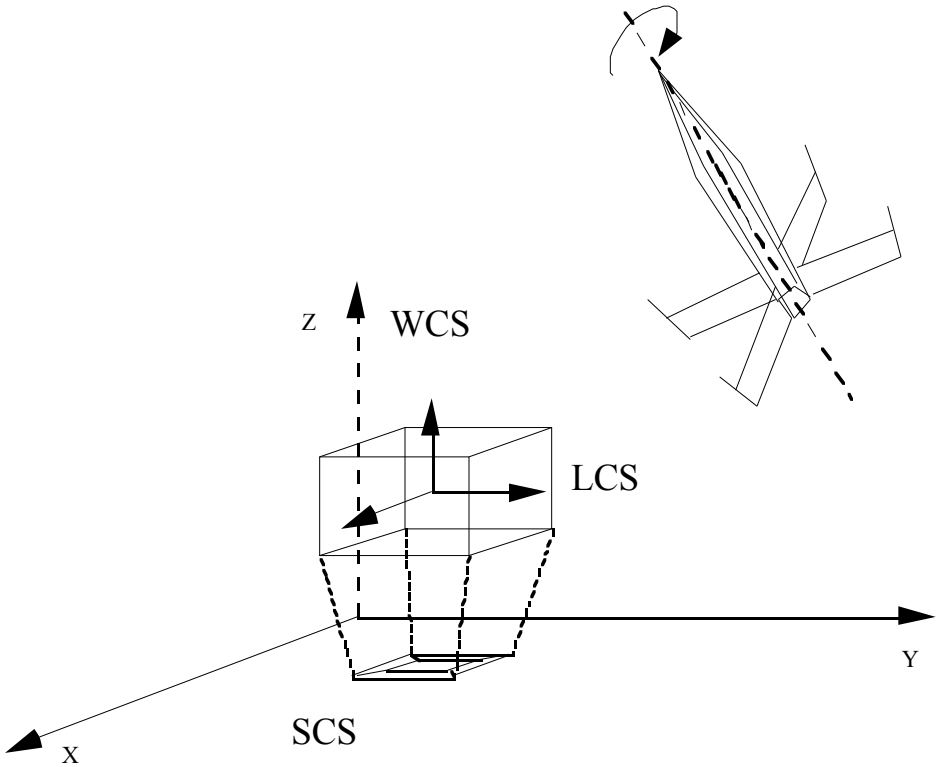
Christian Franz

3D GrafSys

Version 1.1

for programmers



Notice:

My address is

Christian Franz
Sonneggstrasse 61
CH-8006 Zurich

Swizerland

email cfranz@iiic.ethz.ch
tel. +1-261 26 96 (+ = your code for Swizerland)

If you have any questions or bug reports or would like to see other features implemented, please feel free to contact me at above address.

Note: As you will notice throughout the documentation, English is not my primary language. There are bound to be many mistakes. If you find some, please take the time to write them down and (e)mail them to me so I can correct them.

What it is

Didn't you always have this great game in mind where you needed some way of drawing three-dimensional scenes?

Didn't you always wanted to write this program that visualized the structure of three-dimensional molecules?

And didn't the task of writing your 3D conversions routines keep you from actually doing it?

Well if the answer to any of the above questions is 'Yes, but what has it to do with this package???' , read on.

GrafSys is a THINK Pascal/C library that provides you with simple routines for building, saving and loading (as resources), and manipulating (independent rotating around arbitrary achses, translating and scaling) three dimensional objects. **Objects**, not just simple single-line drawings.

GrafSys supports full 3D clipping, animation and some (primitive) hidden-line/hidden-surface drawing with simple commands from within your program.

GrafSys also supports full eye control with both perspective and parallel projections (If you can't understand a word, don't worry, this is just showing off for those who know about it. The docs try to explain what it all means later on).

GrafSys provides a powerful interface to supply your own drawing routines with data so you can use GrafSys to do the 3D transformations and your own routines to do the actual drawing. (Note that GrafSys also provides drawing routines so you don't have to worry about that if you don't want to)

GrafSys was compiled with the direct 68881/2 option set, so you must have a machine equipped with a mathCo or the software won't run. I do know this is a drawback, but since I couldn't get the [expletion deleted] `fixed` type to work, I chose to use the mathCo.

If demand is big enough I will convert the GrafSys to an object-class library. However, I feelt that the way it is implemented now makes it easier to use for a lot more people than the select 'OOP-Guild'.

Overview

The 3D Graphics Package is a set of routines that will allow you easily incorporate 3D Graphics and animations into your programs. It supports hidden-line removal (in experimental stage), full clipping, perspective viewing, independent rotation of objects, even along arbitrary achses. The programmer has full control over perspective, eye location etc.

To enable more advanced programmers to implement or support their own optimized drawing environments, the package also sports a low-level interface where all relevant data for drawing the object can be obtained.

This document will try to give first an overview over the fundamental aspects of three-dimensional graphics and how they are implemented in this package and then show you how to use it in your own programs.

How to use GrafSys

To use the GrafSys, include the file GrafSys.lib and GrafSys.Int into your project.

If you plan on using the provided screen drawing routines, you will also have to include the files Screen3D.lib and Screen3D.int into your project.

Since there are now different versions of the GrafSys, refer to Chapter '881 versus Fixed Point Arithmetic' on how to use the different versions.

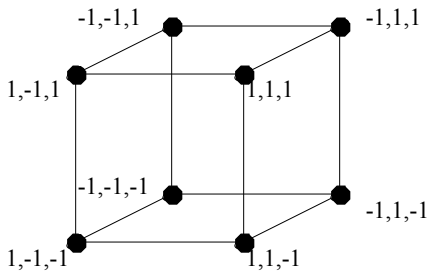
Fundamentals

Point, Line, Polygon, Object

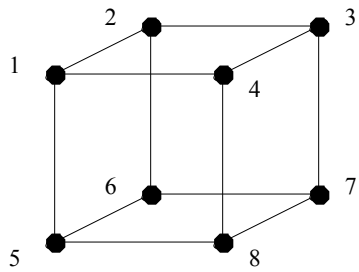
Almost all 3D Graphics with this package should be done with so-called Objects. Although the package supports separate conversion and drawing of 3D lines as well, it is optimized to handle '3D Objects'. These objects are usually a collection of Points, Lines and Polygons that logically belong together. If you group all this data into one single object, drawing and transforming becomes a simple task and requires no additional headhache (or sore fingers while programming) from you. To make you familiar with the concept of objects, lets start with a simple one - a cube:

Point

It should be no news to you that a cube has eight corners (if it is news to you, maybe you should throw away those fancy role-playing game dice and get a normal six-sided (1d6) dice and count...). You guessed it, these corners define the cube. To begin building an object, we start by adding these points with the `AddPoint` to it. Note that the Object should have been allocated previously using the `NewObject` procedure. As you will notice later, the order in which you add the Points *does* make a difference, so be sure to number them correctly.



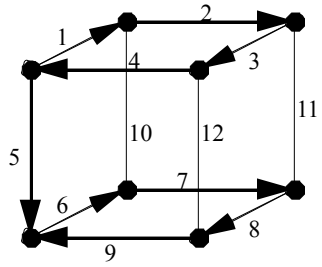
Cube's Coordinates



Cube's Point Numbering

Line

Now that we have entered all the points this object requires, we can start on the lines that connect the points. Note that in this package you will only see lines (or surfaces) but never single points. In the drawing you can see a total of twelve lines. These lines also have to be added to the object, so the routines know what to draw. Again, the efficiency of your objects heavily depends on how good you organize your lines. What you should do is try to find a way to connects as many points as possible without lifting your pen. The more times you have to interrupt your drawing, the more the routines have to calculate the new beginning points of your lines



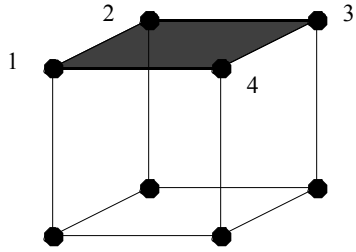
In above example you can see that we can draw continously nine of the twelve lines, only lines 10, 11 and 12 have to be drawn separately. Lines are added to an Object using the `AddLine` procedure. The package automatically tests if a previous line connects the new one, so you don't have to worry about setting any parameters for beginnings and endings of lines. Note that this also applies if you later use the (more advanced) `InsertLine` and `DeleteLine` procedures. If all you are using are wireframe objects (that is objects that do not use hidden line removal or the like) you are done generating the Object and you can skip to the paragraph 'Viewing an Object'.

Ploygon

Polygons are only needed in hidden-line/hidden-surface drawing.

Sometimes there is more you want than just wireframe models. Since real objects normally contain surfaces, this package provides a simple hidden-line/hidden-surface removal strategy. Note that this is still in experimental stage and both performance and results are not too great.

To build a surface you group previously defined lines into a polygon. in our example, the cube has six sides and thus our object will contain six polygons. A polygon is defined quite similar to lines, you simply list all the points *in the correct order* that make out a polygon. A polygon contains a maximum of 10 points. From the last point, the package automatically draws back to the first point thus closing the polygon. The plygon for the top surface would thus be a call to `SetPoly` with the Points 1,2,3 and 4 (filling the rest of the parameters with zeros).



Then calling `AddPoly` with the just generated polygon will add it to the object. Repeat this with all the other sides of the cube and you have defined all surfaces. The order in which you define the surfaces is of no importance for a change.I know that this is still way too cumbersome. If anyone out there has some better ideas I would like to hear them.

Object

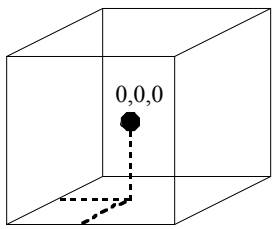
After you described the 3D image, the object is all you need to pass the routines to transform and view it. Before you can add any lines, points or polygons to an object you have to create it (i.e. allocate memory for it). Do this with the `NewObject` and `GetNewObject` procedures. Usually you would use the `GetNewObject` to load previously defined objects and never build them in a program using the `AddPoint`, `AddLine` or `AddPoly` routines. Keep in mind however, that the exist and you can use them to change an object on the fly.

Viewing An Object

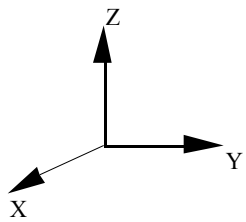
Before we can view an object we just created, there are a few (view) things that you should be aware of. In this section I will try to explain such arcane things as 'world coordinates' and 'object coordinates' and how they tie together. If you kow about this stuff already, I suggest you just skim the paragraphs and look if my definitions match yours to avoid confusion.

object coordinates and world 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*. We did just this when we created the cube object. The origin of the cube is in it's center as can easily be seen (if you have problems following me, look into the '**Techniques for designing an object**' section and try locating the point [0,0,0] within the cube).



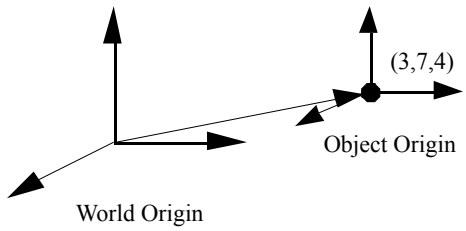
Origin Of The Cube



The Coordinate System

When you design an object, you specify all points in the object's coordinate system. Then, when viewing it, you place the object somewhere in the world. You do this by specifying which point in the

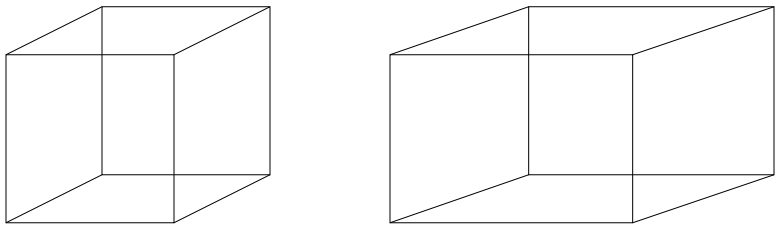
world would correspond to your objects origin. Got it? Well I didn't the first time, so here is yet another figure to forget:



In the figure, the origin of the object was placed at the world coordinates [3,7,4]. Normally you would now have to recalculate all your points. Luckily, this is what the package is for and it does it automatically for you. It does even more as you will soon find out.

scaling

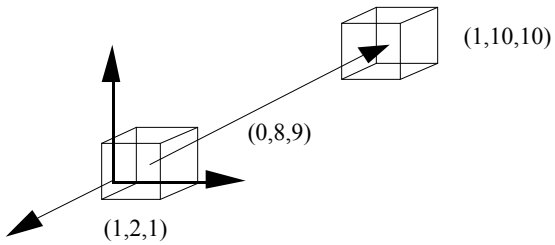
The first thing you can do is that you can scale an object. This means nothing but enlarging or reducing the object along any of its achses. Use the procedure `ObjScale` to do this.



