length of that vector. That is:

A = Tangent vector at joint point

L = the length of the vector

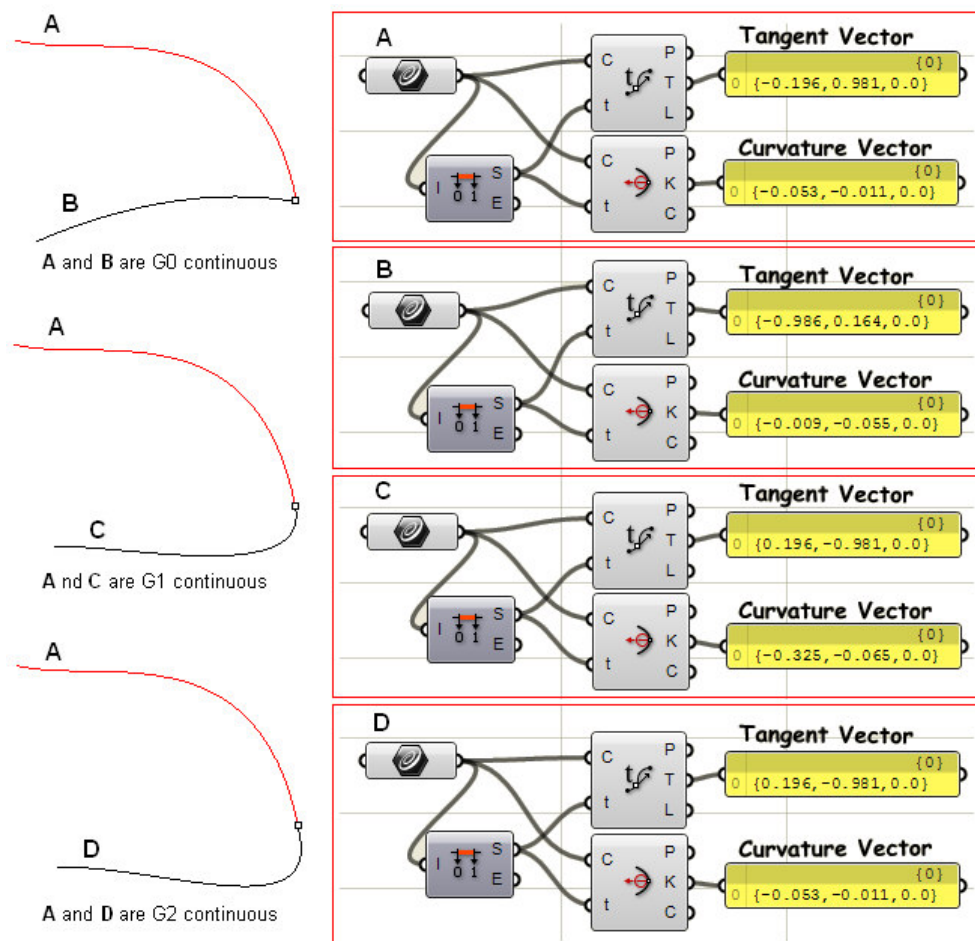


Figure (24): Examining curve continuity

Note that:

Curves **A** and **B** are **G0** continuous (different tangent vector at joint)

Curves **A** and **C** are **G1** continuous (same tangent vector at joint)

Curves **A** and **D** are **G2** continuous (G1 and curvature agrees at the joint)

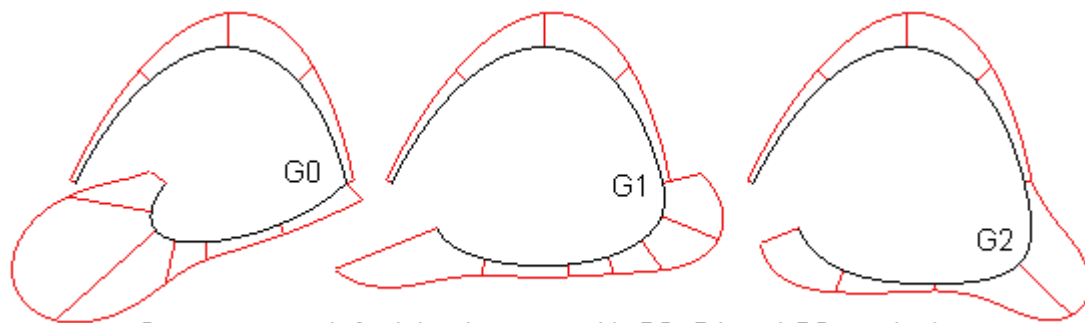
Curvature

Curvature is a widely used concept in modeling 3-D curves and surfaces. Curvature is defined as *the change in inclination of a tangent to a curve over unit length of arc. For a circle or sphere, it is the reciprocal of the radius.*

At any point on a curve in the plane, the line best approximating the curve that passes through this point is the tangent line. We can also find the best approximating circle that passes through this point and is tangent to the curve. The reciprocal of the radius of this circle is the curvature of the curve at this point.

The best approximating circle can lie either to the left or to the right of the curve. If we care about this, then we establish a convention, such as giving the curvature positive sign if the circle lies to the left and negative sign if the circle lies to the right of the curve. This is known as signed curvature.

Curvature values of joined curves indicate continuity between these curves as in the following illustration.



Curvature graph for joined curves with G0, G1 and G2 continuity

For surfaces, normal curvature is one generalization of curvature to surfaces. Given a point on the surface and a direction lying in the tangent plane of the surface at that point, the normal section curvature is computed by intersecting the surface with the plane spanned by the point, the normal to the surface at that point, and the direction. The normal section curvature is the signed curvature of this curve at the point of interest.

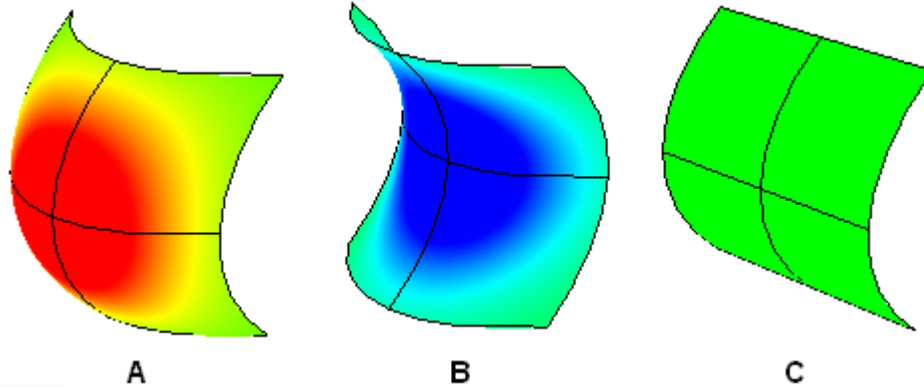
If we look at all directions in the tangent plane to the surface at our point, and we compute the normal curvature in all these directions, then there will be a maximum value and a minimum value.

