Christian Franz

3D GrafSys

Version 2.0

for programmers

© 1993 by Christian Franz

Copyright Notice:

You may use this software, the Source and its documentation free of charge for any non-commercial use. This includes using it for writing public-domain or other *freeware* programs. If you use any part of this software in your non-commercial programs you *must* include the lines

"uses Christian Franz 3D GrafSys 2.0
©1992, 93 by Christian Franz"

in both the program's documentation and 'About...' dialog.

If you like the GrafSys or its documentation, I'm asking you to also write me a postcard.

Permission is granted to freely distribute this package and its accompanying documentation as long as neither is modified in any way nor any fees are charged other than the usual downloading fees on commercial bulletin boards.

If you include the GrafSys into a PD/Shareware Bundle (e.g. CD-ROM) you must send me a complimentary copy of any such bundle that contains the GrafSys.

For commercial use of this software (for shareware programs and any other purpose), its source or its documentation you must contact me and have my written consent. Usually all I want in return is a free registered copy of your finished work.

GrafSys is not not in the Public Domain. I, Christian Franz, retain all rights to both source and documentation.

My address is

Christian Franz
Sonneggstrasse 61
CH-8006 Zurich

Switzerland

email cfranz@home.malg.imp.com
tel. +1-261 26 96 (+ = your code for Switzerland)

Danksagungen

Ich möchte allen danken, die mich bei meiner Diplomarbeit unterstützt haben.

Besonders bedanken möchte ich mich bei *Adrian Brüngger*, meinem Assistenten, und *Michele De Lorenzi* für ihre Hinweise und Anregungen, sowie *Anita Fischer* für ihre Unterstützung bei der englischen Sprache.

What is 3D GrafSys?

GrafSys is a hierarchical object-oriented class library for THINK Pascal. It is designed to facilitate easy 3D graphics and animations in your programs. GrafSys supports full 3D control of graphical objects and electronic eye. Graphical objects can be independently rotated (around arbitrary axes), translated and scaled. Objects can inherit *transformations* (rotation, scaling and translation) from other objects. GrafSys supports dynamic (i.e. on-the-fly) and multiple inheritance of transformations and an unlimited number of so-called operators (matrices) per object.

The GrafSys provides objects for 3D points, lines and whole objects that can contain up to 8000 lines in full RGB color and more than 250'000 points. GrafSys also supports ultra-fast polygon filling using the triangulation approach. With it you can easily implement hidden-surface removal.

Designed for fast and simple to program animations, GrafSys supports an AutoErase feature where the object automatically erases its previous image before redrawing itself. For flicker-free animations GrafSys also provides easy to use Off-Screen bit map handling.

GrafSys is a combination of procedures and a powerful, extensible class library that can be easily curtailed to your specific needs. For example the general-purpose 3D object 'TSObject3D' understands over fifty messages for such diverse things as building a point/line database, rotating and drawing itself.

About this documentation:

This documentation describes how to use the software package 'GrafSys'. It is divided into four major parts. Part one 'General Discussion of 3D' describes the general principles of 3D visualization and how they apply in GrafSys. Part two 'How to use 3D GrafSys' describes how to use the GrafSys in your programs. Part three 'Implementation of GrafSys' provides an in-depth view on how certain aspects of the GrafSys were implemented and can easily be skipped. Part four 'GrafSys and GeoBench' describes the interface GrafSys provides for the GeoBench.

A Note To The Reader

GrafSys 2.0 is the much-improved descendant to GrafSys 1.x which has been available for some time. Many users of GrafSys requested the source code as to be possible to curtail it to their specific needs. For reasons that will become soon appearant, I had to decline their requests.

GrafSys 2.0 is a fully object-oriented 3D graphics library. The transition from 1.x to 2.0 was more than a simple translation. It was in effect my Diploma Thesis and as such required four months of full-time design and coding. Some aspects of the GrafSys might appear strange to you (e.g. the 200K point requirement or the whole part four of the documentation). Please keep in mind that the GrafSys was curtailed to meet specific needs (the XYZ GeoBench of the *Institut für Theoretische Informatik* at the *Swiss Federal Institute of Technology Zürich (ETHZ)*) and therefore some aspects of this work might not be what you expected from the 1.x version.

This documentation is a much more comprehensive manual than the one that came with the 1.x version, which many people commented positive on. It was fun to write but took quite some time, so please be sure to have a look at it. I hope you like it.

The GrafSys represents one of my greatest achievements so far (it was accepted as a diploma thesis and recieved top ranks). Since I would like you to share the fruits of my work, I will now release the GrafSys 2.0, this time along with the source code. The only source that I am not willing to release so far are the assembler versions of the ultra-fast triangle-drawing routine (the compiled code *is* included). However, since I have dedicated a whole chapter in part three to this topic and included the Pascal code for it, I think many of you will be able to code their own variants of it.
The reason for not publishing the source code is very simple: I'm using it for a current project that will hopefully enhance my financial position. If I'm done with it, I will also include the assembler sources.

Please be sure to read the Copyright Notice carefully as to avoid sad misunderstandings.

Have fun,
Christian Franz.

Part I

General Discussion of 3D

Overview

This part of the GrafSys documentation explains the concept of 3D visualization using a two-dimensional medium (such as your computer's screen). Then we will proceed to the fundamental entities and the operations on these. Finally we will discuss the electronic eye that is used to view a scene and define how objects are drawn.

Visualization

Basically, 3D visualization is a fancy name for something very common. If you take a photograph of a building and somebody tells you that the camera performed a three-to-two-dimensional viewing transformation he will probably earn nothing more than a funny look. But this viewing transformation is the heart of the package. It is really not very complicated once you grasp the fundamentals. What the camera did to produce a picture is very simple (at least as long we put the chemistry involving the film aside). It reproduces a flat (two-dimensional) representation of a three-dimensional object.

fig I.1 : Projection of an object

As you can see in fig I.1 the light passes along the edges until it reaches the film. This is called *projection* of an object. The light rays are called *projection beams*. There are different techniques for projecting an object: *parallel projection* and *perspective projection*. In parallel projection the projection beams are parallel. In the resulting image formerly parallel lines (in the 3D object) remain parallel and relative dimensions are preserved. The depth information is lost. Therefore the resulting image does not look realistic. This technique is often used for blueprints etc.

In perspective projection all projection beams converge in the so-called *Center of Projection*. This is where all the projection beams originate. Somewhere we interpose a *Projection Plane*, in our analogy the film. In the projection parallel lines only remain parallel if they are parallel to the viewing plane. Relative dimensions of the object are lost since lines close to the projection center appear longer than those further away. The center of projection is often called the *eye-point* or simply the *eye*. Perspective projection has a singularity that is easily explained. The closer you get to the eye-point the larger the projection gets. If your object extends into the eye itself its size becomes infinite. To avoid this situation the GrafSys will not draw anything between the eye and the projection plane (see 'clipping').

fig I.2a: parallel projection (left) and perspective projection (right)

For further description of the different eye and projection settings, please refer to the chapter 'Eye'.

Adaptation to the computer screen visualization is fairly straightforward. In computer space we use the screen as the projection plane. To facilitate things we let the projection plane coincide with the XY-Plane (i.e. the Z=0 plane). This way we can very easily draw the object using both techniques. Imagine we have a point with the coordinates [x,y,z] that we want to project onto the screen. To parallel-project the point to screen coordinates, we use the simple formula

h := xc + x   (xc is center horizontally of screen)
v := yx + y   (yc is center vertically of screen)

In parallel projection we ignore the z-coordinate and only use the x and y coordinates as offsets from the screen center.
In perspective projection we use the z-coordinate to modify the x any y values. The further away from the projection plane, the closer x and y

should be to xc and yc, respectively. This is of course very simple to accomplish by dividing the x and y values by the distance from the eye. Let d = distance of Eye from projection plane. Thus,

and when projecting we just use above values as with parallel projection:

$$h := xc + xp$$
$$v := yc + yp$$

The beauty of this lies in the usage of the d parameter. If we agree not to draw anything that falls behind the projection plane (i.e. the z component is negative), the z/d parameter can never approach -1 where xp and yp will become singular.

fig I.2b: The eye is always assumed behind the projection plane which can be regarded as the screen. No objects behind the projection plane are drawn

Again, please refer to the section about clipping for further discussion of this problem. For further discussion on how the GrafSys implements projection, please refer to Part III 'Implementation of GrafSys'.

Fundamental Entities

All of the above is nice and interesting but until now we have no way of defining what to transform. In GrafSys all objects are defined via *points* and *lines* in Cartesian coordinates. Since it is an object-oriented library you can easily change this to any other way you like. Keep in mind, however, that the basic transformation algorithm will always work with Cartesian coordinates so you will have to provide you own conversion algorithms. The following discussion assumes that you will be using the supplied TSObject3D object (described below) or it's descendants with it's methods.

Points And Lines

Almost all 3D Graphics with this package should be done with so-called *3D Objects* (these should not be confused with the OOP term 'object' which is simply a data structure). Although the package supports separate conversion and drawing of 3D Points and 3D Lines as well it is optimized to handle 3D Objects. These objects are usually a collection of points and lines that logically belong together. If you group all this data into one single object, drawing and transforming becomes a simple task and requires no additional headache (or sore fingers while programming) from you. The collection of points and lines in a 3D object is sometimes referred to as the 'object's database'.

Points

Points are defined in Cartesian coordinates, i.e. each point has three coordinates that indicate how far away the point from the origin is. The origin is assumed at [0,0,0]. Routines are provide to add to, delete from and change Points in the database. With objects you access points with reference numbers, i.e. the third, fourth or tenth point in the database. For help on how to define these points please refer to 'How to design a 3D object' in part II of this documentation.

Lines

Lines always connect two points. You specify the starting point and the ending point of the line. Routines for adding, deleting and changing lines in the database are provided. Lines are accessed through reference numbers very much like points are.

3D Objects are drawn by walking through the database drawing all lines incrementally (points that are not connected by lines are *not* drawn). Note that the sequence in which you specify the lines can affect the performance of the GrafSys. This is because the GrafSys is smart enough not to recalculate the starting point of a line if the previous line ends at the same point, cutting the overhead of moving the pen to a new location (in techno-speak the `MoveTo` Toolbox routine will not be called). Imagine we had to define a cube. The eight corners must be connected with twelve lines.

fig I.3: optimal sequence of lines (left) and worst sequence (right)

As you can see in fig I.3 the same object may be defined in many different ways. The definition on the left requires 4 `MoveTo` (at lines 1, 10, 11 and 12) and 12 `LineTo` calls while the definition on the right requires 12 `MoveTo` and 12 `LineTo` calls.

Polygons (Triangles)

The normal version of GrafSys does not support true polygons. However, GrafSys provides you with a special ultra-fast triangle fill procedure that you can use to implement hidden-line and hidden-surface removal. The easiest way is to subclass the `TSObject3D` (see below) to add these features. There is a demo program that demonstrates this in conjunction with Back-Face Removal.