Normal Cube Cube Sclaed Along Y-Achsis

translating

Moving an object around in the world (i.e. moving the object origin) is called translating. A fancy name for something relatively dull. You translate an object by giving displacements (i.e. how many units in direction of x, y and z).



In above figure, the cube's origin at (1,2,1) was moved to (1,10,10) by passing a displacement of (0,8,9). Translate an object using the ObjTranslate procedure.

Keep in mind that these displacements are given in world coordinates and

are unaffected by any scaling or orientation of an object. To be

specific, if you turned an object for 90 degrees around its Z-achsis, a X-displacement will *not* become a Y-displacement.

rotating

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. This package is not different. First, the object is rotated around the X-, then Y- and finally Z-Achsis. If you keep this in mind, you should not be surprised too often.
In addition to normal object rotation (also called local rotation), the package supports global rotation. The difference is that while a normal roation will rotate all the objects points around its origin, a global rotate will rotate the object around the world's origin.
A rotation is given in increments, i.e. how many **radiants** (not degrees!) you want to turn the object *further* around the corresponding achses.

You can also set the rotation of the object to absolute values using the `SetObjectRot` procedure and get the current rotation values by calling the `GetObjectRot` function.

Note: to convert between radiants and degrees, use the following:

```
const
        degree = 0.01745329;
```

and multiply all your angles (given in degrees) with the constant. This will convert it to radiants, e.g:

```
ObjRotate(myObject, 45*degree, 15*degree, phd*degree);
```

free rotate

Since it is not enough to rotate an object along it's three main achses, the package supports rotation around an arbitrary achsis (both local and global). For this, you specify two points and an increment (in radiants). The achsis of rotation runs through the two points. Here it is important that you pass the points in the correct order, otherwise the rotation will be into the opposite direction.

the eye

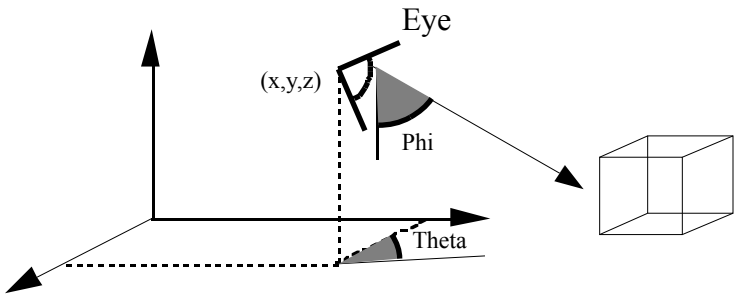
There are two different sets of parameters that specify the 'eye' or the point from which you look at the world. If you look at an Object, the way it is displayed on the screen depends on several aspects:

- from which direction you look at it

• how far away you are from the object

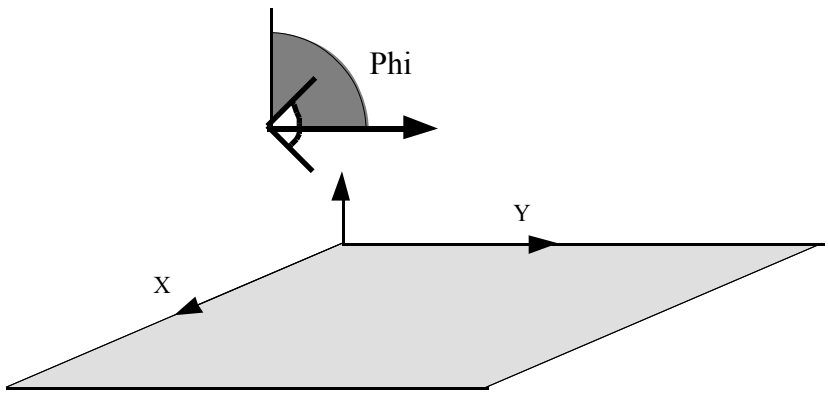
• what projection type you are using
• what kind of electronic lens you have selected

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).

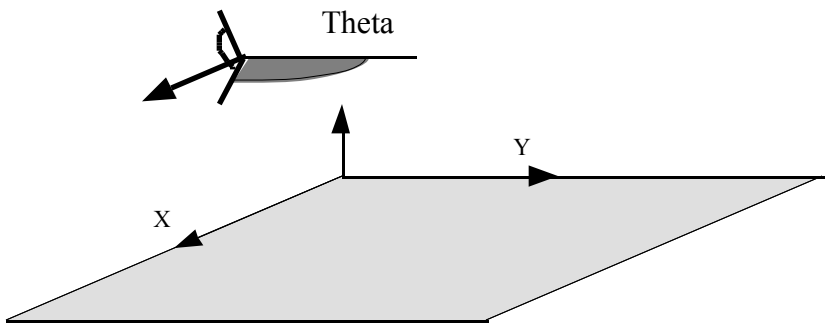


An Eye is defined by a point and three angles

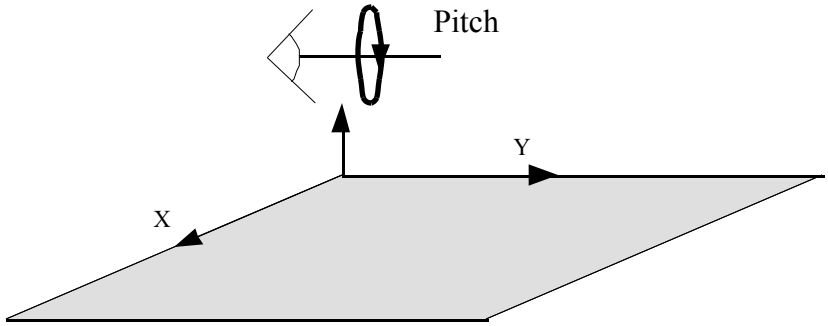
After defining the Point where the camera is set up, you specify how much the camera deviates from to Z-achsis towards the Y-achsis. This angle is called Phi. If you specify a Phi angle of zero, the camera would be facing straight down the Z-achsis, an angle of 90 degrees (remember to convert to radiants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



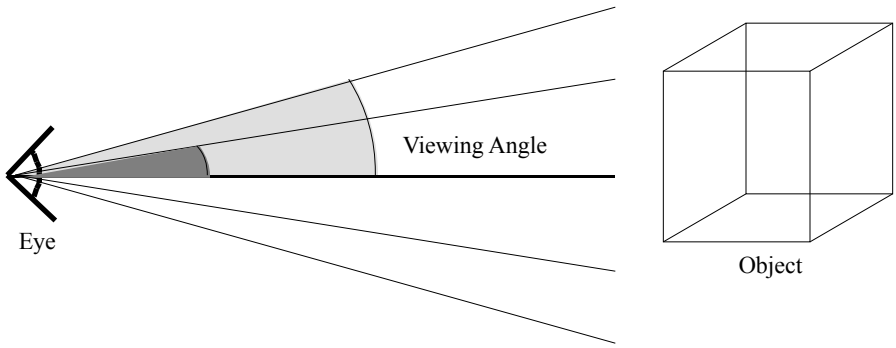
Next, with Theta, you tell the package how far you would like to turn the camera around the Z-Achsis:



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')

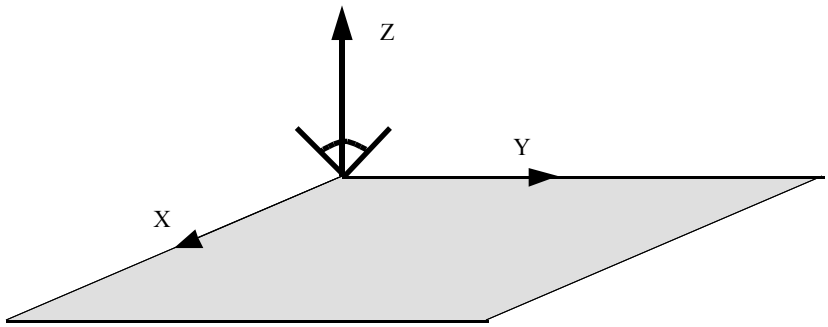


The last parameter affects your graphics only if you have are using perspective projection. It is called 'Viewing Angle' and simulates the electronic lens. If you use small angles, your eye shows only a very small part of the world but enlages it manyfold. This would be a 'Zoom Lens'. Large angles show a much bigger portion of the world, but these will be smaller and you have to get closer to enlarge them (but hey, this is just like in real life).



A viewing angle of zero tells the package that you want to switch to parallel projection (see below)

IMPORTANT: I know this will confuse you, especially since the pictures imply otherwise. Anyway, if you set the eye to reside in the world origin and set all angles to zero, the eye is looking - as I said befor straight down the z-achsis. This means that actually the eye is looking in the positive direction of the Z-achsis:

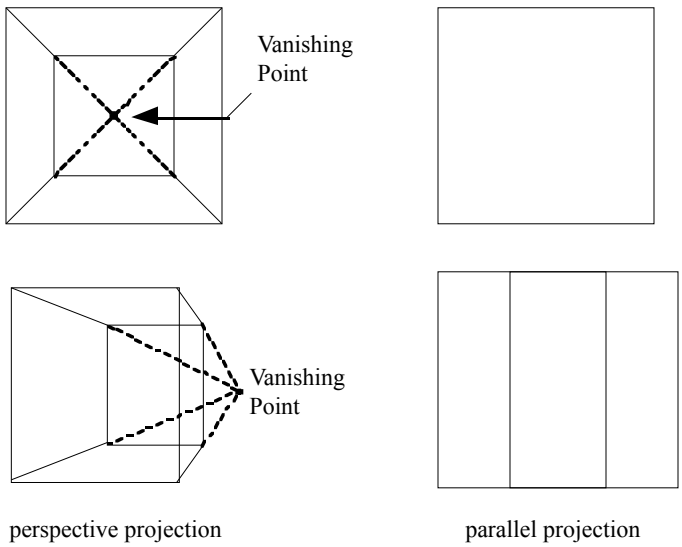


The eye is located immediately behind the point you specified, so anything in front of it will be displayed normally, the rest will be displayed either mirrored (if clipping is off) or not at all (if clipping is on). Clipping so far only works in conjunction with the 'Screen Objects' (see way down below).

viewing options

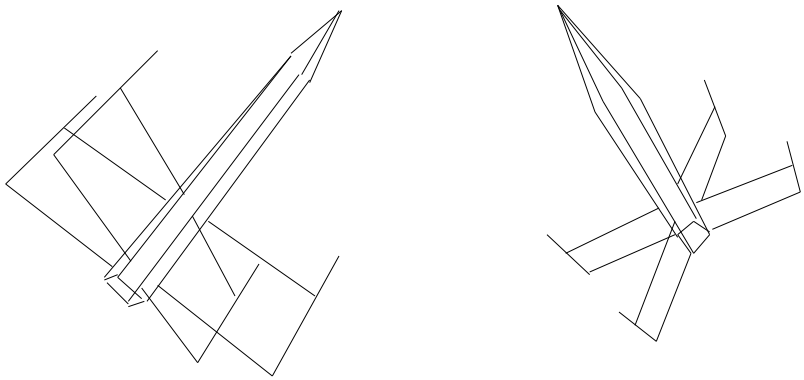
The package supports two ways of drawing the objects: parallel and perspective. The difference is instantly clear. In perspective projection, things that are further away are smaller than those closer to the eye. In parallel projection, all lines on the screen remain the same length, regardless how far 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 don't stay parallel. In parallel projection, parallel lines stay parallel.



The same cube, once with perspective and once with parallel projection

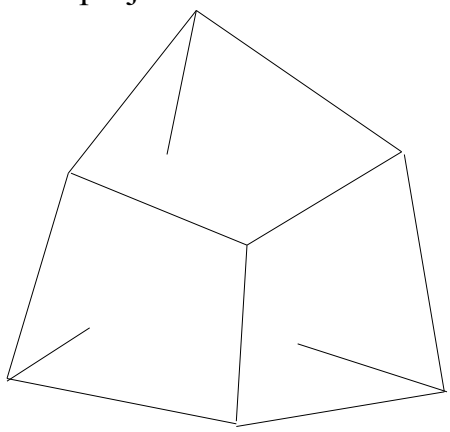
In above example you can very easily see that perspective projection is the way you are used to in normal pictures while parallel projection you probably know from floor plans or construction sheets. To turn on perspective projection, pass a viewing angle that is unequal to zero. To turn on parallel projection, pass a viewing angle of zero.



two perspective projections of the same object

Sometimes it might be useful to have a fixed camera location. In this case you can turn of the eye transformations. The eye will be fixed at location (0,0,0) and look straight down the Z-achsis. 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. To turn off the eye transformations, pass FALSE to the `UseEye` parameter.