Principal curvatures

The principal curvatures of a surface at a point are the minimum and maximum of the normal curvatures at that point. The principal curvatures are used to compute the *Gaussian* and *mean* curvatures of the surface.

Gaussian curvature

The Gaussian curvature of a surface at a point is the product of the principal curvatures at that point. The tangent plane of any point with positive Gaussian curvature touches the surface locally at a single point, whereas the tangent plane of any point with negative Gaussian curvature cuts the surface



A: Positive curvature when surface is bowl-like

B: Negative curvature when surface is saddle-like

C: Zero curvature when surface is flat in at least one direction (plane, cylinder, etc.)

Mean curvature

The mean curvature of a surface at a point is one half the sum of the principal curvatures at that point. Any point with zero mean curvature has negative or zero Gaussian curvature.

Surfaces with zero mean curvature everywhere are minimal surfaces. Surfaces with constant mean curvature everywhere are often referred to as constant mean curvature (CMC) surfaces.

Physical processes which can be modeled by CMC surfaces include the formation of soap bubbles, both free and attached to objects. A soap bubble, unlike a simple soap film, encloses a volume and exists in equilibrium where slightly greater pressure inside the bubble is balanced by the area-minimizing forces of the bubble itself.

Minimal surfaces are the subset of CMC surfaces where the curvature is zero everywhere.

Physical processes which can be modeled by minimal surfaces include the formation of soap films spanning fixed objects, such as wire loops. A soap film is not distorted by air pressure (which is equal on both sides) and is free to minimize its area. This contrasts with a soap bubble, which encloses a fixed quantity of air and has unequal pressures on its inside and outside. Mean curvature is useful for finding areas of abrupt change in the surface curvature.

Algorithms for evaluating parametric curves

De Casteljau⁶ algorithm for evaluating cubic Bézier curves

Named after its inventor, Paul De Casteljau, this algorithm evaluates Bézier curves using a recursive method.

We will show the algorithm to find any point on the curve at parameter t using De Casteljau algorithm using Grasshopper. We will need the following input:

4 points A, B, C, D

t , which is a parameter on the curve within curve domain (0-1)

Output:

Point on curve that is at parameter t

Solution steps:

1. Find point M at t parameter on line AB
2. Find point N at t parameter on line BC
3. Find point O at t parameter on line CD
4. Find point P at t parameter on line MN
5. Find point Q at t parameter on line NO
6. Find point R at t parameter on line PQ

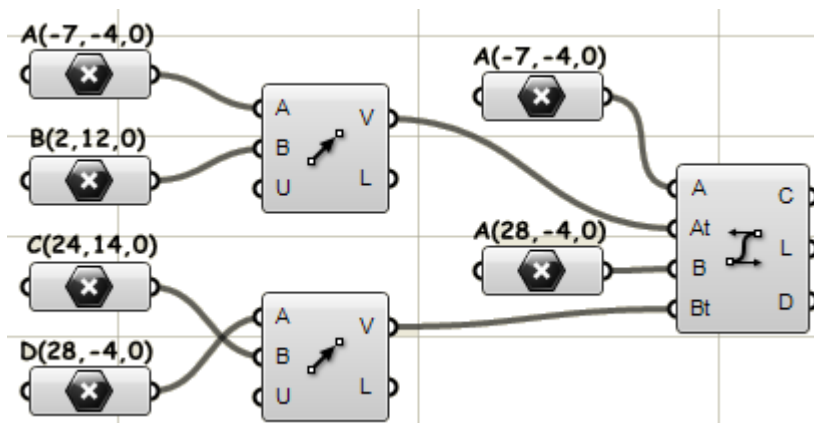
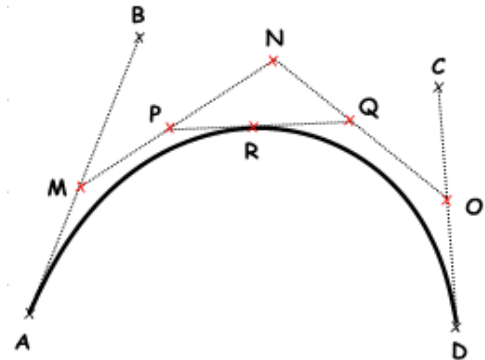


Figure (25): Create Bézier curve in GH

This is the Grasshopper definition to evaluate a parameter on a Bézier curve using De Casteljau algorithm. Note that you can change t value between 0 and 1 to find points between the start and end of the Bézier curve.

⁶ De Casteljau's algorithm details are found in http://en.wikipedia.org/wiki/De_Casteljau%27s_algorithm

