Genesis Programming Specification

Author: Steven Woodman
Date: 16/8/96
Revision: 6

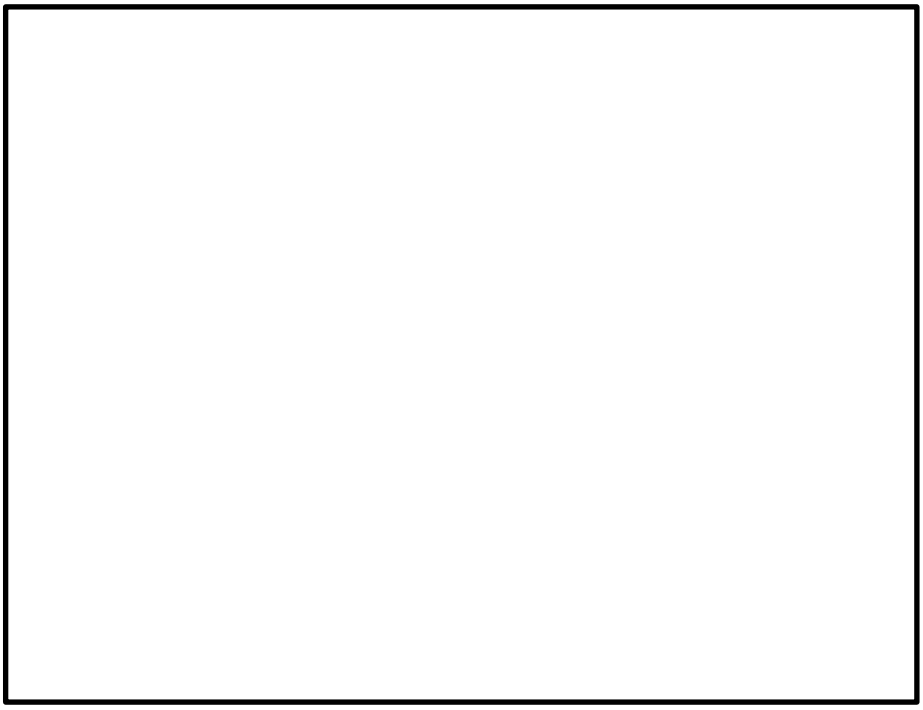


TABLE OF CONTENTS

Geometry Specification

Introduction

This part of the spec describes the Geometry API. It assumes an understanding of programming in C++ and a basic knowledge of programming in Windows. Knowledge of mathematics is not a prerequisite although a basic understanding of vectors and matrices is helpful.

Figure 1 shows the various component parts of the Genesis package and how they relate to one another.

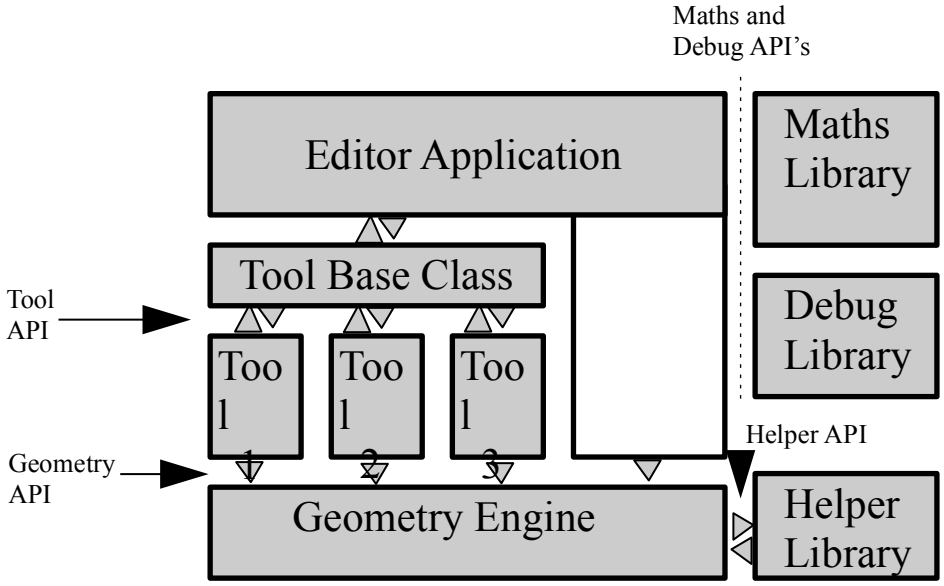


Figure 1. Components of Genesis

Geometry is the part of Genesis that actually stores the data structures representing the 3D models and performs all the 3D rendering. The editor is an application that calls the Geometry API as the user performs various actions on the program. For instance when the user presses the 'Pos' button on the control bar the editor makes a call to the SetCamera routine in order to position the camera ready for rendering. When the user selects the 'Render' menu item, a call to Render is made. A user will never need to know these details, however there are three scenarios in which a programmer might want to understand how the API works.

- Providing a new Geometry engine
- Writing new tools for the editor
  - Using Geometry for applications

Whichever scenario you are interested in the maths and debug libraries supplied in this package will prove invaluable programming aids (See the 'Maths Library' and 'Debug Library' sections).

Which compiler?

The various Genesis library and import library files were produced using Microsoft Visual C++. Although any compiler can be used to compile source code for Genesis, some linkers might experience problems trying to link to these lib files. If you experience problems email the author; spike@silicond.demon.co.uk stating which compiler you are using and we'll attempt to furnish you with a new set of libs.

Providing a new Geometry engine

The default Geometry engine is a single DLL which provides software for 3D graphics processing. It doesn't support accelerated hardware because at the time of writing we couldn't possibly anticipate what manner of 3D acceleration hardware might become available in the future. When third parties release their new 3D processing systems, whether hardware or software based, Genesis can support them simply by writing a DLL to interface the Geometry API to the new hardware/software. Imagine for example that the XYZ computer company came out with a board and a software interface that gave PC's the ability to render at over a million polygons a second, then with a few days work Genesis with all its powerful editing tools could be rendering much faster and completely transparently from the users viewpoint.

To create a new Geometry engine you will create a DLL implementing the Geometry API but which maps those functions onto the software functions provided by the third party 3D company. A set of routines is supplied in this package to make this job easier (See the 'Geometry Helper Library' section).

Writing new tools

The Genesis editor is very powerful in the respect that if a function is needed which isn't available, with a little programming knowledge you can create your own extensions thereby tailoring Genesis to your own particular set of problems. For instance, an architect might spend most of his time creating doors, all of which are different but all share basic characteristics. By writing a door creation tool, he can position the cursor where he wants the door, press a button in the tool box representing his custom tool, enter a few basic parameters into a dialog box, and hey presto, a door appears. In this way users can build their own libraries of tools all of which are seemlessly integrated into the program. In this way Genesis can be considered as little more than a shell designed to support a vast variety of 'plug in' tools to cover every conceivable problem in 3D design.

To create a new tool you will create a DLL implementing a C++ class derived from a class that we supply (See the 'Writing Tools' section). Don't worry if you are not familiar with C++ as we provide a C++ template source file. All you need to do is to fill in the relevant functions using C. You will use the Geometry API plus functions in the base class to implement the functionality of the tool. Minimal knowledge of programming in Windows is required.

Some of the more general tools we thought might be useful are listed below. Basically anything which manipulates 3D models can be implemented as a tool. However there are in fact an unlimited number considering the huge range of applications Genesis might be applied to:

1. Primitive creation types (Creates primitive objects)
    - Patches
    - Spheres/Semi spheres
    - Cylinders
    - Boxes
    - Cones
    - Torus's

- Ellipsoids/General curved area creators

1. Traditional types (Often found on more conventional modellers)
   - Lathe (Spins a 2D outline into a 3D shape)
   - Extrude (Expands a 2D outline to give it 3D depth)
   - Copy (Makes a copy of an object)
     - Text input window (Calls other tools when you type in commands like 'sphere rad=10;')
1. Savers/Loaders (for various file formats)
   - 3DS
   - DXF
     - WAD (Doom game file format)
1. CSG tools (Enables new objects to be created by combining existing ones)
   - Union operator (joins two objects as one object; a=b|c)
   - Intersection operator (creates one object from the overlapping volume of two others; a=b&c)
   - Subtraction operator (subtracts one object from another; a=b-c)
     - Exclusive OR operator (creates an object as the volume of two other objects excluding any intersection; a=b^c)
1. Imaginative ideas
   - Fractal landscape generator
     - 'Physical law' animation tools

Using Geometry for applications

The Geometry engine can be used via its API to construct new applications totally separate from the Genesis editor. For instance; Flight simulators, medical imaging applications, oil field data visualisation, 3D games to mention but a few. The advantages of using the Geometry API are twofold. Firstly all the difficult 3D stuff is already taken care of as well as a lot of other useful bits such as routines to manipulate 3D vectors and matrices etc. (See 'Maths Library' section). Secondly, any new hardware appearing for which a geometry engine is written (see above) will automatically work with the new application. So your flight simulator might not look too quick on a slow 486 PC but when XYZ's new 3D processor board hits the market suddenly without any additional programming effort your 486 looks like a professional simulator.

To write a new application the Geometry DLL is used as a stand alone DLL, just like any other, which is called from your application according to the Geometry API.

How the Geometry API works

The Geometry design is a hierarchical one. Objects such as vertices, patches and normals belong to objects. Objects belong to co-ordinate systems as do lights. Co-ordinate systems belong to other co-ordinates system. Co-ordinate system along with all their associated objects can be rendered. The following sections describe the terminology further.

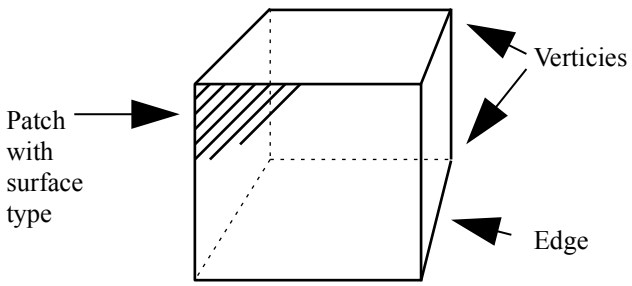


Figure 2. Construction of a cube

Vertices

A vertex is a point in 3D space described by an x, y and a z value. A cube has 6 vertices or corners, 12 edges, and 8 faces.

Patches

A patch is a polygon or face defined by 3 or more vertices, such as the 8 faces of the cube. You build up 3D models by connecting patches together exactly like a patchwork quilt. A patch has a surface type indicator defining its colour, texture and reflection characteristics. Patches are always flat but can be made to appear curved by associating normals with each of its vertices (see below). Because they are always flat care should be taken when creating patches of more than 4 vertices that all the vertices lie on a plane otherwise Geometry will politely inform you of your error.

Patches are one sided entities meaning they are intended to be viewed only from one side. An attempt to view it from the other side will in fact make it invisible. This might sound strange but is actually quite logical. For instance as you look around your cube you are always seeing the individual patches from the same side, to view them from the other side you will need to go inside the cube. If you did this you would actually see straight through to the outside because the patches would be invisible. What you need to do is design the inside of the cube to prevent you seeing out, i.e. you need 8 more patches inside facing the other way. The reason for this is that it makes rendering quite a bit faster. In cases where you don't want to go inside the cube you only need 8 single sided patches which render much quicker than 8 double sided ones. This is a technique used by almost all real time rendering systems.

How do you specify which side is which

Geometry adopts the convention that all patches must have their vertices ordered clockwise when viewed from the correct side. It is the ordering of the vertices which defines which side is the solid looking side. When creating patches you must think about this, if you order them wrongly your object will be visible from the inside, and probably completely invisible from outside.

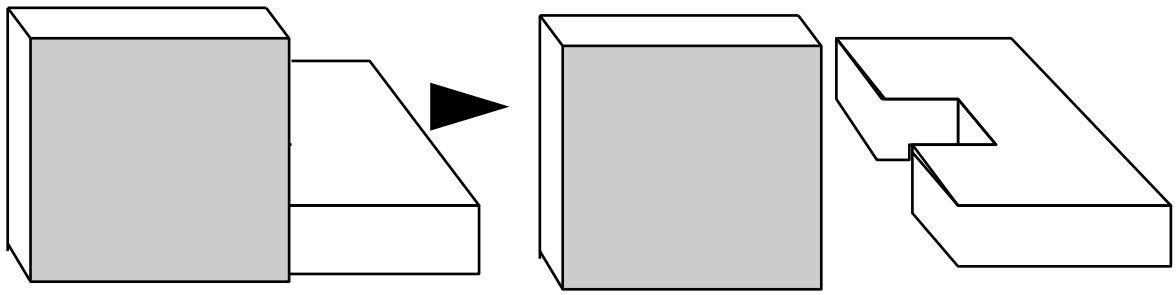


Figure 3. Penetrating patches

Penetrating patches

Care should be taken when constructing objects that one patch should not penetrate another as not all Geometry engines support penetrating surfaces. Although this is not a problem for the default engine it shouldn't be relied upon. If you want an object to look like it has been pierced with another the patches should be designed to give this effect without actually penetrating. The editor's CSG union tool will allow users to do this.

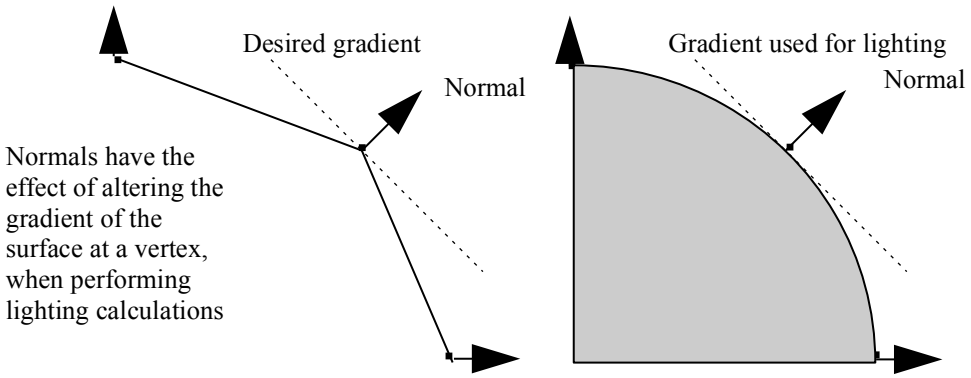


Figure 4. Normals

Normals

A normal is a direction specified in 3D by an x, y and z value, rather like a vertex. Normals are used to make objects appear curved. By associating a normal with each corner of our cube, we are telling the renderer that when it is working out the light shading on those points, to assume that the gradient of the surface at that point is such the normal's direction sticks out at 90 degrees to it. In fact this isn't so, as we might have 3 or more patches converging at a vertex (as in the cube example), but thanks to the normal the shading is generated as if it were a single face at that point. If we tell Geometry that an object we are constructing is to appear curved the 'autosmooth' feature can generate the normal information automatically for us. There are some instance however when we would like to create them explicitly.

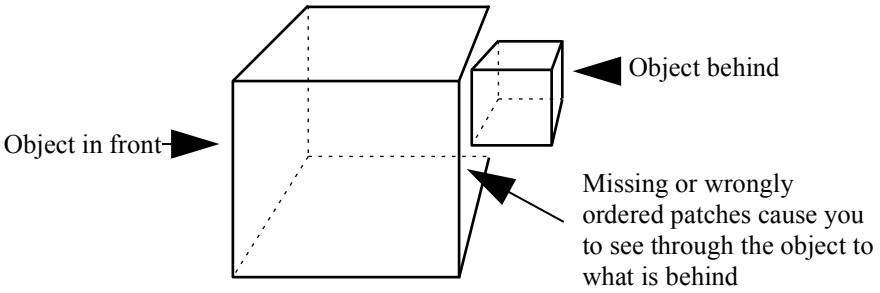


Figure 5. Missing or wrongly ordered patches

Objects

An object is a 3D model constructed out of patches. Since patches are single sided it should be fully enclosed by patches so that we cannot see the inside of any patch from any angle. As mentioned above you would in fact not see it at all giving the appearance of a hole in your object. Another way to think about it is that each edge of every patch should have another patch connected to it. Geometry will not complain if you attempt to render an incomplete object but the results can look confusing seen from some angles.

The editor's 'Enclosed' tool will highlight any unconnected edges.

Lights

As well as a 3D position, lights can have characteristics such as intensity, colour and direction although the default Geometry engine actually only takes notice of the position and intensity.

Ambient Light

Real life lighting very rarely leaves you completely in the dark. This is because there is always light coming from somewhere, e.g. sunlight, moonlight, a far off street lamp etc. In 3D modeller environments we find it difficult to account for all these things, so to ensure that at least a little bit of light falls even on surfaces where no light sources can reach, we use ambient light. The ambient light setting is simply an intensity value added to those generated by the light sources when rendering a scene.

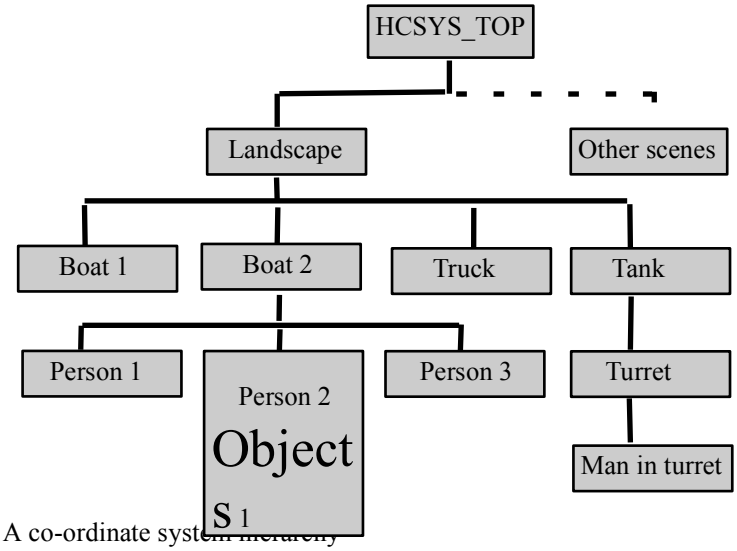


Figure 6. A co-ordinate system hierarchy

Co-ordinate systems

Every 3D co-ordinate you specify for lights, vertices, normals and therefore patches and objects is relative to an origin, i.e. point 0, 0, 0. The position of the origin relative to whatever other stuff might get rendered is defined by the co-ordinate system. All its associated objects, lights and textures can be moved quickly and easily relative to everything else simply by moving the co-ordinate system. The co-ordinate system moves and everything in it moves. As an example, imagine a huge landscape. This would be defined in the top level co-ordinate system by loads of patches. In this landscape there is a river, and on the river a large boat containing lots of people. We want to animate this boat moving down the river. As the boat moves twisting and turning as it goes, so do all the people.
When creating the animation we have to define the position of the boat at each frame, but we shouldn't also have to define the position of every individual person. We do this by creating a co-ordinate system as a child of the landscape's co-ordinate system. and make the boat and everything in it belong it. We then only have to move the co-ordinate system.

Co-ordinate systems are also useful even if we're not making animation's as it provides us with a new origin and axis to work with when designing objects at awkward angles to the x, y and z axis's in the parent co-ordinate system.

How do we define the position of a co-ordinate system

This brings us to the most difficult mathematical concept used in Geometry. The straight answer is; a 4x4 matrix. Some of you will know what I mean, some will require a further explanation.

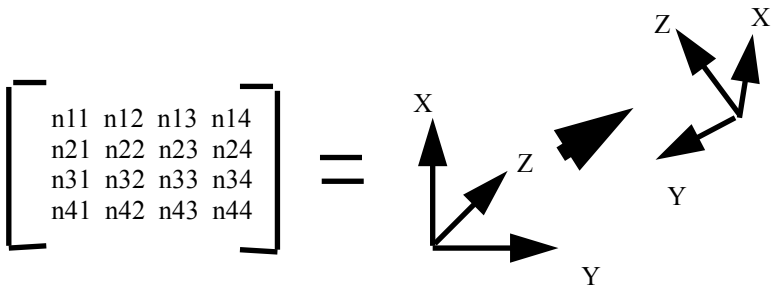


Figure 7. Matrices

Matrices

A matrix is a set of numbers laid out in a rectangle. Matrices can have any number of columns by any number of rows. The matrices we're concerned with in Geometry are all 4x4. When defining the position of a new co-ordinate system its not just a matter of specifying the 3D co-ordinates of a new origin because we also have to define the 'orientation' of the new axis. The 'orientation' means the angle of the new x, y and z axis relative to the parent co-ordinate system. The x axis isn't necessarily parallel to the parents x axis it could be rotated a bit in the y axis and bit in the z axis and even enlarged or shrunk relative to the parent. Matrices were designed to represent this type of information.