An additional parameter, 'Clipping', can be set. Clipping is techno-speak for eliminating those lines of a graphic, that 'fall off' the screen. More precise, it is eliminating those parts of a line, that fall off. It is very important to clip those lines that fall behind the eye or very close to it, since they behave verry erratically there (try looking at your finger at about 0.2 inches from *your* eye and you will understand). However, so far clipping is only supportet with the use of screen objects that will be discussed later. Clipping is only useful in perspective projection.



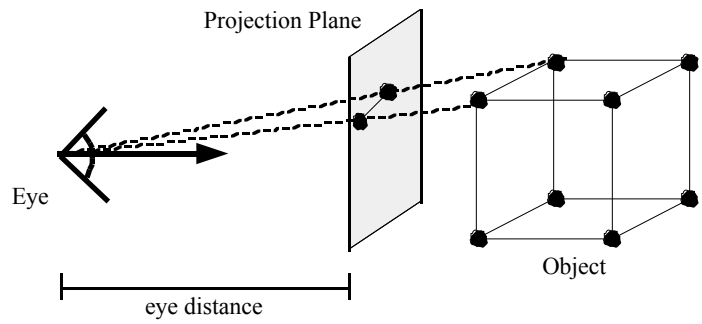
In above example, the (perspective projection of the) upper corner of the cube has been clipped because it came too close to the eye location.

All above mentioned parameters are set with the `SetEye` routine.

how the eye works

Well, to make some things clear that maybe havn't gotten across, lets make it first a bit more complicated. When I was talking about the eye, I was actually talking about the projection plane. But before you give up, let me tell you, that it is really nothing to worry about. You see, the eye is really just a tiny point and if we projected everything into the eye, you would end up with just a single dot and nothing else.

Instead, if you specify the location (and orientation) of the projection plane, you 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).



When you defined the viewing angle, the package used 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'). Why this, you might ask. The answer is very easy. This way, if you resize the projection plane (i.e. on a smaller monitor) the eye distance gets recalculatet and the scene is scaled to fit into the new projection plane. In other words, no matter how big or small your screen (or projection plane), the same scene fits on it if you use the same view angle. Note that this is only true for perspective projections.

Since the difference between eye and projection plane is only of technical interest, I will use the word eye where I should have used projection plane. Especially because eye only has three letters.

Point transformation

As you by now probably have figured 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.

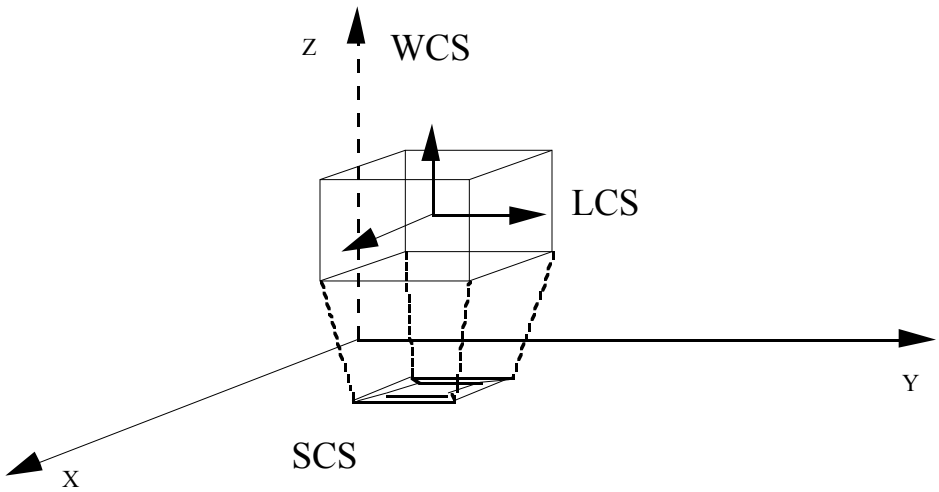
Anyway, to give you a better understanding on what goes on behind the screen, read on (you may skip the next paragraph if you are easily bored).

Coordinate Systems

Things are really getting confusing now. If you define an object, you define all the points in a coordinate system that is special to this and only this object. This coordinate system we will call the Local Coordinate System (LCS). Now, if we transform the object (rotate, translate or scale it), the objects points get changed to other position. However, since all points within the object remain in the same position to each other, we say that the LCS gets transformed.

The new locations of the various points are transformed according to your translation, rotation and scaling settings into a new coordinate system called the World Coordinate System (WCS).

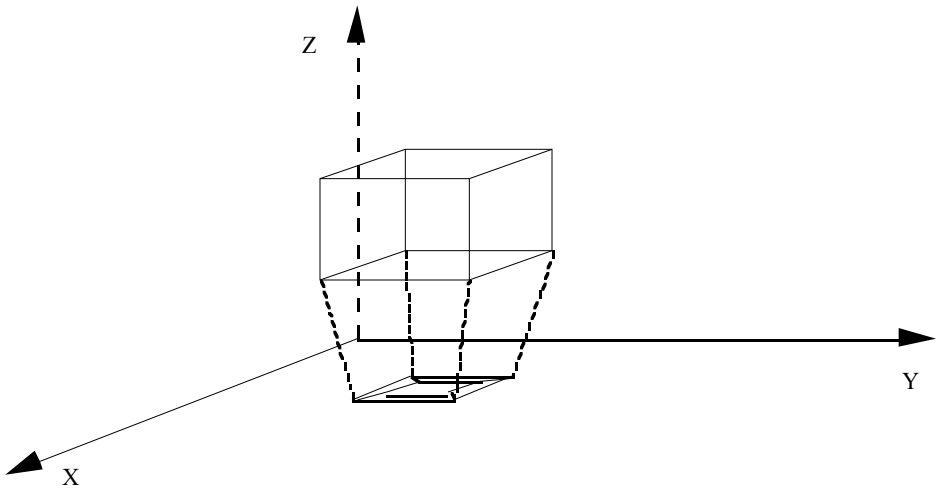
After they are transformed, the points get projected onto the screen. These (now two-dimensional) points reside in the Screen Coordinate System (SCS).



The graphic package supports all different coordinate systems. With the GetPoint routine you access the LCS of the object, with the ObjPoint and ObjPointArb you have points from the WCS and the ToScreen command converts points from WCS to the SCS. NOte that the ObjPoint and ObjPointArb do *not* use the Eye settings (i.e. you must use the ToScreen command to include eye settings after ObjPoint).

Eye Coordinate System

There is a little difficulty that I have to explain. Although it seems that the WCS is the definite coordinate system before the points appear on the screen, this is not true. If you are using the Eye, the points get transformed yet another time into the Eye Coordinate System. This is very important to remember.



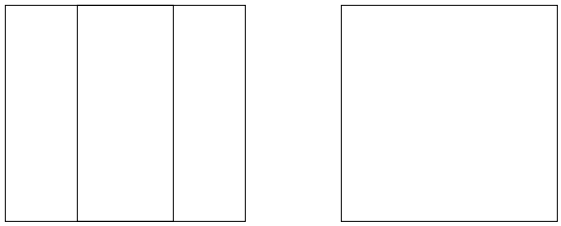
Points are alway projected onto the XY-plane

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-achsis. If we are using the eye, the points from the WCS are transformed again according to the eye settings.
However, whenever you request transformations to WCS, you will automatically receive ECS if you are using the eye.

Care must be taken if you are using the ToScreen command. Here you can easily distinguish between WCS and ECS. If you are using the eye, the point will first be transformed from WCS to ECS and then projected on the screen.
Once again, keep in mind, that all points are projected onto the XY plane after being converted to ECS.

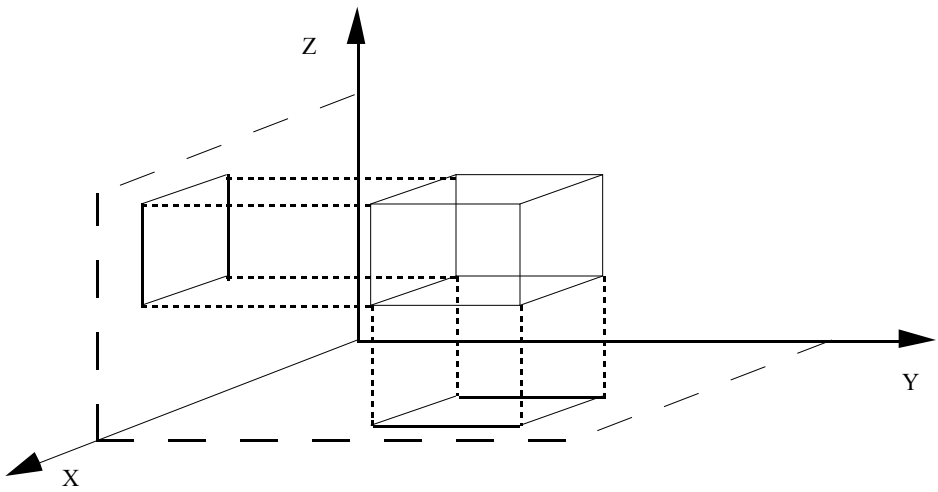
Techniques For Designing An Object

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.

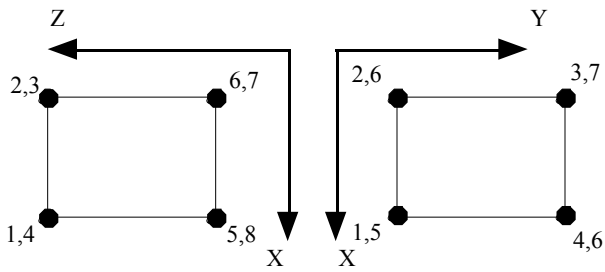


The cube in parallel projection, left rotated, right with rotation of zero

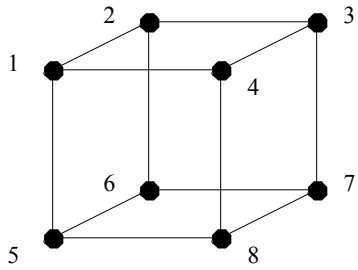
When drawing a 3D object on paper, points that were unique in space become ambigous on paper. Especially in parallel projections as can be seen in above figure. If you look at the cube on the right side, you notice that at each corner two points come to lie on top of each other. If I would point on one,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



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



Cube in two scetches

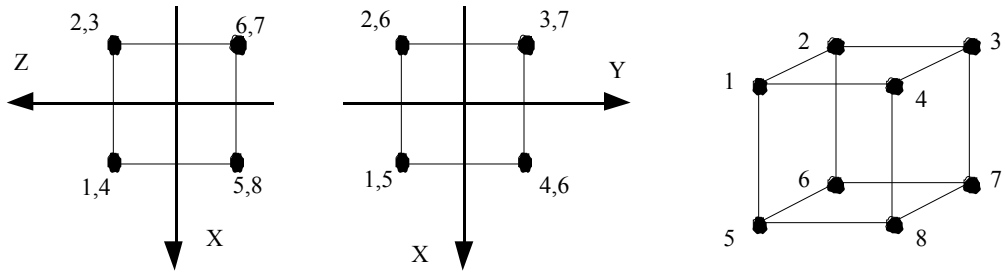


Cube in three dimensions

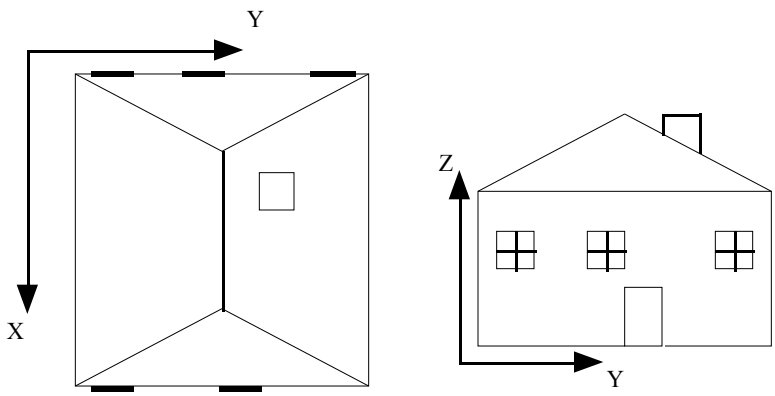
As you can see, no two points fall onto the same point in both scetches. To get each points coordinates, all you have to do is look it up in each scetch and read off the coordinates as you would do it with any normal 2D-Graph.
There is something very important to remember that becomes obvious if you look closely: both graphs have one common achsis (here it is the X-achsis). A point must *always* have the same coordinates on the common achsis of both scetches. If it doesn't, you have made a mistake. This is an easy way to proof your scetch.

You might have noticed that in order to produce the scetches we used the XY and the XZ plane. As you know, there is also the ZY plane. Yes, you could have used this one instead of the XZ plane. In fact, you can use any combination of two of the tree planes to generate the scetches.

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



As you can see, there is no problem if coordinates have negative values. As another example, look at the scetch of a house. Note how in the front view you can not tell where the smokestack nor the windows are located. Only the top view can clarify this.