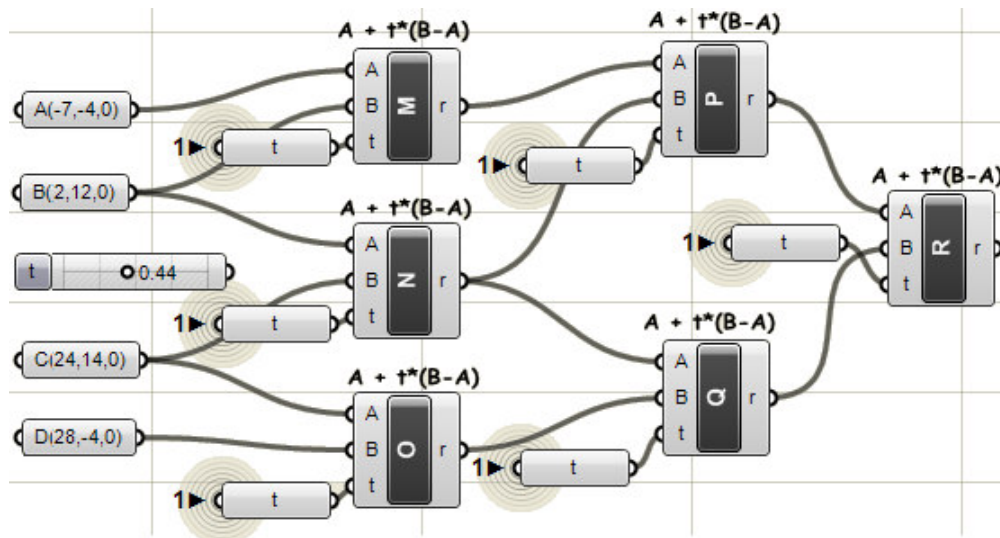


Figure (26): Evaluate points on a Bezier curve using De Casteljau algorithm

De Boor⁷ algorithm for evaluating NURBS curves

DeBoor's algorithm is a generalization of De Casteljau algorithm for Bezier curves. It is numerically stable and is widely used to evaluate a point on NURBS curves in 3D applications. The following is an example to evaluate a point on a degree 3 NURBS curve using DeBoor's algorithm⁸.

Input

7 control points P_0 to P_6

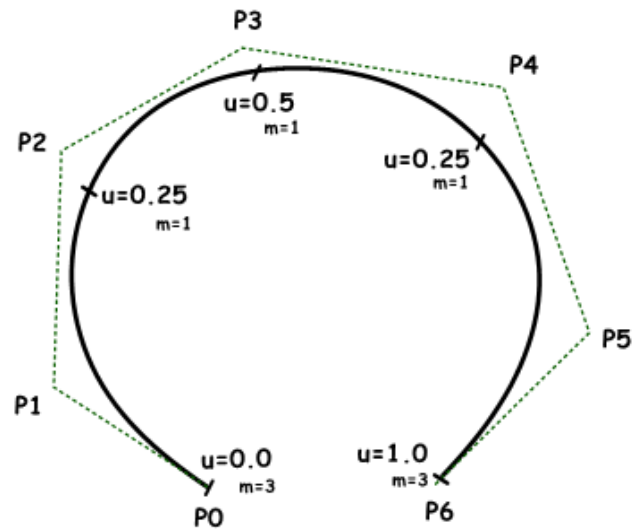
Knot vector:

$u_0 = 0.0$
 $u_1 = 0.0$
 $u_2 = 0.0$
 $u_3 = 0.25$
 $u_4 = 0.5$
 $u_5 = 0.75$
 $u_6 = 1.0$
 $u_7 = 1.0$
 $u_8 = 1.0$

Output:

Point on curve that is at $u=0.4$

Solution steps:



⁷ De Boor's algorithm details are found in http://en.wikipedia.org/wiki/De_Boor's_algorithm

⁸ The general description of the algorithm and the example details are found in: <http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/de-Boor.html>

1. Calculate coefficients for the first iteration:

$$A_c = (u - u_1) / (u_{1+3} - u_1) = 0.8$$

$$B_c = (u - u_2) / (u_{2+3} - u_2) = 0.53$$

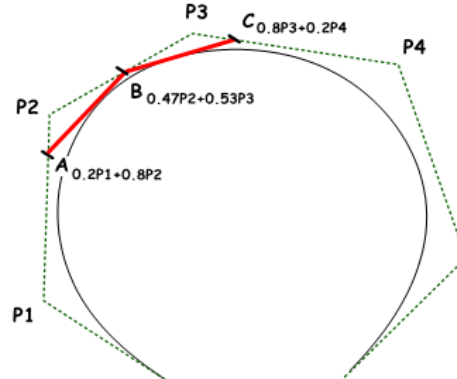
$$C_c = (u - u_3) / (u_{3+3} - u_3) = 0.2$$

2. Calculate points using coefficient data:

$$A = 0.2P_1 + 0.8P_2$$

$$B = 0.47P_2 + 0.53P_3$$

$$C = 0.8P_3 + 0.2P_4$$



3. Calculate coefficients for the second iteration:

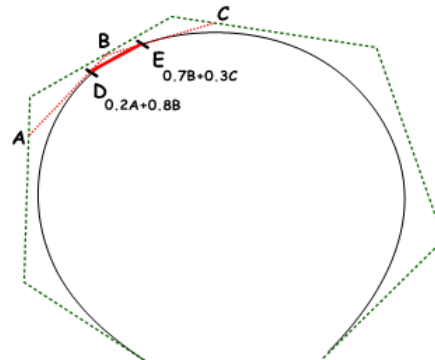
$$D_c = (u - u_2) / (u_{2+3-1} - u_2) = 0.8$$

$$E_c = (u - u_3) / (u_{3+3-1} - u_3) = 0.3$$

4. Calculate points using coefficient data:

$$D = 0.2A + 0.8B$$

$$E = 0.7B + 0.3C$$

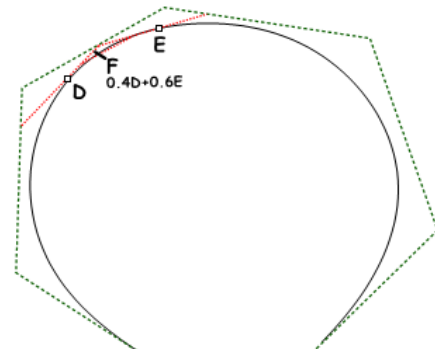


5. Calculate the last coefficient

$$F_c = (u - u_3) / (u_{3+3-2} - u_3) = 0.6$$

6. Find the point on curve at u=0.4 parameter

$$F = 0.4D + 0.6E$$



This is the Grasshopper definition to evaluate the $u=0.4$ parameter on a NURBS curve using DeBoor's algorithm.

