Autodesk University
“ICE: Design Tools”

Todd Akita, Psyop
September 2011

How to use this material

Building from the ground up

For many technical directors and technical artists, ICE has become a gateway to understanding the math behind the computer graphics tools we use on a daily basis. It's ease of use encourages experimentation and being able to visualize results quickly, of course means faster learning.

The material here is a condensed version of the Math Review portion of sessions presented in 2010 and 2011 for Autodesk Japan. It is intended to provide a brief, if remedial review of some basic math fundamentals for technical artists, and is meant to help bridge the gap between what many of us probably studied in high school, and what we need on a daily basis.

In the interest of brevity (the course this presentation was designed for was only two hours long), I have omitted details for such things as how the component-wise solutions for the cross-product are computed, and how Matrices multiplications and rotations work.

Generally speaking, in practice it's often more important to know ***when*** to use those operations than it is to know how to compute them from scratch, those specifics can easily be found online on Wikipedia, or in any introductory book to computer graphics. Recommended reading for those interested in studying the subject further could include David Eberly's "Geometric Tools for Computer Graphics", and John Vince's "Mathematics for Computer Graphics".

Trigonometry

The Pythagorean Theorem

Trigonometry's Pythagorean theorem let's us find the length of any side of a right triangle based on the lengths of the other two sides. The Pythagorean Theorem is $\mathbf{A}^2 + \mathbf{B}^2 = \mathbf{C}^2$, where **A** and **B** represent the lengths of a right triangle's sides, and **C** is the length of it's hypotenuse or diagonal edge *(fig. 1a)*.

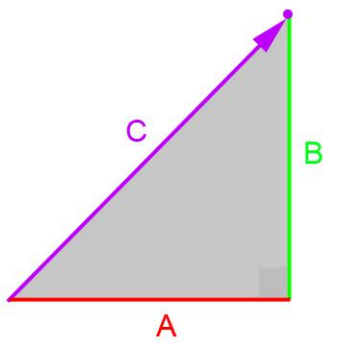


fig. 1a

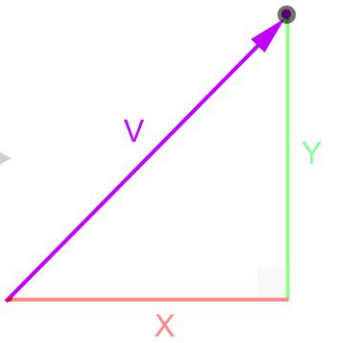


fig. 1b

Solving for the length of a right triangle's side **C** is the same as solving for the length of a 2d vector **V**, where **V**'s x and y coordinates (**V**.x and **V**.y) are the supporting sides of the right triangle *(fig. 1b)*.

We can extend the Pythagorean theorem from 2d (x, y) to 3d (x, y, z) by adding the square of a third component. The Pythagorean Theorem in 3d is $\mathbf{A}^2 + \mathbf{B}^2 + \mathbf{C}^2 = \mathbf{D}^2$.

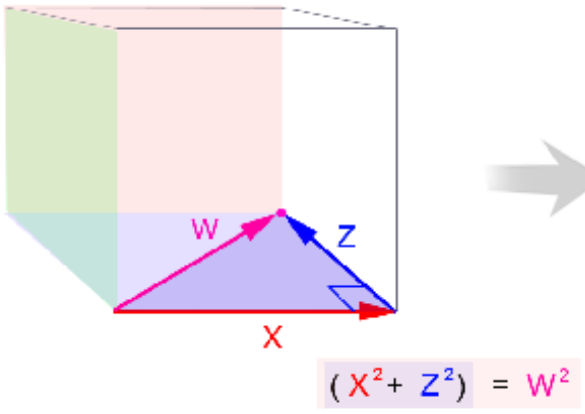


fig. 2a

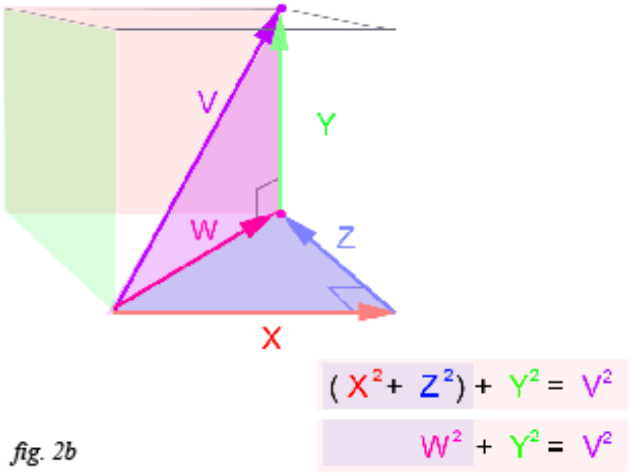


fig. 2b

We can also visualize the pythagorean theorem in 3d as as the combination of two 2d triangles formed by it's XYZ components. The first triangle (with hypotenuse **W**) lies on the XZ plane with **W**.x and **W**.z as it's sides *(fig. 2a)*, and the second triangle (with hypotenuse **V)** stands upright and uses **W** in the XZ plane, and **V**.y as it's supporting sides *(fig. 2b)*.

So a triangle (and the pythagorean theorem) in 3d can be reduced to 2d ($\mathbf{A}^2 + \mathbf{B}^2 = \mathbf{C}^2$), since a triangle is always planar and flat, even if it lives in 3d space.

However in practice (i.e. in ICE), if we want to solve for the length of a 3d triangle's hypotenuse (or vector **V**) manually, we will probably need to solve it as $\mathbf{A}^2 + \mathbf{B}^2 + \mathbf{C}^2 = \mathbf{D}^2$, as in the top of *fig. 2c*.

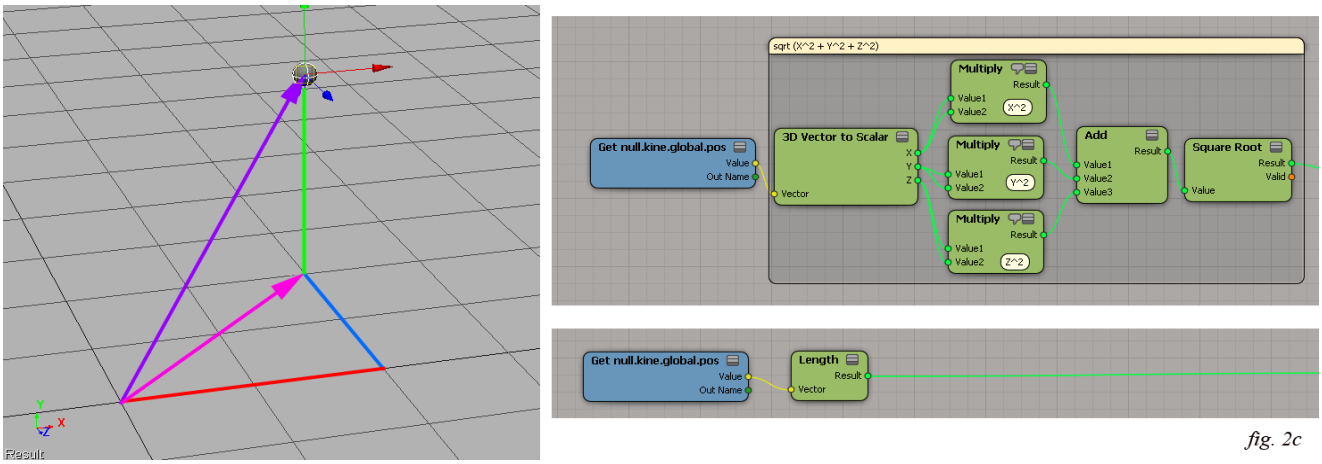


fig. 2c

Obviously, if all we need is the length of a vector, it's far easier to use the "length" node provided by Softimage, and in practice it's almost always better to take advantage of the factory nodes that are available (and it's pretty safe to say they have trigonometry well covered out of the box!).

However, a lot of the power of ICE comes from having the ability to design the tools you need as the project evolves. Much of that flexibility only becomes available once you have the know-how to apply fundamental mathematics towards solving a series of sub-tasks that will get you closer to your creative goal.

Sine, Cosine, and Tangent

Given any interior angle, the Sine, Cosine, or Tangent functions provide us with the relative length ratios of any of it's other two sides. Conversely, the inverse Trigonometric functions ArcSine, ArcCosine, and ArcTangent perform the opposite mapping, where given the lengths of any two of it's sides, we can solve for any interior angle *(fig. 3)*.

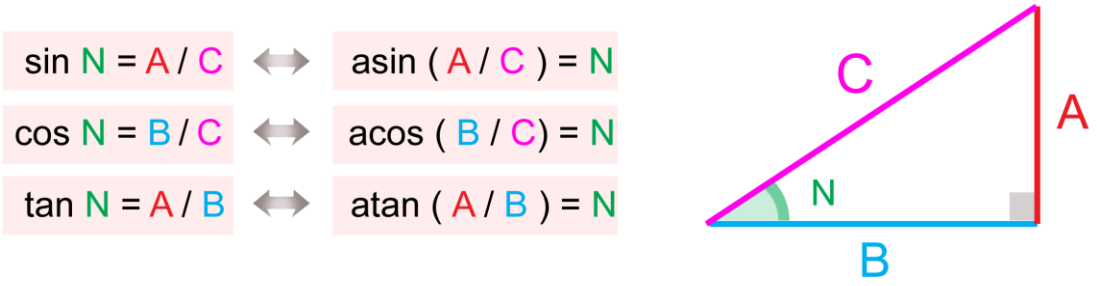


fig. 3

For example, given angle **N** of a right triangle, we could use the cosine of angle **N** to solve for the relative lengths of sides **A** and **B**. Conversely, given the lengths of sides **A** and **B**, we could use arccosine to solve for angle **N** *(fig. 4)*.

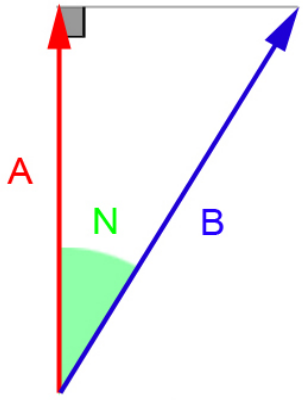
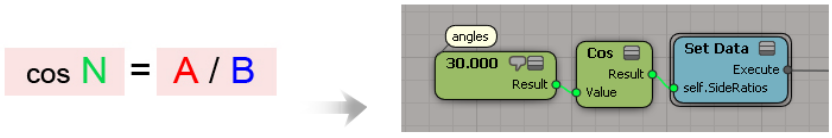
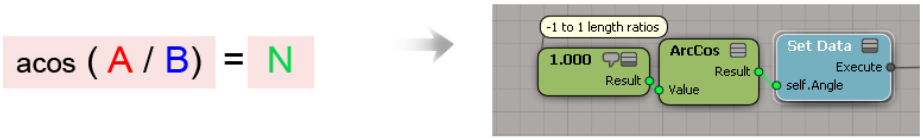


fig. 4



Cosine converts Angles to Length Ratios



ArcCosine converts Length Ratios to Angles

It is always important to select the correct function for the sides whose lengths are known
In this case, because different sides are known, we select Tangent instead of Cosine *(fig. 4a)*.