Top View
(Windows shown)

Front View

However, in the top view you can't see how the windows look like or how high they are etc. Conversely, the fromt view dosn't schow that the first window from the left appears on *both* sides of the house. But both scetches taken together do define every point.
Note also that windows that happen to be on the left or right wall (as seen from top view) would show up in neither scetch. In these cases it might be necessary to draw another (third) scetch to define the remaining windows.

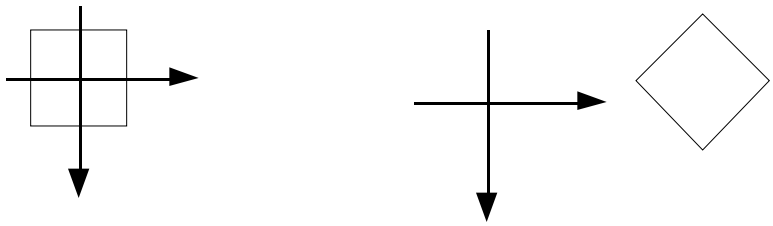
Advanced Topics

Free Rotating Tools

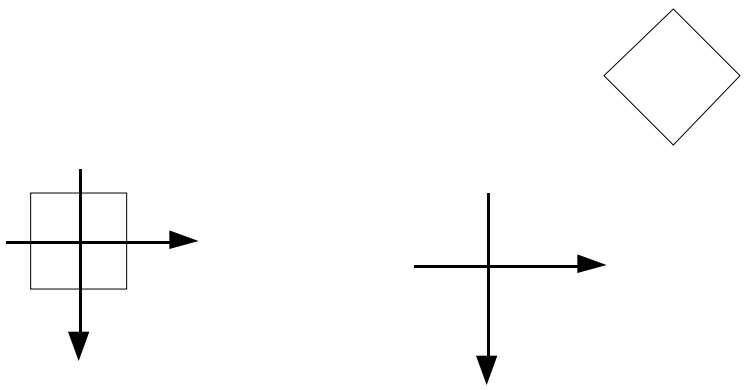
In addition to operations that enable you to rotate, translate and scale your objects to your hearts content, there are additional ways to transform your objects.

These opeartions too consist of rotating and translating, but there is a certain twist to it:
While calling the normal ObjRotate or ObjTranslate routines has no effect on the other and the order in which they are executed is of no consequence, all xxx**Free**yyy or xxx**Arb** *do* depend on their order of execution. This means that to undo the operations you have to do them with inverted signs in the *exact reverse order*. With the normal rotation and translation commands this is not necessary.
This is because the GrafSys provides a mathematical entity (called matrix) for rotating and translating each. If you keep translation and rotation apart, it is of no importance in what order they are executed.

As will be explained later, the GrafSys always rotates first and then translates. Sometimes however you want the package first to translate and then to rotate. The results are quite different as the following figure easily shows:



First rotating then translating the rotated square



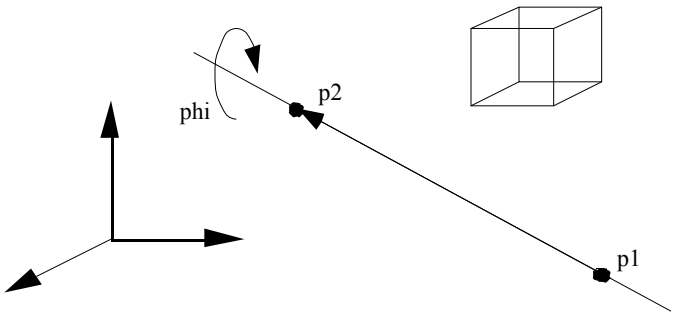
First translating then rotating the translated square

For this, every object has an additional matrix (actually there are two additional matrices, see below) called the FreeFormMatrix. On this matrix you can operate both rotation and translation. However, these rotate and translate operations are dependent on the order they are executed since they both operate on the same matrix and are not kept apart. This matrix is provided to

- give you more flexibility and
- enable you to do things that you couldn't do otherwise (like first translating and then rotating an object.

It is up to the programmer how to use the matrices. You can either use only the freeform matrix to archieve above results or use the TranslationMatrix and FreeFormMatrix (and never touch the RotationMatrix) or any other possible combination.

In additon to the normal rotation achses, the package provides you with means of rotating an object around any arbitrary achsis. You define an achsis by two points in the 3D space. This is called arbitrary rotation. The rotation achsis runs through the two points, looking from the first to the second point.



You normally would use this arbitrary rotation if an object follows another one in a certain way like a hand follows an arm (meaning of course that hand and arm are different objects). In this case you would define two points at the end of the arm, use the ObjPoint command to get the locations of these points, translate the hand to it and then rotate the hand around the achsis to its correct position.

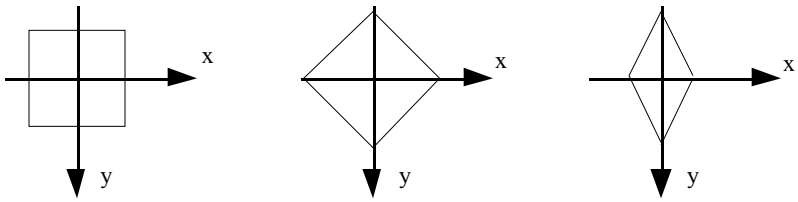
The transformations or: the importance of sequence of execution

The GrafSys provides four matrices for describing the 'state' or orientation the object is in. Normally, one would be enough, but to make operations easily reversible you have to provide more than one. The following describes the four matrices (and a fifth, the master- or Eye-matrix) and how they are used.

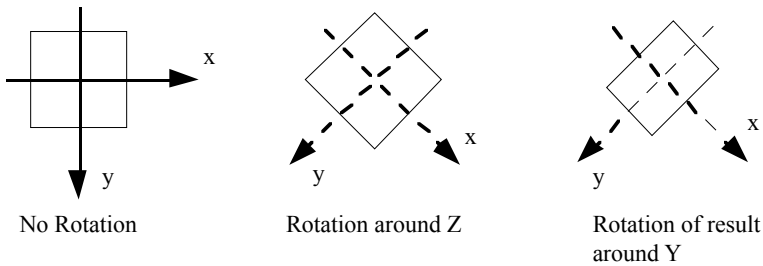
The matrices

Trot

The first matrix is called Trot. It contains the rotation around the three main achses of the object. Note that here you might get different results than you have expected. If you rotate an object 45 degrees first around the Z-Achsis and then around the Y-Achsis it doesn't mean that the Y-Achsis 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-achsis.

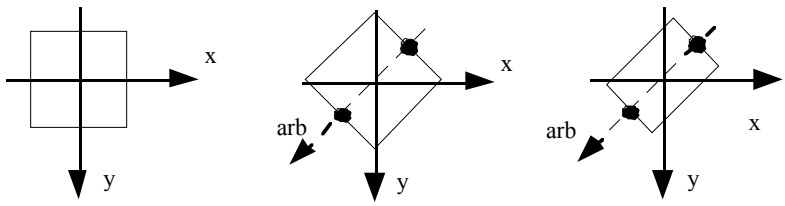


Some people migth have expected the following result:



But since this is dependent on which rotation you execute first, this would make it near impossible to program anything with it, since you always have to know which operation was executed when.

To archive above results you have to youse the following method:



No Rotation

Rotation around Z building arbitrary achsis

Rotation of result around arbitrary achsis

As you can see, the ability to rotate around any achsis gives you quite some flexibility although the 3D transformation are somewhat limited.

Ttrans

The second matrix is one of the easiest to understand. It holds the information used for translation and scaling and works just as you think it would.

Tanyrot

Tanyrot is an additional matrix you can use. It is used whenever you are using the ObjRotateArb operations. Since rotating around an arbitrary achsis always involves translating if the achsis doesn't run through the origin, the arbitrary rotations require their own matrix. If the achsis doesn't run through the origin, the results of operation depend on their order and undoing them will require them being in the exact reverse order.

Tfreeform

To add another level of freedom to the object I added the FreeFormMatrix. Any operation (rotate, arb. rotate and translate) is available to this matrix. But keep in mind that if you mix translation and rotation the operations become dependent on order of execution.

Eye: Master Transform

Another, albeit global matrix exist. This matrix is only used when the GrafPort3D's UseEyeFlag is set to TRUE. This matrix holds the transformations necessary to transform the object according to the settings of the eye. Since it gets recalculated every time you change the eye, it doesn't matter in what order you call the Eye.

Order of evaluation

Since sometimes it does get important, here is the order that the transformations get applied to the object:

- first the object is rotated according to Trot
- then the arbitrary rotations are added according to Tanyrot
- then the object is translated according to Ttrans
- then the xxxFreeyyy operations according to Tfreeform
- then the Eye transformations are applied.

Visualization: the Screen3D Unit

So far we only talked about objects and how to transform them. But how on earth do we get them on the screen so we can actually see them?

For this the unit Screen3D provides a set of operations (routines) that make it quite easy for you to do just that. There are two ways to draw objects. The more simplistic one just draws the object into whatever QuickDraw (QD) GrafPort you want.
The less simple one (but still by far not complicated) involves a data structure I call the ScreenObject. This is the data structure you would use if you have your own drawing routines. The ScreenObject holds all data necessary to draw your object anywhere and them some.

DrawObject

The Screen3D unit sports two commands that can draw an object to the current QD GrafPort. Both transform and convert the object and then draw it. fDrawObject also supports erasing the previously drawn object on repeated calls (called AutoErase), making it the command of choice for animations. However, the realy cool drawing routines are the ones involving the ScreenObjects.

ScreenObjects

ScreenObjects are variables that store additional information that is gatherd during the transformation process. The ScreenObjects holds the screen positions of all lines in a convenient array for ultra-fast access, converted points so you can implement your own hidden-line algorithms (although the DrawHLObject supports its own version of hidden-line/hidden-surface) or anything else you can think of.
Two operations, CalcScreenObject and CCalcScreenObject do exactly this, filling the ScreenObject with all this information. The command DrawScreenObject is the Screen3D routine to draw a prepared ScreenObject. If however the hasChanged flag in the MasterObject is still FALSE, you don't have to call the two conversion routines since the data is still valid. In this case CalcScreenObject and CCalcScreenObject only convert the object if you pass a forceCalc value of TRUE.

Warning: Although any routine that changes rotation, translation or scaling of an object also sets the hasChanged flag to TRUE, changing the Eye or GrafPort3D does *not* do so. The CalcScreenObject or CCalcScreenObject routines automatically detect if this situation arises and recalculates the object even if the hasChanged flag is FALSE.

All ScreenObjects are be attached to an object that becomes the MasterObject (in an object oriented environment a ScreenObject would be an instance of a normal GrafObject3D and all this attaching and unlinking would not be necessary). A ScreenObject is attached to a object via the AttachScreenObject call.

Using ScreenObjects is transparent if you use the standard calls once you have them attached to an object. Note that each object may only have one ScreenObject.

Generating objects for your programs

Since so far there is no interactive editor for building objects so you can later use them in your programs, you have to do it the hard way:

Use the project 'BuildObject.π'. This is a skeleton program for building and viewing an object. You have to code the object description into the MakeObject procedure (replace the code that builds the house and garden). The project uses a resource file called BuildObject.rsc. In this you will find a ResEdit TMPL (template) and (after the program has successfully run) the 3Dob resource containing your object.
After building all your objects paste them into the program you are writing.

Sometime, I will try to write an ObjectEdit program to create and modify objects but this will be a long time coming. Even better would be a ResEdit-Extension but that is too much for me right now. If someone out there thinks he can do it, please send me a copy. If it is good, why don't we include it into this package?

Using 3D Graphics Package

The 3D Graphics Package is divided into separate Parts, one called GrafSys and one Screen3D. The former is the actual 3D transformation unit, while the latter contains the routines that draw on the screen.

Call InitGraf at the beginning of your program. This sets up the required variables and initializes the transformation and projection packages within.

The GrafSys uses ports similar to Quickdraw. The main difference is that you can have multiple GrafPort3D in a single Quickdraw GrafPort. A GrafPort3D should always reside inside a Quickdraw GrafPort. Call NewGrafPort at least once. This sets up the projection plane and initializes the projector to parallel projection. This call should be immediately followed by a call to SetEye that will define the viewangle.

After this, you usually load your objects from resource or create them with the commands NewObject, AddPoint, AddLine and AddPoly. Once an object is done constructing or loading, use the ObjRotate, ObjTranslate and ObjScale to manipulate it. Call SetEye if you want to move the camera.

Note: object manipulation commands (rotate and translate) fall in two different cathegories:

All the ObjRotate, ObjTranslate and ObjScale routines are independent from each other and in which sequence they are executed.

In contrast, the Obj**Free**Rotate and Obj**Free**Translate commands all depend upon their order and different orders of calling will have different results (they have a cummulative effect). Those routines were added to give you an additional degree of freedom but you should be careful if you use them since an unexperienced user cannot predict what effect a change in the sequence of commands will have.