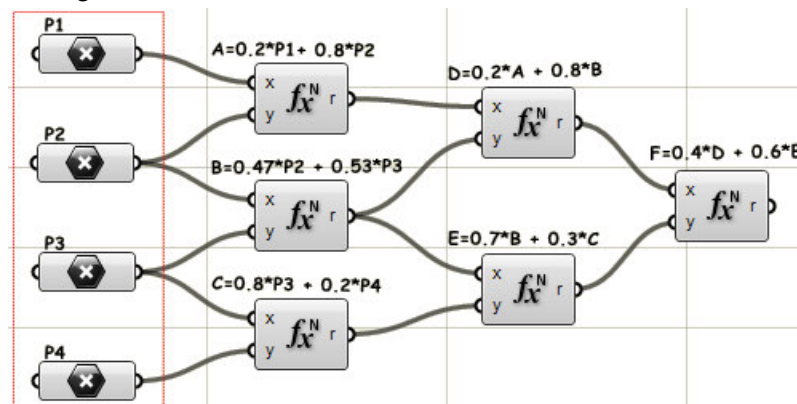


Figure (27): Evaluate a point on a NURBS curve using De Boor algorithm

NURBS curves

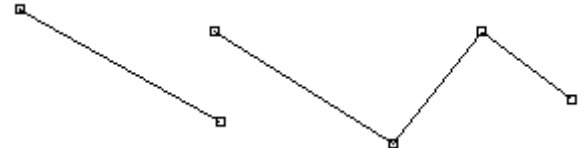
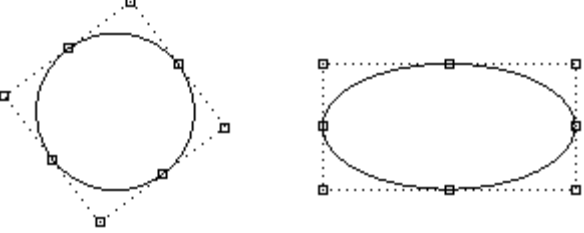
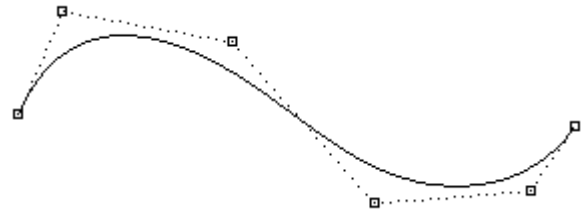
NURBS is an accurate mathematical representation of curves and surfaces that is highly intuitive to edit.

There are many books and references for those of you interested in an in-depth reading about NURBS (<http://en.wikipedia.org/wiki/NURBS>). A basic understanding of NURBS is necessary to help you use the NURBS modeler more effectively.

Four attributes define a NURBS curve: degree, control points, knots, and evaluation rules:

Degree

Degree is a whole positive number. Rhino allows working with any degree starting with 1. Degree 5 is also common, but the degrees above 5 are not very useful in the real world. Following are a few examples of curves and their degree:

<p>Lines and polylines are degree 1 NURBS curves. Order = 2 (order = degree + 1)</p>	
<p>Circles and ellipses are examples of degree 2 NURBS curves. They are also rational or non-uniform curves. Order = 3.</p>	
<p>Free-form curves are usually represented as degree 3 NURBS curves. Order = 4</p>	

Control points

The control points of a NURBS curve is a list of at least (degree+1) points. The most common way to change the shape of a NURBS curve is through moving its control points.

Control points have an associated number called a **weight**. With a few exceptions, weights are positive numbers. When a curve's control points all have the same weight (usually 1), the curve is called non-rational. We will have an example showing how to change the weights of control points interactively in Grasshopper.

Knots or knot vector

Each NURBS curve has a list of numbers associated with it that is called a knot vector. Knots are a little harder to understand and set, but luckily there are SDK functions that do the job for you. Nevertheless, there are few things that will be useful to learn about knot vectors.

Knots are parameter values

The knots are a non-decreasing list of parameter values. There are degree-1 more knots than control points. Usually, for non-periodic curves, the first degree many knots are the same and the last degree many are the same. The domain of the curve is between these extreme knot values.

Knot multiplicity

The multiplicity of a knot is the number of times it is listed in the knot vector. The multiplicity of a knot can not be more than the degree of the curve. Knot multiplicity is used to control continuity at the corresponding curve point.

Full-multiplicity knot

A fully multiple knot has multiplicity equal to the curve degree. At a fully multiple knot there is a corresponding control point, and the curve goes through this point. For example, clamped curves have knots with full multiplicity at the ends of the curve. This is why end control points coincide with curve end points. An interior fully multiple allows for a kink in the curve at the corresponding point.

Simple knot

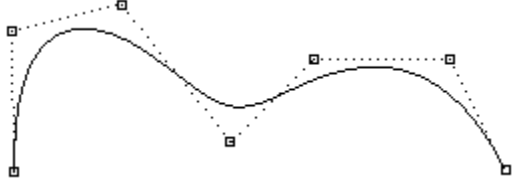
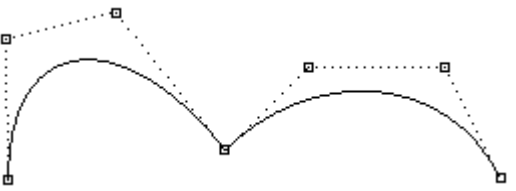
A knot with value appearing only once.

Uniform knot vector

A uniform knot vector satisfies two conditions:

1. Knots start with a full-multiplicity knot, are followed by simple knots, and terminate with a full-multiplicity knot. The values are increasing and equally spaced. This is typical of clamped curves. Periodic curves work differently as we will see later.

Here are two curves with identical control points but different knot vectors:

Degree = 3 Number of control points = 7 knot vector = (0,0,0,1,2,3,5,5,5)	
Degree = 3 Number of control points = 7 knot vector = (0,0,0,1,1,1,4,4,4) Note: Full knot multiplicity in the middle creates a kink and the curve is forced to go through the associated control point.	

Evaluation rule

The evaluation rule uses a mathematical formula that takes a number and assigns a point. The formula involves the degree, control points, and knots.

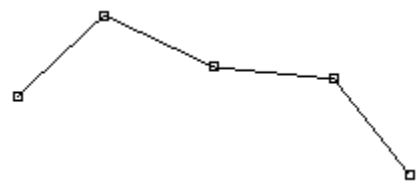
Using this formula, SDK functions can take a curve parameter and produce the corresponding point on that curve. A parameter is a number that lies within the curve domain. Domains are usually increasing and they consist of two numbers: minimum domain parameter ($m_t(0)$) that is usually the start of the curve and maximum ($m_t(1)$) at the end of the curve.