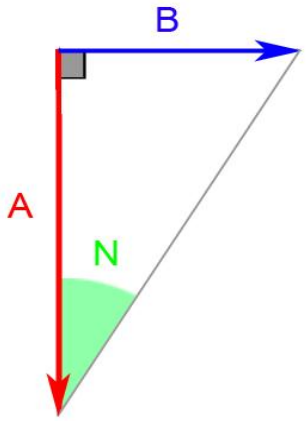
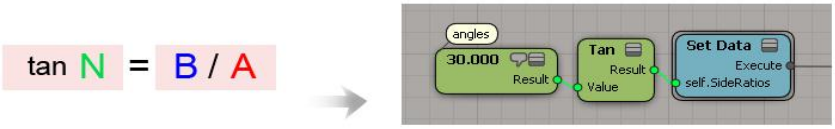
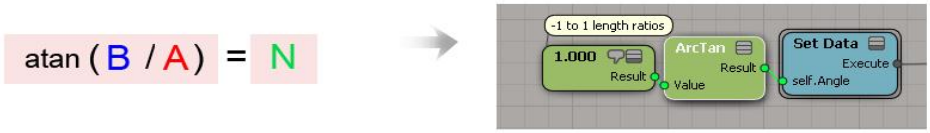


fig. 4a



Tangent converts Angles to Length Ratios



ArcTangent converts Length Ratios to Angles

There is a 3-part imaginary word (in English) called "**Soh-Cah-Toa**" that is often used as a mnemonic for remembering the trigonometric functions that correspond to the sides we want to solve for.

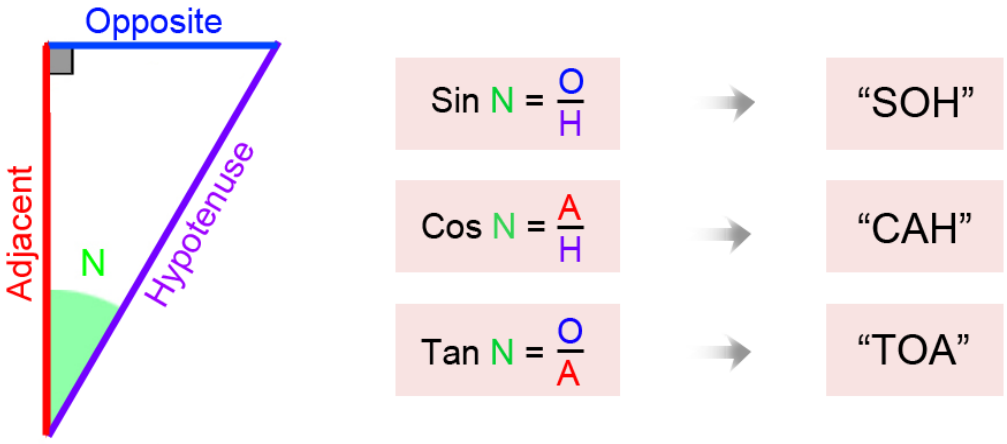


fig. 4b

The first letter of each syllable in the mnemonic corresponds to a trigonometric function (e.g. **S**ine, **C**osine, or **T**angent) that is paired with the the length ratio of two of the sides (e.g. **A**djacent, **O**pposite, or **H**ypotenuse).

For example:
"**SOH**" is **S**ine = **O**pposite over **H**ypotenuse
"**CAH**" is **C**osine = **A**djacent over **H**ypotenuse
"**TOA**" is **T**angent = **O**pposite over **A**djacent

Of course we can also use the same mnemonic to find the appropriate inverse trigonometric function as well.

In the example below, we are given the lengths of the **opposite** and **adjacent** angles, and by dividing them we can use the inverse trigonometric function **ArcTangent** to find the angle **N** (*fig. 4c).*

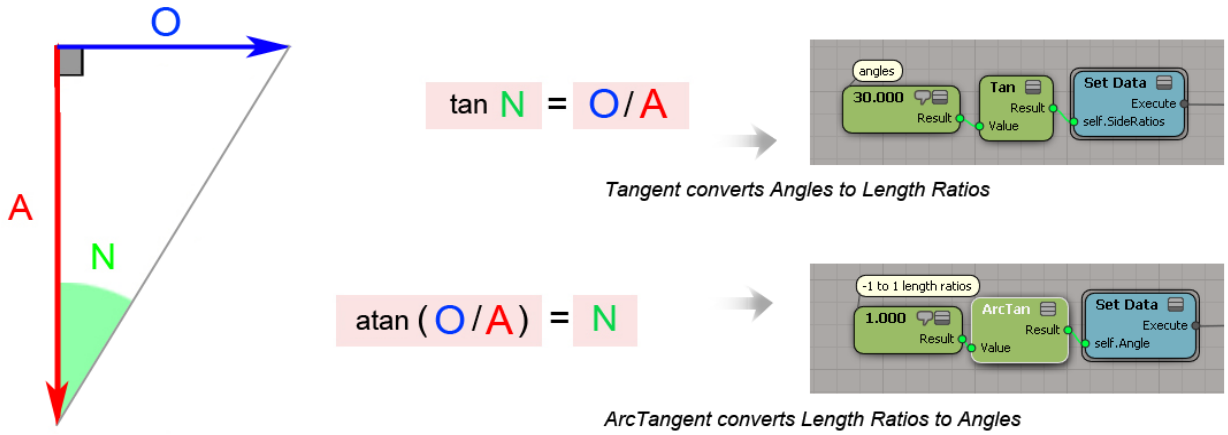
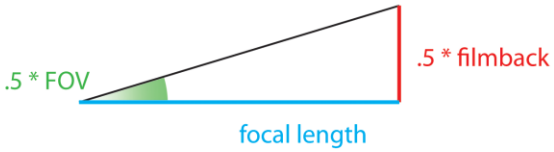


fig. 4c

A practical use for the inverse trigonometric functions could be solving for the (horizontal) FOV angle

of a camera given it's focal length and camera film-back size. Because we are converting length ratios to angles, we can use *ArcTangent* to solve for the camera FOV *(fig. 5)*.



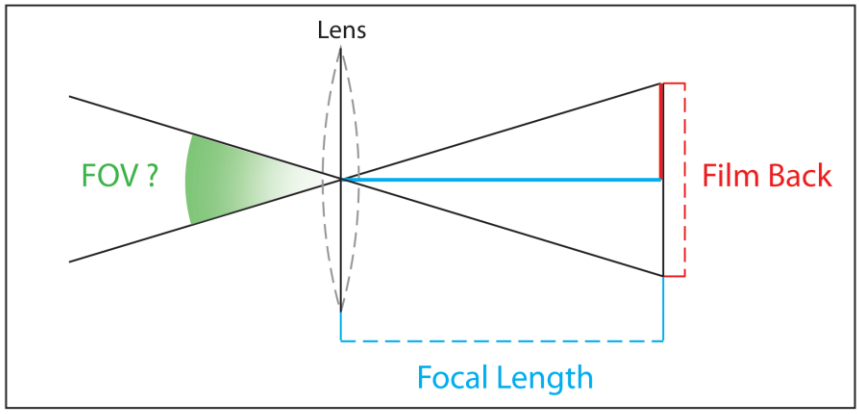


fig. 5

The equation we want to use is: **FOV = 2 * atan (.5 * filmback / focal_length)**
Notice that we only use *half* the length of the filmback, and that we need to multiply the resulting FOV angle by 2, since though we solve for the FOV angle on a right triangle, the full FOV is double that.

Visualizing Sine and Cosine

It may help us better understand the relationship between a triangle's sides and it's interior angles by examining how the length ratios of a triangle's sides change as it's hypotenuse rotates upwards. Suppose we rotate a line segment (our triangle's hypotenuse) upwards (counter-clockwise) at even angular intervals *(fig 6)*.

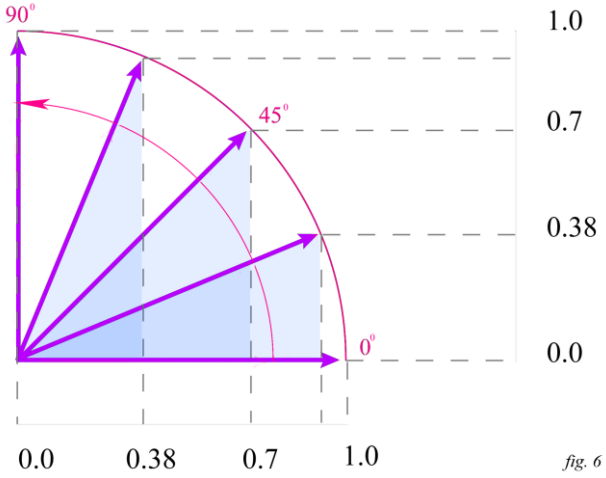


fig. 6

As it rotates upwards, we can see that it's length projected downward on the X-axis decreases until its 'shadow' on the X-axis disappears altogether. You may also notice that the length of our vector's projection (or 'shadow') on the X axis at 45 degrees is actually about .7 *(fig. 6)*.
We can observe then, that the relationship between the rotational increments and length of it's

projection is not linear. Graphing the length of our edge projected horizontally against the Y axis generates a curve that oscillates smoothly between -1 and 1 *(fig. 7)*.