Its not obvious from looking at a matrix what type of transformations it represents. Thankfully the maths library supplies a C++ class and a set of C functions that make working with matrices easy.

The earth viewed in the wrong coordinate system would look something like this (back to front). The coordinate system used depends of the x, y and z values of your original data.

Figure 8. Co-ordinate system 'handedness'

Co-ordinate system 'handedness'

Two types of co-ordinate system can be defined; left and right handed. It is necessary to tell Geometry which type you have defined otherwise the clockwise ordering of vertices in a patch is meaningless. To find out which type you want, position your right hand so that you index finger and second finger are at right angles to each other and your thumb is pointing up. In a right handed co-ordinate system if the x axis has the direction of the index finger and the y the direction of the second finger then the z axis will have the direction of the thumb. The same applies to a left handed co-ordinate system if you use your left hand.

The first co-ordinate system you define will be a child of HCSYS_TOP. This is a left handed co-ordinate system, so unless the matrix used to describe the new system specifically reverses either the

x, y or z axis, then this will also be a left handed system. Geometry cannot work this out from the matrix, so it needs to be told.

When constructing your data you must bear in mind the co-ordinate system it belongs too. If for example you designed a model of the earth for a right handed co-ordinate system but displayed it in a left handed system then America would end up on the right and Europe and Asia on the left. No amount of rotating or moving the object would correct this problem. If something like this happens to you then look closely at which type of co-ordinate system you should be using for you data.

Surface Types

All patches have a surface type. A surface type not only describes the colour of the surface but its texture and how highlights appear on the surface. The surface type does not say whether a patch is curved. This is an attribute of the patch itself.

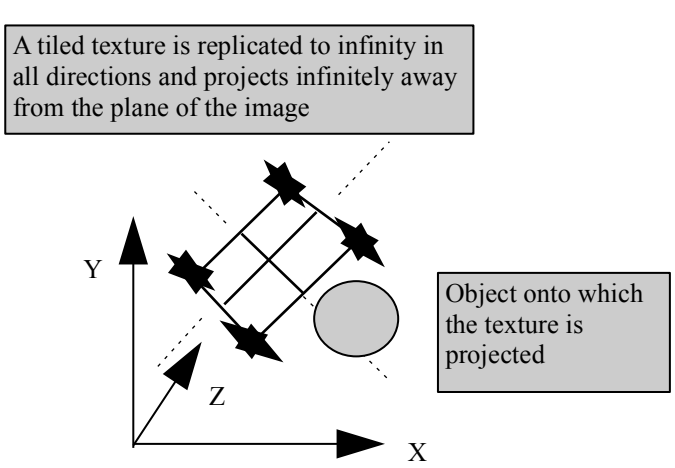


Figure 9. Tiled projected textures

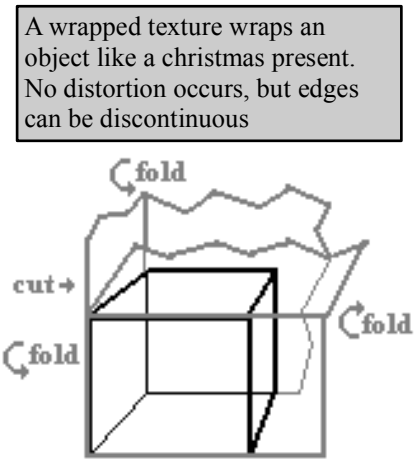


Figure 10. Wrapped textures

Textures

Texturing is a way of making your object look more detailed and realistic by projecting an image onto it. For instance, if you have a wall you can use a brick texture or a wood grain. Several methods of applying texture could be used depending on the underlying geometry engine. Two are described here. The default engine uses the second one.

Projected textures

The way to imagine a projected texture is to think of an invisible image positioned somewhere in your scene. Although you cannot see the image directly, the plane of the image is projected infinitely through the scene in both directions and the image ‘rubs off’ on any patch which uses the surface type of this texture. Think of it a bit like a film projector that shines on anything placed in front of it. The only difference is the film projector image gets bigger the further in front the object or screen is moved. Also with a film projector everything in front gets projected onto, not just selected patches.

Wrapped textures

The default engine uses a technique common to many real-time 3D games. Imagine that the texture is a piece of Christmas wrapping paper and the object being textured is a present. Folding the paper (texture) around a simple object such as a cube, until it is totally covered is a straight forward process. Suppose however that the object is a battleship created from tens of thousands of polygons which are to be wrapped in a rusty metallic texture. The principal is exactly the same.

Both methods have advantages and disadvantages. Wrapping is generally faster because as long as the texture doesn't move it can be rendered many times without needing re-wrapping. You are bound to get discontinuous edges on some textured objects. The default engine does its best to eliminate the effect of this on some objects by carefully selecting how the texture should be folded. The projection method is slower and although it doesn't give discontinuous textures at edges it does so at the expense of distorting the texture. Projection methods do allow for spherical and cylindrical type texture projections. Many other methods exist for applying texture to objects which are not discussed here.

Tiled textures

If you have a huge wall you want to texture and you have a small image of part of a brick wall to use as the texture you have a problem. To make the bitmap fit the wall you have to scale it up. This is quite easily done but now you have a huge wall made of a few huge bricks! What you need to do is make the texture 'tiled'. Genesis can place multiple copies of the image alongside each other and above and below, like tiling an infinitely large bathroom wall. Every point in space will then be within 'range' of the texture.

If you want to use tiled textures you should ensure that the image you use is 'tileable', in other words, if you place multiple copies alongside each other, you should not see the joins. If you can see the joins the wall will look like you've simply pasted up posters of bricks rather than used the real thing.

If you choose not to tile your texture and you have a patch using the surface type which is not in range of the texture, then this patch is coloured using the basic surface colour of the surface type.

Recursive textures

A recursive texture is where each pixel value in your image refers not to a colour in the palette of the image, but rather to another Genesis surface type. This means you could define a surface type which appears to have brick on top and through holes in the brick you can see wood for instance. You can even define pixel values to represent ***real*** holes in your object that show through to objects behind. The default Genesis Geometry engine does not support recursive textures.

Bump mapping

With bump mapping the pixel values in the image refer to the height of a bump which is to appear at that point on any patch its projected onto. Obviously no real physical bump appears on the object, but the shading at the point is modified to make the surface appear irregular at that point. The effect is extremely convincing as long as you don't get too close to the object. The default Geometry engine does not support bump mapping.

Texture orientation and size

Having discussed how textures work, we still haven't addressed the question of how we specify the orientation of the texture. Its quite simple really, as before we use a 4x4 matrix to define the orientation relative to the origin of a co-ordinate system. If you use an identity matrix (this is really a null matrix, one that does nothing) then your texture will be aligned with the x and y axis of the co-ordinate system and will project through the z. The size will be such that if your image is 100 pixels wide by 80 high then it will extend from the origin to x=100, and y=80. Any other matrix can be used to position, rotate, and scale the image away from the origin. This is true regardless of the method the engine uses to apply the texture (projected or wrapped etc.).

Non-linear texture projections

Projection texture matrices can also be used to define non-linear projections. For instance, suppose you want to project bricks onto a sphere. With a normal linear projection as described above, your bricks will distort as they go around the sides of the sphere, like putting a football in front of a film projector. The way around this is to use a spherical mapping. In other words create a matrix to map points on a sphere onto a flat image. Cylindrical, conic and even toriodal projections might come in handy for other types of objects. Wrapped textures will not behave in this way, so this is really an engine dependent feature.

Specular highlights

Specular reflections are what you see when you take a shiny surface such as a metal tray, and angle it towards a light. You will often see brightly lit areas where the light bounces off the object towards your eye. The exact shape, size and brightness of these highlights depends on what the surface is made of. Since the default Geometry engine (or any using Gouraud as opposed to Phong shading) performs its lighting calculations at each vertex only, the more vertices you have the more accurately the lighting will mimic the surface type. Even if you have a flat square surface, to get the most visually accurate highlights the surface should be made of lots of small flat patches rather than one large one.

In technical terms the specular reflection characteristics relates the amount of light reflected to the angle of incidence (the angle made by the light striking a point on the surface and the normal of that point).

Transparent surfaces

As well as all the above a surface can have associated with it a transmissive value, in other words its degree of transparency. A value of 1.0 would make the surface completely invisible whereas 0 would be completely solid. The default Geometry engine does not support transparent surfaces.

Using the API

Custom tool writers have an easy life as the editor does most things for you. Most of the time you will be using only a handful of calls specifically to do with create vertices and patches etc. However a full understanding of how the API works is useful for when you want to expand your tool writing to cover more advanced topics. See the separate section 'Writing custom tools'.

Application writers using the API will need a full understanding of the API, as will those developing new Geometry engines.

The first call to the Geometry DLL must be Initialise, and it can only be called once. The last call must be Terminate. No furthers calls can be made after a Terminate. After Initialise the next likely call that a standalone application is likely to make will probably be AddCoorSystem, since every object must belong to a co-ordinate system. If we're writing a tool this would have been done for us by the editor. As explained co-ordinate systems are built up in hierarchies. A top level co-ordinate system has HCSYS_TOP (a constant defined in 'geometry.h') as its parent. Other co-ordinate systems defined underneath these are known as its children.

After creating some patches you can render an image by placing the camera into one of these co-ordinate systems using the SetCamera call and then call Render. Again, custom tools writers wont need to worry about this. When a co-ordinate system is rendered everything in it, its child co-ordinate systems and everything in them, and its parent co-ordinate systems and everything in them are rendered. The only thing which isn't rendered are other top level co-ordinate systems. So each top level co-ordinate system can be considered a completely separate scene, much like separate documents in a word processor. The one that gets rendered is the one we SetCamera into.

The image gets rendered into a windows Device Independent Bitamp (DIB) which is created by geometry's Render function. The application can maintain as many DIBs or rendered images as it likes. Notice how the editor does this. It has at least one for each top level co-ordinate system. When the window's window needs to be painted (on the WM_PAINT message) with the bitmap, we can call another Geometry function, UpdateImage, to transfer the image to the screen.

Geometry API Reference

To use the Geometry API either from your own application or from a custom tool, you must include the 'geometry.h' C++ include file and link to 'geometry.lib'. All Geometry functions return a GeomErr value. Usually this will be GERR_OK (value 0) if the function succeeded otherwise will either represent an internal processing error such as 'out of memory', or information such as 'vertex not visible'. The calling code should check these return codes and display any serious internal errors to the user in a message box. The GetErrorText function can be used to get a complete text description of the error and whereabouts internally it occurred. Internal processing errors more often than not indicate a bug in the calling code, such as trying to create a patch out of just two vertices. When your code is debugged they should go away. If serious errors persist when they shouldn't do we would be grateful if you could fill in a bug report form and send it to us, in order for us to correct bugs in subsequent releases. We will also endeavour to send you an update as soon as the bug is fixed.

Many functions accept handles to objects. It is the responsibility of the tool/application writers to ensure that these handles are valid, because although there is a 'invalid handle' error, you cannot rely on the Geometry engine being able to check its validity.

Use of C++ maths classes

Often a Geometry function will require a C++ class defined in the maths library as a parameter, a Vec for instance would be passed to AddVertex. A pure C interface will soon be made available where instead you will pass a pointer to a maths 'Vector' structure.

General functions

GeomErr Initialise (void);

Comments

Called to initialise Geometry. Must be called first.

Return codes

GERR_OUT_OF_MEMORY

GeomErr Terminate (void);

Comments

Called to terminate Geometry and release all its allocated resource. Must be called last.

Return codes

GeomErr DefSurfType (SurfType *pst,
bool bReuse,
HanSurf *phsur);

pst Pointer to a SurfType structure.

bReuse If TRUE this call will return the handle of an identical type if one exists.

phsur Returns a handle to a surface type.

Comments

Creates a surface type definition. See the SurfType structure definition for more details. The user data field of the SurfType structure is ignored and is set to zero by the engine. To set this to another value the SetUserData call is used. The hsurNext field of the SurfType structure is also ignored but is used when querying an existing surface type.

Return codes

GERR_OUT_OF_MEMORY
GERR_TOO_MANY_SURFACE_TYPES
GERR_BITMAP_FILE_NOT_FOUND
GERR_NOT_A_BMP_FILE

GeomErr DelSurfType (HanSurf hsur);

hsur Handle to surface type to delete.

Comments

Deletes a surface type as long as no patches are using it.

Return codes

GERR_INVALID_HANDLE
GERR_IN_USE

GeomErr QrySurfType (HanSurf hsur,
SurfType *pst,
BITMAPINFOHEADER *pbmih,
ulong *pulNumPats);

hsur Handle to surface type to query.

pst SurfType structure to receive the information about the surface.

pbmih Pointer to windows BITMAPINFOHEADER structure which will receive details of the texture bitmap if appropriate.

pulNumPats Number of patches using this surface.

Comments

Queries information about a given surface type.

Return codes

GERR_INVALID_HANDLE

GeomErr ModSurfType (HanSurf hsur,
SurfType *pst);

hsur Handle to surface type to modify.

pst SurfType structure used to set surface characteristics.

Comments

Modifies a surface type.

The user data field in the SurfType structure is ignored. The only way to change the user data of a surface type is using the SetUserData call.

Return codes

GERR_INVALID_HANDLE

GeomErr GetFirstSurfType (HanSurf *phsur);

phsur Handle of the first surface type defined in geometry.

Comments

Returns the handle of the first surface type defined. To enumerate all other surface types, query this one and use the hsurNext field of the SurfType structure to query the next, and so on.

Return codes

GeomErr SetCamera (HanCoorSys hcsys,
Vec &vecPos,
Vec &vecDir,
Vec &vecUp,
ViewParams *pvp);

hcsys Co-ordinate system to set the camera in.

vecPos The position of the camera within this co-ordinate system.

vecDir The direction in which the camera is facing. It takes the form of an absolute 3D position within the co-ordinate system.

vecUp The direction which will be 'up' in the rendered image. It takes the form of an absolute 3D position within the co-ordinate system.

pvp Defines the viewing parameters such as viewing angle and depth of view of the camera.

Comments

Sets the camera at a position within the co-ordinate system. The camera can only be in one co-ordinate system at a time. Setting it in another co-ordinate system will remove it from this one. If you need to use Get3DLine and Get3DPoint to get 2D screen co-ordinates of 3D points, you must SetCamera first into the appropriate co-ordinate system.

Technical note: Apart from defining the viewing parameters, and setting which top level co-ordinate system gets rendered, this call is really nothing more than a convenient way of modifying the matrix of the top level co-ordinate system. Each top level co-ordinate system is a scene in its own right. Each is a child of HCSYS_TOP which is in fact the *camera itself*! The Z axis of HCSYS_TOP is the axis along which the camera points. Setting the camera in any co-ordinate system simply has the effect of modifying the matrix of the top level child in the scene, such that the camera *appears* to be at the position given within the specified system. In fact the camera is not really in this system at all, it is the great granddad of *all* co-ordinate systems! With a clever piece of coding you could achieve the same

result by using ModCoorSys to modify the matrix of the top level child in the scene. NB. It is not essential to understand this concept in order to program the API.

Return codes

GeomErr Render (BITMAPINFO *huge *ppbmi);

ppbmi Pointer to a pointer to a windows DIB.

Comments

Before rendering the *ppbmi value should be set to NULL. This will cause Geometry to create a DIB of the size specified in the ViewParams structure of the last SetCamera call and return a pointer to it in *ppbmi. On subsequent renders if we pass in the same value of *ppbmi Geometry will reuse the bitmap. If the requested size changes on any subsequent SetCamera call or circumstances within Geometry change, for instance, implementation specific options might allow your Geometry engine to switch between 8 and 24 bit output, then the old bitmap is freed and a new bitmap created. Each time we pass a value of NULL, we force a new bitmap to be created. To transfer the image to a device context such as the screen, call the UpdateImage function.

The render bitmaps created by this call are removed by Geometry when Terminate is called, however since these bitmaps can be very large it is advisable to remove them if no longer required by calling DeleteRenBitmap.

If the render resulted in visible clipping at the front clipping plane then GERR_VISIBLE_FRONT_CLIP is returned. This should not be interpreted as an error. This is useful for applications such as flight simulators where the camera, viewpoint, cockpit or whatever you want to call it, can *crash* into objects in the scene. If any visible clipping occurs at the front clipping plane then this can be considered as crashing into the object. The application should be aware however that if you approach a large patch and in a single render step move from one side to the other, then geometry will not return this code. In such cases the application might need to perform additional checking.

The default Geometry engine is very sensitive about the ‘handedness’ of the co-ordinate systems being rendered. When creating a co-ordinate system it has no way of telling if the co-ordinate system you’ve created is really of the type you’ve told it. Since Geometry’s renderer relies on this information to optimise its performance it is likely to go wrong if say, you created a left hand co-ordinate system but told Geometry it is right handed. If you experience objects which appear inside out when rendering check your co-ordinate systems very carefully.

Return codes

GERR_IMAGE_SIZE_NOT_MULT_4
GERR_BITMAP_TOO_SMALL
GERR_TOO_MANY_BITMAPS
GERR_VISIBLE_FRONT_CLIP

GeomErr DeleteRenBitmap (BITMAPINFO huge *pbmi);

pbmi A pointer the BITMAPINFO structure created by the Render call.

Comments

Tells Geometry to remove a bitmap created by the Render call.

Return codes

GeomErr UpdateImage (ulong ulHDC,
BITMAPINFO huge *pbmi,
ushort usX,
ushort usY);

ulHDC A ulong value holding the HDC to paint the rendered image to.

pbmi Pointer to the rendered DIB (created by calling Render).

usX Horizontal position in the DC of where the image will appear.

usY Vertical position in the DC of where the image will appear.

Comments

Transfers a rendered DIB to a device context. This call could result in realising the colour palette.

Return codes

GeomErr SetUserData (Handle han,
HandleType htype,
ulong ulUser);

han Handle of the item to set the user data for.

htype The type of the 'han' handle;

COORSYS_HANDLE
OBJECT_HANDLE
PATCH_HANDLE
VERTEX_HANDLE
NORMAL_HANDLE
SURFACE_HANDLE
LIGHT_HANDLE

ulUser The user data value.

Comments

This routine is provided so that the application can associate instance data with a particular geometry object, patch, light etc. This 32 bit data value can be used to store a pointer to further data about the object. For instance the application might associate a name with each object or light. If the item is deleted either directly or indirectly (e.g. patches get deleted as a result of DelObject) geometry will call the application back giving it a chance to remove its instance data.

Return codes

GeomErr SetDelCallback (fnDelCallback *DelCallback);

DelCallback The address of a callback function which is called when any geometry item is deleted.

Comments

This routine is used to set the address of a callback function which is called every time any geometry item (e.g. objects, lights, patches etc.) is deleted. The application could associate instance data with a geometry item which would need removing when the item is deleted.

The callback function is called *before* the item is deleted so the handle is still valid.

Return codes

GeomErr GetErrorText (GeomErr gerr,
char *szBuff,
ushort usBuffSize);

gerr The GeomErr value returned from a Geometry call.

szBuff Pointer to a character buffer to hold the error string.

usBuffSize Set to indicate the size of szBuff.

Comments

Can be called to get Geometry to return an error string describing the error which occurred. The string will contain the name of the module and the line number on which the error occurred.

Return codes

GERR_BUFFER_TOO_SMALL

Co-ordinate system functions

GeomErr AddCoorSys (HanCoorSys hcsysParent,
Mat &matToParent,
Mat &matFromParent,
ushort usType,
HanCoorSys *phcsys);

hcsysParent Handle to parent co-ordinate system

matToParent Matrix describing the transformation from child to parent. In other words this matrix transforms a point in the child co-ordinate system to its corresponding position relative to the parent co-ordinate system.

matFromParent The inverse or opposite matrix to matToParent. NB. The maths library can easily generate the inverse of a matrix by preceding it with a minus sign (e.g. -mat).

usType This value indicates whether the co-ordinate system being defined is right handed or left handed. It should be either CT_RIGHTHAND or CT_LEFTHAND.

phcsys Returns a handle to a new co-ordinate system.

Comments

Creates a co-ordinate system as a child of the parent. Use HCSYS_TOP to create a top level co-ordinate system. If the matrices do not specifically reverse one of the co-ordinate system axis's then this co-ordinate system will be the same 'handedness' as its parent. It is important that the usType parameter correctly identifies the type of the co-ordinate system.

