# CT-analyser (v1.3.2)
# Creating plug-ins, a quick overview

## Introduction

CT-analyser is designed for the Skyscan micro-CT user to obtain quantitative data and visual models from scanned datasets of objects. It accompanies all Skyscan systems and its application and methods are the same for analysis of all scanned objects, both in-vivo and in-vitro.

CT-analyser also supports 3rd party plug-ins with true 3D array access. This manual is a quick overview based on sample code, how to create such a plug-in. A more detailed manual with advanced tips and techniques will follow in the near future.

## The programming language

A choice has been made to use Microsoft Visual C++ 6.0 as base language for our plug-in example. But it can easily be ported to Microsoft Visual C++ .NET (v2002 or v2003). Current versions do not support. NET-managed code, because the .NET has problems finding its correct assemblies when the project uses other assemblies and Microsoft has confirmed that this is a problem still to be solved. But using .NET is possible if you install all the assemblies in the GAC using an installer.

## The plug-ins

The Plug-in is conventional dll file, but with a renamed file extension '.ctp' to simplify detecting the plug-in by CTAn. E.g.: DemoPlg.ctp, and can be acompanied with a help file like DemoPlg.hlp or DemoPlg.chm.

This plug-in can be put anywhere on the computer, but CTAn must point its plug-in folder to that folder and only one folder is allowed. The most logical choice would be *C:\ Program files\Skyscan\CTAn\Plug-ins*. Multiple plug-ins can be put into that same folder as long as there are no conflicting file names. One Plug-in file can contain multiple plug-in functionalities.

## The Skyscan demo plug-ins

Skyscan provides a demo plug-in in 2 variants. One is a MFC ready project: DemoPlg (MFC), the other is a pure Windows version not using MFC: DemoPlg (W32).

We are going to write something about DemoPlg (MFC)

## The short overview of DemoPlg (MFC)

If you open the project, then you will find in DemoPlg.cpp the entry point code for every parameter implemented in this plug-in. In the demo sample you can find an Invert-filter and an And-Filter.

```
// entry point
extern "C" __declspec(dllexport) CPlugIn* CreatePlugIn( DWORD id, LPCTSTR lpszStdOut,
                                                        LPCTSTR lpszStdErr ) {
        switch( id ) {
          case INV_FLT:
                return new CInvFlt( lpszInvFltName, lpszInvFltDesc, lpszStdOut, lpszStdErr );
          case AND_FLT:
                return new CAndFlt( lpszAndFltName, lpszAndFltDesc, lpszStdOut, lpszStdErr );
        }
        return NULL;
}
```

We will continue with the CinvFlt plug-in which is defined in InvFlt.cpp and InvFlt.h.

```
typedef struct tag_INVCONF
{
    UINT mask;
} INVFLTCONFIG, *LPINVFLTCONFIG;

class CInvFlt: public CPlugIn {
public:
        CInvFlt( LPCTSTR lpszName, LPCTSTR lpszDescription, LPCTSTR lpszStdOut, LPCTSTR pszStdErr );
        virtual ~CInvFlt();
        virtual BOOL Run( IDatasetInfo& di, IPlgProgress &pr );
        virtual BOOL About();
        virtual BOOL Config();
        virtual BOOL Help();
private:
        BOOL DoConfig( BOOL bRun );
        INVFLTCONFIG m_config;
};
```

We have 2 important elements:

- Structure: **INVFLTCONFIG** that defines our configuration information customable for any plug-in, as long as it stays within one monolithic block. E.g. strings should reserve enough space to be contained into the structure, instead of a pointer to another location.
- Plug-in class: **cInvFlt** that inherits from the Skyscan default plug-in **CPlugin**.

Note that **INVFLTCONFIG** is assigned to attribute m_Config, and later we will see that in the constructor, that this m_Config will be marked for internal use.

If you look at **cInvFlt** then we also see 4 methods.

- Run (): Starts the actual processing, and contains your calculation code.
- About (): *(Optional)* Contains your about box functionality to show the about box.
- Config ()*: (Optional)* Contains your configuration box functionality to show the configuration window.
  *Note: No dataset data is available at this time, only during Run ().*
- Help (): *(Optional)* Contains your help functionality to show the help file.
  *Note: Support for *.chm or *.hlp files are possible.*

We will explain these functions in more details a little bit further.

The constructor must tell what functionality it supports.

```
CInvFlt::CInvFlt( LPCTSTR lpszName, LPCTSTR lpszDescription, LPCTSTR lpszStdOut, LPCTSTR pszStdErr )
                  : CPlugIn( PLG_IMAGE_8 | PLG_IMAGE_16 | PLG_IMAGE_32 | PLG_OUT_IMAGE_GR | PLG_CONFIG
                    | PLG_ABOUT | PLG_HELP, lpszName, lpszDescription, lpszStdOut, pszStdErr )
{
        // Set default configuration values before assigning
        m_config.mask = 255;
        // Assign configuration. Plug-in need this to manage configurations
        AssignConfigBlob( &m_config, sizeof( m_config ));
}
```

The above sample shows you that CinvFlt will provide support for 8, 16 and 32-bit images (PLG_IMAGE_8, PLG_IMAGE_16 and PLG_IMAGE_32) in future releases, but at this time CTAn v1.3.2 only supports 8 bit images.

Next constants changes the way the dataset bytes are formatted:

- PLG_INP_IMAGE_BW is used, when the plug-in accepts black and white images like threshold values (either byte value 0 or 255 for 8 bit or 0 and 65535 for 16 bit.
  *(Information purposes)*
- PLG_INP_IMAGE_GR is used, when the plug-in accepts grey scale values.
  *(Information purposes)*
- By using PLG_OUT_IMAGE_BW or PLG_OUT_IMAGE_GR you can change the dataset byte format.
  *E.g.: For a custom threshold plug-in you use PLG_INP_IMAGE_GR as grey level input and PLG_OUT_IMAGE_BW as threshold binary output.*

The example also shows that we support an about box, help functionality and configuration window. (PLG_CONFIG, PLG_ABOUT and PLG_HELP).

*Note: In case the mask is used then you should also add PLG_MASK_8, PLG_MASK_16 and PLG_MASK_32 and if you modify the mask then PLG_OUT_MASK_GR must be added. The same rules work like PLG_INP_IMAGE_BW, PLG_INP_IMAGE_GR …*

Finally you see that the m_Config is assigned to the configuration storage handler and that it is initialized to some default settings before assigning.

Functions: About(), Config() and Help() are pretty straightforward to use, so we are not going to discuss these in this quick overview.

The Run() method is our processing method, everything needed for processing is done here. When we leave this method, all the processing has been done.

```
BOOL CInvFlt::Run( IDatasetInfo& di, IPlgProgress &pr )
{
        if( !IsValid( di )) return FALSE;

        BOOL batch = di.IsBatchMode();
        if( !batch && !DoConfig( TRUE )) return FALSE;

…
```

The Run() method has 2 important parameters:

- IDatasetInfo: Passing on information regarding the dataset coming from CTAn.
- IPlgProgress: Feedback for the CTAn progress bar so the user can see how far we have proceeded.

```
class IDatasetInfo
{
public:
 virtual UUID GetId() = 0;              // Unique GUID unique for one dataset processing sequence
 virtual DWORD GetStyle() = 0;          // PLG_IMAGE_8, PLG_IMAGE_16,... PLG_MASK_8, PLG_MASK_16...
 virtual VOID*** GetImageSet( BOOL bReadOnly ) = 0;   // 3-Dimensional array of the Image data
 virtual VOID*** GetMaskSet( BOOL bReadOnly ) = 0;    // 3-Dimensional array of the image mask
 virtual SIZE GetImageSize() = 0;       // Width and height of one cross section in X-Y plane.
 virtual DWORD GetImageCount() = 0;     // Number of images in in the 3D array in Z access.
 virtual DWORD GetImageOrigin() = 0;    // First cross section sequence number from original dataset.
 virtual DWORD GetImageOffset() = 0;    // Sequence step interval. = the distance between 2 cross
                                        // section layers.
 virtual double GetPixelLenght() = 0;   // Size of X pixel in milimeter!
 virtual LPCTSTR GetDatasetName() = 0;  // Filemask + path + extension of source dataset
 virtual BOOL IsBatchMode() = 0;        // When TRUE then we are running in batch mode
 virtual SELECTION GetSelection() = 0;  // Top and bottom 3D array number of our selected area to be
                                        // analyzed.
 virtual int GetUnitType() = 0;         // Default unit specified in the CTAn 0 - pixel, 1 - um,
                                        // 2 - mm, 3 - inch
 virtual DWORD GetImageReduction() = 0; // The amount of reduction from the source dataset
};
```

```
class IPlgProgress
{
public:
  virtual int SetStep( int step ) = 0; // Step interval when Step() is called
  virtual int StepIt() = 0;            // Jumps SetStep() numbers on the progress bar.
  virtual int SetPos( int pos ) = 0;   // Direcly sets the position on the progress bar
  virtual void SetRange( int min, int max ) = 0;// The minimum and maximum range representing 0..100%
                                                // on the progress bar
  virtual void SetText( LPCTSTR text ) = 0;     // Text that is also shown on the progress bar. e.g.
                                                // time to go?
  virtual BOOL IsInterrupted() = 0;    // When TRUE then we cancelled the processing
};
```

Typical start up use is shown here.

Note the setting-up the Progress bar (pr.SetRange()), getting the top and bottom indexes Z-axis, for our upper and lower cross-section range.

```
BOOL CInvFlt::Run( IDatasetInfo& di, IPlgProgress &pr )
{
        if( !IsValid( di )) return FALSE;

        BOOL batch = di.IsBatchMode();
        if( !batch && !DoConfig( TRUE )) return FALSE;
        char dbuf[10], tbuf[10];
        fprintf( stderr, "[ %s %s ]\t", _strdate( dbuf ), _strtime( tbuf ) );

        fprintf( stderr, "%s started...\n", GetName());

        fprintf( stdout, "[ %s %s ]\t", _strdate( dbuf ), _strtime( tbuf ) );
        puts( GetName());
        fprintf( stdout, "mask = %d\n", m_config.mask );

        DWORD dz = di.GetImageCount();
        SELECTION sel = di.GetSelection();
        if( sel.top < dz ) dz = 1 + sel.top;
        SIZE sz = di.GetImageSize();
        pr.SetRange( sel.bottom, dz );
        pr.SetPos( 0 );
        pr.SetStep( 1 );

        switch( IMAGE_TYPE( di.GetStyle())) { // select type of image
               case PLG_IMAGE_8: {
                 BYTE *** _img = ( BYTE*** )di.GetImageSet( FALSE ); //cast pointer to valid value
                 for( DWORD z = sel.bottom; z < dz; z++ ) {
                       pr.StepIt();
                       if( pr.IsInterrupted()) {
….
```

Also important in the above part of the code is the fact that there are 2 standard output Streams:

- stdout: Intended to store calculation information and sometimes error codes when an error occurred. This is the scientific result.
- stderr: Intended for tracing information or more detailed error codes.

Accessing the 3D image array is very simple:

In the case you have an 8 bit image data (IMAGE_TYPE (di.GetStyle ()) ==PLG_IMG_8) then you have an array of Byte***, in the case of 16 bit then a array of WORD***. For example:

```
switch( IMAGE_TYPE( di.GetStyle())) { // select type of image
        case PLG_IMAGE_8: {
               BYTE *** _img = ( BYTE*** )di.GetImageSet( FALSE );  //cast pointer to valid value
…
```

The same holds true for the image mask:

Then is the test (**MASK**_TYPE (di.GetStyle())==PLG_**MASK**_8)

And the pointer BYTE*** mask= (BYTE***)di.Get**Mask**Set(FALSE);
*Note: setting the flag to TRUE indicates that we use the dataset as read-only, else set it to FALSE.*

A more extended sample can be found here:

```
switch( IMAGE_TYPE( di.GetStyle())) { // select type of image
        case PLG_IMAGE_8: {
           BYTE *** _img = ( BYTE*** )di.GetImageSet( FALSE );       //cast pointer to valid value
          for( DWORD z = sel.bottom; z < dz; z++ ) {
                pr.StepIt();
                if( pr.IsInterrupted()) {
                        fprintf( stderr, "[ %s %s ]\t", _strdate( dbuf ), _strtime( tbuf ) );
                        fprintf( stderr, "%s canceled by user\n", GetName());
                        return FALSE;
                }
                for( int y = 0; y < sz.cy; y++ ) {
                        for( int x = 0; x < sz.cx; x++ ) {
                                _img[ z ][ y ][ x ] ^= m_config.mask;
                                }
                        }
                }
        }
        break;
        case PLG_IMAGE_16: {
                WORD *** _img = ( WORD*** )di.GetImageSet( FALSE );
                for( DWORD z = sel.bottom; z < dz; z++ ) {

…
```

Some notes:

- The x, y, z access is written as img[z][y][x] in the array.
- If you have a 8 bit image data, then you could have a 8 bit image mask
- You both have an image pointer and a mask pointer at the same time.
- The array is a byte of *** pointers, this means that '**img++**' will not advance the pointer one x position, but will generate an access violation since the memory layout is in a random order because the complete image is stored in a memory mapped file and might not be in memory all the time but stored on the hard disk. Although it has a performance penalty, you are only limited to 2GB depending on your OS, plug-in memory use and if you have enough free hard disk space. You have true 3D random access.
- If you work in 2D images and want to speed up calculations: Then allocate one memory block, copy all bytes for that one layer to this memory block, and this way you are sure that all memory bytes are in Sequentional order stored in memory. Giving a performance boost because of the caching mechanism of the processor.
- Returning function result TRUE will tell CTAn that everything went smooth, FALSE that something have failed.
- Do not display dialog boxes with user input when running in batch mode. And if you need something like that, then provide an option in the configuration window to disable it.
- You are responsible for accessing all the layers, the progress bar notification and stopping the loop when someone presses Cancel (pr.IsInterrupted()).
- A dataset preview and configuration dialog box is possible during Run() but make sure that in the dialog box does not popup during batch mode.