To view an object, first call TransformObject and then DrawObject to draw it on the Screen. If you are using the ScreenObjects, use CalcScreenObject and DrawScreenObject instead.

Using the ScreenObject for your own Drawing Routines

If you have implemented your own perversely-fast graphics routines you might not want to use the in the Screen3D provided drawing routines since they rely on the normal QuickDraw routines. GrafSys provides you with an easy interface that you can use to get all the data you need to draw the object. This interface is the ScreenObject. It really is nothing else but a data structure that contains all relevant data of the *transformed* object. You can use this data to do anything that you like.

Lets have a closer look at the ScreenObject:

```
ScreenObjPtr = ^ScreenObj;

ScreenObj = record
  nhmin, nhmax, nvmin, nvmax: integer; (* new rect     *)
                               (* from last calculation *)
  hmin, hmax, vmin, vmax: integer;
     (* Rect in which ScreenObject from SECOND LAST    *)
     (* call to ClacScreenObj was drawn                *)
  Point: PointArray; (* Transformed Points of object *)
  deepz: real; (* maximum z of all Transformed Points. *)
               (* Used for Scene-Building/HL/HS Alg.   *)
  maxPoint, maxLine, maxPoly: integer;
     (* number of Points,  Lines and Polygons in this  *)
     (* Object                                         *)
  Line: LineArray;  (* Lines as defined in Parent *)
  screenx: screenPts; (* x- and y-coords of all Points *)
  screeny: screenPts; (* after transformation          *)
  Autoerase: Boolean;
  EraseType: Integer;
  screen1x: ScreenArray; (* x-coordinates for clipped *)
                         (* lines in CxxxScreenObj    *)
  screen1y: ScreenArray; (*          - " -           *)
  screen2x: screenArray; (* used in Line-Clipping mode*)
  screen2y: screenArray; (*         - " -            *)
  screenLines : Integer; (*         - " -            *)
  newLine: newLineArray; (*         - " -            *)
  Polygons: PolyArray;   (* Polygons as in Parent     *)
 end;
```

The ScreenObject contains some fields that are specific for use with the DrawScreenObject routines. However, you can use them as well in your own Programs.

nhmin, nvmin, nhmax and nvmax are four integers reserved for calculating the screen boundaries of the object to draw. CalcScreenObject and

CCalcScreenObject place the information after transforming the object here.

hmin, vmin, hmax and vmax contain the objects screen boundaries from the last time the object was drawn. This is of course used by the DrawScreenObject routine to erase the old image. After drawing, DrawScreenObjet copies the contents of the nhxxx and nvxxx variables into these locations.

Point contains the coordinates of all the object points *after transformation*. You can use this information for your own depth sorting algorithms. Note that after transformation for the eye the coordinate system is moved rather than the eye. This means that the eye will always look at the XY plane.
If for example you implemented a flight simulator and moved the eye around the world, after transformation other objects distances to the eye are their distances to the global origin. This makes collision detection and distance calculation very easy.

Warning: If you are using the Fixed-Point version of the GrafSys, all coordinates are given in Fixed data type and you have to convert the X, Y and Z coodinates using the Fix2X call.

deepz contains the maximum (largest) Z coordinate of an object after transformation. SIce the eye (after tzransformation) is looking at the XY plane straight down the Z-achsis, use this value for queuing objects. The greater their deepz value, the further the object is from the eye. A negative values means that the whole object is behind the eye and should not be drawn if clipping is on.

maxPoint, maxLine and maxPoly contain the number of Points, Lines and Polygons so far defined in this obejct.

screenx and screeny are two arrays that contain the screen coordinates of each transformed point.

Autoerase is a copy of the same flag used in the master object. Note that you shouldn't rely on the correctnes of this value and rather look it up in the master object itself.

EraseType contains the method of how to erase the object prior to redrawing it if Autoerase is true. Note that so far no matter what you specify the object gets erased by erasing the bounding rect.

screen1x/y and screen2x/y are four arrays that contain all screen coordinates for all lines (aka 'Line Buffer'). These coordinates are the same as in screenx/y except that this buffer is optimized for drawing:

It contains the screen coordinates of all *Lines* i.e. to draw line #5 you would issue

```
MoveTo(screen1x[5],screen1y[5]);
DrawTo(screen2x[5],screen2y[5]);
```

As you can see, this can speed up drawing cosiderably.

Note: If you are using CCalcObject and clipping, those lines that completely fall offscreen will not show up in this array. Lines that are partially clipped will have their correct screen coordinates in here.

screenLines is the number of lines that are currently contained in the line buffer. Note that this number can be radically different from the number of lines defined in the object. If for example a line falls completely off the screen, the number of lines will be one less than in the objects definition.

newLine is an array that contains only boolean values. If a line begins at a new screen position and the cursor must be moved there via the MoveTo procedure, its corresponding value will be true. Otherwise you may skip the MoveTo command and simply continue drawing from the last position.

Polygons contain the polygon definitions as in master object.

To illustrate how to use the ScreenObject, Iook at how the Screen3D units DrawScreenObject command works:

```
procedure DrawScreenObject (theObject: GrafObjPtr);

  var
    index: Integer;
    r: Rect;
    x, y: integer;
    theScrnObj: ScreenObjPtr;
    thePort: Graf3DPtr;

begin
  theScrnObj := theObject^.ScreenObjLink;
                              (* get the screenObject *)
  if theScrnObj = nil then (* failsafe *)
    Exit(DrawScreenObject);

  with theScrnObj^ do
    begin
      if Autoerase then
        begin
```

```
GetGrafPort(thePort);
```

```
            SetRect(r, theScrnObj^.hmin, theScrnObj^.vmin,
                   theScrnObj^.hmax,theScrnObj^.vmax);
            EraseRect(thePort^.viewPlane);
          end; (* if autoerase *)

(* now draw the object. Use the Line Buffer for this *)
        for index := 1 to screenLines do
          begin
            if newLine[index] then
              MoveTo(screen1x[index], screen1y[index]);
            LineTo(screen2x[index], screen2y[index]);
          end;

(* since clipping might have destroyed/rendered useless the
  min/max values, rebuild them *)
        hmax := -32000;
        hmin := 32000;
        vmax := -32000;
        vmin := 32000;

        for index := 1 to screenLines do
          begin
            x := screen1x[index];
            y := screen1y[index];
            if x > hmax then (* do bounds checking *)
              hmax := x;
            if x < hmin then
              hmin := x;
            if y > vmax then
              vmax := y;
            if y < vmin then
              vmin := y;
            x := screen2x[index];
            y := screen2y[index];
            if x > hmax then (* do bounds checking *)
              hmax := x;
            if x < hmin then
              hmin := x;
            if y > vmax then
              vmax := y;
            if y < vmin then
              vmin := y;
          end;
        hmax := hmax + 1;
        vmax := vmax + 1;
        hmin := hmin - 1;
        vmin := vmin - 1;

      end; (* with *)
  end;
```

881 versus FixedPoint Arithmetic

Response to the initial publication of the GrafSys caught me completely off-guard. An overwhelming number of people asked me if it was possible to supply a version that uses fixed-point arithmetic instead of relying on the 881 math coprocessor.

As a result, there are now two versions of the GrafSys library. Those libraries that contain the word *'fix'* in its name work with any Macintosh. This is called 'the fixed version'. The other (original) version still requires at least a 020 processor and a math coprocessor.

Some people commented on the fact that Fixed-Point arithmetic is 'wickedly fast'. I was really astonished to see just how fast these routines were. If precision is not an issue, you might want to use the fixed version since it works with more macs.

Using the 881 Version

To use the GrafSys, include the file GrafSys.lib and GrafSys.Int into your project.

If you plan on using the provided screen drawing routines, you will also have to include the files Screen3D.lib and Screen3D.int into your project.

If you plan on writing two versions of the same program one using the 881 version, the other the fixed version, make sure you read the 'Compatability' paragraph, below.

Using FixedPoint Version

The fixed version runs on any Mac. Instead of using the math coprocessor it uses fixed point arithmetic that is lighning fast but not as accurate as real numbers. You should not use big numbers when using the fixed point version. Numbers greater than 32000 will surely produce strange results under certain conditions, numbers grater than 65000 are illegal. Note that coordinates easily can become this large if you use large values for both coordinates and translation.

To use the GrafSys, include the file GrafSys.fix.lib and GrafSys.fix.Int into your project. In addition, you must include the SANElib.lib into your project.

If you plan on using the provided screen drawing routines, you will also have to include the files Screen3D.fix.lib and Screen3D.fix.int into your project.

Compatability

The two versions of GrafSys are *Source Level* compatible. Well, almost. If you use the RealVector4 type in your programs instead of the Vector4 type you will have no compatability problems.

Object resources (the '3Dob' type) are *totally compatible*. The fixed library automatically loads and converts the floating point definitions to fixed-point while loading and back prior to writing them.

Make sure you never directly access an objects point definition since they are different in the two versions. Instead, always use the GetPoint, AddPoint, and ChangePoint routines. This way you will never have compatability problems.

```
procedure InitGrafSys;
```

Call this procedure only once at the beginning of your program. It will initialize the transformation routines and set aside memory for internal variables.

```
procedure NewGrafPort (thePlane: Rect; var the3DPort:
           Graf3DPtr);
```

NewGrafPort allocates memory for the new GrafPort3D and initializes it. The parameter thePlane is a rectangle that defines the size of the projection plane. The center of projection is set into the center of thePlane and the size of the viewplane is set identical to the projection plane. ProjectionType is set to parallel and the UseEye variable set to FALSE (meaning that all calls to TransformObject will ignore the current eye location and orientation). The eye is initialized to reside in the origin (0,0,0) and looking straight up into the positive z-direction. Clip and HiddenLine is ist to FALSE.

If you want the whole window's content as a GrafPort3D, pass theWindow^.portRect as parameter for thePlane.

Note that unlike Quickdraw, GrafSys places the origin of its coordinate system for drawing in the center of the viewing plane.

The current GrafPort3D is set to this new port.

```
procedure SetGrafPort (the3DPort: Graf3Dptr);
```

SetGrafPort sets the current GrafPort3D to the one specified. For all following transformations, this GrafPort3D's settings are used.

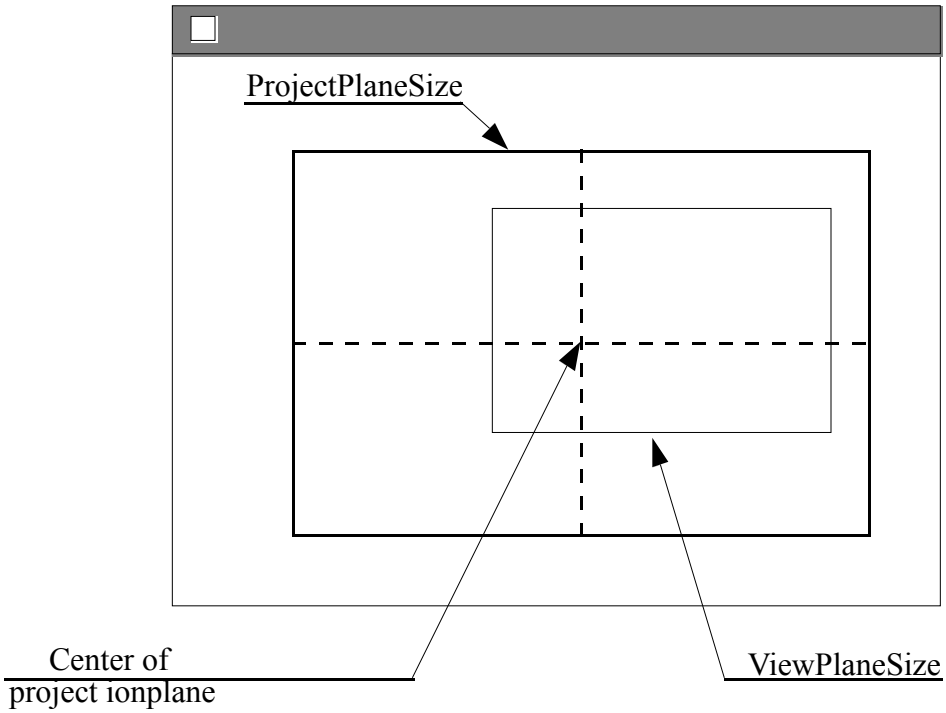
```
procedure GetGrafPort (var the3DPort: Graf3dptr);
```

GetGrafPort returns a pointer to the current GrafPort3D.

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

SetView sets the size of the projection plane and viewplane. The projection plane ist used to calculate the various perspective parameters. The viewplane 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.

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:



Note: This procedure operates on the current active GrafPort3D!

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

SetCenter repositions the center of the projectplane to the given coordinates. x and y should be in the windo's local coordinates.

