

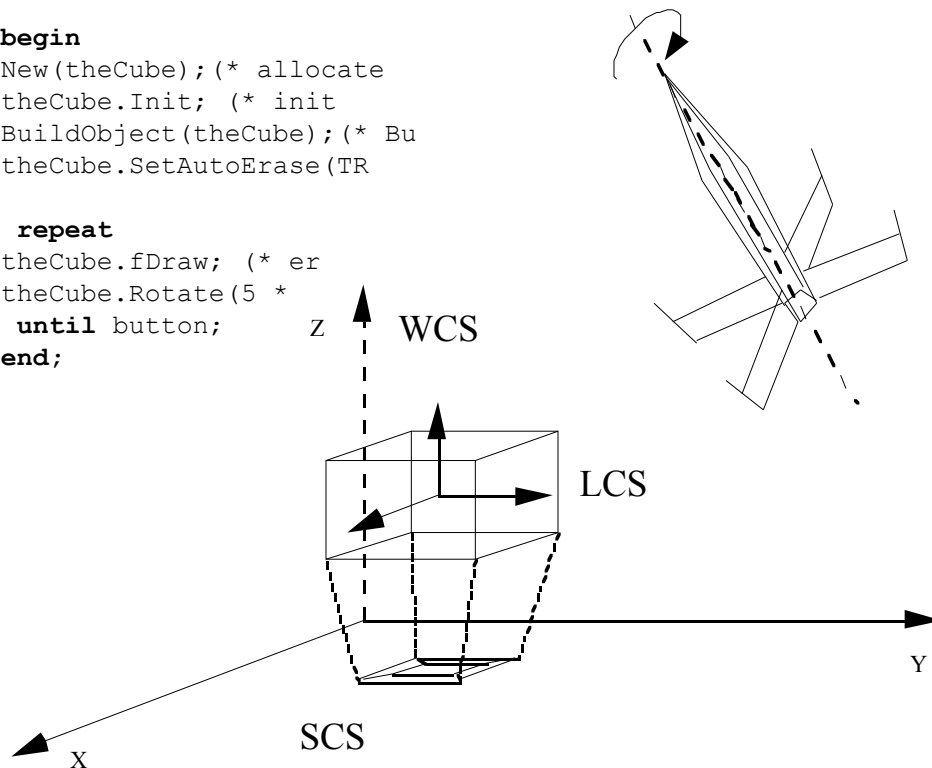
Christian Franz

3D GrafSys

Version 2.0

for programmers

```
begin  
New(theCube); (* allocate  
theCube.Init; (* init  
BuildObject(theCube); (* Bu  
theCube.SetAutoErase(TR  
  
  repeat  
theCube.fDraw; (* er  
theCube.Rotate(5 *  
  until button;  
end;
```



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 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 Brünger*, 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 'TObject3D' 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.

GrafSys Documentation

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 apparent, 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.

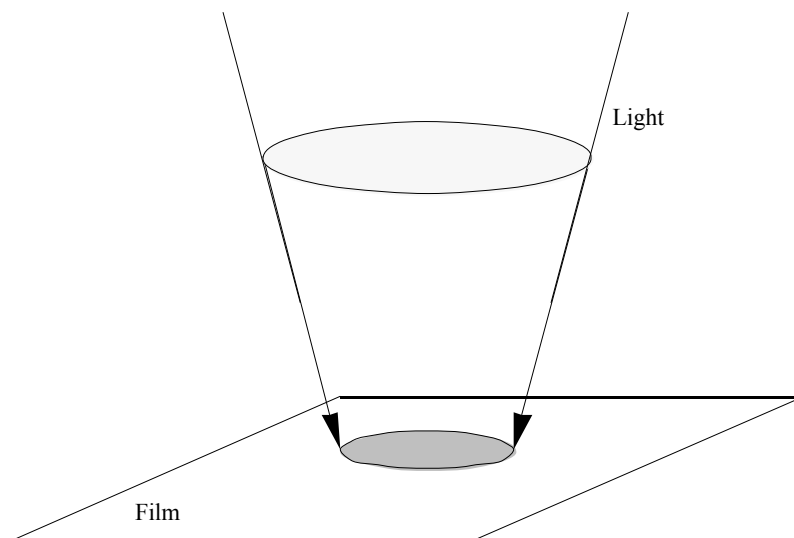


fig I.1 : Projection of an object

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.

GrafSys Documentation

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').

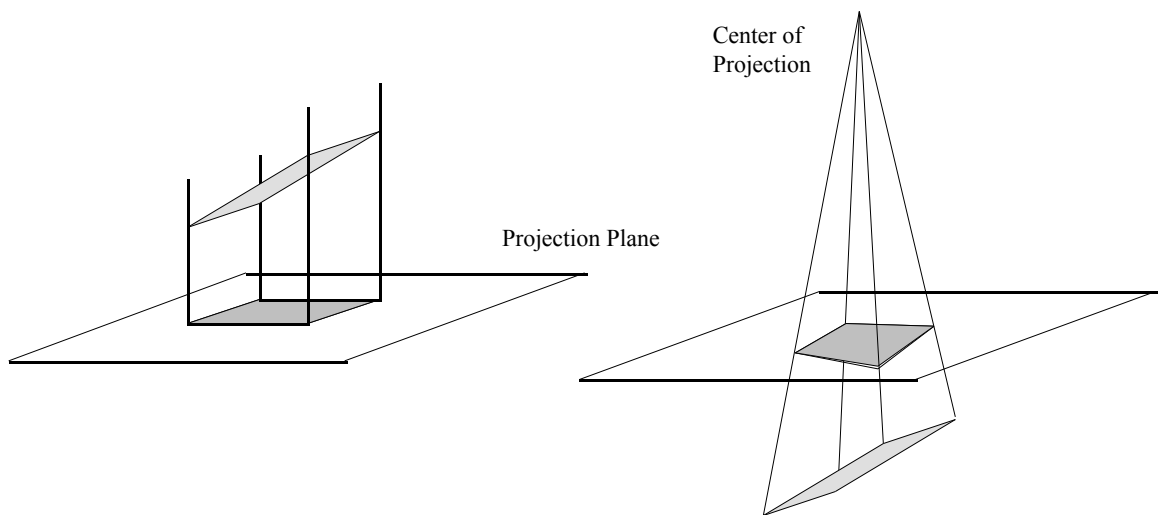


fig I.2a: parallel projection (left) and perspective projection (right)

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

$$\begin{aligned}h &:= xc + x & (xc \text{ is center horizontally of screen}) \\v &:= yc + y & (yc \text{ is center vertically of screen})\end{aligned}$$

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 and y values. The further away from the projection plane, the closer x and y

GrafSys Documentation

should be to x_c and y_c , 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,

$$xp = \xi\left(\frac{\delta}{\delta + \zeta}\right) = \xi\left(\frac{1}{\zeta/\delta + 1}\right)$$

$$yp = \psi\left(\frac{\delta}{\delta + \zeta}\right) = \psi\left(\frac{1}{\zeta/\delta + 1}\right)$$

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.

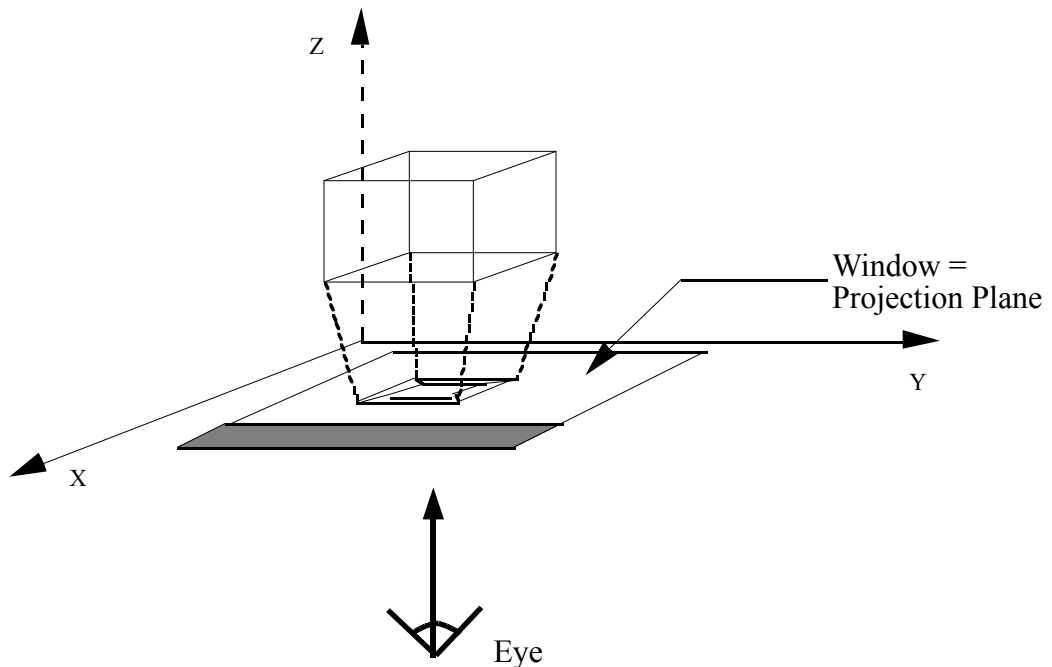


fig I.2b: The eye is always assumed behind the projection plane which can be regarded as the screen. No objects behind the projection plane are drawn

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'.

GrafSys Documentation

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 your own conversion algorithms. The following discussion assumes that you will be using the supplied TSOBJ3D object (described below) or its descendants with its 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 provided 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 technospeak the MOVE TO Toolbox routine will not be called). Imagine we had to define a cube. The eight corners must be connected with twelve lines.

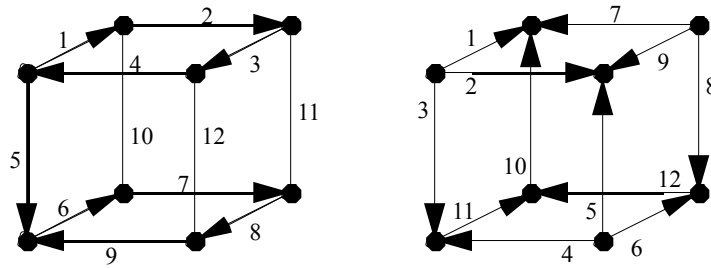


fig I.3: optimal sequence of lines (left) and worst sequence (right)

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).

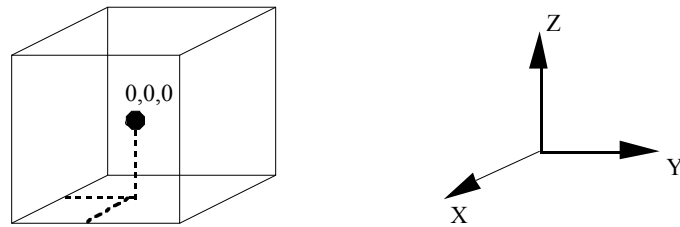


fig I.4: Object origin (left) and coordinate system (right)

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 object's origin. This coordinate system is called World Coordinate System (WCS).

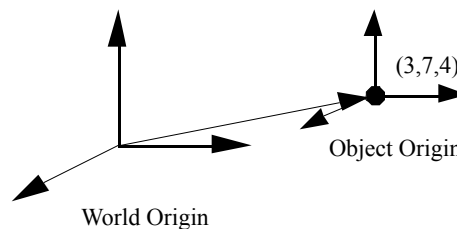


fig I.5: Difference between World Origin and Object Origin

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.

GrafSys Documentation

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.

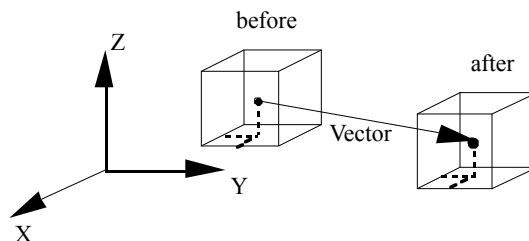


fig I.6: Relative translation

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.

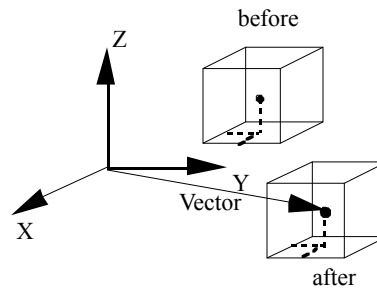


fig I.7: Absolute translation using the same vector as in relative translation

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.

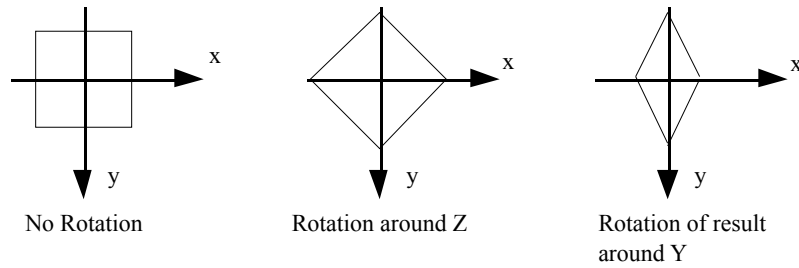


fig I.8a: result of two sequential rotations

Some people might have expected the following result:

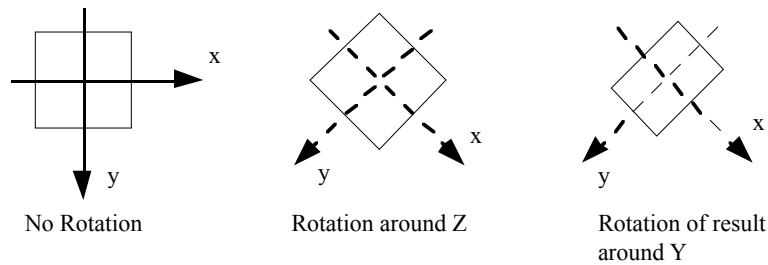


fig I.8b: Incorrect (but expected) result of two sequential rotations

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:

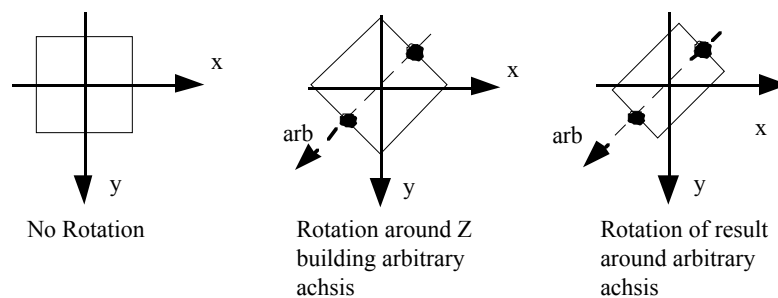


fig I.8c: Achieving expected result through arbitrary rotation

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.

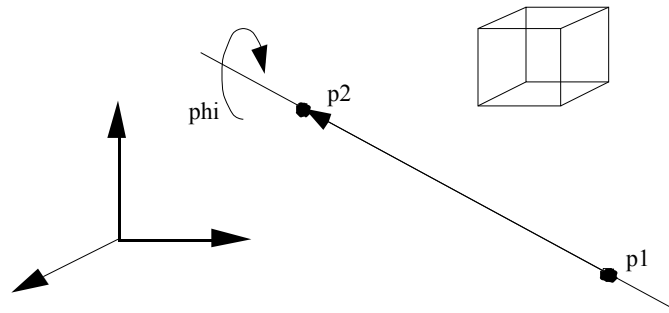


fig I.8c: Rotation around an arbitrary axis

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; (*  $\pi$  / 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.

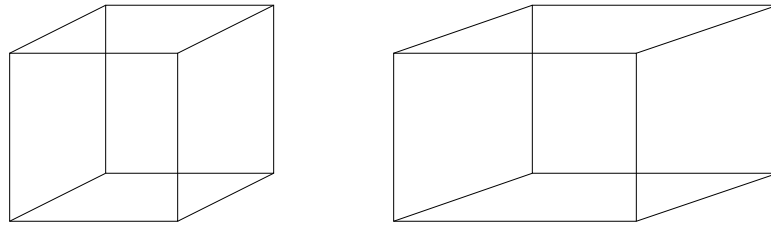


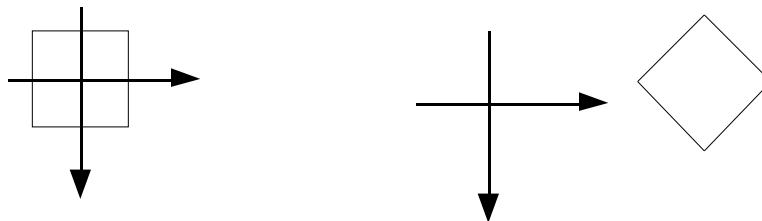
fig I.9: Normal cube (left) and scaled along one axis (right)

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.

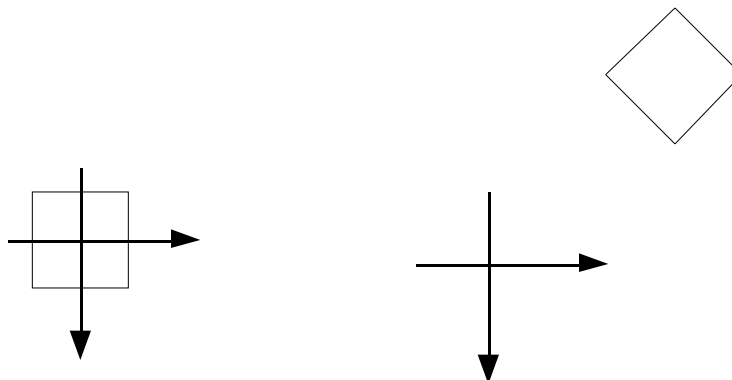
Order Dependencies

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 order 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:



First rotating then translating the rotated square



First translating then rotating the translated square

fig I.10: Order dependencies

GrafSys Documentation

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:

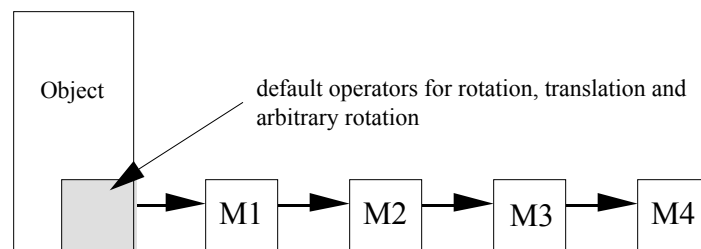


fig I.11: Free-order transformation operators

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'.

GrafSys Documentation

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)

GrafSys Documentation

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 world's 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:

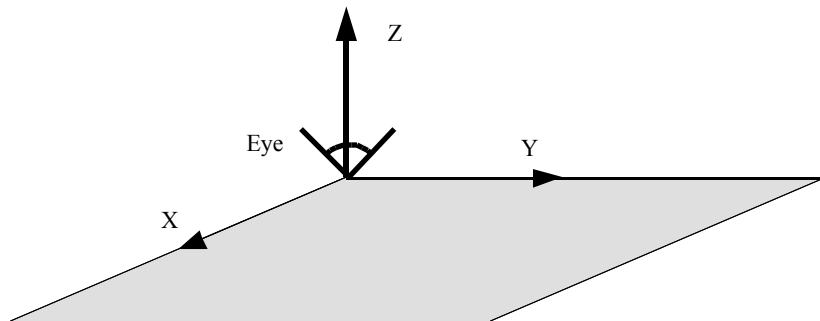


fig I.12 : The eye default direction

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.

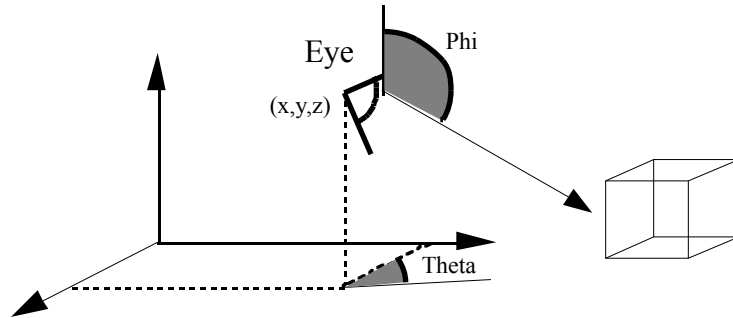


fig I.13 : Eye in action

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:

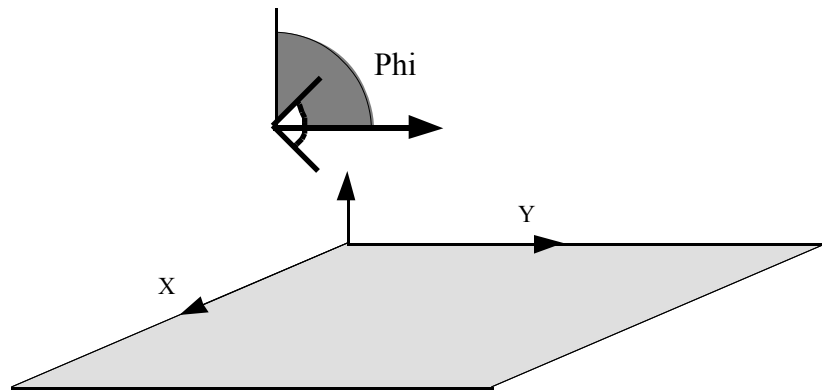


fig I.14: Definition of phi

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

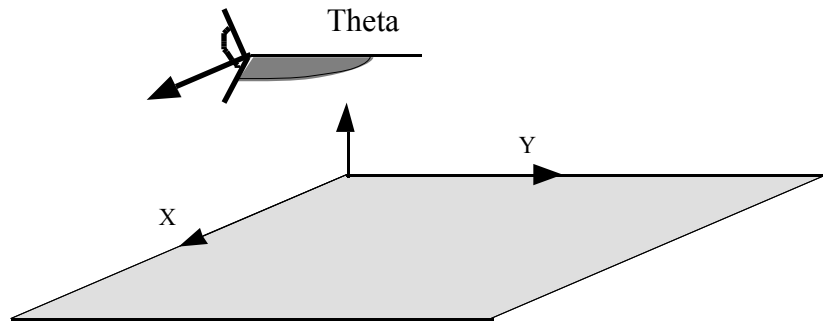


fig I.15: Definition of theta

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')

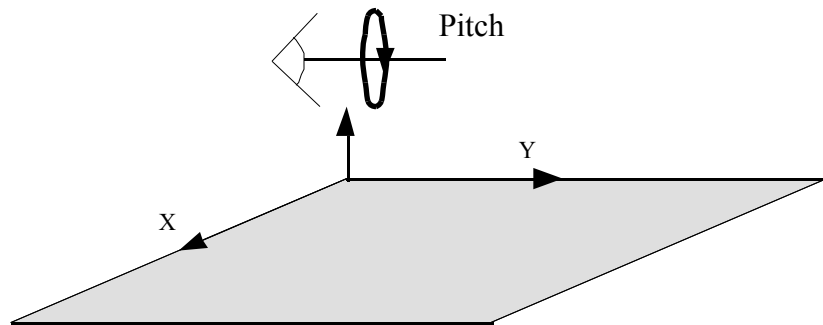


fig I.16: Definition of pitch

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.

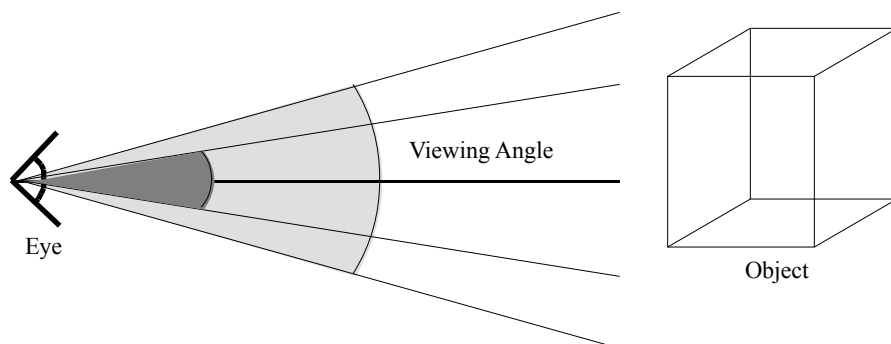


fig I.17a: Definition of viewing angle.

A viewing angle of *zero* tells the package that you want to switch to parallel

GrafSys Documentation

projection (see below). The viewing angle defines how far

GrafSys Documentation

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.

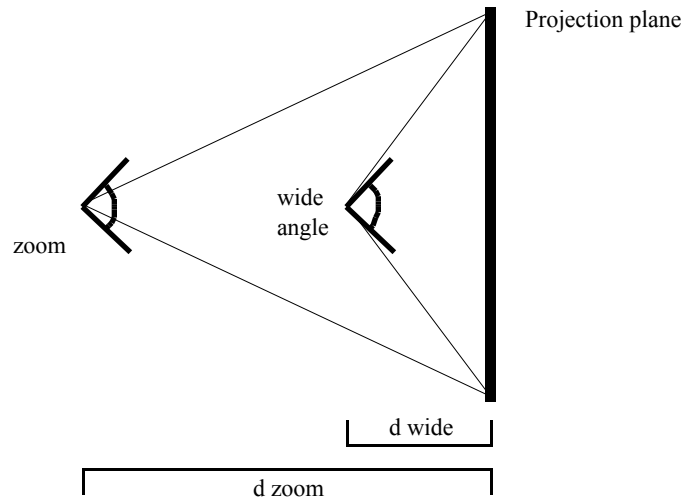


fig I.17b: d parameter and viewing angle

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.

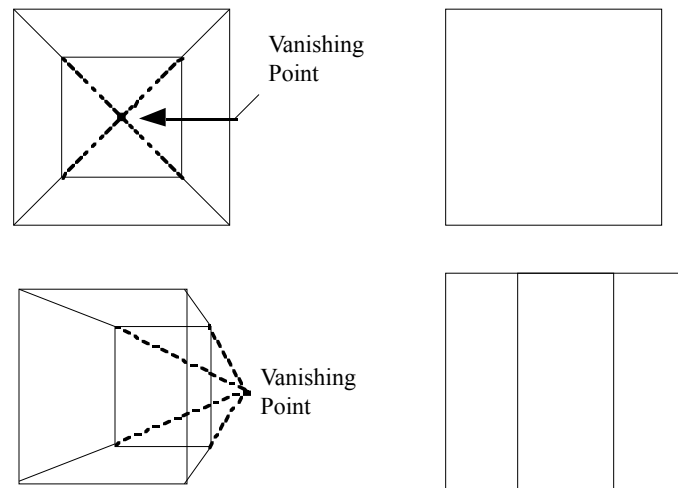


fig I.18a: perspective projection (left) and parallel projection (right)

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.

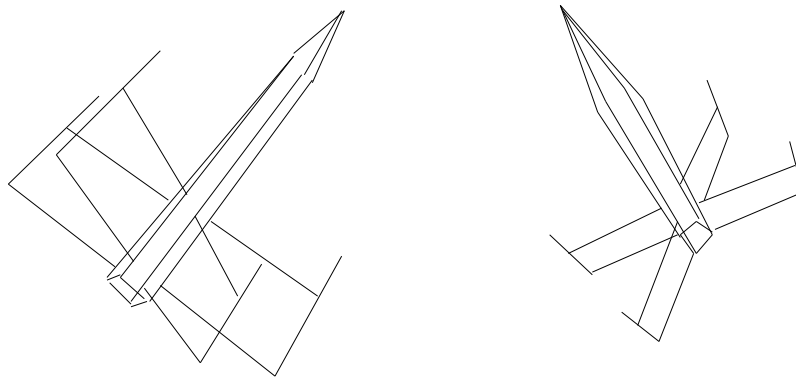


fig I.18b: two perspective projections of the same object

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

GrafSys Documentation
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.

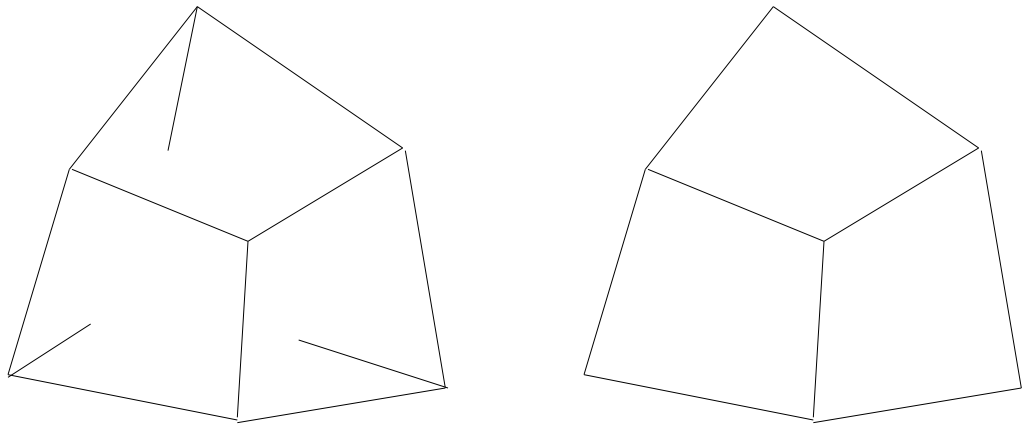


fig I.19: Clipped cube. Left with clipping to projection plane, right with eliminating offending lines

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)

GrafSys Documentation

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).

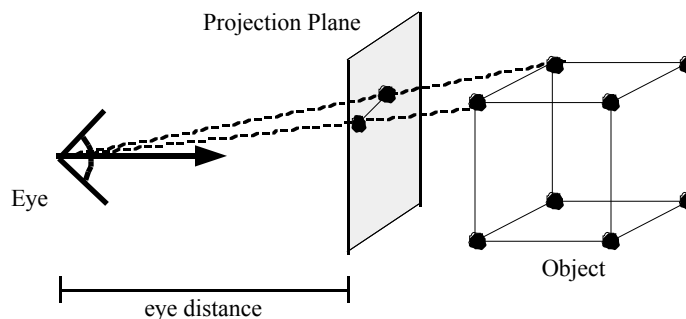


fig I.20: Eye Distance and projection plane

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.

GrafSys Documentation

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).