Coordinate Systems

When defining many objects (e.g. chairs in a room) it would be quite tedious to go and measure the coordinates of each chair relative to a common origin and then enter those points into the database. Instead it is much easier to define an object in its own world where we can place the origin wherever we want to (this is especially important if we want to rotate the object as you will see later on). Using a technique called translation we will then move the object to its location (in this case the location in the room). To do this we use different coordinate systems: Model Coordinates and World Coordinates (there are more coordinate systems involved but these two are the only ones you must know about)

Model Coordinates

When you design an object, you usually instinctively place an origin (i.e. the point with the coordinates of [0,0,0]) somewhere and define all other points relative to this *object origin*. The origin of the cube pictured below is in its center as can easily be seen. Each object has its own origin. Therefore the coordinates of the points that model the object reside in the so-called Model Coordinate System (MCS).

fig I.4: Object origin (left) and coordinate system (right)

World Coordinates

When you design an object, you specify all points in the object's coordinate system. Then, when viewing it, you place the object somewhere in the world. You do this by specifying which point in the world would correspond to your objects origin. This coordinate system is called World Coordinate System (WCS).

fig I.5: Difference between World Origin and Object Origin

In the figure, the origin of the object was placed at the world coordinates [3,7,4]. If the object had a point with the coordinates [0,0,3] in Model Coordinates, the same point would have (after moving to [3,7,4]) the coordinates [3,7,7] in world coordinates. Usually this would mean you had to recalculate all your points. If you are using the 3D Objects this is done automatically.

Fundamental Operations

But why should you be confused with all these different coordinate systems? What is their use? Not only is it much easier to define an object in a local coordinate system. But the real advantage is apparent when you want to move, rotate or scale a single object without having to change everything else in the world. These actions (moving, rotation and scaling) are called transformations. The mathematics to this is quite simple and described in part III of the documentation. You do not need any understanding of the underlying math to use the GrafSys efficiently. If you change the state of an object somehow (by adding, deleting or changing lines or use any of below described operations on it) it remembers this and recalculates itself when necessary.

Translation

Moving the (local) origin of an object means that you move all other points of the object as well along with the origin for the same displacement. This is called translation. For example, if you drive your car one mile down the road, you 'translate' it one mile. GrafSys supports two different kinds of translations: relative and absolute. With relative translation you specify a vector (that is a direction and a distance) and the object will be translated from its current position in the direction and for the length of the vector.

fig I.6: Relative translation

Note however, that when specifying the translation vector you do not calculate the point from where (before) to where (after) but just define the *displacement* (i.e. displace five units in direction of x, six in y and one in z).

In absolute translation you again specify a vector. Now however, instead of displacing the object you place it at an absolute location specified by the vector. The result is a translation of the object from the worlds origin for the given vector.

fig I.7: Absolute translation using the same vector as in relative translation

Note that the displacement vector is always specified in the currently active coordinate systems. This is important to remember only when using the order-dependent transformations (see below). When translating an object that has previously been rotated, translation usually takes place in the rotated coordinate system. If you do not use order-dependent transformations (i.e. no FF operator, described below) translation is always performed in the model coordinate system.

Rotation

The most dramatic advantage of model coordinate systems is obvious when we rotate a single object in a scene (a scene is a collection of objects that are visible on the same screen). While translation of objects could easily be done through simply adding the vector of displacement to all points in question, rotation is not that simple. When you rotate something, the first question is not as you might expect 'how many degrees' but 'around what?' Rotation is always done around an axis (called the *rotation axis*) which is nothing more than a vector in space defined by two points.

The GrafSys supports four different forms of rotation: around the object's x, y and z axis and around an arbitrary axis. If you rotate around one of the main axes you do not have to specify the axis explicitly (because it is obvious).

Note that rotating an object is not as simple as it sounds and you might get different results from what you have expected. If you rotate an object 45 degrees first around the Z-Axis and then around the Y-Axis it does not mean that the Y-Axis of the second rotation is tilted by 45 degrees. Rather, the object is taken out of the coordinate system, rotated by 45 degrees and the result of this operation is *placed back* into the coordinate system and then taken out again to be rotated around the Y-axis.

fig I.8a: result of two sequential rotations

Some people might have expected the following result:

fig I.8b: Incorrect (but expected) result of two sequential rotations

But since this is dependent on which rotation you execute first, this would make it almost impossible to program anything with it, because you always have to know which operation was executed when.
To achieve above results you have to use the following technique:

fig I.8c: Achieving expected result through arbitrary rotation

As you can see, the ability to rotate around any axis gives you additional flexibility.

Rotation around an arbitrary axis is a bit more complicated since you have to specify two points P1, P2 around which to rotate. In this case you have to be careful to specify the two points in the correct sequence since the axis is always oriented looking from P1 to P2.

fig I.8c: Rotation around an arbitrary axis

If you interchange these points, the rotation is done in the inverse direction. Care must be taken when using arbitrary rotation since it consists of a mixture of translation and rotation. They cannot easily be undone and are applied only after the default-operator rotation and translations are done. Therefore, you should not use any of the default rotation or scaling if you use the arbitrary rotation commands except if you know exactly what you are doing. See 'Order Of Transformation', below.

Again, GrafSys provides routines for both relative and absolute rotation. Rotation around the main axes is done through the same routine while arbitrary rotations have their own.

Object rotation is given in **radians** (not degrees!).

> Note: to convert between radians and degrees, use the following:
>
> ```
>     const
>         degree = 0.01745329; (* π / 180 *)
> ```
>
> and multiply all your angles (given in degrees) with the constant. This will convert it to radians e.g.
>
> ```
>         alpha := 90 * degree;
> ```

Scaling
Another thing GrafSys lets you do is scaling an object. That is enlarging or reducing the object along any of its axes.

fig I.9: Normal cube (left) and scaled along one axis (right)

This *scaling* can only be done along the main axes (i.e. X, Y and Z). Note that different scaling factors along the axes can destroy the orthogonality of the coordinate system.

Order Dependencies

Usually, rotating an object is harder than it seems at first. More often than not, the results are not what you expect. This is because normally the rotations are done sequentially and not simultaneously. In that aspect this package is not different. First, the object is rotated around the X-, then Y- and finally around the Z-Axis. If you keep this in mind, you should not be surprised too often.
Still the problem of oder dependencies remains along with another one: Translating an object and then rotating the translated object yields a totally different result from rotating first and then translating the rotated object as the following example illustrates:

fig I.10: Order dependencies

Some graphics packages tackle this problem by defining the order once and for all (e.g. SubLogic's A23D1 3D Graphics Package), others let you define operators and leave the sequence for you to figure out (PHIGS, GKS). I have taken a combined approach. When displaying an object, GrafSys first rotates and then translates the object using the default built-in operators (also called '*default standard*' and '*default arbitrary*' operators, respectively). The normal translate, rotate and scale procedures (messages for the OOP folks) work on these default operators. Since this is not always flexible enough, GrafSys provides each object with an unlimited number of additional *operators* (actually just matrices) that can be linked to the object. It is up to the programmer what she/he does with it. For convenience the operator objects understand the translate, rotate and scale messages. This at first somewhat cumbersome approach has one distinct advantage that even PHIGS or GKS do not provide: using this and the provided inheritance mechanism you can build object hierarchies on-the-fly (i.e. dynamically) while your program is running and even change them. Also this approach allows multiple inheritance (if you ever should find a need for that).

Free Transformation

As described above, the GrafSys supports both default (i.e. fixed-order) transformation and so-called free-order transformation (also called 'FF transformation'). The usage is simple. You tell your object to allocate a new transformation operator (a matrix) and define all your operations on it. For example, if you needed first translation and then rotation you would first allocate a new operator, translate, then allocate another operator and then rotate. The object automatically keeps track of all allocated operators:

fig I.11: Free-order transformation operators

The operators are evaluated in the sequence they have been placed in the chain. Procedures exist to pre- and post-concatenate the operators. The above-mentioned inheritance works with these free-order operators. For more information on inheritance please refer to part II, chapter 'Inheritance'.

Order Of Transformation

Since it is important to know what operation is executed in what order, here is the sequence of transformation:

- first, the build-in rotations around x, y, and z are executed (in that order) using the '*default standard operator*'
- then the object is translated and scaled using the default standard translate/scaling operator
- then the arbitrary axis-rotations are applied
- then in the order the programmer specified the free-order operators are executed (if allocated)

Eye

Everything discussed so far would enable you to view scenes in 3D on your screen. If you want to see an object, simply move it until it appears on your screen. This happens when it's above the XY plane and close enough to the worlds origin. If you cannot see the object, move it around until you do. This is completely operable and suffices in most cases. However, this is like moving the mountain instead of going to it. It would be much simpler (from the programmer's point of view) if we moved the eye until we see the objects (in relativity terms this is of course the same since the underlying math will do the same if you specify an eye location, only automatically). If you open a new screen (or window) for 3D graphics, it defaults to switching the *eye* off. If you switch it on, you may specify a location from where you look at your objects.

Eye Coordinates

The eye is a bit more complicated to describe and you should first try to work without it. If everything works fine without it, turn it on to gain even more control over your scene.
If the eye is turned off, you are always viewing from below the XY plane straight up the Z-axis, the eye is located at the world's origin:

fig I.12 : The eye default direction

If you turn the eye on, you must specify a point in world coordinates from where you wish to look into the world. This is the *eye location*. Think of it as the place where you put the camera. Anything in front of it will be displayed normally. Objects behind the eye will be displayed either mirrored (if clipping is off) or not at all (if clipping is on).

There are two different sets of parameters that specify the eye (i.e. the point from which you look at the world). If you look at an object, the way it is displayed on the screen depends on several aspects:

- from which direction you look at it
- how far away you are from the object
- what projection type you are using
- what kind of electronic lens you have selected

If you regard the eye as a camera and the screen as the film the picture is projected on, things might become a bit easier to understand. First, the camera has to be placed somewhere in the world. You do this by specifying a location in the normal way (as a point) with [x,y,z] as its coordinates.

fig I.13 : Eye in action

After defining the point where the camera is set up, you specify how much the camera deviates from the Z-axis towards the Y-axis. This angle is called Phi. If you specify a phi angle of zero, the camera would be facing straight up the Z-axis, an angle of 90 degrees (remember to convert to radians before calling the routine) aligns the camera with the Y-axis, thus being parallel to the XY-plane:

fig I.14: Definition of phi

Next, with *Theta*, you tell the package how far you would like to turn the camera around the Z-Axis:

fig I.15: Definition of theta

A third angle, called *Pitch* defines, how far you would like to turn the camera around its viewing direction. An angle of zero means no pitch (i.e. parallel to the 'horizon')

fig I.16: Definition of pitch

The last parameter affects your graphics only if you are using perspective projection. It is called *Viewing Angle* and simulates the electronic lens. If you use small angles, your eye shows only a very small part of the world but enlarges it many-fold. This would be a 'Zoom Lens'. Large angles show a much bigger portion of the world, but these will be smaller and you have to get closer to enlarge them (just like in real life). Note that the viewing angle ranges from 0 to 180 degrees.

fig I.17a: Definition of viewing angle.

A viewing angle of *zero* tells the package that you want to switch to parallel projection (see below). The viewing angle defines how far behind the projection plane the eye resides. A large viewing angle (close to 90 degrees) places the eye close to the projection plane (wide angle lens) and a small angle places the eye far away (zoom lens). To calculate the actual distance the eye must know the size of the projection plane. Therefore the eye in GrafSys is tied to the window where you want to view the object.

fig I.17b: d parameter and viewing angle

If you re-size the window you should make sure to tell the GrafSys to recalculate the internal eye parameters. Otherwise you might get surprising results.

