ManagedITK: .NET Wrappers for ITK

Release 3.6.0.2

Dan Mueller[1]

June 23, 2008

[1]Philips Healthcare, PII Development, Best, Netherlands

Abstract

ManagedITK generates wrappers around ITK for .NET languages. These wrappers can be used with any CLR language, including C#, VB.NET, IronPython, and others. ManagedITK is useful for a number of reasons, including the ability to rapidly create graphical user interfaces using Windows Forms (also known as `System.Windows.Forms`). Full source code and many in-depth examples accompany this article. Pre-compiled .NET assemblies can also be downloaded for easy integration into standalone C# applications.

Keywords: *C#, C++/CLI, .NET, CLR, ITK, medical image processing*

1 Introduction

The Insight Toolkit (ITK) [4] is an open-source software system designed for image segmentation and registration — particularly for medical images. ITK is implemented in C++ and supports multiple platforms including Windows, Unix, MacOSX, etc. The toolkit is organised in a data-flow architecture: process objects (eg. filters) consume data objects (eg. images). The toolkit is implemented using generic programming principles, whereby objects are templated over different types at compile-time. The architecture and design of ITK, together with the complied C++ implementation, provides for efficiency, speed, and flexibility. However, interpretive or scripting languages are useful for rapidly prototyping applications and developing graphical user interfaces (GUIs). ITK has recently upgraded it's wrapping system to WrapITK [3].

WrapITK produces automatically generated wrappers for common ITK objects using CableSwig[1]. SWIG[2], upon which CableSwig is built, supports the generation of wrappers from C++ to various scripting languages including Perl, PHP, Python, Tcl, Ruby and PHP, as well as other non-scripting languages such as C#, Lisp, Java, Modula-3 and OCAML. However, CableSwig derives from an

[1]Cable Automates Bindings for Language Extension (Cable) Simplified Wrapper and Interface Generator (Swig): http://www.itk.org/HTML/CableSwig.html

[2]Simplified Wrapper and Interface Generator: http://www.swig.org/

old version of SWIG and at the time of writing only supports Python, Java, and Tcl. It is desirable to generate ITK wrappers for CLR[3] languages, such as C#, VB.NET and IronPython. The .NET CLR is a popular platform which provides a large body of pre-coded solutions to common programming problems, including a powerful GUI application programming interface (API) known as `System.Windows.Forms` or simply Windows Forms.

This article describes ManagedITK — a project which generates ITK wrappers for .NET CLR languages. It extends (probably more accurately mangles) WrapITK, but is a separate system which does not use CableSWIG or SWIG. The generated wrappers are created in a semi-automated fashion, tailored to produce a set of managed classes with well-defined methods, properties, and events. As such, the generated wrappers are not 100% pure (ie. in some cases the ITK API has been altered to better fit the Common Language Infrastructure (CLI) architecture, especially with the addition of properties). The next section describes the features of ManagedITK, providing a discussion of both advantages and disadvantages. Following this, the use of the pre-compiled assemblies is discussed, as well as the compilation process from the included source files. The bulk of the remaining text is devoted to in-depth examples demonstrating the nuts-and-bolts of using the .NET CLR wrappers.

2 Features

As with any engineering solution, ManagedITK offers a number of benefits at the cost of various trade-offs. The advantages of ManagedITK include:

Rapid GUI Development: The Windows Forms library can be used to quickly and easily create graphical user interfaces which interact with ITK data and process objects. For examples see Section 4.1.6, and Section 4.4.1.

Run-time Type Specification: Most ManagedITK objects provide “runtime-type wrapper” objects which can be used to specify the explicit type at run-time, rather than compile-time (as with the native C++ templates). For example, the `itkImage` class provides a runtime-type wrapper around the explicit types `itkImage_UC2`, `itkImage_SS2`, `itkImage_F2`, etc, allowing the user to choose the type at run-time.

Simplified Event Handling: Native ITK events can be easily observed using managed `delegates`. See Section 4.3.2 for more details.

Multi-language Support: Supported languages include C#, C++/CLI, VB.NET, IronPython[4], F#[5], Chrome (Object Pascal)[6], and other languages which target the .NET CLR.

[3] The CLR (Common Language Runtime) is the Microsoft Windows implementation of the CLI (Common Language Infrastructure). Basically it is a Windows implementation of a virtual machine able to run CIL (Common Intermediate Language) bytecode. See Figure 1.

[4] http://www.codeplex.com/IronPython/Wiki/View.aspx

[5] http://research.microsoft.com/fsharp

[6] http://www.chromesville.com

Object Browser and Auto-complete: The documentation from ManagedITK assemblies can be viewed in the Object Browser to discover managed classes, methods, properties, and events. The Visual Studio auto-complete feature can be used to read documentation while coding. See Figure 2 and Figure 3.

Unfortunately ManagedITK suffers from a number of disadvantages:

Windows Platform Only: ManagedITK has a dependency on the `vcredist` assemblies. These libraries are only supported by the Windows.NET CLR.

Memory Management: The CLR is an interpreted architecture which uses a garbage collector (GC) to manage memory. As such, there are no guarantees regarding object finalization: the garbage collector reclaims memory as it sees fit. This means that even though an image may no longer be in scope, there is no guarantee that the native `itk::Image` resources have been freed. To help alleviate the issue, every ManagedITK object implements the `IDisposable` interface, providing a `Dispose()` method[7]. Calling this method forces the native object to free its resources, however — in some respects — this practice defeats the purpose of the `itk::SmartPointer`. In our experience it is not *imperative* to call `Dispose()`, nonetheless it has been provided to support deterministic finalization.

Virtual Memory Allocation: Unfortunately we have found the use of virtual memory is limited in comparison to a native C++ implementation. In our experience, a process running on a system with 1 GB physical RAM can only allocate approximately 1 GB of memory. A process running on a system with 2+ GB physical RAM can allocate the full theoretical 2 GB of memory before an `OutOfMemoryException` is thrown.

Performance: Most ManagedITK classes achieve similar performance to their native counterparts. However, a single managed-to-native transition (when managed code calls native code) reduces the execution speed. Optimal performance is achieved when such transitions are minimised[8]. For most `itk::ProcessObjects` the `Update()` method encapsulates the processing, which inherently minimises these transitions. This means, for example, that a majority of managed `itk::ImageToImageFilter` subclasses will execute with similar speed to a natively compiled executable. Other objects (such as `itkImageIterators`) exhibit many managed-to-native transitions by design, and as such their performance is heavily degraded. In the future, the performance of the `ManagedITK.Image.Iterators` assembly *may* be improved by reworking the dividing line between the managed and native worlds.

Semi-automated Wrapping: The wrappers are generated in a semi-automated manner: hard-coded `cmake` files emit C++/CLI code for managed methods, properties, and events (ie. CableSWIG and/or SWIG are *not* used to automatically generate the wrappers). Therefore, if new properties or methods are added to a native ITK object, the wrapper files need to be *manually* updated.

[7] See http://msdn2.microsoft.com/en-us/library/aa730837(VS.80).aspx#cplusclibp_topic3 for more details.
[8] See http://msdn2.microsoft.com/en-us/library/aa730837(VS.80).aspx#cplusclibp_topic5 for more details.

Coverage: While ManagedITK provides coverage for a large majority of objects it does not provide *full* coverage. Objects currently not supported include: shaped neighborhood iterators, statistics (samples, histograms, etc.), deformable transforms, FFT, and complex IO. A full list of wrapped classes is provided with the binaries.

Automated Regression Testing: There are currently no automated regression tests covering the managed wrappers. CMake does not support managed projects and as such its testing mechanism can not be used in this case. It is desirable to add some form of automated testing in the future, perhaps using NUnit.

When deciding if ManagedITK suits your particular needs, please take into account the advantages and disadvantages described above. ManagedITK will not be suitable for all applications, but may be useful for rapid development of simple GUI-based imaging applications.

3 Using ManagedITK

3.1 Using the Pre-compiled Assemblies

Accompanying this article is the full source-code to compile the ManagedITK wrappers. However, unless you need to make changes or additions to the provided assemblies, it is recommended you use the pre-compiled assemblies. To use these assemblies in an application, follow these steps:

1. Download and unzip the pre-compiled assemblies from: The Insight Journal.
2. Ensure the .NET Framework 2.0 is installed on your system. See here to download the .NET Framework Redistributable Package `dotnetfx.exe`.
3. Install the Microsoft Visual C++ 2005 Redistributable Package `vc_redist_x86.exe` available from here. You may be required to reboot your computer after installing these redistributable libraries.
4. Open Visual Studio and create a new project eg. a “Console Application”.
5. Expand the project in the Solution Explorer, right-click “References”, and then select “Add References...”.
6. Select the “Browse” tab and browse to the folder where you unzipped the pre-compiled assemblies.
7. Select the `ManagedITK.Common.dll` assembly and any other required assemblies (use the Ctrl key to select multiple files). Click OK.
8. You are now ready to use ITK from your .NET project!