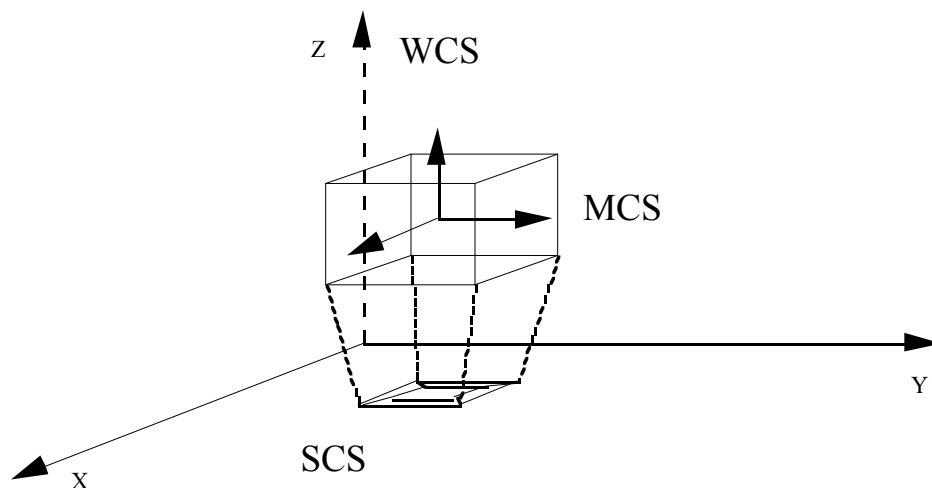


fig I.21: The different coordinate systems and they relation

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

GrafSys Documentation

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.

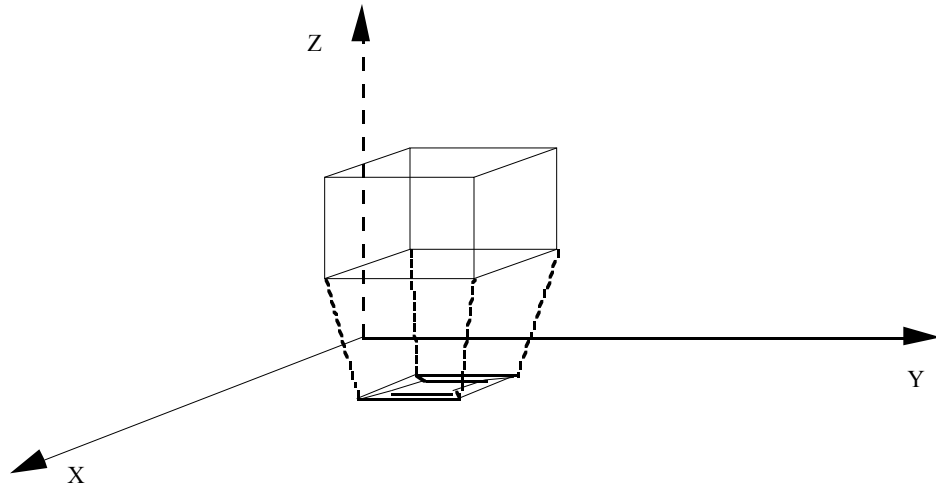


fig I.22: Points are always projected onto the XY-plane

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

GrafSys Documentation

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 *TObject3D* 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.

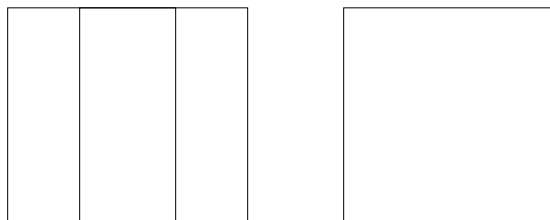


fig II.1 Cube in parallel projection, left rotated, right with rotation of zero

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

GrafSys Documentation

pick almost any two views, it is wise to choose special sketches: the top view and one of two side views.

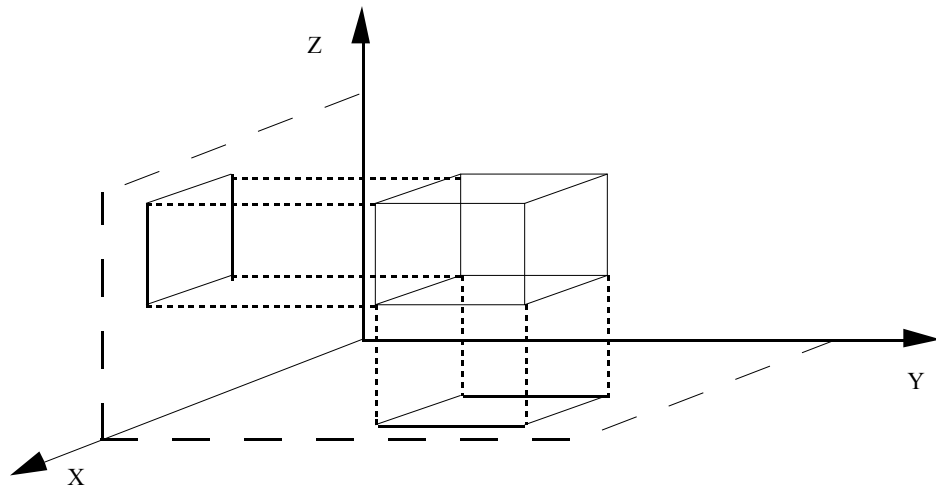
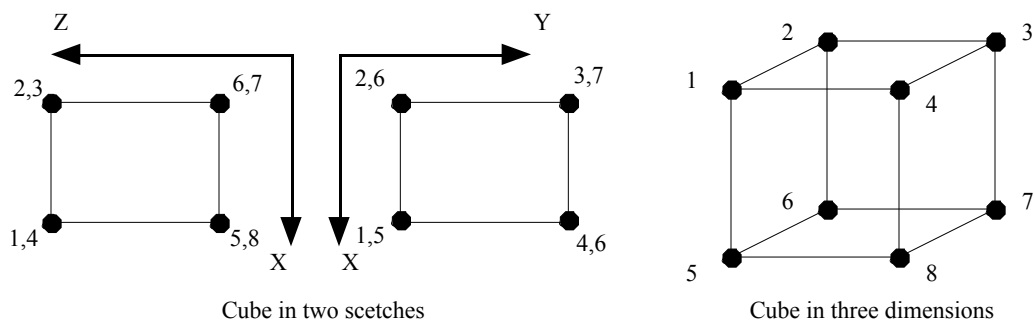


fig II.2: parallel-projecting an object onto two separate planes

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



Cube in two sketches

Cube in three dimensions

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 point's 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 three 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

GrafSys Documentation

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:

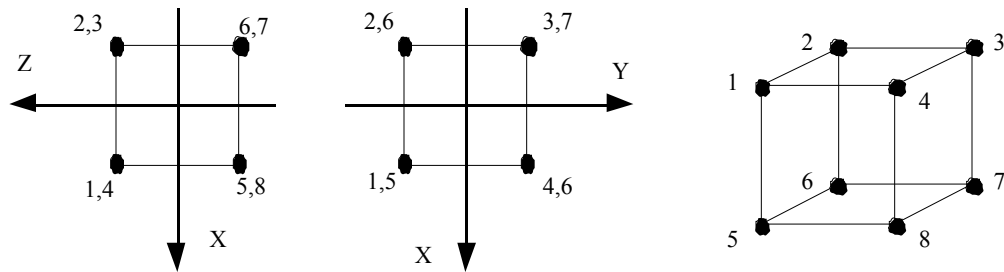


fig II.3 Cube sketch and origin, 3D view of same cube

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.

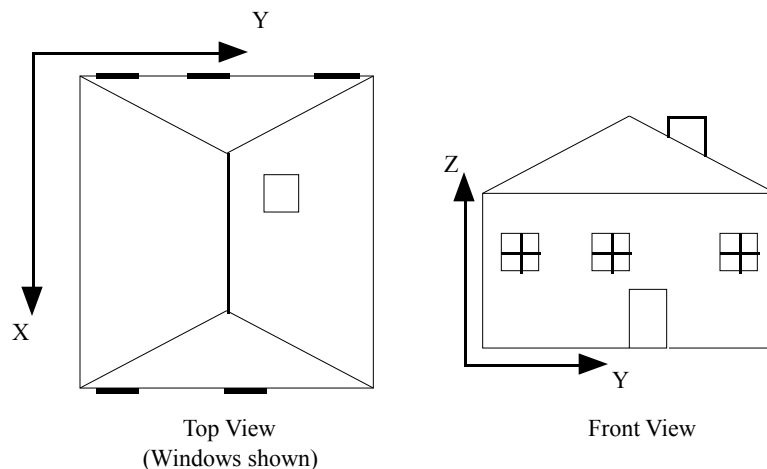


fig II.4: Two sketches to define a house object

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.

GrafSys Documentation

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 `TObject3D`) 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 `TObject3D` 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:

GrafSys Documentation

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 your program will not compile if you switch between versions of the GrafSys.

GrafSys Documentation

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
- 1Clr 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 1Clr 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-Trip 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;

GrafSys Documentation

```
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 2.00 α .

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

GrafSys Documentation

then tries to read in a resource of type '3Dob'

GrafSys Documentation

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.

GrafSys Documentation

```
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.

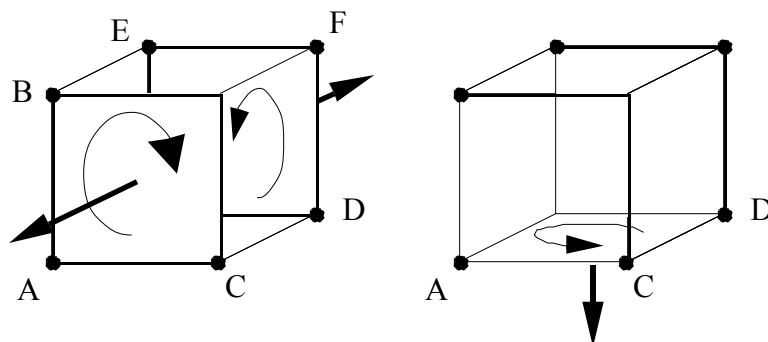
If you use FillTriangle to draw directly to the screen make sure that the currently active port is frontmost (i.e. is not obscured by anything) and that you bracket the FillTriangle calls with ShieldCursor and ShowCursor. TheColor is the direct pixel color you want the triangle to have. If you use the 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:

GrafSys Documentation

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.

GrafSys Documentation

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

GrafSys Documentation

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;
```

GrafSys Documentation

EyeKoord: Vector4;

GrafSys Documentation

```
    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

GrafSys Documentation

		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

GrafSys Documentation

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`

GrafSys Documentation

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*:

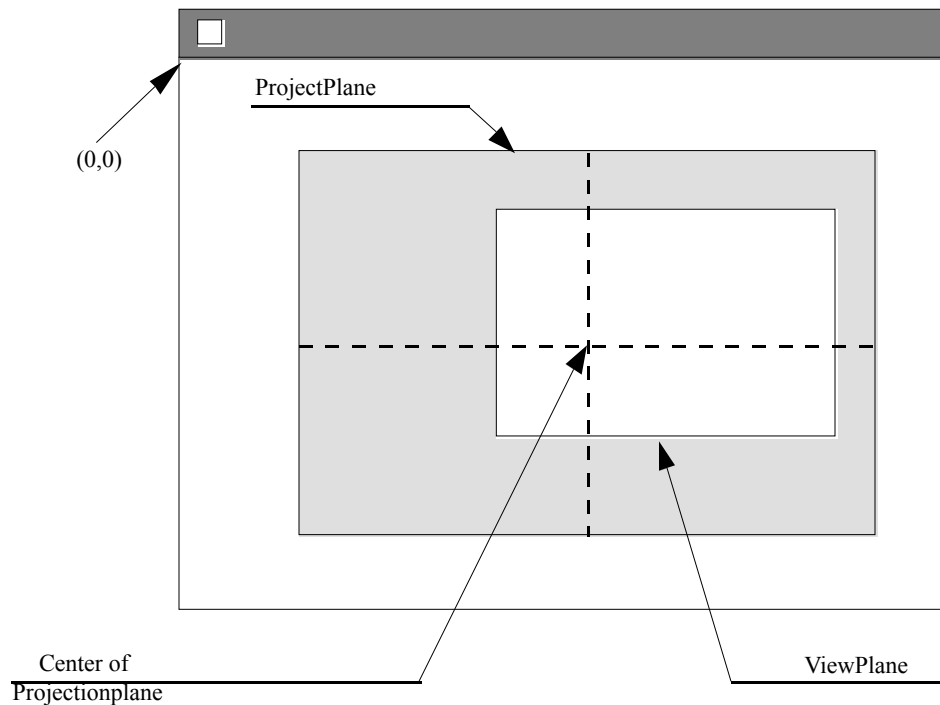


fig II.5: ProjectPlane, ViewPlane, Center of Projection and Window Origin (0,0)

Note: This procedure operates on the current active 3D GrafPort and QuickDraw's currently active GrafPort!

Note: This procedure sets the ClipRgn to ViewPlaneSize of QuickDraw's currently active window!

Note: If you manually change the ClipRgn of a 3D window be sure to restore it to its settings before calling any drawing routines.

```
procedure SetCenter (x, y: Integer);
```

SetCenter repositions the center of the project plane to the given coordinates. X and y should be in the window's local coordinates. This does not move the projection plane itself.

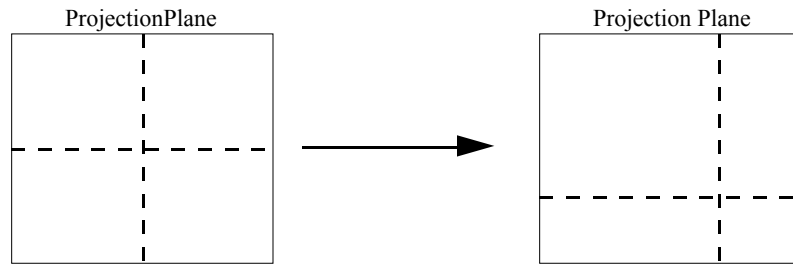


fig II.6: Effect of SetCenter on the Projection Plane

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

Operations Affecting the Eye

All routines that manipulate the eye work on the currently active 3D port. The eye has three major attributes (location, orientation and view angle) plus one minor attribute (clipping type). A master switch tells the GrafSys whether you want to use eye transformations or not. The projection type (parallel or perspective) is set indirectly through the view angle and must be set even if you do not use the eye.

```
procedure SetEye (UsesEye: Boolean; location:
    Vector4; thePhi, theTheta, thePitch,
    theViewangle: real; clipType: clippingType);
```

SetEye sets the attributes for projection. The UseEye parameter tells the package if after transforming an object additional eye transformations should be applied (thus making it possible to move the camera around) or if the eye is fixed at the world's origin and is looking straight up. If your eye is fixed or you are only moving objects, set UseEye to FALSE since it will slightly speed up the calculations.

Location specifies the eye's coordinates (i.e. where the camera is located in the world), phi, theta and pitch define how far from the z-axis towards the y-axis the camera should turn (phi), how far around the z-axis the camera should turn (theta) and how far around the current looking direction the camera should turn (pitch).

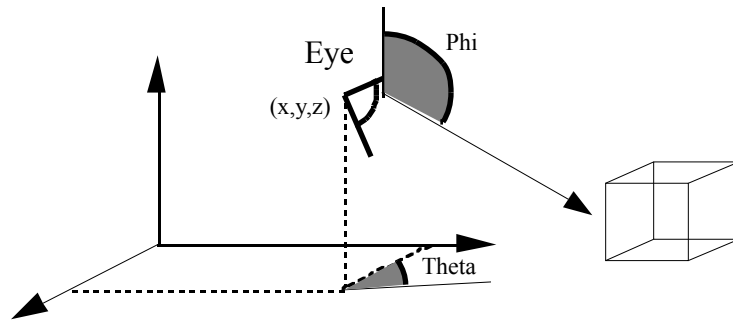


fig II.7 : Eye attributes location and orientation

The view angle parameter tells the package what kind of electronic lens you are using. A small value (≈ 0.1) would be a telephoto zoom and a large value ($\approx \pi = 3.14\dots$) a wide-angle lens. The viewangle parameter only affects the perspective drawing. A setting of zero or greater than 6.28 means parallel projection, otherwise perspective projection.

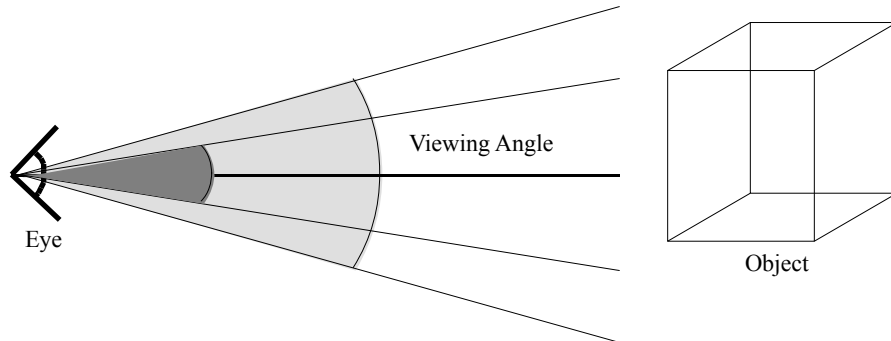


fig II.8: Viewing angle

ClipType tells the eye what kind of clipping the eye transformation should perform. Clipping only has an effect if you use the provided Draw methods and works even when you switch the eye off. There are three clipping techniques currently implemented:

- *None*: No clipping provided (fastest). Use it only if you are sure no part of the object can get very close to or behind the eye.
- *Arithmetic*: This (slowest) variant clips all lines that penetrate the projection plane to the point where they intersect it. Lines that are completely behind the projection plane are removed.
- *Fast*: This technique eliminates all lines that at least fall partially behind the projection plane (i.e. have a negative z component).

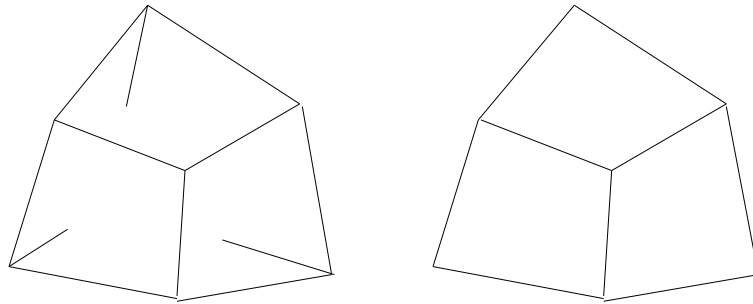


fig II.9: Clipping techniques. Arithmetic (left) and fast (right)

The following clipping types and projection types are defined by GrafSys:

```
Type
ProjectionTypes = (parallel, perspective);
clippingType = (none, arithmetic, fast);
```

Other clipping or projection types can be defined. In that case you have to extend the Draw and Transform methods to support them.

```
procedure GetEye (var UsesEye: Boolean;
  var location: Vector4; thePhi, theTheta,
  thePitch, theViewangle: real;
  var clipType: clippingType);
```

GetEye returns the currently active eye's settings (these are in turn derived from the currently active 3D port). See SetEye for a description of the parameters.

```
procedure ToScreen (thePoint: Vector4;
  var h, v: INTEGER);
```

Given a 3D point in *world coordinates*, ToScreen applies the Eye Transformation (if the eye is switched on) and then the selected projection (parallel or perspective) and returns the screen coordinates for this point.

```
procedure ProjectPoint (thePoint: Vector4;
  var h, v: integer);
```

Given a 3D point in *world coordinates* ProjectPoint will use the currently selected projection method (parallel or perspective) and return the screen coordinates for this point. Unlike ToScreen no eye

GrafSys Documentation

transformations are applied (i.e. this procedure is equal to a call to ToScreen with UseEye set to FALSE).

Vector and Matrix Manipulation

The GrafSys has built-in Vector and Matrix manipulation procedures. These work on the Vector4 and Matrix4 types that are GrafSys version-dependent (FPU or fixed-point arithmetic). Use them if you need efficient Vector-Matrix and Matrix-Matrix multiplication or Vector-Vector addition/subtraction. You would need these routines only if you want to implement your own transformation methods (which is strongly recommended against).

```
procedure InitMatrix;
```

A call to this routine initializes the Matrix Package. You should never call it yourself because it is called indirectly if you initialize the GrafSys.

```
function Identity: Matrix4;
```

This procedure returns an Identity matrix of type Matrix4. Identity is defined as

$$\text{Identity} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For better performance the Matrix4 type carries an IdentityFlag to test if the matrix is identity. This is set to TRUE.

```
function MMult (var A, B: Matrix4): Matrix4;
```

Ultra-fast, efficient Matrix4 x Matrix4 multiplication procedure. Tests to see if one matrix is Identity. The result will always have the Identity flag killed except if both matrices are Identity. Result := A x B

```
function VMult (x: Vector4; var A: Matrix4)
: Vector4;
```

Ultra-fast, efficient Vector4 x Matrix4 multiplication procedure.

GrafSys Documentation

$$\text{Result} = [x \quad y \quad z \quad 1] * \begin{bmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix}$$

```
function Transpose (A: Matrix4): Matrix4;
```

Generates the transpose of Matrix A. It is used normally to calculate the inverse matrix of a rotation matrix since their inverse is equal to their transposed form (This is much faster than calculating the inverse using Gauss-Jordan or the Givens-rotational techniques).

```
function VSub (x, y: Vector4): Vector4;
```

Subtracts the Vector4 y from the Vector4 x.

```
function VAdd (x, y: Vector4): Vector4;
```

Adds y to x.

GrafSys Documentation

Off-Screen Buffers (Pixel Maps)

To support flicker-free animations and fast polygon (triangle) filling, GrafSys provides you with routines to efficiently manipulate off-screen pixel maps.

What it is

Animating an object usually involves an erase-redraw cycle that is repeatedly executed, thus giving the impression of movement. Since both erasing and redrawing usually take a while, the image on the screen flickers. Flicker occurs whenever the erase-redraw cycle takes longer than the monitor's scan beam needs to draw the screen (actually it occurs whenever your object drawing and the scan-line drawing collide, i.e. part of the image is redrawn, part is still erased/old).



fig II.OS1: The animation cycle (from left to right). The old image is first erased and then redrawn. On slow machines this will produce flicker.

This becomes more noticeable the slower the machine you are using is. To reduce or avoid flicker we need a way to instantaneously put the image on-screen. Usually this is done drawing the image in a buffer off-screen (called off-screen pixel map) and then copying it over the old image on-screen thus redrawing and erasing the image simultaneously.

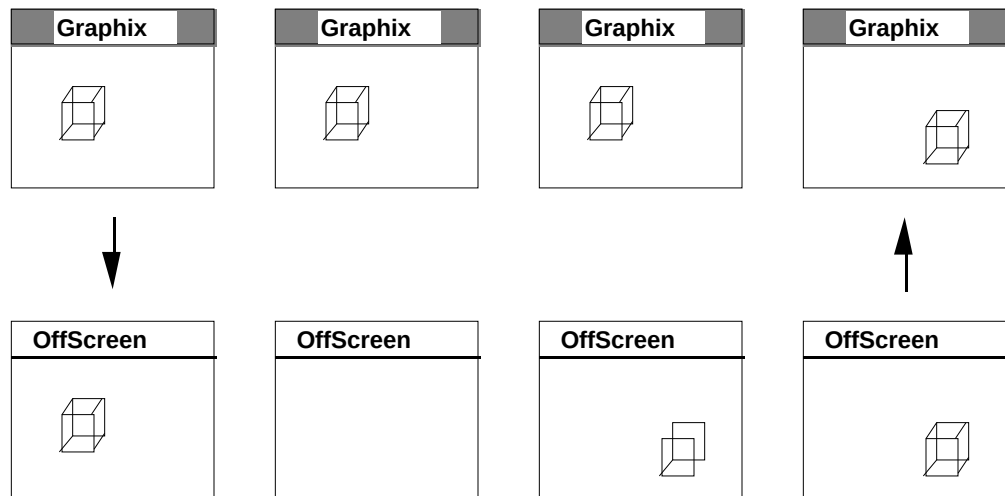


fig II.OS2: The off-screen buffer contains the same data as the on-screen image. The image is erased and redrawn off-screen and then copied to the screen using very fast bit-copy procedures. That way the image on the screen is replaced instantaneously

However, off-screen pixel maps are usually not trivial to use and require a sound understanding of the Macintosh OS. Therefore, GrafSys provides you with a simpler concept that makes the usage of off-screen pixel maps much easier to use. The trade-off is that with GrafSys off-screen manipulation methods you lose some of the flexibility advanced programmers have when they implement off-screen PixMaps themselves.

Using Off-Screen Pixel Maps

In 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

GrafSys Documentation

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;
```

GrafSys Documentation

```
cCantChangeOffscreen      = -16  
cNoOSAttached              = -17;
```

GrafSys Documentation

```
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 pointed to by theWindow or want to change the color table used with the off-screen buffer. The off-screen buffer will be resized to match the 3D window's size (as defined by theWindow^.portRect).

TheColors contains a handle to the color table you want to use with the off-screen PixMap. Pass -1 if you do not want to change it.

If successful, ChangeOffScreen returns noErr. Otherwise it will not change anything and return an error-code that can be interpreted with the standard GrafSys InterpretError procedure.

```
function CloseOffscreen (theWindow: WindowPtr)  
    : integer;
```

Use CloseOffscreen to dispose of the off-screen buffer attached to the 3D window pointed to by theWindow. If you use the Dispos3DWindow routine described above to destroy the 3D window you do not have to call CloseOffScreen. It will be called implicitly by Dispos3DWindow. But it is still

GrafSys Documentation

a good idea to deallocate the off-screen buffer explicitly because if you forget to do so, you will waste a lot of memory.

GrafSys Documentation

If successful, CloseOffScreen returns noErr. Otherwise it will not delete anything and return an error-code that can be interpreted with the standard GrafSys InterpretError procedure.

```
function BeginOSDraw (theWindow: WindowPtr)
    : integer;
```

Calling this function will cause all further drawing to be done on the off-screen buffer that is attached to the 3D window pointed to by theWindow.

Note: BeginOSDraw will in effect issue a QuickDraw SetPort and SetGDevice call to the off-screen pixel map that is attached to the 3D window. If you want to temporally draw to other windows, make sure you call EndOSDraw first. Otherwise nothing will appear on the screen!

If successful, BeginOSDraw returns noErr. Otherwise it will not do anything and return an error-code that can be interpreted with the standard GrafSys InterpretError procedure.

```
function EndOSDraw (theWindow: WindowPtr): integer;
```

This function complements BeginOSDraw. It restores drawing to the window pointed to by theWindow.

Note: EndOSDraw also restores the current GDevice. If you started drawing to an off-screen pixel map, you should call EndOSDraw before attempting to draw onto the screens.

If successful, EndOSDraw returns noErr. Otherwise it will not do anything and return an error-code that can be interpreted with the standard GrafSys InterpretError procedure.

```
function CopyOS2Screen (theWindow: WindowPtr;
    theRect: Rect; copyMode: Integer): integer;
```

Use CopyOS2Screen to copy a portion of the off-screen buffer to the corresponding portion of the 3D window pointed to by theWindow. The portion is defined by the rectangle theRect (in local Macintosh window coordinates). CopyMode defines the way the pixels should be copied and is

GrafSys Documentation

one of the normal Macintosh transfer modes (`srcCopy` etc.).

GrafSys Documentation

```
procedure FastPixErase (pixH: PixMapHandle;  
    color: integer);
```

FastPixErase is a special assembler routine that erases the 8 bit deep off-screen pixel map whose handle is stored in pixH. The whole pixel map is set to the value passed in color (a value between 0 and 255).

Warning:

FastPixErase does not check if the pixel map is actually 8 bits deep. It calculates the size by the number of bytes to set using the rowBytes and bounds entries in the PixMap. If the PixMap actually uses less than 8 bits per pixel, other memory gets erased and your program will probably crash (if you are lucky).

Warning:

Never use FastPixErase for on-screen pixel maps! Your program will definitely crash and the screen will be totally messed up!

GrafSys Documentation

Hidden-Line and Hidden-Surface Animations

As mentioned before, GrafSys only provides you with the tools to implement hidden-line and hidden-surface removal/animation. The simplest method, the Painter's Algorithm, will be discussed now.

GrafSys supplies you with special routines for ultra-fast 2D triangle filling. If you have surfaces in your object model them using triangles. Enter the points into the object as you would normally.

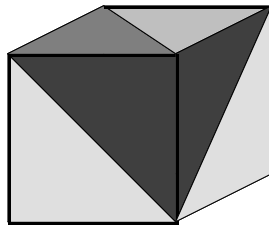


fig HS.1 : Modeling a cube using triangles

Now, instead of drawing lines to visualize the object, you draw triangles using `FillTriangle`. Use the converted point's screen coordinates to define the triangle's corners. The object will appear solid compared to the so-called wire frame rendering the GrafSys uses. Any polygon can be modeled using triangles, so this approach is universal.

However, before you draw the triangles, you should do two steps:

- If your object is convex (such as the cube), you can eliminate those surfaces (triangles) that are invisible using a technique called 'Back-Face Removal'. Use the `IsVisible` procedure for detecting those triangles. Note that you do not have to call `IsVisible` every time. In the above example, a surface is always made up by two triangles. If one of them is invisible, so will be the other. If you model objects by surfaces, test only once for visibility of a given surface, triangulate and draw only if visible.
- Prior to drawing you must sort the triangles according to their (transformed) deepest Z coordinate. That way, surfaces that are further removed from the eye are drawn first. Those parts that are closer to the eye and obscure parts of the other surface are drawn over it. Hidden surfaces are thus removed automatically.

If you use the supplied off-screen drawing procedures to build the object off-screen and then use `CopyOS2Screen` to transfer it onto the screen you have impressive hidden-surface animations at almost no additional cost.

GrafSys Documentation

Predefined Objects In The GrafSys

GrafSys is a combination of global procedures used to communicate with the Mac OS (the window management routines) and a class library for the actual 3D transformation and drawing. The various ancestors of the final objects aren't really important except if you want to extend the library at a lower level yourself. The following sections will introduce you to the GrafSys Class Library from the bottom up. Note that the single most important Object in GrafSys is the TSOBJECT3D.