Note: This procedure operates on the current active GrafPort3D!

```
procedure SetProjection (theGrafPort: Graf3DPtr;
          projectionType: INTEGER);
```

SetProjection sets the projection type for the GrafPort3D. projectionType can either be parallel or perspective. Any other value is not defined.

Operations to edit objects

Object

```
function NewObject: GrafObjPtr;
```

NewObject allocates memory for a new GrafObject. The object is initialized to no rotation, no translation, scaling of 1 (= no scaling) and contains no Points, Lines or Polygons. AutoErase and hasdrawn are set to FALSE, and EraseType to ObjRectFill. No ScreenObject is attached, so scrnObjLink is set to nil.
All lines are initialized as to draw from point zero to point zero (i.e. illegal points).

NewObject returns a pointer to the newly created object.

Call this procedure every time you want to construct an object from scratch.

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

GetNewObject allocates memory and initializes an object like NewObject and then tries to read in a resource of type '3Dob' with the specified ID. This resource contains all points, lines and polygons for this object and they are copied into the object.
GetNewObject returns a pointer to the newly created object.

```
function GetNewNamedObject (theObjectName: Str255):
```

GetNewNamedObject is the same as GetNewObject except that it tries to read a resource with the specified name.

```
procedure SaveObject (theObject: GrafObjPtr; theName:
           Str255; ID: integer);
```

Given a pointer to 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.

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

The parameter theName defines the name the resource will have.

```
procedure SaveNamedObject (theObject: GrafObjPtr; 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 resource.

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

Point

```
function AddPoint (theObject: GrafObjPtr; x, y, z: Real;
          var PointCount: integer): boolean;
```

Given a pointer to an object, AddPoint will add this point to the objects point description. x, y and z are the points coordinates.
The procedure returns with the current number of points in the object. If for some reason the procedure was unable to add the point to the object, it will return FALSE, otherwise TRUE.

```
function DeletePoint (theObject: GrafObjPtr; PointNumber:
          integer): Boolean;
```

DeletePoint will remove a point from an objects point description. All line descriptions are updatet to correctly reflect the change. If there is a line that references the point to be deleted, the routine does nothing and returns with FALSE, otherwise with true.

theObject is the object from which you want to delete a point, PointNumber is the index of the point to delete. If PointNumber is greater than the current number of points in the object or smaller than one, DeletePoint does nothing and returns FALSE.

Warning: If you delete a point that is also part of a polygon, DeletePoint will *not* return FALSE and go on deleting it. The result of an operation with an illegal point reference is not defined.

DeletePoint will also rebuild the newLine arguments for each line (see AddLine for details).

```
procedure GetPoint (theObject: GrafObjPtr; thePoint:
```

```
            integer; var x, y, z: REAL);
```

Given a pointer to an object, GetPoint will return the coordinates of the point with index thepoint in the variables x, y and z. If the index to the point is illegal (<1 or greater than the number of points in the object) the routine returns (0,0,0) as coordinates.

```
procedure ChangePoint (theObject: GrafObjPtr; thePoint:
            integer; x, y, z: real);
```

ChangePoint will change the coordinates of the point with the index thePoint in the object that is pointed to by theObject to the values specified in x, y and z. If the index to the point is illegal (<1 or greater than the number of points in the object) the routine does nothing.

Line

```
function AddLine (theObject: GrafObjPtr; src, tgt:
           integer): Boolean;
```

Given a pointer to an object, AddLine will append a line to the objects line description. src is the index of the point where the line begins and tgt the index of the line where it should draw to.
If any of the points is invalid (<1 or greater than the number of points in the object) or for some other reasons cannot add a line to the object, the procedure does nothing and returns FALSE, otherwise TRUE.

Together with the line description (from point src to point tgt) the GrafSys stores a flag that tells if this new line connects to the previous to accelerate drawing. AddLine updates this information.

```
function DeleteLine (theObject: GrafObjPtr; theLine:
           integer): Boolean;
```

DeleteLine will remove the line with index theLine from the object pointed to by theObject. If the index to the line is illegal (<1 or greater than the number of lines in the object) the procedure does nothing and returns FALSE, otherwise it returns TRUE.

DeleteLine will update the NewLine information after deleting a line from an object.

```
function ChangeLine (theObject: grafObjPtr; theLine:
          integer; src, tgt: integer): Boolean;
```

ChangePoint will change the src and tgt of the line with the index theLine in the object that is pointed to by theObject to the values specified in src and tgt.
If the index to the line is illegal (<1 or greater than the number of lines in the object) or one of the points illegal <1 or greater than the number of points in the object) the routine does nothing and returns FALSE, otherwise it returns TRUE.
ChangePoint updates the NewLine information in the object.

```
procedure GetLine (theObject: GrafObjPtr; theLine:
           integer; var src, tgt: integer; var newline:
           boolean);
```

GetLine will return the following information about the line with index theLine in the object pointed to by theObject:
From which point index in src to which point index in tgt and also if this line connects with the previous line (newline = FALSE) or if this line requires recalculation of startpoint (newline = TRUE).

If the index to the line is illegal (<1 or greater than the number of lines in the object) the procedure returns zero for both src and tgt and FALSE for newline.

Polygon

```
function SetPoly (p1, p2, p3, p4, p5, p6, p7, p8, p9, p10:
          integer): polygon;
```

SetPoly returns a polygon data structure that can be added to an objects polygon description. The parameters p1 through p10 contain the point indices. A value of zero tells the routine that the description ends here and the polygon is to be closed by drawing back to point p1.

Note: A polygon is alwasy closed. You don't have to specify the last point index since this will be the same as the first.

Note: Polygons are limited to 10 points each.

Warning: The SetPoly routine has no ways of checking if a point you specified will be legal in an object. Be very careful if you specify points that are not yet entered into an object.

```
procedure AddPolygon (theObject: GrafObjPtr; thePolygon:
           Polygon; var PolyRef: Integer);
```

AddPolygon adds a previously with SetPoly defined polygon to the object pointed to by theObject. The routine returns this polygons index in PolyRef.
If the operation for some reason was unsuccessful, AddPoly returns a zero as PolyRef.

```
function AddPointToPolygon (theObject: GrafObjPtr;
          thePolyref, thePointRef: Integer): boolean;
```

AddPointToPolygon adds another point to the polygon description of the object pointed to by theObject. thePolyRef is the index of the polygon you want to add a point to and thePointRef is the index of the point that you want to add.
The procedure returns TRUE if the point was added successfully, and FALSE otherwise.

Warning: AddPointToPolygon does not check if a point you specified is legal. Be very careful if you specify points that are not yet entered into the object.

Operations to manipulate objects locally, orderindependent

```
procedure ResetObject (theObject: GrafObjPtr);
```

ResetObject will set the object pointed to by theObject back to translation (0,0,0) rotation (0,0,0), Autoerase to FALSE, hasDrawn to FALSE and EraseType to ObjRectFill. The hasChanged attribute is set to TRUE.

translating

```
procedure ObjTranslate (theObject: GrafObjPtr; dx, dy, dz:
           Real);
```

ObjRotate will translate (i.e. move the origin of) the object pointed to by theObject. dx, dy and dz indicate how much *further* to translate the object along their respective achsis. Translation affects the Ttrans matrix. Translation is independent from any rotation of the object that has been done before. Translation does not affect rotation by ObjRotate or SetObjRot.

To translate an object to a specific position, use the SetObjTranslate, below.

```
procedure SetObjTranslate (theObject: GrafObjPtr; xTrans,
          yTrans, zTrans: Real);
```

ObjRotate will translate (i.e. move the origin of) the object pointed to by theObject to the global position specified in xTrans, yTrans and zTrans. Translation affects the Ttrans matrix. Translation is independent from any rotation of the object that has been done before. Translation does not affect rotation by ObjRotate or SetObjRot.

```
procedure GetObjTranslate (theObject: grafObjPtr; var
          xTrans, yTrans, zTrans: Real);
```

GetObjRot returns the global position of the object pointed to by theObject into the variables xTrans, yTrans and zTrans.

Note: GetObjTrans returns only the translation of the object that has been done with the ObjTranlate or SetObjTranslate, *not* with the xxx**Free**yyy or xxx**Arb**yyy Translate procedures. If you used the xxx**Free**yyyy or xxx**Arb**yyy operations, use the ObjPointArb procedure with the coordinates (0,0,0) as argument to get the correct translation values.

rotating

```
procedure ObjRotate (theObject: GrafObjPtr; dXrot, dYrot,
          dZrot: real);
```

ObjRotate will rotate the object pointed to by theObject around it's local origin. dXrot, dYrot and dZrot indicate how much *further* (in radiants) to rotate the object around their respective achsis. A positive value indicates a clockwise turn, a negative value counterclockwise. Rotation affects the Trot matrix. Rotation is independent from any translation of the object that has been done before. Rotation does not affect translation by ObjTranslate or SetObjTranslate.

To rotate an object to a specific angle, use the SetObjRot, below.

```
procedure SetObjRot (theObject: GrafObjPtr; Xrot, Yrot,
          Zrot: real);
```

ObjRotate will rotate the object pointed to by theObject around it's local origin. Xrot, Yrot and Zrot indicate to what angle (in radiants) to rotate the object around their respective achsis. A positive value indicates a clockwise turn, a negative value counterclockwise. Rotation affects the Trot matrix. Rotation is independent from any translation of the object that has been done before. Rotation does not affect translation by ObjTranslate or SetObjTranslate.

```
procedure GetObjRot (theObject: GrafObjPtr; var Xrot,
          Yrot, Zrot: real);
```

GetObjRot returns the local rotation values of the object pointed to by theObject into the variables Xrot, Yrot and Zrot.

Note: GetObjRot returns only the rotation of the object that has been done with the ObjRotate or SetObjRot, *not* with the xxx**Free**Rot or xxx**Arb**yyy procedures. If you used the xxx**Free**Rot or xxx**Arb**yyy procedures, you have to use the ObjPointArb procedure and do some calculating.

scaling

```
procedure ObjScale (theObject: GrafObjPtr; sx, sy, sz:
          Real);
```

ObjScale increments the scaling factors for the object pointed to by theObject 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-achsis). 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 achsis into nonexistence. Negative scaling will produce mirror-effects (I guess)

```
procedure SetObjScale (theObject: GrafObjPtr; xScale,
          yScale, zScale: Real);
```

ObjScale sets the scaling factors for the object pointed to by theObject 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-achsis). 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 achsis into nonexistence. Negative scaling will produce mirror-effects

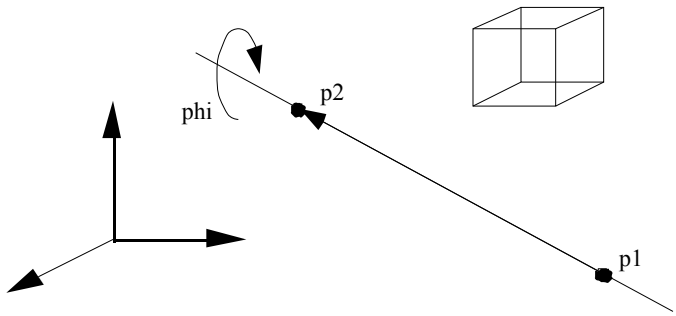
```
procedure GetObjScale (theObject: grafObjPtr; var xScale,
          yScale, zScale: Real);
```

GetObjScale returns the currently set scale factors of the object pointed to by theObject into the variables xScale, yScale and zScale.

arbitrary rotation

```
procedure ObjRotateArb (theObject: GrafObjPtr; p1, p2:
          Vector4; phi: Real);
```

ObjRotateArb rotates the object pointed to be theObject phi radiants further around an achsis defined by the two 3D poins p1 and p2.



The rotational achsis is defined as the line connecting p1 with p2, looking from p1 to p2. A positive angle means clockwise rotation. Note that the points p1 and p2 are given in the objects local coordinate system.

Note: Using ObjRotateArb with an achsis that doesn't run through the objects origin will falsify the relults returned by GetObjTranslate. Using ObjRotateArb will falsify the results returned by GetObjRot.

Note: The results of this command are strongly dependent on the order in which you call them. If you have two different achses called a1 and a2, first rotating around achsis a1 and then around a2 gives a different result than first rotating around a2 and then around a1. You should really be

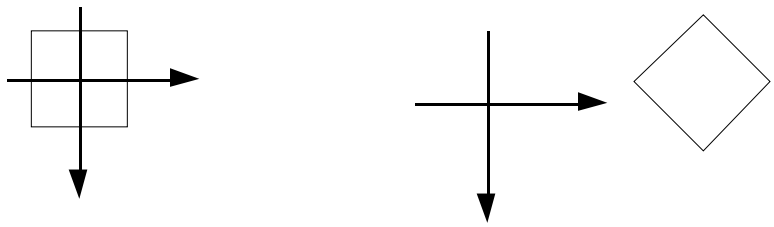
knowing what you are doing if you are using this command. See also the discussion of the xxx**Free**yyy commands below.