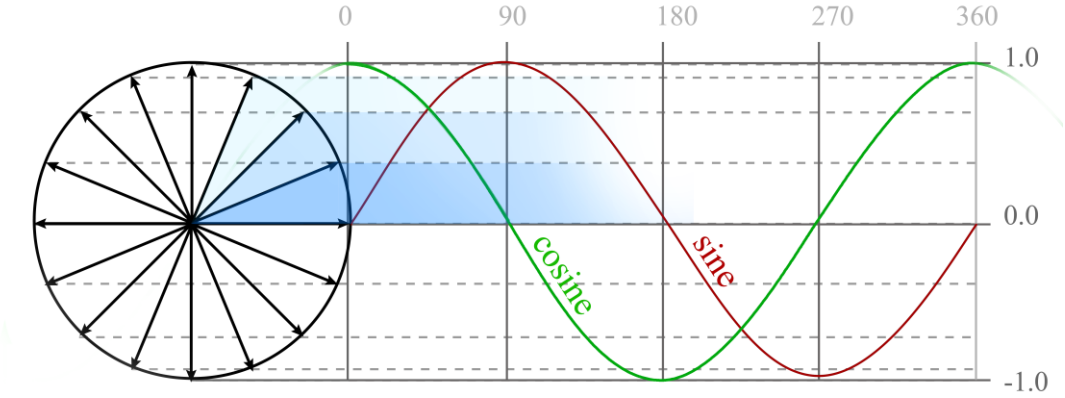
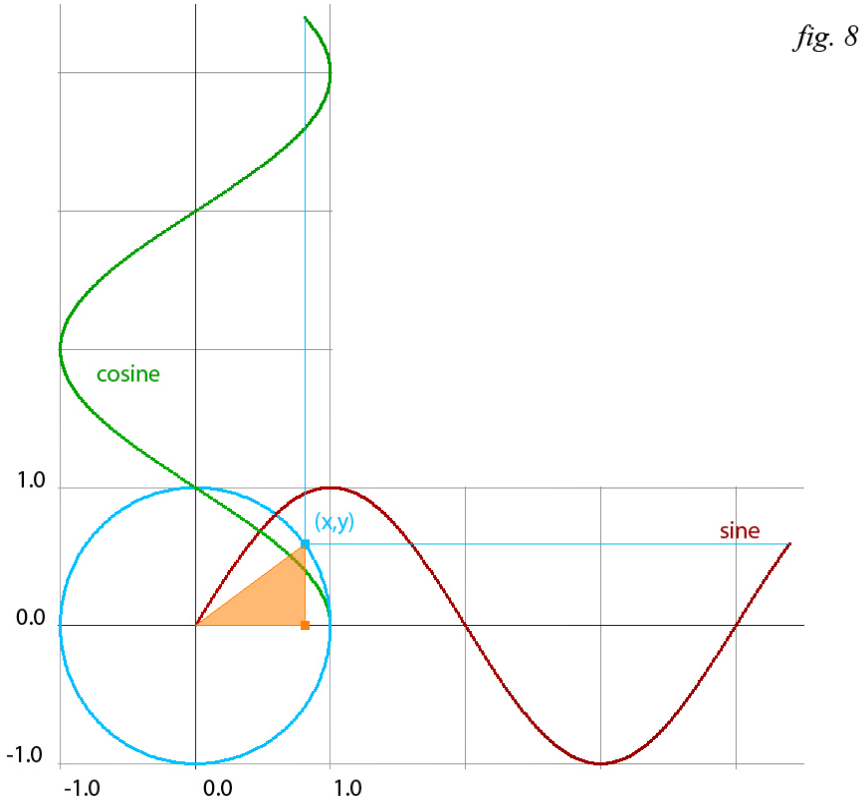


fig. 7

We can observe that this graph is periodic (i.e. it cycles and repeats) at even intervals of 360 degrees or 2pi radians. We can also see that the shape of the curves for sine and cosine are similar, only different in that they are offset by 90 degrees *(fig. 7)*.

In the diagram below, we've inscribed our triangle in a unit circle, and have plotted the sine and cosine of a constantly increasing rotation along the X and Y axes *(fig.8)*.

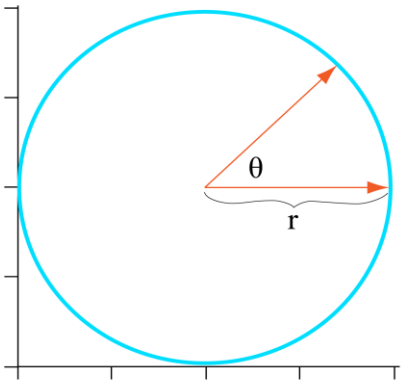
fig. 8



We can observe that the cosine plot follows the X-coordinate of the triangle's corner as it slides along the edge of our circle. Likewise, the sine plot follows the Y-coordinate of our triangle's corner *(fig.8)*.

Notice also how the intersection of Cosine along the X axis and Sine along the Y axis describes a

perfect circle coincident with the edge as the interior angle increases from 0 to 360 *(fig. 9)*.



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

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

r = radius
θ = angle

fig. 9

The parametric formula for a circle is:

x = r * cos(angle)
y = r * sin(angle)

where 'r' is the radius of our circle and 'angle' is a quantity varying between 0-360 *(fig. 9)*.

Trigonometry Applied

Because they can describe a perfect circle in whole or in part, Sine and Cosine can also be used to perform rotations around an axis.

In the example below, I am performing a 2d rotation of my geometry's point positions (in the XY plane, around the Z axis) by using sine and cosine to modify their x and y components *(fig. 10)*.

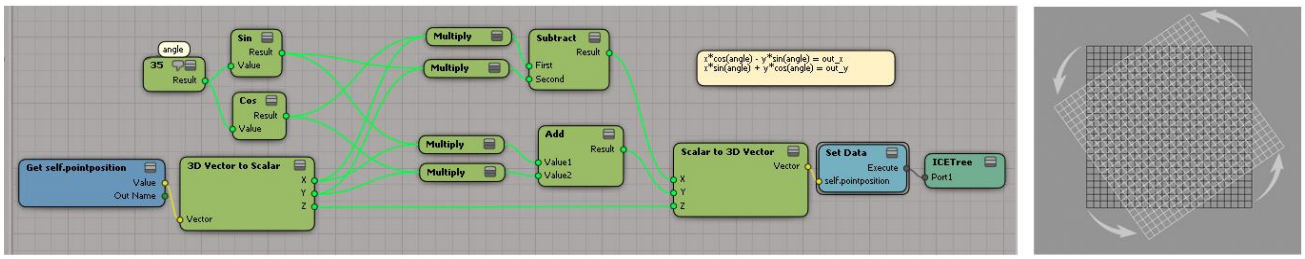


fig. 10

This example is of a 2d rotation in the XY plane, but we can apply a series of planar rotations around the X, Y and Z axes to describe a full rotation in 3d space.

In the next example, we can observe how the oscillating shapes of the Sine and Cosine functions can be used to to create a wave-like ripple deformation. Because they are ***periodic*** and because they repeat and cycle, Sine and Cosine are probably one of the most commonly used functions in procedural textures and deformations *(fig. 11)*.

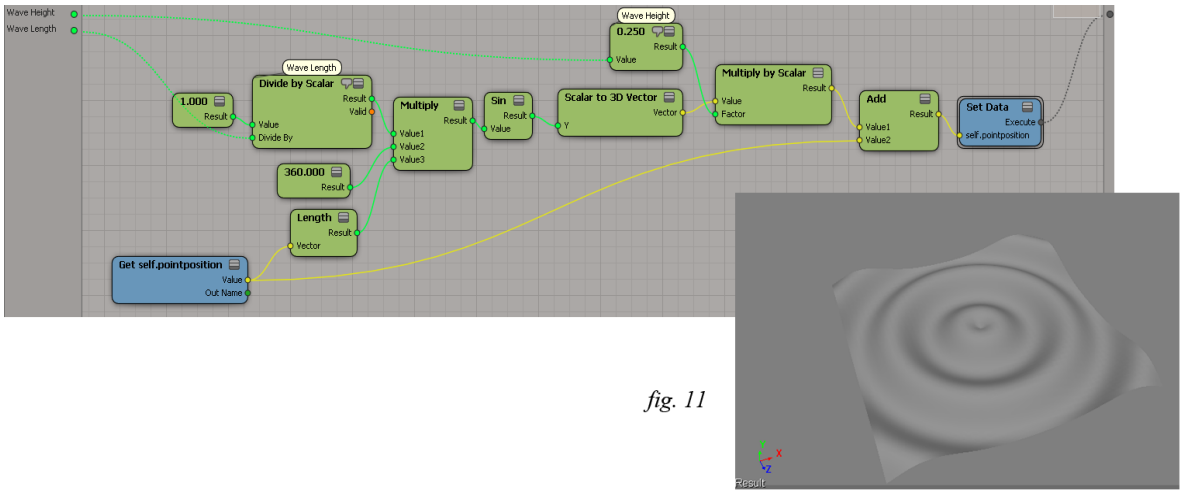


fig. 11

We can also use trigonometry to perform tasks like rectangular/polar coordinate conversions using the inverse trigonometric functions. Here we use ArcTangent to map the angle/distance of a pixel/particle from the center of the flower to a rectangular X/Y position *(fig. 12)*.

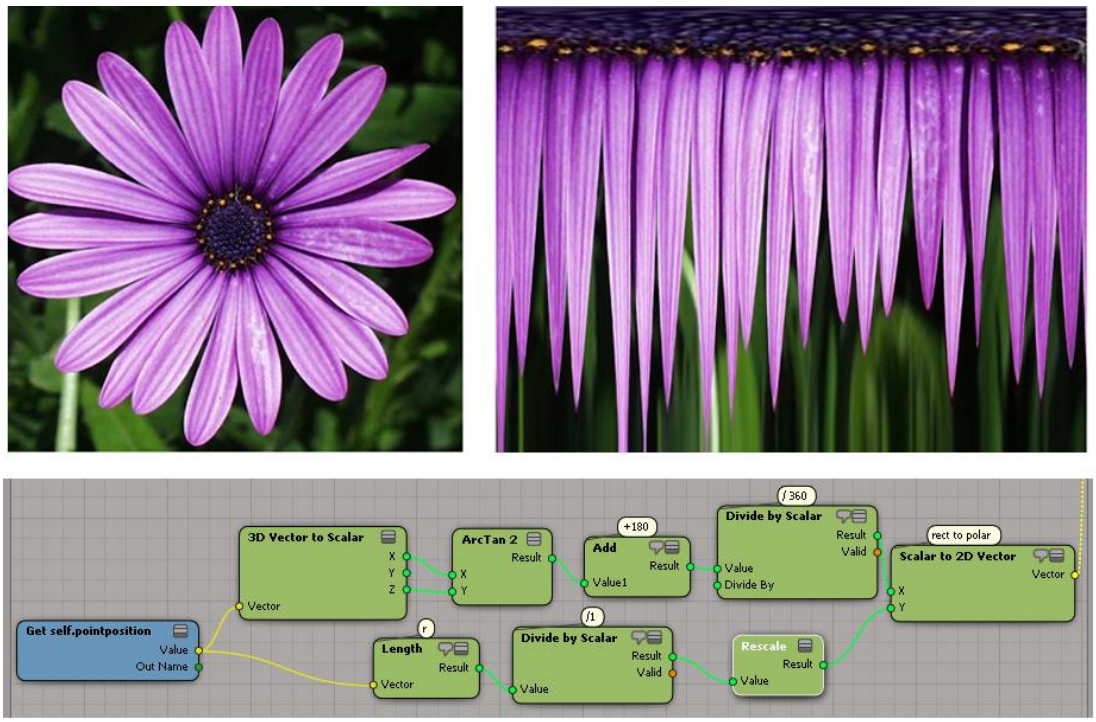


fig. 12

If you look closely, you may have noticed that in the ICE tree I am using ***ArcTan2***, instead of the standard ***ArcTan***. This is because ***ArcTan2*** returns the angular range of a full circle (from -180 to 180), while ***ArcTan*** returns the angular range of a semi-circle (from -90 to 90).