The Object Hierarchy

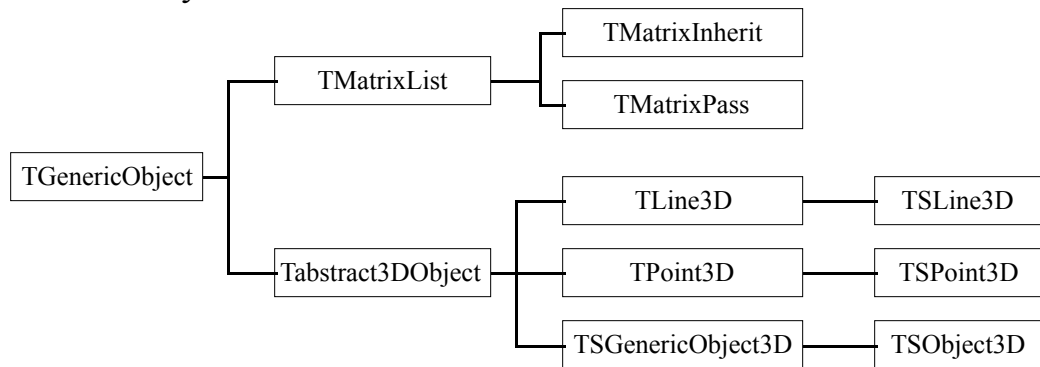


fig II.10: Class Hierarchy

The hierarchy of the GrafSys objects is quite simple to explain:

- Root is the TGenericObject that implements a common base for all GrafSys objects and contains methods to clone, kill, test etc. objects.
- The next level contains the first specialized objects that implement the core functionality of all their children. There are two such objects one for the transformation operators (TMatrixList) and one for 3D objects (TAbstract3DObject).
- The following levels all add to the functionality while becoming more and more specialized. While the sons of the TMatrixList are quite uninteresting and exist solely for implementation purpose of dynamic transformation inheritance the subclassing of the TAbstract3DObject is quite interesting. It splits into three distinct subclasses, one each for 3D Point, 3D Line and 3D Object.
- The last Level, indicated by a leading 'TS' implement the front-end interface for Macintosh programming and are the classes you should subclass if you intend to extend the hierarchy. These objects know how to draw on the Macintosh screen (into windows), how to use the eye and how to clip.

The Base Type

TGenericObject is the base object of GrafSys. Here you find the methods for housekeeping, reporting errors and other things.

GrafSys Documentation

The Matrix Types

The Matrix types are the transformation operators the GrafSys uses for inheritance and free-order transformation. They know how to rotate, scale, translate and handle inheritance.

The Point Type

The first 3D oriented type is the point. This object knows how to transform and in later incarnations how to draw themselves.

The Line Type

A double-point connected with a line would be a better description. It could have been implemented as a child of the 3D Point type but it was decided to define it as a full-fledged own class directly inheriting from the `TAbstract3DObject` since this makes it a bit more flexible

The Object Type

The object type is both ancestor 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, `TObject3D` is the single most powerful object in the class hierarchy and should be the base of most of your class extensions.

How To Use `TObject3D`

Since `TObject3D` 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 `TObject3D` can inherit transformation sequences from other `TObject3D` objects and knows how to draw itself on the screen.

If you want to use the `TObject3D` 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.

GrafSys Documentation

- 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 `TObject3D` was designed to hold a very large amount of data it allocates memory dynamically in order to minimize memory waste. If you clone a `TObject3D` 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 `TObject3D`. 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`.

GrafSys Documentation

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 its 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

GrafSys Documentation

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`, `FFRotArbAchs` and `FFReset`. To allocate and pre- or post-concatenate the operators use `FFNewPreConcat` and `FFNewPostConcat`.

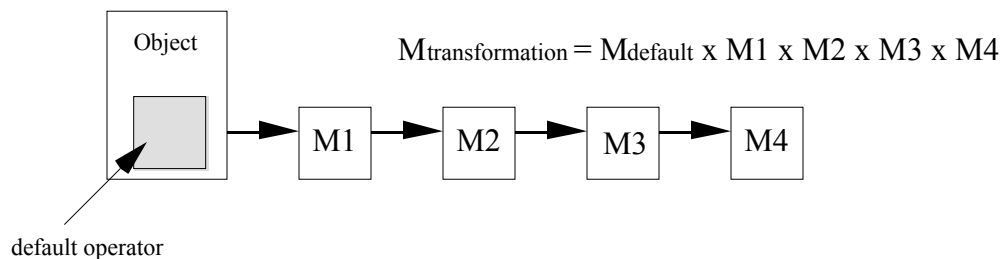


fig II.11 : Functionality of the FF operators

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 `TObject3D` only has two additional attributes that you can change. They are the *AutoErase* and *UseBounds* attributes. When *AutoErase* is set, calling the `Draw` or `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 `TObject3D` 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.

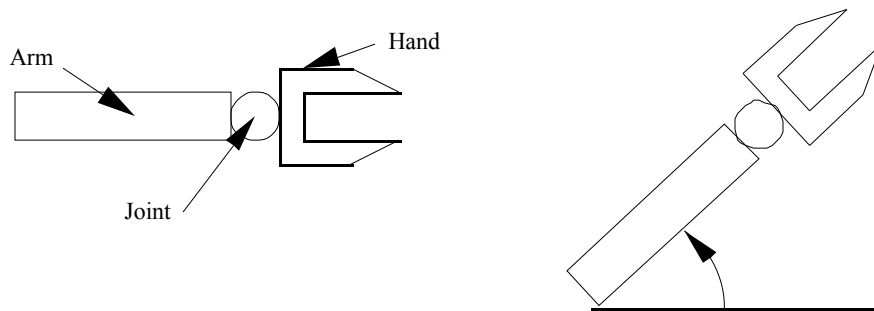


fig II.12 : Robot arm and hand (left) and rotated (right)

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.

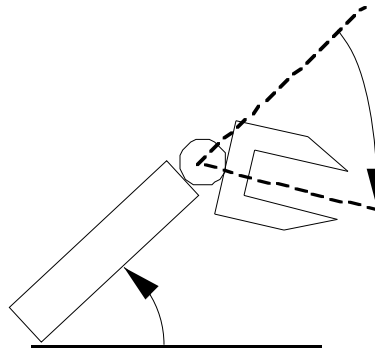


fig II.13: Hand rotated relative to rotated arm

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.

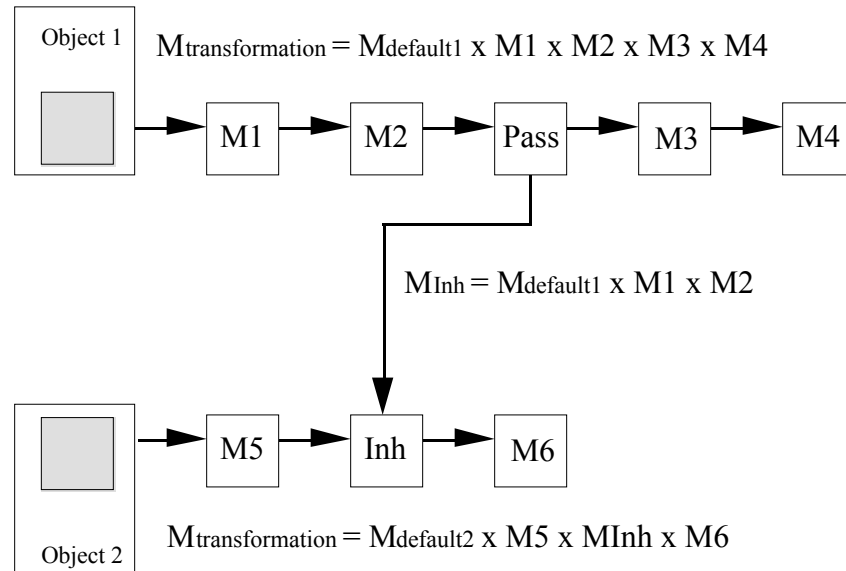


fig II.14a: Object 2 inherits free transformations from Object 1

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 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.

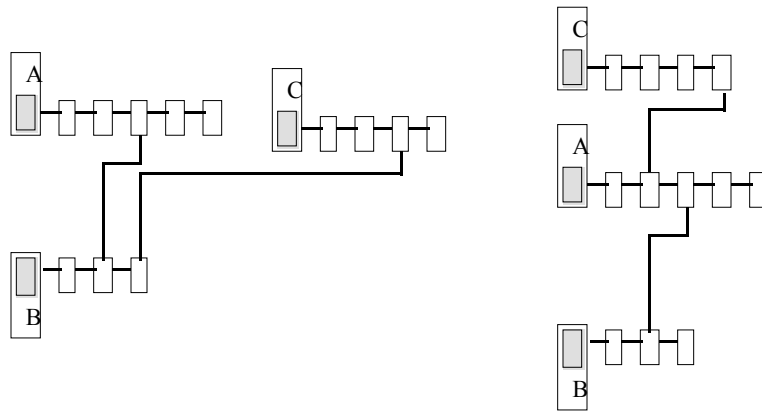


fig II.14b: Multiple inheritance (left) and corrected inheritance (right)

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.

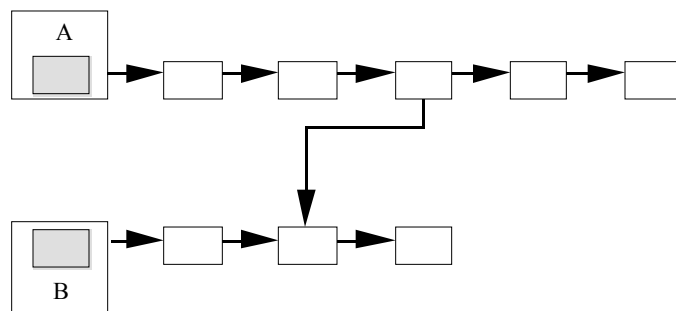


fig II.15a: B inherits from A

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:

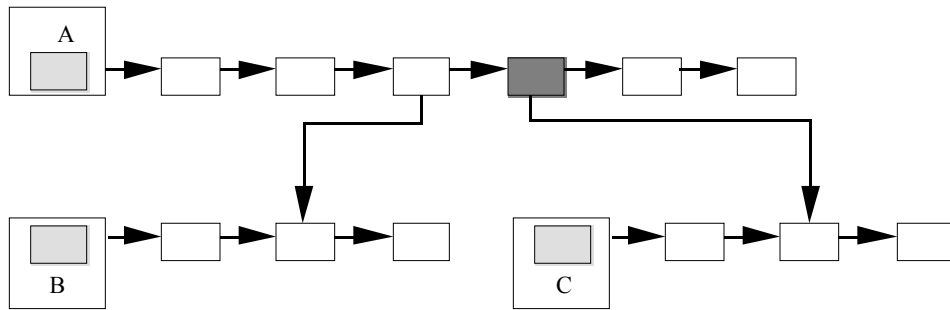


fig II.15b: C is cloned from B. A new PassOn operator was installed in father object

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!

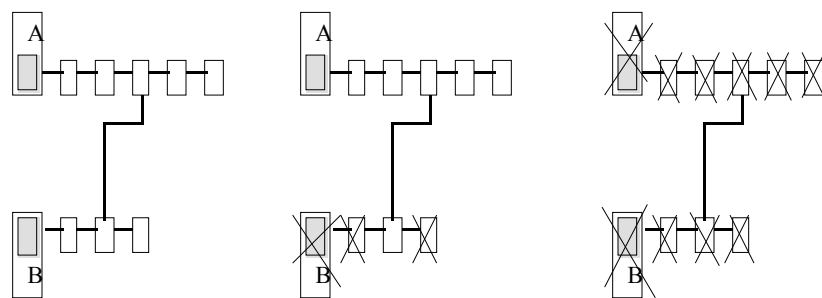


fig II.16: B inherits from A. Effect of killing B (middle) versus killing A (right)

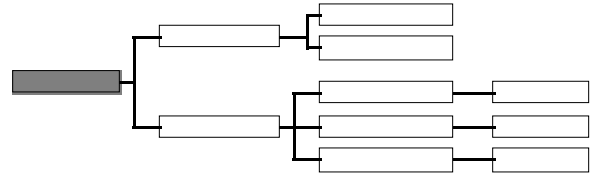
This means that if a son is father to another object, the 'grandson' will be killed as well if its father is killed etc.

GrafSys Documentation

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

GrafSys Documentation

		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.

GrafSys Documentation

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

GrafSys Documentation

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:

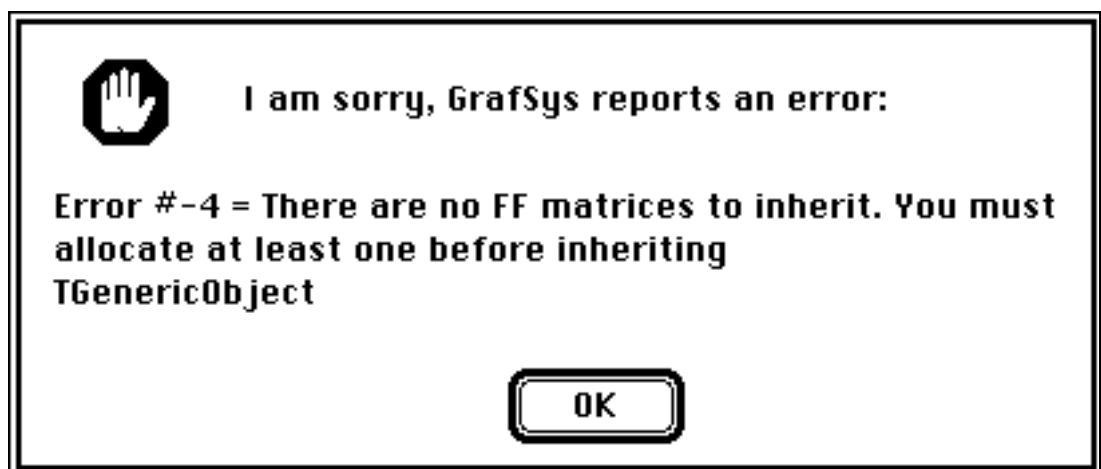


fig II.17: Standard error alert

```
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:

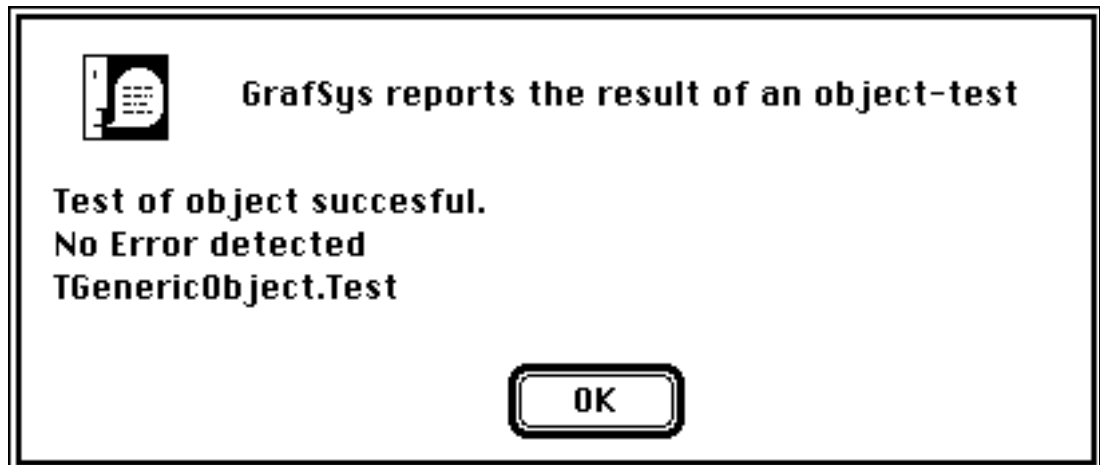


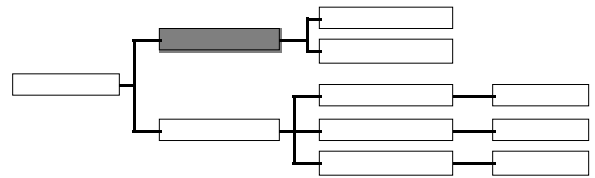
fig II.18: Standard test notification alert

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



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 TSOject3D 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 TSOject3D 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 TSOject3D.

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;
```

GrafSys Documentation

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

GrafSys Documentation

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.

GrafSys Documentation

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

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

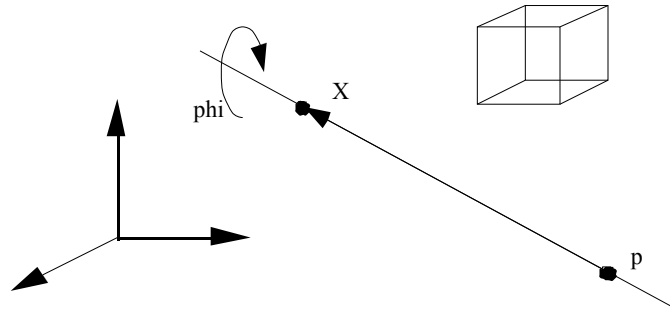
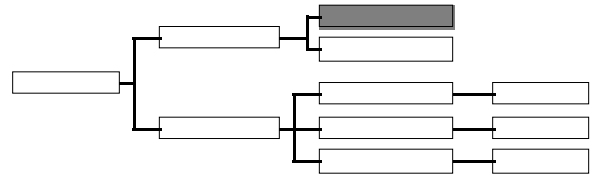


fig II.19: Arbitrary rotation

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

The result is multiplied with M and stored in M .



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

Superclass	TMatrixList
Subclasses	none

Don't.

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

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);
```

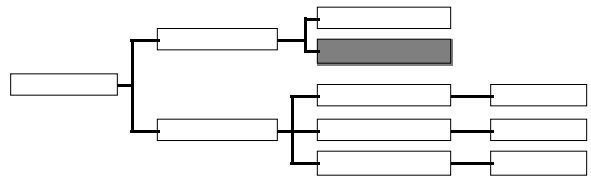
```
procedure Init;  
override;
```

GrafSys Documentation

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

GrafSys Documentation

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

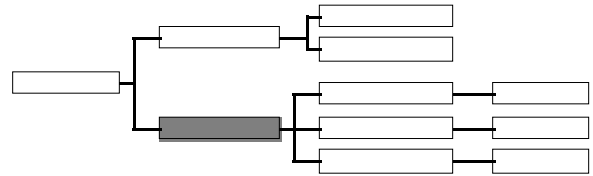
Inherited methods:

```
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.



Introduction

Tababstract3DObject 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 Tababstract3DObject

Directly after allocating the Tababstract3DObject 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

GrafSys Documentation

		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 <code>calcTransform</code> will reset it.
versionsID	longint	Used for synchronization with eye to detect if a recalculation of <code>xForm</code> is required since <code>xForm</code> also holds eye transformation
hasChanged	Boolean	Indicates that a <code>calcTransform</code> call changed the transformed point description. This is used to flag to <code>Draw</code> 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.

GrafSys Documentation

```
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.

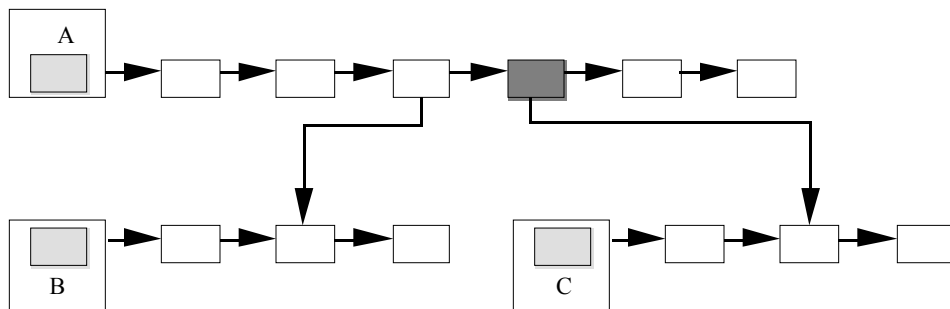


fig II.20: The result of cloning B yielding C

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

GrafSys Documentation

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 xScale, yScale and zScale.

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

Scale sets the scaling factors for the object to 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 scaling 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.

SetScale operates on the default operator as reflected by the object's variables xScale, yScale and zScale.

GrafSys Documentation

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

RotArb rotates the object phi radians further around an axis defined by the two 3D points p and x. RotArb operates on one of the default operators (on the arbRot operator to be specific).

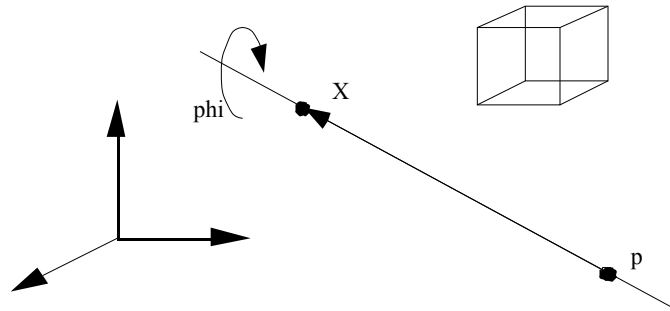


fig II.21: Arbitrary rotation

The rotational axis is defined as the line connecting p with x, looking from p to x. A positive angle means clockwise rotation. Note that the points p and x are given in the object's local coordinate system.

Note: Using RotArb with an axis that does not run through the objects origin will make the coordinates (xTrans, yTrans and zTrans) of the object reflect incorrect positions since it moves the origin. Transformation will still be correct but you must use ForeignPoint with [0,0,0] to get the correct origin in world coordinates.

Note: The results of this command are strongly dependent on the order in which you call them. If you have two different axes called a1 and a2, first rotating around axis a1 and then around a2 gives a different result than first rotating around a2 and then around a1. You should really be knowing what you are doing if you are using this command. I strongly recommend to use only one arbitrary rotation and do not mix it with other default transformations.

```
procedure ResetArb;
```

This method resets the default operator RotArb to Identity.

GrafSys Documentation

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

Displaces the object's origin for the vector [dx,dy,dz]. FFTranslate operates on the currently active FF operator indicated by currentFF. Note that this translation is applied after other transformations are applied. They do not affect the settings of xTrans, yTrans or zTrans, so to get the correct position of the object's origin you must use the ForeignPoint method with [0,0,0] as argument.

```
procedure FFRotate (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.

FFRotate operates on the currently active FF operator indicated by currentFF. Note that this rotation is applied after other transformations are applied. They do not affect the settings of xTrans, yTrans, zTrans, xrot, yrot or zrot. To get the correct position of the object's origin you must use the ForeignPoint method with [0,0,0] as argument.

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

FFScale increments the scaling factors for the object pointed to by theObject by the given values.

FFScale operates on the currently active FF operator indicated by currentFF. Note that this scaling is applied after other transformations are applied. They do not affect the settings of xTrans, yTrans, zTrans, xrot, yrot, zrot, xScale, yScale or zScale. To get the correct position of the object's origin you must use the ForeignPoint method with [0,0,0] as argument.

GrafSys Documentation

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

FFRotArbAchsis rotates the object phi radians further around an axis defined by the two 3D points p and x. FFRotArbAchsis operates on the currently active FF operator indicated by currentFF.

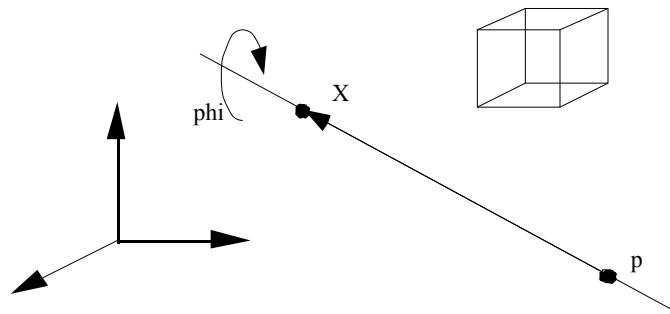


fig II.21: Arbitrary rotation

The rotational axis is defined as the line connecting p with x, looking from p to x. A positive angle means clockwise rotation. Note that the points p and x are given in the object's local coordinate system.

Note that this transformation is applied after other transformations are applied. They do not affect the settings of xTrans, yTrans, zTrans, xrot, yrot or zrot. To get the correct position of the object's origin you must use the ForeignPoint method with [0,0,0] as argument.

Note: The results of this command are strongly dependent on the order in which you call them. If you have two different axes called a1 and a2, first rotating around axis a1 and then around a2 gives a different result than first rotating around a2 and then around a1. You should really be knowing what you are doing if you are using this command. I strongly recommend to use only one arbitrary rotation per operator and do not mix it with default transformations.

```
procedure FFReset;
```

This command resets the currently active FF matrix (indicated by currentFF) to Identity.

GrafSys Documentation

```
function FFNewPostConcat: TMatrixList;
```

This message causes a new FF operator to be allocated and appended to the chain of FF operators.

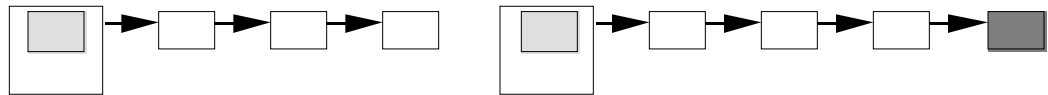


fig II.22: Post concatenation of operator. Left before operation, right after.

The currentFF field is set to the new operator.

```
function FFNewPreConcat: TMatrixList;
```

This message causes a new FF operator to be allocated and placed in front of the chain of FF operators.

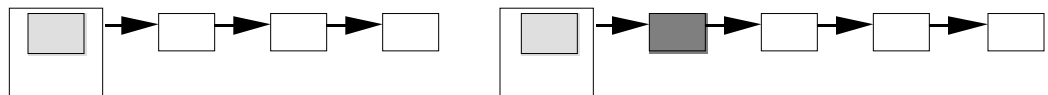


fig II.23: Pre concatenation of operator. Left before operation, right after.

The currentFF field is set to the new operator.

```
function FFActivate (theFF: TMatrixList)
    : Boolean;
```

Makes theFF the currently active operator. FFActivate returns true if operation successful, FALSE otherwise. If theFF does not belong to the object, FFActivate will also return false.

```
function FFPassOn: TMatrixPass;
```

Use this method to generate a link in the father's operator list. It returns a PassOn operator that is subsequently used in another object to receive the transformation. The PassOn operator is always appended to the list of current operators.

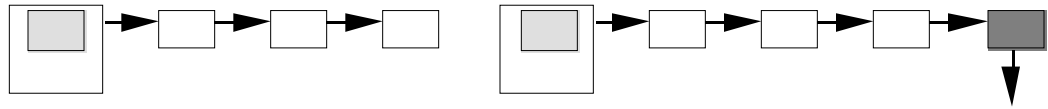


fig II.24: Father object (left) before calling FFPassOn, right with new PassOn operator

FFPassOn is usually directly followed by a FFInherit message to the son object with the result of this call as parameter:

```
theLink := father.FFPassOn;
son.FFInherit(theLink);
```

or even better

```
son.FFInherit(father.FFPassOn);
```

Never (ever!) call FFInherit twice with the same PassOn operator or you would lose at least one link and the result of killing or cloning is unpredictable.

The setting of currentFF is unchanged.

```
procedure FFInherit (var FatherList:
    TMatrixPass);
```

This method takes a PassOn operator from a father object and generates an Inherit operator that is placed at the end of the current FF chain. On all subsequent calls to calcTransform the father's transformation results are included at the position where the inherit operator is located.

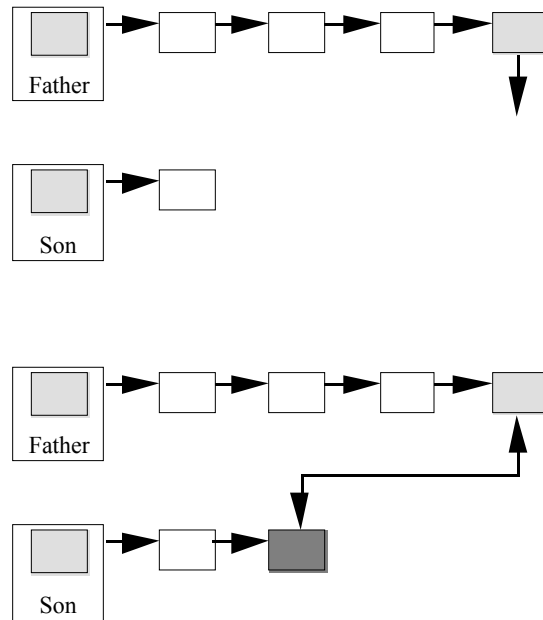


fig II.25: Son inherits from father. Top before calling FFinherit, bottom after.

Never (ever!) call FFinherit twice with the same PassOn operator or you would lose at least one link and the result of killing or cloning is unpredictable.

The setting of currentFF is unchanged.

```
procedure CalcTransform;
```

This is the central calculation call. The different operators are evaluated and the result is stored in the xForm field. The order of evaluation is:

- x-rotation (from default operator)
- y-rotation (from default operator)
- z-rotation (from default operator)
- Translation (from default operator)
- Scaling (from default operator)
- Arbitrary rotation as specified by RotArb
- All FF operators in the order they appear in the FF chain.

Note that no eye transformation is applied at this level.

If a PassOn operator is encountered while evaluating the FF chain, the current result is placed into the Inherit operator of the son.

GrafSys Documentation

```
function ForeignPoint (p: Vector4): Vector4;
```

Using the current xForm operator the point p (in model coordinates) is converted to world coordinates. No eye transformation is applied.

If you haven't done so, you must first call CalcTransform to set up the xForm operator prior to calling ForeignPoint.

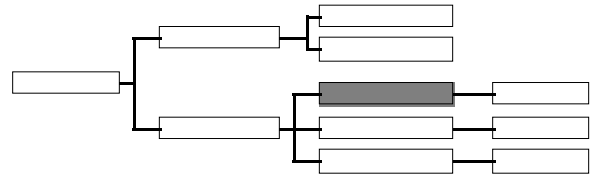
```
function WorldToModel(wc : Vector4) : Vector4;
```

This is the reverse to ForeignPoint. It transforms a point given in world coordinates to a point in model coordinates according to xForm. Again, make sure that you called CalcTransform to initialize the xForm field.

```
procedure Draw;
```

Draw is not supported at this level since Tabstract3DObject knows nothing of points or the like. It's just included to provide a common method over the whole 3D object tree.

Calling this method will only set the ErrorCode field to cBadMethodCall.



Introduction

TLine3D is a general-purpose object to support 3D lines at an abstract level. There should never be a reason why you want to use this object. It's only included to make the separation between levels of the GrafSys consistent. This level knows about points so the TLine3D supports the two logical points: Startpoint and Endpoint of the 3D line.

Heritage

Superclass Tabstract3DObject
Subclasses TSLine3D

Using TLine3D

TLine3D is the next step up from Tabstract3DObject. It supports manipulation of two points (the start- and endpoints) and eye transformation. Handling is like Tabstract3DObject.

Variables

Name	Type	Description
FromLoc	Vector4	A 3D point describing the start point of the line in model coordinates.
ToLoc	Vector4	A 3D point describing the end point of the line in model coordinates.