```
procedure ResetAnyRot (theObject: GrafObjPtr);
```

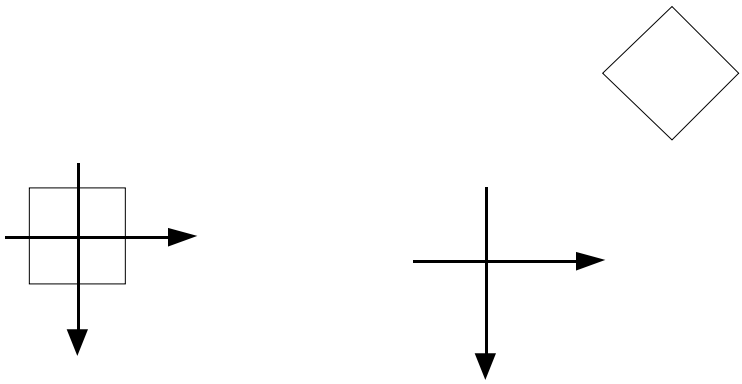
ResetAnyRot will cancel any rotations about arbitrary achses that you have done previously.

Operations to manipulate objects globally, orderdependent

Note: The three following routines are not independent from each other. A different order of these commands will have different results.



First rotating then translating the rotated square



First translating then rotating the translated square

Make sure you know what you are doing or you will be surprised by the results.

Note: Using any of the following three routines will falsify the results returned by GetObjRot and GetObjTranslate.

```
procedure ObjFreeRotate (theObject: GrafObjPtr; dXrot,
          dYrot, dZrot: real);
```

ObjFreeRotate will rotate the object pointed to by theObj for dXrot, dYrot and dZrot radiants further around the x-, y- and z-achses.

```
procedure ObjFreeRotateArb (theObject: GrafObjPtr; p1, p2:
          Vector4; phi: Real);
```

ObjFreeRotate will rotate the object pointed to by theObject phi radiants further around the arbitrary achsis defined by the two points p1 and p2.

```
procedure ObjFreeTranslate (theObject: GrafObjPtr; dx, dy,
          dz: Real);
```

ObjFreeRotate will translate the object pointed to by theObj for dx, dy and dz units further along the x-, y- and z-achses.

```
procedure ObjFreeReset (theObject: GrafObjPtr);
```

ObjFreeReset will cancel any previous xxx**Free**yyy commands used on the object pointed to by theObject.

Operations affecting Eye setting

Make sure that you call SetEye at least once in your program to set up the electronic camera and initialize the viewangle. Remember that the GrafPort3D is initialized to parallel projection and UseEye to FALSE. If you move the Eye to any other location than (0,0,0), set UseEye to TRUE or no change will happen.

```
procedure SetEye (UseEye: Boolean; x, y, z: REAL; phi,
          theta, pitch: real; viewangle: real; clipping:
          boolean);
```

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

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

camera should turn (pitch).

The viewangle 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 (≈ π = 3.14...) a wideangle lens. The viewangle parameter only affects the perspective drawing. A setting of zero or greater than 6.28 means parallel projection.
the clipping parameter is used only in the CCalcScreenObject procedure and controls if those lines that get too close to the eye will get clipped.

```
procedure geteye (var UseEye: Boolean; var x, y, z, phi,
           theta, pitch, viewangle: real; var clipping:
           boolean);
```

GetEye returns the current GrafPort3D's eye settings. See SetEye and 'Viewing an object : the eye' for the description of the parameters.

Miscellaneous Operations

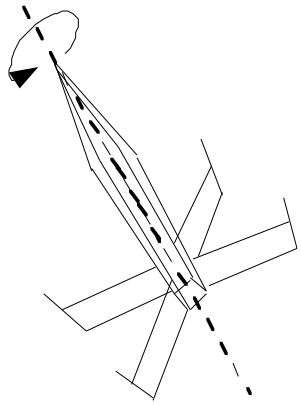
```
function ObjPoint (theObject: GrafObjPtr; thePoint:
          Integer): Vector4;
```

ObjPoint will return the world coordinates (*not* eye coordinates even if you are using the eye) of the point with the index thePoint from the object pointed to by theObject.
Note that all scaling, rotation and translation of the object apply to the conversion of this point.
To convert this point to screen coordinates (incluing eye transformation) use the ToScreen command.

```
function ObjPointArb (theObject: GrafObjPtr; x, y, z:
          Real): Vector4;
```

ObjPoint will return the world coordinates (*not* eye coordinates even if you are using the eye) of a point with the coordinates (x,y,z) in the local coordinate system of the object pointed to by theObject.
Note that all scaling, rotation and translation of the object apply to the conversion of this point.
To convert this point to screen coordinates (incluing eye transformation) use the ToScreen command.

You would usually use the two ObjPoint procedures to rotate an object around an arbitrary achsis. If you had an object like the one below and wanted to rotate it around its center, you would do the following:



Pass the index for the point of the bow to the ObjPoint routine. Save the result as p1. Pass the local coordinates of the point where the achsis goes through the stern to the ObjPointArb (or define another point there in the object point description and pass this point index). The result will be p2. Call ObjFreeRotate with p1 and p2 defining the achsis. Note that the rotational achsis should run through the objects origin or this will not work. Note also that you cannot use any other xxxFreeyyy command or the rotation will not be what you wanted.

```
procedure SetAutoErase (theObject: GrafObjPtr; Flag:
           Boolean);
```

SetAutoErase will set the AutoErase Flag in the object pointed to by theObject to the value of Flag. This flag is later used by the Screen3D module. If you don't use those routines, you can use it for your own purpose.

```
procedure ToScreen (x, y, z: real; var h, v: INTEGER);
```

ToScreen calculates the screen position of a 3 dimensional point as it would be seen from the eye using the current projection setting. If UseEyeFlag of the current GrafPort3D is set to FALSE then it will just convert the point to its corresponding screen coordinates using the current projection setting.

Operations to transform an Object

```
procedure TransformObject (theObject: GrafObjPtr; var
           xPointBuf, ypointBuf: screenPts; var hmin,
           vmin, hmax, vmax: integer; var deepz: Real;
           var Points: PointArray);
```

The TransformObject is the central routine of the graphic system. It will convert the object pointed to by theObject according to its rotation, translation, scaling and the GrafPort3D's eye settings.

After converting, all the objects converted points are stored in Points.
The highest z-coordinate of the converted object is returned in deepz.
xPointBuf and yPointBuf contain the screen coordinates for each point in the object.
hmin, hmax, vmin and vmax contain the bounding rectangle that would just enclose the object if it were drawn on the screen.

You can use all this data for your custom drawing routines. If you plan on using the supplied drawing routines from the Screen3D unit you will never have to call TransformObject yourself.

Drawing to the screen (Screen3D Unit)

The unit Screen3D contains a collection of procedures that makes both the handling of objects and arbitrary drawing in 3D very easy. You don't have to care about where to erase or how to draw lines. The package takes care of this for you.
It also supports clipping of lines that come too close to the eye and a limited version of Hidden-Line/Hidden-Surface drawing.

For most operations, the unit uses a special data structure called the ScreenObject. The structure contains all necesary data required to draw objects on the screen. If you have your own optimized drawing routines, you might still want to use the screen objetcs in conjunction with the xCalcScreenObject operations and then use your own drawing routines.

Operations for simple 3D drawing

```
procedure MoveTo3D (x, y, z: Real);
```

MoveTo3D moves the QuickDraw cursor to the screen location where the 3D point specified by (x,y,z) would be projected. If the UseEyeFlag is TRUE eye conversion will also be used.

```
procedure LineTo3D (x, y, z: Real);
```

LineTo3D draws a line from the current the QuickDraw cursorposition to the screen location where the 3D point specified by (x,y,z) would be projected. If the UseEyeFlag is TRUE eye conversion will also be used.

Operations to draw objects

```
procedure DrawObject (theObject: GrafObjPtr);
```

DrawObject draws the object pointed to by theObject to the current Quickdraw GrafPort. The procedure updates the objects bounding rect parameters. DrawObject does not support the AutoErase feature. Actual drawing is a bit slower than fDrawObject so using this routine to draw repeatedly on the screen might result in a flicker. The hasChanged attribute is *not* cleared.

```
procedure fDrawObject (theObject: GrafObjPtr);
```

fDrawObject is the main object drawing routine. It draws the object pointed to by theObject to the current Quickdraw GrafPort.Drawing is a lot quicker than DrawObject but overall performance is a bit slower due

to line-buffering. If you are using offscreen bitmaps, use DrawObject instead.

fDrawObject supports the AutoErase flag. If the hasDrawn flag is TRUE, fDrawObject will erase the bounding rectangle (that should contain the bounds from the previous drawing).
Then it will draw the object, update the bounding rectangle and set hasDrawn to TRUE and hasChanged to FALSE.

Note: Although three different erase types are defined (ObjRectFill, XorLines, WhiteLine and BlackLines) only the first is supported so far. No matter what EraseType you specify, it will always be ObjRectFill. This method is filling the bounding rectangle with the current QD GrafPort's background pattern and color.

Operations on ScreenObjects

```
function NewScreenObject: ScreenObjPtr;
```

NewScreenObject returns a pointer to a newly allocated memory block that contains the initialized ScreenObject.

```
procedure UpdateScreenObject (theObject: GrafObjPtr);
```

UpdateScreenObject transfers vital information from the associated object pointed to by theObject to the ScreenObject that is linked to it. If no ScreenObject is linked (attached) the theObject, the routine does nothing.

'Vital information' in this case are the settings of the AutoErase, Changed, hasDrawn and EraseType variables. Then it copies the contents of the associated line and polygon descriptions to the ScreenObject (the point descriptions are *not*copied since their transformed values are stored there).
After this, UpdateScreenObject copies the maxPoint, maxLine and maxPolygon descriptions (i.e. the current number of points, lines and polygons in the object) to the screenobject.

Call this procedure whenever you made changes to either flag value or changed the object description (point, line or polygon).

Warning: If you change the line description of the object and don't pass the information on, strange things may happen that I don't dare to imagine (i.e. very very

strange. Remember what happened to Harry Kammer? Don't know him? Strange, isn't it?)

However, you don't have to call UpdateScreenObject when you have changed the rotation, scaling or translation of an object.

Note: This operation becomes useless once the OOP version of the GrafSys is ready.

```
procedure CalcScreenObject (theObject: grafObjPtr;
           forceCalc: Boolean);
```

CalcScreenObject is to a ScreenObject what TransformObject is to a normal object. It Transformes the object pointed to by theObject and stores the results of the transformation into the attached ScreenObject pointed to by theObject^.ScreenObjLink.

CalcScreenObject then does some additional processing, building up a buffer containing all lines that must be drawn and preparing the object to be drawn by fast specialzed routines (usually DrawScreenObject).

CalcScreenObject only transforms the object if it's hasChanged flag is set to TRUE or the eye settings have been changed. You can force a recalculation by passing the value TRUE to the forceCalc parameter.

After calculating and updating the ScreenObject, the objects hasChanged flag is set to FALSE.

Warning: Although any routine that changes rotation, translation or scaling of an object also sets the hasChanged falg to TRUE, changing the Eye or GrafPort3D does *not* do so. CalcScreenObject has code included to detect this situation and will act accordingly.

If no ScreenObject is attached to theObject, the routine does nothing.

```
procedure AttachScreenObject (theScrnObj: ScreenObjPtr;
           theObject: GrafObjPtr);
```

AttachScreenObject links the ScreenObject pointed to by theScrnObj to the object pointed to by theObject. It must be executed at least once every time you link an object to a ScreenObject or want to change the link.
AttachScreenObject calls UpdateScreenObject once to initialize the ScreenObject to its new master object

Note: This operation becomes useless once the OOP version of the GrafSys is ready.

Note: Each object may only have one ScreenObject.

```
procedure UnLinkScreenObject (theObject: GrafObjPtr; var
          theScrnObj: ScreenObjPtr);
```

UnlinkScreenObject severs the link between the object pointed to by theObject and returns a pointer to the ScreenObject that was cut off.

Note: This operation becomes useless once the OOP version of the GrafSys is ready.