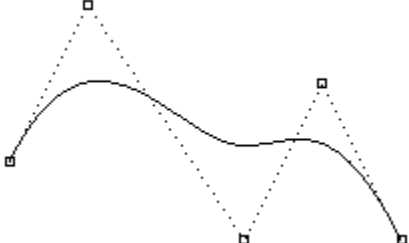
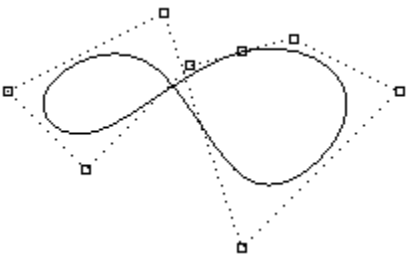
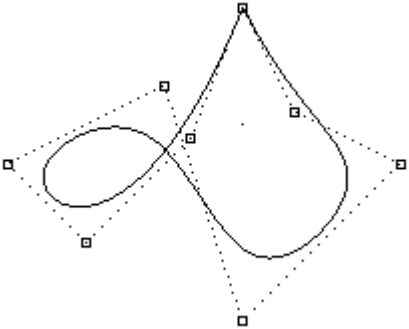
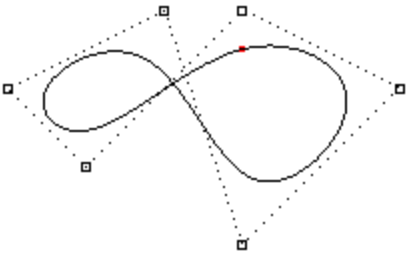
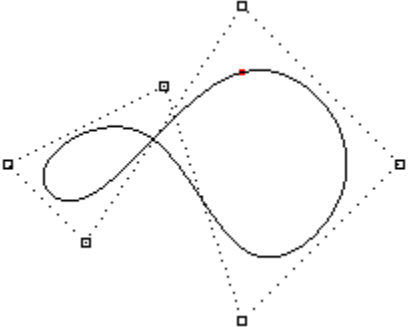
Characteristics of NURBS curves

In order to create a NURBS curve, you will need to provide the following information:

- Dimension, which is typically 3.
- Degree, (sometimes use “order” which is degree + 1)
- Control points (array of points).
- Knot vector (array of numbers).
- Specify if the curve is rational (we will explain the notion of rational curves in the weights discussion).

Using a 3D modeler, you will typically need to specify the degree of the curve and control points. The rest of the information necessary to construct a NURBS curves would be generated automatically. Selecting an end point to coincide with the start point would typically create a periodic smooth closed curve. The following figure shows open (clamped) curve, closed curve that is not periodic and closed curve that is periodic. We will discuss clamped vs periodic curves in the next section.

Degree 1 open curve. Note how the curve goes through all control points.	
---	--

<p>Degree 3 open curve. Both curve ends coninside with end control points.</p>	
<p>Degree 3 closed (non-periodic) curve. Start and end point of the curve overlap a control point.</p>	
<p>Moving the control point of a non-periodic curve causes a kink and the curve does not look smooth.</p>	
<p>Degree 3 closed periodic curve. Note that curve end/start does not go through a control point. That point is called "curve seam". It is noted in red in the figure.</p>	
<p>Moving control points of a periodic curve does not affect curve smoothness or cause any kink.</p>	

Clamped vs. periodic NURBS curves

Clamped curves are curves (usually open) where curve ends coincide with end control points. Periodic curves are smooth closed curves. The best way to understand the differences between the two is through comparing control points.

The following component creates clamped NURBS curve and prints control points and knot vector:

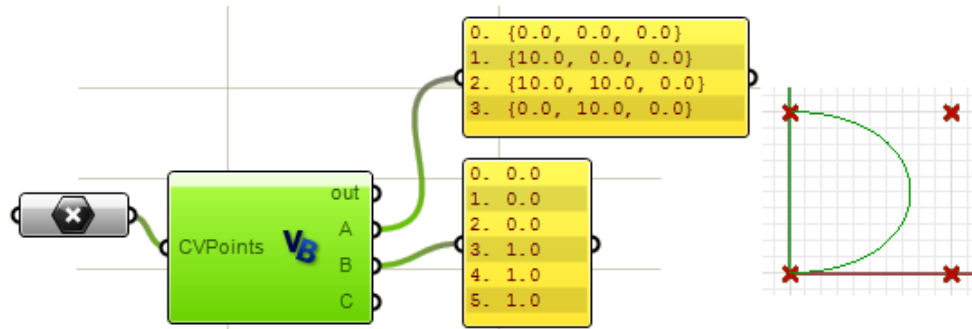


Figure (28): Analyze clamped NURBS curves

Here is the periodic curve using same input (control points and curve degree):

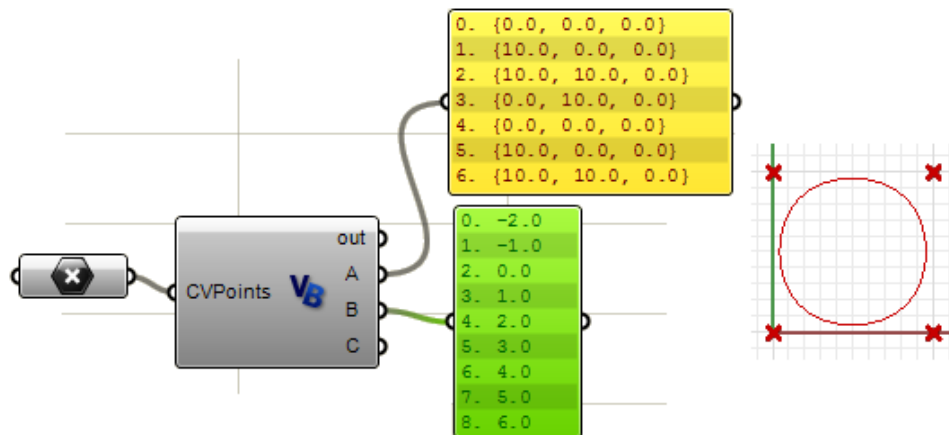


Figure (29): Analyze periodic NURBS curves

Notice that the periodic curve turned the four input points into seven control points ($4 + \text{degree}$), while the clamped curve used only four control points. The knot vector of the periodic curve uses only simple knots, while the clamped curve start and end knots have full multiplicity.

Here are some examples of degree 2 curves. As you may have guessed, the number of control points and knots of periodic curves change when degree changes.