There are many other uses for trigonometry but in short, the Sine, Cosine and Tangent functions and their inverse relate a triangle's interior angles to the proportional lengths of it's sides, and it is up to us to figure out how to apply those relationships towards solving the problem at hand.

Vectors

We learned earlier that we can use the Pythagorean Theorem to find the length of a line segment, or vector by using it's x, y, or z components as the sides of a right triangle *(fig. 13)*.

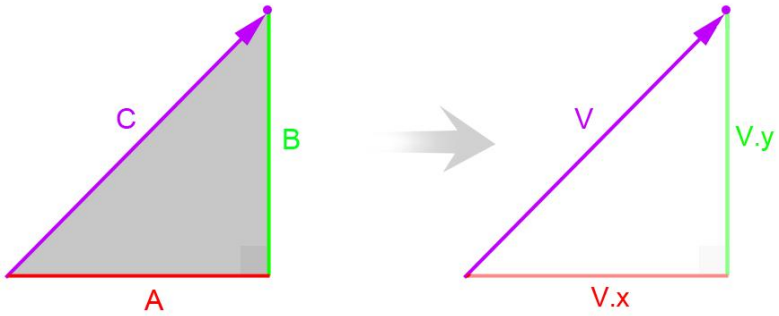


fig. 13

One way to think of a vector is as an arrow with both a head and tail. In this case our point in space is the head of the vector, and the tail of our vector lies at the origin (0,0,0).

A vector has two independent properties we are interested in. Those properties are direction (or orientation), and magnitude (or length) *(fig. 14)*.

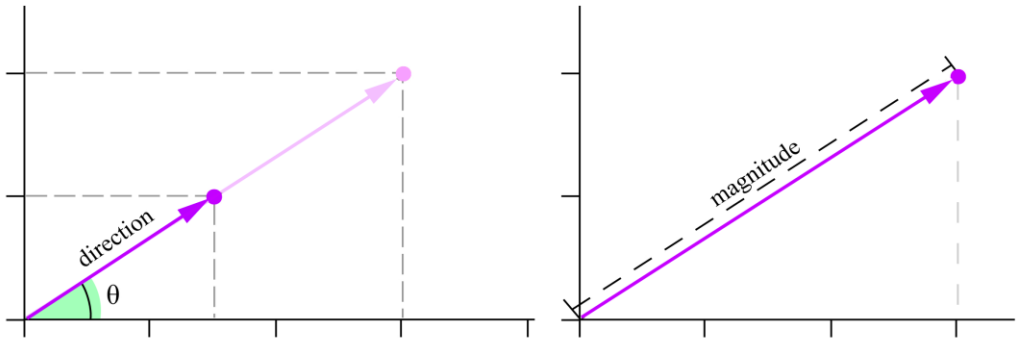


fig. 14

Vector Addition

Adding two vectors is analogous to chaining them head to tail *(fig. 15)*.

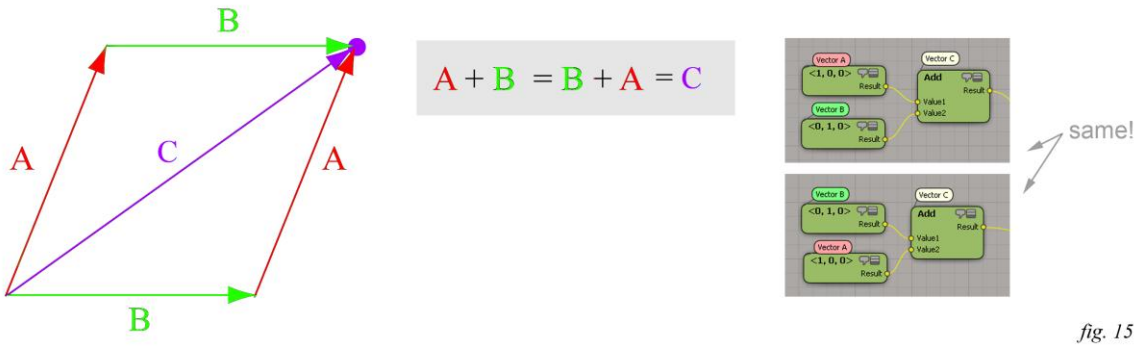


fig. 15

Vector Addition is also commutative, in other words, **A**+**B** = **B**+**A**, and changing the order of the operands does not affect the result.

This also means that when we sum (add) the forces in a particle simulation (say gravity + wind), adding them in a different order (wind + gravity) will still give us the same net result *(fig. 16)*.

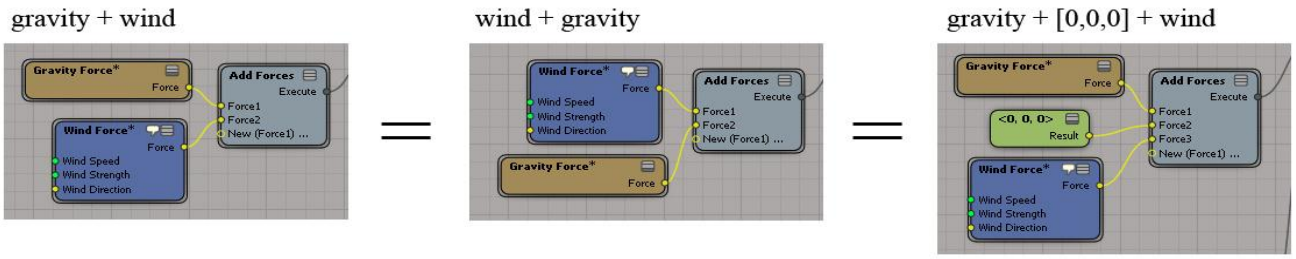


fig. 16

Vector Subtraction

Just as we can add vectors together, we can also subtract them from each other.

Subtracting a vector is the same as adding it's inverse, or *negating* it.

Negating a Vector is the same as multiplying each of it's components by -1. The new negated vector now points in the opposite direction, but has the same length as the original *(fig. 17)*.

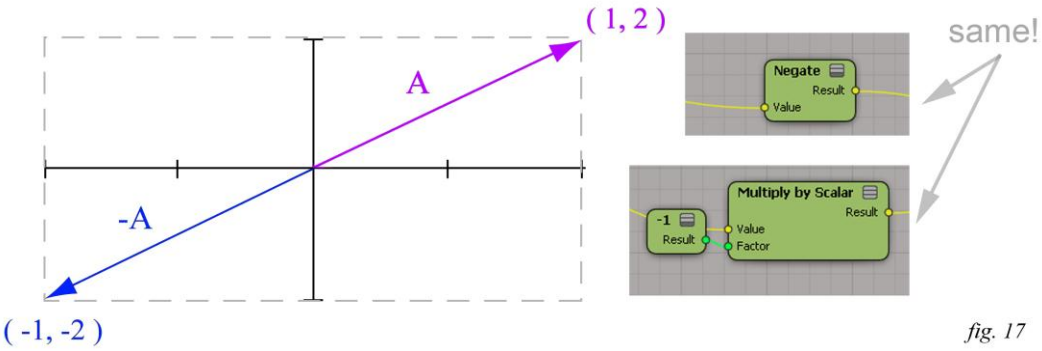


fig. 17

Vector subtraction is often used when you need to express relative distance between two points *(fig 18)*.

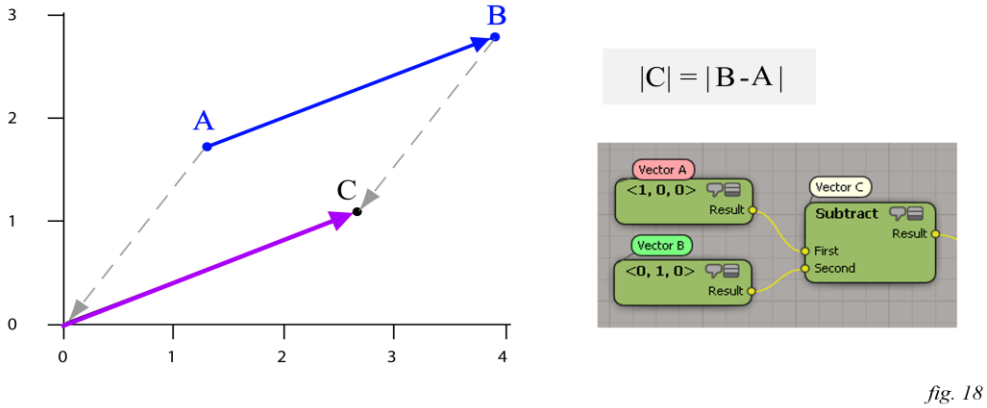


fig. 18

For example, if we have two points, **A** and **B**, subtracting the position of **A** from **B** gives us the vector **C**, which describes the relative distance between them.

So **C** now represents **B** as if the entire set of **A** and **B** were offset by the inverse of **A**, putting **A** at the origin.

Vector/Scalar Multiplication (Scaling a vector)

We can scale a vector by multiplying each of its components by the factor we want to scale it by.
We can observe that scaling a vector changes it's magnitude, or length, but not it's direction *(fig. 19)*.

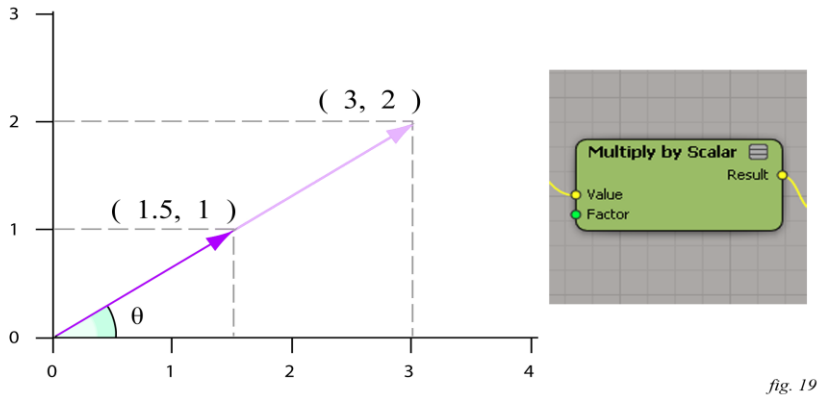


fig. 19

Vector/Scalar Division (Normalizing a vector)

If we divide each of a vector's components by it's length, the resulting vector will have a length of 1.0.
A unit vector is any vector with a length of 1.0. Any unit vector can be considered normalized *(fig 20)*.