Eye Transformations

The eye can be moved around just like any other object. However, since a great deal of calculation is involved with moving the eye, you must keep track of the eye's position. There is only one central procedure to set the eye parameters. Therefore, if you update the eye it is not necessary to tell your objects that they have to recalculate themselves since they detect this situation automatically.

Viewing Options

Projection

The package supports two ways of projecting the objects: parallel and perspective. In perspective projection, things that are further away are smaller than those closer to the eye. In parallel projection, all lines on the screen remain the same length regardless how far away they are from the eye.

In perspective projection, all lines tend to shrink towards a certain point that is far, far away, the so-called *Vanishing Point*. Parallel lines usually do not stay parallel. In parallel projection, parallel lines stay parallel.

fig I.18a: perspective projection (left) and parallel projection (right)

In the above example you can very easily see that perspective projection is the way you are used to in normal pictures while parallel projection you probably know from floor plans or construction blueprints. To turn on perspective projection, pass a viewing angle that is not equal to zero. To turn on parallel projection, pass a viewing angle of zero.

fig I.18b: two perspective projections of the same object

Sometimes it might be useful to have a fixed camera location. In this case you can turn off the eye transformations. The eye will be fixed at location (0,0,0) and look straight down the Z-axis. Now instead of moving the camera, you have to move all your objects, but if you only have one object, this might be useful, since turning off the eye transformation makes recalculating the object a bit faster.

Clipping

An additional parameter *clipping* can be set. Clipping is tech-speak for eliminating those lines of a graphic, that 'fall off' the screen. To be more precise, it is eliminating those parts of a line, that fall off. Usually, there are three ways of clipping:

- None,
- eliminating those lines that fall off part-wise, and
- clipping them to the point where they penetrate the viewing plane.

It is very important to clip those lines that fall behind the eye or very close to it, since they behave very erratically there (try looking at your finger at about 0.2 inches from *your* eye and you will understand). Clipping is usually only useful in perspective projection.

fig I.19: Clipped cube. Left with clipping to projection plane, right with eliminating offending lines

In above example, the (perspective projection of the) closest corner of the cube has been clipped because it came too close to the eye location. GrafSys supports all three clipping methods, called *none* (no clipping at all), *arithmetic* (as in the left picture, above) and *fast* (right picture, above)

Drawing An Object

Objects in GrafSys are fairly smart. Sending them a *Draw* message causes them to draw themselves using all current rotations and eye settings. If a situation arises where the object must recalculate something either because its state or the eye changed the object detects this and does it automatically. The object can even erase its previous image if you want it to.
Before the object can successfully be drawn, there are quite some transformations going on behind the scene inside GrafSys. For a full comprehension of what is going on, read on. However, the following is only for the technically inclined and can easily be skipped since it is not essential for using the library.

The Eye Revisited

When describing the eye, we were actually talking about the projection plane. In reality the eye is just a tiny point and if we projected everything into the eye, we would end up with just a single black pixel and nothing else.

Instead, if we specify the location (and orientation) of the projection plane, we also define the location of the eye. The eye of course sits somewhere directly behind the projection plane and looks straight onto it. The projection plane has a variable size and usually is a rectangle inside one of your windows. The package draws onto the projection plane (i.e. inside this rectangle).

fig I.20: Eye Distance and projection plane

When we define the viewing angle, the package uses this angle in conjunction with the current projection plane size to calculate how far the eye would sit behind the projection plane (the *Eye Distance*). This way, if we re-size the projection plane (i.e. on a smaller monitor) the eye distance is recalculated and the scene scaled to fit into the new projection plane. In other words, no matter how big or small our screen (or projection plane), the same scene fits on it if you use the same view angle. Note that this is only true for perspective projections.

Since the difference between eye and projection plane is only of technical interest, we will use the word eye where we should have used projection plane (especially since 'eye' has only three letters).

Point Transformation Revisited

As I have pointed out, a lot happens to a point from the moment it is defined to the one it is drawn. As a matter of fact, this is probably the reason why you are using this package.
Anyway, to give you a better understanding on what goes on behind the screen, read on (you may skip the next paragraphs if you are easily bored).

If we define an object, we define all the points in a coordinate system that is special to this and only this object. As mentioned before this is called the Model Coordinate System (MCS). Now, if we transform the object (rotate, translate or scale it), the object's points are changed to other positions. However, since all points within the object remain in the same position relative to each other, we say that the MCS gets transformed.
The new locations of the various points are transformed according to our translation, rotation and scaling settings into a new coordinate system called the World Coordinate System (WCS).
After they are transformed, the points are projected onto the screen. These (now two-dimensional) points reside in the *Screen Coordinate System* (SCS).

fig I.21: The different coordinate systems and they relation

The graphic package supports all different coordinate systems. With special routines we can access the MCS, WCS and SCS representation of a specific point.

Although it seems that the WCS is the final coordinate system before the points appear on the screen, this is not always the case. If you are using the Eye, the points are transformed yet another time into the *Eye Coordinate System* (ECS). This is very important to remember.

fig I.22: Points are always projected onto the XY-plane

The package always uses the XY plane as the projection plane and rotates the WCS according to the eye settings. This means that instead of moving the screen that you project on in the world, we rather move the world around the screen.
If you are not using the eye, ECS and WCS are the same. Everything is plotted looking up the Z-axis. If we are using the eye, the points from the WCS are transformed again according to the eye settings.
However, whenever you request transformations into WCS, you will automatically receive ECS if you are using the eye.

Hidden-Line and Hidden-Surface Animation
GrafSys does not provide you with full-blown hidden-line or hidden-surface removal. Since there are so many different methods readily available, it rather provides you with the necessary tools to do so and leaves it up to the programmer to implement her/his method of choice.
GrafSys provides you with routines to test if a surface can be seen from the current eye settings (Back-Face Removal), routines to draw triangles very fast and routines for using off-screen pixel maps. Combined, these can be used to implement simple, efficient and hidden-surface animations.

Part II

How to use 3D GrafSys

Overview

Part II of this documentation tackles the more technical aspects of the GrafSys. This part is strictly for programmers that intend to use the GrafSys.

First we will see a short run-down on how to efficiently design a 3D Object on paper before entering it into the data base. Then we will discuss the procedures and objects the GrafSys provides you with. Starting with the global procedures that handle 3D GrafPorts and windows on the Macintosh operating system we will then continue on to a short introduction into Off-Screen Pixel Maps. After that follows the description of the class hierarchy of the different objects in GrafSys and the messages they understand, focusing mainly on the central object *TSObject3D* and how to use the state inheritance feature the GrafSys supports.

Finally you will find a full documentation on all the messages the objects in GrafSys support and a description of the few global procedures. This part concludes with some advanced topics and caveats plus a short example on how to extend the GrafSys.

How To Design A 3D Object

Since designing an object involves bringing it down on paper first, many people experience some difficulties at first. This often comes from the fact that paper is a two-dimensional medium while our objects are three dimensional.

fig II.1 Cube in parallel projection, left rotated, right with rotation of zero

When drawing a 3D object on paper, points that were unique in space become ambiguous on paper. Especially in parallel projections, as can be seen in above figure. If you look at the cube on the right side, you notice that at each corner two points come to lie on top of each other. If I were to point on one, you would not know which one I mean, the one in front or the one in the back. What we have to do is to draw *two* sketches of the same object, looking from different sides, so every point has two distinct positions, one in each sketch. While in each sketch two points can still overlap, no two same points overlap in both sketches. Although you can pick almost any two views, it is wise to choose special sketches: the top view and one of two side views.

fig II.2: parallel-projecting an object onto two separate planes

If we now number all corners and project them onto the two sketches, we will come up with something like this:

fig II.3: The cube as two sketches and in 3D

As you can see, no two points fall onto the same point in both sketches. To get each points coordinates, all you have to do is look it up in each sketch and read off the coordinates as you would do with any normal 2D-Graph.
There is something very important to remember that becomes obvious if you look closely: both graphs have one common axis (here it is the X-axis). A point must *always* have the same coordinates on the common axis of both sketches. If it does not, you have made a mistake. This is an easy way to proof your sketch.
You might have noticed that in order to produce the sketches we used the XY and the XZ plane. As you know, there is also the ZY plane. Yes, you could have used this one instead of the XZ plane. In fact, you can use any combination of two of the tree planes to generate the sketches.

This object's origin (the point with the coordinates [0,0,0]) lies outside the object. Try locating it. While it sometimes might be useful to place the origin outside an object, remember that the object will rotate around its origin, and not its center as you might perceive it. In our demo object above, we also use a cube. This is the sketch that I used to produce the coordinates:

fig II.3 Cube sketch and origin, 3D view of same cube

As you can see, there is no problem if coordinates have negative values. As another example, look at the sketch of a house. Note how in the 'Front View' below you can neither tell where the smokestack nor the windows are located. Only the 'Top View' can clarify this.

fig II.4: Two sketches to define a house object

However, in the top view you cannot see what the windows look like. Conversely, the front view does not show that the first window from the left appears on *both* sides of the house. But both sketches taken together define every point.
Note also that windows that happen to be on the left or right wall (as seen from the top view) would show in neither sketches. In this case it might be necessary to draw another (third) sketch to define the remaining windows.

Using GrafSys

General Usage

To use the GrafSys you simply include the GrafSys library files into your project and copy the resources into your project's resource file. Then you have to decide if you want to use the library 'as is' or if you need to extend the objects provided. In general you only need to extend the objects (usually the TSObject3D) if you have complicated objects that have parts that move relative to each other such as a robot arm with one or more flexible joints etc. GrafSys supports this through the use of a hierarchy that enables the programmer to dynamically allocate and deallocate hierarchies of objects while the program is running.

Program Structure

A typical program has three distinct parts:

- Initialization of the GrafSys package, opening 3D windows and initializing the eyes
- Allocating objects and building (loading) their database, setting their attributes (such as AutoErase etc.)
- Animating or simply drawing the object. Note that GrafSys allows you to change the object's database even after their initial loading/building is done

For the first step, 'Initialization' you would use the general procedures that are not object-oriented but classic Pascal procedures. Note that you must initialize the GrafSys before you build any 3D objects. Note also that you must have opened at least one 3D Window and initialized the eye before you may call any of the object's transformation methods or those that call them indirectly (such as `Draw` or `fDraw`, see below). If you open a 3D window an eye is automatically attached to it. One eye is both the minimum and maximum number of eyes you can attach to it. You cannot subdivide a window into two or more 3D ports that have different eyes attached (well you *could* actually if you try really hard by copying the data structure and assigning a different eye to a portion of it - don't do it! There is another approach that is described in the 'Caveats' chapter). Procedures exist to manipulate the eye in many different ways. See 'Using The Eye', below.