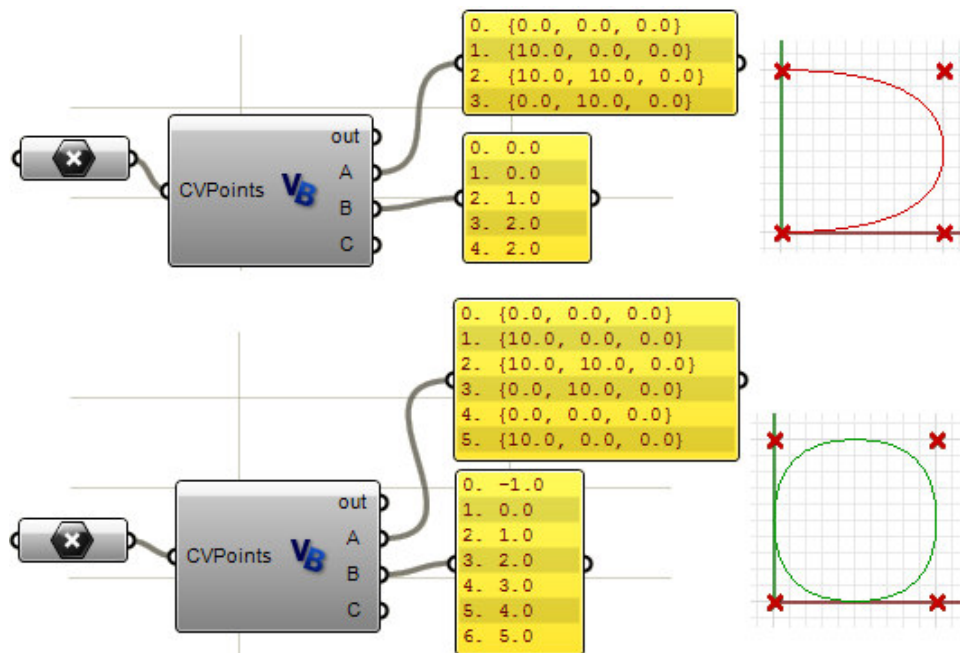


Figure (30): Analyze degree 2 NURBS curves

Weights

Weights of control points in a uniform NURBS curve are set to 1, but this number can vary in rational NURBS curves. The following example shows how to modify weights of control points interactively in Grasshopper.

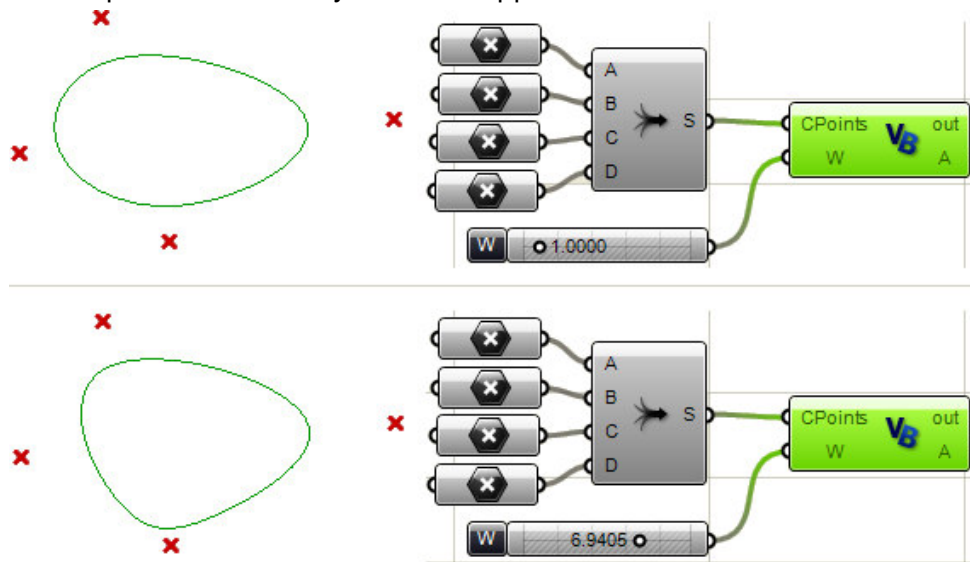


Figure (31): Analyze weights in NURBS curves

NURBS surfaces

You can think of NURBS surfaces as a grid of NURBS curves that go in two directions. The shape of a NURBS surface is defined by a number of control points and the degree of that surface in each one of the two directions (u- and v-directions).

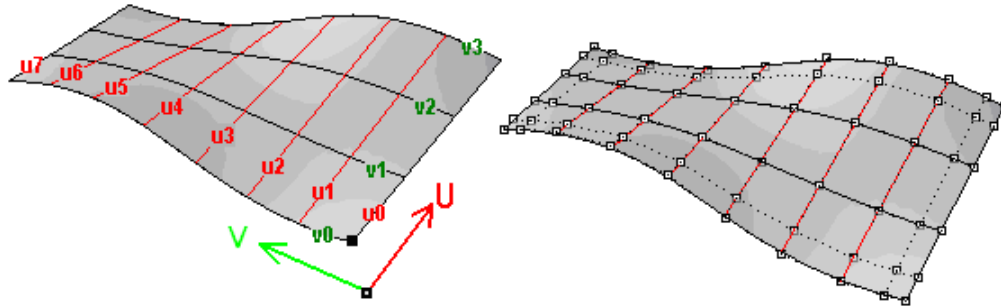


Figure (31): NURBS surface domain

NURBS surfaces can be trimmed or untrimmed. Trimmed surfaces use an underlying NURBS surface and closed curves to cut a specific shape of that surface. Each surface has one closed curve that defines the outer border (*outer loop*) and non-intersecting closed inner curves to define holes (*inner loops*). A surface with an outer loop that is the same as that of its underlying NURBS surface and that has no holes is what we refer to as an *untrimmed* surface.