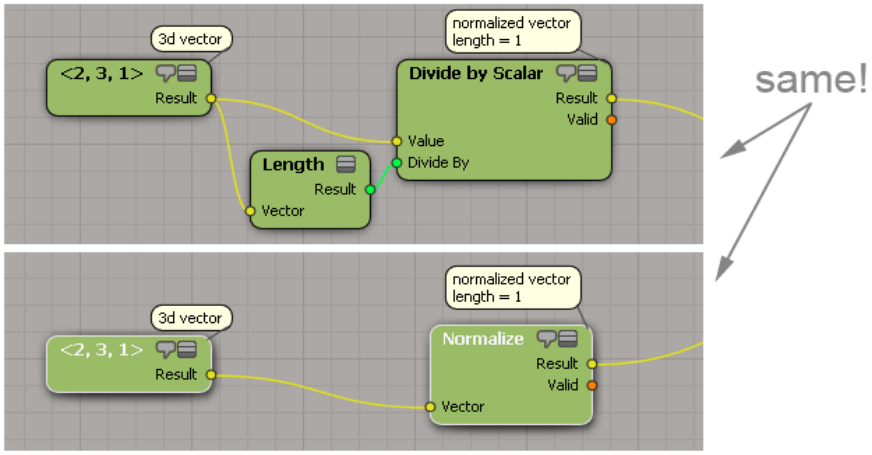


fig. 20

Vector/Vector Multiplication

We'll look at two kinds of vector/vector multiplication, one is the dot-product, and other is the cross-product.

The dot-product (or scalar product) produces a scalar value and is commutative, or indifferent to the order of operands.

The second kind, the cross-product (or vector product) produces a vector value, and is not commutative so we will need to be mindful of the order of operands when using the cross-product.

The Dot Product (Scalar Product)

One way to visualize the dot product is as the projection of one vector on to another.

In the example below, we can see that the length of vector **A** projected on to **B** is 1 when they are parallel, 0 when they are perpendicular, and -1 when they are parallel but facing opposite directions *(fig. 21)*.

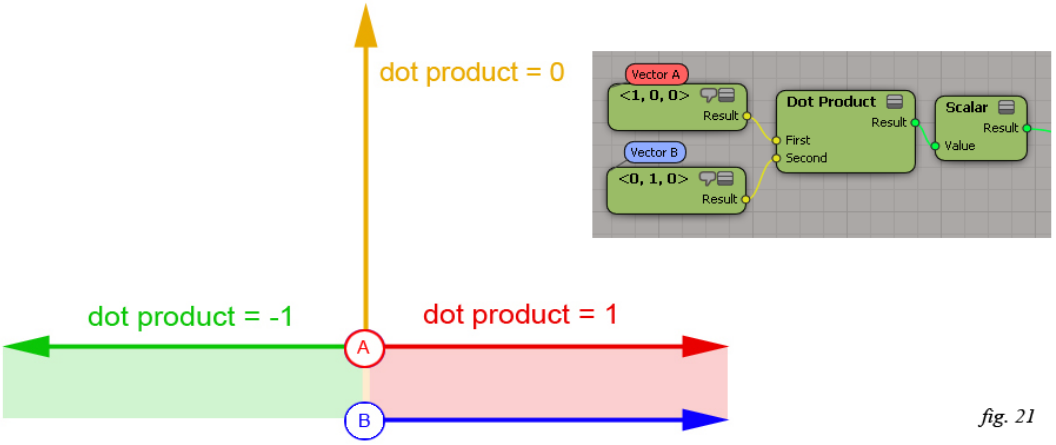
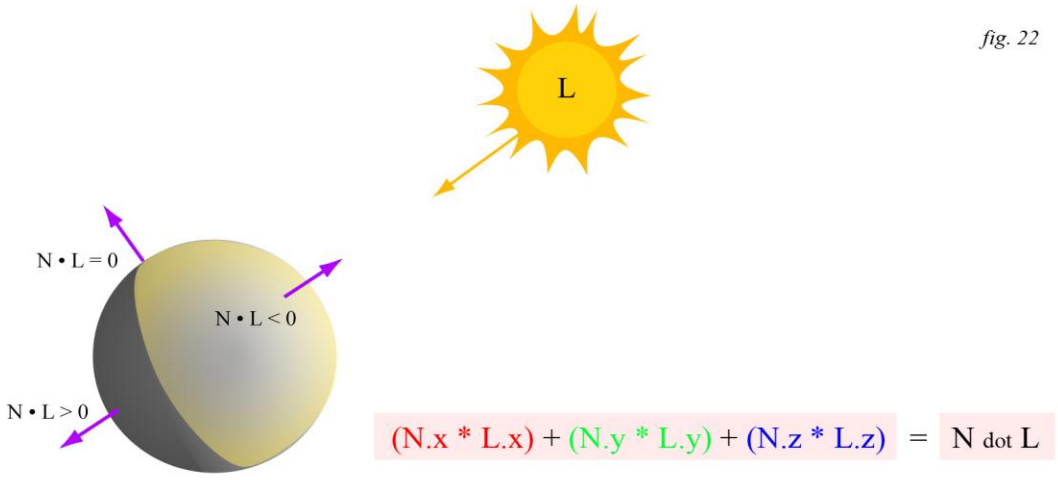


fig. 21

We can also observe that if either operand points in the opposite direction, the sign (positive or negative) of the dot product changes *(fig. 21)*.

Because of this characteristic, the dot product is frequently used in back-face detection and for diffuse lambert shading, which is simply the dot product of the surface normal against the incoming light direction *(fig. 22)*.

fig. 22



We can compute the dot product of vectors A and B manually by multiplying each component pair, and then adding them all up as in *figure 22*.

In order to get results within the -1 to 1 range, we need to operate on **unit** vectors with a length of 1. To ensure this, we can divide our vectors by their length (or **normalize** them as in *fig. 20*) beforehand, or divide the result of our dot product afterwards by the product of both of their lengths.

We can also apply this math to perform 2d relighting in a compositing package using a normals pass (where RGB channels represent XYZ normal direction vectors) multiplied by a solid color representing the incoming light direction *(fig. 23)*.

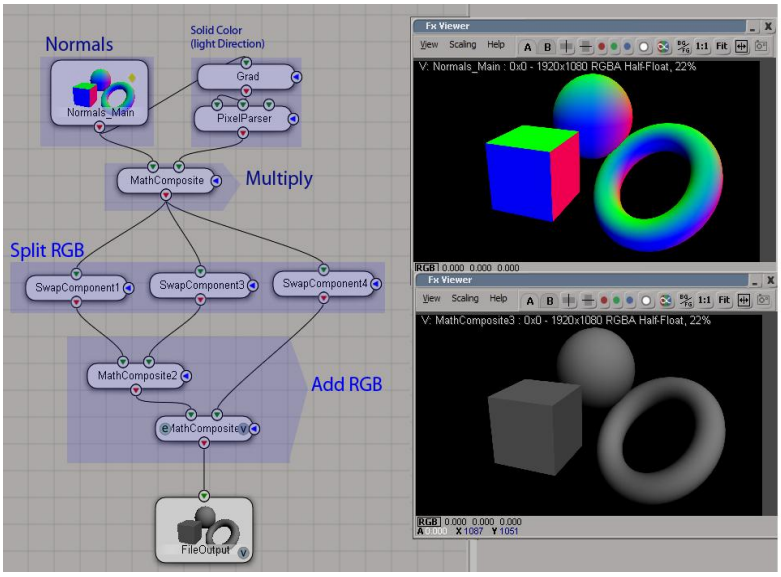


fig. 23

We can also fake diffuse shading on pointclouds without meshing them, by using the average neighbor direction as our "shading normal" and performing a dot product with a vector representing the incoming light direction *(fig. 24, 24a)*.

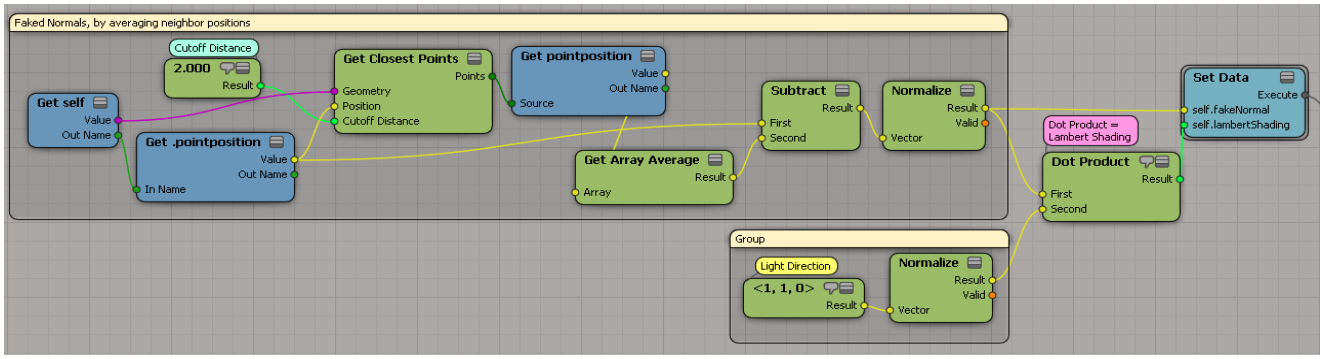


fig. 24

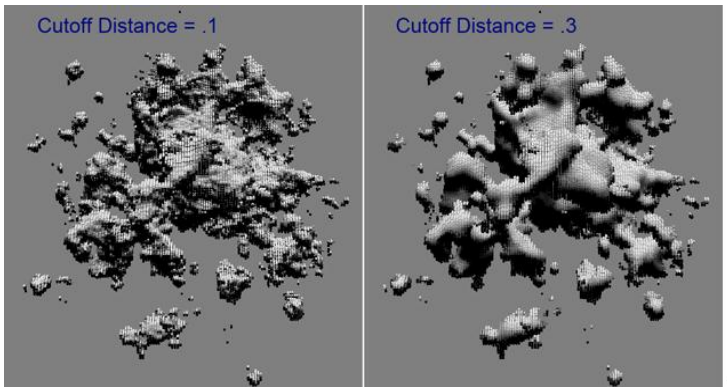


fig. 24a

Modifying the cutoff distance for the neighbor search has the effect of smoothing out the “normals” by taking a higher average of neighboring points *(fig. 24a)*. Of course, we could also simply store the fake “normals” as a vector-valued ICE attribute, and perform the dot product afterward in compositing (as in *fig. 23*).