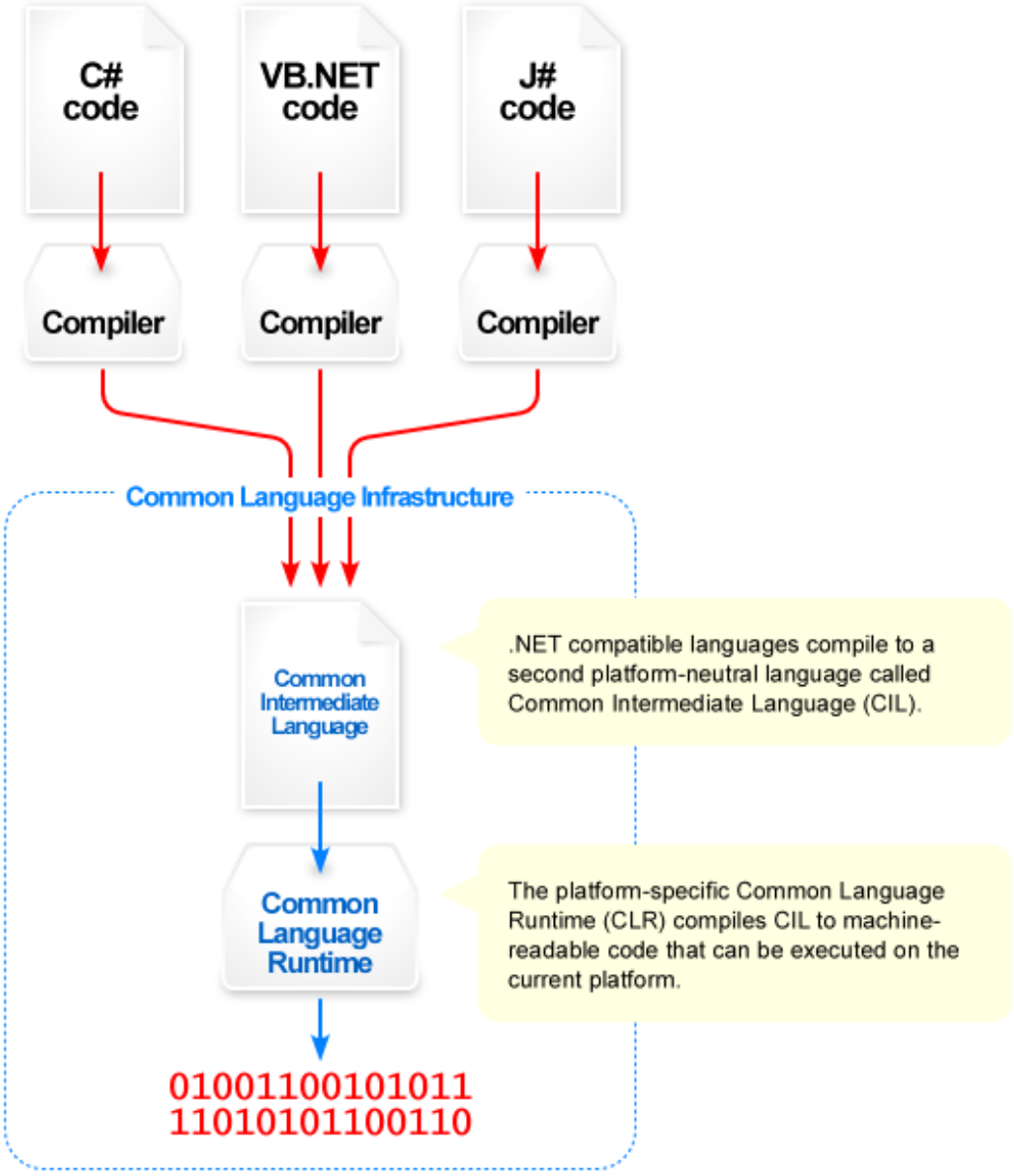
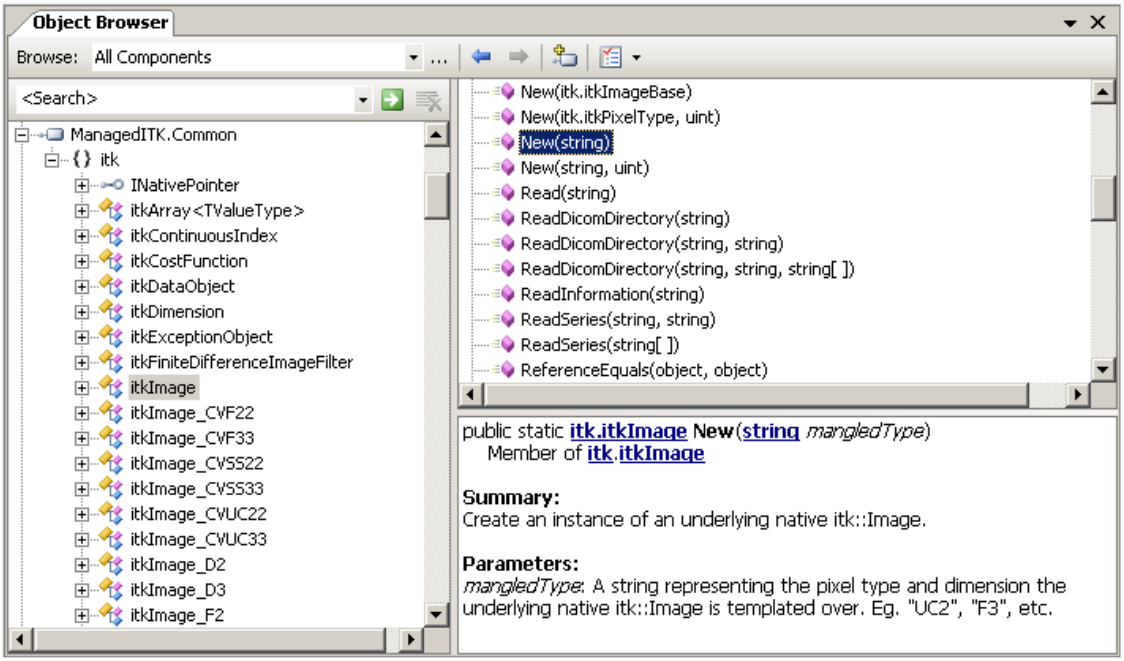


Figure 1: Overview of the Common Language Infrastructure (CLI) [1].



(a)

Figure 2: Using the Visual Studio Object Browser.

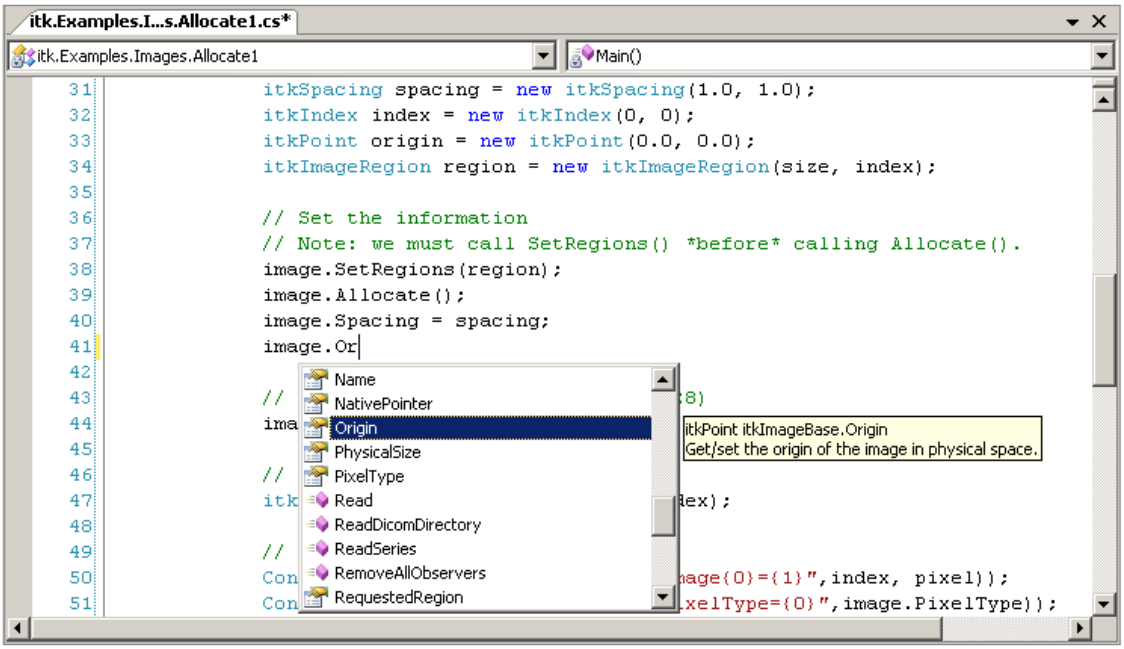


Figure 3: Using Visual Studio Auto-completion.

3.2 Compiling the Assemblies

To compile the ManagedITK assemblies from the source-code with Microsoft Visual Studio 8.0, follow these steps:

1. **Download Source:** The latest and greatest source files can be downloaded from: The Insight Journal

2. **Unzip Source:** Unzip the source to a folder, such as `C:/Insight-Toolkit/Wrapping/ManagedITK`.

3. **Patch ITK:** Follow the instructions in the `Patch` folder to patch your ITK source to fix a bug caused by the Visual Studio 8.0 compiler. Hopefully these patches can be included in the ITK CVS source in the future.

4. **Configure and Build ITK:** Configure and build ITK 3.2.0 or greater with `ITK_USE_REVIEW = ON`.

5. **Configure ManagedITK:** Open CMake and set the source-code to the folder containing the unzipped source and the build path to the desired location (as of v3.2.0.2 ManagedITK supports both in-source and out-of-source builds). The example projects currently have the build path hard-coded, expecting `Buildx86`; if you chose another location you will have to manually change the references for each C# project under the `Examples` folder. Click the Configure button and select "Visual Studio 8 2005"[9]. Set the `ITK_DIR` and `WRAP_*` variables as desired. Click the OK button to finish the configuration.

6. **Convert to Managed Projects:** *Before* opening the projects, run the `FinishCMake.bat` file located in the build folder. Because CMake does not support managed projects, this batch file is required to convert the generated `vcproj` files into managed projects.

7. **Open Solution:** Open the `ManagedITK.sln` solution file. If you have a dual processor or multi-threaded machine, set the parallel project build option as desired. To configure this feature, go to Tools>Options>Projects and Solutions>Build and Run and type the maximum number of parallel builds in the text box.

8. **Build Projects:** Build all the projects by right-clicking `ALL_BUILD` and selecting `Build`. The `ALL_BUILD` project automatically builds all the projects taking care of dependencies.

[9] ManagedITK requires .NET Framework 2.0 or greater. It has only been tested using Visual Studio 8.0 2005 (with SP1), however it *should* work with Visual C++ 2005 Express Edition.

4 Examples

4.1 Images

4.1.1 Allocating an Image

The source code for this section can be found in the file `Examples/Images/itk.Examples.Images.Allocate1.cs`.

This example illustrates how to manually construct and allocate a managed `itk::Image`. Firstly, we use the `itk` namespace (this step is not necessary but makes the code more user friendly[10]):

```
1 using itk;
```

Recall in native ITK, images are templated over the pixel type and the number of dimensions. Templates are not supported by the CLR, so ManagedITK explicitly wraps images with different template combinations. ManagedITK also includes numerous *wrapper* types, however this particular example only uses an *explicit* type. Similar to native ITK, the image is created using the `New()` method:

```
1             // Create an image using an explicit type
2             // Note: "UC2" stands for itk::Image< unsigned char, 2 >.
3             itkImageBase image = itkImage_UC2.New();
```

Finally, we create and set an image region, `Allocate` the memory, and fill the buffer:

```
1             // Create some image information
2             itkImageRegion region = new itkImageRegion(size, index);
3
4             // Set the information
5             // Note: we must call SetRegions() *before* calling Allocate().
6             image.SetRegions(region);
7             image.Allocate();
8
9             // Fill the image with gray (ie. 128)
10            image.FillBuffer(128);
```

Executing the example gives the following output:

```
> itk.Examples.Images.Allocate1
Image[0, 0]=128
PixelType=Unsigned Char
Dimension=2
Size=[128, 128]
Spacing=[1.0, 1.0]
Origin=[000.00, 000.00]
```

[10] The additional `itk` at the front of the name of every managed class was incorporated for two reasons: 1. a similar convention is used for other wrappings (eg. VTK .NET Wrappers: http://vtkdotnet.sourceforge.net), and 2. doing so allows both the native and managed versions to be used without conflict in languages which support both (eg. C++/CLI).

4.1.2 Reading Image Information

The source code for this section can be found in the file `Examples/Images/itk.Examples.Images.ReadInformation1.cs`.

The native `itk::ImageIOBase::ReadImageInformation()` method gives applications the ability to peek at the image information without unnecessarily reading the image data. A similar static method is provided by `itkImageBase.ReadInformation()`, which returns an `itkImageInformation` structure containing the number of dimensions, pixel type, size, spacing and origin.

```
1            // Read the image file information
2            itkImageInformation info = itkImageBase.ReadInformation(args[0]);
```

The information structure can be used to choose the image type at *run-time*, as shown in this example:

```
1            // Use file information to create and read an image
2            itkImageBase image = itkImage.New(info.PixelType, info.Dimension);
```

Executing the example gives the following output:

```
> itk.Examples.Images.ReadInformation1 cthead1.png
Information --------------------------
PixelType=RGB Unsigned Char
Dimension=2
Size=[256, 256]
Spacing=[1.0, 1.0]
Origin=[000.00, 000.00]
Image --------------------------------
Name=cthead1.png
PixelType=RGB Unsigned Char
Dimension=2
Size=[256, 256]
Spacing=[1.0, 1.0]
Origin=[000.00, 000.00]
Buffer=57064620
```

4.1.3 Reading and Writing Images

The source code for this section can be found in the file `Examples/Images/itk.Examples.Images.ReadWrite1.cs`.

One of the benefits of ManagedITK is the ability to easily choose the image type at *run-time*, rather than compile-time. This example shows how to read and write an image, using a command line argument to determine the type at run-time.

Firstly, we use the `itk` namespace:

```
using itk;
```

Next, we use a command line string argument (such as "UC2" or "IF3") to instantiate an image:

```
            // Use the image type from the command line to create an image
            itkImageBase image = itkImage.New(args[0]);
```

For sake of ease the itk::ImageFileReader, itk::ImageFileWriter, and other associated objects have been directly incorporated into the managed itkImage class. This means that image IO can be accomplished very easily only using the image object. Note that the underlying native itk::Image is not actually created until a call to one of the following methods is made: Read(), ReadSeries(), ReadDicomDirectory(), or Allocate(). The Read() method takes a single String argument for the input file name:

```
            // Read the image from disk
            image.Read(args[1]);
```

Similarly the Write() method takes a single String argument for the output file name:

```
            // Write the image to disk
            image.Write(args[2]);
```

Executing the example gives the following output:

```
> itk.Examples.Images.ReadWrite1 "F2" cthead1.png cthead1_OUT.mhd
Name=cthead1.png
PixelType=Float
Dimension=2
Size=[256, 256]
Spacing=[1.0, 1.0]
Origin=[000.00, 000.00]
Buffer=57065616
```

4.1.4 Reading DICOM Images

The source code for this section can be found in the file Examples/Images/itk.Examples.Images.ReadDicom1.cs.

The itkImageBase.ReadDicomDirectory() method uses the native itk::GDCMImageIO to read an image from a DICOM directory. As discussed in the previous example, calling this method creates the underlying native itk::Image. The ReadDicomDirectory() method expects a String specifying the directory, however overrides also exist allowing you to specify the series id and/or restrictions.

In this example we are expecting the DICOM directory to hold a 3D scalar image with SignedShort component type:

```
            // Create an explicit image type
            itkImageBase image = itkImage_SS3.New();
```

The DICOM directory is passed in via a command line argument:

```
            // Read the DICOM image from the given directory
            image.ReadDicomDirectory(args[0]);
```

Finally, the extracted image can be saved in a different format: I recommend the MetaImage format (*.mhd):

```
            // Write the image to disk
            image.Write(args[1]);
```

We have not included a DICOM series in the examples, however executing the example with an appropriate directory gives the following output:

```
> itk.Examples.Images.ReadDicom1 D:\AA1\AA1\AA12345 C:\Temp\AA12456.mhd
Name=D:\AA1\AA1\AA12345
PixelType=Signed Short
Dimension=3
Size=[144, 144, 234]
Spacing=[4.0, 4.0, 4.0]
Origin=[-286.59, -216.59, -1041.35]
Buffer=66715680
```

4.1.5 Reading and Writing Image Series

The source code for this section can be found in the file `Examples/Images/itk.Examples.Images.ReadWriteSeries1.cs`.

This example explains the usage of the `itkImageBase.ReadSeries()` and `itkImageBase.WriteSeries()` methods.

The `itkImageBase.ReadSeries()` method expects an array of file names. Alternatively it expects a base path and a file name containing the wildcard character '`*`' (referred to as a *pattern*). An example for the pattern argument might be: `myfilename*.png`.

```
            // Read the image series using a pattern
            image.ReadSeries(args[0], args[1]);
```

The `itkImageBase.WriteSeries()` method expects a file name format, with the '`{0}`' string. This format placeholder is replaced with the series id or slice number. The format of the series id can also be specified: for example `"000"` forces the series id to have at least three digits. Any numeric format string supported by `Int32.ToString()` can be passed in as the series id format.

```
            // Write the image to disk as a series
            image.WriteSeries(args[2], args[3]);
```

Executing the example with `a7b1.jpg`, `a7b2.jpg`, `a7b3.jpg`, and `a7b4.jpg` in the path `C:/itk` gives the following output:

```
>itk.Examples.Images.ReadWriteSeries1 "C:/itk" "a7b*.png" "a7c{0}.jpg" "00"
PixelType=Unsigned Char
Dimension=3
Size=[72, 72, 4]
Spacing=[1.0, 1.0, 1.0]
Origin=[000.00, 000.00, 000.00]
Buffer=57044976
```

and writes the files: `a7c00.jpg`, `a7c01.jpg`, `a7c02.jpg`, and `a7c03.jpg`.

4.1.6 Displaying images using System.Drawing.Bitmap

The source code for this section can be found in the file `Examples/Images/itk.Examples.Images.FormBitmap1.cs`.

This example shows how to use ManagedITK with `System.Drawing.Bitmap`. It should be noted that the `System.Drawing` namespace is a managed wrapper around the Windows GDI+ library, and as such only supports 8-bit images (which is far from ideal for medical imaging applications). ManagedITK can be used to perform operations on images with a pixel depth greater than 8-bits, however only 8-bit images can be displayed using `System.Drawing.Bitmap`[11].

To display an `itkImageBase` using the `System.Drawing` namespace we must firstly use the required assemblies:

```
using System.Drawing;
using System.Drawing.Imaging;
using System.Windows.Forms;
using itk;
```

Next, we create and read a 2D 8-bit scalar image:

```
            // Read the image
            this.m_Image = itkImage_UC2.New();
            this.m_Image.Read(filename);
```

Next, we set the pixel format to support 8-bit grayscale images:

```
            // Set pixel format as 8-bit grayscale
            PixelFormat format = PixelFormat.Format8bppIndexed;
```

[11] OpenGL supports a much broader range of pixel types. Tao is a .NET binding around the OpenGL library suitable for use with ManagedITK: http://taoframework.com.

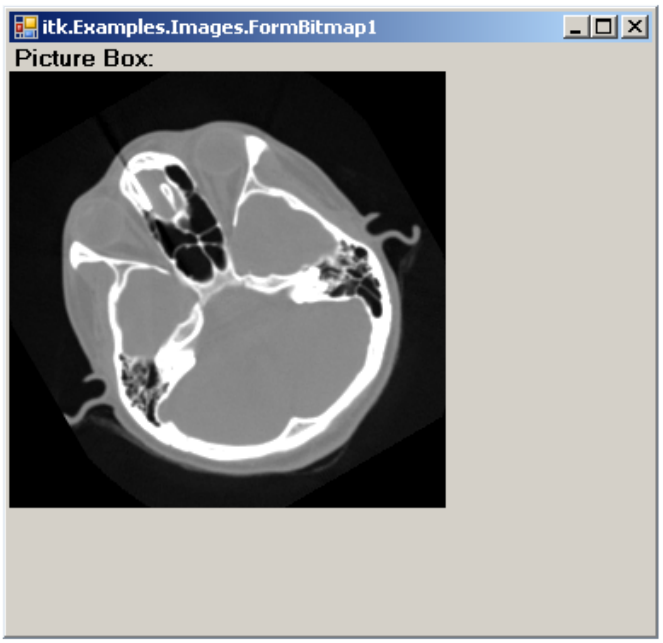


Figure 4: Output from `Examples/Images/itk.Examples.Images.Bitmap1.cs`.

The data in `Bitmap` objects are not tightly packed, rows of bytes are stored in multiples of four (the multiple-four width of an image known as the *stride*). If the width and stride are not equal, we must perform some non-trivial operations to copy the image buffer into the `Bitmap`. However, if the width and stride are equal, we can simply use the `Bitmap` constructor[12]:

```
1                // Width = Stride: simply use the Bitmap constructor
2                bitmap = new Bitmap(image.Size[0],  // Width
3                                    image.Size[1],  // Height
4                                    image.Size[0],  // Stride
5                                    format,         // PixelFormat
6                                    image.Buffer    // Buffer
7                                    );
```

Finally, we set the `Bitmap.Palette`:

```
1            // Set a color palette
2            bitmap.Palette = this.CreateGrayscalePalette(format, 256);
```

Executing the example as below opens the window shown in Figure 4:

```
> itk.Examples.Images.Bitmap1 cthead1.png
```

[12]The method described in this example uses the `itk::Image::GetBufferPointer()`. This is not an ideal solution and — from discussions on the `insight-developers` mailing list — can not be guaranteed to exist in the future. Making this method public has been considered poor design because it forces the `itk::Image` to store data in a contiguous array. However, for the moment it remains a suitable means to obtain the image data.

4.2 Iterators

The source code for this section can be found in the file `Examples/Iterators/itk.Examples.Iterators.ImageRegionIterator1.cs`.

This example shows how to use an `ImageRegionIterator` to iterate over the each pixel in an image. The interesting aspect to note is the use of the `foreach` syntax.

We begin by creating and reading an image using an explicit type:

```
1            // Read an explicitly typed image
2            itkImageBase input = itkImage_UC2.New();
3            input.Read(args[0]);
```

Next, we allocate an empty image for the output:

```
1            // Allocate an empty output image
2            itkImageBase output = itkImage_UC2.New();
3            itkImageRegion region = input.LargestPossibleRegion;
4            output.SetRegions(region);
5            output.Allocate();
6            output.FillBuffer(0);
7            output.Spacing = input.Spacing;
```

Next, we create two iterators, one to walk the input image and one to walk the output image. Note that iterators are created using the normal C# `new` command, this is because the native ITK iterators are not derived from `itk::SmartPointer` and do not use the factory method of instantiation.

```
1            // Create iterators to walk the input and output images
2            itkImageRegionConstIterator_IUC2 inputIt;
3            itkImageRegionIterator_IUC2 outputIt;
4            inputIt = new itkImageRegionConstIterator_IUC2(input, region);
5            outputIt = new itkImageRegionIterator_IUC2(output, region);
```

We now walk over both images, setting the output value as the input value. Note the use of the `foreach` statement: ManagedITK iterators implement the `System.IEnumerable` interface allowing us to use this syntax (alternatively a simple `for` loop could have been used). Also note that the `foreach` statement can only be used on *one* iterator, so we must remember to increment any other iterators manually:

```
1            // Walk the images using the iterators
2            foreach (itkPixel pixel in inputIt)
3            {
4                outputIt.Set(pixel);
5                outputIt++;
6            }
```

This example can be executing using the following code to copy the input image:

```
> itk.Examples.Iterators.ImageRegionIterator1 cthead1.png cthead1_COPY.png
```

4.3 Filters

4.3.1 Gradient Magnitude

The source code for this section can be found in the file `Examples/Filters/itk.Examples.Filters.GradientMagnitude1.cs`.

This example introduces how to use `itkImageToImageFilter` objects, with the `itkGradientMagnitudeImageFilter` as a simple case study. The most important difference has to do with the `itkImageSource.GetOutput()` method.

Firstly, as usual, we use the `itk` namespace:

```
using itk;
```

For code clarity we can define an alias for the `itkGradientMagnitudeImageFilter`:

```
using FilterType = itk.itkGradientMagnitudeImageFilter;
```

Next, we create our input and output images, using the command line to specify the type. Note that we have used the `itkImageBase.New(itkImageBase image)` method for the output image, which creates an explicit type the same as the given image:

```
            // Setup input and output images
            itkImageBase input = itkImage.New(args[0]);
            itkImageBase output = itkImage.New(input);
```

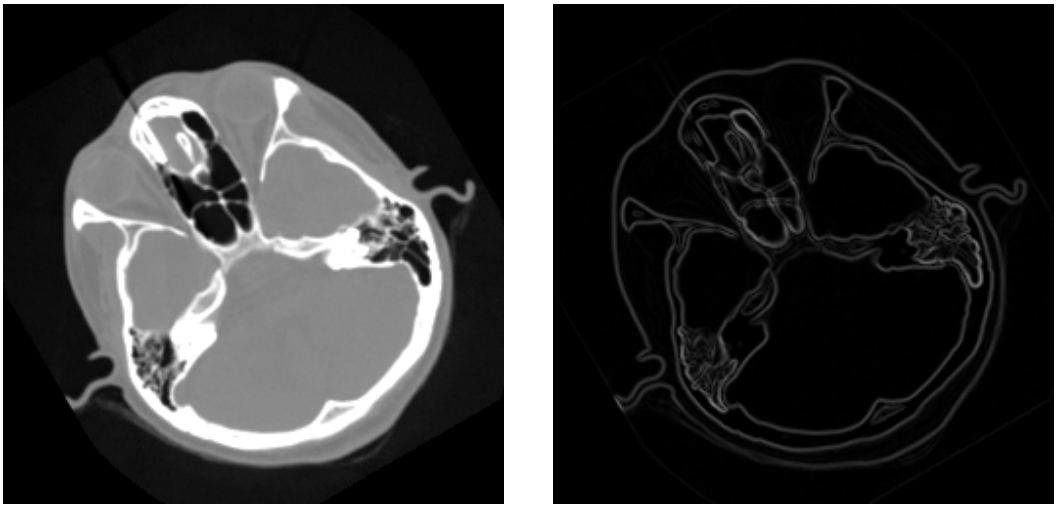
Next, we read the input image:

```
            // Read the input image
            input.Read(args[1]);
```

Creating and applying the filter is similar to native ITK, except the template arguments are moved to the `New()` method. In the case below, we have specified the type parameters as `input, output`. This creates the filter type by appending the `itkObject.ManagedTypeString` from all type parameters: eg. `"IUC2" + "IUC2" = "IUC2IUC2"`:

```
            // Apply the filter
            FilterType filter = FilterType.New(input, output);
            filter.SetInput(input);
            filter.Update();
```

If the template parameters we specify do not create a valid `ManagedTypeString` when appended together, we would receive an exception similar to below:



(a) Input (b) Output

Figure 5: Output from `itk.Examples.Filters.GradientMagnitude1.cs`.

```
itk.itkInvalidWrappedTypeException: Could not create an instance of
'itk.itkGradientMagnitudeImageFilter_IUC2'. The given type may not be supported
or may be invalid.
---> System.NullReferenceException:
The type 'itk.itkGradientMagnitudeImageFilter_IUC2' could not be found in
ManagedITK.Filtering.Common, Version=0.0.0.0, Culture=neutral
   at itk.itkGradientMagnitudeImageFilter.New(String mangledType)
   --- End of inner exception stack trace ---
   at itk.itkGradientMagnitudeImageFilter.New(String mangledType)
   at itk.itkGradientMagnitudeImageFilter.New(INativePointer[] types)
   at itk.Examples.Filters.GradientMagnitude1.Main(String[] args)
      in itk.Examples.Filters.GradientMagnitude1.cs:line 26
```

Finally, we retrieve the output by calling `itkImageSource.GetOutput()` with the `output` object (the object is required to specify the managed type). We could also use the `GetOutput()` method which returns an `IntPtr` to the native type, but this is of little use in the current example.

```
1            filter.GetOutput(output);
```

Executing the example as below gives the output shown in Figure 5:

```
> itk.Examples.Filters.GradientMagnitude1 UC2 cthead1.png cthead1_G.png
```

4.3.2 Sigmoid (and Managed Events)

The source code for this section can be found in the file `Examples/Filters/itk.Examples.Filters.Sigmoid1.cs`.

This example demonstrates how to observe native ITK events using managed events and `delegates` (the managed type-safe equivalent of C-style function pointers).

We firstly create the input and output images, reading the desired type from the command line:

```
            // Create input and output images
            itkImageBase input = itkImage.New(args[0]);
            itkImageBase output = itkImage.New(input);
```

Next, we read the input image from disk:

```
            // Read the input image
            input.Read(args[1]);
```

Now we create the filter, and set the input, output and filter parameters:

```
            // Apply the Sigmoid filter
            FilterType filter = FilterType.New(input, output);
            filter.Started += new itkEventHandler(filter_Started);
            filter.Progress += new itkProgressHandler(filter_Progress);
            filter.Ended += new itkEventHandler(filter_Ended);
            filter.SetInput(input);
            filter.GetOutput(output);
            filter.Alpha = Double.Parse(args[2]);
            filter.Beta  = Double.Parse(args[3]);
```

In the code block above note how we add observers to the `Started`, `Progress`, and `Ended` events. To observe an event, we simply create a `delegate` which references a function containing the desired logic for the event. For example, in the `Started` event handler we print out a string to the console with the time the filter started:

```
    static void filter_Started(itkObject sender, itkEventArgs e)
    {
        string message = "{0}: Started at {1}";
        itkProcessObject process = sender as itkProcessObject;
        Console.Write(String.Format(message, process.Name, DateTime.Now));
    }
```

Finally we write the output to disk. Note that the `Write` method will automatically update any upstream filters:

```
            // NOTE: Any upstream filters will be automatically updated
```

Executing the example as below gives the following output:

```
>itk.Examples.Filters.Sigmoid1 UC2 cthead1.png -60.0 90.0 cthead1_SIGMOID.png
itk.itkSigmoidImageFilter: Started at 19/01/2007 10:16:03 AM
001% 002% 003% 004% 005% 006% 007% 008% 009% 010%
011% 012% 013% 014% 015% 016% 017% 018% 019% 020%
021% 022% 023% 024% 025% 026% 027% 028% 029% 030%
031% 032% 033% 034% 035% 036% 037% 038% 039% 040%
```

```
041% 042% 043% 044% 045% 046% 047% 048% 049% 050%
051% 052% 053% 054% 055% 056% 057% 058% 059% 060%
061% 062% 063% 064% 065% 066% 067% 068% 069% 070%
071% 072% 073% 074% 075% 076% 077% 078% 079% 080%
081% 082% 083% 084% 085% 086% 087% 088% 089% 090%
091% 092% 093% 094% 095% 096% 097% 098% 099% 100%
itk.itkSigmoidImageFilter: Ended at 19/01/2007 10:16:03 AM
```

4.4 Segmentation

4.4.1 Binary Threshold (and Simple GUI)

The source code for this section can be found in the file `Examples/Filters/itk.Examples.Segmentation.FormBinaryThreshold1.cs`.

This example shows how to use the `itkBinaryThresholdImageFilter`. In addition is shows how to use `System.Windows.Forms` to control and monitor a filter which is running on a separate thread.

Firstly we use various namespaces:

```
1 using System;
2 using System.IO;
3 using System.Drawing;
4 using System.Threading;
5 using System.Windows.Forms;
```

For sake of ease we introduce a type alias:

```
1 using FilterType = itk.itkBinaryThresholdImageFilter;
```

Using the Visual Studio Windows Designer, we create a `Form` with a menu containing two items: “Open” and “Exit”. When the user clicks the “Open” menu item, the user will be prompted for an image file name:

```
1         const String allext = "*.mhd;*.png;*.jpg;*.bmp;*.tif;*.vtk";
2
3         // Show an open file dialog
4         OpenFileDialog ofd = new OpenFileDialog();
5         ofd.CheckFileExists = true;
6         ofd.CheckPathExists = true;
7         ofd.Filter = "Meta Header files (*.mhd)|*.mhd";
8         ofd.Filter += "|PNG Image files (*.png)|*.png";
9         ofd.Filter += "|JPEG Image files (*.jpg)|*.jpg";
10        ofd.Filter += "|BMP Image files (*.bmp)|*.bmp";
11        ofd.Filter += "|TIFF Image files (*.tif)|*.tif";
12        ofd.Filter += "|VTK Image files (*.vtk)|*.vtk";
13        ofd.Filter += "|All Image files (" + allext + ")|" + allext;
```

```
        ofd.FilterIndex = 7;
        ofd.Multiselect = false;
        ofd.RestoreDirectory = true;
        ofd.Title = "Select image to process";
        DialogResult result = ofd.ShowDialog(this);
```

If the user selects a valid file name and clicks “OK”, the processing thread will start:

```
        // Start the processing thread
        ParameterizedThreadStart threadStart;
        threadStart = new ParameterizedThreadStart(this.DoWork);
        Thread thread = new Thread(threadStart);
        thread.Start(ofd.FileName);
```

In the new thread, we firstly determine the image information and create input and output images:

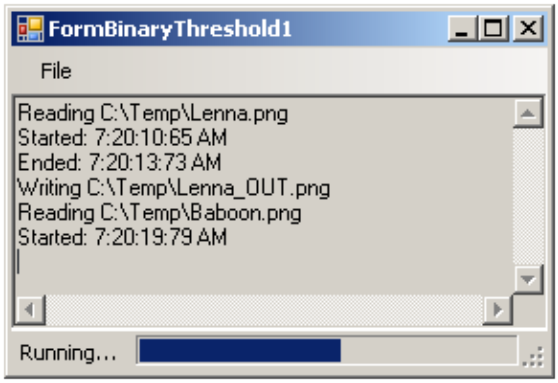
```
            // Create the input and output types
            itkImageBase input; itkImageBase output;
            input  = itkImage.New(itkPixelType.F,  info.Dimension);
            output = itkImage.New(itkPixelType.UC, info.Dimension);
```

After reading the input image, the filter is started:

```
            // Apply the filter
            FilterType filter = FilterType.New(input, output);
            filter.Started += new itkEventHandler(FilterStarted);
            filter.Progress += new itkProgressHandler(FilterProgress);
            filter.Ended += new itkEventHandler(FilterEnded);
            filter.SetInput(input);
            filter.LowerThreshold = 100;
            filter.UpperThreshold = 255;
            filter.OutsideValue = 0;
            filter.InsideValue = 255;
            filter.Update();
            filter.GetOutput(output);
```

The `Started`, `Progress`, and `Ended` events are observed with thread-safe methods which update the GUI as the filter is running. For example, the progress observer increments a `ProgressBar`:

```
    void FilterProgress(itkProcessObject sender, itkProgressEventArgs e)
    {
        // Make thread safe
        if (this.InvokeRequired)
        {
            this.Invoke(new itkProgressHandler(this.FilterProgress),
                        sender, e);
            return;
        }

        // Update progress bar
        this.stripProgressBar.Value = e.ProgressAsPercentage;
    }
```



(a) Screen shot

(b) Input

(c) Output

Figure 6: Output from `itk.Examples.Segmentation.BinaryThreshold1.cs`.

Executing the example as below gives the output shown in Figure 6:

```
>itk.Examples.Segmentation.BinaryThreshold1
```

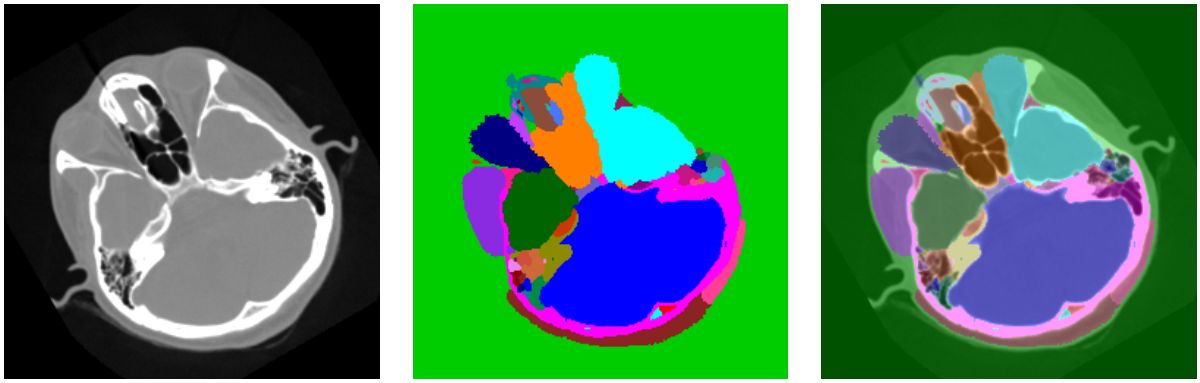
4.4.2 Watershed (and Pipeline)

The source code for this section can be found in the file `Examples/Segmentation/itk.Examples.Segmentation.Watershed1.cs`.

This example shows how to use the `itkWatershedImageFilter`. It also demonstrates the usage of a pipeline.

Firstly, we use the `itk` namespace and in addition we create some aliases for code clarity:

```
1 using itk;
2
3 using PixelType     = itk.itkPixelType;
4 using ImageType     = itk.itkImageBase;
5 using WatershedType = itk.itkWatershedImageFilter;
6 using RelabelType   = itk.itkRelabelComponentImageFilter;
```



(a) Input (b) Coloured Output (c) Coloured Overlay

Figure 7: Output from `itk.Examples.Segmentation.Watershed1.cs`.

Next, we create the input and output label images:

```
            // Create the input and label images
            ImageType input = itkImage.New(args[0]);
            input.Read(args[1]);
            ImageType label = itkImage.New(PixelType.UL, input.Dimension);
```

We now set up the watershed filter, noting that its `New()` method only expects a single type argument:

```
            // Watershed
            WatershedType filterWatershed = WatershedType.New( input );
            filterWatershed.SetInput( input );
            filterWatershed.Threshold = Double.Parse(args[2]);
            filterWatershed.Level = Double.Parse(args[3]);
            filterWatershed.Update();
```

Finally, we connect the filter output to a relabeller using similar syntax as native ITK:

```
            // Relabel
            RelabelType filterRelabel = RelabelType.New( label, label );
            filterRelabel.SetInput( filterWatershed.GetOutput() );
            filterRelabel.Update();
            filterRelabel.GetOutput( label );
```

Executing the example as below gives the output shown in Figure 7:

```
>itk.Examples.Segmentation.Watershed1 UC2 cthead1.png 0.3 0.22
 cthead1_WATERSHED.mhd
```

4.4.3 Level Set Segmentation

The source code for this section can be found in the file `Examples/Segmentation/itk.Examples.Segmentation.LevelSet1.cs`.

This example shows how to use the `itkCurvesLevelSetImageFilter`.

Firstly, we create an alias for code clarity:

```
using LevelSetType = itk.itkCurvesLevelSetImageFilter;
```

Next, we create the initial, speed, and output images by reading the dimensionality from the command line:

```
            // Create the initial, feature and output images
            itkPixelType pixeltype = itkPixelType.F;
            uint Dimension = UInt32.Parse(args[0]);
            itkImageBase initial = itkImage.New(pixeltype, Dimension);
            itkImageBase speed = itkImage.New(pixeltype, Dimension);
            itkImageBase output  = itkImage.New(pixeltype, Dimension);
            Console.WriteLine("Reading initial: " + args[1]);
            initial.Read(args[1]);
            Console.WriteLine("Reading speed: " + args[2]);
            speed.Read(args[2]);
```

We now create and set up the `itkCurvesLevelSetImageFilter`:

```
            // Level Set
            LevelSetType levelset = LevelSetType.New(initial, speed,
                                                     pixeltype);
            levelset.Started += new itkEventHandler(LevelSetStarted);
            levelset.Iteration += new itkEventHandler(LevelSetIteration);
            levelset.Ended += new itkEventHandler(LevelSetEnded);
            levelset.SetInitialImage(initial);
            levelset.SetFeatureImage(speed);
            levelset.PropagationScaling = 5.0;
            levelset.CurvatureScaling = 3.0;
            levelset.AdvectionScaling = 1.0;
            levelset.MaximumRMSError = 0.01;
            levelset.NumberOfIterations = 600;
            levelset.Update();
            levelset.GetOutput(output);
```

We watch for the `Started`, `Iteration`, and `Ended` events. For the iteration event, we write out the number of elapsed iteration in a nice block format:

```
    static void LevelSetIteration(itkObject sender, itkEventArgs e)
    {
        LevelSetType levelset = sender as LevelSetType;
        UInt32 i = levelset.ElapsedIterations;
```

```
        String str = i.ToString("000 ");
        if (i == 1)       Console.Write(str);
        if (i % 100 == 0) Console.WriteLine();
        if (i %  10 == 0) Console.Write(str);
    }
```

Finally, we write the output:

```
            // Write the output image to disk
            Console.WriteLine("Writing output: " + args[3]);
            output.Write(args[3]);
```

Executing the example as below gives the output shown in Figure 8:

```
>itk.Examples.Segmentation.LevelSet1 2 BrainProtonDensitySlice_INITIAL.mhd
 BrainProtonDensitySlice_SPEED.mhd BrainProtonDensitySlice_LEVELSET.mhd
Reading initial: BrainProtonDensitySlice_INITIAL.mhd
Reading feature: BrainProtonDensitySlice_SPEED.mhd
Started: 4/04/2007 9:47:19 AM
001 010 020 030 040 050 060 070 080 090
100 110 120 130 140 150 160 170 180 190
200 210 220 230 240 250 260 270 280 290
300 310 320 330 340 350 360 370 380 390
400 410 420 430 440 450 460 470 480 490
500 510 520 530 540 550 560 570 580 590
600
Ended: 4/04/2007 9:47:20 AM
Writing output: BrainProtonDensitySlice_LEVELSET.mhd
```

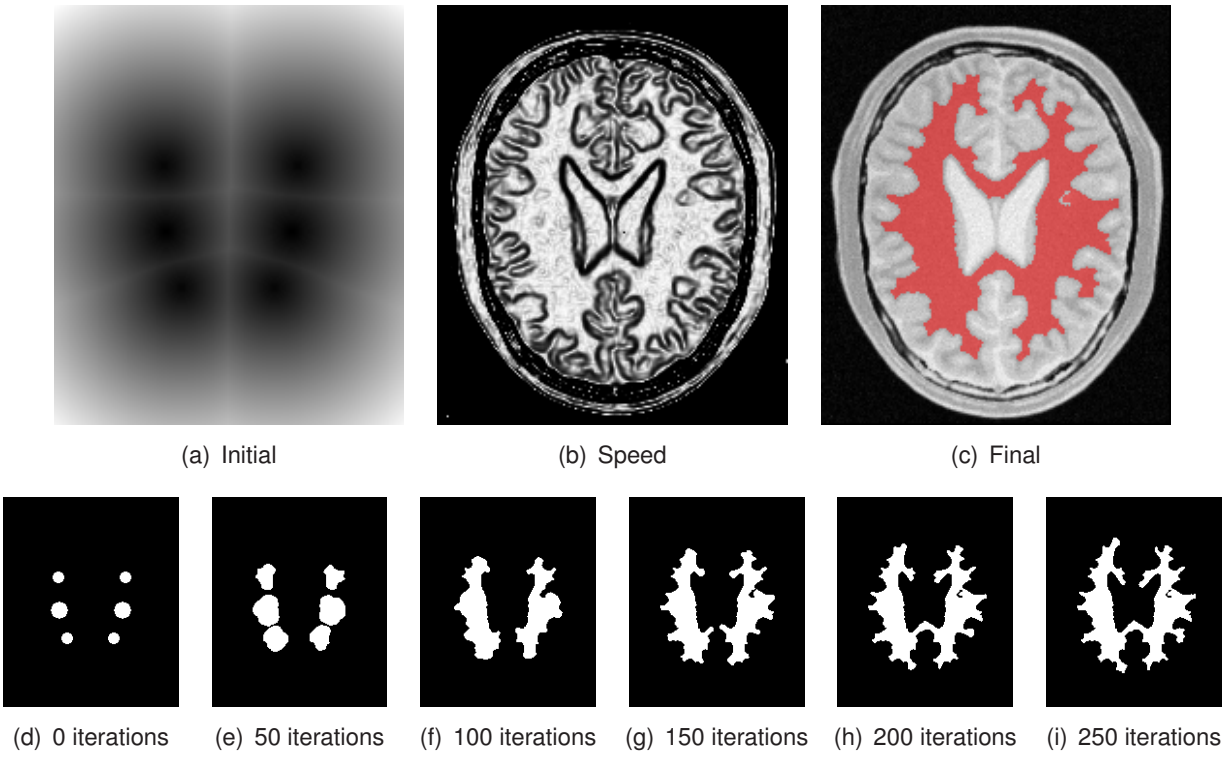
4.5 Registration

The source code for this section can be found in the file `Examples/Registration/itk.Examples.Registration.Translation1.cs`.

This example shows how to use the Registration wrappers. It requires four different assemblies: `ManagedITK.Common.dll`, `ManagedITK.Interpolators.dll`, `ManagedITK.Registration.dll`, and `ManagedITK.Transform.dll`.

Firstly, we use the `itk` namespace and define some aliases:

```
using itk;

using ImageType = itk.itkImage_UC2;
using InterpolatorType = itk.itkLinearInterpolateImageFunction_IUC2D;
using TransformType = itk.itkTranslationTransform_D2;
using ResampleType = itk.itkResampleImageFilter_IUC2IUC2;
using MetricType = itk.itkMeanSquaresImageToImageMetric_IUC2IUC2;
using OptimizerType = itk.itkRegularStepGradientDescentOptimizer;
using RegistrationType = itk.itkImageRegistrationMethod_IUC2IUC2;
```



(a) Initial (b) Speed (c) Final

(d) 0 iterations (e) 50 iterations (f) 100 iterations (g) 150 iterations (h) 200 iterations (i) 250 iterations

Figure 8: Output from `itk.Examples.Segmentation.LevelSet1.cs`.

Next, we read the fixed image from the command line:

```
            // Read the fixed image from the command line
            itkImageBase imageFixed = ImageType.New();
            imageFixed.Read(args[0]);
```

We now construct a moving image by translating the fixed image by a known amount:

```
            // Create a moving image
            itkImageBase imageMoving = itkImage.New(imageFixed);
            InterpolatorType interpolator = InterpolatorType.New();
            TransformType transform = TransformType.New();
            transform.Translate(new itkVector(7.5, 12.0));
            ResampleType filterResample = ResampleType.New();
            filterResample.SetInput(imageFixed);
            filterResample.SetInterpolator(interpolator);
            filterResample.SetTransform(transform);
            filterResample.OutputSize = imageFixed.Size;
            filterResample.OutputSpacing = imageFixed.Spacing;
            filterResample.Update();
            filterResample.GetOutput(imageMoving);
```

Next, we set up the metric and optimizer (the optimizer values were taken from the ITK Software Guide):

```
            // Create metric
            MetricType metric = MetricType.New();

            // Create optimiser
            OptimizerType optimizer = OptimizerType.New();
            optimizer.Iteration += new itkEventHandler(optimizer_Iteration);
            optimizer.MaximumStepLength = 4.00;
            optimizer.MinimumStepLength = 0.01;
            optimizer.NumberOfIterations = 200;
```

Note that we add an observer to the optimizer `Iteration` event:

```
    static void optimizer_Iteration(itkObject sender, itkEventArgs e)
    {
        OptimizerType optimizer = sender as OptimizerType;
        String message = "{0}:   {1}";
        String iteration = optimizer.CurrentIteration.ToString("000");
        String position = optimizer.CurrentPosition.ToString();
        Console.WriteLine(String.Format(message, iteration, position));
    }
```

Finally, we plug everything together and start the registration algorithm:

```
            // Create registration method
            RegistrationType registration = RegistrationType.New();
            registration.SetFixedImage(imageFixed);
            registration.SetMovingImage(imageMoving);
            registration.SetTransform(transform);
            registration.SetInterpolator(interpolator);
            registration.SetMetric(metric);
            registration.SetOptimizer(optimizer);
            registration.InitialTransformParameters = transform.Parameters;
            registration.StartRegistration();
```

Executing the example as below gives the output shown in Figure 9:

```
>itk.Examples.Registration.Translation1 cthead1.png
START: [007.50, 012.00]
000:   [008.20, 008.06]
001:   [005.34, 005.28]
002:   [002.74, 002.23]
003:   [001.32, -001.51]
004:   [-001.14, -004.66]
005:   [-003.17, -008.11]
006:   [-006.29, -010.61]
007:   [-008.82, -013.71]
008:   [-007.60, -012.13]
009:   [-005.97, -010.97]
010:   [-006.83, -011.48]
011:   [-007.63, -012.08]
012:   [-007.15, -011.93]
013:   [-007.40, -011.94]
014:   [-007.64, -012.00]
015:   [-007.52, -012.01]
016:   [-007.40, -012.01]
017:   [-007.46, -012.00]
018:   [-007.52, -011.99]
019:   [-007.49, -012.00]
020:   [-007.50, -012.00]
END:   [-007.50, -012.00]
```

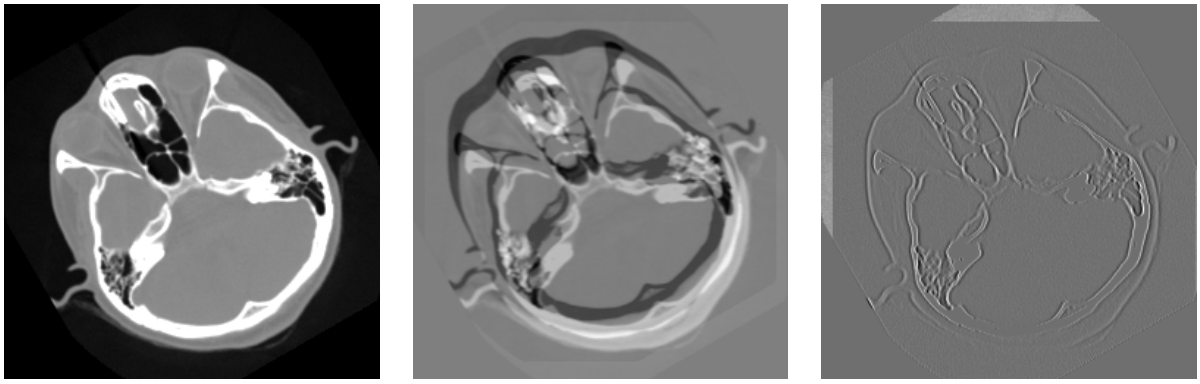
4.6 Meshes

The source code for this section can be found in the file `Examples/Meshes/itk.Examples.Meshes.TriangleMesh1.cs`.

This example shows how to convert a triangle mesh to a binary image.

Firstly, we declare some typedefs for use with meshes:

```
            // Setup typedefs
            itkPixelType pixel = itkPixelType.D;
            itkDimension dim = new itkDimension(3);
            itkMeshTraits traits = itkMeshTraits.Static();
```



(a) Fixed (b) Fixed minus Moving (c) Fixed minus Result

Figure 9: Output from `itk.Examples.Registration.Translation1.cs`.

Next, we use `itkRegularSphereMeshSource` to programmatically create a mesh. Notice that the traits object is passed to the mesh `New()` method:

```
            // Create mesh
            itkMesh mesh = itkMesh.New(pixel, dim, traits);
            itkRegularSphereMeshSource source =
                itkRegularSphereMeshSource.New(mesh);
            source.Center = new itkPoint(32.0, 32.0, 32.0);
            source.Scale = new itkVector(32.0, 32.0, 32.0);
            source.Resolution = 4;
            source.Update();
            source.GetOutput(mesh);
```

Finally, we convert the mesh to an image using `itkTriangleMeshToBinaryImageFilter`:

```
            // Convert mesh to image
            itkImageBase output = itkImage.New(itkPixelType.UC, dim.Dimension);
            itkTriangleMeshToBinaryImageFilter filter =
                itkTriangleMeshToBinaryImageFilter.New(mesh, output);
            filter.SetInput(mesh);
            filter.Tolerance = 0.001;
            filter.Size = new itkSize(100, 100, 100);
            filter.Update();
            filter.GetOutput(output);
```

4.7 IronPython

The source code for this section can be found in the file `Examples/IronPython/IronPythonSpeedImage1.cs`.

This example simply demonstrates that ManagedITK can be used by any language which tar-

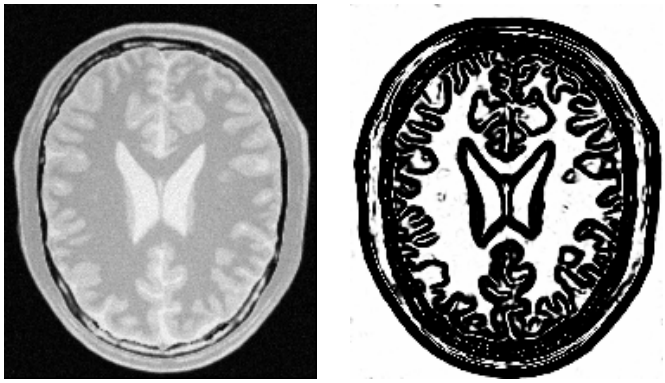
gets the .NET CLR, including IronPython[13]. This example computes a speed image useful for the `itkFastMarchingImageFilter`:

```
# Import default references
import System, clr, sys

# Import ITK references
clr.AddReference("ManagedITK.Common")
clr.AddReference("ManagedITK.Filtering.Common")
clr.AddReference("ManagedITK.Filtering.Intensity")
from itk import *

# Create aliases
ImageType = itkImage_F2
GradientFilterType = itkGradientMagnitudeRecursiveGaussianImageFilter
SigmoidFilterType  = itkSigmoidImageFilter
RescaleFilterType  = itkRescaleIntensityImageFilter

# Create the output image
output = ImageType.New( )

# Open input image
image = ImageType.New( )
image.Read( "BrainProtonDensitySlice.png" )

# Gradient Magnitude
filterGrad = GradientFilterType.New( image, image )
filterGrad.SetInput( image )
filterGrad.Sigma = 1.3
filterGrad.NormalizeAcrossScale = False

# Sigmoid
filterSigmoid = SigmoidFilterType.New( image, image )
filterSigmoid.SetInput( filterGrad.GetOutput() )
filterSigmoid.Alpha = -0.3
filterSigmoid.Beta  =  3.0

# Rescale
filterRescale = RescaleFilterType.New( image, image )
filterRescale.SetInput( filterSigmoid.GetOutput() )
filterRescale.OutputMinimum = itkPixel( image.PixelType,  0.0 )
filterRescale.OutputMaximum = itkPixel( image.PixelType,  1.0 )

# Write the output
filterRescale.GetOutput( output )
output.Write( "BrainProtonDensitySlice_SPEED.mhd" )
```

Executing the example as below gives the output shown in Figure 10:

[13] IronPython is an implementation of the Python programming language running on .NET. It is well integrated with the rest of the .NET Framework and makes all .NET libraries easily available to Python programmers, while maintaining full compatibility with the Python language. See http://www.codeplex.com/IronPython/Wiki/View.aspx.

```
>ipy IronPythonSpeedImage1.py
```



(a) Input (b) Speed Image Output

Figure 10: Output from `Examples/IronPython/IronPythonSpeedImage1.py`.

5 Frequently Asked Questions (FAQ)

Why do I need to install `vcredist_x86`?

ManagedITK is a set of C++/CLI wrapper classes around the native ITK code: it is not 100% managed code. As a result of using the C++/CLI Interop mechanism for mixing native and managed code, all ManagedITK objects depend on the `x86_Microsoft.VC80.CRT` side-by-side assemblies. The `vcredist_x86` executable packages these dependencies and installs them to the `C:/Windows/WinSxS` directory. The pre-compiled assemblies were compiled with Visual Studio 8.0 SP1, and therefore the SP1 version of `vcredist_x86` **must** be used (which is included with the pre-compiled assemblies)[14].

Does ManagedITK work with Mono?

As far we know, ManagedITK will not work with Mono (and hence on Linux platforms supporting by Mono). ManagedITK is not 100% managed code, it depends on the `x86_Microsoft.VC80.CRT` side-by-side assemblies. This dependency (as far as we know...) binds it to the Windows platform. There has been some talk of migrating WrapITK (a totally separate project) to a pure SWIG implementation which could allow for the generation of 100% pure C# wrappers (which would probably be compatible with Mono).

[14] Microsoft has not yet release a version of `vcredist_x86` SP1. However the RTM version can be downloaded here.

Are all ManagedITK objects managed wrappers around native objects?

No, not all ManagedITK objects are managed wrappers around native ITK objects. To simplify the use of common objects most of the objects in the `ManagedITK.Common` assembly are 100% managed (except for the explicit `itkImage` types eg. `itkImage_UC2`). Unfortunately, this design decision has resulted in performance issues with the `ManagedITK.Image.Iterators` assembly, meaning the line between the managed and native worlds may have to be moved in the near future.

How do I use the ManagedITK assemblies?

Section 3.1 explains how to use the pre-compiled ManagedITK assemblies. Basically you create a project, right-click "References", select "Add Reference...", browse to the location containing the ManagedITK assemblies, select the desired assemblies (ensuring to always select `ManagedITK.Common.dll`), and click "OK". The `Examples` folder contains numerous projects using the ManagedITK assemblies in this manner.

How do I determine the `types` parameter for `New()` methods?

All natively wrapped objects have two types of wrappers: explicit-type wrappers (eg. `itkThresholdImageFilter_IF2`) and runtime-type wrappers (eg. `itkThresholdImageFilter`). The runtime-type wrappers require a valid sequence of type parameters passed into the `New()` method.

You can determine which parameters to pass the `New()` method by inspecting the explicit-type wrappers in the Object Browser. To open the Object Browser in Visual Studio select `View>Object Browser`. Expand the desired assembly and scroll to the explicit-type wrappers for the filter or object. In the above example the `itkThresholdImageFilter` has a number of explicit types: `itkThresholdImageFilter_IF2`, `itkThresholdImageFilter_IF3`, `itkThresholdImageFilter_UC2`, `itkThresholdImageFilter_UC3`, etc. The suffix (ie. "IF2") indicates it expects a single image type to be passed to the `New()` method.

Some more complex examples include `itkAddImageFilter` which expects three images (indicated by the "IF2IF2IF2" suffix) or `itkLinearInterpolateImageFunction` which expects one image and a pixel type (indicated by the "IF2D" suffix). See `Examples/Interpolators/itk.Examples.Interpolators.Linear1.cs` for an example.

How do I monitor ITK events?

Common native ITK events (ie. Started, Ended, Aborted, Modified, Iteration, and Progress) have been exposed as managed events. To monitor the managed event, simply use a `delegate` to attach an observer. See Section 4.3.2 for an in-depth example.

Why is my `ImageIterator` so slow?

It is a known issue that `ImageIterator` objects in ManagedITK are slow. This results from the managed-to-native transition which occurs when incrementing the iterator. The problem may be alleviated in the future by simplifying or removing the `itkPixel` class. In the meantime — if speed is an issue for your application — you might consider creating a native custom filter and wrapping it using an external project.

How do I use ManagedITK and OpenGL?

ManagedITK can be easily integrated with OpenGL using Tao (Tao a set of .NET wrappers around OpenGL functions created as part of the Mono project). Go to http://taoframework.com to download the latest assemblies.

How do I use ManagedITK and VTK?

ManagedITK can be integrated with VTK using a VTK .NET wrapper, of which at least two currently exist: http://vtkdotnet.sourceforge.net or http://herakles.zcu.cz/research/vtk.net. An external project is supplied within the `Examples/ExternalProjects/VTK` folder which provides wrappers for `ImageToVTKImageFilter` and `VTKImageToImageFilter`. View the `Readme.txt` file in the project directory for build instructions.

How do I show an image using `System.Drawing.Bitmap`?

See Section 4.1.6 which walks through `Examples/Images/itk.Examples.Images.FormBitmap1.cs`.

How do I wrap an external project?

There will probably come a time when you want to wrap one of your newly created customised filters. Rather than integrate this into the ManagedITK source structure, an external project mechanism (similar to WrapITK) has been provided.

The `Examples/ExternalProjects/Skeletonize` folder contains an example of using such an external project. In this example we wrap a distance-ordered homotopic thinning filter taken from the Insight Journal [2].

The first step is to create a `CMakeLists.txt` file specifying that we are using ManagedITK to wrap an external project. An important aspect to note is the setting of *both* the group and subgroup names ("`Image`" and "`Topology`" in this example).

```
1 PROJECT(WrapSkeletonize)
2
3 # Find required packages
```

```
FIND_PACKAGE(ITK REQUIRED)
FIND_PACKAGE(ManagedITK REQUIRED)

# Use required packages
INCLUDE(${ITK_USE_FILE})
INCLUDE(${MANAGED_ITK_USE_FILE})

# Wrap the project
BEGIN_MANAGED_WRAP_EXTERNAL_PROJECT("Skeletonize")
  SET(MANAGED_WRAPPER_OUTPUT "${CMAKE_BINARY_DIR}")
END_MANAGED_WRAP_EXTERNAL_PROJECT()
```

The next step is to create a CMake file to wrap each class. In our example we created two such files: `managed_itkChamferDistanceTransformImageFilter.cmake` and `managed_itkSkeletonizeImageFilter.cmake`. Each CMake wrapper file must specify the class to wrap, the template parameters, and the managed methods/properties to emit. Look through the `Source/Modules` folders for other examples of such files.

```
# Begin the wrapping
WRAP_CLASS("itk::SkeletonizeImageFilter")

  # Wrap the class for INT and REAL types
  WRAP_IMAGE_FILTER_INT(2)
  WRAP_IMAGE_FILTER_REAL(2)

  # Create a method body
  SET(body "")
  SET(body "${body}m_PointerToNative->SetOrderingImage( ")
  SET(body "${body}(NativeType::OrderingImageType*)image->NativePointer.ToPointer()"
  SET(body "${body} );")

  # Emit a managed method
  BEGIN_MANAGED_METHOD("SetOrderingImage")
    SET(MANAGED_METHOD_SUMMARY      "Set the ordering (ie. distance) image.")
    SET(MANAGED_METHOD_RETURN_TYPE  "void")
    SET(MANAGED_METHOD_PARAMS       "itkImageBase^ image")
    SET(MANAGED_METHOD_TYPE_BODY    "${body}")
    SET(MANAGED_METHOD_WRAPPER_BODY "iInstance->SetOrderingImage( image );")
  END_MANAGED_METHOD()

  # Create a method body
  SET(body "")
  SET(body "${body}m_PointerToNative->SetOrderingImage( ")
  SET(body "${body}(NativeType::OrderingImageType*)imgPtr.ToPointer()")
  SET(body "${body} );")

  # Emit a managed method
  BEGIN_MANAGED_METHOD("SetOrderingImage")
    SET(MANAGED_METHOD_SUMMARY      "Set the ordering (ie. distance) image.")
    SET(MANAGED_METHOD_RETURN_TYPE  "void")
    SET(MANAGED_METHOD_PARAMS       "IntPtr imgPtr")
    SET(MANAGED_METHOD_TYPE_BODY    "${body}")
```

```
    SET(MANAGED_METHOD_WRAPPER_BODY "iInstance->SetOrderingImage( imgPtr );")
  END_MANAGED_METHOD()

# End the wrapping
END_WRAP_CLASS()
```

Next, open CMake and point it to the external project folder, and configure. Run the `FinishCMake.bat` file created in the external project folder and then open the generated solution file. Choose the build type (eg. Debug, Release, etc.) and build the project.

6 Conclusion

This article presented ManagedITK: a set of .NET wrappers for the Insight Toolkit (ITK). These wrappers have a number of nice features including: support for all CLR languages (including C#, VB.NET, and IronPython), the ability to rapidly prototype graphical user interfaces using Windows Forms, and simplified event handling. Various examples were discussed. Future development should focus on the issue of reducing the managed-to-native transition overhead (particularly for `ImageIterator` objects), increasing the coverage (especially to include support for complex IO, meshes, point sets, and deformable registration), and adding automated regression tests. For suggestions, additions or bugs, feel free to contact us[15].

[15]Corresponding author: Dan Mueller: dan.muel@gmail.com or dan.mueller@philips.com.

References

[1] Wikipedia Community. Microsoft .NET Framework. Technical report, Wikimedia Foundation, 2007.
http://en.wikipedia.org/wiki/Microsoft_.NET_Framework. 1

[2] Julien Lamy. Digital topology. *The Insight Journal*, July - December, 2006. 5

[3] Gaetan Lehmann, Zachary Pincus, and Benoit Regrain. WrapITK: Enhanced languages support for the Insight Toolkit. *The Insight Journal*, January - June, 2006. 1

[4] T. Yoo, M. Ackerman, W. Lorensen, W. Schroeder, V. Chalana, S. Aylward, D. Metaxes, and R. Whitaker. Engineering and algorithm design for an image processing API: A technical report on ITK - the Insight Toolkit. *Proc. of Medicine Meets Virtual Reality*, pages 586–592, 2002. 1