The matrices can define scaling and shearing as well as basic orientation. However a matrix is not acceptable if it alters the basic topology of the geometry. For instance, a mirror type matrix will alter the 'type' of the co-ordinate system (only use this if you also specify the correct usType parameter). A matrix that warps space into say, a cone or cylindrical shape is completely unacceptable. A matrix consisting entirely of zeros, would be the Genesis equivalent of a black hole. In space physical laws break down inside black holes. Genesis will also break down if you use a matrix like this!

Return codes

GERR_OUT_OF_MEMORY

GeomErr DelCoorSys (HanCoorSys hcsys);

hcsys Co-ordinate system to delete.

Comments

Deletes a co-ordinate system and any objects belonging to that system including child co-ordinate systems and their objects.

Return codes

GeomErr QryCoorSys (HanCoorSys hcsys, CoorSysInfo *pcsi);

hcsys Co-ordinate system to query.

pcsi Pointer to a CoorSysInfo structure to receive the information.

Comments

Returns information about a co-ordinate system and all its associated objects enabling us to navigate our way around the co-ordinate system hierarchy.

If you query HCSYS_TOP, the only relevant field is usType, which will be left-hand.

If you query any top level co-ordinate systems (i.e. direct children of HCSYS_TOP) you cannot rely on the matrix fields of the CoorSysInfo structure being what you originally set them to. The reason for this apparent oddity is that under the covers these matrices define the position of the scene relative to the camera, so as the camera 'moves', so these matrices change.

Return codes

GeomErr ModCoorSys (HanCoorSys hcsys,
Mat &matToParent,
Mat &matFromParent,
ushort usTransType,
ushort usNewType);

hcsys Co-ordinate system to query.

matToParent Matrix to define the new position of the co-ordinate system relative to its parent. It can be used either to replace the old matrix or to modify it.

matFromParent Inverse of matToParent. This must be supplied. If it is not known you can always pass in (-matToParent).

usTransType Indication of how the new matToParent matrix is to modify the existing one. Can be one of the following values;

MCS_TRANS_LAST	The new matrix is applied as; old_matrix*new_matrix
MCS_TRANS_FRIST	The new matrix is applied as; new_matrix*old_matrix
MCS_TRANS_REPLACE	The new matrix simply replaces the old

usNewType The new type of the co-ordinate system as defined by the transformation. Zero can be used if the transformation hasn't changed type.

Comments

Modifies the position of a co-ordinate system relative to its parent. If the transformation results in the type of the co-ordinate system changing (e.g. from right-hand to left-hand) then we must tell the engine the new type. If the co-ordinate system contains any objects or child co-ordinate system then its type cannot be changed.

If the matrix of a top level child co-ordinate system (any child of HCSYS_TOP) is changed and queried later it cannot be guaranteed to be the same. The reason for this is that the SetCamera call also modifies the matrix the top level child of the scene it is set in (see the technical note on the SetCamera call).

Return codes

GERR_INVALID_OPP_IN_TOP
GERR_CANT_CHANGE_TYPE

GeomErr Get3DLine (HanCoorSys hcsys,
Vec &vecA,
Vec &vecB,
float *pfStartX,
float *pfStartY,
float *pfEndX,
float *pfEndY);

hcsys Handle of co-ordinate system points belong to.

vecA Position in co-ordinate system of first point.

vecB Position in co-ordinate system of second point.

pfStartX Returns the screen X co-ordinate of start of line.

pfStartY Returns the screen Y co-ordinate of start of line.

pfEndX Returns the screen X co-ordinate of end of line.

pfEndY Returns the screen Y co-ordinate of end of line.

Comments

Returns the 2D screen co-ordinates of a line segment defined by two points; vecA and vecB, within the co-ordinate system hcsys. This routine can be used to generate lines corresponding to points in a rendered image which can then be superimposed on the rendered image. This way it makes it look as if Geometry's renderer can support 3D lines as well as patches. Of course this is not quite true as the lines do not go through the hidden surface/line process.

Unlike the 2D screen co-ordinates returned on calls like QryVertex, this routine doesn't require a Render to have taken place, however it does require that the camera is set in an appropriate co-ordinate system for the point to be visible, otherwise GERR_NOT_VISIBLE is returned.

Return codes

GERR_NOT_VISIBLE

GeomErr Get3DPoint (HanCoorSys hcsys,
Vec &vec,
float *pfX,
float *pfY);

hcsys Handle of co-ordinate system points belong to.

vec Position of point in co-ordinate system.

pfX Returns the screen X co-ordinate of the point.

pfY Returns the screen Y co-ordinate of the point.

Comments

Returns the 2D screen co-ordinates of a point; vec, within the co-ordinate system hcsys.

Unlike the 2D screen co-ordinates returned on calls like QryVertex, this routine doesn't require a Render to have taken place, however it does require that the camera is set in an appropriate co-ordinate system for the point to be visible, otherwise GERR_NOT_VISIBLE is returned.

Return codes

GERR_NOT_VISIBLE

Object construction functions

GeomErr AddObject (HanCoorSys hcsys,
float fNewVertTol,
HanObject *phobj);

hcsys Handle to co-ordinate system to add object to.

fNewVertTol Vertices in this object cannot be closer than this value in all of x, y and z.

phobj Returns a handle to the new object.

Comments

Adds a new object to a co-ordinate system.

When new vertices are added to this object a check is made to see if it is closer than fNewVertTol in x, y and z. If it is the vertex is re-used. Notice that fNewVertTol is not the distance from the existing vertices, this is given by the formula; MaxDistFromVert=sqrt((NewVertTol^2)*3).

Return codes

GeomErr DelObject (HanObject hobj);

hobj Handle to the object being deleted.

Comments

Deletes an object along with all its patches, vertices and normals from a co-ordinate system.

Return codes

GeomErr QryObject (HanObject hobj, ObjectInfo *poi);

hobj Handle to the object to query.

poi Points to an ObjectInfo structure to receive the information.

Comments

Returns information on an object.

Return codes

GeomErr MoveObject (HanObject hobj, Vec &vec);

hobj Handle to the object to move.

vec The vector to move the object along.

Comments

This function moves an entire object within its co-ordinate system. The most efficient way of moving objects for instance, during a real-time animation, is to assign the object to its own co-ordinate system which is then moved relative to its parent. If however an object has to be moved within the system, each vertex is turn has to moved.

Although less efficient than moving a co-ordinate system, this routine is far more efficient than calling ModVertex for each vertex. The reason for this is that each time a single vertex is moved the engine has to check if any of the patches using it has four or more vertices. If they do the chances are that the vertices in these patches no longer lie on a plane and these patches might have to be split in two to accommodate the move. If every vertex in the object moves by the same vector this is not necessary so using this function saves on processing and prevents patches being split in two.

Return codes

GeomErr ScaleObject (HanObject hobj, float fScale);

hobj Handle to the object to scale.

fScale The factor to scale the object by.

Comments

This function scales an entire object within its co-ordinate system. The most efficient way of scaling objects for instance, during a real-time animation, is to assign the object to its own co-ordinate system which is then scaled relative to its parent. This routine is however more efficient than moving each vertex in the object (see MoveObject for more details).

Return codes

GeomErr PrtObject (HanObject hobj);

hobj Handle to the object to print.

Comments

Used purely as a debugging aid for when writing Geometry engines. This routine can be called to print out a text description of an objects data structures. It can be invoked from the editor using a special key combination. Unless compiled with the _DEBUG macro definition this routine should do nothing.

Return codes

GeomErr AddVertex (HanObject hobj,
Vec &vec,
HanVertex *phver);

hobj Handle to object to add vertex to.

vec 3D position of the vertex within the co-ordinate system.

phver Returns a handle to the vertex.

Comments

Creates a new vertex belonging to the object.

Return codes

GeomErr DelVertex (HanObject hobj, HanVertex hver);

hobj Handle to object we want to delete the vertex from.

hver Handle to the vertex to be deleted.

Comments

Deletes a vertex from an object. The vertex is only deleted if no patches are using it.

Return Codes

GERR_IN_USE

GeomErr QryVertex (HanObject hobj,
HanVertex hver,
VertInfo *pvi);

hobj Handle to object containing vertex to query.

hver Handle to the vertex to query.

pvi Pointer to a VertInfo structure to receive the information.

Comments

Returns information about a vertex. The fScrnX and fScrnY elements of the VertInfo structure contain the screen x and y co-ordinates of this vertex the last time a render was performed. If the vertex was not visible for whatever reason, e.g. it was out of view, or the last render was applied to a different co-ordinate system, then GERR_NOT_VISIBLE is returned.

The default Geometry engine will indicate that the vertex was visible if it was within the view and used by a patch which was facing the viewpoint even if the vertex was obscured by a closer object. If on the other hand the vertex was on the back face of an object, it will indicate that it was not visible.

Return codes

GERR_NOT_VISIBLE

GeomErr ModVertex (HanObject hobj,
HanVertex hver,
Vec &vec);

hobj Handle to object containing vertex to modify.

hver Handle to vertex to modify.

vec New position.

Comments

Modifies the position of a vertex. The next time a render is performed the vertex will appear in its new position. If the new position affects a patch with 4 or more sides, such that it no longer lies on a plane, then the patch will be split into two or more patches.

To move or scale an entire object the MoveObject and ScaleObject functions should be used.

Return codes

GERR_COINCIDENT_POINTS
GERR_THIN_SEGMENT
GERR_COMPLEX_OUTLINE

GeomErr AddNormal (HanObject hobj
Vec &vec,
HanNormal *phnor);

hobj Handle to the object to add the normal to.

vec 3D direction vector of the normal. It specifies a 3D 'direction' rather than an absolute point in 3D space.

phnor Returns a handle to the new normal.

Comments

Adds a normal to an object.

Return codes

GeomErr DelNormal (HanObject hobj, HanNormal hnor);

hobj Handle to the object containing the normal to delete.

hnor Handle to the normal to delete.

Comments

Deletes a normal from an object as long as no patches are using it.

Return Codes

GERR_IN_USE

GeomErr QryNormal (HanObject hobj,
HanNormal hnor,
NormInfo *pni);

hobj Handle to the object containing the normal to query.

hnor Handle to normal to query.

pni Pointer to a NormInfo structure to receive the information.

Comments

Returns information about a normal. Since a single normal can be used by vertices from many patches we need to supply the handle of a vertex in order to generate the 2D screen co-ordinates of the normal when used by that vertex. The normal makes a line extending from the vertex to a distance of 1 co-ordinate system unit away in the 3D direction given when the normal was created. If this line or any part of it was visible at the last render, its 2D screen co-ordinates are returned, otherwise the return code is GERR_NOT_VISIBLE.

Return Codes

GERR_NOT_VISIBLE

GeomErr ModNormal (HanObject hobj,
HanNormal hnor,
Vec &vec);

hobj Handle to the object containing the normal.

hnor Handle to normal to modify.

vec New direction vector.

Comments

Modifies the direction of a normal vector and therefore the surface gradient at all vertices using the normal. At the next render the shading at all those vertices using the normal will have changed to reflect the new surface gradients.

Return codes

GeomErr QryEdge (HanObject hobj,
HanEdge hedg,
EdgeInfo *pei);

hobj Handle to the object containing the edge.

hedg Handle to the edge to be queried.

pei Pointer to an EdgeInfo structure to receive the information.

Comments

Returns information about an edge. Although we do not explicitly create the edges, we create vertices and patches and Geometry takes care of the edges, it is sometimes useful to have access to the edges. Suppose we want to draw a wire frame of the object (exactly as the editor does). We could iteratively query each patch in the object, query the screen positions of each of its vertices in turn and draw them.

However each edge of each patch is also used by another patch meaning each edge (and therefore the entire object) will get drawn twice. Querying the edges directly avoids this.

Return codes

```
GeomErr DefPatch (HanObject hobj,
                  ushort usNumEdges,
                  ushort usFlags,
                  float fSmoothAng,
                  HanVertex *ahver,
                  HanNormal *ahnor,
                  HanSurf hsur,
                  ushort *pusNumPats,
                  HanPatch *ahpat);
```

hobj Handle to object to add patch to.

usNumEdges The number of edges, and therefore vertices and normals in the patch. This can be any number from 3 upwards. If the Geometry engine does not support patches of more than say 3 or 4 edges then the patch will be automatically split up into more than one patch resulting in more than one handle being returned.

usFlags Flags controlling the creation of the patch. It can be any of the following values combined with C's OR operator ('|');

DP_AUTOSMOOTH	Signifies that the patch should appear curved even although we have not defined any normals. The normals should be created automatically by Geometry by averaging out the plane normals of all patches converging on a vertex.
DP_SMOOTH_BY_SURF	Signifies that autosmoothing is applied only to patches adjoining this one if they have the same surface type.
DP_SMOOTH_BY_ANGLE	Signifies that autosmoothing is applied only to patches adjoining this one if they angle made at the join is less than that given by the fSmoothAng parameter.
DP_DONT_VALIDATE	Signifies that this patch should not be validated. If your code is fully debugged and tested it can use this flag to say, 'I am confident that this patch definition is OK'. Using this flag speeds up the creation process, otherwise the following checks are performed; • No two points can be in exactly the same spot • All points do not lie along a straight line • All points lie on a plane (or within a small tolerance of)
DP_CONVEX	Signifies that the polygon is convex (has no internal angles greater than 180 degrees). This flag should not be used if we're not certain.

As with DP_DONT_VALIDATE it can speed up the creation process, (depending on the Geometry engine).

DP_ATTEMPT_TO_FIX_PLANE Signifies that if the points are not quite on a plane then we should split the outline into two or more planes in order to be able to create it. If the points do not lie anywhere near a common plane then geometry will not be able to discern any kind of clockwise/anticlockwise ordering and will still generate a GERR_NOT_ON_A_PLANE error code.

fSmoothAng Specifies the angle to use for autosmoothing expressed in radians. Only takes effect if DP_SMOOTH_BY_ANGLE is used.

ahver Array of handles to vertices. The number of vertices is given by the usNumEdges parameter. The vertices must be specified in a clockwise order when viewing the patch from the correct side. No attempt should be made to define a patch using vertices belonging to another object.

ahnor Array of handles to normals. The number of normals is given by the usNumEdges parameter. Each normal relates to the corresponding vertex in ahver, e.g. ahnor[0] applies to aver[0] etc. This parameter is ignored if the DP_AUTOSMOOTH flag is given. No attempt should be made to define a patch using normals belonging to another object.

hsur Handle to the surface type to be used for the patch.

pusNumPats A pointer to a ushort used to return the number of patches actually created (Large concave outlines will almost definitely be split up into more than one patch). Before calling the ushort should be set to the size of the ahpat buffer.

ahpat Buffer to hold the handle(s) of the patch(es) created.

Comments

Creates a patch out of 3 or more predefined vertices. Any number of vertices can be used to define the outline of the patch as long as they lie on a plane, do not form a straight line or cross over at any point.

The autosmooth feature takes the worry out of having to define normals for curved surfaces. If the patch is deleted the vertices and normals defining it are not deleted. This is true even if the autosmooth feature was used to create the normals automatically. In this case the patch should be queried to find the handles of the normals it uses before it is deleted and then the normals can be deleted using DelNormal.

The outline may be split into more than one patch depending on how many vertices the Geometry engine can use in a single patch and whether or not it can support concave patch outlines. The default Geometry engine supports patches of no more than four vertices and they must be convex. An attempt to create a patch with more than four vertices, or with a concave outline will result in more than one patch being created.

Return codes

GERR_COINCIDENT_POINTS
GERR_THIN_SEGMENT
GERR_NOT_ON_PLANE
GERR_OUT_OF_MEMORY
GERR_NOT_ENOUGH_POINTS
GERR_COMPLEX_OUTLINE

GeomErr DelPatch (HanObject hobj, HanPatch hpat);

hobj Handle to the object containing the patch to delete.

hpat Handle to the patch to delete.

Comments

Deletes a patch from an object. Deleting a patch does not delete any vertices or normals used by the patch. Also be aware that even if the autosmooth feature was used to automatically create normals for the patch, these will not be deleted either. To delete these the patch should be queried to find the handles of the normals which can then be deleted *after* the patch.

Return codes

GeomErr QryPatch (HanObject hobj,
HanPatch hpat,
PatchInfo *ppi);

hobj Handle to the object containing the patch to query.

hpat Handle to the patch to query.

ppi Pointer to a PatchInfo structure to hold the information.

Comments

Returns information about a patch. The ahver and ahnor elements of the PatchInfo structure should point to buffers to receive the handles of the vertices and normals and the usNumEdges element should indicate how many handles there is room for in the buffers. NB. No Geometry engine implementation should have more than MAX_PATCH_EDGES edges in any patch, so it is a good idea to set usNumEdges to this. The ahver and ahnor fields can be set to NULL if we're not interested in the handles. If GERR_BUFFER_TOO_SMALL is returned the usNumEdges field of the PatchInfo structure will contain the size required.

Return codes

GERR_BUFFER_TOO_SMALL

GeomErr QryVertPatch (ulong *pulRef, HanPatch *phpat);

pulRef Pointer to a ulong holding a reference to a patch. This value is returned by the ulRef element of the VertInfo structure on a call to QryVertex.

phpat Returns a handle to the next patch using this vertex.

Comments

To query which patches use a particular vertex, first query the vertex, and then pass the ulRef value to QryVertPatch. This will return a handle to the first patch using that vertex. Subsequent calls to QryVertPatch will return other patches until a NULL_HANDLE is returned.

Return codes

GeomErr ModPatchSurf (HanObject hobj,
HanPatch hpat,
HanSurf hsur)

hobj The object the patch belongs to.

hpat The patch to change.

hsur The surface type to colour the patch with.

Comments

Changes the surface type used by a patch.

Return codes

Lighting functions

GeomErr AddLight (HanCoorSys hcsys,
LightDef *pld,
HanLight *phli);

hcsys Handle to co-ordinate system to add light to.

pld Points to a LightDef structure containing details of the light.

phli Returns a handle to a new light.

Comments

Creates a new light belonging to the co-ordinate system. The user data field of the LightDef structure is ignored and is set to zero by the engine. To set this to another value the SetUserData call is used. The hliNext field of the LightDef structure is also ignored but is used when querying an existing light.

Return codes

GeomErr DelLight (HanLight hli);

hli Handle to the light to be deleted.

Comments

Deletes a light from the co-ordinate system.

Return Codes

GeomErr QryLight (HanLight hli, LightDef *pld);

hli Handle to light to query.

pld Pointer to a LightDef structure to receive the information.

Comments

Returns information about a light.

Return codes

GeomErr ModLight (HanLight hli, LightDef *pld);

hli Handle to light to modify.

pld Pointer to LightDef structure containing the new parameters.

Comments

Modifies the characteristics of a light, including possibly its position.

The user data field in the LightDef structure is ignored. The only way to change the user data of a light is using the SetUserData call.

Return codes

GeomErr SetAmbient (float fInt);

fInt Ambient light intensity.

Comments

Sets the intensity of the ambient light in a scene. The ambient value applies to all co-ordinates systems.

Return codes

GERR_INVALID_LIGHT_INT

Loading and Saving

GeomErr SaveScene (HFILE hfile,
HanCoorSys hcsys,
ulong *pulNumCSys,
HanCoorSys *ahcsys,
ulong *pulNumObjs,
HanObject *ahobj,
fnLoadSaveCallback *Report);

hfile	Handle of a file to save to.
hcsys	Co-ordinate system to save.
pulNumCSys	Indicates the size of the ahcsys buffer in terms of how many handles it can hold. On return this value is modified to the total number of co-ordinate systems saved, and therefore the number of handles in the ahcsys array.
ahcsys	An array which on return contains the handles of all the co-ordinate systems saved in the order in which they were saved in the file. If you are not interested in the handles, this parameter can be NULL if we set *pulNumCSys to zero.
pulNumObjs	Indicates the size of the ahobj buffer in terms of how many handles it can hold. On return this value is modified to the total number of objects saved, and therefore the number of handles in the ahobj array.
ahobj	An array which on return contains the handles of all the objects saved in the order in which they were saved in the file. If you are not interested in the handles, this parameter can be NULL if we set *pulNumObjs to zero.
Report	The address of a callback function to inform the application of how far through the load its done (see the fnLoadSaveCallback function).