Methods

Inherited methods:

```

procedure HandleError;
procedure ResetError;
function Test (opcode: integer): integer;
function Clone: TGenericObject;
procedure Translate (dx, dy, dz: real);
procedure SetTranslation (x, y, z: real);
procedure Rotate (dx, dy, dz: real);
procedure SetRotation (x, y, z: real);
procedure Scale (dx, dy, dz: real);

```


GrafSys Documentation

```
procedure SetScale (x, y, z: real);  
procedure RotArb (p, x: Vector4; phi: real);  
procedure ResetArb;
```

GrafSys Documentation

```
procedure FFTranslate (dx, dy, dz: real);
procedure FFRotate (dx, dy, dz: real);
procedure FFScale (dx, dy, dz: real);
procedure FFRotArbAchsis (p, x: Vector4;
    phi: real);
procedure FFReset;
function FFNewPostConcat: TMatrixList;
function FFNewPreConcat: TMatrixList;
function FFActivate (theFF: TMatrixList)
    : Boolean;
function FFPassOn: TMatrixPass;
procedure FFinherit (var FatherList:
    TMatrixPass);
procedure CalcTransform;
function ForeignPoint (p: Vector4): Vector4;
function WorldToModel(wc : Vector4) : Vector4;
procedure Draw;
procedure Kill;
```

Other Methods:

```
procedure Init;
override;
```

Init calls the inherited Init method and then initializes the start point and endpoint to [0,0,0].

```
procedure Reset;
override;
```

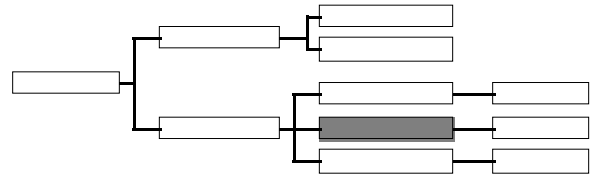
Reset is like Init except that it does not call the inherited Init but the inherited Reset instead (i.e. resetting all operators). Startpoint and endpoint are set to [0,0,0].

```
procedure SetKoords (K1, K2: Vector4);
```

Use this procedure to set the coordinates of the startpoint and endpoint of the 3D line.

```
procedure GetKoords (var K1, K2: Vector4);
```

This method returns the coordinates of startpoint and endpoint of the 3D line in K1 and K2, respectively.



Introduction

TPoint3D is a general-purpose object to support 3D points at an abstract level. There should never be a reason why you want to use this object. It's only included to make the separation between levels of the GrafSys consistent. This level knows about points.

Heritage

Superclass TAbstract3DObject

Subclasses TSPoint3D

Using TPoint3D

TPoint3D is the next step up from TAbstract3DObject. It supports manipulation of one point and eye transformation. Handling is like TAbstract3DObject.

Variables

Name	Type	Description
Koord	Vector4	The coordinates of the point in model coordinates.

Methods

Inherited methods:

```
procedure HandleError;  
procedure ResetError;  
function Test (opcode: integer): integer;  
function Clone: TGenericObject;  
procedure Translate (dx, dy, dz: real);  
procedure SetTranslation (x, y, z: real);  
procedure Rotate (dx, dy, dz: real);  
procedure SetRotation (x, y, z: real);  
procedure Scale (dx, dy, dz: real);  
procedure SetScale (x, y, z: real);  
procedure RotArb (p, x: Vector4; phi: real);  
procedure ResetArb;  
procedure FFTranslate (dx, dy, dz: real);  
procedure FFRotate (dx, dy, dz: real);  
procedure FFScale (dx, dy, dz: real);
```

GrafSys Documentation

```
procedure FFRotArbAchsis (p, x: Vector4;  
    phi: real);  
procedure FFReset;  
function FFNewPostConcat: TMatrixList;  
function FFNewPreConcat: TMatrixList;
```

GrafSys Documentation

```
function FFActivate (theFF: TMatrixList)
    : boolean;
function FFPassOn: TMatrixPass;
procedure FFInherit (var FatherList:
    TMatrixPass);
procedure CalcTransform;
function ForeignPoint (p: Vector4): Vector4;
function WorldToModel (wc : Vector4) : Vector4;
procedure Draw;
procedure Kill;
```

Other Methods:

```
procedure Init;
override;
```

Init calls the inherited Init method and then initializes the coordinates to [0,0,0].

```
procedure Reset;
override;
```

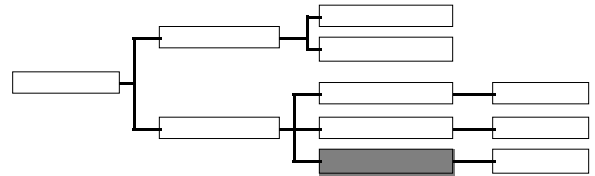
Reset is like Init except that it does not call the inherited Init but the inherited Reset instead (i.e. resetting all operators). The coordinates are set to [0,0,0].

```
procedure SetKoords (Koordinates: Vector4);
```

Use this procedure to set the coordinates of the 3D point.

```
function GetKoords: Vector4;
```

This method returns the coordinates of the 3D point.



Introduction

The next level of the general-purpose 3D object introduces points and eye transformation. Methods are provided to extend and modify the object's point data base and to project them to screen coordinates.

Heritage

Superclass Tabstract3DObject
Subclasses TSOBJECT3D

Using TSGenericObject3D

Use this object if you want to be able to manipulate a large mass of 3D points but do not need the ability to manipulate lines as in the TSOBJECT3D subclass or want to implement different line algorithms yourself. Handling is like Tabstract3DObject except for the additional methods for adding and deleting points.

Points are always referenced by their index number. Thus, the third point you add will be referenced by 'point index 3'. Note that this number is not static since TSGenericObject3D supports methods to remove points and if you delete 'point index 2' the third point will become the new number two.

Variables

Name	Type	Description
theBufs	array[0..MaxBuffers] of Point3DBufPtr	Array that points to the buffers for 3D points.
currentBuf	Point3DBufPtr	Points to the currently used Point buffer. Only used for fast sequential access.
currentIndex	integer	Index of currently used point. Internal use only. Cannot be relied on to contain a valid number.
numPoints	longint	Number of points in object's data base
Bounds	Rect	Rectangle specifying the current bounding rectangle.
oldBounds	Rect	Rectangle containing

GrafSys Documentation

screenXform Matrix4

last bounding rectangle.
Transformation matrix
used for all point
transformations. This
matrix includes the eye
transformations if
UseEye is set in the
current 3D GrafPort,
else it's equal to xForm.

Methods

Inherited methods:

```
procedure HandleError;
procedure ResetError;
function Test (opcode: integer): integer;
procedure Translate (dx, dy, dz: real);
procedure SetTranslation (x, y, z: real);
procedure Rotate (dx, dy, dz: real);
procedure SetRotation (x, y, z: real);
procedure Scale (dx, dy, dz: real);
procedure SetScale (x, y, z: real);
procedure RotArb (p, x: Vector4; phi: real);
procedure ResetArb;
procedure FFTranslate (dx, dy, dz: real);
procedure FFRotate (dx, dy, dz: real);
procedure FFScale (dx, dy, dz: real);
procedure FFRotArbAchsis (p, x: Vector4;
    phi: real);
procedure FFReset;
function FFNewPostConcat: TMatrixList;
function FFNewPreConcat: TMatrixList;
function FFActivate (theFF: TMatrixList)
    : boolean;
function FFPassOn: TMatrixPass;
procedure FFInherit (var FatherList:
    TMatrixPass);
procedure CalcTransform;
procedure Draw;
```


GrafSys Documentation

Other Methods:

```
procedure Init;  
override;
```

The Init method will first call all inherited Init methods and then initialize the bounding rectangles to empty rectangles. All point buffer pointers are initialized (to nil), numPoints set to -0 (no points in database), currentIndex to -1 and a first point buffer is allocated. currentBuf is set to this buffer as well as the first buffer pointer (theBuf[0]).

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

This method will first call the inherited Clone methods and then clone any allocated buffers for 3D points.

```
procedure Reset;  
override;
```

Reset will call all inherited reset routines (i.e. resetting all operators) and then reset the bounding rectangles to empty.

```
procedure Kill;  
override;
```

Kill will first deallocate all allocated point buffers and then call all inherited kill methods (i.e. killing all operators and sons).

```
procedure GenIndex (pointIndex: longint;  
                    var BufIndex, bufOffset: integer);
```

For internal use only, GenIndex calculates the buffer number and the offset into the buffer a point with pointIndex has. If you want to access a point, always use the ChangePoint or GetPoint methods described below.

```
function AddPoint (x, y, z: real): longint;
```

GrafSys Documentation

AddPoint will add a 3D point with the coordinates $[x,y,z]$ to the current point database. It returns the point's index if successful or -1

GrafSys Documentation

if not. The reason why AddPoint failed can be read from ErrorCode.

Note that the external point count is one-based while the internal representation is zero-based. To avoid confusion, always use the provided methods to access points and never try to access points directly.

```
function DeletePoint (index: longint): Boolean;
```

DeletePoint will remove the point whose index you specified from the object's database. If the method is successful, it returns TRUE, false otherwise (ErrorCode contains the reason).

All points beyond index will be moved forward once.

Note that DeletePoint does not deallocate any buffers that have been allocated even if one becomes completely free. Only the Kill method will release memory that was allocated.

If you pass -1 as index to the point to remove, *all points in the database will quickly be deleted.*

```
procedure GetPoint (index: longint;  
                  var x, y, z: real);
```

GetPoint returns the coordinates of the point whose index you specified. If you specify an illegal index, it will return [0,0,0] as coordinates and ErrorCode will be set.

```
function ChangePoint (index: longint; x, y, z:  
                    real): Boolean;
```

ChangePoint changes the coordinates of the point whose index you specified to the new coordinates [x,y,z]. It returns TRUE if the operation was successful, FALSE otherwise (ErrorCode set).

```
procedure Transform (forceCalc: Boolean);
```

Transform will calculate the transformation matrix (xForm) and apply the eye transformations from the currently active 3D GrafPort. Then all points in the database are transformed.

Warning: A 3D GrafPort **must** be open!

GrafSys Documentation

Eye transformations are applied only if the UseEye flag is set in the currently active 3D GrafPort. The active projection type is applied no matter what the setting of UseEye.

```
procedure Transform2 (forceCalc: Boolean);
```

Transform2 will calculate the transformation matrix (xForm) and apply the eye transformations from the currently active 3D GrafPort. Then all points in the database are transformed. When transforming the points, Transform2 gathers data for the bounding rectangle and stores them in Bounds.

Warning: A 3D GrafPort **must** be open!

Eye transformations are applied only if the UseEye flag is set in the currently active 3D GrafPort. The active projection type is applied no matter what the setting of UseEye.

```
function TransformedPoint (index: longint)
    : Vector4;
```

Call TransformedPoint to get the world coordinates of a point whose index you specified. Depending on the UseEye status the eye transformations are applied.

Warning: A 3D GrafPort **must** be open!

If the index you specified was illegal, TransformedPoint returns [0,0,0] and ErrorCode is set.

```
procedure CalcBounds;
```

This method calculates the bounding rectangle for the whole object as it would appear on the screen and places it in oldBounds. For this it calls scans through all transformed points. Therefore you must have called Transform (or Transform2) before calling CalcBounds.

```
function ForeignPoint (p: Vector4): Vector4;
override
```

GrafSys Documentation

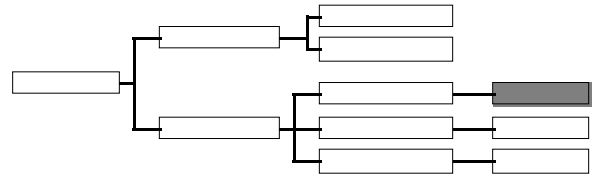
This is the same as the inherited procedure except that eye transformations are applied if the current 3D GrafPort's UseEye flag is set. Unlike the inherited procedure this method detects if it

GrafSys Documentation

has to call CalcTransform and will do so if necessary to initialize the screenXform matrix.

```
function WorldToModel(wc : Vector4) : Vector4;  
override
```

This is the same as the inherited procedure except that eye transformations are applied if the current 3D GrafPort's UseEye flag is set. Unlike the inherited procedure this method detects if it has to call CalcTransform and will do so if necessary to initialize the screenXform matrix.



Introduction

TSLine3D is a 3D line object that knows how to draw itself. In all other aspects it is like its ancestor TLine3D.

Heritage

Superclass TLine3D

Subclasses none

Using TSLine3D

Handling is exactly like TLine3D except that now drawing is supported.

Variables

no additional variables at this instance.

Methods

Inherited methods:

```
procedure HandleError;
procedure ResetError;
function Test (opcode: integer): integer;
function Clone: TGenericObject;
procedure Translate (dx, dy, dz: real);
procedure SetTranslation (x, y, z: real);
procedure Rotate (dx, dy, dz: real);
procedure SetRotation (x, y, z: real);
procedure Scale (dx, dy, dz: real);
procedure SetScale (x, y, z: real);
procedure RotArb (p, x: Vector4; phi: real);
procedure ResetArb;
procedure FFTranslate (dx, dy, dz: real);
procedure FFRotate (dx, dy, dz: real);
procedure FFScale (dx, dy, dz: real);
procedure FFRotArbAchs (p, x: Vector4;
    phi: real);
procedure FFReset;
function FFNewPostConcat: TMatrixList;
function FFNewPreConcat: TMatrixList;
function FFActivate (theFF: TMatrixList)
```

GrafSys Documentation

```
      : boolean;  
function FFPassOn: TMatrixPass;  
procedure FFIinherit (var FatherList:  
    TMatrixPass);
```

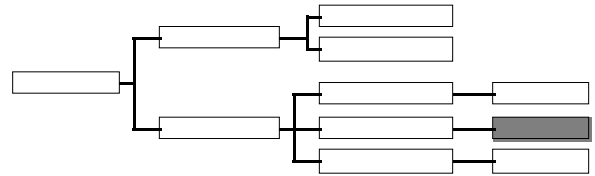

GrafSys Documentation

```
procedure CalcTransform;  
function ForeignPoint (p: Vector4): Vector4;  
function WorldToModel (wc : Vector4) : Vector4;  
procedure Kill;  
procedure Init;  
procedure Reset;  
procedure SetKoords (Koordinates: Vector4);  
function GetKoords: Vector4;
```

Other Methods:

```
procedure Draw;  
override;
```

This method will draw the line as seen under the current eye settings and with current projection. No clipping is supported.



Introduction

TSPoint3D is a 3D point object that knows how to draw itself. For this it has a size parameter that defines how large the point should be if drawn on the screen. In all other aspects it is like its ancestor TPoint3D.

Heritage

Superclass TPoint3D
Subclasses none

Using TSPoint3D

Handling is exactly like TLine3D except that now drawing and point size is supported. Point size defaults to 2, i.e. the point will be drawn as a rectangle with a size of 4*4 (the size of each side is *twice* the size parameter).

Variables

Name	Type	Description
Size	integer	Size of point when drawn. Used to create a rectangle whose sides are twice as long as the size parameter.

Methods

Inherited methods:

```

procedure HandleError;
procedure ResetError;
function Test (opcode: integer): integer;
function Clone: TGenericObject;
procedure Translate (dx, dy, dz: real);
procedure SetTranslation (x, y, z: real);
procedure Rotate (dx, dy, dz: real);
procedure SetRotation (x, y, z: real);
procedure Scale (dx, dy, dz: real);
procedure SetScale (x, y, z: real);
procedure RotArb (p, x: Vector4; phi: real);
procedure ResetArb;
procedure FFTranslate (dx, dy, dz: real);
procedure FFRotate (dx, dy, dz: real);
  
```

GrafSys Documentation

```
procedure FFScale (dx, dy, dz: real);  
procedure FFRotArbAchsis (p, x: Vector4;  
    phi: real);
```

GrafSys Documentation

```
procedure FFReset;  
function FFNewPostConcat: TMatrixList;  
function FFNewPreConcat: TMatrixList;  
function FFActivate (theFF: TMatrixList)  
    : boolean;  
function FFPassOn: TMatrixPass;  
procedure FFInherit (var FatherList:  
    TMatrixPass);  
procedure CalcTransform;  
function ForeignPoint (p: Vector4): Vector4;  
function WorldToModel (wc : Vector4) : Vector4;  
procedure Kill;  
procedure Init;  
procedure SetKoords (Koordinates: Vector4);  
function GetKoords: Vector4;
```

Other Methods:

```
procedure Init;  
override
```

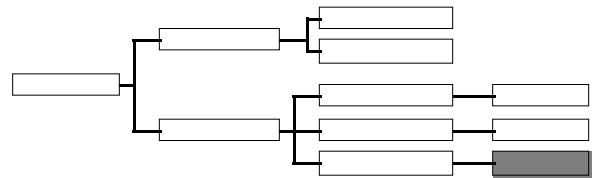
Initializes the object (by calling all inherited Init methods) and then sets size to 2.

```
procedure Reset;  
override;
```

Resets the object (by calling all inherited Reset methods) and then sets the size parameter to 2.

```
procedure Draw;  
override;
```

This method will draw the point as seen under the current eye settings and with current projection and size parameter. No clipping is supported.



Introduction

This is the main GrafSys 3D object. It supports up to 8000 lines, full RGB color (i.e. billions of colors) and clipping. Use this object when extending the GrafSys. Usually, however this object is enough to satisfy most needs.

Heritage

Superclass TObjectGenericObject3D
Subclass none

Using TObject3D

Basic handling is like TObjectGenericObject3D. Lines can be added in a way similar to the way you add points. Lines are always defined as connecting two points in the database (identified by their index). The order in which you defined the lines can have dramatic impact on the performance of GrafSys, so be sure to read Part I, 'Fundamentals' of this documentation.

the default Color for lines is assumed to be black (RGB 0,0,0). When you change a line's color to A, all subsequent lines will be drawn using A as well until you specify a new color.

Variables

Name	Type	Description
Lines	LineBufPtr	Buffer for line descriptions. Holds a maximum of 8000 lines.
numLines	integer	Number of lines currently in database
AutoErase	Boolean	If this flag is set the Draw and fDraw methods will erase the previously drawn image (actually it will erase the contents of the oldBounds rectangle.
UseBounds	Boolean	If this flag is set the fDraw and Draw procedures collect bounds rectangle information into the Bounds rectangle.

Methods

Inherited methods:

GrafSys Documentation

```
procedure HandleError;
```

GrafSys Documentation

```
procedure ResetError;
function Test (opcode: integer): integer;
procedure Translate (dx, dy, dz: real);
procedure SetTranslation (x, y, z: real);
procedure Rotate (dx, dy, dz: real);
procedure SetRotation (x, y, z: real);
procedure Scale (dx, dy, dz: real);
procedure SetScale (x, y, z: real);
procedure RotArb (p, x: Vector4; phi: real);
procedure ResetArb;
procedure FFTranslate (dx, dy, dz: real);
procedure FFRotate (dx, dy, dz: real);
procedure FFScale (dx, dy, dz: real);
procedure FFRotArbAchsis (p, x: Vector4;
    phi: real);
procedure FFReset;
function FFNewPostConcat: TMatrixList;
function FFNewPreConcat: TMatrixList;
function FFActivate (theFF: TMatrixList)
    : boolean;
function FFPassOn: TMatrixPass;
procedure FFInherit (var FatherList:
    TMatrixPass);
procedure CalcTransform;
function ForeignPoint (p: Vector4): Vector4;
function WorldToModel(wc : Vector4) : Vector4;
procedure SetKoords (Koordinates: Vector4);
function GetKoords: Vector4;
procedure GenIndex (pointIndex: longint;
    var BufIndex, bufOffset: integer);
function AddPoint (x, y, z: real): longint;
procedure GetPoint (index: longint;
    var x, y, z: real);
function ChangePoint (index: longint; x, y, z:
    real): boolean;
procedure Transform (forceCalc: boolean);
procedure Transform2 (forceCalc: boolean);
function TransformedPoint (index: longint)
    : Vector4;
procedure CalcBounds;
function ForeignPoint (p: Vector4): Vector4;
function WorldToModel(wc : Vector4) : Vector4;
```

GrafSys Documentation

Other Methods:

```
procedure Init;  
override;
```

Call this method immediately after allocating the object. It calls all inherited Init methods and allocates memory for the line buffer. numLines is set to zero, AutoErase and useBounds to FALSE.

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

Clone produces an exact copy of the object. It calls all inherited Clone methods (to clone allocated point buffers and handle inheritance cloning, see Clone for TabstractObject3D) and then clones the line buffer.

```
procedure Reset;  
override;
```

Reset calls all inherited Reset methods (thus resetting all operators) and then sets AutoErase and useBounds to FALSE.

```
procedure Kill;  
override;
```

Kill calls all inherited Kill methods (thus killing sons and owned operators and deallocates memory for point buffers) and releases the memory for the line buffer.

```
function AddLine (fIndex, tIndex: longint)  
: integer;
```

Use this function to add a line to the object's data base. fIndex and tIndex are the indices of the points as defined in the database. AddLine returns the index of the line so you can access it later.

Note that this method does not change any line color information. Use the ChangeLineColor method, below.

If AddLine returns a negative index the operation was unsuccessful and ErrorCode contains the reason.

GrafSys Documentation

```
function ChangeLine (LineIndex, fIndex, tIndex  
                    : longint): Boolean;
```

ChangeLine replaces the line description (from point index , to point index) of line LineIndex to the new description passed in fIndex and tIndex). ChangeLine returns TRUE if operation was successful, FALSE otherwise (ErrorCode set).

```
function ChangeLineColor (LineIndex: longint;  
                          theColor: RGBColor): Boolean;
```

ChangeLineColor changes the color the lines are being drawn with from this line on to theColor. RGBColor is the standard Mac OS color definition. The function returns TRUE if the operation was successful, FALSE otherwise. Default line color is black (RGB 0,0,0).

```
function GetLineColor (LineIndex: longint;  
                      var theColor: RGBColor;  
                      var ChangeHere: Boolean): Boolean;
```

GetLineColor returns the color the specified line will be drawn with. If ChangeHere is true, this is the first line drawn with theColor. If operation successful, the function returns TRUE, otherwise theColor returns black (RGB 0,0,0) and ErrorCode is set. The result of ChangeHere is not defined.

```
function KeepLineColor (LineIndex: longint)  
                      : Boolean;
```

Since the line color only changes at lines you specify and then keeps that color until changed again, you can use KeepLineColor on the line where you used ChangeLineColor to undo the changes. Note that this Method is only effective on lines that you actually used ChangeLineColor on.

Imagine we have four lines $\langle l_1, l_2, l_3, l_4 \rangle$ and used ChangeLineColor with Red on l_2 . Therefore, the colors for the lines will be l_1 (black), l_2 (red), l_3 (red) and l_4 (red). If you Use KeepLineColor on l_3 , nothing happens since the color changes at line l_2 and KeepLineColor does nothing. If on the other hand used KeepLineColor on l_2 (where the color changes from Black to Red) the new colors would be l_1 (black), l_2 (black), l_3 (black) and l_4 (black).

```
function DeleteLine (LineIndex: integer)
    : Boolean;
```

DeleteLine will remove a line description from the object's database. All lines with an index greater than LineIndex will be moved down one index. Note that if the line you delete contains color change information this will be lost. Imagine we have four lines $\langle l_1, l_2, l_3, l_4 \rangle$ and used ChangeLineColor with Red on l_2 . Therefore, the colors for the lines will be l_1 (black), l_2 (red), l_3 (red) and l_4 (red). If you deleted l_2 (which carries color change information), the new colors will be l_1 (black), l_2 (black), l_3 (black). Therefore you should use GetLineColor with the line you want to erase so you can save the change color information if you want to.

```
function DeletePoint (index: longint): Boolean;
override;
```

This method checks to see if the point that you want to delete is referenced by a line. If so, the point cannot be deleted and DeletePoint returns FALSE with ErrorCode set accordingly. Otherwise, it calls the inherited DeletePoint method.

```
procedure GetLine (lineIndex: integer;
    var src, tgt: longint);
```

This method returns the index of startpoint and endpoint of the line with index lineIndex. If the lineIndex is illegal, GetLine will return zero for both start and endpoint and ErrorCode will be set.

```
procedure BuildNewLines;
```

This method is for internal use only. It is called when line information is changed. Each line carries a flag that tells the drawing routines if a QuickDraw MoveTo is necessary. A call to BuildNewLine calculates that flag for all lines in the database.

```
procedure CollectLineData;
```

GrafSys Documentation

This method is for internal use only. It preprocesses all line information and puts the screen coordinates of all lines (clipped and projected according to the current 3D GrafPort) into a special buffer to be used with fast drawing procedures.

```
procedure SetAutoerase (TurnOn: Boolean);
```

Sets the object's AutoErase flag according to TurnOn. If turned on, it will call the CalcBounds method to collect data so the next call to Draw or fDraw can erase the last image.

```
procedure SetUseBounds (TurnOn: Boolean);
```

This method sets the object's useBounds variable according to TurnOn. When switched on, Draw and fDraw will collect bounding rectangle information for you to implement your own auto-erase procedures.

```
procedure Draw;  
override;
```

Draws the object onto the current active QD GrafPort using the current active 3D GrafPort's settings. If AutoErase set it will first the Rect specified in oldBounds, then move the contents of Bounds to oldBounds. Then the method calls transform (or Transform2) if necessary and plots all lines according to UseEye and clipping mode.

If useBounds or AutoErase, Draw will collect bounding rectangle information and store it in the Bounds field.

```
procedure fDraw;
```

This procedure is exactly like Draw, except that it pre-calculates all line information and then uses a tight loop to display all lines. Overall performance is a bit slower than the Draw method, but the actual drawing is a bit faster. Use Draw if you draw into off-screen bit/pixel maps and fDraw if you draw onto the screen.

```
procedure Erase;
```

GrafSys Documentation

This procedure calculates the current bounding rectangle and then erases the corresponding portion of the screen.

GrafSys Documentation

Caveats: Using SetPort vs. Set3DPort

As stated above, it is usually impossible to use two different eye settings with the same Window. However, if you take advantage of the subtle difference between the two procedures `SetPort` (QuickDraw) and `Set3DPort` (GrafSys) you can achieve the desired result of two or more different eye settings on one window. You may even draw objects onto non-3D windows (which is desirable if you wanted to draw to off-screen pixel maps or the printer).

But first, let us explore the differences between QuickDraw's `SetPort` and the `Set3DPort` routines. For all drawing QuickDraw uses a global variable called `'thePort'` (called `'currentPort'` throughout this documentation). It holds a pointer to the currently active drawing environment (called `GrafPort`) that QuickDraw draws into [InsideMac]. All QuickDraw drawing takes place inside these `GrafPorts`. A `GrafPort` is a data structure that holds a variety of information such as what kind of pen you are using, where the pen is located, what font is selected etc. Every time you switch windows you actually switch `GrafPorts`. No matter where, if you draw into a dialog, a window, a print record etc.; wherever you draw, you draw into a `GrafPort`.

The currently active `GrafPort` is stored inside `'thePort'`, the `currentPort` variable. When you call `SetPort` you actually change the contents of that variable. Thus QuickDraw always knows in which `GrafPort` to draw and what pen to use by looking into the `currentPort` variable.

GrafSys uses a similar approach. All information about the eye etc. is stored in a 3D `GrafPort` that is an extension to the `GrafPort` (actually it is an extension of the newer `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.

GrafSys Documentation

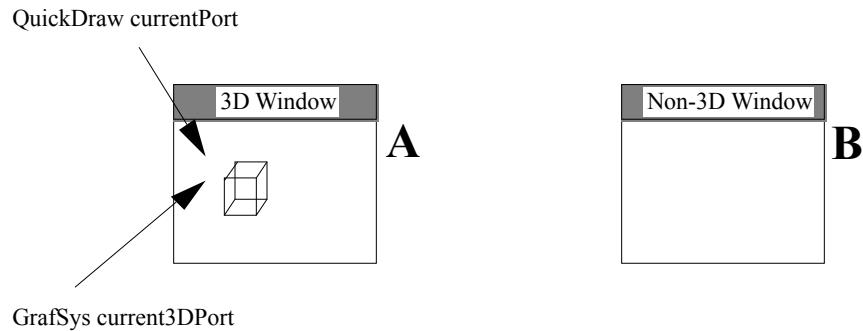


fig II.26: Normally QuickDraw points to window A. Drawing takes place in A. GrafSys points to A. A's eye settings will be used.

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.

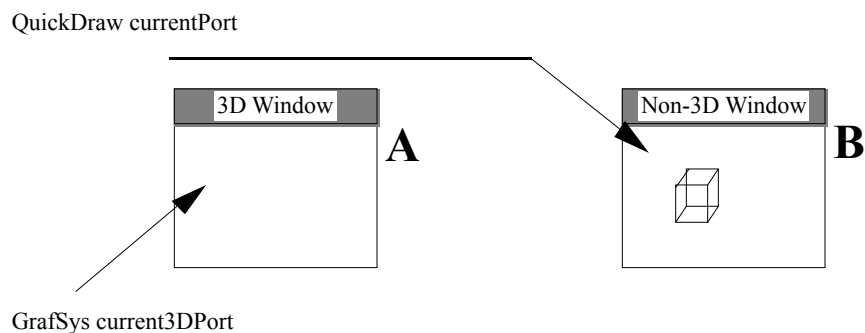


fig II.27: QuickDraw points to window B. Drawing takes place in B. GrafSys points to A. A's eye setting will be used but the image will appear in B.

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

GrafSys Documentation

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:

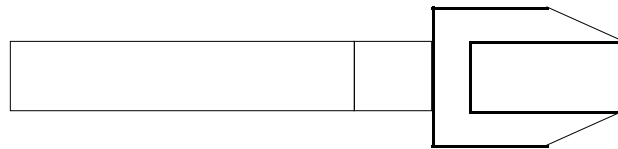


fig II.28: Robot arm

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.

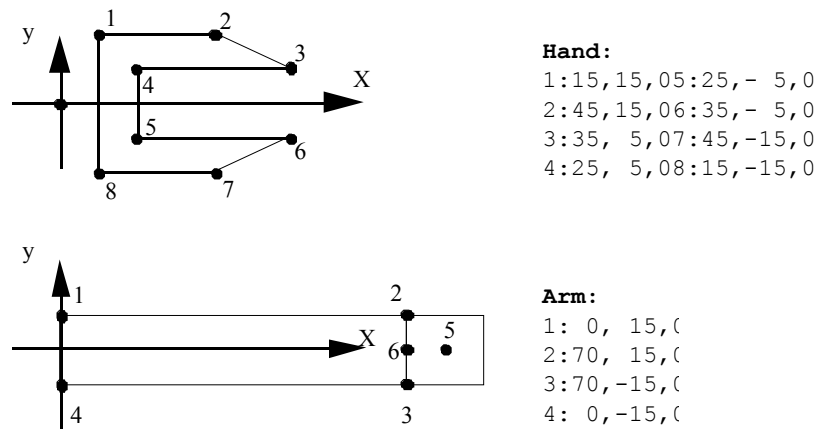


fig II.29: Database definition of hand (top) and arm (bottom).

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.

GrafSys Documentation

```
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, 0);
  OK := Obj.AddPoint(15, -15, 0);
  OK := Obj.AddLine(1, 2);
  OK := Obj.AddLine(2, 3);
  OK := Obj.AddLine(3, 4);
  OK := Obj.AddLine(4, 5);
  OK := Obj.AddLine(5, 6);
  OK := Obj.AddLine(6, 7);
  OK := Obj.AddLine(7, 8);
  OK := Obj.AddLine(8, 1);
end;
```

There is only one thing worth mentioning here: The definition of points 5 and 6 that are used to model the joint since the normal 3D GrafSys does not support circles. How we overcome this restriction is covered in the next chapter.

Modeling theRobot

The robot arm is modeled using the TSOBJECT3D as building-stones, the inheritance is the cement that glues the parts together. We use two TSOBJECT3D objects to model the two parts, Arm and Hand. They are enclosed into a new data structure 'robot'.

GrafSys Documentation

type

```
robot = object
  Arm: TSOBJECT3D;
  Hand: TSOBJECT3D;
  joint: TMatrixList; (* used to translate hand to end of arm *)
  jointRect: Rect;
  jointRadius: integer;
  procedure Init;
  procedure draw;
end;
```

To move the hand to the end of the arm, we use a FF operator 'joint' that has its own field in the data structure. To draw the joint we use a little trick. Since GrafSys does not support circles or spheres, we use the Mac toolbox `FrameOval` command that draws the outline of an oval inside a specified rectangle. The rectangle is simple to generate: the fifth point in the arm data base is the center for the joint. Since we move in 2D the radius of the joint will not change and we calculate it only once. Using the Mac toolbox `InsetRect` command with the radius we generate a square of the desired size around the joint's center coordinates and draw it.

Since the arm moves the hand, the hand needs to inherit the arm's movements. But when? The hand needs to be rotated independently before we apply the arms transformations. The correct sequence for all hand transformations are:

- Rotate the hand as desired (using default operators)
- Move the hand to the end of the arm (using 'joint' FF operator)
- Execute inherited transformations (using inherit FF operator)

Therefore, the hand needs two FF operators, one normal and one inherit operator. All these connections are done in the `robot.Init` method:

```
procedure robot.Init;
```

```
  var
```

```
    tmp: TMatrixPass;
    h1, v1, h, v: integer;
```

```
  begin
```

```
    New(Arm);
    Arm.Init;
    BuildArm(Arm);
    Arm.SetUseBounds(TRUE);
    New(Hand);
    Hand.Init;
    BuildHand(Hand);
    Hand.SetUseBounds(TRUE);
    joint := Hand.FFNewPostConcat;
    tmp := Arm.FFPassOn;
    Hand.FFInherit(tmp);
    joint.TMTranslate(85, 0, 0);
    ProjectPoint(Arm.transformedPoint(5), h, v); (* calc radius *)
    ProjectPoint(Arm.transformedPoint(6), h1, v1);
```

GrafSys Documentation

```
jointRadius := abs(h1 - h);  
end;
```

To draw the object, we simply call the two TSOBJECT3D Draw methods. Since we want to erase the past image ourselves we do not set the AutoErase flag. However, since we do need the Bounds information for erasing, we set the useBounds flag.

procedure robot.draw;

begin

```
(* erase old image *)
EraseRect(Arm.Bounds);
EraseRect(Hand.Bounds);
EraseRect(jointRect);
Arm.fDraw;
Hand.fDraw; (* now draw the joint *)
ProjectPoint(Arm.transformedPoint(5), jointRect.right, jointRect.bottom);
jointRect.left := jointRect.right;
jointRect.top := jointRect.bottom;
InsetRect(jointRect, -jointRadius, -jointRadius);
FrameOval(jointRect);
```

end;

Drawing the joint is easy. We get the coordinates for the center and create a rectangle that frames only this point. Then we expand it to all four sides for the radius and presto! have a square that exactly fits the joint circle.

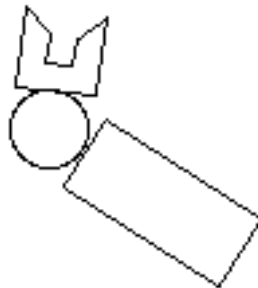


fig II.30: Result of the Draw message

Animating the robot

Animation now is quite simple:

- send arm and/or hand a rotate message with desired rotation
- send robot Draw message

repeat

```
theRobot.Arm.Rotate(0, 0, 2.5 * degrees);
theRobot.Hand.Rotate(0, 0, 5 * degrees);
theRobot.Draw;
delay(10, dummyLong);
dummyBool := GetNextEvent(everyevent, theEvent);
SystemTask;
```

until button;

GrafSys Documentation

The calls to `GetNextEvent` and `SystemTask` are only executed to remain compatible with running background applications and drivers.

Full Code Listing

program RobotArm;

{demonstrates inheritance of transformation on a 2D robot hand }

uses

Matrix, Transformations, GrafSysCore, GrafSysScreen, GrafSysObject,
Resources;

const

degrees = 0.01745329; (* $\pi/180$ *)
cTheWindow = 400;

type

robot = **object**

Arm: TSOBJECT3D;
Hand: TSOBJECT3D;
joint: TMatrixList; (* used to translate hand to end of arm *)
jointRect: Rect;
jointRadius: integer;
procedure Init;
procedure draw;

end;

var

theRobot: robot;
EyeLoc: Vector4;
theWindow: WindowPtr;

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;

GrafSys Documentation

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);
```

GrafSys Documentation

```
OK := Obj.AddPoint(35, -5, 0);
OK := Obj.AddPoint(45, -15, 0);
OK := Obj.AddPoint(15, -15, 0);
OK := Obj.AddLine(1, 2);
OK := Obj.AddLine(2, 3);
OK := Obj.AddLine(3, 4);
OK := Obj.AddLine(4, 5);
OK := Obj.AddLine(5, 6);
OK := Obj.AddLine(6, 7);
OK := Obj.AddLine(7, 8);
OK := Obj.AddLine(8, 1);
```

end;

procedure robot.Init;

var

```
tmp: TMatrixPass;
hl, vl, h, v: integer;
```

begin

```
New(Arm);
Arm.Init;
BuildArm(Arm);
Arm.SetUseBounds(TRUE);
New(Hand);
Hand.Init;
BuildHand(Hand);
Hand.SetUseBounds(TRUE);
joint := Hand.FFNewPostConcat;
tmp := Arm.FFPassOn;
Hand.FFInherit(tmp);
joint.TMTranslate(85, 0, 0);
ProjectPoint(Arm.transformedPoint(5), h, v);
ProjectPoint(Arm.transformedPoint(6), hl, vl);
jointRadius := abs(hl - h);
```

end;

procedure robot.draw;

begin

```
(* erase old image *)
EraseRect(Arm.Bounds);
EraseRect(Hand.Bounds);
EraseRect(jointRect);
Arm.fDraw;
Hand.fDraw; (* now draw the joint *)
ProjectPoint(Arm.transformedPoint(5), jointRect.right, jointRect.bottom);
jointRect.left := jointRect.right;
jointRect.top := jointRect.bottom;
InsetRect(jointRect, -jointRadius, -jointRadius);
FrameOval(jointRect);
```

end;

var

```
dummyLong: longint;
dummyBool: Boolean;
theEvent: EventRecord;
```

begin

```
InitGrafSys;
theWindow := GetNew3DWindow(cTheWindow, pointer(-1));
SetVector4(EyeLoc, 0, 0, -1);
SetEye(TRUE, EyeLoc, 0, 0, 0, 90 * degrees, none);
New(theRobot);
```


GrafSys Documentation

```
theRobot.Init;  
theRobot.Draw;
```

repeat

```
theRobot.Arm.Rotate(0, 0, 2.5 * degrees);  
theRobot.Hand.Rotate(0, 0, 5 * degrees);  
theRobot.Draw;  
delay(10, dummyLong);
```

GrafSys Documentation

```
dummyBool := GetNextEvent(everyevent, theEvent);  
SystemTask;  
until button;  
end.
```

Part III

Implementation of GrafSys

GrafSys Documentation

Overview

This part of the documentation is for the technically inclined reader and can easily be skipped. The first chapter 'Mathematics' contains detailed information about the mathematics involved and requires fundamental skills in linear algebra. The second chapter 'Implementation' discusses certain points of interest within the GrafSys implementation.

Mathematics

The underlying mathematics to GrafSys is fairly simple providing you have some understanding of linear algebra. All transformations are really just matrix manipulations (usually matrix-matrix or vector-matrix multiplication). For a quick rehearsal of linear algebra I suggest you read 'Linear Algebra' published by the McGraw-Hill Book Company ('Edition Schaum' in German) which is aimed at the high-school level reader.

Mathematics of Transformations

Before we can discuss the mathematics involved in transforming the points in the object database we first have to remove some mathematically unaesthetic obstacles. After that I will explain 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

GrafSys Documentation

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

$$[x', y', z', c'] = [\xi \ \psi \ \zeta \ \chi] \begin{bmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix}$$

The operator always has the following general appearance:

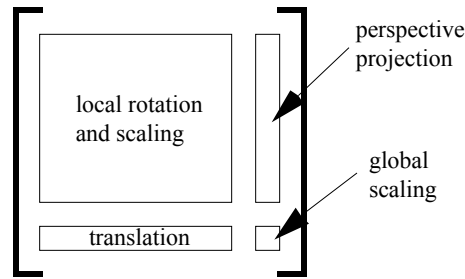


fig III.1: General structure of transformation matrix

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 $[dx, dy, dz]$:

$$[x', y', z', c'] = [\xi \ \psi \ \zeta \ \chi] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \delta\xi & \delta\psi & \delta\zeta & 1 \end{bmatrix}$$

GrafSys Documentation

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.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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 vector

$$\bar{v} = \begin{bmatrix} \xi_2 - \xi_1 \\ \eta_2 - \eta_1 \\ \zeta_2 - \zeta_1 \end{bmatrix} \text{ and } \bar{u} = \frac{\bar{w}}{\|\bar{w}\|} = \begin{bmatrix} \alpha \\ \beta \\ \chi \end{bmatrix}$$

- The order of the points you specify defines the positive direction or rotation. Then you specify the angle θ to rotate.
- To rotate around this axis, we have to go through five distinct steps:

1. Translate the rotation axis that it passes through the world's origin. This can be easily done by using the first point as translation vector:

$$T = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ -\xi_1 & -\eta_1 & -\zeta_1 & 1 \end{bmatrix}$$

2. Rotate the axis of rotation that it falls onto one of the main coordinate axis. We will always let the arbitrary axis fall onto the Z-axis. To accomplish this, we first rotate \bar{u} about the X-axis until it falls into the XZ plane and then rotate \bar{u} about the Y-axis until it coincides with the Z-axis:

For the first part (rotation about X-axis) we will not calculate the angle α but calculate directly the cosine and sine because we have all we need in \bar{u} already [Hearn86]:

$$\cos \alpha = \frac{\chi}{\delta} \text{ and } \sin \alpha = \frac{\beta}{\delta} \text{ with } d = \sqrt{\beta^2 + \chi^2}$$

Therefore, using the definition for rotation about X-axis, we get

$$R_x(\alpha) = \begin{bmatrix} 1 & & & \\ & \frac{\chi}{\delta} & \frac{\beta}{\delta} & \\ & -\frac{\beta}{\delta} & \frac{\chi}{\delta} & \\ & & & 1 \end{bmatrix}$$

This rotation is then followed up by the rotation about the Y-axis for β radians. Again we do not have to actually calculate β since $\cos \beta = \delta$ and $\sin \beta = -\alpha$. Thus, the rotation matrix looks like this:

$$Ry(\beta) = \begin{bmatrix} \delta & & \alpha \\ & 1 & \\ -\alpha & & \delta \\ & & & 1 \end{bmatrix}$$

3. Rotate for θ about the Z-axis
4. Inverse-rotate to restore original rotation axis orientation
5. Inverse-translate to bring axis back to where it belong

The whole transformation can be expressed as series of matrix multiplications:

$$RA(\theta) = T \cdot Rx(\alpha) \cdot Ry(\beta) \cdot Rz(\theta) \cdot Ry^{-1}(\beta) \cdot Rx^{-1}(\alpha) \cdot T^{-1}$$

As mentioned above, the inverse rotation matrix can easily be generated by calculating the transpose of the original rotation matrix (that way we never have to calculate α nor β)

Scaling

To scale along the three major axes we use the matrix

$$S = \begin{bmatrix} \sigma_x & & & \\ & \sigma_y & & \\ & & \sigma_z & \\ & & & 1 \end{bmatrix} \text{ where } \sigma_x, \sigma_y \text{ and } \sigma_z \text{ are the scaling factors.}$$

Note that using different values for scaling in X, Y and Z direction will destroy the orthogonality of the coordinate system [Brügg91].

Note also that scaling an object with transformation (i.e. it is removed from the origin) moves it further away from the origin.

Although GrafSys does not support arbitrary scaling (i.e. scaling with respect to a fixed point) it can be readily simulated by performing the following operations:

GrafSys Documentation

If we want to scale for $S=(sx, sy, sz)$ with respect to $P=(x, y, z)$ we would generate a matrix $SA(S, P)$ with the following attributes:

$$SA(S, P) = T(-x, -y, -z) \cdot S(sx, sy, sz) \cdot T(x, y, z)$$

or

$$SA(S, P) = \begin{bmatrix} \sigma_x & & & \\ & \sigma_y & & \\ & & \sigma_z & \\ 1 - \sigma_x & 1 - \sigma_y & 1 - \sigma_z & 1 \end{bmatrix}$$

GrafSys Documentation

Viewing in 3D

Mathematics of Projections

Projecting an image to the screen is pretty simple. As mentioned in part I there are two principal projection methods: parallel and perspective projection. Projecting a 3D point onto the screen takes two steps. The first is transforming it according to the transformation matrix (i.e. all rotations, translations etc. you performed upon it including the eye transformation). Let us call this matrix x_F , all operators (matrices) you defined OP_i . If you have n operators defined, we get

$$x_F = OP_1 \cdot OP_2 \cdot \dots \cdot OP_n [\cdot OP_{Eye}]$$

After we have the finale transformation matrix x_F we transform all Points P_i that are in an object's data base.

$$P_i' = (x', y', z') = P_i \cdot x_F$$

Now that we have the coordinates as seen from the eye we can project them onto the screen. In parallel projection we simply forget the z coordinate and use x' and y' to plot the point.

Since the origin of the Macintosh window unfortunately lies in the upper left corner [InsideMac], GrafSys uses an internal point structure to relocate the center to (xc, yc) .

$$\begin{aligned} h &:= xc + x & (xc \text{ is center horizontally of screen}) \\ v &:= yc + y & (yc \text{ is center vertically of screen}) \end{aligned}$$

If we use perspective projection we use the depth information in z' (that indicates how far removed the point from the projection plane is) to generate a perspective view. The further away from the projection plane, the closer the point moves to the vanishing point which is (in this case) the center of the screen.

Let d = distance of eye from projection plane, $d > 0$. Thus,

$$xp = \xi \left(\frac{\delta}{\delta + \xi} \right) = \xi \left(\frac{1}{\xi/d + 1} \right)$$

$$yp = \psi \left(\frac{\delta}{\delta + \xi} \right) = \psi \left(\frac{1}{\xi/d + 1} \right)$$

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

GrafSys Documentation

$$\begin{aligned}h &:= xc + xp \\v &:= yc + yp\end{aligned}$$

In perspective projection you should never draw points that fall behind the projection plane (i.e. whose z coordinate is negative). If you do this you risk that the term z/d becomes -1 and the projection becomes singular (i.e. your program is likely to crash). GrafSys will usually ignore a division by zero and just return nightmarish results that will likely clutter your screen and return you bounding rectangles from hell. But it will not bomb on you.

Clipping

Clipping lines in an object is obviously very important. Depending on the desired performance you can choose from three predefined techniques:

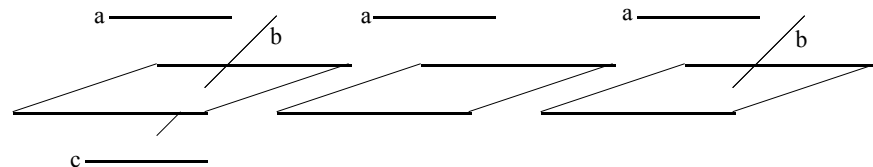


fig III.2: Clipping techniques. None (left), fast (middle) and arithmetic (right).
Line a is entirely above, b intersects and c is entirely behind the projection plane.

- *None*: This is the fastest. No clipping whatsoever is done and you must make sure that no line ever crosses or falls behind the projection plane.
- *Fast*: This technique is still very fast but handles lines that at least partially fall off the screen. Handling is very simple: the line is not drawn.
- *Arithmetic*: This technique is the slowest but returns the best results. Lines that are entirely in front of the projection plane are drawn immediately, lines that completely fall behind the projection plane are removed. Those lines that intersect (i.e. 'go through') will be clipped to the point where they penetrate the projection plane. The mathematics involved is pretty simple and straightforward. Having both startpoint S and endpoint E we generate the line description in the standard parametric description:

$$g = \begin{bmatrix} \sigma_x^S \\ \sigma_y^S \\ \sigma_z^S \end{bmatrix} + \tau \star \begin{bmatrix} \varepsilon_x^E - \sigma_x^S \\ \varepsilon_y^E - \sigma_y^S \\ \varepsilon_z^E - \sigma_z^S \end{bmatrix}$$

GrafSys Documentation

We now look for the intersection of g with the plane whose equation is

$$z=0$$

(Yes, this is one of the advantages of using the XY-plane as projection plane). This is of course very simple to solve since the problem degenerates to the one-variable equation

$$0 = \sigma_z + t * (\varepsilon_z - \sigma_z)$$

which immediately leads to

$$t = -\frac{\sigma_z}{(\varepsilon_z - \sigma_z)}$$

Since this parameter t can directly be calculated and applied, calculation is still fairly fast. We achieve this by always forcing the endpoint to be illegal (i.e. behind the plane) and the startpoint legal. Now we always calculate the new coordinates for the endpoint and the formula is readily given as

$$\begin{aligned} ex' &= \sigma_x - \frac{\sigma_z}{(\varepsilon_z - \sigma_z)} * (\varepsilon_x - \sigma_x) \\ \varepsilon x' &= \sigma_x - \frac{\sigma_z}{(\varepsilon_z - \sigma_z)} * (\varepsilon_x - \sigma_x) \\ \varepsilon_z &= 0 \end{aligned}$$

The question remains if this algorithm is robust. The obvious case where it breaks is when g is parallel to the projection plane and therefore never penetrates the plane. In this case ez equals sz and therefore $(ez-sz)$ becomes zero and t singular. However, this can never happen since we know that if the line is parallel to $Z=0$ it is either completely visible (and drawn) or completely invisible (removed). In both cases the clipping algorithm is skipped.

The only thing to remember before drawing is that the clipped line must be (perspective) projected and included into the boundary check. This is automatically done in the GrafSys.

GrafSys Documentation

Implementation

In this chapter I will discuss some interesting problems I encountered writing GrafSys and how I solved or (in some cases) circumvented them. Again, this is only for the technically inclined reader and can easily be skipped. Central to almost all problems discussed is the need for speed and OS compatibility.

Compatibility and System Requirements

I have taken great care to make the use of the GrafSys transparent to the user and Macintosh OS as well. This was very difficult in many cases since I had to make some assumptions as to what machines would be used with the GrafSys and which will not. Since one of the main goals was speed, setting hundreds of flags as to which functions are implemented and which not was simply out of question. I had to draw the line somewhere and still maintain compatibility across the majority of the Macintosh product line.

For example take the color implementation. The Macintosh Plus does not know about color if it is running System 4.2 or below. If, on the other hand you ran your Plus with System 6.x and 32-bit QuickDraw (default with System 7) it would know about it. The same goes for GWorlds [InsideMac, Brig92] or other System X.Y dependent routines.

Then there was the problem of manipulating Pixel- and/or BitMaps. The problem is not immediately apparent until you run into a 32-Bit 'dirty' ROM Macintosh. The Memory Manager in older Macs used only 24 of the 32 available address bits for its addresses and of course some 'clever' programmer used the remaining bits for housekeeping. This is similar to the problem many companies face nowadays that the turn of the millennium is near and many programmers cleverly allocated too few bits to accommodate numbers larger than 99.

On the Mac you run into major problems if you try to reference a 'dirty' address since one to eight of the highest bits can be set wrongly. A routine `StripAddress` is provided just to fix this problem [InsideMac], but only if you are in 24 bit mode. Of course the `StripAddress` function is only implemented in the new ROMs or comes with System 5.0 or above.

The minimum requirements for running the Fixed-Point GrafSys is therefore a Mac Plus with System 6.x (or above) and Color QD installed. The minimum requirement for running FPU GrafSys is a Mac with FPU or 68040 Processor and System 6.x (or above) with Color QD installed. The Fixed-Point version gave me some headache as you will see below.

GrafSys Documentation

The temptation was great to cut the line a bit higher and make QuickDraw GX a requirement. This would have made the GrafSys a whole lot smaller and probably even faster, but firstly GX was still beta when I wrote the GrafSys and secondly I thought it a bit overkill to force you to install Display Postscript just to have some 3D graphics on your machine.

Fixed vs. Floating-Point Math

Why did I have to do this? The answer is quite simple: It was one of the requirements that GrafSys runs with GeoBench and this program comes in two versions: FPU and non-FPU. This was (and still is) a major hassle because it imposes a data abstraction between core routines and other GrafSys procedures. As a result, it took a big performance hit. The other obvious solution to compile two different versions specifically written for each version was unacceptable to me since then the two versions weren't code compatible anymore and you had to change your code according to which version you are using.

In the end I decided to settle for a mix of both options. I did impose a data abstraction between inner core (matrix manipulation) and outer core (lowest objects) and was therefore free to code different versions of the inner core, each curtailed to the current GrafSys version.

Still the speed degradation is noticeable (about 5% performance drop to a straight-coded version with no data abstraction).

Fixed-Point Arithmetic

The fixed-point arithmetic routines are a standard Macintosh data type [InsideMac] and allow fast manipulation of fractions. Since they are based on the integer data type, addition and subtraction are handled the same way as integers. For multiplication and division the Macintosh Toolbox provides quick routines. On the whole, the outlook was good, especially since it has the same range as the integer data type but provides 16-Bit accuracy (i.e. a resolution of 1/65535).

The Lookup-Table approach

To further boost performance of the fixed-point GrafSys I tried to implement fast look-up tables for sine and cosine functions. Using tables in resource form one could use different accuracies and simple procedure calls to immediately return the sine, cosine and tangents results in fixed-point form. Performance test showed good results on machines with no FPU installed. On FPU-equipped Macs, however, even look-up tables were slower than using SANE (Standard Apple Numeric Environment) calls (see Appendix B: 'Look-Up Table Performance Tests').

GrafSys Documentation

After using a profiler I finally found out that the time used to calculate sines, cosines or tangents is unnoticeable compared to the time spent looking up points, transforming and projecting them. Using a faster but lower-precision look-up table does not gain anything noticeable. This approach was not successful.

The Bottom Line

While look-up tables for trigonometric functions seemed to make sense at first, they did not really improve performance since 3D transformation does not make heavy use of trigonometric functions. On the other hand the usage of fixed-point arithmetic is imperative when using non-FPU equipped Macintosh computers since their floating-point handling is pitiful at best. Using fixed-point arithmetic increases performance by a factor of 100 and more by sacrificing some accuracy and range of value.

If GrafSys wants to be compatible with both versions of GeoBench it has to come in two versions.

The Memory Allocation Problem

Another prime requirement was that GrafSys should be able to handle huge amounts of points. This again stems from the fact that GrafSys must work with GeoBench which can model geographic data (and believe me, they use gargantuan databases). Again I had to cut the line somewhere and after some talk with Adrian Brünnger I drew the line at currently 256'000 points per object. At 256'000 point each object needs 7.7 MB of storage.

The first problem with this is that THINK Pascal has problems addressing records larger than 32K (obviously by inefficiently using the 16-bit indirect offset addressing mode [Kelly88]). Then, relating directly to this is that if we have an object of this size, extending it will become a major problem as well, because THINK Pascal uses 16-bit indirect addressing for accessing instance variables [Sphar91].

If we somehow solve this problem, allocating a single object will still require more memory than a Macintosh Plus can address. This is called 'inefficient memory management' or sometimes simply 'stupid'.

Therefore, I used another approach that allocates memory as the database grows. To keep up with performance, however, the memory for points is allocated in chunks of 32K blocks, which is enough for 1024 points (which also neatly circumvents the 32K index problem). The current implementation uses a block of 256 pointers that point to the various point buffers. This allows for reasonably fast random access to the points by minimizing memory fragmentation:

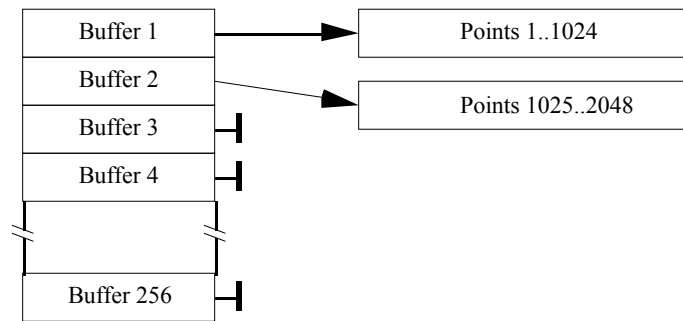


fig III.3: Buffer allocation of an object that has between 1025 and 2048 points in it's data base. Only two point buffers have been allocated

Now, the minimum size for an object would be about 50K and grows another 32K every 1024 points you add. An object with 256'000 points entered now weighs in at 8.3 MB which yields an internal fragmentation of 0.6 MB or 7%. The worst fragmentation per allocated Block would be 32K of memory compared to 7.6 MB if we allocated monolithic storage for the database.

This method of memory allocation allows average access times for better performance but has the problem of limiting the size of the database to 256'000 points. I thought this a reasonable size since if you really needed to access a bigger data bases, you can easily create new objects that consist of multiple supplied objects and just extend the `AddPoint` methods (you do not have to modify the `Draw` methods if you use inheritance) to suit your needs.

Since this method of memory handling is similar to the way operating systems allocate disk storage [Tanen87], one immediately thinks about double- or even triple indexing to gain more address space. I thought about implementing this and although it might be thrilling to be able to access more than 25 billion points, but it would seriously degrade performance and will not improve much besides the ability to access more points than you would ever need (the memory requirements would of course be 30 times the number of points accessed since a point takes up 30 bytes).

Another way to increase the number of points in the database would be to double the number of buffer pointers. This is feasible and will not hurt performance at all. To do this, all you have to do is change the `MaxBuffers` constant and recompile the library.

The same method, albeit much simpler was used for buffering of line information. In its current version, a `TObject3D` supports only 8000 lines that are stored in a buffer that is allocated immediately when the object receives an `Init` message. To extend this, you should adopt the same technique as described above.

Inheritance

The implementation of inheritance is quite simple. Still it took me three attempts to get its current form. The key to success was subclassing the TMatrixList for two link pairs TMatrixPass and TMatrixInherit. The former is used to automatically pass on the current xForm operator whenever the object gets recalculated. The latter (TMatrixInherit) is used to receive the parent's current xForm operator (i.e. all transformations until now) and behaves much like any other operator.

Now, why do I need a pair of operators where one would surely suffice? In it's first incarnation I only used a single operator that held a pointer to the operator that would inherit. This was possible in a way but required the calculation routines of both objects to check whether a link existed or the father changed. The real problem became apparent when I tried to inherit the same operator to two objects (like it happens when cloning the son). Another problem was what I should do when I killed either son or father. Finally I decided on introducing the PassOn object that was neutral to the xForm operator and just stored it in the son's Inherit operator. This way, cloning the son was easy, since all that was needed was to insert another PassOn operator in the fathers FF chain.

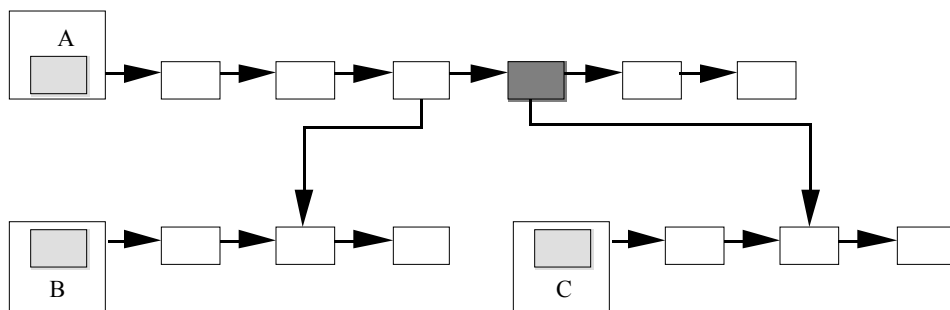


fig III.4: Cloning a son (C is clone of B). A new PassOn operator was installed in father object

But what should be done when cloning the father?

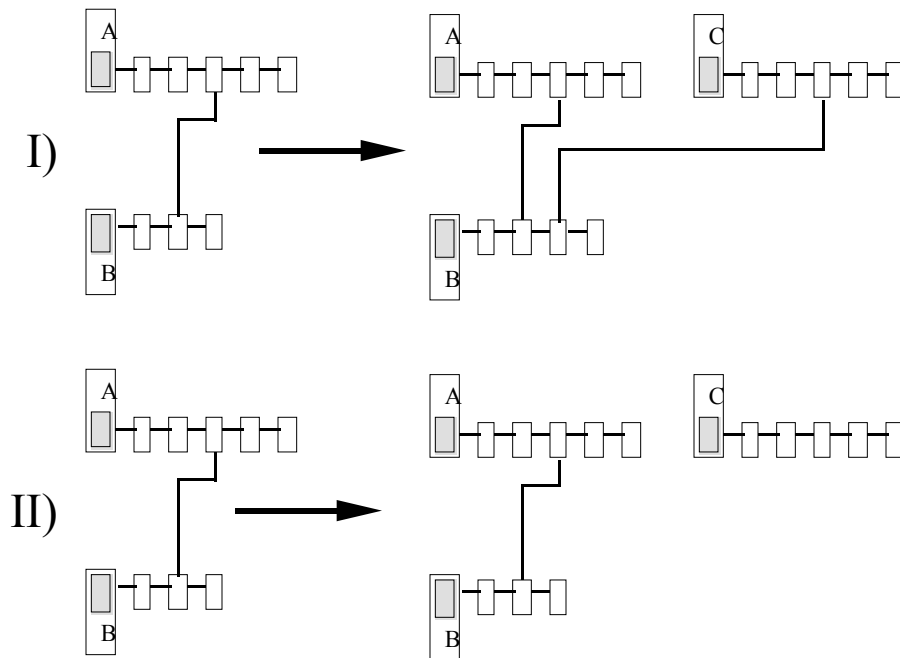


fig III.5: Two possible ways to clone a father. The top version also clones the PassOn operator, the bottom removes the inheritance link between clone and son.

After some thought I found out that cloning the link does not make much sense at all and decided that the clone will not pass on (another possibility would be to clone the sons as well but then the programmer would have no possibility to recover the handles to the newly allocated objects and therefore this would not make that much sense either).

This has the disadvantage that clones are not really clones but just 'twins' as I would like to call them. They look the same but have different ways of acting. Also, cloning the son (which inserts a new link in the father) should have likewise results when cloning the father. However, I decided that common sense should override logic principle here.

Another problem surfaced when I implemented the `Kill` methods. What should be done when an object passes on transformations? Again, I opted for a pragmatic approach: Killing a father means that their sons (that move relative to the father anyway) will be killed and their sons as well. If you want to change this you must first break the inheritance link (by removing all PassOn-Inherit pairs) and then kill the object. Only removing the PassOn operator will result in a crash since passing a `calcTransform` message to the son will cause it to access the fathers PassOn link.

When killing a son, things are much simpler. The PassOn-Inherit pair is not removed, only the reference to the owner. When the father is killed, the pair is automatically removed.

The calcTransform Process In Detail

CalcTransform is the object's main transformation evaluation routine. It evaluates all operators, owned and inherited, and builds the master transformation operator `xForm` that is used when the object's points are transformed. `xForm` is first initialized to Identity. Then all operators for this object are applied to `xForm` consecutively in the following order:

- Default standard operator in the following Order: X-rotation, Y-rotation, Z-rotation, translation, scaling
- Default arbitrary operator as it currently is
- All FF operators in the sequence they are put in the chain

Default Standard Operators

To optimize speed, there is no real default standard operator. Rather, the object saves current rotation, scale and position (i.e. translation) in separate variables. Then, when beginning the calcTransform process, `xForm` is initialized from these values. This saves calls to matrix-matrix multiplications, since all transformations are independent from each other (rotation and translation are never mixed). Since this is not true for any other operator this trick only applies to the default standard operator.

Default Arbitrary

After setting up `xForm` using the default standard operators, the default arbitrary operator (the `arbRot` field) is multiplied to `xForm`. This operator is used whenever the object receives a `RotArb` call.

FF Chain And Inheritance

Finally, all free-order operators are multiplied to `xForm` in the order they are encountered in the chain. Whenever a `PassOn` operator is encountered, the current values of `xForm` is copied to the `Inherit` operator in the son's FF chain. Also, the son's '`objChanged`' flag is set to signal the `Draw` procedures that it has to recalculate the `xForm` operator. If an `Inherit` operator is encountered, the contents is simply multiplied to `xForm`.

After the last FF operator has been multiplied, `xForm` contains all transformations that have been applied to the object except for eye transformations.

Speed Tricks and Trade-Offs

To maximize performance, I have used a lot of tricks. In other places, I had to use slower techniques to meet design goals. Still, there is no

GrafSys Documentation

'hack' involved that could crash the program in later versions of the system software. The most notable 'Tricks and Trade-Offs' are:

Tricks:

Look-Up Table (discontinued)

To minimize calculation time for sine, cosine and tangents, I introduced a look-up table for the fixed-point GrafSys version. However, results showed a neglectible gain (less than one millisecond every second) on non-FPU machines and losses on FPU-equipped Macs.

Transpose for inverse Rotation

In the arbitrary rotation routines, I used the transpose of a rotational matrix instead of calculating the inverse matrix or using the inverse angle (which would also cost a call to the ArcTan function).

Fast Matrix-Matrix And Vector-Matrix Multiplication

The matrix-matrix and vector-matrix multiplications are optimally coded. There is no index calculation involved, every single multiplication is coded, no index variable ever set. This way the compiler generates only offsets instead of register-relative [Kelly88] with offsets and the offset-calculation overhead is removed [ThPasRef, Moess91] and constants are used instead.

Identity Flag

To further speed matrix-matrix and vector-matrix multiplication, each matrix carries an Identity flag. If set, the matrix is mathematically neutral and multiplication can be skipped.

NewLine-Flag

When drawing an object to a GrafPort, the drawing routines check if the line's NewLine flag is set. If true, the routine must evaluate the new starting point and move the pen to the starting location. If false, the line continues from the position we left off. This only needs to be calculated once for each object and can improve performance, since calls to ToolBox procedures such as MoveTo require a lot of time [Mark89, Mark90, MBugRef].

Default Standard Operator Hoax

When using the default standard operator, no real matrix-manipulation is used but direct variables are set. This way I could avoid calls to the matrix-multiplication routines

GrafSys Documentation

Trade-Offs

OOP run-time binding

GrafSys takes the second worst performance hit from a design specification. Since it should be an object-oriented, flexible and

GrafSys Documentation

full fledged library that can be curtailed to anyone's need, it uses run-time binding. This is slower than a static library with a fixed set of functions [Sphar91]. To minimize the impact of run-time binding, I have placed the most important routines as low as possible while still retaining the semblance of a hierarchy.

Buffering for points and lines

The requirements to accommodate hundreds of thousands of points forced me to implement a low level memory management. This will of course slow the manipulation of objects considerably. However, to minimize this, I have incorporated only simple indirection and used powers of two as elements per buffer. Still, this is where GrafSys loses most on complex objects.

FPU and Fixed-Math

Forcing me to implement two versions of the GrafSys also takes its toll. Because now the internal data definition is opaque, the user must go through specially provided routines to manipulate the point's data inside the database. This cuts performance.

The Piggy-Back Technique

The single most important trick used implementing the GrafSys was emulating type inheritance for the Macintosh OS window data type. Since a 3D window should be transparent to both user and system I needed an efficient way of attaching the vital 3D data structures to the normal window data. That way the user could treat a 3D window just like any other window (important for event processing, drawing etc.) and GrafSys would be able to retrieve data such as eye setting and projection. The same was necessary for the off-screen package.

The solution was very simple once thought of. The Mac uses a memory manager that keeps track of the size of blocks. When you use the toolbox `GetNewWindow` trap, it first asks the memory manager to allocate space for the window data structure and then fills in the fields according to the window appearance. Luckily the designers of the toolbox made it an option to provide the storage yourself. This feature was intended for programmers who wanted to minimize memory waste and did their own memory management (which was important on early Macs) or for people who did not use resources.

Although originally not designed for it, the possibility to provide your own storage space for the window data is the key to the piggy-back technique. If we allocate a block of memory that was n Bytes larger than the windows data structure and passed this to the `GetNewWindow` command, we have the last n bytes for ourselves that can be used to store anything.

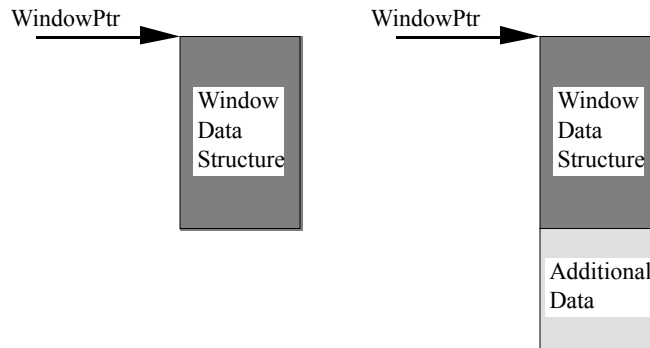


fig III.6: Normal GrafPort data allocation (left) and with space for addition data (right)

And this is exactly the way I implemented the storage for the 3D data. The first part of the 3D GrafPort is exactly as in a CWindowRecord, the remainder is private GrafSys data.

The fortunate side effect is that all Mac toolbox routines accept this enlarged data block, and only modify the first standard fields. Still, if you pass a 3D GrafPort to the standard DisposWindow procedure, the whole block gets deallocated, not just the normal window data structure.

Resource Access

Since it is very cumbersome and bad practice to hard-code object data descriptions (points, lines, polygons and line colors) into a program and the Mac supports resource files, GrafSys should be able to store and retrieve object databases to and from resources.

The biggest problem writing these resources was the Pascal language definition, not the Mac toolbox. Looking at the resource definition it immediately becomes obvious that the resource is variable-sized. Unfortunately, Pascal does not allow dynamic arrays. The solution to this problem is straightforward and comes close to C programming (i.e. better left untold).

I declared a HackType as a packed array [0..0] of Byte. When accessing the resources I simply turned off range checking and accessed 'illegal' indexes. When saving a resource I did the same. Before saving the resource, however I had to pre-calculate the size of the resource as it would be after creation and then allocate a block of memory exactly the same size. After storing the information the rest was very simple since the Resource Manager works closely together with the Memory Manager and all I had to do was declaring the block of memory as a Resource Handle and the Resource Manager did the rest.

GrafSys Documentation

Off-Screen Pixel Maps

Off-Screen Pixel Maps were one of the most difficult aspect of the whole GrafSys (in conjunction with the high-speed triangle filling). Although the current implementation of the GrafSys looks as if using off-screen pixel maps is a snap, it is actually not. This is because the Mac supports multiple monitors, multiple color models and, even worse, multiple pixel types (indexed and direct).

The first approach I used was that I simply allocated a chunk of memory of the desired size and tried to substitute it for the normal pixel map ('PixMap') that the window used. The idea was that now QuickDraw unwittingly drew into the now-off-screen pixel map. The problem with this approach was two-fold:

- Under normal circumstances QuickDraw would use the substituted PixMap as the normal pixel map. However, since I used the same GrafPort, it also used the same clip region. So, if the window was half obscured by another window, QuickDraw would not draw into the part that was obscured.
- When trying to copy to the screen the exact inverse problem occurred. I could not use `CopyBits` since it requires a second PixMap structure that I hadn't allocated. But when I wrote my own copy routines they would ignore the port's clip regions and overwrite other windows that obscured parts of the original port.

The next step was that I created a whole off-screen port and simply set the current QuickDraw port to this whenever I wanted to draw into off-screen PixMaps. This was of course very elegant and of course it did not work either, even though it was documented like this in *Inside Macintosh Vol I*. Further research showed that this method did indeed work, but not on the newer Macs that use color. This is because the newer Macs (actually all Macs that use Color QD) use the probably worst documented data structure called GDevice for internal housekeeping. The GDevice data structure stores one vital piece of information that is not accessible through the GrafPort: the inverse table for color information [TN#163, TN#193, *InsideMac V*].

After some experimentation I found out that every off-screen PixMap should have an off-screen GDevice to describe how it uses color and pixels [TN#079, TN#120]. GDevices are usually used to keep track of the different video cards installed into the Macintosh and are handled by the system [*InsideMac V*]. Needless to say, if you allocate an off-screen PixMap you have to construct a GDevice yourself and keep track of it yourself. Otherwise the `CopyBits` procedure will do anything imaginable (and then some) except what you expect [TNTS_C].

GrafSys Documentation

The third step was to think 'to hell with it all' and use GWorlds, the new System 7 extension to 32-bit QD that allegedly made off-screen PixMaps simple to use. However, I decided against this because of the requirement to run under System 6.x with Color QD.

The final step was prompted when reading [TN#120, TNOSB] and [TNPOSE]. The solution was to create both a standard off-screen GDevice and CGrafPort for every off-screen PixMap. That way, I was able to use the off-screen PixMap as any other port, translate any color and use CopyBits for maximum compatibility. The 3D GrafPort data type has a dedicated structure to hold this information. Then I decided to define all off-screen PixMaps as 8 bit deep (256 Colors) and wrote routines to make manipulation of the off-screen PixMaps almost transparent to the programmer.

The 8 bit depth was chosen for maximum performance of the triangle fill algorithms that had to be optimized for a fixed bit-depth (see Triangle-Fill below).

The idea to think of the off-screen PixMap as a shadow for drawing and support only this mode with fixed size, depth and congruent coordinate systems proved to be very successful and led to the only six (!) routines required to create, maintain and dispose of the off-screen PixMaps and maintaining full compatibility with all Macintosh models.

Triangles

Although the GrafSys was designed for speed, it only provided it for transformation and some drawing. However, when trying to use it for hidden-line or hidden-surface animations, the programmer was left to his/her own devices. I looked for a general and efficient method to allow a programmer to implement hidden-line and hidden-surface animations.

The Problem with Polygons

To implement surfaces, the programmer usually groups lines to a polygon, sorts them by ascending z values and calls the Macintosh Toolbox `PaintPoly` procedure to paint them one over the other to create the illusion of hidden-surfaces. This is basically the so-called painter's-algorithm (a.k.a. depth-sort) [Foley90, Hearn86] and the simplest method to implement hidden-surface removal. Although the painter's algorithm usually fails for surfaces that alternately obscure each other, I chose it because

- it is simple and fast

GrafSys Documentation

- the data base in each object is static (with regard to animation, not the program itself) and alternately obscuring surfaces can be detected and broken down into separate surfaces that do not.

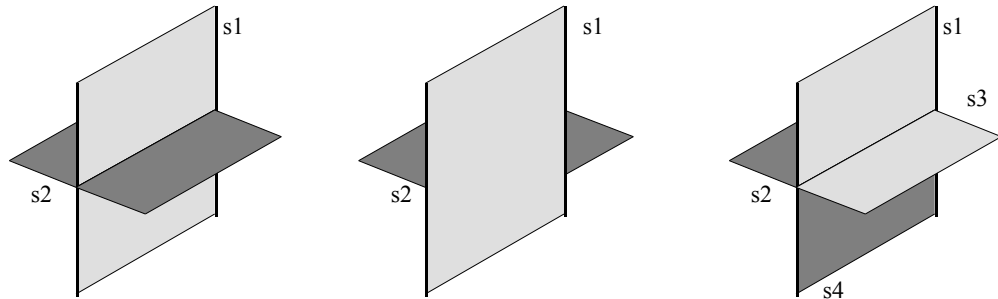


fig III.7: Alternately obscuring surfaces s1 and s2 (left), how they are drawn with the painter's algorithm (middle) and the two surfaces broken down into four separate surfaces s1 - s4 that do not overlap alternately.

Objects that are specifically designed for animations (e.g. ground objects in a flight simulator or the geologic data bases in the GeoBench) never have alternately obscuring surfaces so this problem does not exist there.

- better algorithms like depth-buffering (which can easily be implemented using a 32-bit deep off-screen pixel map that reserves 24 bits for z information and 8 bits for pixel information) use too much memory and have a great impact on performance due to high demands on number processing.
- GeoBench provides a 'triangulation' approach for rendering its scenes that is particularly fit for this algorithm.

The painter's algorithm breaks down into two parts: sorting the polygons according to their z-values (those furthest away are drawn first) and then drawing them. The first step can be done with a great number of methods; usually its a QuickSort derivative that does it and this part is of no further interest. The only thing noteworthy here is that while scanning through the polygons we can apply a technique called *back-face removal* to reduce the number of polygons drawn. For discussion of this subject, please see 'Back-Face Removal', below.

Now let us focus on the second part of the painter's algorithm: the painting itself. Normally, to draw a polygon the programmer would write something like

GrafSys Documentation