The general-purpose object TSObject3D sports a so-called 'AutoErase' feature to facilitate easy animation. If you decide to use this feature you do not have to keep track of the position and size of your object where it was drawn last. Simple animations are a snap if you use it since the `Draw` messages will then automatically erase the previously drawn image. Without going into further details let us look at a simple code fragment that rotates a cube around its X-axis in front of you:

```
var
  theCube : TSObject3D;
  theWindow : WindowPtr;

begin
InitGrafSys; (* initialize the package *)
  theWindow := GetNew3DWindow(cTheWindow, pointer(-1));
SetVector4(EyeLoc, 0, 0, -500);
SetEye(TRUE, EyeLoc, 0, 0, 0, 90 * degrees, fast);
   (* now eye and the package are ready to use   *)
   (* the 3D graphics will be drawn in theWindow *)

  New(theCube);     (* allocate space for the 3D object *)
  theCube.Init;          (* init me *)
  BuildObject(theCube);  (* Build object database *)
  theCube.SetAutoErase(TRUE);
   (* now the cube is ready to animate *)

  repeat
     theCube.fDraw; (* erase it and redraw it*)
     theCube.Rotate(5 * degrees, 0, 0);
  until button;
end;
```

Number Representation In GrafSys

Since not all Macintosh Computers are equipped with a Floating-Point Processing Unit (FPU) you have to decide which version of the GrafSys you want to use. There are two available. The normal version works with real-numbers and direct calls to the FPU. This is the fastest. The other version (GrafSys.fixed) uses fixed-point arithmetic. This way the GrafSys still delivers an acceptable speed but overall performance is visibly reduced. Use the fixed-point version whenever you want to use your program on any Macintosh, the other one for development and when you are sure that it is only used on FPU-equipped Macs. Programs written for FPU that are run on machines without it are known for spectacular crashes.

Whatever internal number representation is used is of no importance to your code since the interface to the core routines will do any conversion for you. To the outside world the GrafSys always seems to work with real numbers so you do not have to change a single line of code if you switch GrafSys versions.

GrafSys uses a special data structure called *Vector4* to communicate some internal number data. You should *never* assume anything about the number format in this variable. *Always* use the supplied conversion routines `SetVector4` and `GetVector4` for accessing its contents. If you do not follow this advice you program will not compile if you switch between versions of the GrafSys.

Resources

If you use the supplied TSObject3D object in your programs you can store and retrieve the object's database into and from resource files [InsideMac, ResEdRef]. Note that only data definitions but not operator definitions are saved to resource. GrafSys uses two resource types:

- `3Dob` for storage of point, data and polygon definitions
- `lClr` to store line-color information.

For any given object the IDs must match, i.e. if your `3Dob` resource has the ID of 1234, the `lClr` resource must have the same ID or it will not be loaded.

General Procedures

This section describes procedures supplied with the GrafSys outside the object definitions, such as number conversion or procedures to convert 3D points to 2D screen coordinates.

```
procedure InitGrafSys
```

This routine must be called before you can call any other routine or method in the GrafSys package. It initializes certain data structures that are required for all other operations. If you do not call InitGrafSys be prepared for some especially nasty nil-Trap crashes (if you are lucky).

```
function InterpretError (theErr : integer) : Str255;
```

The different GrafSys methods can produce a variety of error codes. If any of the supplied objects returns an error and you need a description in written text, call this procedure. Note however, that all GrafSys errors provide a method to display an alert with the current error number and description so you need this function only if you want to override these methods or write to a log file. The following error codes are currently defined:

```
const
cNoFFallocated          = -1;
cOutOfMem               = -2;
cBadMethodCall          = -3;
cNothingToInherit       = -4;
cTooManyPoints          = -5;
cIllegalPointIndex      = -6;
cTooManyLines       = -7;
cIllegalLineIndex       = -8;
cCantDeletePoint        = -9;
```

```
cNotOwner                    = -10;
```

```
    cBadFF                    = -11;
    cBadFFType                = -12;
    cCantLoadRes              = -13;
    cNo3DWindow               = -14;
    cCantCreateOffscreen      = -15;
    cCantChangeOffscreen      = -16
    cNoOSAttached        = -17;
    cCantUseWindowCLUT   = -18;
```

```
function GrafSysVersion: longint;
```

Use this function to determine the current version of the GrafSys. The high-order two bytes contain the major release number, the low-order two bytes the minor release number. Thus the hex number 00010007 would mean release 01.07. Likewise the number 00020201 means a release of 2.21. A non-zero first byte means alpha (01) or beta (02) release, e.g. 01020000 is GrafSys version 2.00α.

As mentioned before, GrafSys supplies procedures to convert real numbers to the internal number format so you can pass on or retrieve information from your objects.

```
procedure SetVector4(var v : Vector4; x,y,z : real);
```

Use this procedure to convert three coordinates in real-number representation to the internally used 3D point definition. You should never try to access v directly yourself or try to 'smartly' take advantage of some alleged knowledge of its contents.

```
procedure GetVector4(v : Vector4; var x,y,z : real);
```

This is the counterpart to SetVector4. Use it to convert an internal-number format to real numbers. You should never try to access v directly yourself or try to 'smartly' take advantage of some alleged knowledge of its contents.

```
function GetNewObject (theObjectID: INTEGER)
     : TSObject3D;
```

GetNewObject allocates memory and initializes an object like NewObject and then tries to read in a resource of type `'3Dob'`

with the specified ID. This resource contains all points, lines and polygons for this object and they are copied into the object.
If this was successful, it tries to load and locate a `'lClr'` line-color definition resource with the same ID.
GetNewObject returns the newly created object.
If the resource you specified does not exist, GetNewObject returns an empty initialized Object.

```
function GetNewNamedObject (theName: Str255)
    : TSObject3D;
```

GetNewNamedObject is the same as GetNewObject except that it tries to read a resource with the specified name. If the named resource has been loaded successfully, GetNewNamedObject will look for the 'lClr' resource with the *same ID* (i.e. the name is irrelevant!) as the '3Dob' resource.

```
procedure SaveObject (Obj: TSObject3D; theName:
    Str255; ID: integer);
```

Given an object, SaveObject writes the objects point, line and polygon definitions to the current open resource file into a resource of type '3Dob' with the given ID. Then it writes out the 'lClr' line-color information resource with the same ID as the 3Dob resource.

Warning: If a resource with the same ID already exists, it gets replaced.

The parameter theName defines the name the resources will have.

```
procedure SaveNamedObject (Obj: TSObject3D;
    theName: Str255; var ID: integer);
```

Same as SaveObject except that the name is significant for saving. The procedure returns the ID that was assigned for the resources.

Warning: If a resource with the same name already exists, it gets replaced.

```
procedure FillTriangle (p1: Point; p2: point;
    p3: Point; theColor: Integer; useQD: Boolean);
```

FillTriangle is a highly specialized routine that uses an ultra-fast algorithm to draw the triangle defined by p1, p2 and p3 on the currently active GrafPort. This routine is written for 8-bit 'deep' devices (mainly off-screen pixel maps, see below) and can either write to off-screen pixel maps or directly to the screen. If the currently active port is set to something other than 256 colors/grays (8 bit), then the normal QuickDraw `PaintPolygon` procedures are used.
If you use FillTriangle to draw directly to the screen make sure that the currently active port is frontmost (i.e. is not obscured by anything) and that you bracket the `FillTriangle` calls with `ShieldCursor` and `ShowCursor`.
TheColor is the direct pixel color you want the triangle to have. If you use the RGB color space, use `Color2Index` to convert the RGB color to the closest match.
UseQD can be used to override the fast polygon drawing procedure and use normal QuickDraw calls instead.

Use FillTriangle to build your hidden-surface animation techniques.

```
function IsVisible (k, l, m: Vector4): Boolean;
```

IsVisible checks to see what side of the plane defined by the three points k, l and m is looking toward the eye. If IsVisible returns TRUE, the front side is showing, if it returns FALSE, you are looking at the back side. IsVisible is usually used for back-face removal on convex polyhedron. In order to function correctly, you must order k, l and m correctly, i.e. clockwise:

If you look at the cube pictured above, you define the sides that face outside by writing the direction of the clock on it and then specify the points in that order. For example, if you wanted to define a cube with surfaces that all face outside, you would have the following definitions:

Front: A, B, C
Back: D, F, E
Bottom: D, A, C   etc.

(of course you can rotate the points, only the sequence must be clockwise, CAB would define the same front just as BCA). It helps if you affix the vector pointing outside to find the correct orientation.
Use IsVisible to build your hidden-surface animation techniques.

Using The Eye

One of the central elements of a graphics package is usually the eye. A good Graphics package must be easy to control and yet be flexible enough to satisfy almost any needs. This is of course impossible if you want to have adequate performance. Implementing a m-eye-with-n-projection-plane system would not be much more work but degrade performance to a minimum while making programming (i.e. using it) a nightmare. As will be pointed out later the eye is tied to a very common Macintosh data structure (the window) that is predestined for it and limited to one eye per window. This enables the programmer to use 3D windows as any other windows and it makes the few additional routines to manipulate the eye intuitive to use.

If you use the provided drawing methods you do not have to keep track of those objects that need to be recalculated if the eye has changed since the objects detect this situation automatically.

Mac Windows

The eye is always attached to a 3D window. Although you can use any Macintosh Toolbox routine (such as `SetPort` or `TxFont`) on a 3D window, the reverse is not true. You cannot attach an eye to a normal Mac window yourself. This is because GrafSys attaches its own information to the window data structure where the Mac OS would not disturb it. For a description of this technique, please refer to part III of this documentation. 3D windows are in one aspect different to normal windows. The origin is initially placed in the center of the window (the Mac places the origin in the upper left corner) and the positive Y direction is up (down on the Mac). However, this only applies to 3D points and lines. If you use normal QuickDraw routines to draw in a 3D window you will find everything as usual.
Like QuickDraw the GrafSys uses an internal pointer to the currently active 3D GrafPort. Likewise some routines affect the currently active 3D GrafPort. Care must be taken since this is not necessarily the currently active QuickDraw `GrafPort`. Be sure you know what you are doing if you mix the normal `SetPort` and `Set3DPort` routines. For further discussion of this see the description of the `Set3DPort` routine and the 'Caveat' chapter, below.

All GrafSys drawing takes place in normal Macintosh windows that have been extended to support 3D graphics. As mentioned in part I of this documentation the eye-distance (i.e. the distance between eye and projection plane) is calculated using the view angle parameter and the size of the projection plane. When you open a window the projection plane is assumed to be the same size as the windows content region. GrafSys distinguishes between *ProjectPlane* and *ViewPlane*. All drawing takes place inside the ViewPlane while the ProjectPlane is used to transform the object and calculate the eye-distance. Usually the ViewPlane is a rectangle inside the projection plane. The center of projection is placed in the center of the projection plane. Routines exist to relocate the origin and to move and resize both projection and view plane.

GrafSys provides routines to allocate windows analogous to those found in Inside Macintosh with the only restriction that you can no longer specify your own storage space for window data but must let GrafSys allocate space for the GrafPort storage (if you always passed `nil` as reference to `wStorage` in your GetNewWindow or NewWindow calls you do not have to change anything). When allocating a new window, GrafSys will always allocate a CWindow. This should have no effect on your programs except that it will not run on the Mac 512K.
When you are done using a 3D window, you can use the normal Toolbox `DisposWindow` procedure (although this is not recommended if you are using off-screen pixel maps, described below). All memory allocated, including the memory reserved for the eye, is deallocated.

Use `GetNew3DWindow` and `New3DWindow` to allocate 3D windows. To set the 3D GrafPort to a certain 3D window use the `Set3DPort` procedure. To check if a certain window is a 3D window use the `Is3DPort` function.

The 3D GrafPort Data Structure
The 3D GrafPort is an extension to the normal CGrafPort (Macintosh Toolbox standard data type). You should never access it directly but use the procedures provided. Since GrafSys uses a technique called 'piggy-back data' to impose its structure on the normal QuickDraw GrafPort data structure, you must type-cast a `WindowPtr` to GrafSys `TPort3DPtr` to access the fields. No guarantee is given that the data structure is really there so you must be quite sure about what you are doing.

```
Type
TPort3DPtr = ^TPort3D;
TPort3D = record
     theWindow: CWindowRecord;
     versionType: OSType;
     theOffscreen: WindowPtr;
     ProjectionPlane: rect;
     ViewPlane: rect;
     left, right, top, bottom: integer;
     center: point;
     useEye: Boolean;
```

```
EyeKoord: Vector4;
```

```
          ViewPoint: Vector4;
          phi, theta, pitch: real;
          ViewAngle: real;
          d: real;
          MasterTransform: Matrix4;
          projection: ProjectionTypes;
          clipping: ClippingType;
          versionsID: longint;
      end;
```

Name	**Type**	**Description**
theWindow	CWindowRecord	QuickDraw CGrafPort data structure
versionType	OSType	Used for identification of 3D GrafPort
theOffscreen	TOffscreenRec	Record structure that holds information that the off-screen package uses. In Effect it contains pointers to the off-screen port and GDevice
ProjectionPlane	rect;	Logical size of the projection plane.
ViewPlane	rect;	Logical size of the view plane.
left	integer	Left coordinate (local) of view plane. Internal use only
right	integer	Right coordinate (local) of view plane. Internal use only
top	integer	Top coordinate (local) of view plane. Internal use only
bottom	integer	Bottom coordinate (local) of view plane. Internal use only
center	point	Coordinates of the logical center of projection in local window coordinates.
useEye	Boolean	Tells transformation methods to use the eye settings for transformation.
EyeKoord	Vector4	Eye location in world coordinates.
ViewPoint	Vector4	(not used)
phi	real	Phi parameter for eye as described below
theta	real	Theta parameter for eye as

		described below
pitch	real	Pitch parameter for eye as described below
ViewAngle	real	ViewAngle parameter for eye as described below. A view angle of zero means parallel projection
d	real	Distance of eye behind projection plane
MasterTransform	Matrix4	Eye transformation matrix
projection	ProjectionTypes	Projection type used with this 3D GrafPort. Either parallel or perspective
clipping	ClippingType	Clipping type used with this 3D GrafPort. Valid types are none, arithmetic and fast.
versionsID	longint	Used by the objects to detect a change in the eye settings.

Operations Affecting the 3D GrafPort

```
function GetNew3DWindow (ID: integer; behind: ptr)
    : WindowPtr;
```

Use this function to open a new 3D window. The Mac tries to load a 'WIND' and corresponding 'wctb' (optional) resource with the number ID. Behind points to a window behind which this window is to be opened. If you want to open the window in front of all windows, pass pointer(-1) as parameter. The currently active 3D Port is set to this window as well as QuickDraw's current GrafPort.

The eye is automatically initialized to [0,0,0,0,0,0], parallel projection and clipping to none. The center of projection is set to the center of the window's portRect. The eye is switched off.
GetNew3DWindow returns a pointer to the newly opened window.

```
function New3DWindow (boundsRect: Rect; title:
    Str255; visible: BOOLEAN; procID: Integer;
    behind: WindowPtr; goAwayFlag: BOOLEAN; refCon:
    longint): WindowPtr;
```

New3DWindow opens a new 3D window. The parameters are the same as for NewWindow as described in Inside Macintosh. Note that there

is no wStorage pointer since New3DWindow always allocates memory for the window itself.
The currently active 3D Port is set to this window.
The eye is automatically initialized to [0,0,0,0,0,0], parallel projection and clipping to none. The center is set to the center of the window's portRect. The eye is switched off.
New3DWindow returns a pointer to the newly opened window.

```
procedure Dispos3DWindow (theWindow: WindowPtr);
```

This procedure releases the memory associated with the 3D window pointed to by theWindow. If you do not use off-screen buffering (as described below) you might as well use the normal DisposWindow procedure to close, remove and deallocate the window.

Note: If you use off-screen buffering it is safer to use this procedure since it will also release the off-screen buffer associated with this 3D window (if it was allocated).

When drawing 3D entities into windows GrafSys uses an internal data structure called *current3Dport* that points to the currently active 3D port similar to QuickDraw's current GrafPort variable. The differences are subtle. Since GrafSys uses the Mac Toolbox calls to draw lines it *draws* using QuickDraw's current GrafPort variable. However it *transforms* according to the eye settings of the currently active 3D port. Since a 3D port holds information about the eye setting, projection types and clipping it is important that you always set the currently active 3D port to the 3D window you are using. However, QuickDraw's current GrafPort and GrafSys current 3D port do not necessarily have to be the same since calling QuickDraw's `SetPort` routine does not affect the setting of the current3Dport. How to use this feature to achieve some effects that are otherwise impossible (e.g. drawing 3D images on non-3D windows) will be described in the 'Caveats' chapter of this documentation.

Keep in mind that with using Set3DPort you specify the currently active eye settings that are attached to the 3D Port. All transformation and drawing routines use this information.

```
procedure Set3DPort (the3DPort: WindowPtr);
```

Set3DPort sets the current 3D GrafPort to the one specified. For all following transformations, this 3D port's settings are used. Set3DPort calls `SetPort` (Mac Toolbox) to set QuickDraw's current GrafPort to the window you specified.

Note: When you change the 3D GrafPort you also change to the eye settings associated with this window.

Note: If you specify a window that is not a 3D only QuickDraw's current GrafPort will be changed. To find out if a window is a 3D window, use the `Is3DPort` function described below

```
procedure Get3DPort (var the3DPort: WindowPtr);
```

Returns a pointer to the currently active 3D GrafPort. Note that this does not necessarily mean that this is the currently active window (i.e. the GrafPort QuickDraw draws into). This procedure merely returns the window whose eye setting GrafSys is using. Should Get3DPort return `nil`, trying to draw using GrafSys will result in a crash. In this case you have either not allocated a 3D window or just deallocated the currently active 3D port. This is analogous to trying to draw into a window that has been disposed of.

```
function Is3DPort (thePort: WindowPtr): Boolean;
```

Use this function to find out if a window is a 3D window or not. It returns TRUE if GrafSys allocated the window, FALSE otherwise.

```
procedure SetView (ProjectPlaneSize,
    ViewPlaneSize: Rect);
```

SetView sets the size of the projection plane and view plane. The projection plane is used to calculate the various perspective parameters. The view plane rectangle specifies a clipping region for drawing. Note that unlike QuickDraw, GrafSys places the origin of its coordinate system for drawing in the center of the viewing plane.
The center of projection is set to the center of ProjectPlaneSize.

Note: Although the coordinate system for drawing is the center of the viewing plane, ProjectPlaneSize and ViewPlaneSize should be given in the window's local coordinates as described in Inside Macintosh:

fig II.5: ProjectPlane, ViewPlane, Center of Projection and Window Origin (0,0)

Note: This procedure operates on the current active 3D GrafPort and Quic

GrafSys, off-screen pixel maps are always tied to a 3D window. Any 3D window may only use *one* off-screen buffer. The off-screen buffer is always the *same size* as the 3D window and uses the same coordinate system. The off-screen buffer always uses *256 (8-Bit) color*, no matter what your 3D window uses.

To use off-screen PixMaps you have to follow six simple rules:

- Allocate an off-screen buffer for a 3D window using the `AttachOffscreen` procedure. This creates the off-screen buffer and attaches it to the 3D window data structure. Since the off-screen buffer is always 8 bit 'deep' (i.e. uses 256 colors/grays), it will use a significant amount of memory. For example, a window of 640 by 320 pixels uses 200K of memory. Make sure that there is enough memory available.
- When resizing the 3D window, resize the off-screen buffer using `ChangeOffscreen`.
- To begin drawing to the off-screen pixel map, call `BeginOSDraw`. From now on all QuickDraw drawing commands draw to your off-screen buffer. When done drawing to the

off-screen buffer, call `EndOSDraw`. Now all drawing will be done to screen again.

- To copy a portion of the off-screen buffer to your window, use `CopyOS2Screen`. This will transfer a rectangular portion of the off-screen buffer to the screen. If the screen uses a different bit depth or color table, the colors are converted to their on-screen representation.
- If you are done using the off-screen buffer, use `CloseOffscreen` to deallocate the buffer. If you forget to do this you will end up fragmenting the heap and probably run out of memory soon.
- If you need a fast routine to erase a whole off-screen pixel map, you can use the `FastPixErase` routine.

Therefore all you normally have to do to add off-screen buffering to your animations is to bracket your normal animation code with BeginOSDraw and EndOSDraw calls and add a CopyOS2Screen command.

Normal Animation Code

```
...
repeat
       theObject.Erase
       theObject.Draw


until HellFreezesOver
...
```

Off-Screen Animation Code

```
(* Offscreen buffer must have *)
(* been allocated *)
...
repeat
       BeginOSDraw(the3DWindow)
       theObject.Erase
       theObject.Draw
       EndOSDraw(the3DWindow)
       CopyOS2Screen(...)
until HellFreezesOver
...
```

Off-Screen Handling Routines

To simplify off-screen buffer handling, GrafSys imposes some rules that normally do not apply to off-screen pixel maps. Firstly, off-screen pixel maps are always 256-colors/grays. Secondly, the buffer should always be the same size as the window it buffers (this is not mandatory but you should follow this guideline). Thirdly, the off-screen buffer procedures only work on 3D windows, not on the normal Macintosh window.

The off-screen package defines some error codes that can be interpreted using the InterpretError procedure the GrafSys provides. The following error codes are defined:

const

```
cNo3DWindow              = -14;
cCantCreateOffscreen     = -15;
cCantChangeOffscreen     = -16
cNoOSAttached       = -17;
```

```
    cCantUseWindowCLUT  = -18;
```

```
function AttachOffScreen (theWindow: WindowPtr;
    theColors: CTabHandle): integer;
```

AttachOffScreen creates an off-screen buffer for the 3D window pointed to by theWindow. theWindow must be a 3D window. If it is a normal Macintosh window, the procedure will exit without allocating a buffer. The buffer allocated will always use 256 colors/grays.
theColors specifies the color look-up table (CLUT) to use for the off-screen buffer. If you pass -1 (as you normally would), the off-screen buffer uses the same CLUT theWindow uses. If you pass -2 as argument, AttachOffScreen will load the default 256 color CLUT the system provides.

If successful, AttachOffScreen returns noErr and the buffer is allocated and attached to the 3D window. Otherwise it will not allocate anything and return an error-code that can be interpreted with the standard GrafSys InterpretError procedure.

```
function ChangeOffscreen (theWindow: WindowPtr;
     theColors: CTabHandle): integer;
```

Use this function whenever you resize the 3D window po to the Point and Line subclasses and a class of its own. Subclassed twice more, it knows first about a collection of points and then about a collection of lines over the point collection. The latest incarnation, TSObject3D is the single most powerful object in the class hierarchy and should be the base of most of your class extensions.

How To Use TSObject3D

Since TSObject3D is so powerful it is important that you fully understand its abilities and how to work with it. Built-in into this class is a database that may contain up to 250'000 points and (currently) 8000 lines. Special algorithms implement high-speed access to this tremendous amount of points while still minimizing memory allocation.
A TSObject3D can inherit transformation sequences from other TSObject3D objects and knows how to draw itself on the screen.

If you want to use the TSObject3D there are only a few simple steps to follow and you can have fast and simple animation:

- Before allocating any 3D objects, initialize the GrafSys package using `InitGrafSys` and open at least one 3D Window using Get3DWindow or GetNew3DWindow. If you want to use it, initialize the eye using `SetEye`. If you do not do it, remember the eye defaults to parallel projection, no clipping and the eye switched off (which has no effect on clipping and projection type).
- Directly after allocating the object, pass the `Init` message.
- Build the point and line database. GrafSys supports a way to store and retrieve this database in resources so it might be a good idea to design an object with an object-editor and then simply import the data through resources keeping the code small.

- Set the object's attributes such as AutoErase etc.
- Animate the object by calling the `Draw` or `fDraw` messages repeatedly.

Cloning

All objects understand the `Clone` message. This message will cause the object to produce an exact copy of itself. While this would normally simply mean a call to the Macintosh Toolbox `HandToHand` procedure, things are actually not as simple as it seems. Since the TSObject3D was designed to hold a very large amount of data it allocates memory dynamically in order to minimize memory waste. If you clone a TSObject3D all dynamically allocated buffers for point and line information will be cloned as well (automatically). The same goes for the user-installed operators that are equally cloned. Note that there is a slight irregularity involved if you use inheritance as will be explained in that chapter, below.

Killing

The first thing you need to know after allocating an object is how to ever get rid of it again. Since a Pascal `Delete` command would not deallocate the different buffers the `Init` method allocated, you need a safe way to dispose a TSObject3D. Invoking the Kill message will cause the object to first deallocate all buffers used for points, lines and additional operators (see below). In case another object inherited from this object it will be killed as well. See *Inheritance*, below.

Defining Points

Points are defined by passing their model coordinates as real numbers to the point-defining procedures. To retrieve them they are referenced through index numbers. You should never assume anything about the internal data format in which the points are stored or where to find them since this can change with different versions of the GrafSys. Points *may be added, deleted or changed between successive draws* of the object. Relevant messages are `AddPoint`, `DeletePoint` and `ChangePoint`.

Defining Lines

Lines are defined by passing the reference numbers of the two points the line connects. You should never assume anything about the internal data format in which the lines are stored or where to find them since this can change with different versions of the GrafSys. Lines may be added, deleted or changed between successive draws of the object. Relevant messages are `AddPoint`, `DeletePoint` and `ChangePoint`.

Color

GrafSys supports the full RGB color space. Each line can be assigned a color. Since it is sensible to assume that if a line has a certain color the next line will have the same (object coherency), you only specify the lines where you *change* the color. This means that if you told a line to change its color (e.g. to red) all subsequent lines (i.e. all lines that have a higher index number) will have the same color until another line changes its color.
Note that this can lead to problems if you delete a line that changes it's color since that information is lost. TSObject3D does not check to prevent this since this effect could well be intentional.
TSObject3D provides methods to change the line color and reset it (this means that the line should be drawn in the previously selected color). The default color for lines is black (RGB black). This means that if you do not ever change color in your object, the whole object will be drawn in black lines. Relevant messages are `GetLineColor`, `ChangeLineColor` and `KeepLineColor`.

Transformations

The TSObject3D understands two sets of transformations which are identical except that they work on different operators (matrices). Transformations are messages to the object to rotate, scale or translate. Rotation, scale and translation can be either in/decremented or set to absolute values. Rotation can be done around the three major axes (X, Y, Z) and around any arbitrary axis. Scaling can only be done along the three major axes.

Fixed-order (default standard) Transformations

GrafSys distinguished between default-order transformation that work on the built-in operators and free-order transformations that work on the user-allocated operators. The built-in operators (also called default standard operators) are executed in the following order rotation (X first, then Y, then Z), translation, scaling, arbitrary rotation. Relevant messages are `Translate`, `SetTranslation`, `Rotate`, `SetRotation`, `Scale`, `SetScale`, `RotArb` and `ResetArb`.

Free-order (optional) Transformations

After evaluating the arbitrary-rotation operator the free-order operators are applied. Free-order operators are used when you want to deviate from the predefined order of transformation or want to inherit transformations. The free-order operators are attached to the 3D object through a linked list. Methods exist to either pre- or postconcatenate an operator to the current list. You cannot remove an operator from the list except when you kill the whole object in which case the free-order operators get deallocated

automatically. When using the 3D object's methods to manipulate the free-order operators you can only manipulate the latest allocated operator. Note however that since the operators understand the different transformation messages themselves you would normally access them directly without going through the 3D object.
Relevant messages to manipulate the operators (through the 3D object) are `FFTranslate`, `FFRotate`, `FFScale`, `FFRotArbAchsis` and `FFReset`. To allocate and pre- or post-concatenate the operators use `FFNewPreConcat` and `FFNewPostConcat`.

fig II.11 : Functionality of the FF operators

To evaluate all operators use the `CalcTransform` message. If you use the supplied `Draw` and `fDraw` messages to draw your objects you never have to call CalcTransform yourself. CalcTransform handles all inheritance by itself.

Attributes
The TSObject3D only has two additional attributes that you can change. They are the *AutoErase* and *UseBounds* attributes. When AutoErase is set, calling the `Draw` of `fDraw` methods will cause them to erase the part of the current active window that corresponds to the *bounding rectangle* (the bounding rectangle is the smallest rectangle into which the image fits) of the previously drawn image. Usually this will only erase the image drawn just before. If you changed ports with QuickDraw's `SetPort` routine, however, it will have unpredictable results.
If you only set the UseBounds attribute, the `fDraw` and `Draw` methods collect the bounds information but will not erase the last image drawn.
Relevant messages are `SetAutoErase` and `SetUseBounds`.

Using Resources
Reading from and writing to resources using the object is very easy. Usually you would create objects using an interactive 3D object editor and save the objects to a resource file. This eliminates the need for long and tedious object definitions inside a program. The TSObject3D supports resources. The routines for accessing resources have been covered in the 'General Procedures' section, above.

Inheritance

One of the most powerful features of the TSObject3D is the possibility to inherit transformations to other TSObject3Ds. For this GrafSys uses two special instances of the operator. When inheriting transformation you have to specify two objects: The Father (who supplies the transformations) and the Son, who inherits.

What it is

Imagine we want to model a robot arm with a hand. The hand is mounted on a flexible joint. If we move the arm, the hand has to be moved along with it. If we rotate the arm, the hand has to stay at the end of the arm and must be rotated accordingly. As it turns out, exactly the same transformations done to the arm must be repeated for the hand.

fig II.12 : Robot arm and hand (left) and rotated (right)

So instead of repeating the same rotations for the hand over and over again it would be much more efficient if we could simply pass the (already calculated) transformations on to the hand that moves relative to the arm.

fig II.13: Hand rotated relative to rotated arm

If we now rotate the hand we could either apply all transformations first from the arm and then from the hand or we could use the previously calculated arm transformations and then apply the hand transformation.

GrafSys supports this kind of linking one object to another through a specialized FF operator.

How to use it

Usage is pretty simple. With the father, allocate all operators that you want another object to inherit. Then allocate a special 'PassOn' operator. This will be postconcatenated to the father's operator list.

fig II.14a: Object 2 inherits free transformations from Object 1

Locate the object that should inherit the transformations. Send it the FFInherit message and a special operator will be created that links the son to the father. Note that any operators that have been allocated previously by the son will remain previous to the inheritance.

If, for example, you allocated two operators A and B before you passed the FFInherit message and then allocated another operator D, each time you calculate the transformation, A and B will be executed, then the inherited transformations and then D.

While it is it very common that a father has multiple sons, using this technique it is also possible to implement multiple inheritance (i.e. a son with more than one father). If you should ever find a need for this, however, you are probably doing something wrong because an object seldom moves relative to two objects at once. If it does, you can be quite sure that one of them actually moves relative to the other and the correct implementation would be a hierarchy of inheritances.

fig II.14b: Multiple inheritance (left) and corrected inheritance (right)

However, there is no guarantee that this will always work and there might indeed be some strange reason why you should want to do this. Therefore GrafSys supports multiple inheritance.

Relevant messages are `FFInherit` and `FFPassOn`.

Note that the default transformations are *always* inherited from the father object. Likewise, the default transformations are always applied before any other transformations (including inherited) are applied.

Cloning

All objects understand the `Clone` message. This will produce an exact copy of the object. However, there are some subtle details involved when cloning an object that inherits transformations.

fig II.15a: B inherits from A

Since all operators that are allocated are cloned as well, so will be the inherit operator. In this special case, the *father's FF operator chain will be modified* and *another PassOn operator inserted* so that both clones now correctly inherit the same information from the father:

fig II.15b: C is cloned from B. A new PassOn operator was installed in father object

If, on the other hand you clone a father object, the PassOn operator is **not** cloned to avoid double inheritance, since it would make no sense at all. If, for example, a son inherited a translation of 100 along the X-axis, cloning the father would result in inheriting this *twice*, translating the son for 200. Therefore, be careful when cloning an object that passes on, since the clone in this case is just a twin.

Killing

Again, the possibility of inheritance can give cause for headache. Imagine you had an object B that inherited from object A. If you now send B a `Kill` message, there is no problem. But what if we send the father a Kill message? We would end up with a dangling inheritance. The next time we try to calculate the transformation we would end up with a reference to a dead object. Since inheriting means logically that one object (the 'slave') moves relative to another (the 'master'), it is sensible to assume that if the master is removed, the slave should be removed as well since it has nothing to move relative to.
Therefore killing a father will result in killing all sons as well!

fig II.16: B inherits from A. Effect of killing B (middle) versus killing A (right)

This means that if a son is father to another object, the 'grandson' will be killed as well if its father is killed etc.

The GrafSys Class Library

This chapter is the reference section for all messages and methods currently implemented in the class library.

TGenericObject

Introduction

TGenericObject is an abstract class to provide a common denominator for all GrafSys objects and provide standardized house-keeping and error-handling messages. It is the root class.

Heritage

Superclass	none
Subclasses	TMatrixList
	Tabstract3DObject

Using TGenericObject

TGenericObject provides the GrafSys with a standard for error-handling and error-notification as well as house-keeping chores.

Variables

Variable	**Type**	**Description**
`ErrorCode`	`Integer`	Result of last operation

The following error codes are defined by GrafSys:

Label	**Value**	**Description**
noErr	0	No error encountered
cNoFFallocated	-1	Operation tried on a FF matrix when none was allocated
cOutOfMem	-2	Cannot allocate memory for this operation
cBadMethodCall	-3	You called a method that should be instanced but not called
cNothingToInherit	-4	[obsolete]
cTooManyPoints	-5	Model's database is full. Maximum number of points exceeded
cIllegalPointIndex	-6	You tried to access a point with an illegal index (either

		negative or larger than number of points defined)
cTooManyLines	-7	Too many lines defined.
cIllegalLineIndex	-8	You tried to access a line with an illegal index (either the index was negative or larger than the number of lines defined)
cCantDeletePoint	-9	The point cannot be deleted because at least one line references it
cNotOwner	-10	The matrix you specified does not belong to this object
cBadFF	-11	The matrix you specified has not been allocated
cBadFFType	-12	The matrix you specified cannot be activated
cCantLoadRes	-13	The resource you specified (either by name or ID) cannot be loaded. This usually happens when it cannot be found.

Methods

```
procedure Init;
```

Call Init only once for every object and directly after allocating it. This message will cause the object to reset itself to a predefined state and allocate all buffers it needs to function properly.
This method just resets ErrorCode to noErr.

```
procedure Reset;
```

Reset will reset the object to its default predefined state. It is like Init except that no buffers are allocated.

```
procedure Kill;
```

Kill will release the memory associated with the object. Use this method to only release the memory the object itself occupies. Other instances of this message will also deallocate buffers associated with the object.

```
function Clone: TGenericObject;
```

Clone returns an exact copy of your object. Use this method to just clone the object but not the associated buffers. This way the two objects (clone and original) share the same buffers. All instances of this method will also clone the different allocated buffers.

```
procedure HandleError;
```

HandleError is the default error handler for the GrafSys. If called, it looks up ErrorCode and tries to translate the error to text. It displays the error code inside a Stop-Alert:

fig II.17: Standard error alert

```
procedure ResetError;
```

This method resets ErrorCode to its default value, noErr.

```
function Test (opcode: integer): integer;
```

This is the basic sanity-check routine. You can instance it to include your own checking routines. Opcode can be used to pass anything to the checking routine. GrafSys ignores the opcode parameter and only checks the ErrorCode. In any case Test will display an Information-Alert similar to the one below:

fig II.18: Standard test notification alert

Test returns the ErrorCode when done.

Resources

GrafSys uses two resources for it's error-handling and testing:

Resource Type	**ID**
DITL	32700
ALRT	32700

TMatrixList

Introduction

TMatrixList is the central transformation operator type. Higher GrafSys objects use this type to implement free-order transformations and inheritance of transformation.

Heritage

Superclass	TGenericObject
Subclasses	TMatrixInherit
	TMatrixPass

Using TMatrixList

All TMatrixList objects understand messages to rotate, scale and translate. You would normally use them in conjunction with the TSObject3D free-order transformation feature. You should never allocate a TMatrix object yourself but let other (higher) GrafSys methods do this. Once allocated, though you can (and normally would) directly access them to tell them to rotate etc.

If you modeled above mentioned robot arm, you would let the TSObject3D allocate the FF operator (which is of TSMatrix3D type). Later in your program however you would tell the operator directly to transform without going through the TSObject3D.

Variables

Name	**Type**	**Description**
M	Matrix4	This is the actual transformation matrix
next	TMatrixList	Next operator in list
owner	TGenericObject	Object that own this operator

Methods

Inherited methods:

```
function Clone: TGenericObject;
procedure Kill;
procedure HandleError;
procedure ResetError;
function Test (opcode: integer): integer;
```

Other Methods:

```
procedure Init;
override;
```

Init initializes the object first by calling the inherited Init method and then by setting the M to Identity. Next and owner are set to `nil`.

```
procedure Reset;
override;
```

Reset is like Init except that owner and Next aren't set to `nil`. M is set to Identity.

```
procedure TMRotate (dx, dy, dz: real);
```

TMRotate calculates the matrix required to accomplish a rotation of dx radians further around the X-axis, dy radians further around the Y-axis and dz radians around the Z-axis. The result is multiplied with M and stored in M.

```
procedure TMScale (dx, dy, dz: real);
```

TMScale calculates the matrix required to accomplish a scaling of a factor of dx along the X-axis, dy along the Y-axis and dz along the Z-axis. The result is multiplied with M and stored in M.

```
procedure TMTranslate (dx, dy, dz: real);
```

TMScale calculates the matrix required to accomplish a translation of dx along the X-axis, dy along the Y-axis and dz along the Z-axis. The result is multiplied with M and stored in M.

```
procedure TMRotArbAchsis (p, x: Vector4;
         phi: real);
```

TMRotate calculates the matrix required to accomplish a rotation of phi radians around the axis defined by the two points p and x.

fig II.19: Arbitrary rotation

The axis is defined as looking from the point P to X. A positive phi will rotate clockwise, a negative counter-clockwise.

The result is multiplied with M and stored in M.

TMatrixInherit

Introduction

This instance of TMatrixList is for internal use only and should not be used by you. Usage is for inheritance of transformations.

Heritage

Superclass	TMatrixList
Subclasses	none

Using TMatrixInherit

Don't.

Variables

Name	**Type**	**Description**
upLink	TMatrixList	Pointer to link in FF chain of father
meTheSon	Tabstract3DObject	Pointer to owner

Methods

Inherited methods:

```
 function Clone: TGenericObject;
 procedure Kill;
 procedure HandleError;
 procedure ResetError;
 function Test (opcode: integer): integer;
 procedure Reset;
 procedure TMRotate (dx, dy, dz: real);
 procedure TMScale (dx, dy, dz: real);
 procedure TMTranslate (dx, dy, dz: real);
 procedure TMRotArbAchsis (p, x: Vector4;
     phi: real);
```

Other Methods:

```
procedure Init;
override;
```

Calls inherited Init and then sets upLink and meTheSon to nil.

TMatrixPass

Introduction

This instance of TMatrixList is for internal use only and should not be used by you. Usage is for inheritance of transformations.

Heritage

Superclass	TMatrixList
Subclasses	none

Using TMatrixPass

Don't.

Variables

Name	**Type**	**Description**
downLink	TMatrixList	Pointer to link in FF chain of son
meTheFather	Tabstract3DObject	Pointer to owner

Methods

Inherited methods:

```
 function Clone: TGenericObject;
 procedure Kill;
 procedure HandleError;
 procedure ResetError;
 function Test (opcode: integer): integer;
 procedure Reset;
 procedure TMRotate (dx, dy, dz: real);
 procedure TMScale (dx, dy, dz: real);
 procedure TMTranslate (dx, dy, dz: real);
 procedure TMRotArbAchsis (p, x: Vector4;
     phi: real);
```

Other Methods:

```
procedure Init;
override;
```

Calls inherited Init and then sets downLink and meTheFather to nil.

Tabstract3DObject

Introduction

Tabstract3DObject is the basic 3D object. It implements the methods for handling transformations and FF operators as well as inheritance. Use this object whenever you want to build objects that do not use Cartesian coordinates since this object knows nothing about points, lines or anything else about the object database.

Heritage

Superclass	TGenericObject
Subclasses	TLine3D
	TPoint3D
	TSGenericObject3D

Using Tabstract3DObject

Directly after allocating the Tabstract3DObject call the `Init` method to initialize the default and free-order operators. All default operators are set to Identity, the translation and rotation values are set to zero, the scale factors are set to one. No FF operator is allocated, so the `FFMatrix` and `currentFF` fields are set to nil.

Use the `calcTransform` message to evaluate the different operators in the described order (default rotation, default translation, default scaling, default arbitrary rotation, FF list, eye) to generate the xForm operator that you can use to transform points.

Use the `Kill` method to deallocate the object and `Reset` to reset the object to the predefined state of zero rotation, zero translation and scale of one (this will also reset all allocated FF operators, see below).

Variables

Name	**Type**	**Description**
xTrans	real	X coordinate in world coordinates of the object's origin. Used to build the default standard operator.
yTrans	real	Y coordinate in world coordinates of the object's origin. Used to build the default standard operator.
zTrans	real	Y coordinate in world coordinates of the object's origin. Used to build the default standard operator.

xScale	real	Scaling factor for the object's X axis. Used to build the default standard operator.
yScale	real	Scaling factor for the object's Y axis. Used to build the default standard operator.
zScale	real	Scaling factor for the object's Z axis. Used to build the default standard operator.
xrot	real	Rotation (in radians) of object around its X-axis. Used to build the default standard operator.
yrot	real	Rotation (in radians) of object around its Y-axis. Used to build the default standard operator.
zrot	real	Rotation (in radians) of object around its Z-axis. Used to build the default standard operator.
xForm	Matrix4	Final transformation matrix. This is the result of all transformations.
arbRot	Matrix4	All arbitrary operations on default operator are stored here.
currentFF	TMatrixList	Points to the currently active FF operator.
FFMatrix	TMatrixList	Points to first element in the FF operator chain.
objChanged	Boolean	Indicates if the object's data base has changed. A call to calcTransform will reset it.
versionsID	longint	Used for synchronization with eye to detect if a recalculation of xForm is required since xForm also holds eye transformation
hasChanged	Boolean	Indicates that a `calcTransform` call changed the transformed point description. This is used to flag to `Draw` methods that they have to redraw.

Methods

Inherited methods:

```
 procedure HandleError;
 procedure ResetError;
 function Test (opcode: integer): integer;
```

Other Methods:

```
procedure Init;
override;
```

Init calls the inherited Init method and then initializes the default and free-order operators. All default operators are set to Identity, the translation and rotation values are set to zero, the scale factors are set to one. No FF operator is allocated, so the `FFMatrix` and `currentFF` fields are set to nil.

```
procedure Kill;
override;
```

Use Kill to deallocate the memory associated with this object. This method will also deallocate all owned FF operators *and all objects that inherit transformations from this object*. See the 'Inheritance' chapter, above.

```
procedure Reset;
override;
```

Reset calls the inherited Reset method and then resets the default and free-order operators. All default operators are set to Identity, the translation and rotation values are set to zero, the scale factors are set to one. All FF operator are reset to identity.

```
function Clone: TGenericObject;
override;
```

Clone returns an (almost) exact clone of the object. All FF operators owned by the object are cloned as well. If the original object inherits transformation from another object (called the father), Clone will place a PassOn FF operator into the FF chain of the father directly behind the PassOn operator for the original.

fig II.20: The result of cloning B yielding C

In above example B inherited from A. When B was cloned (yielding C) a PassOn operator was inserted in A's FF operator chain. If on the other hand an object passes on, this link to the son will **not** be cloned; the link is lost to avoid double inheritance.

```
procedure Translate (dx, dy, dz: real);
```

Displaces the object's origin for the vector [dx,dy,dz]. Translate operates on the default operator as reflected by the objects variables xTrans, yTrans and zTrans.

```
procedure SetTranslation (x, y, z: real);
```

Moves the object's origin to the world coordinates [x,y,z]. SetTranslation operates on the default operator as reflected by the objects variables xTrans, yTrans and zTrans.

```
procedure Rotate (dx, dy, dz: real);
```

Rotates the object further around its main axes. If rotation is positive, it will be clockwise. dx specifies the amount (in radians) around the X-axis, dy around the Y-axis, dz around the Z-axis. Rotate operates on the default operator as reflected by the object's variables xrot, yrot and zrot.

```
procedure SetRotation (x, y, z: real);
```

Sets the object's rotation around the main axes to the amount specified. If rotation is positive, it will be clockwise. dx specifies the amount (in radians) around the X-axis, dy around the Y-axis, dz around the Z-axis. SetRotation operates on the default operator as reflected by the object's variables xrot, yrot and zrot.

```
procedure Scale (dx, dy, dz: real);
```

Scale increments the scaling factors for the object by the given values. Scaling is independent from any previous translation or rotation (i.e. it will scale the object along its original local x, y and z-axis). A (resulting) setting of 1 means no scaling, a setting of 2 means double size, a setting of 3 triple size etc. A factor of zero will shrink that axis into nonexistence. Negative scaling will produce mirror-effects (I guess).
Scale operates on the default operator as reflected by the object's variables xS

CGrafPort) data structure. That is why the `TPort3DPtr` is compatible with the `WindowPtr` (or `GrafPtr` for that matter). The first part is the same. GrafSys uses a `current3DPort` variable similar to QuickDraw's currentPort variable 'thePort'. If you call Set3DPort, GrafSys sets its own current3DPort variable to the window you specify and then tells QuickDraw to set its own currentPort variable to the same window.

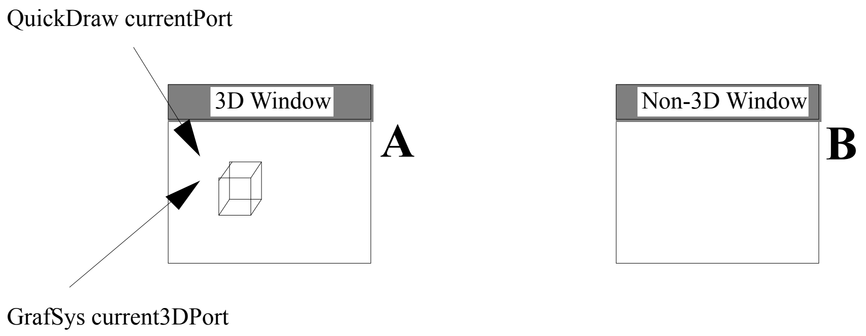


fig II.26: Normally QuickDraw points to window A. Drawing takes place in A. GrafSys points to A. A's eye settings will be used.

Since GrafSys uses QuickDraw's `MoveTo` and `LineTo` procedures to draw lines, all drawing will take place in the GrafPort QuickDraw's currentPort variable points to. And here is the clinch.
You can now call QuickDraw's SetPort to set its currentPort to another window; GrafSys would not be notified about this. It will still assume that you are drawing into the window pointed to by current3Dport and therefore use its eye settings.

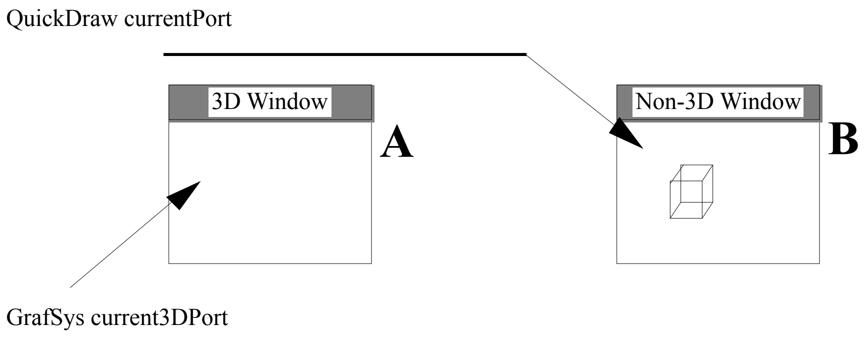


fig II.27: QuickDraw points to window B. Drawing takes place in B. GrafSys points to A. A's eye setting will be used but the image will appear in B.

Imagine we had two windows, A and B. A is a 3D window, B is not. Now, to set GrafSys up we will use `Set3DPort(A)`. GrafSys and QuickDraw will now be drawing into window A. Now, use QuickDraw's `SetPort(B)`. All actual drawing will from now on take place in window B. But since the GrafSys current3DPort pointer still points to A, GrafSys will use A's 3D window structure. That way you can draw 3D graphics into non-3D ports. Use the same technique to allow two eye settings for one window: allocate a second 3D window, hide it but use its eye settings to draw the object on the first window by using SetPort to set the QuickDraw port to it.

Another application for this technique is when you want to print the object. Once again, you first use Set3DPort to initialize the projection routines and then continue with a call to `PrOpenDoc` [InsideMac]. Another, simpler approach would be to surround the Draw message by

`OpenPicture` and `ClosePicture` calls and then draw the picture on the printing device [InsideMac, TN021, TN059, TN297].

Example 1: Robot Arm (transformation inheritance)
Let us begin with a fairly simple demonstration of the GrafSys capabilities. The first demo, RobotArm is a mere 2D animation of the previously mentioned robot arm to demonstrate transformation inheritance and how you would model such an object.

Robot Arm Points and Lines
The whole arm will look like this:

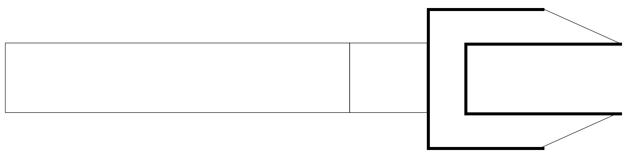
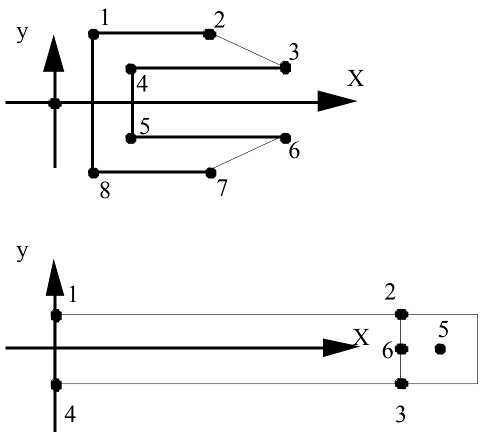


fig II.28: Robot arm

The problem is that the whole robot arm has parts that can move independently from one another (i.e. the hand can rotate without moving the arm). We therefore have to subdivide the robot arm into two parts: Arm and Hand.
The first step in modeling the arm is defining the point and line database for both objects.



```
Hand:
1:15,15,05:25,- 5,0
2:45,15,06:35,- 5,0
3:35, 5,07:45,-15,0
4:25, 5,08:15,-15,0
```

```
Arm:
1: 0, 15,
2:70, 15,
3:70,-15,
4: 0,-15,
```

fig II.29: Database definition of hand (top) and arm (bottom).
Line numbers are not shown

As can be seen, the origin of the hand is outside the object and is placed into the center of the joint that is attached to the arm. For building the object's databases we define two objects, BuildHand and BuildArm.

```
procedure BuildArm (Obj: TSObject3D);

 var
  OK: longint;

begin
 OK := Obj.AddPoint(0, 15, 0);
 OK := Obj.AddPoint(70, 15, 0);
 OK := Obj.AddPoint(70, -15, 0);
 OK := Obj.AddPoint(0, -15, 0);
 OK := Obj.AddLine(1, 2);
 OK := Obj.AddLine(2, 3);
 OK := Obj.AddLine(3, 4);
 OK := Obj.AddLine(4, 1);

 OK := Obj.AddPoint(85, 0, 0); (* joint is circle with this center  *)
 OK := Obj.AddPoint(70, 0, 0); (* joint radius calculated with this *)

end;

procedure BuildHand (Obj: TSObject3D);
 var
  OK: longint;

begin
 OK := Obj.AddPoint(15, 15, 0);
 OK := Obj.AddPoint(45, 15, 0);
 OK := Obj.AddPoint(35, 5, 0);
 OK := Obj.AddPoint(25, 5, 0);
 OK := Obj.AddPoint(25, -5, 0); (* used to model joint *)
 OK := Obj.AddPoint(35, -5, 0); (* used to model joint *)
 OK := Obj.AddPoint(45, -15
```

the four major transformations (translation, rotation, arbitrary rotation and scaling).

Homogenous Coordinates

All points in the GrafSys have three coordinates: x, y, z. The most simple transformation, the translation, can be expressed as a simple vector-addition. But when we come to rotation or scaling or (even worse) to arbitrary rotation –where translation and rotation are mixed– the transformation becomes a matrix multiplication [Foley90]. To make things more consistent, we need a way to express all transformations the same way. Using linear algebra, we project this 3D problem into the 4-dimensional space. Each vector [x,y,z] now becomes [x,y,z,c]. Note that you can choose any value for c since this transformation is equivocal. The mathematical implication of c is a scaling factor [Pavli82] and we would normally set it to 1 [Hearn86] so the transformation becomes unequivocal.

Note: The GrafSys always uses the [x,y,z,1] representation. This is transparent to the user and there is no sure way (and no actual use) for you to manipulate the fourth coordinate.

Since we now have transformed a three-dimensional problem into four dimensions all operations become matrix-multiplications and

we have a universal way to transform a point (i.e. through matrix-matrix multiplication). The new representation using four coordinates is also called 'using *homogenous coordinates*'.
Since we are now using four-dimensional coordinates, everything is simple to implement. The operator is always a 4x4 matrix and transforming a point (or vector) has always the form

The operator always has the following general appearance:

fig III.1: General structure of transformation matrix

As it immediately becomes apparent we can store both translation and rotation in the same operator as long as they do not get mixed (as in arbitrary rotation). However I do not recommend you do this unless you know exactly what you are doing.

Multiple Transformations

The greatest advantage of using homogenous coordinates becomes obvious when we want to perform multiple transformations onto the same object. Now all we have to do is perform a *matrix multiplication* for each transformation. The resulting matrix is the operator that will perform all transformations at once according to their sequence of execution.

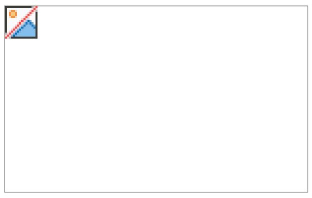
Translation

This is a very simple transformation. All you do is displacing the coordinates for a certain vector [dx,dy,dz]:

To reverse this translation, simply pass [-dx,-dy,-dz] as parameters.

Rotation

Rotation about the main axes (X, Y, Z) is fairly simple. There are three principal rotation matrices that are used: Let θ be the angle you want to rotate.



Rotation about X   Rotation about Y   Rotation about Z

To reverse the operation you can do two things: Either use the negative angle or transpose the matrix (which becomes obvious if you look at the rotation matrix more closely). The GrafSys uses the latter approach since it is faster than recalculating the sine and cosines.

Arbitrary Rotation

Arbitrary Rotation can get rather unpleasant if you do not adhere to the following guidelines:

- Never use two arbitrary rotations with the same operator except the reverse operation (in this case it is much faster to use the Reset method in GrafSys).
- Never use any other transformation with the operator that does an arbitrary rotation.
- Using arbitrary rotation will displace the object's origin so GrafSys' internal procedures will incorrectly reflect the position. If you use arbitrary rotation you must use the `TransformedPoint` message with [0,0,0] to get the correct origin position. This is because arbitrary rotation mixes translation and rotation and the results are not easily predictable.

GrafSys implements arbitrary rotation the following way which has the distinct advantage that it uses only previously defined operations [Foley90]:

- First you specify an axis by defining two points P1 and P2 on the axis. This defines a