Comments

Saves a co-ordinate system to the file given by hfile. All objects and lights belonging to this co-ordinate system will be saved including any children and all their objects. The last four parameters will return the
handles of all co-ordinate systems and objects saved. This is very useful for applications like the editor. The editor associates a name with each co-ordinate system and object. These names are unknown to Geometry and so will not get saved. However the editor can save the names associated with each handle ***after*** the main Geometry information in the file. When the file gets loaded again by LoadScene we get told the new handles which we then simply have to associate with the names in the file.

Return codes

GeomErr LoadScene (HFILE hfile,
HanCoorSys hcsys,
Char *szTexPath,
ulong *pulNumCSys,
HanCoorSys *ahcsys,
ulong *pulNumObjs,
HanObject *ahobj,
fnLoadSaveCallback *Report);

hfile	Handle of a file to load from.
hcsys	Co-ordinate system to load under.
szTexPath	A pointer to a path specification for where to search for texture bitmaps if they are not found in the current working directory. This would typically be set to the directory where the model file is, or else a special texture directory. Any number of paths can be seperated by semi colons, but each must have a terminating back slash e.g.; "c:\genesis\textures\;c:\windows\bmps\" or else be a null string. The pointer cannot be NULL.
pulNumCSys	Indicates the size of the ahcsys buffer in terms of how many handles it can hold. On return this value is modified to the total number of co-ordinate systems loaded, and therefore the number of handles in the ahcsys array.
ahcsys	An array which on return contains the handles of all the co-ordinate systems loaded in the order in which they existed in the file. If you are not interested in the handles, this parameter can be NULL if we set *pulNumCSys to zero.
pulNumObjs	Indicates the size of the ahobj buffer in terms of how many handles it can hold. On return this value is modified to the total number of objects loaded, and therefore the number of handles in the ahobj array.
ahobj	An array which on return contains the handles of all the objects loaded in the order in which they existed in the file. If you are not interested in the handles, this parameter can be NULL if we set *pulNumObjs to zero.
Report	The address of a callback function to inform the application of how far through the load its done (see the fnLoadSaveCallback function).

Comments

Loads a co-ordinate system from the file given by hfile. All objects and lights belonging to this co-ordinate system will be loaded including any children and all their objects. The top level co-ordinate system in the file will become a child of hcsys. To load an entirely new scene hcsys should be set to HCSYS_TOP. For a fuller description of the last four parameters see SaveScene.

Return codes

GERR_INVALID_GEN_FILE

GeomErr LoadLWObject (HFILE hfile,
HanCoorSys hcsys,
char *szTexPath,
HanObject *phobj,
fnLoadSaveCallback *Report);

hfile	Windows handle of the file to load from.
hcsys	Handle of the co-ordinate system the object will be added to.
szTexPath	A pointer to a path specification for where to search for texture bitmaps if they are not found in the current working directory. This would typically be set to the directory where the model file is, or else a special texture directory. Any number of paths can be seperated by semi colons, but each must have a terminating back slash e.g.; "c:\genesis\textures\;c:\windows\bmps\" or else be a null string. The pointer cannot be NULL.
phobj	A pointer to the handle of the object that will be created.
Report	The address of a callback function to inform the application of how far through the load its done (see the fnLoadSaveCallback function).

Comments

Loads a lightwave object from an .lwo file into the specified co-ordinate system.

Callback functions

void fnLoadSaveCallback (ulong ulNumBytes, ushort usReportType);

ulNumBytes If usReport type is LSREPORT_LOADING this is the number of bytes actually written to the file. If usReport type is LSREPORT_SAVING this is the number of bytes currently read from the file. If usReport type is LSREPORT_POLYGONS this is actually the number of polygons processed during loading of non Genesis file formats (due to the differences in the way various file formats store polygons, processing can sometimes be a time consuming task).

usReportType Determines what the ulNumBytes parameter refers to. Can be either;

LSREPORT_LOADING
LSREPORT_SAVING
LSREPORT_POLYGONS

Comments

This callback function is used by application to give the user some feedback during the loading or saving of large files to or from Genesis. The address of this function is passed to the various loading/saving routines.

void fnDelCallback (Handle han, HandleType htype, ulong ulUser);

han Handle of the item being deleted.

htype The type of the 'han' handle. Can be any of the *_HANDLE types that are used for the SetUserData call.

ulUser The user data value of the item being deleted.

Comments

This routine is called by geometry when any item such as a patch, light, surface type etc. is deleted. The SetDelCallback function is used to pass the address of this routine to geometry. If SetDelCallback is not called, then no callback is made.

This callback function is useful if you want to assign additional data (or user data) to an item. The application will often allocate memory and store the pointer to this memory in the user dword (see SetUserData call). The application can then use this callback to delete this data.

Structures and types

```
typedef struct Colourtag          // col
        {
        byte                      byRed;
        byte                      byGrn;
        byte                      byBlu;
        } Colour;
```

byRed Red component.

byGrn Green component.

byBlu Blue component.

Comments

Specifies a true RGB colour value.

```
typedef struct VertInfotag        // vi
        {
        Vec                       vec;
        ulong                     ulNumPatches;
        ulong                     ulRef;
        float                     fScrnX;
        float                     fScrnY;
        ulong                     ulUser;
        HanVertex                 hverNext;
        } VertInfo;
```

vec 3D position of vertex.

ulNumPatches Number of patches using this vertex.

ulRef Reference to the first patch. To find all the patches using this vertex pass this value to QryVertPatch.

fScrnX The screen X co-ordinate of this vertex after the last Render call.

fScrnY The screen Y co-ordinate of this vertex after the last Render call.

ulUser The user data value.

hverNext Handle to the next vertex in this object.

Comments

Contains information about a queried vertex.

```
typedef struct NormInfotag        // ni
        {
        Vec                       vec;
        ulong                     ulUser;
        HanNormal                 hnorNext;
        } NormInfo;
```

vec 3D direction vector of normal.

ulUser The user data value.

hnorNext Handle to next normal in this object.

Comments

Contains information about a queried normal.

```
typedef struct PatchInfotag       // pi
        {
        ushort                    usNumEdges;
        ushort                    usFlags;
        HanSurf                   hsur;
        HanVertex                 *ahver;
        HanNormal                 *ahnor;
        Vec                       vecNorm;
        ulong                     ulUser;
        HanPatch                  hpatNext;
        } PatchInfo;
```

usNumEdges Describes the size of the buffers pointed to by ahver and ahnor.

usFlags Set of bitwise OR'd flags describing the patch;

PI_VISIBLE	Patch was visible at last render. The default geometry engine sets this on if the patch was facing us, but was maybe obscured by a closer object
PI_FLAT	If on object is flat, otherwise it is curved

hsur Handle to the surface used by the patch.

ahver Pointer to a buffer in which the handles of the vertices will be returned.

ahnor Pointer to a buffer in which the handles of the normals will be returned.

vecNorm Normal of the plane of the patch.

ulUser The user data value.

hpatNext Handle to the next patch in the object.

Comments

Contains information about a queried patch. The ahver and ahnor pointers should be set to point to the buffers to receive the handles *before* the query call and usNumEdges variable be set to the size of the ahver and ahnor buffers.

```
typedef struct EdgeInfotag          // ei
    {
    HanVertex           hverA;
    HanVertex           hverB;
    HanPatch            hpatAB;
    HanPatch            hpatBA;
    HanEdge             hedgNext;
    } EdgeInfo;
```

hverA Handle to the vertex at one endpoint.

hverB Handle to the vertex at the other endpoint.

hpatAB Handle to the patch which uses the edge from vertex A to vertex B NULL_HANDLE if not used.

hpatBA Handle to the patch which uses the edge from vertex B to vertex A NULL_HANDLE if not used.

hedgNext Handle to the next edge in this object.

Comments

Contains information about a queried edge. Each edge should be used twice if the object is fully enclosed by patches. If the object isn't fully enclosed one of the patch handles will be set to NULL_HANDLE. Since edges are created and deleted automatically by the geometry engine you will not receive an edge used by no patches as any such edges would have been deleted.

```
typedef struct ObjectInfotag                    // oi
        {
        ulong                   ulNumVerts;
        ulong                   ulNumNorms;
        ulong                   ulNumPatches;
        ulong                   ulNumEdges;
        float                   fNewVertTol;
        ulong                   ulMemUsed;
        ushort                  usCoorType;
        HanCoorSys              hcsys;
        HanVertex               hverFirst;
        HanNormal               hnorFirst;
        HanPatch                hpatFirst;
        HanEdge                 hedgFirst;
        ulong                   ulUser;
        HanObject               hobjNext;
        } ObjectInfo;
```

ulNumVerts Number of vertices defined in the object

ulNumNorms Number of normals defined in the object

ulNumPatches Number of patches defined in the object.

ulNumEdges Number of edges defined in the object.

float Indication of how close new vertices can be to neighbouring vertices without getting re-used.

ulMemUsed Total memory used by object and all its vertices, patches etc. (in bytes).

usCoorType Type of co-ordinate system object belongs to;

CT_LEFTHAND	Co-ordinate system is left handed
CT_RIGHTHAND	Co-ordinate system is right handed

hcsys Handle of the co-ordinate system this object belongs to.

hverFirst Handle to the first vertex in this object.

hnorFirst Handle to the first normal in this object.

hpatFirst Handle to the first patch in this object.

hedgFirst Handle to the first edge in this object.

ulUser The user data value.

hobjNext Handle to the next object in the co-ordinate system.

Comments

Contains information about a queried object.

```
typedef struct CoorSysInfotag          // csi
    {
    ushort              usType;
    Mat                 matToParent;
    Mat                 matFromParent;
    HanCoorSys          hcsysParent;
    HanCoorSys          hcsysNext;
    HanCoorSys          hcsysFirstChild;
    HanLight            hliFirst;
    HanObject           hobjFirst;
    ulong               ulUser;
    } CoorSysInfo;
```

usType Type of co-ordinate system.

CT_LEFTHAND	Co-ordinate system is left handed
CT_RIGHTHAND	Co-ordinate system is right handed

matToParent Matrix describing the orientation of this co-ordinate system relative to the parent. The matrix will convert a point in our co-ordinate system into that of the parent.

matFromParent The inverse of matToParent. This matrix will convert a point in the parent co-ordinate system into ours.

hcsysParent Handle to parent co-ordinate system.

hcsysNext Handle to next sibling co-ordinate system, i.e. the next co-ordinate system which also has hcsysParent as its parent.

hcsysFirstChild Handle to the first of our child co-ordinate system, i.e. the first co-ordinate system which has us as the parent (query this co-ordinate system to find the next one).

hliFirst Handle to the first light in this co-ordinate system (query this light to find the next one).

hobjFirst Handle to the first object in this co-ordinate system (query this object to find the next one).

ulUser The user data value.

Comments

Contains information about a queried co-ordinate system.

```
typedef struct LightDeftag          // ld
    {
    Vec                 vecPos;
    Vec                 vecDir;
    ushort              usFlags;
    float               fInt;
    float               fAng;
    Colour              col;
    float               fHalfIntDist;
    HanCoorSys          hcsys;
    ulong               ulUser;
    HanLight            hliNext;
    } LightDef;
```

vecPos 3D position of light.

vecDir 3D position towards which the light is directed.

usFlags Flags describing the light;

LI_DIRECTIONAL	Light has a directional beam
LI_DIMINISHING	Light diminishes with distance

fInt Intensity of the light from 0 to 1.

fAng Angle of the lights beam (if directional).

col Colour of the light.

fHalfIntDist If the LI_DIMINISHING flag is set, this value show how far the light can travel before its intensity halves.

hcsys The co-ordinate system the light belongs to.

ulUser The user data value.

hliNext Handle of the next light in this co-ordinate system.

Comments

Contains information about a light. Can be used to set the light or query the light. NB. The default Geometry engine does not support directional or coloured lights.

```
typedef struct SurfTypetag          // st
    {
    Colour              col;
    ushort              usFlags;
    ushort              usSpecRefChar;
    float               fShineCoef;
    float               fTransCoef;
    HanCoorSys          hcsysTex;
    Mat                 matTex;
    HanSurf             hsurSurf0;
    HanSurf             hsurSurf1;
    HanSurf             hsurSurf2;
    HanSurf             hsurSurf3;
    float               fBumpHeight;
    ulong               ulUser;
    HanSurf             hsurNext;
    char                szFNTex[128];
    } SurfType;
```

col Basic colour of the surface.

usFlags A set of bitwise OR'd flags defining the type of surface.

DS_TEXTURE	Surface is textured
DS_RECURSIVE_TEXTURE	The texture is a recursive texture. The default Geometry engine does not support recursive textures.
DS_TILE_TEXTURE	The texture will be tiled. If the texture is not tiled the basic surface colour 'col' will show through where the texture does not reach, or where hsurSurf0-4 is set to NULL_HANDLE for recursive textures.

DS_BUMPED — The texture will be bump mapped. It is not valid to have this on as well as DS_RECURSIVE_TEXTURE. The default Geometry engine does not support bump mapped textures.

usSpecRefChar This value describes ***only*** how the specular highlights will appear on the surface. It does not mean that an SRC_GOLD surface will have a gold colour but given enough patches the positioning and size of the specular highlights will mimic a surface made of gold

The following types are permitted;

SRC_DULL	(Surface has no specular highlights)
SRC_CONSTANT	(50% reflection regardless of incidence angle)
SRC_LINEAR	(Amount of reflected light increases in proportion to incidence angle)
SRC_SILVER	(Produces large diffuse highlights)
SRC_GOLD	(Produces dull ring shaped highlights)
SRC_GLASS	(Produces highlights only on the horizon edges of objects)
SRC_BRASS	(The default Geometry engine does not implement
SRC_COPPER	any further ,but maps them to the nearest of the
SRC_ALUMINIUM	above...)
SRC_IRON	
SRC_PLASTIC	
SRC_WATER	
SRC_CHINA	
SRC_LEATHER	
SRC_SILK	

For completeness sake the following types are defined but mean the same as SRC_DULL as they are not shiny surfaces;

SRC_ROCK	(same as SRC_DULL...)
SRC_RUBBER	
SRC_WOOD	
SRC_FABRIC	

fShineCoef Shinyness coefficient. This values ranges from 1 to 10. Large values mean highly polished, and therefore very reflective surfaces. These generate very small but bright highlights with sharply defined edges (e.g. a snooker ball). Small values produces large, dull, diffuse highlights (e.g. a piece of paper).

fTransCoef Transmissive coefficient. A value ranging from 0 to 1 to describe how transparent the surface is. A value of 1 would make the surface invisible. The default Geometry engine does not support transmissive surfaces.

hcsysTex The handle of the co-ordinate system the texture is defined relative to. See below for further details. NB. Because a texture is defined in one co-ordinate system, doesn't mean it cant be used by patches in another. Textures pervade all of 3D space, however because top level co-ordinate systems aren't related to each other by matrices like parent-child-sibling relationships, it does means that the orientation of a texture in one scene won't have any meaning in other and may come out looking confusing. The ModSurfType call can be used to modify the co-ordinate system a texture is defined in.

matTex The orientation of the texture bitmap within the co-ordinate system. When a texture is defined the bitmap is projected through 3D space and 'rubs off' on

any patch using this surface type. If you supply the identity matrix here then the orientation of the bitmap is such that the bottom left is at the origin of the co-ordinate system, and the bottom edge proceeds along the x axis for however many pixels wide the bitmap is, e.g. If we have a 100 pixel wide bitmap the bottom left will be at 0, 0, 0 and the bottom right at 100, 0, 0 in the co-ordinate system. Similarly the top of the bitmap proceeds up the y axis of the co-ordinate system. The bitmap is projected through the z axis to +/- infinity. By changing this matrix you can alter the plane of the bitmap or scale it to any size.

hsurfSurf0-3 A recursive bitmap is one where the colour values in the bitmap refer not to colours in the bitmaps palette, but to other surfaces types which can in turn be textured, or even recursive textures. The surface handles for the colour values 0 to 3 can be set here. A value of NULL_HANDLE means the surface's colour (defined by 'col') shows through, whereas a value of HSUR_SEE_THROUGH means that the object has a hole in it at those points showing objects that may be behind. The default Geometry engine does not support recursive textures.

fBumpHeight If the DS_BUMPED flag is set, the colour values in the bitmap refer to height values for a bumpy surface. The *effective* height of the maximum colour value is set using this parameter. The default Geometry engine does not support bump mapping.

ulUser The user data value.

hsurNext The next surface type defined in geometry.

szFNTex The full name (including path and extension) of a windows bitmap file to use as a texture.

Comments

Defines a surface type. This structure can be used to both set, query and modify a surface type.

```
typedef struct ViewParamstag     // vp
        {
        float                   fAngX;
        float                   fAngY;
        float                   fClipFront;
        float                   fClipBack;
        ushort                  usResX;
        ushort                  usResY;
        } ViewParams;
```

fAngX The horizontal viewing angle.

fAnyY The vertical viewing angle.

fClipFront	The distance from the viewpoint of the front clipping plane.
fClipBack	The distance from the viewpoint of the back clipping plane.
usResX	The horizontal size of bitmap to render.
usResY	The vertical size of bitmap to render.

Comments

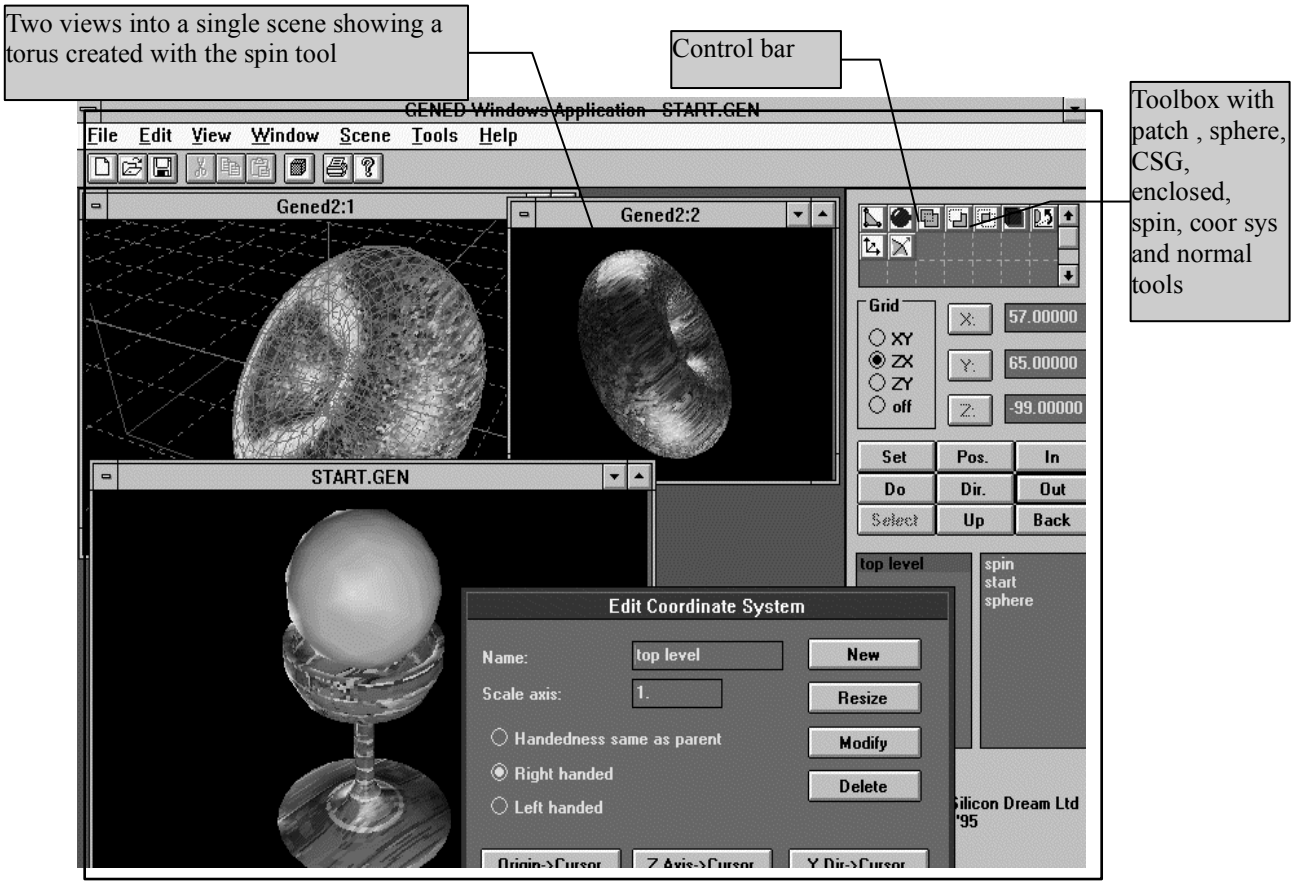
Defines the viewing parameters for the SetCamera function.

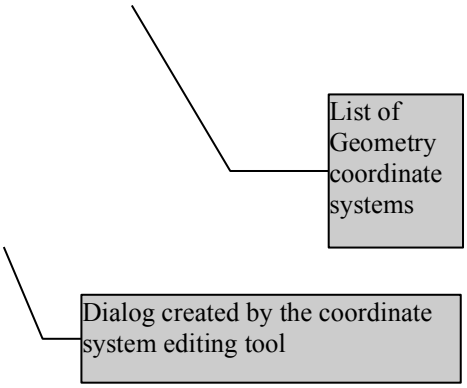
Tool Interface

Introduction

Although custom tools are implemented as a C++ class, an understanding of the C++ language is not essential as we have supplied a template source file; toolexmp.cpp, defining the class. All you need to do is to fill in the functions which can be written in normal C language. The source to the patch tool is also supplied as an example (see patch.cpp). Only a basic understanding of programming the windows operating system is required as the only windows calls you are likely to make are the GDI drawing calls such as, MoveTo and LineTo. Dialog box calls will also be required if your tool needs a configuration dialog box. An understanding of the Geometry API is required.

The screen shot below is useful to refer to when reading this section.





You will notice that some of the Geometry API functions are also available through the tool interface and have been slightly modified. This *does not* mean that you can't use the original Geometry versions There are two reasons we have done this; Some functions, DefPatch for example are built on top of Geometry's DefPatch and provide additional functionality, for instance if an error occurs during the DefPatch call (say 'out of memory') the error will be reported to the user in a message box and everything created by the tool up to that point will be erased. This is the sort of thing many tools would have to implement, so to save time and effort the tool base class DefPatch does it for you (if you want it to).

The other reason for duplicating some functions is that the editor gives names to things like objects and co-ordinates systems. Since Geometry only has the concept of handles and not names, if your tool needs to create an object called 'MyBox' it needs to call the editor version of AddObject for the name to appear in the list box on the control bar. If you use the Geometry version it will still be visible and will be rendered, but without a name it wont be selectable at a single mouse click from the listbox and any tools that require the name of an object to modify will not be able to process the object.

Writing custom tools

Each tool is implemented in a windows DLL by writing a C++ class derived from the base tool class; 'Tool'. The base class is implemented in tool.dll. Your derived class should include tool.h and should be linked to tool.lib.

Your DLL should include a resource file defining a 16 pixel wide by 15 pixel high bitmap with a resource ID of one. This bitmap contains an image which will automatically be placed on a button in the toolbox to enable your tool to be invoked. If you are using Microsoft App Studio this can be done by entering; <id>=1 in the id field of the properties dialog for the bitmap.

The tool object

The DLL should provide a routine to create and delete an object belonging to your tool class which will be called by the editor. The IMPLEMENT_OBJECT macro will do this for you. The following line must be added to your main source file;

```
IMPLEMENT_OBJECT(<your_class_name>)
```

You must also add the following to the exports section of your .def file;

```
CreateTool
DeleteTool
```

The editor will create one of your tool objects for each *scene* that exists. Note, this is not the same as the view. You can have several views into a single scene. It works in the same in same way that a word processor can have several documents loaded at once, and you might have several windows showing different parts of the same document. The reason for this is that it makes tool writing considerably simpler as a tool can be constructing in different scenes at once without the tool code having to manage a complete set of variables relating to each scene. Therefore a tool defines just one set of member variables that it needs to do its job.

The IMPLEMENT_OBJECT macro also calls the Debug library's DebStart function, so if your tool uses the library it should not call it again.

The tool's configuration dialog box

If your tool defines a configuration dialog box, you generally want the dialog to apply to all tools belonging to your object. For instance when editing in scene A, you configure your tool in a certain way. You then switch to scene B, since you see only one tool of your type in the toolbox, you assumes it is configured the way you just set it. Now, if your configuration data is stored as member variables within your object there will be a separate configuration for each tool, i.e. one for each scene. So generally it is best to make your configuration variables *static* members or global.

How to tell the editor about your tool

The name of your DLL should be added to the Gened.ini file in the main windows directory, under the section [Tools] as follows;

```
[Tools]
0=patch
1=sphere
2=box
:
n=<name of your DLL>
```

The value of n must be the next unused consequecutive number.

Overriding tool functions

Whenever the 'Set' or 'Do' buttons on the control bar is pressed, or the 'Undo' menu item is selected an appropriate function is called in the currently selected tool. The tool base class provide default functions which actually do nothing. To detect these events you can override these functions in your derived class. The following table shows the events which trigger tool functions;

Event	*Called when*
OnSelect	The tool has been selected from the toolbox
OnUnSelect	An attempt has been made to select another tool
OnButtonDown	The mouse is in the view and the left button is pressed
OnMouseMove	The left button is pressed and the mouse moves

OnButtonUp	The left mouse button is released
OnSet	The ‘Set’ button on the control bar is pressed
OnDo	The ‘Do’ button on the control bar is pressed
OnViewChange	The another view into the scene becomes active
OnUndo	The ‘Undo’ menu item is selected and this was the last active tool
FreeUndo	When another tool modifies the scene
OnCommand	The user selects a menu/toolbar button create by this tool
OnCommandUpdate	Windows requires the status of a tools menu/toolbar button is updated
OnPaste	A clipboard paste operation is performed and this tool is the paste receiver
OnConfigure	The user requests to configure the tool

Calling sequence

The very first call to all tools is Initialise. Override it if you need to initialise any variables. The next likely call to your tool will be when it is selected from the toolbox, the OnSelect routine will be invoked. Then as the user clicks and moves the mouse in the view you will receive OnButtonDown, OnMouseMove and OnButtonUp events. Then when the user presses the ‘Set’ button, having positioned the 3D cursor, the OnSet call is invoked. This can be invoked as many times as your tool requires. If your tool requires an indefinite number of Set presses, then the ‘Do’ button can be used to do whatever is required to complete the tools action. For instance, the patch tool uses Set to position each vertex. Since you can define an indefinite number of vertices, the Do button is used to create the patch. The interpretation of the buttons is entirely up to you.

All events are directed to the currently selected tool only, apart from, obviously OnSelect, OnUndo and FreeUndo.

Modifying and undoing

For your custom tool to be seamlessly integrated with the editor, it must be capable of undoing any action it has performed on the scene. If and when a tool modifies any aspect of a scene, whether its creating or deleting part of an object, moving a light or even changing the colour of a patch it must call the ‘ModifiedScene’ function. This has two effects. First of all it marks the scene as having changed and will give the user a warning if he tries to close the editor down without having saved the scene first. Secondly it marks this tool for calling should the user select the ‘Undo’ menu item. This tool can now be called to undo its action even after it has been de-selected, at least until another tool calls the ModifiedScene routine.

Drawing into the view

During the construction of a primitive the tool will probably need to draw into the views showing the object being constructed, to give the user some feedback about what is going on. There are two ways of doing this. The tool can implement the DrawSoFar routine so that whenever the view gets updated it gets a chance to draw into the DC representing the view window. The patch tool draws a dotted line between the cursor positions at each point Set was pressed using this technique. The patch isn’t actually created on Geometry until the Do button is pressed. Alternatively, more complex tools might construct the object on Geometry in several steps (e.g. during Set presses, or on mouse move events) in which case the editor can redraw it for you. The sphere tool does this. A combination of these techniques can also be used.

When the screen view needs repainting, say if part of it is uncovered by a window on top, the editor redraws the rendered image along with its wire frame and then calls the selected tool’s DrawSoFar routine passing it the device context handle of the view window. The DrawSoFar routine will be called for each view into the object which needs updating. If we attempt to draw into the view from any call

other than DrawSoFar, then we have to consider the fact that there might be several views into this object all of which will need updating. For this reason drawing should be reserved exclusively for the DrawSoFar routine.

The view will probably need updating after a Set/Do press or an Undo selection. Although we cannot draw on these events we can return a value which will cause the view(s) to be repainted, and therefore DrawSoFar to be called.

Drawing return codes

For each view into the scene the editor maintains two bitmaps. The first is the full colour rendered bitmap generated by Geometry's Render call. The second is a monochrome overlay containing the image of the x, y, and z axis, the objects wire frame, and optionally a set of grid points. When the object changes all this needs to be updated; the image must be re-rendered and the overlay regenerated. When a tool is in a constructing state, we might want to draw the object quickly. For instance the sphere or box tool requires the radius or corner to be dragged by the mouse and released when you release the button. Given that there could be several views into the object there is not time to re-render and regenerate the overlay for every view as the mouse moves. The whole thing would be too slow. The following return codes from event functions can tell the editor how to redraw quickly.

REDRAW_ALL

This is typically used only once when the tool has finished its business and tells the editor to re-render the main bitmaps and regenerate all overlays for each view. The screen image for each view is then refreshed from both of these. This takes the most amount of time and so it is unlikely you will want to use this from a mouse move event.

Generally it takes the windows GDI longer to draw the overlay than it takes the engine to render the image. For this reason the box and sphere tools use this return code from the mouse move event only when the overlay is turned off.

The current tools DrawSoFar routine will also be called.

REDRAW_NONE

Pretty much self explanatory. This return code will not affect the on screen image in any way.

REDRAW_OBJECT_WIRE

This causes the editor to draw the wire frame of the last object created by the tool base class's AddObject function directly onto the screen using the GDI XOR mode. The XOR mode has the effect that if you draw something into a DC twice, it actually removes it leaving you with the original image. Using this return code does not cause the screen image to get refreshed from the rendered and overlay bitmaps, so it is very fast.

The box and sphere tools use this return code to update the wire frame of the box on the mouse move event when dragging a corner to get the correct box size (see the 'Dragging with the mouse' section).

Again the current tools DrawSoFar routine is called.

REDRAW_TOOL

The REDRAW_ALL and REDRAW_OBJECT_WIRE return codes result in the editor redrawing the object as it exists within Geometry (i.e.. as seen through Geometry's query functions). If the object within Geometry hasn't changed, returning these codes will be wasteful as the on screen image wont change. Instead the REDRAW_TOOL return code results in just the tools DrawSoFar routine being called. The patch tool uses this after setting a vertex.

REDRAW_NOTOOL

This causes the editor to simply refresh the screen from its internal bitmaps, thereby effectively removing any image drawn as a result of a REDRAW_OBJECT_WIRE code, or anything drawn by a tool's DrawSoFar routine. If a tool has used either of these to draw an object during a construction phase but undo was selected before the operation completed, then this return code can be used to erase the 'construction image' on the screen. If we actually constructed parts of an object on Geometry then we must remember to remove them as well otherwise the next time a render is performed the object will reappear!

The DrawSoFar routine is *not* called as a result.

REDRAW_REFRESH

This is the same as a REDRAW_NOTOOL except that the tool's DrawSoFar routine is called as well.

REDRAW_SHADING

This return code re-renders the main bitmap but doesn't redraw the overlay. This is only really use then if the geometry itself doesn't change, otherwise the overlay won't match the rendered picture. An example is the when the light tool moves a light The objects don't move so the overlay doesn't need updating but the shading on the objects changes.

Again the DrawSoFar routine is called.

Getting 2D screen co-ordinates

Any drawing performed in DrawSoFar is done using the windows GDI. The 2D window co-ordinates you need to perform the drawing operations can be got using Geometry calls such as Get3DPoint and Get3DLine. The Geometry API spec states that before you can use these calls you must set the camera in the appropriate co-ordinate system. Obviously without specifying a camera view the 3D co-ordinates you work with will not have a 2D counterpart. Before DrawSoFar is called the editor sets the camera at the viewpoint in the co-ordinate system of the view being redrawn so the tool doesn't have to worry about this. On event calls like OnSet and OnMouseMove the camera is set in the currently *active* view, i.e.. the one that has the focus.

Dragging with the mouse

The sphere tool creates a small sphere when Set is pressed, if the 'quickdraw' configuration option is switched off. It then traps the OnMouseMove event and scales the sphere to the correct radius. By returning REDRAW_OBJECT_WIRE it gets the editor to draw the wire frame of the sphere (assuming the overlay is on).

If it is more appropriate to perform drawing ourselves then simply use GDI's ROP2 setting of XORPEN to draw and erase your image in the DrawSoFar routine. You may of course prefer to save the background before drawing on it, but in general we have found that XOR drawing is faster. The only thing to remember is that if you change any of the DC's attributes such as the selected pen, drawing mode etc. you must set them back to the originals before DrawSoFar returns. NB. This is a general requirement of programming in Windows.

XOR'ing in multiple views

If you wish to implement dragging on the mouse move event as described above there is just one other thing to bear in mind. Consider the following sequence of events. Assume that this is part of a box construction tool. We 'Set' the cursor at one corner then move the cursor to the opposite corner (during which the box is being dynamically sized in the view[s]), and press 'Do'. Further assume we have two views into this scene.

1. **OnSet event** - Set the 3D co-ordinates of the first corner to those of the cursor. Set the 3D co-ordinates of the far corner to the same as the first and return REDRAW_TOOL to start the XOR drawing (initially a point sized cube as both corners are in the same place)
2. **OnDrawSoFar call** - Called for the *first* view as a result of returning REDRAW_TOOL from OnSet. Being the first DrawSoFar for this view there is no previous construction image to remove, so simply draw the image in XOR mode (i.e.. a cube) using Get3DPoint to convert your 3D corners into 2D window co-ordinates and draw them using the GDI. Store the 2D co-ordinates of your construction image so you can erase it later
3. **OnDrawSoFar call** - Called for the *second* view. Do exactly the same as for the previous DrawSoFar but for the second view
4. **OnMouseMove event** - Set the 3D co-ordinates of the opposite corner to the cursor position, and return REDRAW_TOOL
5. **OnDrawSoFar call** - Called for the *first* view as a result of returning REDRAW_TOOL from the OnMouseMove event. Redraw the old construction image in XOR mode to erase the image and redraw the image in XOR mode using the opposite corner in its *new* position
6. **OnDrawSoFar call** - Called for the *second* view. Do exactly the same as for the previous DrawSoFar but for the second view
7. **Do steps 4, 5 and 6** - For however many mouse move events we get
8. **OnDo event** - Construct the cube using Geometry and return REDRAW_ALL to get it rendered properly

You will notice that because we have two views we get two DrawSoFar calls every time we return REDRAW_TOOL from an event. This means that we must store two sets of co-ordinates in order to remove the previous XOR image. For this purpose DrawSoFar gets called with a parameter to instance data relating to each particular view.

DrawSoFar parameters

The first parameter contains the DC handle of the view. You need this when using GDI drawing commands.

The second parameter is a pointer to a 60 byte area of memory stored as part of each view known as the view instance data. You can use this memory to store whatever you like, e.g. co-ordinates of the construction image. If you need more memory than this you can allocate your own and store a pointer to the memory in the view instance data. The SetViewData call in the tool base class can be used to initialise the instance data at any time (say, from the OnSet event).

The final parameter is a boolean value which says why this routine has been called. Not only will you receive DrawSoFar calls as a result of return codes from events but also if part of the view was invalidated by a window on top being removed. In normal windows programming this will generate a WM_PAINT message. If this happens and your tool is in the process of constructing an XOR image it must not erase the previous image because the view is invalid in that region. As a general rule, if this

bInvalid parameter is TRUE simply draw the XOR image as if it were the first OnDrawSoFar for this view.

General guidelines

Help on using tools

As a general rule any tool should provide instructions to the user on how to operate the tool. The recommended approach is at each stage inform the user what to do next by writing a message in the status bar at the bottom of the main window using the WriteMessage function. See the Sphere or Patch tools as an example.

Displaying text strings

Whenever your tool displays any text it is a good idea to load the text from a string table resource. This way if your tool ever gets used in other countries, language translation is a simple matter of going through the string table and recompiling to get a new DLL, rather than having to search through the source code for literal strings. The tool base class functions such as WriteMessage and MessageBox aid this by allowing you to present string resource ids as well as character pointers.

Co-ordinate system types

Any tool which creates patches has to consider the fact it could be invoked in either a left-hand or a right-hand co-ordinate system. The significance is in the ordering of the vertices. Remember vertices should always be seen to progress clockwise around the patch. If your tool works fine in one system but not in the other then you probably need to reverse the ordering in the system which doesn't work.

User Data

User data can be associated with geometry items such as objects, patches, lights, surfaces etc. The user data is a single 32 bit value. Tools cannot modify this value directly as the editor uses the user dwords to store pointers to its internal information. If a tool needs to store instance for a geometry item, it must use the appropriate tool functions such as AddUserData. This way the editor can chain together the instance data it needs as well as those from any tools.

A note for C Users

For those not familiar with C++, you can use the toolexmp.cpp file as a base to work from. The functions defined in the 'Overidables' section are functions you can implement yourself if you're interested in trapping those particular events. Empty functions are defined but commented out in toolexmp.cpp. If you want to trap mouse moves events simply uncomment it and add your code using normal C. Remember C++ is a superset of the C language.

The functions defined in the section titled 'Support' are supplied for you to call should the need arise. One thing you will realise about C++ is that a function is specified not just by its name but also by its

parameters, meaning you can have two or more functions with the same name as long as they have different parameters (NB. The MessageBox support function is an example).

Some variables are also defined in the ‘Support’ section. In C++ terminology these are public variables defined in the base class. Non C++ programmers can just think of them as global variables containing information which may or may not be of use to your tool. You must not attempt to alter these variables.

Overridables

virtual ~Tool();

Comments

Destructor for your tool object. This function must be overridden even if it does nothing. As with any normal C++ class, if you create a constructor for your object which allocates resources, the destructor must release these resources. C users can leave this function empty.

virtual void Initialise();

Comments

Called once when the DLL is loaded. This call can be used to initialise variables.

virtual Redraw OnSelect(Vec &vec);

vec Current position of the 3D cursor.

Comments

Called when the user selects the tool from the toolbox.

virtual Redraw OnUnSelect(bool *pbOkToChange)

pbOkToChange Set this to TRUE if it is OK for the editor to change the current tool.

Comments

Called when the user tries to select another tool from the toolbox. If the tool is in the middle of a complicated operation it can either set *pbOkToChange to FALSE or it can either abort its operation or complete it without further action from the user. What happens here is up to the tool writer. Remember the tool can always undo what its just done if the user isn't happy.

When the clipboard pastes an object into the scene the paste receiver tool is automatically selected in order for the user to position the object. This means that an OnUnSelect event is sent to the current tool if this is not the paste receiver. If the tool sets *pbyOkToChange to FALSE the user is presented with a message saying that the paste cannot be carried out until the selected tool completes its operation.

---

virtual void OnConfigure();

Comments

Called when the tool is selected and the user then selects the Configure item from the Tool menu. Typically this call is overridden to construct a dialog box enabling the user to enter configuration information for the tool.

---

virtual Redraw OnButtonDown(Vec &vec);

vec Current position of the 3D cursor.

Comments

Called when the tool is selected and the user presses the left mouse button when the mouse is in a view allowing the 3D cursor to be moved. The vec parameter gives the co-ordinates of the 3D cursor. The editor assumes the tool is in a constructing state only after the first Set press, so if you override this function, construction should not start until Set has been pressed. The Redraw return code can be used to tell the editor how to efficiently redraw the view.

---

virtual Redraw OnMouseMove(Vec &vec);

vec Current position of the 3D cursor.

Comments

Called when the 3D cursor moves within a view. NB. Windows programmers should realise that this is not quite the same as a WM_MOUSEMOVE message as the left button has to be depressed while the mouse is over a view.

---

virtual Redraw OnButtonUp(Vec &vec);

vec Current position of the 3D cursor.

Comments

Called when the tool is selected and the user releases the left mouse button when the mouse is in a view.

virtual Redraw OnSet(Vec &vec);

vec Current position of the 3D cursor.

Comments

Called when the 'Set' button is clicked after a tool has been selected.

virtual Redraw OnUndo();

Comments

Called when user wishes to undo the effect of a tool. This can be called even if the tool is not selected. If the currently selected tool has not modified the scene in any way and the user selects ‘Undo’ from the ‘Edit’ menu, then the last tool to call ModifiedScene is called to undo.

If we are not the selected tool we should only return REDRAW_ALL after undoing, as anything else might effect whatever the selected tool has drawn into the view(s). For instance returning REDRAW_NOTOOL will erase whatever the selected tool has drawn.

virtual Redraw OnDo(Vec &vec);

vec Current position of the 3D cursor.

Comments

Called when the user presses the ‘Do’ button on the control bar. Typically this is used to end use of the selected tool.

virtual void OnViewChange(HanCoorSys hcsys, Vec &vec);

hcsys The co-ordinate system of the new active view.

vec Current position of the 3D cursor.

Comments

Called when the user clicks the mouse in a new view making it active. Since there is a separate object of each tool type for each scene the tool will not be notified of a view change into a different scene. In fact the new active scene could have a different tool type currently selected.

This event is also called if a tool calls the base class version of AddCoorSys with the bAddView parameter set to TRUE. In this case the OnViewChange event is called ***before*** the AddCoorSys call returns.

---

virtual Redraw OnPaste(Vec &vec,
HanObject *ahobj,
ulong ulNumObjects);

vec Current position of the 3D cursor.

ahobj A list of ulNumObjects object handles.

ulNumObjects The number of objects pasted into the view.

Comments

Only the paste receiver tool receives this event. When a clipboard paste operation is perform the tool is automatically selected and this event is called to position one or more objects in the view. To make a tool the paste receiver tool the PasteReceiver function must be called. In the default configuration the move tool is also the paste receiver.

When the paste receiver moves the pasted objects it should not consider this as modifying the scene, so it should not call ModifyScene as a result. When a paste is performed the 'undo' tool (the one which gets called to undo the last action if undo is selected from the menu) is set internally to the editors clipboard. If undo is selected the editor deletes all pasted objects. If the paste receiver calls ModifiedScene then the it will be called to undo the move only (not the entire paste).

---

virtual void FreeUndo();

vec Position to set the cursor too.

Comments

When a tool modifies a scene, whether its creating or deleting part of an object, moving something or creating a light etc. it must call ModifiedScene. This marks the tool as the one which must perform an 'undo' if the user requests it. Sometimes a tool will need to allocate memory or other resources in order to be able to undo what it has done. If another tool then modifies the scene this one becomes the 'undo' tool and the previous tool can then free up its resources.

This event is used to tell a tool that if it allocated any resources for the purposes of undoing, then they are no longer required and it can free them. If the same tool modifies a scene twice, say the sphere tool is used to create two spheres without another tool being selected in the mean time, it can (but doesn't need to) call ModifiedScene twice. However it will only receive one FreeUndo event when another tool modifies the scene. The tool should therefore free any resources it has kept before the second operation.

The subtract tool creates two archives of the objects it uses in a subtract operation so that it can re-create the original objects if undo is selected. When it receives this event if deletes the archives.

virtual void OnCommand(ushort usID)

usID The command ID of the menu item or toolbar button.

Comments

Called when a menu item or toolbar button created by this tool is selected by the user. Menu items and toolbars can be added to the main frame window using standard windows calls. The tool must however use only the range of IDs from 0x9000 to 0x9009.

The editor alone will receive notifications that these have been selected. These notifications are passed on to the tool using this event.

virtual CmdUpdate OnCommandUpdate(ushort usID, char *szText, ushort usBuffSize)

usID The command ID of the menu item or toolbar button.

szText Can be used to set the text of a menu item. This is ignored unless we return SETTEXT.

usBuffSize The size of the supplied szText buffer.

Comments

This event is called when windows requires the status of a menu item or toolbar button to be updated and is usually called when the menu drops down or when there are no other messages in the queue for toolbar buttons. The value returned determines the status of the control. If SETTEXT is returned the supplied buffer should contain the text to set the menu item to. Possible return values are;

```
ENABLE
DISABLE
CHECK
UNCHECK
INDETER
RADIOON
RADIOOFF
SETTEXT
```

```
virtual void DrawSoFar(HDC hdc,
                       HanCoorSys hcsys,
                       void *pvViewData,
                       bool bInvalid);
```

hdc	A handle to the device context of the view being redrawn.
hcsys	Co-ordinate system handle of the view being redrawn.
pvViewData	Pointer to instance data for the view.
bInvalid	The region being redrawn was invalidated, probably by a window on top being removed.

Comments

Called when the view is being redrawn to give the tool a chance to draw the 'object so far' into the view.

Support

Protected member variables

hinst	Instance handle of the derived tool DLL.
hwnd	Handle to the main frame window.
hcsysTopLevel	Handle of top level co-ordinate system in this scene.
hsurActive	Currently active surface type. This is the selected palette entry or the selected surface in the surface type dialog box. The value is NULL_HANDLE if no surface is selected.
usSmoothFlags	A set of autosmooth flags as defined by the Autosmooth dialog box. The flags are the same set of autosmooth flags that are passed to geometry's DefPatch call.
fSmoothAng	The autosmooth threshold angle as set by the autosmooth dialog box expressed in radians.
hcsysActive	The co-ordinate system of the currently active view.
bOverlay	Is TRUE if the overlay is switched on in the active view (hcsysActive).
Mode	The mode of the active view (hcsysActive)
ahverPrim	Array of handles to vertices created (ulNumVerts contains how many). See StartPrim for details.
hverPrim	Last handle created by the base class's AddVertex call.
ulNumVerts	Number of vertex handles in ahverPrim.
ahnorPrim	Array of handles to normals created (ulNumNorms contains how many). See StartPrim for details.
hnorPrim	Last handle created by the base class's AddNormal call.
ulNumNorms	Number of normal handles in ahnorPrim.
ahpatPrim	Array of handles to patches created (ulNumPats contains how many). See StartPrim for details
ulNumPats	Number of patch handles in ahpatPrim.

void WriteMessage(int iResId);
void WriteMessage(const char *szMsg);

iResId The resource id of the string within the DLLs resource file.

szMsg Pointer to a null terminated string.

Comments

Called to write a message in status bar at the bottom of the main window.

---

int MessageBox(int iTitleId, int iMsgId,
UINT uiStyle =MB_ICONEXCLAMATION);
int MessageBox(int iTitleId, char *szMsg,
UINT uiStyle =MB_ICONEXCLAMATION);
int MessageBox(int iTitleId, GeomErr gerr);

iTitleId The resource id of a string to use for the message box title.

iMsgId The resource id of the main message string.

szMsg A pointer to the message string.

gerr A Geometry error value. The routine will query an appropriate error string from Geometry, and will use the MB_ICONEXCLAMATION style.

uiStyle One of windows MB_ values to indicate the type of message box. This defaults to MB_ICONEXCLAMATION if omitted.

Comments

Puts a message box on the screen. There are three variations on this function. If a tool gets an internal processing error return code from Geometry it should use the third form to notify the user, and then abort whatever action it was performing.

The return code is the same as for the windows MessageBox function.

```
bool StartPrim (HanObject hobj,
                ulong ulMaxVerts,
                ulong ulMaxNorms,
                ulong ulMaxPats,
                bool bReportErrs,
                bool bUndoOnErr);
```

hobj	Handle to the object we want to add to.
ulMaxVerts	Maximum number of vertices we will be adding.
ulMaxNorms	Maximum number of normals we will be adding.
ulMaxPats	Maximum number of patches we will be adding.
bReportErrs	If this is set to TRUE any Geometry errors that occur during AddVertex, AddNormal, or DefPatch (e.g. 'out of memory' or 'vertices not on a plane') will be reported to the user in an message box.
bUndoOnErr	If this is set to TRUE and a Geometry error occurs then every patch, vertex and normal added to the object since the StartPrim call will be removed.

Comments

This routine can be used along with the base class's AddVertex, AddNormal and DefPatch routines to add elements to an object. They are usually more convenient than Geometry's version of these functions because the handles to the elements are automatically stored in arrays allocated by this routine. For instance a tool will usually want to access say the handle for the 8th vertex that it created. If we use the base class version of the AddVertex function we can access this handle using the ahverPrim protected member variable of the tool base class, i.e.. ahverPrim[7] for the 8th vertex. Similarly ahnorPrim[7] will access the 8th normal created and ahpatPrim[7] will give you the 8th patch. What is more, if we need to undo, the RemovePrim function will remove all the elements created so far.

It should be considered that some geometry engines will only support triangular patches so each call to DefPatch could result in several patches being created. Make sure you leave enough room in the ulMaxPats parameter for the maximum number of patches that could result.

Once the tool has finished constructing and we have no further interest in the handles, we must call the EndPrim function to release the handle arrays. If during any 'Add' function a serious error is returned by Geometry a description of the error can be automatically reported to the user and the RemovePrim function called to remove all elements created. The return code is then passed back to the tool. In this case you must not call EndPrim.

If the tool has any reason to call RemovePrim itself it should not call EndPrim as the RemovePrim routine does this for you.

If you call say AddVertex to create more vertices than you originally specified on the StartPrim call the function will not fail, but the handle will not be stored either, therefore if you call RemovePrim you might find it doesn't remove all of your object.

StartPrim can be called with an object already containing vertices, patches and normals.

This routine returns TRUE if it succeeded.

---

void EndPrim ();

Comments

Called after a StartPrim and when the tool has no further need for handles to the elements its created. This routine removes the buffers allocated by StartPrim.

---

void RemovePrim ();

Comments

Removes all primitive elements (verts, normals and patches) added to the object the tool is working on. Should be called only if a StartPrim has been called.

GeomErr AddVertex (Vec &vec);

vec Position of vertex.

Comments

Adds a vertex to the object specified by a call to StartPrim and records its handle in the ahverPrim array. The hver base class variable can be used as a more convenient way of accessing the last handle created by an AddVertex call.

If Geometry reports an error this is passed to the user in a message box.

This call can only be used after a call to StartPrim. See StartPrim comment and Geometry's AddVertex function for a full description.

GeomErr AddNormal (Vec &vec);

vec Direction of normal.

Comments

Adds a normal to the object specified by a call to StartPrim and records its handle in the ahnorPrim array. The hnor base class variable can be used as a more convenient way of accessing the last handle created by an AddNormal call.

If Geometry reports an error this is passed to the user in a message box.

This call can only be used after a call to StartPrim. See StartPrim comment and Geometry's AddNormal function for a full description.

GeomErr DefPatch (ushort usNumEdges,
ushort usFlags,
float fSmoothAng,
HanVertex *ahver,
HanNormal *ahnor,
HanSurf hsur);

usNumEdges The number of edges, and therefore vertices and normals in the patch. This can be any number from 3 upwards.

usFlags Flags controlling the creation of the patch.

fSmoothAng Specifies the angle to use for autosmoothing expressed in radians. Only takes effect if DP_SMOOTH_BY_ANGLE flag is used.

ahver Array of handles to vertices. The number of vertices is given by the usNumEdges parameter.

ahnor Array of handles to normals. The number of normals is given by the usNumEdges parameter.

hsur Handle to the surface type to be used for the patch. If you use NULL_HANDLE here then the currently selected surface type will be used. If no surface type has been selected then any available type will be used. If no surface types exists then one will be created.

Comments

Adds a patch to the object specified by a call to StartPrim and records its handle in the ahpatPrim array. Unlike AddVertex and AddNormal, DefPatch doesn't have an equivalent hpat variable as the call could result in several patches being created. The ahpat array must be used to access the handles.

If Geometry reports an error this is passed to the user in a message box.

This call can only be used after a call to StartPrim. See StartPrim comment and Geometry's DefPatch function for a full description.

HanObject AddObject(HanCoorSys hcsys,
char *szName,
ushort usBuffSize,
float fNewVertTol,
HanObject hobj)

hcsys Handle to co-ordinate system to add object too.

szName The name of the object to appear in the object list box. The actual name used is returned in this buffer as the name supplied might already be in use. If this is the case a number is added to the end of the name.

usBuffSize Size of the szName buffer. To ensure that the modified name will fit you can set this to MAX_NAME_BUFF_LEN but ensure that the supplied string uses no more than MAX_NAME_LEN of the buffer. If the actual name used is of no interest this can be set to zero, so the buffer will not be modified. This allows use of a literal string, e.g. AddObject(hcsys, "box", 0...)

fNewVertTol The tolerance setting for creating new vertices rather than reusing old ones.

hobj If the object already exists within Geometry and you simply want to give it a name, the existing object handle can be supplied here.

Comments

This function *does not* have to be used in conjunction with StartPrim and EndPrim. It simply calls Geometry's AddObject function to start an empty object in the given co-ordinate system. However by supplying a name, the object gets added to the object list on the control panel thereby allowing the entire object to be selected by highlighting its name.

To delete this object the standard geometry function ::DelObject can be used.

HanCoorSys AddCoorSys(HanCoorSys hcsysParent,
char *szName,
ushort usBuffSize,
HanCoorSys hcsys,
Mat &matToParent,
Mat &matFromParent,
ushort usType,
bool bAddView);

hcsysParent Handle to the parent co-ordinate system.

szName The name of the coor sys to appear in the list box. The actual name used is returned in this buffer as the name supplied might already be in use. If this is the case a number is added to the end of the name.

usBuffSize Size of the szName buffer. To ensure that the modified name will fit you can set this to MAX_NAME_BUFF_LEN but ensure that the supplied string uses no more than MAX_NAME_LEN of the buffer. If the actual name used is of no interest this can be set to zero, so the buffer will not be modified. This allows use of a literal string, e.g. AddCoorSys(hcsys, "new", 0...)

hcsys If the coor sys already exists within Geometry and you simply want to give it a name, the existing handle can be supplied here.

matToParent Matrix describing the transformation from child to parent. In other words this matrix transforms a point in the child co-ordinate system to its corresponding position relative to the parent co-ordinate system.

matFromParent The inverse or opposite matrix to matToParent. NB. The maths library can easily generate the inverse of a matrix by preceding it with a minus sign (e.g. -mat).

usType This value indicates whether the co-ordinate system being defined is right handed or left handed. It should be either CT_RIGHTHAND or CT_LEFTHAND.

bAddView If this is set to TRUE a new view window will appear.

Comments

Creates a new co-ordinate system as a child hcsysParent. Its name will appear in the coor sys list box. There are many advantages of using a named co-ordinate system rather than creating an unnamed one using the Geometry API, for instance, a named co-ordinate system can be used in the surface type dialog to specify an orientation for a texture.

To delete this co-ordinate system the standard geometry function ::DelCoorSys can be used.

HanObject GetObjectHandle(const char *szName);

szName The name of the object whose handle is to be returned.

Comments

Returns the handle of the named object. A NULL_HANDLE is returned if the name was not recognised.

HanObject GetCoorSysHandle(const char *szName);

szName The name of the coor sys whose handle is to be returned.

Comments

Returns the handle of the named co-ordinate system. A NULL_HANDLE is returned if the name was not recognised.

void GetObjectName(HanObject hobj, char *szName, ushort usBuffSize);

hobj The handle of the object whose name we want returned.

szName The name of the object.

usBuffSize Length of the szName buffer.

Comments

Returns the name of a given object. A zero length string is returned if the handle was not recognised.

void GetCoorSysName(HanCoorSys hcsys, char *szName, ushort usBuffSize);

hcsys The handle of the coor sys whose name we want returned.

szName The name of the coor sys.

usBuffSize Length of the szName buffer.

Comments

Returns the name of a given coor sys. A zero length string is returned if the handle was not recognised.

void ForceRedraw(Redraw rd, HanCoorSys hcsys=NULL_HANDLE);

rd The redraw code to use.

hcsys The handle of the co-ordinate system whose view is to be redrawn. If more than one view window exists for this coor sys they are all redrawn. If a NULL_HANDLE is passed, every view into the scene (i.e.. all co-ordinate systems) is redrawn.

Comments

Forces a redraw of one or more views. This function is useful if a tool constructs a dialog box which has buttons that can be invoked independently of any events from the editor. For instance the coor sys editing tool has a dialog whose buttons could invoke a redraw. All resulting DrawSoFar calls will be made ***before*** this call returns.

---

void DrawMarker(HDC hdc, int iX, int iY, ushort usType);

hdc Handle to device context to draw into.

iX Device context X co-ordinate for marker.

iY Device context Y Co-ordinate for marker.

usType Type of marker to draw. Can be one of the following values;

MT_BOX	A small square marker
MT_PLUS	A '+' shaped marker
MT_CROSS	An 'X' type cross marker
MT_DIAMOND	A small diamond shaped marker

Comments

Draws a marker into the device context at the specified position using the current attribute (pen, colour, mode etc.) settings of the DC.

---

void DrawArrow(HDC hdc, int iX, int iY, ushort usFlags);

hdc Device context to draw into.

iX Device context x co-ordinate of the arrow head.

iY Device context y co-ordinate of the arrow head.

usFlags Type of marker to draw. Can be one of the following values;

DA_FILLED	The arrow head is filled in
DA_FIXEDSIZE	The arrow head is fixed in size as opposed to proportional to the length of the arrow body

Comments

Draws an arrow from the current position in the device context to iX, iY.

---

bool ActivateCSysWin(HanCoorSys hcsys);

hcsys Handle of co-ordinate system whose view is to be activated.

Comments

Activates the view window associated with a co-ordinate system. This means bringing it on top of the other windows and highlighting its title bar. An OnViewChange event will be received ***before*** this call returns.

---

void ModifiedScene();

Comments

This routine ***must*** be called after any tool modifies the scene in any way. Not only does it mark the tool as the one which should perform an undo if the user requests it, but also it marks the document as modified such that a warning will be issued if an attempt is made to close the editor without saving it first.

---

void SetViewData(void *pvData, ushort usSize);

pvData Pointer to the view data.

usSize Size of the view data.

Comments

Sets the view data for every view in the scene to the contents of pvData.

---

void BackgroundYield(const char *szWrite=NULL);

szWrite An optional string which is written to the status bar at the bottom of the screen.

Comments

If a tool has a lot of processing to perform which could take several seconds, even minutes, then somewhere in its processing loop it should call this function. This prevents the entire system being ‘locked up’ during the processing. The optional string can be used to print a message to status bar to say for instance; “25% done”. Notice how the CSG tools use this.

void *AddUserData(HanObject hobj,
Handle han,
HandleType htype,
ulong ulSize);

hobj The parent object of the item to add the data to. This parameter is only needed if the item is a patch, a vertex or a normal.

han The handle of the item. This can be any standard geometry handle.

htype The type of the 'han' handle. This can be any one of;

```
COORSYS_HANDLE
OBJECT_HANDLE
PATCH_HANDLE
VERTEX_HANDLE
NORMAL_HANDLE
SURFACE_HANDLE
LIGHT_HANDLE
```

ulSize Size of the user data required.

Comments

If a tool needs to store instance data for a geometry item, say a patch, light or co-ordinate system, this routine should be used. It is not safe to allocate memory and use geometry's SetUserData call to store the pointer in the items user data, as many other tools and the editor itself may also need to store instance data. Using this routine not only allocates the memory for you but chains together the user data blocks from all other tools. The returned pointer points to a block of memory of ulSize bytes.

Only one set of user data can exist for each tool. If more is needed the first lot should be deleted using RemoveUserData before calling here again.

The user data is automatically deleted when the item is deleted from geometry.

void *GetUserData(HanObject hobj,
Handle han,
HandleType htype);

hobj The parent object the item belongs to. This is only required if the item is a patch, a vertex or a normal.

han The handle of the item we want the user data for.

htype The type of the handle (see AddUserData).

Comments

The pointer returned points to the user data created by this tool for the geometry item given by ‘han’.

---

bool RemoveUserData(HanObject hobj,
Handle han,
HandleType htype);

hobj The parent object the item belongs to. This is only required if the item is a patch, a vertex or a normal.

han The handle of the item we want the user data deleted from.

htype The type of the handle (see AddUserData).

Comments

Removes the user data allocated by a tool using the AddUserData call. If TRUE is returned the memory was freed successfully.

---

bool IsSelected(HanObject hobj,
Handle han,
HandleType htype);

hobj The parent object of the patch or vertex we want to query. This is ignored if we are querying an object.

han The handle of the item we want to query.

htype The type of the handle. This can be any one of;

```
OBJECT_HANDLE
PATCH_HANDLE
VERTEX_HANDLE
```

Comments

Returns TRUE if the geometry item is selected.

Handle GetNextSelected(Handle hanParent,
ulong *pulRef,
HandleType htype);

hanParent The parent item of the item we are querying. If we want to find the next selected object in a specific co-ordinate system we must set this to the handle of the co-ordinate system. If we want to find the next selected patch or vertex in an object, this must be set to the owning object.

pulRef A reference value used by this routine. To query the first selected object in a co-ordinate system or the first vertex/patch in an object we must first set the value pointed to, to zero.

htype The type of the handle. This can be any one of;

```
OBJECT_HANDLE
PATCH_HANDLE
VERTEX_HANDLE
```

Comments

Returns the handle of the next selected object in a co-ordinate system or the next selected vertex or patch in an object. Object, vertices and patches are selected using the editor 'select mode'. The first call must be made with (*pulRef) set to zero. Subsequent calls will then return all selected items. When NULL_HANDLE is returned there are no more selected items in the object/co-ordinate system. An object is selected if all of its vertices are selected.

HanCoorSys QryNextCoorSys(ulong *pulRef, CoorSysInfo *pcsi);

pulRef A pointer to a reference value that this routine uses to find its place in the enumeration. On the first call to this routine set (*pulRef) to NULL_HANDLE).

pcsi A pointer to a structure used to receive information about the next co-ordinate system.

Comments

It is quite common to have to write code which performs processing for every co-ordinate system in a scene. Because of the hierarchical nature of the co-ordinate systems the code needed to enumerate them (return each in sequence) can be quite complex. This is a handy routine which can be called multiple times to return a query structure for each co-ordinate system in the scene.

The first time the routine is called (*pulRef) should be set to NULL_HANDLE. This routine then returns the handle of the next coordinate system in the scene along with its query structure.When NULL_HANDLE is returned there are no more co-ordinate systems to process. The following code can be used to perform processing for all co-ordinate systems in a scene;

```
ulRef=NULL_HANDLE;
hcsys=QryNextCoorSys(&ulRef, &csi);
while (hcsys!=NULL_HANDLE)
    {
    /* Perform processing here using info from csi... */

    QryNextCoorSys(&ulRef, &csi);
```

```
}
```

void EditInThisView(HanCoorSys hcsys);

hcsys Co-ordinate system to edit in. Use NULL_HANDLE to indicate any co-ordinate system can be used.

Comments

Most primitive creation type tools (for instance the editor's sphere and box tools) will only allow you to create the primitive in the view window (or co-ordinate system) you started it in. For instance the sphere tool wont allow you to press 'Set' (for the sphere centre) in one window and 'Do' in another. This routine can be used to say that any button presses must be made when the view for the specified co-ordinate system is active. If any other is active the tool is not notified of the event. This forces creation to happen in a specific window. If this is used, when creation has finished or if 'Undo' is selected then you must call it again with NULL_HANDLE. This ensures that the next operation can be started in any window.

void MakeObjWireframe(HanObject hobj);

hobj Object to add to the wireframe redraw list.

Comments

If a tool returns a REDRAW_OBJECT_WIRE code from an event, any objects (using the base class AddObject call) created since the tool was selected will be drawn in wire frame rather than rendered. Subsequent return codes from events cause the old image to be erased (using the GDI's XOR mode) and redrawn enabling fast wireframe animation which can be performed on the OnMouseMove event for instance. However it might be that the tool requires an existing object drawn in wire frame. The move tool does this for any objects which are selected when the mouse moves. This function adds other objects to the wire frame redraw list.

void NoneWireframe();

Comments

Clears the wire frame redraw list. See the MakeObjWireframe call.

void PasteReceiver();

Comments

Nominates this tool as the paste receiver tool. Only one tool can nominate itself as a paste receiver. In the default configuration the move tool is the paste receiver tool. When a clipboard paste operation is performed on the editor the pasted object(s) need positioning in exactly the same way you can paste into a 2D graphics paint program (except in 3D). The paste receiver tool will be automatically selected and will receive a special OnPaste event when pasting occurs. The OnPaste event passes the tool a list of handles for objects to be positioned.

---

void AutosmoothDlg();

Comments

This can be called by a tool to bring up the Autosmooth dialog box. Any changes made in this dialog will be reflected in the public base class variables usSmoothFlags and fSmoothAng. Most primitive creation type tools use these to control the smoothness of the primitives they create. Some tools also include a button on their configuration dialogs to access this dialog.

---

void SurfaceDlg();

Comments

This can be called by a tool to bring up the Surface Type dialog box. If the currently active surface type changes as a result of the user clicking on another type then the public base class variables hsurActive will reflect this. Most primitive creation type tools use this as the surface type to use for new primitives. Some tools also include a button on their configuration dialogs to access this dialog.

---

void ChangeMode(Mode mode);

mode The mode to set the active view to. Possible values are;

MODE_EDIT
MODE_VIEW
MODE_FLIGHT
MODE_SELECT

Comments

This call can be used to change the mode of the active view. The edit mode is most useful for tools as mouse movement in edit mode causes the 3D cursor to move. Most tools require use of the 3D cursor to position and create primitives. The view tool on the other hand is a very basic tool which does nothing more than put the active view into view mode. Using this call, tools have the ability to change the mode on the users behalf to allow the user to better view something it has just done or select a set of patches to be worked on etc.

void SetCursor(Vec &vec);

vec Position to set the cursor too.

Comments

Sets the position of the 3D cursor in the active view. The light tool uses this to set the cursor to the position of the light selected in its dialog box.

HanArchive ArchiveObject(HanCoorSys hcsysOrigin,
HanObject hobj,
bool bRemove);

hcsysOrigin The co-ordinate system which the archive is stored relative to. All vertices and textures used by the object are converted relative to this co-ordinate system.

hobj The object to store in the archive.

bRemove Set to TRUE if the object should be deleted from geometry, as in a clipboard cut operation.

Comments

This call archives an object. An object archive is a complete description of an object including all its vertices, normals and any surface types that it uses (including textures), in a block of memory. The handle returned is not a geometry handle but a special base class handle which identifies the archive. Archives are extremely useful for recreating objects which have been deleted (see the CreateFromArchive function).

The hcsysOrigin parameter does not have to be the co-ordinate system the object belongs to. All vertex position and texture positions will be converted relative to hcsysOrigin.

The subtract tool uses archives to undo the effect of a subtraction operation.

The archive handle is returned. If the function fails NULL_HANDLE is returned.

HanObject CreateFromArchive(HanArchive harch,
HanCoorSys hcsys);

harch The handle of the archive to extract from.

hcsys The co-ordinate system we want to extract to. If NULL_HANDLE objects is extracted back into the co-ordinate system it was originally copied from.

Comments

This function recreates an object from an archive created by the ArchiveObject call. The hcsys parameter determines which co-ordinate system it will be created in. If this is the same as the hcsysOrigin parameter used on the ArchiveObject call then the object will be created in *exactly* the same place in 3D space that it was taken from, assuming of course it was deleted when originally archived. If it was not deleted you will end up with two identical objects in exactly the same place.

Just because the object might re-appear in the same place doesn't mean of course that it belongs to the same co-ordinate system. The co-ordinate system it belongs to now is given by hcsys.

If hcsys is NULL_HANDLE the object is copied back into its original co-ordinate system. The editor uses this facility to undo a clipboard cut operation.

The return value is the handle of the newly created object.

---

void DeleteArchive(HanArchive harch)

harch The handle of the archive to delete.

Comments

Deletes an object archive.

Quick start to writing tools

The following source file is provided as a template to work from when creating new tools. Remember you must define a 16x15 pixel bitmap with an id of one in your resource file, and you must export CreateTool and DeleteTool in the .def file (the functions are implemented by the IMPLEMENT_OBJECT macro).

From then on simply uncomment any events you are interested in trapping. The whole of the geometry, tool interface, maths and debug API's are available to you as well as the windows API.

Finally put an entry in the Gened.ini file so the editor knows of the existence of your tool.

```
/*-------------------------------------------------------------------------
                                ExampleTool
                                -----------

           Example tool class

           (C) Silicon Dream Ltd 1995

  -------------------------------------------------------------------------
Changes:                                                        Date:
* Created file                                                  01/06/96
*/

#include <tool.h>

#include "resource.h"

// YOU DO: Change the class name below to describe your tool and anywhere else
//            this class name is used in this file
class ExampleTool: public Tool
           {
           private:
                       // YOU DO: Declare your private variables here. eg.
                       //  Vec                 vecCursorPos;
                       //            (Non C++ users: 'private' means those
                       //             variables which have global scope from
                       //             within the functions you define here)

                       // YOU DO: Uncomment any functions to trap events you're interested in
                       // void _cppdyn Initialise();                          // Called once when DLL loaded
                       // Redraw _cppdyn OnSelect(Vec &vec);                  // Called when tool is selected
                       // Redraw _cppdyn OnUnSelect(bool *pbOkToChange);      // Called when tool is selected
                       // void _cppdyn OnConfigure();                         // Called to configure tool
                       // Redraw _cppdyn OnButtonDown(Vec &vec);              // Called when button pressed
                       // Redraw _cppdyn OnMouseMove(Vec &vec);               // Called when mouse moves with button down
                       // Redraw _cppdyn OnButtonUp(Vec &vec);                // Called when button is released
                       // Redraw _cppdyn OnSet(Vec &vec);             // Called when the 'Set' button is clicked
                       // Redraw _cppdyn OnDo(Vec &vec);                      // Called to end use of tool
                       // Redraw _cppdyn OnUndo();                            // Called when user wishes to undo effect of tool
                       // void _cppdyn FreeUndo();                            // Called when tool no longer needs to keep..
                       // void _cppdyn OnViewChange(HanCoorSys hcsys, Vec &vec); // Called when active view changes
                       // Redraw _cppdyn OnPaste(Vec &vec, HanObject *ahobj,      // Called when objects are pasted into the..
                       //                        ulong ulNumObjects);
                       // void _cppdyn OnCommand(ushort usID);                // Called when a tool specific menu item..
                       // CmdUpdate _cppdyn OnCommandUpdate(ushort usID,,     // Called when a tool specific menu item..
                       //                                   char *szText, ushort usBuffSize);
                       // void _cppdyn DrawSoFar(HDC hdc, HanCoorSys hcsys,       // Called to draw object so far into the view
                       //                        void *pvViewData, bool bInvalid);

           public:
                       _cppdyn ~ExampleTool() {};
           };

IMPLEMENT_OBJECT(ExampleTool)

// YOU DO: Uncomment and fill in any functions for events you want to trap.
//             (Non C++ users: If you create any functions of your own which need
//              to access your private variables (see above), you must make the
//              funtion part of your class by declaring it along with the other
//              functions (see above), and prefixing the function name with
//              'classname::' in the implementation)

/*
void _cppdyn ExampleTool::Initialise()
           {
           }
*/
/*
Redraw _cppdyn ExampleTool::OnSelect(Vec &vec)
           {
           return REDRAW_NONE;
           }
*/
/*
Redraw _cppdyn ExampleTool::OnUnSelect(bool *pbOkToChange)
           {
           *pbOkToChange=TRUE;
           return REDRAW_NONE;
           }
*/
/*
void _cppdyn ExampleTool::OnConfigure()
           {
           }
*/
/*
Redraw _cppdyn ExampleTool::OnButtonDown(Vec &vec)
           {
           return REDRAW_NONE;
           }
*/
```

```
/*
Redraw _cppdyn ExampleTool::OnMouseMove(Vec &vec)
            {
            return REDRAW_NONE;
            }
*/
/*
Redraw _cppdyn ExampleTool::OnButtonUp(Vec &vec)
            {
            return REDRAW_NONE;
            }
*/
/*
Redraw _cppdyn ExampleTool::OnSet(Vec &vec)
            {
            return REDRAW_NONE;
            }
*/
/*
Redraw _cppdyn ExampleTool::OnDo(Vec &vec)
            {
            return REDRAW_NONE;
            }
*/
/*
Redraw _cppdyn ExampleTool::OnUndo()
            {
            return REDRAW_NONE;
            }
*/
/*
void _cppdyn ExampleTool::FreeUndo()
            {
            }
*/
/*
void _cppdyn ExampleTool::OnViewChange(HanCoorSys hcsys, Vec &vec)
            {
            }
*/
/*
Redraw _cppdyn ExampleTool::OnPaste(Vec &vec, HanObject *ahobj, ulong ulNumObjects)
            {
            return REDRAW_NONE;
            }
*/
/*
void _cppdyn ExampleTool::OnCommand(ushort usID)
            {
            }
*/
/*
CmdUpdate _cppdyn ExampleTool::OnCommandUpdate(ushort usID, char *szText, ushort usBuffSize)
            {
            return ENABLE;
            }
*/
/*
void _cppdyn ExampleTool::DrawSoFar(HDC hdc, HanCoorSys hcsys, void *pvViewData, bool bInvalid)
            {
            }
*/
```

Maths Library

The maths library is implemented in a single DLL containing useful functions and classes for manipulating graphical related objects such as vectors and matrices. Currently Geometry accepts parameters to its functions as C++ classes so the C version of the library is not really useful for programming tools or applications. However eventually a C version of the API will also be included.

This section is split into two, the first part describing the C support, the second describing the C++ support.

To use the DLL functions and classes you must include either the 'maths.h' include file (not to be confused with C's math.h), for C users, or 'cppmaths.h' for C++ users. Both must then link to 'maths.lib'.

The C++ class interface is built on top of the C maths library and provides no additional functionality that is not available in the C library, however the C++ version allows programs to be written which manipulate vectors and matrices using algebraic formula and are therefore easier to read. For instance;

```
VectorA=VectorB+VectorC;
```

is easier to understand than;

```
MthAddVec(&VectorA, &VectorB, &VectorC);
```

For C Users

Not yet written

For C++ Users

The C++ support is probably best described in terms of the class definitions and examples of the operations available on those classes, rather than a function by function breakdown.

Four types of class are available;

Vec	Implements a three element floating point Cartesian co-ordinate (x, y and z).
Lvec	Implements a four element floating point Cartesian co-ordinate (x, y, z and w).
Polar	Implements a three element floating point polar co-ordinate (theta, phi and rho).
Mat	Implements a 4x4 floating point matrix.

Vectors

A vector can be used to store the position of a point in 3D space or alternatively a direction relative to another point. The following operations are defined for vectors;

Vec vec(x, y, z)	Initialise a vector with up to three values
Vec vecA(vecB)	Initialise a vector with another vector
Vec vec(lvec)	Initialise a vector with a long vector (loses last component)
((Vec) lvec)	Convert a long vector to a vector (loses last component)
Vec vec(pol)	Initialise a vector with a polar vector (conversion performed)
((Vec) pol)	Convert a polar vector to a vector (conversion performed)
vecA+vecB	Add two vectors
vecA-vecB	Subtract a vector from another
vec*f	Multiply a vector by a scalar
vec*mat	Multiply a vector by a matrix
vec/f	Divide a vector by a scalar
vecA+=vecB	Add two vectors (overwrite original)
vecA-=vecB	Subtract a vector from another (overwrite original)
vec*=f	Multiply a vector by a scalar (overwrite original)
vec*=mat	Multiply a vector by a matrix (overwrite original)
vec/=f	Divide a vector by a scalar (overwrite original)
-vec	Returns the vector inverted
vecA==vecB	Are two vectors equal?
vecA!=vecB	Are two vectors not equal?
vec.Set(x, y, z)	Set a vector's components
vec.SetX(y)	Set a vector's X component
vec.SetY(y)	Set a vector's Y component
vec.SetZ(z)	Set a vector's Z component
vec.X()	Get x component
vec.Y()	Get y component
vec.Z()	Get z component
vec.Len()	Get length of vector
vecA.Dot(vecB)	Get dot product of two vectors
vecA.Cross(vecB)	Get cross product of two vectors
vec.AddrVec()	Gets address of Vector structure member (WARNING: Use only to interface to maths C functions)

Long Vectors

Long vectors are designed primarily to allow multiplication by matrices. The following operations are defined for long vectors;

LVec lvec(x, y, z, w)	Initialise a long vector with up to four values
LVec lvecA(lvecB)	Initialise a long vector with another long vector
LVec lvec(vec)	Initialise a long vector with a vector (last component becomes 1.0)
((LVec) vec)	Convert a long vector to a vector (last component becomes 1.0)
lvec*mat	Multiply a long vector with a matrix
lvec*=mat	Multiply a long vector with a matrix (overwrite original)
lvecA==lvecB	Are two long vectors equal?
lvecA!=lvecB	Are two long vectors not equal?
lvec.Set(x, y, z, w)	Set a long vector's components
lvec.SetX(x)	Set a long vector's X component
lvec.SetY(y)	Set a long vector's Y component
lvec.SetZ(z)	Set a long vector's Z component
lvec.SetW(w)	Set a long vector's W component
lvec.X()	Get x component
lvec.Y()	Get y component
lvec.Z()	Get z component
lvec.W()	Get w component (homogeneous co-ordinate)
lvec.AddrLVec()	Gets address of LongVec structure member (WARNING: Use only to interface to maths C functions)

Polar Vectors

The following operations are defined for polar vectors;

Polar pol(t, p, r)	Initialise a polar with up to three values
Polar polA(polB)	Initialise a polar with another polar
Polar pol(vec)	Initialise a polar with a vector (conversion performed)
((Polar) vec)	Convert a vector to a polar (conversion performed)
polA==polB	Are two polars equal?
polA!=polB	Are two polars not equal?
pol.Set(t, p, r)	Set a polar's components
pol.SetTheta(t)	Set a polar's theta component
pol.SetPhi(p)	Set a polar's phi component
pol.SetRho(r)	Set a polar's rho component
pol.Theta()	Get theta component
pol.Phi()	Get phi component
pol.Rho()	Get rho component

pol.AddrVec()	Gets address of Vector structure member (WARNING: Use only to interface to maths C functions)

Matrices

The following operations are defined for matrices;

Mat mat	Initialise a matrix with the identity matrix
Mat matA(matB)	Initialise a matrix with another matrix
Mat mat(XROT, a)	Initialise a matrix with a rotation or scale value (or leave it unset)
Mat mat(TRANSL, vec)	Initialise a matrix with a translation value (or leave it unset)
Mat mat(vecOrigin, vecZAxis, vecYDir)	Initialise a matrix by defining a new origin, a direction for the z axis, and given these constraints, a direction towards which the y axis will point.
matA*matB	Multiply two matrices
matA*=matB	Multiply two matrices (overwrite original)
matA==matB	Are two matrices equal?
matA!=matB	Are two matrices not equal?
-mat	Compute inverse of matrix (i.e. when multiplied by this gives identity)
mat.Set(f1,..f16)	Sets elements of a matrix
mat.MoveParent(vec)	Moves coor sys defined by matrix relative to its parent
mat.MoveChild(vec)	Moves coor sys defined by matrix relative to itself
mat.ScaleParent(x, y, z)	Scale coor sys defined by matrix relative to its parent
mat.ScaleChild(x, y, z)	Scale coor sys defined by matrix relative to its own origin
mat.XRotParent(a)	Rotates coor sys defined by matrix about parents x axis
mat.XRotChild(a)	Rotates coor sys defined by matrix about its own x axis
mat.YRotParent(a)	Rotates coor sys defined by matrix about parents y axis
mat.YRotChild(a)	Rotates coor sys defined by matrix about its own y axis
mat.ZRotParent(a)	Rotates coor sys defined by matrix about parents z axis
mat.ZRotChild(a)	Rotates coor sys defined by matrix about its own z axis
mat.ReverseX()	Reverses direction of x axis of coor sys defined by matrix
mat.ReverseY()	Reverses direction of y axis of coor sys defined by matrix
mat.ReverseZ()	Reverses direction of z axis of coor sys defined by matrix
mat.AddrMat()	Gets address of Matrix structure member (WARNING: Use only to interface to maths C functions)

Debug Library

The debug library is not really part of Genesis but is a very helpful debugging aid whether you're writing tools, geometry engines, applications, in fact any code what so ever. It can be used in windows or non-windows programs, DLLs, C, or C++. What is more it presents you with just one set of very fast and very simple memory management functions.

The first call for any application or DLL must be to DebStart.

To allocate memory use Debug's Malloc and Free calls (note the capitals letters to distinguish from C's memory functions). If using C++ you can use New and Delete (again with capitals). At any time, a call to DebListMem will output to a file showing any memory which has been allocated, the size of the memory, the module and line it was allocated from and whether it was allocated with Malloc or New. This is usually called when an application or DLL terminates to ensure all memory was freed.

If you're programming under windows 3.x the Malloc and New functions are *not* limited to allocating 64K segments.

The DebOut function can be used exactly like printf but will output to a file. Each line appearing in the debug output file is timestamped to within the accuracy of the system clock. The time given is the time in seconds since the first call to DebOut. What is more since the output is buffered into a 64K area of memory it has practically zero impact on the performance of your application and so can be used to time 'speed critical' parts of your code. When the buffer becomes full and the file has to be flushed to disk a line appears in the file saying that a flush happened at this point in case there is a discrepancy in the time stamps.

The debug filename is set by the string passed on the very first call to DebOut. This string should contain a valid filename. Directly after the filename you can put a '+' character to indicate that the data should also be directed to an additional output stream. Under windows the additional output stream will be the debug window, in a regular C program it will be the screen. Because many DLL's could all be printing on the additional output stream simultaneously the output can look confusing and the timestamps wont necessarily be continuous.

If writing C code simply include 'debug.h' and C++ users should include 'cppdebug.h'. In both cases you should define the _DEBUG macro, this can be done with /D _DEBUG on the compile line. The program must then be linked to debug.lib.

Here is an example run;

```
DebStart();                                  // Must be first call
DebOut("C:\app.deb+");                       // Open debug file (with debug win output also)
DebOut("Debug file has been opened");
DebOut("Variable fLength is %f", fLength);  // Output floating point variable
pvoid1=Malloc(20);                           // Allocates 20 bytes
pvoid2=Malloc(30);                           // Allocates 30 bytes
pobj=New Obj;                                // Allocate a C++ object called Obj
DebOut(FLUSH);                               // Forces the debug file to be flushed to disk
DebOut("More debug");
DebListMem();                                // List all unfreed memory to debug file
DebOut("Closing file now");
DebOut(STOP);                                // Flushes and closes the debug file
Free(pvoid1);                                // Free memory...
```

```
Free(pvoid2);
Delete(pobj);
```

The following output will be generated;

```
0.000: Debug file has been opened
0.000: Variable fLength is 5.75
0.000: --------: (Flushed)
0.030: More debug
0.030: Allocated memory dump:
0.033:   Size   |     File     | Line  |  Alloc type
0.033: ---------+--------------+-------+--------------
0.033:     20   |     app.c    |  128  |   Malloc
0.033:     30   |     app.c    |  129  |   Malloc
0.033:    106   |     app.c    |  130  |    New
0.040: Closing file now
```

If your code is compiled without the _DEBUG macro defined then the memory allocation will not store the debugging information required by DebMemList. What is more the DebOut routine will not be included as part of your application and so calls to it should be enclosed in #ifdef's to prevent linker errors, e.g.;

```
#ifdef _DEBUG
DebOut("A line of debug");
#endif
```

Instances of debug library

If you are building a large application consisting of many DLLs, and statically linked libraries, then it is sometimes not clear how many instances of the debug library you have. Basically whenever you use the linker and pass it the debug library's name, then you get an instance of the library. The upshot of this is that you cannot open a debug file in your application and expect one of your DLLs to write to the same file. Any DLLs your application use we're linked separately and therefore have their own copy of the debug library. Therefore they should open their own debug file on the first call to DebOut. Statically linked libraries however are not linked independently and are therefore considered part of the application or DLL using it.

Writing a Geometry engine

Writing a Geometry engine is the simple process of writing a DLL whose interface conforms to the Geometry API specification and which maps those functions onto its own or a third party rendering engine or a hardware based interface.

Of course there may be many differences between the new renderer and Genesis's Geometry engine. For instance the renderer might not have the concept of a co-ordinate system. However as long as it has the ability to move objects independently of one another then a co-ordinate system can be implemented in the DLL which hides the inadequacies of the renderer. It might also be that the renderer uses integer rather than floating point co-ordinates. Again this is easy to remedy, you could adopt the convention that floating point numbers in the range 0 to 100 get scaled to integer numbers 0 to 100,000 meaning that floating point numbers with an effective resolution of 0.001 are supported. As long as the co-ordinates are converted back on any query function then a Geometry application will not know the difference.

Errors

The geometry.h file includes all of the standard geometry errors. If a particular error code is listed in the Geometry API spec as being returned from a particular function, for instance, DefPatch returns GERR_NOT_ON_PLANE if its points do not lie on a plane, then any Geometry engine ***must*** return the same error should the condition occur. This is because some applications may check for this specific return code and if it returns a more general error, such as GERR_INT_PROCESSING_ERROR then the behaviour of the application might change. The mark of a good Geometry engine is one whose applications work identically to the default one.

If an error occurs in the new engine which a standard error does not describe adequately then you can use GERR_INT_PROCESSING_ERROR or define your own code whose value must be GERR_UNKNOWN_ERROR (defined in geometry.h) or greater. Such errors can have their own descriptions implemented in the engine’s GetErrorText function.

Handles

All handles are defined as 32 bit unsigned integers. The geometry engine can use this value to mean whatever is most convenient. For instance the default Geometry stores a pointer to the appropriate object in this value. It can also be used as an index into an array, however, the value 0 cannot be used as this is reserved for the NULL_HANDLE, so any such arrays must be based at one.

Handles *do not* have to be validated according to the spec. To perform proper validation on all handles at every call could be time consuming, so if it is not quickly and easily achieved, then do not worry about implementing it. The onus is on the application and tool writers to provide valid handles to calls.

Unsupported features

If a Geometry engine does not support a particular feature say, bumped mapped textures, it should still accept DefSurfType calls defining bump mapped textures *without* returning a bad error. When rendered, the texture should just appear as a normal image mapped texture. What is more if the surface is queried with QrySurfType it should return a SurfType structure with the bump mapped flag set, otherwise as a particular model is saved and loaded into editors with different engines then the model will change. If it is loaded then saved again by an editor whose engine does not support bump mapping, and then the resulting file is loaded into an engine that does, the texture will have lost its 'bumpy' quality. Likewise engines not supporting 'curved' patches must still store the normal information.

Helper library

A library is provided which contains functions common to any Geometry engine thereby preventing any duplication of work on behalf of the programmer. The library includes;

- Validating patches (checks all points lie on a plane etc.)
- Splitting patches into smaller (possibly concave) patches
- Computing patch normals
- Computing normals for 'autosmooth' patches
- Loading and saving scenes
- Loading bitmaps for textures
    - Providing text descriptions of Geometry errors

If the Geometry engine is compiled with the _DEBUG macro defined then we must link to the debug version of the helper library genhelpd.lib. Otherwise we should link to the release version genhelp.lib. Notice there is no include file for the helper library, everything you need is defined in geometry.h.

API's

Any Geometry engine has at its disposal the whole of the maths, debug and helper APIs. It should not attempt to use the tool interface as there is no concept of tools in the Geometry engine. Tools are purely an invention of the editor application.

GeomErr RegisterError (GeomErr gerrIn, char *szFNIn, ushort usLineIn);

gerrIn Geometry error to register.

szFNIn Filename of the module in which the error occurred

usLineIn The line number within the module at which the error occurred.

Comments

Registers a Geometry error. If an error occurs somewhere within the Geometry engine, whether it is the result of the application passing in bad parameters (e.g. GERR_INVALID_HANDLE), or due to circumstances beyond the apps control (e.g. GERR_OUT_OF_MEMORY) then the error must be passed back to the app. It is quite likely that the app will then call the GetErrorText function to see what went wrong and possibly to report the error to the user. If the Geometry engine registers the error with this routine ***before*** returning the error code then it gives the GetErrorText routine in the helper library the ability to build a more helpful error string indicating where exactly the error occurred.

The Gerr() macro defined in geometry.h calls the RegisterError function without worrying about how to work out the module name and line number.

The following example code shows the recommended way of returning errors;

```
if (pmem==NULL)
        return Gerr(GERR_OUT_OF_MEMORY);
```

If the app then passes the returned error to GetErrorText which in turn invokes the helper library function GetHlpErrorText, then the following text will be returned;

Out of memory. Error occurred at line 850 in module geometry.cpp.

If we returned the error without invoking the macro;

```
if (pmem==NULL)
        return GERR_OUT_OF_MEMORY;
```

then the following text will be returned from GetErrorText;

Out of memory.

Implementation specific errors with values above GERR_UNKNOWN_ERROR can also be registered.

GeomErr GetHlpErrorText (GeomErr gerr, char *szBuff, ushort usBuffSize);

gerr The error whose text is to be returned.

szBuff Buffer to contain message.

usBuffSize Size of the supplied buffer.

Comments

This function can be called from the Geometry engine's GetErrorText function. It supplies standard text descriptions for all of Geometry's errors. If an implementation specific error is used (one with a value above GERR_UNKNOWN_ERROR) then the buffer will contain the following;

> 'An implementation specific error occurred. Error was reported at line 850 in module geometry.cpp. Implementation interprets error as:'

and GERR_UNKNOWN_ERROR will be returned. The GetErrorText function can then append its own implementation defined description of the error if there is space in the buffer. strlen can be used to find the length of the returned string. If the GetHlpErrorText function runs out of buffer space then it copies as much of the error text as possible and puts three period characters, '...' in the end of the buffer and returns GERR_BUFFER_TOO_SMALL.

GeomErr ValidPatch (Vec *avec, ushort usNumVecs, ushort usFlags);

avec A list of points to be used for the vertices.

usNumVecs Number of points in avec.

usFlags Flags used to indicate the kind of validity checks to perform;

VP_CHECK_THIN_SEGMENT	Checks that no 'thin' segments appear in the patch outline. A thin segment is where part of the outline doubles back on itself at a 180 degree angle, forming an infinitely thin line rather than an area.
VP_CHECK_ON_PLANE	Checks that all the points lie on a plane or within a tolerance of the plane surface. The tolerance used is proportional to the size of the patch and is approximately 1/50th the greatest distance across any two points on the patch outline. This really only applies to patches with more 4 or more vertices.

Comments

Two checks are performed regardless of the flag setting, first that at least three points have been supplied, and secondly that no consequetive points are coincident, i.e. are in exactly the same place. One check which is isn't performed is whether any line segments in the patch's outline cross over thereby making the ordering clockwise in one part and anticlockwise in the other. We also do not check for concave outlines as to some renderers a concave patch might well be a valid patch. To those that don't support concave patches the split function can be called to chop it up into smaller convex patches.

In general this routine should not be called from DefPatch if the DP_DONT_VALIDATE flag is used in the call to DefPatch. This is because this call can be time consuming if many patches are being created, and if the application programmer is confident that no invalidate patches will be passed then DP_DONT_VALIDATE can speed things up.

GeomErr ExtractVecs (HanObject hobj, ushort usType, Handle *ahan, ushort usNum, Vec **pavec);

hobj Handle to object containing vertices.

usType Are we extracting vertices or normals? Value can be EV_NORMS or EV_VERTS.

ahan Array of handles to vertices or normals.

usNum Number of handles in ahan array.

pavec The address of a pointer to Vec. The pointer to the allocated memory holding the vectors is returned here.

Comments

This routine extracts the vectors from a list of vertices or normals using QryVertex or QryNormal respectively. A buffer is allocated to contain the vectors. This buffer must be freed by the calling code using the Free call.

This routine is useful for extracting vertex positions in order to pass to routines such as ValidPatch, Split and CompNormal.

GeomErr CompNormal (Vec *avec, ushort usNumVecs, ushort usFlags, Vec *pvecNorm);

avec Set of points to use for patch.

usNumVecs Number of points in avec.

usFlags Flags to specify the significance of the normal;

CN_DIRECTION_IMPORTANT	The normal will point to the side from which the points proceed in a clockwise direction. Also if we set this flag we must try and set the CONVEX flag (if we know for sure that the outline is convex) and the co-ordinate system type flag (appropriate to the co-ordinate system the points belong to).
CN_CONVEX	If we know for certain that the outline is convex, then we should set this flag as it can reduce the amount of processing this routine has to do. If CN_DIRECTION_IMPORTANT is not set then this is not important.

CT_LEFTHAND	If we know that this patch belongs to a left-hand coor sys then we should set this flag as it can reduce the amount of processing this routine has to do. If CN_DIRECTION_IMPORTANT is not set then this is not important.
CT_RIGHTHAND	If we know that this patch belongs to a right-hand coor sys then we should set this flag as it can reduce the amount of processing this routine has to do. If CN_DIRECTION_IMPORTANT is not set then this is not important.

pvecNorm A unit length normal vector is returned here.

Comments

Computes the normal of a set of points. No check is made to see if the points lie on a plane (use ValidPatch for this). If the CN_DIRECTION_IMPORTANT is set then a lot more processing is needed to work out which side the normal should be facing. This processing can be reduced by specifying as many of the other parameters as possible. Concave patches always need the full processing if CN_DIRECTION_IMPORTANT is set.

If the patch outline contains some points very close together then this routine is selective in deciding which three points to use to generate the normal. This ensures we get an accurate a value as possible as the accuracy of this normal is crucial to the operation of some of the editors tools.

GeomErr Split (Vec *avecMain, ushort usNumMainVecs, ushort usMaxPerPatch, ushort usFlags, ushort *pusNumSets, void **apvSets, ushort *ausNumInSets);

avecMain Set of points to split up.

usNumMainVecs Number of points in avecMain.

usMaxPerPatch The maximum number vertices per patch that this Geometry engine will allow.

usFlags Flags controlling the operation;

SP_SUPPORT_CONCAVE	Indicates whether the Geometry engine supports concave patch outlines.
SP_RETURN_INDEX	Rather than returning an array of vector arrays in (*apvSets) instead we return an array of ushort arrays, where each ushort is an index into the original set of points passed in.

pusNumSets The number of sets returned, i.e.. the number of smaller outlines the main outline was split into.

apvSets A pointer to an array of size (*pusNumSets) containing pointers to either arrays of vectors arrays of ushorts (if SP_RETURN_INDEX was given). Each of these arrays describes a set of points. If SP_RETURN_INDEX was specified than a variable declared; ushort **aausIndex, should be passed to apvSets, otherwise the variable should be; Vec **aavec; All arrays are allocated by this routine.

ausNumInSet An array of ushorts indicating the size of the individual set arrays. For example ausNumInSet[2] contains the size of the array pointed to by apvSet[2]. In other words set 2 consists of elements apvSet[2][0] to apvSet[2][ausNumIntSet[2]].

Comments

This routine splits up an outline defined by a set of points into a number of sets. The sets can be expressed themselves as points or as indexes into the original array avecMain. The sets are allocated by this routine, but it is the responsibility of the calling code to Free the memory when it has finished with it.

The following code fragment will free up all memory;

```
for (us=usNumSets; us--;)
        Free(aausIndex[us]);        // Free all sets
Free(aausIndex);                    // Free array of pointers to sets
Free(ausNumInSet);                  // Free array of sizes of sets
```

Although we can pass in any number for usMaxPerPatch, in practice the sets produced will not contain more than 4 points each, although these can be concave if SP_SUPPORT_CONCAVE is set.

GeomErr Autosmooth (HanObject hobj, HanVertex *ahver, ushort usNumVerts, Vec &vecNorm, ushort usFlags, HanSurf hsur, float fSmoothAng, HanNormal *ahnor, ASUndoBuff *pasub);

hobj Handle to object to add ‘autosmoothed’ patch too.

ahver Array of vertex handles defining the patch.

usNumVerts Number of handles in ahver. Also specifies the size of the ahnor buffer.

vecNorm The normal of the patch to be ‘autosmoothed’. The normal should point to the empty side of the patch.

usFlags Flags controlling the operation;

DP_SMOOTH_BY_SURF	This patch is to be smoothed only with neighbouring patches which have the same surface type.
DP_SMOOTH_BY_ANGLE	This patch is to be smoothed only with neighbouring patches which form an angle less than fSmoothAng with this patch.

hsur Surface type of the ‘autosmoothed’ patch.

fSmoothAng Specifies autosmooth angle threshold if DP_SMOOTH_BY_ANGLE is set.

ahnor An array of handles to normals returned.

pasub Pointer to an autosmooth undo buffer. If the geometry engine wishes to abort this operation it can pass this to the UndoAS routine to undo any changes made. If not it ***must*** pass it to the ComitAS.

Comments

Given an array of vertex handles defining a new patch, this routine can determine the set of normals to use to get this patch to appear smooth given the criteria defined in usFlags. It does this by looking at the surrounding patches and averaging out the surface normals of the patches to create the normals. If normals already exist at these points they are modified, otherwise new ones are created. The set of normals to use in defining the patch is passed back in the ahnor parameter. The returned pointer pasub should be passed either to UndoAS or to CommitAS if the patch creation succeeded.

GeomErr CommitAS (ASUndoBuff *pasub);

pasub Pointer to the autosmooth undo buffer.

Comments

Commits the changes made to the normals by the Autosmooth function. Either this or UndoAS must be called at some point after an Autosmooth call.

GeomErr UndoAS (ASUndoBuff *pasub);

pasub Pointer to the autosmooth undo buffer.

Comments

Undoes the changes made to the normals by the Autosmooth function. Either this or CommitAS must be called at some point after an Autosmooth call.

GeomErr LoadBmp (char *szFN,
BITMAPINFO huge **ppbmi);

szFN The name of a .bmp file to open.

ppbmi The address of a pointer to a bitmap which will be returned after the bitmap is loaded. The memory is allocated by this routine.

Comments

Loads a windows bitmap or IFF file with the specified name. Even if the bitmap is an IFF file it is presented to the calling code as a windows BITMAPINFO structure. HAM and Extra halfbrite IFF files are also supported.

The correct amount of memory is automatically allocated and a pointer to it is returned in (*ppbmi). The calling code must free the memory using the Debug library's Free call. If the bitmap does not exist then GERR_BITMAP_FILE_NOT_FOUND is returned. If the file has an incorrect format GERR_NOT_A_BMP_FILE is returned.

GeomErr SaveSceneHlp (HFILE hfile,
HanCoorSys hcsys,
ulong *pulNumCSys,
HanCoorSys *ahcsys,
ulong *pulNumObjs,
HanObject *ahobj,
fnLoadSaveCallback *Report);

hfile Windows handle of the file to save too.

hcsys Handle of the co-ordinate system to save.

pulNumCSys Size of ahcsys (in number of handles). On return holds the total number of co-ordinate systems saved.

ahcsys An array of handles to co-ordinate systems saved.

pulNumObjs Size of ahobj (in number of handles). On return holds the total number of objects saved.

ahobj An array of handles to objects saved.

Report The address of a callback function to inform the application of how far through the save its done.

Comments

This function takes exactly the same parameters as Geometry's SaveScene function, so the implementation of the geometry function is nothing more than calling this helper routine.

```
GeomErr LoadSceneHlp (HFILE hfile,
                      HanCoorSys csysParent,
                      Char *szTexPath,
                      ulong *pulNumCSys,
                      HanCoorSys *ahcsys,
                      ulong *pulNumObjs,
                      HanObject *ahobj,
                      fnLoadSaveCallback *Report);
```

hfile	Windows handle of the file to load from.
hcsysParent	Handle of the co-ordinate system which will be the parent of the one being loaded.
szTexPath	A pointer to a path specification for where to search for texture bitmaps if they are not found in the current working directory. This would typically be set to the directory where the model file is, or else a special texture directory. Any number of paths can be seperated by semi colons, but each must have a terminating back slash e.g.; "c:\genesis\textures\;c:\windows\bmps\" or else be a null string. The pointer cannot be NULL.
pulNumCSys	Size of ahcsys (in number of handles). On return holds the total number of co-ordinate systems loaded.
ahcsys	An array of handles to co-ordinate systems loaded.
pulNumObjs	Size of ahobj (in number of handles). On return holds the total number of objects loaded.
ahobj	An array of handles to objects loaded.
Report	The address of a callback function to inform the application of how far through the load its done.

Comments

This function takes exactly the same parameters as Geometry's LoadScene function, so the implementation of the geometry function is nothing more than calling this helper routine.

GeomErr LoadLWObjectHlp (HFILE hfile,
HanCoorSys hcsys,
char *szTexPath,
HanObject *phobj,
fnLoadSaveCallback *Report);

hfile	Windows handle of the file to load from.
hcsys	Handle of the co-ordinate system the object will be added to.
szTexPath	A pointer to a path specification for where to search for texture bitmaps if they are not found in the current working directory. This would typically be set to the directory where the model file is, or else a special texture directory. Any number of paths can be seperated by semi colons, but each must have a terminating back slash e.g.; "c:\genesis\textures\;c:\windows\bmps\" or else be a null string. The pointer cannot be NULL.
phobj	A pointer to the handle of the object that will be created.
Report	The address of a callback function to inform the application of how far through the load its done.

Comments

This function takes exactly the same parameters as Geometry's LoadLWObject function, so the implementation of the geometry function is nothing more than calling this helper routine.