```
(* draw a 3-D transformed parallelogram *)
(* the transformed points a-d contain *)
(* the screen coordinates *)
thePoly := OpenPoly;
  MoveTo(a.h, a.v);
  LineTo(b.h, b.v);
  LineTo(c.h, c.v);
  LineTo(d.h, d.v);
  LineTo(a.h, a.v);
ClosePoly;
RGBForeColor(myColor);
PaintPoly(thePoly);
KillPoly(thePoly);
```

Although this approach works, its not particularly fast. It requires ten calls to the Toolbox to draw a simple parallelogram. Two of the calls (`OpenPoly` and `KillPoly`) also require calls to the Memory Manager which may cause the OS to compact the application heap (BIG performance hit) [InsideMac].

Triangles as Specialized Polygons

The great advantage of the above method is that `QuickDraw` accepts any polygon (up to a size of 32K or 8192 points). Its drawback is that it is slow because of the generalized approach and the necessity of allocating and deallocating memory for the data structure. If we provide a highly specialized routine that handles only a special case of polygons we can speed up the drawing process dramatically. With regard to `GeoBench`'s triangulation rendering of scenes, the optimal specialized polygon is the triangle. With it you can build any other polygon through triangulation. Therefore we need a very fast and efficient triangle drawing procedure.

Drawing a Triangle

Let us look at a general triangle ABC:

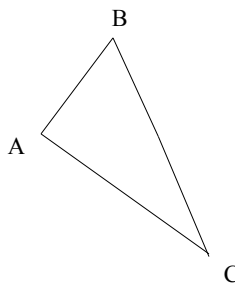


fig III.8: General triangle

To draw a triangle on a raster display (or in our case a pixel map) we could use the obvious approach and calculate the start and ending points of the pixels in the triangle and then fill all the pixels in between.

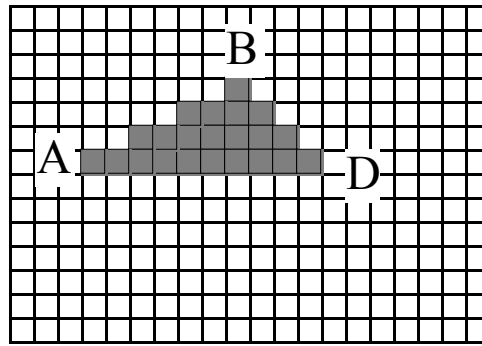


fig III.9: Filling a triangle

The start and endpoints will of course be calculated using the Bresenham line algorithm [Stamm89, Wirth89, Foley90, Hearn86, Pavli82]. However, if we look more closely we can simplify the whole process. If we separate the triangle into two triangles that have a scan-parallel side we can further simplify the drawing process:

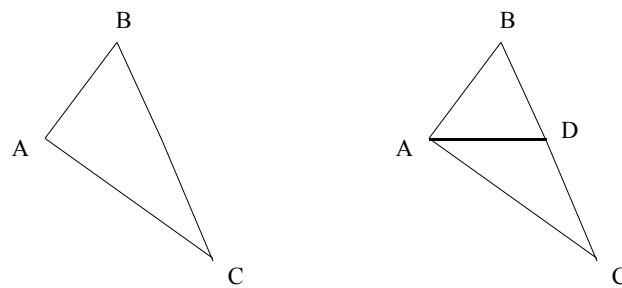


fig III.10: Partitioning a triangle in two scan-parallel parts

After calculating the point D (that is on the same scan-line as A, i.e. has the same Y coordinate) we can write two separate procedures that fill the two scan-parallel triangles ABD and ADC separately. The first procedure fills top-down, the other one bottom-up.

Since we always fill the triangle top-down or bottom-up we do not have to differentiate between drawing up-down or left-right as in the classic Bresenham algorithm. When drawing top-down, the next X coordinate of the start point can be calculated as

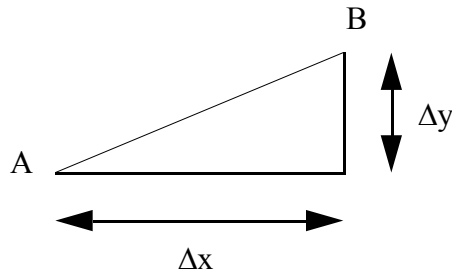


fig III.11 : Calculating the next x position

$$dx := \frac{\Delta\xi}{\Delta\psi};$$

$$\xi := \xi + \delta\xi$$

We therefore come to the following algorithm for drawing a general triangle:

Assign the three points p1, p2 and p3 to ABC in such order that B is the uppermost, C the lowermost and A the point in between.

Convert this general triangle ABC to two scan-parallel triangles ABD and ADC. A must have a lower x coordinate than D and C must be higher up than A and D. Furthermore, A and D must be on the same scan-line. Moving down means increasing Y (this is the normal Mac down direction)

```

xStart := B.x
xEnd := B.x
y := B.y
repeat
    fill all pixels on row Y from Xstart to Xend
    xStart := xStart + dXstart
    xEnd := xEnd + dXend
    y := y + 1
until Y = A.y
    
```

Do the same with the triangle ADC except filling from bottom up.

This is the basic algorithm used for filling the triangle. Below now follows the full Pascal code that can be used as a first approximation for the top-down triangle fill:

GrafSys Documentation

```
(* TopDown requires that a.v = d.v and *)  
(* that b.v < a.v and that a.h < d.h   *)
```

```
procedure TPTopDown (a, b, d: point);
```

```
  var
```

```
    deltaxleft, deltaxright: real;  
    dx, dy: integer;  
    ys, ye: integer;  
    xs, xe: real;
```

```
begin
```

```
  dy := a.v - b.v;  
  dx := a.h - b.h;  
  deltaXleft := dx / dy; (* ATTN: dy may be zero ! *)  
  dx := d.h - b.h;  
  deltaXright := dx / dy;  
  DrawPoint(b.h,b.v)  
  xs := b.h;  
  ys := b.v;  
  xe := b.h;  
  ye := b.v; (* now both point to the start point *)
```

```
  while ye < a.v do
```

```
    begin (* go down one line and plot from s to e horizontal line *)  
      ys := ys + 1;  
      ye := ye + 1;  
      xs := xs + deltaXleft;  
      xe := xe + deltaXright;  
      MoveTo(xs, ys);  
      LineTo(xe, ye);
```

```
    end;
```

```
end;
```

The code for bottom-up is similar.

Implementation

The actual implementation is a four-step process. Firstly, I implemented the whole algorithm in Pascal to test it. Secondly I translated it into C (THINK C) and re-tested it. Thirdly I translated the C code into 680000 Assembler. The fourth step was optimizing the assembler code for greater speed. The following paragraphs discuss certain aspects and tricks in the implementation and why I chose a certain approach. For those who are not too familiar with assembler there is a short 68000 instruction summary in Appendix D.

C and Assembler

Why did I use C in the first place? Well, C is a mid-level language and much closer to the machine than Pascal. In C one can directly assign registers to variables. This step allowed me to perform some optimizations while still retaining some readability.

Furthermore, THINK C provides a built-in assembler that allows the use of labels for variables. This means that one can write

GrafSys Documentation

and C would automatically generate the correct offset and base register [ThCRef]. The normal code would have looked something like

```
move.w    thePoint+2(a6),d0
```

with a great possibility of generating the wrong offset or using the wrong base register. Also, the possibility to mix assembler and C allowed me to step-wise translate and test the program. Also, THINK C's debugger also supports source-level debugging of assembler code, which helped a lot debugging the program.

8 Bit only

When writing the code I realized that I had to settle for a certain pixel-depth, since writing generic code for filling pixels would slow the algorithm down to the level QuickDraw works on (which is still surprisingly fast). The other possibility, writing a dedicated routine for each possible pixel-depth, did not look too attractive to me since the Mac is able to translate one bit-depth into another when using CopyBits. Thus, I settled for the (to me) easiest pixel depth (8 bits per pixel) since this is also the depth I used for off-screen drawing. As an additional boon this made offset calculation especially simple.

word size access

The first time I implemented the actual assembler code I realized with horror that the code was only marginally faster than a patched QuickDraw call to PaintPoly. After some research I realized why: the 68000 Processor is a 16/32-bit machine. Access to memory is done in word size and only to even addresses [Kelly88]. However, the 68000 offers three different opcode sizes, Byte, Word and LongWord (1 Byte, 2 Bytes and 4 Bytes). While word- and longword-sized access have to be to even addresses, a byte-size access can also be done to odd an addresses. In that case, the processor internally accesses the next lower even address word-sized and shifted out the lower 8 bits. This of course wastes lots of cycles.

To speed up the code I added a short overhead to the scan-line fill algorithm that aligned the fill addresses to even and then filled word-size. The method can in effect be coded as follows:

```
if current address odd then  
    paint pixel byte-size, increment address  
  
(* address is now even *)  
save end address and make it even by killing last bit  
while current address <= end address do
```

GrafSys Documentation

**paint two pixels at a time,
increment current address by two
whileend**

**(* now restore end address to see *)
(* if we need to plot another pixel *)
restore end address
if current address < end address then
paint last pixel byte wise**

This little trick sped up the fill algorithm by over 150%. In assembler the address arithmetic is much simpler than it looks from Pascal:

```

                                move.l    a0,d0
                                andi.l    #0x00000001,d0        ; is it even?
                                beq        @alignedleft
                                move.b     d2,(a0)+              ; draw point and update, a0 now
even
                                cmpa.l     a0,a1                  ; done drawing?
                                ble        @nextline              ; yes, fill next line

@alignedleft
                                /* check if last pixel follows immediately. If so, draw and */
                                /* go on to next row */
                                cmpa.l     a0,a1
                                bne        @alignright
                                move.b     d2,(a0)                ; draw point and go
                                bra        @nextline              ; to next line

                                /* now align the right bound. this is done by masking out */
                                /* the last bit of the address and plotting by checking */
                                /* against the new boundary */
@alignright
                                move.l     a1,d0
                                andi.l     #0xFFFFFFE,d0        ; mask last bit
                                movea.l    d0,a3                  ; and save end address in a1

                                /* now fill scanline up to the even end address */
@setpix
                                move.w     d2,(a0)+              ; draw horiz. line
                                cmpa.l     a0,a3                  ; done drawing?
                                bgt        @setpix

                                /* now check to see if end address was originally odd */
                                cmpa.l     a0,a1                  ; did we mask out one?
                                ble        @nextline              ; nope, next line
                                move.b     d2,(a0)                ; yes, draw and go
                                bra        @nextline              ; to next line

@finished
```

The above assembler code fills a scan line from the address stored in a0 to the one in a1 with as much word-sized accesses in between as possible. There is only one optimization that is not apparent from the Pascal code. If you look at the instructions directly following the @alignedleft label you will see that the code checks immediately if we are done to avoid

GrafSys Documentation

unwanted drawing. This is expressed implicitly in the Pascal while statement and was

GrafSys Documentation

coded like this to avoid an unwanted check at the beginning of each drawing iteration.

fixed arithmetic

Another possible bottleneck in the triangle routines becomes apparent if we look at the following lines:

```
xs := xs + deltaXleft;
xe := xe + deltaXright;
```

Unfortunately, `deltaXleft` and `deltaXright` are real numbers, their fractional part being very important. Floating-point arithmetic is very cumbersome and slows down execution speed considerably. Even with only 100 lines to fill we need 200 floating point operations.

A better method would be using fixed-point arithmetic. In fixed-point arithmetic, addition and subtraction can be done like any other binary operation, only multiplication and division are more complicated. If you look at the code, you will realize that only two divisions per call are required but two additions each scan line. The Mac supports fixed-point division and multiplication and thus this problem was solved easily. The Macintosh definition for the Fixed data type is the following [InsideMac]:

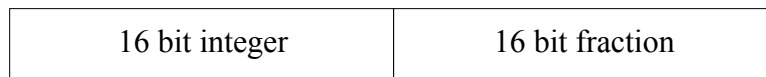


fig III.12: Fixed data type

Thus, adding two fixed-point numbers amounts to nothing more than adding two 32-bit numbers in assembler. To access the leading 16-bit integer, a simple `swap` followed by an `andi` is sufficient:

```
move.l    deltaXleft,d0
add.l     d0,xs          ; xs := xs + deltaXleft
move.l    deltaXright,d0
add.l     d0,xs          ; xs := xs + deltaXleft

move.l    xs,d0          ; calc start pixel
swap      d0
andi.l    #0x0000FFFF,d0
```

Setting up the variables is equally simple:

```
deltaXleft = FixRatio(dx, dy); /* this is C Code */
deltaXright = FixRatio(dx, dy);
```

Since THINK C allows this mixture of C and assembler this is no problem for the compiler [ThCRef]. `FixRatio` is a standard Mac toolbox call.

Degenerate Triangles

Another dramatic speedup over QuickDraw can be achieved if one checks to see if the triangle degenerates to a horizontal line or even a single point. This happens very often in 3D transformations if the triangle is far removed from the eye. In landscape views in GeoBench for example, about 10-25% of all triangles degenerate to a single point. Since the triangle routines check for this condition before any other drawing takes place, a speedup of a factor of 5 to 12 (overall) can occur on complex scenes.

Boundary Check

When moving from the abstract drawing plane that is infinite in both directions to the actual device to draw upon, we must check for boundaries. This is critical in three aspects:

- If we draw outside of the boundaries, we will destroy memory that does not belong to the graphics device. This can either be other screen memory (at best) or memory for variables or even worse, our own program. We therefore need to clip the drawing to the area defined.
- Boundary checks should be smart enough to stop drawing or calculation at the first possible moment but
- should not slow down the algorithm

Meeting both the last two criteria simultaneously is almost impossible but a good approximation can be reached by placing the checks sensibly.

The boundaries are defined as the port's `portRect` [InsideMac] and can easily be accessed through the `GetPort` procedure. Using this and the `PixMap` definition [InsideMac] one can not only calculate the boundaries but also the start and end addresses of the scan line to fill.

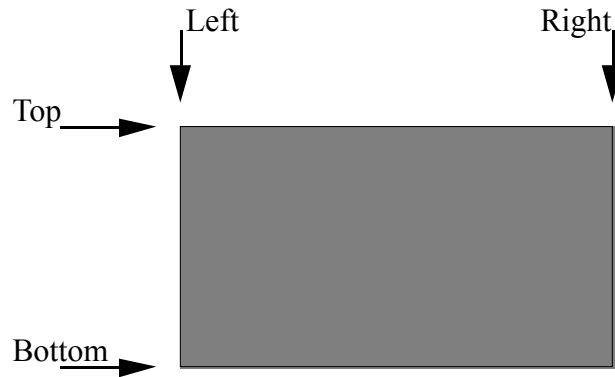


fig. III.13: Boundaries in a Graphics Device/Port

Making the boundary checks smart enough can be done by employing some object coherency: If we are drawing top-down, for example, we can stop the whole drawing process the instant our y exceeds the bottom boundary. Similar conditions exist for drawing bottom-up.

Clipping a scan-line can likewise be easily implemented. If the start pixel is left of the left boundary, the start pixel will be set to the left boundary. Likewise, if the end pixel is right of the righter bound, it is set equal to the righter bound. The triangle is now clipped to the bounding box. And now comes the smart part: If the start pixel is right of the end pixel it means that the whole line is either entirely left or right outside the bounding box and should not be drawn. Using these three simple checks makes more complicated checks superfluous.

The last criteria, no impact on execution speed can never be matched totally since a check and branch always costs cycles. We therefore must try to minimize the checks and place them strategically. For example it would not be wise to place the Y -check inside the loop that fills the scan line, since we do not change the Y -coordinate inside it.

Thus we arrive at the following pseudo-code for drawing the triangles into a bounding box (Top-Down):

```
while  $y < a.v$  do
    if  $y > \text{Bottom}$  then exit loop
    calc adrs for pixel start and pixel end
    if  $\text{start} < \text{left}$  then  $\text{start} := \text{left}$ 
    if  $\text{end} > \text{right}$  then  $\text{end} := \text{right}$ 
    if  $\text{start} < \text{end}$  then
        fill row from start to end pixel
whileend
```

GrafSys Documentation

The checks from assembler are straightforward and offer no real surprise. The sequence `swap d0, andi.l 0xFFFF,d0` is the normal fixed-to-integer conversion since `xs` and `xe` (the pixel start and end values) are of Fixed type. All checks are done outside the actual drawing loop.

```
/* check for valid bounds left */
        move.l      xs,d0                ; get left start
swap      d0
andi.l    #0x0000FFFF,d0
cmp.w     bounds.left,d0                ; if xleft < left
bge       @lt1
move.w    bounds.left,d0                ; then xleft := left
@lt1     move.w      d0,xpos              ; save it
        add.l      d4,d0                ; add base adress
movea.l   d0,a0
move.l    xe,d0                        ; get end pixel
swap      d0
andi.l    #0x0000FFFF,d0

/* check for valid bounds right */
cmp.w     bounds.right,d0               ; if xright > right
ble       @lt2
move.w    bounds.right,d0
@lt2     cmp.w      xpos,d0              ; if xright < xleft
blt       @next                        ; then next iteration
add.l     d4,d0                        ; add base adress
movea.l   d0,a1                        ; put end pixel into a1

/* check valid bounds top, bottom */
cmp.w     bounds.top,d5
ble       @next                        ; still not inside
                                           ; the window
cmp.w     bounds.bottom,d5             ; if ys > bounds.bottom then
bgt       @done                        ; can abort this plot, no
                                           ; bottom-up required

/* now plot the row */
```

Writing to the screen

Although I originally did not intend to implement it, the current version of the triangle paint routines are able to draw directly to the screen, provided the screen is set to 8-bit pixel depth (256 colors). This was made possible by a few additional lines of code that accessed the current GrafPort's `portRect` and it's `pixel Maps` bounds fields. There are great differences between writing to screen memory and a memory-based pixel map:

- Screen Memory usually resides in a video card and
- to access Screen Memory one must go over the NuBus (performance bottleneck).
- to access Screen Memory one must switch to 32-bit MMU mode. In this mode, however, you may not use the Memory Manager or certain Toolbox routines.
- the memory associated with the pixel map is just a small portion of the

GrafSys Documentation

whole pixel map and the bytes per row have no

GrafSys Documentation

resemblance with the actual width of the pixel map. Furthermore, the pixel map bounds and the ports bounds do not match:

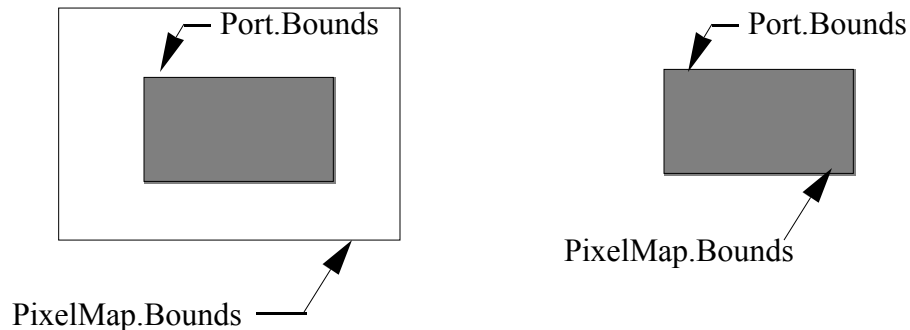


fig III.14: Bounds of a Port in Screen Memory (left) do not match with the pixel map's bounds as they do in off-screen pixel maps (right)

There are a number of reasons why its not a good idea to write directly to the screen [Brig92, InsideMac]:

- It is bad to assume a certain bit-depth of the screen (this violates the Apple Human Interface Guidelines)
- It's likely to break in the future on new video devices that support a radically different imaging model (e.g. Display Postscript or chunky pixels)
- Writing directly to the screen loses QuickDraw's clipping ability
- The code will probably fail if the port crosses two or more monitors
- Writing directly to the screen blasts the Help Manager's windows
- QuickDraw's ability to map pictures and pixel maps from one color environment to another is lost
- The ability to print is restricted since the printer only understands QuickDraw. One must therefore use `CopyBits` to copy the graphics to the printing GrafPort at a great loss of detail.

To avoid most of the problems [Brig92] suggest two possible alternatives, both of which have been implemented in addition to the possibility to draw to the screen:

- *Always supply a QuickDraw alternative.* The triangle routines can be forced to use QuickDraw calls and do so automatically if the depth of the pixel map is not equal to 8 bits per pixel. The QuickDraw calls used internally are significantly faster than the normal sequence since they do not have to allocate a polygon structure.

GrafSys Documentation

- *Draw into off-screen pixel maps and use CopyBits to copy the image to screen.* This is up to the programmer but I strongly encourage you to do so. Remember that GrafSys supports this with the `BeginOSDraw`, `EndOSDraw` and `CopyOS2Screen` procedures.

If you use the triangle routines to draw directly to the screen you must be careful to ensure that the window you draw into is frontmost since the triangle drawing routines ignore any `ClipRgn` settings. If you draw a triangle directly on the screen and the window is partly obscured by another window, the triangle could be drawn onto the obscuring window. To avoid this, always draw into the off-screen pixel map and use `CopyBits` which does honor the `ClipRgn` to write to your window. Make sure you issue a `ShieldCursor` before and a `ShowCursor` after calling the Triangle routines to avoid unwanted 'special effects'.

Results

The results I achieved with the new specialized triangle routines are somewhat ambiguous. They are faster in most aspects but under certain circumstances they can be even slower than using `QuickDraw`. The following paragraphs discuss these circumstances and how to avoid them.

Normal Operation

In normal (i.e. average) operation with average-size triangles, the triangle fill procedure yields a speedup factor of 2 to 3, depending on the size of the triangle and how much is clipped. The larger the triangle, the less the performance gain.

Borderline cases

Performance gain is zero if the triangle obscures the whole pixel map (port). In this case a `PaintRect` call would be much faster, `QuickDraw`'s `PaintPoly` procedure is exactly as fast as the `FillTriangle` procedure.

If the whole triangle falls off the screen, `PaintPoly` and `FillTriangle` are almost identical in speed. `PaintPoly` is a bit faster since it checks if the bounding box falls out of bounds. However, this gain is lost due to the necessity of calling ten toolbox procedures to set up and destroy the triangle.

If the triangle almost degenerates to a single point, `FillTriangle` is up to 6 times faster than `PaintPoly` since its overhead for setting up the data structure is much smaller

GrafSys Documentation

compared to the time spend drawing. In addition to that `FillTriangle` specifically checks for this condition and is therefore much faster.

Processors with Cache

Since the drawing loops in the `FillTriangle` procedure are much tighter due to the higher specialization, using a processor with instruction caching yields better results over QuickDraw's `PaintPoly`. On a 25 MHz 68040 with Data- and Instruction caching on, the performance gain is boosted to an average factor of 4 to 5, with degenerate triangles to 12 to 15. Since the triangle procedures are written in 68000 code they do not take advantage of the specialized addressing modes the 68020 or newer processors provide, so performance will not degrade under a factor of 1 on 68000 equipped Macintoshes.

Dedicated Hardware (only direct to screen)

There is, however, a case where using QuickDraw is consistently faster than the `FillTriangle` procedure. This is when the Mac is equipped with special hardware to accelerate drawing such as the 8•24 GC Video Card with Graphics Co-processor. In this case no procedure can hope to compete with the asynchronously running RISC processor. Since there are Macs that have this or other graphics co-processors installed (e.g. the newer Macintosh '*' av machines) there must be a way to avoid the unwanted slowdown without great code modification.

Therefore, the `FillTriangle` procedure has a parameter that tells it to use the QD procedures. In normal programs it would look something like

```
... (* initialization code *)
if hasQDaccel then useQD := TRUE
    else useQD := FALSE;

...
(* now draw the triangle ABC to the screen *)
FillTriangle(A,B,C,myColor,useQD);
```

Using this method you can have the best of both worlds: Fast triangles when no graphics co-processor is available and faster QD calls if there is.

Note however that some graphics co-processors are only faster if you are actually drawing to the screen. If you happen to draw into off-screen pixel maps, you may find that the `FillTriangle` procedures are faster. This depends on the type of graphics co-processor you have, so maybe you should set up a little timing procedure that tests which procedure is actually faster.

Back-Face Removal

In order to speed up drawing, one can try to eliminate surfaces that cannot be seen. While eliminating these surfaces on general objects can be quite complicated, it is very easy for convex polyhedron. In these cases you can eliminate a surface simply by determining which side is facing the eye. If one is looking at the backside, you cannot see it because it will be always obscured by other surfaces.

Eliminating these surfaces is very easy. Since after transformation the eye is always sitting at the world's origin and is looking straight up, it would be looking at a backside whenever the normal vector of a surface has a negative z component.

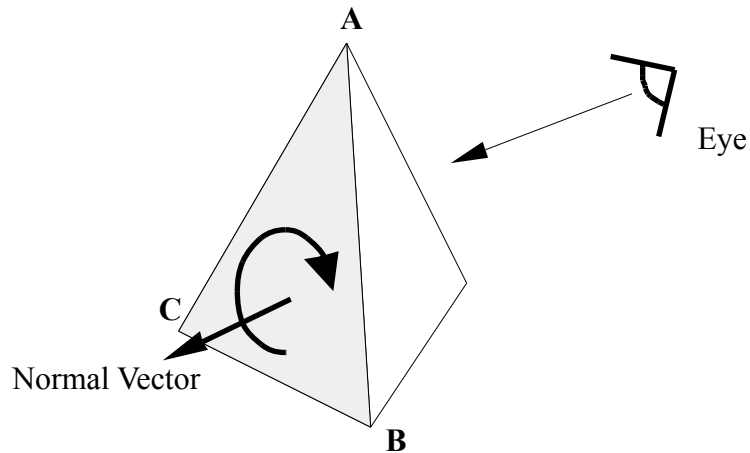


fig III.15: Backface identification. The eye cannot see the shaded surface. This is because looking from the eye, the order of the plane definition Points ABC would look reversed and the normal vector would face inwards (when seen from the eye).

In order to define a surface (or in our case a plane since GrafSys does not support curved surfaces) three points suffice. If we label these points A, B and C and count them clockwise, the direction of the normal vector \mathbf{n} is defined. Note that the direction of \mathbf{n} is defined by the order you pass the points. If you reverse the order (e.g. B, A, C), the normal vector would point into the opposite direction.

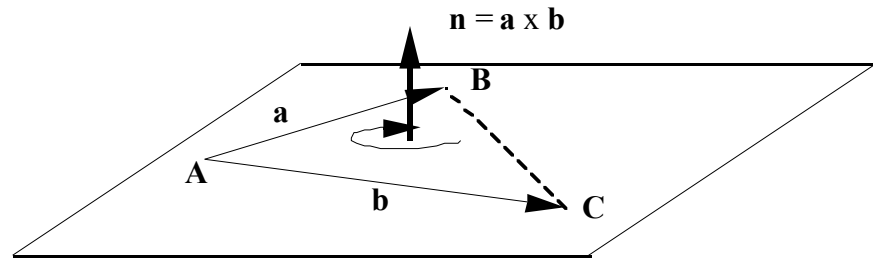


fig III.16 : Definition of $\mathbf{n} := \mathbf{a} \times \mathbf{b}$.

To remove a backface, all we have to do is test for a 'backface situation', i.e. to look if the normal vector of the plane defined by the three points A, B and C has a negative z component. If it does, the plane cannot be seen from the eye and does not have to be drawn. Thus, to detect the backface situation, all we have to do is to look at the z component of \mathbf{n} .

Let

$$\mathbf{a} = \mathbf{B} - \mathbf{A}$$

$$\mathbf{b} = \mathbf{C} - \mathbf{A}$$

$$\mathbf{n} = \mathbf{a} \times \mathbf{b}$$

However, for our implementation only the z coordinate of \mathbf{n} is relevant. Therefore, we do not need to implement a whole vector multiplication algorithm but simply calculate \mathbf{n}_z .

According to the definition of vector multiplication,

$$\mathbf{n}_z = \mathbf{a}_x * \mathbf{b}_y - \mathbf{a}_y * \mathbf{b}_x$$

And since

$$\mathbf{a}_x = \mathbf{B}_x - \mathbf{A}_x$$

$$\mathbf{a}_y = \mathbf{B}_y - \mathbf{A}_y$$

$$\mathbf{b}_x = \mathbf{C}_x - \mathbf{A}_x$$

$$\mathbf{b}_y = \mathbf{C}_y - \mathbf{A}_y$$

we can simply write the following function that returns true if the plane defined by k, l and m has a normal vector with positive z component:

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

var

nz: real;
ax, bx, ay, by: real;

GrafSys Documentation

begin

```
{ax := (l[1] - k[1]);{}
```

GrafSys Documentation

```
{bx := (m[1] - k[1]);{  
{ay := (l[2] - k[2]);{  
{by := (m[2] - k[2]);{  
{nz := ax * by - ay * bx;{  
nz := (l[1] - k[1]) * (m[2] - k[2]) - (l[2] - k[2]) * (m[1] - k[1]);  
IsVisible := nz >= 0;  
end;
```

Since this routine directly accesses the vector components, it is implemented in the Matrix package and coded depending on the version either in floating point (881 version) or fixed-point arithmetic.

The function IsVisible can now be used to detect if a surface is visible or not to implement hidden-surface removal.

Part IV

GrafSys and GeoBench

Overview

This section describes how the GrafSys was integrated into the GeoBench and what steps and adaptations were necessary to accomplish it. We start with the description of the library and its files and then continue with the changes that were necessary in the GeoBench itself.

Including The Library

Usually, the GrafSys comes in a pre-compiled monolithic block and including the library is nothing more than just including library and interface file into the project. However, since GeoBench made some slight adaptations within the GrafSys necessary, I chose to include the source files and not the pre-compiled library. This also gives me the opportunity to describe what part of the GrafSys I implemented in which source file.

The full GrafSys consists of twelve separate files that each implements a certain aspect:

- *Matrix2*: This unit implements all (speed-optimized) matrix-matrix and matrix-vector manipulation. Furthermore, this is the unit where implementation-specific code for fixed-point and floating-point arithmetic resides. All further units use the Vector4 data type that is defined here.
- *Transformation2*: Implements the transformational procedures required to set up and handle rotation, translation and scaling.
- *OffScreen Core*: Here are the low-level off-screen port and device generation, resizing and disposal routines
- *GrafSys Core*: This is the central GrafSys unit. It defines the common objects as well as the matrix and basic 3D object. The central transformation calculation method and all inheritance mechanisms are defined here.
- *GrafSys Aux*: This unit defines some less important methods for the 3D Point and 3D Line objects.
- *GrafSys Screen*: Here the 3D Window and all its manipulation routines are defined. Additionally, the 3D object is instanced to handle the point data base. All methods to access and manipulate points in the data base are found here.
- *GrafSys Object*: Again the 3D object is instanced, this time to handle lines. All methods to access and manipulate lines are found here.
- *ResourceAccess*: This unit implements the low-level procedures required to create and access the resources required by the GrafSys. A description of the GrafSys resource format can be found in Appendix C.

GrafSys Documentation

- *Resource*: This unit implements the loading and saving of 3D objects. It's mainly just a hi-level interface to the procedures defined in Resource Access.
- *OffScreen Graf*: This unit contains all the hi-level off-screen graphics routines required to create, use and destroy off-screen pixel maps. It implements the shadowing technique for 3D windows and offers a simple interface for a customized CopyBits routine. Furthermore, it handles palette access and CLUT management.
- *GrafSys C lib*: This (compiled) unit contains the assembler routines for high-speed triangle drawing and pixel map erase. Since Pascal cannot handle assembler, this unit has to come pre-compiled. The name might be misleading since it implies that the triangle routines are written in C. Alas, they originally were (hence the name) but were subsequently translated to assembler.
- *GrafSys C int*: This is the interface file for the assembler routines.

GrafSys Documentation

Changes To GeoBench

Besides the obvious changes (including the library files), there were numerous changes the original source required in order to get the GrafSys running with GeoBench. I have tried to keep these changes to a minimum and therefore made some subtle changes to the GrafSys as well so that they cooperate better. The main problem with integrating the GrafSys into GeoBench was that GeoBench scatters the 3D imaging and viewing procedures over many units. Furthermore, GeoBench takes a different approach to 3D transformation.

General (global) Changes

Some changes had to be performed throughout all GeoBench units. Since GrafSys uses the Macintosh window structure to store its own eye transformation data, I had to change all window manipulation routines inside GeoBench to take this into account.

NewWindow

Throughout the GeoBench, whenever a new window was opened, it now opens a 3D window instead. This is immediately followed by a call to `SetEye` to initialize the projection and clipping variables. Note that there is no conflict with the way `TransSkel` uses windows.

SetPort

All `SetPort` procedures have been replaced with `Set3DPort` calls. This procedure sets `QuickDraw`'s current port to the indicated and also the current 3D port, but only if the window is a 3D window.

Color Model

I noticed that GeoBench uses the HSB color model. Interestingly, GeoBench provides its own color conversion and management procedures instead of the toolbox HSV manipulation procedures. At first I changed this but later I decided against it and changed it back.

Other Changes

The greatest changes in GeoBench were done in those units that did the 3D transformations and projections. They were the units

- *graphics*

I changed the `DrawCoordSys` procedure. Now GeoBench loads a 3D object while initialization that exactly resembles the coordinate cross with labels. The seventeen calls to `drawLineSegment3d` are now replaced by a single call to `theCoodSys.draw`. Speedup factor is around 12 to 15

GrafSys Documentation

depending on the scale factor. For more details on this change please read the chapter '3D Graphics First Step', below.

GrafSys Documentation

- *triangulation3d*

To speed up the triangle drawing process I replaced the `OpenPoly-BuildPoly-ClosePoly-DrawPoly` sequence with a call to `FillTriangle`. For more details on this change please read the chapter 'Triangles', below.

GeoBench Interface

To adapt GrafSys to GeoBench there were numerous changes required. I added a new unit that facilitates a smoother integration of GeoBench and GrafSys. This unit is mainly used to translate the two different drawing models. In GeoBench the view-up direction is actually down (since the positive y direction on the Mac is downwards) and the origin at the upper left corner. Additionally, GeoBench uses a little undocumented (thanks a lot, folks!) hack to allow zooming. A patch to GrafSys now supports this zooming feature.

Triangles

The high-speed triangle drawing routines were specifically designed for the GeoBench since the spectacular geologic pictures created with it use the Triangulation approach.



fig IV.1: Churfirsten (part of a mountain range in Switzerland) drawn with GeoBench/GrafSys using Triangulation.

For this I had to change the pattern-fill algorithm to a pure color fill algorithm. Additionally, 3D triangles may be clipped and the resulting figure may not be a triangle anymore, becoming a trapeze or pentagon. In this case the old polygon-style drawing routines are used. Although designed for speed, I had to discover that more than 98% of the time used to draw above example were used in the overhead of calling different methods. Since this is intrinsic to the way GeoBench stores and handles data, I was unable to change this.

GrafSys Documentation

As an additional change the picture is now built off-screen and then copied onto the screen at once using the GrafSys off-screen procedures. I originally intended to buffer every 3D window with an off-screen pixel map. However, the extremely high demands on memory forced me to abort this plan because my development system ran out of memory the moment I opened the first window. Currently, GeoBench allocates an off-screen buffer when it wants to draw a triangulated object, draws it off-screen, copies it on-screen and then deallocates the buffer.

Still, the advantages of permanent off-screen buffers are immediately apparent: currently a redraw (as required when a window update event occurs) of above picture takes about ten to twelve seconds on a Centris 650, while redrawing it through `CopyBits` from off-screen takes about three tenth of a second. If the off-screen buffer was to be permanent, window updates would be much faster.

3D Graphics First Step: CoordSys

The first and most difficult step in integrating GrafSys into GeoBench was synchronizing the two viewing transformations. GrafSys is primarily geared toward speed, while GeoBench's main goal is obviously robust code. But there are more differences between the two packages:

- While GrafSys employs a window-oriented one-eye/one-window strategy with independent object rotation and translation, GeoBench uses an all-in-one approach. The eye is tied to a frame data structure and one window may have multiple frames, although only one frame per window is shown at any given time.
- The view-up vector points down in GeoBench and the origin is at the upper left corner. Furthermore, GeoBench implements zooming through a dubious (and totally undocumented) `m` and `d` parameter pair.
- To make things worse, GeoBench uses numerous routines to convert 3D coordinates to a 2D projection and these routines are scattered over many units plus they are called through callback techniques, so tracing the various transformations was a nightmare.

In GeoBench the actual drawing of the coordinate system is done through repeatedly calling `drawLineSegment3d` who in turn call `drawLineSegment` who call an `execute` procedure to execute the passed round function and then call the QuickDraw `MoveTo` and `LineTo` procedure to actually draw the lines. In between, the line is first clipped to the viewing pyramid and then clipped to the drawing plane. Each start and end point is transformed separately. This means, for example, that the origin

GrafSys Documentation

[0,0,0] get transformed three times since three lines (one for each axis) begin there.

My main goal was therefore to integrate GrafSys to boost performance (if possible) and code readability while retaining maximum compatibility and patching as few lines as possible. This has been done using the following techniques:

The Eye And Transformation Adaptation

The (GrafSys) eye was reprogrammed to always follow the current frame, not the current window. For GrafSys this is logically the same since each frame has a window and only one frame is active per window at the same time.

Then, I had to adjust the GeoBench eye transformations to those used in GrafSys. After some detailed studies, I found out that

- GeoBench and GrafSys use the same viewing transformation technique except for the view-up direction and origin placement
- GeoBench provides but does not use the user transformation matrix. All transformations are immediately applied to the eye transformation matrix.
- Zooming is done using a zooming parameter (instead of scaling and transformation as would be the correct way)

Therefore I wrote a procedure called `SyncEye` that does the following:

- The GrafSys origin is placed to the coordinates indicated by the frame.
- The current transformation matrix contained in the frame is translated to the GrafSys `Matrix4` transformation operator type. This is just done for cleaner code and not really necessary. Since the user transformation matrix is ignored by GeoBench, so does the `SyncEye` procedure. If you want to change this, all you have to do is to convert both to `Matrix4` type. Then multiply the user transformation with the eye transformation matrix to receive the final transformation matrix (note that this is done implicitly when the view matrix is set up).
- Finally, the zoom parameters (`dx`, `dy`, `mx` and `my`) are copied into the current eye data structure. This data structure was extended just for this purpose and is not the standard GrafSys structure any more.

A small patch in the GrafSys transformation method now includes the zoom parameters. The patch directly applies these parameters

GrafSys Documentation

when the 3D points are transformed but before the projection methods (parallel projection, perspective projection) are executed.

A different (and maybe even cleaner) possibility would have been to convert the zoom parameters to translate and scale factors, but the effect would be the same, so I took the simple (and faster) approach. The correct method is to allocate an FF operator (postconcat) and set the scale to [mx, my, 1] and translation to [dx, dy, 0] but my patch is much simpler to understand.

After implementing these minute changes, calling SyncEye prior to calling an object's draw method is all that is required to make GrafSys work correctly with GeoBench. As an additional boon I was able to use the many-eyes-per-window feature implemented in GeoBench through the frames because GrafSys now synchronizes the 3D eye with both 3D window and frame.

Clipping: 3D vs. 2D

GeoBench uses a very elaborate clipping mechanism: All lines are first clipped to the viewing pyramid, projected onto the 2D plane and then clipped to the QuickDraw drawing plane. After some experimentation I found out that the first (and complicated) step is unnecessary. The only important step is clipping the lines after projecting them (note that in perspective projection GrafSys also adds clipping to those lines that penetrate the projection plane) to 2D. However, since QuickDraw clips those lines to the visible region anyways, this clipping is usually not necessary.

The GeoBench version of GrafSys clips lines the following way:

- Lines that penetrate the projection plane are clipped to the penetration point according to the `ArithmeticClip` method described in Part II, above. This is only done in perspective projection.
- Lines that go outside QuickDraw's drawing plane (-32'000 to +32'000 in both X and Y direction) are clipped to the drawing plane using the Cohen-Sutherland 2D clipping algorithm [Foley90, Hornu89, Pavli82, Hearn86].
- QuickDraw clips the lines to the window while drawing [InsideMac]

CoordSys Model (in resource)

After I adapted the GrafSys transformation routines to act exactly like GeoBench did, all I had to do was to replace the seventeen `drawLineSegment3d` calls with a single call to GrafSys' draw method. For this I modeled the coordinate system as a

GrafSys Documentation

TObject3D, saved it in resource format and pre-load it when the program initializes.

Now DrawCoordSys simply synchronizes the eye with the currently active frame and then draws the coordinate system using the fast (and straightforward) GrafSys routines.

3D Graphics full integration

After I adapted the core routines for transformation and projection I started to adapt GeoBench to utilize GrafSys for drawing 3D polygons as described in [Brünn91]. The current implementation uses 3D objects for all drawing. However, the 3D objects are GrafSys representations of the object they are attached to. This means that almost all data is duplicated, albeit GrafSys uses a more compact form.

A more efficient implementation would have been to use the GrafSys data base as a repository for all point data within a 3D polygon. However, this would have been a severe limitation to GeoBench which can model points using real, integer and other representations. Therefore I decided against it.

The Polygon adaptation

Originally, I intended to simply extend the `layerVector` data type to include a `TObject3D` that modeled the different layers in a single 3D object. However, I soon found out that this would not work, since the polygons had to be either simply drawn in wire frame format (as GrafSys provides) or filled, if the visibility flag in GeoBench was set. This GrafSys could not provide. Therefore, I had to extend GrafSys for a new layer model 3D object (as I suspected from the beginning).

TLayer3dObj

The first approach extended the `TObject3D` type to add polygon handling and drawing. Since GeoBench's polygons are of the poly-line kind (two successive points are connected, no other connections except for the first and last point who are connected), implementation was simple. An array held the start points for each polygon. Drawing was equally simple. I began with the first point and continued collecting successive points into the polygon until I reached the first point of the next polygon (since GrafSys accesses points through index, this was very fast) and closed the polygon (i.e. connected the last point with the first). If visibility was set, I used QuickDraw's `FillPoly`, otherwise `FramePoly`. Note that since I used QuickDraw's polygons, I would have problems if the total number of points in the polygon exceeded 8'192 (32K). But since the original GeoBench used the same approach, I figured this

GrafSys Documentation

was fair. GrafSys itself would have no problems framing larger polygons (up to 250'000 points), but it cannot fill them.

Unfortunately, this first attempt had two major flaws:

- GeoBench provides three different ways to draw the layers: Top-Bottom, Bottom-Top and Middle-Out. The TLayer3dObj only supported one way: First-defined-First-drawn. This is fundamentally different from the three ways GeoBench draws 3D polygons.
- The flow of control used in GeoBench was totally disrupted. Furthermore, GeoBench theoretically supports multi-dimensional polygon arrays (due to the way the LayerVector data structure was designed). If in later changes to GeoBench this was used, the Tlayer3dObj will fail to work while the current implementation (without GrafSys) is able to handle it.

Therefore, I had to come up with another idea.

TPolygon3dObj

The next incarnation went further down the object hierarchy. Instead of one 3D object per layerVector, I designed a new object that was attached to each layer. This way, drawing the layerVector does not disrupt the original flow of control, it only cuts out the overhead involved in the actual drawing. Furthermore, now the layers are always drawn in the desired order without additional code.

TPolygon3dObj is a descendant to TSGenericObject3D, not TSOBJECT3D and overrides only the draw procedure. Since an object only models a single polygon, the draw method is even simpler than the one I wrote for the TLayer3dObj. Also, this design allows for easy adaptation to multi-dimensional polygons if anyone should ever find a need for that.

As an additional feature it is now possible to rotate, translate or scale each layer independently since points in an GrafSys object always have a Z coordinate that can be transformed. GeoBench uses only a global first-z and z-increment for the layerVector, GrafSys models each point independently. Also, it is now possible to model non-planar (e.g. curved) polygons since the points in each layer do not have to reside on the same z plane.

This approach proved not only to be efficient but also fitted well into the GeoBench since only a minimum amount of changes was

GrafSys Documentation

required; the object methods for TPolygon3dObj are defined in the GrafSysInterface unit.

Changes To GeoBench

As it turned out, only three methods and one object had to be changed in GeoBench in order to accept GrafSys:

- *Unit layer:*
The layer object itself had to be extended for one instance variable, the 3D object of type TPolygon3dObj.

`layer.interactiveOutput` now calls the GrafSys drawing methods to draw the polygon it models. All the local procedures are now obsolete. However, I decided not to delete them in case someone still needed them for reference.

- *Unit layerVector:*
`layerVector.setFirstZAndDz` now adds a call to a new method that sets all z values in the layer's 3D object to the given z coordinate.

`polyLayerVector.execute` is noticeably extended. It has three new local procedures to translate the various layer and point objects to GrafSys 3D objects. The main procedure, `Build3DPolys` breaks up the `polyLayerVector` into layers and calls `LayerTo3dObj` for each layer. This procedure allocates a new TPolygon3dObj and collects all points that reside in this layer into the 3D object using `Vector2Element` and `PointsTo3dObj`.

Problems Encountered

The biggest problem with the GrafSys integration into GeoBench was clearly the memory allocation problem. Development was done on a Centris 650 with only 8 MB of main storage, clearly not enough for a 1.4 MB (code size) memory hog like GeoBench. A single `polylayer` vector in GeoBench gobbles up an easy 100K minimum because of the numerous objects involved. Adding the equally memory-hungry GrafSys became a major nightmare since THINK Pascal ran out of memory immediately, especially when the debugging option was turned on.

All different kind of intermittend fatal errors occurred and THINK invariably traced them to the `SANE881.lib`. Needless to say, they did not occur there but somewhere in between. The most encountered error was the Bus Error ('Type 1 Error') which usually occurs if a procedure tries to allocate an object, gets a nil reference because

GrafSys Documentation

memory ran out and then tried to access an instance variable or worse, passed the `Init` message.

For this reason, the unit `GeoBenchUtility` provides a dedicated `failNil` procedure that checks for this situation. Unfortunately, not all procedures use it. For example, the `initLayer` procedure does not check if the allocated objects really exist. This does not really matter until one tried to triangulate a `polyLayerVector`, in which case the program aborted with said `Bus Error` somewhere in the middle of triangulation. This kind of annoying errors greatly impeded program testing and adaptation.

The second biggest problem originated in the highly object-oriented structure of `GeoBench`. Although this is usually an advantage in programming, it becomes a severe obstacle if one tried to patch certain procedures that are called via callback or procedural parameters (e.g. the `forAll` message). This is amplified by the problem that `GeoBench` relies on a custom version of `TransSkel` as backbone, the skeleton application from pre-OOP days that contains more bugs than the Indonesian rain forest.

After these problems were mastered, adapting the `GrafSys` to `GeoBench` was not that difficult any more (this of course also stemmed from the fact that `GrafSys` was designed with `GeoBench` in mind). The current adaptation is well below the maximum possible integration (as mentioned above, 3D objects are just translation of their `GeoBench` analogons) but it works well and almost seamless.

I also refrained from removing any 3D transformation and projection code (although it is never executed) and even went so far as to use `GeoBench`'s eye calculation procedures to guarantee maximum compatibility.

GrafSys Documentation

Conclusion

Although GrafSys is now fully integrated into GeoBench, the performance improvements leaves a bit to be desired. However, the problem lies with GeoBench, not with GrafSys. The Fast Triangle debacle is symptomatic for all the problems I had with speed improvement. The actual drawing and transformations are faster for a factor of 10 and above. However, the overhead introduced by message passing and run-time binding take up more than 95% of the time required to draw a frame. If we had a performance factor of 10, this would only result in an overall speed gain of 4%.

On the other hand, GeoBench now has a broad, versatile, fast and easy-to-use 3D graphics fundament that will make further 3D experiments much easier to implement. Also, if the new experiments are designed with GrafSys in mind from ground up, they can profit from the speed designed into it. Fluent animations with many objects and/or hidden-line/hidden-surface removal with frame rates of 20 per second are now easily possible.

I hope that GrafSys improves current and coming versions of GeoBench as much as I envisioned and that its structure and power is of good use for other projects as well.

Christian Franz,
September 1st, 1993

Appendix A

GrafSys Reference

GrafSys Documentation
Non-OOP procedures

General Procedures

```
procedure InitGrafSys
function InterpretError (theErr : integer) : Str255;
function GrafSysVersion: longint;
procedure SetVector4(var v : Vector4; x,y,z : real);
procedure GetVector4(v : Vector4; var x,y,z : real);
function GetNewObject (theObjectID: INTEGER)
    : TObject3D;
function GetNewNamedObject (theName: Str255)
    : TObject3D;
procedure SaveObject (Obj: TObject3D; theName:
    Str255; ID: integer);
procedure SaveNamedObject (Obj: TObject3D;
    theName: Str255; var ID: integer);
procedure FillTriangle (p1: Point; p2: point;
    p3: Point; theColor: Integer; useQD: Boolean);
function IsVisible (k, l, m: Vector4): Boolean;
```

3D GrafPort/Window Manipulation

```
function GetNew3DWindow (ID: integer; behind: ptr)
    : WindowPtr;
function New3DWindow (boundsRect: Rect; title:
    Str255; visible: BOOLEAN; procID: Integer;
    behind: WindowPtr; goAwayFlag: BOOLEAN; refCon:
    LongInt): WindowPtr;
procedure Dispos3DWindow (theWindow: WindowPtr);
procedure Set3DPort (the3DPort: WindowPtr);
procedure Get3DPort (var the3DPort: WindowPtr);
function Is3Dport (thePort: WindowPtr): Boolean;
procedure SetView (ProjectPlaneSize,
    ViewPlaneSize: Rect);
procedure SetCenter (x, y: Integer);
```

GrafSys Documentation

Eye Manipulation

```
procedure SetEye (UsesEye: Boolean; location:
    Vector4; thePhi, theTheta, thePitch,
    theViewangle: real; clipType: clippingType);
procedure GetEye (var UsesEye: Boolean;
    var location: Vector4; thePhi, theTheta,
    thePitch, theViewangle: real;
    var clipType: clippingType);
procedure ToScreen (thePoint: Vector4;
    var h, v: INTEGER);
procedure ProjectPoint (thePoint: Vector4;
    var h, v: integer);
```

Matrix/Vector Manipulation

```
procedure InitMatrix;
function Identity: Matrix4;
function MMult (var A, B: Matrix4): Matrix4;
function VMult (x: Vector4; var A: Matrix4)
    : Vector4;
function Transpose (A: Matrix4): Matrix4;
function VSub (x, y: Vector4): Vector4;
function VAdd (x, y: Vector4): Vector4;
```

Off-Screen Buffering

```
function AttachOffScreen (theWindow: WindowPtr;
    theColors: CTabHandle): integer;
function ChangeOffscreen (theWindow: WindowPtr;
    theColors: CTabHandle): integer;
function CloseOffscreen (theWindow: WindowPtr)
    : integer;
function BeginOSDraw (theWindow: WindowPtr)
    : integer;
function EndOSDraw (theWindow: WindowPtr): integer;
function CopyOS2Screen (theWindow: WindowPtr;
    theRect: Rect; copyMode: Integer): integer;
```

OOP Objects and Methods

GrafSys Documentation
TGenericObject

```
procedure Init;
```

GrafSys Documentation

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

TMatrixList

Inherited

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

Instanced

```
procedure Init;  
override;  
procedure Reset;  
override;  
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);
```

TMatrixInherit

Inherited

```
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);
```

Instanced

```
procedure Init;
```

GrafSys Documentation
override;

Inherited

```
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);
```

Instanced

```
procedure Init;  
override;
```

TAbstract3DObject

Inherited

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

Instanced

```
procedure Init;  
override;  
procedure Kill;  
override;  
procedure Reset;  
override;  
function Clone: TGenericObject;  
override;  
procedure Translate (dx, dy, dz: real);  
procedure SetTranslation (x, y, z: real);  
procedure Rotate (dx, dy, dz: real);  
procedure SetRotation (x, y, z: real);  
procedure Scale (dx, dy, dz: real);  
procedure SetScale (x, y, z: real);  
procedure RotArb (p, x: Vector4; phi: real);  
procedure ResetArb;
```

GrafSys Documentation

```
procedure FFTranslate (dx, dy, dz: real);  
procedure FFRotate (dx, dy, dz: real);  
procedure FFScale (dx, dy, dz: real);  
procedure FFRotArbAchsis (p, x: Vector4; phi: real);
```

GrafSys Documentation

```
procedure FFReset;
function FFNewPostConcat: TMatrixList;
function FFNewPreConcat: TMatrixList;
function FFActivate (theFF: TMatrixList) : boolean;
function FFPassOn: TMatrixPass;
procedure FFInherit (var FatherList: TMatrixPass);
procedure CalcTransform;
function ForeignPoint (p: Vector4): Vector4;
function WorldToModel(wc : Vector4) : Vector4;
procedure Draw;
```

TLine3D

Inherited

```
procedure HandleError;
procedure ResetError;
function Test (opcode: integer): integer;
function Clone: TGenericObject;
procedure Translate (dx, dy, dz: real);
procedure SetTranslation (x, y, z: real);
procedure Rotate (dx, dy, dz: real);
procedure SetRotation (x, y, z: real);
procedure Scale (dx, dy, dz: real);
procedure SetScale (x, y, z: real);
procedure RotArb (p, x: Vector4; phi: real);
procedure ResetArb;
procedure FFTranslate (dx, dy, dz: real);
procedure FFRotate (dx, dy, dz: real);
procedure FFScale (dx, dy, dz: real);
procedure FFRotArbAchsis (p, x: Vector4; phi: real);
procedure FFReset;
function FFNewPostConcat: TMatrixList;
function FFNewPreConcat: TMatrixList;
function FFActivate (theFF: TMatrixList): boolean;
function FFPassOn: TMatrixPass;
procedure FFInherit (var FatherList: TMatrixPass);
procedure CalcTransform;
function ForeignPoint (p: Vector4): Vector4;
function WorldToModel(wc : Vector4) : Vector4;
procedure Draw;
procedure Kill;
```


GrafSys Documentation

```
procedure Init;  
override;  
procedure Reset;  
override;  
procedure SetKoords (K1, K2: Vector4);  
procedure GetKoords (var K1, K2: Vector4);
```

TPoint3D

Inherited

```
procedure HandleError;  
procedure ResetError;  
function Test (opcode: integer): integer;  
function Clone: TGenericObject;  
procedure Translate (dx, dy, dz: real);  
procedure SetTranslation (x, y, z: real);  
procedure Rotate (dx, dy, dz: real);  
procedure SetRotation (x, y, z: real);  
procedure Scale (dx, dy, dz: real);  
procedure SetScale (x, y, z: real);  
procedure RotArb (p, x: Vector4; phi: real);  
procedure ResetArb;  
procedure FFTranslate (dx, dy, dz: real);  
procedure FFRotate (dx, dy, dz: real);  
procedure FFScale (dx, dy, dz: real);  
procedure FFRotArbAchsis (p, x: Vector4; phi: real);  
procedure FFReset;  
function FFNewPostConcat: TMatrixList;  
function FFNewPreConcat: TMatrixList;  
function FFActivate (theFF: TMatrixList): boolean;  
function FFPassOn: TMatrixPass;  
procedure FFInherit (var FatherList: TMatrixPass);  
procedure CalcTransform;  
function ForeignPoint (p: Vector4): Vector4;  
function WorldToModel(wc : Vector4) : Vector4;  
procedure Draw;  
procedure Kill;
```

Instanced

```
procedure Init;  
override;  
procedure Reset;
```


GrafSys Documentation

override;

procedure SetKoords (Koordinates: Vector4);

function GetKoords: Vector4;

TSGenericObject

Inherited

```
procedure HandleError;
procedure ResetError;
function Test (opcode: integer): integer;
procedure Translate (dx, dy, dz: real);
procedure SetTranslation (x, y, z: real);
procedure Rotate (dx, dy, dz: real);
procedure SetRotation (x, y, z: real);
procedure Scale (dx, dy, dz: real);
procedure SetScale (x, y, z: real);
procedure RotArb (p, x: Vector4; phi: real);
procedure ResetArb;
procedure FFTranslate (dx, dy, dz: real);
procedure FFRotate (dx, dy, dz: real);
procedure FFScale (dx, dy, dz: real);
procedure FFRotArbAchsis (p, x: Vector4; phi: real);
procedure FFReset;
function FFNewPostConcat: TMatrixList;
function FFNewPreConcat: TMatrixList;
function FFActivate (theFF: TMatrixList): boolean;
function FFPassOn: TMatrixPass;
procedure FFInherit (var FatherList: TMatrixPass);
procedure CalcTransform;
procedure Draw;
```

Instanced

```
procedure Init;
override;
function Clone: TGenericObject;
override;
procedure Reset;
override;
procedure Kill;
override;
procedure GenIndex (pointIndex: longint;
    var BufIndex, bufOffset: integer);
function AddPoint (x, y, z: real): longint;
function DeletePoint (index: longint): boolean;
procedure GetPoint (index: longint;
```

GrafSys Documentation

```
    var x, y, z: real);  
function ChangePoint (index: longint;  
    x, y, z: real): boolean;
```

GrafSys Documentation

```
procedure Transform (forceCalc: boolean);
procedure Transform2 (forceCalc: boolean);
function TransformedPoint (index: longint): Vector4;
procedure CalcBounds;
function ForeignPoint (p: Vector4): Vector4;
override
function WorldToModel(wc : Vector4) : Vector4;
override
```

TSLine3D

Inherited

```
procedure HandleError;
procedure ResetError;
function Test (opcode: integer): integer;
function Clone: TGenericObject;
procedure Translate (dx, dy, dz: real);
procedure SetTranslation (x, y, z: real);
procedure Rotate (dx, dy, dz: real);
procedure SetRotation (x, y, z: real);
procedure Scale (dx, dy, dz: real);
procedure SetScale (x, y, z: real);
procedure RotArb (p, x: Vector4; phi: real);
procedure ResetArb;
procedure FFTranslate (dx, dy, dz: real);
procedure FFRotate (dx, dy, dz: real);
procedure FFScale (dx, dy, dz: real);
procedure FFRotArbAchsis (p, x: Vector4; phi: real);
procedure FFReset;
function FFNewPostConcat: TMatrixList;
function FFNewPreConcat: TMatrixList;
function FFActivate (theFF: TMatrixList): boolean;
function FFPassOn: TMatrixPass;
procedure FFInherit (var FatherList: TMatrixPass);
procedure CalcTransform;
function ForeignPoint (p: Vector4): Vector4;
function WorldToModel(wc : Vector4) : Vector4;
procedure Kill;
procedure Init;
procedure Reset;
procedure SetKoords (Koordinates: Vector4);
function GetKoords: Vector4;
```

Instanced

GrafSys Documentation

```
procedure Draw;  
override;
```

TSPoint3D

Inherited

```
procedure HandleError;  
procedure ResetError;  
function Test (opcode: integer): integer;  
function Clone: TGenericObject;  
procedure Translate (dx, dy, dz: real);  
procedure SetTranslation (x, y, z: real);  
procedure Rotate (dx, dy, dz: real);  
procedure SetRotation (x, y, z: real);  
procedure Scale (dx, dy, dz: real);  
procedure SetScale (x, y, z: real);  
procedure RotArb (p, x: Vector4; phi: real);  
procedure ResetArb;  
procedure FFTranslate (dx, dy, dz: real);  
procedure FFRotate (dx, dy, dz: real);  
procedure FFScale (dx, dy, dz: real);  
procedure FFRotArbAchsis (p, x: Vector4; phi: real);  
procedure FFReset;  
function FFNewPostConcat: TMatrixList;  
function FFNewPreConcat: TMatrixList;  
function FFActivate (theFF: TMatrixList): boolean;  
function FFPassOn: TMatrixPass;  
procedure FFInherit (var FatherList: TMatrixPass);  
procedure CalcTransform;  
function ForeignPoint (p: Vector4): Vector4;  
function WorldToModel (wc : Vector4) : Vector4;  
procedure Kill;  
procedure Init;  
procedure SetKoords (Koordinates: Vector4);  
function GetKoords: Vector4;
```

Instanced

```
procedure Init;  
override  
procedure Reset;  
override;  
procedure Draw;
```

GrafSys Documentation
override;

Inherited

```
procedure HandleError;  
procedure ResetError;  
function Test (opcode: integer): integer;  
procedure Translate (dx, dy, dz: real);  
procedure SetTranslation (x, y, z: real);  
procedure Rotate (dx, dy, dz: real);  
procedure SetRotation (x, y, z: real);  
procedure Scale (dx, dy, dz: real);  
procedure SetScale (x, y, z: real);  
procedure RotArb (p, x: Vector4; phi: real);  
procedure ResetArb;  
procedure FFTranslate (dx, dy, dz: real);  
procedure FFRotate (dx, dy, dz: real);  
procedure FFScale (dx, dy, dz: real);  
procedure FFRotArbAchsIs (p, x: Vector4; phi: real);  
procedure FFReset;  
function FFNewPostConcat: TMatrixList;  
function FFNewPreConcat: TMatrixList;  
function FFActivate (theFF: TMatrixList): boolean;  
function FFPassOn: TMatrixPass;  
procedure FFInherit (var FatherList: TMatrixPass);  
procedure CalcTransform;  
function ForeignPoint (p: Vector4): Vector4;  
function WorldToModel(wc : Vector4) : Vector4;  
procedure SetKoords (Koordinates: Vector4);  
function GetKoords: Vector4;  
procedure GenIndex (pointIndex: longint;  
    var BufIndex, bufOffset: integer);  
function AddPoint (x, y, z: real): longint;  
procedure GetPoint (index: longint;  
    var x, y, z: real);  
function ChangePoint (index: longint;  
    x, y, z: real): boolean;  
procedure Transform (forceCalc: boolean);  
procedure Transform2 (forceCalc: boolean);  
function TransformedPoint (index: longint): Vector4;  
procedure CalcBounds;  
function ForeignPoint (p: Vector4): Vector4;  
function WorldToModel(wc : Vector4) : Vector4;
```

Instanced

GrafSys Documentation
procedure Init;

GrafSys Documentation

```
override;
function Clone: TGenericObject;
override;
procedure Reset;
override;
procedure Kill;
override;
function AddLine (fIndex, tIndex: longint)
    : integer;
function ChangeLine (LineIndex, fIndex, tIndex
    : longint): boolean;
function ChangeLineColor (LineIndex: longint;
    theColor: RGBColor): boolean;
function GetLineColor (LineIndex: longint;
    var theColor: RGBColor;
    var ChangeHere: boolean): Boolean;
function KeepLineColor (LineIndex: longint)
    : boolean;
function DeleteLine (LineIndex: integer)
    : Boolean;
function DeletePoint (index: longint): boolean;
override;
procedure GetLine (lineIndex: integer;
    var src, tgt: LongInt);
procedure BuildNewLines;
procedure CollectLineData;
procedure SetAutoerase (TurnOn: Boolean);
procedure SetUseBounds (TurnOn: Boolean);
procedure Draw;
override;
procedure fDraw;
procedure Erase;
```

Appendix B

Look-Up Table Performance Tests

GrafSys Documentation

Look-Up Table Performance Tests

The performance tests were done using the three supplied TimeTrig programs. Each program called 100'000 times the look-up Sine, Cosine and Tangents functions and compared the results with the time it took the system to perform the same task using either SANE, direct 030 code and 030 code plus direct FPU calls.

A performance factor greater than one means that the look-up tables are faster by that factor.

Macintosh SE, System 7.1, no FPU, no extensions
(68000 Processor, 8 MHz)

Test Program: 'Test Trig plain'

Test	System	HiPerf Trigs	Performance Factor
Sin	89603	23191	3.86
Cos	81815	23186	3.53
Tan	105345	21704	4.85

Macintosh LC, System 7.1, no FPU, no extensions
(68020 Processor, 16 MHz)

Test Program: 'Test Trig plain'

Test	System	HiPerf Trigs	Performance Factor
Sin	17913	6312	2.84
Cos	15912	6324	2.52
Tan	19908	5935	3.35

Test Program: 'Test Trig 030'

Test	System	HiPerf Trigs	Performance Factor
Sin	17908	6111	2.93
Cos	15906	6079	2.62
Tan	19904	5731	3.48

GrafSys Documentation

Macintosh IIcx, System 7.1, FPU, no extensions
(68030 Processor, 16 MHz)

Test Program: 'Test Trig plain'

Test	System	HiPerf Trigs	Performance Factor
Sin	9174	4577	2.00
Cos	7679	4575	1.68
Tan	10545	4314	2.44

Test Program: 'Test Trig 030'

Test	System	HiPerf Trigs	Performance Factor
Sin	8859	4460	1.99
Cos	7428	4465	1.66
Tan	10161	4347	2.34

Test Program 'Test Trig 030/881'

Test	System	HiPerf Trigs	Performance Factor
Sin	176	315	0.56
Cos	176	325	0.54
Tan	204	323	0.63

GrafSys Documentation

Macintosh Centris 650, System 7.1, FPU on-Chip, no extensions
(68040 Processor, 25 MHz)

Test Program: 'Test Trig plain'

Test	System	HiPerf Trigs	Performance Factor
Sin	328	260	1.26
Cos	315	259	1.22
Tan	346	270	1.28

Test Program: 'Test Trig 030'

Test	System	HiPerf Trigs	Performance Factor
Sin	329	243	1.35
Cos	316	242	1.31
Tan	347	256	1.36

Test Program 'Test Trig 030/881'

Test	System	HiPerf Trigs	Performance Factor
Sin	147	196	0.75
Cos	145	196	0.74
Tan	148	203	0.73

Appendix C

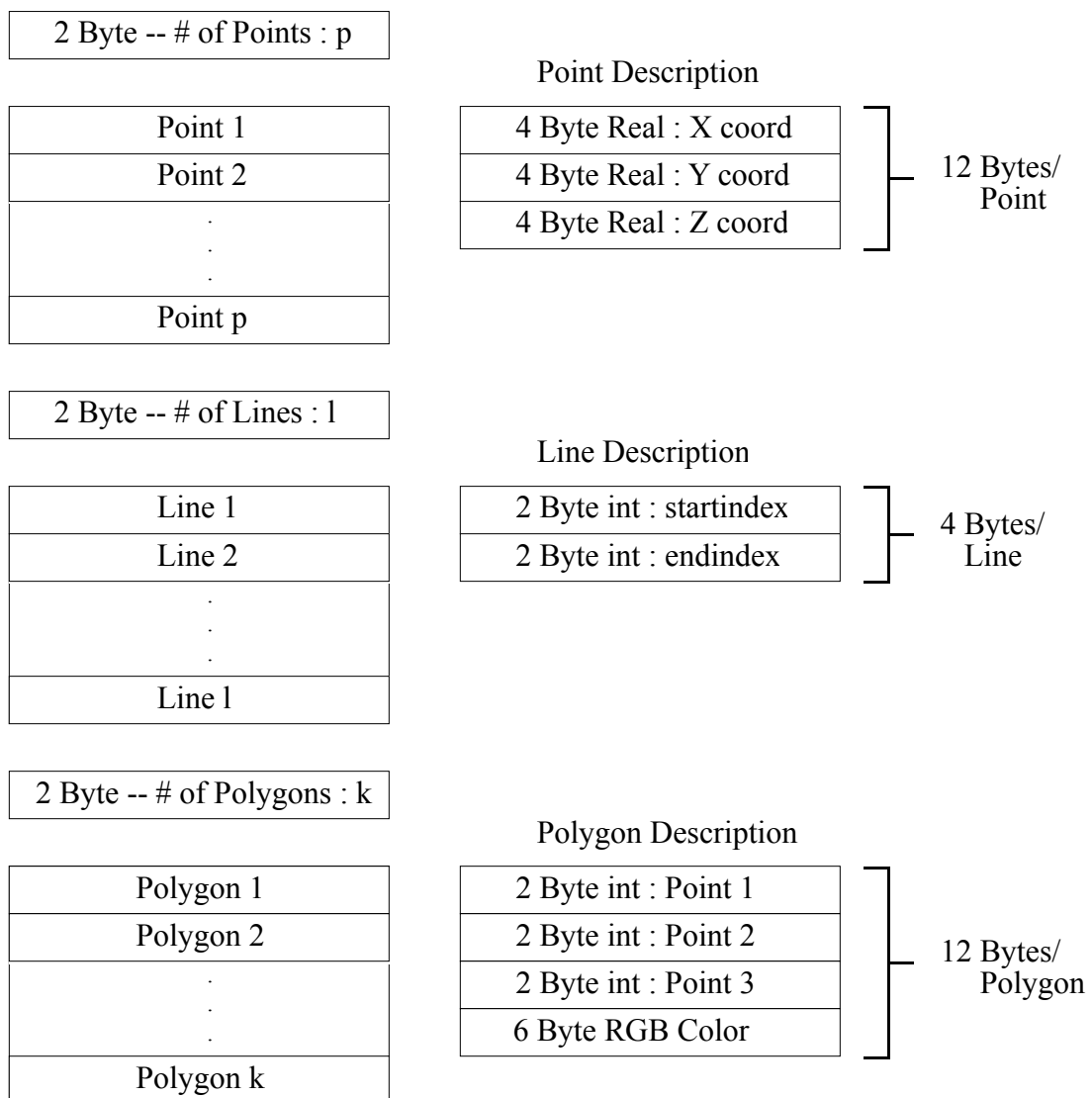
Resource Formats

GrafSys Documentation

3Dob Resource Format

GrafSys provides commands to save and get object descriptions into and from resources, respectively. The resource type used is '3Dob'. I strongly encourage accessing and saving resources by name, since it is much easier to understand. And yes, I know that all you C enthusiasts (what an euphemism for 'obnoxious little freak...' 😊) frown on this and you think that 'everything is integer' but I **am** in favor of speaking names etc. (and I do certainly **not** adhere to 'if it was hard to write, it should be hard to read').

But enough of this. Here's the description of the '3Dob' resource:

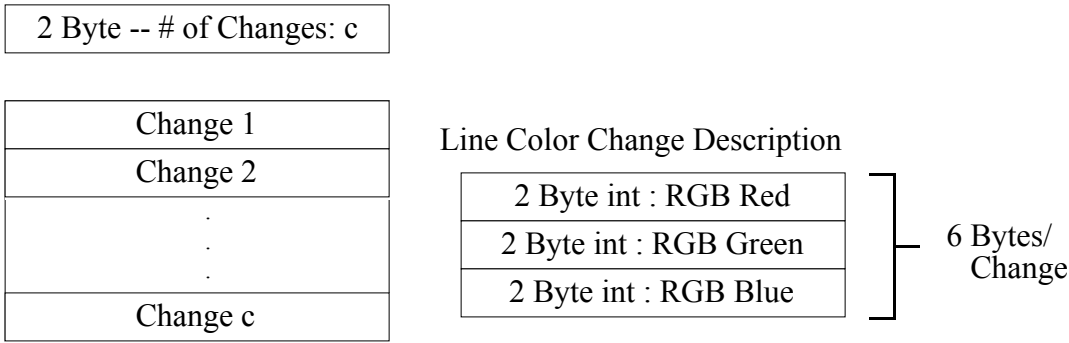


Therefore the smallest possible 3Dob resource is 6 bytes :

- 00 (2 byte zero Point count)
- 00 (2 byte zero Line count)
- 00 (2 byte zero Polygon count)

IClr Resource Format

The IClr resource must have the same ID as the corresponding 3Dob resource or it will not be loaded by the GrafSys.



GrafSys Documentation
Literature

- [Brig92] Brigham Stevens/Bill Gushwan, Graphical Truffles, "develop", August 1992
- [Brügg91] Adrian Brügger, Schichtenmodelle in der XYZ-GeoBench, Diplomarbeit, 1991
- [Foley90] Foley/vanDam/Feiner/Hughes, Computer Graphics, Second Edition, Addison Wesley, 1990
- [Hearn86] Donald Hearn/M. Pauline Baker, Computer Graphics, Prentice-Hall International, 1986
- [Hornu89] Christoph Hornung/Josef Pöpsel, 3-D à la carte, c't 4/1989, S. 240ff
- [InsideMac] Inside Macintosh Vol I, II, IV, V, VI, Addison Wesley
- [Kelly88] Stan Kelly-Bootle/Bob Fowler, 68000 Primer, The Waite Group, First Edition, 1988
- [Mark89] Dave Mark, Macintosh Programming Primer Volume II, Addison-Wesley, 1st Edition 1989
- [Mark90] Dave Mark, Macintosh Programming Primer Volume I, Addison-Wesley, 3rd Edition 1990
- [MBUGRef] MacsBug Reference and Debugging Guide for MacsBug Version 6.2, Addison-Wesley, 1991
- [Moess91] H. Moessenboeck, 'Compilerbau', Handschriftliches Skriptum zur Vorlesung im SS91 an der ETH Zürich, VIS-Archiv
- [MPWRef] MPW 3.0 Reference, Apple Computer 1992
- [Pavli82] Theo Pavlidis, Algorithms for Graphics and Image Processing, Computer Science Press, 1982
- [ResEdRef] ResEdit Reference for ResEdit version 2.1, Addison-Wesley, 1991
- [Sphar91] Chuck Sphar, Object-Oriented Programming Power, Microsoft Press, 1st Edition 1991

GrafSys Documentation

- [Spitz93] Daryl Spitzer (Spitzer@mindlink.bc.ca), "Frequently Asked Questions List for Comp.Sys.Mac.Programmer", January 12, 1993, Part I and II, available via anonymous ftp at mac.archive.umich.edu and sumex-aim.stanford.edu.
- [Stamm89] Beat Stamm, Algorithms for Drawing Thick Lines and Curves on Raster Devices, ETH 107, May 1989
- [Tanen87] Andrew S. Tanenbaum, Operating Systems, Prentice-Hall International, 1st Edition 1987
- [ThCRef] THINK C User manual, Symantec Corp, 1991
- [ThPasRef] THINK Pascal 4.0 Manual, Symantec Corp. 1991
- [TNxxx] Macintosh Technical Notes, Apple Computer.
#021: Quickdraw's Internal Picture Definition, March 1988
#046: Seperate Resource Files, March 1988
#059: Pictures and Clip Regions, March 1988
#078: Resource Manager Tips, March 1988
#091: Optimizing for the Laser Writer, March 1988
#101: CreateResFile and the Poor Man's Search Path, March 1988
#117: Compatability: Why and How, March 1988
#120: Drawing Into Off-Screen Pixel Map, April 1989
#163: Adding Color with CopyBits, March 1988
#181: Every Picture Comment..., March 1988
#193: So Many Bitmaps, So Little Time, December 1989
#211: Palette Manager Changes In System 6.0.2, October 1, 1988
#214: New Resource Manager Calls, October 1, 1988
#257: Slot Interrupt Prio-Techniques, October 1989
#297: Pictures And The Printing Manager, April 1991
OSB Off-Screen Bitmaps, June 1990
POSE Principa Off-Screen Graphics Environments, March 1992
QDQA Basic Quickdraw Q&As, May 1993
RBR2 RowBytes Revealed II, May 1993
TS_C Time and Space and _CopyBits, June 1990
- [Wirth89] Niklaus Wirth, "Drawing Lines, Circles, and Ellipses in a Raster", ETH 117, Oktober 1989

Index

GrafSys Documentation

Index