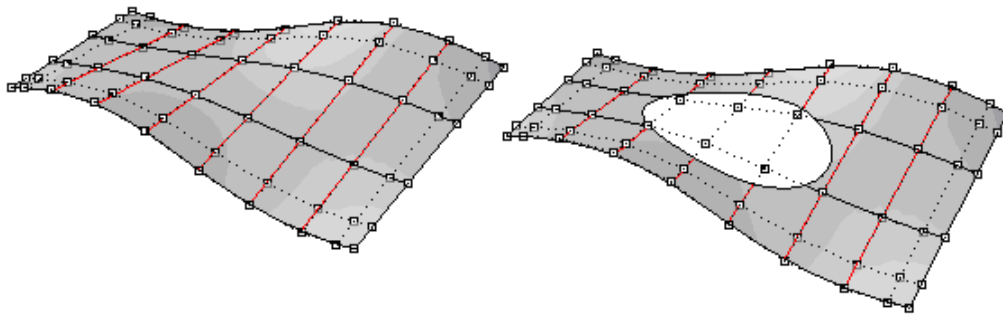


Figure (32): Trimmed NURBS surface

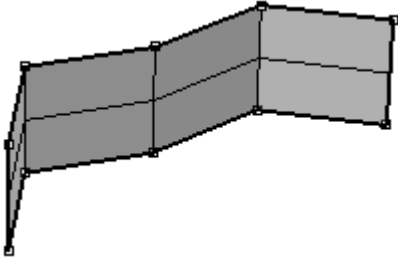
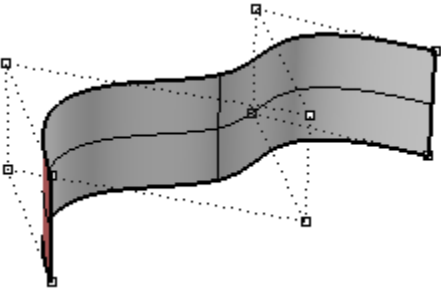
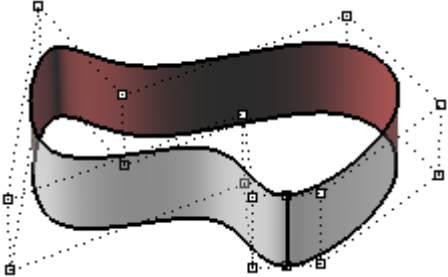
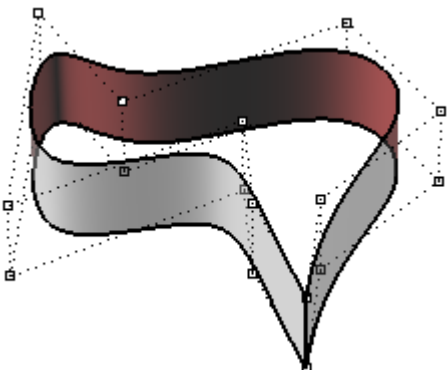
The surface on the left is untrimmed. The surface on the right is the same surface trimmed with an elliptical hole. Notice that the NURBS structure of the surface does not change when trimming.

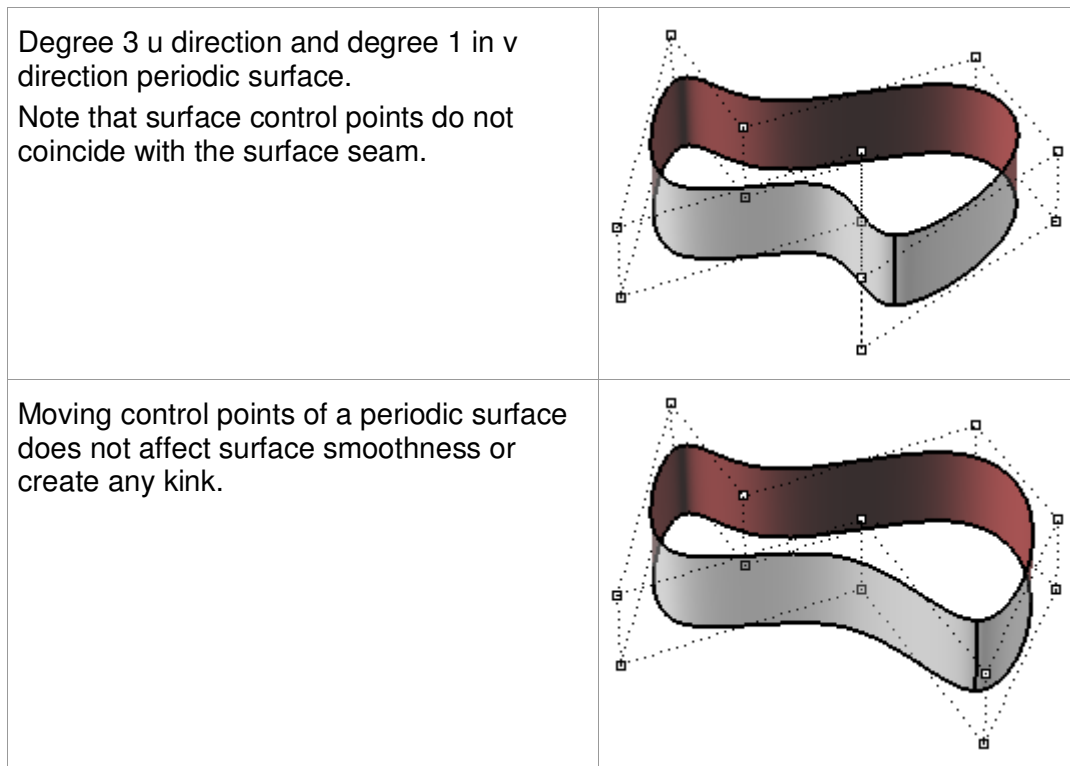
Characteristics of NURBS surfaces

NURBS surfaces characteristics are very similar to that of the NURBS curves except there is one additional direction. NURBS surfaces holds the following information:

- Dimension, which is typically 3.
- Degree in u and v directions: (sometimes use “order” which is degree + 1)
- Control points in u and v directions
- Knot vector (2D array of numbers).
- Specify if the surface is rational.

As with the NURBS curves, you will unlikely need to know the details of how to create a NURBS surface since the 3D modeler will typically provide good set of tools to help. You can always rebuild surfaces (and curves for that matter) to new degree and number of control points. Also the type of the surface can be open, closed or periodic. Here are few examples of surfaces with different degrees, number of control points and whether they are open or periodic:

<p>Degree 1 surface in both u and v directions. All control points lay on the surface</p>	
<p>Degree 3 u direction and degree 1 in v direction open surface. Note how surface corners coincide with corner control points.</p>	
<p>Degree 3 u direction and degree 1 in v direction closed (non-periodic) surface. Note that there are control points that coincide with surface seam.</p>	
<p>Moving control points of a closed (non-periodic) surface causes a kink and the surface does not look smooth.</p>	



Polysurfaces

A polysurface consists of two or more (possibly trimmed) NURBS surfaces joined together. Each surface has its own parameterization and u-,v-directions that do not have to match. Polysurfaces and trimmed surfaces are represented using what is called boundary representation (brep for short). It basically describes surface, edge, and vertex geometry with trimming data and relationships among different parts. For example, the brep describes each face, its surrounding edges and trims, normal direction relative to the surface, relationship with neighboring faces and so on. Breps can also be called *solids* when they are closed or watertight.

