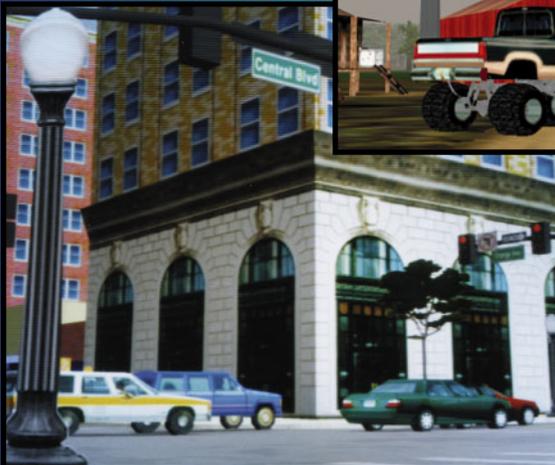


REAL 3D PRO-1000

Professional Series



Developer's Guide

REAL 3D PRO-1000

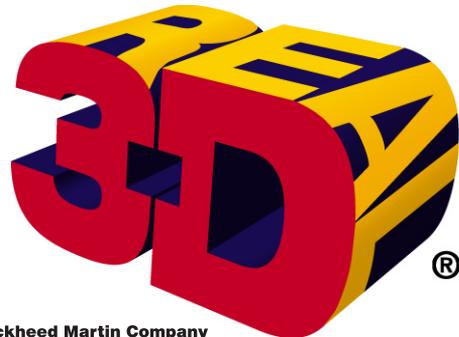
Professional Series

Developer's Guide

December 1997

Version 4.0e

Copyright Real 3D, 1997
All Rights Reserved



a Lockheed Martin Company



IMPORTANT NOTICE

While every effort has been made to ensure the quality of this document, the information and material contained in this book is provided "as is", without warranty of any kind, expressed or implied, including without limitation any warranty concerning the accuracy, adequacy, or completeness of such information or material or the results to be obtained from using such information or material. Neither Real 3D, Lockheed Martin Corporation nor the authors shall be responsible for any claims attributed to errors, omissions, or other inaccuracies in the information or material contained in this book. In no event shall the Corporation be liable for direct, indirect, special, incidental, or consequential damages arising from the use of such information or material.

This document contains the intellectual property of Real 3D, a Lockheed Martin Company, and may not be copied or duplicated in any form, in whole or in part, without the prior written permission of Real 3D or Lockheed Martin Corporation.

The Real 3D® products are protected by one or more of the following patents:

4205389, 4446529, 4583185, 4586038, 4612482, 4692880, 4714428, 4715005, 4727365,
4811245, 4821212, 4825391, 4841467, 4855937, 4862388, 4868771, 4888711, 4905164,
4958305, 4965745, 4974176, 5126726, 5187754, 5191642, 5268996, 5293467, 5319744,
5357579, 5367615, 5420970.



Real 3D® is a registered trademark of Lockheed Martin Corporation. (Reg. US. Pat. & TM Office)

All other product names and services identified throughout this publication are trademarks or registered trademarks of their respective companies. Such uses, or the use of any trade name, is not intended to convey endorsement or other affiliation with this publication.

Caution: Use of Real 3D and Lockheed Martin Corporation products for the unauthorized copying, modification, distribution, or creation of derivative works from copyrighted works such as images, photographs, text, drawings, films, videotape, music, sound recordings, recorded performances, or portions thereof may violate applicable copyright laws.

Manufactured in the United States of America.

Real 3D

a Lockheed Martin Company
12506 Lake Underhill Road
Orlando, Florida 32825-5002

web: <http://www.real3d.com/>

e-mail: real3d@real3d.com

tele: 1-800-393-7730 (U.S.A.)

1-407-306-7302 (Worldwide)

fax: 1-407-306-3358 (Worldwide)



CONTENTS – NEXT PAGE

REAL 3D Developer's Guide	1
Title Page	2
Important Notice	3
Contents	4
Introduction	28
About This Guide	28
About This Software	30
Hardware Overview	31
System Architecture	31
System Configurations	33
Display Interface	34
Hardware Capacity	35
Coordinate Sets	35
Polygons	35
Texture	35
Concepts and Object Data Types	36
API Concepts and Conventions	36
Blending and Levels-of-Detail	36
Lighting and Weather Effects	37
Coordinate Systems	38
Order of Rotation Angles	38
World	38



CONTENTS – NEXT PAGE

Local	39
Viewpoint	40
Offline Formatting Mode	41
Dynamic Paging	41
Video Output Controls	43
Gamma Correction	43
Soft Edge Blending	43
API Data Types	44
Device	44
Viewport	45
Viewpoint	45
Database	45
Models	46
Pre-formatted	46
Run-time	46
Instancing	46
Animation Sequences	47
Culling Nodes	47
LOD Culling Nodes	48
Articulated Part Culling Nodes	48
Animation Culling Nodes	48
Instance Culling Nodes	48
Instance Reference Culling Nodes	48
Point Lights	49
Instance Sets	49



CONTENTS – NEXT PAGE

Polygons	50
Layered Polygons	50
Translucency	50
High Priority Polygons	50
Shading and Lighting	50
Texture	51
Color	51
Level-of-Detail (LOD) Table	51
Color Table	51
Functional Groups	52
Configuration Functions	52
Hardware Interface and Global Functions	52
Device Functions	54
Viewport Functions	56
Database Functions	58
Viewpoint Functions	59
Blend Table Functions	60
Color Table Functions	61
Model Functions	62
Animation Functions	67
Culling Node Functions	68
LOD Culling Node Functions	70



CONTENTS – NEXT PAGE

Articulated Part Culling Node Functions	71
Animation Culling Node Functions	73
Instance Culling Node Functions	74
Instance Reference Culling Node Functions	75
Point Light Functions	76
Instance Set Functions	77
Polygon Functions	78
Vertex Functions	80
Texture Functions	81
Microtexture Functions	82
Matrix Functions	83
MultiGen Loader Functions	85
SAMPLE APPLICATIONS	86
Sample 1	87
Sample 2	90
Sample 3	93
Sample 4	98



CONTENTS – NEXT PAGE

FUNCTION REFERENCE	102
Configuration Functions	102
PRO_ConfigureMemory	102
PRO_AbbreviatedConfigureMemory	104
PRO_SetLogicalUnitOn	106
PRO_SetLogicalUnitOff	107
Hardware Interface and Global Functions	108
PRO_Init	108
PRO_InitOffline	110
PRO_Stop	111
PRO_DisplayDatabase	112
PRO_LoadModelDynamically	113
PRO_UnloadDynamicModel	115
PRO_SetGenlock	116
PRO_SetUpdateRate	117
PRO_SetOverloadMode	118
PRO_SetDisplaySelect	119
PRO_SetForegroundColor	120
PRO_SetBackgroundColor	121
PRO_SetGamma	122
PRO_SetPointLightDetail	123
PRO_SetPointLightFeatureType	124
PRO_SetPointLightSize	125
PRO_SetPointLightPolygon	126
PRO_SetMicrotextureMapCount	127



CONTENTS – NEXT PAGE

PRO_SetTexturePageSize	128
PRO_ReloadLogo	129
PRO_GetNumberOfLogicalUnits	130
PRO_GetNumberOfAvailableTexels	131
PRO_GetMicrotextureMapCount	132
PRO_GetFirmwareRevision	133
PRO_GetModelNumber	134
PRO_GetRealtimeClockCount	135
PRO_GetProcessingTime	136
PRO_GetFrameRateEstimate	137
PRO_GetPolygonMemorySize	138
PRO_GetDefaultDevice	139
Device Functions	140
PRO_Device class constructor	140
PRO_Device class destructor	142
GetStatus	143
AttachViewport	144
DetachViewport	145
SetGamma	146
SetGammaRGB	147
ActivateGamma	149
DeactivateGamma	150
SaveGammaToROM	151
WriteBlendMemory	152
ActivateBlendMemory	154



CONTENTS – NEXT PAGE

DeactivateBlendMemory	155
SaveBlendMemoryToROM	156
SetForegroundColor	157
SetForegroundColor	158
SetBackgroundColor	159
SetBackgroundColor	160
SetDisplaySelect	161
SetOverloadMode	163
GetLogicalUnit	164
IsValidDisplaySelect	165
GetSyncSelectInfo	166
HasAttachedViewports	167
GetGamma	168
GetForegroundColor	169
GetBackgroundColor	170
GetDisplaySelect	171
GetOverloadMode	172
GetLineRate	173
IsHighRes	174
HasExpandedTexture	175
GetFirmwareRevision	176
GetModelNumber	177
GetRealtimeClockCount	178
GetProcessingTime	179
GetFrameRateEstimate	180
GetCurrentFrameCount	181
GetHWPolygonMemorySize	182



CONTENTS – NEXT PAGE

Viewport Functions	183
PRO_Visport class constructor	183
PRO_Visport class destructor	184
GetStatus	185
AttachDefaultBlendTable	186
AttachDefaultDatabase	187
AttachBlendTable	188
AttachDatabase	189
AttachViewpoint	190
DetachViewpoint	191
AttachChildViewport	192
DetachChildViewport	193
SetFieldOfView	194
SetFieldOfView	195
SetDisplayExtents	196
SetSunIlluminationData	197
SetSunVector	198
SetFogParameters	199
SetFogRanges	201
SetFogColor	202
SetLightFogColor	203
SetLightSourceColor	204
SetLightSourceParameters	205
SetLightSourceStartRange	206
SetLightSourceExtent	207



CONTENTS – NEXT PAGE

SetLobeParameters	208
SetViewportOff	209
SetViewportOn	210
SetDisplaySelect	211
SetInvertFlags	212
GetViewpoint	213
GetAttachedDevice	214
GetFieldOfView	215
GetFieldOfView	216
GetDisplayExtents	217
GetSunIlluminationData	218
GetSunVector	219
GetFogParameters	220
GetFogRanges	221
GetFogColor	222
GetLightFogColor	223
GetLightSourceColor	224
GetLightSourceParameters	225
GetLobeParameters	226
GetPriority	227
Database Functions	228
PRO_Database class constructor	228
PRO_Database class destructor	229
GetStatus	230
AttachModel	231



CONTENTS – NEXT PAGE

DetachModel	232
ActivateAllModels	233
DeactivateAllModels	234
HasActiveModels	235
Viewpoint Functions	236
PRO_Viewpoint class constructor	236
PRO_Viewpoint class destructor	237
GetStatus	238
SetPosition	239
SetPosition	240
SetOrientation	241
SetOrientation	242
SetOrientation	243
GetPosition	244
GetPosition	245
GetOrientation	246
GetOrientation	247
GetLocalForward	248
GetLocalLeft	249
GetLocalUp	250
AttachViewpoint	251
DetachViewpointFromParent	252
Blend Table Functions	253
PRO_LOD_Table class constructor	253
PRO_LOD_Table class destructor	254



CONTENTS – NEXT PAGE

GetStatus	255
SetRangeBlendParameters	256
SetSizeBlendParameters	257
GetRangeBlendParameters	258
GetSizeBlendParameters	259
GetAllocatedTableSize	260
Color Table Functions	261
PRO_Color_Table class constructor	261
PRO_Color_Table class destructor	262
GetStatus	263
SetColor	264
SetColor	265
GetAllocatedTableSize	266
Model Functions	267
PRO_Model class constructor	267
PRO_Model class constructor	269
PRO_Model class constructor	270
PRO_Model class constructor	271
PRO_Model class destructor	272
GetModelStatus	273
SetPosition	274
SetPosition	275
SetPosition	276
SetPosition	277
SetOrientation	278



CONTENTS – NEXT PAGE

SetOrientation	279
SetOrientation	280
SetOrientation	281
SetOrientation	282
SetOrientation	283
Activate	284
Deactivate	285
AttachColorTable	286
SetRangeLOD	287
SetSizeLOD	288
UserMatrix	289
IsStatic	290
IsActive	291
GetPosition	292
GetPosition	293
GetPosition	294
GetPosition	295
GetOrientation	296
GetOrientation	297
GetOrientation	298
GetOrientation	299
GetLocalForward	300
GetLocalForward	301
GetLocalForward	302
GetLocalForward	303
GetLocalLeft	304



CONTENTS – NEXT PAGE

GetLocalLeft	305
GetLocalLeft	306
GetLocalLeft	307
GetLocalUp	308
GetLocalUp	309
GetLocalUp	310
GetLocalUp	311
GetNumberOfAnimationSequences	312
GetAnimationSequence	313
GetAnimationSequence	314
GetAnimationSequenceName	315
GetNumberOfArticulatedParts	316
GetArticulatedPartMatrixIndex	317
GetArticulatedPartName	318
GetNumberOfTextureMaps	319
GetTextureMapName	320
ProcessImmediateModeData	321
ProcessMorphedPolygons	322
AddCullingNode	323
AddChild	324
AddPointLight	325
AddInstanceSet	326
AddPolygon	327
AddChild	328
EnableVertexSharing	329
DisableVertexSharing	330



CONTENTS – NEXT PAGE

SetAutoDeleteCullingData	331
SetAutoDeletePolygonData	332
SetAutoDestructCullingData	333
SetAutoDestructPolygonData	334
StoreArticulatedPart	335
StoreAnimationSequence	336
StoreTextureData	337
StoreTextureData	338
Save	339
Animation Functions	340
AnimationStartsVisible	340
AnimationEndsVisible	341
AnimationStartsInvisible	342
AnimationEndsInvisible	343
ActivateSequence	344
DeactivateSequence	345
AnimationRunsForward	346
AnimationRunsBackwards	347
AnimationRunsMirrored	348
AnimationIsUserControlled	349
AnimationRunsForever	350
SetAnimationCycleTime	351
SetAnimationCycleTime	352
SetAnimationRepeatCount	353
SetFrameActive	354



CONTENTS – NEXT PAGE

SetFrameInactive	355
GetAnimationCycleTime	356
GetAnimationCycleTime	357
GetNumberOfAnimationFrames	358
GetCurrentFrame	359
GetAnimationState	360
Culling Node Functions	361
PRO_Culling_Node class constructor	361
PRO_Culling_Node class destructor	362
GetStatus	363
ActivateNode	364
DeactivateNode	365
AddChild	366
AddChild	367
AddPointLight	368
AddInstanceSet	369
DetachChild	370
DetachChild	371
AttachColorTable	372
SetRangeLOD	373
SetSizeLOD	374
SetFeatureType	375
SetGroupLayerNumber	376
SetVolumeSizeClamp	377
SetAutoDeleteCullingData	378



CONTENTS – NEXT PAGE

SetAutoDeletePolygonData	379
SetAutoDestructCullingData	380
SetAutoDestructPolygonData	381
FormatAndFlushPolygons	382
GetFeatureType	383
GetBoundingBox	384
GetLODNumber	385
GetGroupLayerNumber	386
GetCentroid	387
HasAssociatedMatrix	388
LOD Culling Node Functions	389
PRO_LOD_Culling_Node class constructor	389
PRO_LOD_Culling_Node class destructor	390
AddChild	391
AddChild	392
Articulated Part Culling Node Functions	393
PRO_6DOF_Culling_Node class constructor	393
PRO_6DOF_Culling_Node class constructor	394
PRO_6DOF_Culling_Node class destructor	395
SetName	396
SetPosition	397
SetPosition	398
SetOrientation	399
SetOrientation	400
SetOrientation	401



CONTENTS – NEXT PAGE

SetMotionExtents	402
SetMotionExtents	403
SetMatrix	404
ReplaceMatrix	405
GetName	406
GetPosition	407
GetOrientation	408
GetMatrix	409
Animation Culling Node Functions	410
PRO_Animation_Culling_Node class constructor	410
PRO_Animation_Culling_Node class destructor	411
AddChild	412
AddChild	413
SetName	414
GetName	415
GetAnimationSequence	416
Instance Culling Node Functions	417
PRO_Instance_Culling_Node	417
ProcessImmediateModeData	418
EnableVertexSharing	419
DisableVertexSharing	420
Instance Reference Culling Node Functions	421
PRO_Instance_Reference_Culling_Node class constructor	421
SetInstanceLocation	423
GetInstanceLocation	424



CONTENTS – NEXT PAGE

Point Light Functions	425
PRO_Point_Light class constructor	425
PRO_Point_Light class destructor	426
SetColor	427
SetColor	428
SetPosition	429
GetPosition	430
Instance Set Functions	431
PRO_Instance_Set class constructor	431
PRO_Instance_Set class destructor	432
SetColor	433
SetColor	434
SetPosition	435
GetPosition	436
SetActiveInstanceCount	437
GetActiveInstanceCount	438
GetNumberOfInstances	439
Polygon Functions	440
PRO_Polygon class constructor	440
PRO_Polygon class destructor	441
AddVertex	442
AddChild	443
DetachChild	444
AttachTexture	445
DetachTexture	446



CONTENTS – NEXT PAGE

AttachMicroTexture	447
DetachMicroTexture	448
SetColor	449
SetColor	450
SetColorIndex	451
SetSensorColorIndex	452
EnableLightingEffects	453
DisableLightingEffects	454
SetTranslucency	455
SetLightModifier	456
SetHighPriority	457
PolygonIsLayered	458
IsDoubleSided	459
DoSmoothShading	460
DoFlatShading	461
IsNotShaded	462
VerticesAreClockwise	463
VerticesAreCounterClockwise	464
SetXSmoothing	465
SetYSmoothing	466
SetXMirror	467
SetYMirror	468
SetNPScale	469
SetTranslucencyPatternSelect	470
EnableAutoVertexDelete	471
DisableAutoVertexDelete	472



CONTENTS – NEXT PAGE

GetLODNumber	473
GetVertexCount	474
GetCompressedVertexCount	475
GetVertex	476
IsTextured	477
Vertex Functions	478
PRO_Vertex class constructor	478
PRO_Vertex class constructor	479
PRO_Vertex class destructor	480
SetCoordinates	481
SetCoordinates	482
SetMorphCoordinates	483
SetTextureCoordinates	484
SetVertexNormal	485
SetFixedShadingIntensity	486
OffsetVertex	487
ScaleVertex	488
TransformVertex	489
GetCoordinates	490
GetCoordinates	491
GetTextureCoordinates	492
GetVertexNormal	493
IsTextured	494
IsShaded	495



CONTENTS – NEXT PAGE

Texture Functions	496
PRO_Texture class constructor	496
PRO_Texture class constructor	498
PRO_Texture class destructor	500
IsContourTexture	501
SetContourThreshold	502
SetContourThreshold	503
EnableAlpha	504
DisableAlpha	505
SetTextureFileName	506
KeepDataAfterLoading	507
FreeDataAfterLoading	508
DeleteDataAfterLoading	509
GetTextureFileName	510
GetOrigTextureMapWidth	511
GetOrigTextureMapHeight	512
IsTextureRGB	513
IsMicroTexture	514
Microtexture Functions	515
PRO_MicroTexture class constructor	515
PRO_MicroTexture class constructor	517
PRO_MicroTexture class destructor	518



CONTENTS – NEXT PAGE

Matrix Functions	519
PRO_Matrix class constructor	519
PRO_Matrix class destructor	520
GetStatus	521
SetPosition	522
SetPosition	523
SetPosition	524
SetOrientation	525
SetOrientation	526
SetOrientation	527
SetLocalForward	528
SetLocalLeft	529
SetLocalUp	530
GetPosition	531
GetPosition	532
GetPosition	533
GetOrientation	534
GetOrientation	535
GetLocalForward	536
GetLocalLeft	537
GetLocalUp	538
AttachMatrix	539
DetachFromParent	540



CONTENTS – NEXT PAGE

MultiGen Loader Functions	541
PRO_MGen_Model class constructor	541
PRO_MGen_Model class destructor	543
GetPolyCount	544
GetVertCount	545
Glossary	546
Appendix – Cross References of C++ to C Function Calls	550
Configuration Functions	550
Hardware Interface and Global Functions	551
Device Functions	553
Viewport Functions	558
Database Functions	564
Viewpoint Functions	565
Blend Table Functions	567
Color Table Functions	569
Model Functions	570
Animation Functions	581
Culling Node Functions	584
LOD Culling Node Functions	588
Articulated Part Culling Node Functions	589
Animation Culling Node Functions	593



CONTENTS – NEXT PAGE

Instance Culling Node Functions	595
Instance Reference Culling Node Functions	596
Point Light Functions	597
Instance Set Functions	598
Polygon Functions	600
Vertex Functions	604
Texture Functions	607
Microtexture Functions	609
Matrix Functions	610
Index	614



INTRODUCTION

Thank you for purchasing the Real 3D PRO-1000 Professional Series Image Generator and welcome to the world of Real 3D Graphics. Our goal is to provide you with the highest quality product, superior customer support, and a product that is easy to use.

ABOUT THIS GUIDE

The Real 3D PRO-1000 Professional Series Developer's Guide documents the Applications Programming Interface (API) for the Real 3D PRO-1000 series of image generators. This API is a library of functions that provides a higher level of interface to the PRO-1000 rendering engine. Figure 1 shows a typical graphics application in which the PRO/API library interfaces with the PRO-1000 hardware to prepare and render graphical objects. Detailed descriptions of the PRO/API graphics library functions, as well as general information about the data objects used by the library, are contained in this guide. A general overview of the PRO-1000 hardware capabilities is also provided.

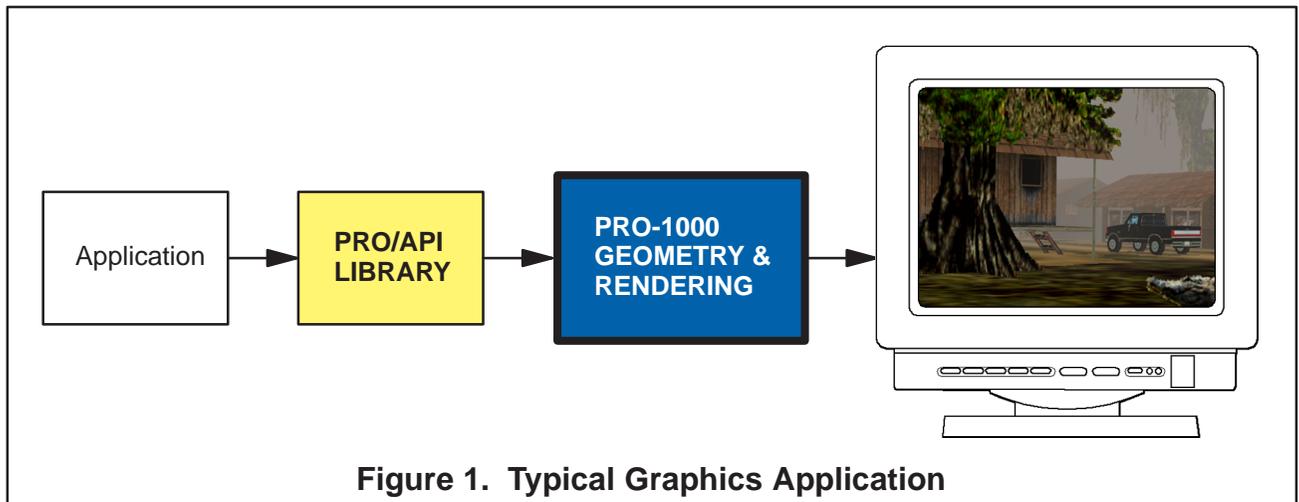


Figure 1. Typical Graphics Application



Introduction

The PRO/API library functions can be grouped according to their functionality. To locate a specific function, refer to the *Functional Group* section.

To further assist the application developer, an Adobe PDF version of this document is available from Real 3D. This full color electronic document provides thumbnails, bookmarks, and hot links to various portions of this guide. Information contained in the PRO-1000 Professional Series Developer's Guide is arranged in the following sections: Each section of this manual features a section overview highlighting key topics and concepts presented in that section. The back portions of this manual contain a glossary and index. If you require assistance not available in this guide, please refer to the *Title Page* for Real 3D contact information.

- ***Introduction*** – describes the objectives of the PRO-1000 API and explains the use, function and structure of the Developer's Guide
- ***Hardware Overview*** – provides a brief overview of system performance and hardware features of the PRO-1000 Professional Series Image Generator
- ***Concepts and Object Data Types*** – describes API real-time concepts, objectives and advantages of the PRO-1000 image generator
- ***Functional Groups*** – provides an overall description of the functions available through the PRO-1000 API. These functions are sorted by group type and are described in detail in the *Function Reference* section
- ***Sample Applications*** – sample programs are provided to illustrate concepts and features of PRO-1000 API. These programs are provided to augment the detailed descriptions of specific functions described in the *Function Reference* section
- ***Function Reference*** – describes in detail each function available through the PRO-1000 API library
- ***Glossary*** – provides easy access to definitions of unfamiliar terms or acronyms
- ***Index*** – a fast cross-reference to specific subjects by page location



ABOUT THIS SOFTWARE

The PRO/API library functions provide the mechanism to create the display list supported by the PRO-1000 hardware, manipulate the display list for specific application needs, and load the graphical objects into the hardware memory for rendering. The graphical object types are described in more detail in the Functional Group Section.

The PRO/API library runs on personal computers that are based on Intel architecture, running Microsoft Windows NT. The PRO/API also runs on Sun SPARCStations running Solaris 2.X. This manual primarily presents PRO/API functions callable from C++ application programs; C-callable functions are also documented.

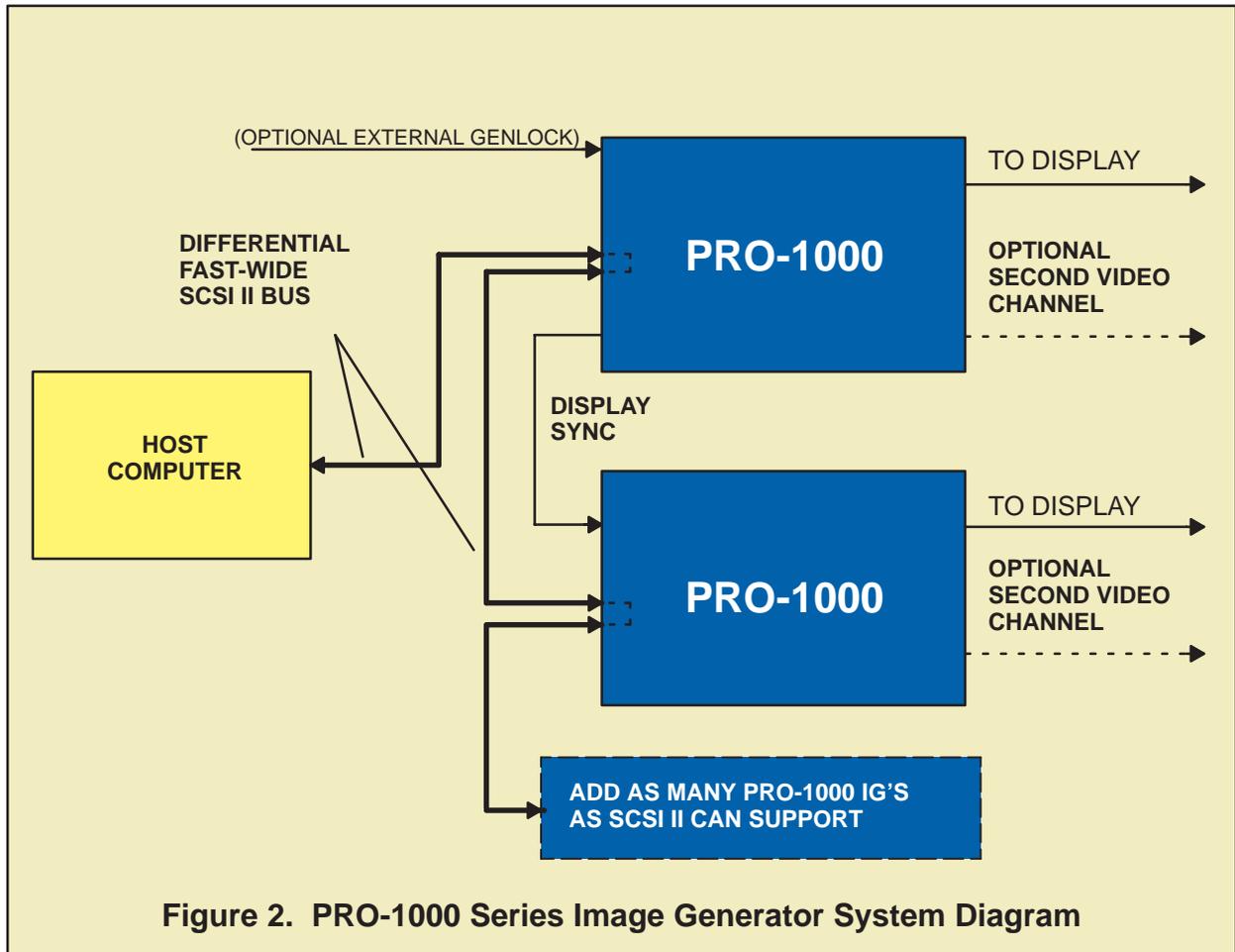


HARDWARE OVERVIEW

The REAL 3D PRO-1000 visual system is a true low-cost, high-performance image generator for serious simulation applications. All image related computations are performed in the image generator. The REAL 3D PRO-1000 is controlled by a user-supplied personal computer or workstation which acts as host for control programs. The Image Generator system off-loads intensive floating point computations and all “out-the-window” image management tasks from the host computer to achieve a sustained image update rate of 30 or 60 frames per second. The basic PRO-1000 pixel fill rate is 50 megapixels per second, and may be expanded to 100 megapixels per second by simply adding a second pixel fill daughter board to the system. A high-resolution model with four pixel boards offers a 200 megapixel per second fill rate. This is especially important for high resolution and 60 Hz update requirements. Additionally, 8 MB of on-board MIP-mapped texture random access memory is available. REAL 3D’s patented MIP-mapped, tri-linear interpolation and microtexture algorithms are used to process visual scenes of unmatched quality. Anti-aliasing is performed as a post process using a Patent Pending proprietary image processing algorithm.

SYSTEM ARCHITECTURE

A system configuration diagram of the PRO-1000 series image generator is depicted in Figure 2. As illustrated, multiple PRO-1000 systems may be “chained” together to provide superior application flexibility and system expandability. A differential fast-wide SCSI II bus provides the interface from the user provided host computer to the PRO-1000 system(s). The host computer may be a PC or a workstation.





SYSTEM CONFIGURATIONS

The REAL 3D PRO-1000 Professional Series image generator supports multiple configurations. The standard PRO-1000 configurations are listed by model number in Table 1. Please contact REAL 3D for specific system configuration information.

Table 1. PRO-1000 Series System Configurations

MODEL NUMBER	PIXEL PROCESSORS	PIXEL WRITES
1100	1	50M
1200	2	100M
1400	4	200M



DISPLAY INTERFACE

The PRO-1000 display resolution is programmable. The standard set of resolutions supplied with the PRO-1000 system are listed in Table 2 and Table 3. This set may be customized.

Table 2. PRO-1000 Standard Resolutions (Single Channel)

RESOLUTION	INTERLACE	DISPLAY UPDATE RATE	MODEL NUMBER
640H X 480V	non-interlace	60 Hz.	1100/1200/1400
1024H X 768V	non-interlace	60 Hz.	1400
512H X 486V	interlace	59.94 Hz.*	1100/1200
720H X 486V	interlace	59.94 Hz.*	1400
720H X 576V	interlace	50 Hz.	1400

* – Application programmers should treat this as 60 Hz.

The following table shows the standard configurations supported in dual monitor mode. In the dual monitor mode, both monitors must have the same resolution.

Table 3. PRO-1000 Standard Resolutions (Dual Channel)

RESOLUTION	INTERLACE	DISPLAY UPDATE RATE	MODEL NUMBER
640H X 480V	non-interlace	60 Hz.	1400
512H X 486V	interlace	59.94 Hz.*	1200
640H X 486V	interlace	59.94 Hz.*	1400

* – Application programmers should treat this as 60 Hz.



HARDWARE CAPACITY

COORDINATE SETS

Any model or viewpoint that will be positioned or rotated must have an associated direction cosine matrix. Storage is provided for up to 4096 direction cosine matrices, which allows up to 4096 coordinates sets to be processed.

POLYGONS

Eight megabytes of polygon memory is provided. With 8 MB of memory, a typical database can include over 100,000 polygons of active database storage. Additional database memory can be added in 8 MB increments by adding memory daughter boards, up to a total of 32 MB of polygon memory capable of storing over 400,000 polygons.

TEXTURE

There are 4 million texture memory locations. An option exists to increase texture storage to 16 million texture memory locations. Each texture memory location is a 16-bit word. Texture map size may range from 32 by 32 to 1024 by 1024 texels in increments of powers of 2. Texture maps may be square (e.g., 64 x 64) or rectangular (e.g., 256 x 512). Microtexture and texture MIP levels are included in the texel count.



CONCEPTS AND OBJECT DATA TYPES

API CONCEPTS AND CONVENTIONS

BLENDING AND LEVELS-OF-DETAIL

Most realtime computer graphics systems require the use of polygon and processing load management controls in order to maintain a steady rendering update rate. The number of polygons processed by the rendering engine and the pixel fill rates are typically the two most significant measurements affecting a realtime system's ability to maintain the desired graphics update rate (overload).

One of the features available in the PRO-1000 series system which allows users to manage the realtime rendering process more efficiently is the use of levels of detail (LODs). LODs provide a simple mechanism for the user to define how elements that make up the scene being rendered may be replaced by simpler versions of themselves as they get further away. The PRO-1000 series achieves a smooth transition between the different LODs by using a complementary translucency algorithm which gradually replaces one LOD by another as a function of distance. This avoids distracting anomalies due to the abrupt switch of model/object versions that is typical of lower performance systems.

LOD blending is also used to smoothly eliminate graphic objects that are too small to provide any meaningful contribution to the scene being rendered. This is done by using translucency to fade the object out of the rendered scene as a function of distance from the eyepoint, and then completely discarding the object from the display list at run-time. This saves realtime resources and helps to maintain a high graphics update rate. Up to four LODs are supported by the PRO-1000 system.

LOD blending and discard are influenced by the user through the use of blending or feature types. The PRO-1000 supports 127 application controllable blend types per viewport. Each element of the display list (or database) hierarchy may be assigned a blend type. The different pieces of the graphics scene may then be controlled independently, based on the



blend types assigned by the application. These LOD control parameters define the subtended angle or range the LOD transitions are to occur. Each of the 127 available LOD control entries per viewport contains four sets of maximum and minimum angles or ranges over which a given LOD exists, as well as a transition range or angle over which the LOD complementary translucency transition or feature discard takes place.

LIGHTING AND WEATHER EFFECTS

The PRO-1000 series systems provide a number of different ways in which polygons may be affected by the system lighting parameters. There are several types of polygon lighting available to the application programmer:

1. Self luminous lighting
2. Fixed polygon shading
3. Flat sun lighting
4. Smooth polygon shading

The self-illuminating model option disables all sun lighting effects on a polygon. Polygons using this kind of illumination model are always rendered using their full color, which never gets modulated based on the sun lighting parameters.

The fixed shading polygon illumination model is similar in nature to the self-illumination model in that polygon colors are not affected by the sun lighting parameters. However, the user may specify a fixed shading weight per vertex which will be used to interpolate a shading value for every pixel covered by the polygon on the screen.

The flat sun lighting model calculates a single color intensity modulation value for a polygon, based on the alignment of the polygon with the sun direction, the sun intensity, and the amount of ambient light (minimum color modulation).

The smooth shading lighting model uses vertex normals and the sun direction, intensity, and ambient light parameters to calculate a per vertex lighting value that gets interpolated and applied on a per-pixel basis.

The PRO-1000 series systems provide a realistic polygon fogging algorithm that can be used to control the amount of fading applied to the rendered scene on a per pixel basis. The application may define a set of fog behavior parameters as well as the color of the fog.



The fog parameters available to the user are the distance at which a pixel is to be fully fogged, and either the amount of fog (normalized between 0.0 and 1.0) at the eyepoint OR the distance at which fog starts to have an effect.

COORDINATE SYSTEMS

All coordinate systems in the PRO-1000 system follow the same conventions. This is an important point for application developers. The emphasis here is on the word CONVENTION. There is no coordinate set that is inherently required by the system; however, a set of CONVENTIONS is used in the PRO-1000 Application Programming Interface (PRO/API) software to manipulate the display list. The general coordinate set conventions used by the PRO/API is a Z-up right-handed coordinate system with rotation performed in a X,Y, and Z order.

ORDER OF ROTATION ANGLES

All PRO/API object and viewpoint angles are expressed as rotation angles about the x, y, and z axes. All rotations are counterclockwise positive about the axis of rotation. To transform an object, the object is first rotated about the x axis (roll), then about the y axis (pitch), then about the z axis (yaw). Note that yaw is not heading as normally assumed in many simulation applications; rather, heading = $(90.0^\circ - \text{yaw})$.

Since not all applications use the same coordinate set definitions and conventions, the PRO/API software provides a generalized way of specifying rotation matrices so that the application may use its own coordinate set definitions and conventions.

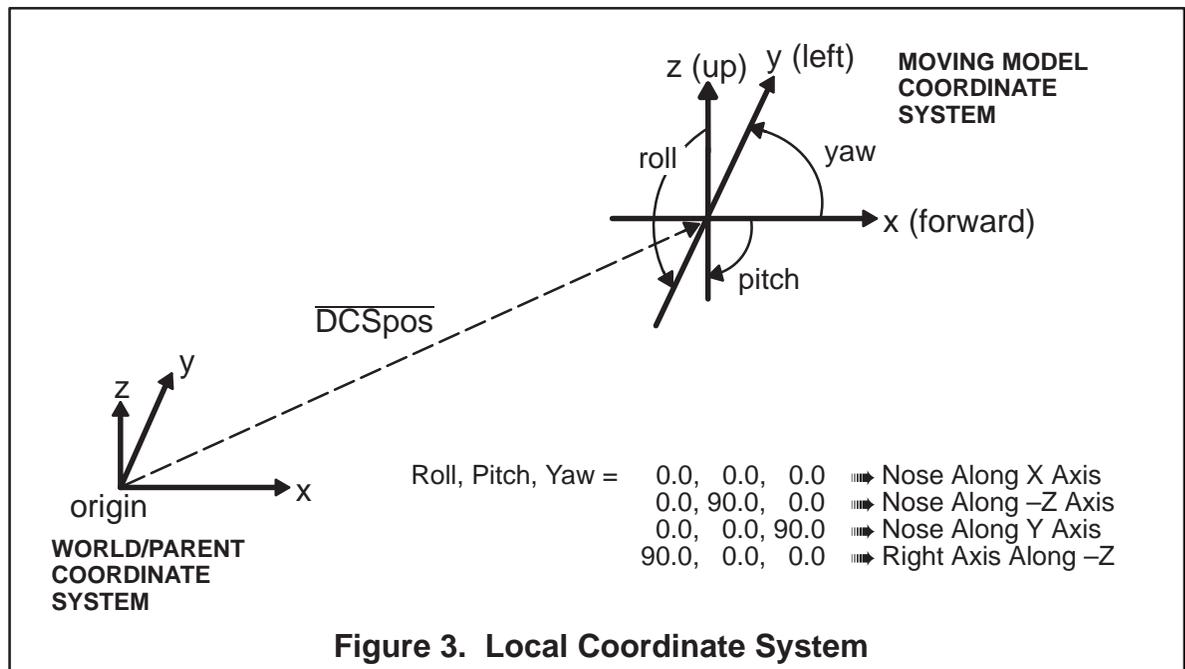
WORLD

The world coordinate system is in reality nonexistent in the system; however, it must be noted that all the geometry of high-level graphics constructs (pre-processed or immediate mode) is defined as being relative to a common reference point (0.0, 0.0, 0.0). The one exception is graphic objects that are associated with a separate coordinate set, in which case their geometry is assumed to be relative to the parent coordinate set. All user controllable coordinate set definitions have a parent, and at the highest levels of the display list hierarchy, the world coordinate system is the parent.



LOCAL

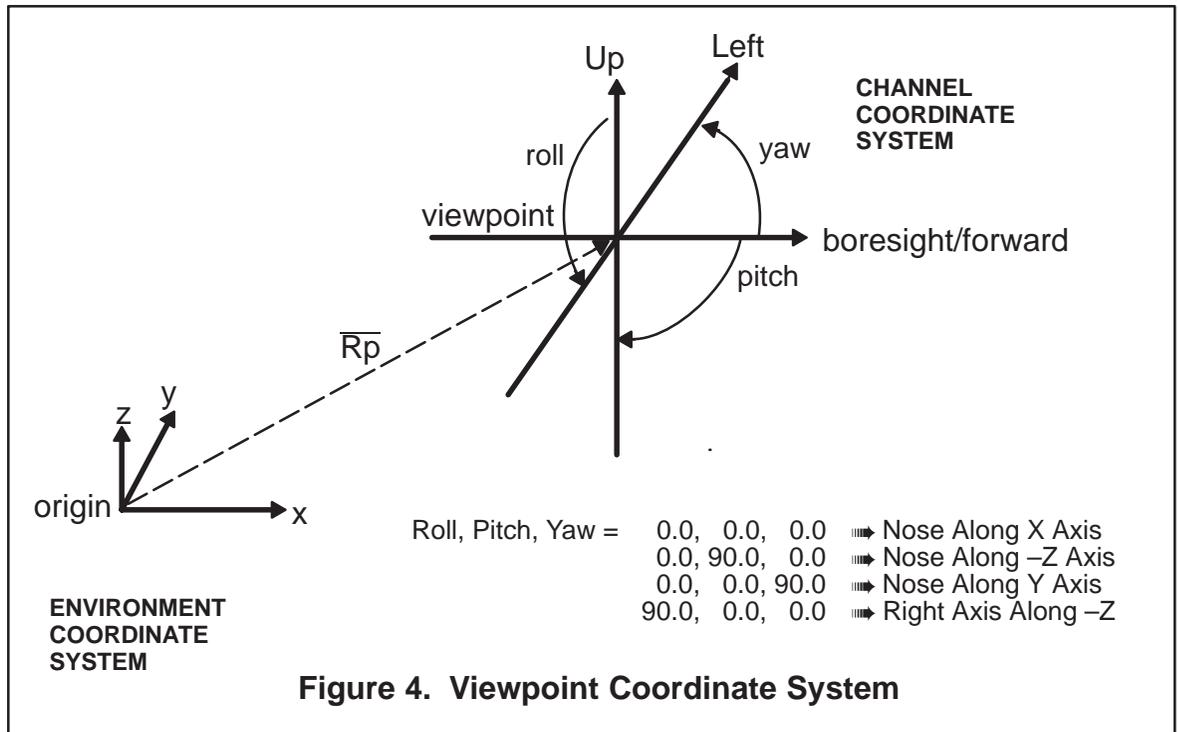
Local coordinate systems are used to manipulate display list elements that require up to six degrees of freedom. The position of the local coordinate set is relative to its parent (the world in most cases) and so are its rotations, as depicted in Figure 3. Fifteen levels of coordinate set nesting (levels of articulation) are supported and the different levels of nesting are defined as parent relative. The \overline{DCSpos} vector represents the position of the local coordinate set relative to the world/parent coordinate set.





VIEWPOINT

The viewpoint coordinate system (eyepoint or viewpoint), as depicted in Figure 4, behaves in the same way as any other coordinate system, with the exception that its position and orientation must always be relative to the world coordinate system. The \bar{R}_p vector represents the location of the viewpoint coordinate system relative to the world origin.





OFFLINE FORMATTING MODE

Offline formatting mode allows the use of the PRO/API internal display list structures to process a model and save the formatted data to file. In offline formatting mode, the display list is not actually sent to the renderer. This means that a complex model can be formatted on a system with no PRO–1000 system attached and saved models can be loaded multiple times without incurring the overhead of run–time formatting.

When developing a formatter application using offline mode, PRO/API objects relating to rendering, such as devices, blend tables, viewports, and databases, need not be created.

DYNAMIC PAGING

Dynamic paging allows applications to load pre–formatted models (processed offline and saved to file) at run time while making sure that real–time performance on the host is not compromised. This dynamic paging is especially useful for applications whose database storage requirements (both texture and geometry) exceed the existing system capacity and do not require all the data to be loaded at the same time. Applications such as flight simulation and extended driving training are able to move smoothly about large area databases which can not be all held in memory.

Dynamic paging, or dynamic database update, is supported for both texture and geometry and its memory management controls are invisible to the user. The functionality provided by the PRO/API in support of this feature has been designed to hide from the application all the file I/O and CPU intensive functions normally associated with loading a model. This is accomplished by time–slicing the complete database load procedure in such a way that it adjusts to the instantaneous demands placed on the CPU by the host application and only uses the spare time left over after processing a real–time field time to perform its function.

The net result and visual impact of dynamic update is that the application can, at any time during its execution, request a new model load, and at some time later the model is made available to the host application through the setting of application pointers and the invocation of user callback when the model load has been completed.

Dynamic paging also supports a model load request queueing mechanism that allows the host applications to request as many model loads as desired, independent of any other model load request it may have previously issued.



Concepts and Object Data Types

In order to maintain system performance, the application needs to ensure that the model load requests do not exceed the system storage capacity. In a typical application, once a visible portion of a given large area database has been loaded, the new model load requests are normally balanced by model deletions, thus allowing the hardware memory management mechanisms built into the API to reclaim and reuse any system and host memory previously used by unloaded (deleted) models.

As part of the dynamic database paging process, texture maps may be reduced in resolution if the texture memory demands are such that they exceed the total system texture memory capacity. There is no performance impact when this occurs, but image resolution will be reduced if the requested space is not readily available at the time of the model load request.



VIDEO OUTPUT CONTROLS

GAMMA CORRECTION

Gamma correction provides a means of color balancing physically different display devices (multiple monitors, projectors, etc.). This feature is especially useful when dealing with display devices with different color output profiles that are required to match as closely as possible to provide a more visually convincing composite image.

The interface provided by the PRO/API allows the host application to adjust the intensity profile of each video output color component (red, green or blue) independently through either a standard single gamma correction value or through the use of color look-up tables. The color look-up tables are 8-bit value tables containing 256 entries, each representing a level of video input and output (8-bits in, 8-bits out), with zero representing the darkest color intensity input into and output from the table, and 255 being the maximum allowable value.

SOFT EDGE BLENDING

Soft edge blending is supported only on high-resolution (1400 series) systems. This is an advanced display device image matching feature intended to be used in conjunction with display projectors so that images that are output by physically separate devices may be seamlessly combined over a single large-display surface.

This feature can be used to account for the non-uniform image intensity profiles usually observed when using display projectors (the center of the display is brighter than the borders due to the lighting profile of the beam produced to project the image in the final surface).

The interface provided by the PRO/API allows applications to define a matrix of color intensity attenuation values across the complete display output (up to two video outputs on a 1400) on a per color component basis (red, green and blue). This attenuation factor matrix is defined at a resolution that is lower than that of the physical display by a factor of eight (8), effectively allowing for the specification of a single triplet of attenuation values (red, green and blue) for every 8 pixels on the screen across the vertical and horizontal directions.

The intermediate pixel attenuation values are calculated by the hardware through the use of bilinear interpolation between the four nearest user defined grid points in screen space.



API DATA TYPES

DEVICE

A device corresponds to a physical PRO–1000 system. The device type allows attributes such as overload mode, display select/line rate, foreground and background color to be specified individually for each unit. This is especially important if the attached PRO–1000 units are of different types (e.g., a 1200 and a 1400). Devices are identified by a logical unit number, which corresponds to the order in which the systems are physically chained on the SCSI bus.

The PRO/API supports the concept of a default device. Initially, this device acts as a data broadcast mechanism to all active attached units. This allows default attributes to be set which apply to all devices. However, once an application creates a device object corresponding to a particular unit, it is assumed that the new device is distinct from all others, and therefore default attribute changes are no longer applied to that device. This implies an order dependency that application programmers should recognize. For example, code snippet one, listed below, results in all devices (including the new device) having a red background; code snippet two results in device 0 having the default black background and all other devices having a red background.

1. `PRO_Init() ;`
`PRO_SetBackgroundColor(255, 0, 0) ;`
`dev0 = new PRO_Device(0);`
2. `PRO_Init() ;`
`dev0 = new PRO_Device(0) ;`
`PRO_SetBackgroundColor(255, 0, 0) ;`

Since the default device is a concept rather than a physical device, it has no intrinsic status information associated with it. Requests for hardware feedback (e.g., frame rate, model number) from the default device returns data from the first active logical unit.



VIEWPORT

A viewport defines a rectangular area of the screen in which the 3D scene will be rendered. As many viewports as desired can be processed; however, the number of viewports may be limited by the processing resources of the system. Viewports may overlap and may be contained within another viewport. The hierarchy of the viewports defines their priority, and thus the order in which viewports are processed. A viewport has a lower priority than its children, but the same priority as its siblings. If the priority levels of multiple viewports are the same and the viewports overlap, the polygons will merge and become one combined scene in the overlapping areas.

A viewport defines its size and position on the display, and its field-of-view. It also defines sun illumination parameters, sun position, fog attributes, and headlight (lobe) position and size. A viewport must be attached to a device in order to output the viewport's data to that device only. At creation, all viewports are set to broadcast to all active logical units, that is, they are initially attached to the default device.

VIEWPOINT

A viewpoint allows control over the viewing position and orientation of a viewport. Multiple viewports may be controlled by a single viewpoint. Viewpoints can also be attached to other viewpoints, making viewpoint relative dynamics easy to implement.

DATABASE

A database defines a group of models that comprise a scene. For a database to be rendered, it must have attached models and must itself be attached to a viewport. Multiple viewports may be attached to the same database (scene).



MODELS

The PRO/API supports two types of model/geometry data: pre-formatted models that are loaded from a file on disk, and run-time generated models. A model may have sub-parts that can be positioned and oriented relative to their topological parent. A model or its articulated parts may have multiple representations (or animation frames) and levels of detail. Model instancing is supported, allowing the same geometry to be referenced more than once in the database hierarchy without replicating polygon data. If the polygons associated with a model or culling node are created so that they reference a color table, each model or culling node instance may reference a separate color table.

PRE-FORMATTED

Pre-formatted models are models whose geometry and attribute data were processed “offline” and stored in a disk file. When a model is created through the PRO/API, the name of the model is provided and the file(s) are read into the display list structures for processing.

RUN-TIME

Run-time models are models whose geometry and attribute data are determined by an application. At run-time, the application creates culling nodes and polygons, sets their attributes, and links them together to create the desired model hierarchy.

INSTANCING

Instancing allows the applications to share display list segments to reference the same model geometry from different places in the database. Individual instances can be positioned differently, but share the same polygons.



ANIMATION SEQUENCES

Animation sequences are created by modeling multiple versions of the same model (animation frames) and cycling through them at run-time. Animation sequences may be used to depict effects such as explosions, fire, or flashing lights. Animation sequences are supported for polygon data and texture animation, as well as model swapping.

The PRO API supports a variety of mechanisms for displaying animation sequences. An application may specify that an animation cycle runs through its frames in a forward sequence or in a reverse sequence, or that it runs forward and then back again to its initial frame. The user determines whether the animation sequence should be under full user control or whether it should run automatically after activation. “Automatic” animations may be run for a set number of iterations, for a set period of time, or run continuously until deactivated. These animations must be activated through an explicit function call. A user-controlled animation is manipulated directly and only by the application, giving the ability to activate or deactivate any combination of animation frames during any real-time cycle. No separate function call is required to start or stop a user-controlled animation. The application also controls whether an animation is visible before it is activated and whether it remains visible after it is deactivated.

CULLING NODES

Culling nodes are used to determine if a model is in the field-of-view and to select the level-of-detail to be rendered.

Culling nodes are connected by child and sibling linkages to create a directed tree-structured database hierarchy. A culling node can have either culling nodes or polygons as its children and can only be attached to a single parent, with the noted exception of instance nodes. The culling node bounding volume encompasses all of its children for real-time culling purposes. A culling node that points directly to polygons is called a “leaf node”. There is a limit of 15 levels of child nodes (nesting), not including the polygon nodes. Once the culling hierarchy is built, it must be attached to a model class instance before it can be rendered.

A culling node may reference a color table that becomes the default for all of the node’s children. Each subsequent generation in the culling hierarchy may reference a different color table. This allows a different color tables to be used for each instance of a model.



There are several sub–types of culling nodes, described in the following paragraphs.

LOD CULLING NODES

LOD culling nodes are used to build a culling hierarchy for an object with different representations at different levels of detail. Each child (culling node or polygon) added to an LOD culling node specifies the level of detail at which the child is valid. Up to four LODs are supported by the PRO-1000 system. The LOD transition between the different representations of the model is controlled by the feature type of the culling node and the blend table used by the viewport in which the geometry is being displayed.

ARTICULATED PART CULLING NODES

An Articulated Part culling node, or six degree–of–freedom node, is used to define geometry that can move relative to the parent coordinate set to which it is attached. Fifteen levels of coordinate set nesting (levels of articulation) are supported.

ANIMATION CULLING NODES

Animation culling nodes are used to build a culling hierarchy for an object with different representations, or animation frames. which can be turned on and off by the application. Each child (culling node or polygon) added to an Animation culling node specifies the frame for which the child is valid.

INSTANCE CULLING NODES

Instance culling nodes define the top of a shared display list segment that can be referenced from other parts of the scene display list.

INSTANCE REFERENCE CULLING NODES

Instance Reference culling nodes are used to place shared geometry. When an Instance Reference culling node is created, it must be passed the Instance culling node to which it refers. An Instance Reference node is considered a leaf node; its “child” is the shared geometry segment. An Instance Reference may be attached to a parent node and may not have any other children, but may have siblings.



If the shared geometry represents the entire object (e.g., a tree), the Instance Reference is attached directly to the model, and the model is used to position and rotate the reference. If the shared geometry represents a portion of the entire object and that portion requires individual movement (e.g., a foot on a human), the Instance Reference is attached to an Articulated Part node which is used to position and rotate the reference. In the latter example, of course, the entire human can be positioned or rotated at the model level.

POINT LIGHTS

A Point Light is used to create an instance of a point luminous feature. The size, feature type, and number of sides of the point light model may be customized.

INSTANCE SETS

An Instance Set is a culling node which defines a set of point features. Each feature is positioned individually. This type of culling node can be used to simulate particles.



POLYGONS

A polygon is a fully rendered, textured, shaded, anti-aliased, fogged, illuminated, translucent geometric object. The polygon node contains the data necessary to define a textured and shaded polygon. This data includes its vertex list, translucency, texture, shading, and color.

LAYERED POLYGONS

A layered polygon is a polygon that lies in the plane of another polygon. The visual priority of layered polygons is based on the hierarchy of the polygons. A layered polygon has a lower priority than its children, and the same priority as its siblings.

TRANSLUCENCY

Polygon translucency is specified at the polygon node level. Also, a culling node may compute translucency for level-of-detail transitions. In this case, the translucency is inherited for all child nodes. The final polygon translucency is the product of the inherited translucency from the culling nodes, the inherent polygon translucency, and the translucency due to texture.

HIGH PRIORITY POLYGONS

A high priority polygon can be specified and will have visual priority over all other polygons in the viewport.

SHADING AND LIGHTING

There are several modes of shading. A shading intensity is computed at each vertex that has a vertex normal defined. If a vertex normal is not provided, a vertex may define its own fixed shading value, or have it computed by using the polygon normal as the vertex normal (flat shading). The shading intensities at each vertex are interpolated across the polygon to compute a shading value at each pixel that lies on the polygon. Sun shading uses the sun direction, sun intensity, ambient light, and vertex normals to compute shading. The vertex normals are user defined vectors representing the gradient of the surface whose curvature will be used to create a seamless, smoothly shaded object.



TEXTURE

Each polygon may be textured with color or monochrome texture. The texture map may modulate the color, intensity, and/or translucency of a polygon. An x and y normalized texture address at each vertex of the polygon maps a texture pattern on the polygon. Texture maps may be defined by the user to have any size, but will be clamped to a maximum size of 1024 x 1024.

COLOR

A polygon may reference a color table, or it may store the actual RGB color value.

LEVEL-OF-DETAIL (LOD) TABLE

The LOD table regulates the level-of-detail of a model displayed in the viewport to which the table is attached. The LOD table may be used to load manage the viewport. The LOD table can have up to 127 entries for different feature types. Each culling node may be assigned to one of the 127 feature types via an LOD index (1 to 127). The LOD controls are based upon either the subtended angle of the model from the viewpoint or the range from the viewpoint. As a model moves away from the viewpoint, its subtended angle becomes smaller, so fewer polygons are needed to represent the model. Therefore, a lower LOD version of the model can be selected, leaving more polygons available for more important features. Complementary translucency blending is provided for model level-of-detail transitions.

COLOR TABLE

A color table can store up to 4096 colors, each of which is a 24-bit RGB value. A color table may be referenced by a culling node. A color table is not required if all polygons store actual RGB values instead of an offset into a color table.



FUNCTIONAL GROUPS

Configuration Functions

The Configuration functions provide control over the PRO-1000 hardware configuration. They allow customization of the hardware memory allocation and allow deactivation of attached PRO-1000 units.

Function Name	Description
PRO_ConfigureMemory	Sets the memory configuration.
PRO_AbbreviatedConfigureMemory	Sets the memory configuration.
PRO_SetLogicalUnitOn	Allows display list data to be processed on this device.
PRO_SetLogicalUnitOff	Inhibits display list data from being processed on this device.

Hardware Interface and Global Functions

The Hardware Interface and Global functions are not associated with any specific display list objects. These functions are used to communicate with the PRO-1000 hardware, to get status back from the hardware, and to configure global attributes ranging from point light features (size and feature type) to texture memory page size to background color.

Function Name	Description
PRO_Init	Initializes communication with the PRO-1000 hardware.
PRO_InitOffline	Initializes display list when running in offline mode.
PRO_Stop	Shuts down communication with the PRO-1000 hardware.
PRO_DisplayDatabase	Processes the display list.
PRO_LoadModelDynamically	Loads a preformatted model dynamically.
PRO_UnloadDynamicModel	Unloads a dynamically loaded model.
PRO_SetGenlock	Sets the GENLOCK between multiple PRO-1000 devices.



Hardware Interface and Global Functions – Continued

Function Name	Description
PRO_SetUpdateRate	Sets the realtime update rate.
PRO_SetOverloadMode	Obsolete. Use equivalent Device Function.
PRO_SetDisplaySelect	Obsolete. Use equivalent Device Function.
PRO_SetForegroundColor	Obsolete. Use equivalent Device Function.
PRO_SetBackgroundColor	Obsolete. Use equivalent Device Function.
PRO_SetGamma	Obsolete. Use equivalent Device Function.
PRO_SetPointLightDetail	Sets the number of edges of the lights.
PRO_SetPointLightFeatureType	Sets the point light size clamp feature type.
PRO_SetPointLightSize	Sets the physical culling size of the lights.
PRO_SetPointLightPolygon	Sets the polygon to be used as point light model.
PRO_SetMicrotextureMapCount	Sets the number of microtexture maps.
PRO_SetTexturePageSize	Obsolete.
PRO_ReloadLogo	Reloads the spinning Real3D logo.
PRO_GetNumberOfLogicalUnits	Returns the number of attached PRO-1000 units.
PRO_GetNumberOfAvailableTexels	Returns the amount of texture memory.
PRO_GetMicrotextureMapCount	Returns the number of microtexture maps.
PRO_GetFirmwareRevision	Obsolete. Use equivalent Device Function.
PRO_GetModelNumber	Obsolete. Use equivalent Device Function.
PRO_GetRealtimeClockCount	Obsolete. Use equivalent Device Function.
PRO_GetProcessingTime	Obsolete. Use equivalent Device Function.
PRO_GetFrameRateEstimate	Obsolete. Use equivalent Device Function.
PRO_GetPolygonMemorySize	Gets the size of Polygon memory in bytes.
PRO_GetDefaultDevice	Gets a pointer to the PRO_Device object representing the default device (logical unit -1).



Device Functions

The Device class functions are used to assign graphical objects to a specified PRO-1000 device.

Function Name	Description
PRO_Device constructor	Creates an instance of a PRO_Device.
PRO_Device destructor	Destroys an instance of a PRO_Device.
GetStatus	Gets the status of a PRO_Device object instantiation.
AttachViewport	Attaches a viewport to a PRO_Device.
DetachViewport	Detaches a viewport from a PRO_Device.
SetGamma	Sets the gamma correction for the device based on a factor.
SetGammaRGB	Sets the gamma correction for the device using tables of component color values.
ActivateGamma	Turns on hardware gamma table processing.
DeactivateGamma	Turns off hardware gamma table processing.
SaveGammaToROM	Saves table in gamma memory to ROM.
WriteBlendMemory	Writes data to blend memory.
ActivateBlendMemory	Turns on soft edge blending.
DeactivateBlendMemory	Turns off soft edge blending.
SaveBlendMemoryToROM	Saves table in blend memory to ROM.
SetForegroundColor	Sets the foreground color for the device using 8-bit color values.
SetForegroundColor	Sets the foreground color for the device using normalized color values.
SetBackgroundColor	Sets the background color for the device using 8-bit color values.
SetBackgroundColor	Sets the background color for the device using normalized color values.



Device Functions – Continued

Function Name	Description
SetDisplaySelect	Sets the display sync select based on number of displays and desired line rate.
SetOverloadMode	Sets the realtime processing overload behavior.
GetLogicalUnit	Gets the logical unit associated with the PRO_Device.
IsValidDisplaySelect	Determines whether specified combination of display type and line rate is supported for the PRO_Device
GetSyncSelectInfo	Returns display type and line rate corresponding to each hardware sync file.
HasAttachedViewports	Reports whether device has attached viewports.
GetGamma	Returns the gamma value of the device.
GetForegroundColor	Gets the foreground color of the device.
GetBackgroundColor	Gets the background color of the device.
GetDisplaySelect	Returns the device's display mode.
GetOverloadMode	Returns the current overload mode.
GetLineRate	Returns the line rate set for the device.
IsHighRes	Tests for a high resolution system.
HasExpandedTexture	Tests for expanded texture memory.
GetFirmwareRevision	Returns the firmware version of the PRO device.
GetModelNumber	Returns the model number of the PRO device.
GetRealtimeClockCount	Returns the device's image display processing time.
GetProcessingTime	Returns the device's image display processing time in seconds.
GetFrameRateEstimate	Returns the device's estimated frame rate.
GetCurrentFrameCount	Returns the device's image display processing time in HW clocks.
GetHWPolygonMemorySize	Returns the device's polygon memory size.



Viewport Functions

The Viewport class functions are used to configure and control a viewport, a rectangular area on the 2D display for viewing the 3D data.

Function Name	Description
PRO_Viewport constructor	Creates an instance of a PRO_Viewport object.
PRO_Viewport destructor	Destroys an instance of a PRO_Viewport object.
GetStatus	Gets the status of a PRO_Viewport object instantiation.
AttachDefaultBlendTable	Associates the default blend table with a viewport.
AttachDefaultDatabase	Associates the default database with a viewport.
AttachBlendTable	Associates a blend table with a viewport.
AttachDatabase	Associates a database with a viewport.
AttachViewpoint	Associates a viewpoint with a viewport.
DetachViewpoint	Detaches a viewpoint.
AttachChildViewport	Attaches a viewport to a higher priority viewport.
DetachChildViewport	Detaches a viewport from its parent viewport.
SetFieldOfView	Defines the field-of-view.
SetFieldOfView	Defines an asymmetric field-of-view.
SetDisplayExtents	Sets the extents of the viewport in the physical display.
SetSunIlluminationData	Sets the parameters which control sun illumination effects.
SetSunVector	Sets the position of the sun.
SetFogParameters	Sets the parameters that control fog effects.
SetFogRanges	Sets the near and far ranges for fog effects.
SetFogColor	Sets the color of the fog.
SetLightFogColor	Sets the color of the light lobe on fog.
SetLightSourceColor	Sets the color of the light lobe on polygons.



Viewport Functions – Continued

Function Name	Description
SetLightSourceParameters	Sets the parameters that control light source illumination effects.
SetLightSourceStartRange	Sets the range to the light source.
SetLightSourceExtent	Sets distance which the light source illuminates.
SetLobeParameters	Sets the parameters that define the light lobe.
SetViewportOff	Deactivates a viewport.
SetViewportOn	Activates a viewport.
SetDisplaySelect	Selects the display.
SetInvertFlags	Sets the mirror flags as specified.
GetViewpoint	Gets the associated viewpoint.
GetAttachedDevice	Gets the associated device.
GetFieldOfView	Gets the symmetric field of view angles.
GetFieldOfView	Gets the asymmetric field of view half-angles.
GetDisplayExtents	Gets the extents of the viewport in the physical display.
GetSunIlluminationData	Gets the parameters which control sun illumination effects.
GetSunVector	Gets the position of the sun.
GetFogParameters	Gets the parameters that control fog effects.
GetFogRanges	Gets the near and far ranges used for fog effects,
GetFogColor	Gets the color of the fog.
GetLightFogColor	Gets the color of the light lobe on fog.
GetLightSourceColor	Gets the color of the light lobe on polygons.
GetLightSourceParameters	Gets the parameters that control light source illumination effects
GetLobeParameters	Gets the parameters that define the light lobe.
GetPriority	Returns the viewport's priority level.



Database Functions

The Database class functions are used to manage graphical objects. A Database is a collection of model root nodes that can be culled and (potentially) rendered.

Function Name	Description
PRO_Database constructor	Creates an instance of a PRO_Database object.
PRO_Database destructor	Destroys an instance of a PRO_Database object.
GetStatus	Gets the status of a PRO_Database object instantiation.
AttachModel	Attaches a model to a PRO_Database.
DetachModel	Detaches a model from a PRO_Database.
ActivateAllModels	Activates all models attached to the scene.
DeactivateAllModels	Deactivates all models attached to the scene.
HasActiveModels	Indicates whether any attached models are active.



Viewpoint Functions

The Viewpoint class functions handle the positioning and orientation of a viewpoint attached to a Viewport object.

Function Name	Description
PRO_Viewpoint constructor	Creates an instance of a PRO_Viewpoint object.
PRO_Viewpoint destructor	Destroys an instance of a PRO_Viewpoint object.
GetStatus	Gets the status of a PRO_Viewpoint object instantiation.
SetPosition	Sets the position of the viewpoint. Uses double-precision values.
SetPosition	Sets the position of the viewpoint. Uses single-precision values.
SetOrientation	Sets the orientation of the viewpoint. Uses double-precision values.
SetOrientation	Sets the orientation of the viewpoint. Uses single-precision values.
SetOrientation	Sets the orientation of the viewpoint. Uses a matrix of single-precision values.
GetPosition	Gets the current position of a viewpoint.
GetPosition	Gets the current position of a viewpoint. Returns values via reference variables.
GetOrientation	Gets the current orientation of a viewpoint.
GetOrientation	Gets the current orientation of a viewpoint. Returns values via reference variables.
GetLocalForward	Gets the forward vector in viewpoint coordinates.
GetLocalLeft	Gets the left vector in viewpoint coordinates.
GetLocalUp	Gets the up vector in viewpoint coordinates.
AttachViewpoint	Attaches a viewpoint to a viewpoint.
DetachViewpointFromParent	Detaches a viewpoint from its parent.



Blend Table Functions

A Blend Table, or LOD Table, contains parameters that specify how groups of objects are blended in and out of the rendered scene at various LODs.

Function Name	Description
PRO_LOD_Table constructor	Creates an instance of a PRO_LOD_Table.
PRO_LOD_Table destructor	Destroys an instance of a PRO_LOD_Table.
GetStatus	Gets the status of a PRO_LOD_Table object instantiation.
SetRangeBlendParameters	Sets blend parameters based on range.
SetSizeBlendParameters	Sets blend parameters based on subtended angle.
GetRangeBlendParameters	Gets blend parameters based on range.
GetSizeBlendParameters	Gets blend parameters based on subtended angle.
GetAllocatedTableSize	Returns the allocated number of entries.



Color Table Functions

The Color Table class functions are used to establish a table of Red–Green–Blue (RGB) color values that can be referenced by polygon objects.

Function Name	Description
PRO_Color_Table constructor	Creates an instance of a PRO_Color_Table.
PRO_Color_Table destructor	Destroys an instance of a PRO_Color_Table.
GetStatus	Gets the status of a PRO_Color_Table object instantiation.
SetColor	Sets the RGB value of a table entry. Uses integer values.
SetColor	Sets the RGB value of a table entry. Uses single–precision values.
GetAllocatedTableSize	Returns the allocated number of entries.



Model Functions

The Model class functions allow manipulation of a database object. The object may be defined by a pre-formatted model file that contains all of the object's geometry and attributes, or the object may be built up at runtime by adding culling volumes and polygons to it.

Function Name	Description
PRO_Model constructor	Creates an instance of a PRO_Model using pre-formatted input data.
PRO_Model constructor	Creates an instance of a PRO_Model that will be formatted "at runtime".
PRO_Model constructor	Creates an instance of a PRO_Model referencing shared geometry.
PRO_Model constructor	Creates an instance of a PRO_Model that is a copy of another PRO_Model.
PRO_Model destructor	Destroys an instance of a PRO_Model.
GetModelStatus	Returns the status of a model.
SetPosition	Sets the model's position. Takes an index that specifies the main model or any of its articulated parts. Takes double-precision values.
SetPosition	Sets the model's position. Takes an index that specifies the main model or any of its articulated parts. Takes single-precision values.
SetPosition	Sets an articulated part's position by specifying its name. Takes double-precision values.
SetPosition	Sets an articulated part's position by specifying its name. Takes single-precision values.
SetOrientation	Sets the model's orientation. Takes an index that specifies the main model or any of its articulated parts. Takes double-precision values.
SetOrientation	Sets the model's orientation. Takes an index that specifies the main model or any of its articulated parts. Takes single-precision values.
SetOrientation	Sets an articulated part's orientation by specifying its name. Takes double-precision values.



Model Functions – Continued

Function Name	Description
SetOrientation	Sets an articulated part's orientation by specifying its name. Takes single-precision values.
SetOrientation	Sets the model's orientation. Takes an index that specifies the main model or any of its articulated parts. Takes a matrix of single-precision values.
SetOrientation	Sets an articulated part's orientation by specifying its name. Takes a matrix of single-precision values.
Activate	Activates an inactive model.
Deactivate	Deactivates an active model.
AttachColorTable	Associates a color table with a model.
SetRangeLOD	Indicates that the model's blend data will be interpreted as ranges.
SetSizeLOD	Indicates that the model's blend data will be interpreted as sizes.
UserMatrix	Specifies the matrix to be used by the model.
IsStatic	Returns type of model.
IsActive	Returns current state of model.
GetPosition	Gets the model's position. Takes an index that specifies the main model or any of its articulated parts. Returns double-precision values.
GetPosition	Gets the model's position. Takes an index that specifies the main model or any of its articulated parts. Returns single-precision values.
GetPosition	Gets an articulated part's position by specifying its name. Returns double-precision values.
GetPosition	Gets an articulated part's position by specifying its name. Returns single-precision values.
GetOrientation	Gets the model's orientation. Takes an index that specifies the main model or any of its articulated parts. Returns double-precision values.



Model Functions – Continued

Function Name	Description
GetOrientation	Gets the model's orientation. Takes an index that specifies the main model or any of its articulated parts. Returns single-precision values.
GetOrientation	Gets an articulated part's orientation by specifying its name. Returns double-precision values.
GetOrientation	Gets an articulated part's orientation by specifying its name. Returns single-precision values.
GetLocalForward	Gets a model's forward vector. Takes an index that specifies the main model or any of its articulated parts. Passes back data using reference variables.
GetLocalForward	Gets a model's forward vector. Takes an index that specifies the main model or any of its articulated parts.
GetLocalForward	Gets an articulated part's forward vector by specifying the name of the part. Passes back data using reference variables.
GetLocalForward	Gets an articulated part's forward vector by specifying the name of the part.
GetLocalLeft	Gets a model's left vector. Takes an index that specifies the main model or any of its articulated parts. Passes back data using reference variables.
GetLocalLeft	Gets a model's left vector. Takes an index that specifies the main model or any of its articulated parts.
GetLocalLeft	Gets an articulated part's left vector by specifying the name of the part. Passes back data using reference variables.
GetLocalLeft	Gets an articulated part's left vector by specifying the name of the part.
GetLocalUp	Gets a model's up vector. Takes an index that specifies the main model or any of its articulated parts. Passes back data using reference variables.
GetLocalUp	Gets a model's up vector. Takes an index that specifies the main model or any of its articulated parts.



Model Functions – Continued

Function Name	Description
GetLocalUp	Gets an articulated part's up vector by specifying the name of the part. Passes back data using reference variables.
GetLocalUp	Gets an articulated part's up vector by specifying the name of the part.
GetNumberOfAnimationSequences	Gets the total number of animations associated with the model.
GetAnimationSequence	Gets the PRO_Animation object by index.
GetAnimationSequence	Gets the PRO_Animation object specified by name.
GetAnimationSequenceName	Gets the name of specified animation sequence.
GetNumberOfArticulatedParts	Gets the number of sub-parts in the model.
GetArticulatedPartMatrixIndex	Gets the index of a sub-part specified by name
GetArticulatedPartName	Gets the name of specified articulated part.
GetNumberOfTextureMaps	Gets the number of texture maps attached to polygons of this model.
GetTextureMapName	Gets the name of specified texture map.
ProcessImmediateModeData	Flushes immediate mode data to the hardware.
ProcessMorphedPolygons	Process geometry and texture deltas for immediate mode polygons.
AddCullingNode	Adds a culling node to the hierarchy of the model.
AddChild	Adds a culling node to the hierarchy of the model.
AddPointLight	Adds a point light culling node to the hierarchy of the model.
AddInstanceSet	Adds an instance set culling node to the hierarchy of the model.
AddPolygon	Adds a polygon to the hierarchy of the model.
AddChild	Adds a polygon to the hierarchy of the model.
EnableVertexSharing	Enables vertex sharing among polygons of this model.
DisableVertexSharing	Disables vertex sharing among polygons of this model.



Model Functions – Continued

Function Name	Description
SetAutoDeleteCullingData	Indicates that PRO_Culling_Node objects in hierarchy should be automatically deleted along with parent culling nodes.
SetAutoDeletePolygonData	Indicates that PRO_Polygon objects in hierarchy should be automatically deleted along with parent culling nodes.
SetAutoDestructCullingData	Indicates that PRO_Culling_Node objects in hierarchy should be automatically deleted when formatting is completed.
SetAutoDestructPolygonData	Indicates that PRO_Polygon objects in hierarchy should be automatically deleted when formatting is completed.
StoreArticulatedPart	Associates a 6DOF node with a model.
StoreAnimationSequence	Associates an animation node with a model.
StoreTextureData	Associates a PRO_Texture object with a model.
StoreTextureData	Associates a texture map with a model.
Save	Saves a run-time formatted model to file.



Animation Functions

The Animation class functions specify the characteristics and control the functionality of animation sequences associated with a model.

Function Name	Description
AnimationStartsVisible	Sets an animation to be visible at activation.
AnimationEndsVisible	Sets the animation to remain visible when deactivated.
AnimationStartsInvisible	Sets an animation to be invisible at activation.
AnimationEndsInvisible	Sets the animation to be invisible when deactivated.
ActivateSequence	Activates an animation sequence.
DeactivateSequence	Deactivates the animation sequence.
AnimationRunsForward	Specifies a forward–running animation.
AnimationRunsBackwards	Specifies a backward–running animation.
AnimationRunsMirrored	Specifies an animation that runs forward, then backward.
AnimationIsUserControlled	Specifies an animation that runs according to user controls.
AnimationRunsForever	Specifies an animation that runs continuously.
SetAnimationCycleTime	Sets the time for one cycle through the animation. Takes a double–precision value.
SetAnimationCycleTime	Sets the time for one cycle through the animation. Takes a single–precision value.
SetAnimationRepeatCount	Sets the number of times to run through the animation.
SetFrameActive	Activates the specified animation frame.
SetFrameInactive	Deactivates the specified animation frame.
GetAnimationCycleTime	Gets the time for one cycle through the animation. Passes back single–precision value.
GetAnimationCycleTime	Gets the time for one cycle through the animation. Passes back double–precision value.
GetNumberOfAnimationFrames	Gets the number of animation frames.
GetCurrentFrame	Gets the current animation frame.
GetAnimationState	Indicates whether an animation sequence is active or inactive.



Culling Node Functions

The Culling Node class functions are used to build a database hierarchy that will be used by the hardware to determine which graphical objects should be rendered.

Function Name	Description
PRO_Culling_Node constructor	Creates an instance of a PRO_Culling_Node object.
PRO_Culling_Node destructor	Destroys an instance of a PRO_Culling_Node object.
GetStatus	Returns status of PRO_Culling_Node instantiation.
ActivateNode	Indicates that culling node should be processed by PRO-1000 hardware.
DeactivateNode	Indicates that culling node should be ignored by PRO-1000 hardware.
AddChild	Adds a child culling node to a culling node.
AddChild	Adds a child polygon to a culling node.
AddPointLight	Adds a point light to a culling node.
AddInstanceSet	Adds a child instance set to a culling node.
DetachChild	Detaches a child culling node from its parent.
DetachChild	Detaches a child polygon from its parent.
AttachColorTable	Associates a color table with a culling node.
SetRangeLOD	Indicates that the node's blend data will be interpreted as ranges.
SetSizeLOD	Indicates that the node's blend data will be interpreted as sizes.
SetFeatureType	Sets the feature type for blending.
SetGroupLayerNumber	Sets a group layer number.
SetVolumeSizeClamp	Sets an object to clamp at the last LOD transition size.



Culling Node Functions – Continued

Function Name	Description
SetAutoDeleteCullingData	Indicates that child culling nodes should be destroyed with their parent.
SetAutoDeletePolygonData	Indicates that child polygons should be destroyed with their parent.
SetAutoDestructCullingData	Indicates that child culling nodes should be destroyed with their parent.
SetAutoDestructPolygonData	Indicates that child polygons should be destroyed with their parent.
FormatAndFlushPolygons	Formats polygons and downloads to hardware immediately.
GetFeatureType	Gets the feature type used for blending.
GetBoundingBox	Retrieves the current bounding box of a node.
GetLODNumber	Gets the LOD number of the culling node.
GetGroupLayerNumber	Gets the group layer number.
GetCentroid	Gets the centroid of the culling volume.
HasAssociatedMatrix	Returns whether the culling node supports articulation.



LOD Culling Node Functions

The LOD Culling Node class functions are used to define culling nodes which have indirect children representing levels of detail in the culling hierarchy.

Function Name	Description
PRO_LOD_Culling_Node constructor	Creates an instance of a PRO_LOD_Culling_Node object.
PRO_LOD_Culling_Node destructor	Destroys an instance of a PRO_LOD_Culling_Node object.
AddChild	Adds a child culling node to the culling hierarchy at the specified LOD.
AddChild	Adds a child polygon to the culling hierarchy at the specified LOD.



Articulated Part Culling Node Functions

Articulated Part Culling Nodes are used to create geometry that can move relative to the coordinate set of the parent culling node to which they are attached. These nodes are known as dynamic coordinate sets or six-degree-of-freedom nodes.

Function Name	Description
PRO_6DOF_Culling_Node constructor	Creates a PRO_6DOF_Culling_Node object.
PRO_6DOF_Culling_Node constructor	Creates a PRO_6DOF_Culling_Node object with a defined user matrix.
PRO_6DOF_Culling_Node destructor	Destroys a PRO_6DOF_Culling_Node object.
SetName	Names the 6DOF Node
SetPosition	Sets the position of the 6DOF node. Uses double-precision values.
SetPosition	Sets the position of the 6DOF node. Uses single-precision values.
SetOrientation	Sets the orientation of the 6DOF node. Uses double-precision values.
SetOrientation	Sets the orientation of the 6DOF node. Uses single-precision values.
SetOrientation	Sets the orientation of the 6DOF node. Passes a matrix of floating point values.
SetMotionExtents	Sets the motion limits for the 6DOF node. Uses double-precision values.
SetMotionExtents	Sets the motion limits for the 6DOF node. Uses single-precision values.
SetMatrix	Overrides API matrix with a user matrix.
ReplaceMatrix	Overrides the current user matrix with another.
GetName	Retrieves the name of the 6DOF Node



Articulated Part Culling Node Functions – Continued

Function Name	Description
GetPosition	Gets the current position of the 6DOF node.
GetOrientation	Gets the current orientation of the 6DOF node.
GetMatrix	Retrieves the 6DOF node's matrix.



Animation Culling Node Functions

Animation Culling Nodes are used to manage sets of models representing frames of an animation or pieces of models that can be turned on and off as required by the host application. The `PRO_Animation_Culling_Node` class creates a `PRO_Animation_Sequence` object that can also be used to transfer control of the animation sequencing to the PRO/API.

Function Name	Description
PRO_Animation_Culling_Node constructor	Creates a <code>_PRO_Animation_Culling_Node</code> object.
PRO_Animation_Culling_Node destructor	Destroys a <code>_PRO_Animation_Culling_Node</code> object.
AddChild	Adds a child culling node to the culling hierarchy for the specified frame of animation.
AddChild	Adds a child polygon to the culling hierarchy for the specified frame of animation.
SetName	Names the animation sequence.
GetName	Retrieves the name of the animation node.
GetAnimationSequence	Retrieves the associated <code>PRO_Animation_Sequence</code> .



Instance Culling Node Functions

Instance Culling Nodes are used to define the top of a shared display list segment that can be referenced later on from other parts of the scene display list. They are normally useful in reducing the polygon storage requirements for replicated features, such as trees, by storing a single copy of the geometry, and simply referencing (and locating) it multiple times with little storage overhead.

Function Name	Description
PRO_Instance_Culling_Node constructor	Creates an instance of a PRO_Instance_Culling_Node object.
ProcessImmediateModeData	Flushes immediate mode polygons to the hardware.
EnableVertexSharing	Enables vertex sharing when processing the shared geometry.
DisableVertexSharing	Disables vertex sharing when processing the shared geometry.



Instance Reference Culling Node Functions

Instance Reference Culling Nodes are used to place the geometry linked to PRO_Instance_Culling_Node objects in the display list. The shared geometry is processed as if it were located at a given position with little storage overhead. Instance references may not have other nodes attached as children.

Function Name	Description
PRO_Instance_Reference constructor	Creates an instance of a PRO_Instance_Reference node.
SetInstanceLocation	Sets the position of the instance reference.
GetInstanceLocation	Gets the position of the instance reference.



Point Light Functions

The Point Light class is used to create instances of point luminous features. Point lights are handled as polygons that always face the viewpoint and are not affected by lighting, so they seem luminous during night scenes.

Function Name	Description
PRO_Point_Light constructor	Creates an instance of a point light.
PRO_Point_Light destructor	Destroys an instance of a point light.
SetColor	Defines the color of the light. Takes normalized color values (0.0 to 1.0).
SetColor	Defines the color of the light. Takes 8-bit values (0 to 255).
SetPosition	Sets the position of the point light.
GetPosition	Gets the position of the point light.



Instance Set Functions

The Instance Set class is used to create a group of instances of point features, to achieve effects such as dust particles.

Function Name	Description
PRO_Instance_Set constructor	Creates an instance of an instance set.
PRO_Instance_Set destructor	Destroys an instance of an instance set.
SetColor	Defines the color of the instances in the set. Takes normalized floating point values (0.0 to 1.0).
SetColor	Defines the color of the instances in the set. Takes 8-bit values (0 to 255).
SetPosition	Sets the position of an instance in the set.
GetPosition	Returns the position of an instance in the set.
SetActiveInstanceCount	Sets the number of instances in the set which are active.
GetActiveInstanceCount	Gets the number of active instances in the set.
GetNumberOfInstances	Returns the total number of instances in the set.



Polygon Functions

The Polygon class functions are used to define the geometry of and assign attributes to polygons.

Function Name	Description
PRO_Polygon constructor	Creates an instance of a PRO_Polygon object.
PRO_Polygon destructor	Destroys an instance of a PRO_Polygon object.
AddVertex	Adds a vertex to the definition of a polygon.
AddChild	Adds a child polygon to a polygon.
DetachChild	Detaches a child polygon from its parent polygon.
AttachTexture	Associates the polygon with a texture map.
DetachTexture	Disassociates the polygon from its texture map.
AttachMicroTexture	Associates the polygon with a microtexture map.
DetachMicroTexture	Disassociates the polygon from its microtexture map.
SetColor	Sets the polygon color. Takes 8-bit values (0 to 255).
SetColor	Sets the polygon color. Takes normalized values (0.0 to 1.0).
SetColorIndex	Sets the color table index for out-the-window colors.
SetSensorColorIndex	Sets the color table index for sensor colors.
EnableLightingEffects	Indicates that polygon is not luminous and its color will be affected by shading, ambient light, and sun intensity.
DisableLightingEffects	Indicates that polygon is luminous and its color will not be affected by shading, ambient light, and sun intensity.
SetTranslucency	Sets the translucency value for a polygon.
SetLightModifier	Sets the amount of effect fog will have on a luminous polygon.
SetHighPriority	Indicates that polygon will have priority over other polygons in the viewport.



Polygon Functions – Continued

Function Name	Description
PolygonIsLayered	Indicates a stencil polygon.
IsDoubleSided	Sets the double-sided attribute of a polygon.
DoSmoothShading	Indicates that the polygon will be smooth shaded.
DoFlatShading	Indicates that the polygon will be flat shaded.
IsNotShaded	Indicates that the polygon will not be shaded.
VerticesAreClockwise	Sets the direction of vertex ordering to clockwise.
VerticesAreCounterClockwise	Sets the direction of vertex ordering to counter-clockwise.
SetXSmoothing	Enables texture wrap blending in the x (u) direction.
SetYSmoothing	Enables texture wrap blending in the y (v) direction.
SetXMirror	Enables mirroring in the x (u) direction.
SetYMirror	Enables mirroring in the y (v) direction.
SetNPScale	Biases the texture sampling rate.
SetTranslucencyPatternSelect	Sets the blend/translucency pixel hole pattern select.
EnableAutoVertexDelete	Indicates that vertices associated with the polygon will be deleted when polygon is deleted.
DisableAutoVertexDelete	Indicates that vertices associated with the polygon will not be deleted when polygon is deleted.
GetLODNumber	Gets the face LOD for a polygon.
GetVertexCount	Gets the number of vertices associated with a polygon.
GetCompressedVertexCount	Gets the number of vertices associated with a polygon after vertex sharing has been determined.
GetVertex	Returns a vertex.
IsTextured	Tests for valid texture attachment.



Vertex Functions

The Vertex class functions are used to define and manipulate a vertex.

Function Name	Description
PRO_Vertex constructor	Creates an instance of a PRO_Vertex object with the specified x,y,z coordinates.
PRO_Vertex constructor	Creates an instance of a PRO_Vertex object with no specified coordinates.
PRO_Vertex destructor	Destroys an instance of a PRO_Vertex object.
SetCoordinates	Sets the coordinates of a vertex. Values are single-precision.
SetCoordinates	Sets the coordinates of a vertex. Values are double-precision.
SetMorphCoordinates	Sets the coordinates of a vertex without rounding.
SetTextureCoordinates	Sets the texture coordinates of a vertex.
SetVertexNormal	Sets the vector defining the vertex normal.
SetFixedShadingIntensity	Sets a fixed lighting value for the vertex.
OffsetVertex	Defines an offset vector for the vertex coordinates.
ScaleVertex	Defines a scale vector for the vertex coordinates.
TransformVertex	Defines a transformation matrix for the vertex coordinates.
GetCoordinates	Returns the vertex coordinates as single-precision values.
GetCoordinates	Returns the vertex coordinates as double-precision values.
GetTextureCoordinates	Returns the vertex texture coordinates as single-precision values.
GetVertexNormal	Returns the vertex normal.
IsTextured	Returns TRUE if the vertex is textured.
IsShaded	Returns TRUE if the vertex is shaded.



Texture Functions

The Texture class functions are used to define an image that will be used as a texture map for polygons.

Function Name	Description
PRO_Texture constructor	Creates an instance of a PRO_Texture object for a color image.
PRO_Texture constructor	Creates an instance of a PRO_Texture object for a monochrome image.
PRO_Texture destructor	Destroys an instance of a PRO_Texture object.
IsContourTexture	Sets the contour texture attribute of a PRO_Texture object.
SetContourThreshold	Defines the threshold between opacity and translucency. Takes an 8-bit value (0 to 255).
SetContourThreshold	Defines the threshold between opacity and translucency. Takes a normalized value (0.0 to 1.0).
EnableAlpha	Enables processing of alpha channel data.
DisableAlpha	Disable processing of alpha channel data.
SetTextureFileName	Sets name of source texture file.
KeepDataAfterLoading	Indicates that map data should be retained after loading.
FreeDataAfterLoading	Indicates that map data should be released after loading. For use if data was allocated using “alloc” functions.
DeleteDataAfterLoading	Indicates that map data should be released after loading. For use if data was allocated using “new”.
GetTextureFileName	Returns name of source texture file.
GetOrigTextureMapWidth	Returns image width before scaling or rotating.
GetOrigTextureMapHeight	Returns image height before scaling or rotating.
IsTextureRGB	Indicates whether map is a color image.
IsMicrotexture	Indicates whether map is a microtexture map.



Microtexture Functions

The Microtexture class functions are used to define an image that will be used as a microtexture map for polygons. The Microtexture class inherits from the PRO_Texture class. All PRO_Texture functions still apply.

Function Name	Description
PRO_MicroTexture constructor	Creates an instance of a PRO_MicroTexture object for a color image.
PRO_MicroTexture constructor	Creates an instance of a PRO_MicroTexture object for a monochrome image.
PRO_MicroTexture destructor	Destroys an instance of a PRO_MicroTexture object.



Matrix Functions

The Matrix class is an abstraction of a direction cosine matrix, which stores a centroid and a 3x3 transformation matrix.

Function Name	Description
PRO_Matrix constructor	Creates an instance of a PRO_Matrix object.
PRO_Matrix destructor	Destroys an instance of a PRO_Matrix object.
GetStatus	Gets the status of a PRO_Matrix object instantiation.
SetPosition	Sets the position stored in the matrix. Uses double-precision values.
SetPosition	Sets the position stored in the matrix. Uses single-precision values.
SetPosition	Sets the position stored in the matrix. Uses a vector of single-precision values.
SetOrientation	Sets the orientation stored in the matrix. Uses double-precision values.
SetOrientation	Sets the orientation stored in the matrix. Uses single-precision values.
SetOrientation	Sets the orientation stored in the matrix. Uses a matrix of single-precision values.
SetLocalForward	Sets the forward vector.
SetLocalLeft	Sets the left vector.
SetLocalUp	Sets the up vector.
GetPosition	Gets the position stored in a matrix. Returns double-precision values.
GetPosition	Gets the position stored in a matrix. Returns single-precision values.
GetPosition	Gets the position stored in a matrix. Returns values via reference variables.
GetOrientation	Gets the orientation stored in a matrix.



Matrix Functions – Continued

Function Name	Description
GetOrientation	Gets the orientation stored in a matrix. Returns values via reference variables.
GetLocalForward	Gets the forward vector.
GetLocalLeft	Gets the left vector.
GetLocalUp	Gets the up vector.
AttachMatrix	Attach a matrix to another matrix.
DetachFromParent	Detach a matrix from its parent matrix.



MultiGen Loader Functions

The `PRO_MGen_Model` class is used to load in MultiGen models (.flt files). When the .flt file name is passed to the constructor, the file is loaded and a model object is created. Articulated parts, animations, LODs, and textures are all supported. Also supported are the Real3D comments and extended attributes that are specific to the PRO-1000 product. This class is inherited from the `PRO_Model` class.

Function Name	Description
PRO_MGen_Model constructor	Creates an instance of a <code>PRO_MGen_Model</code> .
PRO_MGen_Model destructor	Destroys an instance of a <code>PRO_MGen_Model</code> .
GetPolyCount	Returns the number of polygons in the model.
GetVertCount	Returns the number of vertices in the model.



SAMPLE APPLICATIONS

A few naming conventions are used throughout the PRO/API library. All PRO/API header files are prefixed by “pro_”. All PRO/API classes have the “PRO_” prefix. General purpose functions (not methods of a class) are also prefixed with “PRO_”.

All of these applications have been compiled with a C++ compiler and tested on both a Pentium PC running Windows NT 3.51 and a Sun running Solaris 2.4. However, these samples should not necessarily be used as templates for a real application. They are intended only to illustrate concepts and functions which will be used in most application development efforts.

Note that the pre-formatted models used in these examples were selected for illustration purposes only. The concepts presented in each sample should apply to all pre-formatted models with similar attributes.



Sample Applications

Sample 1

This application loads a pre-formatted model and rotates the model about its z axis.

The first step in any application using the PRO/API library functions is a call to `PRO_Init()`. This function initializes the hardware and the internal structures of the PRO/API. After this call, the application may create PRO/API objects and attach them together to build the desired display list, which is downloaded to the hardware via a call to `PRO_DisplayDatabase()`. The last step in any application is a call to `PRO_Stop()`. All dynamically allocated PRO/API objects must be destroyed before this call, which halts communication to the hardware and deletes the API internal structures.

In this application, a single pre-formatted model is loaded and attached to a database. The database is then attached to a viewport that has an associated viewpoint. Here, most viewport attributes are allowed to default. The viewport is attached to a device that specifies the PRO-1000 unit which will display the rendered data. In this example, `PRO_DisplayDatabase()` is called each time through a loop that modifies the orientation of the model.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pro_globals.hh>
#include <pro_api_data.hh>
#include <pro_db_entity.hh>
#include <pro_memmgr.hh>
#include <pro_device.hh>
#include <pro_viewport.hh>
#include <pro_viewpoint.hh>
#include <pro_database.hh>
#include <pro_tables.hh>
#include <pro_model.hh>
#include <pro_data_block.hh>

/*_____*/
```



Sample Applications

```
int main ( int argc , char *argv[] )
{
    PRO_Status    status ;
    PRO_Device    *dev0 = NULL;
    PRO_Viewport  *viewport = NULL;
    PRO_Viewpoint *viewpoint = NULL;
    PRO_Database  *scene = NULL;
    PRO_Model     *model1 = NULL;
    long          i ;
    double        yaw ;

    // Initialize the hardware. The default memory configuration is used.
    status = PRO_Init() ;

    // Instantiate a device corresponding to logical unit (SCSI ID) 0.
    dev0 = new PRO_Device(0) ;

    // Instantiate a model with the specified pre-formatted filename.
    model1 = new PRO_Model ( "texcube" ) ;
    if ( model1->GetModelStatus() != PRO_SUCCESS )
    {
        printf("Failed loading model 'texcube'.\n") ;
        delete model1 ;
        PRO_Stop() ;
        return -1 ;
    }

    // Set the model to interpret blend values as ranges.
    model1->SetRangeLOD() ;

    // Set the model's initial position and orientation.
    model1->SetPosition( 0.0, 0.0, 0.0 ) ;
    model1->SetOrientation( 0.0, 0.0, 30.0 ) ;

    // Instantiate a database and attach the model to the scene.
    scene = new PRO_Database () ;
    scene->AttachModel ( model1 ) ;
}
```



Sample Applications

```
// Instantiate a viewport and attach the scene to the viewport.
viewport = new PRO_Visport() ;
viewport->AttachDatabase ( scene ) ;

// Instantiate a viewpoint and attach the viewpoint to the viewport.
viewpoint = new PRO_Viewpoint() ;
viewport->AttachViewpoint ( viewpoint ) ;

// Attach the viewport to the device.
dev0->AttachVisport( viewport ) ;

// Set the viewpoint's initial position and orientation.
viewpoint->SetPosition ( -20.0, 0.0, 20.0 ) ;
viewpoint->SetOrientation ( 0.0 , 45.0 , 0.0 ) ;

// Spin the model around on its z axis for a while.
for(i=0; i<3600; i++)
{
    yaw = (double)(i % 360) ;
    model1->SetOrientation ( 0.0 , 0.0 , yaw ) ;

    // Send display list to hardware for rendering.
    status = PRO_DisplayDatabase() ;
}

// Delete all the dynamically allocated objects.
if ( viewpoint) delete viewpoint ;
if ( visport) delete visport ;
if ( scene) delete scene ;
if ( model1) delete model1 ;
if ( dev0) delete dev0 ;

// Halt communication with the hardware.
status = PRO_Stop() ;

return 0 ;
}
```



Sample Applications

Sample 2

This application loads a pre-formatted model that has articulated parts, and rotates the articulated part about its y axis.

As in Sample Application #1, this example begins with a call to `PRO_Init()` and ends with a call to `PRO_Stop()`. It also instantiates a single pre-formatted model and the same associations are made between model, database, viewport, viewpoint, and device. In this example, however, the model has an articulated part that is manipulated independent of the main model. An index is obtained for the articulated part in the model's hierarchy. That index is passed to functions that set the position and orientation of the sub-part. This sample program demonstrates by loading a model of a railroad gate, and moving its arm up and down.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pro_globals.hh>
#include <pro_api_data.hh>
#include <pro_db_entity.hh>
#include <pro_memmgr.hh>
#include <pro_device.hh>
#include <pro_viewport.hh>
#include <pro_viewpoint.hh>
#include <pro_database.hh>
#include <pro_tables.hh>
#include <pro_model.hh>
#include <pro_data_block.hh>

/*_____*/

int main ( int argc , char* argv[] )
{
    PRO_Status    status ;
    PRO_Device    *dev0 = NULL;
    PRO_Visport   *viewport = NULL;
    PRO_Viewpoint *viewpoint = NULL;
    PRO_Database  *scene = NULL;
```



Sample Applications

```
PRO_Model  *model1 = NULL;
long       i, up ;
long       arm_id = 0 ;
double     pitch ;

// Initialize the hardware. The default memory configuration is used.
status = PRO_Init() ;

// Instantiate a device corresponding to logical unit (SCSI ID) 0.
dev0 = new PRO_Device(0) ;

// Set the background color to yellow so you can see the black model.
dev0->SetBackgroundColor( (long)255, (long)255, (long)0 ) ;

// Instantiate a model with the specified pre-formatted filename.
model1 = new PRO_Model ( "rrgate" ) ;
if ( model1->GetModelStatus() != PRO_SUCCESS )
{
    printf("Failed loading model %s.\n", model_name ) ;
    delete model1 ;
    PRO_Stop() ;
    return -1 ;
}

// Set the model to interpret blend values as ranges.
model1->SetRangeLOD() ;

// Set the model's initial position and orientation.
model1->SetPosition( 0.0, 0.0, 0.0 ) ;
model1->SetOrientation( 0.0, 0.0, 90.0 ) ;

// Get an index corresponding to the named art part.
arm_id = model1->GetArticulatedPartMatrixIndex( "arm" ) ;

// Set the art part's initial position and orientation.
model1->SetPosition( 0.0 , 0.0 , 3.0 , arm_id ) ;
model1->SetOrientation( 0.0 , 0.0 , 0.0 , arm_id ) ;

// Instantiate a database and attach the model to the scene.
scene = new PRO_Database () ;
scene->AttachModel ( model1 ) ;
```



Sample Applications

```
// Instantiate a viewport and attach the scene to the viewport.
viewport = new PRO_Visport() ;
viewport->AttachDatabase ( scene ) ;

// Instantiate a viewpoint and attach the viewpoint to the viewport.
viewpoint = new PRO_Viewpoint() ;
viewport->AttachViewpoint ( viewpoint ) ;

// Attach the viewport to the device.
dev0->AttachVisport( viewport ) ;

// Set the viewpoint's initial position and orientation.
viewpoint->SetPosition ( -100.0, 0.0, 70.0 ) ;
viewpoint->SetOrientation ( 0.0 , 35.0 , 0.0 ) ;

// Move the art part up and down for a while.
up = 0 ;
for (i=0; i<3600; i++)
{
    pitch = (double)(i % 90) ;

    if ( ( i % 90 ) == 0 ) up = !up ;
    if ( up ) pitch = ( 90 - pitch ) ;
    model1->SetOrientation ( 0.0 , pitch, 0.0, arm_id ) ;

    // Send display list to hardware for rendering.
    status = PRO_DisplayDatabase() ;
}

// Delete all the dynamically allocated objects.
if ( viewpoint) delete viewpoint ;
if ( visport) delete visport ;
if ( scene)    delete scene ;
if ( model1)  delete model1 ;
if ( dev0)    delete dev0 ;

// Halt communication with the hardware.
status = PRO_Stop() ;

return 0 ;
}
```



Sample 3

This application loads four pre-formatted models. Each model has an animation sequence that is set up to run in a different way.

As in the previous examples, this example begins with a call to `PRO_Init()` and ends with a call to `PRO_Stop()`. This example loads the same pre-formatted model four times, placing each model at a different position. Each model contains one animation sequence. The first model's animation sequence is set to run forward through its frames continuously, with one cycle set to take 1 second. The second model's animation is set to run backward through its frames continuously, with one cycle set to take 2 seconds. The third model's sequence is set to run through its frames first forward then back again, continuously, with each cycle taking 3 seconds. The last model's animation sequence is user controlled. The sample application runs in a loop, each time setting the frame of the user-controlled animation to be displayed, and then calling `PRO_DisplayDatabase()` to write the display list for rendering.

```
#include <stdlib.h>
#include <stdio.h>
#include <pro_symbol_table.hh>
#include <pro_external_reference.hh>
#include <pro_globals.hh>
#include <pro_db_entity.hh>
#include <pro_memmgr.hh>
#include <pro_device.hh>
#include <pro_viewport.hh>
#include <pro_viewpoint.hh>
#include <pro_database.hh>
#include <pro_tables.hh>
#include <pro_model.hh>
#include <pro_data_block.hh>
#include <pro_animation.hh>

/*_____*/

int main ( int argc, char *argv[] )
{
    PRO_Status    status ;
```



Sample Applications

```
PRO_Device  *dev0 = NULL ;
PRO_Viewport *viewport = NULL ;
PRO_Viewport *viewpoint = NULL;
PRO_Database *scene = NULL;
PRO_Model   *model1 = NULL;
PRO_Model   *model2 = NULL;
PRO_Model   *model3 = NULL;
PRO_Model   *model4 = NULL;
PRO_Animation_Sequence *animation_sequence = NULL;
long        i ;

// Initialize the hardware. The default memory configuration is used.
status = PRO_Init() ;

// Instantiate a device corresponding to logical unit (SCSI ID) 0.
dev0 = new PRO_Device(0) ;

// Instantiate models with the specified pre-formatted filename.
model1 = new PRO_Model ( "case10" ) ;
model2 = new PRO_Model ( "case10" ) ;
model3 = new PRO_Model ( "case10" ) ;
model4 = new PRO_Model ( "case10" ) ;
if ( model1->GetModelStatus() != PRO_SUCCESS ||
    model2->GetModelStatus() != PRO_SUCCESS ||
    model3->GetModelStatus() != PRO_SUCCESS ||
    model4->GetModelStatus() != PRO_SUCCESS )
{
    printf("Failed loading model 'case10'.\n") ;
    if ( model1 ) delete model1 ;
    if ( model2 ) delete model2 ;
    if ( model3 ) delete model3 ;
    if ( model4 ) delete model4 ;
    PRO_Stop() ;
    return -1 ;
}

// Set the initial position and orientation of each model.
if ( model1 ) model1->SetPosition ( 0.0 , 100.0 , 100.0 ) ;
```



Sample Applications

```
if ( model2 ) model2->SetPosition ( 0.0 , -100.0 , 100.0 ) ;
if ( model3 ) model3->SetPosition ( 0.0 , -100.0 , -100.0 ) ;
if ( model4 ) model4->SetPosition ( 0.0 , 100.0 , -100.0 ) ;

// Instantiate a database and attach the model to the scene.
scene = new PRO_Database ( ) ;
if ( model1 ) scene->AttachModel ( model1 ) ;
if ( model2 ) scene->AttachModel ( model2 ) ;
if ( model3 ) scene->AttachModel ( model3 ) ;
if ( model4 ) scene->AttachModel ( model4 ) ;

// Instantiate a viewport and attach the scene to the viewport.
viewport = new PRO_Viewport ( ) ;
viewport->AttachDatabase ( scene ) ;

// Instantiate a viewpoint and attach the viewpoint to the viewport.
viewpoint = new PRO_Viewpoint ( ) ;
viewport->AttachViewpoint ( viewpoint ) ;

// Attach the viewport to the device.
dev0->AttachViewport ( viewport ) ;

// Set the viewpoint's initial position and orientation.
viewpoint->SetPosition ( -800.0 , 0.0 , 0.0 ) ;
viewpoint->SetOrientation ( 0.0 , 00.0 , 0.0 ) ;

if ( model1 )
{
    // Get the first model's animation sequence by name.
    animation_sequence = model1->GetAnimationSequence( "g1" ) ;

    // Set the animation's behavior and activate it.
    animation_sequence->AnimationRunsForever ( ) ;
    animation_sequence->AnimationRunsForward ( ) ;
    animation_sequence->SetAnimationCycleTime( 1.0 ) ;
    animation_sequence->ActivateSequence ( ) ;
}

if ( model2 )
{
```



Sample Applications

```
// Get the second model's animation sequence by name.
animation_sequence = model2->GetAnimationSequence("g1" );

// Set the animation's behavior and activate it.
animation_sequence->AnimationRunsForever() ;
animation_sequence->AnimationRunsBackwards() ;
animation_sequence->SetAnimationCycleTime(2.0) ;
animation_sequence->ActivateSequence() ;
}

if ( model3 )
{
// Get the third model's animation sequence by name.
animation_sequence = model3->GetAnimationSequence("g1" );

// Set the animation's behavior and activate it.
animation_sequence->AnimationRunsForever() ;
animation_sequence->AnimationRunsMirrored() ;
animation_sequence->SetAnimationCycleTime(3.0) ;
animation_sequence->ActivateSequence() ;
}

if ( model4 )
{
// Get the fourth model's animation sequence by name.
// This animation sequence is user-controlled (the default state).
animation_sequence = model4->GetAnimationSequence("g1" );

// Set the animation's behavior and activate it.
animation_sequence->AnimationStartsInvisible() ;
animation_sequence->AnimationEndsInvisible() ;
}

if ( model1 ) model1->SetOrientation ( 0.0, 0.0, 90.0 ) ;
if ( model2 ) model2->SetOrientation ( 0.0, 0.0, 90.0 ) ;
if ( model3 ) model3->SetOrientation ( 0.0, 0.0, 90.0 ) ;
if ( model4 ) model4->SetOrientation ( 0.0, 0.0, 90.0 ) ;

for ( i = 0 ; i < 1100 ; i++ )
{
```



Sample Applications

```
// Activate one frame of the user-controlled animation.
if ( model4 ) animation_sequence->SetFrameActive ( i % 11 ) ;

// Send display list to hardware for rendering.
status = PRO_DisplayDatabase() ;

// Deactivate the frame of the user-controlled animation.
if ( model4 ) animation_sequence->SetFrameInactive ( i % 11 ) ;
}

// Delete all the dynamically allocated objects.
if ( viewport ) delete viewport ;
if ( scene ) delete scene ;
if ( model1 ) delete model1 ;
if ( model2 ) delete model2 ;
if ( model3 ) delete model3 ;
if ( model4 ) delete model4 ;
if ( viewpoint ) delete viewpoint ;
if ( dev0 ) delete dev0 ;

// Halt communication with the hardware.
status = PRO_Stop() ;

return 0 ;
}
```



Sample Applications

Sample 4

This application builds an immediate mode model and rotates the model about its z axis while morphing its vertices in realtime.

This sample application is an immediate mode version of Sample Application #1. This example begins with a call to `PRO_Init()` and ends with a call to `PRO_Stop()`. It instantiates a single model and the same associations are made between model, database, viewport, viewpoint, and device. In this example, however, the model is not created from a file with geometry and attributes predefined. Instead, vertices are created and added to polygons, which in turn are assigned attributes and added to the model to define its geometry. Here, the polygons are two quadrilaterals with a shared edge. The application cycles at 60 Hz, rotating the model and modifying its vertices each time the model is rendered.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <pro_globals.hh>
#include <pro_api_data.hh>
#include <pro_db_entity.hh>
#include <pro_memmgr.hh>
#include <pro_device.hh>
#include <pro_viewport.hh>
#include <pro_viewpoint.hh>
#include <pro_database.hh>
#include <pro_tables.hh>
#include <pro_model.hh>
#include <pro_polygon.hh>
#include <pro_data_block.hh>

/*_____*/

int main ( int argc , char *argv[] )
{
    PRO_Status    status ;
    PRO_Device    *dev0 = NULL;
```



Sample Applications

```
PRO_Viewport  *viewport = NULL;
PRO_Viewpoint *viewpoint = NULL;
PRO_Database  *scene = NULL;
PRO_Model     *model1 = NULL;
PRO_Vertex    vertices[6] ;
PRO_Polygon   *polygon1 = NULL;
PRO_Polygon   *polygon2 = NULL;
long          i ;
double        yaw ;

// Initialize the hardware. The default memory configuration is used.
status = PRO_Init() ;

// Instantiate a device corresponding to logical unit (SCSI ID) 0.
dev0 = new PRO_Device(0) ;

// Instantiate a model to which polygons will be attached.
model1 = new PRO_Model() ;
if ( model1->GetModelStatus() != PRO_SUCCESS )
{
    printf("Failed instantiating model.\n") ;
    delete model1 ;
    PRO_Stop() ;
    return -1 ;
}

// Set the model's initial position and orientation.
model1->SetPosition( 0.0, 0.0, 0.0 ) ;
model1->SetOrientation( 0.0, 0.0, 0.0 ) ;

// Instantiate a database and attach the model to the scene.
scene = new PRO_Database () ;
scene->AttachModel ( model1 ) ;

// Instantiate a viewport and attach the scene to the viewport.
viewport = new PRO_Viewport() ;
viewport->AttachDatabase ( scene ) ;
```



Sample Applications

```
// Instantiate a viewpoint and attach the viewpoint to the viewport.
viewpoint = new PRO_Viewpoint() ;
viewport->AttachViewpoint ( viewpoint ) ;

// Attach the viewport to the device.
dev0->AttachViewport( viewport ) ;

// Set the viewpoint's initial position and orientation.
viewpoint->SetPosition ( -800.0 , 0.0 , 0.0 ) ;
viewpoint->SetOrientation ( 0.0 , 0.0 , 0.0 ) ;

// Initialize the vertex coordinates for two quadrilaterals.
vertices[0].SetCoordinates ( 0.0 , 100.0 , 100.0 ) ;
vertices[1].SetCoordinates ( 0.0 , -100.0 , 100.0 ) ;
vertices[2].SetCoordinates ( 0.0 , -100.0 , -100.0 ) ;
vertices[3].SetCoordinates ( 0.0 , 100.0 , -100.0 ) ;
vertices[4].SetCoordinates ( 100.0 , -100.0 , -100.0 ) ;
vertices[5].SetCoordinates ( 100.0 , 100.0 , -100.0 ) ;

// Create two immediate mode polygons
polygon1 = new PRO_Polygon() ;
polygon2 = new PRO_Polygon() ;

// Attach polygons to model
model1->AddPolygon ( polygon1 ) ;
model1->AddPolygon ( polygon2 ) ;

// Make the polygons double sided so that they are always visible.
polygon1->IsDoubleSided() ;
polygon2->IsDoubleSided() ;

// Assemble the vertex list for polygon 1
polygon1->AddVertex ( &vertices[0] ) ;
polygon1->AddVertex ( &vertices[1] ) ;
polygon1->AddVertex ( &vertices[2] ) ;
polygon1->AddVertex ( &vertices[3] ) ;

// Assemble the vertex list for polygon 2
polygon2->AddVertex ( &vertices[0] ) ;
```



Sample Applications

```
polygon2->AddVertex ( &vertices[1] );
polygon2->AddVertex ( &vertices[4] );
polygon2->AddVertex ( &vertices[5] );

// Spin the model around on its z axis for a while.
for(i=0; i<3600; i++)
{
    // Change the vertex coordinates in real-time.
    vertices[0].SetCoordinates ( 0.0 , 100.0 + (30.0*cos(i/10.0)) , 100.0 + (30.0*cos(i/10.0)) );
    vertices[1].SetCoordinates ( 0.0 , -100.0 - (30.0*cos(i/10.0)) , 100.0 + (30.0*cos(i/10.0)) );
    vertices[2].SetCoordinates ( 0.0 , -100.0 + (30.0*cos(i/10.0)) , -100.0 - (30.0*cos(i/10.0)) );
    vertices[3].SetCoordinates ( 0.0 , 100.0 - (30.0*cos(i/10.0)) , -100.0 - (30.0*cos(i/10.0)) );

    yaw = (double)(i % 360) ;
    model1->ProcessImmediateModeData() ;
    model1->SetOrientation ( 0.0 , 0.0 , yaw ) ;

    // Send display list to the hardware for rendering.
    status = PRO_DisplayDatabase() ;
}

// Delete all the dynamically allocated objects.
if ( viewpoint ) delete viewpoint ;
if ( viewport ) delete viewport ;
if ( scene ) delete scene ;
if ( model1 ) delete model1 ;
if ( dev0 ) delete dev0 ;
if ( polygon1 ) delete polygon1 ;
if ( polygon2 ) delete polygon2 ;

// Halt communication with the hardware.
status = PRO_Stop() ;

return 0 ;
```



FUNCTION REFERENCE

Configuration Functions

The Configuration functions provide customized control over the PRO-1000 hardware configuration. Note that making configuration changes may be done only before communication with the PRO-1000 has been established (before PRO_Init is called). By default all logical units are set to receive data. A default memory configuration is provided by the PRO/API. It is not mandatory to use these functions.

Configuration functions are the only PRO/API functions which may be invoked before a call to PRO_Init.

(Configuration Functions)

PRO_ConfigureMemory

Sets the memory configuration.

```
PRO_Status PRO_ConfigureMemory ( long num_viewports,  
                                long num_blend_tables,  
                                long num_databases,  
                                long num_models_per_db,  
                                long num_matrices,  
                                long update_memsize,  
                                long animation_frame_count ) ;
```

Description

Sets the hardware memory configuration to support the number of entities specified by the user.



Function Reference

Arguments

<code>long num_viewports</code>	Maximum number of PRO_Viewport objects to be instantiated by the application.
<code>long num_blend_tables</code>	Maximum number of PRO_LOD_Table objects to be instantiated by the application.
<code>long num_databases</code>	Maximum number of PRO_Database objects to be instantiated by the application.
<code>long num_models_per_db</code>	Maximum number of PRO_Model objects to be associated with any one PRO_Database.
<code>long num_matrices</code>	Maximum number of coordinate sets to be used across all models in all databases in all viewports. The default value is 1024, and the maximum value supported by the hardware is 4096.
<code>long update_memsize</code>	The size in bytes of the maximum amount of data to be passed from the application to the renderer. The default is 393216 bytes (96K 32-bit words). The value must be a multiple of 4 bytes.
<code>long animation_frame_count</code>	The maximum number of frames in any animation. The default is zero.

Return Value

PRO_SUCCESS if successful.
PRO_FAILURE if function was invoked after calling PRO_Init.

Application Notes

This function may be invoked prior to calling function PRO_Init(), but at no other time.



Function Reference

(Configuration Functions)

PRO_AbbreviatedConfigureMemory

Sets the memory configuration.

```
PRO_Status PRO_AbbreviatedConfigureMemory ( long num_matrices,  
                                             long update_memsize,  
                                             long pingpong_memsize ) ;
```

Description

Sets the hardware memory configuration to support the number of entities specified by the user.

Arguments

long <i>num_matrices</i>	Maximum number of coordinate sets to be used across all models in all databases in all viewports. The default value is 4096, which is the maximum number supported by the hardware.
long <i>update_memsize</i>	The size in bytes of the maximum amount of data to be passed from the application to the renderer. The default is 393216 bytes (96K 32-bit words). The value must be a multiple of 4 bytes.
long <i>pingpong_memsize</i>	The size in bytes of culling memory where viewports, animation sequences, and databases are stored. The default is 16384 bytes. The value must be a multiple of 4 bytes.

Return Value

PRO_SUCCESS if successful.
PRO_FAILURE if function was invoked after calling PRO_Init.



Function Reference

Application Notes

This function may be invoked prior to calling function `PRO_Init()`, but at no other time.



Function Reference

(Configuration Functions)

PRO_SetLogicalUnitOn

Allows display list data to be processed on this device.

```
PRO_Status PRO_SetLogicalUnitOn ( long lun ) ;
```

Description

Indicates that display list data should be sent to and be processed by the PRO-1000 unit associated with the specified logical unit. This is the default state for all attached units.

Arguments

long <i>lun</i>	The logical unit number of the device. The default value is -1.
-----------------	--

Return Value

PRO_SUCCESS if successful.
PRO_FAILURE if function was invoked after calling PRO_Init().

Application Notes

This function may be invoked prior to calling function PRO_Init(), but at no other time.

The default value for the logical unit number (-1) denotes the default device. The characteristics and initial attributes of the default device are described in the Device section in the Concepts and Object Data Types chapter.



Function Reference

(Configuration Functions)

PRO_SetLogicalUnitOff

Inhibits display list data from being processed on this device.

```
PRO_Status PRO_SetLogicalUnitOff ( long lun ) ;
```

Description

Indicates that display list data should not be sent to or processed by this PRO-1000 unit. This is used so that two applications running on the same host computer can communicate with a different subset of attached PRO-1000 units.

Arguments

long <i>lun</i>	The logical unit number of the device. The default value is -1.
-----------------	--

Return Value

PRO_SUCCESS if successful.
PRO_FAILURE if function was invoked after calling PRO_Init().

Application Notes

This function may be invoked prior to calling function PRO_Init(), but at no other time.

The default value for the logical unit number denotes the default device. The characteristics and initial attributes of the default device are described in the Device section in the Concepts and Object Data Types chapter.



Hardware Interface and Global Functions

The Hardware Interface and Global functions are used to communicate with the PRO-1000 hardware, to get status back from the hardware, and to configure general attributes which are not associated with any specific display list object.

(Hardware Interface and Global Functions)

PRO_Init

Initializes communication with the PRO-1000 hardware.

```
PRO_Status PRO_Init ( PRO_Update_Rate update_rate,  
                     PRO_Overload_Mode overload_mode,  
                     PRO_Display_Select display_select,  
                     PRO_Line_Rate line_rate ) ;
```

Description

Establishes communications with the graphics engine and initializes the control structures used by the PRO/API graphics library. With the exception of calls to Configuration functions, no graphics library functions may be called and no graphics library objects may be instantiated before PRO_Init() is invoked.

Arguments

PRO_Update_Rate <i>update_rate</i>	The rate at which graphics updates are processed, based on the display line rate. Values are UPDATE_60_HERTZ, UPDATE_50_HERTZ, UPDATE_30_HERTZ, and UPDATE_25_HERTZ. The default value is UPDATE_60_HERTZ. See PRO_SetUpdateRate for details.
------------------------------------	---



Function Reference

PRO_Overload_Mode <i>overload_mode</i>	The processing overload behavior of the hardware. Values are OVERLOAD_ON, OVERLOAD_OFF and EXTENDED_OVERLOAD. The default is OVERLOAD_OFF. See PRO_SetOverloadMode for details.
PRO_Display_Select <i>display_select</i>	The number of output devices supported by the graphics engine. Values are SINGLE_DISPLAY, DUAL_DISPLAY, and STEREO_DISPLAY. The default is SINGLE_DISPLAY.
PRO_Line_Rate <i>line_rate</i>	The displayed lines and pixels. Values are dependent on firmware configuration. The default is VGA_640X480. See PRO_SetDisplaySelect for details.

Return Value

PRO_SUCCESS or PRO_FAILURE.

Application Notes

This function must be the first called by the application after (optionally) calling any memory configuration or logical unit enable/disable functions. This function may not be used again by the application until PRO_Stop has been invoked.

The update rate must match the selected display line rate. For stereo, use the UPDATE_120_HERTZ update rate, the STEREO_DISPLAY display select, and the VGA_640X480 line rate.



Function Reference

(Hardware Interface and Global Functions)

PRO_InitOffline

Initializes display list for offline mode.

```
PRO_Status PRO_InitOffline ( void ) ;
```

Description

Initializes the control structures used by the PRO/API graphics library for processing in offline mode. Offline mode is used to process models and save formatted data to a file; in this mode, no data will be rendered by the PRO-1000. No other functions should be called before PRO_InitOffline() is invoked.

Arguments

None.

Return Value

PRO_SUCCESS or PRO_FAILURE.

Application Notes

This function is an alternative to calling PRO_Init().

This function must be the first called by the application. This function may not be used again by the application until PRO_Stop has been invoked.



Function Reference

(Hardware Interface and Global Functions)

PRO_Stop

Shuts down communication with the PRO-1000 hardware.

```
PRO_Status PRO_Stop ( void ) ;
```

Description

Shuts down communications to the PRO-1000 graphics engine and frees up all memory associated with graphics library structures. Any PRO/API objects that were allocated dynamically must be deleted before calling PRO_Stop.

Arguments

None.

Return Value

PRO_SUCCESS or PRO_FAILURE.

Application Notes

No PRO/API objects are valid after calling PRO_Stop. Only calls to PRO/API Configuration functions may be made between PRO_Stop and a call to PRO_Init.



Function Reference

(Hardware Interface and Global Functions)

PRO_DisplayDatabase

Processes the updates to the display list.

```
PRO_Status PRO_DisplayDatabase ( void ) ;
```

Description

Processes updates to the display list and writes them to the PRO-1000 for rendering.

Arguments

None.

Return Value

PRO_SUCCESS or PRO_FAILURE.

Application Notes

The picture on the display will not change until this function is invoked.



(Hardware Interface and Global Functions)

PRO_LoadModelDynamically

Loads a pre-formatted model dynamically.

```
void *PRO_LoadModelDynamically ( PRO_Model **model_pointer_address,  
                                char *filename,  
                                char *texture_path,  
                                char *uf_path,  
                                long high_priority_model,  
                                void (*user_callback)( PRO_Model *model,  
                                                       void *user_data ),  
                                void *user_data );
```

Description

This function is used to issue a request for a dynamic model load. The model requested will be added to the dynamic load model queue and processed at a later time as allowed by the host system CPU and I/O load. Completion of a model load request is signalled by a valid model pointer and, if requested, a user callback will be invoked.

Arguments

PRO_Model **model_pointer_address	Address of user pointer which will be initialized to zero during this call and then set to a valid PRO_Model pointer value when the requested model load is complete.
char *filename	The name of file (full path but no extension) containing model data to be loaded.
char *texture_path	The path to texture maps. By default, the directory containing the model is used.



Function Reference

<code>char *<i>uf_path</i></code>	The path to pre-formatted external reference files. By default, the directory containing the model is used.
<code>long <i>high_priority_model</i></code>	Indicates whether this is a high priority model whose load request should be processed immediately. This flag defaults to 0 (false).
<code>void (*<i>user_callback</i>) (PRO_Model *<i>model</i>, void *<i>user_data</i>)</code>	User function to be called after the model load request has been processed. By default, no callback is made.
<code>void *<i>user_data</i></code>	Pointer to the data that will be passed to the user defined callback function when invoked. By default, no user data is passed.

Return Value

An identifier that is unique for every model loaded. The unique ID can be used to unload the model in the cases when the dynamic load request has not yet been satisfied.

Application Notes

The picture on the display will not change when this function is invoked.



Function Reference

(Hardware Interface and Global Functions)

PRO_UnloadDynamicModel

Unloads a dynamically loaded model.

```
void PRO_UnloadDynamicModel ( void *id );
```

Description

Removes a model from the dynamic load model queue. The ID that is passed must be an ID returned by the PRO_LoadModelDynamically function.

Arguments

void *id	ID returned by PRO_LoadModelDynamically.
----------	--

Return Value

None.

Application Notes

This function may be called only if the dynamic model request is pending. If called with an invalid or no longer current ID (the model has already been loaded), this function will do nothing.



Function Reference

(Hardware Interface and Global Functions)

PRO_SetGenlock

Sets the GENLOCK between multiple PRO-1000 units.

```
void PRO_SetGenlock ( long master_lun ) ;
```

Description

GENLOCKS the displays when there is more than one PRO-1000 unit or when there is an external device driving the display sync.

Arguments

long <i>master_lun</i>	The ID of the master PRO-1000. Values are 0 through (number of units – 1), PRO_EXTERNAL_GENLOCK, or PRO_NO_GENLOCK. The default is 0.
------------------------	---

Return Value

None.

Application Notes

At initialization, PRO-1000 units are not GENLOCK'd.



(Hardware Interface and Global Functions)

PRO_SetUpdateRate

Sets the realtime update rate.

```
PRO_Status PRO_SetUpdateRate ( PRO_Update_Rate update_rate, long lun ) ;
```

Description

Sets the realtime update rate to the full display update rate (60 or 50 hertz) or to half the display update rate (30 or 25 hertz).

Refer to the Hardware Overview section for Display Interface characteristics.

Arguments

PRO_Update_Rate <i>update_rate</i>	The rate at which graphics updates are processed, based on the display line rate. Valid values are UPDATE_60_HERTZ, UPDATE_30_HERTZ, UPDATE_50_HERTZ, and UPDATE_25_HERTZ.
long <i>lun</i>	The logical unit number of the device. The default value is -1.

Return Value

PRO_SUCCESS if successful.
PRO_INVALID_LUN if an invalid logical unit number was provided.
PRO_UNSUPPORTED_MODE if an invalid update rate was provided.

Application Notes

The default value for the logical unit number denotes the default device. The characteristics and initial attributes of the default device are described in the Device section in the Concepts and Object Data Types chapter.



(Hardware Interface and Global Functions)

PRO_SetOverloadMode

Sets the realtime processing overload behavior.

This function is obsolete.

To set the overload mode for a device, instantiate a PRO_Device object for a specific logical unit and invoke Device Function SetOverloadMode().

To set the overload mode for the default device (logical unit -1), get a pointer to the default PRO_Device object using function PRO_GetDefaultDevice(). The characteristics and initial attributes of the default device are described in the Device section in the Concepts and Object Data Types chapter.



(Hardware Interface and Global Functions)

PRO_SetDisplaySelect

Sets the display sync select based on number of displays and the desired line rate.

This function is obsolete.

To set the display sync select for a device, instantiate a PRO_Device object for a specific logical unit and invoke Device Function SetDisplaySelect().

To set the display sync select for the default device (logical unit -1), get a pointer to the default PRO_Device object using function PRO_GetDefaultDevice(). The characteristics and initial attributes of the default device are described in the Device section in the Concepts and Object Data Types chapter.



(Hardware Interface and Global Functions)

PRO_SetForegroundColor

Sets the foreground color of the display.

This function is obsolete.

To set the foreground color for a device, instantiate a PRO_Device object for a specific logical unit and invoke Device Function SetForegroundColor().

To set the foreground color for the default device (logical unit -1), get a pointer to the default PRO_Device object using function PRO_GetDefaultDevice(). The characteristics and initial attributes of the default device are described in the Device section in the Concepts and Object Data Types chapter.



(Hardware Interface and Global Functions)

PRO_SetBackgroundColor

Sets the background color of the display.

This function is obsolete.

To set the background color for a device, instantiate a PRO_Device object for a specific logical unit and invoke Device Function SetBackgroundColor().

To set the background color for the default device (logical unit -1), get a pointer to the default PRO_Device object using function PRO_GetDefaultDevice(). The characteristics and initial attributes of the default device are described in the Device section in the Concepts and Object Data Types chapter.



(Hardware Interface and Global Functions)

PRO_SetGamma

Sets the gamma correction for the device.

This function is obsolete.

To set the gamma correction for a device, instantiate a PRO_Device object for a specific logical unit and invoke Device Function SetGamma().

To set the gamma correction for the default device (logical unit -1), get a pointer to the default PRO_Device object using function PRO_GetDefaultDevice(). The characteristics and initial attributes of the default device are described in the Device section in the Concepts and Object Data Types chapter.



Function Reference

(Hardware Interface and Global Functions)

PRO_SetPointLightDetail

Sets the number of edges of the lights.

```
void PRO_SetPointLightDetail ( long number_of_sides ) ;
```

Description

Sets the number of edges of the point light model.

Arguments

long <i>number_of_sides</i>	The number of sides for the point light polygon. Valid number of sides are 4 through 31.
-----------------------------	---

Return Value

None.

Application Notes

None.



Function Reference

(Hardware Interface and Global Functions)

PRO_SetPointLightFeatureType

Sets the point light size clamp feature type.

```
void PRO_SetPointLightFeatureType ( long feature_type );
```

Description

Sets the feature type for the point light polygon. The blending and volume clamping can be controlled by the blend table entry corresponding to the input *feature_type* index.

Arguments

<i>long feature_type</i>	The feature type (blend table entry index) of the point light model.
--------------------------	--

Return Value

None.

Application Notes

None.



Function Reference

(Hardware Interface and Global Functions)

PRO_SetPointLightSize

Sets the physical culling size of the lights.

```
void PRO_SetPointLightSize ( float size ) ;
```

Description

Sets the physical culling size of the polygon to be used as the point light model in database units.

Arguments

float *size* The radius of the point light polygon. The default value is 1.0.

Return Value

None.

Application Notes

None.



Function Reference

(Hardware Interface and Global Functions)

PRO_SetPointLightPolygon

Sets the polygon to be used as point light model.

```
void PRO_SetPointLightPolygon ( PRO_Polygon *point_light_polygon ) ;
```

Description

Defines the polygon to be used as a point light model.

Arguments

PRO_Polygon * <i>point_light_polygon</i>	A pointer to the PRO_Polygon object to be used as the point light polygon.
--	--

Return Value

None.

Application Notes

This function may be used if the default polygon provided by the PRO/API does not meet the user requirements. The point light polygon must be modeled in the Y-Z plane.



Function Reference

(Hardware Interface and Global Functions)

PRO_SetMicrotextureMapCount

Sets the number of microtexture maps.

```
void PRO_SetMicrotextureMapCount ( long count ) ;
```

Description

Sets the number of microtexture maps to be used. This value defaults to eight on a PRO-1000 with expanded texture memory; otherwise, the default is four. This value can be set to zero, four on any system. This value can be set to eight on a PRO-1000 with expanded texture memory.

Arguments

<code>long <i>count</i></code>	The number of microtexture maps. Valid settings are zero, four, or eight (expanded texture memory systems only).
--------------------------------	--

Return Value

None.

Application Notes

This function must be invoked before any textures are created. This function affects the texture memory configuration so it may not be called again once any PRO_Texture objects have been created.



(Hardware Interface and Global Functions)

PRO_SetTexturePageSize

Sets the page size in texture memory.

This function is obsolete.



Function Reference

(Hardware Interface and Global Functions)

PRO_ReloadLogo

Reloads the spinning Real 3D model.

```
void PRO_ReloadLogo ( void ) ;
```

Description

Reloads the default database, which is the spinning Real 3D logo.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Hardware Interface and Global Functions)

PRO_GetNumberOfLogicalUnits

Returns the number of attached PRO-1000 devices.

```
long PRO_GetNumberOfLogicalUnits ( void ) ;
```

Description

Returns the number of PRO-1000 units found on the SCSI bus.

Arguments

None.

Return Value

The number of PRO-1000 systems found, or zero if the function is called before PRO_Init.

Application Notes

None.



Function Reference

(Hardware Interface and Global Functions)

PRO_GetNumberOfAvailableTexels

Returns the size of texture memory in texels.

```
long PRO_GetNumberOfAvailableTexels ( void ) ;
```

Description

Returns the available texture memory size, in texels.

Arguments

None.

Return Value

The number of free texels currently available in texture memory.

Application Notes

None.



Function Reference

(Hardware Interface and Global Functions)

PRO_GetMicrotextureMapCount

Returns the number of microtexture maps.

```
long PRO_GetMicrotextureMapCount ( void ) ;
```

Description

Returns the maximum number of microtexture maps defined by the current texture memory configuration.

Arguments

None.

Return Value

The maximum number of microtexture maps.

Application Notes

If not explicitly set by user, this value defaults to eight on a PRO-1000 with expanded texture memory; four on other systems.



Function Reference

(Hardware Interface and Global Functions)

PRO_GetFirmwareRevision

Returns the firmware version of the PRO device.

This function is obsolete.

To get the firmware revision associated with a PRO-1000 device, use the Device Function GetFirmwareRevision(). This requires a PRO_Device object for each logical unit.



(Hardware Interface and Global Functions)

PRO_GetModelNumber

Return the model number of the PRO device.

This function is obsolete.

To get the model number associated with a PRO–1000 device, use the Device Function GetModelNumber(). This requires a PRO_Device object for each logical unit.



(Hardware Interface and Global Functions)

PRO_GetRealtimeClockCount

Get the processing time of a given logical unit.

This function is obsolete.

To get the real-time clock count associated with a PRO-1000 device, use the Device Function `GetRealtimeClockCount()`. This requires a `PRO_Device` object for each logical unit.



(Hardware Interface and Global Functions)

PRO_GetProcessingTime

Get the processing time of a given logical unit.

This function is obsolete.

To get the processing time associated with a PRO-1000 device, use the Device Function `GetProcessingTime()`. This requires a `PRO_Device` object for each logical unit.



(Hardware Interface and Global Functions)

PRO_GetFrameRateEstimate

Get the frame update rate of a given logical unit.

This function is obsolete.

To get the frame rate associated with a PRO-1000 device, use the Device Function `GetFrameRateEstimate()`. This requires a `PRO_Device` object for each logical unit.



Function Reference

(Hardware Interface and Global Functions)

PRO_GetPolygonMemorySize

Returns the size of Polygon memory.

```
long PRO_GetPolygonMemorySize ( void ) ;
```

Description

Gets the size in bytes of the polygon area in hardware memory. This value will vary based on the number of pixel daughter boards in the PRO-1000. The standard polygon memory is eight megabytes. Each additional polygon memory board (up to 3) adds eight megabytes of polygon storage capacity. If there is more than one PRO-1000 unit attached, the smallest polygon memory size across all units will be returned.

Arguments

None.

Return Value

The smallest polygon memory size (in bytes) across all attached PRO-1000 units.

Application Notes

None.



Function Reference

(Hardware Interface and Global Functions)

PRO_GetDefaultDevice

Returns the default PRO Device object.

```
PRO_Device *PRO_GetDefaultDevice ( void ) ;
```

Description

Gets a pointer to the PRO_Device object representing the default device.

Arguments

None.

Return Value

A pointer to the PRO_Device object representing the default device.

Application Notes

The characteristics and initial attributes of the default device are described in the Device section in the Concepts and Object Data Types chapter.



Device Functions

The Device Class functions are used to assign graphical objects to a specified PRO-1000 device.

(Device Functions)

PRO_Device class constructor

Creates an instance of a PRO Device.

```
PRO_Device ( long logical_unit, long use_gamma_from_rom ) ;
```

Description

Creates an instance of a PRO_Device. The logical unit number (LUN) reflects the order in which PRO-1000 units are detected on the SCSI bus, and it is not necessarily the same as the SCSI ID. For example, two PRO-1000 units on the bus, with SCSI ID's 5 and 6, would be assigned LUNs 0 and 1.

If a gamma correction table has been stored in ROM for the PRO-1000 device, the stored table can be used as the initial state for the device. By default, a stored table is not used. If a stored gamma table is not used, the device inherits its initial gamma value from the default device, which has an initial gamma value of 1.0.

Arguments

long *logical_unit*

The logical ID of the PRO-1000 unit. Valid values are 0 through 7, or -1 to specify broadcast to all attached PRO-1000 units. The default value is -1.

long *use_gamma_from_rom*

Indicates whether gamma table stored in ROM should be used when device is instantiated. This flag defaults to 0 (false).



Function Reference

Return Value

None.

Application Notes

Device objects inherit attributes (gamma, foreground and background color, overload mode and display select) set for the default device before the device is instantiated.

The GetStatus function should be called after object instantiation.



Function Reference

(Device Functions)

PRO_Device class destructor

Destroys an instance of a PRO Device.

```
~PRO_Device ( void ) ;
```

Description

Destroys an instance of a PRO_Device.

Arguments

None.

Return Value

None.

Application Notes

Any viewports attached to the PRO_Device being destroyed will be attached to the default device.



Function Reference

(Device Functions)

GetStatus

Gets the status of a PRO Device object instantiation.

```
PRO_Status GetStatus ( void ) ;
```

Description

Returns the status of object instantiation.

Arguments

None.

Return Value

PRO_SUCCESS if object creation was successful.

PRO_INVALID_LUN if an invalid logical unit was specified.

PRO_INVALID_LINE_RATE if the requested display select/line rate combination is not supported on the device.

Application Notes

If status PRO_INVALID_LINE_RATE is returned, the device has been defaulted to its first valid sync select.



Function Reference

(Device Functions)

AttachViewport

Attaches a viewport to a PRO Device.

```
PRO_Status AttachViewport ( PRO_Viewport *viewport_ptr ) ;
```

Description

Attaches a viewport to a PRO_Device. This defines the PRO-1000 unit that will render the viewport and its attached scenes. By default, a viewport is rendered on all active devices.

Arguments

PRO_Viewport * <i>viewport_ptr</i>	A pointer to a PRO_Viewport object.
------------------------------------	-------------------------------------

Return Value

PRO_SUCCESS if successful.

PRO_FAILURE if the maximum number of viewports were already attached.

Application Notes

None.



Function Reference

(Device Functions)

DetachViewport

Detaches a viewport from a PRO Device.

```
void DetachViewport ( PRO_Visport *viewport_ptr ) ;
```

Description

Indicates that the viewport should no longer be displayed on a device. Once a viewport is detached from a specific device, it reverts to being rendered on all active devices.

Arguments

PRO_Visport * <i>viewport_ptr</i>	A pointer to a PRO_Visport object.
-----------------------------------	------------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Device Functions)

SetGamma

Sets the gamma correction for the device.

```
PRO_Status SetGamma ( float gamma, long bank ) ;
```

Description

Sets the gamma correction table for the device using the specified factor as a seed to calculate the table. The initial setting for the default device is a gamma of 1.0. If the device is a high-resolution model using a dual display mode, different gamma tables may be loaded into memory for each channel using the *bank* argument. Gamma table processing is activated by this function.

Arguments

float *gamma*
long *bank*

The gamma value for the device.
Where to load the gamma table. This is selectable only for model 1400 series systems. Values are -1, 0 or 1. The default value is -1, which loads the table into both memory banks.

Return Value

PRO_FAILURE if *bank* 1 was requested for a device that is not a high-resolution system; PRO_SUCCESS otherwise.

Application Notes

Device objects attributes (gamma, foreground and background color, overload mode and display select) are initialized to the settings for the default device at the time of the object's creation. Changes made to the default device attributes after a device object's instantiation will not be propagated to that device.



Function Reference

(Device Functions)

SetGammaRGB

Sets the gamma correction for the device.

```
PRO_Status SetGammaRGB ( unsigned char *gamma_table_red,  
                          unsigned char *gamma_table_green,  
                          unsigned char *gamma_table_blue,  
                          long bank );
```

Description

Sets the gamma correction table for the device using the specified arrays of intensities for the red, green, and blue bands. If the device is a high-resolution model using a dual display mode, different gamma tables may be loaded into memory for each channel using the *bank* argument. Gamma table processing is activated by this function.

Arguments

unsigned char * <i>gamma_table_red</i>	Pointer to array of values for red band data. Values in array are 0 to 255.
unsigned char * <i>gamma_table_green</i>	Pointer to array of values for green band data. Values in array are 0 to 255.
unsigned char * <i>gamma_table_blue</i>	Pointer to array of values for blue band data. Values in array are 0 to 255.
long <i>bank</i>	Where to load the gamma table. This is selectable only for model 1400 series systems. Values are -1, 0 or 1. The default value is -1, which loads the table into both memory banks.

Return Value

PRO_FAILURE if *bank* 1 was requested for a device that is not a high-resolution system; PRO_SUCCESS otherwise.



Application Notes

Device objects attributes (gamma, foreground and background color, overload mode and display select) are initialized to the settings for the default device at the time of the object's creation. Changes made to the default device attributes after a device object's instantiation will not be propagated to that device.



Function Reference

(Device Functions)

ActivateGamma

Turns on gamma table processing.

```
void ActivateGamma ( void ) ;
```

Description

Activates gamma table processing. By default, gamma table processing is active on a device.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Device Functions)

DeactivateGamma

Turns off gamma table processing.

```
void DeactivateGamma ( void ) ;
```

Description

Deactivates gamma table processing. By default, gamma table processing is active on a device.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Device Functions)

SaveGammaToROM

Saves table in gamma memory to ROM.

```
void SaveGammaToROM ( void ) ;
```

Description

Stores the table(s) in gamma memory to ROM, for retention between power cycles.

Arguments

None.

Return Value

None.

Application Notes

Use of the stored gamma table is controlled by a flag passed to the PRO_Device constructor.



Function Reference

(Device Functions)

WriteBlendMemory

Writes data to blend memory.

```
PRO_Status WriteBlendMemory ( long start_i, long start_j,  
                             long size_i, long size_j,  
                             unsigned char *bm_data_red,  
                             unsigned char *bm_data_green,  
                             unsigned char *bm_data_blue,  
                             long bank ) ;
```

Description

The blend memory has an entry for each eight pixels of screen space, where each entry is an attenuation value (for red, green, and blue independently) between 0 and 255. For dual line rates, the blend memory is split vertically into two sections which can be written individually using the *bank* argument.

Arguments

long <i>start_i</i>	Start line, defining upper left corner.
long <i>start_j</i>	Start pixel, defining upper left corner.
long <i>size_i</i>	Number of lines. Values are 1 to 96.
long <i>size_j</i>	Number of pixels per line. Values are 1 to 128.
unsigned char * <i>bm_data_red</i>	Pointer to array of values for red band data. Values in array are 0 to 255.
unsigned char * <i>bm_data_green</i>	Pointer to array of values for green band data. Values in array are 0 to 255.
unsigned char * <i>bm_data_blue</i>	Pointer to array of values for blue band data. Values in array are 0 to 255.
long <i>bank</i>	Which blend memory bank to load. Values are 0 or 1.



Function Reference

Return Value

PRO_FAILURE if this function was invoked on a device that is not a high-resolution system or if the value of bank is not 0 or 1; PRO_SUCCESS otherwise.

Application Notes

This function is valid only on model 1400 series systems.



Function Reference

(Device Functions)

ActivateBlendMemory

Turns on soft edge blending.

```
void ActivateBlendMemory ( void ) ;
```

Description

Activates soft edge blending. By default, soft edge blending is active on a high-resolution device.

Arguments

None.

Return Value

None.

Application Notes

This function is valid only for 1400 series systems.



Function Reference

(Device Functions)

DeactivateBlendMemory

Turns off soft edge blending.

```
void DeactivateBlendMemory ( void ) ;
```

Description

Deactivates soft edge blending. By default, soft edge blending is active on a high-resolution device.

Arguments

None.

Return Value

None.

Application Notes

This function is valid only for 1400 series systems.



Function Reference

(Device Functions)

SaveBlendMemoryToROM

Saves table in blend memory to ROM.

```
void SaveGammaToROM ( void ) ;
```

Description

Stores the table(s) in blend memory to ROM, for retention between power cycles.

Arguments

None.

Return Value

None.

Application Notes

This function is valid only for 1400 series systems.



Function Reference

(Device Functions)

SetForegroundColor

Sets the foreground color for the device.

```
void SetForegroundColor ( long red, long green, long blue ) ;
```

Description

Sets the foreground color for the device using 8-bit color values. The initial setting for the default device is a black foreground color.

Arguments

long <i>red</i>	The intensity of the red component of the color. Value is zero through 255.
long <i>green</i>	The intensity of the green component of the color. Value is zero through 255.
long <i>blue</i>	The intensity of the blue component of the color. Value is zero through 255.

Return Value

None.

Application Notes

Device objects attributes (gamma, foreground and background color, overload mode and display select) are initialized to the settings for the default device at the time of the object's creation. Changes made to the default device attributes after a device object's instantiation will not be propagated to that device.



Function Reference

(Device Functions)

SetForegroundColor

Sets the foreground color for the device.

```
void SetForegroundColor ( float red, float green, float blue ) ;
```

Description

Sets the foreground color for the device using normalized color values. The initial setting for the default device is a black foreground color.

Arguments

long <i>red</i>	The intensity of the red component of the color. Value is 0.0 through 1.0.
long <i>green</i>	The intensity of the green component of the color. Value is 0.0 through 1.0.
long <i>blue</i>	The intensity of the blue component of the color. Value is 0.0 through 1.0.

Return Value

None.

Application Notes

Device objects attributes (gamma, foreground and background color, overload mode and display select) are initialized to the settings for the default device at the time of the object's creation. Changes made to the default device attributes after a device object's instantiation will not be propagated to that device.



Function Reference

(Device Functions)

SetBackgroundColor

Sets the background color for the device.

```
void SetBackgroundColor ( long red, long green, long blue ) ;
```

Description

Sets the background color for the device. The initial setting for the default device is a black background color.

Arguments

long <i>red</i>	The intensity of the red component of the color. Value is zero through 255.
long <i>green</i>	The intensity of the green component of the color. Value is zero through 255.
long <i>blue</i>	The intensity of the blue component of the color. Value is zero through 255.

Return Value

None.

Application Notes

Device objects attributes (gamma, foreground and background color, overload mode and display select) are initialized to the settings for the default device at the time of the object's creation. Changes made to the default device attributes after a device object's instantiation will not be propagated to that device.



Function Reference

(Device Functions)

SetBackgroundColor

Sets the background color for the device.

```
void SetBackgroundColor ( float red, float green, float blue ) ;
```

Description

Sets the background color for the device using normalized color values. The initial setting for the default device is a black background color.

Arguments

long <i>red</i>	The intensity of the red component of the color. Value is 0.0 through 1.0.
long <i>green</i>	The intensity of the green component of the color. Value is 0.0 through 1.0.
long <i>blue</i>	The intensity of the blue component of the color. Value is 0.0 through 1.0.

Return Value

None.

Application Notes

Device objects attributes (gamma, foreground and background color, overload mode and display select) are initialized to the settings for the default device at the time of the object's creation. Changes made to the default device attributes after a device object's instantiation will not be propagated to that device.



Function Reference

(Device Functions)

SetDisplaySelect

Sets the display sync select.

```
PRO_Status SetDisplaySelect ( PRO_Display_Select display_select,  
                             PRO_Line_Rate line_rate );
```

Description

Sets the display sync select based on the number of displays and the display resolution (number of pixels x number of lines).

Each PRO-1000 unit is programmed to handle 8 display select/line rate combinations, which may differ from the standard set of eight in the default firmware. The supported line rates are as follows:

VGA_640X480	(in default set)
SVGA_1024X768	(in default set)
NTSC_505X480	
CGA_512X384	
NTSC_512X486	(in default set)
NTSC_640X486	(in default set)
NTSC_720X486	(in default set)
PAL_720X576	(in default set)

Refer to the Hardware Overview section for Display Interface characteristics.

Arguments

PRO_Display_Select <i>display_select</i>	The number of displays supported by the graphics engine. Values are SINGLE_DISPLAY, DUAL_DISPLAY, and STEREO_DISPLAY.
PRO_Line_Rate <i>line_rate</i>	The displayed lines and pixels. Values are dependent on firmware configuration.



Function Reference

Return Value

PRO_SUCCESS if successful.

PRO_INVALID_LINE_RATE if the requested display select/line rate combination is not supported on the device

Application Notes

Device objects attributes (gamma, foreground and background color, overload mode and display select) are initialized to the settings for the default device at the time of the object's creation. Changes made to the default device attributes after a device object's instantiation will not be propagated to that device.

If the requested display select/line rate combination is not supported for this device, the device will be set to its first valid sync select. For the default firmware, that setting is SINGLE_DISPLAY, VGA_640X480.

For stereo, use the STEREO_DISPLAY display select and the VGA_640X480 line rate.



Function Reference

(Device Functions)

SetOverloadMode

Sets the realtime processing overload behavior.

```
PRO_Status SetOverloadMode ( PRO_Overload_Mode overload_mode );
```

Description

Sets the realtime processing overload behavior. When OVERLOAD_OFF is defined, the system will drop polygons in order to maintain the user specified update rate. When OVERLOAD_ON is defined, the system will halve the frame rate in an overload condition (60 Hz. will drop to 30 Hz., or 30 Hz. will drop to 15 Hz.). If there is not enough time to render the scene at half the frame rate, polygons will be dropped. When EXTENDED_OVERLOAD is specified, all polygons will be displayed, regardless of how long it takes. Realtime frame rates are not enforced in extended overload mode.

Arguments

PRO_Overload_Mode *overload_mode*

The processing overload behavior of the hardware.
Valid values are OVERLOAD_ON, OVERLOAD_OFF, or
EXTENDED_OVERLOAD.

Return Value

PRO_SUCCESS if successful.

PRO_UNSUPPORTED_MODE if an invalid overload mode was provided.

Application Notes

Device objects attributes (gamma, foreground and background color, overload mode and display select) are initialized to the settings for the default device at the time of the object's creation. Changes made to the default device attributes after a device object's instantiation will not be propagated to that device.



Function Reference

(Device Functions)

GetLogicalUnit

Returns the logical unit associated with a PRO Device.

```
long GetLogicalUnit ( void ) ;
```

Description

Returns the logical unit number to which a device is associated.

Arguments

None.

Return Value

The logical unit number.

Application Notes

None.



Function Reference

(Device Functions)

IsValidDisplaySelect

Indicates whether selection is valid on this device.

```
long IsValidDisplaySelect ( PRO_Display_Select display_select,  
                           PRO_Line_Rate line_rate ) ;
```

Description

Indicates whether the specified number of displays and display resolution combination is supported on this device. See SetDisplaySelect() for details about settings.

Arguments

PRO_Display_Select <i>display_select</i>	The number of displays supported by the graphics engine. Values are SINGLE_DISPLAY, DUAL_DISPLAY, and STEREO_DISPLAY.
PRO_Line_Rate <i>line_rate</i>	The displayed lines and pixels. Values are dependent on firmware configuration.

Return Value

One if combination is valid for this device; zero if combination is invalid.

Application Notes

For stereo, use the STEREO_DISPLAY display select and the VGA_640X480 line rate.



Function Reference

(Device Functions)

GetSyncSelectInfo

Gets number of displays and line rate associated with the display sync select.

```
void GetSyncSelectInfo ( long sync_select,  
                        PRO_Display_Select &display_select,  
                        PRO_Line_Rate &line_rate );
```

Description

Gets the number of displays (single or dual) and the line rate associated with the display sync select. This function queries the hardware about the display settings it supports.

Arguments

long <i>sync_select</i>	Display sync select, Values are 1 through PRO_MAX_LINE_RATES.
PRO_Display_Select & <i>display_select</i>	The number of displays supported by the graphics engine.
PRO_Line_Rate & <i>line_rate</i>	The displayed lines and pixels.

Return Value

None.

Application Notes

None.



Function Reference

(Device Functions)

HasAttachedViewports

Indicates whether a device has attached viewports.

long HasAttachedViewports (void) ;

Description

Indicates whether a device has any viewports attached.

Arguments

None.

Return Value

One if any viewports are attached.
Zero if no viewports are attached to the device.

Application Notes

None.



Function Reference

(Device Functions)

GetGamma

Returns the gamma value of the device.

```
float GetGamma ( void ) ;
```

Description

Returns the gamma value of the device.

Arguments

None.

Return Value

The gamma value of the device.

Application Notes

This function is valid only if function SetGamma() was used to set the gamma factor. If the gamma table was set using the SetGammaRGB() function or was read from ROM, the gamma factor returned by this function will not match the actual gamma table in memory.

The GetStatus function will return PRO_GAMMA_NO_MATCH if the gamma table was not generated using the gamma factor.



Function Reference

(Device Functions)

GetForegroundColor

Gets the foreground color of the device.

```
void GetForegroundColor ( long &red, long &green, long &blue ) ;
```

Description

Gets the foreground color of the device.

Arguments

long &red	The intensity of the red component of the color.
long &green	The intensity of the green component of the color.
long &blue	The intensity of the blue component of the color.

Return Value

None.

Application Notes

None.



Function Reference

(Device Functions)

GetBackgroundColor

Gets the background color of the device.

```
void GetBackgroundColor ( long &red, long &green, long &blue ) ;
```

Description

Gets the background color of the device.

Arguments

long &red	The intensity of the red component of the color.
long &green	The intensity of the green component of the color.
long &blue	The intensity of the blue component of the color.

Return Value

None.

Application Notes

None.



Function Reference

(Device Functions)

GetDisplaySelect

Returns the display mode of the device.

```
PRO_Display_Select GetDisplaySelect ( void ) ;
```

Description

Returns the actual display mode.

Arguments

None.

Return Value

SINGLE_DISPLAY or DUAL_DISPLAY.

Application Notes

This function can be used to determine the display mode if PRO_INVALID_LINE_RATE was returned from the GetDisplaySelect or GetStatus functions.



Function Reference

(Device Functions)

GetOverloadMode

Returns the current overload mode.

```
PRO_Overload_Mode GetOverloadMode ( void ) ;
```

Description

Returns the current overload processing mode.

Arguments

None.

Return Value

OVERLOAD_ON, OVERLOAD_OFF, or EXTENDED_OVERLOAD.

Application Notes

None.



Function Reference

(Device Functions)

GetLineRate

Returns the device line rate.

```
PRO_Line_Rate GetLineRate ( void ) ;
```

Description

Returns the actual device line rate.

Arguments

None.

Return Value

One of the supported line rates. See SetDisplaySelect for details.

Application Notes

This function can be used to determine the display resolution if PRO_INVALID_LINE_RATE was returned from the GetDisplaySelect or GetStatus functions.



Function Reference

(Device Functions)

IsHighRes

Tests for a high resolution system.

```
long IsHighRes ( void ) ;
```

Description

Returns true if the device is a PRO-1400 or PRO-1410.

Arguments

None.

Return Value

1 if the device is a PRO-1400 or PRO-1410.
0 if the device is not a PRO-1400 or PRO-1410.

Application Notes

Status for the default device (logical unit -1) does not exist, since the default device is a PRO/API concept and not a physical device. If the default device is queried, status for the first active device will be returned. See the Device section in the Concepts and Object Data Types chapter for more details about the default device.



Function Reference

(Device Functions)

HasExpandedTexture

Tests for expanded texture memory.

```
long HasExpandedTexture ( void ) ;
```

Description

Returns true if the device has expanded texture memory (32 megabytes).

Arguments

None.

Return Value

One if the device has expanded texture memory.
Zero otherwise.

Application Notes

Status for the default device (logical unit -1) does not exist, since the default device is a PRO/API concept and not a physical device. If the default device is queried, status for the first active device will be returned. See the Device section in the Concepts and Object Data Types chapter for more details about the default device.



Function Reference

(Device Functions)

GetFirmwareRevision

Returns the firmware version of the PRO device.

```
char* GetFirmwareRevision ( void ) ;
```

Description

Returns the firmware version of the PRO device.

Arguments

None.

Return Value

Pointer to a string that contains the firmware version.

Application Notes

Status for the default device (logical unit -1) does not exist, since the default device is a PRO/API concept and not a physical device. If the default device is queried, status for the first active device will be returned. See the Device section in the Concepts and Object Data Types chapter for more details about the default device.



Function Reference

(Device Functions)

GetModelNumber

Returns the model number of the PRO device.

```
long GetModelNumber ( void ) ;
```

Description

Returns the model number of the PRO device.

Arguments

None.

Return Value

1400 if the device is a PRO-1400 or PRO-1410.

1200 if the device is a PRO-1200 or PRO-1210.

1100 if the device is a PRO-1100 or PRO-1110.

Application Notes

Status for the default device (logical unit -1) does not exist, since the default device is a PRO/API concept and not a physical device. If the default device is queried, status for the first active device will be returned. See the Device section in the Concepts and Object Data Types chapter for more details about the default device.



Function Reference

(Device Functions)

GetRealtimeClockCount

Returns the device's image display processing time.

```
unsigned long GetRealtimeClockCount ( void ) ;
```

Description

Returns the number of hardware clocks that it took to process the last image. This can be used to determine the stress level of the system.

Arguments

None.

Return Value

Number of clock cycles (30 nanoseconds per cycle)

Application Notes

Status for the default device (logical unit -1) does not exist, since the default device is a PRO/API concept and not a physical device. If the default device is queried, status for the first active device will be returned. See the Device section in the Concepts and Object Data Types chapter for more details about the default device.



Function Reference

(Device Functions)

GetProcessingTime

Returns the device's image display processing time.

```
float GetProcessingTime ( void ) ;
```

Description

Returns the time (in seconds) that it took to process the last image. This can be used to determine the stress level of the system.

Arguments

None.

Return Value

Returns the time in floating point seconds that it took to process the image.

Application Notes

Status for the default device (logical unit -1) does not exist, since the default device is a PRO/API concept and not a physical device. If the default device is queried, status for the first active device will be returned. See the Device section in the Concepts and Object Data Types chapter for more details about the default device.



Function Reference

(Device Functions)

GetFrameRateEstimate

Get the device's frame update rate.

```
float GetFrameRateEstimate ( void ) ;
```

Description

Returns the effective display update rate in frames per second. This can be used to measure the stress level of the system.

Arguments

None.

Return Value

Returns the frame rate estimate in frames per second.

Application Notes

Status for the default device (logical unit -1) does not exist, since the default device is a PRO/API concept and not a physical device. If the default device is queried, status for the first active device will be returned. See the Device section in the Concepts and Object Data Types chapter for more details about the default device.



Function Reference

(Device Functions)

GetCurrentFrameCount

Returns the time since the system was initialized.

```
long GetCurrentFrameCount ( void ) ;
```

Description

Returns the current real time frame number since the hardware was initialized. This is the number of 60 or 30 Hertz frames.

Arguments

None.

Return Value

Number of 30 or 60 hertz frames elapsed since the system was initialized.

Application Notes

Status for the default device (logical unit -1) does not exist, since the default device is a PRO/API concept and not a physical device. If the default device is queried, status for the first active device will be returned. See the Device section in the Concepts and Object Data Types chapter for more details about the default device.



Function Reference

(Device Functions)

GetHWPolygonMemorySize

Returns the device's polygon memory size.

```
long GetHWPolygonMemorySize ( void ) ;
```

Description

Returns the polygon memory size (in bytes) of the PRO device. The standard polygon memory is eight megabytes. Each additional polygon memory board (up to 3) adds eight megabytes of polygon storage capacity.

Arguments

None.

Return Value

8388608 for 8 megabytes of polygon memory.
16777216 for 16 megabytes of polygon memory.
25165824 for 24 megabytes of polygon memory.
33554432 for 32 megabytes of polygon memory.

Application Notes

Status for the default device (logical unit -1) does not exist, since the default device is a PRO/API concept and not a physical device. If the default device is queried, status for the first active device will be returned. See the Device section in the Concepts and Object Data Types chapter for more details about the default device.



Viewport Functions

The Viewport class functions are used to configure and control a viewport, a rectangular area on the 2D display for viewing the 3D data. A viewport's attributes define which databases (scenes) will be displayed in the "window" and which blending table to use. The attributes also control functions such as field-of-view, sun illumination, fog, headlights, and color table selection. A viewport must have an attached viewpoint that controls the position and orientation of the viewport.

(Viewport Functions)

PRO_Visport class constructor

Creates an instance of a PRO_Visport object.

```
PRO_Visport ( void ) ;
```

Description

Creates an instance of a PRO_Visport.

Arguments

None.

Return Value

None.

Application Notes

The GetStatus function should be called after object instantiation.



Function Reference

(Viewport Functions)

PRO_Visport class destructor

Destroys an instance of a PRO Viewport object.

```
~PRO_Visport ( void ) ;
```

Description

Destroys a PRO_Visport instance.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetStatus

Gets the status of a PRO Viewport object instantiation.

```
PRO_Status GetStatus ( void ) ;
```

Description

Returns the status of object instantiation.

Arguments

None.

Return Value

PRO_SUCCESS if object creation was successful.

PRO_OUT_OF_DISPLAY_LIST_MEMORY if hardware memory could not be allocated for the entity.

PRO_OUT_OF_HOST_MEMORY if object data members cannot be allocated.

Application Notes

None.



Function Reference

(Viewport Functions)

AttachDefaultBlendTable

Associates the default blend table with a viewport.

```
PRO_Status AttachDefaultBlendTable ( void ) ;
```

Description

Associates the default blend table with a PRO_Viewport object. The default blend table sets the blend ranges so that all database objects are visible. Attachment of the default blend table occurs automatically when a PRO_Viewport is instantiated.

Arguments

None.

Return Value

PRO_SUCCESS if attachment is successful.
PRO_FAILURE otherwise.

Application Notes

None.



Function Reference

(Viewport Functions)

AttachDefaultDatabase

Associates the default database with a viewport.

```
PRO_Status AttachDefaultDatabase ( void ) ;
```

Description

Associates an empty database with a PRO_Viewport object. Attachment of the default database occurs automatically when a PRO_Viewport is instantiated.

Arguments

None.

Return Value

PRO_SUCCESS if attachment is successful.
PRO_FAILURE otherwise.

Application Notes

None.



Function Reference

(Viewport Functions)

AttachBlendTable

Associates a blend table with a viewport.

```
PRO_Status AttachBlendTable ( PRO_LOD_Table *lod_table );
```

Description

Associates a blend table with a PRO_V viewport object.

Arguments

PRO_LOD_Table *lod_table	A pointer to the PRO_LOD_Table object to be associated with the viewport.
--------------------------	---

Return Value

PRO_SUCCESS if attachment is successful.
PRO_FAILURE otherwise.

Application Notes

Only one blend table may be associated with a given viewport at any time. One blend table may be associated with multiple viewports.



Function Reference

(Viewport Functions)

AttachDatabase

Associates a database with a viewport.

```
PRO_Status AttachDatabase ( PRO_Database *database ) ;
```

Description

Associates a database with a PRO_Viewport object.

Arguments

PRO_Database * <i>database</i>	The pointer to the PRO_Database object to be associated with the viewport.
--------------------------------	--

Return Value

PRO_SUCCESS if attachment is successful.
PRO_FAILURE otherwise.

Application Notes

Only one database may be associated with a given viewport at any time. One database may be associated with multiple viewports.



Function Reference

(Viewport Functions)

AttachViewpoint

Associates a viewpoint with a viewport.

```
PRO_Status AttachViewpoint ( PRO_Viewpoint *viewpoint ) ;
```

Description

Associates a viewpoint with a viewport. This provides the mechanism to control what is seen in the viewport, since it is the viewpoint that can be positioned and rotated.

Arguments

PRO_Viewpoint * <i>viewpoint</i>	A pointer to a PRO_Viewpoint object that will be associated with the viewport.
----------------------------------	--

Return Value

PRO_SUCCESS if attachment is successful.
PRO_FAILURE otherwise.

Application Notes

Only one viewpoint may be associated with a given viewport at any time. One viewpoint may be associated with multiple viewports.



Function Reference

(Viewport Functions)

DetachViewpoint

Detaches a viewpoint from a viewport.

```
PRO_Status DetachViewpoint ( void ) ;
```

Description

Deletes a viewpoint's association with a viewport. This function has the effect of making the viewport invisible until another viewpoint is attached to it.

Arguments

PRO_Viewpoint * <i>viewpoint</i>	A pointer to a PRO_Viewpoint object.
----------------------------------	--------------------------------------

Return Value

PRO_SUCCESS if detachment is successful.
PRO_FAILURE otherwise.

Application Notes

None.



Function Reference

(Viewport Functions)

AttachChildViewport

Attaches a viewport to a higher priority viewport.

```
void AttachChildViewport ( PRO_Visport *child_visport ) ;
```

Description

Attaches a viewport to another and reassigns the viewport overlay priorities such that the attached viewport has higher visual priority than the one to which it is being attached. The child viewport, by definition, has higher visual priority and is overlaid on top of its parent.

Arguments

PRO_Visport *child_visport	A pointer to a PRO_Visport object that will be attached as a child of this viewport.
----------------------------	--

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

DetachChildViewport

Detaches a viewport from a higher priority viewport.

```
void DetachChildViewport ( PRO_Viewport *child_viewport ) ;
```

Description

Deletes the association between a viewport and a viewport of higher priority.

Arguments

PRO_Viewport * <i>child_viewport</i>	A pointer to a PRO_Viewport object.
--------------------------------------	-------------------------------------

Return Value

None.

Application Notes

When a viewport is detached from its parent, its priority is reset to zero (lowest priority).



Function Reference

(Viewport Functions)

SetFieldOfView

Defines the field-of-view.

```
void SetFieldOfView ( float horizontal_fov, float vertical_fov ) ;
```

Description

Defines a symmetric field-of-view by specifying the horizontal and vertical angles. The maximum is a 180 degree horizontal and 180 degree vertical field-of-view. The default provides a 40 degree horizontal and 30 degree vertical field-of-view.

Arguments

float <i>horizontal_fov</i>	Horizontal field-of-view angle in degrees. The default value is 40.0 degrees.
float <i>vertical_fov</i>	Vertical field-of-view angle in degrees. The default value is 30.0 degrees.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

SetFieldOfView

Defines an asymmetric field-of-view.

```
void SetFieldOfView ( float left, float right, float top, float bottom ) ;
```

Description

Defines the field-of-view by specifying the horizontal (left and right) and vertical (top and bottom) half angles. The maximum total field-of-view is 180 degrees horizontal and 180 degrees vertical. The default provides a 40 degree horizontal and 30 degree vertical field-of-view. The angles do not have to be symmetrical about their axes.

Arguments

float <i>left</i>	Left field-of-view half angle in degrees. The default value is -20.0 degrees.
float <i>right</i>	Right field-of-view half angle in degrees. The default value is 20.0 degrees.
float <i>top</i>	Top field-of-view half angle in degrees. The default value is 15.0 degrees.
float <i>bottom</i>	Bottom field-of-view half angle in degrees. The default value is -15.0 degrees.

Return Value

None.

Application Notes

The values for the left and bottom angles must be negative, and the values for the right and top angles must be positive.



Function Reference

(Viewport Functions)

SetDisplayExtents

Sets the extents of the viewport in the physical display.

```
void SetDisplayExtents ( float up_left_i, float up_left_j,  
                        float viewport_size_i, float viewport_size_j ) ;
```

Description

Sets the extents of the viewport in terms of the physical display. The upper left-hand corner of the viewport and the viewport's height and width are specified as a normalized fraction of the screen size.

Arguments

float <i>up_left_i</i>	Position of viewport's upper left as percentage of vertical screen space. Value is 0.0 to 1.0. The default value is 0.0.
float <i>up_left_j</i>	Position of viewport's upper left as percentage of horizontal screen space. Value is 0.0 to 1.0. The default value is 0.0.
float <i>viewport_size_i</i>	Vertical size of viewport as a percentage of vertical screen space. The default value is 1.0.
float <i>viewport_size_j</i>	Horizontal size of viewport as a percentage of horizontal screen space. The default value is 1.0.

Return Value

None.

Application Notes

If the display extents require that the viewport size be clipped to the display size, the field of view will be modified as well so that the image is not distorted.



Function Reference

(Viewport Functions)

SetSunIlluminationData

Sets the parameters that control sun illumination effects.

```
void SetSunIlluminationData ( float ambient_contribution, float sun_contribution ) ;
```

Description

Sets the parameters that control the sun illumination effects on the objects in scenes (databases) attached to the viewport. The default provides an ambient lighting of 0.5 and a sun intensity of 0.5. These values should not add to more than 1.0 or be negative.

Arguments

float <i>ambient_contribution</i>	The minimum modulation on face color due to the lighting source. The default value is 0.5.
float <i>sun_contribution</i>	The variation above the ambient lighting of the face color due to the orientation of the polygon to the sun vector. The valid range is 0.0 to 1.0. The default is 0.5.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

SetSunVector

Sets the position of the sun.

```
void SetSunVector ( float sun_x, float sun_y, float sun_z ) ;
```

Description

Sets the vector that points from the origin of the environment to the light source. The vector need not be normalized.

Arguments

float <i>sun_x</i>	The x component of the vector describing the sun's position. The default value is 0.5.
float <i>sun_y</i>	The y component of the vector describing the sun's position. The default value is 0.5.
float <i>sun_z</i>	The z component of the vector describing the sun's position. The default value is 0.5.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

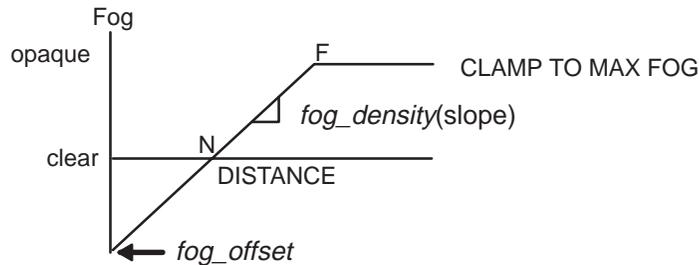
SetFogParameters

Sets the parameters that control fog effects.

```
void SetFogParameters ( float fog_attenuation, float ambient_fog,  
                        float fog_offset, float fog_density ) ;
```

Description

Sets the parameters that control fog effects for the viewport.



$$fog_density = \frac{1}{F - N}$$

$$fog_offset = -N \times fog_density$$

Arguments

- | | |
|------------------------------|--|
| float <i>fog_attenuation</i> | Defines the fog density used by the lobe light model to decrease the headlight effects as a function of range. It is a measure of the fog particle light interference. If the color of the light on the fog (see SetLightFogColor()) is zero, fog_attenuation has no effect.
The valid range is 0.0 to 1.0. The default is 0.0. |
| float <i>ambient_fog</i> | The amount of light applied to the fog.
The valid range is 0.0 to 1.0. The default value is 0.6. |
| float <i>fog_offset</i> | Fog value at the viewpoint. The default value is -0.333. |
| float <i>fog_density</i> | Rate at which fog increases as a function of range.
The valid range is 0.0 to 1.0. The default value is 0.0. |



Function Reference

Return Value

None.

Application Notes

See also `SetFogRanges`.



Function Reference

(Viewport Functions)

SetFogRanges

Sets the near and far ranges for fog effects.

```
void SetFogRanges ( float near_distance, float far_distance ) ;
```

Description

Sets the near and far ranges for fog effects.

Arguments

float <i>near_distance</i>	The range from the eyepoint where fogging starts.
float <i>far_distance</i>	The range from the eyepoint where the data is fully fogged.

Return Value

None.

Application Notes

This function performs the necessary calculations that would otherwise be required by the SetFogParameters function. It is meant to provide a more intuitive interface for fog parameter definition.



Function Reference

(Viewport Functions)

SetFogColor

Sets the color of the fog.

```
void SetFogColor ( long red, long green, float blue ) ;
```

Description

Sets the fog color to the specified RGB value.

Arguments

long <i>red</i>	The intensity of the red component of fog color. The valid range is 0 through 255; the default is 240.
long <i>green</i>	The intensity of the green component of fog color. The valid range is 0 through 255; the default is 230.
long <i>blue</i>	The intensity of the blue component of fog color. The valid range is 0 through 255; the default is 220.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

SetLightFogColor

Sets the color of the light lobe on fog.

```
void SetLightFogColor ( long light_fog_color ) ;
```

Description

Sets the color of the light on the fog. The color value is specified as follows:

color value 0	OFF
color value 1	BLUE
color value 2	GREEN
color value 3	CYAN
color value 4	RED
color value 5	PURPLE
color value 6	YELLOW
color value 7	WHITE

Arguments

long light_fog_color The color of the light on the fog. Valid range is 0 through 7. The default value is 5.

Return Value

None.

Application Notes

The amount of colored light visible on the fog is determined by the `fog_attenuation`, which is set using function `SetFogParameters`. If the `fog_attenuation` is zero, no colored light will be visible on the fog.



Function Reference

(Viewport Functions)

SetLightSourceColor

Sets the color of the light lobe on polygons.

```
void SetLightSourceColor ( long light_color );
```

Description

Sets the color of the light lobe on a polygon. The color value is specified as follows:

color value 0	OFF
color value 1	BLUE
color value 2	GREEN
color value 3	CYAN
color value 4	RED
color value 5	PURPLE
color value 6	YELLOW
color value 7	WHITE

Arguments

long *light_color* The color of the light on polygons. Valid range is 0 through 7. The default value is 0.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

SetLightSourceParameters

Sets the parameters that control light source illumination effects.

```
void SetLightSourceParameters ( float lt_start_range, float light_extent ) ;
```

Description

Sets the parameters that control the illuminating effects of the light source.

Arguments

float <i>lt_start_range</i>	The range to the point where the light source begins to be visible. The default value is 6.0.
float <i>light_extent</i>	The distance over which the light source will illuminate the scene. The intensity of the light source will be held constant over this distance. The default value is 1000.0.

Return Value

None.

Application Notes

The light source color must be set to a value other than 'OFF' (using function SetLightSourceColor) to see the light on polygons.



Function Reference

(Viewport Functions)

SetLightSourceStartRange

Sets the range to light source.

```
void SetLightSourceStartRange ( float lt_start_range ) ;
```

Description

Sets the range to the point where the light source begins to be visible.

Arguments

float <i>lt_start_range</i>	The range to the point where the light source begins to be visible. The default value is 6.0.
-----------------------------	---

Return Value

None.

Application Notes

The light source color must be set to a value other than 'OFF' (using function SetLightSourceColor) to see the light on polygons.



Function Reference

(Viewport Functions)

SetLightSourceExtent

Sets the distance which light source illuminates.

```
void SetLightSourceExtent ( float light_extent ) ;
```

Description

Sets the distance over which the light source will illuminate the scene. The intensity of the light source will be held constant over this distance.

Arguments

float <i>light_extent</i>	The distance over which the light source will illuminate the scene. The intensity of the light source will be held constant over this distance. The default value is 1000.0.
---------------------------	--

Return Value

None.

Application Notes

The light source color must be set to a value other than 'OFF' (using function SetLightSourceColor) to see the light on polygons.



Function Reference

(Viewport Functions)

SetLobeParameters

Sets the parameters that define the light lobe.

```
void SetLobeParameters ( float lobe_position_i, float lobe_position_j,  
                        float lobe_size_i, float lobe_size_j ) ;
```

Description

Sets the parameters that describe the position and size of a light source. The parameters describe the center and size of an ellipse in terms of a fraction of viewport extents relative to the viewport's upper-left corner.

Arguments

float <i>lobe_position_i</i>	Vertical position of the center of the light as a percentage of the viewport display. Valid values are 0.0 to 1.0; the default is 0.5.
float <i>lobe_position_j</i>	Horizontal position of the center of the light as a percentage of the viewport display. Valid values are 0.0 to 1.0; the default is 0.5.
float <i>lobe_size_i</i>	Height of the light as a percentage of the viewport display Valid values are 0.0 to 1.0; the default is 0.05.
float <i>lobe_size_j</i>	Width of the light as a percentage of the viewport display Valid values are 0.0 to 1.0; the default is 0.05.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

SetViewportOff

Deactivates a viewport.

```
void SetViewportOff ( void ) ;
```

Description

Sets a viewport to inactive so it is not processed by the hardware. By default, a viewport is active.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

SetViewportOn

Activates a viewport.

```
void SetViewportOn ( void ) ;
```

Description

Sets a viewport to active so it is processed by the hardware. This is the default state for a viewport. A viewport cannot be active unless it has a valid database and a valid viewpoint attached.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

SetDisplaySelect

Selects the display.

```
void SetDisplaySelect ( short selected_display );
```

Description

Selects the physical display to which the viewport contents will be output. All viewports default to display zero.

Arguments

short *selected_display* The output display. Valid settings are zero and one.

Return Value

None.

Application Notes

This selection is meaningful only if the device to which the viewport is attached has been set in a dual display mode.



Function Reference

(Viewport Functions)

SetInvertFlags

Sets the mirror flags.

```
void SetInvertFlags ( long invert_horiz, long invert_vert ) ;
```

Description

Indicates whether a viewport display should be mirrored horizontally and/or vertically.

Arguments

long *invert_horiz*

When set to 1, indicates that the viewport display should be mirrored horizontally.

long *invert_vert*

When set, indicates that the viewport display should be mirrored vertically.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetViewpoint

Gets the associated viewpoint.

```
PRO_Viewpoint *GetViewpoint ( void ) ;
```

Description

Returns the viewpoint associated with the viewport.

Arguments

None.

Return Value

A pointer to the PRO_Viewpoint object that is attached to the viewport.

Application Notes

None.



Function Reference

(Viewport Functions)

GetAttachedDevice

Gets the device to which viewport is attached.

```
PRO_Device *GetAttachedDevice ( void ) ;
```

Description

Gets the PRO_Device object to which the viewport has been attached. If the viewport has not been attached to a device by the user application, this function will return the default device.

Arguments

None.

Return Value

A pointer to the PRO_Device object to which the viewport is attached.

Application Notes

None.



Function Reference

(Viewport Functions)

GetFieldOfView

Gets the symmetric field-of-view angles.

```
void GetFieldOfView ( float &horizontal_fov, float &vertical_fov ) ;
```

Description

Gets the horizontal and vertical field-of-view angles which have been defined for the viewport.

Arguments

float &horizontal_fov	Horizontal field-of-view in degrees.
float &vertical_fov	Vertical field-of-view in degrees.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetFieldOfView

Gets the asymmetric field-of-view angles.

```
void GetFieldOfView ( float &left, float &right, float &top, float &bottom ) ;
```

Description

Gets the horizontal (left and right) and vertical (top and bottom) half angles which define the field-of-view for the viewport.

Arguments

float &left	Left field-of-view half angle in degrees.
float &right	Right field-of-view half angle in degrees.
float &top	Top field-of-view half angle in degrees.
float &bottom	Bottom field-of-view half angle in degrees.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetDisplayExtents

Gets the extents of the viewport in the physical display.

```
void GetDisplayExtents ( float &up_left_i, float &up_left_j,  
                        float &viewport_size_i, float &viewport_size_j ) ;
```

Description

Gets the extents of the viewport in terms of the physical display. The upper left-hand corner of the viewport and the viewport's height and width are specified as a fraction of the screen size.

Arguments

float &up_left_i	Position of the viewport's upper left corner as percentage of vertical screen space.
float &up_left_j	Position of the viewport's upper left corner as percentage of horizontal screen space.
float &viewport_size_i	Vertical size of the viewport as a percentage of vertical screen space.
float &viewport_size_j	Horizontal size of the viewport as a percentage of horizontal screen space.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetSunIlluminationData

Gets the parameters that control sun illumination effects.

```
void GetSunIlluminationData ( float &ambient_contribution, float &sun_contribution ) ;
```

Description

Gets the parameters that control the sun illumination effects on the objects in scenes (database) attached to the viewport.

Arguments

float &ambient_contribution	The minimum modulation on face color due to the lighting source.
float &sun_contribution	The variation above the ambient lighting of the face color due to the orientation of the polygon to the sun vector.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetSunVector

Gets the position of the sun.

```
void GetSunVector ( float &sun_x, float &sun_y, float &sun_z ) ;
```

Description

Gets the vector that points from the origin of the environment to the light source.

Arguments

float &sun_x	The x component of the vector describing the sun's position.
float &sun_y	The y component of the vector describing the sun's position.
float &sun_z	The z component of the vector describing the sun's position.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetFogParameters

Gets the parameters that control fog effects.

```
void GetFogParameters ( float &fog_attenuation, float &ambient_fog,  
                        float &fog_offset, float &fog_density ) ;
```

Description

Gets the parameters that control fog effects for the viewport.

Arguments

float &fog_attenuation	The fog density used by the lobe light model to decrease the headlight effects as a function of range.
float &ambient_fog	The amount of light applied to the fog.
float &fog_offset	Fog value at the viewpoint.
float &fog_density	Rate at which fog increases as a function of range.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetFogRanges

Gets the near and far ranges for fog effects.

```
void GetFogRanges ( float &near_distance, float &far_distance ) ;
```

Description

Gets the range from the eyepoint to the start of fog effects and the range to the end of fog effects.

Arguments

float &*near_distance*

The near range of fog effects.

float &*far_distance*

The far range of fog effects.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetFogColor

Gets the color of the fog.

```
void GetFogColor ( long &red, long &green, long &blue ) ;
```

Description

Gets the RGB value of the fog color.

Arguments

long &red	The intensity of the red component of fog color.
long &green	The intensity of the green component of fog color.
long &blue	The intensity of the blue component of fog color.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetLightFogColor

Gets the color of the light lobe on fog.

```
void GetLightFogColor ( long &light_fog_color ) ;
```

Description

Gets the color of the light on the fog. The color value is specified as follows:

color value 0	OFF
color value 1	BLUE
color value 2	GREEN
color value 3	CYAN
color value 4	RED
color value 5	PURPLE
color value 6	YELLOW
color value 7	WHITE

Arguments

long &*light_fog_color* The color of the light on the fog.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetLightSourceColor

Gets the color of the light lobe on polygons.

```
void GetLightSourceColor ( long &light_color ) ;
```

Description

Gets the color of the light lobe on a polygon. The color value is specified as follows:

color value 0	OFF
color value 1	BLUE
color value 2	GREEN
color value 3	CYAN
color value 4	RED
color value 5	PURPLE
color value 6	YELLOW
color value 7	WHITE

Arguments

long &*light_color* The color of the light on polygons.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetLightSourceParameters

Gets the parameters that control light source illumination effects.

```
void GetLightSourceParameters ( float &lt_start_range, float &light_extent ) ;
```

Description

Gets the parameters that control the illuminating effects of the light source.

Arguments

float <_start_range	The range to the point where the light source begins to be visible.
float &light_extent	The distance over which the light source will illuminate the scene. The intensity of the light source will be held constant over this distance.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetLobeParameters

Gets the parameters that define the light lobe.

```
void GetLobeParameters ( float &lobe_position_i, float &lobe_position_j,  
                        float &lobe_size_i, float &lobe_size_j ) ;
```

Description

Gets the parameters that describe the position and size of a light source. The parameters describe the center and size of an ellipse in terms of a fraction of viewport extents.

Arguments

float &lobe_position_i	Vertical position of the center of the light as a percentage of the viewport display.
float &lobe_position_j	Horizontal position of the center of the light as a percentage of the viewport display.
float &lobe_size_i	Height of the light as a percentage of the viewport display.
float &lobe_size_j	Width of the light as a percentage of the viewport display.

Return Value

None.

Application Notes

None.



Function Reference

(Viewport Functions)

GetPriority

Gets the viewport's priority level.

```
long GetPriority ( void ) ;
```

Description

Returns the viewport's priority level. The priority defines how the viewport will be drawn in relation to its parent or child viewports. Higher priority viewports are displayed on top of those with lower priority.

Arguments

None.

Return Value

The priority level of the viewport.

Application Notes

None.



Database Functions

The Database Class functions are used to manage graphical objects. A Database is a collection of models (root nodes) that can be culled and rendered.

(Database Functions)

PRO_Database class constructor

Creates an instance of a PRO Database.

```
PRO_Database ( long number_of_children ) ;
```

Description

Creates an instance of a PRO_Database.

Arguments

long <i>number_of_children</i>	The number of root nodes to be managed. The default is the maximum number of models specified in the call to PRO_Configure_Memory(), or 256 if the default memory configuration is used.
--------------------------------	--

Return Value

None.

Application Notes

The GetStatus function should be called after object instantiation.



Function Reference

(Database Functions)

PRO_Database class destructor

Destroys an instance of a PRO Database.

```
~PRO_Database ( void ) ;
```

Description

Destroys an instance of a PRO_Database.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Database Functions)

GetStatus

Gets the status of a PRO Database object instantiation.

```
PRO_Status GetStatus ( void ) ;
```

Description

Returns the status of object instantiation.

Arguments

None.

Return Value

PRO_SUCCESS if object creation was successful

PRO_OUT_OF_DISPLAY_LIST_MEMORY if hardware memory could not be allocated for the entity.

PRO_OUT_OF_HOST_MEMORY if object data members cannot be allocated.

Application Notes

None.



Function Reference

(Database Functions)

AttachModel

Attaches a model to a PRO Database.

```
void AttachModel ( PRO_Model *model_ptr ) ;
```

Description

Attaches a model to a PRO_Database. A model must be attached to a database to be a candidate for rendering.

Arguments

PRO_Model * <i>model_ptr</i>	A pointer to a PRO_Model object.
------------------------------	----------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Database Functions)

DetachModel

Detaches a model from a PRO Database.

```
void DetachModel ( PRO_Model *model_ptr );
```

Description

Removes a model from the models managed by a PRO_Database.

Arguments

PRO_Model * <i>model_ptr</i>	A pointer to a PRO_Model object.
------------------------------	----------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Database Functions)

ActivateAllModels

Activates all models attached to the scene.

```
void ActivateAllModels ( void ) ;
```

Description

Activates all models attached to the scene.

Arguments

None.

Return Value

None.

Application Notes

The same result can be achieved by calling `model->Activate()` for each model attached to the database.



Function Reference

(Database Functions)

DeactivateAllModels

Deactivates all models attached to the scene.

```
void DeactivateAllModels ( void ) ;
```

Description

Deactivates all models attached to the scene.

Arguments

None.

Return Value

None.

Application Notes

The same result can be achieved by calling `model->Deactivate()` for each model attached to the database.



Function Reference

(Database Functions)

HasActiveModels

Indicates whether any attached models are active.

```
long HasActiveModels ( void ) ;
```

Description

Indicates whether any of the models attached to the database are active.

Arguments

None.

Return Value

One if any models are active.
Zero if no models are active.

Application Notes

None.



Viewpoint Functions

The Viewpoint class functions handle the positioning and orientation of a viewpoint attached to a Viewport object. The viewpoint defines the scene that is rendered as well as the viewing position and orientation.

(Viewpoint Functions)

PRO_Viewpoint class constructor

Creates an instance of a PRO Viewpoint.

```
PRO_Viewpoint ( void ) ;
```

Description

Creates an instance of a PRO_Viewpoint.

Arguments

None.

Return Value

None.

Application Notes

The GetStatus function should be called after object instantiation.



Function Reference

(Viewpoint Functions)

PRO_Viewpoint class destructor

Destroys an instance of a PRO Viewpoint.

```
~PRO_Viewpoint ( void ) ;
```

Description

Destroys an instance of a PRO_Viewpoint.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

GetStatus

Gets the status of a PRO Viewpoint object instantiation.

```
PRO_Status GetStatus ( void ) ;
```

Description

Returns the status of object instantiation.

Arguments

None.

Return Value

PRO_SUCCESS if object creation was successful.

PRO_OUT_OF_DISPLAY_LIST_MEMORY if hardware memory could not be allocated for the entity

PRO_OUT_OF_HOST_MEMORY if object data members cannot be allocated.

Application Notes

None.



Function Reference

(Viewpoint Functions)

SetPosition

Sets the position of the viewpoint.

```
void SetPosition ( double x, double y, double z ) ;
```

Description

Sets the viewpoint position to the Cartesian coordinates x,y,z.
The default position is (0.0, 0.0, 0.0).

Arguments

double x	The x coordinate of the viewpoint position.
double y	The y coordinate of the viewpoint position.
double z	The z coordinate of the viewpoint position.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

SetPosition

Sets the position of the viewpoint.

```
void SetPosition ( float x, float y, float z ) ;
```

Description

Sets the viewpoint position to the Cartesian coordinates x,y,z.
The default position is (0.0, 0.0, 0.0).

Arguments

float x	The x coordinate of the viewpoint position.
float y	The y coordinate of the viewpoint position.
float z	The z coordinate of the viewpoint position.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

SetOrientation

Sets the orientation of the viewpoint.

```
void SetOrientation ( double roll, double pitch, double yaw ) ;
```

Description

Rotates the viewpoint according to the angles *roll*, *pitch* and *yaw*. The default values of these angles are all zero, which makes the viewpoint look in the direction of the positive x axis.

Arguments

double <i>roll</i>	The rotation about the x axis, in degrees.
double <i>pitch</i>	The rotation about the y axis, in degrees.
double <i>yaw</i>	The rotation about the z axis, in degrees.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

SetOrientation

Sets the orientation of the viewpoint.

```
void SetOrientation ( float roll, float pitch, float yaw ) ;
```

Description

Rotates the viewpoint according to the angles *roll*, *pitch* and *yaw*. The default values of these angles are all zero, which makes the viewpoint look in the direction of the positive x axis.

Arguments

float <i>roll</i>	The rotation about the x axis, in degrees.
float <i>pitch</i>	The rotation about the y axis, in degrees.
float <i>yaw</i>	The rotation about the z axis, in degrees.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

SetOrientation

Sets the orientation of the viewpoint.

```
void SetOrientation ( float *in_matrix ) ;
```

Description

Rotates the viewpoint by the 3x3 matrix. The ordering of the matrix elements is as follows:

0	1	2
Forward .X	Forward .Y	Forward .Z
3	4	5
Left .X	Left .Y	Left .Z
6	7	8
Up .X	Up .Y	Up .Z

Arguments

float *in_matrix

A pointer to the 3x3 matrix containing the rotations to be applied to the viewpoint.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

GetPosition

Gets the position of a viewpoint.

```
void GetPosition ( double *x, double *y, double *z ) ;
```

Description

Gets the viewpoint position.

Arguments

double *x	Pointer to the x coordinate of the viewpoint position.
double *y	Pointer to the y coordinate of the viewpoint position.
double *z	Pointer to the z coordinate of the viewpoint position.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

GetPosition

Gets the position of a viewpoint.

```
void GetPosition ( double &x, double &y, double &z ) ;
```

Description

Gets the viewpoint position.

Arguments

double &x	The x coordinate of the viewpoint position.
double &y	The y coordinate of the viewpoint position.
double &z	The z coordinate of the viewpoint position.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

GetOrientation

Gets the orientation of a viewpoint.

```
void GetOrientation ( double *roll, double *pitch, double *yaw ) ;
```

Description

Gets the viewpoint orientation in degrees.

Arguments

double *roll	Pointer to the roll angle of the viewpoint orientation.
double *pitch	Pointer to the pitch angle of the viewpoint orientation.
double *yaw	Pointer to the yaw angle of the viewpoint orientation.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

GetOrientation

Gets the orientation of a viewpoint.

```
void GetOrientation ( double &roll, double &pitch, double &yaw ) ;
```

Description

Gets the viewpoint orientation in degrees.

Arguments

double &roll	The roll angle of the viewpoint orientation.
double &pitch	The pitch angle of the viewpoint orientation.
double &yaw	The yaw angle of the viewpoint orientation.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

GetLocalForward

Gets the forward vector in viewpoint coordinates.

```
void GetLocalForward ( float *x, float *y, float *z ) ;
```

Description

Gets the forward (boresight) vector in local viewpoint coordinates.

Arguments

float *x	Pointer to the x component of the forward vector.
float *y	Pointer to the y component of the forward vector.
float *z	Pointer to the z component of the forward vector.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

GetLocalLeft

Gets the left vector in viewpoint coordinates.

```
void GetLocalLeft ( float *x, float *y, float *z ) ;
```

Description

Gets the left-right vector in local viewpoint coordinates.

Arguments

float *x	Pointer to the x component of the left-right vector.
float *y	Pointer to the y component of the left-right vector.
float *z	Pointer to the z component of the left-right vector.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

GetLocalUp

Gets the up vector in viewpoint coordinates.

```
void GetLocalUp ( float *x, float *y, float *z ) ;
```

Description

Gets the top-bottom vector in local viewpoint coordinates.

Arguments

float *x	Pointer to the x component of the up-down vector.
float *y	Pointer to the y component of the up-down vector.
float *z	Pointer to the z component of the up-down vector.

Return Value

None.

Application Notes

None.



Function Reference

(Viewpoint Functions)

AttachViewpoint

Attaches a child viewpoint.

```
void AttachViewpoint ( PRO_Viewpoint *child_viewpoint ) ;
```

Description

Attaches viewpoint as a child of another.

Arguments

PRO_Viewpoint **child_viewpoint* Pointer to be set to a child viewpoint.

Return Value

None.

Application Notes

When a viewpoint is attached as a child of another, its position and orientation are interpreted as being relative to the parent's, to any degree of nesting.



Function Reference

(Viewpoint Functions)

DetachViewpointFromParent

Detaches a child viewpoint from its parent.

```
void DetachViewpointFromParent ( void ) ;
```

Description

Detaches viewpoint from its parent viewpoint.

Arguments

None.

Return Value

None.

Application Notes

None.



Blend Table Functions

A Blend Table, or Level-of-Detail (LOD) Table, contains parameters that specify how groups of objects are blended in and out of the rendered scene at various LODs. Up to four LODs are supported. By default, blending parameters are specified as a start range and a delete range for each LOD. Alternatively, blend parameters may be specified as angles.

Objects may be grouped by feature type. All objects with the same feature type will be blended according to the same set of parameters. By default, all objects have a feature type of zero.

(Blend Table Functions)

PRO_LOD_Table class constructor

Creates an instance of a PRO LOD Table.

```
PRO_LOD_Table ( long num_feature_types );
```

Description

Creates an instance of a PRO_LOD_Table.

Arguments

long <i>num_feature_types</i>	The number of blending types with which objects may be associated. The default value is 128, which is the maximum supported by the hardware.
-------------------------------	--

Return Value

None.

Application Notes

The GetStatus function should be called after object instantiation.



Function Reference

(Blend Table Functions)

PRO_LOD_Table class destructor

Destroys an instance of a PRO LOD Table.

```
~PRO_LOD_Table ( void ) ;
```

Description

Destroys an instance of a PRO_LOD_Table.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Blend Table Functions)

GetStatus

Gets the status of a PRO LOD Table object instantiation.

```
PRO_Status GetStatus ( void ) ;
```

Description

Returns the status of object instantiation.

Arguments

None.

Return Value

PRO_SUCCESS if object creation was successful.

PRO_OUT_OF_DISPLAY_LIST_MEMORY if hardware memory could not be allocated for the entity.

PRO_OUT_OF_HOST_MEMORY if object data members cannot be allocated.

Application Notes

None.



Function Reference

(Blend Table Functions)

SetRangeBlendParameters

Sets blend parameters based on range.

```
PRO_Status SetRangeBlendParameters ( long feature_type, long lod_number,  
float start_range, float delete_range ) ;
```

Description

Sets the blending parameters for a specified feature type at a specified Level-of-Detail (LOD). Objects are transitioned in or out of the scene based on the specified start and delete ranges.

Arguments

long <i>feature_type</i>	Feature type. Value is zero through (num_feature_types – 1).
long <i>lod_number</i>	Level-of-Detail. Value is zero through 3. The most detailed version of the object/model is normally associated with LOD zero, while the coarsest is normally associated with LOD 3.
float <i>start_range</i>	Range at which object at this LOD will begin to transition into the scene.
float <i>delete_range</i>	Range at which object at this LOD will be deleted from the scene.

Return Value

PRO_SUCCESS if successful.
PRO_FAILURE if feature type is out of range.

Application Notes

None.



Function Reference

(Blend Table Functions)

SetSizeBlendParameters

Sets blend parameters based on subtended angle.

```
PRO_Status SetSizeBlendParameters ( long feature_type, long lod_number,  
float start_angle, float delete_angle );
```

Description

Sets the blending parameters for a specified feature type at a specified Level-of-Detail (LOD). Objects are transitioned in or out of the scene based on the specified start and delete subtended angles.

Arguments

long <i>feature_type</i>	Feature type. Value is zero to (num_feature_types – 1).
long <i>lod_number</i>	Level-of-Detail. Value is zero through 3. The highest detail version of the object/model is normally associated with LOD zero, while the coarsest is associated with LOD index 3.
float <i>start_angle</i>	Subtended angle in degrees at which object at this LOD will begin to transition into the scene.
float <i>delete_angle</i>	Subtended angle in degrees at which object at this LOD will be deleted from the scene (blended out).

Return Value

PRO_SUCCESS if successful.
PRO_FAILURE if feature type is out of range.

Application Notes

None.



Function Reference

(Blend Table Functions)

GetRangeBlendParameters

Gets blend parameters based on range.

```
void GetRangeBlendParameters ( long feature_type, long lod_number,  
                             float &start_range, float &delete_range ) ;
```

Description

Gets the blending parameters for a specified feature type at a specified Level-of-Detail (LOD).

Arguments

long <i>feature_type</i>	Feature type. Value is zero through (num_feature_types – 1).
long <i>lod_number</i>	Level-of-Detail. Value is zero through 3.
float & <i>start_range</i>	Range at which object at this LOD will be transitioned into the scene.
float & <i>delete_range</i>	Range at which object at this LOD will be transitioned out of the scene.

Return Value

None.

Application Notes

None.



Function Reference

(Blend Table Functions)

GetSizeBlendParameters

Gets blend parameters based on subtended angle.

```
void GetSizeBlendParameters ( long feature_type, long lod_number,  
                             float &start_angle, float &delete_angle ) ;
```

Description

Gets the blending parameters for a specified feature type at a specified Level-of-Detail (LOD).

Arguments

long <i>feature_type</i>	Feature type. Value is zero through (num_feature_types – 1).
long <i>lod_number</i>	Level-of-Detail. Value is zero through 3.
float & <i>start_angle</i>	Subtended angle in degrees at which object at this LOD will be transitioned into the scene.
float & <i>delete_angle</i>	Subtended angle in degrees at which object at this LOD will be transitioned out of the scene.

Return Value

None.

Application Notes

None.



Function Reference

(Blend Table Functions)

GetAllocatedTableSize

Returns the allocated number of entries.

```
long GetAllocatedTableSize ( void ) ;
```

Description

Returns the allocated number of entries in the blend table.

Arguments

None.

Return Value

The allocated number of entries in the blend table.

Application Notes

None.



Color Table Functions

The Color Table class functions are used to establish a table of Red-Green-Blue (RGB) color values that can be referenced by polygon objects.

(Color Table Functions)

PRO_Color_Table class constructor

Creates an instance of a PRO Color Table.

```
PRO_Color_Table ( long table_size ) ;
```

Description

Creates a PRO_Color_Table instance.

Arguments

long <i>table_size</i>	An optional argument specifying the number of entries in the color table. The default table size is 4096, which is the maximum size supported by the hardware.
------------------------	--

Return Value

None.

Application Notes

The GetStatus function should be called after object instantiation.



Function Reference

(Color Table Functions)

PRO_Color_Table class destructor

Destroys an instance of a PRO Color Table.

```
~PRO_Color_Table ( void ) ;
```

Description

Destroys an instance of a PRO_Color_Table.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Color Table Functions)

GetStatus

Gets the status of a PRO Color Table object instantiation.

```
PRO_Status GetStatus ( void ) ;
```

Description

Returns the status of object instantiation.

Arguments

None.

Return Value

PRO_SUCCESS if object creation was successful.

PRO_OUT_OF_DISPLAY_LIST_MEMORY if hardware memory could not be allocated for the entity.

PRO_OUT_OF_HOST_MEMORY if object data members cannot be allocated.

Application Notes

None.



Function Reference

(Color Table Functions)

SetColor

Sets the RGB value of a table entry.

```
PRO_Status SetColor ( long index, long red, long green, long blue ) ;
```

Description

Sets an entry in the PRO_Color_Table to the specified RGB value.

Arguments

long <i>index</i>	The index into the color table. Value is zero through (<i>table_size</i> – 1).
long <i>red</i>	The intensity of the red component of the color. Valid range is zero through 255.
long <i>green</i>	The intensity of the green component of the color. Valid range is zero through 255.
long <i>blue</i>	The intensity of the blue component of the color. Valid range is zero through 255.

Return Value

Returns PRO_SUCCESS if successful.
PRO_FAILURE if an invalid index is specified.

Application Notes

None.



Function Reference

(Color Table Functions)

SetColor

Sets the RGB value of a table entry.

```
PRO_Status SetColor ( long index, float red, float green, float blue ) ;
```

Description

Sets an entry in the PRO_Color_Table to the specified RGB value.

Arguments

long <i>index</i>	The index into the color table. Value is zero through (table_size – 1).
float <i>red</i>	The intensity of the red component of the color. Valid range is 0.0 through 1.0.
float <i>green</i>	The intensity of the green component of the color. Valid range is 0.0 through 1.0.
float <i>blue</i>	The intensity of the blue component of the color. Valid range is 0.0 through 1.0.

Return Value

Returns PRO_SUCCESS if successful.
PRO_FAILURE if an invalid index is specified.

Application Notes

None.



Function Reference

(Color Table Functions)

GetAllocatedTableSize

Returns the allocated number of entries.

```
long GetAllocatedTableSize ( void ) ;
```

Description

Returns the allocated number of entries in the color table.

Arguments

None.

Return Value

The allocated number of entries in the color table.

Application Notes

None.



Model Functions

The Model class functions allow manipulation of a database object. The object may be defined by a pre-formatted model file that contains all of the object's geometry and attributes, or the object may be built up "at run-time" by adding culling nodes and polygons to it. A model may have sub-parts, or articulated parts that may be positioned and rotated relative to the main model. A model or its articulated parts may have multiple representations (versions, or animation frames) which can be cycled by the application.

(Model Functions)

PRO_Model class constructor

Creates an instance of a PRO Model.

```
PRO_Model ( char *preformatted_filename,  
            char *texture_path,  
            char *external_reference_path ) ;
```

Description

Creates an instance of a PRO_Model by reading a pre-formatted model from disk. This requires the model to be pre-formatted by the PRO-1000 offline formatter or by using the PRO/API offline formatter mode.

Arguments

char *preformatted_filename	The name of file (full path but no extension) containing model data to be loaded.
char *texture_path	The path to texture maps. By default, the directory containing the model is used.
char external_reference_path	The path to pre-formatted external reference files. By default, the directory containing the model is used.



Function Reference

Return Value

None.

Application Notes

The GetModelStatus function should be called after object instantiation.



Function Reference

(Model Functions)

PRO_Model class constructor

Creates an instance of a PRO_Model.

```
PRO_Model ( long static_model ) ;
```

Description

Creates an instance of a PRO_Model to which culling nodes and polygons will be attached. This is the general purpose run-time model constructor, for models to be formatted and downloaded to the hardware at run-time.

Arguments

<i>long static_model</i>	Indicates whether model will have an associated matrix, or coordinate set. Valid values are zero (dynamic model) or one (static model). The default value is zero.
--------------------------	--

Return Value

None.

Application Notes

The GetModelStatus function should be called after object instantiation.

If the model was created as a static model, all matrix-related operations (e.g, SetPosition, SetOrientation , GetLocalForward) are invalid.



Function Reference

(Model Functions)

PRO_Model class constructor

Creates an instance of a PRO_Model.

```
PRO_Model ( PRO_Instance_Culling_Node *instanced_geometry,  
            PRO_Matrix *user_matrix ) ;
```

Description

Creates an instance of a PRO_Model which references shared geometry ONLY. Optionally, a user-defined matrix may be specified, providing the mechanism for a single matrix to be shared across models.

Arguments

PRO_Instance_Culling_Node *instanced_geometry	Pointer to the instance culling node that holds the model geometry.
PRO_Matrix *user_matrix	A pointer to the shared matrix. By default, a shared matrix is not provided.

Return Value

None.

Application Notes

Models created in this manner may not have ANY other geometry objects attached to it. If a model is required which contains an instance reference and other geometry objects, the model should be created using the general purpose run-time constructor.

The GetModelStatus function should be called after object instantiation.



Function Reference

(Model Functions)

PRO_Model class constructor

Creates an instance of a PRO_Model.

```
PRO_Model ( PRO_Model *model_to_copy ) ;
```

Description

Creates an instance of a PRO_Model which is a copy of another PRO_Model object. The two models share the same geometry, but have separate matrices and animation sequences to manipulate the shared geometry independently.

Arguments

PRO_Model * <i>model_to_copy</i>	Pointer to a PRO_Model object.
----------------------------------	--------------------------------

Return Value

None.

Application Notes

Additional culling nodes and polygons cannot be added to a PRO_Model which has been instantiated as a copy.

The GetModelStatus function should be called after object instantiation.



Function Reference

(Model Functions)

PRO_Model class destructor

Destroys an instance of a PRO Model.

```
~PRO_Model ( void ) ;
```

Description

Destroys an instance of a PRO_Model.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetModelStatus

Returns the status of a model.

```
PRO_Status GetModelStatus ( void ) ;
```

Description

Returns the status of model load / creation. This method should be called after each model is instantiated to ensure that other PRO_Model operations will be valid.

Arguments

None.

Return Value

PRO_SUCCESS if the model was instantiated successfully.
PRO_OUT_OF_HOST_MEMORY if memory allocation operations failed.
PRO_FAILURE if a general error condition was encountered.

Application Notes

None.



Function Reference

(Model Functions)

SetPosition

Sets the model's position.

```
void SetPosition ( double x, double y, double z, long index ) ;
```

Description

Sets the model position to the Cartesian coordinates x,y,z. The default position is (0.0, 0.0, 0.0). The index specifies whether to position the model itself or one of its sub-parts (articulated parts). For an articulated part, the position is relative to its parent's position.

Arguments

double x	The x coordinate of the model position.
double y	The y coordinate of the model position.
double z	The z coordinate of the model position.
long index	The sub-part index to be positioned. The valid range is zero through GetNumberOfArticulatedParts(). The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

SetPosition

Sets the model's position.

```
void SetPosition ( float x, float y, float z, long index ) ;
```

Description

Sets the model position to the Cartesian coordinates x,y,z. The default position is (0.0, 0.0, 0.0). The index specifies whether to position the model itself or one of its sub-parts (articulated parts). For an articulated part, the position is relative to its parent's position.

Arguments

float x	The x coordinate of the model position.
float y	The y coordinate of the model position.
float z	The z coordinate of the model position.
long index	The sub-part index to be positioned. The valid range is zero through GetNumberOfArticulatedParts(). The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes



Function Reference

(Model Functions)

SetPosition

Sets an articulated part's position.

```
void SetPosition ( double x, double y, double z, char *part_name ) ;
```

Description

Sets the articulated part position to the Cartesian coordinates x,y,z. The default position is (0.0, 0.0, 0.0). The position is relative to the parent position.

Arguments

double x	The x coordinate of the articulated part position.
double y	The y coordinate of the articulated part position.
double z	The z coordinate of the articulated part position.
char *part_name	The name of the part to be positioned.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

SetPosition

Sets an articulated part's position.

```
void SetPosition ( float x, float y, float z, char *part_name ) ;
```

Description

Sets the articulated part position to the Cartesian coordinates x,y,z. The default position is (0.0, 0.0, 0.0). The position is relative to the parent position.

Arguments

float x	The x coordinate of the articulated part position.
float y	The y coordinate of the articulated part position.
float z	The z coordinate of the articulated part position.
char *part_name	The name of the part to be positioned.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

SetOrientation

Sets the model's orientation.

```
void SetOrientation ( double roll, double pitch, double yaw, long index ) ;
```

Description

Rotates the model based on the roll, pitch, and yaw angles. The default angles are (0.0, 0.0, 0.0). The index specifies whether to rotate the model itself or one of its sub-parts (articulated parts). For an articulated part, the rotations are relative to its parent's orientation.

Arguments

double <i>roll</i>	The rotation about the x axis, in degrees.
double <i>pitch</i>	The rotation about the y axis, in degrees.
double <i>yaw</i>	The rotation about the z axis, in degrees.
long <i>index</i>	The sub-part index to be rotated. The valid range is zero through <code>GetNumberOfArticulatedParts()</code> . The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

SetOrientation

Sets the model's orientation.

```
void SetOrientation ( float roll, float pitch, float yaw, long index ) ;
```

Description

Rotates the model based on the roll, pitch, and yaw angles. The default angles are (0.0, 0.0, 0.0). The index specifies whether to rotate the model itself or one of its sub-parts (articulated parts). For an articulated part, the rotations are relative to its parent's orientation.

Arguments

float <i>roll</i>	The rotation about the x axis, in degrees.
float <i>pitch</i>	The rotation about the y axis, in degrees.
float <i>yaw</i>	The rotation about the z axis, in degrees.
long <i>index</i>	The sub-part index to be rotated. The valid range is zero through <code>GetNumberOfArticulatedParts()</code> . The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

SetOrientation

Sets an articulated part's orientation.

```
void SetOrientation( double roll, double pitch, double yaw, char *part_name );
```

Description

Rotates the articulated part based on the roll, pitch, and yaw angles. The default angles are (0.0, 0.0, 0.0). The rotations are relative to its parent's orientation.

Arguments

double <i>roll</i>	The rotation about the x axis, in degrees.
double <i>pitch</i>	The rotation about the y axis, in degrees.
double <i>yaw</i>	The rotation about the z axis, in degrees.
char * <i>part_name</i>	The name of the part to be positioned.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

SetOrientation

Sets an articulated part's orientation.

```
void SetOrientation ( float roll, float pitch, float yaw, char *part_name ) ;
```

Description

Rotates the articulated part based on the roll, pitch, and yaw angles. The default angles are (0.0, 0.0, 0.0). The rotations are relative to its parent's orientation.

Arguments

float <i>roll</i>	The rotation about the x axis, in degrees.
float <i>pitch</i>	The rotation about the y axis, in degrees.
float <i>yaw</i>	The rotation about the z axis, in degrees.
char * <i>part_name</i>	The name of the part to be positioned.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

SetOrientation

Sets a model's orientation.

```
void SetOrientation ( float *in_matrix, long index ) ;
```

Description

Rotates the model by the provided rotation matrix. The index specifies whether to rotate the model itself or one of its sub-parts (articulated parts). For an articulated part, the rotations are relative to its parent's orientation.

The ordering of the matrix elements is as follows:

0	1	2
Forward .X	Forward .Y	Forward .Z
3	4	5
Left .X	Left .Y	Left .Z
6	7	8
Up .X	Up .Y	Up .Z

Arguments

float *in_matrix
float index

Pointer to the 3x3 rotation matrix.
The sub-part index to be rotated. The valid range is zero through GetNumberOfArticulatedParts(). The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

Matrices defined in this way must be orthonormal matrices, that is, no scaling or skewing is allowed. If a matrix with scaling or skewing is passed to this function, unpredictable polygon culling, lighting anomalies and texturing artifacts may be introduced into the rendered image.



Function Reference

(Model Functions)

SetOrientation

Sets an articulated part's orientation.

```
void SetOrientation ( float *in_matrix, char *part_name ) ;
```

Description

Rotates the articulated part by the provided rotation matrix. The rotations are relative to its parent's orientation.

The ordering of the matrix elements is as follows:

0	1	2
Forward .X	Forward .Y	Forward .Z
3	4	5
Left .X	Left .Y	Left .Z
6	7	8
Up .X	Up .Y	Up .Z

Arguments

float *in_matrix	Pointer to the 3x3 rotation matrix.
char *part_name	The name of the part to be positioned.

Return Value

None.

Application Notes

Matrices defined in this way must be orthonormal matrices, that is, no scaling or skewing is allowed. If a matrix with scaling or skewing is passed to this function, unpredictable polygon culling, lighting anomalies and texturing artifacts may be introduced into the rendered image.



Function Reference

(Model Functions)

Activate

Activates an inactive model.

```
void Activate ( void ) ;
```

Description

Enables the processing of this model by the hardware.

Arguments

None.

Return Value

None.

Application Notes

By default, all models are active.



Function Reference

(Model Functions)

Deactivate

Deactivates an active model.

```
void Deactivate ( void ) ;
```

Description

Removes a model from the active display list. The model data stays resident in the hardware memory but is flagged as inactive.

Arguments

None.

Return Value

None.

Application Notes

By default, all models are active.



Function Reference

(Model Functions)

AttachColorTable

Associates a color table with a model.

```
void AttachColorTable ( PRO_Color_Table *color_table );
```

Description

Associates a color table with a model. This is used for models whose polygons contain color table indices rather than actual RGB values.

Arguments

PRO_Color_Table *color_table	Pointer to a PRO_Color_Table object.
------------------------------	--------------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

SetRangeLOD

Indicates that the model's blend data will be interpreted as ranges.

```
void SetRangeLOD ( void ) ;
```

Description

Indicates that the model will interpret associated blending parameters as range values.

Arguments

None.

Return Value

None.

Application Notes

The blend tables attached to the viewports in which the model is visible must be consistent in definition (size or range) with the model's setting.



Function Reference

(Model Functions)

SetSizeLOD

Indicates that the model's blend data will be interpreted as sizes.

```
void SetSizeLOD ( void ) ;
```

Description

Indicates that the model will interpret associated blend data as subtended angles.

Arguments

None.

Return Value

None.

Application Notes

The blend tables attached to the viewports in which the model is visible must be consistent in definition (size or range) with the model's setting.



Function Reference

(Model Functions)

UserMatrix

Defines the matrix to be used by the model.

```
void UserMatrix ( PRO_Matrix *user_matrix, long index ) ;
```

Description

This function allows the user to define which matrix is to be used by a model or articulated part. This provides the mechanism for a single matrix to be shared across models. The index specifies either the main model (index zero) or one of its sub-parts.

Arguments

PRO_Matrix *user_matrix	A pointer to the shared matrix.
long index	The sub-part index. The valid range is zero through GetNumberOfArticulatedParts(). The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

This function applies only to pre-formatted models.



Function Reference

(Model Functions)

IsStatic

Returns type of model.

```
long IsStatic ( void ) ;
```

Description

Indicates whether the model has an associated matrix.

Arguments

None.

Return Value

Returns 1 if the model is static (has no associated matrix) and 0 otherwise.

Application Notes

None.



Function Reference

(Model Functions)

IsActive

Returns current state of model.

```
long IsActive ( void ) ;
```

Description

Indicates whether model is active in any associated database.

Arguments

None.

Return Value

Returns 1 if the model is active and 0 otherwise.

Application Notes

None.



Function Reference

(Model Functions)

GetPosition

Gets the model's position.

```
void GetPosition ( double &x, double &y, double &z, long index ) ;
```

Description

Gets the Cartesian coordinates (x,y,z) of the model's position. The index specifies whether to get the position the model itself or one of its sub-parts (articulated parts). For an articulated part, its position is relative to the parent position.

Arguments

double &x	The x coordinate of the model position.
double &y	The y coordinate of the model position.
double &z	The z coordinate of the model position.
long <i>index</i>	The sub-part index. The valid range is zero through <code>GetNumberOfArticulatedParts()</code> . The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetPosition

Gets the model's position.

```
void GetPosition ( float &x, float &y, float &z, long index ) ;
```

Description

Gets the Cartesian coordinates (x,y,z) of the model's position. The index specifies whether to get the position the model itself or one of its sub-parts (articulated parts). For an articulated part, its position is relative to the parent position.

Arguments

float &x	The x coordinate of the model position.
float &y	The y coordinate of the model position.
float &z	The z coordinate of the model position.
long <i>index</i>	The sub-part index. The valid range is zero through <code>GetNumberOfArticulatedParts()</code> . The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetPosition

Gets an articulated part's position.

```
void GetPosition ( double &x, double &y, double &z, char *part_name ) ;
```

Description

Gets the Cartesian coordinates (x,y,z) of the articulated part's position. The position is relative to the parent position.

Arguments

double &x	The x coordinate of the articulated part position.
double &y	The y coordinate of the articulated part position.
double &z	The z coordinate of the articulated part position.
char *part_name	The name of the articulated part.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetPosition

Gets an articulated part's position.

```
void GetPosition ( float &x, float &y, float &z, char *part_name ) ;
```

Description

Gets the Cartesian coordinates (x,y,z) of the articulated part's position. The position is relative to the parent position.

Arguments

float &x	The x coordinate of the articulated part position.
float &y	The y coordinate of the articulated part position.
float &z	The z coordinate of the articulated part position.
char *part_name	The name of the articulated part.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetOrientation

Gets the model's orientation.

```
void GetOrientation ( double &roll, double &pitch, double &yaw, long index ) ;
```

Description

Gets the rotation angles (roll,pitch,yaw) of the model's orientation. The index specifies whether to get the orientation of the model itself or one of its sub-parts (articulated parts). For an articulated part, the rotations are relative to its parent's orientation.

Arguments

double &roll	The rotation about the x axis, in degrees.
double &pitch	The rotation about the y axis, in degrees.
double &yaw	The rotation about the z axis, in degrees.
long index	The sub-part index. The valid range is zero through GetNumberOfArticulatedParts(). The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetOrientation

Gets the model's orientation.

```
void GetOrientation ( float &roll, float &pitch, float &yaw, long index ) ;
```

Description

Gets the rotation angles (roll,pitch,yaw) of the model's orientation. The index specifies whether to get the orientation of the model itself or one of its sub-parts (articulated parts). For an articulated part, the rotations are relative to its parent's orientation.

Arguments

float &roll	The rotation about the x axis, in degrees.
float &pitch	The rotation about the y axis, in degrees.
float &yaw	The rotation about the z axis, in degrees.
long index	The sub-part index. The valid range is zero through GetNumberOfArticulatedParts(). The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetOrientation

Gets an articulated part's orientation.

```
void GetOrientation ( double &roll, double &pitch, double &yaw, char *part_name ) ;
```

Description

Gets the roll, pitch, and yaw rotation angles for the articulated part. The rotations are relative to its parent's orientation.

Arguments

double &roll	The rotation about the x axis, in degrees.
double &pitch	The rotation about the y axis, in degrees.
double &yaw	The rotation about the z axis, in degrees.
char *part_name	The name of the articulated part.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetOrientation

Gets an articulated part's orientation.

```
void GetOrientation ( float &roll, float &pitch, float &yaw, char *part_name ) ;
```

Description

Gets the roll, pitch, and yaw rotation angles for the articulated part. The rotations are relative to its parent's orientation.

Arguments

float &roll	The rotation about the x axis, in degrees.
float &pitch	The rotation about the y axis, in degrees.
float &yaw	The rotation about the z axis, in degrees.
char *part_name	The name of the articulated part.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetLocalForward

Gets a model's forward vector.

```
void GetLocalForward ( float &x, float &y, float &z, long index ) ;
```

Description

Gets the forward vector in model coordinates.

Arguments

float &x	The x component of the forward vector.
float &y	The y component of the forward vector.
float &z	The z component of the forward vector.
long <i>index</i>	The sub-part index. The valid range is zero through <code>GetNumberOfArticulatedParts()</code> . The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetLocalForward

Gets a model's forward vector.

```
void GetLocalForward ( float *x, float *y, float *z, long index ) ;
```

Description

Gets the forward vector in model coordinates.

Arguments

float *x	Pointer to the x component of the forward vector.
float *y	Pointer to the y component of the forward vector.
float *z	Pointer to the z component of the forward vector.
long index	The sub-part index. The valid range is zero through GetNumberOfArticulatedParts(). The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetLocalForward

Gets an articulated part's forward vector.

```
void GetLocalForward ( float &x, float &y, float &z, char *part_name ) ;
```

Description

Gets the forward vector in model coordinates.

Arguments

float &x	The x component of the forward vector.
float &y	The y component of the forward vector.
float &z	The z component of the forward vector.
char *part_name	The name of the articulated part.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetLocalForward

Gets an articulated part's forward vector.

```
void GetLocalForward ( float *x, float *y, float *z, char *part_name ) ;
```

Description

Gets the forward vector in model coordinates.

Arguments

float *x	Pointer to the x component of the forward vector.
float *y	Pointer to the y component of the forward vector.
float *z	Pointer to the z component of the forward vector.
char *part_name	The name of the articulated part.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetLocalLeft

Gets a model's left vector.

```
void GetLocalLeft ( float &x, float &y, float &z, long index ) ;
```

Description

Gets the left vector in model coordinates.

Arguments

float &x	The x component of the left vector.
float &y	The y component of the left vector.
float &z	The z component of the left vector.
long <i>index</i>	The sub-part index. The valid range is zero through <code>GetNumberOfArticulatedParts()</code> . The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetLocalLeft

Gets a model's left vector.

```
void GetLocalLeft ( float *x, float *y, float *z, long index ) ;
```

Description

Gets the left vector in model coordinates.

Arguments

float *x	Pointer to the x component of the left vector.
float *y	Pointer to the y component of the left vector.
float *z	Pointer to the z component of the left vector.
long index	The sub-part index. The valid range is zero through GetNumberOfArticulatedParts(). The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetLocalLeft

Gets an articulated part's left vector.

```
void GetLocalLeft ( float &x, float &y, float &z, char *part_name ) ;
```

Description

Gets the left vector in model coordinates.

Arguments

float &x	The x component of the left vector.
float &y	The y component of the left vector.
float &z	The z component of the left vector.
char *part_name	The name of the articulated part.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetLocalLeft

Gets an articulated part's left vector.

```
void GetLocalLeft ( float *x, float *y, float *z, char *part_name ) ;
```

Description

Gets the left vector in model coordinates.

Arguments

float *x	Pointer to the x component of the left vector.
float *y	Pointer to the y component of the left vector.
float *z	Pointer to the z component of the left vector.
char *part_name	The name of the articulated part.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetLocalUp

Gets a model's up vector.

```
void GetLocalUp ( float &x, float &y, float &z, long index ) ;
```

Description

Gets the up vector in model coordinates.

Arguments

float &x	The x component of the up vector.
float &y	The y component of the up vector.
float &z	The z component of the up vector.
long <i>index</i>	The sub-part index. The valid range is zero through <code>GetNumberOfArticulatedParts()</code> . The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetLocalUp

Gets a model's up vector.

```
void GetLocalUp ( float *x, float *y, float *z, long index ) ;
```

Description

Gets the up vector in model coordinates.

Arguments

float *x	Pointer to the x component of the up vector.
float *y	Pointer to the y component of the up vector.
float *z	Pointer to the z component of the up vector.
long index	The sub-part index. The valid range is zero through GetNumberOfArticulatedParts(). The default value is 0, which corresponds to the main model.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetLocalUp

Gets an articulated part's up vector.

```
void GetLocalUp ( float &x, float &y, float &z, char *part_name ) ;
```

Description

Gets the up vector in model coordinates.

Arguments

float &x	The x component of the up vector.
float &y	The y component of the up vector.
float &z	The z component of the up vector.
char *part_name	The name of the articulated part.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetLocalUp

Gets an articulated part's up vector.

```
void GetLocalUp ( float *x, float *y, float *z, char *part_name ) ;
```

Description

Gets the up vector in model coordinates.

Arguments

float *x	Pointer to the x component of the up vector.
float *y	Pointer to the y component of the up vector.
float *z	Pointer to the z component of the up vector.
char *part_name	The name of the articulated part.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

GetNumberOfAnimationSequences

Gets the total number of animations associated with the model.

```
long GetNumberOfAnimationSequences ( void ) ;
```

Description

Returns the total number of animation sequences associated with the model.

Arguments

None.

Return Value

The number of animation sequences associated with the model.

Application Notes

See also `StoreAnimationSequence`.



Function Reference

(Model Functions)

GetAnimationSequence

Gets the PRO AnimationSequence object by index.

```
PRO_AnimationSequence GetAnimationSequence ( long index ) ;
```

Description

Returns a pointer to the PRO_AnimationSequence object by the specified index. This handle allows the user to specify the characteristics of the animation sequence.

Arguments

long <i>index</i>	Index of the animation sequence. Value is zero through (GetNumberOfAnimationSequences() - 1) .
-------------------	--

Return Value

Pointer to the associated PRO_AnimationSequence object.
NULL if no match is found.

Application Notes

The total number of animation sequences associated with this model can be obtained using the function GetNumberOfAnimationSequences. See also StoreAnimationSequence.



Function Reference

(Model Functions)

GetAnimationSequence

Gets the PRO Animation object specified by name.

```
PRO_Animation GetAnimationSequence ( char *animation_name ) ;
```

Description

Returns a pointer to the PRO_Animation object by the specified name. This handle allows the user to specify the characteristics of the animation sequence.

Arguments

char * <i>animation_name</i>	Name of the animation sequence.
------------------------------	---------------------------------

Return Value

Pointer to the associated PRO_Animation object.
NULL if no match is found.

Application Notes

See also StoreAnimationSequence.



Function Reference

(Model Functions)

GetAnimationSequenceName

Gets the name of specified animation sequence.

```
char * GetAnimationSequenceName ( long index ) ;
```

Description

Returns the name of an animation sequence specified by an index.

Arguments

long <i>index</i>	Index of the animation sequence. Value is zero through (GetNumberOfAnimationSequences() -1).
-------------------	--

Return Value

Name of the animation sequence.

Application Notes

The total number of animation sequences associated with this model can be obtained using the function GetNumberOfAnimationSequences. See also StoreAnimationSequence.



Function Reference

(Model Functions)

GetNumberOfArticulatedParts

Gets the number of sub-parts in the model.

```
long GetNumberOfArticulatedParts ( void ) ;
```

Description

Returns the number of articulated parts in the model.

Arguments

None.

Return Value

The number of articulated parts associated with the model.

Application Notes

See also `StoreArticulatedPart`.



Function Reference

(Model Functions)

GetArticulatedPartMatrixIndex

Gets the index of a sub-part specified by name.

```
long GetArticulatedPartMatrixIndex ( char *articulated_part_name ) ;
```

Description

Returns the index associated with a named articulated part. Articulated part matrix indices are numbered starting with one, since index zero corresponds to the main model. Using this sub-part index to manipulate the articulated part is more efficient than using its name.

Arguments

<code>char *articulated_part_name</code>	Name of an articulated part.
--	------------------------------

Return Value

The sub-part index for the named articulated part. Returns -1 if the specified name is not associated with an articulated part of this model.

Application Note

See also `StoreArticulatedPart`.



Function Reference

(Model Functions)

GetArticulatedPartName

Gets the name of specified articulated part.

```
char * GetArticulatedPartName ( long index ) ;
```

Description

Returns the name of an articulated part specified by an index.

Arguments

long <i>index</i>	Index of the articulated part. Value is zero through (GetNumberOfArticulatedParts() -1).
-------------------	--

Return Value

Name of the articulated part.

Application Notes

The index used here is not the matrix index, but simply a zero-relative index into an array of articulated parts names. See also StoreArticulatedPart.



Function Reference

(Model Functions)

GetNumberOfTextureMaps

Gets the number of texture maps associated with the model.

```
long GetNumberOfTextureMaps ( void ) ;
```

Description

Returns the number of texture maps used by any polygons in this model.

Arguments

None.

Return Value

The number of texture maps associated with the model.

Application Notes

See also StoreTextureData.



Function Reference

(Model Functions)

GetTextureMapName

Gets the name of specified texture map.

```
char * GetTextureMapName ( long index ) ;
```

Description

Returns the name of a texture map specified by an index.

Arguments

long <i>index</i>	Index of the texture map. Value is zero through (GetNumberOfTextureMaps() -1).
-------------------	--

Return Value

Name of the texture map.

Application Notes

See also StoreTextureData.



Function Reference

(Model Functions)

ProcessImmediateModeData

Flushes immediate mode polygons to the hardware.

```
void ProcessImmediateModeData ( void ) ;
```

Description

Processes the display list data attached to this model and flushes the data to the hardware.

Arguments

None.

Return Value

None.

Application Notes

This function is not used for pre-formatted models.



Function Reference

(Model Functions)

ProcessMorphedPolygons

Process geometry and texture deltas for immediate mode polygons.

```
void ProcessMorphedPolygons ( void ) ;
```

Description

Processes partial updates to immediate mode polygons that have already been downloaded via ProcessImmediateModeData.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

AddCullingNode

Adds a culling node to the hierarchy of the model.

```
PRO_Status AddCullingNode ( PRO_Culling_Node *culling_node_ptr ) ;
```

Description

Adds a culling node as a child of the model.

Arguments

PRO_Culling_Node *culling_node_ptr	A pointer to the PRO_Culling_Node object being added to the model's hierarchy.
------------------------------------	--

Return Value

PRO_SUCCESS if culling node is successfully added to the model's hierarchy.
PRO_FAILURE otherwise.

Application Notes

None.



Function Reference

(Model Functions)

AddChild

Adds a culling node to the hierarchy of the model.

```
PRO_Status AddChild ( PRO_Culling_Node *culling_node_ptr );
```

Description

Adds a culling node as a child of the model.

Arguments

PRO_Culling_Node * <i>culling_node_ptr</i>	A pointer to the PRO_Culling_Node object being added to the model's hierarchy.
--	--

Return Value

PRO_SUCCESS if culling node is successfully added to the model's hierarchy.
PRO_FAILURE otherwise.

Application Notes

None.



Function Reference

(Model Functions)

AddPointLight

Adds a Point Light to a model.

```
PRO_Status AddPointLight ( PRO_Point_Light *a_point_light );
```

Description

Adds a point light to a model.

Arguments

PRO_Point_Light *a_point_light A pointer to a PRO_Point_Light.

Return Value

PRO_SUCCESS if the PRO_Point_Light was attached to the model successfully.
PRO_FAILURE if an error was encountered.

Application Notes

None.



Function Reference

(Model Functions)

AddInstanceSet

Adds a Instance Set culling node to the hierarchy of the model.

```
PRO_Status AddInstanceSet ( PRO_Instance_Set *instance_set_ptr );
```

Description

Adds an instance set as a child of the model.

Arguments

PRO_Instance_Set *instance_set_ptr A pointer to the PRO_Instance_Set object being added to the model's hierarchy.

Return Value

PRO_SUCCESS if culling node is successfully added to the model's hierarchy
PRO_FAILURE otherwise.

Application Notes

None.



Function Reference

(Model Functions)

AddPolygon

Adds a polygon to the hierarchy of the model.

```
PRO_Status AddPolygon ( PRO_Polygon *polygon_ptr );
```

Description

Adds a polygon as a child of the model.

Arguments

PRO_Polygon * <i>polygon_ptr</i>	A pointer to the PRO_Polygon object being added to the model's hierarchy.
----------------------------------	---

Return Value

PRO_SUCCESS if polygon is successfully added to the model's hierarchy.
PRO_FAILURE otherwise.

Application Notes

None.



Function Reference

(Model Functions)

AddChild

Adds a polygon to the hierarchy of the model.

```
PRO_Status AddChild ( PRO_Polygon *polygon_ptr );
```

Description

Adds a polygon as a child of the model.

Arguments

PRO_Polygon * <i>polygon_ptr</i>	A pointer to the PRO_Polygon object being added to the model's hierarchy.
----------------------------------	---

Return Value

PRO_SUCCESS if polygon is successfully added to the model's hierarchy.
PRO_FAILURE otherwise.

Application Notes

None.



Function Reference

(Model Functions)

EnableVertexSharing

Enables vertex sharing among polygons of this model.

```
void EnableVertexSharing ( void ) ;
```

Description

Enables vertex sharing among polygons associated with this model. By default, vertex sharing is enabled.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

DisableVertexSharing

Disables vertex sharing among polygons of this model.

```
void DisableVertexSharing ( void ) ;
```

Description

Disables vertex sharing among polygons associated with this model. By default, vertex sharing is enabled.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

SetAutoDeleteCullingData

Indicates that culling nodes should be destroyed with their parent.

```
void SetAutoDeleteCullingData ( void ) ;
```

Description

Indicates that culling nodes associated with a model should be destroyed automatically when the model is deleted. The default state requires the application to destroy all culling nodes explicitly. This function is essentially the same as `SetAutoDestructCullingData`.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

SetAutoDeletePolygonData

Indicates that polygons should be destroyed with their parent.

```
void SetAutoDeletePolygonData ( void ) ;
```

Description

Indicates that polygons associated with a model should be destroyed automatically when the model is deleted. The default state requires the application to destroy all polygons explicitly.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

SetAutoDestructCullingData

Indicates that culling nodes should be destroyed after formatting.

```
void SetAutoDestructCullingData ( void ) ;
```

Description

Indicates that culling nodes associated with a model should be destroyed automatically when the model is deleted. The default state requires the application to destroy all culling nodes explicitly. This function is essentially the same as `SetAutoDeleteCullingData`.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

SetAutoDestructPolygonData

Indicates that polygons should be destroyed after formatting.

```
void SetAutoDestructPolygonData ( void ) ;
```

Description

Indicates that polygons associated with a model should be destroyed automatically after they have been formatted and downloaded to the PRO-1000. The default state requires the application to destroy all polygons explicitly.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

StoreArticulatedPart

Associates a 6DOF node with a model.

```
void StoreArticulatedPart ( PRO_6DOF_Culling_Node *articulated_part_node ) ;
```

Description

Associates an articulated (6DOF) culling node with a model. This allows an application to manipulate the articulated part through the PRO_Model object, instead of manipulating it directly through the PRO_6DOF_Culling_Node object.

Arguments

PRO_6DOF_Culling_Node *articulated_part_node	A pointer to a PRO_6DOF_Culling_Node.
--	---------------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

StoreAnimationSequence

Associates an animation node with a model.

```
void StoreAnimationSequence ( PRO_Animation_Culling_Node *animation_node );
```

Description

Associates an animation node with a model. This allows an application to manipulate the animation through the PRO_Model object, instead of manipulating it directly through the PRO_Animation_Culling_Node object.

Arguments

PRO_Animation_Culling_Node *animation_node	A pointer to a PRO_Animation_Culling_Node.
--	--

Return Value

None.

Application Notes

None.



Function Reference

(Model Functions)

StoreTextureData

Associates a PRO Texture object with a model.

```
long StoreTextureData ( PRO_Texture *texture_map ) ;
```

Description

Associates a texture map with a model. This allows an application to query the PRO_Model object about the names of texture maps applied to its polygons.

Arguments

PRO_Texture *texture_map	A pointer to a PRO_Texture object.
--------------------------	------------------------------------

Return Value

The index into the array of this model's texture map names.

Application Notes

None.



Function Reference

(Model Functions)

Save

Saves a run-time formatted model to file.

```
PRO_Status Save ( char *model_name ) ;
```

Description

Saves a run-time formatted model to file. Once saved to file, a formatted model may be used to instantiate a PRO_Model using the pre-formatted model constructor. This allows a model to be processed into hardware format once and reloaded multiple times without the overhead of formatting each time.

Arguments

char *model_name

The name of the model, to be used to name output files.

Return Value

PRO_FAILURE if the model has not been formatted or if the API was not initialized in offline mode; PRO_SUCCESS otherwise.

Application Notes

This function is valid only in offline mode.



Animation Functions

The Animation class functions specify the characteristics and control the functionality of animation sequences associated with a model or a PRO_Animation_Culling_Node. Note that PRO_Animation objects are not instantiated directly. Instead, pointers to PRO_Animation objects are retrieved using methods of the associated PRO_Model object or PRO_Animation_Culling_Node object. All animations default to user controlled, starts visible, ends visible.

(Animation Functions)

AnimationStartsVisible

Sets an animation to be visible at activation.

```
void AnimationStartsVisible ( void ) ;
```

Description

Sets the state of an animation sequence to be visible when activated. This is the default state for an animation sequence. For animations which do not want a visible initial frame, function AnimationStartsInvisible should be invoked before activating the animation.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

AnimationEndsVisible

Sets the animation to remain visible when deactivated.

```
void AnimationEndsVisible ( void ) ;
```

Description

Sets the state of an animation sequence to remain visible when deactivated. This is the default state for an animation sequence. For animations which do not want a visible last frame, function AnimationEndsInvisible should be invoked before activating the animation.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

AnimationStartsInvisible

Sets an animation to be invisible at activation.

```
void AnimationStartsInvisible ( void ) ;
```

Description

Sets the state of an animation sequence to be invisible until activated.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

AnimationEndsInvisible

Sets the animation to be invisible when deactivated.

```
void AnimationEndsInvisible ( void ) ;
```

Description

Sets the state of an animation sequence to be invisible when deactivated.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

ActivateSequence

Activates an animation sequence.

```
void ActivateSequence ( void ) ;
```

Description

For animations that are not user controlled, this function starts processing the animation sequence according to previously-set characteristics (e.g., runs forward, runs for 10 seconds). This function does not apply to animations that have been declared user-controlled.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

DeactivateSequence

Deactivates the animation sequence.

```
void DeactivateSequence ( void ) ;
```

Description

Stops processing of an animation sequence. Depending upon previous settings, the last displayed frame may or may not remain visible. This function does not apply to user-controlled animations.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

AnimationRunsForward

Specifies a forward-running animation.

```
void AnimationRunsForward ( void ) ;
```

Description

Specifies that the animation will run forward through its frames, from its initial to its last frame.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

AnimationRunsBackwards

Specifies a backward-running animation.

```
void AnimationRunsBackwards ( void ) ;
```

Description

Specifies that the animation will run backward through its frames, from its last to its initial frame.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

AnimationRunsMirrored

Specifies an animation that runs forward, then backward.

```
void AnimationRunsMirrored ( void ) ;
```

Description

Specifies that the animation will run forward through its frames, from its initial to its last frame, and then back again to its initial frame.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

AnimationIsUserControlled

Specifies an animation that runs according to user controls.

```
void AnimationIsUserControlled ( void ) ;
```

Description

Specifies that all frame activation and deactivation operations are provided by the host application. In this mode, the user must specify when to activate and deactivate each frame of the animation.

Arguments

None.

Return Value

None.

Application Notes

Use functions `SetFrameActive` and `SetFrameInactive` to control each frame of the animation.



Function Reference

(Animation Functions)

AnimationRunsForever

Specifies an animation that runs continuously.

```
void AnimationRunsForever ( void ) ;
```

Description

Specifies that the animation will run through its frames continuously. Use of this call removes the animation from the user controlled state.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

SetAnimationCycleTime

Sets the time for one cycle through the animation.

```
void SetAnimationCycleTime ( double time_in_seconds );
```

Description

Sets the time in seconds that one cycle through all frames of the animation sequence will take.

Arguments

double <i>time_in_seconds</i>	Time in seconds to process the entire animation.
-------------------------------	--

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

SetAnimationCycleTime

Sets the time for one cycle through the animation.

```
void SetAnimationCycleTime ( float time_in_seconds ) ;
```

Description

Sets the time in seconds that one cycle through all frames of the animation sequence will take.

Arguments

float <i>time_in_seconds</i>	Time in seconds to process the entire animation.
------------------------------	--

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

SetAnimationRepeatCount

Sets the number of times to run through the animation.

```
void SetAnimationRepeatCount ( long count ) ;
```

Description

Sets the number of times to run through the entire animation.

Arguments

long <i>count</i>	Number of iterations for the animation sequence.
-------------------	--

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

SetFrameActive

Activates the specified animation frame.

```
void SetFrameActive ( long frame_index );
```

Description

Sets the specified frame of the animation to an active state. This function is used when an animation is user controlled to activate animation frames as required by the application.

Arguments

long <i>frame_index</i>	Frame number to activate. Value is zero through (<code>GetNumberOfAnimationFrames()</code> -1).
-------------------------	---

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

SetFrameInactive

Deactivates the specified animation frame.

```
void SetFrameInactive ( long frame_index );
```

Description

Sets the specified frame of the animation to an inactive state. This function is used when an animation is user controlled to deactivate animation frames as required by the application.

Arguments

long <i>frame_index</i>	Frame number to deactivate. Value is zero through (GetNumberOfAnimationFrames() -1).
-------------------------	--

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

GetAnimationCycleTime

Gets the time for one cycle through the animation.

```
void GetAnimationCycleTime ( float &time_in_seconds ) ;
```

Description

Gets the time in seconds that one cycle through all frames of the animation sequence will take.

Arguments

float & <i>time_in_seconds</i>	Time in seconds to process the entire animation.
--------------------------------	--

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

GetAnimationCycleTime

Gets the time for one cycle through the animation.

```
void GetAnimationCycleTime ( double &time_in_seconds ) ;
```

Description

Gets the time in seconds that one cycle through all frames of the animation sequence will take.

Arguments

double &time_in_seconds	Time in seconds to process the entire animation.
-------------------------	--

Return Value

None.

Application Notes

None.



Function Reference

(Animation Functions)

GetNumberOfAnimationFrames

Gets the number of animation frames.

```
long GetNumberOfAnimationFrames ( void ) ;
```

Description

Gets the number of frames comprising the animation.

Arguments

None.

Return Value

The number of frames in the animation sequence.

Application Notes

None.



Function Reference

(Animation Functions)

GetCurrentFrame

Gets the current animation frame.

```
long GetCurrentFrame ( void ) ;
```

Description

Gets the current frame of the animation.

Arguments

None.

Return Value

The current frame of the animation sequence.

Application Notes

None.



Function Reference

(Animation Functions)

GetAnimationState

Indicates whether an animation sequence is active or inactive.

```
long GetAnimationState ( void ) ;
```

Description

Gets the state of the animation, indicating whether it is currently active or inactive.

Arguments

None.

Return Value

One if the animation is active, or zero if it is inactive.

Application Notes

None.



Culling Node Functions

The Culling Node Class functions are used to build a database hierarchy that will be used by the hardware to determine which graphical objects should be rendered.

(Culling Node Functions)

PRO_Culling_Node class constructor

Creates an instance of a PRO Culling Node.

```
PRO_Culling_Node ( void ) ;
```

Description

Creates an instance of a PRO_Culling_Node.

Arguments

None.

Return Value

None.

Application Notes

Function GetStatus should be called to verify object instantiation.



Function Reference

(Culling Node Functions)

PRO_Culling_Node class destructor

Destroys an instance of a PRO Culling Node.

```
~PRO_Culling_Node ( void ) ;
```

Description

Destroys an instance of a PRO_Culling_Node.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Culling Node Functions)

GetStatus

Returns status of culling node instantiation.

```
PRO_Status GetStatus ( void ) ;
```

Description

Returns status of PRO_Culling_Node object instantiation.

Arguments

None.

Return Value

PRO_SUCCESS if successful.

PRO_OUT_OF_HOST_MEMORY if host memory allocation fails.

PRO_OUT_OF_DISPLAY_LIST_MEMORY if hardware memory has no space for the culling node.

Application Notes

None.



Function Reference

(Culling Node Functions)

ActivateNode

Activates culling node.

```
void ActivateNode ( void ) ;
```

Description

Indicates that culling node and its children should be processed by the PRO-1000 hardware.

Arguments

None.

Return Value

None.

Application Notes

This is the default state for all culling nodes.



Function Reference

(Culling Node Functions)

DeactivateNode

Deactivates culling node.

```
void DeactivateNode ( void ) ;
```

Description

Indicates that culling node and its children should be ignored by the PRO-1000 hardware. All culling nodes are active by default.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Culling Node Functions)

AddChild

Attaches a child culling node to a culling node.

```
void AddChild ( PRO_Culling_Node *child_ptr ) ;
```

Description

Attaches a child culling node to a PRO_Culling_Node.

Arguments

PRO_Culling_Node * <i>child_ptr</i>	A pointer to a PRO_Culling_Node object.
-------------------------------------	---

Return Value

None.

Application Notes

None.



Function Reference

(Culling Node Functions)

AddChild

Attaches a child polygon to a culling node.

```
void AddChild ( PRO_Polygon *child_ptr );
```

Description

Attaches a child polygon to a PRO_Culling_Node.

Arguments

PRO_Polygon *child_ptr	A pointer to a PRO_Polygon object.
------------------------	------------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Culling Node Functions)

AddPointLight

Adds a point light to a culling node.

```
PRO_Status AddPointLight ( PRO_Point_Light *a_point_light );
```

Description

Adds a point light as a child of a culling node.

Arguments

PRO_Point_Light *a_point_light A pointer to a PRO_Point_Light.

Return Value

PRO_SUCCESS if the PRO_Point_Light was attached successfully.
PRO_FAILURE if an error was encountered.

Application Notes

None.



Function Reference

(Culling Node Functions)

AddInstanceSet

Adds an instance set to a culling node.

```
PRO_Status AddInstanceSet ( PRO_Instance_Set *instance_set_ptr );
```

Description

Adds an instance set as a child of a culling node.

Arguments

PRO_Instance_Set *instance_set_ptr A pointer to a PRO_Instance_Set.

Return Value

PRO_SUCCESS if the PRO_Instance_Set was attached successfully.
PRO_FAILURE if an error was encountered.

Application Notes

None.



Function Reference

(Culling Node Functions)

DetachChild

Detaches a child culling node from a culling node.

```
void DetachChild ( PRO_Culling_Node *child_ptr ) ;
```

Description

Detaches a child culling node from a PRO_Culling_Node.

Arguments

PRO_Culling_Node * <i>child_ptr</i>	A pointer to a PRO_Culling_Node object.
-------------------------------------	---

Return Value

None.

Application Notes

Can also be used to detach point lights and instance sets (attached using AddPointLight or AddInstanceSet) from their parent culling node.



Function Reference

(Culling Node Functions)

DetachChild

Detaches a child polygon from a culling node.

```
void DetachChild ( PRO_Polygon *child_ptr ) ;
```

Description

Detaches a child polygon from a PRO_Culling_Node.

Arguments

PRO_Polygon * <i>child_ptr</i>	A pointer to a PRO_Polygon object.
--------------------------------	------------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Culling Node Functions)

AttachColorTable

Associates a color table with a culling node.

```
void AttachColorTable ( PRO_Color_Table *color_table );
```

Description

Associates a color table with a culling node. This defines the default color table for all children of this node.

Arguments

PRO_Color_Table *color_table	Pointer to a PRO_Color_Table object.
------------------------------	--------------------------------------

Return Value

None.

Application Notes

A color table is needed only if any attached polygons have a color index defined.



Function Reference

(Culling Node Functions)

SetRangeLOD

Indicates that the culling node's blend data will be interpreted as ranges.

```
void SetRangeLOD ( long traverse_hierarchy ) ;
```

Description

Indicates that the culling node will interpret associated blend data as ranges.

Arguments

<code>long <i>traverse_hierarchy</i></code>	Indicates whether to propagate the size LOD attribute to all child culling nodes in this node's hierarchy. The default is 1 (TRUE).
---	---

Return Value

None.

Application Notes

None.



Function Reference

(Culling Node Functions)

SetSizeLOD

Indicates that the culling node's blend data will be interpreted as sizes.

```
void SetSizeLOD ( long traverse_hierarchy ) ;
```

Description

Indicates that the culling node will interpret associated blend data as subtended angles.

Arguments

<code>long <i>traverse_hierarchy</i></code>	Indicates whether to propagate the size LOD attribute to all child culling nodes in this node's hierarchy. The default is 1 (TRUE).
---	---

Return Value

None.

Application Notes

None.



Function Reference

(Culling Node Functions)

SetFeatureType

Sets the feature type for blending.

```
void SetFeatureType ( long feature_type );
```

Description

Sets the feature type for this culling node. The feature type is used as an index into a Blend Table.

Arguments

long *feature_type* Feature type for blending. The valid values are zero through 127.

Return Value

None.

Application Notes

Blending is used for both load management and size clamping of objects.



Function Reference

(Culling Node Functions)

SetGroupLayerNumber

Sets the layer number for group layering.

```
void SetGroupLayerNumber ( long layer_number ) ;
```

Description

Sets the layer number for group layering. The group layer number sets the drawing priority for all polygons under the culling node.

Arguments

long <i>layer_number</i>	Group layer number. Zero indicates the lowest priority layer.
--------------------------	---

Return Value

None.

Application Notes

When group layering is used, the geometry being layered must all be coplanar. In addition, all culling nodes whose drawing order is to be controlled by this attribute must be children of the same parent node.

See also `FormatAndFlushPolygons` for special application handling of group layering.



Function Reference

(Culling Node Functions)

SetVolumeSizeClamp

Sets an object to clamp at the last LOD transition size.

```
void SetVolumeSizeClamp ( void ) ;
```

Description

Sets the culling node to volume size-clamp at the last LOD transition according to the blend table. This operation is valid only for leaf nodes (culling nodes whose only children are polygons).

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Culling Node Functions)

SetAutoDeleteCullingData

Indicates that child culling nodes should be destroyed with their parent.

```
void SetAutoDeleteCullingData ( void ) ;
```

Description

Indicates that child culling nodes should be destroyed automatically when their parent culling node is deleted. The default state requires the application to delete all culling nodes explicitly.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Culling Node Functions)

SetAutoDeletePolygonData

Indicates that child polygons should be destroyed with their parent.

```
void SetAutoDeletePolygonData ( void ) ;
```

Description

Indicates that child polygons should be destroyed automatically when their parent culling node is destroyed. The default state requires the application to destroy all polygons explicitly.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Culling Node Functions)

SetAutoDestructCullingData

Indicates that child culling nodes should be destroyed with their parent.

```
void SetAutoDestructCullingData ( void ) ;
```

Description

Indicates that child culling nodes should be destroyed automatically when their parent culling node is deleted. The default state requires the application to delete all culling nodes explicitly. This is essentially the same function as `SetAutoDeleteCullingData` and it is only provided for naming consistency.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Culling Node Functions)

SetAutoDestructPolygonData

Indicates that child polygons should be destroyed with their parent.

```
void SetAutoDestructPolygonData ( void ) ;
```

Description

Indicates that child polygons should be destroyed automatically after they have been formatted and downloaded to the PRO-1000. The default state requires the application to destroy all polygons explicitly.

Arguments

None.

Return Value

None.

Application Notes

Geometry formatted with this option may only be processed once since the polygon data will be destroyed once the hardware display list data has been sent to the hardware.



Function Reference

(Culling Node Functions)

FormatAndFlushPolygons

Formats polygons and downloads to hardware.

```
void FormatAndFlushPolygons ( long stencil_polygons, long enable_vertex_sharing ) ;
```

Description

Formats polygons and downloads to hardware immediately. This function is useful for applications such as database loaders which process large amounts of data and want to format subsets of this data incrementally.

Arguments

long <i>stencil_polygons</i>	Indicates whether to stencil (layer) child polygons. This is necessary if the application is dealing with group layering.
long <i>enable_vertex_sharing</i>	Indicates that vertex sharing should be used during formatting.

Return Value

None.

Application Notes

Under normal conditions (when `FormatAndFlushPolygons` is NOT used), group layering is automatically handled by the PRO/API. However, when the application takes over the order of formatting of objects by using this function, the layered polygon attribute (`stencil`) must be defined by the application.



Function Reference

(Culling Node Functions)

GetFeatureType

Gets the feature type for blending.

```
long GetFeatureType ( void ) ;
```

Description

Returns the feature type set for this culling node.

Arguments

None.

Return Value

The feature type for this node.

Application Notes

None.



Function Reference

(Culling Node Functions)

GetBoundingBox

Retrieves the current bounding box of a node.

```
void GetBoundingBox ( float *xmin , float *ymin , float *zmin,  
                    float *xmax , float *ymax , float *zmax ) ;
```

Description

Retrieves the current calculated bounding box of a culling node. Normally the bounding boxes are either user defined (for 6DOF nodes) or calculated by the software during the ProcessImmediateModeData method of the PRO_Model class.

Arguments

float *min_x	The minimum x value for this node's bounding box.
float *min_y	The minimum y value for this node's bounding box.
float *min_z	The minimum z value for this node's bounding box.
float *max_x	The maximum x value for this node's bounding box.
float *max_y	The maximum y value for this node's bounding box.
float *max_z	The maximum z value for this node's bounding box.

Return Value

None.

Application Notes

The data returned will be valid only after calling ProcessImmediateModeData for the model to which this node is associated.



Function Reference

(Culling Node Functions)

GetLODNumber

Gets the LOD number for the culling node.

```
long GetLODNumber ( void ) ;
```

Description

Returns the LOD number set for this culling node.

Arguments

None.

Return Value

The LOD number for this node.

Application Notes

None.



Function Reference

(Culling Node Functions)

GetGroupLayerNumber

Gets the group layer number for the culling node.

```
long GetGroupLayerNumber ( void ) ;
```

Description

Returns the group layer number set for this culling node.

Arguments

None.

Return Value

The group layer number for this node.

Application Notes

None.



Function Reference

(Culling Node Functions)

GetCentroid

Gets the centroid of the culling volume.

```
void GetCentroid ( PRO_Vector *centroid ) ;
```

Description

Gets the centroid of the culling volume. Centroid (node locations) are normally calculated by the PRO/API when ProcessImmediateModeData is called, except for the nodes for which the user defined a specific location (instance references).

Arguments

PRO_Vector *centroid	Pointer to the vector defining the centroid.
----------------------	--

Return Value

None.

Application Notes

This data will be valid only after ProcessImmediateModeData has been called for the model to which this node is associated.



Function Reference

(Culling Node Functions)

HasAssociatedMatrix

Returns whether the culling node supports articulation.

```
long HasAssociatedMatrix ( void ) ;
```

Description

Indicates whether the culling node has an associated matrix (i.e., supports articulation).

Arguments

None.

Return Value

One if the culling node has an associated matrix, or zero if it does not.

Application Notes

None.



LOD Culling Node Functions

The LOD Culling Node Class functions are used to define culling nodes which have indirect children representing levels of detail (LODs) in the culling hierarchy.

(LOD Culling Node Functions)

PRO_LOD_Culling_Node class constructor

Creates an instance of a PRO LOD Culling Node.

```
PRO_LOD_Culling_Node ( void ) ;
```

Description

Creates an instance of a PRO_LOD_Culling_Node. This is a PRO_Culling_Node object whose child culling nodes or polygons will be considered LODs.

Arguments

None.

Return Value

None.

Application Notes

The maximum number of LODs handled by the PRO-1000 system is four.



Function Reference

(LOD Culling Node Functions)

PRO_LOD_Culling_Node class destructor

Destroys an instance of a PRO LOD Culling Node.

```
~PRO_LOD_Culling_Node ( void ) ;
```

Description

Destroys an instance of a PRO_LOD_Culling_Node.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(LOD Culling Node Functions)

AddChild

Attaches a child culling node at specified LOD.

```
void AddChild ( PRO_Culling_Node *child_ptr , long child_index ) ;
```

Description

Attaches a child culling node to a PRO_LOD_Culling_Node.

Arguments

PRO_Culling_Node **child_ptr*
long *child_index*

A pointer to a PRO_Culling_Node object.
The level-of-detail index for the child.
Valid range is zero through three.

Return Value

None.

Application Notes

None.



Function Reference

(LOD Culling Node Functions)

AddChild

Attaches a child polygon at specified LOD.

```
void AddChild ( PRO_Polygon *child_ptr , long child_index ) ;
```

Description

Attaches a child polygon to a PRO_LOD_Culling_Node.

Arguments

PRO_Polygon **child_ptr*
long *child_index*

A pointer to a PRO_Polygon object.
The level-of-detail index for the child.
Valid range is zero through three.

Return Value

None.

Application Notes

None.



Articulated Part Culling Node Functions

Articulated Part (6DOF) Culling Nodes are used to create geometry that can move relative to the coordinate set of the parent culling node to which they are attached. This kind of node is also known as a dynamic coordinate set or six-degree-of-freedom node.

(Articulated Part Culling Node Functions)

PRO_6DOF_Culling_Node class constructor

Creates a PRO 6DOF Culling Node object.

```
PRO_6DOF_Culling_Node ( void ) ;
```

Description

Creates an instance of a PRO_6DOF_Culling_Node.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Articulated Part Culling Node Functions)

PRO_6DOF_Culling_Node class constructor

Creates a PRO 6DOF Culling Node object.

```
PRO_6DOF_Culling_Node ( PRO_Matrix *user_matrix ) ;
```

Description

Creates an instance of a PRO_6DOF_Culling_Node to be controlled by the supplied matrix.

Arguments

PRO_Matrix **user_matrix* Pointer to a shared matrix.

Return Value

None.

Application Notes

It is the user's responsibility to delete the user matrix after the 6DOF node has been deleted.



Function Reference

(Articulated Part Culling Node Functions)

PRO_6DOF_Culling_Node class destructor

Destroys a PRO 6DOF Culling Node object.

```
~PRO_6DOF_Culling_Node ( void ) ;
```

Description

Destroys an instance of a PRO_6DOF_Culling_Node.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Articulated Part Culling Node Functions)

SetName

Names the 6DOF node.

```
void SetName ( char* name ) ;
```

Description

Sets the name of the PRO_6DOF_Culling_Node to the user specified string.

Arguments

char* *name* A pointer to a character string.

Return Value

None.

Application Notes

This name is the same that can be used to access the articulated part names from the PRO_Model class. See also the PRO_model function called StoreArticulatedPart.



Function Reference

(Articulated Part Culling Node Functions)

SetPosition

Sets the position of the 6DOF node.

```
void SetPosition ( double x , double y , double z ) ;
```

Description

Sets the position of the PRO_6DOF_Culling_Node relative to its parent's coordinate set.

Arguments

double x	The x coordinate of the 6DOF node's position.
double y	The y coordinate of the 6DOF node's position.
double z	The z coordinate of the 6DOF node's position.

Return Value

None.

Application Notes

None.



Function Reference

(Articulated Part Culling Node Functions)

SetPosition

Sets the position of the 6DOF node.

```
void SetPosition ( float x , float y , float z ) ;
```

Description

Sets the position of the PRO_6DOF_Culling_Node relative to its parent's coordinate set.

Arguments

float x	The x coordinate of the 6DOF node's position.
float y	The y coordinate of the 6DOF node's position.
float z	The z coordinate of the 6DOF node's position.

Return Value

None.

Application Notes

None.



Function Reference

(Articulated Part Culling Node Functions)

SetOrientation

Sets the orientation of the 6DOF Node.

```
void SetOrientation ( double roll , double pitch , double yaw ) ;
```

Description

Sets the orientation of the PRO_6DOF_Culling_Node relative to its parent's coordinate set.

Arguments

double <i>roll</i>	The rotation about the x axis, in degrees.
double <i>pitch</i>	The rotation about the y axis, in degrees.
double <i>yaw</i>	The rotation about the z axis, in degrees.

Return Value

None.

Application Notes

None.



Function Reference

(Articulated Part Culling Node Functions)

SetOrientation

Sets the orientation of the 6DOF Node.

```
void SetOrientation ( float roll , float pitch , float yaw ) ;
```

Description

Sets the orientation of the PRO_6DOF_Culling_Node relative to its parent's coordinate set.

Arguments

float <i>roll</i>	The rotation about the x axis, in degrees.
float <i>pitch</i>	The rotation about the y axis, in degrees.
float <i>yaw</i>	The rotation about the z axis, in degrees.

Return Value

None.

Application Notes

None.



Function Reference

(Articulated Part Culling Node Functions)

SetOrientation

Sets the orientation of the 6DOF node.

```
void SetOrientation ( float *in_matrix ) ;
```

Description

Rotates the 6DOF node by the 3x3 matrix. The meaning of the matrix elements are as follows:

0	1	2
Forward .X	Forward .Y	Forward .Z
3	4	5
Left .X	Left .Y	Left .Z
6	7	8
Up .X	Up .Y	Up .Z

Arguments

float *in_matrix

A pointer to the 3x3 matrix containing the rotations to be applied to the 6DOF node.

Return Value

None.

Application Notes

None.



Function Reference

(Articulated Part Culling Node Functions)

SetMotionExtents

Sets the motion limits for the 6DOF node.

```
void SetMotionExtents ( double xmin , double ymin , double zmin ,  
                        double xmax , double ymax , double zmax ) ;
```

Description

Sets the bounding box extents for the PRO_6DOF_Culling_Node motion.

Arguments

double <i>xmin</i>	The minimum x position of the motion bounding box.
double <i>ymin</i>	The minimum y position of the motion bounding box.
double <i>zmin</i>	The minimum z position of the motion bounding box.
double <i>xmax</i>	The maximum x position of the motion bounding box.
double <i>ymax</i>	The maximum y position of the motion bounding box.
double <i>zmax</i>	The maximum z position of the motion bounding box.

Return Value

None.

Application Notes

This call MUST be made is the node is ever going to be located at a position other than at the parent's coordinate set origin (0.0,0.0,0.0).



Function Reference

(Articulated Part Culling Node Functions)

SetMotionExtents

Sets the motion limits for the 6DOF node.

```
void SetMotionExtents ( float xmin , float ymin , float zmin ,  
                        float xmax , float ymax , float zmax ) ;
```

Description

Sets the bounding box extents for the PRO_6DOF_Culling_Node motion.

Arguments

float <i>xmin</i>	The minimum x position of the motion bounding box.
float <i>ymin</i>	The minimum y position of the motion bounding box.
float <i>zmin</i>	The minimum z position of the motion bounding box.
float <i>xmax</i>	The maximum x position of the motion bounding box.
float <i>ymax</i>	The maximum y position of the motion bounding box.
float <i>zmax</i>	The maximum z position of the motion bounding box.

Return Value

None.

Application Notes

This call MUST be made is the node is ever going to be located at a position other than at the parent's coordinate set origin (0.0,0.0,0.0).



Function Reference

(Articulated Part Culling Node Functions)

SetMatrix

Overrides API matrix with a user matrix.

```
void SetMatrix ( PRO_Matrix *user_matrix );
```

Description

Sets the pointer of the PRO_6DOF_Culling_Node's matrix to that of a user defined matrix. By default, the PRO/API will automatically assign a unique matrix to each PRO_6DOF_Culling_Node that is created by the application.

Arguments

PRO_Matrix *user_matrix	A pointer to a shared matrix.
-------------------------	-------------------------------

Return Value

None.

Application Notes

Matrices defined in this way will not be deallocated when the node's destructor is called, and it is up to the application to free up these matrices when appropriate.



Function Reference

(Articulated Part Culling Node Functions)

ReplaceMatrix

Overrides user matrix with another user matrix.

```
void ReplaceMatrix ( PRO_Matrix *user_matrix ) ;
```

Description

Replaces the node's current user matrix with a new user matrix.

Arguments

PRO_Matrix *user_matrix A pointer to a shared matrix.

Return Value

None.

Application Notes

This function is valid only when the 6DOF node has been created with a user matrix or after a previous call to SetMatrix has been made to override the PRO/API matrix.

It is the user's responsibility to delete the user matrix after the 6DOF node has been deleted.



Function Reference

(Articulated Part Culling Node Functions)

GetName

Retrieves the 6DOF's name.

```
char* GetName ( void ) ;
```

Description

Returns the name of the PRO_6DOF_Culling_Node.

Arguments

None.

Return Value

A pointer to a character string.

Application Notes

None.



Function Reference

(Articulated Part Culling Node Functions)

GetPosition

Gets the position of the 6DOF node.

```
void GetPosition ( double *x, double *y, double *z ) ;
```

Description

Gets the position of the 6DOF node. The position is relative to its parent position.

Arguments

double *x	Pointer to the x coordinate of the 6DOF node position.
double *y	Pointer to the y coordinate of the 6DOF node position.
double *z	Pointer to the z coordinate of the 6DOF node position.

Return Value

None.

Application Notes

None.



Function Reference

(Articulated Part Culling Node Functions)

GetOrientation

Gets the orientation of the 6DOF node.

```
void GetOrientation ( double *roll, double *pitch, double *yaw ) ;
```

Description

Gets the orientation of the 6DOF node. The orientation is relative to its parent orientation.

Arguments

double *roll	Pointer to the roll angle, in degrees.
double *pitch	Pointer to the pitch angle, in degrees.
double *yaw	Pointer to the yaw angle, in degrees.

Return Value

None.

Application Notes

None.



Function Reference

(Articulated Part Culling Node Functions)

GetMatrix

Retrieves the 6DOF's matrix.

```
PRO_Matrix* GetMatrix ( void ) ;
```

Description

Returns a pointer to the matrix of the PRO_6DOF_Culling_Node.

Arguments

None.

Return Value

A pointer to the 6DOF node's PRO_Matrix object.

Application Notes

None.



Animation Culling Node Functions

Animation Culling Nodes are used to manage sets of models representing frames of an animation, or pieces of models that can be turned on and off as required by the host application. The `PRO_Animation_Culling_Node` class creates a `PRO_Animation_Sequence` object that can also be used to transfer control of the animation sequencing to the PRO/API.

(Animation Culling Node Functions)

PRO_Animation_Culling_Node class constructor

Creates a PRO Animation Culling Node object.

```
PRO_Animation_Culling_Node ( long number_of_frames , char* name ) ;
```

Description

Creates an instance of a `PRO_Animation_Culling_Node` with the specified number of frames of animation.

Arguments

long <i>number_of_frames</i>	The number of frames in the animation.
char* <i>name</i>	The name of the animation.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Culling Node Functions)

PRO_Animation_Culling_Node class destructor
Destroys a PRO Animation Culling Node object.

```
~PRO_Animation_Culling_Node ( void ) ;
```

Description

Destroys an instance of a PRO_Animation_Culling_Node.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Culling Node Functions)

AddChild

Attaches a child culling node for the specified frame.

```
void AddChild ( PRO_Culling_Node *child_ptr , long frame_number ) ;
```

Description

Attaches a child culling node to a PRO_Animation_Culling_Node.

Arguments

PRO_Culling_Node **child_ptr*
long *frame_number*

A pointer to a PRO_Culling_Node object.
The animation frame number at which the child node will be attached

Return Value

None.

Application Notes

None.



Function Reference

(Animation Culling Node Functions)

AddChild

Attaches a child polygon for specified frame.

```
void AddChild ( PRO_Polygon *child_ptr , long frame_number ) ;
```

Description

Attaches a child polygon to a PRO_Animation_Culling_Node.

Arguments

PRO_Culling_Node **child_ptr*
long *frame_number*

A pointer to a PRO_Culling_Node object.
The animation frame number at which the child node will be attached.

Return Value

None.

Application Notes

None.



Function Reference

(Animation Culling Node Functions)

SetName

Names the animation sequence.

```
void SetName ( char* name ) ;
```

Description

Sets the name of the PRO_Animation_Culling_Node to the *name*.

Arguments

char* *name* A pointer to a character string.

Return Value

None.

Application Notes

This name is the same name used when accessing the animation sequences through the model to which this node is associated. See also the PRO_Model function StoreAnimationSequence.



Function Reference

(Animation Culling Node Functions)

GetName

Retrieves the name of the animation node.

```
char* GetName ( void ) ;
```

Description

Returns the name of the PRO_Animation_Culling_Node.

Arguments

None.

Return Value

A pointer to the name of the node.

Application Notes

None.



Function Reference

(Animation Culling Node Functions)

GetAnimationSequence

Retrieves the associated PRO Animation Sequence.

```
PRO_Animation_Sequence* GetAnimationSequence ( void ) ;
```

Description

Returns a pointer to the PRO_Animation_Sequence object.

Arguments

None.

Return Value

A PRO_Animation_Sequence pointer.

Application Notes

None.



Instance Culling Node Functions

Instance Culling Nodes are used to define the top of a shared display list segment that can be referenced later from other parts of the scene display list. They are useful in reducing the polygon storage requirements for replicated features, such as trees, by storing a single copy of the geometry, and simply referencing (and locating) it multiple times with little storage overhead.

(Instance Culling Node Functions)

PRO_Instance_Culling_Node

Creates the top of a shared geometry set.

```
PRO_Instance_Culling_Node ( void ) ;
```

Description

Creates an instance of a PRO_Instance_Culling_Node. This node defines the top of a shared portion of the display list and may be attached only to instance reference nodes.

Arguments

None.

Return Value

None.

Application Notes

An Instance node must have all of its geometry attached to it before it is attached to ANY PRO_Instance_Reference_Culling_Node objects.



Function Reference

(Instance Culling Node Functions)

ProcessImmediateModeData

Flushes immediate mode polygons to the hardware.

```
void ProcessImmediateModeData ( void ) ;
```

Description

Processes the data attached to this instance culling node and flushes it to the hardware.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Instance Culling Node Functions)

EnableVertexSharing

Enables vertex sharing among shared geometry polygons.

```
void EnableVertexSharing ( void ) ;
```

Description

Enables vertex sharing among polygons of the shared geometry. By default, vertex sharing is disabled.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Instance Culling Node Functions)

DisableVertexSharing

Disables vertex sharing among shared geometry polygons.

```
void DisableVertexSharing ( void ) ;
```

Description

Disables vertex sharing among polygons of the shared geometry. This is the default state for processing shared geometry.

Arguments

None.

Return Value

None.

Application Notes

None.



Instance Reference Culling Node Functions

Instance Reference Culling Nodes are used to place the geometry linked to PRO_Instance_Culling_Node objects in the display list. The shared geometry is processed as if it were located at a given position with little storage overhead. Instance references may not have other nodes attached as children.

(Instance Reference Culling Node Functions)

PRO_Instance_Reference_Culling_Node class constructor

Creates a instance reference node.

```
PRO_Instance_Reference_Culling_Node  
    ( PRO_Instance_Culling_Node* instanced_geometry,  
      float x_instance_location,  
      float y_instance_location,  
      float z_instance_location ) ;
```

Description

Creates an instance of an instance reference. Instance references are really culling nodes that serve as links to PRO_Instance_Culling_Node objects, except that they provide a means of placing the shared geometry at any desired position in the database.



Function Reference

Arguments

<code>PRO_Instance_Culling_Node* <i>instanced_geometry</i></code>	A pointer to a <code>PRO_Instance_Culling_Node</code> .
<code>float <i>x_instance_location</i></code>	The x position of the <code>PRO_Instance_Culling_Node</code> .
<code>float <i>y_instance_location</i></code>	The y position of the <code>PRO_Instance_Culling_Node</code> .
<code>float <i>z_instance_location</i></code>	The z position of the <code>PRO_Instance_Culling_Node</code> .

Return Value

None.

Application Notes

The default position for all reference nodes is 0.0,0.0,0.0.

An Instance node must have all of its geometry attached to it before it is attached to ANY `PRO_Instance_Reference_Culling_Node` objects.



Function Reference

(Instance Reference Culling Node Functions)

SetInstanceLocation

Defines the position of an instance reference.

```
void SetInstanceLocation ( float centroid_x, float centroid_y, float centroid_z );
```

Description

Defines the position of a reference to shared geometry.

Arguments

float <i>centroid_x</i>	The x position of the instance reference.
float <i>centroid_y</i>	The y position of the instance reference.
float <i>centroid_z</i>	The z position of the instance reference.

Return Value

None.

Application Notes

None.



Function Reference

(Instance Reference Culling Node Functions)

GetInstanceLocation

Gets the position of an instance reference.

```
void GetInstanceLocation ( float *centroid_x, float *centroid_y, float *centroid_z );
```

Description

Defines the position of a reference to shared geometry.

Arguments

float *centroid_x	Pointer to the x position of the instance reference.
float *centroid_y	Pointer to the y position of the instance reference
float *centroid_z	Pointer to the z position of the instance reference

Return Value

None.

Application Notes

None.



Point Light Functions

The Point Light class is used to create instances of point luminous features. Point lights are handled as polygons that always face the viewpoint and are not affected by lighting, so they seem luminous during night scenes.

(Point Light Functions)

PRO_Point_Light class constructor

Creates an instance of a point light.

```
PRO_Point_Light ( float x , float y , float z ) ;
```

Description

Creates an instance of a point light. Point lights are rendered as self-illuminating polygons of a user defined detail.

Arguments

<i>x</i>	The x location of the point light.
<i>y</i>	The y location of the point light.
<i>z</i>	The z location of the point light.

Return value

None.

Application Notes

The point light blend type used to control its clamped size on the screen, its detail (number of vertices) and its actual physical size are controlled by the following functions:

```
PRO_SetPointLightDetail ( long number_of_sides ) ;
```

```
PRO_SetPointLightFeatureType ( long rtsl ) ;
```

```
PRO_SetPointLightSize ( float size ) ;
```

See also PRO_SetPointLightPolygon



Function Reference

(Point Light Functions)

PRO_Point_Light class destructor

Destroys an instance of a point light.

```
~PRO_Point_Light ( void ) ;
```

Description

Destroys an instance of a PRO_Point_Light.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Point Light Functions)

SetColor

Defines the color of the light.

```
void SetColor ( float red , float green , float blue ) ;
```

Description

Defines the color of the light. Point lights are white by default.

Arguments

float <i>red</i>	The normalized (0.0 to 1.0) red component of the point light color.
float <i>green</i>	The normalized (0.0 to 1.0) green component of the point light color.
float <i>blue</i>	The normalized (0.0 to 1.0) blue component of the point light color.

Return Value

None.

Application Notes

None.



Function Reference

(Point Light Functions)

SetColor

Defines the color of the light.

```
void SetColor ( long red , long green , long blue ) ;
```

Description

Defines the color of the light. Point lights are white by default.

Arguments

long <i>red</i>	The integer (0 to 255) red component of the point light color.
long <i>green</i>	The integer (0 to 255) green component of the point light color.
long <i>blue</i>	The integer (0 to 255) blue component of the point light color.

Return Value

None.

Application Notes

None.



Function Reference

(Point Light Functions)

SetPosition

Defines the position of the light.

```
void SetPosition ( float centroid_x, float centroid_y, float centroid_z );
```

Description

Defines the position of the point light.

Arguments

float <i>centroid_x</i>	The x position of the point light.
float <i>centroid_y</i>	The y position of the point light.
float <i>centroid_z</i>	The z position of the point light.

Return Value

None.

Application Notes

None.



Function Reference

(Point Light Functions)

GetPosition

Gets the position of the light.

```
void GetPosition ( float *centroid_x, float *centroid_y, float *centroid_z ) ;
```

Description

Defines the position of the point light.

Arguments

float *centroid_x	Pointer to the x position of the point light.
float *centroid_y	Pointer to the y position of the point light.
float *centroid_z	Pointer to the z position of the point light.

Return Value

None.

Application Notes

None.



Instance Set Functions

The Instance Set class is used to create a group of instances of point features. The individual instances in the set can be used to simulate effects such as dust particles.

(Instance Set Functions)

PRO_Instance_Set class constructor

Creates an instance of an instance set.

```
PRO_Instance_Set ( long number_of_instances );
```

Description

Creates an instance of an Instance Set.

Arguments

<i>long number_of_instances</i>	The number of instance entities in the group.
---------------------------------	---

Return value

None.



Function Reference

(Instance Set Functions)

PRO_Instance_Set class destructor

Destroys an instance of an instance set.

```
~PRO_Instance_Set ( void ) ;
```

Description

Destroys an instance of an Instance Set.

Arguments

None.

Return value

None.



Function Reference

(Instance Set Functions)

SetColor

Defines the color of the instances.

```
void SetColor ( float red , float green , float blue ) ;
```

Description

Defines the color of the instances in the set.

Arguments

float <i>red</i>	The normalized (0.0 to 1.0) red component of the color.
float <i>green</i>	The normalized (0.0 to 1.0) green component of the color.
float <i>blue</i>	The normalized (0.0 to 1.0) blue component of the color.

Return Value

None.

Application Notes

None.



Function Reference

(Instance Set Functions)

SetColor

Defines the color of the instances.

```
void SetColor ( long red , long green , long blue ) ;
```

Description

Defines the color of the instances in the set.

Arguments

long <i>red</i>	The integer (0 to 255) red component of the color.
long <i>green</i>	The integer (0 to 255) green component of the color.
long <i>blue</i>	The integer (0 to 255) blue component of the color.

Return Value

None.

Application Notes

None.



Function Reference

(Instance Set Functions)

SetPosition

Defines the position of a set member.

```
void SetPosition ( float centroid_x, float centroid_y, float centroid_z, long index );
```

Description

Defines the position of an instance in the set.

Arguments

float <i>centroid_x</i>	The x position of the instance.
float <i>centroid_y</i>	The y position of the instance.
float <i>centroid_z</i>	The z position of the instance.
long <i>index</i>	The index identifying the instance in the set to be positioned.

Return Value

None.

Application Notes

None.



Function Reference

(Instance Set Functions)

GetPosition

Gets the position of a set member.

```
void GetPosition ( float *centroid_x, float *centroid_y, float *centroid_z, long index ) ;
```

Description

Defines the position of an instance in the set.

Arguments

float *centroid_x	Pointer to the x position of the point light.
float *centroid_y	Pointer to the y position of the point light.
float *centroid_z	Pointer to the z position of the point light.
long index	The index identifying the instance in the set.

Return Value

None.

Application Notes

None.



Function Reference

(Instance Set Functions)

SetActiveInstanceCount

Sets the number of active instances.

```
void SetActiveInstanceCount ( long count ) ;
```

Description

Sets the number of active instances in the set.

Arguments

long <i>count</i>	The number of active instances.
-------------------	---------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Instance Set Functions)

GetActiveInstanceCount

Gets the number of active instances.

```
long GetActiveInstanceCount ( void ) ;
```

Description

Gets the number of active instances in the set.

Arguments

None.

Return Value

The number of active instances.

Application Notes

None.



Function Reference

(Instance Set Functions)

GetNumberOfInstances

Gets the number of instances.

```
long GetNumberOfInstances ( void ) ;
```

Description

Gets the total number of instances in the set.

Arguments

None.

Return Value

The number of instances in the set.

Application Notes

None.



Polygon Functions

The Polygon Class functions are used to define the geometry of and assign attributes to polygons.

(Polygon Functions)

PRO_Polygon class constructor

Creates an instance of a PRO Polygon.

```
PRO_Polygon ( void ) ;
```

Description

Creates an instance of a PRO_Polygon.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

PRO_Polygon class destructor

Destroys an instance of a PRO Polygon.

```
~PRO_Polygon ( void ) ;
```

Description

Destroys an instance of a PRO_Polygon.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

AddVertex

Adds a vertex to the definition of a polygon.

```
void AddVertex ( PRO_Vertex *vertex_ptr );
```

Description

Adds a vertex to the definition of a polygon.

Arguments

PRO_Vertex **vertex_ptr* A pointer to a PRO_Vertex object.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

AddChild

Attaches a child polygon to a polygon.

```
void AddChild ( PRO_Polygon *child_ptr ) ;
```

Description

Attaches a child polygon to a PRO_Polygon object. The child polygon will have a higher drawing priority than its parent.

Arguments

PRO_Polygon *child_ptr	A pointer to a PRO_Polygon object.
------------------------	------------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

DetachChild

Detaches a child polygon from a polygon.

```
void DetachChild ( PRO_Polygon *child_ptr ) ;
```

Description

Detaches a child polygon from its parent polygon.

Arguments

PRO_Polygon * <i>child_ptr</i>	A pointer to a PRO_Polygon object.
--------------------------------	------------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

AttachTexture

Associates the polygon with a texture map.

```
void AttachTexture ( PRO_Texture *texture_map ) ;
```

Description

Associates the polygon with a texture map.

Arguments

PRO_Texture *texture_map	A pointer to a PRO_Texture object.
--------------------------	------------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

DetachTexture

Disassociates the polygon from a texture map.

```
void DetachTexture ( void ) ;
```

Description

Breaks the association between the polygon and a texture map.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

AttachMicroTexture

Associates the polygon with a microtexture map.

```
void AttachMicroTexture ( PRO_MicroTexture *microtexture_map ) ;
```

Description

Associates the polygon with a microtexture map.

Arguments

PRO_MicroTexture *microtexture_map A pointer to a PRO_MicroTexture object.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

DetachMicroTexture

Disassociates the polygon from a microtexture map.

```
void DetachMicroTexture ( void ) ;
```

Description

Breaks the association between the polygon and a microtexture map.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

SetColor

Sets the polygon color.

```
void SetColor ( long red, long green, long blue ) ;
```

Description

Sets the polygon color to the specified RGB value.

Arguments

long <i>red</i>	The intensity of the red component of the color. Valid range is 0 through 255.
long <i>green</i>	The intensity of the green component of the color. Valid range is 0 through 255.
long <i>blue</i>	The intensity of the blue component of the color. Valid range is 0 through 255.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

SetColor

Sets the polygon color.

```
void SetColor ( float red, float green, float blue ) ;
```

Description

Sets the polygon color to the specified RGB value.

Arguments

float <i>red</i>	The intensity of the red component of the color. Valid range is 0.0 through 1.0.
float <i>green</i>	The intensity of the green component of the color. Valid range is 0.0 through 1.0.
float <i>blue</i>	The intensity of the blue component of the color. Valid range is 0.0 through 1.0.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

SetColorIndex

Sets the out-the-window color table index.

```
void SetColorIndex ( long color_index ) ;
```

Description

Sets the color table index for a polygon.

Arguments

long <i>index</i>	The color table index. Valid range is 0 through (color table size – 1). The default and maximum color table size is 4096.
-------------------	---

Return Value

None.

Application Notes

There must be a color table defined for a culling node above this polygon in the hierarchy.



Function Reference

(Polygon Functions)

SetSensorColorIndex

Sets the sensor color table index.

```
void SetSensorColorIndex ( long color_index );
```

Description

Sets the sensor color table index for a polygon.

Arguments

long <i>index</i>	The color table index.
-------------------	------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

EnableLightingEffects

Enables lighting effects on the polygon.

```
void EnableLightingEffects ( void ) ;
```

Description

Enables lighting effects on the polygon. This indicates that the polygon is not luminous and will be affected by shading, ambient light, and sun intensity. This is the default for polygons.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

DisableLightingEffects

Disables lighting effects on the polygon.

```
void DisableLightingEffects ( void ) ;
```

Description

Disables lighting effects on the polygon. This indicates that the polygon is luminous and will not be affected by shading, ambient light, and sun intensity.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

SetTranslucency

Sets the translucency value of a polygon.

```
void SetTranslucency ( float translucency ) ;
```

Description

Sets the translucency of a polygon.

Arguments

float <i>translucency</i>	Amount of translucency applied to polygon. Values range from 0.0 to 1.0, with 0.0 being fully translucent and 1.0 being fully opaque. The default setting for all new polygons is 1.0.
---------------------------	--

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

SetLightModifier

Sets the amount that a luminous polygon will burn through fog.

```
void SetLightModifier ( float percent_burn_thru ) ;
```

Description

Sets the amount of effect that fog effects will have on a luminous polygon.

Arguments

float <i>percent_burn_thru</i>	Amount that a luminous polygon will burn through fog. Valid range is 0.0 to 1.0. 0.0 is completely fogged; 1.0 has no fog.
--------------------------------	--

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

SetHighPriority

Indicates that the polygon has higher priority than others in scene.

```
void SetHighPriority ( long high_priority ) ;
```

Description

Indicates that the polygon has higher priority than others in the scene, and therefore behaves as if it were in an overlay plane.

Arguments

long <i>high_priority</i>	High priority flag. 1 sets the high priority attribute. 0 clears the high priority attribute. By default, polygons are not high priority.
---------------------------	---

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

PolygonIsLayered

Indicates a stencil polygon.

```
void PolygonIsLayered ( void ) ;
```

Description

Indicates that the polygon is layered which temporarily disable z-buffer operations while the polygon is being rendered.

Arguments

None.

Return Value

None.

Application Notes

This function is normally not called by applications unless the application itself takes over the drawing order of the polygons.



Function Reference

(Polygon Functions)

IsDoubleSided

Sets the double-sided attribute of a polygon.

```
void IsDoubleSided ( void ) ;
```

Description

Indicates that a polygon is double-sided. The default state for the polygon is single-sided.

Arguments

None.

Return Value

None.

Application Notes

None



Function Reference

(Polygon Functions)

DoSmoothShading

Indicates that the polygon will be smooth shaded.

```
void DoSmoothShading ( void ) ;
```

Description

Sets the polygon to use its vertex normals to apply shading to the face. The default shading mode for all polygons is flat shading.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

DoFlatShading

Indicates that the polygon will be flat shaded.

```
void DoFlatShading ( void ) ;
```

Description

Sets the polygon to use its polygon normal to apply shading to the face. This is the default setting for all new polygons.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

IsNotShaded

Indicates that the polygon will not be shaded.

```
void IsNotShaded ( void ) ;
```

Description

Sets the polygon to have no shading applied to the face.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

VerticesAreClockwise

Sets the direction of vertex ordering.

```
void VerticesAreClockwise ( void ) ;
```

Description

Indicates that the direction of vertex ordering is clockwise. The default convention for polygon vertex ordering is clockwise.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

VerticesAreCounterClockwise

Sets the direction of vertex ordering.

```
void VerticesAreCounterClockwise ( void ) ;
```

Description

Indicates that the direction of vertex ordering is counter-clockwise.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

SetXSmoothing

Enables texture wrap blending in the x (u) direction.

```
void SetXSmoothing ( void ) ;
```

Description

Sets the smoothing of the texture wraps along the x axis of the texture. This will smooth the edges of the texture when it is wrapped. By default, texture is not smoothed across wraps.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

SetYSmoothing

Enables texture wrap blending in the y (v) direction.

```
void SetYSmoothing ( void ) ;
```

Description

Sets the smoothing of the texture wraps along the y axis of the texture. This will smooth the edges of the texture when it is wrapped. By default, texture is not smoothed across wraps.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

SetXMirror

Enables mirroring in the x (u) direction.

```
void SetXMirror ( void ) ;
```

Description

Sets the x-mirror flag for the texture wrapping. The texture will be mirrored along its x-axis when it wraps. By default, texture is not mirrored.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

SetYMirror

Enables mirroring in the y (v) direction.

```
void SetYMirror ( void ) ;
```

Description

Sets the y-mirror flag for the texture wrapping. The texture will be mirrored along its y-axis when it wraps. By default, texture is not mirrored.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

SetNPScale

Biases the texture sampling rate.

```
void SetNPScale ( float np_scale );
```

Description

Sets the texture lod scale for the polygon. A value greater than 1 will increase the range of the transition of texture level of detail. A value less than 1 will decrease the range of the transition of texture level of detail.

Arguments

float <i>np_scale</i>	The scale value.
-----------------------	------------------

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

SetTranslucencyPatternSelect

Sets the blend/translucency pixel hole pattern select.

```
void SetTranslucencyPatternSelect ( long pattern_select ) ;
```

Description

Sets the polygons translucency pattern select used by the hardware while processing translucent polygons. Translucency is achieved by inducing holes in the polygon pixel mask. The pattern select defines which of two possible patterns should be used for a given polygon.

Arguments

long <i>pattern_select</i>	0 or 1. The pattern selected by “0” has the pixels and holes inverted from the pattern selected by “1”.
----------------------------	---

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

EnableAutoVertexDelete

Indicates that vertices will be deleted with polygon.

```
void EnableAutoVertexDelete ( void ) ;
```

Description

Indicates that vertices associated with the polygon will be deleted automatically when polygon object is deleted.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

DisableAutoVertexDelete

Indicates that vertices will not be deleted with polygon.

```
void DisableAutoVertexDelete ( void ) ;
```

Description

Indicates that vertices associated with the polygon will not be deleted automatically when polygon object is deleted. Vertex object deletion is the responsibility of the application. This is the default state for all polygons unless otherwise defined by the application.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Polygon Functions)

GetLODNumber

Gets the LOD number of a polygon.

```
long GetLODNumber ( void ) ;
```

Description

Returns the face LOD number of a polygon.

Arguments

None.

Return Value

The LOD number assigned to the polygon.

Application Notes

None.



Function Reference

(Polygon Functions)

GetVertexCount

Gets the number of vertices in a polygon.

```
long GetVertexCount ( void ) ;
```

Description

Returns the number of vertices associated with a polygon before vertex sharing.

Arguments

None.

Return Value

The number of vertices in the polygon.

Application Notes

None.



Function Reference

(Polygon Functions)

GetCompressedVertexCount

Gets the number of vertices in a polygon after vertex sharing.

```
long GetCompressedVertexCount ( void ) ;
```

Description

Returns the number of vertices associated with a polygon after vertex sharing.

Arguments

None.

Return Value

The number of vertices in the polygon after vertex sharing.

Application Notes

None.



Function Reference

(Polygon Functions)

GetVertex

Gets a vertex.

```
PRO_Vertex *GetVertex ( long vertex_number );
```

Description

Returns a pointer to the PRO_Vertex object associated with the specified vertex in the polygon definition.

Arguments

long *vertex_number*

Index describing the vertex number to be retrieved. Valid range is 0 through (GetVertexCount – 1).

Return Value

A pointer to a PRO_Vertex object.

Application Notes

None.



Function Reference

(Polygon Functions)

IsTextured

Tests for valid texture attachment.

```
long IsTextured ( void ) ;
```

Description

Returns 1 if the polygon has a texture applied to it.

Arguments

None.

Return Value

1 if the polygon is textured.
0 if the polygon is not textured.

Application Notes

None.



Vertex Functions

The Vertex Class functions are used to define and manipulate a vertex.

(Vertex Functions)

PRO_Vertex class constructor

Creates an instance of a PRO Vertex.

```
PRO_Vertex ( float x, float y, float z ) ;
```

Description

Creates an instance of a PRO_Vertex with the specified vertex coordinates.

Arguments

float x	The x coordinate of a vertex.
float y	The y coordinate of a vertex.
float z	The z coordinate of a vertex.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

PRO_Vertex class constructor

Creates an instance of a PRO_Vertex.

```
PRO_Vertex ( void ) ;
```

Description

Creates an instance of a PRO_Vertex.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

PRO_Vertex class destructor

Destroys an instance of a PRO Vertex.

```
~PRO_Vertex ( void ) ;
```

Description

Destroys an instance of a PRO_Vertex.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

SetCoordinates

Sets the coordinates of a vertex.

```
void SetCoordinates ( float x, float y, float z ) ;
```

Description

Sets the coordinates of a vertex to the specified x,y,z values.

Arguments

float x	The x coordinate of a vertex.
float y	The y coordinate of a vertex.
float z	The z coordinate of a vertex.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

SetCoordinates

Sets the coordinates of a vertex.

```
void SetCoordinates ( double x, double y, double z ) ;
```

Description

Sets the coordinates of a vertex to the specified x,y,z values.

Arguments

double x	The x coordinate of a vertex.
double y	The y coordinate of a vertex.
double z	The z coordinate of a vertex.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

SetMorphCoordinates

Sets the vertex coordinates without rounding.

```
void SetMorphCoordinates ( float x, float y, float z ) ;
```

Description

Sets the vertex coordinates. This function does not perform the rounding error checks made by SetCoordinates(), with the possible side effect of geometry mismatches between polygons with shared geometry but attached to different culling nodes.

Arguments

float x	The x coordinate of a vertex.
float y	The y coordinate of a vertex.
float z	The z coordinate of a vertex.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

SetTextureCoordinates

Sets the texture coordinates of a vertex.

```
void SetTextureCoordinates ( float x, float y ) ;
```

Description

Sets the texture coordinates of a vertex to the specified values. Texture coordinates are relative to the lower left corner of the texture map.

Arguments

float x	The normalized x texture coordinate of a vertex.
float y	The normalized y texture coordinate of a vertex.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

SetVertexNormal

Sets the vertex normal of a vertex.

```
void SetVertexNormal ( float normal_x, float normal_y, float normal_z ) ;
```

Description

Sets the vertex normal to the specified x,y,z values.

Arguments

float <i>normal_x</i>	The x coordinate of vertex normal.
float <i>normal_y</i>	The y coordinate of vertex normal.
float <i>normal_z</i>	The z coordinate of vertex normal.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

SetFixedShadingIntensity

Sets a fixed lighting value for the vertex.

```
void SetFixedShadingIntensity ( float vertex_intensity ) ;
```

Description

Sets a fixed lighting value for the vertex.

Arguments

float <i>vertex_intensity</i>	Lighting value.
-------------------------------	-----------------

Return Value

None.

Application Notes

Polygons whose vertices all have fixed intensities will not be affected by lighting parameters.



Function Reference

(Vertex Functions)

OffsetVertex

Defines a vector for translating the vertex.

```
void OffsetVertex ( float x_offset, float y_offset, float z_offset ) ;
```

Description

Defines a vector for translating the vertex coordinates.

Arguments

float <i>x_offset</i>	The x component of the offset vector.
float <i>y_offset</i>	The y component of the offset vector.
float <i>z_offset</i>	The z component of the offset vector.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

ScaleVertex

Defines a vector for scaling.

```
void ScaleVertex ( float x_scale, float y_scale, float z_scale ) ;
```

Description

Defines a vector for scaling the vertex coordinates.

Arguments

float <i>x_scale</i>	The x scaling factor for the vertex.
float <i>y_scale</i>	The y scaling factor for the vertex.
float <i>z_scale</i>	The z scaling factor for the vertex.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

GetCoordinates

Gets the coordinates of a vertex.

```
void GetCoordinates ( float &x, float &y, float &z ) ;
```

Description

Gets the coordinates of a vertex.

Arguments

float &x	The x coordinate of a vertex.
float &y	The y coordinate of a vertex.
float &z	The z coordinate of a vertex.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

GetCoordinates

Gets the coordinates of a vertex.

```
void GetCoordinates ( double &x, double &y, double &z ) ;
```

Description

Gets the coordinates of a vertex.

Arguments

double &x	The x coordinate of a vertex.
double &y	The y coordinate of a vertex.
double &z	The z coordinate of a vertex.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

GetTextureCoordinates

Gets the texture coordinates of a vertex.

```
void GetTextureCoordinates ( float &x, float &y );
```

Description

Gets the texture coordinates of a vertex.

Arguments

float &x	The normalized x texture coordinate of a vertex.
float &y	The normalized y texture coordinate of a vertex.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

GetVertexNormal

Gets the vertex normal of a vertex.

```
void GetVertexNormal ( float &normal_x, float &normal_y, float &normal_z );
```

Description

Gets the vertex normal.

Arguments

float &normal_x	The x coordinate of vertex normal.
float &normal_y	The y coordinate of vertex normal.
float &normal_z	The z coordinate of vertex normal.

Return Value

None.

Application Notes

None.



Function Reference

(Vertex Functions)

IsTextured

Returns whether vertex is textured.

```
long IsTextured ( void ) ;
```

Description

Indicates whether or not the vertex is textured.

Arguments

None.

Return Value

One if vertex is textured, or zero if it is not.

Application Notes

None.



Function Reference

(Vertex Functions)

IsShaded

Returns whether vertex is shaded.

```
long IsShaded ( void ) ;
```

Description

Indicates whether or not the vertex is shaded (has a vertex normal).

Arguments

None.

Return Value

One if vertex is shaded, or zero if it is not.

Application Notes

None.



Texture Functions

Texture class functions are used to define an image that will be used as a texture map for polygons. The origin of an image is assumed to be its lower left corner.

(Texture Functions)

PRO_Texture class constructor

Creates an instance of a PRO Texture.

```
PRO_Texture ( unsigned long   image_size_x,  
              unsigned long   image_size_y,  
              unsigned char   *red_band,  
              unsigned char   *green_band,  
              unsigned char   *blue_band,  
              unsigned char   *alpha_band,  
              long             enable_auto_alpha_detect ) ;
```

Description

Creates an instance of a PRO_Texture for a color image. If the image does not have alpha band data, a NULL pointer should be passed for the alpha band pointer. If no alpha band pointer argument is provided, it will default to NULL.

When automatic alpha data detection is enabled, every data value in the alpha band will be examined in order to determine the hardware format which will result in optimal image quality. This feature is disabled by default.



Function Reference

Arguments

unsigned long <i>image_size_x</i>	Size of the texture map in the x direction. Valid values are 32, 64, 128, 256, 512, and 1024.
unsigned long <i>image_size_y</i>	Size of the texture map in the y direction. Valid values are 32, 64, 128, 256, 512, and 1024.
unsigned char <i>*red_band</i>	Pointer to an array of red component values for a texture.
unsigned char <i>*green_band</i>	Pointer to an array of green component values for a texture.
unsigned char <i>*blue_band</i>	Pointer to an array of blue component values for a texture.
unsigned char <i>*alpha_band</i>	Pointer to an array of alpha channel values for a texture. This argument defaults to NULL.
long <i>enable_auto_alpha_detect</i>	Enables automatic detection of valid translucency values. This flag defaults to 0 (false).

Return Value

None.

Application Notes

The pointers to image band data are used directly by the PRO/API, so addresses of temporary variables should not be used.

Automatic detection of alpha channel data significantly affects the amount of time required to process a texture map and should not be enabled unless the application cannot make its own determination about the validity of alpha band data and the dynamic range of the translucency values. If this feature is enabled, the threshold between opacity and translucency should be set by the application using the `SetContourThreshold` function.



Function Reference

(Texture Functions)

PRO_Texture class constructor

Creates an instance of a PRO Texture.

```
PRO_Texture ( unsigned long   image_size_x,  
              unsigned long   image_size_y,  
              unsigned char    *intensity_band,  
              unsigned char    *alpha_band,  
              long             enable_auto_alpha_detect ) ;
```

Description

Creates an instance of a PRO_Texture for a monochrome image. If the image does not have alpha band data, a NULL pointer should be passed for the alpha band pointer. If no alpha band pointer argument is provided, it will default to NULL.

When automatic alpha data detection is enabled, every data value in the alpha band will be examined in order to determine the hardware format which will result in optimal image quality. This feature is disabled by default.

Arguments

unsigned long <i>image_size_x</i>	Size of the texture map in the x direction. Valid values are 32, 64, 128, 256, 512, and 1024.
unsigned long <i>image_size_y</i>	Size of the texture map in the y direction. Valid values are 32, 64, 128, 256, 512, and 1024.
unsigned char <i>*intensity_band</i>	Pointer to an array of monochrome component values for a texture.
unsigned char <i>*alpha_band</i>	Pointer to an array of alpha channel values for a texture. This argument defaults to NULL.
long <i>enable_auto_alpha_detect</i>	Enables automatic detection of valid translucency values. This flag defaults to 0 (false).



Function Reference

Return Value

None.

Application Notes

The pointers to image band data are used directly by the PRO/API, so addresses of temporary variables should not be used.

Automatic detection of alpha channel data significantly affects the amount of time required to process a texture map and should not be enabled unless the application cannot make its own determination about the validity of alpha band data and the dynamic range of the translucency values. If this feature is enabled, the threshold between opacity and translucency should be set by the application using the `SetContourThreshold` function.



Function Reference

(Texture Functions)

PRO_Texture class destructor

Destroys an instance of a PRO Texture.

```
~PRO_Texture ( void ) ;
```

Description

Destroys an instance of a PRO_Texture.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Texture Functions)

IsContourTexture

Sets the contour texture attribute of a texture map.

```
void IsContourTexture ( void ) ;
```

Description

Indicates that a texture is contour texture.

Arguments

None.

Return Value

None.

Application Notes

Contour texture is defined as a single bit texture translucency mask with anti-aliased translucent edges.



Function Reference

(Texture Functions)

SetContourThreshold

Sets the breakpoint between opacity and translucency.

```
voidSetContourThreshold ( long threshold ) ;
```

Description

Sets the breakpoint between opacity and translucency for a texture map. Values above the threshold are considered totally opaque and values below the threshold are considered totally translucent. This is used when a texture map has been declared to be contour texture. The default threshold is 63.

Arguments

<code>long <i>threshold</i></code>	Threshold between opacity and translucency. The range of values is 0 to 255.
------------------------------------	--

Return Value

None.

Application Notes

Function `IsContourTexture()` should be called for this texture map.



Function Reference

(Texture Functions)

SetContourThreshold

Sets the breakpoint between opacity and translucency.

```
voidSetContourThreshold ( float threshold ) ;
```

Description

Sets the breakpoint between opacity and translucency for a texture map. Values above the threshold are considered totally opaque and values below the threshold are considered totally translucent. This is used when a texture map has been declared to be contour texture. The default threshold is 63.

Arguments

float <i>threshold</i>	Threshold between opacity and translucency. The range of values is 0.0 to 1.0.
------------------------	--

Return Value

None.

Application Notes

Function `IsContourTexture()` should be called for this texture map.



Function Reference

(Texture Functions)

EnableAlpha

Enables alpha channel processing.

```
void EnableAlpha ( void ) ;
```

Description

Indicates that alpha channel data should be processed. This is the default for all texture maps with valid translucency data.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Texture Functions)

DisableAlpha

Disables alpha channel processing.

```
void DisableAlpha ( void ) ;
```

Description

Indicates that alpha channel data should not be processed. By default, translucency data processing is enabled for all texture maps with valid translucency data.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Texture Functions)

SetTextureFileName

Sets the name of the source texture file.

```
void SetTextureFileName ( char *filename ) ;
```

Description

Sets the name of the file containing the source texture data.

Arguments

<code>char *filename</code>	Pointer to the file name string.
-----------------------------	----------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Texture Functions)

KeepDataAfterLoading

Indicates that map should be retained after loading.

```
void KeepDataAfterLoading ( void ) ;
```

Description

Indicates that the texture map data should be retained in memory after loading the map to the PRO-1000 hardware. This is the default.

Arguments

None.

Return Value

None.

Application Notes

In this case, it is up to the application to free or delete the image data AFTER the PRO/API has processed the texture map, that is, after PRO_DisplayDatabase has been called at least once since the creation of the PRO_Texture object using the image data.



Function Reference

(Texture Functions)

FreeDataAfterLoading

Indicates that map should be released after loading.

```
void FreeDataAfterLoading ( void ) ;
```

Description

Indicates that the texture map data should be released after loading the map to the PRO-1000 hardware. This routine should be used when the application has allocated the texture data by using the “malloc” system function. The default is to retain the data after loading.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Texture Functions)

DeleteDataAfterLoading

Indicates that map should be released after loading.

```
void DeleteDataAfterLoading ( void ) ;
```

Description

Indicates that the texture map data should be released after loading the map to the PRO-1000 hardware. This routine should be used when the application has allocated the texture data through the C++ “new” operator. The default is to retain the data after loading.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Texture Functions)

GetTextureFileName

Gets the name of the source texture file.

```
char *GetTextureFileName ( void ) ;
```

Description

Gets the source filename assigned to the texture map.

Arguments

None.

Return Value

A pointer to the name string.

Application Notes

None.



Function Reference

(Texture Functions)

GetOrigTextureMapWidth

Gets the original image width.

```
unsigned long GetOrigTextureMapWidth ( void ) ;
```

Description

Gets the user defined texture map width.

Arguments

None.

Return Value

The original image width.

Application Notes

None.



Function Reference

(Texture Functions)

GetOrigTextureMapHeight

Gets the original image height.

```
unsigned long GetOrigTextureMapHeight ( void ) ;
```

Description

Gets the original user defined texture map height.,

Arguments

None.

Return Value

The original image height.

Application Notes

None.



Function Reference

(Texture Functions)

IsTextureRGB

Indicates whether image is color map.

unsigned long IsTextureRGB (void) ;

Description

Indicates whether the texture map contains color image data.

Arguments

None.

Return Value

One (TRUE) if the texture map contains color image data; zero (FALSE) if the map is monochrome.

Application Notes

None.



Function Reference

(Texture Functions)

IsMicroTexture

Indicates whether map is a microtexture map.

```
unsigned long IsMicroTexture ( void ) ;
```

Description

Indicates whether the texture map was instantiated as a microtexture map.

Arguments

None.

Return Value

One (TRUE) if the map is a microtexture map; zero otherwise.

Application Notes

None.



Microtexture Functions

Microtexture class functions are used to define an image that will be used as a microtexture map for polygons. The origin of an image is assumed to be its lower left corner. The Microtexture class inherits from the PRO_Texture class. All PRO_Texture functions still apply. The number of microtexture maps supported by the PRO-1000 is dependent on the hardware configuration.

(Microtexture Functions)

PRO_MicroTexture class constructor

Creates instance of a PRO MicroTexture.

```
PRO_MicroTexture ( unsigned long image_size_x,  
                  unsigned long image_size_y,  
                  unsigned char *red_band,  
                  unsigned char *green_band,  
                  unsigned char *blue_band ) ;
```

Description

Creates an instance of a PRO_MicroTexture for a color image.



Function Reference

Arguments

unsigned long <i>image_size_x</i>	Size of the microtexture map in the x direction. Valid values are 32, 64, 128, 256, 512, and 1024.
unsigned long <i>image_size_y</i>	Size of the microtexture map in the y direction. Valid values are 32, 64, 128, 256, 512, and 1024.
unsigned char <i>*red_band</i>	Pointer to an array of red component values for a microtexture.
unsigned char <i>*green_band</i>	Pointer to an array of green component values for a microtexture.
unsigned char <i>*blue_band</i>	Pointer to an array of blue component values for a microtexture.

Return Value

None.

Application Notes

The pointers to image band data are used directly by the PRO/API, so addresses of temporary variables should not be used.

Microtexture is supported only on polygons whose base texture map has contour translucency or no translucency.



Function Reference

(Microtexture Functions)

PRO_MicroTexture class constructor

Creates an instance of a PRO MicroTexture.

```
PRO_MicroTexture ( unsigned long image_size_x,  
                  unsigned long image_size_y,  
                  unsigned char *intensity_band );
```

Description

Creates an instance of a PRO_MicroTexture for a monochrome image.

Arguments

unsigned long <i>image_size_x</i>	Size of the microtexture map in the x direction. Valid values are 32, 64, 128, 256, 512, and 1024.
unsigned long <i>image_size_y</i>	Size of the microtexture map in the y direction. Valid values are 32, 64, 128, 256, 512, and 1024.
unsigned char <i>*intensity_band</i>	Pointer to an array of monochrome component values for a microtexture.

Return Value

None.

Application Notes

The pointers to image band data are used directly by the PRO/API, so addresses of temporary variables should not be used.

Microtexture is supported only on polygons whose base texture map has contour translucency or no translucency.



Function Reference

(Microtexture Functions)

PRO_MicroTexture class destructor

Destroys an instance of a PRO_MicroTexture.

```
~PRO_MicroTexture ( void ) ;
```

Description

Destroys an instance of a PRO_MicroTexture.

Arguments

None.

Return Value

None.

Application Notes

None.



Matrix Functions

The Matrix class functions allows definition of a position vector and a 3x3 transformation matrix. A PRO_Matrix is attached to an object (model, 6DOF node) to control its placement and rotation. A PRO_Matrix may be attached to more than one object. The PRO/API automatically attaches a PRO_Matrix to non-static models and 6DOF culling nodes, but this can be replaced by a user-created matrix to enable matrix sharing.

(Matrix Functions)

PRO_Matrix class constructor

Creates an instance of a PRO Matrix.

```
PRO_Matrix ( void ) ;
```

Description

Creates an instance of a PRO_Matrix.

Arguments

None.

Return Value

None.

Application Notes

Function GetStatus should be called to verify object instantiation.



Function Reference

(Matrix Functions)

PRO_Matrix class destructor

Destroys an instance of a PRO Matrix.

```
~PRO_Matrix ( void ) ;
```

Description

Destroys an instance of a PRO_Matrix.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

GetStatus

Returns status of object instantiation.

```
PRO_Status GetStatus ( void ) ;
```

Description

Returns status of PRO_Matrix object instantiation.

Arguments

None.

Return Value

PRO_SUCCESS if successful.

PRO_OUT_OF_HOST_MEMORY if host memory allocation fails.

PRO_OUT_OF_DISPLAY_LIST_MEMORY if the number of matrices has been exceeded.

Application Notes

The PRO-1000 has an ultimate limit of 4096 matrices, but the default memory configuration allows for 1024 matrices. See PRO_Configure_Memory for details.



Function Reference

(Matrix Functions)

SetPosition

Sets the position vector.

```
void SetPosition ( double x, double y, double z ) ;
```

Description

Sets the position vector to the Cartesian coordinates x,y,z. The default position is (0.0, 0.0, 0.0).

Arguments

double x	The x coordinate of the position.
double y	The y coordinate of the position.
double z	The z coordinate of the position.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

SetPosition

Sets the position vector.

```
void SetPosition ( float x, float y, float z ) ;
```

Description

Sets the position vector to the Cartesian coordinates x,y,z. The default position is (0.0, 0.0, 0.0).

Arguments

float x	The x coordinate of the position.
float y	The y coordinate of the position.
float z	The z coordinate of the position.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

SetPosition

Sets the position vector.

```
void SetPosition ( PRO_Vector *position ) ;
```

Description

Sets the position vector.

Arguments

PRO_Vector * <i>position</i>	Pointer to the x,y,z position vector.
------------------------------	---------------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

SetOrientation

Sets the transformation matrix.

```
void SetOrientation ( double roll, double pitch, double yaw ) ;
```

Description

Specifies the angles roll, pitch and yaw which define the stored transformation matrix. The default values of these angles are all zero, which rotates associated objects to look in the direction of the positive x axis.

Arguments

double <i>roll</i>	The rotation about the x axis, in degrees.
double <i>pitch</i>	The rotation about the y axis, in degrees.
double <i>yaw</i>	The rotation about the z axis, in degrees.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

SetOrientation

Sets the transformation matrix.

```
void SetOrientation ( float roll, float pitch, float yaw ) ;
```

Description

Specifies the angles roll, pitch and yaw which define the stored transformation matrix. The default values of these angles are all zero, which rotates associated objects to look in the direction of the positive x axis.

Arguments

float <i>roll</i>	The rotation about the x axis, in degrees.
float <i>pitch</i>	The rotation about the y axis, in degrees.
float <i>yaw</i>	The rotation about the z axis, in degrees.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

SetOrientation

Sets the transformation matrix.

```
void SetOrientation ( float *in_matrix ) ;
```

Description

Specifies the 3x3 transformation matrix to be stored. The meaning of the matrix elements are as follows:

0	1	2
Forward .X	Forward .Y	Forward .Z
3	4	5
Left .X	Left .Y	Left .Z
6	7	8
Up .X	Up .Y	Up .Z

Arguments

float *in_matrix

A pointer to the 3x3 matrix containing the rotations to be applied by the PRO_Matrix object.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

SetLocalForward

Sets the forward vector.

```
void SetLocalForward ( PRO_Vector *local_forward );
```

Description

Sets the forward (boresight) vector. Component values are in the parent's local matrix coordinates.

Arguments

<code>PRO_Vector *local_forward</code>	Pointer to the forward vector.
--	--------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

SetLocalLeft

Sets the left vector.

```
void SetLocalLeft ( PRO_Vector *local_left );
```

Description

Sets the left vector. Component values are in the parent's local matrix coordinates.

Arguments

PRO_Vector *local_left	Pointer to the left-right vector.
------------------------	-----------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

SetLocalUp

Sets the up vector.

```
void SetLocalUp ( PRO_Vector *local_up );
```

Description

Sets the up vector. Component values are in the parent's local matrix coordinates.

Arguments

PRO_Vector *local_up	Pointer to the up-down vector.
----------------------	--------------------------------

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

GetPosition

Gets the position vector.

```
void GetPosition ( double *x, double *y, double *z ) ;
```

Description

Gets the x, y, z position.

Arguments

double *x	Pointer to the x coordinate of the position.
double *y	Pointer to the y coordinate of the position.
double *z	Pointer to the z coordinate of the position.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

GetPosition

Gets the position vector.

```
void GetPosition ( float *x, float *y, float *z ) ;
```

Description

Gets the x, y, z position.

Arguments

float *x	Pointer to the x coordinate of the position.
float *y	Pointer to the y coordinate of the position.
float *z	Pointer to the z coordinate of the position.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

GetPosition

Gets the position vector.

```
void GetPosition ( double &x, double &y, double &z ) ;
```

Description

Gets the x, y, z position.

Arguments

double &x	The x coordinate of the position.
double &y	The y coordinate of the position.
double &z	The z coordinate of the position.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

GetOrientation

Gets the rotation angles.

```
void GetOrientation ( double *roll, double *pitch, double *yaw ) ;
```

Description

Gets the rotation angles derived from the stored transformation matrix.

Arguments

double <i>*roll</i>	Pointer to the roll angle, in degrees.
double <i>*pitch</i>	Pointer to the pitch angle, in degrees.
double <i>*yaw</i>	Pointer to the yaw angle, in degrees.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

GetOrientation

Gets the rotation angles.

```
void GetOrientation ( double &roll, double &pitch, double &yaw ) ;
```

Description

Gets the rotation angles derived from the stored transformation matrix.

Arguments

double &roll	The roll angle, in degrees.
double &pitch	The pitch angle, in degrees.
double &yaw	The yaw angle, in degrees.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

GetLocalForward

Gets the forward vector in matrix coordinates.

```
void GetLocalForward ( float *x, float *y, float *z ) ;
```

Description

Gets the forward (boresight) vector coordinates.

Arguments

float *x	Pointer to the x component of the forward vector.
float *y	Pointer to the y component of the forward vector.
float *z	Pointer to the z component of the forward vector.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

GetLocalLeft

Gets the left vector in matrix coordinates.

```
void GetLocalLeft ( float *x, float *y, float *z ) ;
```

Description

Gets the left vector coordinates.

Arguments

float *x	Pointer to the x component of the left-right vector.
float *y	Pointer to the y component of the left-right vector.
float *z	Pointer to the z component of the left-right vector.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

GetLocalUp

Gets the up vector in matrix coordinates.

```
void GetLocalUp ( float *x, float *y, float *z ) ;
```

Description

Gets the up vector coordinates.

Arguments

float *x	Pointer to the x component of the up-down vector.
float *y	Pointer to the y component of the up-down vector.
float *z	Pointer to the z component of the up-down vector.

Return Value

None.

Application Notes

None.



Function Reference

(Matrix Functions)

AttachMatrix

Attaches a child matrix.

```
void AttachMatrix ( PRO_Matrix *child_matrix ) ;
```

Description

Attaches a child matrix.

Arguments

PRO_Matrix *child_matrix	Pointer to a child matrix.
--------------------------	----------------------------

Return Value

None.

Application Notes

The position and orientation of the attached matrix will be interpreted as being relative to coordinate set of the parent matrix.



Function Reference

(Matrix Functions)

DetachFromParent

Detaches a child matrix.

```
void DetachFromParent ( void ) ;
```

Description

Detaches a child matrix from its parent.

Arguments

None.

Return Value

None.

Application Notes

None.



MultiGen Loader Functions

The PRO_MGen_Model class is used to load in MultiGen models (.flt files). When the .flt file name is passed to the constructor, the file is loaded and a model object is created. Articulated parts, animations, LODs, and textures are all supported. Also supported are the Real3D comments and extended attributes that are specific to the PRO-1000 product. This class is inherited from the PRO_Model class.

(MultiGen Loader Functions)

PRO_MGen_Model class constructor

Creates an instance of a PRO MGen Model.

```
PRO_MGen_Model (char *filename, unsigned long flags, float MaxScaleFactor) ;
```

Description

Creates an instance of a PRO_MGen_Model by reading a MultiGen .flt file from disk.

Arguments

char *filename	The name of the MultiGen .flt file (with extension).
unsigned long flags	Controls behavior of model processing. The default value is zero and the following values may be OR'd in: PRO_DISABLE_GRP_LAYERING (disable group layering) PRO_STATIC_MODEL (model is static) PRO_3D_LINES_SMOOTH (force smooth shading)
float MaxScaleFactor	Maximum scale factor to be used when scaling models

Return Value

None.



Function Reference

Application Notes

The environment variable “TXTPATH” is used by the loader to find the texture files called out by the .flt file. The current directory is searched first, followed by a subdirectory (if it exists) called “textures”, followed by the path pointed to by this environment variable.

The environment variable “FLTEXTERNPATH” is used by the loader to find any external reference files called out by the .flt file. The current directory is searched first, followed by a subdirectory (if it exists) called “externals”, followed by the path pointed to by this environment variable.



Function Reference

(MultiGen Loader Functions)

PRO_MGen_Model class destructor

Destroys an instance of a PRO MGen Model.

```
~PRO_MGen_Model ( void ) ;
```

Description

Destroys an instance of a PRO_MGen_Model.

Arguments

None.

Return Value

None.

Application Notes

None.



Function Reference

(MultiGen Loader Functions)

GetPolyCount

Returns the number of polygons in the model.

```
long GetPolyCount ( void ) ;
```

Description

Returns the number of polygons in the Multigen model.

Arguments

None.

Return Value

The number of polygons in the Multigen model.

Application Notes

None.



Function Reference

(MultiGen Loader Functions)

GetVertCount

Returns the number of vertices in the model.

```
long GetVertCount ( void ) ;
```

Description

Returns the number of vertices in the Multigen model.

Arguments

None.

Return Value

The number of vertices in the Multigen model.

Application Notes

None.



GLOSSARY

Anti-Aliasing	Aliasing problems can be classified as Spatial or Temporal. Spatial aliasing occurs when jagged edges and similar anomalies form from insufficient sampling of faces being processed. Temporal aliasing is a similar effect but appears only when the image is in motion. Anti-aliasing is a corrective measure for this problem and is accomplished through two methods: Interlaced smoothing and oversampling of subpixel edges.
API	Applications Programming Interface
Articulated Part	A sub-part of a model that has independent movement.
Blending	The process of transitioning from one feature to another.
Blue	The 'blue' content of a color.
Color Table	A table containing all of the color information for the database or moving model.
Contour Texture	A binary texture translucency mask.
Coordinate Set	Basis set relative to which the display list geometry is defined.
Database	The representation of a virtual environment that drives the generated image.
Data Culling	Filtering of database information to derive only that information required to generate scenes within the viewpoint FOV.
Device	A PRO-1000 unit with a unique SCSI ID.
Direction Cosine Matrix	An orthonormal geometry transform.
Display	A physical video output device.



Glossary

Double Sided	A polygon attribute that disables backfacing tests.
Fading	The reduction of visible light emanation from objects as the distance from the observer to the object increases.
Feature Type	A load management attribute used to blend sets of graphical objects.
Field-of-View (FOV) ..	The area visible for a given simulation. Usually defined by left, right, top and bottom half angles.
GENLOCK	Logical synchronization of PRO-1000 devices.
Gouraud method	Method used for curved surface shading.
Green	The 'green' content of a color.
IG	Image Generator.
Illumination	The amount of light associated with a light source that artificially illuminates the database.
Immediate Mode	The generation of graphical objects at runtime.
Interpolation	Mathematical approximation of unknown data between two known points.
Leaf Node	A culling node whose children are polygons.
Level-of-Detail	Each face in the database (except terrain) is assigned a Level-of-Detail (LOD) which gives the IG a mechanism to balance its load and to avoid processing unnecessary detail in the scene. During each processing cycle, the IG determines the LOD to be displayed for each terrain cluster and model, based on the projected size of the feature. The more distant features that appear small will be displayed in their coarsest LOD while the closer, larger features may be displayed in the finest LOD.
Line of Sight (LOS) ...	Viewray from the viewpoint through a pixel on the screen.



Glossary

Local Coordinates	Coordinates defined in terms of x,y,z relative to some predefined origin, usually the centroid of an object in the simulated environment. Same as flat earth coordinates.
Microtexture	Fine resolution texture that appears when an object is at extremely close range to the viewpoint.
Microtexture Map	Image map containing microtexture data.
Model Instance	A mechanism for sharing portions of a display list.
Modulation	The variation of polygon color due to texture and lighting.
Moving Models	Models designated for movement in the database, such as vehicles, aircraft, spacecraft, etc.
Overload	A condition that inhibits a constant real-time frame rate.
%blend	The percentage of the LOD range (end) where the higher LOD transitions to the lower level.
Pitch	Counterclockwise positive rotation about the y axis.
Pixel	Picture element. The smallest element that may show light on a display.
Position	The location of the viewpoint or model within the database.
Pre-formatted Model .	A preprocessed display list segment.
Red	The 'red' content of a color.
Renderer	The realtime image processing system.
RGB	Defines the Red/Green/Blue components of a color.
Roll	Counterclockwise positive orientation rotation about the X axis.



Glossary

Root Node	The culling node at the top of a model display list hierarchy.
Shading	Progressive darkening of areas on objects in an IG scene to simulate a directional light source.
Static Model	A model with no associated coordinate set.
Sun Vector	Vector defining the direction from which the simulated sunlight comes.
Texel	Texture element.
Texture Map	An image in digital form which may be color or monochrome.
Texture MIP-map	Simplified versions of a texture map used for trilinear interpolation.
Texturing	Variations of light levels and color for pixels within a scene to enhance realism and 3-D illusion.
Translucent Faces	Faces displayed in an IG scene that are transparent to some degree.
Update Rate	The number of cycles per unit of time the IG requires to process and send completed scene data to the display interface.
Vector	The definition of a point in space in terms of a direction and a distance (usually defined by two points).
Viewpoint	The instantaneous position (location and attitude) used by the IG to place the “viewer” in the database.
Viewport	A rectangular area of the screen into which a 3D scene will be rendered.
Yaw	Counterclockwise, positive rotation about the Z axis.



Hardware Interface and Global Functions – Continued

Function Name	C Function Prototype
PRO_SetTexturePageSize	PRO_Status PRO_SetTexturePageSize (long texture_page_size) ;
PRO_ReloadLogo	void PRO_ReloadLogo (void) ;
PRO_GetNumberOfLogicalUnits	long PRO_GetNumberOfLogicalUnits(void) ;
PRO_GetNumberOfAvailableTexels	long PRO_GetNumberOfLogicalUnits(void) ;
PRO_GetMicrotextureMapCount	long PRO_GetMicrotextureMapCount (void) ;
PRO_GetPolygonMemorySize	long PRO_GetModelNumber (long lun) ;
PRO_GetDefaultDevice	pro_device_t *PRO_GetDefaultDevice(void) ;



Device Functions

The handle returned by the PRO_Device "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Device constructor	<code>pro_device_t * dev_CreateDevice (long logical_unit) ;</code>
PRO_Device destructor	<code>void dev_DeleteDevice (pro_device_t *this_ptr) ;</code>
GetStatus	<code>PRO_Status dev_GetStatus(pro_device_t *this_ptr) ;</code>
AttachViewport	<code>PRO_Status dev_AttachViewport(pro_device_t *this_ptr , pro_viewport_t *viewport_ptr) ;</code>
DetachViewport	<code>void dev_DetachViewport(pro_device_t *this_ptr , pro_viewport_t *viewport_ptr) ;</code>
SetGamma	<code>void dev_fSetGamma (pro_device_t *this_ptr , float gamma_value , long bank) ;</code>



Device Functions – Continued

Function Name	C Function Prototype
SetGammaRGB	void dev_SetGammaRGB (pro_device_t *this_ptr , unsigned char *gamma_table_red , unsigned char *gamma_table_green , unsigned char *gamma_table_blue , bank) ;
ActivateGamma	void dev_ActivateGamma (pro_device_t *this_ptr) ;
DeactivateGamma	void dev_DeactivateGamma (pro_device_t *this_ptr) ;
SaveGammaToROM	void dev_SaveGammaToROM (pro_device_t *this_ptr) ;
WriteBlendMemory	PRO_Status dev_WriteBlendMemory (pro_device_t *this_ptr , long start_i , long start_j , long size_i , long size_j , unsigned long *bm_data_red , unsigned long *bm_data_green , unsigned long *bm_data_blue , long bank) ;
ActivateBlendMemory	void dev_ActivateBlendMemory (pro_device_t *this_ptr) ;



Device Functions – Continued

Function Name	C Function Prototype
DeactivateBlendMemory	void dev_DeactivateBlendMemory (pro_device_t *this_ptr) ;
SaveBlendMemoryToROM	void dev_SaveBlendMemoryToROM(pro_device_t *this_ptr) ;
SetForegroundColor	void dev_ISetForegroundColor (pro_device_t *this_ptr , long red, long green, long blue) ;
SetForegroundColor	void dev_ISetForegroundColor (pro_device_t *this_ptr , float red, float green, float blue) ;
SetBackgroundColor	void dev_ISetBackgroundColor (pro_device_t *this_ptr , long red, long green, long blue) ;
SetBackgroundColor	void dev_ISetBackgroundColor (pro_device_t *this_ptr , float red, float green, float blue) ;
SetDisplaySelect	void dev_SetDisplaySelect (pro_device_t *this_ptr , PRO_Display_Select display_select , PRO_Line_Rate line_rate) ;
SetOverloadMode	PRO_Status dev_SetOverloadMode (pro_device_t *this_ptr , PRO_Overload_Mode overload_mode) ;



Device Functions – Continued

Function Name	C Function Prototype
GetLogicalUnit	long dev_GetLogicalUnit (pro_device_t *this_ptr);
IsValidDisplaySelect	long dev_IsValidDisplaySelect (pro_device_t *this_ptr , PRO_Display_Select display_select , PRO_Line_Rate line_rate);
GetSyncSelectInfo	void dev_GetSyncSelectInfo (pro_device_t *this_ptr , long sync_select , PRO_Display_Select *display_select , PRO_Line_Rate *line_rate);
HasAttachedViewports	long dev_HasAttachedViewports (pro_device_t *this_ptr);
GetGamma	float dev_GetGamma (pro_device_t *this_ptr);
GetForegroundColor	void dev_lpGetForegroundColor (pro_device_t *this_ptr , long *red, long *green, long *blue);
GetBackgroundColor	void dev_lpGetBackgroundColor (pro_device_t *this_ptr , long *red, long *green, long *blue);
GetDisplaySelect	PRO_Display_Select dev_GetDisplaySelect (pro_device_t *this_ptr);



Device Functions – Continued

Function Name	C Function Prototype
GetOverloadMode	PRO_Overload_Mode dev_GetOverloadMode (pro_device_t *this_ptr) ;
GetLineRate	PRO_Line_Rate dev_GetLineRate (pro_device_t *this_ptr) ;
IsHighRes	long dev_IsHighRes (pro_device_t *this_ptr) ;
HasExpandedTexture	long dev_HasExpandedTexture(pro_device_t *this_ptr) ;
GetFirmwareRevision	char * dev_GetFirmwareRevision (pro_device_t *this_ptr) ;
GetModelNumber	long dev_GetModelNumber (pro_device_t *this_ptr) ;
GetRealtimeClockCount	unsigned long dev_GetRealtimeClockCount(pro_device_t *this_ptr) ;
GetProcessingTime	float dev_GetProcessingTime (pro_device_t *this_ptr) ;
GetFrameRateEstimate	float dev_GetFrameRateEstimate (pro_device_t *this_ptr) ;
GetCurrentFrameCount	long dev_GetCurrentFrameCount (pro_device_t *this_ptr) ;
GetHWPolygonMemorySize	long dev_GetHWPolygonMemorySize(pro_device_t *this_ptr) ;



Viewport Functions

The handle returned by the PRO_Viewport "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Viewport constructor	pro_viewport_t * vp_CreateViewport (void) ;
PRO_Viewport destructor	void vp_DeleteViewport (pro_viewport_t *this_ptr) ;
GetStatus	PRO_Status vp_GetStatus (pro_viewport_t *this_ptr) ;
AttachDefaultBlendTable	PRO_Status vp_AttachDefaultBlendTable (pro_viewport_t *this_ptr) ;
AttachDefaultDatabase	PRO_Status vp_AttachDefaultDatabase (pro_viewport_t *this_ptr) ;
AttachBlendTable	PRO_Status vp_AttachBlendTable (pro_viewport_t *this_ptr , pro_lod_table_t *api_lod_table) ;
AttachDatabase	PRO_Status vp_AttachDatabase (pro_viewport_t *this_ptr , pro_database_t *api_database) ;
AttachViewpoint	PRO_Status vp_AttachViewpoint (pro_viewport_t *this_ptr , pro_viewpoint_t *api_viewpoint) ;
DetachViewpoint	PRO_Status vp_DetachViewpoint (pro_viewport_t *this_ptr) ;
AttachChildViewport	void vp_AttachChildViewport (pro_viewport_t *this_ptr , pro_viewport_t *child_viewport) ;



Viewport Functions – Continued

Function Name	C Function Prototype
DetachChildViewport	void vp_DetachChildViewport (pro_viewport_t *this_ptr , pro_viewport_t *child_viewport) ;
SetFieldOfView	void vp_fSetFieldOfView (pro_viewport_t *this_ptr , float horizontal_fov, float vertical_fov) ;
SetFieldOfView	void vp_fSetAsymmetricFieldOfView (pro_viewport_t *this_ptr , float left, float right , float top, float bottom) ;
SetDisplayExtents	void vp_fSetDisplayExtents(pro_viewport_t *this_ptr , float up_left_i, float up_left_j , float size_i, float size_j) ;
SetSunIlluminationData	void vp_fSetSunIlluminationData (pro_viewport_t *this_ptr , float min_modulation , float variation_in_modulation) ;
SetSunVector	void vp_fSetSunVector (pro_viewport_t *this_ptr , float sunx, float suny, float sunz) ;



Viewport Functions – Continued

Function Name	C Function Prototype
SetFogParameters	void vp_fSetFogParameters (pro_viewport_t *this_ptr , float fog_attenuation , float ambient_fog , float fog_offset , float fog_density) ;
SetFogRanges	void vp_fSetFogRanges (pro_viewport_t *this_ptr , float near_dist, float far_dist) ;
SetFogColor	void vp_lSetFogColor (pro_viewport_t *this_ptr , long r, long g, long b) ;
SetLightFogColor	void vp_lSetLightFogColor (pro_viewport_t *this_ptr , long light_rgb_fog) ;
SetLightSourceColor	void vp_lSetLightSourceColor (pro_viewport_t *this_ptr , long light_rgb) ;
SetLightSourceParameters	void vp_fSetLightSourceParameters(pro_viewport_t *this_ptr , float lt_start_range, float light_extent) ;



Viewport Functions – Continued

Function Name	C Function Prototype
SetLobeParameters	void vp_fSetLobeParameters(pro_viewport_t *this_ptr , float lobe_position_i , float lobe_position_j , float lobe_size_i , float lobe_size_j) ;
SetViewportOff	void vp_SetViewportOff (pro_viewport_t *this_ptr) ;
SetViewportOn	void vp_SetViewportOn (pro_viewport_t *this_ptr) ;
SetDisplaySelect	void vp_SetDisplaySelect (pro_viewport_t *this_ptr, short sel) ;
SetInvertFlags	void vp_SetInvertFlags (pro_viewport_t *this_ptr , long invert_v, long invert_w) ;
GetViewpoint	pro_viewpoint_t * vp_GetViewpoint (pro_viewport_t *this_ptr) ;
GetAttachedDevice	pro_device_t * vp_GetAttachedDevice(pro_viewport_t *this_ptr) ;
GetFieldOfView	void vp_fpGetFieldOfView (pro_viewport_t *this_ptr , float *horizontal, float *vertical) ;
GetFieldOfView	void vp_fpGetAsymmetricFieldOfView (pro_viewport_t *this_ptr , float *left, float *right , float *top, float *bottom) ;



Viewport Functions – Continued

Function Name	C Function Prototype
GetDisplayExtents	<pre>void vp_fpGetDisplayExtents (pro_viewport_t *this_ptr , float *up_left_i, float *up_left_j , float *size_i , float *size_j) ;</pre>
GetSunIlluminationData	<pre>void vp_fpGetSunIlluminationData (pro_viewport_t *this_ptr , float *ambient_light , float *sun_intensity) ;</pre>
GetSunVector	<pre>void vp_fpGetSunVector (pro_viewport_t *this_ptr , float *sunx , float *suny , float *sunz) ;</pre>
GetFogParameters	<pre>void vp_fpGetFogParameters (pro_viewport_t *this_ptr , float *fog_attenuation , float *ambient_fog , float *fog_offset , float *fog_density) ;</pre>
GetFogRanges	<pre>void vp_fpGetFogRanges (pro_viewport_t *this_ptr , float *near_dist, float *far_dist) ;</pre>
GetFogColor	<pre>void vp_lpGetFogColor (pro_viewport_t *this_ptr , long *r, long *g, long *b) ;</pre>



Viewport Functions – Continued

Function Name	C Function Prototype
GetLightFogColor	void vp_lpGetLightFogColor (pro_viewport_t *this_ptr , long *light_rgb_fog) ;
GetLightSourceColor	void vp_lpGetLightSourceColor(pro_viewport_t *this_ptr , long *light_rgb) ;
GetLightSourceParameters	void vp_fpGetLightSourceParameters (pro_viewport_t *this_ptr , float *lt_start_range , float *light_extent) ;
GetLobeParameters	void vp_fpGetLobeParameters(pro_viewport_t *this_ptr , float *lt_src_position_i , float *lt_src_position_j , float *size_i , float *size_j) ;
GetPriority	long vp_GetPriority(pro_viewport_t *this_ptr) ;



Database Functions

The handle returned by the PRO_Database "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Database constructor	pro_database_t * db_CreateDatabase (long num_children) ;
PRO_Database destructor	void db_DeleteDatabase (pro_database_t *this_ptr) ;
GetStatus	PRO_Status db_GetStatus (pro_database_t *this_ptr) ;
AttachModel	void db_AttachModel (pro_database_t *this_ptr , pro_model_t *model_ptr) ;
DetachModel	void db_DetachModel (pro_database_t *this_ptr , pro_model_t *model_ptr) ;
ActivateAllModels	void db_ActivateAllModels (pro_database_t *this_ptr) ;
DeactivateAllModels	void db_DeactivateAllModels (pro_database_t *this_ptr) ;
HasActiveModels	long db_HasActiveModels (pro_database_t *this_ptr) ;



Viewpoint Functions

The handle returned by the PRO_Viewpoint "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Viewpoint constructor	pro_viewpoint_t * vpt_CreateViewpoint (void) ;
PRO_Viewpoint destructor	void vpt_DeleteViewpoint (pro_viewpoint_t *this_ptr) ;
GetStatus	PRO_Status vpt_GetStatus (pro_viewpoint_t *this_ptr) ;
SetPosition	void vpt_dSetPosition (pro_viewpoint_t *this_ptr , double x , double y , double z) ;
SetPosition	void vpt_fSetPosition (pro_viewpoint_t *this_ptr , float x , float y , float z) ;
SetOrientation	void vpt_dSetOrientation (pro_viewpoint_t *this_ptr , double r , double p , double y) ;
SetOrientation	void vpt_fSetOrientation (pro_viewpoint_t *this_ptr , float r , float p , float y) ;
SetOrientation	void vpt_fpSetOrientation (pro_viewpoint_t *this_ptr , float *in_matrix) ;



Viewpoint Functions – Continued

Function Name	C Function Prototype
GetPosition	void vpt_dpGetPosition (pro_viewpoint_t *this_ptr , double *x, double *y, double *z);
GetOrientation	void vpt_dpGetOrientation (pro_viewpoint_t *this_ptr , double *r, double *p, double *y);
GetLocalForward	void vpt_fpGetLocalForward (pro_viewpoint_t *this_ptr , float *x , float *y , float *z);
GetLocalLeft	void vpt_fpGetLocalLeft (pro_viewpoint_t *this_ptr , float *x , float *y , float *z);
GetLocalUp	void vpt_fpGetLocalUp (pro_viewpoint_t *this_ptr , float *x , float *y , float *z);
AttachViewpoint	void vpt_AttachViewpoint (pro_viewpoint_t *this_ptr , pro_viewpoint_t *child_viewpoint);
DetachViewpointFromParent	void vpt_DetachViewpointFromParent (pro_viewpoint_t *this_ptr);



Blend Table Functions

The handle returned by the PRO_LOD_Table "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_LOD_Table constructor	pro_lod_table_t * btab_CreateLodTable (long number_of_rtsls) ;
PRO_LOD_Table destructor	void btab_DeleteLodTable (pro_lod_table_t *this_ptr) ;
GetStatus	PRO_Status btab_GetStatus (pro_lod_table_t *this_ptr) ;
SetRangeBlendParameters	void btab_SetRangeBlendParameter(pro_lod_table_t *this_ptr , long rtsl, long lod , float start_range , float delete_range) ;
SetSizeBlendParameters	void btab_SetSizeBlendParameters (pro_lod_table_t *this_ptr , long rtsl, long lod , float start_angle , float delete_angle) ;
GetRangeBlendParameters	void btab_GetRangeBlendParameters(pro_lod_table_t *this_ptr , long rtsl, long lod , float *start_range , float *delete_range) ;



Blend Table Functions – Continued

Function Name	C Function Prototype
GetSizeBlendParameters	void btab_GetSizeBlendParameters (pro_lod_table_t *this_ptr , long rtsl, long lod , float *start_angle , float *delete_angle) ;
GetAllocatedTableSize	long btab_GetAllocatedTableSize (pro_lod_table_t *this_ptr) ;



Color Table Functions

The handle returned by the PRO_Color_Table "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Color_Table constructor	pro_color_table_t * ctab_CreateColorTable (long table_size) ;
PRO_Color_Table destructor	void ctab_DeleteColorTable (pro_color_table_t *this_ptr) ;
GetStatus	PRO_Status ctab_GetStatus (pro_color_table_t *this_ptr) ;
SetColor	PRO_Status ctab_ISetColor (pro_color_table_t *this_ptr , long index , long red, long green, long blue) ;
SetColor	PRO_Status ctab_fSetColor (pro_color_table_t *this_ptr , long index , float red, float green, float blue) ;
GetAllocatedTableSize	long ctab_GetAllocatedTableSize(pro_color_table_t *this_ptr) ;



Model Functions

The handle returned by the PRO_Model "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Model constructor	<code>pro_model_t * mod_CreatePreformattedModel(char* file_name , char* texture_path , char* uf_path) ;</code>
PRO_Model constructor	<code>pro_model_t * mod_CreatePreformattedModel(long is_static_model) ;</code>
PRO_Model constructor	<code>pro_model_t * mod_CreateModelCopy (pro_model_t *model_to_copy) ;</code>
PRO_Model constructor	<code>pro_model_t * mod_CreateInstanceReferenceModel (pro_instance_culling_node_t *instanced_geometry , pro_matrix_t *user_matrix) ;</code>
PRO_Model destructor	<code>void mod_DeleteModel (pro_model_t *this_ptr) ;</code>
GetModelStatus	<code>PRO_Status mod_GetModelStatus (pro_model_t *this_ptr) ;</code>
SetPosition	<code>void mod_dSetModelPosition (pro_model_t *this_ptr , double x, double y, double z) ;</code>
SetPosition	<code>void mod_fSetModelPosition (pro_model_t *this_ptr , float x, float y, float z) ;</code>



Model Functions – Continued

Function Name	C Function Prototype
SetPosition	<pre>void mod_dSetArtPartPositionByIndex (pro_model_t *this_ptr , double x, double y, double z , long art_part_index);</pre>
SetPosition	<pre>void mod_fSetArtPartPositionByIndex (pro_model_t *this_ptr , float x, float y, float z , long art_part_index);</pre>
SetPosition	<pre>void mod_dSetArtPartPositionByName (pro_model_t *this_ptr , double x, double y, double z , char *art_part_name);</pre>
SetPosition	<pre>void mod_fSetArtPartPositionByName (pro_model_t *this_ptr , float x, float y, float z , char *art_part_name);</pre>
SetOrientation	<pre>void mod_dSetModelOrientation (pro_model_t *this_ptr , double r, double p, double y);</pre>
SetOrientation	<pre>void mod_fSetModelOrientation (pro_model_t *this_ptr , float r, float p, float y);</pre>



Model Functions – Continued

Function Name	C Function Prototype
SetOrientation	void mod_fpSetModelOrientation (pro_model_t *this_ptr, float *matrix) ;
SetOrientation	void mod_dSetArtPartOrientationByIndex (pro_model_t *this_ptr , double r, double p, double y , long art_part_index) ;
SetOrientation	void mod_fSetArtPartOrientationByIndex (pro_model_t *this_ptr , float r, float p, float y , long art_part_index) ;
SetOrientation	void mod_fpSetArtPartOrientationByIndex (pro_model_t *this_ptr , float *matrix , long art_part_index) ;
SetOrientation	void mod_dSetArtPartOrientationByName (pro_model_t *this_ptr , double r, double p, double y , char *art_part_name) ;



Model Functions – Continued

Function Name	C Function Prototype
SetOrientation	void mod_fSetArtPartOrientationByName (pro_model_t *this_ptr , float r, float p, float y , char *art_part_name);
SetOrientation	void mod_fpSetArtPartOrientationByName (pro_model_t *this_ptr , float *matrix , char *art_part_name);
Activate	void mod_Activate (pro_model_t *this_ptr);
Deactivate	void mod_Deactivate (pro_model_t *this_ptr);
AttachColorTable	void mod_AttachColorTable (pro_model_t *this_ptr , pro_color_table_t *color_table);
SetRangeLOD	void mod_SetRangeLOD (pro_model_t *this_ptr);
SetSizeLOD	void mod_SetSizeLOD (pro_model_t *this_ptr);
UserMatrix	void mod_UserMatrix (pro_model_t *this_ptr , pro_matrix_t *user_matrix , long matrix_index);
IsStatic	long mod_IsStatic (pro_model_t *this_ptr);
IsActive	long mod_IsActive (pro_model_t *this_ptr);



Model Functions – Continued

Function Name	C Function Prototype
GetPosition	void mod_dpGetModelPosition (pro_model_t *this_ptr , double *x, double *y, double *z) ;
GetPosition	void mod_fpGetModelPosition (pro_model_t *this_ptr , float *x, float *y, float *z) ;
GetPosition	void mod_dpGetArtPartPositionByIndex (pro_model_t *this_ptr , double *x, double *y, double *z , long art_part_index) ;
GetPosition	void mod_fpGetArtPartPositionByIndex (pro_model_t *this_ptr , float *x, float *y, float *z , long art_part_index) ;
GetPosition	void mod_dpGetArtPartPositionByName (pro_model_t *this_ptr , double *x, double *y, double *z , char *art_part_name) ;
GetPosition	void mod_fpGetArtPartPositionByName (pro_model_t *this_ptr , float *x, float *y, float *z , char *art_part_name) ;



Model Functions – Continued

Function Name	C Function Prototype
GetOrientation	void mod_dpGetModelOrientation (pro_model_t *this_ptr , double *r, double *p, double *y) ;
GetOrientation	void mod_fpGetModelOrientation (pro_model_t *this_ptr , float *r, float *p, float *y) ;
GetOrientation	void mod_fpGetArtPartOrientationByName (pro_model_t *this_ptr , float *r, float *p, float *y , char *art_part_name) ;
GetOrientation	void mod_fpGetArtPartOrientationByIndex (pro_model_t *this_ptr , float *r, float *p, float *y , long art_part_index) ;
GetOrientation	void mod_dpGetArtPartOrientationByName (pro_model_t *this_ptr , double *r, double *p, double *y , char *art_part_name) ;



Model Functions – Continued

Function Name	C Function Prototype
GetOrientation	<pre>void mod_fpGetArtPartOrientationByName (pro_model_t *this_ptr , float *r, float *p, float *y , char *art_part_name);</pre>
GetLocalForward	<pre>void mod_fpGetModelLocalForward (pro_model_t *this_ptr , float *x, float *y, float *z);</pre>
GetLocalForward	<pre>void mod_fpGetArtPartLocalForwardByIndex (pro_model_t *this_ptr , float *x, float *y, float *z , long art_part_index);</pre>
GetLocalForward	<pre>void mod_fpGetArtPartLocalForwardByName (pro_model_t *this_ptr , float *x, float *y, float *z , char *art_part_name);</pre>
GetLocalLeft	<pre>void mod_fpGetModelLocalLeft (pro_model_t *this_ptr , float *x, float *y, float *z);</pre>



Model Functions – Continued

Function Name	C Function Prototype
GetLocalLeft	void mod_fpGetArtPartLocalLeftByIndex (pro_model_t *this_ptr , float *x, float *y, float *z , long art_part_index) ;
GetLocalLeft	void mod_fpGetArtPartLocalLeftByName (pro_model_t *this_ptr , float *x, float *y, float *z , char *art_part_name) ;
GetLocalUp	void mod_fpGetModelLocalLeft (pro_model_t *this_ptr , float *x, float *y, float *z) ;
GetLocalUp	void mod_fpGetArtPartLocalUpByIndex (pro_model_t *this_ptr , float *x, float *y, float *z , long art_part_index) ;
GetLocalUp	void mod_fpGetArtPartLocalUpByName (pro_model_t *this_ptr , float *x, float *y, float *z , char *art_part_name) ;
GetNumberOfAnimationSequences	long mod_GetNumberOfAnimationSequences (pro_model_t *this_ptr) ;



Model Functions – Continued

Function Name	C Function Prototype
GetAnimationSequence	<pre>pro_animation_t * mod_GetAnimationSequenceByIndex (pro_model_t *this_ptr , long animation_index) ;</pre>
GetAnimationSequence	<pre>pro_animation_t * mod_GetAnimationSequenceByName (pro_model_t *this_ptr , char *animation_name) ;</pre>
GetAnimationSequenceName	<pre>pro_animation_t * mod_GetAnimationSequenceByName (pro_model_t *this_ptr , char *animation_name)</pre>
GetNumberOfArticulatedParts	<pre>long mod_GetNumberOfArticulatedParts (pro_model_t *this_ptr) ;</pre>
GetArticulatedPartMatrixIndex	<pre>long mod_GetArticulatedPartMatrixIndex (pro_model_t *this_ptr , char *art_part_name) ;</pre>
GetArticulatedPartName	<pre>char * mod_GetArticulatedPartName (pro_model_t *this_ptr , long index) ;</pre>
GetNumberOfTextureMaps	<pre>long mod_GetNumberOfTextureMaps (pro_model_t *this_ptr) ;</pre>



Model Functions – Continued

Function Name	C Function Prototype
GetTextureMapName	char * mod_GetTextureMapName (pro_model_t *this_ptr , long index);
ProcessImmediateModeData	void mod_ProcessImmediateModeData(pro_model_t *this_ptr);
ProcessMorphedPolygons	void mod_ProcessMorphedPolygons (pro_model_t *this_ptr);
AddCullingNode	PRO_Status mod_AddCullingChild (pro_model_t *this_ptr , pro_culling_node_t *a_culling_node);
AddChild	PRO_Status mod_AddCullingChild (pro_model_t *this_ptr , pro_culling_node_t *a_culling_node);
AddPointLight	PRO_Status mod_AddPointLight (pro_model_t *this_ptr , pro_point_light_t *a_point_light);
AddInstanceSet	PRO_Status mod_AddInstanceSet (pro_model_t *this_ptr , pro_instance_set_t *an_instance_set);
AddPolygon	PRO_Status mod_AddPolygonChild (pro_model_t *this_ptr , pro_polygon_t *a_polygon);
AddChild	PRO_Status mod_AddPolygonChild (pro_model_t *this_ptr , pro_polygon_t *a_polygon);
EnableVertexSharing	void mod_EnableVertexSharing (pro_model_t *this_ptr);



Model Functions – Continued

Function Name	C Function Prototype
DisableVertexSharing	<code>void mod_DisableVertexSharing (pro_model_t *this_ptr) ;</code>
SetAutoDeleteCullingData	<code>void mod_SetAutoDeleteCullingData (pro_model_t *this_ptr) ;</code>
SetAutoDeletePolygonData	<code>void mod_SetAutoDeletePolygonData (pro_model_t *this_ptr) ;</code>
SetAutoDestructCullingData	<code>void mod_SetAutoDestructCullingData (pro_model_t *this_ptr) ;</code>
SetAutoDestructPolygonData	<code>void mod_SetAutoDestructPolygonData (pro_model_t *this_ptr) ;</code>
StoreArticulatedPart	<code>void mod_StoreArticulatedPart (pro_model_t *this_ptr , pro_6dof_culling_node_t *art_part_node) ;</code>
StoreAnimationSequence	<code>void mod_StoreAnimationSequence (pro_model_t *this_ptr , pro_animation_culling_node_t *anim_node) ;</code>
StoreTextureData	<code>long mod_StoreTextureData (pro_model_t *this_ptr , pro_texture_t *texture_map) ;</code>
StoreTextureData	<code>long mod_StoreTextureDataByName (pro_model_t *this_ptr , char *map_name) ;</code>
Save	<code>PRO_Status mod_Save (pro_model_t *this_ptr, char *model_name) ;</code>



Animation Functions

The handle returned by the PRO_Model GetAnimationSequence() function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
AnimationStartsVisible	void aseq_AnimationStartsVisible (pro_animation_t *this_ptr);
AnimationEndsVisible	void aseq_AnimationEndsVisible (pro_animation_t *this_ptr);
AnimationStartsInvisible	void aseq_AnimationStartsInvisible (pro_animation_t *this_ptr);
AnimationEndsInvisible	void aseq_AnimationEndsInvisible (pro_animation_t *this_ptr);
ActivateSequence	void aseq_ActivateSequence (pro_animation_t *this_ptr);
DeactivateSequence	void aseq_DeactivateSequence (pro_animation_t *this_ptr);
AnimationRunsForward	void aseq_AnimationRunsForward (pro_animation_t *this_ptr);
AnimationRunsBackwards	void aseq_AnimationRunsBackwards (pro_animation_t *this_ptr);



Animation Functions – Continued

Function Name	C Function Prototype
AnimationRunsMirrored	void aseq_AnimationRunsMirrored (pro_animation_t *this_ptr);
AnimationIsUserControlled	void aseq_AnimationIsUserControlled (pro_animation_t *this_ptr);
AnimationRunsForever	void aseq_AnimationRunsForever (pro_animation_t *this_ptr);
SetAnimationCycleTime	void aseq_fSetAnimationCycleTime (pro_animation_t *this_ptr , float animation_time_in_seconds);
SetAnimationCycleTime	void aseq_dSetAnimationCycleTime (pro_animation_t *this_ptr , double animation_time_in_seconds);
SetAnimationRepeatCount	void aseq_ISetAnimationRepeatCount (pro_animation_t *this_ptr, long count);
SetFrameActive	void aseq_ISetFrameActive (pro_animation_t *this_ptr , long frame_index);



Animation Functions – Continued

Function Name	C Function Prototype
SetFrameInactive	void aseq_ISetFrameInactive (pro_animation_t *this_ptr , long frame_index);
GetAnimationCycleTime	void aseq_fpGetAnimationCycleTime (pro_animation_t *this_ptr , float *cycle_time);
GetAnimationCycleTime	void aseq_dpGetAnimationCycleTime (pro_animation_t *this_ptr , double *cycle_time);
GetNumberOfAnimationFrames	long aseq_GetNumberOfAnimationFrames (pro_animation_t *this_ptr);
GetCurrentFrame	long aseq_GetNumberOfAnimationFrames (pro_animation_t *this_ptr);
GetAnimationState	long aseq_GetAnimationState(pro_animation_t *this_ptr);



Culling Node Functions

The handle returned by the PRO_Culling_Node "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function prototype
PRO_Culling_Node constructor	pro_culling_node_t * cn_CreateCullingNode (void) ;
PRO_Culling_Node destructor	void cn_DeleteCullingNode (pro_culling_node_t *this_ptr) ;
GetStatus	PRO_Status cn_GetStatus (pro_culling_node_t *this_ptr) ;
ActivateNode	void cn_ActivateNode(pro_culling_node_t *this_ptr) ;
DeactivateNode	void cn_DeactivateNode(pro_culling_node_t *this_ptr) ;
AddChild	PRO_Status cn_AddCullingChild (pro_culling_node_t *this_ptr , pro_culling_node_t *culling_child) ;
AddChild	PRO_Status cn_AddPolygonChild (pro_culling_node_t *this_ptr , pro_polygon_t *polygon_child) ;
AddPointLight	PRO_Status cn_AddPointLight (pro_culling_node_t *this_ptr , pro_point_light_t *point_light) ;
AddInstanceSet	PRO_Status cn_AddInstanceSet (pro_culling_node_t *this_ptr , pro_instance_set_t *light_set) ;



Culling Node Functions – Continued

Function Name	C Function prototype
DetachChild	void cn_DetachCullingChild (pro_culling_node_t *this_ptr , pro_culling_node_t *culling_child) ;
DetachChild	void cn_DetachPolygonChild (pro_culling_node_t *this_ptr , pro_polygon_t *polygon_child) ;
AttachColorTable	void cn_AttachColorTable (pro_culling_node_t *this_ptr , pro_color_table_t *color_table) ;
SetRangeLOD	void cn_SetRangeLOD (pro_culling_node_t *this_ptr , long traverse_hierarchy) ;
SetSizeLOD	void cn_SetSizeLOD (pro_culling_node_t *this_ptr , long traverse_hierarchy) ;
SetFeatureType	void cn_ISetFeatureType (pro_culling_node_t *this_ptr , long ftype) ;
SetGroupLayerNumber	void cn_ISetGroupLayerNumber(pro_culling_node_t *this_ptr , long layer_num) ;
SetVolumeSizeClamp	void cn_SetVolumeSizeClamp(pro_culling_node_t *this_ptr) ;



Culling Node Functions – Continued

Function Name	C Function prototype
SetAutoDeleteCullingData	void cn_SetAutoDeleteCullingData (pro_culling_node_t *this_ptr);
SetAutoDeletePolygonData	void cn_SetAutoDeletePolygonData (pro_culling_node_t *this_ptr);
SetAutoDestructCullingData	void cn_SetAutoDestructCullingData (pro_culling_node_t *this_ptr);
SetAutoDestructPolygonData	void cn_SetAutoDestructPolygonData (pro_culling_node_t *this_ptr);
FormatAndFlushPolygons	void cn_FormatAndFlushPolygons (pro_culling_node_t *this_ptr , long stencil_polygons , long enable_vertex_sharing);
GetFeatureType	long cn_GetFeatureType (pro_culling_node_t *this_ptr);
GetBoundingBox	void cn_fpGetBoundingBox (pro_culling_node_t *this_ptr , float *min_x, float *min_y, float *min_z , float *max_x, float *max_y, float *max_z);



Culling Node Functions – Continued

Function Name	C Function prototype
GetLODNumber	long cn_GetLODNumber (pro_culling_node_t *this_ptr);
GetGroupLayerNumber	long cn_GetGroupLayerNumber (pro_culling_node_t *this_ptr);
GetCentroid	void cn_fpGetCentroid (pro_culling_node_t *this_ptr , float *x, float *y, float *z);
HasAssociatedMatrix	long cn_HasAssociatedMatrix(pro_culling_node_t *this_ptr);



LOD Culling Node Functions

The handle returned by the PRO_LOD_Culling_Node "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_LOD_Culling_Node constructor	<code>pro_lod_culling_node_t * lcn_CreateLODCullingNode (void) ;</code>
PRO_LOD_Culling_Node destructor	<code>void lcn_DeleteLODCullingNode (pro_lod_culling_node_t *this_ptr) ;</code>
AddChild	<code>PRO_Status lcn_AddCullingChild (pro_lod_culling_node_t *this_ptr , pro_culling_node_t *culling_child , long lod_number) ;</code>
AddChild	<code>PRO_Status lcn_AddPolygonChild (pro_lod_culling_node_t *this_ptr , pro_polygon_t *polygon_child , long lod_number) ;</code>



Articulated Part Culling Node Functions

The handle returned by the PRO_6DOF_Culling_Node "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_6DOF_Culling_Node constructor	<code>pro_6dof_culling_node_t * dof_Create6DOFCullingNode (void) ;</code>
PRO_6DOF_Culling_Node constructor	<code>pro_6dof_culling_node_t * dof_Create6DOFCullingNodeWithMatrix (pro_matrix_t *user_matrix) ;</code>
PRO_6DOF_Culling_Node destructor	<code>void dof_Delete6DOFCullingNode (pro_6dof_culling_node_t *this_ptr) ;</code>
SetName	<code>void dof_SetName (pro_6dof_culling_node_t *this_ptr , char *name) ;</code>
SetPosition	<code>void dof_dSetPosition (pro_6dof_culling_node_t *this_ptr , double x , double y , double z) ;</code>
SetPosition	<code>void dof_fSetPosition (pro_6dof_culling_node_t *this_ptr , float x , float y , float z) ;</code>
SetOrientation	<code>void dof_dSetOrientation (pro_6dof_culling_node_t *this_ptr , double r , double p , double y) ;</code>



Articulated Part Culling Node Functions – Continued

Function Name	C Function Prototype
SetOrientation	<pre>void dof_fSetOrientation (pro_6dof_culling_node_t *this_ptr , float r , float p , float y) ;</pre>
SetOrientation	<pre>void dof_fpSetOrientation (pro_6dof_culling_node_t *this_ptr , float *matrix) ;</pre>
SetMotionExtents	<pre>void dof_dSetMotionExtents (pro_6dof_culling_node_t *this_ptr , double xmin , double ymin , double zmin , double xmax , double ymax , double zmax) ;</pre>



Articulated Part Culling Node Functions – Continued

Function Name	C Function Prototype
SetMotionExtents	<pre>void dof_fSetMotionExtents (pro_6dof_culling_node_t *this_ptr , float xmin , float ymin , float zmin , float xmax , float ymax , float zmax);</pre>
SetMatrix	<pre>void dof_SetMatrix (pro_6dof_culling_node_t *this_ptr , pro_matrix_t *user_matrix);</pre>
ReplaceMatrix	<pre>void dof_ReplaceMatrix (pro_6dof_culling_node_t *this_ptr , pro_matrix_t *user_matrix);</pre>
GetName	<pre>char * dof_GetName (pro_6dof_culling_node_t *this_ptr);</pre>
GetPosition	<pre>void dof_dpGetPosition (pro_6dof_culling_node_t *this_ptr , double *x, double *y, double *z);</pre>



Articulated Part Culling Node Functions – Continued

Function Name	C Function Prototype
GetOrientation	<pre>void dof_dpGetOrientation (pro_6dof_culling_node_t *this_ptr , double *r, double *p, double *y) ;</pre>
GetMatrix	<pre>pro_matrix_t * dof_GetMatrix (pro_6dof_culling_node_t *this_ptr) ;</pre>



Animation Culling Node Functions

The handle returned by the PRO_Animation_Culling_Node "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Animation_Culling_Node constructor	<pre>pro_animation_culling_node_t * acn_CreateAnimationCullingNode (short number_of_frames, char *name) ;</pre>
PRO_Animation_Culling_Node destructor	<pre>void acn_DeleteAnimationCullingNode (pro_animation_culling_node_t *this_ptr);</pre>
AddChild	<pre>PRO_Status acn_AddCullingChild (pro_animation_culling_node_t *this_ptr , pro_culling_node_t *culling_child , long animation_frame) ;</pre>
AddChild	<pre>PRO_Status acn_AddPolygonChild (pro_animation_culling_node_t *this_ptr , pro_polygon_t *polygon_child , long animation_frame) ;</pre>
SetName	<pre>void acn_SetName (pro_animation_culling_node_t *this_ptr , char *name) ;</pre>



Animation Culling Node Functions – Continued

Function Name	C Function Prototype
GetName	<pre>void acn_SetName (pro_animation_culling_node_t *this_ptr , char *name);</pre>
GetAnimationSequence	<pre>pro_animation_t * acn_GetAnimationSequence (pro_animation_culling_node_t *this_ptr);</pre>



Instance Culling Node Functions

The handle returned by the PRO_Instance_Culling_Node "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Instance_Culling_Node constructor	<code>pro_instance_culling_node_t * icn_CreateInstanceCullingNode (void) ;</code>
ProcessImmediateModeData	<code>void icn_ProcessImmediateModeData (pro_instance_culling_node_t *this_ptr) ;</code>
EnableVertexSharing	<code>void icn_EnableVertex_Sharing (pro_instance_culling_node_t *this_ptr) ;</code>
DisableVertexSharing	<code>void icn_DisableVertex_Sharing (pro_instance_culling_node_t *this_ptr) ;</code>



Instance Reference Culling Node Functions

The handle returned by the `PRO_Instance_Reference_Culling_Node` "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Instance_Reference_constructor	<pre>pro_instance_ref_culling_node_t * ircn_CreateInstanceReferenceCullingNode (pro_instance_culling_node_t *instanced_geometry , float x_instance_location , float y_instance_location , float z_instance_location) ;</pre>
SetInstanceLocation	<pre>void ircn_fSetInstanceLocation (pro_instance_ref_culling_node_t *this_ptr , float x, float y, float z) ;</pre>
GetInstanceLocation	<pre>void ircn_fpGetInstanceLocation (pro_instance_ref_culling_node_t *this_ptr , float *x, float *y, float *z) ;</pre>



Point Light Functions

The handle returned by the PRO_Point_Light "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Point_Light constructor	pro_point_light_t * pl_CreatePointLight (float x, float y, float z) ;
PRO_Point_Light destructor	void pl_DeletePointLight (pro_point_light_t *this_ptr) ;
SetColor	void pl_lSetColor (pro_point_light_t *this_ptr , long red , long green , long blue) ;
SetColor	void pl_fSetColor (pro_point_light_t *this_ptr , float red , float green , float blue) ;
SetPosition	void pl_fSetPosition (pro_point_light_t *this_ptr , float x, float y, float z) ;
GetPosition	void pl_fpGetPosition (pro_point_light_t *this_ptr , float *x, float *y, float *z) ;



Instance Set Functions

The handle returned by the PRO_Instance_Set "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Instance_Set constructor	pro_instance_set_t * is_CreateInstanceSet (long number_in_set) ;
PRO_Instance_Set destructor	void is_DeleteInstanceSet (pro_instance_set_t *this_ptr) ;
SetColor	void is_ISetColor (pro_instance_set_t *this_ptr , long red, long green, long blue) ;
SetColor	void is_fSetColor (pro_instance_set_t *this_ptr , float red, float green, float blue) ;
SetPosition	void is_fSetPosition (pro_instance_set_t *this_ptr , float x, float y, float z , long index) ;
GetPosition	void is_fpGetPosition (pro_instance_set_t *this_ptr , float *x, float *y, float *z , long index) ;



Instance Set Functions – Continued

Function Name	C Function Prototype
SetActiveInstanceCount	void is_ISetActiveInstanceCount (pro_instance_set_t *this_ptr , long count) ;
GetActiveInstanceCount	long is_GetActiveInstanceCount (pro_instance_set_t *this_ptr) ;



Polygon Functions

The handle returned by the PRO_Polygon "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Polygon constructor	<code>pro_polygon_t * poly_CreatePolygon (void) ;</code>
PRO_Polygon destructor	<code>void poly_DeletePolygon (pro_polygon_t *this_ptr) ;</code>
AddVertex	<code>void poly_AddVertex (pro_polygon_t *this_ptr , pro_vertex_t *a_vertex) ;</code>
AddChild	<code>void poly_AddChild (pro_polygon_t *this_ptr , pro_polygon_t *child_polygon) ;</code>
DetachChild	<code>void poly_DetachChild (pro_polygon_t *this_ptr , pro_polygon_t *child_polygon) ;</code>
AttachTexture	<code>void poly_AttachTexture (pro_polygon_t *this_ptr , pro_texture_t *texture_map) ;</code>
DetachTexture	<code>void poly_DetachTexture (pro_polygon_t *this_ptr) ;</code>
AttachMicroTexture	<code>void poly_AttachMicroTexture (pro_polygon_t *this_ptr , pro_microtexture_t *microtexture_map) ;</code>
DetachMicroTexture	<code>void poly_DetachMicroTexture (pro_polygon_t *this_ptr) ;</code>



Polygon Functions – Continued

Function Name	C Function Prototype
SetColor	void poly_ISetColor (pro_polygon_t *this_ptr , long red, long green, long blue) ;
SetColor	void poly_fSetColor (pro_polygon_t *this_ptr , float red, float green, float blue) ;
SetColorIndex	void poly_ISetColorIndex (pro_polygon_t *this_ptr , long color_index) ;
SetSensorColorIndex	void poly_ISetSensorColorIndex (pro_polygon_t *this_ptr , long color_index) ;
EnableLightingEffects	void poly_EnableLightingEffects (pro_polygon_t *this_ptr) ;
DisableLightingEffects	void poly_DisableLightingEffects (pro_polygon_t *this_ptr) ;
SetTranslucency	void poly_fSetTranslucency (pro_polygon_t *this_ptr , float translucency) ;
SetLightModifier	void poly_fSetLightModifier (pro_polygon_t *this_ptr , float percent_burn_thru) ;
SetHighPriority	void poly_ISetHighPriority (pro_polygon_t *this_ptr , long high_priority) ;



Polygon Functions – Continued

Function Name	C Function Prototype
PolygonIsLayered	void poly_PolygonIsLayered (pro_polygon_t *this_ptr) ;
IsDoubleSided	void poly_IsDoubleSided (pro_polygon_t *this_ptr) ;
DoSmoothShading	void poly_DoSmoothShading (pro_polygon_t *this_ptr) ;
DoFlatShading	void poly_DoFlatShading (pro_polygon_t *this_ptr) ;
IsNotShaded	void poly_IsNotShaded (pro_polygon_t *this_ptr) ;
VerticesAreClockwise	void poly_VerticesAreClockwise (pro_polygon_t *this_ptr) ;
VerticesAreCounterClockwise	void poly_VerticesAreCounterClockwise (pro_polygon_t *this_ptr) ;
SetXSmoothing	void poly_SetYSmoothing (pro_polygon_t *this_ptr) ;
SetYSmoothing	void poly_SetXSmoothing (pro_polygon_t *this_ptr) ;
SetXMirror	void poly_SetYMirror (pro_polygon_t *this_ptr) ;
SetYMirror	void poly_SetXMirror (pro_polygon_t *this_ptr) ;
SetNPScale	void poly_fSetNPScale (pro_polygon_t *this_ptr , float np_scale) ;
SetTranslucencyPatternSelect	void poly_IsSetTranslucencyPatternSelect (pro_polygon_t *this_ptr , long pattern_select) ;
EnableAutoVertexDelete	void poly_EnableAutoVertexDelete(pro_polygon_t *this_ptr) ;



Polygon Functions – Continued

Function Name	C Function Prototype
DisableAutoVertexDelete	void poly_DisableAutoVertexDelete(pro_polygon_t *this_ptr);
GetLODNumber	long poly_GetLODNumber(pro_polygon_t *this_ptr);
GetVertexCount	long poly_GetVertexCount(pro_polygon_t *this_ptr);
GetCompressedVertexCount	long poly_GetCompressedVertexCount (pro_polygon_t *this_ptr);
GetVertex	pro_vertex_t * poly_GetVertex(pro_polygon_t *this_ptr, long i);
IsTextured	long poly_IsTextured (pro_polygon_t *this_ptr);



Vertex Functions

The handle returned by the PRO_Vertex "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Vertex constructor	<code>pro_vertex_t * vtx_CreateVertexWithCoordinates (float x , float y , float z) ;</code>
PRO_Vertex constructor	<code>pro_vertex_t *vtx_CreateVertex (void) ;</code>
PRO_Vertex destructor	<code>void vtx_DeleteVertex (pro_vertex_t *this_ptr) ;</code>
SetCoordinates	<code>void vtx_fSetCoordinates (pro_vertex_t *this_ptr , float x , float y , float z) ;</code>
SetCoordinates	<code>void vtx_dSetCoordinates (pro_vertex_t *this_ptr double x , double y , double z) ;</code>
SetMorphCoordinates	<code>void vtx_fSetMorphCoordinates (pro_vertex_t *this_ptr , float x , float y , float z) ;</code>
SetTextureCoordinates	<code>void vtx_fSetTextureCoordinates (pro_vertex_t *this_ptr , float u, float v) ;</code>



Vertex Functions – Continued

Function Name	C Function Prototype
SetVertexNormal	void vtx_fSetVertexNormal (pro_vertex_t *this_ptr , float normal_x , float normal_y , float normal_z) ;
SetFixedShadingIntensity	void vtx_fSetFixedShadingIntensity (pro_vertex_t *this_ptr , float vertex_intensity) ;
OffsetVertex	void vtx_fOffsetVertex (pro_vertex_t *this_ptr , float x_offset , float y_offset , float z_offset) ;
ScaleVertex	void vtx_fScaleVertex (pro_vertex_t *this_ptr , float x_offset , float y_offset , float z_offset) ;
TransformVertex	void vtx_fpTransformVertex (pro_vertex_t *this_ptr , float *matrix) ;



Vertex Functions – Continued

Function Name	C Function Prototype
GetCoordinates	void vtx_fpGetCoordinates(pro_vertex_t *this_ptr , float *x, float *y, float *z);
GetCoordinates	void vtx_dpGetCoordinates(pro_vertex_t *this_ptr , double *x, double *y, double *z);
GetTextureCoordinates	void vtx_fpGetTextureCoordinates(pro_vertex_t *this_ptr , float *u, float *v);
GetVertexNormal	void vtx_fpGetVertexNormal(pro_vertex_t *this_ptr , float *norm_x , float *norm_y , float *norm_z);
IsTextured	long vtx_IsShaded (pro_vertex_t *this_ptr);
IsShaded	long vtx_IsTextured (pro_vertex_t *this_ptr);



Texture Functions

The handle returned by the PRO_Texture "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_Texture constructor	<pre>pro_texture_t * tex_CreateColorTextureMap (unsigned long image_size_x , unsigned long image_size_y , unsigned char *red_band , unsigned char *green_band , unsigned char *blue_band , unsigned char *alpha_band , long auto_alpha_detect_enable);</pre>
PRO_Texture constructor	<pre>pro_texture_t * tex_CreateMonoTextureMap (unsigned long image_size_x , unsigned long image_size_y , unsigned char *intensity_band , unsigned char *alpha_band , long auto_alpha_detect_enable);</pre>
PRO_Texture destructor	<pre>void tex_DeleteTextureMap (pro_texture_t *this_ptr);</pre>



Texture Functions – Continued

Function Name	C Function Prototype
IsContourTexture	void tex_IsContourTexture (pro_texture_t *this_ptr);
SetContourThreshold	void tex_IsSetContourThreshold (pro_texture_t *this_ptr , long threshold);
SetContourThreshold	void tex_fSetContourThreshold (pro_texture_t *this_ptr , float threshold);
EnableAlpha	void tex_EnableAlpha (pro_texture_t *this_ptr);
DisableAlpha	void tex_DisableAlpha (pro_texture_t *this_ptr);
SetTextureFileName	void tex_SetTextureFileName (pro_texture_t *this_ptr , char *file_name);
KeepDataAfterLoading	void tex_KeepDataAfterLoading (pro_texture_t *this_ptr);
FreeDataAfterLoading	void tex_FreeDataAfterLoading (pro_texture_t *this_ptr);
GetTextureFileName	char * tex_GetTextureFileName (pro_texture_t *this_ptr);
GetOrigTextureMapWidth	unsigned long tex_GetOrigTextureMapWidth (pro_texture_t *this_ptr);
GetOrigTextureMapHeight	unsigned long tex_GetOrigTextureMapHeight(pro_texture_t *this_ptr);
IsTextureRGB	unsigned long tex_IsTextureRGB (pro_texture_t *this_ptr);
IsMicrotexture	unsigned long tex_IsMicroTexture(pro_texture_t *this_ptr);



Microtexture Functions

The handle returned by the PRO_MicroTexture "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function Prototype
PRO_MicroTexture constructor	<pre>pro_microtexture_t * tex_CreateColorMicroTextureMap (unsigned long size_x , unsigned long image_size_y , unsigned char *red_band , unsigned char *green_band , unsigned char *blue_band) ;</pre>
PRO_MicroTexture constructor	<pre>pro_microtexture_t * tex_CreateMonoMicroTextureMap (unsigned long size_x , unsigned long size_y , unsigned char *intensity_band) ;</pre>
PRO_MicroTexture destructor	<pre>void tex_DeleteMicroTextureMap (pro_microtexture_t *this_ptr) ;</pre>



Matrix Functions

The handle returned by the PRO_Matrix "constructor" function(s) should be passed to all other C-callable functions with arguments of this type.

Function Name	C Function prototype
PRO_Matrix constructor	pro_matrix_t * mtx_CreateMatrix (void) ;
PRO_Matrix destructor	void mtx_DeleteMatrix (pro_matrix_t *this_ptr) ;
GetStatus	PRO_Status mtx_GetStatus (pro_matrix_t *this_ptr) ;
SetPosition	void mtx_dSetPosition (pro_matrix_t *this_ptr , double x , double y , double z) ;
SetPosition	void mtx_fSetPosition (pro_matrix_t *this_ptr , float x , float y , float z)
SetOrientation	void mtx_dSetOrientation (pro_matrix_t *this_ptr , double r , double p , double y) ;
SetOrientation	void mtx_fSetOrientation (pro_matrix_t *this_ptr , float r , float p , float y) ;



C++ to C Function Calls Cross Reference

Function Name	C Function prototype
SetOrientation	<pre>void mtx_fpSetOrientation (pro_matrix_t *this_ptr , float *in_matrix);</pre>
SetLocalForward	<pre>void mtx_fSetLocalForward (pro_matrix_t *this_ptr , float x , float y , float z);</pre>



Matrix Functions – Continued

Function Name	C Function prototype
SetLocalLeft	void mtx_fSetLocalLeft (pro_matrix_t *this_ptr , float x , float y , float z) ;
SetLocalUp	void mtx_fSetLocalUp (pro_matrix_t *this_ptr , float x , float y , float z) ;
GetPosition	void mtx_dpGetPosition (pro_matrix_t *this_ptr , double *x, double *y, double *z) ;
GetPosition	void mtx_fpGetPosition (pro_matrix_t *this_ptr , float *x, float *y, float *z) ;
GetOrientation	void mtx_dpGetOrientation (pro_matrix_t *this_ptr , double *r, double *p, double *y) ;
GetLocalForward	void mtx_fpGetLocalForward (pro_matrix_t *this_ptr , float *x , float *y , float *z) ;
GetLocalLeft	void mtx_fpGetLocalLeft (pro_matrix_t *this_ptr , float *x , float *y , float *z) ;
GetLocalUp	void mtx_fpGetLocalUp (pro_matrix_t *this_ptr , float *x , float *y , float *z) ;



Matrix Functions – Continued

Function Name	C Function prototype
AttachMatrix	void mtx_AttachMatrix (pro_matrix_t *this_ptr , pro_matrix_t *child_matrix) ;
DetachFromParent	void mtx_DetachFromParent (pro_matrix_t *this_ptr) ;



Index – Next Page

A

abbreviated configure memory,
 PRO_AbbreviatedConfigureMemory, 52, 104,
 550

about this guide, 28

about this software, 30

Activate, model functions, 63, 284, 573

activate blend memory, ActivateBlendMemory, 54,
 154, 554

activate gamma, ActivateGamma, 54, 149, 554

ActivateAllModels, database functions, 58, 233,
 564

ActivateBlendMemory, device functions, 54, 154,
 554

ActivateGamma, device functions, 54, 149, 554

ActivateNode, culling node functions, 68, 364,
 584

ActivateSequence, animation functions, 67, 95,
 96, 344, 581

AddChild

animation culling node functions

 child culling node, 73, 412, 593

 child polygon, 73, 413, 593

culling node functions, 68, 366, 584

 polygon, 68, 367, 584

model functions

 culling node, 65, 324, 579

 polygon, 65, 328, 579

 polygon functions, 78, 443, 600

AddCullingNode, model functions, 65, 323, 579

AddInstanceSet

 culling node functions, 68, 369, 584

 model functions, 65, 326, 579

AddPointLight

 culling node functions, 68, 368, 584

 model functions, 65, 325, 579

AddPolygon, model functions, 65, 100, 327, 579

AddVertex, polygon functions, 78, 100, 101,
 442, 600

Adobe, PDF files, 29

animation culling node functions, 73, 410, 593,
 594

AddChild

 child culling node, 73, 412, 593

 child polygon, 73, 413, 593

GetAnimationSequence, 73, 416, 594

GetName, 73, 415, 594

PRO_Animation_Culling_Node constructor, 73,
 410, 593

PRO_Animation_Culling_Node destructor, 73,
 411, 593

SetName, 73, 414, 593

animation culling nodes, API data types, 48

animation functions, 67, 340, 581, 582, 583

 ActivateSequence, 67, 95, 96, 344, 581

 AnimationEndsInvisible, 67, 96, 343, 581

 AnimationEndsVisible, 67, 341, 581

 AnimationIsUserControlled, 67, 349, 582

 AnimationRunsBackward, 96

 AnimationRunsBackwards, 67, 347, 581



Index – Next Page

- AnimationRunsForever, 67, 95, 96, 350, 582
- AnimationRunsForward, 67, 95, 346, 581
- AnimationRunsMirrored, 67, 96, 348, 582
- AnimationStartsInvisible, 67, 96, 342, 581
- AnimationStartsVisible, 67, 340, 581
- DeactivateSequence, 67, 345, 581
- GetAnimationCycleTime
 - double precision, 67, 357, 583
 - single precision, 67, 356, 583
- GetAnimationState, 67, 360, 583
- GetCurrentFrame, 67, 359, 583
- GetNumberOfAnimationFrames, 67, 358, 583
- SetAnimationCycleTime
 - double precision, 67, 95, 96, 351, 582
 - single precision, 67, 352, 582
- SetAnimationRepeatCount, 67, 353, 582
- SetFrameActive, 67, 97, 354, 582
- SetFrameInactive, 67, 97, 355, 583
- animation sequence, models, 47
- AnimationEndsInvisible, animation functions, 67, 96, 343, 581
- AnimationEndsVisible, animation functions, 67, 341, 581
- AnimationIsUserControlled, animation functions, 67, 349, 582
- AnimationRunsBackward, animation functions, 96
- AnimationRunsBackwards, animation functions, 67, 347, 581
- AnimationRunsForever, animation functions, 67, 95, 96, 350, 582
- AnimationRunsForward, animation functions, 67, 95, 346, 581
- AnimationRunsMirrored, animation functions, 67, 96, 348, 582
- animations, 47
- AnimationStartsInvisible, animation functions, 67, 96, 342, 581
- AnimationStartsVisible, animation functions, 67, 340, 581
- anti-aliasing, 31, 50, 546
- API, 38, 44, 546
 - about this software, 30
 - C callable functions, 30
 - C++ applications, 30
 - concepts and conventions, 36
 - function reference, 102
 - functional groups, 52
 - C++ to C Functions, 550
 - library functions, 29, 30
 - PRO-1000, 28
 - Sun Solaris, 30
 - Windows NT, 30
- API data types, 44
- color table, 51
- culling nodes, 47
 - animation culling nodes, 48
 - articulated part culling nodes, 48
 - instance culling nodes, 48
 - instance reference culling nodes, 48
 - instance sets, 49
 - LOD culling nodes, 48
 - point lights, 49



Index – Next Page

- database, 45
- device, 44
- level-of-detail, 51
- models, 46
 - animation sequence, 47
 - immediate mode, 46
 - instancing, 46
 - pre-formatted, 46
- polygons, 50
- viewpoint, 45
- viewport, 45
- applications programming interface, PRO-1000, 28
- articulated part, 546
- articulated part culling node functions, 71, 72, 393, 589, 590, 591, 592
 - GetMatrix, 72, 409, 592
 - GetName, 71, 406, 591
 - GetOrientation, 72, 408, 592
 - GetPosition, 72, 407, 591
 - PRO_6DOF_Culling_Node constructor, 71, 393, 394, 589
 - PRO_6DOF_Culling_Node destructor, 71, 395, 589
 - ReplaceMatrix, 71, 405, 591
 - SetMatrix, 71, 404, 591
 - SetMotionExtents
 - double precision, 71, 402, 590
 - single precision, 71, 403, 591
 - SetName, 71, 396, 589
 - SetOrientation
 - double precision, 71, 399, 589
 - matrix, 71, 401, 590
 - single precision, 71, 400, 590
 - SetPosition
 - double precision, 71, 397, 589
 - single precision, 71, 398, 589
- articulated part culling nodes, API data types, 48
- attach viewport, AttachViewport, 54, 144, 553
- AttachBlendTable, viewport functions, 56, 188, 558
- AttachChildViewpoint, viewport functions, 56, 192, 558
- AttachColorTable
 - culling node functions, 68, 372, 585
 - model functions, 63, 286, 573
- AttachDatabase, viewport functions, 56, 89, 92, 95, 99, 100, 189, 558
- AttachDefaultBlendTable, viewport functions, 56, 186, 558
- AttachDefaultDatabase, viewport functions, 56, 187, 558
- AttachMatrix, matrix functions, 84, 539, 613
- AttachMicroTexture, polygon functions, 78, 447, 600
- AttachModel, database functions, 58, 88, 91, 95, 99, 231, 564
- AttachTexture, polygon functions, 78, 445, 600
- AttachViewpoint
 - viewpoint functions, 59, 251, 566
 - viewport functions, 56, 190, 558
- AttachViewport, device functions, 54, 88, 89, 90, 92, 94, 95, 99, 100, 144, 553



Index – Next Page

B

background color

- GetBackgroundColor, 55, 170, 556
- PRO_SetBackgroundColor, 53, 121
- SetBackGroundColor, 91
- SetBackgroundColor
 - 8-bit, 54, 159, 555
 - normalized, 54, 158, 160, 555

blend table

- AttachBlendTable, 56, 188, 558
- AttachDefaultBlendTable, 56, 186, 558

blend table functions, 60, 253, 567, 568

- GetAllocatedTableSize, 60, 260, 568
- GetRangeBlendParameters, 60, 258, 567
- GetSizeBlendParameters, 60, 259, 568
- GetStatus, 60, 255
- PRO_LOD_Table constructor, 60, 253, 567
- PRO_LOD_Table destructor, 60, 254, 567
- SetRangeBlendParameters, 60, 256, 567
- SetSizeBlendParameters, 60, 257, 567

blend types, application controllable, 36

blending, 36, 546, 548

- ActivateBlendMemory, 54, 154, 554
- DeactivateBlendMemory, 54, 155, 555
- GetAllocatedTableSize, 60, 260, 568
- GetRangeBlendParameters, 60, 258, 567
- GetSizeBlendParameters, 60, 259, 568
- level-of-detail, 36
- SaveBlendMemoryToROM, 54, 156, 555

- SetRangeBlendParameters, 60, 256, 567
- SetSizeBlendParameters, 60, 257, 567
- WriteBlendMemory, 54, 152, 554

blue, 546

bounding sphere, 47

C

C callable functions

- API, C++ to C Functions, 550
- PRO API, 30

C++ applications, PRO API, 30

C++ to C Functions Cross Reference, functional groups, C++ to C Functions, 550

child node, 47

child pointers, 47

clock

- GetRealtimeClockCount, 55, 178, 557
- PRO_GetRealtimeClockCount, 53, 135

color, 44

- GetBackgroundColor, 55, 170, 556
- GetForegroundColor, 55, 169, 556
- polygons, 51
- PRO_SetBackgroundColor, 53, 121
- PRO_SetForegroundColor, 53, 120
- SetBackGroundColor, 91
- SetBackgroundColor
 - 8-bit, 54, 159, 555
 - normalized, 54, 158, 160, 555
- SetForegroundColor, 8-bit, 54, 157, 555



Index – Next Page

- color table, 47, 546
 - API data types, 51
 - polygons, 51
- color table functions, 61, 261, 569
 - GetAllocatedTableSize, 61, 266, 569
 - GetStatus, 61, 263, 567, 569
 - PRO_Color_Table constructor, 61, 261, 569
 - PRO_Color_Table destructor, 61, 262, 569
 - SetColor
 - integer, 61, 264, 569
 - single precision, 61, 265, 569
- communication, PRO_Stop, 52, 111, 551
- concepts and object data types, 29, 36
- configuration functions, 52, 102, 550
 - PRO_AbbreviatedConfigureMemory, 52, 104, 550
 - PRO_Configure_Memory, 550
 - PRO_ConfigureMemory, 52, 102
 - PRO_SetLogicalUnitOff, 52, 107, 550
 - PRO_SetLogicalUnitOn, 52, 106, 550
- configure memory
 - PRO_Configure_Memory, 550
 - PRO_ConfigureMemory, 52, 102
- contour texture, 546
- controls, video output, 43
 - gamma correction, 43
 - soft edge blending, 43
- coordinate set, 546
- coordinate sets, capacity, 35
- coordinate systems, 38
 - local, 39
 - order of rotation angles, 38
 - viewpoint, 40
 - world, 38
- Cross Reference – C++ to C, API, 550
- culling node, 51
- culling node functions, 68, 69, 361, 389, 584, 585, 586, 587
 - ActivateNode, 68, 364, 584
 - AddChild, 68, 366, 584
 - polygon, 68, 367, 584
 - AddInstanceSet, 68, 369, 584
 - AddPointLight, 68, 368, 584
 - AttachColorTable, 68, 372, 585
 - DeactivateNode, 68, 365, 584
 - DetachChild, 68, 370, 585
 - polygon, 68, 371, 585
 - FormatAndFlushPolygons, 69, 382, 586
 - GetBoundingBox, 69, 384, 586
 - GetCentroid, 69, 387, 587
 - GetFeatureType, 69, 383, 586
 - GetGroupLayerNumber, 69, 386, 587
 - GetLODNumber, 69, 385, 587
 - GetStatus, 68, 363, 584
 - HasAssociatedMatrix, 69, 388, 587
 - PRO_Culling_Node constructor, 68, 361, 584
 - PRO_Culling_Node destructor, 68, 362, 584
 - SetAutoDeleteCullingData, 69, 378, 586
 - SetAutoDeletePolygonData, 69, 379, 586
 - SetAutoDestructCullingData, 69, 380, 586
 - SetAutoDestructPolygonData, 69, 381, 586



Index – Next Page

- SetFeatureType, 68, 375, 585
- SetGroupLayerNumber, 68, 376, 585
- SetRangeLOD, 68, 373, 585
- SetSizeLOD, 68, 374, 585
- SetVolumeSizeClamp, 68, 377, 585
- culling nodes, API data types, 47
- customer support, Real 3D, 28

- D**
- data culling, 546
- database, 546
 - ActivateAllModels, 58, 233, 564
 - API data types, 45
 - AttachDatabase, 56, 189, 558
 - AttachDefaultDatabase, 56, 187, 558
 - AttachModel, 58, 231, 564
 - DeactivateAllModels, 58, 234, 564
 - DetachModel, 58, 232, 564
 - PRO_DisplayDatabase, 52, 89, 92, 97, 101, 112, 551
 - PRO_LoadModelDynamically, 52, 113, 551
 - PRO_UnloadDynamicModel, 52, 115, 551
- database functions, 58, 228, 564
 - ActivateAllModels, 58, 233, 564
 - AttachModel, 58, 88, 91, 95, 99, 231, 564
 - DeactivateAllModels, 58, 234, 564
 - DetachModel, 58, 232, 564
 - GetStatus, 58, 230, 564
 - HasActiveModels, 58, 235, 564
 - PRO_Database constructor, 58, 88, 90, 91, 94, 95, 99, 228, 564
 - PRO_Database destructor, 58, 229, 564
- database hierarchy, 47
- Deactivate, model functions, 63, 285, 573
- deactivate blend memory, DeactivateBlendMemory, 54, 155, 555
- deactivate gamma, DeactivateGamma, 54, 150, 554
- DeactivateAllModels, database functions, 58, 234, 564
- DeactivateBlendMemory, device functions, 54, 155, 555
- DeactivateGamma, device functions, 54, 150, 554
- DeactivateNode, culling node functions, 68, 365, 584
- DeactivateSequence, animation functions, 67, 345, 581
- DeleteDataAfterLoading, texture functions, 81, 509
- detach viewport, DetachViewport, 54, 145, 553
- DetachChild
 - culling node functions, 68, 370, 585
 - polygon, 68, 371, 585
 - polygon functions, 78, 444, 600
- DetachChildViewpoint, viewport functions, 56, 193, 559
- DetachFromParent, matrix functions, 84, 540, 613



Index – Next Page

- DetachMicroTexture, polygon functions, 78, 448, 600
- DetachModel, database functions, 58, 232, 564
- DetachTexture, polygon functions, 78, 446, 600
- DetachViewpoint, viewport functions, 56, 191, 558
- DetachViewpointFromParent, viewpoint functions, 59, 252, 566
- DetachViewport, device functions, 54, 145, 553
- developer's guide, PRO-1000, 28
- device
 - API data types, 44
 - default, 44
 - SCSI ID, 546
- device functions, 54, 55, 140, 553, 554, 555, 556, 557
 - ActivateBlendMemory, 54, 154, 554
 - ActivateGamma, 54, 149, 554
 - AttachViewport, 54, 88, 89, 90, 92, 94, 95, 99, 100, 144, 553
 - DeactivateBlendMemory, 54, 155, 555
 - DeactivateGamma, 54, 150, 554
 - DetachViewport, 54, 145, 553
 - GetBackgroundColor, 55, 170, 556
 - GetCurrentFrameCount, 55, 181, 557
 - GetDisplaySelect, 55, 171, 556
 - GetFirmwareRevision, 55, 176, 557
 - GetForegroundColor, 55, 169, 556
 - GetFrameRateEstimate, 55, 180, 557
 - GetGamma, 55, 168, 556
 - GetHWPolygonMemorySize, 55, 182, 557
 - GetLineRate, 55, 173, 557
 - GetLogicalUnit, 55, 164, 556
 - GetModelNumber, 55, 177, 557
 - GetOverloadMode, 55, 172, 557
 - GetProcessingTime, 55, 179, 557
 - GetRealtimeClockCount, 55, 178, 557
 - GetStatus, 54, 143, 553
 - GetSyncSelectInfo, 55, 166, 556
 - HasAttachedViewports, 55, 167, 556
 - HasExpandedTexture, 55, 175, 557
 - IsHighRes, 55, 174, 557
 - IsValidDisplaySelect, 55, 165, 556
 - PRO_Device constructor, 54, 88, 90, 91, 94, 98, 99, 140, 553
 - PRO_Device destructor, 54, 142, 553
 - SaveBlendMemoryToROM, 54, 156, 555
 - SaveGammaToROM, 54, 151, 554
 - SetBackgroundColor, 91
 - SetBackgroundColor
 - 8-bit, 54, 159, 555
 - normalized, 54, 158, 160, 555
 - SetDisplaySelect, 55, 161, 555
 - SetForegroundColor, 8-bit, 54, 157, 555
 - SetGamma, 54, 146, 553
 - SetGammaRGB, 54, 147, 554
 - SetOverloadMode, 55, 163, 555
 - WriteBlendMemory, 54, 152, 554
- devices, 44
- differential fast-wide SCSI II, PRO-1000 interface, 31
- direction cosine matrix, 546



Index – Next Page

DisableAlpha, texture functions, 81, 505, 608

DisableAutoVertexDelete, polygon functions, 79, 472, 603

DisableLightingEffects, polygon functions, 78, 454, 601

DisableVertexSharing
instance culling node functions, 74, 420, 595
model functions, 65, 330, 580

display, 546
GetDisplayExtents, 57, 217, 562
GetLineRate, 55, 173, 557
GetSyncSelectInfo, 55, 166, 556
SetDisplayExtents, 56, 196, 559
SetDisplaySelect, 57, 211, 561

display database, PRO_DisplayDatabase, 52, 89, 92, 97, 101, 112, 551

display interface
multiple channel, 34
options, 34
single channel, 34
standard line rates, 34

display select, 44
GetDisplaySelect, 55, 171, 556
IsValidDisplaySelect, 55, 165, 556
PRO_SetDisplaySelect, 53, 119
SetDisplaySelect, 55, 161, 555

DoFlatShading, polygon functions, 79, 461, 602

DoSmoothShading, polygon functions, 79, 460, 602

double sided, 547

dynamic database paging, Save, formatted model, 66, 339, 580

dynamic paging, 41

E

easy to use, PRO-1000, 28

electronic documents, PRO-1000, 29

EnableAlpha, texture functions, 81, 504, 608

EnableAutoVertexDelete, polygon functions, 79, 471, 602

EnableLightingEffects, polygon functions, 78, 453, 601

EnableVertexSharing
instance culling node functions, 74, 419, 595
model functions, 65, 329, 579

expanded texture, HasExpandedTexture, 55, 175, 557

F

fading, 36, 547

feature type, 547

field-of-view, 47, 547
GetFieldOfView
asymmetric, 57, 216, 561
symmetric, 57, 215, 561

firmware revision
GetFirmwareRevision, 55, 176, 557
PRO_GetFirmwareRevision, 53, 133



Index – Next Page

- fixed polygon shading, 37
- flat sun lighting, 37
- fog, 37, 50
 - color and density, 45
 - GetFogColor, 57, 222, 562
 - GetFogParameters, 57, 220, 562
 - GetFogRanges, 57, 221, 562
 - GetLightFogColor, 57, 223, 563
 - SetFogColor, 56, 202, 560
 - SetFogParameters, 56, 199, 560
 - SetFogRanges, 56, 201, 560
 - SetLightFogColor, 56, 203, 560
- fog parameters, 38
- foreground color
 - GetForegroundColor, 55, 169, 556
 - PRO_SetForegroundColor, 53, 120
 - SetForegroundColor, 8-bit, 54, 157, 555
- FormatAndFlushPolygons, culling node functions, 69, 382, 586
- formatting mode, 41
- frame count, GetCurrentFrameCount, 55, 181, 557
- frame rate
 - GetFrameRateEstimate, 55, 180, 557
 - PRO_GetFrameRateEstimate, 53, 137
- FreeDataAfterLoading, texture functions, 81, 508, 608
- function reference, 29
 - animation functions, 340
 - API, 102
 - blend table functions, 253
 - color table functions, 261
 - configuration functions, 102
 - culling node functions, 361, 389
 - database functions, 228
 - device functions, 140
 - hardware interface and global functions, 108
 - model functions, 267
 - MultiGen Loader functions, 85, 541
 - polygon functions, 440
 - texture functions, 496
 - vertex functions, 478
 - viewpoint functions, 236
 - viewport functions, 183
- functional group, overview, 30
- functional groups, 29
 - animation culling node functions, 73, 410, 593, 594
 - animation functions, 67, 581, 582, 583
 - API, 52
 - C++ to C Functions, 550
 - articulated part culling node functions, 71, 72, 393, 589, 590, 591, 592
 - blend table functions, 60, 567, 568
 - color table functions, 61, 569
 - configuration functions, 52, 550
 - culling node functions, 68, 69, 584, 585, 586, 587
 - database functions, 58, 564
 - device functions, 54, 55, 553, 554, 555, 556, 557
 - hardware interface and global functions, 52, 53, 551, 552



Index – Next Page

instance culling node functions, 74, 417, 595
instance reference culling node functions, 75, 421, 596
instance set functions, 77, 431, 598, 599
LOD culling node functions, 70, 588
matrix functions, 83, 519, 610, 612, 613
microtexture functions, 82, 515, 609
model functions, 62, 63, 64, 65, 66, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580
point light functions, 76, 425, 597
polygon functions, 78, 79, 600, 601, 602, 603
sorted by type, 29
texture functions, 81, 607, 608
vertex functions, 80, 604, 605, 606
viewpoint functions, 59, 565, 566
viewport functions, 56, 57, 558, 559, 560, 561, 562, 563

G

gamma

ActivateGamma, 54, 149, 554
DeactivateGamma, 54, 150, 554
GetGamma, 55, 168, 556
PRO_SetGamma, 53, 122
SaveGammaToROM, 54, 151, 554
SetGamma, 54, 146, 553
SetGammaRGB, 54, 147, 554

genlock, 547

PRO_SetGenlock, 52, 116, 551

get background color, GetBackgroundColor, 55, 170, 556
get current frame count, GetCurrentFrameCount, 55, 181, 557
get default device, PRO_GetDefaultDevice, 53, 139, 552
get display select, GetDisplaySelect, 55, 171, 556
get firmware revision
 GetFirmwareRevision, 55, 176, 557
 PRO_GetFirmwareRevision, 53, 133
get foreground color, GetForegroundColor, 55, 169, 556
get frame rate estimate,
 PRO_GetFrameRateEstimate, 53, 137
get gamma, GetGamma, 55, 168, 556
get HW polygon memory size,
 GetHWPolygonMemorySize, 55, 182, 557
get logical unit, GetLogicalUnit, 55, 164, 556
get microtexture map count,
 PRO_GetMicrotextureMapCount, 53, 132, 552
get model number
 GetModelNumber, 55, 177, 557
 PRO_GetModelNumber, 53, 134
get number of available texels,
 PRO_GetNumberOfAvailableTexels, 53, 131, 552
get number of logical units,
 PRO_GetNumberOfLogicalUnits, 53, 130, 552



Index – Next Page

- get overload mode, `GetOverloadMode`, 55, 172, 557
- get polygon memory size,
`PRO_GetPolygonMemorySize`, 53, 138, 552
- get processing time
`GetProcessingTime`, 55, 179, 557
`PRO_GetProcessingTime`, 53, 136
- get realtime clock count
`GetRealtimeClockCount`, 55, 178, 557
`PRO_GetRealtimeClockCount`, 53, 135
- get status, `GetStatus`, 54, 143, 553
- `GetActiveInstanceCount`, instance set functions, 77, 438, 599
- `GetAllocatedTableSize`, color table functions, 61, 266, 569
- `GetAllocatedTableSize`, blend table functions, 60, 260, 568
- `GetAnimationCycleTime`, animation functions
double precision, 67, 357, 583
single precision, 67, 356, 583
- `GetAnimationSequence`
animation culling node functions, 73, 416, 594
model functions
by name, 65, 314, 578
by sequence, 65, 95, 96, 313, 578
- `GetAnimationSequenceName`, model functions, 65, 315, 578
- `GetAnimationState`, animation functions, 67, 360, 583
- `GetArticulatedPartMatrixIndex`, model functions, 65, 91, 317, 578
- `GetArticulatedPartName`, model functions, 65, 318, 578
- `GetAttachedDevice`, viewport functions, 57, 214, 561
- `GetBackgroundColor`, device functions, 55, 170, 556
- `GetBoundingBox`, culling node functions, 69, 384, 586
- `GetCentroid`, culling node functions, 69, 387, 587
- `GetCompressedVertexCount`, polygon functions, 79, 475, 603
- `GetCoordinates`, vertex functions
double precision, 80, 491, 606
single precision, 80, 490, 606
- `GetCurrentFrame`, animation functions, 67, 359, 583
- `GetCurrentFrameCount`, device functions, 55, 181, 557
- `GetDisplayExtents`, viewport functions, 57, 217, 562
- `GetDisplaySelect`, device functions, 55, 171, 556
- `GetFeatureType`, culling node functions, 69, 383, 586
- `GetFieldOfView`, viewport functions
asymmetric, 57, 216, 561
symmetric, 57, 215, 561
- `GetFirmwareRevision`, device functions, 55, 176, 557



Index – Next Page

- GetFogColor, viewport functions, 57, 222, 562
- GetFogParameters, viewport functions, 57, 220, 562
- GetFogRanges, viewport functions, 57, 221, 562
- GetForegroundColor, device functions, 55, 169, 556
- GetFrameRateEstimate, device functions, 55, 180, 557
- GetGamma, device functions, 55, 168, 556
- GetGroupLayerNumber, culling node functions, 69, 386, 587
- GetHWPolygonMemorySize, device functions, 55, 182, 557
- GetInstanceLocation, instance reference culling node functions, 75, 424, 596
- GetLightFogColor, viewport functions, 57, 223, 563
- GetLightSourceColor, viewport functions, 57, 224, 563
- GetLightSourceParameters, viewport functions, 57, 225, 563
- GetLineRate, device functions, 55, 173, 557
- GetLobeParameters, viewport functions, 57, 226, 563
- GetLocalForward
- matrix functions, 84, 536, 612
 - model functions
 - articulated part
 - reference variable, 64, 576
 - single precision, 64, 302
 - index
 - reference variables, 64, 300, 576
 - single precision, 64, 301, 576
- GetLocalLeft
- matrix functions, 84, 537, 612
 - model functions
 - articulated part
 - reference variables, 64, 306, 577
 - single precision, 64, 307
 - index
 - reference variable, 64, 303, 576
 - reference variables, 304
 - single precision, 64, 305, 577
- GetLocalUp
- matrix functions, 84, 538, 612
 - model functions
 - articulated part
 - reference variables, 65, 310, 577
 - single precision, 65
 - index
 - reference variables, 64, 308, 577
 - single precision, 64, 309, 577
- GetLODNumber
- culling node functions, 69, 385, 587
 - polygon functions, 79, 473, 603
- GetLogicalUnit, device functions, 55, 164, 556
- GetMatrix, articulated part culling node functions, 72, 409, 592
- GetModelNumber, device functions, 55, 177, 557
- GetModelStatus, model functions, 62, 88, 91, 94, 99, 273, 570



Index – Next Page

- GetName
 - animation culling node functions, 73, 415, 594
 - articulated part culling node functions, 71, 406, 591
- GetNumberOfAnimationFrames, animation functions, 67, 358, 583
- GetNumberOfAnimationSequences, model functions, 65, 312, 577
- GetNumberOfArticulatedParts, model functions, 65, 316, 578
- GetNumberOfInstances, instance set functions, 77, 439
- GetNumberOfTextureMaps, model functions, 65, 319, 578
- GetOrientation
 - articulated part culling node functions, 72, 408, 592
 - matrix functions
 - matrix, 83, 534, 612
 - reference variables, 84, 535
 - model functions
 - double precision articulated part, 64, 298, 575
 - double precision model, 63, 296, 575
 - single precision articulated part, 64, 299, 576
 - single precision model, 64, 297, 575
 - viewpoint functions
 - double precision, 59, 246, 566
 - reference variables, 59, 247
- GetOrigTextureMapHeight, texture functions, 81, 512, 608
- GetOrigTextureMapWidth, texture functions, 81, 511, 608
- GetOverloadMode, device functions, 55, 172, 557
- GetPolyCount, MultiGen Loader functions, 85, 544
- GetPosition
 - articulated part culling node functions, 72, 407, 591
 - instance set functions, 77, 436, 598
 - matrix functions
 - double precision, 83, 531
 - reference variables, 83, 533, 612
 - single precision, 83, 532, 612
 - model functions
 - double precision articulated part, 63, 294, 574
 - double precision model, 63, 292, 574
 - single precision articulated part, 63, 295, 574
 - single precision model, 63, 293, 574
 - point light functions, 76, 430, 597
 - viewpoint functions
 - double precision, 59, 244, 566
 - reference variable, 59, 245
- GetPriority, viewport functions, 57, 227, 563
- GetProcessingTime, device functions, 55, 179, 557
- GetRangeBlendParameters, blend table functions, 60, 258, 567
- GetRealtimeClockCount, device functions, 55, 178, 557
- GetSizeBlendParameters, blend table functions, 60, 259, 568
- GetStatus
 - blend table functions, 60, 255
 - color table functions, 61, 263, 567, 569



Index – Next Page

culling node functions, 68, 363, 584
database functions, 58, 230, 564
device function, 54, 143, 553
matrix functions, 83, 521, 610
viewpoint functions, 59, 238, 565
viewport functions, 56, 185, 558

GetSunIlluminationData, viewport functions, 57, 218, 562

GetSunVector, viewport functions, 57, 219, 562

GetSyncSelectInfo, device functions, 55, 166, 556

GetTextureCoordinates, vertex functions, 80, 492, 606

GetTextureFileName, texture functions, 81, 510, 608

GetTextureMapName, model functions, 65, 320, 579

GetVertex, polygon functions, 79, 476, 603

GetVertexCount
MultiGen Loader functions, 85
polygon functions, 79, 474, 603

GetVertexNormal, vertex functions, 80, 493, 606

GetViewpoint, viewport functions, 57, 213, 561

glossary, 29, 546

Gouraud method, shading, 547

graphical objects, 30

graphics application, 28

graphics library functions, 28
green, 547

H

hardware capacity, 35
coordinate sets, 35
polygon memory, 35
texture memory, 35

hardware interface and global functions, 52, 53, 108, 551, 552

point light polygon, 53, 126, 551

PRO_DisplayDatabase, 52, 89, 92, 97, 101, 112, 551

PRO_GetDefaultDevice, 53, 139, 552

PRO_GetFirmwareRevision, 53, 133

PRO_GetFrameRateEstimate, 53, 137

PRO_GetMicrotextureMapCount, 53, 132, 552

PRO_GetModelNumber, 53, 134

PRO_GetNumberOfAvailableTexels, 53, 131, 552

PRO_GetNumberOfLogicalUnits, 53, 130, 552

PRO_GetPolygonMemorySize, 53, 138, 552

PRO_GetProcessingTime, 53, 136

PRO_GetRealtimeClockCount, 53, 135

PRO_Init, 52, 87, 88, 90, 91, 93, 94, 98, 99, 108, 551

PRO_InitOffline, 52, 110, 551

PRO_LoadModelDynamically, 52, 113, 551

PRO_ReloadLogo, 53, 129, 552

PRO_SetBackgroundColor, 53, 121

PRO_SetDisplaySelect, 53, 119

PRO_SetForegroundColor, 53, 120



Index – Next Page

PRO_SetGamma, 53, 122
PRO_SetGenlock, 52, 116, 551
PRO_SetMicrotextureMapCount, 53, 127, 551
PRO_SetOverloadMode, 53, 118
PRO_SetPointLightDetail, 53, 123, 551
PRO_SetPointLightFeatureType, 53, 124, 551
PRO_SetPointLightPolygon, 53, 126, 551
PRO_SetPointLightSize, 53, 125, 551
PRO_SetTexturePageSize, 53, 128, 552
PRO_SetUpdateRate, 53, 117, 551
PRO_Stop, 52, 87, 88, 89, 90, 91, 92, 93, 94, 97, 98, 99, 101, 111, 551
PRO_UnloadDynamicModel, 52, 115, 551
hardware overview, 29, 31
has attached viewports, HasAttachedViewports, 55, 167, 556
HasActiveModels, database functions, 58, 235, 564
HasAssociatedMatrix, culling node functions, 69, 388, 587
HasAttachedViewports, device functions, 55, 167, 556
HasExpandedTexture, device functions, 55, 175, 557
headlight, size and position, 45
high performance, PRO-1000, 31
high priority polygons, polygons, 50
high resolution, IsHighRes, 55, 174, 557

I
IG, PRO-1000, 547
illumination, 50, 547
 GetLightSourceColor, 57, 224, 563
 GetLightSourceParameters, 57, 225, 563
 GetLobeParameters, 57, 226, 563
 GetSunIlluminationData, 57, 218, 562
 GetSunVector, 57, 219, 562
 SetLightFogColor, 56, 203, 560
 SetLightSourceColor, 56, 204, 560
 SetLightSourceExtent, 57, 207
 SetLightSourceParameters, 57, 205, 560
 SetLightSourceStartRange, 57, 206
 SetLobeParameters, 57, 208, 561
image generator, PRO-1000, 31
immediate mode, 547
 models, 46
index, 29
initialize PRO-1000
 PRO_Init, 52, 108, 551
 PRO_InitOffline, 52, 110, 551
instance culling node functions, 74, 417, 595
 DisableVertexSharing, 74, 420, 595
 EnableVertexSharing, 74, 419, 595
 PRO_Instance_Culling_Node constructor, 74, 417, 595
 ProcessImmediateModeData, 74, 418, 595
instance culling nodes, API data types, 48
instance reference culling node functions, 75, 421, 596
 GetInstanceLocation, 75, 424, 596



Index – Next Page

PRO_Instance_Reference constructor, 75, 421, 596
SetInstanceLocation, 75, 423, 596
instance reference culling nodes, API data types, 48
instance set functions, 77, 431, 598, 599
 GetActiveInstanceCount, 77, 438, 599
 GetNumberOfInstances, 77, 439
 GetPosition, 77, 436, 598
 PRO_Instance_Set constructor, 77, 431, 598
 PRO_Instance_Set destructor, 77, 432, 598
 SetActiveInstanceCount, 77, 437, 599
 SetColor
 8-bit values, 77, 434, 598
 normalized values, 77, 433, 598
 SetPosition, 77, 435, 598
instance sets, API data types, 49
instancing, models, 46
interface
 differential fast-wide SCSI II, 31
 host computer, 31
 SCSI II, 32
interpolation, 547
introduction, 29
is high res, IsHighRes, 55, 174, 557
IsActive, model functions, 63, 291, 573
IsContourTexture, texture functions, 81, 501, 608
IsDoubleSided, polygon functions, 79, 100, 459, 602

IsHighRes, device functions, 55, 174, 557
IsMicrotexture, texture functions, 81, 514, 608
IsNotShaded, polygon functions, 79, 462, 602
IsShaded, vertex functions, 80, 495, 606
IsStatic, model functions, 63, 290, 573
IsTextured
 polygon functions, 79, 477, 603
 vertex functions, 80, 494, 606
IsTextureRGB, texture functions, 81, 513, 608
IsValidDisplaySelect, device functions, 55, 165, 556

K

KeepDataAfterLoading, texture functions, 81, 507, 608

L

layered polygons, polygons, 50
leaf node, 47, 547
level-of -detail, API data types, 51
level-of-detail, 36, 37, 547
 blending, 36
library functions, API, 30
light
 GetLightSourceColor, 57, 224, 563
 GetLightSourceParameters, 57, 225, 563
 GetLobeParameters, 57, 226, 563



Index – Next Page

- GetSunIlluminationData, 57, 218, 562
 - SetLightFogColor, 56, 203, 560
 - SetLightSourceColor, 56, 204, 560
 - SetLightSourceExtent, 57, 207
 - SetLightSourceParameters, 57, 205, 560
 - SetLightSourceStartRange, 57, 206
 - SetLobeParameters, 57, 208, 561
 - lighting and weather effects, 37
 - line of sight, 547
 - line rate, 44
 - GetLineRate, 55, 173, 557
 - load management, 36
 - load model dynamically,
 - PRO_LoadModelDynamically, 52, 113, 551
 - local, coordinate system, 39
 - local coordinates, 548
 - LOD culling node functions, 70, 588
 - AddChild
 - child culling node, 70, 391, 588
 - child polygon, 70, 392, 588
 - PRO_LOD_Culling_Node constructor, 70, 389, 588
 - PRO_LOD_Culling_Node destructor, 70, 390, 588
 - LOD culling nodes, API data types, 48
 - logical unit
 - GetLogicalUnit, 55, 164, 556
 - PRO_GetDefaultDevice, 53, 139, 552
 - PRO_SetLogicalUnitOff, 52, 107, 550
 - PRO_SetLogicalUnitOn, 52, 106, 550
 - logical units, PRO_GetNumberOfLogicalUnits, 53, 130, 552
 - logo reload, PRO_ReloadLogo, 53, 129, 552
 - low cost, PRO-1000, 31
- ### M
- matrix functions, 83, 519, 610, 612, 613
 - AttachMatrix, 84, 539, 613
 - DetachFromParent, 84, 540, 613
 - GetLocalForward, 84, 536, 612
 - GetLocalLeft, 84, 537, 612
 - GetLocalUp, 84, 538, 612
 - GetOrientation
 - matrix, 83, 534, 612
 - reference variables, 84, 535
 - GetPosition
 - double precision, 83, 531
 - reference variables, 83, 533, 612
 - single precision, 83, 532, 612
 - GetStatus, 83, 521, 610
 - PRO_Matrix constructor, 83, 519, 610
 - PRO_Matrix destructor, 83, 520, 610
 - SetLocalForward, 83, 528, 611
 - SetLocalLeft, 83, 529, 612
 - SetLocalUp, 83, 530, 612
 - SetOrientation
 - double precision, 83, 525, 610
 - matrix, 83, 527, 611
 - single precision, 83, 526, 610
 - SetPosition
 - double precision, 83, 522, 610
 - single precision, 83, 523, 610
 - vector, 83, 524



Index – Next Page

- memory
 - GetHWPolygonMemorySize, 55, 182, 557
 - PRO_AbbreviatedConfigureMemory, 52, 104, 550
 - PRO_Configure_Memory, 550
 - PRO_ConfigureMemory, 52, 102
- microtexture, 31, 548
 - PRO_GetMicrotextureMapCount, 53, 132, 552
 - PRO_SetMicrotextureMapCount, 53, 127, 551
- microtexture functions, 82, 515, 609
 - PRO_MicroTexture constructor
 - color image, 82, 515, 609
 - monochrome image, 82, 517, 609
 - PRO_MicroTexture destructor, 82, 518, 609
- microtexture map, 548
- MIP-mapped texture, 31, 549
- model
 - ActivateAllModels, 58, 233, 564
 - AttachModel, 58, 231, 564
 - DeactivateAllModels, 58, 234, 564
 - DetachModel, 58, 232, 564
 - HasActiveModels, 58, 235, 564
- model functions, 62, 63, 64, 65, 66, 267, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580
 - Activate, 63, 284, 573
 - AddChild
 - culling node, 65, 324, 579
 - polygon, 65, 328, 579
 - AddCullingNode, 65, 323, 579
 - AddInstanceSet, 65, 326, 579
 - AddPointLight, 65, 325, 579
 - AddPolygon, 65, 100, 327, 579
 - AttachColorTable, 63, 286, 573
 - Deactivate, 63, 285, 573
 - DisableVertexSharing, 65, 330, 580
 - EnableVertexSharing, 65, 329, 579
 - GetAnimationSequence
 - by name, 65, 314, 578
 - by sequence, 65, 95, 96, 313, 578
 - GetAnimationSequenceName, 65, 315, 578
 - GetArticulatedPartMatrixIndex, 65, 91, 317, 578
 - GetArticulatedPartName, 65, 318, 578
 - GetLocalForward
 - articulated part
 - reference variable, 64, 576
 - single precision, 64, 302
 - index
 - reference variables, 64, 300, 576
 - single precision, 64, 301, 576
 - GetLocalLeft
 - articulated part
 - reference variables, 64, 306, 577
 - single precision, 64, 307
 - index
 - reference variable, 64, 303, 576
 - reference variables, 304
 - single precision, 64, 305, 577
 - GetLocalUp
 - articulated part
 - reference variables, 65, 310, 577
 - single precision, 65
 - index
 - reference variables, 64, 308, 577
 - single precision, 64, 309, 577
 - GetModelStatus, 62, 88, 91, 94, 99, 273, 570



Index – Next Page

- GetNumberOfAnimationSequences, 65, 312, 577
- GetNumberOfArticulatedParts, 65, 316, 578
- GetNumberOfTextureMaps, 65, 319, 578
- GetOrientation
 - double precision articulated part, 64, 298, 575
 - double precision model, 63, 296, 575
 - single precision articulated part, 64, 299, 576
 - single precision model, 64, 297, 575
- GetPosition
 - double precision articulated part, 63, 294, 574
 - double precision model, 63, 292, 574
 - single precision articulated part, 63, 295, 574
 - single precision model, 63, 293, 574
- GetTextureMapName, 65, 320, 579
- IsActive, 63, 291, 573
- IsStatic, 63, 290, 573
- PRO_Model constructor
 - copy of another PRO model, 62, 271, 570
 - formatted at runtime, 62, 269, 270, 570
 - pre-formatted, 62, 88, 91, 94, 99, 267, 570
- PRO_Model destructor, 62, 272, 570
- ProcessImmediateModeData, 65, 101, 321, 579
- ProcessMorphedPolygons, 65, 322, 579
- Save, formatted model, 66, 339, 580
- SetAutoDeleteCullingData, 66, 331, 580
- SetAutoDeleteData, 66, 332, 580
- SetAutoDestructCullingData, 66, 333, 580
- SetAutoDestructPolygonData, 66, 334, 580
- SetOrientation
 - double precision articulated part, 62, 280, 572
 - double precision model, 62, 88, 89, 91, 92, 99, 101, 278, 571
 - matrix of single precision, articulated part, 63, 283, 573
 - matrix of single precision, model, 63, 282, 573
 - single precision articulated part, 63, 281, 572
 - single precision model, 62, 279, 571
- SetPosition
 - double precision articulated part, 62, 276, 571
 - double precision model, 62, 88, 91, 94, 95, 99, 274, 570
 - single precision articulated part, 62, 277, 571
 - single precision model, 62, 275, 570
- SetRangeLOD, 63, 88, 91, 287, 573
- SetSizeLOD, 63, 288, 573
- StoreAnimationSequence, 66, 336, 580
- StoreArticulatedPart, 66, 335, 580
- StoreTextureData
 - map name, 66, 338, 580
 - PRO_Texture, 66, 337, 580
- UserMatrix, 63, 289, 573
- model instance, 548
- model number
 - GetModelNumber, 55, 177, 557
 - PRO_GetModelNumber, 53, 134
- model numbers, PRO-1000 series, 33
- models, API data types, 46
 - animation sequence, 47
 - immediate mode, 46
 - instancing, 46
 - pre-formatted, 46
- modulation, 548



Index – Next Page

moving models, 548

MultiGen Loader functions, 85, 541

 GetPolyCount, 85, 544

 GetVertexCount, 85

 PRO_MGen_Model class constructor, 85, 541

 PRO_MGen_Model class destructor, 85, 543

multiple viewports, 45

O

offline formatting mode, 41

OffsetVertex, vertex functions, 80, 487, 605

order of rotation angles, coordinate system, 38

overload, 44, 548

overload mode

 GetOverloadMode, 55, 172, 557

 PRO_SetOverloadMode, 53, 118

 SetOverloadMode, 55, 163, 555

overview

 documentation, 29

 PRO-1000, 28

P

paging, 41

 Save, formatted model, 66, 339, 580

patents

 anti-aliasing, 31

 PRO-1000, 31

PDF files, PRO-1000 documents, 29

personal computer, host computer, 31

pitch, 548

pixel, 548

pixel fill rates, 31

pixel processor, 33

pixel writes, 33

point light, PRO_SetPointLightDetail, 53, 123, 551

point light feature type,

 PRO_SetPointLightFeatureType, 53, 124, 551

point light functions, 76, 425, 597

 GetPosition, 76, 430, 597

 PRO_Point_Light constructor, 76, 425, 597

 PRO_Point_Light destructor, 76, 426, 597

 SetColor

 8-bit color, 76, 428, 597

 normalized color, 76, 427, 597

 SetPosition, 76, 429, 597

point light size, PRO_SetPointLightSize, 53, 125, 551

point lights, API data types, 49

polygon

 GetLightSourceColor, 57, 224, 563

 SetInvertFlags, 57, 212, 561

polygon color, 50

polygon functions, 78, 79, 440, 600, 601, 602, 603

 AddChild, 78, 443, 600



Index – Next Page

- AddVertex, 78, 100, 101, 442, 600
- AttachMicroTexture, 78, 447, 600
- AttachTexture, 78, 445, 600
- DetachChild, 78, 444, 600
- DetachMicroTexture, 78, 448, 600
- DetachTexture, 78, 446, 600
- DisableAutoVertexDelete, 79, 472, 603
- DisableLightingEffects, 78, 454, 601
- DoFlatShading, 79, 461, 602
- DoSmoothShading, 79, 460, 602
- EnableAutoVertexDelete, 79, 471, 602
- EnableLightingEffects, 78, 453, 601
- GetCompressedVertexCount, 79, 475, 603
- GetLODNumber, 79, 473, 603
- GetVertex, 79, 476, 603
- GetVertexCount, 79, 474, 603
- IsDoubleSided, 79, 100, 459, 602
- IsNotShaded, 79, 462, 602
- IsTextured, 79, 477, 603
- PolygonIsLayered, 79, 458, 602
- PRO_Polygon constructor, 78, 99, 100, 440, 600
- PRO_Polygon destructor, 78, 441, 600
- SetColor
 - 8-bit color, 78, 449, 601
 - normalized values, 78, 450, 601
- SetColorIndex, 78, 451, 601
- SetHighPriority, 78, 457, 601
- SetLightModifier, 78, 456, 601
- SetNPScale, 79, 469, 602
- SetSensorColorIndex, 78, 452, 601
- SetTranslucency, 78, 455, 601
- SetTranslucencyPatternSelect, 79, 470, 602
- SetXMirror, 79, 467, 602
- SetXSmoothing, 79, 465, 602
- SetYMirror, 79, 468, 602
- SetYSmoothing, 79, 466, 602
- VerticesAreClockwise, 79, 463, 602
- VerticesAreCounterClockwise, 79, 464, 602
- polygon memory
 - capacity, 35
 - PRO_GetPolygonMemorySize, 53, 138, 552
- polygon memory size,
 - GetHWPolygonMemorySize, 55, 182, 557
- polygon node, 47, 50
- PolygonIsLayered, polygon functions, 79, 458, 602
- polygons
 - API data types, 50
 - color, 51
 - high priority polygons, 50
 - layered polygons, 50
 - SetLightSourceColor, 56, 204, 560
 - shading and lighting, 50
 - texture, 51
 - translucency, 50
- position, 548
- pre-formatted models, 46, 548
- PRO-1000 model number,
 - PRO_GetModelNumber, 53, 134
- PRO-1000 status, GetStatus, 143
- PRO_6DOF_Culling_Node constructor, articulated part culling node functions, 71, 393, 589
 - user matrix, 71, 394, 589



Index – Next Page

- PRO_6DOF_Culling_Node destructor, articulated part culling node functions, 71, 395, 589
- PRO_AbbreviatedConfigureMemory, configuration functions, 52, 104, 550
- PRO_Animation_Culling_Node constructor, animation culling node functions, 73, 410, 593
- PRO_Animation_Culling_Node destructor, animation culling node functions, 73, 411, 593
- PRO_Color_Table constructor, color table functions, 61, 261, 569
- PRO_Color_Table destructor, color table functions, 61, 262, 569
- PRO_Configure_Memory, configuration functions, 550
- PRO_ConfigureMemory, configuration functions, 52, 102
- PRO_Culling_Node constructor, culling node functions, 68, 361, 584
- PRO_Culling_Node destructor, culling node functions, 68, 362, 584
- PRO_Database constructor, database functions, 58, 88, 90, 91, 94, 95, 99, 228, 564
- PRO_Database destructor, database functions, 58, 229, 564
- PRO_Device constructor, device functions, 54, 88, 90, 91, 94, 98, 99, 140, 553
- PRO_Device destructor, device functions, 54, 142, 553
- PRO_DisplayDatabase, hardware interface and global functions, 52, 89, 92, 97, 101, 112, 551
- PRO_GetDefaultDevice, hardware interface and global functions, 53, 139, 552
- PRO_GetFirmwareRevision, hardware interface and global functions, 53, 133
- PRO_GetFrameRateEstimate, hardware interface and global functions, 53, 137
- PRO_GetMicrotextureMapCount, hardware interface and global functions, 53, 132, 552
- PRO_GetModelNumber, hardware interface and global functions, 53, 134
- PRO_GetNumberOfAvailableTexels, hardware interface and global functions, 53, 131, 552
- PRO_GetNumberOfLogicalUnits, hardware interface and global functions, 53, 130, 552
- PRO_GetPolygonMemorySize, hardware interface and global functions, 53, 138, 552
- PRO_GetProcessingTime, hardware interface and global functions, 53, 136
- PRO_GetRealtimeClockCount, hardware interface and global functions, 53, 135
- PRO_Init, hardware interface and global functions, 52, 87, 88, 90, 91, 93, 94, 98, 99, 108, 551
- PRO_InitOffline, hardware interface and global functions, 52, 110, 551
- PRO_Instance_Culling_Node constructor, instance culling node functions, 74, 417, 595



Index – Next Page

- PRO_Instance_Reference constructor, instance reference culling node functions, 75, 421, 596
- PRO_Instance_Set constructor, instance set functions, 77, 431, 598
- PRO_Instance_Set destructor, instance set functions, 77, 432, 598
- PRO_LoadModelDynamically, hardware interface and global functions, 52, 113, 551
- PRO_LOD_Culling_Node constructor, LOD culling node functions, 70, 389, 588
- PRO_LOD_Culling_Node destructor, LOD culling node functions, 70, 390, 588
- child culling node, 70, 391, 588
 - child polygon, 70, 392, 588
- PRO_LOD_Table constructor, blend table functions, 60, 253, 567
- PRO_LOD_Table destructor, blend table functions, 60, 254, 567
- PRO_Matrix constructor, matrix functions, 83, 519, 610
- PRO_Matrix destructor, matrix functions, 83, 520, 610
- PRO_MGen_Model class constructor, MultiGen Loader functions, 85, 541
- PRO_MGen_Model class destructor, MultiGen Loader functions, 85, 543
- PRO_MicroTexture constructor, microtexture functions
- color image, 82, 515, 609
 - monochrome image, 82, 517, 609
- PRO_MicroTexture destructor, microtexture functions, 82, 518, 609
- PRO_Model constructor, model functions
- copy of another PRO model, 62, 271, 570
 - formatted at runtime, 62, 269, 270, 570
 - pre-formatted, 62, 88, 91, 94, 99, 267, 570
- PRO_Model destructor, model functions, 62, 272, 570
- PRO_Point_Light constructor, point light functions, 76, 425, 597
- PRO_Point_Light destructor, point light functions, 76, 426, 597
- PRO_Polygon constructor, polygon functions, 78, 99, 100, 440, 600
- PRO_Polygon destructor, polygon functions, 78, 441, 600
- PRO_ReloadLogo, hardware interface and global functions, 53, 129, 552
- PRO_SetBackgroundColor, hardware interface and global functions, 53, 121
- PRO_SetDisplaySelect, hardware interface and global functions, 53, 119
- PRO_SetForegroundColor, hardware interface and global functions, 53, 120
- PRO_SetGamma, hardware interface and global functions, 53, 122
- PRO_SetGenlock, hardware interface function, 52, 116, 551



Index – Next Page

- PRO_SetLogicalUnitOff, configuration functions, 52, 107, 550
- PRO_SetLogicalUnitOn, configuration functions, 52, 106, 550
- PRO_SetMicrotextureMapCount, hardware interface and global functions, 53, 127, 551
- PRO_SetOverloadMode, hardware interface and global functions, 53, 118
- PRO_SetPointLightDetail, hardware interface and global functions, 53, 123, 551
- PRO_SetPointLightFeatureType, hardware interface and global functions, 53, 124, 551
- PRO_SetPointLightPolygon, hardware interface and global functions, 53, 126, 551
- PRO_SetPointLightSize, hardware interface and global functions, 53, 125, 551
- PRO_SetTexturePageSize, hardware interface and global functions, 53, 128, 552
- PRO_SetUpdateRate, hardware interface and global functions, 53, 117, 551
- PRO_Stop, hardware interface and global functions, 52, 87, 88, 89, 90, 91, 92, 93, 94, 97, 98, 99, 101, 111, 551
- PRO_Texture constructor, texture functions
color image, 81, 496, 607
monochrome image, 81, 498, 607
- PRO_Texture destructor, texture functions, 81, 500, 607
- PRO_UnloadDynamicModel, hardware interface and global functions, 52, 115, 551
- PRO_Vertex constructor, vertex functions, 80, 99, 478, 604
with no specified coordinates, 80, 479, 604
- PRO_Vertex destructor, vertex functions, 80, 480, 604
- PRO_Viewpoint constructor, viewpoint functions, 59, 88, 89, 90, 92, 94, 95, 99, 100, 236, 565
- PRO_Viewpoint destructor, viewpoint functions, 59, 237, 565
- PRO_Viewport constructor, viewport functions, 56, 183, 558
- PRO_Viewport destructor, viewport functions, 56, 184, 558
- PRO-1000
Developer's Guide, 29
electronic documents, 29
- PRO-1000 firmware, GetFirmwareRevision, 55, 176, 557
- PRO-1000 model number, GetModelNumber, 55, 177, 557
- PRO-1000 status, GetStatus, 54, 553
- ProcessImmediateModeData
instance culling node functions, 74, 418, 595
model functions, 65, 101, 321, 579
- processing time
GetProcessingTime, 55, 179, 557
PRO_GetProcessingTime, 53, 136



Index – Next Page

ProcessMorphedPolygons, model functions, 65,
322, 579

Q

quality product, PRO-1000 Professional Series, 28

R

Real 3D, electronic documentation, 29

Real 3D graphics, PRO-1000 Professional Series,
28

realtime clock

 GetRealtimeClockCount, 55, 178, 557

 PRO_GetRealtimeClockCount, 53, 135

realtime resources, 36

red, 548

reload logo, PRO_ReloadLogo, 53, 129, 552

renderer, 548

rendering engine, PRO-1000, 28

ReplaceMatrix, articulated part culling node
 functions, 71, 405, 591

RGB, 548

roll, 548

root node, 549

S

sample applications, 29, 86

 sample 1, 87

 sample 2, 90

 sample 3, 93

 sample 4, 98

Save, model functions, formatted model, 66, 339,
580

save blend memory to ROM,
 SaveBlendMemoryToROM, 54, 156, 555

save gamma to ROM, SaveGammaToROM, 54,
151, 554

SaveBlendMemoryToROM, device functions, 54,
156, 555

SaveGammaToROM, device functions, 54, 151,
554

ScaleVertex, vertex functions, 80, 488, 605

SCSI interace, 44

self luminous lighting, 37

serious simulation applications, PRO-1000, 31

set background color
 PRO_SetBackgroundColor, 53, 121
 SetBackGroundColor, 91
 SetBackgroundColor
 8-bit, 54, 159, 555
 normalized, 54, 158, 160, 555

set display select
 PRO_SetDisplaySelect, 53, 119
 SetDisplaySelect, 55, 161, 555



Index – Next Page

- set foreground color
 - PRO_SetForegroundColor, 53, 120
 - SetForegroundColor, 8-bit, 54, 157, 555
- set gamma
 - PRO_SetGamma, 53, 122
 - SetGamma, 54, 146, 553
- set gamma RGB, SetGammaRGB, 54, 147, 554
- set genlock, PRO_SetGenlock, 52, 116, 551
- set logical unit off, PRO_SetLogicalUnitOff, 52, 107, 550
- set logical unit on, PRO_SetLogicalUnitOn, 52, 106, 550
- set microtexture map count,
 - PRO_SetMicrotextureMapCount, 53, 127, 551
- set overload mode
 - PRO_SetOverloadMode, 53, 118
 - SetOverloadMode, 55, 163, 555
- set point light detail, PRO_SetPointLightDetail, 53, 123, 551
- set point light feature type,
 - PRO_SetPointLightFeatureType, 53, 124, 551
- set point light polygon,
 - PRO_SetPointLightPolygon, 53, 126, 551
- set point light size, PRO_SetPointLightSize, 53, 125, 551
- set texture page size, PRO_SetTexturePageSize, 53, 128, 552
- set update rate, PRO_SetUpdateRate, 53, 117, 551
- SetActiveInstanceCount, instance set functions, 77, 437, 599
- SetAnimationCycleTime, animation functions
 - double precision, 67, 95, 96, 351, 582
 - single precision, 67, 352, 582
- SetAnimationRepeatCount, animation functions, 67, 353, 582
- SetAutoDeleteCullingData
 - culling node functions, 69, 378, 586
 - model functions, 66, 331, 580
- SetAutoDeletePolygonData
 - culling node functions, 69, 379, 586
 - model functions, 66, 332, 580
- SetAutoDestructCullingData
 - culling node functions, 69, 380, 586
 - model functions, 66, 333, 580
- SetAutoDestructPolygonData
 - culling node functions, 69, 381, 586
 - model functions, 66, 334, 580
- SetBackgroundColor, device functions, 91
- SetBackgroundColor, device functions
 - 8-bit, 54, 159, 555
 - normalized, 54, 158, 160, 555
- SetColor
 - color table functions
 - integer, 61, 264, 569
 - single precision, 61, 265, 569
 - instance set functions
 - 8-bit values, 77, 434, 598
 - normalized values, 77, 433, 598



Index – Next Page

- point light functions
 - 8-bit color, 76, 428, 597
 - normalized color, 76, 427, 597
- polygon functions
 - 8-bit color, 78, 449, 601
 - normalized values, 78, 450, 601
- SetColorIndex, polygon functions, 78, 451, 601
- SetContourThreshold, texture functions
 - 8-bit, 81, 502, 608
 - normalized, 81, 503, 608
- SetCoordinates, vertex functions
 - double precision, 80, 482, 604
 - single precision, 80, 100, 101, 481, 604
- SetDisplayExtents, viewport functions, 56, 196, 559
- SetDisplaySelect
 - device functions, 55, 161, 555
 - viewport functions, 57, 211, 561
- SetFeatureType, culling node functions, 68, 375, 585
- SetFieldOfView, viewport functions
 - asymmetric, 56, 195, 559
 - symmetric, 56, 194, 559
- SetFixedShadingIntensity, vertex functions, 80, 486, 605
- SetFogColor, viewport functions, 56, 202, 560
- SetFogParameters, viewport functions, 56, 199, 560
- SetFogRanges, viewport functions, 56, 201, 560
- SetForegroundColor, device functions, 8-bit, 54, 157, 555
- SetFrameActive, animation functions, 67, 97, 354, 582
- SetFrameInactive, animation functions, 67, 97, 355, 583
- SetGamma, device functions, 54, 146, 553
- SetGammaRGB, device functions, 54, 147, 554
- SetGroupLayerNumber, culling node functions, 68, 376, 585
- SetHighPriority, polygon functions, 78, 457, 601
- SetInstanceLocation, instance reference culling node functions, 75, 423, 596
- SetInvertFlags, viewport functions, 57, 212, 561
- SetLightFogColor, viewport functions, 56, 203, 560
- SetLightModifier, polygon functions, 78, 456, 601
- SetLightSourceColor, viewport functions, 56, 204, 560
- SetLightSourceExtent, viewport functions, 57, 207
- SetLightSourceParameters, viewport functions, 57, 205, 560
- SetLightSourceStartRange, viewport functions, 57, 206
- SetLobeParameters, viewport functions, 57, 208, 561
- SetLocalForward, matrix functions, 83, 528, 611



Index – Next Page

- SetLocalLeft, matrix functions, 83, 529, 612
- SetLocalUp, matrix functions, 83, 530, 612
- SetMatrix, articulated part culling node functions, 71, 404, 591
- SetMorphCoordinates, vertex functions, double precision, 80, 483, 604
- SetMotionExtents, articulated part culling node functions
 - double precision, 71, 402, 590
 - single precision, 71, 403, 591
- SetName
 - animation culling node functions, 73, 414, 593
 - articulated part culling node functions, 71, 396, 589
- SetNPScale, polygon functions, 79, 469, 602
- SetOrientation
 - articulated part culling node functions
 - double precision, 71, 399, 589
 - matrix, 71, 401, 590
 - single precision, 71, 400, 590
 - matrix functions
 - double precision, 83, 525, 610
 - matrix, 83, 527, 611
 - single precision, 83, 526, 610
 - model functions
 - double precision articulated part, 62, 280, 572
 - double precision model, 62, 88, 89, 91, 92, 99, 101, 278, 571
 - matrix of single precision, articulated part, 63, 283, 573
 - matrix of single precision, model, 63, 282, 573
 - single precision articulated part, 63, 281, 572
 - single precision model, 62, 279, 571
 - viewpoint functions
 - double precision, 59, 89, 92, 95, 100, 241, 565
 - matrix of single precision, 59, 243, 565
 - single precision, 59, 242, 565
- SetOverloadMode, device functions, 55, 163, 555
- SetPosition
 - articulated part culling node functions
 - double precision, 71, 397, 589
 - single precision, 71, 398, 589
 - instance set functions, 77, 435, 598
 - matrix functions
 - double precision, 83, 522, 610
 - single precision, 83, 523, 610
 - vector, 83, 524
 - model functions
 - double precision articulated part, 62, 276, 571
 - double precision model, 62, 88, 91, 94, 95, 99, 274, 570
 - single precision articulated part, 62, 277, 571
 - single precision model, 62, 275, 570
 - point light functions, 76, 429, 597
 - viewpoint functions
 - double precision, 59, 89, 92, 95, 100, 239, 565
 - single precision, 59, 240, 565
- SetRangeBlendParameters, blend table functions, 60, 256, 567
- SetRangeLOD
 - culling node functions, 68, 373, 585
 - model functions, 63, 88, 91, 287, 573
- SetSensorColorIndex, polygon functions, 78, 452, 601
- SetSizeBlendParameters, blend table functions, 60, 257, 567



Index – Next Page

- SetSizeLOD
 - culling node functions, 68, 374, 585
 - model functions, 63, 288, 573
- SetSunIlluminationData, viewport functions, 56, 197, 559
- SetSunVector, viewport functions, 56, 198, 559
- SetTextureCoordinates, vertex functions, 80, 484, 604
- SetTextureFileName, texture functions, 81, 506, 608
- SetTranslucency, polygon functions, 78, 455, 601
- SetTranslucencyPatternSelect, polygon functions, 79, 470, 602
- SetVertexNormal, vertex functions, 80, 485, 605
- SetViewportOff, viewport functions, 57, 209, 561
- SetViewportOn, viewport functions, 57, 210, 561
- SetVolumeSizeClamp, culling node functions, 68, 377, 585
- SetXMirror, polygon functions, 79, 467, 602
- SetXSmoothing, polygon functions, 79, 465, 602
- SetYMirror, polygon functions, 79, 468, 602
- SetYSmoothing, polygon functions, 79, 466, 602
- shading, 37, 50, 549
 - Gouraud method, 547
- shading and lighting, polygons, 50
- sibling pointers, 47
- smooth polygon shading, 37
- smooth shading lighting model, 37
- static model, 549
- status, GetStatus, 54, 143, 553
- stop PRO-1000 communication, PRO_Stop, 52, 111, 551
- StoreAnimationSequence, model functions, 66, 336, 580
- StoreArticulatedPart, model functions, 66, 335, 580
- StoreTextureData, model functions
 - map name, 66, 338, 580
 - PRO_Texture, 66, 337, 580
- sun illumination, 45
 - GetSunIlluminationData, 57, 218, 562
 - GetSunVector, 57, 219, 562
 - SetSunIlluminationData, 56, 197, 559
- sun lighting, 37
- sun position, 45
- Sun Solaris, 30
- sun vector, 549
 - SetSunVector, 56, 198, 559
- superior application flexibility, PRO-1000, 31
- sync select, GetSyncSelectInfo, 55, 166, 556
- system architecture, PRO-1000 Professional Series, 31
- system configuration, 31



Index – Next Page

system configurations

model 1100, 1200, and 1400, 33

optional, 33

standard, 33

system diagram, 32

system expandability, PRO-1000, 31

T

texel, 549

texels, PRO_GetNumberOfAvailableTexels, 53,
131, 552

texture, 50

HasExpandedTexture, 55, 175, 557

polygons, 51

PRO_SetTexturePageSize, 53, 128, 552

texture functions, 81, 496, 607, 608

DeleteDataAfterLoading, 81, 509

DisableAlpha, 81, 505, 608

EnableAlpha, 81, 504, 608

FreeDataAfterLoading, 81, 508, 608

GetOrigTextureMapHeight, 81, 512, 608

GetOrigTextureMapWidth, 81, 511, 608

GetTextureFileName, 81, 510, 608

IsContourTexture, 81, 501, 608

IsMicrotexture, 81, 514, 608

IsTextureRGB, 81, 513, 608

KeepDataAfterLoading, 81, 507, 608

PRO_Texture constructor

color image, 81, 496, 607

monochrome image, 81, 498, 607

PRO_Texture destructor, 81, 500, 607

SetContourThreshold

8-bit, 81, 502, 608

normalized, 81, 503, 608

SetTextureFileName, 81, 506, 608

texture map, 549

texture memory

capacity, 35

MIP-mapped, 31

texturing, 549

timing, GetFrameRateEstimate, 55, 180, 557

TransformVertex, vertex functions, 80, 489, 605

translucency, 36

polygons, 50

translucency blending, LOD transitions, 51

translucent faces, 549

tri-linear interpolation, 31

U

unload dynamic model,

PRO_UnloadDynamicModel, 52, 115, 551

update rate, 36, 549

30 or 60 hertz, 31

PRO_SetUpdateRate, 53, 117, 551

UserMatrix, model functions, 63, 289, 573



Index – Next Page

V

- valid display select, IsValidDisplaySelect, 55, 165, 556
- vector, 549
- vertex functions, 80, 478, 604, 605, 606
 - GetCoordinates
 - double precision, 80, 491, 606
 - single precision, 80, 490, 606
 - GetTextureCoordinates, 80, 492, 606
 - GetVertexNormal, 80, 493, 606
 - IsShaded, 80, 495, 606
 - IsTextured, 80, 494, 606
 - OffsetVertex, 80, 487, 605
 - PRO_Vertex constructor
 - with no specified coordinates, 80, 479, 604
 - with specified coordinates, 80, 99, 478, 604
 - PRO_Vertex destructor, 80, 480, 604
 - ScaleVertex, 80, 488, 605
 - SetCoordinates
 - double precision, 80, 482, 604
 - single precision, 80, 100, 101, 481, 604
 - SetFixedShadingIntensity, 80, 486, 605
 - SetMorphCoordinates, double precision, 80, 483, 604
 - SetTextureCoordinates, 80, 484, 604
 - SetVertexNormal, 80, 485, 605
 - TransformVertex, 80, 489, 605
- vertex list, 50
- VerticesAreClockwise, polygon functions, 79, 463, 602
- VerticesAreCounterClockwise, polygon functions, 79, 464, 602
- video output contols, 43
 - gamma correction, 43
 - soft edge blending, 43
- viewpoint, 45, 549
 - API data types, 45
 - AttachViewpoint, 56, 190, 558
 - coordinate system, 40
 - DetachViewpoint, 56, 191, 558
 - GetViewpoint, 57, 213, 561
- viewpoint functions, 59, 236, 565, 566
 - AttachViewpoint, 59, 251, 566
 - DetachViewpointFromParent, 59, 252, 566
 - GetLocalForward, 59, 248, 566
 - GetLocalLeft, 59, 249, 566
 - GetLocalUp, 59, 250, 566
 - GetOrientation
 - double precision, 59, 246, 566
 - reference variables, 59, 247
 - GetPosition
 - double precision, 59, 244, 566
 - reference variable, 59, 245
 - GetStatus, 59, 238, 565
 - PRO_Viewpoint constructor, 59, 88, 89, 90, 92, 94, 95, 99, 100, 236, 565
 - PRO_Viewpoint destructor, 59, 237, 565
 - SetOrientation
 - double precision, 59, 89, 92, 95, 100, 241, 565
 - matrix of single precision, 59, 243, 565
 - single precision, 59, 242, 565
 - SetPosition
 - double precision, 59, 89, 92, 95, 100, 239, 565
 - single precision, 59, 240, 565



Index – Next Page

- viewport, 51, 549
 - API data types, 45
 - AttachChildViewpoint, 56, 192, 558
 - AttachViewport, 54, 144, 553
 - DetachChildViewpoint, 56, 193, 559
 - DetachViewport, 54, 145, 553
 - GetAttachedDevice, 57, 214, 561
 - SetFieldOFView
 - asymmetric, 56, 195, 559
 - symmetric, 56, 194, 559
 - SetViewportOff, 57, 209, 561
 - SetViewportOn, 57, 210, 561
- viewport functions, 56, 57, 183, 558, 559, 560, 561, 562, 563
 - AttachBlendTable, 56, 188, 558
 - AttachChildViewpoint, 56, 192, 558
 - AttachDatabase, 56, 89, 92, 95, 99, 100, 189, 558
 - AttachDefaultBlendTable, 56, 186, 558
 - AttachDefaultDatabase, 56, 187, 558
 - AttachViewport, 56, 190, 558
 - DetachChildViewpoint, 56, 193, 559
 - DetachViewport, 56, 191, 558
 - GetAttachedDevice, 57, 214, 561
 - GetDisplayExtents, 57, 217, 562
 - GetFieldOfView
 - asymmetric, 57, 216, 561
 - symmetric, 57, 215, 561
 - GetFogColor, 57, 222, 562
 - GetFogParameters, 57, 220, 562
 - GetFogRanges, 57, 221, 562
 - GetLightFogColor, 57, 223, 563
 - GetLightSourceColor, 57, 224, 563
 - GetLightSourceParameters, 57, 225, 563
 - GetLobeParameters, 57, 226, 563
 - GetPriority, 57, 227, 563
 - GetStatus, 56, 185, 558
 - GetSunIlluminationData, 57, 218, 562
 - GetSunVector, 57, 219, 562
 - GetViewpoint, 57, 213, 561
 - PRO_Viewport constructor, 56, 183, 558
 - PRO_Viewport destructor, 56, 184, 558
 - SetDisplayExtents, 56, 196, 559
 - SetDisplaySelect, 57, 211, 561
 - SetFieldOFView
 - asymmetric, 56, 195, 559
 - symmetric, 56, 194, 559
 - SetFogColor, 56, 202, 560
 - SetFogParameters, 56, 199, 560
 - SetFogRanges, 56, 201, 560
 - SetInvertFlags, 57, 212, 561
 - SetLightFogColor, 56, 203, 560
 - SetLightSourceColor, 56, 204, 560
 - SetLightSourceExtent, 57, 207
 - SetLightSourceParameters, 57, 205, 560
 - SetLightSourceStartRange, 57, 206
 - SetLobeParameters, 57, 208, 561
 - SetSunIlluminationData, 56, 197, 559
 - SetSunVector, 56, 198, 559
 - SetViewportOff, 57, 209, 561
 - SetViewportOn, 57, 210, 561
- viewport priority, GetPriority, 57, 227, 563
- viewports, HasAttachedViewports, 55, 167, 556



Index – Next Page

W

Windows NT, 30
workstation, host computer, 31
world, coordinate system, 38
write blend memory, WriteBlendMemory, 54, 152,
554
WriteBlendMemory, device functions, 54, 152,
554

Y

yaw, 549