Let's revisit the idea of the dot product as the length of the projection of (or the shadow cast by) vector **A** (in red) on to vector **B** (in blue) *(fig. 25)*.

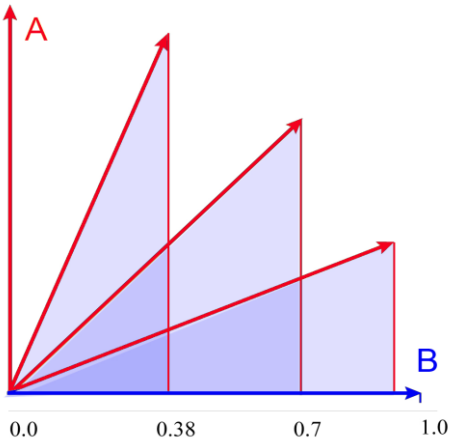


fig. 25

You may notice that this looks very similar to what we saw in the trigonometry review when we projected our edge rotating counter clockwise on to the global X-axis *(fig. 6)*.

In fact, the dot product of two vectors is equal to the cosine of the angle between them. So you could say that the dot product is really a cosine in disguise *(fig. 26)*.

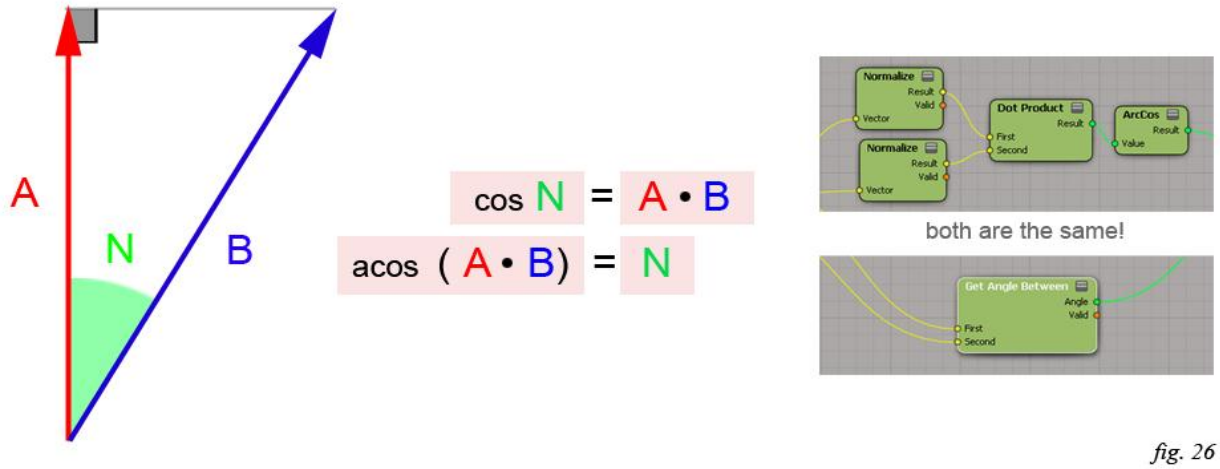


fig. 26

Knowing this, we can also find the angle between the two vectors by finding the inverse cosine, or arccosine of their dot product *(fig. 26)*.

The Cross Product (Vector Product)

The other form of Vector/Vector multiplication is called the Cross Product (or Vector Product), and involves multiplying two vectors together to produce a third vector.

The vector produced by the cross product is always perpendicular (90 degrees) to *both* of the two input vectors *(fig. 27)*.

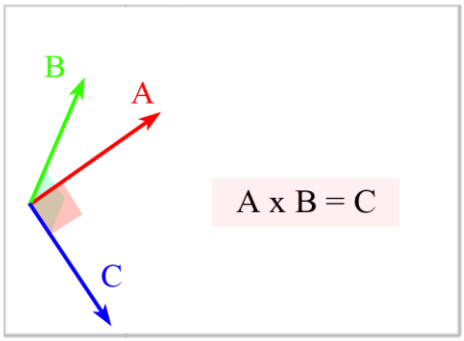


fig. 27

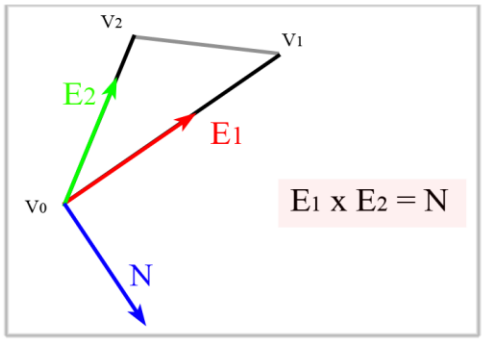


fig. 28

This makes the cross product useful for tasks such as calculating a triangle's normal based on two of it's edges *(fig. 28)*, or for finding the rotation axis of a spinning body.

One way to visualize the cross product is to imagine the two input vectors as spokes of a wheel, and the resulting output vector as the wheel's axle *(fig. 29)*.

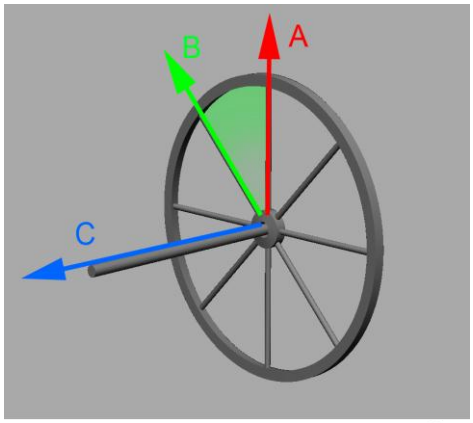


fig. 29

However the vector cross-product (unlike the dot product) is NOT commutative, so switching the order of operands inverts the direction of the resulting vector *(fig. 30)*.

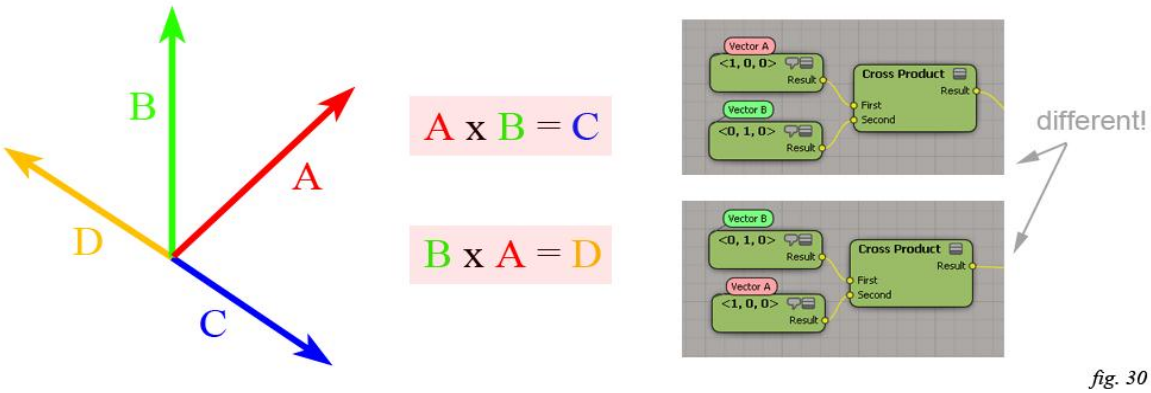


fig. 30

There is a common mnemonic to remember which way the resulting vector should point, which is called the “right hand rule”. It states that if you hold your hand like this, so that your index finger is vector A, and middle finger is vector B, your thumb will indicate where the normal vector will be pointing *(fig. 31)*.

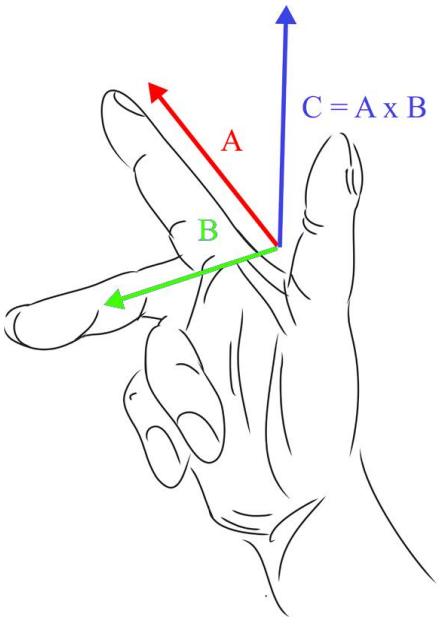


fig. 31

Using the Cross-Product to define a Coordinate Space or Reference Frame

The three vectors or axes that describe the roof's orientation are sometimes called ***Basis Vectors*** because they can be scaled and combined to describe any point within that space.

If I have 3 axes describing the roof's coordinate space and a position vector on the plane of the roof, then I now have enough information to describe the roof's reference plane (or local space) *(fig. 32)*.

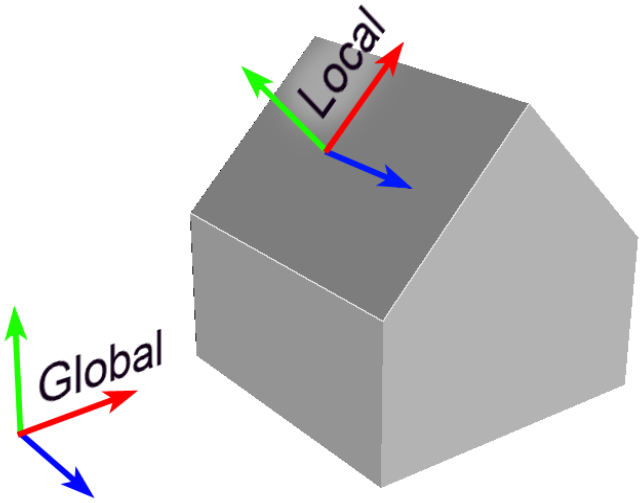


fig. 32

Suppose I need to place an object (like a chimney) on the roof of this house. If I had a coordinate space that were aligned to the slope of the roof, then I could place the object so that Y=0 within that

coordinate space (and lies flat along the X/Z plane).
I can use the cross product and three of the roof's vertex positions to solve for 3 basis vectors aligned with the roof as follows *(fig. 33a, fig. 33b)*:

1) Subtract vertex position **V0** from **V1**, for vector **E1** (**E1** will double as my new **X** axis).
2) Subtract vertex position **V0** from **V2**, for vector **E2**.
3) Find the cross product of **E1** and **E2** which will be my new **Y** axis (and is the triangle normal).
4) Normalize my new **X** and **Y** axes
5) Find the cross product of the **X** and **Y** axes for my new **Z** axis.

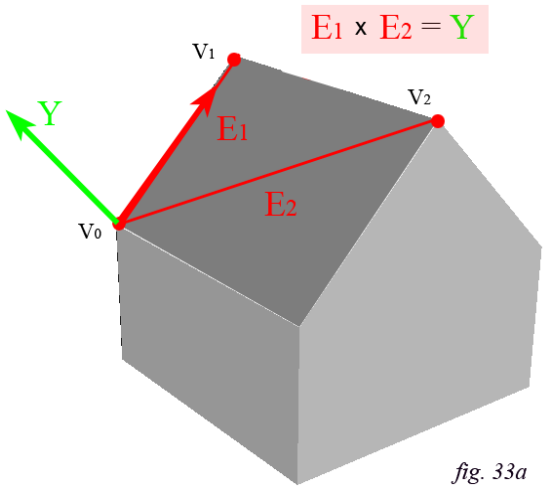


fig. 33a

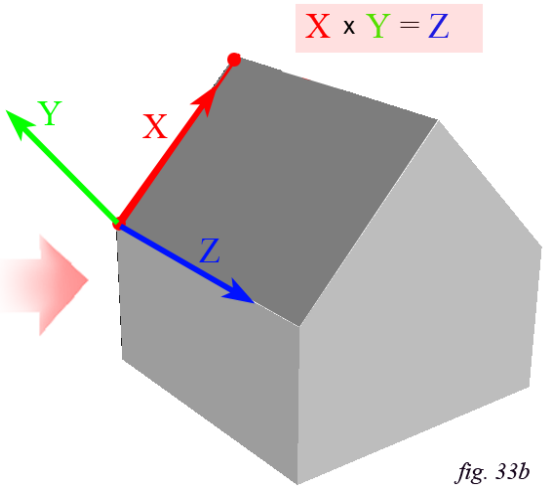


fig. 33b

Now that we have 3 direction vectors, we have enough information to describe the orientation of the roof. However, our 3 direction vectors alone do not include any information about the position or the origin of that space.

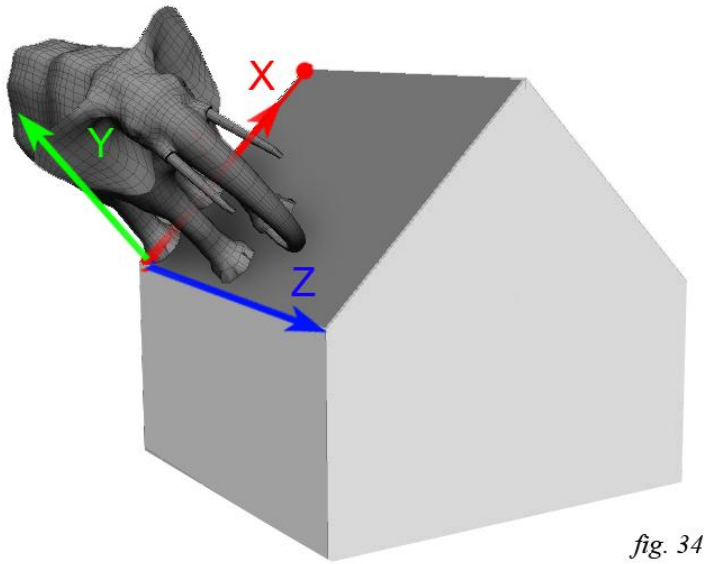
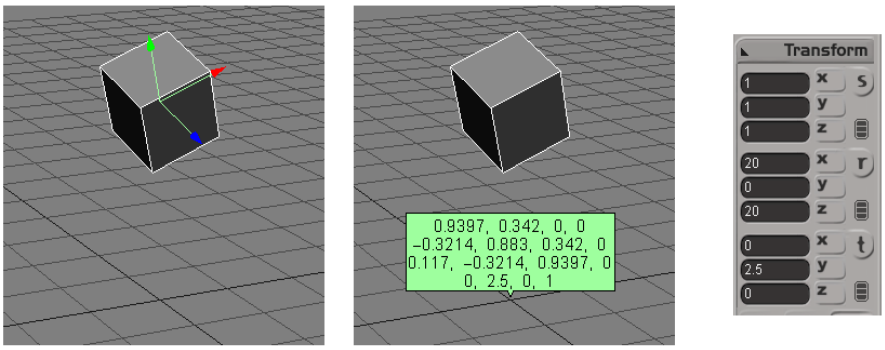


fig. 34

In the next section we will look at how 3 direction vectors describing orientation along with a position vector can be stored within a 4x4 matrix to describe a full transformation with Scaling, Rotation, and Translation.

Matrices/Transforms

Probably the first thing anyone learns how to do in 3d is to use SRT manipulators to Translate, Rotate, and Scale objects to position and arrange them. In ICE, if we try to display the global transforms of an object in the ICE tree, we have the option to either display it graphically as 'axes', or display it numerically - which displays it as a 4x4 Matrix *(fig. 35)*.



Display SRT as 4x4 "Axes" | Display SRT as 4x4 "Numeric" | SRT display

fig. 35

A matrix is a basically a grid of data with rows and columns. A 4x4 transformation matrix typically has 4 rows and 4 columns, for a total of 16 entries.

Matrices are convenient for managing transformations in the sense that we can perform a 'parent' operation simply by multiplying two of them together. Or by multiplying one by the inverse of another we can perform the equivalent of an 'un-parenting' operation.

Let's take a look at the anatomy of a matrix. A good place to start with is what's called the identity matrix *(fig. 36)*.

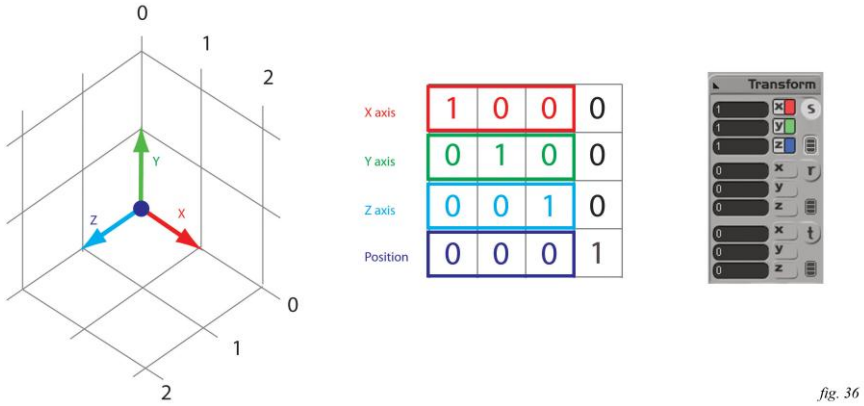
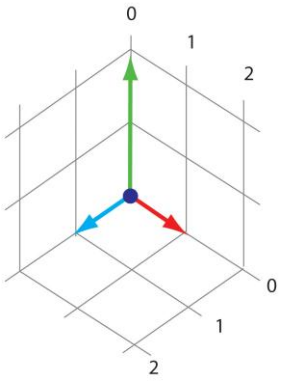


fig. 36

A 4x4 identity matrix is a "zeroed" transform. In other words it's the matrix representation of a transformation with Scaling = 1, and zero for Rotation and Translation. We can also observe that it has ones in the diagonal, and zeros everywhere else.

Each row corresponds to the direction that each of it's orientation axes point towards. In this case we have an object at the origin, with 'zeroed' transforms (and identity matrix for it's transform) so that it's

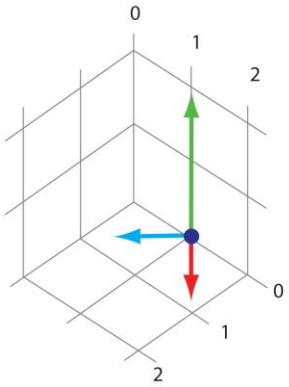
orientation axes are aligned with the world XYZ axes.
If I scale the object on it's Y axis, we will notice that the scaling of that axis corresponds to the length of the vector in the second row *(fig. 37)*.



X axis	1	0	0	0
Y axis	0	2	0	0
Z axis	0	0	1	0
Position	0	0	0	1

fig. 37

When we first introduced vectors, we also introduced the concept of a vector's magnitude being independent of it's orientation. Similarly, it is possible to store Scaling and Rotation as independent properties superimposed upon each other in the same matrix.



X axis	.7	0	.7	0
Y axis	0	2	0	0
Z axis	-.7	0	.7	0
Position	1	0	0	1

fig. 38

Scaling and orientation cohabit in the sense that the **magnitude** of each vector describes its **scaling** along each axis, while the **direction** describes the **orientation** of each axis *(fig. 38)*.

Recall that in an earlier example we created three orthonormal basis vectors from two arbitrary vectors as below:

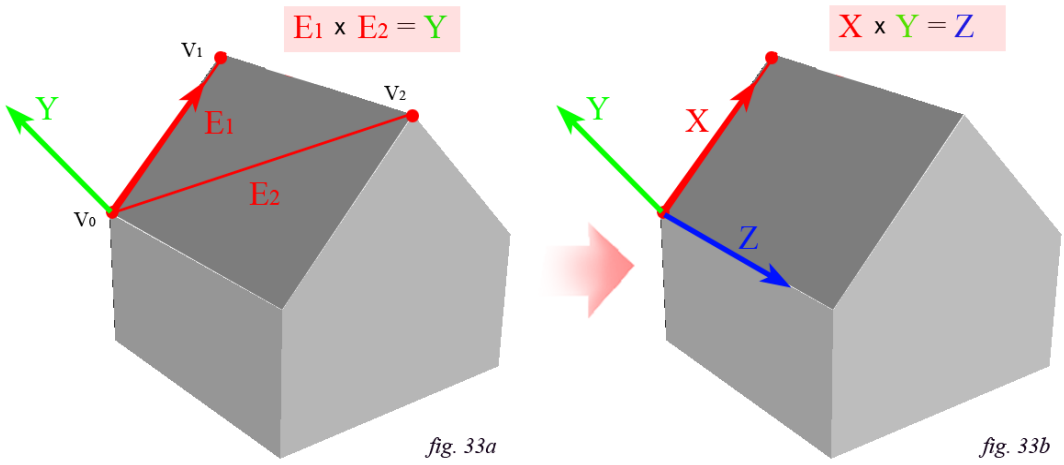


fig. 33a *fig. 33b*

We can now use these three vectors in addition to a fourth position vector (**V0** in *fig. 33a*) to build a full 4x4 Matrix describing Scaling, Rotation and Translation *(fig. 39)*.

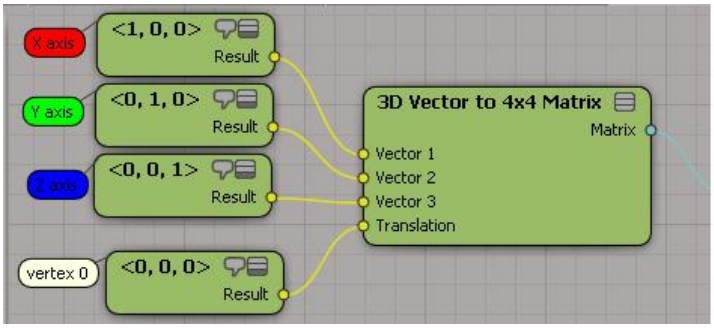


fig. 39

Using Matrices

Matrices are useful to perform tasks such as "parenting" one object to another, or to transform objects from one space to another (e.g. Global to Local or vice versa).

Let's take a look at the earlier example of the house where we used the cross-product to solve for three orthonormal basis vectors describing the local space of the roof *(fig. 33a, 33b)*. Suppose that we wanted to place an object (like this elephant) flat on the roof of this house *(fig. 34)*.

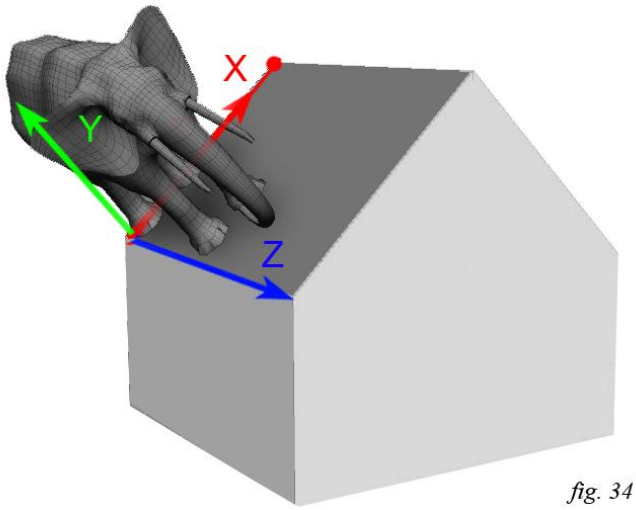


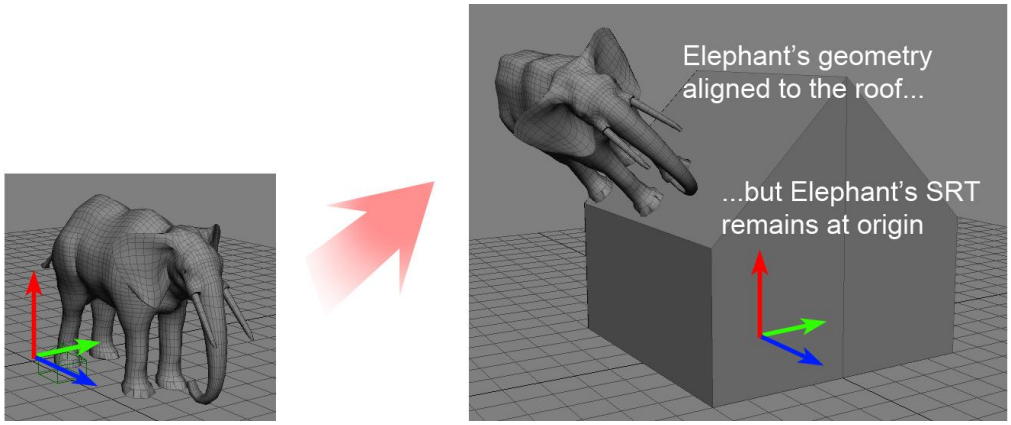
fig. 34

There are 2 ways to accomplish this task:

1) By **deforming** it's geometry, so that it's transforms remain stationary at the origin but it's *point positions* are transformed in to position (similar to a skinning or envelope deformation)
2) By **transforming** it in to position, so that it's point positions remain un-deformed but it's *transforms* are modified to align the object with the roof.

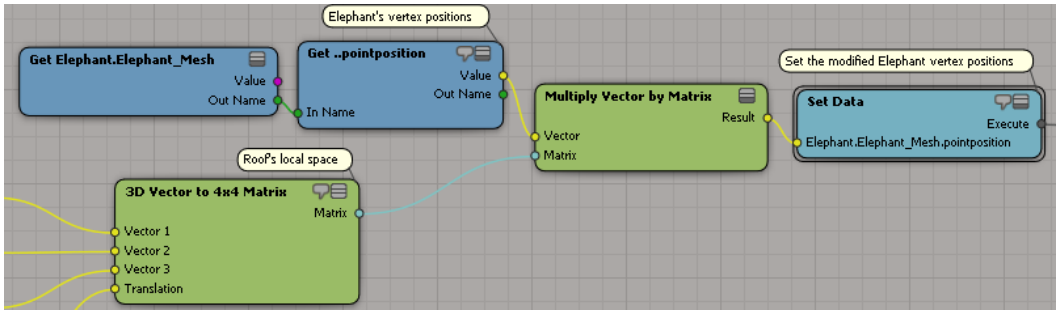
Vector/Matrix Multiplication (deforming or transforming points)

Now that we have a 4x4 transformation matrix describing the position and orientation of our roof, it is fairly straightforward to place an object (like a chimney, shingles, or this elephant) on it.



First we need to make sure that the object we want to place sits exactly on the X/Z plane in world space (as in the image on the left). I have also slightly offset the elephant relative to the origin by the same amount I want the elephant offset from the corner vertex of the roof.

Then we simply multiply the object's point positions by the transformation matrix of the roof (as in the ICE tree below). This will transform (or more specifically, deform) the points of the elephant by our 4x4 matrix (resulting in the image in the upper right).

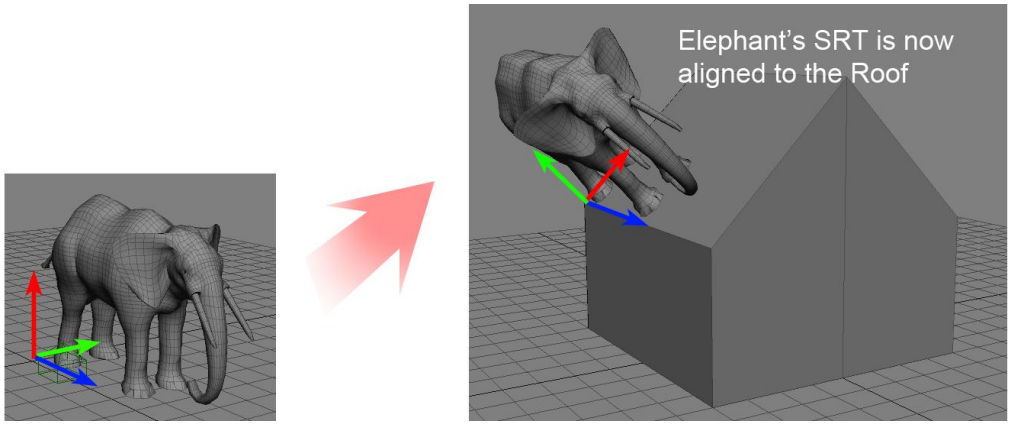


Note that because we are applying the transformation of the roof as a deformation to the elephant's geometry, the SRT of the elephant stays at the origin and remains unchanged,

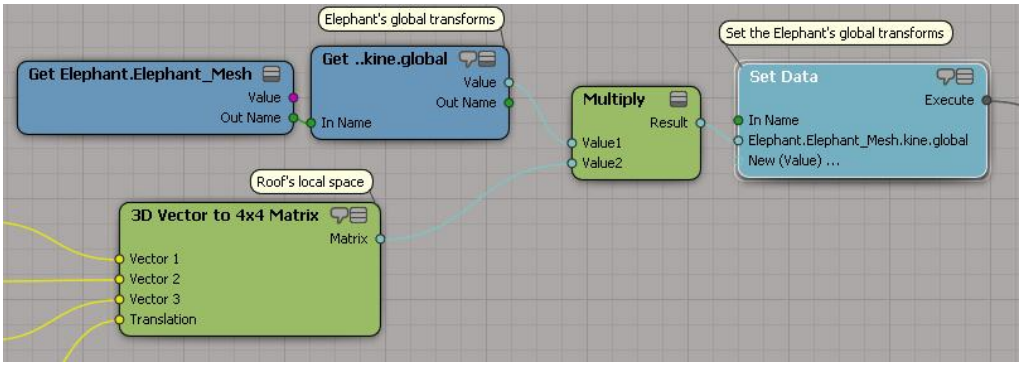
This kind of deformation is similar to building a skinning or envelope deformer in ICE from scratch, except that instead of one matrix deforming the mesh you would typically have an array of matrices (for every bone in the skeleton) with a corresponding array of bone weights.

Matrix/Matrix Multiplication (“parenting”, or concatenating transforms)

Sometimes we just want to apply a transformation directly – for example we may simply want to modify the elephant's transforms so that it is aligned to the roof, without modifying it's geometry.



To do that, we simply multiply the elephant's transform matrix by the matrix representing the local space of the roof.



In doing so, the transforms of the elephant have been appended (or concatenated) to the transforms describing the local space of the roof, and the point positions of the elephant's geometry relative to it's own center remain unchanged.

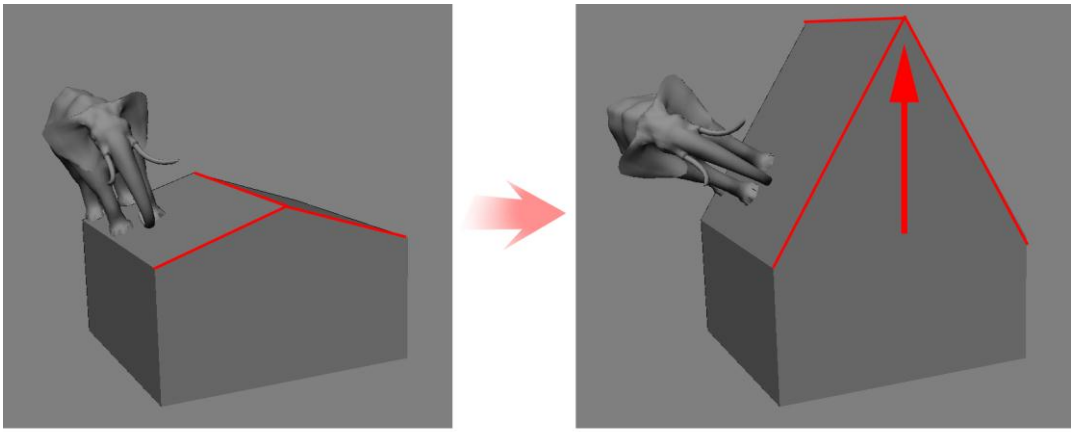


fig. 42.

What we have is a geometric constraint that pins the elephant to the roof plane, allowing us to deform the roof while maintaining the desired orientation of the elephant relative to the roof edges.