In the case of following box, it is made out of 6 untrimmed surfaces joined together:

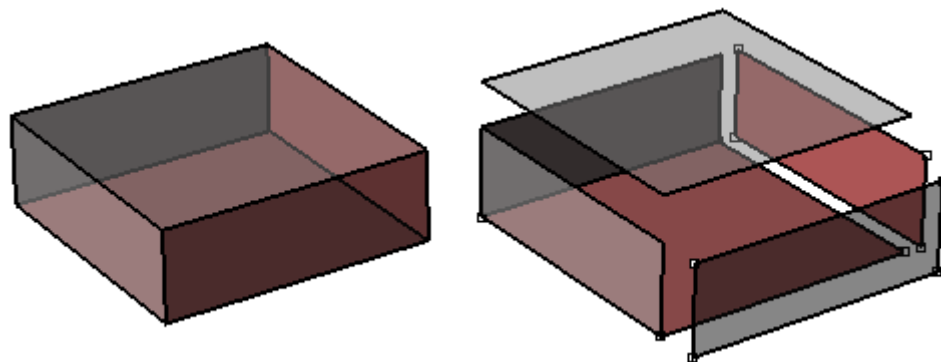


Figure (33): Polysurfaces are made out of joined NURBS surfaces

Many polysurfaces are made out of joined trimmed surfaces. The top and bottom faces of the cylinder in the following example are trimmed out of planar surfaces.

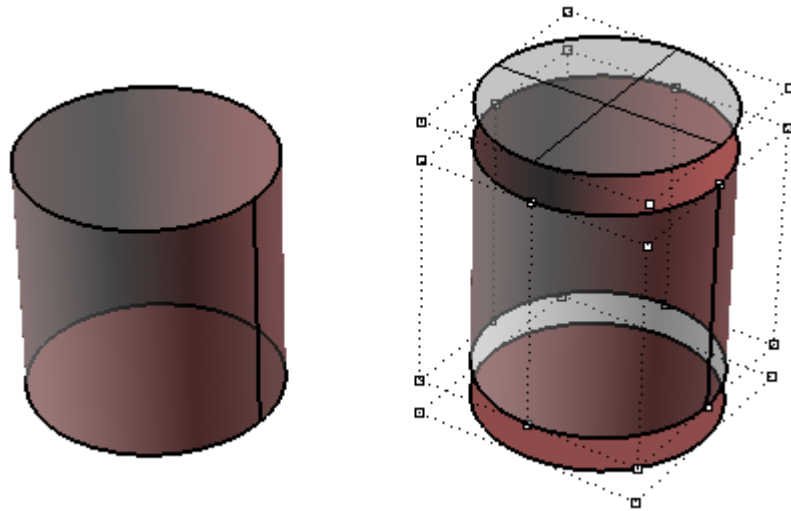
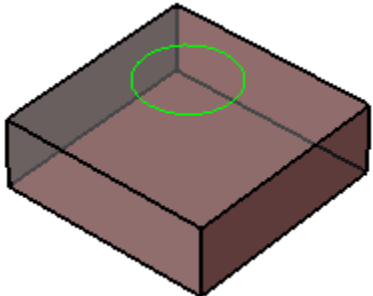
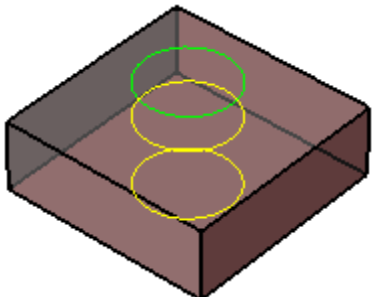


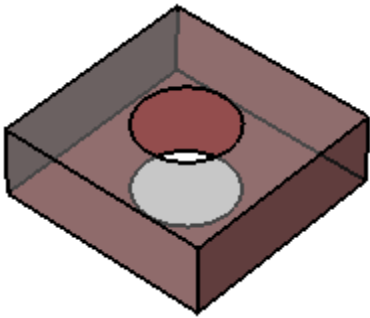

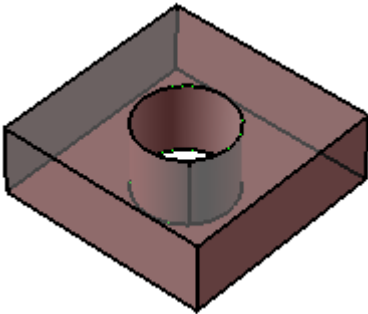
Figure (34): Polysurface faces can be trimmed NURBS surfaces

We saw that editing NURBS curves a surfaces is very intuitive and can be done interactively by moving control points. However, editing polysurfaces and maintain joined edges of different faces is not as intuitive. There are tools that are provided by the 3D NURBS modelers to edit or deform polysurfaces directly, but it is also common to explode a polysurface to its component surfaces, edit those, and then joint back together. In that process, the user must maintain that edges align correctly within tolerance to be able to join successfully.

Example

Show the steps of create a round hole in a box.

<p>Start with a box and a circle that marks the location and diameter of the hole.</p>	
<p>Project the circle to the box.</p>	

<p>Use projection circles to trim top and bottom faces of the box.</p>	
<p>Use projection circles to create rolled surface between them.</p>	
<p>Join the trimmed surfaces and the hole wall all together.</p>	

References

Edward Angel, "Interactive Computer Graphics with OpenGL," Addison Wesley Longman, Inc., 2000.

James D Foley, Steven K Feiner, John F Hughes, "Introduction to Computer Graphics" Addison-Wesley Publishing Company, Inc., 1997.

James Stewart, "Calculus," Wadsworth, Inc, 1991.

Kenneth Hoffman, Ray Kunze, "Linear Algebra", Prentice-Hall, Inc., 1971

Rhinoceros® help document, Robert McNeel and Associates, 2009.