```
procedure DrawScreenObject (theObject: GrafObjPtr);
```

DrawScreenObject draws the object pointed to by theObject to the current Quickdraw GrafPort. It uses the attached ScreenObject for much faster drawing. The ScreenObject attached to theObject should have been properly filled in prior to calling DrawScreenObject. Usually you this by calling CalcScreenObject or CCalcScreenObject. If, however, theObject.^hasChanged is FALSE, you don't need to call these routines, since the data in the ScreenObject is still valid.

If no ScreenObject is attached to theObject, the routine does nothing.

DrawScreenObject supports the AutoErase flag. If the hasDrawn flag is TRUE, fDrawObject will erase the bounding rectangle (that should contain the bounds from the previous drawing).
Then it will draw the object, update the bounding rectangle and set hasDrawn to true.

Note: Although three different erase types are defined (ObjRectFill, XorLines, WhiteLine and BlackLines) only the first is supported so far. No matter what EraseType you specify, it will always be ObjRectFill. This method is filling the bounding rectangle with the current QD GrafPort's background pattern and color.

Operations for clipped line-drawing

```
procedure CCalcScreenObject (theObject: grafObjPtr;
          forceCalc: Boolean);
```

CCalcScreenObject is exactly like CalcScreenObject except that it generates all information necessary for clipping. All lines that after transformation go through the projection plane are clipped to the point where they intersect.

After calculating and updating the ScreenObject, the objects hasChanged flag is set to FALSE.

CalcScreenObject only transforms the object if it's hasChanged flag is set to TRUE or the eye settings have been changed. You can force a recalculation by passing the value TRUE to the forceCalc parameter.

Warning: Although any routine that changes rotation, translation or scaling of an object also sets the hasChanged falg to TRUE, changing the Eye or GrafPort3D does *not* do so. CCalcScreenObject has code included to detect this situation and will act accordingly.

Operations for Hidden-Line/Hidden-Surface drawing

```
procedure DrawHLScreenObject (theScrnObj: ScreenObjPtr);
```

DrawHLScreenObject is still in experimental stage. It attempts to draw the object pointed to by theObject. For its calculations it uses the ScreenObject attached to theObject.

If no ScreenObject is attached to theObject, the routine does nothing.

DrawHLScreenObject attempts a simple approach to hidden surface/hidden line drawing:

All surfaces have been defined as polygons. The procedure then sorts all transformed polygons according to their maximum depth and then draws them beginning with the deepest (= greatest z value) polygon until drawing the closest polygon.

Summary of Commands GrafSys

Constants

```
Pi = 3.14159265;
```

```
(* projection Types *)
```

```
 parallel = 0;
 perspective = 1;

(* Erase Types  *)
 ObjRectFill = 0;
 XorLines = 1;
 WhiteLines = 2; (* draw all Lines in White *)
 BlackLines = 3; (* draw all Lines in Black *)
```

Data Types

```
 Eye3D = record
   location: Point3D;
   phi: Real;
   theta: Real;
   pitch: Real;
   ViewAngle: Real;
  end;

 Graf3DPtr = ^Grafport3D;
 Grafport3D = record
   ProjectionPlane: Rect;
   ViewPlane: Rect;
   left, right, top, bottom: Integer; (* Window rect *)
   center: Point; (* center of Viewplane *)
   MasterTransform: Matrix4; (* Matrix for pretrafo for *)
                             (* eye-coords             *)
   eye: Eye3D;  (* the Eye of the Camera *)
   UseEyeFlag: Boolean; (* FALSE --> eye is always at   *)
                        (* (0,0,0) and looks straight   *)
                        (* down z *)
   d: Real; (* Perspective Parameter set by Viewangle   *)
   Clip: Boolean; (* Tells algorithm if to clip to the  *)
                  (* Z=0 Plane after trafo              *)
   HiddenLine: Boolean; (* use Hidden-Line Algorithm on *)
                        (* Object                       *)
   projectionType: INTEGER; (* parallel or perspective  *)
   versionID: LongInt;      (* used to identify changes *)
  end;

 screenArray = array[1..MXL] of integer;
     (* note : mxL, for all screencoords will be stored *)
     (*        for all lines *)
 screenPts = array[1..MXP] of integer;
 newLineArray = array[1..MXl] of Boolean;


 ScreenObjPtr = ^ScreenObj;
 ScreenObj = record
   nhmin, nhmax, nvmin, nvmax: integer; (* new rect     *)
                                (* from last calculation *)
   hmin, hmax, vmin, vmax: integer;
```

```
   (* Rect in which ScreenObject from SECOND LAST    *)
   (* call to ClacScreenObj was drawn                *)
Point: PointArray; (* Transformed Points of object *)
deepz: real; (* maximum z of all Transformed Points. *)
             (* Used for Scene-Building/HL/HS Alg.   *)
```

```
   maxPoint, maxLine, maxPoly: integer;
      (* number of Points,  Lines and Polygons in this  *)
      (* Object                                         *)
   Line: LineArray;  (* Lines as defined in Parent *)
   screenx: screenPts; (* x- and y-coords of all Points *)
   screeny: screenPts; (* after transformation          *)
   Autoerase: Boolean;
   EraseType: Integer;
   screen1x: ScreenArray; (* x-coordinates for clipped *)
                          (* lines in CxxxScreenObj    *)
   screen1y: ScreenArray; (*          - " -            *)
   screen2x: screenArray; (* used in Line-Clipping mode*)
   screen2y: screenArray; (*         - " -             *)
   screenLines : Integer; (*         - " -             *)
   newLine: newLineArray; (*         - " -             *)
   Polygons: PolyArray;   (* Polygons as in Parent     *)
  end;

 GrafObjPtr = ^GrafObject;
 GrafObject = record
   versionID: LongInt; (* used to identify changes in *)
                        (* Eye*)
   hasChanged: Boolean; (* will be set TRUE after any *)
                        (* change to Object           *)
   x, y, z: Real; (* position of object's Origin in   *)
                  (* 3D-space                          *)
   xRot, yRot, zRot: Real; (* rotation of Object to   *)
                           (* its own origin          *)
   sx, sy, sz: Real; (* objects scaling factors       *)
   Trot: Matrix4; (* internal use : object's trafo-   *)
                  (* matrix for local rotation        *)
   Ttrans: Matrix4; (* internal use : objects trafo-  *)
                    (* Matrix for translation and     *)
                    (* scaling                        *)
   Tanyrot: Matrix4; (* internal use : matrix for     *)
                     (* additional rotation around an *)
                     (* arbitrary achsis              *)
   Tfreeform: Matrix4; (* internal use only : free    *)
                       (* translation of object       *)
   maxPoint: Integer; (* number of points in Object   *)
   maxLine: Integer; (* number of Lines in Object     *)
   maxPoly: Integer; (* number of Polygons in Object  *)
   Point: PointArray; (* All Points of this object    *)
   Line: LineArray;   (* All lines                    *)
   Polygons: PolyArray; (* All polygons               *)
   vmax: integer; (* this objects maximal and minimal *)
   vmin: integer; (* screen coords after last draw    *)
   hmin: integer;
   hmax: integer;

(* AutoErase and hasDrawn are only used in the Screen3D *)
(* package with the Draw(Screen)Object routine          *)
```

```
AutoErase: Boolean; (* Flag for use with vmax..hmax *)
                     (* and the fDrawObject routine  *)
hasDrawn: Boolean; (* internal use only : for use  *)
                   (* with erase flag              *)
EraseType: INTEGER;(* What kind of Erase-Technique *)
                   (* for Autoerase                *)
```

```
 ScreenObjLink: ScreenObjPtr; (* attached screen-   *)
                              (* object. defaults to NIL *)
end;
```

Routines

GrafPort3D Routines

```
procedure InitGrafSys;
procedure NewGrafPort (thePlane: Rect; var the3DPort:
          Graf3DPtr);
procedure SetGrafPort (the3DPort: Graf3Dptr);
procedure GetGrafPort (var the3DPort: Graf3dptr);
procedure SetView (ProjectPlaneSize, ViewPlaneSize: Rect);
procedure SetCenter (x, y: INTEGER);
procedure SetEye (UseEye: Boolean; x, y, z: REAL; phi,
          theta, pitch: real; viewangle: real; clipping:
          boolean);
procedure geteye (var UseEye: Boolean; var x, y, z, phi,
          theta, pitch, viewangle: real; var clipping:
          boolean);
procedure setprojection (theGrafPort: Graf3DPtr;
          projectionType: INTEGER);
```

Operations to edit objects

```
function NewObject: GrafObjPtr;
function GetNewObject (theObjectID: INTEGER): GrafObjPtr;
function GetNewNamedObject (theObjectName: Str255):
          GrafObjPtr;
procedure SaveObject (theObject: GrafObjPtr; theName:
          Str255; ID: integer);
procedure SaveNamedObject (theObject: GrafObjPtr; theName:
          Str255; var ID: integer);
function AddPoint (theObject: GrafObjPtr; x, y, z: Real;
          var PointCount: integer): boolean;
function DeletePoint (theObject: GrafObjPtr; PointNumber:
          integer): Boolean;
procedure GetPoint (theObject: GrafObjPtr; thePoint:
          integer; var x, y, z: REAL);
procedure ChangePoint (theObject: GrafObjPtr; thePoint:
          integer; x, y, z: real);

function AddLine (theObject: GrafObjPtr; src, tgt:
          integer): Boolean;
function DeleteLine (theObject: GrafObjPtr; theLine:
          integer): Boolean;
function ChangeLine (theObject: grafObjPtr; theLine:
          integer; src, tgt: integer): Boolean;
procedure GetLine (theObject: GrafObjPtr; theLine:
          integer; var src, tgt: integer; var newline:
          boolean);
```

```
function SetPoly (p1, p2, p3, p4, p5, p6, p7, p8, p9, p10:
          integer): polygon;
procedure AddPolygon (theObject: GrafObjPtr; thePolygon:
           Polygon; var PolyRef: Integer);
```

```
function AddPointToPolygon (theObject: GrafObjPtr;
          thePolyref, thePointRef: Integer): boolean;
```

Operations to manipulate objects locally, orderindependent

```
procedure ResetObject (theObject: GrafObjPtr);
procedure ObjRotate (theObject: GrafObjPtr; dXrot, dYrot,
           dZrot: real);
procedure SetObjRot (theObject: GrafObjPtr; Xrot, Yrot,
           Zrot: real);
procedure GetObjRot (theObject: GrafObjPtr; var Xrot,
           Yrot, Zrot: real);
procedure ObjTranslate (theObject: GrafObjPtr; dx, dy, dz:
           Real);
procedure SetObjTranslate (theObject: GrafObjPtr; xTrans,
           yTrans, zTrans: Real);
procedure GetObjTranslate (theObject: grafObjPtr; var
           xTrans, yTrans, zTrans: Real);
procedure ObjScale (theObject: GrafObjPtr; sx, sy, sz:
           Real);
procedure SetObjScale (theObject: GrafObjPtr; xScale,
           yScale, zScale: Real);
procedure GetObjScale (theObject: grafObjPtr; var xScale,
           yScale, zScale: Real);
procedure ObjRotateArb (theObject: GrafObjPtr; p1, p2:
           Vector4; phi: Real);
procedure ResetAnyRot (theObject: GrafObjPtr);
```

Operations to manipulate objects globally, orderdependent

```
procedure ObjFreeRotate (theObject: GrafObjPtr; dXrot,
           dYrot, dZrot: real);
procedure ObjFreeRotateArb (theObject: GrafObjPtr; p1, p2:
           Vector4; phi: Real);
procedure ObjFreeTranslate (theObject: GrafObjPtr; dx, dy,
           dz: Real);
procedure ObjFreeReset (theObject: GrafObjPtr);
```

Miscellaneous Operations

```
function ObjPoint (theObject: GrafObjPtr; thePoint:
          Integer): Vector4;
function ObjPointArb (theObject: GrafObjPtr; x, y, z:
          Real): Vector4;
procedure SetAutoErase (theObject: GrafObjPtr; Flag:
           Boolean);
procedure ToScreen (x, y, z: real; var h, v: INTEGER);
```

Operations to transform objects

```
procedure TransformObject (theObject: GrafObjPtr; var
           xPointBuf, ypointBuf: screenPts; var hmin,
           vmin, hmax, vmax: integer; var deepz: Real;
           var Points: PointArray);
```

Operations for simple 3D drawing

```
procedure MoveTo3D (x, y, z: Real);
procedure LineTo3D (x, y, z: Real);
```

Operations to draw objects

```
procedure DrawObject (theObject: GrafObjPtr);
procedure fDrawObject (theObject: GrafObjPtr);
```

Operations on ScreenObjects

```
function NewScreenObject: ScreenObjPtr;
procedure UpdateScreenObject (theObject: GrafObjPtr);
procedure CalcScreenObject (theObject: grafObjPtr;
          forceCalc: Boolean);
procedure AttachScreenObject (theScrnObj: ScreenObjPtr;
          theObject: GrafObjPtr);
procedure UnLinkScreenObject (theObject: GrafObjPtr; var
          theScrnObj: ScreenObjPtr);
procedure DrawScreenObject (theObject: GrafObjPtr);
```

Operations for clipped line-drawing

```
procedure CCalcScreenObject (theObject: grafObjPtr;
          forceCalc: Boolean);
```

Operations for Hidden-Line/Hidden-Surface drawing

```
procedure DrawHLScreenObject (theScrnObj: ScreenObjPtr);
```

Resource Format

GrafSys provides commands to save and get object descriptions into and from resources, respectively. The resourcetype 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 litte 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: