POINTOOLS *Vortex*

API User Guide and Reference

1.5

Pointools Vortex API Documentation

API Version 1.5.1.0 May 2010

POINTOOLS CONFIDENTIAL

This document and parts of the information contained within are confidential.

THE VORTEX ENGINE

Introduction

The Pointools Vortex point cloud engine is the core component driving the Pointools product range. The engine has been developed and continuously refined over 5 years and has been used with data from practically every scanner available during this period. Pointools products are used in a wide range of industries including petro-chemical, security, forensics, defense, architecture, heritage, archeology, engineering, mining, marine, transport, planning and various governmental institutions.

Background

Pointools develop a range of products designed as easy to use solutions for working with scan data. Our debut product was Pointools View which enabled users to large volumes of visualize scan data, take measurements and produce ortho imagery.

Development of Pointools View started in late 2002. At this time medium range scanning solutions were limited to the Cyrax 2500 scanner which produced scans that where each 1million data points. Typical scan projects where composed of 10 or 20 scans or 50-100 scans at the top end. Our engine was designed accordingly and was able to manage the quantity of data that was typical of most scanning projects. View 1.0 excelled in the speed with which point clouds could be visualized and in this respect was unmatched at the time.

By late 2003 it was clear that a different approach was required as scan projects represented by files that were larger than a machine's available RAM where no longer uncommon. The engine was redesigned to manage these datasets efficiently and offer the highest performance of any platform for point cloud visualization on standard hardware.

This is the design on which Vortex is based today and it differs from the original approach. The 2004 engine introduced view based streaming as the method of loading data from disk enabling the viewer to start visualizing the scan data immediately. This is coupled with careful memory management to ensure optimal performance.

Since this time graphics technology has evolved and the bottlenecks have shifted, CPUs have become more commonly multi-core and RAM is inexpensive with most machines being equipped with at least 3Gb. The Vortex engine has adapted to these changes to ensure that whilst it retains the architecture of the 2004 engine it still delivers optimal performance on today's consumer grade and professional grade hardware.

Functionality

The Vortex engine comprises of the point cloud database, the display engine and various plug-in importers and exporters for scan data exchange.

The functionality of the engine can be summarized as:

- **IMPORT AND EXPORT OF SCAN DATA FORMATS**

The following formats are currently supported: Leica ptx, pts, ptz, ptg; Riegl 3dd, rxp, rsp; Topcon cl3, Faro fls, fws; Optech ixf; In addition Z+F zfc should be available soon. Data from other scanners can be imported via a flexible generic ascii importer and the ability to create the POD format is provide enabling addition of additional formats by users of the engine.

- **POINT CLOUD ORGANIZATION IN OPTIMIZED FILE STRUCTURE (POD FILE)**

The structure of Vortex pod files optimizes point data fetch by visible region and enables compression without loss of accuracy.

- **FILE AND SCENE MANAGEMENT**

File management includes opening files, closing files and temporarily unloading / reloading files.

Scene management includes view based loading of point cloud files and careful management of memory resources to deliver optimal performance.

- **POINT CLOUD RENDERING TO OPENGL CONTEXT**

Optimised rendering to OpenGL contexts. User rendering to non-GL contexts is also possible using the querying interface.

- **POINT QUERYING**

Point data can be extracted from the engine using the query functions.

- **POINT EDITING**

Point data can be selected, moved or copied between layers and hidden for segmentation or removal of unwanted artifacts

- **USER CHANNELS**

Additional channels of data can be added to point clouds at run-time and displayed as colour or point position offsets

Features

The Vortex engine has been subject to 6 years of development, improvement and redesign. Today a number of features differentiate the engine's features and performance industry wide:

- **INDEXED FOR HIGH PERFORMANCE**

Data is organized and indexed optimally for fast retrieval and display. Data is compressed but without meaningful loss of accuracy (in relation to capture accuracy) when importing data into the Vortex database format and no loss of density. Large multi-Gb collection of scan files can be imported from scanner formats and indexed into the Pointools pod format in a short time.

- **SUPPORT FOR MASSIVE POINT CLOUD DATASETS**

Multi-billion point datasets can be loaded into the Vortex engine. All of Vortex's functionality is tested with huge datasets to ensure end-users of the engine experience the fluidity and speed of workflow that is associated with Pointools products.

- **DISPLAY ENGINE AND SHADER FLEXIBILITY**

The Vortex engine includes a display component for rendering to OpenGL with a range of shading possibilities are supported by the display engine including rgb, intensity, height, edge and lighting. Shading methods can also be blended and configured via options.

The display engine prioritizes the data within the user's view or tool based area of interest. Scene management responds accordingly to ensure these areas are given load priority to deliver as smooth a visualization experience as possible.

It is also possible to build alternative display pipelines not based on OpenGL, in this case Vortex provides a fast query interface and software shading system that exactly replicates the OpenGL shader based system.

- **FAST AREA SELECTION**

Areas of the point cloud can be quickly selected using rectangle, fence, plane or 3d brush tools. The selection algorithm is extremely fast even on multi-billion point datatsets. If multiple cores are available on the CPU these are all utilized efficiently. Points can be moved or copied into one of seven layers.

- **FAST POINT QUERY AND EXTRACTION**

Point queries can be run quickly against large point cloud datasets and the points extracted to a client buffer. Queries can be configured to return the full query result, a representational sample or split the results into multiple buffers allowing for parallel processing.

- **RAY INTERSECTION FOR POINT SNAPPING**

Ray intersections with the point cloud or nearest point computation can be performed via the API. This provides an easy way to implement point snapping for CAD applications.

THE VORTEX API

Background

The Pointools Vortex API provides the point cloud loading, display and extraction capabilities in Pointools products, including Pointools 4 Rhino and Model for AutoCAD. The API has a clean and easy to integrate interface provided by a single header and binary dll file.

API Design Philosophy

An understanding of the API's design philosophy will be useful in understanding the API usage. A number of key objectives have determined the design:

- **FULL OBJECT LIFETIME MANAGEMENT**

Client code does not have to allocate or manage any objects to use the API. All objects are managed by the engine and referenced via a handle. This frees the client code from all memory management tasks directly related to the use of the API.

- **COMPATIBLE WITH MOST COMPILERS**

The API is binary compatible almost any win32 C++/C compiler since functions are imported at run-time and the calling convention is specified.

- **ABILITY TO EXPOSE ONLY PARTS OF THE INTERFACE**

Applications requiring only part of the interface need only import those functions

- **ABILITY TO EASILY BIND TO DIFFERENT LANGUAGES**

The interface is straight C and can be bound to other languages, or wrapped by C++ into an object model suiting the application

- **UPDATABLE WITHOUT BREAKING CHANGES TO THE INTERFACE**

The engine can be extended, improved and updated by Pointools in most cases without breaking the interface requiring no re-compile for client code.

- **MINIMAL SETUP FOR CLIENT CODE**

Client code implementation can be minimal and quick to implement. A straight C interface means that clients of the API may also choose to better adapt the API to their architecture by means of a C++ wrapper.

- **BINARY DEPENDENCIES ON STANDARD LIBRARIES ONLY**

No additional third party libraries are required; the only requirements are basic system libraries and OpenGL

The Vortex API - Getting Started

The Pointools Vortex API is contained in a single dll file containing scene management and display engine functionality. Note that at this time scan format import and export is not implemented in the API, these functions are available in the PTFormatsIO API. Please contact Pointools support for further information if you require this functionality.

LIBRARY LOADING AND INITIALISATION

Source code to perform the loading of the library and import of functions with error checking is provided with the library. Inclusion and use of these files (pointoolsAPI_import.h/cpp) is the recommended method for importing the library into an application.

After the API is loaded, the following code will initialize it:

```
ptInitialize( clientLicenseCode );
```

You will need to obtain the client license code from Pointools otherwise this will fail. This code is usually provided in the vortexEvalCode.c file distributed with the evaluation pack.

POD FILE LOADING

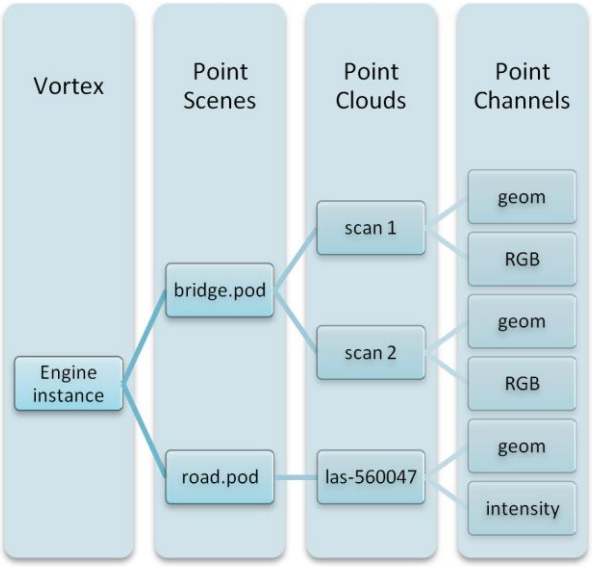
POD files (**PO**int **D**atabase) are the proprietary optimized file type directly loaded by the Vortex Engine. The format is compact and structured for rapid partial loading. The Vortex engine pages the POD file according to the current view requirements using a local cache to minimize IO reads. When querying for points or rendering a large point cloud the engine may also supplement the cache by loading point channels in the current thread.

A POD file consists of:

1. File version and basic structure
2. Extended Meta data
3. Point cloud structure
4. Point data channels

A single POD file contains a point cloud *scene* which contains one or more *point clouds* composed of at least a *geometry* channel and optionally *intensity, rgb* and *normal* data channels. The Vortex engine can load multiple POD files in a single session.

Vortex Engine run-time data model



Run-time data model in the Vortex Engine

Note that the Vortex engine always loads POD files read only enabling multiple engine sessions to access the same file.

The following code segment opens a windows file browser and allows the user to select one or more pods file to load. A handle is returned if the load is successful. This will only be the first handle in the case of multiple files.

In the listing below the lists the handles for the files that have been loaded to std::cout.

```
PThandle ptscene = ptBrowseAndOpenPOD();

if (ptscene != PT_NULL)
{
	int newNumScenes = ptNumScenes();
	PThandle *handles = new PThandle[newNumScenes];
	ptGetSceneHandles( handles );

	for ( int sc=numScenes; sc<newNumScenes; sc++ )
	{
		std::cout << "Pod file: " << handles[sc] << std::endl;
	}
	delete [] handles;
}
```

Alternatively using `ptOpenPOD(PTstr*filename)` will load the file directly without displaying a file browser.

VIEWPORT MANAGEMENT

Pointools API will keep track of shading and other settings per viewport. In order to do this you must notify the API when the active viewport changes. You can use the `ptSetViewportByName(PTstr*name)` method to do this by providing a unique name for the viewport. Note that names like 'top'. 'perspective' ect will not work well since there maybe more than one viewport with a name like this. Also note that there is no need to 'Add' the viewport, although the command does exist in the API.

SHADER SETTINGS

These are very easy to change and include RGB / intensity, lighting, plane shading etc. Here are a few examples: Also see the 'simple' example project for more detail.

```
ptEnable( PT_LIGHTING ); // Enables lighting. Use ptDisable to disable settings.

// changes the intensity ramp to the currently selected index in a combo box.
ptShaderOptioni( PT_INTENSITY_SHADER_RAMP, m_cboIntensityRamp.GetCurSel() );

// set up material properties for lighting
ptShaderOptionf( PT_MATERIAL_AMBIENT, 0.3f );
ptShaderOptionf( PT_MATERIAL_DIFFUSE, 0.7f );
ptShaderOptionf( PT_MATERIAL_SPECULAR, 0.1f );
ptShaderOptionf( PT_MATERIAL_GLOSSINESS, 0.1f );
```

Note that these settings operate per viewport. To copy settings from the active viewport to the others use `ptCopyShaderSettingsToAll();`

DRAWING

Using the API draw functions will in most cases result in the simplest implementation and best performance when rendering points to the viewport. This is possible only when drawing into current OpenGL context (support for DirectX to follow) that you are using in your application. To do this simply use the `ptDrawGL(bool dynamic)` function. Note that creating an OpenGL context and ensuring that it is active is not handled by the API, you must do this yourself.

When using OpenGL the API will not setup a camera modelview or projection matrices, this must be performed by the host application. Once set the entire view setup can be read by Vortex using `ptReadViewFromGL`. This must be done before calling `ptDraw`.

The dynamic flag indicates that Vortex API should try to draw the scene quickly to maintain frame-rate. This may mean that the view is decimated and can be used to provide fast drawing whilst the user is navigating the scene. When the user has finished navigation (usually when the mouse button is up) a full draw settings dynamic to false should be performed.

You can also use `ptDrawSceneGL(PThandle scene)` to draw a particular scene (ie file).
The drawing mode can be over-ridden be using `ptOverrideDrawMode` with `PT_DRAW_MODE_INTERACTIVE`, `PT_DRAW_MODE_STATIC` and `PT_DRAW_MODE_DEFAULT` (to reset). This is useful when a peripheral part of the client code needs to influence the draw mode.

For none OpenGL drawing the point querying interface can be used with settings specifically designed for retrieval of point buffers for drawing. This requires that a valid view comprising of projection, eye space transform and viewport size have been setup via the `ptSetView…` functions. Thereafter `ptCreateFrustumPointsQuery` should be used to setup a query that returns points within the viewing frustum. To control the volume of points that are returned `ptSetQueryDensity` can be used with `PT_QUERY_DENSITY_LIMIT` to limit the overall number of points to a specific amount or `PT_QUERY_DENSITY_VIEW` to return an optimal number of points for the current view.

See the 'tuning' example for an implementation of a simple renderer that uses this method.

UNITS

To ensure correct scaling you should set the host units using `ptSetHostUnits` according to the units used in your application to one of the following:

```
PT_METERS
PT_CENTIMETERS
PT_MILLIMETERS
PT_FEET
PT_FEET_US_SURVEY
PT_INCHES
```

The default units are meters. Note that POD files are unitless by default.

SELECTION REGIONS

The Vortex API allows the selection of areas of the point cloud and subsequent hiding or retrieval of points. Selection can be in one of three of the following modes:

`PT_EDIT_MODE_SELECT`	– Points are selected in the selection region
`PT_EDIT_MODE_UNSELECT`	– Points are deselected in the selection region
`PT_EDIT_MODE_UNHIDE`	– Points are unhidden in the selection region

The mode is set by using `ptSetSelectPointsMode` with one of the constants above.

Selection of points can be performed in screen space (often by the user) or in 3D project space. The following functions perform selection in screen space:

`ptSelectPointsByRect`	Selects points using a screen rectangle. This method is highly optimized and in most cases can be used to give instantaneous feedback to the user.
`ptSelectPointsByFence`	Selects points using a screen fence (polygon). This method is more flexible than the use of a rectangle but does not offer the same performance.

In 3D project space the following functions can be used to select points:

ptSelectPointsByBox	Selects points using an axis aligned bounding box. This method is extremely quick and can be used to implement a user real-time 3d brush selector.
ptSelectPointsBySphere	Selects points using a sphere. This method is also extremely quick and can be used to implement a user real-time 3d brush selector.

The point selection can be inverted and cleared by the use of `ptInvertSelection` and `ptUnselectAll`. Selected points can be hidden with `ptHideSelected`.

It is possible to store the current selection and visible state for later retrieval both within session memory and on disk. Please see the API reference for details on how to do this.

New to version 1.3 of the API is the ability to move or copy points between layers. These can be used to provide CAD layer like functionality or for point catergorisation or segmentation. Layers can be shown / hidden as well as locked.

Currently 6 layers are provided, the maximum number of layers is provided by PT_MAX_LAYERS. This number will be increased in future versions of the API.

QUERYING

Points and their attributes can be extracted from the Vortex API using the querying part of the API interface. A query object is first created using one of a number of methods that return a handle to the object. For example to the `ptCreateBoudingBoxQuery` can be used to create a query for points within an axis aligned bounding box. Once the query is created, client code should allocate a buffer for at least the point geometry, this can be either an array of floats or doubles. In addition buffers for intensity, rgb and normals may also be allocated if required.

Points can then be extracted using the `ptGetQueryPoints` functions which return the number of points extracted. Depending on the size of the buffer and the number of points in the query you may need to call this function a number of times until it returns zero. Each iteration will fill the buffer until all the query points have been extracted. This is most conveniently performed in a while loop with the number of points returned as a condition.

Note that the `ptGetQueryPoints` function takes a number of pointers to intensity, rgb and other buffers. These are optional and it is safe to pass null pointers for attributes you do not need other than point geometry which cannot be a null pointer.

Queries take minimal resources until they are actually executed. This is because a query is also only evaluated when it is executed using `ptGetQueryPoints`. As a result, a SelectedPoints query would return points that are selected when the query is *executed* not those selected when the query was created.

Once a query has returned all its points it will not return more points until `ptResetQuery` is called with the query handle.

USER CHANNELS

Point clouds have a geometry channel (x,y,z) and often RGB and intensity channels also. Additional run-time channels can be added to store arbitrary numeric per point data. These are called User Channels and are not stored directly in the POD file format unlike RGB and intensity. There are a number of applications where this is extremely useful. For example holding temporary values whilst processing points in an algorithm, a user channel removes the burden of tracking point references in client code and can also be stored out-of-core enabling efficient use of memory.

User Channels can also be used to display results of an algorithm by telling Vortex to interpret values as a colour ramp position or RGB value.

A User Channel can be to persist between sessions by using the `ptSaveUserChannels` and `ptLoadUserChannels` file functions. When loaded the channels are automatically mapped to the correct point clouds if these are loaded in Vortex.

To create a user channel use the `ptCreateUserChannel` function and store the returned handle to the channel for onward reference. User channel values are read and written by creating queries and retrieving values using one of the `ptGetDetailedQueryPoints` functions in which additional parameters for User Channel access can be specified. These values are returned to a local buffer you provide, this buffer can be both read from and written to. Changes made to the values in the buffer can be written back to the User Channel by calling `ptSubmitPointChannelUpdate`.

For a demonstration of User Channel usage see the ‘moving-points’ example project.

VORTEX API EXAMPLES

The Pointools Vortex API is distributed with a number of example projects in the 'examples' folder. These are contained in a single Visual Studio 2005 solution file, *examples.sln* which is set up to build out-of-the-box without any changes or specification of additional folders.

Most of the examples use GLUI, a simple OpenGL / glut based UI library to set up the user interface and as much of the peripheral application framework code as possible has been placed in *src/appframework.cpp* to keep the example code clear. The GLUI is used under a LGPL license requiring any modifications to the source code of the library to be made available under the same license terms. To obtain the modified source if required, please contact vortex@pointools.com

SIMPLE

The simple project demonstrates basic use of the API including file loading, shading options and basic view settings for the OpenGL renderer

EDITING

The editing project demonstrates the point selection, point hiding and layers capability of the engine. A number of editing tools are implemented, including screen space selections such as rectangle and polygon and 3d brush selectors.

The available point 6 layers are shown in 6 rows of buttons below the editing tools:

The ***V*** button is used to control visibility

The ***L*** button is used to lock a layer, preventing editing operations from affecting this layer

The ***Layer*** button is used to make a layer current. The current layer is also made visible.

Once points are selected, switching the current layer and using the ***Copy*** or ***Move*** buttons will copy or move points into the current layer.

The example also demonstrates a simple query based renderer, which although still using OpenGL, queries the engine for points and renders these in the example code.

MAKE BMP

The *makebmp* project demonstrates the use of bitmap viewport contexts to create and save a bitmap of the current view

SNAPPING

The *snapping* project demonstrates the use of the ray intersection and nearest point functions to implement accurate point snapping.

MOVE PNTS

The *movepnts* project demonstrates the use of User Channels to add per point values on the whole of or part of a point cloud. Using the OpenGL renderer the channel is rendered as an offset to the point position resulting in a distortion of the point cloud. Though there is little direct application of distorting a point cloud in this way, the example is intended to illustrate use of User Channels and persistence of User

Channels between sessions. Note that persistence will not work correctly on POD files generated from Pointools products before May 2009.

TUNING

The *tuning* example demonstrates some of the available engine tuning parameters and implements a simple query based renderer. The example takes the query renderer further than the *editing* example with a number of settings available that change the behavior of the query renderer.

METADATA

The *metadata* example demonstrates the reading of metadata from a POD file and displays the header information in a grid

VORTEX API COMMAND REFERENCE

Initialization

Initialize

DEFINITION

```
PTbool ptInitialize( const PTubyte *license )
```

DESCRIPTION

Initialises the Vortex engine and checks license state. This must be the called before any other Vortex API function.

PARAMETERS

license	This is a license string provided by Pointools usually held in a variable declared in a file named license.c

RETURN VALUE

PT_TRUE if initialisation is successful.

IsInitialized

DEFINITION

```
PTbool ptIsInitialized
```

DESCRIPTION

Checks if the Vortex engine has been initialized by a previous successful call to `ptInitialise`.

RETURN VALUE

PT_TRUE if initialisation was successful.

SetWorkingFolder

DEFINITION

```
PTvoid ptSetWorkingFolder( const PTstr folder )
```

DESCRIPTION

Sets the working folder for the engine resources. In Vortex 1.5 this the only resources that are loaded via files are the shader files. In Vortex 1.4 and earlier this would include the *ramps* folder containing gradient images for the intensity and plane shader, however this is now longer required and the ramps folder can safely be removed from the working folder.

PARAMETERS

folder	A file path to the working folder

GetWorkingFolder

DEFINITION
```
const PTstr ptGetWorkingFolder
```

DESCRIPTION
Gets the working folder previously set with `ptSetWorkingFolder`

RETURN VALUE
The working folder path as a string

GetVersionString

DEFINITION
```
const PTstr ptGetVersionString
```

DESCRIPTION
Returns a string representing the current Vortex version and build number

RETURN VALUE
A string representing the current Vortex version and build number

---

GetVersionNum

DEFINITION
```
PTvoid ptGetVersionNum( PTubyte *version );
```

DESCRIPTION
Returns a 4 byte size numbers representing the current Vortex version number. For example version 1.5.1.0 would return numbers 1,5,1 and 0. Ie version[0] = 1, version[1]=5 and so on.

PARAMETERS

version	An array of at least 4 unsigned bytes

Release

DEFINITION
```
PTvoid ptRelease
```

DESCRIPTION
Releases all resources associated with the use of the engine. This should only be called when use of the Vortex API is no longer required within the session as part of the application clean-up.

Handle Management

GetCloudHandleByIndex

DEFINITION

```
PThandle ptGetCloudHandleByIndex( PThandle scene, PTuint cloud_index )
```

DESCRIPTION

Returns a point cloud's handle using its index within a given scene. Most functions referencing point clouds require a handle rather than an index to the point cloud. This function enables conversion of a point clouds index in the scene file to a point cloud handle.

PARAMETERS

scene	A valid handle to a scene
index	The zero-based index of the point cloud within the scene

RETURN VALUE

The handle to the point cloud is returned if the function succeeds otherwise a zero handle is returned. Reasons for failure are an invalid scene handle or a point cloud index that is out of bounds.

GetNumCloudsInScene

DEFINITION

```
PTuint ptGetNumCloudsInScene( PThandle scene )
```

DESCRIPTION

Return the number of point clouds that are contained in an loaded scene

PARAMETERS

scene	A valid handle to a scene

RETURN VALUE

The number of point clouds in the scene file. Zero is returned If the scene handle is invalid

Files

OpenPOD

DEFINITION

```
PThandle ptOpenPOD( const PTstr filepath )
```

DESCRIPTION

Opens a POD file and reads the structure into the Vortex engine. A single POD file represents a point cloud scene that is comprised of one or more point clouds.

PARAMETERS

filepath	The file path to the POD file

RETURN VALUE

A valid handle is returned if the operation was successful. A valid handle is a handle with a non-zero value. This is a session handle that used to reference the scene file held in the engine.

OpenPODStructuredStorageStream

DEFINITION

```
PThandle ptOpenPODStructuredStorageStream(const PTstr filepath, PTvoid *stream)
```

DESCRIPTION

Uses the given COM Structured Storage IStream (cast to void) to open a POD file and reads the structure into the Vortex engine. A single POD file represents a point cloud scene that is comprised of one or more point clouds. The filepath is used for reference only and never to actually open a file. The filepath must begin with a URI style protocol typing PTSS e.g. “PTSS://myfile.pod”

PARAMETERS

filepath	URI based name of POD used for reference (must begin PTSS:)
stream	COM structured storage stream already opened on a POD file in a storage.

RETURN VALUE

A valid handle is returned if the operation was successful. A valid handle is a handle with a non-zero value. This is a session handle that used to reference the scene file held in the engine.

IsOpen

DEFINITION

```
PThandle ptIsOpen( const PTstr filepath )
```

DESCRIPTION

Checks if a file is opened by the engine

PARAMETERS

filepath	The file path to the POD file

RETURN VALUE

The file’s corresponding scene handle if the file is open, otherwise a null handle.

BrowseAndOpenPOD

DEFINITION

```
PThandle ptBrowseAndOpenPOD
```

DESCRIPTION

A helper function that opens a win32 file chooser dialog box and allows the user to pick files to open

RETURN VALUE

The handle to the first scene file opened. The scene management functions can be used to determine if more than one file was opened.

Scene Management

NumScenes

DEFINITION

```
PTint ptNumScenes
```

DESCRIPTION

Returns the number of scene files currently managed by the Vortex engine

RETURN VALUE

The number of scene files currently managed

GetSceneHandles

DEFINITION

```
PTint ptGetSceneHandles( PThandle* handles )
```

DESCRIPTION

Get an array of scene handles for all the scenes currently managed by the engine

PARAMETERS

handles	An array of PThandles. To prevent buffer overrun it is important to first check the number of scenes to ensure the buffer is sufficiently large enough

RETURN VALUE

The number of scene handles returned

SceneInfo

DEFINITION

```
PTbool ptSceneInfo( PThandle scene, PTstr name, PTint &clouds,
                    PTuint &num_points, PTuint &specification, PTbool
                    &loaded, PTbool &visible )
```

DESCRIPTION

Gets information about a scene

PARAMETERS

scene	The scene's handle
name	String buffer to return the name of the scene. This should be at least 64 characters in size.
clouds	Int (by ref) to return the number of clouds in the scene.
num_points	Unsigned int (by ref) to return the number of points in the scene.
specification	Returns details about the point cloud data as a bit mask of the following values

PT_HAS_INTENSITY	One or more of the point clouds have an intensity channel
PT_HAS_RGB	One or more of the point clouds have a colour RGB channel
PT_HAS_NORMAL	One or more of the point clouds have a normal RGB channel

loaded *Boolean* (by ref) to return a scene's loaded state

visible *Boolean* (by ref) to return a scene's visibility state

RETURN VALUE
PT_TRUE if successful. Passing an invalid scene handle will return PT_FALSE

SceneFile

DEFINITION
```
const PTstr ptSceneFile( PThandle scene )
```

DESCRIPTION
Gets the file path to a scene's POD file

PARAMETERS
scene The scene's handle

RETURN VALUE
A const C string with a path to the scene's POD file. A null pointer is returned if the scene handle is invalid.

CloudInfo

DEFINITION
```
PTres ptCloudInfo( PThandle cloud, PTstr name, PTuint &num_points,
                   PTuint &specification, PTbool &visible )
```

DESCRIPTION
Gets information about a cloud within a scene.

PARAMETERS
cloud The cloud's handle. To form a valid cloud handle from the scene handle and cloud index use the `ptGetCloudHandleByIndex` function

name String buffer to return the name of the cloud. This should be at least 64 characters in size.

num_points Unsigned int (by ref) to return the number of points in the cloud.

visible *Boolean* (by ref) to return a cloud's visibility state

specification Returns details about the point cloud data as a bit mask of the following values

PT_HAS_INTENSITY	The point cloud has an intensity channel
PT_HAS_RGB	The point cloud has a colour RGB channel
PT_HAS_NORMAL	The point cloud has a normal RGB channel

RETURN VALUE
A value indicating the result of the call. Possible results are:

`PTV_INVALID_HANDLE`	The point cloud handle is invalid
`PTV_SUCCESS`	The function completed successfully

SceneBounds

DEFINITION

```
PTres ptSceneBounds( PThandle scene, PTfloat *lower3, PTfloat *upper3 );
```

DESCRIPTION

Retrieves the bounding extents of a scene in world coordinates.

PARAMETERS

scene	The scene's handle
lower3	An array of 3 float values to receive the lower extent of the box
upper3	An array of 3 float values to receive the upper extent of the box

RETURN VALUE

A value indicating the result of the call. Possible results are:

`PTV_INVALID_HANDLE`	The point cloud handle is invalid
`PTV_SUCCESS`	The function completed successfully

CloudBounds

DEFINITION

```
PTres ptCloudBounds( PThandle cloud, PTfloat *lower3, PTfloat *upper3 );
```

DESCRIPTION

Retrieves the bounding extents of a point cloud in world coordinates.

PARAMETERS

scene	The point cloud's handle
lower3	An array of 3 float values to receive the lower extent of the box
upper3	An array of 3 float values to receive the upper extent of the box

RETURN VALUE

A value indicating the result of the call. Possible results are:

`PTV_INVALID_HANDLE`	The point cloud handle is invalid
`PTV_SUCCESS`	The function completed successfully

ShowScene

DEFINITION

```
PTres ptShowScene( PThandle scene, PTbool visible );
```

DESCRIPTION

Sets the visibility of all point clouds contained in a scene file. The visibility will affect subsequent query and draw operations.

PARAMETERS

scene	The scene's handle
visible	The new desired visible state for scene

RETURN VALUE

A value indicating the result of the call. Possible results are:

`PTV_INVALID_HANDLE`	The scene handle is invalid
`PTV_SUCCESS`	The function completed successfully

ShowCloud

DEFINITION

```
PTres ptShowCloud( PThandle cloud, PTbool visible );
```

DESCRIPTION

Sets the visibility of a point cloud. The visibility will affect subsequent query and draw operations on the point cloud.

PARAMETERS

scene	The point cloud's handle
visible	The new desired visible state for point cloud

RETURN VALUE

A value indicating the result of the call. Possible results are:

`PTV_INVALID_HANDLE`	The point cloud handle is invalid
`PTV_SUCCESS`	The function completed successfully

IsSceneVisible

DEFINITION

```
PTbool ptIsSceneVisible( PThandle scene );
```

DESCRIPTION

Returns the visibility setting for a scene as previously set by `ptShowScene`

PARAMETERS

scene	The scene's handle

RETURN VALUE

A Boolean value indicating the scene's visible setting

IsCloudVisible

DEFINITION

```
PTbool ptIsCloudVisible( PThandle cloud );
```

DESCRIPTION

Returns the visibility setting for a scene as previously set by `ptShowScene`

PARAMETERS

scene The point cloud's handle

RETURN VALUE

A Boolean value indicating the point cloud's visible setting

UnloadScene

DEFINITION

```
PTres ptUnloadScene( PThandle scene )
```

DESCRIPTION

Unloads the scene's point data from the engine. Note that this retains a reference to the scene and does not remove the scene from the engine. This can be used to free memory used by a point cloud when that point cloud is not required for display or query but a referenced must be retained.

PARAMETERS

scene The scene's handle.

RETURN VALUE

A value indicating the result of the call. Possible results are:

`PTV_INVALID_HANDLE`	The scene handle is invalid
`PTV_SUCCESS`	The function completed successfully

ReloadScene

DEFINITION

```
PTres ptReloadScene( PThandle scene )
```

DESCRIPTION

Reloads the scene's point data which was previously unloaded.

PARAMETERS

scene The scene's handle.

RETURN VALUE

A value indicating the result of the call. Possible results are:

`PTV_INVALID_HANDLE`	The scene handle is invalid
`PTV_SUCCESS`	The function completed successfully

RemoveScene

DEFINITION

```
PTres ptRemoveScene( PThandle scene )
```

DESCRIPTION

Removes the scene from the engine and closes the associated POD file. No reference to the scene POD file is retained in the engine session.

PARAMETERS

scene The scene's handle.

RETURN VALUE

A value indicating the result of the call. Possible results are:

`PTV_INVALID_HANDLE`	The scene handle is invalid
`PTV_SUCCESS`	The function completed successfully

RemoveAll

DEFINITION

```
PTvoid ptRemoveAll
```

DESCRIPTION

Removes all scenes from the engine and closes all POD files. All references to scene files are retained in the engine session.

GetLowerBound

DEFINITION

```
PTbool ptGetLowerBound( PTdouble *lower )
```

DESCRIPTION

Gets the lower coordinate bound of all point cloud data currently managed by the engine

PARAMETERS

lower An array of 3 doubles to receive the point representing the lower bound

RETURN VALUE

PT_TRUE if successful. PT_FALSE is returned if no data is managed by the engine.

GetUpperBound

DEFINITION

```
PTbool ptGetUpperBound( PTdouble *upper )
```

DESCRIPTION

Gets the upper coordinate bound of all point cloud data currently managed by the engine

PARAMETERS

upper An array of 3 doubles to receive the point representing the upper bound

RETURN VALUE

PT_TRUE if successful. PT_FALSE is returned if no data is managed by the engine.

Meta Data

ReadPODMeta

DEFINITION

```
PThandle ptReadPODMeta( const PTstr filepath )
```

DESCRIPTION

Reads the meta data from a POD file without loading the point cloud data or adding the POD scene to the Vortex engine.

PARAMETERS

filepath The path to the POD file as a string

RETURN VALUE

A handle to the meta data is returned which can be used in meta data query functions. A zero handle indicates failure.

GetMetaDataHandle

DEFINITION

```
PThandle ptGetMetaDataHandle( PThandle sceneHandle )
```

DESCRIPTION

Returns meta data handle for a POD file already loaded into the Vortex engine. The handle's lifetime is independent of the scene's (POD file's) lifetime so that closing the file does not invalidate the meta data handle.

PARAMETERS

sceneHandle A handle to the POD file scene already loaded in the Vortex engine

RETURN VALUE

A handle to the meta data is returned which can be used in meta data query functions

---

GetMetaData

DEFINITION

```
PTres ptGetMetaData( PThandle metadataHandle, PTstr name,
                     PTint &num_clouds, PTuint64 &num_points,
                     PTuint &scene_spec,
                     PTdouble *lower3, PTdouble *upper3 )
```

DESCRIPTION

Returns basic meta data from a POD file using its meta data handle created previously with one of the above functions. Valid basic meta data is returned for any POD file supported by the Vortex engine.

PARAMETERS

metadataHandle A handle to the meta data created using ptReadPODMeta or ptGetMetaDataHandle

name A character buffer to receive the name of the Scene. The buffer must be at least 128 characters in size.

num_clouds	Receives the number of point clouds in the Scene file
num_points	Receives the total num of points in the Scene file, note the use of 64bit unsigned integer since the number of points can exceed the capacity of a 32bit integer.
scene_spec	Returns details about the point cloud data as a bit mask of the following values

PT_HAS_INTENSITY	The point cloud has an intensity channel
PT_HAS_RGB	The point cloud has a colour RGB channel
PT_HAS_NORMAL	The point cloud has a normal RGB channel

lower3	A pointer to an array of 3 doubles to receive the lower bounding box extent of the point cloud Scene
upper3	A pointer to an array of 3 doubles to receive the lower bounding box extent of the point cloud Scene

RETURN VALUE

A value indicating the result of the call. Possible results are:

PTV_INVALID_HANDLE	The metadata handle is invalid
PTV_SUCCESS	The function completed successfully

GetMetaTag

DEFINITION

```
PTres ptGetMetaTag( PThandle metadataHandle, const PTstr tagName,
                    PTstr value )
```

DESCRIPTION

Returns a specific meta data tag from the Scene. Note that the meta tag may not be available in the Scene.

PARAMETERS

metadataHandle	A handle to the meta data created using ptReadPODMeta or ptGetMetaDataHandle
tagName	The name of the tag formatted as “section.tagname”. The following values are accepted :

"Instrument.ScannerManufacturer"	The Manufacturer of the sensor used to capture the data
"Instrument.ScannerModel"	The name of the scanner Model
"Instrument.ScannerSerial"	The serial number of the scanner
"Instrument.CameraModel"	The camera model used to capture RGB

"Instrument.CameraSerial"	The camera serial number
"Instrument.CameraLens"	The camera lens
"Survey.Company"	The company that captured the data
"Survey.Operator"	The operator name
"Survey.ProjectName"	The project name
"Survey.ProjectCode"	The project code, does not have to conform to particular convention
"Survey.DateOfCapture"	The date of capture, must be specified as YYYY-MM-DD, for example 2009-06-29
"Survey.Site"	Text describing the site or object captured
"Survey.SiteLong"	Site's Longitude. This does not affect data positioning and is for information only
“Survey.SiteLat"	Sites Latitude. This does not affect data positioning and is for information only
"Survey.CoordinateSystem"	Coordinate system descriptor. This does not affect data positioning and is for information only
"Survey.ZipCode" or “Survey.Postcode”	Zip or Postal code of site
"Description.Description"	Description of the scans contents
"Description.Keywords"	Keywords describing data, multiple keywords are separated by semicolons
"Description.Category"	Category, one of: "Aerial Lidar" "Terrestrial Phase Based" "Terrestrial Time of Flight" "Mobile mapping" "Bathymetric" "Photogrammetric"

"Synthesized"

"Audit.ScanPaths"	Original file paths of source input files. To retrieve multiple file paths the function can be called multiple times each time returning one of the file paths until an empty string is returned.
"Audit.OriginalNumScans"	Number of original scans, note this may differ from number of original files
"Audit.CreatorApp"	The application that created the POD file
"Audit.Generation"	The generation of the file, where each modification and resave of the file increments the generation number.
“Audit.DateCreated”	The date the file was originally created, this may differ to the system date created value of the file.

value A string buffer to accept the value of the tag. The buffer should be at least MAX_META_STR_LEN characters long to prevent buffer overrun

RETURN VALUE

A value indicating the result of the call. This will be one of the following values:

`PTV_METATAG_NOT_FOUND`	The meta tag indicated by tagName was not found
`PTV_METATAG_EMPTY`	The meta tag is empty
`PTV_SUCCESS`	The function completed successfully

FreeMetaData

DEFINITION

```
PTvoid ptFreeMetaData( PThandle metadataHandle )
```

DESCRIPTION

Frees memory used by the Vortex engine to hold meta data associated with this handle.

PARAMETERS

metadataHandle A handle to the meta data to be freed. This handle must have been previously created using ptReadPODMeta or ptGetMetaDataHandle. The handle will be invalid after this call.

NumUserMetaSections

DEFINITION

```
PTint ptNumUserMetaSections( PThandle metadataHandle )
```

DESCRIPTION

Returns the number of user meta data sections

PARAMETERS

metadataHandle A handle to the meta data created using ptReadPODMeta or ptGetMetaDataHandle

RETURN VALUE

The number of user meta data sections

NumUserMetaTagsInSection

DEFINITION

```
PTint ptNumUserMetaTagsInSection( PThandle metadataHandle, PTint
                                  sectionIndex )
```

DESCRIPTION

Returns the number of user meta data tags in a section.

PARAMETERS

metadataHandle A handle to the meta data created using ptReadPODMeta or ptGetMetaDataHandle

sectionIndex The zero based index of the section of meta data tags

RETURN VALUE

The number of user meta data tags in a section

UserMetaSectionName

DEFINITION

```
const PTstr ptUserMetaSectionName( PThandle metadataHandle, PTint
                                   sectionIndex )
```

DESCRIPTION

Returns the number of user meta data tags in a section.

PARAMETERS

metadataHandle A handle to the meta data created using ptReadPODMeta or ptGetMetaDataHandle

sectionIndex The zero based index of the section of meta data tags

RETURN VALUE

The name of the user meta data section as a pointer to a string held within the Vortex engine. The string should be copied as its lifetime cannot be guaranteed.

GetUserMetaTagByIndex

DEFINITION

```
PTres  ptGetUserMetaTagByIndex( PThandle metadataHandle,
                                PTint section_index, PTint tag_index,
                                PTstr name, PTstr value )
```

DESCRIPTION

Gets a user meta data tag name and value pair by the section and tag index.

PARAMETERS

metadataHandle	A handle to the meta data created using ptReadPODMeta or ptGetMetaDataHandle
sectionIndex	The zero based index of the section of meta data tags
tagIndex	The zero based index of the user meta tag within the section
name	A string buffer to receive the name of the tag. The buffer should be at least 64 chars in size.
value	A string buffer to receive the value of the tag. The buffer should be at least MAX_META_STR_LEN chars in size.

RETURN VALUE

A value indicating the result of the call. This will be one of the following values:

PTV_INVALID_HANDLE	The metadata handle provided is not valid
PTV_VALUE_OUT_OF_RANGE	the section index or tag index is out of bounds
PTV_VOID_POINTER	The name or value parameter is a null pointer
PTV_SUCCESS	The function completed successfully

GetUserMetaTagByName

DEFINITION

```
PTres  ptGetUserMetaTagByName( PThandle metadataHandle,
                               PTstr sectionDotName, PTstr value )
```

DESCRIPTION

Gets a user meta data tag name and value pair by the section and tag index.

PARAMETERS

metadataHandle	A handle to the meta data created using ptReadPODMeta or ptGetMetaDataHandle
sectionDotName	The case-sensitive name of the meta tag formatted as “section.name”
value	A string buffer to receive the value of the tag. The buffer should be at least MAX_META_STR_LEN chars in size.

RETURN VALUE

A value indicating the result of the call. This will be one of the following values:

`PTV_INVALID_HANDLE`	The metadata handle provided is not valid
`PTV_METATAG_NOT_FOUND`	The metatag indicated by sectionDotName was not found
`PTV_VOID_POINTER`	The name or value parameter is a null pointer
`PTV_SUCCESS`	The function completed successfully

Transformation

SetCloudTransform

DEFINITION

```
PTres ptSetCloudTransform( PThandle cloud, const PTdouble *transform4x4,
                           PTbool row_order )
```

DESCRIPTION

Sets the user transformation for a point cloud. If the point cloud is stored in scan space and has a registration transformation, the user transformation is applied after the registration one.

PARAMETERS

cloud	A handle to the point cloud
transform4x4	An array of 16 double values representing a 4 by 4 transformation matrix
row_order	Boolean indicating the ordering of the values in the matrix. Use PT_TRUE to indicate row major ordering and PT_FALSE for column ordering.

RETURN VALUE

A value indicating the result of the call. This will be one of the following values:

PTV_INVALID_HANDLE	The metadata handle provided is not valid
PTV_VOID_POINTER	The transform4x4 is a null pointer
PTV_SUCCESS	The function completed successfully

SetSceneTransform

DEFINITION

```
PTres ptSetSceneTransform( PThandle scene, const PTdouble *transform4x4,
                           PTbool row_order )
```

DESCRIPTION

Sets the user transformation for a scene.

PARAMETERS

scene	A handle to the scene
transform4x4	An array of 16 double values representing a 4 by 4 transformation matrix
row_order	Boolean indicating the ordering of the values in the matrix. Use PT_TRUE to indicate row major ordering and PT_FALSE for column ordering.

RETURN VALUE

A value indicating the result of the call. This will be one of the following values:

PTV_INVALID_HANDLE	The metadata handle provided is not valid
PTV_VOID_POINTER	The transform4x4 is a null pointer
PTV_SUCCESS	The function completed successfully

GetCloudTransform

DEFINITION

```
PTres ptGetCloudTransform( PThandle cloud, PTdouble *transform4x4,
                           PTbool row_order )
```

DESCRIPTION

Retrieves the user transformation for a point cloud.

PARAMETERS

cloud	A handle to the point cloud
transform4x4	An array of 16 double values to receive the 4 by 4 transformation matrix
row_order	Boolean indicating the desired ordering of the values in the matrix. Use PT_TRUE to indicate row major ordering and PT_FALSE for column ordering.

RETURN VALUE

A value indicating the result of the call. This will be one of the following values:

PTV_INVALID_HANDLE	The metadata handle provided is not valid
PTV_VOID_POINTER	The transform4x4 is a null pointer
PTV_SUCCESS	The function completed successfully

GetSceneTransform

DEFINITION

```
PTres ptGetSceneTransform( PThandle scene, PTdouble *transform4x4,
bool row_order )
```

DESCRIPTION

Retrieves the user transformation for a scene

PARAMETERS

cloud	A handle to the scene
transform4x4	An array of 16 double values to receive the 4 by 4 transformation matrix
row_order	Boolean indicating the desired ordering of the values in the matrix. Use PT_TRUE to indicate row major ordering and PT_FALSE for column ordering.

RETURN VALUE

A value indicating the result of the call. This will be one of the following values:

PTV_INVALID_HANDLE	The metadata handle provided is not valid
PTV_VOID_POINTER	The transform4x4 is a null pointer
PTV_SUCCESS	The function completed successfully

CreateSceneInstance

DEFINITION

```
PThandle ptCreateSceneInstance( PThandle scene )
```

DESCRIPTION

Duplicates a scene and returns a new handle to this scene. The scene may be independently transformed and edited.

PARAMETERS

scene A handle to the scene to be edited

RETURN VALUE

The handle of the new scene instance if successful or else a zero handle to indicate failure

---

Persistence of Viewport Setup

GetPerViewportDataSize

DEFINITION
```
PTuint ptGetPerViewportDataSize
```

DESCRIPTION
Returns the maximum size of the binary blob as a number of bytes. The blob will contain all per viewport data intended for storage.

RETURN VALUE
The size of the binary blob in bytes

GetPerViewportData

DEFINITION
```
PTuint ptGetPerViewportData( PTubyte *data )
```

DESCRIPTION
Retrieves the per viewport settings as a binary blob. This can be written to disk for later restoration of per viewport shader settings.

PARAMETERS

data	an array of unsigned bytes at least the size returned by ptGetViewportDataSize

RETURN VALUE
The actual number of bytes returned in data. This value can be used to write the data to storage.

SetPerViewportData

DEFINITION
```
PTres ptSetPerViewportData( const PTubyte *data )
```

DESCRIPTION
Used to restore per viewport settings from the previously saved binary data.

PARAMETERS

data	A pointer to the binary blob containing the per viewport data as previously retrieved with ptGetPerViewportData.

RETURN VALUE
A value indicating the result of the operation. Valid values are:

PTV_INVALID_BLOCK_VERSION	The data blob contains errors and cannot be processed The data may have been created from a more recent version of the Vortex API than is being used.
PTV_SUCCESS	The operation completed successfully

Errors

GetLastErrorString

DEFINITION
PTstr **ptGetLastErrorString**

DESCRIPTION
Returns the latest error string generated by a Vortex API function call

RETURN VALUE
A immutable English language string describing the error

GetLastErrorCode

DEFINITION
PTres **ptGetLastErrorCode**

DESCRIPTION
Returns the last error code generated by the previous API function call. The error codes are defined in the PointoolsVortexAPI_ResultCodes.h header file. Values above zero are warnings and do not indicate an error. Values below zero are error codes that may require client side handling.

RETURN VALUE
The error code generated by a previous function call. This can be one of:

PTV_FILE_NOT_EXIST	The file that is attempted to be openend does not exist
PTV_FILE_NOT_ACCESSIBLE	The file that is attempted to be openend is not accessible by the file system. The file may be in a locked state.
PTV_FILE_WRONG_TYPE	The file being opened is the wrong type of file although the file extension may be correct
PTV_FILE_COM_ERROR	A COM error has occured. This is only returned by ptBrowseAndOpen file in which a standard file selector dialog is shown.
PTV_FILE_USER_CANCELLED	The user cancelled the file open operation. This is only returned by ptBrowseAndOpen file in which a standard file selector dialog is shown
PTV_FILE_ALREADY_OPENED	The file being attempted to be opened is already open in the Vortex engine
PTV_FILE_NOTHING_TO_WRITE	There is nothing to write in the channel file since there is no channel data

PTV_FILE_WRITE_FAILURE	A write operation failed
PTV_FILE_READ_FAILURE	General read failure
PTV_FILE_FAILED_TO_CREATE	The file failed to be created. This could be due to an invalid path, user rights or Anti-virus software blocking file creation.
PTV_UNKNOWN_ERROR	An unknown error has occurred
PTV_INVALID_PARAMETER	A supplied parameter is invalid
PTV_VALUE_OUT_OF_RANGE	A supplied value is out of range for the function
PTV_INVALID_OPTION	A supplied option value is not value. This happens if an incorrect preprocessor defined constant value is used for a function
PTV_INVALID_VALUE_FOR_PARAMETER	The value supplied is not valid for this parameter
PTV_INVALID_HANDLE	The supplied handle is not valid
PTV_VOID_POINTER	A void pointer was supplied where a pointer to memory was expected
PTV_NOT_INITIALIZED	The Vortex engine has not been initialised with ptInitialise
PTV_NOT_IMPLEMENTED_IN_VERSION	The function has not yet been implemented
PTV_OUT_OF_MEMORY	The function could not be completed due to lack of memory
PTV_LICENSE_EXPIRY	The vortex license has expired
PTV_LICENSE_MODULE_ERROR	There was an error processing the license
PTV_LICENSE_CORRUPT	The license code is corrupt and cannot be read
PTV_MAXIMUM_VIEWPORTS_USED	The maximum number of viewports have already been used and new ones cannot be created
PTV_INVALID_BLOCK_VERSION	A data blob being used to restore viewport or editing settings was created by a more recent version of the Vortex engine
PTV_METATAG_NOT_FOUND	The requested metatag could not be found
PTV_METATAG_EMPTY	The metatag has no value

Coordinate Handling

SetAutoBaseMethod

DEFINITION

```
PTvoid ptSetAutoBaseMethod( PTenum type )
```

DESCRIPTION

This function sets the APIs internal method used to reposition (rebase) geometry to minimize coordinate sizes. This is important when hardware acceleration is being used for rendering (ie OpenGL or Direct3D) in order to avoid loss of precision in the display pipeline.

PARAMETERS

type The method used to auto re-base (reposition) geometry, the following values can be used

`PT_AUTO_BASE_DISABLED`	Auto base is disabled
`PT_AUTO_BASE_CENTRE`	Geometry is centered at the origin
`PT_AUTO_BASE_REDUCE`	Geometry is repositioned to reduce the coordinate size if needed. This is done using a power of 10 round number.

For multiple point cloud files it may be important to ensure that the geometry is only repositioned once. For example in a CAD application where CAD geometry may already reference the first point cloud loaded. To ensure geometry is only re-positioned once add the `PT_AUTO_BASE_FIRST_ONLY` value with the Or (|) operator.

GetAutoBaseMethod

DEFINITION

```
PTenum ptSetAutoBaseMethod
```

DESCRIPTION

Gets the method current set for geometry rebasing.

RETURN VALUE

A PTenum representing the method in use. Valid values are listed in the *SetAutoBase* method documentation.

SetCoordinateBase

DEFINITION

```
PTvoid ptSetCoordinateBase( const PTdouble *coordinateBase )
```

DESCRIPTION

Explicitly set the coordinate base (origin) of all geometry held in the engine.

PARAMETERS

coordinateBase The coordinate representing the new base of the geometry as an array of 3 doubles.

GetCoordinateBase

DEFINITION

```
PTvoid ptSetCoordinateBase ( PTdouble *coordinateBase)
```

DESCRIPTION

Retrieve the current coordinate base. This could the coordinate set by the SetCoorindateBase function or automatically by the API.

PARAMETERS

coordinateBase	Array of 3 doubles to receive the coordinate.

Shading

Enable

DEFINITION
```
PTvoid ptEnable( PTenum shader_option )
```

DESCRIPTION
Enables a display option in the active viewport

PARAMETERS
Option The display option to be enabled. Valid values are:

PT_RGB_SHADER	RGB colour is displayed
PT_INTENSITY_SHADER	Intensity is displayed
PT_BLENDING_SHADER	RGB and Intensity are blended
PT_PLANE_SHADER	Shade by distance from a plane
PT_LIGHTING	Point lighting
PT_ADAPTIVE_POINT_SIZE	Point size is adapted to reduction in dynamic rendering
PT_FRONT_BIAS	More detail is rendered near front during dynamic rendering

---

Disable

DEFINITION
```
PTvoid ptDisable( PTenum shader_option )
```

DESCRIPTION
Disables a display option in the active viewport

PARAMETERS
shader_option The display option to be disabled. Valid values are the same as Enable

---

IsEnabled

DEFINITION
```
PTbool ptIsEnabled ( PTenum shader_option )
```

DESCRIPTION
Checks if a display option is enabled in the active viewport

PARAMETERS
shader_option The display option to be checked. Valid values are the same as Enable

---

PointSize

DEFINITION

```
PTres ptPointSize( PTfloat size )
```

DESCRIPTION

Sets the point display size for the active viewport

PARAMETERS

size The pixel size of each point rendered with ptDrawGL or ptDrawSceneGL. Valid values are between 1 and 10

RETURN VALUE

A value indicated the result of the function, possible values are:

PTVALUE_OUT_OF_RANGE	The point size specified is outside the acceptable range
PTV_SUCCESS	The function completed successfully

ShaderOption

DEFINITION

```
PTres ptShaderOptionf( PTenum option, PTfloat value )
PTres ptShaderOptionfv( PTenum option, PTfloat *value )
PTres ptShaderOptioni( PTenum option, PTint value )
```

DESCRIPTION

Sets a shader option for the active viewport.

PARAMETERS

option The shader option that is being set.

The following are valid options for `ptShaderOptionf`:

PT_PLANE_SHADER_DISTANCE	The distance over which the plane shader operates.
PT_PLANE_SHADER_OFFSET	The offset to the start of the plane shader
PT_MATERIAL_AMBIENT	The strength of the point material's ambient quality (lighting). Valid values are between 0 and 1
PT_MATERIAL_DIFFUSE	The strength of the point material's diffuse quality (lighting) Valid values are between 0 and 1
PT_MATERIAL_SPECULAR	The strength of the point material's specular quality (lighting) Valid values are between 0 and 1
PT_MATERIAL_GLOSSINESS	The glossiness of the point material's (lighting). Valid values are between 0 and 1 with lower values resulting in a Matte surface and higher values a shiny surface
PT_INTENSITY_SHADER_CONTRAST	The contrast setting for the intensity shader. Valid values are between 0 and 360
PT_INTENSITY_SHADER_BRIGHTNESS	The brightness setting for the intensity shader. Valid

values are between 0 and 360

The following are valid options for `ptShaderOptionfv`:

`PT_PLANE_SHADER_VECTOR`	The normal of the plane used in the plane shader. This is an array of 3 floats and should be a unitized vector.

The following are valid options for `ptShaderOptioni`:

`PT_INTENSITY_SHADER_RAMP`	The index of the intensity colour ramp that is used to shade intensity values
`PT_PLANE_SHADER_RAMP`	The index of the plane shader ramp that is used to shade points by distance from a plane
`PT_PLANE_SHADER_EDGE`	The behavior at the edge of the planar shading range (distance)

value The shader option's new value.

Values for PT_PLANE_SHADER_EDGE can be one of the following:

`PT_EDGE_REPEAT`	The shading is repeated (modulated)
`PT_EDGE_CLAMP`	The colour at the edge is clamped
`PT_EDGE_BLACK`	Anything outsied the range is black
`PT_EDGE_MIRROR`	The shade at the edge is mirrored. Note that this is only supported on OpenGL 2.0 when using Vortex's hardware rendering via ptDrawGL

RETURN VALUE

A value indicated the result of the function, possible values are:

`PTV_INVALID_OPTION`	The option parameter is not a valid value
`PTV_SUCCESS`	The function completed successfully

GetShaderOption

DEFINITION

```
PTres ptGetShaderOptionf( PTenum option, PTfloat *value )
PTres ptGetShaderOptionfv( PTenum option, PTfloat *value )
PTres ptGetShaderOptioni( PTenum option, PTint *value )
```

DESCRIPTION

Gets a shader option for the active viewport

PARAMETERS

option	The shader option whose value is being retrieved. Valid values are the same as for the option parameter of ptShaderOption
value	A pointer to a buffer to take the option's values

RETURN VALUE

A value indicated the result of the function, possible values are:

`PTV_INVALID_OPTION`	The option parameter is not a valid value
`PTV_VOID_POINTER`	The pointer to the value buffer is null
`PTV_SUCCESS`	The function completed successfully

ResetShaderOptions

DEFINITION
```
PTvoid ptResetShaderOptions
```

DESCRIPTION
Resets shading options in the active viewport to their default values

---

CopyShaderSettings

DEFINITION
```
PTvoid ptCopyShaderSettings( PTuint dest_viewport )
```

DESCRIPTION
Copys shader settings from the current viewport to the destination viewport

PARAMETERS

dest_viewport	The index of the destination viewport to copy the settings to

---

CopyShaderSettingsToAll

DEFINITION
```
PTvoid ptCopyShaderSettingsToAll
```

DESCRIPTION
Copys shader settings from the current viewport to all other viewports

---

NumRamps

DEFINITION
```
PTint ptNumRamps
```

DESCRIPTION
Gets the overall number of shading ramps loaded by the engine.

RETURN VALUE
The number of shading ramps

RampInfo

DEFINITION
```
const PTstr ptRampInfo( PTint ramp, PTenum *type )
```

DESCRIPTION
Gets information about a particular shading ramp referenced by its index

PARAMETERS

ramp Gets The index of the ramp where *ramp* is less that the number of ramps

type A pointer to a PTenum to receive the ramp type information.
The type will be a value that is a combination of the following values:

`PT_INTENSITY_RAMP_TYPE`	A ramp used to shade intensity
`PT_PLANE_RAMP_TYPE`	A ramp used by the planar distance shader

Note that the constants above may be or'ed together for ramps that are intended for both purposes.

RETURN VALUE
The name of the ramp as a string

AddCustomRamp

DEFINITION
```
PTres ptAddCustomRamp(const PTstr name, PTint numKeys, const PTfloat
                      *positions, const PTubyte* colour3vals,
                      PTbool interpolateInHSL )
```

DESCRIPTION
Adds a user colour ramp to the available ramps in the Vortex engine. If successfully added this ramp will be then be available for use with the shader options that use colour ramps. The colour ramp is described as an array of position / colour value keys which are interpolated to generate the colour ramp.

PARAMETERS
name A unique name for the new colour ramp

numKeys The number of colour key(position and colour value) pairs in the arrays.

positions An array of single float values of size *numKeys* in ascending order of the position of the keys along the ramp. These values will be internally normalized to 0 to 1 so the input values can use any range of values.

colour3vals An array of triple unsigned byte values representing the key's RGB values. The size of the array will therefore be 3 x *numKeys* values.

interpolateInHSL If set true the key colour values will be interpolated in Hue, Saturation, Lightness (HSL) space rather than Red, Green Blue (RGB)

RETURN VALUE
PT_SUCCESS if the function succeeds.

Lighting

LightOption

DEFINITION

```
PTres ptLightOptionf( PTenum Light_option, PTfloat value )
```

DESCRIPTION

Sets a lighting option for the current viewport

PARAMETERS

option The lighting option to change. Valid values are

PT_LIGHT_STRENGTH	Sets the strength of the lighting. Valid values (for value parameter) are between 0 and 2.0

value The new desired value for the lighting option

RETURN VALUE

A value indicated the result of the function, possible values are:

PTV_INVALID_OPTION	The option parameter is not a valid value
PTV_VALUE_OUT_OF_RANGE	The value parameter is outside the acceptable range
PTV_SUCCESS	The function completed successfully

GetLightOption

DEFINITION

```
PTres ptGetLightOptionf( PTenum Light_option, PTfloat *value )
```

DESCRIPTION

Gets a lighting options current value for the current viewport

PARAMETERS

option The option whose value is being requested, valid options are:

PT_LIGHT_VECTOR	The direction from which the light is coming from. A vector represented by 3 float values will be returned
PT_LIGHT_COLOUR	The colour of the light. An RGB value represented by 3 float values will be returned
PT_LIGHT_AMBIENT_COLOUR	The ambient colour of the light. An RGB value represented by 3 float values will be returned
PT_LIGHT_DIFFUSE_COLOUR	The diffuse colour of the light. A RGB value represented by 3 float values will be returned
PT_LIGHT_SPECULAR_COLOUR	The specular colour of the light. A RGB value represented by 3 float values will be returned

PT_LIGHT_ANGLE	The direction of the light source expressed as XYZ euler angles. XYZ rotation values represented by 3 float values will be returned
PT_LIGHT_STRENGTH	The overall strength of the lighting. A single float value is returned

value A float buffer to receive the value at least the size of the returned values

RETURN VALUE
A value indicated the result of the function, possible values are:

PTV_INVALID_OPTION	The option parameter is not a valid value
PTV_NULL_POINTER	The value parameter is null
PTV_SUCCESS	The function completed successfully

CopyLightSettings

DEFINITION
```
PTvoid ptCopyLightSettings( PTuint dest_viewport )
```

DESCRIPTION
Copys lighting settings from the current viewport to the destination viewport

PARAMETERS
dest_viewport The index of the destination viewport to copy the settings to

---

CopyLightSettingsToAll

DEFINITION
```
PTvoid ptCopyLightSettingsToAll
```

DESCRIPTION
Copys lighting settings from the current viewport to all other viewports

---

ResetLightOptions

DEFINITION
```
PTvoid ptResetLightOptions
```

DESCRIPTION
Resets the lighting setup options to their default state

View Parameters

ReadViewFromGL

DEFINITION

```
PTbool ptReadViewFromGL
```

DESCRIPTION

Reads the entire view setup (projection, eye transform and viewport size) from the current OpenGL context

RETURN VALUE

Boolean value indicating success. Returns GL_FALSE if no OpenGL context is current in this thread

---

SetViewProjectionOrtho

DEFINITION

```
PTvoid ptSetViewProjectionOrtho( PTdouble l, PTdouble r, PTdouble b,
                                 PTdouble t, PTdouble n, PTdouble f )
```

DESCRIPTION

Sets the view projection to an parallel view identical to glOrtho

PARAMETERS

l ,r	Specifies the coordinate for the left and right clipping planes
t, b	Specifies the coordinate for the top and bottom clipping planes
n,f	Specifies the coordinate for the near and far clipping planes

---

SetViewProjectionOrtho

DEFINITION

```
PTvoid ptSetViewProjectionOrtho( PTdouble l, PTdouble r, PTdouble b,
                                 PTdouble t, PTdouble n, PTdouble f )
```

DESCRIPTION

Sets the view projection to an parallel view identical to glOrtho

PARAMETERS

l ,r	Specifies the coordinate for the left and right clipping planes
t, b	Specifies the coordinate for the top and bottom clipping planes
n,f	Specifies the coordinate for the near and far clipping planes

---

SetViewProjectionFrustum

DEFINITION

```
PTvoid ptSetProjectionFrustum( PTdouble l, PTdouble r, PTdouble b,
                               PTdouble t, PTdouble n, PTdouble f )
```

DESCRIPTION
Sets the view projection to an perspective frustum

PARAMETERS

l ,r	Specifies the coordinate for the left and right clipping planes
t, b	Specifies the coordinate for the top and bottom clipping planes
n,f	Specifies the coordinate for the near and far clipping planes

---

SetViewProjectionPerspective

DEFINITION

```
PTvoid ptSetProjectionFrustum( PTenum type, PTdouble fov, PTdouble
                               aspect, PTdouble n, PTdouble f )
```

DESCRIPTION
Sets the view projection to an perspective view

PARAMETERS

type Specifies the method used to calculate the projection matrix. One of the following values can be used:

PT_PROJ_PERSPECTIVE_GL	Use the same method as OpenGL
PT_PROJ_PERSPECTIVE_DX	Use the same method as Direct3D
PT_PROJ_PERSPECTIVE_BLINN	Use the Blinn method

fov	Specifies the field-of-view in degrees
aspect	Specifies the aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (height) and can most often be computed using the viewport dimensions.
n	The distance from the viewer to the near clipping plane
f	The distance from the viewer to the far clipping plane

SetViewProjectionMatrix

DEFINITION

```
PTvoid ptSetProjectionMatrix( const PTdouble *matrix, bool row_major )
```

DESCRIPTION
Sets the view projection matrix by directly providing an array of values.

PARAMETERS

matrix	An array of the 16 double values of the 4 x 4 projection matrix
row_major	If true interpret the matrix values as being row ordered (like Direct3D) otherwise values are interpreted as column ordered (like OpenGL).

GetViewProjectionMatrix

DEFINITION

```
PTvoid ptGetViewProjectionMatrix( const PTdouble *matrix )
```

DESCRIPTION

Gets the view's eye space transformation matrix by into an array of values. The matrix values are returned in column major order

PARAMETERS

matrix	An array of the 16 doubles to receive the values of the 4 x 4 projection matrix

SetViewEyeLookAt

DEFINITION

```
PTvoid ptSetViewEyeLookAt(const PTdouble *eye, const PTdouble *target,
                          const PTdouble *up );
```

DESCRIPTION

Sets the eye transformation (modelview in OpenGL) derived from an eye point, a view target and an up vector. The matrix maps the target point to the negative z axis and the eye to the origin, so that when a typical projection matrix is sued the target maps to the centre of the viewport. The matrix produced is identical to gluLookAt.

PARAMETERS

eye	Specifies the location of the viewer as an array of 3 doubles
target	Specifies the point being looked at as an array of 3 doubles
up	Specifies the up vector as an array of three doubles

SetViewEyeMatrix

DEFINITION

```
PTvoid ptSetViewEyeMatrix( const PTdouble *matrix, bool row_major )
```

DESCRIPTION

Sets the view's eye space transformation matrix by directly providing an array of values.

PARAMETERS

matrix	An array of the 16 double values of the 4 x 4 projection matrix
row_major	If true interpret the matrix values as being row ordered (like Direct3D) otherwise values are interpreted as column ordered (like OpenGL).

GetViewEyeMatrix

DEFINITION

```
PTvoid ptGetViewEyeMatrix ( const PTdouble *matrix )
```

DESCRIPTION

Gets the view's eye space transformation matrix by into an array of values. The matrix values are returned in column major order

PARAMETERS

matrix	An array of the 16 doubles to receive the values of the 4 x 4 projection matrix

---

SetViewportSize

DEFINITION

```
PTvoid ptSetViewportSize( PTint left, PTint bottom, PTuint width, PTuint
                          height )
```

DESCRIPTION

Sets the size and position of the viewport used to map normalized device coordinates to the window coordinates

PARAMETERS

left, bottom	Specify the left and bottom coordinates of the viewport, these values can be negative.
Width, height	Specify the size of the viewport in pixels

Editing

SetSelectPointsMode

DEFINITION

```
PTvoid ptSetSelectPointsMode( PTenum select_mode )
```

DESCRIPTION

Sets the mode for subsequent selection operations

PARAMETERS

select_mode The selection mode for subsequent operations. This can be one of the following:

`PT_EDIT_MODE_SELECT`	Points are selected in the selection region
`PT_EDIT_MODE_UNSELECT`	Points are deselected in the selection region
`PT_EDIT_MODE_UNHIDE`	Points are unhidden in the selection region

GetSelectPointsMode

DEFINITION

```
PTenum ptGetSelectPointsMode
```

DESCRIPTION

Gets the mode for subsequent selection operations

RETURN VALUE

The current selection mode as one of following values:

`PT_EDIT_MODE_SELECT`	Points are selected in the selection region
`PT_EDIT_MODE_UNSELECT`	Points are deselected in the selection region
`PT_EDIT_MODE_UNHIDE`	Points are unhidden in the selection region

SetSelectionDrawColor

DEFINITION

```
PTvoid ptSetSelectionDrawColor( const PTubyte *col3 );
```

DESCRIPTION

Sets the color which is used for drawing selected points. This will affect both the Vortex hardware rendering and the RGB values returned by point queries when the RGB mode is set to shader.

PARAMETERS

col3 An array of 3 bytes specifying Red, Green and Blue components of the color

---

GetSelectionDrawColor

DEFINITION

```
PTvoid ptGetSelectionDrawColor(PTubyte *col3 );
```

DESCRIPTION

Returns the color which is used for drawing selected points.

PARAMETERS

col3	An array of 3 bytes to receive the Red, Green and Blue components of the color

---

SelectPointsByRect

DEFINITION

```
PTvoid ptSelectPointsByRect( PTint xpos, PTint ypos,
                             PTint width, PTint height )
```

DESCRIPTION

Selects points with a screen space rectangle. Note that in Vortex screen space Y coordinates increase upwards (OpenGL convention). Use `ptFlipMouseYCoords` if a win32 / MFC mouse position is being used.

PARAMETERS

x_edge	The position of the left edge of the rectangle
y_edge	The position of the bottom edge of the rectangle
width	The width of the rectangle
height	The height of the rectangle

SelectPointsByFence

DEFINITION

```
PTres ptSelectPointsByFence(PTint num_vertices, const PTint *vertices)
```

DESCRIPTION

Selects points with a screen space polygonal fence. Note that in Vortex screen space Y coordinates increase upwards (OpenGL convention). Use `ptFlipMouseYCoords` if a win32 / MFC mouse position is being used.

PARAMETERS

num_vertices	The number of vertices in the polygon, must be at least 3
vertices	An array of vertices in screen space as X, Y integer pairs

RETURN VALUE

A value indicated the result of the function, possible values are:

`PTV_INVALID_PARAMETER`	The num of vertices parameter is invalid
`PTV_VOID_POINTER`	The vertices parameter is null
`PTV_SUCCESS`	The function completed successfully

SelectPointsByCube

DEFINITION

```
PTres ptSelectPointsByCube(const PTfloat* center, PTfloat radius)
```

DESCRIPTION

Selects points within an axis aligned cube

PARAMETERS

center	The center of the cube expressed as an array of 3 float values representing a point
radius	The radius of the cube, ie the half edge length

RETURN VALUE

A value indicated the result of the function, possible values are:

PTV_INVALID_PARAMETER	The radius is zero or less than zero
PTV_SUCCESS	The function completed successfully

SelectPointsByPlane

DEFINITION

```
PTres ptSelectPointsByPlane( const PTfloat *origin, const PTfloat *normal,
                             PTfloat thickness )
```

DESCRIPTION

Selects points that lay within a distance of a plane

PARAMETERS

origin	The origin of the plane, this can be any point the plane passes through. The value is expressed as an array of 3 float values
normal	The plane's normal. This should be a normalized value. The value is expressed as an array of 3 float values
thickness	The thickness to select points within

RETURN VALUE

A value indicating the result of the function, possible values are:

PTV_VOID_POINTER	The origin or normal values are null
PTV_SUCCESS	The function completed successfully

SelectPointsByBox

DEFINITION

```
PTres ptSelectPointsByBox( const PTfloat *lower, const PTfloat *upper )
```

DESCRIPTION

Selects points within an axis aligned box.

PARAMETERS

lower	The lower extents of the box expressed as an array of 3 floats representing the lower box corner
upper	The upper extents of the box expressed as an array of 3 floats representing the upper box corner

RETURN VALUE

A value indicated the result of the function, possible values are:

PTV_VOID_POINTER	The lower or upper values are null
PTV_SUCCESS	The function completed successfully

SelectPointsByOrientedBox

DEFINITION

```
PTres ptSelectPointsByOrientedBox( const PTfloat *lower, const PTfloat *upper,
const
                                   PTfloat* pos, PTfloat* uAxis, PTfloat* vAxis)
```

DESCRIPTION

Selects points using a positioned and oriented bounding box. The box's dimensions are specified in its own local coordinate system. The coordinate system is defined by the position and the U and V axes, which are the X and Y axes of the local coordinate system. The W (Z) axis of the local coordinate system is calculated automatically. U and V must be orthogonal, but not normalized.

PARAMETERS

lower	The lower extents of the box in its own coordinate system as an array of 3 floats
upper	The upper extents of the box in its own coordinate system as an array of 3 floats
pos	The position of the box as an array of 3 float representing the translation vector
uAxis	The U axis represented by an array of 3 floats
vAxis	The V axis represented by an array of 3 floats

RETURN VALUE

A value indicated the result of the function, possible values are:

PTV_VOID_POINTER	One of the parameters is null
PTV_SUCCESS	The function completed successfully

SelectPointsBySphere

DEFINITION

```
PTres ptSelectPointsBySphere( const PTfloat *centre, PTfloat radius )
```

DESCRIPTION

Selects points within a sphere

PARAMETERS

center	The center of the sphere expressed as an array of 3 float values representing a point
radius	The radius of the sphere

RETURN VALUE

A value indicated the result of the function, possible values are:

`PTV_INVALID_PARAMETER`	The radius is zero or less than zero
`PTV_SUCCESS`	The function completed successfully

InvertSelection

DEFINITION
`PTvoid` **`ptInvertSelection`**

DESCRIPTION
Inverts the selection of points so that points that were previously selected are deselected and vise-versa

---

IsolateSelected

DEFINITION
`PTvoid` **`ptIsolateSelected`**

DESCRIPTION
Isolates the selected points with the active layers. Points which are not selected will be hidden. This is the same as a InvertSelection followed by a HideSelected but is more efficient.

UnselectAll

DEFINITION
`PTvoid` **`ptUnselectAll`**

DESCRIPTION
Unselects all points

HideSelected

DEFINITION
`PTvoid` **`ptHideSelected`**

DESCRIPTION
Hides selected points from view

UnhideAll

DEFINITION
`PTvoid` **`ptUnhideAll`**

DESCRIPTION
Shows all points resetting previous hide commands

InvertVisibility

DEFINITION

```
PTvoid ptInvertVisibility
```

DESCRIPTION

Inverts the visibility of points so that points that were previously visible are hidden and vise-versa

---

SetSelectionScope

DEFINITION

```
PTvoid ptSetSelectionScope( PThandle sceneOrCloudHandle )
```

DESCRIPTION

Sets the selection scope restricting subsequent selection operations to a particular point cloud or scene. The selection scope can be reset to global by providing a zero handle as the sceneOrCloudHandle parameter

PARAMETERS

sceneOrCloudHandle	A handle to a scene or point cloud. Use zero to reset the scope to global

---

RefreshEdit

DEFINITION

```
PTvoid ptRefreshEdit
```

DESCRIPTION

Re-applies editing operations in the current edit stack ie all operations since the last clear command. This may be necessary if subsequent to selection more points have been loaded into memory.

ClearEdit

DEFINITION
```
PTvoid ptClearEdit
```

DESCRIPTION
Clears the editing stack, this has the affect of deselecting and unhiding all points and returning the editing system to its startup state

---

StoreEdit

DEFINITION
```
PTvoid ptStoreEdit( const PTstr name )
```

DESCRIPTION
Stores the current edit stack for application later

PARAMETERS
name A string to identify the stored edit stack

RestoreEdit

DEFINITION
```
PTbool ptRestoreEdit( const PTstr name )
```

DESCRIPTION
Restores the current edit stack for identified by its name

PARAMETERS
name A string that identifies the previously stored edit stack

RETURN VALUE
PT_TRUE if successful. PT_FALSE is returned if the edit stack referred to cannot be found

---

RestoreEditByIndex

DEFINITION
```
PTbool ptRestoreEditByIndex( PTint index )
```

DESCRIPTION
Restores the current edit stack for identified by its index

PARAMETERS
index A index that identifies the previously stored edit stack

RETURN VALUE
PT_TRUE if successful. PT_FALSE is returned if the edit stack referred to cannot be found

---

DeleteEdit

DEFINITION
```
PTbool ptDeleteEdit( const PTstr name )
```

DESCRIPTION
Deletes the current edit stack for identified by its name

PARAMETERS
name A string that identifies the previously stored edit stack

RETURN VALUE
PT_TRUE if successful. PT_FALSE is returned if the edit stack referred to cannot be found

---

DeleteEditByIndex

DEFINITION
```
PTbool ptDeleteEditByIndex( PTint index )
```

DESCRIPTION
Deletes the current edit stack for identified by its index

PARAMETERS
index A index that identifies the previously stored edit stack

RETURN VALUE
PT_TRUE if successful. PT_FALSE is returned if the edit stack referred to cannot be found

---

DeleteAllEdits

DEFINITION
```
PTvoid ptDeleteAllEdits
```

DESCRIPTION
Deletes all edit stacks stored in the Vortex engine

NumEdits

DEFINITION
```
PTint ptNumEdits
```

DESCRIPTION
Gets the number of edit stacks stored in the Vortex engine

RETURN VALUE
The number of edit stacks stored in the Vortex engine

EditName

DEFINITION
```
const PTstr ptEditName( PTint index )
```

DESCRIPTION
Gets the name of the edit stack referenced by its index

PARAMETERS

index The indexed position of the edit stack. This will be >0 and <ptNumEdits

RETURN VALUE

The name of the edit stack as a C string

GetEditData

DEFINITION

```
PTint ptGetEditData ( PTint index, PTubyte *data )
```

DESCRIPTION

Provides a binary chunk of data that represents the entire edit stack. This can be used for persistence between sessions of the client application.

PARAMETERS

index The indexed position of the edit stack. This will be >0 and < ptNumEdits

data A pointer to a buffer to receive the binary. The size of this buffer should be at least the number of bytes indicated by ptGetEditDataSize

RETURN VALUE

The number of bytes written to data

GetEditDataSize

DEFINITION

```
PTint ptGetEditDataSize ( PTint index )
```

DESCRIPTION

Calculates the number of bytes required to store the binary chunk of data that represents the entire edit stack.

PARAMETERS

index The indexed position of the edit stack. This will be >0 and <ptNumEdits

RETURN VALUE

The number of bytes required to store the edit stack

CreateEditFromData

DEFINITION

```
PTvoid ptCreateEditFromData ( const PTubyte *data )
```

DESCRIPTION

Creates an edit stack from a binary chunk previously written by ptGetEditdata. This is most often used to restore the named edit stacks between sessions. Usually this data is being read from a project file.

PARAMETERS

data The binary data representing the stored edit stack

Layers

SetCurrentLayer

DEFINITION

```
PTbool ptSetCurrentLayer( PTuint layer )
```

DESCRIPTION

Sets the current layer. The current layer is the target layer for Copy or Move operations. The current layer is always visible and cannot be locked.

PARAMETERS

layer	Layer index from 0 to PT_MAX_LAYERS

RETURN VALUE

Boolean indicating success. PT_FALSE is returned if the layer is locked or out of bounds

---

GetCurrentLayer

DEFINITION

```
PTuint ptGetCurrentLayer
```

DESCRIPTION

Retrieves the current projection matrix used by Vortex for visibility determination

RETURN VALUE

The current layer index

LockLayer

DEFINITION

```
PTbool ptLockLayer( PTuint layer, PTbool lock )
```

DESCRIPTION

Locks or unlocks a layer. Locking a layer prevents point selection in that layer. Layer locking is independent of layer visibility and a locked layer maybe shown or hidden.

PARAMETERS

layer	Layer to lock or unlock as an index from 0 to PT_MAX_LAYERS
lock	Boolean indicating desired lock status

RETURN VALUE

Boolean indicating success. PT_FALSE is returned if the layer is current or out of bounds

---

IsLayerLocked

DEFINITION

```
PTbool ptIsLayerLocked( PTuint layer )
```

DESCRIPTION

Retrieves the locked status of a layer

PARAMETERS

layer Layer to be queried as an index from 0 to PT_MAX_LAYERS

RETURN VALUE

Boolean indicating locked status of layer.

ShowLayer

DEFINITION

```
PTbool ptShowLayer( PTuint layer, PTbool show )
```

DESCRIPTION

Sets the visible property of a layer causing the layer to be shown or hidden

PARAMETERS

layer Layer to show or hide as an index from 0 to PT_MAX_LAYERS

show Boolean indicating desired visibility status

RETURN VALUE

Boolean indicating success. Attempting to hide the current layer returns PT_FALSE

IsLayerShown

DEFINITION

```
PTbool ptIsLayerShown( PTuint layer )
```

DESCRIPTION

Retrieves the visible status of a layer

PARAMETERS

layer Layer to be queried as an index from 0 to PT_MAX_LAYERS

RETURN VALUE

Boolean indicating success. There is no OpenGL context current if PT_FALSE is returned

---

DoesLayerHavePoints

DEFINITION

```
PTbool ptDoesLayerHavePoints( PTuint layer )
```

DESCRIPTION

Returns the points occupancy status of a layer. This can be used to provide a visual cues to the empty / occupied status of a layer. Note that the method returns a pre-computed state and therefore does not incur significant processing overhead

PARAMETERS

layer	Layer to be queried as an index from 0 to PT_MAX_LAYERS

RETURN VALUE

Boolean indicating occupancy status ie. PT_TRUE is returned if there are points in the layer and PT_FALSE if there are not.

ClearPointsFromLayer

DEFINITION

```
PTvoid ptClearPointsFromLayer( PTuint layer )
```

DESCRIPTION

Removes points from the specified layer

PARAMETERS

layer	The layer to remove points from as an index from 0 to PT_MAX_LAYERS

ResetLayers

DEFINITION

```
PTvoid ptResetLayers
```

DESCRIPTION

Clears all layers and places all points into layer 0 ie sets the layer status to the startup state

---

CopySelToCurrentLayer

DEFINITION

```
PTbool ptCopySelToCurrentLayer( PTbool deselect )
```

DESCRIPTION

Copies selected points to the current layer. Points can exist in more than one layer. There is no duplication in this case so that selecting a point in one layer causes it to be selected in all layers.

PARAMETERS

deselect Deselect the points after the copy. Usually this is the desired behavior

RETURN VALUE

Boolean indicating success.

MoveSelToCurrentLayer

DEFINITION

```
PTbool ptMoveSelToCurrentLayer( PTbool deselect )
```

DESCRIPTION

Moves selected points out of all active (ie unlocked) layers to the current layer.

PARAMETERS

deselect Deselect the points after the copy. Usually this is the desired behavior

RETURN VALUE

Boolean indicating success.

Drawing

OverrideDrawMode

DEFINITION

```
PTvoid ptOverrideDrawMode( PTenum mode )
```

DESCRIPTION

Overrides the draw mode forcing drawing into either dynamic or static mode. This is useful when a tool requires a particular draw mode but it is not possible to check this at draw time.

PARAMETERS

mode A constant representing the draw mode. The following values are can be used:

`PT_DRAW_MODE_STATIC`	Draw in static mode. In this mode a full view is rendered, for large volumes of data this may take up to 2 or 3 seconds.
`PT_DRAW_MODE_DYNAMIC`	Draw in dynamic mode. The drawing is optimized and attempts to return without the time frame determined by the frame rate.
`PT_DRAW_MODE_DEFAULT`	Resets the draw mode to the default mode, it is important to do this after the override is no longer required

---

DrawGL

DEFINITION

```
PTvoid ptDrawGL
```

DESCRIPTION

Draws all visible objects to the active OpenGL context. This will use the active context's modelview and projection matrices. It does not affect the pre-call GL state with the exception of initializing resources on the first call in a new context. The view setup is extracted from OpenGL and used to determine visible areas and prioritize point data loading.

Support for OpenGL 1.4 or later is required to use all the features of the display engine. In most cases where sufficient support is not available the display will gracefully drop unsupported features.

Client code that requires non-GL drawing should still set up an OpenGL context for the purpose of setting up a viewing frustum and viewport that Vortex can use. This could be done with an off-screen context so that the client application is not affected. Drawing could then be performed via a visible points query with the query density set to viewing frustum. See the query section of the API documentation.

The call is not asynchronous and returns after drawing has completed.

DrawSceneGL

DEFINITION

```
PTvoid ptDrawSceneGL( PThandle scene, PTbool dynamic )
```

DESCRIPTION

Draws a point cloud scene into the active OpenGL context. See DrawGL for more information on OpenGL state and requirements.

PARAMETERS

mode A constant representing the draw mode. The following values are can be used:

PT_DRAW_MODE_STATIC	Draw in static mode. In this mode a full view is rendered, for large volumes of data this may take up to 2 or 3 seconds.
PT_DRAW_MODE_DYNAMIC	Draw in dynamic mode. The drawing is optimized and attempts to return without the time frame determined by the frame rate.

WeightedPtsLoaded

DEFINITION

```
PTuint ptWeightedPtsLoaded( PTbool reset )
```

DESCRIPTION

Returns the number of points loaded since the last resetting call of the function. The number of points is a weighted value that takes into account the visual significance of the points loaded. This value can be used to determine if the view requires a redraw to show the recently loaded point data.

PARAMETERS

reset Reset the counter to zero

RETURN VALUE

The weighted number of points loaded since the last reset call

---

PtsLoadedInViewportSinceDraw

DEFINITION

```
PTint64 ptPtsLoadedInViewportSinceDraw( PThandle forScene )
```

DESCRIPTION

Returns the number of a scene's points loaded for the current viewport since the last ptDrawGL call. This can be used to determine if a viewport redraw is required to show the recently loaded points.

PARAMETERS

forScene Specifies a particular scene for which the metric is required. Using a zero value returns results for all scenes that are loaded

RETURN VALUE
The number of points loaded in the viewport since the last ptDrawGL call

---

PtsToLoadInViewport

DEFINITION
```
PTint64 ptPtsToLoadInViewport( PThandle forScene, PTbool reCompute )
```

DESCRIPTION
Returns the number of points that are pending for loading to complete the viewport in the current viewport and for a particular scene

PARAMETERS
forScene Specifies a particular scene for which the metric is required. Using a zero value returns results for all scenes that are loaded

reCompute Recompute this value, this will give an exact number at the expense of some CPU overhead.

RETURN VALUE
The number of points that are pending for loading

StartDrawFrameMetrics

DEFINITION
```
PTvoid ptStartDrawFrameMetrics
```

DESCRIPTION
This function should be called before starting to draw a frame to enable the Vortex API to correctly compute the data available to draw since the last frame. If a single ptDrawGL command is used to draw the frame it is not necessary to call this function.

EndDrawFrameMetrics

DEFINITION
```
PTvoid ptEndDrawFrameMetrics
```

DESCRIPTION
This function should be called after drawing of the frame is complete to enable the Vortex API to correctly compute the data available to draw since the last frame.

Units

SetHostUnits

DEFINITION

```
PTvoid ptSetHostUnits( PTenum units )
```

DESCRIPTION

Sets the units used in the client environment. Subsequent draw and query operations will use the units specified

PARAMETERS

units The units to be used. This can be one of the following values:

PT_METERS	Metric meters (default)
PT_DECIMETERS	Decimeters
PT_CENTIMETERS	Centimeters
PT_MILLIMETERS	Millimeters
PT_FEET	Feet
PT_FEET_US	US Survey feet
PT_INCHES	Inches

GetHostUnits

DEFINITION

```
PTenum ptGetHostUnits( void )
```

DESCRIPTION

Return the current units setting set previously with ptSetHostUnits or the default PT_METERS

RETURN VALUE

The units used, possible values are:

PT_METERS	Metric meters (default)
PT_DECIMETERS	Decimeters
PT_CENTIMETERS	Centimeters
PT_MILLIMETERS	Millimeters
PT_FEET	Feet
PT_FEET_US	US Survey feet
PT_INCHES	Inches

Viewports

Viewports are used to track shading settings and camera setup. Correct Viewport and view definition is important for applications that render the point cloud data since the current view settings are used to identify point visibility and drive background loading of point data.

If OpenGL is being used for rendering, each viewport is assumed to have its own GL context

AddViewport

DEFINITION

```
PTint ptAddViewport( PTint index, const PTstr name, PTenum viewportContext )
```

DESCRIPTION

Adds a viewport to the Vortex Engine. This does not create a hardware viewport context but will attach to an existing context as specified by the viewportContext parameter.

PARAMETERS

index	The desired index of the viewport
name	The name of the viewport
viewportContext	The type of viewport being added. Valid values include:

`PT_GL_VIEWPORT`	An OpenGL based viewport
`PT_DX_VIEWPORT`	A DirectX based viewport (not implemented as of 1.5.0.6)
`PT_SW_VIEWPORT`	A viewport that does not attach to a hardware context

RETURN VALUE

The actual allocated index of the viewport

RemoveViewport

DEFINITION

```
PTvoid ptRemoveViewport( PTint index )
```

DESCRIPTION

Removes a viewport from the Vortex engine freeing resources associated with its storage

PARAMETERS

index The index of the viewport to free

SetViewport

DEFINITION

```
PTvoid ptSetViewport( PTint index )
```

DESCRIPTION

Sets the current viewport. Many functions use the current viewport setting that is set with this function. For example subsequent shader settings will only affect the current viewport.

PARAMETERS

index The index of the viewport to set as current

SetViewportByName

DEFINITION

```
PTint ptSetViewportByName( const PTstr name )
```

DESCRIPTION

Set the current viewport to the one identified by the name parameter

PARAMETERS

name The name of the viewport. Note that for many applications names such as Top, Left, Perspective etc may not be appropriate if multiple viewports can have the same viewing direction.

RETURN VALUE

The index of the viewport set as current

CurrentViewport

DEFINITION

```
PTint ptCurrentViewport
```

DESCRIPTION

Returns the current viewport by its index

RETURN VALUE

The current viewport index

EnableViewport

DEFINITION

```
PTvoid ptEnableViewport( PTint index )
```

DESCRIPTION

Sets a viewport's state to enabled so that ptDrawGL will draw in that viewport and visibility computation and data loading will be active for that viewport. This is the default state

PARAMETERS

index The index of the viewport to be enabled.

DisableViewport

DEFINITION

```
PTvoid ptDisableViewport( PTint index )
```

DESCRIPTION

Sets a viewport's state to disabled so that ptDrawGL will not draw in that viewport and visibility computation and data loading will be inactive for that viewport.

PARAMETERS

index The index of the viewport to be disabled.

IsViewportEnabled

DEFINITION

```
PTbool ptIsViewportEnabled( PTint index )
```

DESCRIPTION

Return the enabled state for a viewport

PARAMETERS

index The index of the viewport

RETURN VALUE

A Boolean indicating the enabled state of the viewport

IsCurrentViewportEnabled

DEFINITION

```
PTbool ptIsCurrentViewportEnabled
```

DESCRIPTION

Returns the enabled state for the current viewport

RETURN VALUE

A Boolean representing the enabled state of the current viewport

---

ViewportIndexFromName

DEFINITION

```
PTint ptViewportIndexFromName ( const PTstr name )
```

DESCRIPTION

Return the index of a viewport specified by its name.

PARAMETERS

name The name of the viewport

RETURN VALUE
The index of the viewport. If the returned integer is negative the viewport was not found.

CreateBitmapViewport

DEFINITION
```
PTvoid* ptCreateBitmapViewport(int w, int h, const PTstr name)
```

DESCRIPTION
Creates a off-screen OpenGL viewport for rendering to an image. This is demonstrated in the MakeBitmap example project. The viewport can be made current by name, ie by using ptSetViewportByName

PARAMETERS
w The width of the bitmap

h The height of the bitmap

name The unique name of the viewport

RETURN VALUE
A HBITMAP value cast to void. If the value is not null it is safe to cast the void pointer back to HBITMAP to gain access to the bitmap data.

DestroyBitmapViewport

DEFINITION

```
PTvoid ptDestroyBitmapViewport(const PTstr name)
```

DESCRIPTION
Frees resources used by the bitmap viewport specified by its name. The HBITMAP associated with this viewport will no longer be valid after this is called.

PARAMETERS
name The name of the bitmap viewport

Engine Tuning

DynamicFrameRate

DEFINITION
```
PTvoid ptDynamicFrameRate( PTfloat fps )
```

DESCRIPTION
Sets the frame rate in frames per second that is maintained during dynamic draw mode. Dynamic draw mode is usually active during user navigation operations where less points are drawn to ensure smooth navigation.

Lowering the frame rate helps to retain detail whilst navigating but may cause the navigation to become jerky. Increasing the frame rate results in a smooth navigation but may reduce the view density.

PARAMETERS
fps The frame rate in frames per second. Valid values are from 1 to 30, the default value is 15

GetDynamicFrameRate

DEFINITION
```
PTfloat ptGetDynamicFrameRate
```

DESCRIPTION
Gets the current display frame rate

RETURN VALUE
The frame rate in frames per second

StaticOptimizer

DEFINITION
```
PTvoid ptStaticOptimizer( PTfloat opt )
```

DESCRIPTION
Sets the static optimizer strength. This is the optimizer that optimizes the static drawing. Lowering this value is sometimes necessary if the optimizer is too aggressive and causes some areas to become faint in density.

PARAMETERS
opt The optimizer strength. Valid values are from 0 to 1, default 0.5

GetStaticOptimizer

DEFINITION
```
PTfloat ptGetStaticOptimizer
```

DESCRIPTION
Gets the current static optimizer value

RETURN VALUE
The optimizer strength expressed as a float between 0 and 1.

GlobalDensity

DEFINITION
```
PTvoid ptGlobalDensity( PTfloat density )
```

DESCRIPTION
Sets the global display density. This is used to fade the display of point clouds and can help to compare solid or line geometry against the point cloud which would otherwise be hidden by the dense point cloud.

PARAMETERS
density	The density value. Valid values are from 0 to 1, default 1.0

GetGlobalDensity

DEFINITION
```
PTfloat ptGetGlobalDensity
```

DESCRIPTION
Gets the current global density value

RETURN VALUE
The density expressed as a float between 0 and 1.

SetLoadingPriorityBias

DEFINITION
```
PTres ptSetLoadingPriorityBias ( PTenum bias )
```

DESCRIPTION
Sets the priority bias for background loading of view based point data. This can significantly impact perceived performance. For interior environments for example, it can be more effective to prioritise loading data furthest from the viewer – for exterior the opposite is true. The default method uses a combination of distance from the viewer and centrality to the view to determine loading priority.

PARAMETERS
bias	The engine tuning option to be set. This can be one of the following:

`PT_LOADING_BIAS_SCREEN`	A screen based metric is used to control loading bias. This is the default method
`PT_LOADING_BIAS_NEAR`	Points nearer to the viewer are loaded first
`PT_LOADING_BIAS_FAR`	Points further from the viewer are loaded first
`PT_LOADING_BIAS_POINT`	Point loading priority is based on distance from a point that can be specified with ptSetTuningParameterfv

RETURN VALUE
The density expressed as a float between 0 and 1.

PTV_INVALID_OPTION	The bias value is invalid
PTV_SUCCESS	The function completed successfully

SetTuningParameter

DEFINITION
```
PTres ptSetTuningParameterfv( PTenum param, const PTfloat *values )
```

DESCRIPTION
Sets an engine tuning option.

PARAMETERS
param The engine tuning option to be set. This can be one of the following:

PT_LOADING_BIAS_POINT	If the loading priority bias is set to PT_LOADING_BIAS_POINT then this option will set the actual point from which the loading bias is computed. The point should be passed through *values* as an array of 3 floats representing a point.

Values An array of float values

RETURN VALUE
A value indicating the result of the function, possible values are:

PTV_INVALID_OPTION	The param value is invalid
PTV_SUCCESS	The function completed successfully

GetTuningParameter

DEFINITION
```
PTres ptGetTuningParameterfv( PTenum param, PTfloat *values )
```

DESCRIPTION
Gets an engine tuning option.

PARAMETERS
param The engine tuning option to get. This can be one of the following:

PT_LOADING_BIAS_POINT	The point from which is used when the loading priority is set to PT_LOADING_BIAS_POINT. The point will be returned in values as an array of 3 floats representing a point.

Values An array of float values to receive the result

RETURN VALUE
A value indicating the result of the function, possible values are:

PTV_INVALID_OPTION	The origin or normal values are null
PTV_SUCCESS	The function completed successfully

SetCacheSizeMb

DEFINITION
```
PTvoid ptSetCacheSizeMb( PTuint mb )
```

DESCRIPTION
Sets the in memory cache size for background loading. A larger cache will minimize loading activity and improve performance.

PARAMETERS
mb The amount of memory in mb to use for the cache

GetCacheSizeMb

DEFINITION
```
PTuint ptGetCacheSizeMb
```

DESCRIPTION
Gets the in memory cache size for background loading.

AutoCacheSize

DEFINITION
```
PTuint ptAutoCacheSize
```

DESCRIPTION
Tells Vortex to manage the cache size according to the available memory

Point Search

SetIntersectionRadius

DEFINITION
```
PTres ptSetIntersectionRadius(PTfloat radius)
```

DESCRIPTION
Sets the intersection radius for Ray intersection queries. Since a point has no dimension, an intersection radius specifies the nominal spherical radius to consider when computing point ray intersection.

PARAMETERS

radius	The radius of the point

RETURN VALUE
A value indicating the result of the function, possible values are:

PTV_INVALID_PARAMETER	The radius is zero or less than zero
PTV_SUCCESS	The function completed successfully

GetIntersectionRadius

DEFINITION
```
PTfloat ptGetIntersectionRadius
```

DESCRIPTION
Returns the intersection radius setting

RETURN VALUE
A float representing the intersection radius

FindNearestScreenPoint

DEFINITION
```
PTint ptFindNearestScreenPoint( PThandle scene, PTint screenx, PTint screeny,
                                PTdouble *pnt )
```

DESCRIPTION
If OpenGL rendering is being used, this function will check the depth buffer for the current viewport context and unproject the point at screen, screen. The unproject point is used as a seed point to search for the actual point cloud point projected to this screen position. This can be used to implement fast point snapping on the point cloud.

PARAMETERS

scene	The scene to restrict the search to. Use zero to use a global search
screenx	The x position of the on screen pixel
screeny	The y position of the on screen pixel
pnt	An array of 3 double values to which the resulting point will be written

RETURN VALUE

The number of pixels difference between the screen position and the returned point projected position.

FindNearestScreenPointWDepth

DEFINITION

```
PTint ptFindNearestScreenPointWDepth( PThandle scene, PTint screenx,
        PTint screeny, PTfloat *dpArray4x4, PTdouble *pnt )
```

DESCRIPTION

In cases where an OpenGL context is not accessible or being used by providing depth values around the vertex, this function will search for the nearest point to a screen point in a similar way to ptFindNearestScreenPoint. This can be used to implement fast point snapping on the point cloud.

PARAMETERS

scene	The scene to restrict the search to. Use zero to use a global search
screenx	The x position of the on screen pixel
screeny	The y position of the on screen pixel
dpArray4x4	Array of 16 values arranged in a row major 4 x 4 matrix representing depth values of points around screen point
pnt	An array of 3 double values to which the resulting point will be written

RETURN VALUE

The number of pixels difference between the screen position and the returned point projected position.

FindNearestPoint

DEFINITION

```
PTfloat ptFindNearestPoint( PThandle scene, const PTdouble *search_pnt,
                            PTdouble *nearest )
```

DESCRIPTION

Finds the nearest point to the search point in the point cloud scene. This can be used to implement point snapping in a CAD system where an approximate 3d point can be resolved from the cursor by unprojecting the screen point given a depth value.

PARAMETERS

scene	A handle to the scene to search
search_pnt	The search point as a tuple of doubles (x,y,z)
nearest	A pointer to a tuple of doubles to receive the nearest cloud point

RETURN VALUE

The distance between the search point and nearest cloud point. A negative distance indicates that no point was found.

IntersectRay

DEFINITION

```
PTbool ptIntersectRay(PThandle scene, const PTdouble *origin,
                      const PTdouble *direction, PTdouble
                  *intersection, PTenum densityType, PTfloat
                  densityValue)
```

DESCRIPTION

Finds the nearest point to the start of the ray that lays on the ray within a the intersection radius (see ptSetIntersectionRadius). This can be used to implement point snapping in a CAD system where the camera position and cursor can be used to compute the ray parameters. Performance in this case would be acceptable, even on large datasets. However intensive use for applications such as ray tracing may not be practical.

PARAMETERS

scene	A handle to the scene to search
origin	The ray origin as a tuple of doubles (x,y,z)
direction	The pointer to a tuple of doubles to receive the nearest cloud point
intersection	The nearest intersection point candidate
densityType	The density type setting used to determine the number of points evaluated in the query See ptSetDensityQuery for a complete explanation of this parameter.
densityValue	The density value setting used to determine the number of points evaluated in the query. See ptSetDensityQuery for a complete explanation of this parameter.

RETURN VALUE

PT_TRUE if an intersection was found.

IntersectRayPntIndex

DEFINITION

```
PTbool ptIntersectRayPntIndex( PThandle scene, const PTdouble *origin,
     const PTdouble *direction, PThandle *cloud, PThandle *pntPartA,
     PThandle *pntPartB )
```

DESCRIPTION

Finds the nearest point to the start of the ray that lays on the ray within the intersection radius (see ptSetIntersectionRadius). This can be used to implement point snapping in a CAD system where the camera position and cursor can be used to compute the ray parameters. Performance in this case would be acceptable, even on large datasets. However intensive use for applications such as ray tracing may not be practical.

This function differs to the ptIntersectRay in that it returns a handle to a point that can be used with PointAttributes to retrieve the point's details.

PARAMETERS

scene	A handle to the scene to search
origin	The ray origin as a tuple of doubles (x,y,z)

direction	The pointer to a tuple of doubles to receive the nearest cloud point
cloud	Pointer to receive the cloud handle
pntPartA	Pointer to handle to receive part A of the point handle
pntPartB	Pointools to handle to receive part B of the point handle

RETURN VALUE
PT_TRUE if an intersection was found.

PointData

DEFINITION
```
PTbool ptPointData( PThandle cloud, PThandle pointIndex,
      PTdouble *position, PTshort *intensity, PTubyte *rgb, PTfloat *normal )
```

DESCRIPTION

Retrieves a points position and other data channels from a cloud handle and the points index within the point cloud

PARAMETERS

cloud	The point cloud handle
pointIndex	The points index in the point cloud
position	Array of 3 double values to receive the point's position
intensity	Pointer *to* single short value to receive the point's intensity
rgb	Pointer to 3 unsigned byte values to receive the point's intensity
normal	Pointer to 3 float values to receive the point's normal

RETURN VALUE
PT_TRUE if the function succeeds

PointAttributes

DEFINITION
```
PTuint ptPointAttributes( PThandle cloud, PThandle pntPartA, PThandle pntPartB )
```

DESCRIPTION

Returns a bitmask of the points attributes. This is a combination of the following bit masks:

PT_HAS_INTENSITY	The point has an intensity value
PT_HAS_RGB	The point as an RGB colour value
PT_HAS_NORMALS	The point has a normal vector value

PARAMETERS

cloud The point cloud that the point is in

pntPartA The point's part A handle

pntPartB The point's part B handle

RETURN VALUE
A bitmask of points attributes

GetPointAttribute

DEFINITION
```
PTbool ptGetPointAttribute( PThandle cloud, PThandle pntPartA, PThandle
                            pntPartB, PTuint attribute, void* data )
```

DESCRIPTION

Returns a point's attribute by its handle

PARAMETERS
cloud The point cloud that the point is in

pntPartA The point's part A handle

pntPartB The point's part B handle

attribute The required attribute, this should be one of the following:

`PT_HAS_INTENSITY`	The point has an intensity value
`PT_HAS_RGB`	The point as an RGB colour value
`PT_HAS_NORMALS`	The point has a normal vector value

data A pointer to a buffer to receive the point's attribute

RETURN VALUE
PT_TRUE if the function succeeds

Point Query

CreateSelPointsQuery

DEFINITION

```
PThandle ptCreateSelPointsQuery
```

DESCRIPTION

Creates a query for selected visible points only. This can be used in conjunction with rectangle, fence and other selection tools to develop downstream tools based on a select and operate paradigm with, for example for meshing, primitive fitting and feature extraction.

RETURN VALUE

A handle to the query. This handle is valid until the query is deleted. The query can be used multiple times.

SetQueryDensity

DEFINITION

```
PTres ptSetQueryDensity( PThandle query, PTenum densityType,
                         PTfloat densityValue )
```

DESCRIPTION

Sets the detail level for the query allowing quick processing of a view or density based subset of points.

PARAMETERS

query Handle of the query to update

densityType The level of detail required. This can be one of the following:

`PT_QUERY_DENSITY_FULL`	The query returns every point including points that in held in out-of-core storage useful for algorithms that need to process every point. This is the default behavior, however setting a densityValue less than 1 returns a subset of points, ie a percentage of 100 x the density value.
`PT_QUERY_DENSITY_VIEW`	A view based optimal point set. This is can be used for displaying points or providing a preview of a tool's result.
`PT_QUERY_DENSITY_VIEW_COMPLETE`	A view based optimal point set. This is can be used for displaying points or providing a preview of a tool's result.
`PT_QUERY_DENSITY_LIMIT`	The query returns a subset of points that best represent the entire point set. The number of points to be returned is specified by the *densityValue*.

densityValue A coefficient that modulates the density type. This is applied per region and can be used to evenly reduce the density of points retrieved. Used with PT_QUERY_DENSITY_VIEW the *densityValue* can be used to select a level-of-detail for fast dynamic display.
In the case of `PT_QUERY_DENSITY_LIMIT`, the density value specifies the maximum number of points to be returned.

RETURN VALUE
A value indicating the result of the function, possible values are:

`PTV_INVALID_OPTION`	The densityType parameter is invalid
`PTV_SUCCESS`	The function completed successfully

SetQueryRGBMode

DEFINITION
```
PTres ptSetQueryRGBMode( PThandle query, PTenum mode )
```

DESCRIPTION
Sets the RGB colour retrieval mode. This has no affect if a colour buffer is not provided in the following *ptGetPoints* function.

PARAMETERS
query Handle of the query to update

Mode The RGB colour retrieval mode, this can be one of the following:

`PT_QUERY_RGB_MODE_ACTUAL`	The query returns the actual scan RGB values. This is unaffected by the enabled state of point intensity or the plane shader
`PT_QUERY_RGB_MODE_SHADER`	The query returns rgb values that composite the current shading options. This might include intensity, scan rgb and the planar shader. This means that client code can simply use these RGB values to display the points without considering what shading options are to be applied. Note that this excludes any lighting consideration.

RETURN VALUE
A value indicating the result of the function, possible values are:

`PTV_INVALID_OPTION`	The mode parameter is invalid
`PTV_SUCCESS`	The function completed successfully

CreateVisPointsQuery

DEFINITION
```
PThandle ptCreateVisPointsQuery
```

DESCRIPTION
Creates a query for visible points only. This is all points that have not been hidden by an edit operation or by hiding a point cloud or scene. Note that this may also includes points that are outside the viewing frustum, ie not visible on screen.

RETURN VALUE
A handle to the query. This handle is valid until the query is deleted. The query can be used multiple times and is evaluated on execution.

CreateBoundingBoxQuery

DEFINITION
```
PThandle ptCreateBoundingBoxQuery( PTdouble minx, PTdouble miny,
                                   PTdouble minz, PTdouble maxx,
                                   PTdouble maxy, PTdouble maxz )
```

PARAMETERS
minx	Minimum extent of bounding box in X
miny	Minimum extent of bounding box in Y
minz	Minimum extent of bounding box in Z
maxx	Maximum extent of bounding box in X
maxy	Maximum extent of bounding box in Y
maxz	Maximum extent of bounding box in Z

DESCRIPTION
Creates a query for points within an axis aligned bounding box.

RETURN VALUE
A handle to the query. This handle is valid until the query is deleted. The query can be used multiple times and is evaluated on execution.

CreateBoundingSphereQuery

DEFINITION
```
PThandle ptCreateBoundingSphereQuery ( PTdouble *cen, PTdouble radius )
```

PARAMETERS
cen	Centre point of the sphere as an array of 3 double values
radius	Radius of sphere

DESCRIPTION
Creates a query for points within a sphere.

RETURN VALUE
A handle to the query. This handle is valid until the query is deleted. The query can be used multiple times and is evaluated on execution.

CreateFrustumPointsQuery

DEFINITION

```
PThandle ptCreateFrustumPointsQuery
```

DESCRIPTION

Creates a query for points within the current viewports view frustum. In order to ensure the frustum has been setup the viewport must either use OpenGL (these settings are read from OpenGL) or provide the projection, eye space transform and viewport dimensions via the View parameter functions.

RETURN VALUE

A handle to the query. This handle is valid until the query is deleted. The query can be used multiple times and is evaluated on execution.

CreateKNNQuery

DEFINITION

```
PThandle ptCreateKNNQuery  ( PTfloat *vertices, PTint numQueryVertices,
                             PTint k, PTfloat queryLOD)
```

DESCRIPTION

Creates a K-Nearest Neighbor query for a compact set of query points. Query points are specified in the vertices buffer in (X,Y,Z) triples. The number of vertices specified is passed, along with k, the requested maximum number of neighboring points to find for each query point. LOD specifies the minimum level of detail in the range [0..1] to use during analysis, where 0 is none and 1 is full detail. This algorithm makes use of coherence between points in the query set. For best performance, the query set's extents should be compact and small in relation to the over all scene. Query points may be unique or a subset of the scene. Note that if queries are a subset of the scene being analyzed, the query points themselves will be members of the result set. Multiple scenes and clouds are included in the analysis. A query scope will limit the regions analyzed, defined using setQueryScope(). Use ptGetQueryPointsMultif() to execute and obtain the results. Each query point has a set of results that may contain up to k items. Usually, if k or more items are available in the data set, exactly k items will be returned for each query point. Each query point's results are returned in sorted order with increasing distance from the query point.

RETURN VALUE

A handle to the query. This handle is valid until the query is deleted. The query can be used multiple times and is evaluated on execution.

GetQueryPoints

DEFINITION

```
PTuint ptGetQueryPointsd ( PThandle query, PTuint bufferSize,
                           PTdouble *geomBuffer, PTubyte *rgbBuffer,
                           PTshort *intensityBuffer,
                           PTubyte *selectionBuffer  )

PTuint ptGetQueryPointsf ( PThandle query, PTuint bufferSize,
                           PTfloat *geomBuffer, PTubyte *rgbBuffer,
                           PTshort *intensityBuff,
                           PTubyte *selectionBuff  )
```

DESCRIPTION

Retrieves query point geometry and optionally rgb, intensity and selection channels into one or more buffers. If the buffers are filled by the retrieval the function returns. To get the remaining points the function should be called until it returns 0 points.

PARAMETERS

query The query's handle. This is obtained from one of the query creation functions

bufferSize The size of the buffer to retrieve points as the number of points, not array elements

geomBuffer A pointer to the buffer to retrieve point geometry.
This should be an array of floats (or doubles) that has at least 3 x the number of point elements

rgbBuffer A pointer to the buffer to retrieve point geometry.
This should be an array of PTubyte that has at least 3 x the number of point elements. See *SetQueryRGBMode* for how the RGB retrieval can be configured. A null pointer can be passed for this parameter if point RGB is not required.

intensityBuff A pointer to the buffer to retrieve point intensity values as an array of 16 bit signed shorts.
A null pointer can be passed for this parameter if selection state is not required.

selectionBuff A pointer to the buffer to retrieve point selection / hidden state values as an array of PTubyte. A null pointer can be passed for this parameter if selection state is not required.

RETURN VALUE

The number of points written to the buffers by this iteration.

GetQueryPointsMulti

DEFINITION

```
PTuint ptGetQueryPointsMultid ( PThandle query, PTuint numResultSets,
                                PTuint bufferSize,
                                PTuint *resultSetSize,
                                PTdouble **geomBufferA,
                                PTubyte **rgbBufferA,
```

```
                                PTshort **intenBufferA,
                                PTubyte **selectionBufferA )

PTuint ptGetQueryPointsMultif ( PThandle query, PTuint numResultSets,
                                PTuint bufferSize,
                                PTuint *resultSetSize,
                                PTfloat **geomBufferA,
                                PTubyte **rgbBufferA,
                                PTshort **intenBufferA,
                                PTubyte **selectionBufferA )
```

DESCRIPTION

Retrieves multiple query point geometry and optionally RGB, intensity and selection channels into one or more buffers. This call should be used in conjunction with multiple query types such as ptCreateKNNQuery().

PARAMETERS

query The query's handle. This is obtained from one of the query creation functions

numResultSets The number of result sets expected by the caller. This should usually be equal to the number of query points included when a multi point query was created as one result set is associated with each query point.

bufferSize The size of each result buffer to retrieve points as the number of points, not array elements.

resultSetSize An array of integers in which to return the size of each result set. Usually, the size of this array must be at least equal to numResultSets.

geomBufferA A pointer to an array of pointers to result set buffers to retrieve point geometry. Result set buffers should be arrays of floats (or doubles) with 3 x the number of points as elements.

rgbBufferA A pointer to an array of pointers to result set buffers to retrieve point RGB color. Result set buffers should be arrays of PTubyte with at least 3 x the number of points as elements. See *SetQueryRGBMode* for how the RGB retrieval can be configured. A null pointer can be passed for this parameter if point RGB is not required.

intenBufferA A pointer to an array of pointers to result set buffers to retrieve point intensity values as an array of 16 bit signed shorts. A null pointer can be passed for this parameter if selection state is not required.

selectionBufferA A pointer to an array of pointers to result set buffers to retrieve point selection / hidden state values as an array of PTubyte. A null pointer can be passed for this parameter if selection state is not required

RETURN VALUE

The number of resultSets.

GetCloudProxyPoints

DEFINITION

```
PTuint ptGetCloudProxyPoints( PThandle scene, PTint num_points, PTfloat *pnts,
                              PTubyte *col )
```

DESCRIPTION

Extracts a small number of points that can be used as a scene proxy when the scene is unloaded or hided. This is a convenience function, using a Query with the appropriate density and colour setting will give the same result.

PARAMETERS

scene	A handle to the scene
num_points	The number of points desired
pnts	An array of floats that is at least 3 x num_points in size to receive the point positions
col	An array of unsigned bytes that is at least 3 x num_points in size to receive the point colours

RETURN VALUE

The actual number of points returned which may be lower than num_points

---

GetDetailedQueryPoints

DEFINITION

```
PTuint ptGetDetailedQueryPointsd ( PThandle query, PTuint bufferSize,
                                   PTdouble *geomBuffer, PTubyte *rgbBuffer,
                                   PTshort *intensityBuffer,
                                   PTubyte * selectionBuffer,
                                   PTuint numPointChannels,
                                   const PThandle *pointChanelsReq,
                                   PTvoid **pointChannels)

PTuint ptGetDetailedQueryPointsf ( PThandle query, PTuint bufferSize,
                                   PTfloat *geomBuffer, PTubyte *rgbBuffer,
                                   PTshort *intensityBuffer,
                                   PTubyte * selectionBuffer,
                                   PTuint numPointChannels,
                                   const PThandle *pointChanelsReq,
                                   PTvoid **pointChannels)
```

DESCRIPTION

These are detailed versions of `GetQueryPoints` allowing access to User Channels.

PARAMETERS

See GetQueryPoints above for shared parameters

numPointChannels	The number of user point channels to be returned
pointChannelsReq	A pointer to an array of point channel handles specifying which channels should be returned in the query.
pointChannels	A pointer to an array of point channel buffers to receive the point channels. These are specified as pointers to void and it is important that the buffers are at least the size of user channels bytes-per-point x the *bufferSize* to avoid buffer overrun.

RETURN VALUE

The number of points written to the buffers by this iteration.

DeleteQuery

DEFINITION

```
PTbool ptDeleteQuery( PThandle query )
```

PARAMETERS

query	The query's handle. This is obtained from one of the query creation functions

DESCRIPTION

Free's resources associated with the query. The query handle will be invalidated after the call.

RETURN VALUE

Boolean indicating success. If the query cannot be found PT_FALSE is returned.

ResetQuery

DEFINITION

```
PTbool ptResetQuery( PThandle query )
```

PARAMETERS

query	The query's handle. This is obtained from one of the query creation functions

DESCRIPTION

Resets the query to its creation state.

RETURN VALUE

Boolean indicating success. If the query cannot be found PT_FALSE is returned.

SetQueryScope

DEFINITION

```
PTres ptSetQueryScope ( PThandle query, PTuint cloudOrSceneHandle )
```

DESCRIPTION

By default a query will return points from every point cloud currently held by the Vortex engine. This function can be used to limit the scope of a query to a specific point cloud or point cloud scene (file).

PARAMETERS

Query	A handle to the query whose scope is to be set
cloudOrSceneHandle	A handle to either a point cloud or a point cloud scene to which the queries scope is being limited to

RETURN VALUE

A value indicating the result of the function, possible values are:

`PTV_INVALID_HANDLE`	The cloudOrSceneHandle or query handle is invalid
`PTV_SUCCESS`	The function completed successfully

Interaction

FlipMouseYCoords

DEFINITION

```
PTvoid ptFlipMouseYCoords
```

DESCRIPTION

Flips all screen coordinates in Y. Use this if the incoming mouse coordinates are from win32 or MFC and not OpenGL viewport coordinates. This function affects all functions that use screen coordinates, for example ptSelectPointsByFence.

DontFlipMouseYCoords

DEFINITION

```
PTvoid ptDontFlipMouseYCoords
```

DESCRIPTION

Unflips all screen coordinates in Y. This function affects all functions that use screen coordinates, for example ptSelectPointsByFence.

View Setup

ReadViewFromGL

DEFINITION

```
PTbool ptReadViewFromGL
```

DESCRIPTION

Reads the current view setup from the current OpenGL context. This is used in visibility computation and optimization of view based queries and rendering. If this method is used there is no need to use any further view setup functions found in this section.

RETURN VALUE

Boolean indicating success. There is no OpenGL context current if PT_FALSE is returned

---

SetViewProjectionOrtho

DEFINITION

```
PTvoid ptSetViewProjectionOrtho ( PTdouble l, PTdouble r, PTdouble b,
                                  PTdouble t, PTdouble n, PTdouble f )
```

DESCRIPTION

Sets up an ortho (parallel) view projection based on the frustum plane positions. This is used in visibility computation and optimization of view based queries and rendering.

PARAMETERS

l	Frustum's left plane position
r	Frustum's right plane position
b	Frustum's bottom plane position
t	Frustum's top plane position
n	Frustum's near plane position
f	Frustum's far plane position

SetViewProjectionFrustum

DEFINITION

```
PTvoid ptSetViewProjectionFrustum ( PTdouble l, PTdouble r, PTdouble b,
                                    PTdouble t, PTdouble n, PTdouble f )
```

DESCRIPTION

Sets up a perspective (parallel) view frustum. This is used in visibility computation and optimization of view based queries and rendering.

PARAMETERS

l	Frustum's left plane position
r	Frustum's right plane position
b	Frustum's bottom plane position
t	Frustum's top plane position
n	Frustum's near plane position
f	Frustum's far plane position

SetViewProjectionMatrix

DEFINITION

```
PTvoid ptSetViewProjectionMatrix( const PTdouble *matrix,
                                  bool row_major )
```

DESCRIPTION

Sets up a view projection using a 4x4 projection matrix. This is used in visibility computation and optimization of view based queries and rendering.

PARAMETERS

matrix	An array of the 16 double values of the matrix
row_major	Boolean indicating the ordering of the matrix values

---

SetViewProjectionPerspective

DEFINITION

```
PTvoid ptSetViewProjectionPerspective(PTenum type, PTdouble fov,
PTdouble aspect, PTdouble n, PTdouble f)
```

DESCRIPTION

Sets up a perspective view projection. This is used in visibility computation and optimization of view based queries and rendering.

PARAMETERS

type The model used to compute the projection matrix. Any of the following values can be used:

PT_PROJ_PERSPECTIVE_GL	The projection matrix is calculated to produce a matrix identical to OpenGL's gluPerspective
PT_PROJ_PERSPECTIVE_DX	The projection matrix is calculated to produce a matrix identical to a DirectX perspective matrix

PT_PROJ_PERSPECTIVE_BLINN	The projection matrix is calculated to using the Blinn

fov	The field of view in degrees
aspect	The aspect ratio of the projection. This should normally be the viewport width / height
n	The frustum's near plane position
f	The frustum's far plane position

SetViewEyeLookAt

DEFINITION

```
PTvoid ptSetViewEyeLookAt ( const PTdouble *eye3, const PTdouble
                                *target, const PT double *up3 )
```

DESCRIPTION

Sets up an eye transformation based on an eye position and target position. The result will be identical to OpenGL's gluLookAt matrix

PARAMETERS

eye3	The eye location specified as an array of 3 double values ie x,y,z
target	The location of the target of the view. Thie point will be mapped to the centre of the viewport. The target is specified as an array of 3 doubles.
up3	The up vector specified as an array of 3 doubles. This is usually the Z or Y axis.

SetViewEyeMatrix

DEFINITION

```
PTvoid ptSetViewEyeMatrix( const PTdouble *matrix16, PTbool row_major)
```

DESCRIPTION

Sets up an eye transformation directly by specifying a transformation matrix. Care should be taken to when using matrices that the row_major parameter is correctly specified.

PARAMETERS

matrix16	An array of the 16 double values of the matrix
row_major	Boolean indicating the ordering of the matrix values

---

SetViewportSize

DEFINITION

```
PTvoid ptSetViewportSize(PTint left, PTint bottom, PTuint width, PTuint height )
```

DESCRIPTION

Specifies the viewport size and position that the viewing frustum is mapped to. It is important to specify this correctly in order for the visibility computation to be correctly performed.

PARAMETERS

left	The viewport left position in pixels
bottom	The viewport bottom position in pixels
width	The viewport's width in pixels
height	The viewport's height in pixels

GetViewEyeMatrix

DEFINITION

```
PTvoid ptGetViewEyeMatrix( PTdouble *matrix )
```

DESCRIPTION

Retrieves the current eye transformation matrix used by Vortex for visibility determination

PARAMETERS

matrix	Buffer of 16 double values to receive the matrix values in column order

---

GetViewProjectionMatrix

DEFINITION

```
PTvoid ptGetViewProjectionMatrix( PTdouble *matrix16 )
```

DESCRIPTION

Retrieves the current projection matrix used by Vortex for visibility determination

PARAMETERS

matrix16	Buffer of 16 double values to receive the matrix values in column order

User Channels

CreatePointChannel

DEFINITION

```
PThandle ptCreatePointChannel( PTstr name, PTenum typesize, PTuint
                               multiple, void *defaultValue, PTuint
                               flags)
```

DESCRIPTION

Creates a user points channel enabling storage of arbitrary per point numerical data.

PARAMETERS

name	A unique name for the user channel.
typesize	The size of the per point value data type being stored in bytes.
multiple	The number of values per point
defaultValue	The default value specified as a void pointer to a buffer containing the values. If there are multiple values per point the buffer must contain all these values and be of size *typesize* x *multiple*.
flags	Additional creation options. A bitmask of the following options can be used:

`PT_CHANNEL_OUT_OF_CORE`	The user channel data is stored out-of-core. This will reduce memory overhead significantly at the cost of access performance.

RETURN VALUE

A handle to the user channel.

DeletePointChannel

DEFINITION

```
PTres ptDeletePointChannel( PThandle channel )
```

DESCRIPTION

Frees resources associated with the user channel. After calling this function the channel handle will no longer be valid.

RETURN VALUE

A value indicating the result of the function, possible values are:

`PTV_INVALID_HANDLE`	The channel handle is invalid
`PTV_SUCCESS`	The function completed successfully

SubmitPointChannelUpdate

DEFINITION

```
PTres ptSubmitPointChannelUpdate( PThandle query, PThandle channel )
```

DESCRIPTION

Submits changes made to user channels that have been changed after being retrieved in a buffer using a query. Vortex checks the contents of the buffer using the previously supplied pointer and saves the changed values back to the internal user data structures.

RETURN VALUE

A value indicating the result of the function, possible values are:

PTV_INVALID_HANDLE	The channel handle is invalid
PTV_SUCCESS	The function completed successfully

DrawPointChannelAs

DEFINITION

```
PTres ptDrawPointChannelAs( PThandle channel, PTenum method, PTfloat
                            param1, PTfloat param2 )
```

DESCRIPTION

Instructs the Vortex renderer to interpret the user channel values using a particular method for the purpose of drawing. In most cases only a single interpretation method can be active at one time enabling the rendering of a single User Channel only.

PARAMETERS

channel A handle to the channel to draw

method The method used to interpret the channel for drawing purposes. Any of the following values may be used:

PT_CHANNEL_AS_OFFSET	User channel values are interpreted as x,y,z offsets to the original point positions. The user channel must have a multiple of 3 values per point, ie one for each of the x,y and z components although the type of value can vary. *param1* specifies a scale value that is applied to the offset. *param2* is unused.
PT_CHANNEL_AS_RAMP	User channel values are interpreted as indices into a colour ramp. Index values range from 0 to 1. Values beyond this range will be modulated to fall in range. *param1* specifies the colour ramp to be used *param2* specifies a scaling factor to be applied the user channel values
PT_CHANNEL_AS_ZSHIFT*	Use channel values are interpreted as an offset to the original point's Z value.

	param1 specifies a scale value that is applied to the offset. *param2* is unused.
PT_CHANNEL_AS_RGB*	Use channel values are interpreted as R, G, B values. The user channel must have a multiple of 3 values per point, ie one for each of the x,y and z components although the type of value can vary. *param1* specifies a blend value that is used to blend the RGB values to the original RGB values. Valid values are between 0 (no channel rgb) to 1 (full channel rgb). *param2* is unused.

param1	First parameter, see above for usage
param2	Second parameter, see above for usage

RETURN VALUE

A value indicating the result of the function, possible values are:

PTV_INVALID_OPTION	The *method* parameter is invalid
PTV_INVALID_VALUE_FOR_PARAMETER	The value of *param1* or *param2* for the given method option is invalid
PTV_NOT_IMPLEMENTED_IN_VERSION	The method has not been implemented
PTV_INVALID_HANDLE	The *channel* handle is invalid
PTV_SUCCESS	The function succeeded

NOTES

Methods mark with an * have not been implemented as of release 1.5.0

WriteChannelsFile

DEFINITION

```
PTres ptWriteChannelsFile( const PTstr filename, PTint numChannels,
                           PThandle *channels )
```

DESCRIPTION

Writes a file that stores the specified channels for later retrieval. The channels file references the point clouds by the cloud GUID values, this enables order-independent loading of the channel file and POD file into the Vortex engine.

PARAMETERS

filename	The full path to the file to write. Any extension can be used

numChannels The number of channels to be written to the file. This will be the number of elements in the following *channels* array

channels An array of channel handles specifying the channels to output to the file

RETURN VALUE

A value indicating the result of the function, possible values are:

`PTV_FILE_NOTHING_TO_WRITE`	There is no user channel data to write
`PTV_FILE_WRITE_FAILURE`	There was a write failure
`PTV_FILE_FAILED_TO_CREATE`	The file failed to be created. This might be due to user permissions or file write being block by the OS or Anti-virus software
`PTV_SUCCESS`	The file was written successfully

ReadChannelsFile

DEFINITION

```
PTres ptReadChannelsFile( const PTstr filename, PTint &numChannels,
                          PThandle **channels )
```

DESCRIPTION

Reads a channel file containing one or more point channels. The channels reference the point clouds by the cloud GUID values, this enables order-independent loading of the channel file and POD file into the Vortex engine.

PARAMETERS

filename The full path to the file to write. Any extension can be used

numChannels [out] The number of channels that are read from the file

channels [out] A pointer to a handles pointer that will be set to point to an internal channels array. This will contain the channel handles that are read. Memory at this pointer must not be deleted as it is managed by Vortex.

RETURN VALUE

A value indicating the result of the function, possible values are:

`PTV_FILE_READ_FAILURE`	There was a read failure
`PTV_FILE_WRONG_TYPE`	The file is not a channels file
`PTV_FILE_NOT_ACCESSIBLE`	The file is not accessible
`PTV_FILE_NOT_EXIST`	The file does not exist
`PTV_SUCCESS`	The file was read successfully

SetChannelOOCFolder

DEFINITION

```
PTres ptChannelOOCFolder( const PTstr filepath )
```

DESCRIPTION

Sets the folder for temporary Out-of-Core files. This is by default the operating system's temporary folder.

PARAMETERS

filename	The full path to the folder to be used.

RETURN VALUE

A value indicating the result of the function, possible values are:

`PTV_FILE_NOT_ACCESSIBLE`	The folder is not accessible for writing to
`PTV_SUCCESS`	The folder was set successfully

DeleteAllChannels

DEFINITION

```
PTvoid ptDeleteAllChannels
```

DESCRIPTION

Deletes all the channels in the Vortex engine

VORTEX API INTERFACE

The Pointools Vortex API header file contains definitions of functions and extern declarations. As such, it is as not as easily read as a header file containing function declarations that most programmers are more familiar with. For this reason such an interface is provided here for reference purposes only.

```
/* typedefs */
typedef unsigned int            PTenum;
typedef bool                    PTbool;
typedef int                     PTint;
typedef int                     PTres;
typedef unsigned int            PTuint;
typedef float                   PTfloat;
typedef double                  PTdouble;
typedef short                   PTshort;
typedef unsigned short          PTushort;
typedef char                    PTbyte;
typedef unsigned char           PTubyte;
typedef unsigned __int64        PTuint64;
typedef __int64                 PTint64;
#define PTstr wchar_t*


typedef void                    PTvoid;
typedef PTuint                  PThandle;
typedef unsigned char           PTubyte;

/* Shader Enables */
#define PT_RGB_SHADER                       0x01
#define PT_INTENSITY_SHADER                 0x02
#define PT_BLENDING_SHADER                  0x03
#define PT_PLANE_SHADER                     0x04
#define PT_LIGHTING                         0x05
#define PT_CLIPPING                         0x06

/* Display Enables */
#define PT_ADAPTIVE_POINT_SIZE              0X100
#define PT_FRONT_BIAS                       0X101
#define PT_DELAYED_CHANNEL_LOAD             0X102

/* Shader Settings */
#define PT_PLANE_SHADER_DISTANCE            0x11
#define PT_PLANE_SHADER_VECTOR              0x12
#define PT_PLANE_SHADER_OFFSET              0x13

#define PT_INTENSITY_SHADER_CONTRAST        0x14
#define PT_INTENSITY_SHADER_BRIGHTNESS      0x15

#define PT_RGB_SHADER_CONTRAST              0x16
#define PT_RGB_SHADER_BRIGHTNESS            0x17

#define PT_LIGHT_VECTOR                     0x18
#define PT_LIGHT_ANGLE                      0x19
#define PT_LIGHT_COLOUR                     0x1a
#define PT_LIGHT_AMBIENT_COLOUR             0x1b
#define PT_LIGHT_DIFFUSE_COLOUR             0x1c
#define PT_LIGHT_SPECULAR_COLOUR            0x1d
#define PT_LIGHT_STRENGTH                   0x1f
#define PT_LIGHT_AMBIENT_STRENGTH           0x20
#define PT_LIGHT_DIFFUSE_STRENGTH           0x21
```

```
#define PT_LIGHT_SPECULAR_STRENGTH          0x22

#define PT_INTENSITY_SHADER_RAMP            0x23
#define PT_PLANE_SHADER_RAMP                0x24

#define PT_MATERIAL_AMBIENT                 0X25
#define PT_MATERIAL_DIFFUSE                 0x26
#define PT_MATERIAL_SPECULAR                0x27
#define PT_MATERIAL_GLOSSINESS              0x28

#define PT_PLANE_SHADER_EDGE                0x29

#define PT_EDGE_REPEAT                      0x00
#define PT_EDGE_CLAMP                       0x01
#define PT_EDGE_BLACK                       0x02
#define PT_EDGE_MIRROR                      0x03

/* units */
#define PT_METERS                           0x100
#define PT_DECIMETERS                       0x101
#define PT_CENTIMETERS                      0x102
#define PT_MILLIMETERS                      0x103
#define PT_FEET                             0x104
#define PT_FEET_US                          0x106
#define PT_INCHES                           0x105

/* draw modes */
#define PT_DRAW_MODE_STATIC                 0x01
#define PT_DRAW_MODE_INTERACTIVE            0x02
#define PT_DRAW_MODE_DEFAULT                0x00
#define PT_DRAW_MODE_COMPATIBILITY          0x04

/* selection modes */
#define PT_SELECT                           0x01
#define PT_DESELECT                         0x02
#define PT_SELECT_TOGGLE                    0x03

/*context */
#define PT_GLOBAL_CONTEXT                   0x01
#define PT_SCENE_CONTEXT                    0x02
#define PT_CLOUD_CONTEXT                    0x03
#define PT_VIEWPORT_CONTEXT                 0x04

/* constants */
#define PT_MAX_VIEWPORTS            32
#define PT_TRUE                     true
#define PT_FALSE                    false
#define PT_NULL                     0
#define PT_ERROR                    0

/* coordinate base */
#define PT_AUTO_BASE_DISABLED       0x0
#define PT_AUTO_BASE_CENTER         0x01
#define PT_AUTO_BASE_REDUCE         0x02
#define PT_AUTO_BASE_FIRST_ONLY     0x04

/* ramps */
#define PT_INTENSITY_RAMP_TYPE      0x01
#define PT_PLANE_RAMP_TYPE          0x02

/* point attributes */
#define PT_HAS_INTENSITY            0x01
#define PT_HAS_RGB                  0x02
```

```
#define PT_HAS_NORMAL                   0x04
#define PT_HAS_FILTER                   0x08
#define PT_HAS_ANALYTICAL               0x10

/* editing */
#define PT_EDIT_MODE_SELECT             0X01
#define PT_EDIT_MODE_UNSELECT           0X02
#define PT_EDIT_MODE_UNHIDE             0X03

/* query */
#define PT_QUERY_DENSITY_FULL                   0x01
#define PT_QUERY_DENSITY_VIEW                   0X02
#define PT_QUERY_DENSITY_LIMIT                  0X03
#define PT_QUERY_DENSITY_VIEW_COMPLETE          0X04
#define PT_QUERY_DENSITY_SPATIAL                0x07

#define PT_QUERY_RGB_MODE_ACTUAL                0x04
#define PT_QUERY_RGB_MODE_SHADER                0x05
#define PT_QUERY_RGB_MODE_SHADER_NO_SELECT      0x06

/* imaging */
#define PT_IMAGE_TYPE_COLOUR                    0x01
#define PT_IMAGE_TYPE_NORMAL                    0x02
#define PT_IMAGE_TYPE_DEPTH                     0x03
#define PT_IMAGE_TYPE_BUMP                      0X04

/* tuning */
#define PT_LOADING_BIAS_SCREEN                  0x01
#define PT_LOADING_BIAS_NEAR                    0x02
#define PT_LOADING_BIAS_FAR                     0x03
#define PT_LOADING_BIAS_POINT                   0x04

/* fitting */
#define PT_FIT_MODE_USE_SELECTED                0x01
#define PT_FIT_MODE_USE_INPUT                   0x02

/* eye perspective type */
#define PT_PROJ_PERSPECTIVE_GL                  0x01
#define PT_PROJ_PERSPECTIVE_DX                  0x02
#define PT_PROJ_PERSPECTIVE_BLINN               0x03

/* channel constants */
/* draw as */
#define PT_CHANNEL_AS_OFFSET                    0x01
#define PT_CHANNEL_AS_RAMP                      0x02
#define PT_CHANNEL_AS_ZSHIFT                    0x03
#define PT_CHANNEL_AS_RGB                       0x04

/* options */
#define PT_CHANNEL_OUT_OF_CORE                  0X01

/* meta data */
#define MAX_META_STR_LEN                        512


/* Pointools Vortex API v1.5 */
/* intitialization */
PTbool  ptInitialize( const PTubyte *license );

PTbool  ptIsInitialized();

PTvoid  ptSetWorkingFolder( const PTstr folder );
const  PTstr   ptGetWorkingFolder( void );
```

```
const  PTstr   ptGetVersionString( void );

PTvoid PTAPI  ptGetVersionNum( PTubyte *version );

PTvoid  ptRelease( void );

/* handle management */
PThandle      ptGetCloudHandleByIndex( PThandle scene, PTuint cloud_index );

PTuint        ptGetNumCloudsInScene( PThandle scene );

/* importing scene data */
PThandle      ptOpenPOD( const PTstr filepath );

PThandle      ptIsOpen( const PTstr filepath );

PThandle      ptBrowseAndOpenPOD( void );

/* management */
PTint   ptNumScenes( void );

PTint   ptGetSceneHandles( PThandle *handles );

PTbool  ptSceneInfo( PThandle scene, PTstr name, PTint &clouds, PTuint
&num_points, PTuint &specification, PTbool &loaded, PTbool &visible );

const PTstr   ptSceneFile( PThandle scene );

PTres   ptCloudInfo( PThandle cloud, PTstr name, PTuint &num_points,
       PTuint &specification, PTbool &visible );

PTres   ptSceneBounds( PThandle scene, PTfloat *lower3, PTfloat *upper3 );

PTres   ptCloudBounds( PThandle cloud, PTfloat *lower3, PTfloat *upper3 );

PTres   ptShowScene( PThandle scene, PTbool visible );

PTres   ptShowCloud( PThandle cloud, PTbool visible );

PTbool  ptIsSceneVisible( PThandle scene );

PTbool  ptIsCloudVisible( PThandle cloud );

PTres   ptUnloadScene( PThandle scene );

PTres   ptReloadScene( PThandle scene );

PTres   ptRemoveScene( PThandle scene );

PTvoid  ptRemoveAll();

/* Meta data */
PThandle ptReadPODMeta( const PTstr filepath );

PThandle ptGetMetaDataHandle( PThandle sceneHandle );

PTres  ptGetMetaData( PThandle metadataHandle, PTstr name,
                    PTint &num_clouds, PTuint64 &num_points, PTuint &scene_spec,
                    PTdouble *lower3, PTdouble *upper3 );
```

```
PTres  ptGetMetaTag( PThandle metadataHandle, const PTstr tagName,
                     PTstr value );

PTvoid ptFreeMetaData( PThandle metadataHandle );

/* user metatags */
PTint  ptNumUserMetaSections( PThandle metadataHandle );

const PTstr   PTAPI  ptUserMetaSectionName( PThandle metadataHandle,
       PTint section_index );

PTint  ptNumUserMetaTagsInSection( PThandle metadataHandle,
       PTint section_index );

PTres  ptGetUserMetaTagByIndex( PThandle metadataHandle,
       PTint section_index, PTint tag_index, PTstr name, PTstr value );

PTres  ptGetUserMetaTagByName( PThandle metadataHandle,
       const PTstr sectionDotName, PTstr value );

/* scene duplication */
PThandle      PTAPI  ptCreateSceneInstance( PThandle scene );

/* transformation */
PTres  ptSetCloudTransform( PThandle cloud, const PTdouble *transform4x4,
       bool row_order );

PTres  ptSetSceneTransform( PThandle scene, const PTdouble *transform4x4,
       bool row_order );

PTres  ptGetCloudTransform( PThandle cloud, PTdouble *transform4x3,
       bool row_order );

PTres  ptGetSceneTransform( PThandle scene, PTdouble *transform4x3,
       bool row_order );


/* persistence of viewport setup */
PTuint  ptGetPerViewportDataSize();

PTuint  ptGetPerViewportData( PTubyte *data );

PTres   ptSetPerViewportData( const PTubyte *data );


/* points */
PTuint  ptGetCloudProxyPoints( PThandle scene, PTint num_points, PTfloat *pnts,
       PTubyte *col );

PTuint  ptGetSceneProxyPoints( PThandle cloud, PTint num_points, PTfloat *pnts,
       PTubyte *col );


/* error handling */
PTstr   ptGetLastErrorString( void );

PTres   ptGetLastErrorCode( void );


/* view parameters - these operate in current viewport */
PTbool  ptReadViewFromGL( void );
```

```
PTvoid  ptSetViewProjectionOrtho( PTdouble l, PTdouble r, PTdouble b, PTdouble t,
       PTdouble n, PTdouble f );

PTvoid  ptSetViewProjectionFrustum( PTdouble l, PTdouble r, PTdouble b, PTdouble
       t, PTdouble n, PTdouble f );

PTvoid  ptSetViewProjectionMatrix( const PTdouble *matrix, bool row_major );

PTvoid  ptSetViewProjectionPerspective( PTenum type, PTdouble fov, PTdouble
       aspect, PTdouble n, PTdouble f);

PTvoid  ptSetViewEyeLookAt( const PTdouble *eye3, const PTdouble *target3, const
       PTdouble *up3 );

PTvoid  ptSetViewEyeMatrix( const PTdouble *matrix16, bool row_major );

PTvoid  ptSetViewportSize( PTint left, PTint bottom, PTuint width, PTuint height
);

PTvoid  ptGetViewEyeMatrix( PTdouble *matrix );

PTvoid  ptGetViewProjectionMatrix( PTdouble *matrix16 );

/* draw */
PTvoid  ptOverrideDrawMode( PTenum mode );

PTvoid  ptDrawGL( void );

PTvoid  ptDrawSceneGL( PThandle scene, PTbool dynamic );

PTuint  ptKbLoaded( PTbool reset );

PTuint  ptWeightedPtsLoaded( PTbool reset );

PTint64 ptPtsLoadedInViewportSinceDraw( PThandle forScene );

PTint64 ptPtsToLoadInViewport( PThandle forScene, PTbool reCompute );

PTvoid ptEndDrawFrameMetrics( void );

PTvoid ptStartDrawFrameMetrics( void );


/* units */
PTvoid  ptSetHostUnits( PTenum units );

PTenum  ptGetHostUnits( void );


/* Coordinate Management */
PTvoid  ptSetAutoBaseMethod( PTenum type );

PTenum  ptGetAutoBaseMethod( void );

PTvoid  ptGetCoordinateBase( PTdouble *coordinateBase );

PTvoid  ptSetCoordinateBase( PTdouble *coordinateBase );


/* viewports */
PTint   ptAddViewport( PTint index, const PTstr name );

PTvoid  ptRemoveViewport( PTint index );
```

```
PTvoid  ptSetViewport( PTint index );

PTint   ptSetViewportByName( const PTstr name );

PTvoid  ptCaptureViewportInfo( void );

PTvoid  ptStoreView( void );

PTint   ptCurrentViewport( void );

PTvoid  ptEnableViewport( PTint index );

PTvoid  ptDisableViewport( PTint index );

PTbool  ptIsViewportEnabled( PTint index );

PTbool  ptIsCurrentViewportEnabled( void );


/* offscreen viewport */
PTvoid*  ptCreateBitmapViewport(int w, int h, const PTstr name);

PTvoid  ptDestroyBitmapViewport(const PTstr name);


/* bounds of data */
PTbool  ptGetLowerBound( PTdouble *lower );

PTbool  ptGetUpperBound( PTdouble *upper );


/* shader options */
PTvoid  ptEnable( PTenum option );

PTvoid  ptDisable( PTenum option );

PTbool  ptIsEnabled( PTenum option );

PTres   ptPointSize( PTfloat size );

PTres   ptShaderOptionf( PTenum shader_option, PTfloat value );

PTres   ptShaderOptionfv( PTenum shader_option, PTfloat *value );

PTres   ptShaderOptioni( PTenum shader_option, PTint value );

PTres   ptGetShaderOptionf( PTenum shader_option, PTfloat *value );

PTres   ptGetShaderOptionfv( PTenum shader_option, PTfloat *values );

PTres   ptGetShaderOptioni( PTenum shader_option, PTint *value );

PTvoid  ptResetShaderOptions( void );

PTvoid  ptCopyShaderSettings( PTuint dest_viewport );

PTvoid  ptCopyShaderSettingsToAll( void );

PTint   ptNumRamps( void );

const  PTstr  ptRampInfo( PTint ramp, PTenum *type );
```

```
/* lighting */
PTres   ptLightOptionf( PTenum Light_option, PTfloat value );

PTres   ptLightOptionfv( PTenum Light_option, PTfloat *value );

PTres   ptLightOptioni( PTenum Light_option, PTint value );

PTres   ptGetLightOptionf( PTenum Light_option, PTfloat *value );

PTres   ptGetLightOptioni( PTenum Light_option, PTint *value );

PTvoid  ptCopyLightSettings( PTuint dest_viewport );

PTvoid  ptCopyLightSettingsToAll();

PTvoid  ptResetLightOptions();


/* editing options */
PTres   ptSetSelectPointsMode( PTenum select_mode );

PTenum  ptGetSelectPointsMode( void );

PTvoid  ptSelectPointsByRect( PTint x_edge, PTint y_edge, PTint x2_edge,
       PTint y2_edge, PTint height );

PTres   ptSelectPointsByFence( PTint num_points, const PTint *points );

PTres   ptSelectPointsByCube( const PTfloat *centre, PTfloat radius );

PTres   ptSelectPointsByPlane( const PTfloat *origin, const PTfloat *normal,
       PTfloat thickness );

PTres   ptSelectPointsByBox( const PTfloat *lower, const PTfloat *upper );

PTres  ptSelectPointsByOrientedBox( const PTfloat *lower, const PTfloat *upper,
                                    const PTfloat *pos, PTfloat *uAxis, PTfloat
                           *vAxis );

PTres   ptSelectPointsBySphere( const PTfloat *centre, PTfloat radius );

PTvoid   ptInvertSelection( void );

PTvoid   ptInvertVisibility( void );

PTvoid  ptHideSelected( void );

PTvoid  ptUnhideAll( void );

PTvoid  ptUnselectAll( void );

PTvoid  ptSetSelectionScope( PThandle sceneOrCloudHandle );

PTvoid  ptRefreshEdit( void );

PTvoid  ptClearEdit( void );

PTvoid  ptStoreEdit( const PTstr name );
```

```
PTbool  ptRestoreEdit( const PTstr name );

PTbool  ptRestoreEditByIndex( PTint index );

PTbool  ptDeleteEdit( const PTstr name );

PTbool  ptDeleteEditByIndex( PTint index );

PTvoid  ptDeleteAllEdits( void );

PTint   ptNumEdits( void );

const  PTstr  ptEditName( PTint index );

PTint  ptGetEditData( PTint index, PTubyte *data );

PTint  ptGetEditDataSize( PTint index );

PTvoid  ptCreateEditFromData( const PTubyte *data );


/* point layers */
PTbool  ptSetCurrentLayer( PTuint layer );

PTuint  ptGetCurrentLayer();

PTbool  ptLockLayer( PTuint layer, PTbool lock );

PTbool  ptIsLayerLocked( PTuint layer );

PTbool  ptShowLayer( PTuint layer, PTbool show );

PTbool  ptIsLayerShown( PTuint layer );

PTbool  ptDoesLayerHavePoints( PTuint layer );

PTvoid  ptClearPointsFromLayer( PTuint layer );

PTvoid  ptResetLayers();

PTbool  ptCopySelToCurrentLayer( PTbool deselect );

PTbool  ptMoveSelToCurrentLayer( PTbool deselect );


/* optimisation and rendering options*/
PTvoid       ptDynamicFrameRate( PTfloat fps );

PTfloat      ptGetDynamicFrameRate();

PTvoid       ptStaticOptimizer( PTfloat opt );

PTfloat      ptGetStaticOptimizer();

PTvoid       ptGlobalDensity( PTfloat opt );

PTfloat      ptGetGlobalDensity( void );


/* Query */
PTres         ptSetIntersectionRadius(PTfloat radius);

PTfloat       ptGetIntersectionRadius( void );
```

```
PTint        ptFindNearestScreenPoint( PThandle scene, PTint screenx, PTint
                                       screeny, PTdouble *pnt );

PTint   ptFindNearestScreenPointWDepth( PThandle scene, PTint screenx,
             PTint screeny, PTfloat *dpArray4x4, PTdouble *pnt );

PTfloat  ptFindNearestPoint( PThandle scene, const PTdouble *pnt,
             PTdouble *nearest );

PTbool  ptIntersectRay( PThandle scene, const PTdouble *origin,
             const PTdouble *direction, PTdouble *intersection );

PTbool  ptIntersectRayPntIndex( PThandle scene, const PTdouble *origin,
             const PTdouble *direction, PThandle *cloud, PThandle *pntPartA,
             PThandle *pntPartB );

PTbool  ptPointData( PThandle cloud, PThandle pointIndex,
       PTdouble *position, PTshort *intensity, PTubyte *rgb, PTfloat *normal );

PTuint  ptPointAttributes( PThandle cloud, PThandle pntPartA, PThandle pntPartB
);

PTbool  ptGetPointAttribute( PThandle cloud, PThandle pntPartA, PThandle
                           pntPartB, PTuint attribute, void* data );


/* Selection */
PThandle  ptCreateSelPointsQuery();

PThandle  ptCreateVisPointsQuery();

PThandle  ptCreateFrustumPointsQuery();

PThandle  ptCreateKNNQuery(PTfloat *vertices, PTint numQueryVertice, PTint k,
                            PTfloat lod = 1);

PThandle  ptCreateBoundingBoxQuery( PTdouble minx, PTdouble miny,
       PTdouble minz, PTdouble maxx, PTdouble maxy, PTdouble maxz );

PThandle  ptCreateOrientedBoundingBoxQuery( PTdouble minx, PTdouble miny,
       PTdouble minz, PTdouble maxx, PTdouble maxy, PTdouble maxz, PTdouble posx,
       PTdouble posy, PTdouble posz, PTdouble ux, PTdouble uy, PTdouble uz,
       PTdouble vx, PTdouble vy, PTdouble vz);

PThandle  ptCreateBoundingSphereQuery( PTdouble *cen, PTdouble radius );

PTbool  ptDeleteQuery( PThandle query );

PTbool  ptResetQuery( PThandle query );

PTres  ptSetQueryRGBMode( PThandle query, PTenum mode );

PTres  ptSetQueryDensity( PThandle query, PTenum densityType,
       PTfloat densityValue );

PTres  ptSetQueryScope( PThandle query, PThandle sceneOrCloudHandle );

PTuint  ptGetQueryPointsd( PThandle query, PTuint bufferSize,
       PTdouble *geomBuffer, PTubyte *rgbBuffer, PTshort *intensityBuffer,
       PTubyte *selectionBuffer );
```

```
PTuint  ptGetDetailedQueryPointsd( PThandle query, PTuint bufferSize,
       PTdouble *geomBuffer, PTubyte *rgbBuffer, PTshort *intensityBuffer,
       PTfloat *normalBuffer, PTubyte *filter, PTuint numPointChannels,
       const PThandle *pointChannelsReq, PTvoid **pointChannels );

PTuint  ptGetQueryPointsf( PThandle query, PTuint bufferSize,
       PTfloat *geomBuffer, PTubyte *rgbBuffer, PTshort *intensityBuffer,
       PTubyte *selectionBuffer);

PTuint  ptGetDetailedQueryPointsf( PThandle query, PTuint bufferSize,
       PTfloat *geomBuffer, PTubyte *rgbBuffer, PTshort *intensityBuffer,
       PTfloat *normalBuffer, PTubyte *filter,PTuint numPointChannels,
       const PThandle *pointChannelsReq, PTvoid **pointChannels );

PTuint ptGetQueryPointsMultif( PThandle query, PTuint numResultSets,
       PTuint bufferSize, PTuint *resultSetSize, PTfloat **geomBufferArray,
       PTubyte **rgbBufferArray, PTshort **intensityBufferArray,
       PTubyte **selectionBufferArray);

PTuint ptGetQueryPointsMultid( PThandle query, PTuint numResultSets,
       PTuint bufferSize, PTuint *resultSetSize, PTdouble **geomBufferArray,
       PTubyte **rgbBufferArray, PTshort **intensityBufferArray,
       PTubyte **selectionBufferArray);


/* screen interaction */
PTvoid  ptFlipMouseYCoords( void );

PTvoid  ptDontFlipMouseYCoords( void );


/* tuning and memory management */
PTvoid  ptSetCacheSizeMb( PTuint mb );

PTuint  ptGetCacheSizeMb();

PTvoid  ptAutoCacheSize();

PTres   ptSetLoadingPriorityBias( PTenum bias );

PTenum  ptGetLoadingPriorityBias();

PTres   ptSetTuningParameterfv( PTenum param, const PTfloat *values );

PTres   ptGetTuningParameterfv( PTenum param, PTfloat *values );



/* User data channel */
PThandle      ptCreatePointChannel( PTstr name, PTenum typesize, PTuint
       multiple, void* default_value, PTuint flags );

PTres  ptDeletePointChannel( PThandle channel );

PTres  ptSubmitPointChannelUpdate( PThandle query, PThandle channel );

PTres   ptWriteChannelsFile( const PTstr filename, PTint numChannels,
             PThandle *channels );

PTres   ptReadChannelsFile( const PTstr filename );

PTres  ptDrawPointChannelAs( PThandle channel, PTenum method, PTfloat param1,
             PTfloat param2 );
```

```
PTres  ptSetChannelOOCFolder( const PTstr foldername );

PTvoid ptDeleteAllChannels( void );

#endif
```